Ademir de Alvarenga Oliveira

MetaJ: Um Ambiente para Meta-Programação em Java [1]

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

Belo Horizonte

7 de Outubro de 2004

[1]Esta ferramenta foi desenvolvida em cooperação com o Departamento de Computação da Universidade Federal de Ouro Preto (DECOM – UFOP)

para minha mãe Lavínia;
meu pai Ademir;
irmãos Víctor, Igor, Natália e Izabela;
Maurílio;
tios; primos
e para a doce Daniela.

Agradecimentos

Agradeço ao Departamento de Ciência da Computação da Universidade Federal de Minas Gerais pela oportunidade de participar do seu programa de mestrado. Agradeço à CAPES pelo apoio financeiro.

Aos professores Claudionor José Nunes Coelho Júnior, Christiano Gonçalves Becker, José Monteiro da Mata, Newton Vieira e Nívio Ziviani pelos ensinamentos nas matérias que tive oportunidade de cursar.

Aos amigos Adriano César Machado, Bruno Estolano Grossi, César Francisco de Moura Couto, Fernando Magno Quintão Pereira, Fábio Tirelo, Flávio Humberto Cabral Nunes, Geraldo Antônio Ferreira, Jacques Fux, João Rafael Moraes Nicola, Kristian Magnani dos Santos, Leonardo Barbosa e Oliveira, Patrícia Martins, Ricardo Saraiva de Camargo e Thiago Alves Macambira – pessoas de grande valor que conheci durante o curso – pelas ajudas sempre tão valiosas e pelo companheirismo.

Agradeço também a amiga Mariza Andrade da Silva Bigonha cuja ajuda em momentos difíceis foi preciosa.

Ao meu orientador, amigo e professor Roberto da Silva Bigonha pela atenção, disponibilidade, paciência, e, principalmente, pelos sábios conselhos e ensinamentos.

Ao meu co-orientador e amigo Marcelo de Almeida Maia, com quem venho trabalhando nos últimos cinco anos, pela paciência, atenção, disponibilidade, entusiasmo e pelos ensinamentos que foram essenciais para minha formação.

Aos amigos da UFOP, Thiago Henrique Braga, Denis P. Pinheiro e Rodrigo Geraldo Ribeiro, que tanto contribuíram para a realização deste trabalho.

Agradeço também aos meus primos e tios que sempre foram presença constante na minha vida, dando o apoio necessário para todas as minhas conquistas.

Agradeço à doce, brilhante e tão amada Daniela, cuja presença nos dias mais difíceis me deu forças para continuar.

Agradeço também aos meus irmãos Víctor, Igor, Natália e Izabela, ao Maurílio e ao meu pai Ademir que são amigos e companheiros insubstituíveis.

Finalmente agradeço à minha mãe Lavínia cujos ensinamentos, presença e suporte foram, sem dúvida, os mais importantes.

Resumo

Meta-programas são programas capazes de manipular o código de outros programas. Podem ser encontradas na literatura várias ferramentas especialmente adaptadas para permitir a construção de tal tipo de programas. Estas ferramentas apresentam alguns problemas como a sua complexidade (devido a algumas decisões de projeto adotadas), à falta de recursos básicos (como bibliotecas de I/O, GUI, tratamento de exceções, etc.) facilmente encontrados em linguagens de uso geral e à falta de flexibilidade (permitem apenas a manipulação de programas de uma determinada linguagem). Pela análise destas ferramentas, foi possível também identificar recursos essenciais de uma ferramenta de meta-programação. Neste trabalho, apresentamos MetaJ, um ambiente para meta-programação construído como uma extensão da linguagem Java, que propõe soluções para as deficiências detectadas nas ferramentas encontradas na literatura. MetaJ define quatro abstrações que capturam recursos essenciais de uma ferramenta de meta-programação: referências-p, iteradores, *templates* e *plug-ins*. Com a utilização dessas abstrações, programas Java podem trabalhar facilmente com trechos de programas de qualquer linguagem.

Abstract

Meta-programs are programs which manipulate the code of other programs. Although there is a large number of meta-programming tools available in the literature, these tools have several problems. These problems include their complexity (due to some design strategies), the lack of some basic resources (such as I/O and GUI libraries, exception handling, etc.) and the lack of flexibility (some of them do not allow meta-programers to work with codes of different programming languages). Some meta-programmig essential resources was identified in these tools. This master thesis presents MetaJ, a meta-programming environment built as a Java extension. Its goal is to propose a solution for some of the limitations of the available meta-programing tools. MetaJ provides four elements which capture the essential basic resources of a meta-programming language: p-references, iterators, templates and plug-ins. Meta-programs built with its resources are able to work with codes written in different programming languages.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Linguagens de programação, programas e meta-programas

Talvez sistemas de computação sejam as entidades mais complexas já construídas pelo homem [She01]. No difícil processo de desenvolvimento deste tipo de sistema, as linguagens de programação (LPs) desempenham um papel muito importante: elas são os formalismos utilizados para descrever os procedimentos realizados pelo sistema projetado, i.e., seus algoritmos e estruturas de dados. Logicamente, estes procedimentos não podem ser descritos de qualquer maneira e, por isso, toda LP possui regras:

- sintáticas, que definem a forma como procedimentos devem ser descritos;
- semânticas, que definem o significado de uma descrição.

Descrições que obedecem a estas regras são chamadas de **programas**. Assim como vários outros formalismos, o poder de computação das LPs é limitado pelo poder de computação dos algoritmos. Dizemos que uma LP é *universal* (ou *Turing complete*) se qualquer algoritmo puder ser nela implementado. A universalidade de uma linguagem garante que programas escritos em qualquer formalismo possam ser traduzidos para ela.

Os computadores digitais são projetados de maneira a compreender e serem capazes de executar programas escritos em linguagens universais: as linguagens de máquina[1]. Duas conseqüências devem ser destacadas: (i) programas escritos em qualquer formalismo podem ser traduzidos para linguagens de máquina e, conseqüentemente, executados por um computador digital; (ii) as limitações dos algoritmos se refletem nos computadores digitais.

Atualmente pode ser encontrada uma grande variedade de LPs de alto nível que definem conceitos e abstrações mais adequados para o desenvolvimento de programas de grande porte do que as instruções de baixíssimo nível definidas pelas linguagens de máquina. Estas instruções

[1] Utilizamos o termo linguagens de máquina no plural pois diferentes máquinas compreendem linguagens diferentes.

são muito complicadas e também muito influenciadas por detalhes da arquitetura da máquina utilizada. Desde os anos 50, vários conceitos vêm sendo definidos pelas LPs tendo como um dos principais objetivos oferecer recursos que permitam ao programador abstrair-se dos detalhes da máquina na qual o programa está sendo executado. *Variáveis*, *tipos* (sistemas de tipos), *encapsulamento*, *comandos*, *procedimentos*, *expressões* e *funções* são conceitos que podem ser encontrados em grande parte das linguagens de programação atuais. Através da utilização de compiladores ou de interpretadores, programas que utilizam estes conceitos podem ser traduzidos para instruções da linguagem de máquina, permitindo assim que eles sejam executados por um computador digital.

De uma maneira geral, programas especificam sistemas computacionais que manipulam representações de entidades pertencentes ao universo de discurso do problema em questão. Um sistema de controle de vendas, por exemplo, trabalha com representações de entidades pertencentes ao processo de gerenciamento das vendas de uma empresa (clientes, funcionários, produtos, estoque, lucro, etc.). O universo de discurso de um programa consiste no conjunto de conceitos e entidades que fazem sentido para ele. Os programadores utilizam os recursos definidos pela LP para representar estas entidades, sejam elas concretas ou abstratas. Como conseqüência, desde que seja possível definir uma representação interna para os elementos manipulados, programas podem ser construídos para entender sobre quase tudo que podemos imaginar[2], até mesmo sobre outros programas. São chamados de **meta-programas** aqueles cujo universo de discurso é constituído também por programas. Os programas que são alvo de manipulações são chamados de **programas objeto**[3]. Também é comum a utilização do termo **programas base** para se referir a estes programas (veja [BMR+96]). Nesta dissertação optamos por utilizar o termo "programas objeto", ao invés de "programas base".

Apesar de, a primeira vista, parecerem muito raros, meta-programas são bastante comuns e amplamente utilizados. Na área de linguagens de programação, podemos destacar, dentre várias outras, as seguintes aplicações:

- **tradução**, talvez esta seja a aplicação mais conhecida da meta-programação[4]. Ela consiste em traduzir códigos escritos em uma linguagem fonte em códigos de uma outra linguagem. O tipo mais popular de tradutores são os compiladores, que, geralmente, são utilizados para traduzir programas escritos em uma linguagem de alto nível (C, C++, Pascal) para outra de baixo nível (*assembly*, linguagem de máquina);

- **transformação de programas**, neste caso, o objetivo é realizar modificações automáticas (ou semi-automáticas) em programas para se obter outros que atendam requisitos que

[2]Excluindo-se, é claro, o conjunto dos problemas indecidíveis.

[3]No contexto de meta-programação o termo **programas objeto** é utilizado para se referir tanto aos programas que estão sendo analisados quanto para aqueles que estão sendo gerados, diferentemente da terminologia de compiladores onde os primeiros são chamados de programas fonte. O termo programas objeto é utilizado para ressaltar que eles são **objetos** sobre os quais os meta-programas realizarão operações, sejam elas de análise, geração ou mesmo transformação.

[4]Meta-programação é o ato de se construir meta-programas.

os originais não atendia. Se o requisito for "melhor performance", a tarefa é chamada de **otimização de programas**; se for "melhor estrutura" (arquitetura) do sistema, **reestruturação de programas**.

- **compreensão de programas**, nesta aplicação o objetivo é extrair informações de código para visualizá-lo em uma linguagem de mais alto nível ou mesmo para detectar violações semânticas [Rug95, She01];

- **avaliação parcial**, esta é uma técnica utilizada para melhorar a performance de programas. O objetivo é identificar e realizar o máximo possível de computação antes de sua execução baseando-se em informações sobre algumas das possíveis entradas para o programa [JGS93, She01];

- **mobilidade de código** [TCM98], segundo Sheard, recentemente a meta-programação vem sendo utilizada na área de transporte de programas [She01]. A idéia é: ao invés de utilizar as redes para levar os dados até o programa, utilizá-las para levar o programa até os dados. Neste contexto, os programas são transportados em uma representação que permita que eles sejam analisados para garantir a segurança da máquina hospedeira. Neste caso, meta-programas funcionam como analisadores.

A meta-programação também possui importantes aplicações em engenharia de software, servindo de grande auxílio para aumentar a produtividade em projetos de software [CS92]. Um exemplo é a construção de meta-programas para auxiliar o uso de padrões de programação [MMW01]. Segundo Beck, programadores podem expressar mais facilmente suas intenções com relação ao programa que está sendo construído através da utilização desses padrões [Bec97]. Alguns deles são bastante conhecidos: *padrões de projeto* [GHJV94], *best practice patterns* [Bec97], *heurísticas de projeto* [Rie96], *bad smells* e *padrões de refactorings* [FBB+99, Opd92]. Um problema com essa prática é que as linguagens de programação, bem como os ambientes de desenvolvimento, não oferecem recursos para auxiliá-la [MMW01]. Uma solução para essa limitação é a construção de *meta-programas* para: (i) aplicar ou detectar um determinado padrão em um código já existente, (ii) detectar violações ou gerar código (esqueletos de programas) de acordo com um padrão específico [MMW01], ou mesmo (iii) detectar a ocorrência de *anti-padrões*[5] [BMB+98].

Em princípio, um meta-programa pode ser desenvolvido em qualquer linguagem de programação de uso geral [DV98], como Java, C, Pascal, etc.. Neste caso, os programas objeto são representados internamente como árvores de sintaxe (abstrata) [CI84] construídas a partir das estruturas de dados oferecidas pela linguagem utilizada para especificar o meta-programa: *registros*, em linguagens imperativas (C, Pascal); objetos, em linguagens orientadas-a-objetos (C++,

[5] *Eles (anti-padrões) oferecem o padrão de más soluções e sugerem técnicas de* refactoring *para melhorar essas situações. [BMB+98].*

Java, Object Pascal); tipos de dados algébricos, em linguagens funcionais (ML, Haskel); e termos, em sistemas de reescrita [Vis02]. O problema com essa representação é que manipulá-la não é uma tarefa muito confortável, pois: (i) a distância conceitual entre programas concretos e as operações utilizadas para compor e decompor essas estruturas é muito grande, e (ii) meta-programas nem sempre são escritos por programadores que possuem experiência na construção de compiladores ou no projeto de linguagens de programação [Vis02]. Vale destacar que, apesar de necessárias, as já estabelecias técnicas de compilação não são suficientes para serem adotadas como técnicas gerais de construção de meta-programas. Klint destaca que a reestruturação e compreensão de programas não podem ser tratados simplesmente pela aplicação destas técnicas [Kli03]. Ele justifica tal afirmativa apresentando as seguintes diferenças entre os processos:

- geralmente, a compilação analisa programas escritos em uma única linguagem fonte, enquanto a reestruturação e compreensão de programas pode envolver várias linguagens diferentes. Estas duas últimas tarefas exigem a definição de um *framework* de análise independente da linguagem manipulada;

- um compilador pode abstrair-se do código fonte do programa no mesmo momento em que ele se torna disponível para o sistema. Por outro lado, no caso de reestruturação de programas, é necessário manter a ligação entre a abstração e o código fonte, de forma que seja possível reconstruí-lo;

- a compreensão de programas é uma tarefa que exige uma grande interação com o usuário, diferentemente do processo de compilação que geralmente ocorre em forma de *batch*;

- compreensão e reestruturação de programas são utilizados para fazer *reverse engineering*, enquanto compiladores são utilizados para *forward engineering*. Segundo van den Brant, *forward* e *reverse engineering* são processos inversos: o primeiro consiste em mover de um alto nível de abstração e projeto para um baixo nível de implementação, e o segundo tem como objetivo extrair informações da implementação do sistema (baixo nível) representando-as em um nível diferente de abstração [vdBKV97].

Sheard ainda destaca que a meta-programação não é uma tarefa trivial, a complexidade dos sistemas computacionais é uma das justificativas para essa dificuldade:

> *"(. . .) Programas de grande porte talvez sejam as entidades mais complexas construídas pelos humanos. Programadores utilizam várias ferramentas para gerenciar essa complexidade. Elas são freqüentemente acopladas nas linguagens de programação e incluem: sistemas de tipos (. . .), mecanismos de escopo (. . .) e mecanismos de abstração (. . .). Esses recursos aumentam consideravelmente a complexidade das linguagens onde eles estão embutidos, mas trazem benefícios que superam esta desvantagem. Quando escrevemos programas para manipular programas devemos lidar com essa complexidade duas vezes, uma nos programas que estamos escrevendo, e novamente no que eles estão manipulando." [She01]*

Devido às dificuldades do processo de meta-programar, e à necessidade de construção de meta-programas, existem várias linguagens de domínio específico "ajustadas" para permitir especificá-los mais facilmente [DV98]. Essas linguagens são chamadas de **linguagens de meta-programação**, ou **meta-linguagens**. Utilizaremos o termo **linguagem objeto** para nos referirmos à linguagem utilizada na descrição dos programas objeto. Observe que, apesar de serem "ajustadas" para a meta-progamação, as meta-linguagens não possuem maior poder computacional que as linguagens de programação tradicionais. Elas apenas oferecem recursos que fazem com que a meta-programação seja uma tarefa mais confortável. Muitas vezes, meta-linguagens são concebidas como o agrupamento de vários recursos além da própria linguagem de meta-programação (como ASF+SDF e Stratego, Seções 2.4 e 2.5), neste caso podemos chamá-las também de **ambientes de meta-programação**.

Basicamente, podemos definir duas maneiras de se construir meta-linguagens: (i) estender uma linguagem de uso geral com bibliotecas para meta-programação ou (ii) construír uma linguagem completamente nova que ofereça recursos *built in* para atender aos requisitos da meta-programação.

No primeiro caso, a vantagem é a possibilidade de utilização de todos os recursos já disponibilizados pela linguagem original, como APIs para interfaces gráficas e recursos de I/O, nos meta-programas. Isso é muito importante pois algumas aplicações da meta-programação exigem iteração com o usuário. Mas essa abordagem possui dois problemas: (i) é necessário um maior esforço por parte do (meta) programador para expressar consultas sobre o código fonte e (ii) os meta-programas construídos utilizando a biblioteca são mais difíceis de manter e menos extensíveis [Kli03] do que no caso de linguagens especialmente projetadas para meta-programação.

A outra opção, a definição de uma nova (meta-)linguagem que ofereça recursos *built-in* para meta-programação, tem a vantagem de permitir expressar uma operação sobre o código fonte em alto nível, tornando o meta-programa mais simples. Por outro lado, é mais difícil aprender uma nova linguagem do que uma biblioteca para uma linguagem já conhecida. Essa dificuldade aumenta ainda mais quando a nova linguagem é projetada como sistemas de reescrita ou pertence a paradigmas menos populares como o lógico ou funcional. Outro ponto negativo dessa abordagem é o fato de que, geralmente, uma linguagem é mais complexa de ser construída do que uma biblioteca.

Cordy destaca duas dificuldades em relação ao processo de meta-programar que podem ser consideradas obstáculos para o desenvolvimento de linguagens de meta-programação: (i) a falta de uma abordagem geral para a construção de meta-programas e (ii) a dificuldade no aprendizado da meta-linguagem [CS92].

Apesar das diferentes necessidades dos vários tipos de meta-programas existentes, Cordy afirma que o problema da falta de uma abordagem geral para a meta-programação vem sendo resolvido através da utilização de sistemas de reescrita como TXL [CDMS02][CS92] . Mas, apesar de apresentarem soluções elegantes para o problema, a integração de sistemas de reescrita em

processos de desenvolvimento de *software* completamente baseados em paradigmas imperativos ou orientados a objeto não é uma tarefa muito natural. Vale destacar que, apesar de solucionar o problema da falta de uma abordagem geral para a meta-programação, a utilização de sistemas de reescrita agrava ainda mais o segundo problema apresentado por Cordy: a dificuldade de aprendizado da ferramenta de meta-programação. Para aliviar a grande dificuldade no aprendizado, Cordy destaca que linguagens de meta-programação devem: (i) permitir a utilização de padrões de programas *by example* (ver Seção 2.9.3), (ii) oferecer recursos que sejam simples e de elevado grau de abstração [CS92].

1.2 Objetivos

O objetivo do trabalho apresentado nesta dissertação é projetar e implementar um ambiente para meta-programação. Tendo em vista a facilidade de utilização da ferramenta, o sistema deve possuir seguintes características:

- oferecer uma visão orientada a objetos da meta-programação, em contraste com a abordagem proposta pelas várias linguagens de meta-programação modeladas como sistemas de reescrita;
- permitir a utilização de padrões de programa descritos através da sintaxe concreta da linguagem objeto (*by example*);
- definir abstrações realizadas como classes Java que oferecem serviços aos meta-programas. Desta maneira, a ferramenta não precisará definir uma estrutura rígida de descrição de meta-programas, permitindo maior legibilidade e flexibilidade para os mesmos;
- não realizar modificações na estrutura sintática linguagem Java, ou redefinição de conceitos já existentes;
- permitir fácil acesso aos recursos do ambiente que se tornam disponíveis pela sua importação (**import**);

1.3 Contribuições da dissertação

As principais contribuições deste trabalho são listadas abaixo:

- apresentação e análise de um conjunto de ferramentas para meta-programação disponíveis atualmente (Capítulo 2);
- identificação dos recursos essenciais de uma ferramenta de meta-programação (Seção 2.9.1):(i) padrões de programa, (ii) referências/variáveis para trechos de código objeto, (iii) caminhamentos em código e (iv) componentes que gerenciam características inerentes à linguagem objeto;

- a definição (Capítulo 3) e implementação (Capítulo 4) destes recursos em Java, definido MetaJ, um ambiente para meta-programação projetado como uma extensão de uma linguagem orientada a objetos bastante popular (Java). Este ambiente disponibiliza padrões de programa *by example* e permite fácil acesso aos recursos básicos de Java como bibliotecas de I/O, GUI, tratamento de exceções, etc..

- definição de uma forma semi-automática de se estender gramáticas LALR de linguagens objeto de forma a permitir meta-anotações mantendo a gramática resultante na classe LALR (Seção 3.5.4). Esta definição fez com que MetaJ não necessitasse das técnicas de parser GLR (*Generalized LR Pasing*) e pudesse explorar a eficiência dos parsers LALR, bem como os conhecidos e amplamente utilizados geradores de parsers LALR;

- construção de um processador de gramáticas Cup [Hud99] capaz de construir a gramática meta-anotada citada no item anterior e gerar os componentes utilizados por MetaJ para manipular programas daquela linguagem objeto (Seção 3.6.1);

- definição e implementação de uma linguagem declarativa para consultas em código objeto (SCQL), apresentada como uma extensão de MetaJ (Capítulo 5);

- a implementação da refatoração *Self Encapsulate Field* (Capítulo 6);

1.4 Estrutura da dissertação

Este trabalho está organizado da seguinte maneira:

- No Capítulo 2 é apresentada uma revisão das diferentes ferramentas de meta-programação disponíveis. Neste capítulo são apresentados também os principais requisitos e os recursos essenciais de uma ferramenta de meta-programação, bem como críticas em relação às ferramentas apresentadas;

- O Capítulo 3 apresenta MetaJ – um ambiente para meta-programação em Java. São abordados também os principais recursos e conceitos do ambiente, juntamente com vários pequenos exemplos de utilização destes. O capítulo é finalizado com a apresentação da implementação da refatoração *change_variable_name* em MetaJ;

- O Capítulo 4 trata dos detalhes de implementação de MetaJ;

- O Capítulo 5 apresenta SCQL, uma extensão de MetaJ que permite a descrição declarativa de consultas em programas objeto. Neste serão apresentados também os detalhes de implementação de SCQL. No fim, será apresentada a implementação da refatoração *change_variable_name* [Opd92] destacando-se as vantagens agregadas a MetaJ por SCQL;

- O Capítulo 6 é completamente dedicado à descrição da implementação da refatoração *Self Encapsulate Field* [FBB+99];
- Finalmente, no Capítulo 7 serão apresentadas as conclusões desta dissertação e perspectivas de trabalhos futuros.

Capítulo 2

Ferramentas de Meta-Programação

2.1 Introdução

Neste capítulo são apresentadas algumas ferramentas de meta-programação encontradas na literatura: TXL, ASF+SDF, Stratego, JaTS, TAWK e SCRUPLE. Antes de apresentar seus principais recursos vale destacar dois pontos em comum entre elas.

Primeiramente, estas ferramentas permitem ao programador descrever meta-programas para manipular programas de uma linguagem objeto cuja estrutura sintática seja especificada por uma gramática livre de contexto (GLC). GLCs são formalismos utilizados para descrever linguagens formais, como, por exemplo, as linguagens de programação. Uma descrição livre do contexto é composta por um conjunto R de regras de derivação que definem como as sentenças da linguagem podem ser produzidas. Cada regra de R é especificada utilizando-se terminais, i.e., símbolos que compõem uma sentença, e não-terminais, símbolos temporários que, no processo de produção de uma sentença, são substituídos por uma seqüência de terminais. Formalmente, uma GLC é uma quadrupla (N, T, R, P), onde N é o conjunto dos símbolos não-terminais, $P \in N$ é o símbolo inicial, T o conjunto dos terminais e R o conjunto das regras de derivação. Toda regra possui a seguinte forma: $A \rightarrow B_1 B_2 B_3 \dots B_n$, onde $A \in N$, $n \in \mathbb{N}$ e $B_i \in (N \cup T)$ sendo $i \leq n$. A linguagem definida por uma GLC é o conjunto de todas as sentenças que possam ser construídas tomando-se o símbolo inicial P e substituindo-se continuamente cada ocorrência de um não-terminal Z pelo lado direito de alguma regra $r \in R$ da forma $Z \rightarrow B_1 B_2 B_3 \dots B_n$, onde $B_i \in (N \cup T)$, até que a sentença seja composta apenas por símbolos terminais.

Outro ponto em comum entre as ferramentas é que elas representam programas objeto como árvores sintáticas, sobre as quais as operações especificadas pelo meta-programa serão aplicadas. Estas estruturas de dados são importantes pois, além de armazenar o código do programa, também descrevem os passos tomados no processo de reconhecimento do mesmo.

GLCs são um formalismo consagrado e amplamente utilizado em ciência da computação, da mesma maneira, árvores sintáticas são estruturas de dados bastante conhecidas devido a sua grande utilização na implementação de compiladores. Por esta razão não entraremos em maiores

detalhes sobre estas entidades.

A próxima seção apresentará os requisitos que devem ser atendidos por uma ferramenta de meta-programação. Em seguida, TXL, ASF+SDF, Stratego, JaTS, TAWK e SCRUPLE serão apresentadas.

2.2 Requisitos da Meta-Programação

Abaixo são listados os requisitos que devem ser atendidos por uma ferramenta de meta-programação.

- **Expressividade e legibilidade**: o principal objetivo de uma ferramenta de meta-programação é facilitar a difícil tarefa de meta-programar [She01]. Para isso, é necessário que a linguagem utilizada para escrever meta-programas seja expressiva e de fácil entendimento. É importante que a ferramenta disponibilize abstrações que "escondam" a representação interna (árvore de sintaxe) do programa objeto.

- **Flexibilidade**: é necessário que a ferramenta atenda às várias atividades de meta-programação: geração, análise, transformação, etc.;

- **Facilidade no aprendizado**: segundo Cordy, uma das razões para que a meta-programação ainda não tenha sido amplamente aceita é a acentuada curva de aprendizado[1] da ferramenta [CS92]. É necessário que uma ferramenta de meta-programação seja fácil de aprender.

- **Padrões de programas descritos *by example***: a utilização de padrões descritos *by example*, i.e., utilizando a sintaxe concreta da linguagem objeto, facilita o entendimento das ferramentas. Veja maiores detalhes na Seção 2.9.3.

- **Garantia da correção sintática dos programas objeto**: é necessário que as ferramentas disponibilizem um mecanismo para garantir consistência sintática do programa gerado, pois, na maioria das vezes em que se deseja gerar programas, é necessário que ele esteja de acordo com a gramática da linguagem objeto.

- **Multi-linguagens**: geralmente, sistemas de software são construídos utilizando mais de uma linguagem de programação. Por isso, é necessário que as ferramentas permitam a manipulação de várias linguagens objeto.

No final de cada uma das seções que se seguem, para cada ferramenta apresentada, serão mostradas suas principais falhas em relação aos requisitos descritos acima. Estas deficiências serão apresentadas de maneira resumida nos comentários finais deste capítulo.

[1] Curva de aprendizado é a relação entre o tempo para se executar a tarefa e o número de vezes que tal tarefa foi executada

2.3 TXL

TXL [CDMS02, CHP88, CC93] é uma linguagem de transformação de programas [Cor04], i.e., uma linguagem onde cada meta-programa descreve regras para se alterar o programa objeto recebido como entrada. Cada programa TXL é composto pela GLC da linguagem objeto e pelas regras e funções de transformação. A execução de um programa TXL ocorre em três fases: *parse*, na qual é construída uma árvore sintática para representar o programa objeto que foi recebido em formato textual; *transform*, na qual a árvore construída na fase anterior é transformada segundo as regras de transformação do meta-programa; *unparse*, na qual a árvore resultante da fase anterior é convertida para o formato textual [CC93].

A primeira parte de um programa TXL é a gramática da linguagem objeto, que deve ser especificada através da construção **define**. Cada ocorrência desta construção agrupa todas as formas possíveis de um não-terminal, veja o exemplo abaixo:

Exemplo 2.1 [CC93]:

(a)
```
expression → number
        |   expression "+" number
        |   expression "-" number
```

(b)
```
define expression
            [number]
        |   [expression] + [number]
        |   [expression] - [number]
end define
```

A produção BNF descrita em (a) pode ser feita em TXL como mostrado em (b).

A linguagem oferece também os operadores **repeat** e **list** para expressar repetições de maneira mais fácil. O primeiro é utilizado para definir repetições de determinada estrutura e o segundo para repetições separadas por vírgula (“,”), veja os exemplos abaixo.

Exemplo 2.2 :

(a)
```
statements → statement
        | statements statement
```

(b)
```
define statements
        [ repeat statement ]
end define
```

A letra (b) mostra como o operador **repeat** pode ser utilizado para expressar a lista de *statement*s descrita em BNF na letra (a).

(a)
```
formalParameters → formalParameter
        | formalParameters "," formalParameter
```

(b)
```
define formalParameters
        [ list formalParameter ]
    end define
```

A letra (b) do exemplo acima mostra como o operador **list** pode ser utilizado para expressar a lista de *formalParameter*s descrita em BNF na letra (a).

Existem também construções especiais para declarar comentários (**comments**) e palavras reservadas (**keys**).

A especificação da GLC da linguagem objeto é utilizada como roteiro para construção da árvore sintática que representa o programa objeto recebido como entrada pelo meta-programa TXL. Cada nodo desta árvore é rotulado com um tipo, que corresponde ao nome do não-terminal da gramática que derivou a construção sintática que ele representa.

A segunda parte de um programa TXL são as regras e funções de transformação. Estas entidades são aplicadas sobre a árvore sintática do programa objeto a ser transformado. TXL garante que a aplicação delas sempre preserva a estrutura sintática do programa resultante [CDMS02]. Abaixo é apresentada a sintaxe que deve ser utilizada para descrever uma regra ou função de transformação [CC93]:

```
(function | rule) <nome> <parâmetros formais>
    replace [<tipo>]
        <padrão>
    by
        <substituição>
```

As palavras reservadas **function** e **rule** definem se a declaração corresponde a uma função ou regra de transformação, respectivamente. *<nome>* é o nome da regra ou função. *<parâmetros formais>* é uma seqüência de parâmetros formais que podem ser fornecidos. Cada parâmetro é declarado com um nome e um tipo. Em *<padrão>* deve ser fornecido um padrão de casamento e em *<substituição>*, um de substituição. Os padrões são definidos utilizando-se uma sequência da símbolos teminais da linguagem objeto e variáveis. A primeira ocorrência de cada variável nova no padrão deve aparecer acompanhada com o seu tipo (p. ex., X [expression]). Utilizações subsequentes não devem aparecer com informação de tipo. Para reconhecer padrões de programas, a linguagem utiliza um *parser* descendente recursivo com um limite superior para a profundidade do *backtracking* [vdBSV98] [Cor04].

A semântica da aplicação de uma regra de substituição sobre uma árvore t é: a substituição de todas as sub-árvores de t que possuem tipo *<tipo>*, e que estão de acordo com o padrão de casamento, pelo padrão de substituição. Já a semântica da aplicação de uma função de substituição é: se t casa com o padrão, então o resultado é o padrão de substituição, caso contrário o resultado é a própria árvore t. Abaixo é apresentado um exemplo de regra e um de função de substituição:

Exemplo 2.3 [CC93]:

```
rule TodoDoisPorN N [Number]
    replace [Number]
        2
    by N

function FDoisPorQuarenta
    replace [Number]
        2
    by 40
```

A regra `TodoDoisPorN` aplicada à seqüência `20 2 2 45`, passando como parâmetro o número 50, resulta em `20 50 50 45`. Já a função `FDoisPorQuarenta`, quando aplicada àquela mesma seqüência não realiza nenhuma modificação, pois Number não casa com a seqüência.Porém, quando a função é aplicada isoladamente ao número `2`, resulta em `40`.

A linguagem ainda oferece operadores **deconstruct** e **construct** (para decompor e compor trechos de programas), expressões condicionais (**where**), regras condicionais, dentre outros elementos.

São exemplos de aplicações de TXL: (i) recuperação de informações de projeto a partir de código fonte, (ii) tradução de interfaces, que consiste em converter uma interface descrita em uma linguagem de especificação de interfaces (como IDL [Lam87]) para uma definição de interface em uma linguagem de programação específica (como C, Java, etc.), (iii) extensão de linguagens de programação e (iv) implementação de linguagens de *meta-programação*, a família $\mu^\star$ [CS92] [CDMS02]. Esta é uma família de *meta-linguagens* implementada utilizando TXL. Todas as linguagens $\mu^\star$ são baseadas na mesma filosofia: os *meta-programas* são descritos utilizando trechos da *linguagem objeto* intercalados com *meta-anotações*, as quais são predicados sobre uma base de dados Prolog contendo fatos sobre a *linguagem objeto*.

A principal crítica a TXL está relacionada ao requisito de facilidade no aprendizado da ferramenta. A justificativa para esta é ligada ao fato de que TXL é uma linguagem completamente nova, que define sua própria sintaxe, seus conceitos e comandos, o que dificulta o aprendizado da ferramenta. Outro agravante é o paradigma adotado: sistemas de reescrita são menos populares que linguagens de programação baseadas em paradigmas imperativos ou orientados a objetos. Sistemas de grande porte são muito freqüentemente construídos utilizando-se linguagens e processos de desenvolvimento voltados para os paradigmas imperativo ou orientado a objetos. Neste ambiente, a utilização de TXL implicaria na necessidade de aprendizado de um novo paradigma por parte dos desenvolvedores.

Não foram encontradas, na literatura revisada [CC93] [CHP88] [CDMS02] [Cor04], informações sobre recursos TXL que permitam aos seus meta-programas interagir com o usuário (como bibliotecas de I/O ou GUI – *graphical user interface*). Esta deficiência afeta diretamente

a flexibilidade da ferramenta pois estes recursos são muito importantes para algumas aplicações da meta-programação (ex.: reestruturação e compreensão de programas, ver Seção 1.1). É necessário que ferramentas de meta-programação ofereçam meios simples e flexíveis que permitam a interação entre meta-programas e o usuário. De maneira equivalente, também não foram encontradas informações sobre mecanismos de tratamento de exceções e controle de erros.

Além destas críticas, vale destacar que TXL utiliza um *parser full backtracking top-down* para reconhecer sentenças da linguagem objeto [vdBSV98] [Cor04]. Apesar dele ser capaz de reconhecer qualquer gramática livre de contexto, seu tempo de processamento pode ser muito alto ou mesmo impraticável para algumas gramáticas [Cor04].

2.4 O *Meta-Environment* ASF+SDF

> *O* Meta-Environment *ASF+SDF é um ambiente de desenvolvimento para geração automática de sistemas interativos para manipular programas, especificações ou outros textos escritos em uma linguagem formal [vdBBK03].*

O ambiente é resultado da combinação de dois formalismos ASF [BHK89] (*Algebraic Specification Formalism*) e SDF [Vis97](*Syntax Definition Formalism*). SDF é utilizado para definir a sintaxe concreta e abstrata da linguagem que será manipulada. ASF é o formalismo utilizado para descrever equações que definem a semântica da linguagem. A partir das especificações SDF, o ambiente é capaz de construir a árvore sintática do programa objeto. As equações ASF são vistas pelo ambiente como regras de reescrita e, por isso, possíveis de serem executadas. Estas regras são continuamente aplicadas ao programa objeto recebido como entrada até que ele seja reduzido a uma forma normal, i.e., um programa onde nenhuma regra pode mais ser aplicada.

Especificações ASF+SDF são realizadas através de módulos. Existem construções especiais para controle de visibilidade e importação de entidades. A sintaxe de descrição de um módulo é dada abaixo [vdBBK03]:

module *<nome do módulo>*
(**imports** *<nome de módulo>*$^{+}$)*
((**exports** | **hiddens**) *<entidade exportada ou oculta>*$^{+}$)*
equations
<equação>$^{+}$

Toda descrição de módulo inicia-se pela palavra **module** seguida do nome do módulo que está sendo declarado. Uma seção de importações de módulos (**imports**) pode ser especificada. Em seguida pode também ser declarada uma lista de entidades exportadas (**exports**) ou ocultas (**hiddens**). Finalmente, precedida pela palavra **equations**, uma lista de equações pode ser definida.

Nas próximas seções serão tratadas as principais partes desta declaração: seções **exports**, **hiddens** e **equations**

2.4.1 Entidades exportadas e ocultas (exports e hiddens)

As entidades que podem ser exportadas ou declaradas como ocultas em um módulo são: (i) *sorts*, (ii) sintaxe léxica, (iii) sintaxe livre de contexto, (iv) variáveis.

Sorts

Sorts são os símbolos não-terminais que serão utilizados em descrições léxicas e sintáticas da linguagem objeto. Eles também definem os tipos de construções sintáticas da linguagem objeto, toda construção derivável a partir de um mesmo *sort* são de um mesmo tipo sintático.

Sintaxe léxica (lexical syntax)

Em **lexical syntax** são descritas as regras de formação dos *tokens* da linguagem manipulada. Ela é composta por funções léxicas, que ligam aos *sort*s expressões regulares que definem os *tokens* representados por ele. Para descrever expressões regulares são fornecidos operadores para expressar repetição (`A*`, `A+` ou `An+`, onde n é o número de vezes que `A` deve ser repetido), opção (`A?`), que descreve a parte opcional da regra léxica, alternativa (`A | B`), seqüência (`A B`), etc.. Além disso, as regras poderão utilizar também:

- classes de caracteres (*character class*), como `[0-9]`, para expressar os dígitos de `0` a `9`; `[a-z]`, para expressar os caracteres de `a` a `z`; `[a]`, para expressar o caractere `a`; `[abcd]`, para expressar um dos caracteres `a`, `b`, `c` ou `d`;
- literais, como `"um string"` e `true`.

A sintaxe para definição de uma função léxica é: *<expressão regular> -> <sort>*. Abaixo mostramos um exemplo de módulo ASF+SDF no qual podem ser vistas declarações de *sorts* e de funções léxicas:

Exemplo 2.4 [vdBBK03]:

```
module Numbers
exports
    sorts UnsignedInt SignedInt UnsignedReal Number
    lexical syntax
        [0] | [1-9][0-9]* -> UnsignedInt
        [\+\-]? UnsignedInt -> SignedInt
        UnsignedInt "." UnsignedInt ([eE] SignedInt)? -> UnsignedReal
        UnsignedInt [eE] SignedInt -> UnsignedReal
        UnsignedInt | UnsignedReal -> Number
```

No módulo acima, foram declarados os *sorts* `UnsignedInt`, `SignedInt`, `UnsignedReal` e `Number`. As funções léxicas definem quais os *tokens* são representados por estes *sorts*. A

primeira função define que `UnsignedInt` representa uma seqüência de dígitos entre `0` e `9`. A segunda define que `SignedInt` representa os mesmos *tokens* que `UnsignedInt` precedidos por um dos sinais "`+`" ou "`-`". São definidas também duas expressões regulares para definir `UnsignedReal`, *sort* que representa números reais. Finalmente, uma expressão regular é construída para definir `Number`, que representa números inteiros ou reais sem sinal.

Cada ocorrência de *sort*s no lado esquerdo de uma função léxica é substituída pela expressão regular associada a ele. No Exemplo 2.4 pode ser observado que várias regras foram utilizadas para um mesmo *sort*. O conjunto de *tokens* representado por um *sort* é a união dos conjuntos de *tokens* descritos pelas expressões regulares associadas a ele.

Sintaxe livre do contexto (context-free syntax)

A sintaxe livre do contexto é utilizada para descrever a estrutura sintática concreta e abstrata da linguagem objeto [vdBBK03]. Ela é composta por funções livres do contexto que podem ser definidas de maneira semelhante a funções léxicas: *(<símbolo>)** -> <sort>*, onde `<símbolo>` pode ser um *sort*, uma classe de caracteres ou um literal. A notação utilizada é equivalente à BNF, mas cada regra escrita em ordem inversa. Os operadores disponíveis para a descrição léxica também podem ser utilizados. Veja abaixo um exemplo de descrição sintática livre do contexto.

Exemplo 2.5 :

```
module Declaracoes
exports
    sorts ID, DECL, TYPE, NUMERICTYPE
    lexical syntax
        [a-z][0-9a-z]* -> ID
    context-free syntax
        "int" -> NUMERICTYPE
        "real" -> NUMERICTYPE
        "char" | NUMERICTYPE -> TYPE
        "decl" ID ("," ID)* ":" TYPE -> DECL
```

Neste módulo, foram declarados quatro *sorts*, `ID`, `DECL`, `TYPE` e `NUMERICTYPE`, uma função léxica e quatro livres de contexto. A função léxica define que os *tokens* representados pelo *sort* `ID` são todas aquelas seqüências de dígitos e caracteres entre `a` e `z` começadas por algum caractere entre `a` e `z`. As duas primeiras funções livres de contexto definem que o *sort* `NUMERICTYPE` representa as palavras "`int`" ou "`real`". A terceira define que o *sort* `Type` representa ou um tipo numérico (`NUMERICTYPE`) ou a palavra "`char`". Através destas três regras é possível perceber a equivalência entre a utilização de várias regras sintáticas para definir os *tokens* representados por um *sort* (no caso do `NUMERICTYPE`) e a utilização do operador alternativa (no caso de `TYPE`). Finalmente, os *tokens* representados por (`DECL`)

iniciam-se com a palavra "`decl`" seguida por uma seqüência não vazia de `ID`s separados por virgula, dois pontos e finalizados por um tipo (`TYPE`). Todas ocorrências de literais *string* são acrescentados automaticamente à seção sintaxe léxica.

Prioridades (priorities)

ASF+SDF permite o uso funções sintáticas que especificam regras ambíguas. Para construir o parser especificado por elas, o ambiente utiliza técnicas *Scannerless GLR*. *Scannerless parsing* são técnicas que não utilizam *scanners*, i.e., analisadores léxicos, para separar o texto de entrada em *tokens*. A análise léxica é completamente integrada à análise sintática do texto. *GLR parsing (Generalized LR parsing)* é uma extensão da técnica LR de construção de parsers que permite trabalhar com qualquer gramática livre de contexto, mesmo as ambíguas. No processo de geração de parsers LR, duas são as causas de conflitos na tabela de parser: ambigüidades e a falta de *lookahead*. No primeiro caso, qualquer que seja o caminho tomado pelo processo de reconhecimento resultará em sucesso. No segundo, um dos caminhos resultará em sucesso, os outros falharão. A técnica *GLR* trata estes conflitos da seguinte maneira: para cada caminho possível, é criada uma instância do parser para processá-lo; as instâncias que falharem serão descartadas; se mais de uma instância resultar em sucesso, o parser construirá uma *parser forest*, i.e., uma representação compacta das várias árvores sintáticas possíveis para a sentença processada [Vis97]. Esta representação compacta é, na verdade, uma única árvore em que as ambigüidades são explicitamente marcadas. Muitas, mas não todas, destas possibilidades podem ser eliminadas pelas estratégias de solução de ambigüidades disponibilizadas pelo ambiente: (i) definição de prioridade entre funções livres de contexto (**context-free priorities**); (ii) definição de associatividade (`left`, associatividade a esquerda, `right`, associatividade a direita, `non-assoc`, não associativo); (iii) `prefer` e `avoid`, para indicar a preferência de uma função sobre outra; (iv) `reject` para rejeitar uma função. Maiores detalhes sobre estas e outras estratégias de solução de conflitos podem ser encontrados em [Vis97] e em [vdBBK03].

Variáveis (variables)

Uma declaração de variável é feita de acordo com a seguinte sintaxe: *<nome variável> -> <sort>*, onde *<nome variável>* é uma expressão regular que define quais os *tokens* que representam declarações de variáveis. Estas variáveis aparecerão intercaladas em trechos de programas da linguagem objeto e serão utilizadas nas equações (**equations**). Variáveis são marcações coringa que aparecerão no lugar de construções do tipo *<sort>*. Veja o exemplo abaixo:

Exemplo 2.6 :

```
module Declaracoes
exports
    sorts ID, DECL, TYPE, NUMERICTYPE
```

```
lexical syntax
    [a-z][0-9a-z]* -> ID
context-free syntax
    "int" -> NUMERICTYPE
    "real" -> NUMERICTYPE
    "char" | NUMERICTYPE -> TYPE
    "decl" ID ("," ID)* ":" TYPE -> DECL

hiddens
    variables
    "Id" -> ID
    "Type" [0-9]* -> TYPE
    "Id-List"[\']* -> (ID ",")*
```

`Id`, `Type1`, `Type99`, `Id-List` e `Id-List''` são exemplos de variáveis declaradas.

2.4.2 Equações (equations)

Utilizando-se equações, a semântica dos elementos sintáticos definidos nas seções de sintaxe léxica e livre de contexto pode ser definida. Uma equação consiste de dois *open terms*, i.e., duas seqüências de caracteres que podem ser processadas através das especificações sintáticas [vdBBK03]. *Open terms* podem conter variáveis como marcações coringa que representam determinado tipo de estrutura sintática da linguagem objeto. Abaixo mostramos as quatro formas possíveis de equações:

```
[TagId]  L = R
[TagId]  L = R when C1, C2, ...
[TagId]  C1, C2, ... ===> L = R
[TagId]  C1, C2, ...
         ==================
              L = R
```

onde `L` e `R` são *open terms* e C_i, $i \in \mathbb{N}$, são condições que devem ser verdadeiras para que a equação também seja. L não pode ser a ocorrência de uma única variável. Estas formas se diferem apenas sintáticasmente, todas elas são semanticamente equivalentes.

As equações são executadas como regras de reescrita, i.e., a equação `L = R` é vista como uma regra que reescreve `L` como `R`. Assim, um termo inicial (que está sendo processado) é reduzido até que uma forma normal (forma irredutível) seja encontrada. Nesta situação, o resultado das reduções corresponde à semântica do programa recebido como entrada. A seleção do próximo (sub)termo a ser reduzido é feita utilizando a estratégia *leftmost-innermost*, i.e., escolhe-se o primeiro (sub)termo que ocorre da esquerda para direita em um caminhamento em profundidade. Depois disso, o (sub)termo escolhido é utilizado para escolher qual a regra de redução deve ser

utilizada. Isso é feito tentando casá-lo com o lado esquerdo das equações, de cima para baixo. Quando a regra de redução é encontrada, as condições da equação são avaliadas. Se todas as elas forem satisfeitas, o (sub)termo é substituído pelo lado direito da regra, caso contrário outra regra é procurada.

Exemplo 2.7 :

```
module Fib
exports
    sorts INT
    context-free syntax
        "0" -> INT
        "s" "(" INT ")" -> INT
        "add(" INT, INT ")" -> INT
        "fib(" INT ")" -> INT
hiddens
    variables
        [xy][0-9]* -> INT
equations
    [1] add(s(x), y) = s(add(x,y))
    [2] add(0, y) = y

    [3] x = 0 ==> fib(x) = s(0)
    [4] fib (s(0)) = s(0)
    [5] fib(s(s(x))) =  add(fib(s(x)), fib(x))
```

O módulo acima define equações para a função $fib(x)$, que retorna o número fibonacci de índice x.

De maneira equivalente àquela apresentada pelo Exemplo 2.7, equações podem também ser utilizadas para especificar transformações de programas. Para especificar transformações, as funções de caminhamento são muito importantes.

2.4.3 Funções de caminhamento (*traversal functions*)

Apesar de serem parte da especificação sintática livre de contexto, as funções de caminhamento serão tratadas separadamente devido a sua aplicação no contexto de transformação de programas.À Através destas funções é possível se escolher entre várias ordens de caminhamento pelos termos que constituem os programas objeto. Para cada passo no caminhamento, é verificado se existe alguma equação a ser aplicada. Se existir, então ela é aplicada. Caso contrário, o caminhamento continua.

Exemplo 2.8 [Zaa01]:

```
module Stat2CONTINUE
imports STDCOBOL
exports
    context-free syntax
     "transform" "(" Program ")" -> Program {traversal(trafo)}
    variables
        <declarações de variáveis>
equations
    [1] transform(#Stat) = CONTINUE
```

O módulo acima mostra uma especificação que descreve como transformar todos os *statements* de um programa COBOL no comando `CONTINUE`. Na especificação, são importadas as especificações da linguagem COBOL (módulo STDCOBOL), em seguida, definida uma função de caminhamento `transform` e, finalmente, a equação de transformação. No exemplo foi omitida a declaração da variável `#Stat`.

À primeira vista, parece que funções de caminhamento não são importantes, já que é sempre possível simulá-las através de construções convencionais do ambiente, decompondo cada um dos termos da linguagem, mas em [vdBKV03] são destacadas algumas razões para a utilizá-las. Uma destas é o fato de que a construção de caminhamentos sem estas funções implicaria na necessidade de definição de inúmeras equações para tratar cada construção possível da linguagem manipulada.

Existem três tipos de funções de caminhamento [vdBKV03, vdBBK03, Zaa01]:

- transformadoras: declaradas através da sintaxe

  `f (`S_1`, ..., `S_n`) -> `S_1 **{traversal(trafo)}**,`

  caminham através do seu primeiro argumento. Possivelmente argumentos extra podem ser fornecidos.

- acumuladoras: declaradas através da sintaxe

  `f (`S_1`, `S_2`, ..., `S_n`) -> `S_2 **{traversal(accu)}**,`

  caminham através do seu primeiro argumento enquanto o segundo argumento mantém acumulado um valor. Para cada aplicação, o valor do segundo argumento é atualizado. Nas próximas aplicações o novo valor será utilizado.

- transformadoras e acumuladoras: declaradas através da sintaxe

  `f (`S_1`, `S_2`, ..., `S_n`) -> `S_1`#`S_2 **{traversal(accu, trafo)}**,`

são uma combinação dos dois casos anteriores. O resultado da aplicação é a nova árvore, substituindo o valor do primeiro argumento, e um novo valor para o acumulador.

É possível também definir estratégias de caminhamento: **bottom-up**, para um caminhamento em pós-ordem, **top-down**, para caminhamento em pré-ordem, **break**, para que o caminhamento ignore o ramo atual da árvore depois de uma aplicação de regra bem sucedida, e **continue**, para continuar o caminhamento normalmente mesmo após uma aplicação bem sucedida. Estas estratégias são definidas na declaração de função de caminhamento, veja o exemplo abaixo:

Exemplo 2.9 [vdBKV03]:

```
module Tree-inc
imports Tree-syntax
exports
    context-free syntax
     inc(TREE) -> TREE { traversal(trafo, bottom-up, continue)}
equations
    [1] inc(N) = N + 1
```

Este modulo mostra como percorrer uma árvore (a sintaxe de árvores é definida pelo módulo `Tree-syntax`) incrementando os inteiros desta árvore de uma unidade utilizando uma função de caminhamento transformadora, com as estratégias **bottom-up** e **continue**.

O ambiente ASF+SDF já foi utilizado na implementação de linguagens de domínio específico como fSDL ("*full Structure Definition Language*"), uma linguagem para descrição de estruturas de dados. Neste caso, ASF+SDF foi utilizado para compilar as definições de fSDL para C. Outra linguagem desenvolvida utilizando-se a ferramenta foi RISLA, uma linguagem para especificação de produtos financeiros. ASF+SDF foi também utilizado para prototipação de uma ferramenta para assegurar a segurança das linhas ferroviárias da Alemanha. Para garantir que semáforos instalados na malha ferroviária só se tornem verdes quando for realmente seguro, são utilizados dispositivos chamados de *Vital Processor Interlocking* (VPI). Estes elementos são programados utilizando-se a linguagem *Vital Logic Code* (VLC). Projetos para a verificação de segurança de VPIs utilizaram ASF+SDF para transformar programas VLC ou para traduzí-los em proposições que seriam verificadas utilizando-se verificadores de tautologias (*tautology checkers*) [BDK+96].

Assim como TXL, a principal crítica a ASF+SDF está relacionada à dificuldade no aprendizado da ferramenta. A justificativa para esta crítica é a mesma apresentada para aquela ferramenta, i.e., a dificuldade no seu aprendizado é conseqüência do fato de ASF+SDF ser uma nova linguagem, com novos conceitos e comandos, além do fato da mesma se enquadrar em um paradigma pouco popular (sistema de reescrita). De maneira equivalente a TXL, também não foram encontradas, na literatura revisada [Zaa01] [BDK+96] [vdBBK03] [vdBvDH+01] [dH03], informações sobre recursos ASF+SDF que permitam ao meta-programa interagir com o usuário.

Este fator pode prejudicar bastante a flexibilidade da ferramenta. Além disso, o suporte ao tratamento de exceções é quase ausente [dH03].

A linguagem também não permite que o processo de reescrita seja interrompido prematuramente para, por exemplo, reportar um determinado erro.

> *Para gerar uma mensagem de erro para o usuário é necessário substituir o programa original pelo texto da mensagem, o que pode não ser uma tarefa muito simples. Isso geralmente involve adicionar regras extras para extrair o texto descrevendo os erros encontrados do resto do resultado. [dH03]*

Por outro lado, ASF+SDF utiliza um *parser* GLR para reconhecer trechos de programa objeto, o que é uma melhoria em relação ao *parser* de TXL, apesar de que esta técnica necessita que sejam especificados filtros para resolver ambigüidades na gramática da linguagem objeto.

2.5 Stratego

> *Stratego [Vis00] é uma linguagem para especificação de sistemas de transformação de programas baseado em estratégias de reescrita. [Vis01]*

Um programa em Stratego é composto da especificação sintática da linguagem objeto, regras de transformação e estratégias de aplicação das regras.

Stratego utiliza SDF para permitir que meta-programadores especifiquem a GLC da linguagem objeto. Com esta é possível construir a árvore sintática do programa que será transformado. A sintaxe de descrição léxica e livre de contexto da linguagem objeto é muito parecida com a de ASF+SDF, veja um exemplo abaixo:

Exemplo 2.10 [Vis03]:

```
module Tiger-Statements
imports Tiger-Lexical
exports
    lexical syntax
        [a-zA-Z][a-zA-Z0-9]* -> Var
    context-free syntax
        Var ":=" Exp -> Exp {cons("Assign")}
        "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
        "while" Exp "do" Exp -> Exp {cons("While")}
        Var -> Exp {cons("Var")}
        Exp "+" Exp -> Exp {left, cons("Plus")}
        Exp "-" Exp -> Exp {left, cons("Minus")}
```

O módulo mostra as regras léxicas e livres de contexto da linguagem Tiger. O significado das seções **imports**, **exports**, **lexical syntax** e **context-free syntax** são os mesmos apresentados para ASF+SDF na Seção 2.4.

No exemplo acima foi utilizado um recurso de SDF não descrito anteriormente, os construtores (**cons**). Estes são utilizados para rotular a árvore sintática, indicando qual foi a produção utilizada para derivar o trecho de código representado pelo nodo em questão. Como já foi citado na Seção 2.4, SDF utiliza técnicas de GLR *parsing*.

Uma regra de transformação é descrita através da sintaxe:

```
L: l -> r where s
```

onde `L` é um *label*, `l` um padrão de casamento, `r` um padrão para reescrita e `s` a condição para a aplicação da regra. Os padrões `l` e `r` são descritos utilizando a sintaxe concreta da linguagem objeto. Nestes padrões podem ocorrer variáveis como marcações coringa, representando trechos de programas da linguagem objeto.

As estratégias são um recurso da linguagem que permitem ao programador um controle completo sobre a aplicação das regras de reescrita [Vis03]. Uma estratégia é definida pela composição de vários combinadores. Uma definição de estratégia da forma $f(x_1,x_2,\ldots,x_n) = s$ declara um operador f de aridade n como uma abstração para a estratégia s [Vis01], onde s é uma composição de operadores de estratégias primitivos ou previamente definidos. Veja um exemplo abaixo:

Exemplo 2.11 [Vis03]:

```
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))
```

O exemplo descreve estratégias de transformação de expressões lambda através da redução beta [Vis03]. Em **rules**, a regra de transformação `Beta` é definida. Segundo ela, uma aplicação `App(Abs(x, e1), e2)` é transformada através da aplicação da estratégia `<lsubs>`, definida no pacote `lambda-vars`, aos parâmetros `[(x,e2)]` e `e1`. São definidas também estratégias de redução: `simplify`, que aplica a transformação `Beta` através de um caminhamento *bottom-up*; `eager`, que reduz o argumento de uma função antes de aplicá-lo, mas não reduz dentro de abstrações; `whnf`, que reduz uma expressão para a *weak head normal form*, i.e., não normaliza dentro de abstrações ou dentro de argumentos. As ocorrências de `App` em **strategies** se tratam de utililzações do operador de congruência **App** que são automaticamente definidos por Stratego. Estes operadores serão explicados logo abaixo.

As definições de `simplify`, `eager` e `wnhf`, apresentadas no exemplo acima, utilizam combinadores de estratégias para definir estratégias mais complexas a partir de estratégias mais simples.

Os seguintes combinadores são disponibilizados: (i) seqüência, $\texttt{s}_1;\texttt{s}_2$, aplica $\texttt{s}_1$ e em seguida, $\texttt{s}_2$; (ii) escolha determinística, $\texttt{s}_1\texttt{<+}\ \texttt{s}_2$, tenta $\texttt{s}_1$, se falhar, tenta $\texttt{s}_2$; (iii) escolha não determinística, $\texttt{s}_1 + \texttt{s}_2$, o mesmo que `<+`, mas sem ordem de escolha; (iv) escolha com guarda, $\texttt{s}_1 < \texttt{s}_2 + \texttt{s}_3$, se $\texttt{s}_1$ falha, aplica $\texttt{s}_3$, senão $\texttt{s}_2$; (v) teste, `where(s)`, ignora a transformação se `s` falha; (vi) negação, `not(s)`, bem sucedida se `s` falha e (vii) recursão, `rec x(s)`, permite fazer chamadas à estratégia `s` dentro de sua própria definição através da utilização do nome `x`.

Os combinadores de estratégias agrupam estratégias que repetidamente aplicam transformações à raiz do termo [Vis00]. Para aplicar transformações nos subtermos de um termo são necessários operadores para caminhar através ele. Para isso, a linguagem define operadores de congruência. Para cada construtor `C` definido na especificação livre de contexto da linguagem objeto, uma estratégia $\texttt{C(s}_1\texttt{, s}_2\texttt{, ..., s}_n\texttt{)}$ é definida da seguinte maneira:

$$\texttt{C(s}_1\texttt{, s}_2\texttt{, ..., s}_n\texttt{) (C(t}_1\texttt{, t}_2\texttt{, ..., t}_n\texttt{)) = C(t}'_1\texttt{, t}'_2\texttt{, ..., t}'_n\texttt{)}$$

onde t_i' é o resultado da aplicação da estratégia s_i ao sub-termo t_i de t. Vale destacar a diferença entre **`C`** e `C`. No primeiro caso, **`C`** é o nome do operador de congruência, no segundo, `C` é o nome do construtor do termo `C`. Eles são equivalentes sintaticamente. Veja o exemplo abaixo.

Exemplo 2.12 :

Seja a seguinte especificação de uma termo do $\lambda - calculus$:

```
module lambda-vars
lexical syntax
    [a-zA-Z][a-zA-Z0-9]* -> Var
    <outras definições léxicas>
context-free syntax
    App(T, T) -> T {cons("App")}
    Abs(Var, T) -> T {cons("Abs")}
    Var -> T {cons("Var")}
```

Automaticamente, Stratego define um operador congruência **App** (de aridade 2), um **Abs** (de aridade 2) e um **Var** (de aridade 1). No Exemplo 2.11, o operador de congruência **App** foi utilizado na definição das estratégias `eager` e `whnf`.

Stratego disponibiliza também operadores genéricos de caminhamento para tornar estratégicas mais gerais. Apresentamos alguns deles abaixo:

- *caminho*, `n(s)`, que aplica `s` ao n-ésimo subtermo do termo que está sendo explorado;
- *todos*, `all(s)`, que aplica `s` a todos os subtermos do termo que está sendo explorado;
- *um*, `one(s)`, que aplica `s` de forma não determinística a um dos subtermos do termo que está sendo explorado;

Através destes operadores é possível definir estratégias de caminhamento como `bottomup` (*innermost-leftmost*) e `top-down` [Vis00].

```
bottomup(s) = rec x (all(x); s)
topdown(s) = rec x (s; all(x))
```

Como exemplos de aplicações construídas utilizando Stratego podemos citar: implementação de linguagens funcionais, otimizadores de fluxo de dados sobre árvores de sintaxe [OV02], implementação de compiladores de linguagens domínio específico [vW03], a implementação de *CodeBoost*, uma ferramenta para realizar transformações e otimizações em programas C++ [Bag03] [BKHV03], dentre outras.

A principal crítica a Stratego foi também citada para ASF+SDF e TXL: ela é uma linguagem completamente nova e que foi projetada como um sistema de reescrita, o que pode dificultar bastante seu aprendizado.

Stratego apresenta recursos para I/O, para controle de processos (comunicação entre processos) e *networking*. Por outro lado, não foram encontradas informações sobre recursos para tratamento de exceções.

2.6 JaTS

JaTS é um sistema que permite especificar transformações de tipos Java [CB01] [COS+01]. Diferentemente das linguagens apresentadas até o momento, JaTS só permite manipulação de programas de uma única linguagem objeto: Java. Cada transformação é composta por uma pré-condição e dois padrões de programas: o lado esquerdo e o lado direito da regra. Quando aplicada em um tipo t, o casamento entre t e o lado esquerdo da regra é verificado. Se ele é bem sucedido e a pré-condição é satisfeita, então o resultado da transformação é construído de acordo com o padrão do lado direito. De uma maneira geral, uma transformação JaTS é uma regra de reescrita com guarda. Os padrões em JaTS são descritos utilizando-se a própria sintaxe concreta de Java. Veja um exemplo abaixo.

Exemplo 2.13 [CB01]:

Lado esquerdo

```
class #C extends Object {}
```

Lado direito

```
class #C {}
```

A transformação acima remove a declaração de superclasse de uma declaração de classe cujo corpo é vazio.

Em um padrão, variáveis são utilizadas como representantes de estruturas sintáticas de Java. No Exemplo 2.13, a variável `#C` ocorre como representante de um identificador para o nome da

classe declarada. Elas podem ser especificadas com um tipo que define qual estrutura sintática ela representa, mas em alguns casos, como no exemplo anterior, isso não é necessário pois o tipo da mesma pode ser facilmente inferido. Veja um exemplo de transformação mais elaborada abaixo.

Exemplo 2.14 [CB01]:

Lado Esquerdo

```
class #C extends Object implements #if:Name {}
```

Lado Direito

```
class #C implements #if:Name{}
```

Transformação equivalente àquela apresentada no Exemplo 2.13, exceto pelo fato de que esta transforma tipos Java que possuam cláusula de declaração de interfaces implementadas.

Além de variáveis, padrões podem utilizar trechos opcionais delimitados pelos sinais de maior que e menor que (`< ... >`), veja um exemplo abaixo:

Exemplo 2.15 [CB01]:

Lado Esquerdo

```
class #C extends Object <implements #ifs:NameList>{}
```

Lado Direito

```
class #C <implements #ifs:NameList>{}
```

Este exemplo remove a cláusula de declaração de superclasse para tipos que possuam, ou não, declaração de interfaces implementadas.

A possibilidade de especificação de trechos opcionais é um recurso que aumenta o poder de expressão de padrões pois permite que eles sejam mais genéricos.

Quando utilizado do lado direito de uma regra, um trecho opcional só será reescrito se todas as ocorrências de variáveis delimitadas por `< ... >` possuírem valores, i.e., forem necessárias para que o casamento do lado esquerdo com o tipo a ser transformado seja bem sucedido.

Variáveis também podem ser utilizadas para representar conjuntos de estruturas de um mesmo tipo, veja os exemplos de transformações abaixo.

Exemplo 2.16 [CB01]:

Lado Esquerdo

```
class #C extends Object <implements #ifs:NameList>{
    #attr:FieldDeclaration;
}
```

Lado Direito

```
class #C <implements #ifs:NameList>{
    #attr:FieldDeclaration;
}
```

A transformação acima só pode ser aplicada a classes que possuam apenas um atributo, que será capturado pela variável **#attr**.

Lado Esquerdo

```
class #C extends Object <implements #ifs:NameList>{
    #attrs:FieldDeclarationSet;
    private #type #name;
    #attr:FieldDeclaration;
}
```

Lado Direito

```
class #C <implements #ifs:NameList>{
    #attr:FieldDeclaration;
    #attrs:FieldDeclarationSet;
}
```

Além de remover a declaração de superclasse da classe Java recebida como entrada, a transformação acima também remove a penúltima declaração de campo privado.

Neste exemplo a variável **#attrs** captura as declarações de campos que ocorrem antes daquele que será removido. **#attrs** é uma variável de conjunto, i.e., representa a ocorrência de várias estruturas de tipo **FieldDeclaration**.

Para dar mais flexibilidade às transformações, JaTS permite que no padrão do lado direito sejam especificadas declarações executáveis. Estas declarações utilizam a informação coletada pelas variáveis do lado direito para construírem novos valores, ou melhor dizendo, produzirem código não necessariamente idêntico ao anterior. Elas podem ser especificadas no lugar de qualquer construção sintática. Veja um exemplo abaixo:

Exemplo 2.17 [COS$^+$01]:

Lado Esquerdo

```
class #C extends #SC{
    #ATTR:FieldDeclaration;
}
```

Lado Direito

```
class #C extends #SC{
    private [[ #ATTR.getType() ]] fd;
```

```
    #ATTR:FieldDeclaration;
}
```

A transformação acima adiciona um novo atributo `fd` à classe a ser transformada. Esta classe deverá possuir apenas uma declaração de atributo. O tipo deste único campo é recuperado pela utilização do método `getType` na declaração executável **[[#ATTR.getType()]]** e utilizado como tipo para a declaração de um novo campo.

Seja a classe Java:

```
class ConcreteTest extends AbstractTest{
    private int code;
}
```

A aplicação da transformação sobre ele resulta na seguinte classe:

```
class ConcreteTest extends AbstractTest{
    private int fd;
    private int code;
}
```

São permitidas também declarações iterativas, que são utilizadas para gerar trechos de programa a partir de um mesmo padrão. De maneira equivalente às declarações executáveis, elas só podem aparecer no lado direito de uma transformação. Sua sintaxe é:

forall *<nome de variável>* **in** *<nome de variável que denota conjunto de estruturas>* **do**
    <corpo de declarações>
end

Uma declaração iterativa é composta por: uma variável que denota um conjunto de estruturas sintáticas, ds, um nome de variável d que receberá, a cada iteração, um valor do conjunto ds e um corpo de declarações c que possivelmente farão referência a esta variável. Para cada elemento de ds, uma cópia de c é inserida no local do código onde a declaração iterativa foi realizada, substituindo-se as ocorrências de d pelo elemento atual de ds. Um exemplo de utilização deste recurso é apresentado abaixo.

Exemplo 2.18 :

Lado Esquerdo

```
class Person extends Object <implements #ifs:NameList> {
   #attrs:FieldDeclarationSet;
   #mtds:MethodDeclarationSet;
}
```

Lado Direito

```
class Person <implements #ifs:NameList> {
    #attrs:FieldDeclarationSet;
    #mtds:MethodDeclarationSet;
    forall #a in #attrs do
        public #[[ #a.getFieldType() ]] #[[ (#a.getFieldName(0)).addPreffix("get") ]] () {
            return #[[ #a.getFieldName(0) ]];
        }
    end
}
```

A transformação deste exemplo, além de remover a declaração de superclasse para a classe `Person`, adiciona, para cada atributo declarado na classe, um método "get". Vale destacar a utilização da declaração iterativa para percorrer os valores da variável **#attrs** que captura as declarações de atributos dentro da classe `Person`.

O último elemento de JaTS a ser apresentado são as pré-condições. Estes especificam expressões booleanas que definem condições para que a transformação seja executada. Para o Exemplo 2.18 podemos, por exemplo, definir a pré-condição apresentada abaixo.

Exemplo 2.19 :

precondition `#attrs.size() > 0`

A pré-condição verifica se o conjunto **#attrs** possui, no mínimo, um elemento.

Nos Exemplos 2.17, 2.18 e 2.19 podem ser percebidas, em declarações executáveis, declarações iterativas e expressão de pré-condição, a utilização de métodos como `size`, `getType`, `getFieldName`, etc.. Estes métodos são definidos por JaTS. Existem outros métodos e operadores que podem ser utilizados na descrição do lado direito de uma regra de transformação.

JaTS possui um ambiente onde o usuário pode especificar e executar transformações, o *JaTS Work Environment*. Ele é um ambiente visual que permite especificar uma regra de transformação e escolher os tipos Java para que estas regras sejam aplicadas.

A principal crítica a JaTS está relacionada ao requisito multi-linguagens. JaTS só permite transformação de programas Java, o que limita bastante a ferramenta. Além disso, não foram encontradas informações sobre maneiras de se compor transformações para definir uma transformação de maior granularidade.

Outra crítica à ferramenta é que ela só permite a iteração sobre listas ou conjuntos de estruturas sintáticas. Além disso, não foram encontrados, na literatura revisada [CB01] [COS+01], exemplos de programas JaTS capazes de explorar e atualizar estruturas sintáticas mais complexas da linguagem Java como expressões e comandos. Foram encontradas apenas tranformações sobre tipos e membros de tipos (classes e interfaces) Java.

2.7 TAWK

TAWK é uma extensão da linguagem AWK para permitir casamento de padrões em árvores sintáticas. Ela permite manipulação de códigos C e MUMPS. É possível estendê-la para outras linguagens objeto, mas esta não é uma característica natural da linguagem, isso exigiria um grande esforço por parte do meta-programador. Um programa em TAWK é uma seqüência de pares padrão-ação [GAM96]. Um par padrão-ação é descrito de acordo com a seguinte sintaxe:

```
<expressão> "{" <ação> "}";
```

onde `<acao>` é um trecho de código C e `<expressao>` especifica um padrão de casamento.

O programa objeto é representado como uma árvore sintática e os pares padrão-ação especificados em um programa TAWK são testados na ordem em que aparecem, tomando um caminhamento *top-down* (pré-ordem) na árvore. Os nodos da árvore sintática armazenam, além das informações sobre o tipo de estrutura sintática representada por ele, valores que poderão ser referenciados dentro de um padrão.

TAWK também oferece bibliotecas para facilitar a manipulação de programas. Dentre os recursos oferecidos por elas estão funções para acessar atributos da árvore de sintaxe, iteradores, *printers* e tipos abstratos de dados (como listas, *Set*s e tabelas). A linguagem foi construída sobre a biblioteca Ponder [GA95] que oferece facilidades para construção de árvores de sintaxe abstrata e outros tipos de dados utilizados nas ações. Estender TAWK para outras linguagens objeto (que não C e MUMPS) implica em realizar adaptações em Ponder [GAM96].

Em sua forma mais simples, um padrão obedece a sintaxe `<tipo>:<expReg>:<variavel>`, onde `<tipo>` especifica qual o tipo de estrutura sintática o padrão representa, `<expReg>` é uma expressão regular com a qual o valor armazenado no nodo deve estar de acordo e `<variável>` é um nome de variável a qual a sub-árvore casada com o padrão será ligada. Padrões compostos também podem ser especificados:

1. `padrão`$_1$ `"|"` `padrão`$_2$: este padrão casa com árvores que estiverem de acordo com `padrão`$_1$ ou com `padrão`$_2$;

2. `padrão`$_1$ `padrão`$_2$: casa com árvores irmãs (i.e., árvores que possuem um mesmo nodo pai). A árvore mais a esquerda casa com `padrão`$_1$ e a árvore irmã imediatamente à direita com `padrão`$_2$;

3. `"(" padrão` c_1`,` c_2`, ...,` c_n `")"`: casa com uma árvore cuja raiz está de acordo com `padrão` e seus filhos casam com c_1, c_2, ..., c_n , nesta ordem. Todos os filhos devem casar;

4. `padrão`*: casa com zero ou mais ocorrências consecutivas (ocorrências irmãs) de sub-árvores em concordância com a expressão especificada;

5. `padrão`$^{+}$: casa com uma ou mais ocorrências consecutivas (ocorrências irmãs) de subárvores em concordância com a expressão especificada;

6. `padrão "?"`: casa com uma ou nenhuma ocorrência de árvore em concordância com a expressão especificada;

7. `"[" padrão "]"`: casa árvores que contêm `padrão` como subárvore;

8. `"<" padrão ">"`: utilizado para especificar que `padrão` tem maior precedência;

Veja abaixo um pequeno exemplo de padrão composto:

Exemplo 2.20 [GAM96]:

`(declaration:FUNCTION:$fdef`* `[expression:FUNCTION:$fcall]`*`)`

O padrão acima casa com todas as chamadas de funções que ocorrem dentro de uma declaração de função. Cada chamada será ligada a variável `$fcall` e a declaração será ligada à `$fdef`.

A linguagem também permite a definição de macros utilizando a mesma sintaxe da linguagem C, i.e., `#define <nome macro> <trecho programa>`. O significado de uma definição de macro em TAWK também é idêntico ao significado das definições de macros em C: todas as ocorrências de `<nome macro>` serão substituídas por `<trecho programa>`.

O exemplo abaixo ilustra com mais detalhes estes recursos e também detalhes sobre a a descrição de ações.

Exemplo 2.21 :

```
%{
    #define FDECL <declaration:FUNCTION | declaration:MACRO>
    #define CALL expression:FUNCTION
    String curCaller;
%}
FDECL:$fdef {
    curCaller = FunctionName($fdef);
}
[CALL:$fcall] {
    String called = CallName($fcall);
    if(curCaller != NULL) {
        printf("%s: %s\n"; curCaller, called);
    }
}
FDECL:$fdef {
    curCaller = NULL;
```

```
}
```

O exemplo imprime na saída principal o grafo de chamadas de funções de um programa objeto. Nele, são definidas macros, `FDECL` e `CALL`, e também a variável global, `curCaller`. Em seguida, são definidos três pares padrão-ação:

1. no primeiro, é utilizado um padrão que seleciona declarações de funções ou macros (observe a utilização do tipo `declaration` e das propriedades `MACRO` e `FUNCTION` na definição da macro `FDECL`). A ação ligada a este padrão recupera o nome da função referente à declaração selecionada utilizando `FunctionName`, uma rotina disponibilizada pela biblioteca de TAWK.
2. no segundo par, é especificado um padrão que seleciona chamadas de funções. A ação utiliza a função `CallName` disponibilizada pela biblioteca de TAWK para recuperar o nome da função que está sendo chamada no trecho de programa selecionado pelo padrão. Se `currCaller != NULL`, então uma mensagem é impressa na saída principal.
3. no último par, um padrão idêntico àquele especificado no primeiro par é utilizado, mas a ação diferente: ela faz `currCaller = NULL`.

Devido à ordem de verificação dos padrões, este programa imprime o grafo de chamadas na saída padrão. Para cada nodo que representa uma declaração de função, o casamento com o primeiro par padrão-ação é bem sucedido, e, conseqüentemente, o valor de `currCaller` é definido. Em seguida, o segundo par é também testado, coletando todas as chamadas de função que ocorrem abaixo do nodo que representa a declaração de função (observe que o padrão ocorre entre colchetes (`[...]`)). Para cada chamada de função encontrada é impressa uma mensagem na saída principal. Finalmente, o último par é verificado, o que fará com que `currCaller` perca seu valor.

Assim como no caso de JaTS, a principal crítica a TAWK é a falta de flexibilidade em relação às linguagens objeto possíveis de serem manipuladas. Por outro lado, o fato de permitir ações escritas em C, uma linguagem bastante popular, facilita o entendimento da ferramenta. Outra crítica a TAWK é o fato da estrutura dos seus meta-programas TAWK set muito rígida: eles devem ser especificados apenas como um conjunto de pares padrão-ação.

TAWK define uma ordem de caminhamento pela árvore sintática para realizar a verificação do padrões e execução das ações. A possibilidade de se descrever um padrão que explora uma árvore em busca da ocorrência de uma determinada estrutura (`"[ padrão ]"`) dificulta a compreensão de alguns meta-programas, veja o Exemplo 2.21.

2.8 SCRUPLE

SCRUPLE é uma ferramenta que permite utilização padrões de programas para realizar consultas em código C e PL/AS (uma variação da linguagem PL/1) [PP94]. A máquina de

casamento de padrões de SCRUPLE é capaz de encontrar, no programa objeto, fragmentos de código que estão de acordo com um determinado padrão de programa. Para realizar esta busca, a máquina transforma o programa objeto em uma árvore de sintaxe e o padrão em um autômato finito não determinístico. O casamento é encontrado pela execução do autômato tendo a árvore do programa objeto como entrada.

A linguagem de descrição de padrões de programa disponibilizada é uma extensão da linguagem objeto original permitindo a utilização de marcações coringa, i.e., marcações que representam um determinado tipo de estrutura sintática como expressões, *statement*s, *while*s. Veja exemplos abaixo:

Exemplo 2.22 [PP94]:

(a) O código abaixo mostra um padrão que pode ser utilizado para encontrar todos os *statements while* em que a condição de parada é uma expressão relacional de desigualdade cujo lado direito é a expressão "0".

```
while (# != 0) @;
```

(b) O código abaixo mostra um padrão que pode ser utilizado para encontrar a ocorrência de três *statements if* consecutivos.

```
if # @ ;
if # @ ;
if # @ ;
```

Em SCRUPLE, símbolos como $, @ e # são utilizados para representar os tipos de estruturas sintáticas representadas pela variável declarada. A Tabela 2.1 apresenta quais são os símbolos permitidos e o tipo de estrutura representado por cada um.

Estrutura sintática	**Símbolo para marcação coringa**
declaração	$d
tipo	$t
variável	$v
função	$f
expressão	#
statement	@

Tabela 2.1: Símbolos de SCRUPLE para marcações coringa e as respectivas estruturas sintáticas representadas [PP94]

São permitidos também símbolos para representar coleções de estruturas sintáticas de um determinado tipo. Coleções podem ser classificadas como conjuntos, quando a ordem dos itens representados não é relevante, ou seqüências, quando a ordem dos itens representados é relevante. A Tabela 2.2 apresenta os símbolos utilizados como marcadores nestes casos, o tipo de estrutura representado por cada um e também sua classificação (lista ou conjunto).

Estrutura sintática	Símbolo para marcação coringa	Classificação
declarações	$*d	Conjunto
variáveis	$*v	Conjunto
expressões	#*	Conjunto
*statement*s	@*	Seqüência

Tabela 2.2: Símbolos de SCRUPLE para marcações coringa de coleções, as respectivas estruturas sintáticas representadas e classificação como conjunto ou seqüência de estruturas [PP94]

Veja abaixo exemplos de utilização de marcações coringa de coleções.

Exemplo 2.23 [PP94]:

(a) O padrão abaixo representa uma chamada de função. Os argumentos são capturados pela marcação coringa #*;

```
$f(#*);
```

(b) O padrão abaixo representa um trecho de programa onde ocorrem três *statements if*, possivelmente com outros *statements* entre eles.

```
if # @ ;
@*;
if # @ ;
@*;
if # @ ;
```

(c) O padrão abaixo representa uma seqüência de declarações onde ocorre a declaração da variável `maxval` em alguma posição. Observe que a classificação de $* como conjunto (ver Tabela 2.2), implicou em uma interpretação diferente deste padrão: a declaração de `maxval` pode ocorrer em qualquer posição do programa.

```
int maxval;
$*d
```

Marcações coringa também podem ser nomeadas de acordo com a seguinte sintaxe: `<marcação coringa>"_"<name>`, onde `<name>` é uma seqüência de caracteres ou dígitos. Abaixo mostramos alguns exemplos de marcações coringa nomeadas.

Exemplo 2.24 [PP94]:

(a) O padrão abaixo descreve um trecho de programa onde ocorre uma declaração de variável, cujo nome é capturado pela marcação coringa `$v_1`, e em seguida um *statement* onde o valor desta é incrementado de uma unidade.

```
int $v_1;
$v_1=$v_1 + 1;
```

(b) O padrão abaixo representa um trecho de programa onde ocorre a troca entre os valores das variáveis capturadas por `$v_x` e `$v_y`, utilizando a variável capturada por `$v_tmp` como espaço de armazenamento temporário.

```
$v_tmp = $v_x;
@*;
$v_x = $v_y;
@*;
$v_y = $v_tmp;
```

A linguagem também oferece símbolos para permitir padrões mais expressivos:

- `@{{...{}...}}`, para expressar *statements* aninhados, com profundidade igual ao número de `{`;
- `@{**}`, para expressar *statements* com níveis arbitrários de aninhamentos;
- `@[sttm-type1 | sttm-type2 | ...]`, para expressar a opção entre um dos tipos de *statements*;
- `@<id1, id2,...>, #<id1, id2,...>, $f<id1, id2,...>`, para expressar o uso de algum dos identificadores `id1`, `id2`, ... na estrutura representada.

Ao contrário das linguagens anteriormente apresentadas, SCRUPLE só permite consultas em código, não disponibilizando recursos para transformação ou geração de programas, o que é uma limitação muito grande. A linguagem possui um ambiente que permite executar e navegar pelos resultados de uma consulta.

2.9 Comentários sobre as ferramentas apresentadas

Abaixo mostramos as principais características detectadas nas ferramentas apresentadas neste capítulo.

- Mecanismo de decomposição (*pattern matching*) e reconstrução (geração) de programas através da utilização de padrões;
- Geralmente utilizam a notação *by example* (ver Seção 2.9.3) para especificar padrões de programas;
- Podem ser extensões de linguagens já existentes, ou mesmo novas linguagens construídas especialmente para atender aos requisitos da meta-programação;
- As novas linguagens de meta-programação (criadas especialmente para este fim) são, comumente, definidas como sistemas de reescrita (ex. TXL, ASF+SDF, Stratego e JaTS).

- As linguagens que são concebidas como extensões de outras para atender os requisitos da meta-programação se encontram mais próximas do paradigma imperativo (ex. TAWK).
- Geralmente apresentam estruturas para descrição de meta-programas muito rígidas: pares padrão-ação (TAWK) ou pares padrão de casamento/padrão de reescrita (TXL, ASF+SDF, Stratego e JaTS).
- Todas as ferramentas apresentadas disponibilizam padrões de programas, variáveis para representar ou armazenar trechos de programas objeto, possuem ou permitem definir caminhamentos sobre o código objeto e possuem componentes responsáveis por armazenar as características específicas da linguagem objeto. É possível perceber que estes quatro recursos são essencias para as ferramentas. A falta de uma maneira para se referir a um trecho de programa objeto é muito limitante. Sem isso, não seria possível para o meta-programa modificar a ocorrência de uma determinada estrutura sintática, pois modificações necessitam de uma referência para o elemento que deve ser modificado.

  Caminhamentos sobre o código objeto também são muito importantes, pois meta-programas trabalham com estruturas sintáticas que são formadas pela composição de outras menores. Observe que, em uma linguagem de programação, declarações de variáveis são compostas por identificadores (tipo e nome da variável), declarações de métodos são compostas por identificadores (tipo de retorno, nome do método), parâmetros formais e o corpo do método. Para analisar ou modificar uma estrutura de granularidade fina (identificadores, literais, etc.) a partir de uma estrutura maior (declarações, unidades de compilação, etc.), por exemplo, é necessário uma maneira de explorar esta última, i.e., caminhar pela estrutura em busca do ponto a ser modificado.

  A falta de um componente que gerencie as informações da linguagem objeto também limitaria bastante a ferramenta. Não seria possível, por exemplo, converter um programa que se encontra no formato textual para a representação interna (árvores sintáticas).

  Já padrões de programas possuem um papel complementar, mas não menos importante. A sua classificação como um recurso essencial vem do fato de que a busca por um modelo de construção sintática é muito comum em meta-programas. Sem padrões de programas seria possível realizar tais buscas, mas esta tarefa seria muito complexa, fazendo com que o meta-programador gastasse muito tempo realizando verificações estruturais detalhadas na representação interna do meta-programa.

2.9.1 Os quatro elementos de meta-programação

A seção anterior apresentou quatro recursos essenciais de uma ferramenta de meta-programação: (i) **padrões de programa**, (ii) **variáveis para trechos de código objeto**, (iii) **caminhamentos em código** e (iv) **componentes que gerenciam características inerentes à linguagem objeto**. Vale destacar que, nas ferramentas apresentadas, este último

componente gerencia apenas as informações sintáticas sobre a linguagem objeto. Além destas, estes componentes poderiam também oferecer outras informações como regras de escopo, regras de tipagem de expressões, etc.. Outra maneira de oferecer tais informações seria através da disponibilização de bibliotecas construídas utilizando-se os recursos da meta-linguagem e componentes que gerenciam apenas características sintáticas da linguagem objeto.

Abaixo mostramos como cada um destes quatro componentes aparece implementado nas linguagens apresentadas:

- **TXL**: (i) funções ou regras de transformação utilizam padrões de programa para realizar transformações, (ii) através de variáveis (inseridas em padrões ou especificadas como parâmetros formais de regras ou funções de transformação) é possível referenciar um trecho de código objeto, (iii) o caminhamento no código é predefinido, não é possível modificá-lo e (iv) as especificações da sintaxe (seções **define**, **key**, ...) são componentes que capturam características inerentes à linguagem objeto;

- **ASF+SDF**: (i) padrões de programas ocorrem em equações, (ii) trechos de programa objeto são recebidos pelos programas e podem ser referenciados utilizando-se variáveis (inseridas em padrões de programas), (iii) a iteração no código pode ser especificada pelas funções de caminhamento e (iv) as especificações da sintaxe léxica e livre do contexto são as entidades que capturam detalhes sintáticos específicos da linguagem objeto.

- **Stratego**: assim como em ASF+SDF, (i) padrões de programas ocorrem em regras de reescrita, (ii) trechos de programa objeto são recebidos pelos programas e podem ser referenciados utilizando-se variáveis (inseridas em padrões de programas). Por outro lado, (iii) a iteração sobre o código é definida através de estratégias. (iv) Especificações da sintaxe léxica e livre do contexto são as entidades que capturam detalhes sintáticos da linguagem objeto.

- **JaTS**: (i) padrões de programas ocorrem em regras de transformação de programas Java, (ii) trechos de programa objeto podem ser referenciados dentro de declarações executáveis, declarações iterativas e pré-condições utilizando-se variáveis. É possível (iii) iteragir sobre o código objeto, de uma maneira bastante limitada, através da utilização de declarações iterativas. Como a linguagem foi projetada de maneira a transformar exclusivamente programas Java, (iv) as características inerentes à linguagem objeto são embutidas no sistema dando a ele pouca flexibilidade para manipular diferentes linguagens objeto.

- **TAWK**: (i) padrões de programa são utilizados na descrição de pares padrão-ação. Nos padrões podem ser definidas (ii) variáveis que serão utilizadas dentro do código das ações para manipular trechos de programas. (iii) O caminhamento no código é pré-definido e (iv) a biblioteca Ponder [GAM96] realiza o papel de componente responsável por gerenciar

características inerentes à linguagem objeto. Tornar TAWK multi-linguagens exige um redirecionamento desta biblioteca.

- **SCRUPLE**: apesar de não permitir geração e transformação de programas, esta linguagem disponibiliza (i) padrões de programa para expressar consultas e, (ii) através de variáveis é possível referenciar um trecho de código objeto. (iii) O caminhamento é pré-definido. (iv) As características inerentes à linguagem objeto são capturadas, principalmente, por *parsers* que são responsáveis por construir árvores sintáticas de programas objeto e por construir representações para padrões de programas descritos utilizando a sintaxe concreta da linguagem objeto. A linguagem é pouco flexível com relação à característica multilinguagens.

Os padrões de programa e referências/variáveis para trechos de código objeto são elementos sobre os quais as seguintes operações podem ser aplicadas: pode-se verificar o casamento entre um padrão e um trecho de código (operação casar), imprimir um trecho de código ou padrão (operação imprimir) e também definir o valor de uma variável de código. Por outro lado, os componentes que gerenciam características inerentes à linguagem objeto assumem um papel mais passivo, são apenas utilizados pelos outros elementos.

A flexibilidade da meta-linguagem em permitir a manipulação de múltiplas linguagens objeto está diretamente ligada a possibilidade de se escolher qual será o componente gerenciador das características inerentes à linguagem objeto será utilizado. Observe que ASF+SDF, Stratego e TXL permitem tal flexibilidade pela possibilidade de se especificar (ou importar) as regras sintáticas e léxicas da linguagem objeto juntamente com o meta-programa. Nos outros casos, isso não é possível, os gerenciadores de informações da linguagem objeto estão ligados às outras entidades de meta-programação e, para substituir tais componentes, modificações na estrutura da meta-linguagem devem ser realizadas.

Os quatro elementos apresentados nesta seção são muito importantes pois foram a base para a construção do ambiente de meta-programação proposto nesta dissertação.

2.9.2 Limitações das ferramentas

Abaixo listamos as principais críticas às ferramentas apresentadas.

- É comum construir linguagens completamente novas (TXL, ASF+SDF, Stratego/XT e JaTS) para atender aos requisitos da meta-programação, o que dificulta o aprendizado da ferramenta. Além disso, estas são projetadas como sistemas de reescrita, o que dificulta seu aprendizado e integração em processos de desenvolvimento orientados a objeto ou imperativos. Além disso, algumas destas ferramentas apresentam falhas quanto a disponibilização de recursos que são facilmente encontrados nas linguagens de programação de uso geral: bibliotecas de I/O, GUI, *multi-threading*, tratamento de exceções e comunicação entre processos. É interessante que estes recursos não sejam apenas disponibilizados, mas

também de fácil acesso pelo meta-programa mantendo assim a simplicidade e legibilidade do mesmo.

- TAWK, apesar de ser uma extensão de uma linguagem já existente, não é flexível o suficiente para atender ao requisito de permitir a manipulação de múltiplas linguagens objeto. A linguagem também define uma forma rígida de descrição de meta-programas (pares padrão-ação). A abordagem adotada por esta ferramenta torna a descrição de meta-programas pouco flexível e menos legível.

- As linguagens de meta-programação apresentadas não permitem a reutilização de padrões de programas. É interessante permitir o reuso destes elementos já que, em linguagens cuja gramática é muito extensa (como Java, C, C++, etc.), um padrão pode ser difícil de ser escrito.

- SCRUPLE só permite consultas em código objeto, não atendendo a todas as aplicações da meta-programação.

Devido às deficiências apresentadas acima, é necessário definir uma ferramenta de meta-programação baseada em um paradigma popular (OO ou imperativo), que disponibilize padrões de programa reutilizáveis, recursos básicos como I/O, GUI, *multi-threading*, *networking*, tratamento de exceções, etc.. É interessante também que esta ferramenta permita que meta-programas sejam descritos de maneira bastante flexível, sem impor uma estrutura rígida de programação.

2.9.3 Padrões *by example*: utilizando a sintaxe concreta da linguagem objeto

Um recurso que foi encontrado em todas as meta-linguagens apresentadas nas seções anteriores é a utilização da sintaxe concreta da linguagem objeto para descrever fragmentos de programas objeto, i.e., padrões de programas *by example* [Vis02, CS92].

Para disponibilizar este recurso, a sintaxe da meta-linguagem deve ser estendida para tornar possível a utilização de trechos de programas objeto como dados ou padrões de análise [Vis02]. Para isso, a meta-linguagem e a linguagem objeto devem ser ampliadas com um operadores de *quotation* e *antiquotation*, respectivamente [Vis02, She01]. Esse mecanismo (*quotation/antiquotation*) é chamado de *quasi-quote* [She01]. Os operadores são utilizados para "chavear" entre descrições do meta-programa e de trechos do programa objeto. Em um meta-programa, fragmentos de programa objeto são marcados com operadores de *quotation* e, em uma descrição de um programa objeto, trechos de meta-programa são marcados com operadores de *antiquotation* [Vis02, She01].

As linguagens apresentadas nas seções anteriores utilizam apenas *antiquotation* pois os padrões de programa ocorrem em partes específicas do meta-programa, ou melhor dizendo, padrões não podem ocorrer em qualquer parte do código. SCRUPLE utiliza marcadores \$, %, e # como *antiquotation* para meta-variáveis. No caso de ASF+SDF e Stratego, o usuário define suas

próprias regras de declaração de meta-variáveis, o que evita a necessidade e um marcador de *antiquotation* específico. Podemos dizer que nestes casos são definidas regras de *antiquotation*. No caso de TXL, o marcador de *antiquotation* é "`[ t ]`", onde `t` é um tipo sintático. JaTS utiliza o marcador # para declarações de meta-variáveis dentro de padrões de programas objeto.

É difícil implementar uma meta-linguagem que permita trabalhar com diferentes linguagens objeto utilizando as técnicas de *parser* tradicionais. Os geradores de *parser* baseados nessas técnicas são restritos às gramáticas livres do contexto pertencentes classes LR e LL, que não são fechadas em relação à operação de composição [Vis02]. Isso quer dizer que: sejam duas gramáticas $G \in LR(k)$ e $G' \in LR(k')$, onde $k, k' \in \mathbb{N}$, uma gramática G'' resultante da composição de G e G' pode não ser $LR(k)$, nem $LR(k')$. A conseqüência desta propriedade é uma grande dificuldade em se definir uma maneira automática de se estender uma meta-linguagem para permitir que trechos de programas objeto sejam utilizados dentro de um meta-programa. Esta mesma dificuldade ocorre na extensão da linguagem objeto para permitir a descrição de trechos de meta-programas dentro de fragmentos de programas objeto. Uma maneira de resolver este problema é utilizar técnicas GLR (*generalized LR parsing*) que permitem o reconhecimento de qualquer gramática livre do contexto [Kli03, vdBSVV02, vdBSV98]. Como já foi citado ASF+SDF e Stratego utilizam tecnologia GLR.

2.10 Comentários finais

Neste capítulo apresentamos algumas das principais linguagens de meta-programação atuais. TXL, ASF+SDF e Stratego são linguagens de meta-programação definidas como sistemas de reescrita. A nossa principal crítica a estas linguagens é a dificuldade no aprendizado pois sistemas de reescrita não são muito populares. Além disso, algumas delas não oferecem recursos que são muito comuns em linguagens de uso geral: bibliotecas de I/O, GUI, *multi-threading*, tratamento de exceções e comunicação entre processos.

JaTS é uma linguagem de transformação de programas. A principal deficiência desta linguagem é a falta de flexibilidade quanto as linguagens objeto possíveis de serem manipuladas. Além disso, a ferramenta não possui uma maneira de se compor transformações para se obter transformações de granularidade maior.

Foi apresentada também uma linguagem concebida a partir de visões mais imperativas da meta-programação, TAWK. Nossas críticas a ela são a definição de uma forma de estruturação de meta-programas muito rígida e a falta de flexibilidade quanto as linguagens objeto possíveis de serem manipuladas.

A linguagem SCRUPLE, apesar de permitir a análise de programas baseada em padrões de programa, não permite geração e transformação. As limitações da linguagem mostram que, apesar de necessários, padrões de programas não são suficientes para atender a todos os requisitos da meta-programação.

Pela análise das deficiências destas ferramentas é possível perceber a necessidade de um ambiente de meta-programação baseado em um paradigma popular, que disponibilize os recursos comumente encontrados em linguagens de uso geral, disponibilize padrões de programa *by example* reutilizáveis, seja adaptável para diferentes linguagens objeto e que, idealmente, seja flexível quanto a estrutura de especificação do meta-programa.

Observando-se as características comuns das ferramentas apresentadas foi possível definir quatro entidades básicas que capturam a essência da meta-programação: padrões de programa, referências/variáveis para trechos de código objeto, caminhamentos em código e componentes que gerenciam características inerentes à linguagem objeto. Estes quatro elementos foram a base para a construção do ambiente de meta-programação proposto nesta dissertação.

Capítulo 3

MetaJ: a solução proposta

3.1 Introdução

MetaJ é um ambiente para meta-programação baseado no paradigma orientado a objetos. Ele disponibiliza padrões de programa *by example* reutilizáveis e permite a manipulação de múltiplas linguagens objeto. O ambiente foi concebido como a implementação de classes Java para representar as quatro entidades essenciais para meta-programação apresentadas na Seção 2.9.1: (i) padrões de programa, (ii) variáveis para trechos de código objeto, (iii) caminhamentos em código e (iv) componentes que gerenciam características inerentes à linguagem objeto.

Esta decisão de projeto e implementação trouxe dois grandes benefícios para a ferramenta:

- meta-programas em MetaJ nada mais são que programas Java que utilizam estas abstrações. Isso permite que a descrição de meta-programas seja bastante flexível pois os recursos do ambiente podem ser instanciados e utilizados em qualquer parte do meta-programa, sem a necessidade de uma estrutura fixa de especificação do mesmo;
- meta-programas MetaJ podem utilizar livremente os recursos básicos de Java, como I/O, GUI, *multi-threading*, *networking* e tratamento de exceções.

Na seção que se segue, serão apresentados, juntamente com vários exemplos, cada um dos recursos MetaJ.

3.2 O ambiente MetaJ

Os quatro elementos de meta-programação apresentados na Seção 2.9.1 foram realizados em Java de forma a não modificar as características desta linguagem, o que garante flexibilidade na descrição dos meta-programas, bem como a facilidade na compreensão dos mesmos. Estes recursos foram implementados como abstrações (como classes Java) que oferecem serviços para o meta-programa que os utiliza. São elas:

- **Referências-p**: abstrações que armazenam trechos de programas objeto. Elas escondem a representação interna do código armazenado e oferecem apenas operações que garantem a consistência sintática deste código. O nome referências-p vem de *referências para programas* (em inglês, *p-references – program references*). Maiores detalhes sobre esta abstração serão apresentados na Seção 3.3.

- ***Templates***: abstrações que encapsulam padrões de programa *by example*. Ver maiores detalhes na Seção 3.5.

- **Iteradores**: abstrações que encapsulam o estado de um caminhamento por um programa.

- ***Plug-ins***: abstrações que encapsulam características inerentes à linguagem objeto. Geralmente estes elementos não são acessados diretamente pelos meta-programas, mas, apesar disso, eles são de grande importância já que implementações das abstrações citadas nos itens anteriores necessitam das informações mantidas por eles para garantir as propriedades das operações disponibilizadas. Para cada linguagem objeto manipulada, um *plug-in* contendo informações sobre ela deve ser fornecido.

  Plug-ins podem ser inseridos ou removidos do ambiente quando necessário. Por esta razão, MetaJ atende ao requisito de possibilitar que múltiplas linguagens objetos sejam manipuladas. Todo *plug-in* possui um nome para identificá-lo dentre os outros registrados no ambiente. As outras abstrações utilizam este nome para procurar informações sobre a linguagem objeto que está sendo manipulada.

  Templates e referências-p são abstrações que precisam de informações que dependem da linguagem objeto, por isso é necessário informar a elas o nome do *plug-in* onde tal informação poderá ser encontrada, veja maiores detalhes nas seções 3.3 e 3.5. Como, geralmente, existe apenas um *plug-in* para cada linguagem objeto, o nome do *plug-in* define a linguagem objeto que está sendo manipulada pela abstração.

A Figura 3.1 mostra o relacionamento entre estes elementos. Os *templates* e as referências-p utilizam um identificador para localizar o *plug-in* que gerencia as informações sobre a linguagem objeto que está sendo manipulada. Os iteradores não se relacionam com os *plug-ins*. Criados a partir de referências-p, eles oferecem operações para caminhar pelo trecho de programa objeto encapsulado por ela. Para cada construção sintática alcançada pelo iterador, uma nova referência-p para este trecho de código pode ser criada.

Plug-ins é a abstração mais importante. Eles gerenciam informações sintáticas que são acessadas pelas referências-p e pelos *templates* durante a execução de um meta-programa. Estas informações são extraídas diretamente da gramática livre de contexto que descreve a sintaxe da linguagem objeto. Na Seção 3.6 será mostrado como um *plug-in* é construído. No momento, o importante é ressaltar que a gramática da linguagem objeto utilizada na construção do *plug-in* influencia diretamente na forma de se trabalhar com trechos de programas da linguagem objeto.

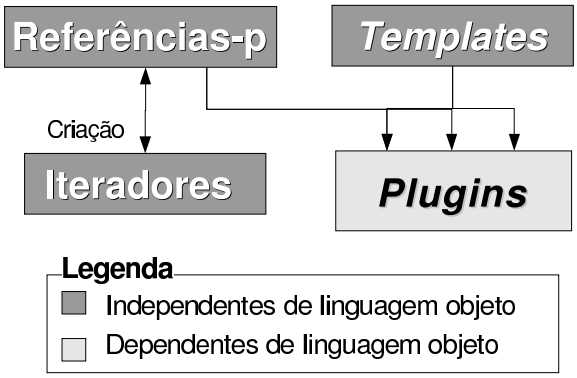


Figura 3.1: Relacionamento entre as abstrações de MetaJ.

Para que seja possível apresentar exemplos de utilizações de referências-p, iteradores e *templates*, as seções e os capítulos que se seguem supõem a existência de um *plug-in* para a linguagem Java registrado no ambiente com o nome "`java`". A gramática utilizada na sua criação é apresentada no Apêndice C. Vale destacar que, nos exemplos que serão apresentados, serão manipulados trechos de programas Java. Isso quer dizer que Java será tanto a meta-linguagem quanto a linguagem objeto.

Dentre as informações mantidas pelos *plug-ins*, a mais importante, e sem a qual não será possível tratar das outras abstrações, é a tipagem de trechos de programas. Em MetaJ, cada trecho de código possui um tipo sintático que corresponde ao não-terminal da gramática da linguagem objeto que o deriva. Veja um exemplo abaixo:

Exemplo 3.1 :

Seja a gramática $G = (T, N, P, R)$, onde:

```
T = {1,2,..., 9, 0, +, -};
N = {Exp, Num};
e R é o conjunto das seguintes regras
  Exp → Exp + Num;
  Exp → Exp - Num;
  Exp → Num;
  Num → Num 1;
  Num → Num 2;
  ...
  Num → Num 9;
  Num → Num 0;
```

O trecho de código `56` possui tipo `Num`. O trecho `556 + 36` possui tipo `Exp`, onde `556` e `36` possuem tipo `Num`. Vale destacar que trechos de código podem possuir sub-trechos que também devem ser tipados.

Geralmente, os *plug-ins* utilizam um símbolo especial como `#, %, ^` e `@` como prefixo dos tipos sintáticos da linguagem objeto. Desta maneira, os tipos sintáticos para a gramática do Exemplo 3.1 seriam representados por um *plug-in* como `#Exp` e `#Num` ou mesmo como `^Exp e ^Num`, ao invés de `Exp` e `Num`. O *plug-in* para a linguagem Java, citado anteriormente, utiliza o símbolo "`#`" como prefixo para os tipos sintáticos.

Além disso, *plug-ins* também são responsáveis por realizar a classificação dos tipos sintáticos e por implementar as propriedades sobre os tipos sintáticos da linguagem objeto. Esta responbilidade é dada a eles devido ao fato de que eles são os componentes que possuem acesso direto às informações sintáticas da linguagem objeto. Quando necessário, as outras abstrações do ambiente pedem para que o *plug-in* classifique um determinado tipo sintático ou verifique uma propriedade específica. Esta classificação influencia no significado de algumas das operações que serão apresentadas nas seções seguintes. Devido a sua importancia, a próxima seção tratará especificamente das regras de classificação de tipos sintáticos (não-terminais da linguagem objeto) e das suas propriedades.

3.2.1 Classificação e propriedades de tipos sintáticos

Seja gramática $G = (N, T, R, P)$ da linguagem objeto utilizada por um *plug-in*, onde N é o conjunto de não-terminais, T o conjunto de terminais, R as regras de derivação e P um símbolo não-terminal de partida, os seus não-terminais (e conseqüentemente as estruturas sintáticas da linguagem objeto) podem ser classificados como:

- **lista uniforme**: é um não-terminal $A \in N$ que possui recursão à direita ou à esquerda (mas não ambas) e que todas as regras em que A ocorre do lado esquerdo são de uma das formas: $A \rightarrow B\ A$, $A \rightarrow A\ B$ ou $A \rightarrow B$, onde $B \in (N \cup T)$. Mais formalmente, $A \in N$ é um não-terminal lista uniforme se, e somente se, $\exists B \in (N \cup T)\ (\forall r \in R\ (left(r) = A \Rightarrow r \in \{A \rightarrow B\ A, A \rightarrow A\ B, A \rightarrow B\}))$, onde $left(r)$ é uma função que retorna o lado esquerdo de uma regra r;

- **lista não uniforme**: é um não-terminal que possui recursão a direita ou a esquerda (mas não ambas), mas que não é uma lista uniforme;

- **simples**: são os não-terminais que não são listas, i.e., não-terminais que não possuem recursão à esquerda ou à direita (mas podem possuir ambas) e que não deriva, vazio diretamente (λ).

- **opcional**: é um não-terminal que pode derivar vazio (λ) diretamente. $A \in N$ é um terminal opcional se, e somente se, $\exists r \in R\ (r = A \rightarrow \lambda)$;

No Apêndice D pode ser encontrada a classificação de cada não-terminal da gramática da linguagem Java que foi utilizada pelo *plug-in* "`java`". Esta gramática é apresentada no Apêndice C.

Existe uma restrição quanto a gramática G utilizada pelo *plug-in*. Seja $L \in N$ classificado como lista, cujas regras de derivação são $L \rightarrow L\ M$ (ou $L \rightarrow M\ L$) e $L \rightarrow N$, onde $M\ e\ N \in (N \cup T)^*$, então não deve ser possível derivar λ (*string* vazio) a partir de M ou N, i.e., as derivações $M \stackrel{*}{\Rightarrow} \lambda$ e $N \stackrel{*}{\Rightarrow} \lambda$ não devem ser possíveis. A presença de tais estruturas sintáticas faria com que a gramática de padrões resultante do processo de geração do *plug-in* (ver Seção 3.6.1) deixasse de pertencer à classe das GLCs LALR.

Quando um não-terminal (tipo sintático) é classificado, a estrutura sintática derivável através dele também está sendo classificada. Vale destacar que os termos "tipo sintático da linguagem objeto", "não-terminal da gramática da linguagem objeto" e "estrutura sintática da linguagem objeto", apesar de poderem, em certos momentos, ser utilizados indiscriminadamente, possuem diferenças sutis. "Não-terminal" é um termo utilizado quando o contexto é a gramática da linguagem e não a linguagem em si. Neste último caso, o termo "estrutura sintática" é mais adequado. No contexto de MetaJ, "tipo sintático" é a terminologia adotada para se falar sobre uma estrutura sintática da linguagem objeto. O termo "construções sintáticas" será utilizado para falar sobre as ocorrências de uma determinada estrutura sintática, ou melhor dizendo, das instâncias de um determinado tipo sintático. O trecho de programa Java "`int var1 = 0`", por exemplo, é uma construção sintática da estrutura sintática *declaração de variável*, assim como "`import java.util.*;`" é uma construção sintática da estrutura sintática *declaração de importação*.

Além de classificá-los, é possível também definir duas propriedades sobre tipos sintáticos. Sejam A e B não-terminais de uma gramática:

- **convertibilidade de tipos**: O tipo sintático `A` é convertível no tipo `B` se, e somente se, $B \stackrel{+}{\Rightarrow} A$, onde $A\ e\ B \in N$, i.e., se o não-terminal B deriva A em um ou mais passos.
- **compatibilidade de tipos**: Existem duas formas de compatibilidade entre tipos `A` e `B`:
  - **substituição**: `A` e `B` são compatíveis para substituição (`A` = `B`) sse $A = B$. A compatibilidade de substituição é uma relação de equivalência entre os tipos. Dois trechos de programa cujos tipos são equivalentes podem ser substituídos sem perda da correção sintática do código onde a substituição foi realizada;
  - **adição**: `A` é compatível para adição em `B` sse B é lista uniforme e A é item da lista, i.e., B é definido pelas regras: $B \rightarrow A\ e\ B \rightarrow BA$ ou $B \rightarrow A\ e\ B \rightarrow AB$.

Tipos sintáticos, classificação de tipos sintáticos e as duas propriedades apresentadas acima são inerentes ao ambiente MetaJ. Elas foram colocadas sobre responsabilidade de *plug-ins* para tornar possível a manipulação de múltiplas linguagens objeto.

Nas seções que se seguem serão apresentadas de forma mais detalhada cada uma das abstrações de MetaJ. As classificações e propriedades de tipos sintáticos apresentadas anteriormente são de grande importância e serão citadas constantemente durante a apresentação das abstrações do ambiente.

3.3 Referências-p

Com o objetivo de oferecer uma maneira simples de se manipular e armazenar trechos de código da linguagem objeto, MetaJ disponibiliza referências-p. Estas são objetos que encapsulam trechos de código da linguagem objeto e oferecem operações que garantem a consistência sintática do código armazenado. Por razão desta propriedade, MetaJ atende ao requisito de garantir a correção sintática dos programas objeto, apresentado na Seção 2.2. Toda referência-p possui um tipo que corresponde a estrutura sintática da linguagem objeto que ela pode armazenar. Os métodos disponibilizados pela abstração utilizam este tipo para garantir a correção sintática do código armazenado.

Em MetaJ, referências-p podem ser criadas livremente, em qualquer parte do meta-programa pela invocação do método estático `createPReference` da classe `metaj.abstractions.MetaJSystem`. Nesta chamada devem ser informados o nome do *plug-in* que deve ser utilizado e o tipo da referência-p, veja o exemplo abaixo:

Exemplo 3.2 :

```
package meusTestes;
import metaj.framework.MetaJSystem;
import metaj.framework.abstractions.reference.PReference;
public class Main {
    public static void main (String args[]){
        ...
        PReference pd = MetaJSystem.
                        createPReference("java","#package_declaration");
        ...
    }
}
```

A referência-p `pd` foi criada com o tipo `#package_declaration`, por isso ela poderá armazenar uma declaração de pacote. Um exemplo de trecho de código que pode ser armazenado por ela é "**package** `meuPacote`" ou mesmo "**package** `eds.arvoresBinarias`".

De maneira equivalente àquela realizada no exemplo anterior, podem ser criadas referências-p de vários outros tipos:

Exemplo 3.3 :

```
package meusTestes;
// importações necessárias
public class Main {
    public static void main (String args[]){
        ...
        PReference id = MetaJSystem.
```

```
                        createPReference("java","#import_declaration"),
            id2 = MetaJSystem.createPReference("java","#import_declaration"),
            ids = MetaJSystem.createPReference("java","#import_declarations"),
            td = MetaJSystem.createPReference("java", "#type_declaration"),
            sttm = MetaJSystem.createPReference("java", "#statement"),
            expr = MetaJSystem.createPReference("java", "#expression");
        ...
    }
}
```

No meta-programa acima foram criadas as referências-p:

- `id` e `id2` do tipo `#import_declaration`, capazes de armazenarem trechos de código Java correspondentes a importações como "**import** `eds.arvoresBinarias.ArvoreSBB;`" ou mesmo "**import** `eds.arvoresBinarias.*;`";
- `ids` do tipo `#import_declarations`, capaz de armazenar trecho de código Java correspondentes a uma sequência de importações como:

  ```
  import eds.arvoresBinarias.ArvoreSBB;
  import eds.arvoresBinarias.*;
  ```

- `td`, do tipo `#type_declaration`, para armazenar declarações de tipo como as declarações de classe e interface descritas abaixo:

  ```
  class A implements I{
      public A () {}
      public void metodo1 (){ return; }
  }
  ```
  ou
  ```
  public interface I {
      public void metodo1 (){ return; }
  } ;
  ```

- `sttm` do tipo `#statement`, capaz de armazenar *statements* Java como "`i++;`" e "`cliente.setNome("José");`";
- `expr` do tipo `#expression`, que armazena expressões como "`5 + 1`" e "`cliente.getNome()`".

É possível também criar referências-p sem tipo. Estas poderão receber qualquer valor, i.e., podem armazenar qualquer tipo de código. Abaixo é apresentado um exemplo de criação de referência-p não tipada.

Exemplo 3.4 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        ...
        PReference generic = MetaJSystem.createPReference();
        ...
    }
}
```

Na criação da referência-p `generic` não é necessário informar nem o nome do *plug-in*, nem o tipo da referência-p.

Como pode ser percebido nos exemplos anteriores, referências-p são manipuladas por meio da interface `metaj.framework.abstractions.reference.PReference`, que define os métodos disponibilizados pela abstração. Estes método serão apresentados no resto desta seção. A Tabela 3.1 apresenta um resumo destes métodos.

Método	**Funcionalidade**

`set`
`add`
`remove`
`replace`
	Modificar o valor
	de uma referência-p

`match`, `matchFile` e `matchStream`
`equals`
`hasType`
`isComposedBy`
`contains`
	Realizar comparações e verificações
	sobre os valores armazenados
	por referências-p

`toString`, `toFile` e `toStream`
`duplicate`
`get`
`getSize`
	Converter para outros formatos, duplicar
	ou recuperar valores e informações

`getIterator`	Retornar um iterador para o trecho de código armazenado pela referência-p

Tabela 3.1: Métodos da interface `PReference`

É possível definir o valor de uma referência-p pela utilização dos métodos `set` da interface `PReference`. Veja exemplos de utilização abaixo:

Exemplo 3.5 :

```
package meusTestes;
// importações necessárias
```

```
import metaj.framework.abstractions.exceptions.*;
public class Main {
    public static void main (String args[]) throws ParseException{
        ...
        id.set("import eds.arvoresBinarias.ArvoreSBB;");
        ids.set("import eds.arvoresBinarias.ArvoreSBB; " +
                "import eds.arvoresBinarias.*;")
        td.set("class A implements I{ " +
               "public A () {} " +
               "public void metodo1 (){ return; } " +
               "}");
        sttm.set("i++;");
        expr.set("5 + 1");
        id2.set(id);
        id.setNull();
        generic.set(id2);
        ...
    }
}
```

O método `set` é polimórfico, pode receber como parâmetro tanto um *string* quanto uma referência-p. No primeiro caso, o trecho recebido é convertido para uma representação interna e passa a ser o valor armazenado pela referência-p. Se o *string* não corresponder a um trecho de programa sintaticamente correto e em concordância com o tipo da referência-p, uma exceção do tipo `ParseException` é lançada. Isso aconteceria, por exemplo, se o trecho "`5 + 1`" fosse atribuído a referência-p `td`, pois "`5 + 1`" não é do tipo `#type_declaration`. Para construir a representação interna do programa que está sendo recebido como parâmetro *string*, a referência-p utiliza seu tipo interno para decidir qual estrutura sintática será processada. Por esta razão, se este método for utilizado com referências-p não tipadas será impossível construir a representação interna e, por causa disso, um erro `MetaJError` ocorrerá.

No segundo caso, a referência-p passa a compartilhar com aquela recebida como parâmetro o mesmo trecho de código. Esta versão de `set` pode ser utilizada para atribuir valores a referências-p não tipadas. No Exemplo 3.5 é atribuído à `generic`, que não possui tipo (veja a declaração no Exemplo 3.4), o mesmo valor armazenado pela `id2`. Após esta atribuição, a referência-p anteriormente não tipada adquire um tipo, o mesmo daquela que foi recebida como parâmetro pelo método `set`.

Existem ainda duas versões adicionais do método `set`: `set(java.io.InputStream in)`, que permite que o valor de uma referência-p seja definido a partir do conteúdo de um objeto `InputStream` recebido como parâmetro, e `setFile(String fileName)`, que permite definir o valor da referência-p como o conteúdo do arquivo de nome `fileName`. Existe também o método `replace` que possui uma semântica bastante parecida com `set`. Ele também redefine o valor da

referência-p. A diferença entre estes dois métodos será apresentada na Seção 3.4. É importante ressaltar que o método `replace` só pode ser aplicado em referências-p cujo tipo é classificado como simples ou lista não uniforme. Ele não é aplicável a referências-p sem tipo ou cujo tipo é lista simples.

A uma referência-p pode ser atribuído, através do método `set`, qualquer trecho de código cujo tipo é convertível para tipo da referência-p. Finalmente, o método `setNull` remove o valor armazenado por ela. Este método **não** faz com que ela perca seu tipo.

Apesar de necessárias, operações de atribuição de valores não são suficientes para dar grande flexibilidade para o meta-programador. Por isso, a interface `PReference` define também os seguintes métodos:

- `boolean equals(PReference r)`: verifica se o trecho de código armazenado é o mesmo daquele armazenado pela referência-p `r` recebida como parâmetro. Não basta que o código seja lexicamente equivalente, ele deve ser o mesmo fisicamente. Abaixo pode ser visto um exemplo de utilização deste método:

  Exemplo 3.6 :

  ```
  ...
  public class Main {
      public static void main (String args[]) throws ParseException{
          ...
          id.set("import eds.arvoresBinarias.ArvoreSBB;");
          id2.set("import eds.arvoresBinarias.ArvoreSBB;");
          System.out.println(id.equals(id2));// resultado "false"
          id.set(id2);
          System.out.println(id.equals(id2));// resultado "true"
          ...
      }
  }
  ```

  Na primeira utilização do método `equals`, apesar dos trechos de código serem sintaticamente equivalentes, o resultado é `false`, já que eles não são a mesma importação.

  Na segunda utilização, o resultado é `true` já que o método `set` utilizado na linha anterior fez com que as duas referências-p `id` e `id2` compartilhassem o mesmo trecho de código.

- `boolean match (...)`: assim como o `set`, este também é um método polimórifico podendo receber como parâmetro uma referência-p, um *string*, um *stream* de entrada (`java.io.InputStream`) ou um arquivo. Independentemente do caso, ele verifica se o código armazenado pela referência-p é sintaticamente equivalente àquele recebido como parâmetro.

Exemplo 3.7 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        ...
        id.set("import eds.arvoresBinarias.ArvoreSBB;");
        id2.set("import eds.arvoresBinarias.ArvoreSBB;");
        System.out.println(id.match(id2));// resultado "true"
        id.set("import eds.arvoresBinarias.*;");
        System.out.println(id.match(id2));// resultado "false"
        ...
    }
}
```

Na primeira utilização do método `match`, apesar de cada referência-p armazenar uma ocorrência da importação em questão, elas são ocorrências sintaticamente equivalentes, o que faz com que resultado da operação seja `true`.

Na segunda utilização, o resultado é `false` pois a utilização do método `set` na linha anterior fez com que a referência `id` assumisse um valor que não é sintaticamente equivalente àquele armazenado pela `id2`.

- `boolean hasType (String t)`: verifica se a referência-p é do tipo recebido como parâmetro. Este tipo é especificado como um *string* e deve ser prefixado com o nome do *plug-in* seguido de um ponto (“.”). Ex.: `"java.#type_declaration"`.

- `String toString ()`: retorna um *string* representando o trecho do código armazenado pela referência-p. Existem outras duas versões deste método: `toFile (String fileName)`, que grava o *string* resultante da operação no arquivo de nome `fileName`, e `toStream (java.io.OutputStream out)`, que utiliza o *stream* de saída `out` para armazenar o resultado da operação;

- `PReference duplicate()`: duplica o código armazenado pela referência-p e o retorna em uma nova referência-p. O tipo da nova referência-p é o mesmo da original. A verificação de igualdade (método `equals`) entre o código original e o duplicado resulta em `false`. Abaixo é apresentado um exemplo de utilização deste método.

Exemplo 3.8 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        ...
```

```
            id.set("import eds.arvoresBinarias.ArvoreSBB;");
            PReference tmp = id.duplicate();
            System.out.println(id.match(tmp));// resultado "true"
            System.out.println(id.equals(tmp));// resultado "false"
            ...
        }
    }
```

No exemplo a referência-p `tmp` é criada pela duplicação de `id`. Em seguida, é verificada a equivalência sintática dos dois trechos de código, esta verificação resulta em `true`. Para finalizar, é verificado se o trecho de código armazenado pelas referências é o mesmo, o resultado é `false`.

A interface `PReference` também define o método `boolean isComposedBy(PReference r)`. Assim como será feito com o `replace`, o método `isComposedBy` também será apresentado na Seção 3.4, onde seu comportamento poderá ser mais bem compreendido.

A linguagem de meta-programação JaTS permite algumas operações especiais sobre tipos sintáticos que são, na verdade, listas de outras estruturas sintáticas (*iterative declarations* e métodos como `size`). De maneira equivalente, a interface `PReference` também oferece métodos específicos para estruturas classificadas como listas (uniformes ou não), i.e., métodos que são permitidos apenas para referências-p cujo tipo é classificado como lista (ver regras de classificação de tipos na Seção 3.2.1, pág 45), são eles: `contains`, `add`, `get` e `remove`. A utilização destes métodos com referências-p cujo tipo é classificado como simples resulta no lançamento de um erro. Referências-p classificadas como listas armazenam não apenas um, mas vários trechos de código de um determinado tipo sintático.

O método `PReference get (int i)` retorna uma referência-p para o item que se encontra na posição `i` da lista. Se a lista for uniforme, a referência-p retornada será do tipo que compõem a lista, por exemplo `#import_declaration` no caso de lista `#import_declarations` e `#type_declaration` no caso de lista `#type_declarations`. Se a lista for não uniforme, será retornada uma referência-p sem tipo para o item da lista. Em uma lista não uniforme definida como $A \rightarrow C \mid B\ A$ ou $A \rightarrow C \mid A\ B$, onde $B\ e\ C \in (N \cup T)^*$, N é o conjunto de não-terminais e T o conjunto de terminais, uma referência-p para um item de uma lista armazena um trecho de programa que pode ser derivado das formas sentenciais B ou C. O exemplo abaixo demonstra a utilização deste método.

Exemplo 3.9 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        PReference ids = MetaJSystem.
                        createPReference("java", "#import_declarations"),
```

```
        vds = MetaJSystem.createPReference("java", "#variable_declarators"),
        ...
        ids.set("import eds.arvoresBinarias.ArvoreSBB; " +
                "import java.util.*; " +
                "import java.io.*;");
        PReference id0 = ids.get(0);
        PReference id1 = ids.get(1);
        PReference id2 = ids.get(2);
        // id0, id1 e id2 são referências-p do tipo #import_declaration.
        // Seus valores são, respectivamente:
        // "import eds.arvoresBinarias.ArvoreSBB;",
        // "import java.util.*;" e "import java.io.*;".
        ...
        vds.set("var0, var1 = i + 1, var2 = 0");
        PReference vd0 = vds.get(0);
        PReference vd1 = vds.get(1);
        PReference vd2 = vds.get(2);
        // vd0, vd1 e vd2 são referências-p sem tipo.
        // Seus valores são, respectivamente: "var0",
        // ", var1 = i + 1" e ", var2 = 0".
        ...
    }
}
```

O código acima declara duas referências-p `ids` e `vds` de tipos `#import_declarations` e `#variable_declarators`, respectivamente. O `#import_declarations` é lista uniforme, `#variable_declarators` é lista não uniforme. Em seguida, a lista de valores de `ids` é construída a partir de um trecho de código Java correspondente a uma seqüência de importações. Três referências-p `id0`, `id1` e `id2` são declaradas, e são atribuídos a elas os valores que se encontram nas posições 0, 1 e 2 da lista `ids`, respectivamente. `id0`, `id1` e `id2` são referências-p para trechos de código `#import_declaration`. A lista de valores de `vds` é agora construída a partir de um trecho de código Java correspondente a uma seqüência de declaradores de variáveis. Finalmente, três referências-p `vd0`, `vd1` e `vd2` são declaradas, e a elas são atribuídos os valores que se encontram nas posições 0, 1 e 2 da lista `vds`, respectivamente. `vd0`, `vd1` e `vd2` são referências-p sem tipo e os valores armazenados por elas são, respectivamente, `"var0"`, `", var1 = i + 1"` e `", var2 = 0"`. Observe a ocorrência da vírgula (“,”) nos dois últimos itens `id1` e `id2` desta lista.

O método `boolean contains(PReference r)` percorre os valores armazenados pela referência-p verificando se o valor armazenado por `r` ocorre. O comportamento deste método pode ser descrito pelo seguinte pseudocódigo:

```
Para cada valor v da lista
```

```
    Se v.equals(r) retorne true
fim
retorne false
```

Diferentemente do método anteriormente citado, o `add` e o `remove` só podem ser utilizados por referências-p cujo tipo é lista uniforme. A utilização do mesmo com referências-p cujo tipo é da classe lista não uniforme, ou simples, resulta em um erro. Valores de listas não uniformes devem ser construídos ou modificados utilizando-se a concatenação de *strings*. Os métodos `add` e `remove` são proibidos para listas não uniformes pois com a sua utilização seria possível construir código sintaticamente incorreto. Abaixo mostramos um exemplo de utilização destes métodos para uma referência-p do tipo `#import_declarations`.

Exemplo 3.10 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        PReference ids = MetaJSystem.
                         createPReference("java", "#import_declarations"),
        ids2 = MetaJSystem.createPReference("java", "#import_declarations");
        ...
        ids.add("import eds.arvoresBinarias.ArvoreSBB;");
        ids.add("import java.util.*;");
        ids.add("import java.io.*;");
        ids2.add(ids);
        System.out.println(ids);
        ids.remove(1);
        ids.match("import eds.arvoresBinarias.ArvoreSBB;") // false
        ids.remove(0);
        ids.match("import java.io.*;") // true
        ids.add("import java.util.*;");
        ids.add("import java.io.*;");
        ...
    }
}
```

Neste exemplo o método `add` é utilizado para adicionar valores do tipo `#import_declaration` na lista `ids`. O método `remove` é utilizado para remover `#import_declaration`s desta lista.

O método `add`, assim como o `set` e o `match`, também possui 4 versões: `add(PReference r)`, `add(String s)`, `add(InputStream in)` e `addFile(String fileName)`. Só podem ser adicionados a uma referência-p trechos de código que sejam compatíveis (para adição ou substituição) com o tipo da mesma.

A interface `PReference` ainda oferece o método `int getSize ()` que retorna o tamanho da lista. Este método pode ser aplicado a qualquer tipo de referência-p, sendo que, no caso de referências-p de tipo simples, o resultado é 0 se a referência-p não possuir valor armazenado e, caso contrário, 1.

Os métodos da interface `PReference` permitem ao meta-programador comparar estruturalmente dois trechos de código objeto, realizar testes de tipos, verificar se uma estrutura é composta por outra (`isComposedBy`), adicionar, remover e recuperar elementos de listas e também converter o valor de uma referência para seu formato textual. Mas é preciso também uma forma de realizar caminhamentos pelo programa objeto, possivelmente realizando verificações e substituições de trechos. Este serviço é oferecido pelos iteradores. O método `getIterator` retorna um iterador para o trecho de código armazenado pela referência-p. Estes elementos serão tratados com mais detalhes na próxima seção.

A Tabela 3.2 apresenta um resumo da permissão para os métodos da interface `PReference`. O objetivo das restrições de cada um dos métodos é garantir a correção sintática do código armazenado. Observe que se fosse possível adicionar ou remover um elemento de uma lista não uniforme, inconsistência sintáticas poderiam ser geradas. Na lista `a, b, c`, por exemplo, removendo o primeiro item temos `, b, c`. Adicionando-se `d` à última posição resultaria em `a, b, c d`. Em ambos os casos, as listas resultantes são sintaticamente incorretas.

	Simples	**Lista uniforme**	**Lista não uniforme**
`set`	Sim	Sim	Sim
`match`	Sim	Sim	Sim
`equals`	Sim	Sim	Sim
`toString`	Sim	Sim	Sim
`duplicate`	Sim	Sim	Sim
`getSize`	Sim	Sim	Sim
`hasType`	Sim	Sim	Sim
`isComposedBy`	Sim	Sim	Sim
`getIterator`	Sim	Sim	Sim
`replace`	Sim	Não	Sim
`get`	Não	Sim	Sim
`contains`	Não	Sim	Sim
`add`	Não	Sim	Não
`remove`	Não	Sim	Não

Tabela 3.2: Permissões para os métodos da interface `PReference`

3.4 Iteradores

MetaJ disponibiliza iteradores para permitir caminhamentos sobre trechos de programas. Estes são uma implementação do padrão de projeto *Iterator* [GHJV94]. Segundo

Gamma [GHJV94], iteradores são objetos utilizados para:

> *Fornecer um meio de acessar, seqüencialmente, os elementos de um objeto agregado sem expor sua representação subjacente. (...) A idéia chave neste padrão é retirar a responsabilidade de acesso e percurso do objeto agregado e colocá-la em um objeto iterador.*

São chamados de objetos agregados aqueles que são definidos através da composição de outros. Exemplos clássicos de objetos agregados são as listas, pilhas, filas e árvores. Iteradores permitem que agregados sejam explorados passo a passo, i.e., um componente por vez, em uma ordem determinada. Para gerenciar este processo iteradores: mantém o estado atual do caminhamento pelo objeto agregado (um ponteiro para o elemento atual), operações para "dar um passo no caminhamento" (alcançar o próximo elemento), e para verificar se ainda existem elementos para serem explorados.

No caso de MetaJ, um objeto agregado é um programa e os iteradores encapsulam um caminhamento *top-down* pelas construções sintáticas que ocorrem no código. Diferentemente das ferramentas apresentadas no Capítulo 2, em que o caminhamento é apenas declarado ou pré-definido, iteradores permitem um controle total do processo de exploração do código permitindo ao meta-programa decidir, a cada passo, se a próxima estrutura sintática será explorada, ignorada, substituída ou mesmo se o processo de caminhamento será interrompido. Para isso, a interface `Iterator` define as seguintes operações: (i) `nextIn`, alcança a próxima construção no caminhamento *top-down*; (ii) `hasNextIn`, verifica se existe mais alguma construção sintática para ser explorada; (iii) `nextOver`, realiza um salto no código, passa para a próxima construção do código ignorando as subestruturas da construção sintática atual; (iv) `hasNextOver`, verifica se existe alguma construção sintática para ser explorada, ignorando-se as subestruturas da construção atual. Uma instância de `Iterator` é criada a partir de uma referência-p com o objetivo de explorar o trecho de código armazenado por ela. Veja um exemplo de utilização de iteradores abaixo:

Exemplo 3.11 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        PReference cd = MetaJSystem.
                        createPReference("java", "#class_declaration");
        ...
        cd.set("public abstract class A extends B { }");
        Iterator td = cd.getIterator();
        it.nextIn();// alcança o trecho  "public" do tipo #modifiers_opt
        PReference atual = it.get();
        System.out.println(atual); // imprime na tela "public abstract"
```

```
        it.nextOver(); // alcança o trecho  "A" do tipo #identifier
        it.nextOver(); // alcança o trecho "extends B" do tipo #super_opt
        ...
    }
}
```

Neste exemplo podemos perceber a utilização de um iterador para percorrer o trecho de programa `public abstract class A extends B { }` do tipo `#class_declaration`. Para explorá-lo, são utilizados os métodos `nextIn` e `nextOver`. Para entender a ordem de caminhamento no código, basta observar a gramática de Java, mais especificamente a produção `class_declaration` que deriva este fragmento de código:

```
...
class_declaration → modifiers_opt CLASS  identifier
                                 super_opt interfaces_opt class_body;
identifier →  IDENTIFIER;
...
```

Na utilização do método `nextIn`, o iterador alcança o trecho `public abstract` cujo tipo é `#modifiers_opt`. Em seguida, por causa da utilização do método `nextOver`, alcança-se a construção `A`, do tipo `#identifier`. Vale destacar que as construções sintáticas que correspondem a terminais da linguagem objeto são ignoradas, por esta razão a palavra reservada `class` não é atingida. Na segunda utilização do método `nextOver`, a estrutura `#super_opt` é alcançada. Observe que a gramática mostra exatamente a próxima estrutura a ser alcançada a cada passo.

O exemplo abaixo ilustra um caminhamento mais longo e também mais complexo:

Exemplo 3.12 :

A partir de uma referência-p `r` encapsulando o trecho de código "`public final String getName() { ... }`" do tipo `#method_declaration`, pode ser criado um iterador `it` para caminhar pelo código. Veja o caminhamento abaixo:

		Tipo da estrutura atual
nextIn:	`public final String getName() { ... }`	`#method_declaration`
nextIn:	`public final String getName()` `{ ... }`	`#method_header`
nextIn:	`public final` `String getName() { ... }`	`#modifiers_opt`
nextIn:	`public final` `String getName() { ... }`	`#modifiers`
nextIn:	`public` `final String getName() { ... }`	`#modifier`
nextIn:	`public` `final` `String getName() { ... }`	`#modifier`

```
   nextIn:  public final [String] getName() { ... }                    #type
 nextOver:  public final String [getName()] { ... }            #method_declarator
   nextIn:  public final String [getName]() { ... }                  #identifier
   nextIn:  public final String getName([]) { ... }    #formal_parameter_list_opt
   nextIn:  public final String getName() [{ ... }]                  #method_body
   nextIn:  public final String getName() [{ ... }]                      #block
   nextIn:  public final String getName() {[ ... ]}         #block_statements_opt
   nextIn:  public final String getName() { [...] }             #block_statements
      ...   ...                                                             ...
```

Este exemplo ilustra o caminhamento *top-down* pelo trecho de código citado anteriormente. Vale destacar a utilização da operação `nextOver` que fez com que as subestruturas do trecho `String`, de tipo `#type`, fossem ignorada. A utilização da operação `nextIn` neste momento resultaria no seguinte caminhamento:

```
    ...
 nextIn:  public final [String] getName() { ... }                    #type
 nextIn:  public final [String] getName() { ... }                    #name
 nextIn:  public final [String] getName() { ... }                 #identifier
 nextIn:  public final String [getName()] { ... }          #method_declarator
 nextIn:  public final String [getName]() { ... }                 #identifier
    ...
```

Apesar de muito úteis, a utilização intensa de iteradores pode fazer com que o programador se sinta "perdido" durante o caminhamento pelo código. Para evitar isso, deve ser utilizado o teste de tipos, i.e., o método `hasType` de referências-p, para controlar o caminhamento. No exemplo abaixo é mostrado um caminhamento para se imprimir todas as utilizações de identificadores (`#identifiers`) em um programa Java.

Exemplo 3.13 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        PReference compunit = MetaJSystem.
                        createReference("java", "#compilation_unit");
        compunit.setFile(args[0]);
        Iterator it = compunit.getIterator();
        while(it.hasNextIn()){
            Reference id = it.get();
            if(id.hasType("java.#identifier")) System.out.println(id);
        }
```

```
    }
}
```

Nem todos os métodos da interface `PReference` foram apresentados na Seção 3.3. A descrição do método `replace` foi omitida naquela situação devido ao fato de que a compreensão do mesmo seria beneficiada pela sua apresentação juntamente com iteradores. Vale lembrar que este método possui semântica bastante parecida com a do `set`, i.e., redefine o valor da referência-p. Mas ele possui uma pequena diferença: quando a referência-p já possui um valor definido, ele realiza a modificação do valor da referência-p e também faz a atualização no contexto onde está o trecho de código armazenado atualmente pela referência-p. O exemplo abaixo ilustra a utilização deste método:

Exemplo 3.14 :

```
...
public class Main {
    public static void main (String args[]) throws ParseException{
        PReference compunit = MetaJSystem.
                        createPReference("java", "#compilation_unit");
        compunit.set("package teste; " +
                     "import java.util.*; " +
                     "class A { } "+
                     "class B { }");
        Iterator it = compunit.getIterator();
        while(it.hasNextIn()){
            PReference id = it.get();
            if(id.hasType("java.#identifier")){
                 id.setValue("novoId");
            }
        }
    }
}
```

Este exemplo é semelhante ao anterior, exceto pelo fato de que o valor da referência-p `compunit` foi definido explicitamente e que o comando `System.out.println (id)` foi substituído por `id.replace("novoId")`. Este meta-programa percorre o valor armazenado por `compunit` substituindo cada ocorrência identificador por "novoId". Por isso, o trecho armazenado por `compunit` após ao final do programa será:

```
package novoId;
import novoId.novoId.*;
class novoId { }
class novoId { }
```

Vale ressaltar novamente que o método `replace` só pode ser aplicado em referências-p cujo tipo é classificado como simples ou lista não uniforme. No caso das listas uniformes, para se realizar tal operação, deve-se substituir cada um de seus itens de maneira independente. Esta restrição é justificada pelo fato de que a aplicação deste método sobre uma lista uniforme pode causar vários efeitos colaterais indesejáveis pois esta lista pode estar compartilhando itens com outras listas uniformes de mesmo tipo sintático.

Outro método de `PReference` que não foi tratado na Seção 3.3 foi o `boolean isComposedBy (PReference r)`. Este método percorre toda a estrutura do código armazenado pela referência-p verificando se o trecho armazenado por `r` (recebida como parâmetro pelo método) ocorre. Seu comportamento pode ser descrito da seguinte maneira:

```
...
public boolean isComposedBy (PReference r){
    // cria-se um iterador para a referência-p em questão
    Iterator it = getIterator ();
    // percorre-se todo o código ...
    while(it.hasNextIn()){
        // se o trecho armazenado por r for encontrado,
        // retorne verdadeiro
        if(it.get().equals(r)) return true;
    }
    // se o programa for completamente percorrido,
    // sem encontrar ocorrência de r, retorne falso
    return false;
}
...
```

É comum que meta-programas necessitem fazer verificações sobre a estrutura do programa percorrido, o que pode ser bastante cansativo utilizando-se apenas o teste de tipos. Iteradores são mais úteis quando utilizados juntamente com *templates*.

3.5 *Templates*

Com o objetivo de permitir decomposição, verificação estrutural e mesmo construção de trechos de programas baseados em um modelo pré-definido, MetaJ disponibiliza *templates*. Esta abstração encapsula padrões de programas descritos utilizando-se a sintaxe concreta da linguagem objeto (*by example*).

Em contraste com algumas das ferramentas de meta-programação apresentadas no Capítulo 2 (TXL, ASF+SDF, Stratego e JaTS), em que as operações sobre padrões de programas (casamento e impressão – conversão para formato textual) são implícitas (resultantes da aplicação de uma regra de reescrita), em MetaJ tais operações são especificadas de maneira explicita, podendo

serem invocadas em qualquer parte do meta-programa. Esta característica dá grande flexibilidade ao meta-programador que pode, utilizando referências-p e iteradores, intercalar operações de casamento de padrão e reescrita de trechos de programas com testes refinados e caminhamentos pelo código objeto.

A Seção 3.5.1 apresentará como devem ser especificados padrões de programas em MetaJ. Na Seção 3.5.3 é apresentada a forma de se declarar e utilizar *templates*.

3.5.1 Padrões de programas *by example* em MetaJ

Em MetaJ, um padrão de programa é uma sentença da linguagem objeto onde podem ocorrer, no lugar de uma construção sintática, dois tipos de meta-anotações: declaração de ***meta-variáveis*** e ***marcação de trecho opcional***. A ocorrência de uma meta-anotação em um padrão sempre se inicia com uma palavra-chave que define qual a estrutura sintática que deveria ocorrer naquela posição do código. Estas palavras-chave correspondem ao tipo sintático da construção sintática que está sendo substituída. Todo tipo sintático deve se iniciar com um símbolo que o diferencie das outras palavras da linguagem objeto, o símbolo de *anti-quotation*. Com já foi citado anteriormente, o *plug-in* java utilizado nos exemplos deste documento utiliza o marcador "#". A seguir são apresentadas as regras para inserção de meta-anotações em padrões de programas.

Meta-Variáveis

Meta-variáveis são meta-anotações que aparecem em padrões para representar a ocorrência de uma determinada estrutura sintática da linguagem objeto. Elas ocorrem como construções coringas. Toda meta-variável possui um tipo e um nome. Seu tipo define qual estrutura sintática é representada por ela. Seu nome a identifica entre todas as outras que ocorrem no padrão.

A sintaxe para declaração de meta-variável é:

```
<tipo> <nome da variável>
```

onde *<tipo>* é uma palavra chave que se inicia com o símbolo de *anti-quotation* ("#", no caso do *plug-in* java) e que define a estrutura sintática representada pela variável de nome *<nome da variável>*.

Diferentes ocorrências de uma mesma meta-variável (i.e., ocorrências com o mesmo nome) em um padrão representam construções sintáticas equivalentes de uma mesma estrutura sintática. Veja abaixo alguns exemplos de utilização de meta-variáveis em padrões.

Exemplo 3.15 :

> Tomando a gramática de expressão apresentada no Exemplo 3.1 e considerando os nomes dos não-terminais da linguagem prefixados com "#" como palavras-chave que definem o tipo da meta-variável, abaixo são apresentados alguns padrões de programa:

- `5 + #Num n`: este padrão representa uma adição cujo primeiro fator é o número `5` e o segundo um número `n`;
- **`#Exp e - #Num n`**: este padrão representa uma subtração cujo primeiro fator é uma expressão `e` e o segundo um número `n` qualquer;
- **`#Exp e1 + #Exp e2 - #Num n`**: este padrão representa uma subtração cujo primeiro fator é uma adição de duas expressões `e1` e `e2` e o segundo o número `n` qualquer;
- `35 - #Num n + #Num n`: este padrão representa uma adição cujo primeiro fator é uma subtração entre `35` e um número qualquer `n`. O segundo fator da adição é, obrigatoriamente, o mesmo número `n` que ocorreu no primeiro fator.

Nos exemplos acima podem ser observadas declarações de meta variáveis. O último exemplo mostra também diferentes ocorrência de uma mesma meta-variável, o que indica diferentes ocorrências de uma mesma construção sintática, neste caso, duas ocorrências de um mesmo número.

Meta-variáveis são classificadas como listas ou simples, de acordo com o seu tipo. Meta-variáveis listas representam a ocorrência de uma seqüência de construções sintáticas na posição do código onde ela aparece. As simples representam a ocorrência de apenas uma. Os exemplos apresentados anteriormente utilizaram apenas meta-variáveis simples.

Meta-variáveis listas são aquelas que representam estruturas sintáticas que foram definidas na gramática da linguagem objeto através da utilização de recursão à esquerda ou à direita, i.e., estruturas que são listas de outras estruturas sintáticas. Estas meta-variáveis podem ocorrer em qualquer posição da lista definida. Veja exemplos abaixo:

Exemplo 3.16 :

Seja a gramática $G = (T, N, P, R)$, onde:

```
T = {|, 1, 2, ..., 9, 0, +, -};
N = {Exps, Exp, Num};
e R é o conjunto das seguintes regras
  Exps → Exps | Exp;
  Exps → Exp;
  Exp → ... //mesma definição do Exemplo 3.1
  Num → ... //mesma definição do Exemplo 3.1
```

As estruturas sintáticas definidas por `Exp` e `Exps` são classificadas como listas não uniforme.

- `5 + 6 | 8 + 7`: este padrão representa uma lista de duas adições `5 + 6` e `7 + 8`;
- **`#Exps aes`** `| 3 + 4`: este padrão representa uma lista de expressões em que a última delas é uma adição `3 + 4`;

- `3 + #Num n #Exps des`: este padrão representa uma lista de expressões em que a primeira de suas componentes é a adição `3 + #Num n`. A meta-variável `des` captura a seqüência de expressões que ocorre depois de `3 + #Num n`.
- `#Exps aes | #Num n + #Num n #Exps des`: este padrão representa uma lista de expressões em que uma de suas componentes é a adição de um número `n` com ele mesmo. As meta-variáveis `aes` e `des` capturam a seqüência de expressões que ocorrem antes e depois de `#Num n + #Num n`, respectivamente.

Uma exceção às regras de declaração de meta-variáveis definidas anteriormente ocorre quando o tipo sintático de uma meta-variável é classificado como lista não uniforme e a lista é definida através de recursão à direita. Neste caso, só poderá ocorrer na lista uma declaração de meta-variável e ela só poderá aparecer como último elemento da lista. Veja um pequeno exemplo abaixo.

Exemplo 3.17 :

Seja a gramática $G = (T, N, P, R)$, onde:

```
T = {1,2,..., 9, 0, +, -};
N = {Exp, Num};
e R é o conjunto das seguintes regras
  Exp → Num + Exp;
  Exp → Num - Exp;
  Exp → Num;
  Num → Num 1;
  Num → Num 2;
  ...
  Num → Num 9;
  Num → Num 0;
```

Esta gramática é bastante parecida com a apresentada no Exemplo 3.1, exceto pelo fato de que as operações de adição e subtração são associativas à direita. Esta modificação fez com que `Exp` se tornasse um não-terminal classificado como lista não uniforme recursiva à direita. Isso implica em uma restrição na descrição de padrões: em uma expressão só poderá ocorrer uma declaração de meta-variável do tipo **#Exp** e ela deverá estar no final da lista.

- `5 + #Exp e`: este padrão representa expressão de adição do literal `5` a uma expressão qualquer;
- `#Exp e + 4` (*Inválido*): este é inválido pois ele declara uma meta-variável do tipo **#Exp** no início da lista;

- `3 + #Num n - #Exps e`: este padrão representa uma adição do literal `3` à sub-expressão `#Num n - #Exps e`, que corresponde a uma subtração entre um número `n` (**#Num**) e uma expressão `e` (**#Exp**). Observe que este padrão é correto pois a meta-variável `e` do tipo lista não uniforme recursiva a direita (**#Exp**) aparece apenas no fim da construção `3 + #Num n - #Exps e` do tipo **#Exp**.

A utilização de variáveis como marcações coringas em padrões é um recurso que a maioria das linguagens de meta-programação oferece. Assim como JaTS, MetaJ permite também a definição de trechos opcionais.

Trecho opcional

Uma definição de trecho opcional descreve um fragmento do padrão, i.e., uma construção sintática, que pode ou não possuir um trecho correspondente no programa objeto. Apesar de tornarem a descrição de padrões um pouco mais complexa, trechos opcionais permitem a especificação de padrões mais genéricos, aumentando a possibilidade de reutilização dos mesmos. A sintaxe para definição de trecho de código opcional é:

<tipo> "[" <padrão de programa do tipo <tipo>>"]#"

onde *<tipo>* é uma palavra chave que se inicia com o símbolo de *anti-quotation* ("#", no caso) e que define o tipo de estrutura sintática que está sendo substituída pela meta-anotação. Os marcadores de trecho opcional "**[**" e "**]#**" não precisam ser necessariamente estes. O primeiro marcador é chamado de **marcador de abertura de opcional** e o segundo de **marcador de fechamento de trecho opcional** Delimitado por estas marcações pode ocorrer um padrão de estrutura sintática do tipo *<tipo>*. Trechos opcionais podem ocorrer em duas situações:

- como representantes de estruturas sintáticas classificadas como opcionais.
- delimitando, em construções sintáticas de uma estrutura sintática classificada como lista, um seqüência opcional de itens;

Veja abaixo alguns exemplos:

Exemplo 3.18 :

Seja a gramática $G = (T, N, P, R)$, onde:

```
T = {ID, int, String, char, public, private, =, ..., 9, 0, +, -, .};
N = {Exp, Num, Decl, Type, VarInit, VarInit_Opt, Mod, Mod_Opt};
e R é o conjunto das seguintes regras
  Decls → Decls Decl | Decl;
  Decl → Mod_Opt Type ID VarInit_Opt.;
```

```
Mod_Opt → Mod | λ ;
Mod → public | private | protected;
Type → int | String | char;
VarInit_Opt → VarInit | λ;
VarInit → = Exp;
Exp → ... //mesma definição do Exemplo 3.1
Num → ... //mesma definição do Exemplo 3.1
```

A gramática acima descreve regras sintáticas para declaração de variáveis. Veja abaixo alguns exemplos de padrões para sentenças desta linguagem.

- `int var1.`: este padrão representa uma sentença que declara a variável `var1`;
- **#Type t** `var1 =` **#Num n** `+ 1.`: este padrão representa uma declaração de variável de nome `var1`, que o tipo é capturado pela meta-variável `t`, e que é inicializada com uma expressão que é uma soma de uma unidade a um número `n`;
- **#Mod_Opt[** `public` **]#** `int var1.`: este padrão representa a declaração de uma variável inteira de nome `var1`, possivelmente pública (observe o terminal `public` entre marcadores opcionais);
- **#Mod_opt[** `#Mod m` **]#** `int var1` **#VarInit_Opt[** `=` **#Exp e** `+ 1` **]#.**: este padrão representa a declaração de uma variável inteira de nome `var1`, onde pode ocorrer um modificador `m` ou uma expressão de inicialização. A expressão, se realmente ocorrer, deverá ser uma adição de uma unidade a uma outra expressão `e` qualquer;
- Observe o padrão abaixo:

```
#Decls[
    int var0.
    int var1 = 0.
    #Type t var2  #VarInit_Opt[ #VarInit vi ]#.
 ]#
 #Mod_opt[ #Mod  m ]# int var3 #VarInit_Opt[ = #Exp e + 1 ]#.
```

  Ele representa uma lista de declarações. A seqüência das três primeiras declarações da lista é opcional, isso faz com que o padrão represente trechos de código onde ou as três declarações ocorrem, ou nenhuma delas ocorre, i.e., a lista tem tamanho 4 ou 1. Apenas a quarta e última declaração é obrigatória. Ela corresponde à declaração de uma variável inteira de nome `var3`, onde podem ocorrer um modificador `m` e também uma expressão de inicialização. Esta expressão, se realmente ocorrer, deverá ser uma adição de uma unidade a uma outra expressão `e` qualquer.

Assim como ocorreu no caso das meta-variáveis, uma exceção às regras de definição de trechos de padrões opcionais ocorre quando o tipo sintático substituído pela meta-anotação é classificado

como lista não uniforme recursiva à direita. Neste caso, a meta-anotação de trecho de padrão opcional não poderá ocorrer no final da construção sintática, mas poderá ocorrer em qualquer outro ponto da mesma. Veja um pequeno exemplo abaixo:

Exemplo 3.19 :

Tomando a gramática de expressões apresentada no Exemplo 3.17.

- `5 #Exp[ + 3 ]# + 4`: nesta expressão, a ocorrência da adição `+ 3` é opcional;
- **`#Num n #Exp[ + 3 ]# #Exp[ - 9 ]# #Exp e`**: nesta expressão, tanto a adição `+ 3` quanto a subtração `- 9` são opcionais. O final da expressão será capturado pela meta-variável `e`;
- **`#Num n #Exp[ + 3 ]#`** (*Inválido*): este padrão é inválido pois o trecho opcional ocorre no final da expressão;

3.5.2 Resumo das regras para escrever padrões de programas

Formalmente, um padrão de programa correto deve estar de acordo com a gramática de padrões de programa da linguagem objeto em questão. A Seção 3.5.4 descreverá as regras para construção desta gramática a partir da gramática da linguagem objeto. Apesar disso, como pode ser percebido nas seções anteriores, é possível descrever padrões sem conhecer estas regras detalhadamente. A Seção 3.5.1 apresentou de maneira informal como devem ser inseridas meta-anotações em uma sentença da linguagem objeto. Um resumo das regras de descrição de padrões é apresentado abaixo:

regras para descrição de padrões: qualquer estrutura sintática da linguagem objeto pode ser substituída por dois tipos de meta-anotações:

- **meta-variáveis:** A sintaxe para declaração de meta-variáveis é *<tipo> <identificador>*. Uma declaração ela aparecer no lugar de qualquer construção sintática da linguagem objeto. *<tipo>* representa á estrutura sintática representada.
- **trechos opcionais:** A sintaxe para definição de trecho opcional de padrões é:

  <tipo> <abertura de opcionais> <padrão <tipo>> <fechamento de opcionais>

  Esta meta-anotação pode ocorrer como substituição de ocorrências de estruturas sintáticas opcionais ou em listas, delimitando os itens que podem ou não ocorrer. *<tipo>* é o tipo de estrutura sintática substituída, no primeiro caso, ou o tipo da lista, no segundo caso. Estes marcadores podem ocorrer delimitando qualquer trecho da lista.

exceção: construções sintáticas de estruturas classificadas como listas não uniformes recursivas à direita: neste caso, os marcadores opcionais não poderão aparecer delimitando o final da lista. Meta-variáveis só poderão aparecer no final da lista.

A linguagem de padrões de programas é bastante limitada quanto a manipulação de estruturas deste tipo. Por esta razão, nestes casos, os padrões utilizados pelo usuário serão bastante restritos e será necessário maior utilização dos outros recursos de MetaJ (iteradores, testes de tipo, operações de referências-p, ...).

3.5.3 Declaração de *templates*

Diferentemente das abstrações referência-p e iterador, que podem ser utilizadas simplesmente pela instanciação de objetos `PReference` e `Iterator`, respectivamente, *templates* devem primeiramente ser declarados para que instâncias possam ser criadas. Para isso, MetaJ define uma *linguagem de declaração de templates*. *Templates* devem ser declarados e compilados separadamente dos programas que os utiliza, o que faz deles elementos reutilizáveis, possíveis de serem importados por qualquer meta-programa. Em outras ferramentas, padrões aparecem declarados dentro do próprio meta-programa impedindo sua reutilização.

Um arquivo contendo declarações de *templates* se inicia por uma declaração de pacote (semelhante àquela encontrada em unidades de compilação Java) seguida pela declaração da linguagem objeto que será utilizada nas declarações de *templates* que se seguirão. O nome da linguagem objeto especificado é, exatamente, o nome do *plug-in* que gerencia as informações sobre a linguagem objeto utilizada na descrição dos *templates*. Finalmente, uma seqüência de declarações de *templates* pode ser especificada. Cada declaração deve obedecer à seguinte sintaxe:

```
template <tipo> <identificador> #{
    <padrão de programa>
}#
```

Na seção *<meta-tipo>* deve ser especificado o tipo do padrão de programa que será encapsulado pelo *template*. Em *<identificador>* deve ser especificado um nome para o *template*. Em seu corpo (entre os símbolos **#{** ... **}#**), deve ser especificado o padrão que será encapsulado. A gramática completa para reconhecer um arquivo contendo declarações de *templates* pode ser encontrada no Anexo A. Veja abaixo exemplos de declarações de *templates*:

Exemplo 3.20 :

```
package  metaj.examples.basicTemplates; // declaração de pacotes
language java; // declaração da linguagem objeto utilizada,
               // i.e., nome do plug-in

/*** Declaração do template ChooseClass ***/
template #compilation_unit ChooseClass #{
    #package_declaration_opt[ package #name pack; ]#
    #import_declarations_opt[ #import_declarations imps ]#
```

```
    #type_declarations[ #type_declarations tds ]#
    #modifiers_opt[ #modifiers m ]# class #identifier classId
                        #super_opt[ extends #name superClassName ]#
                        #interfaces_opt[#interfaces is]# {
        #class_body_declarations_opt[ #class_body_declarations cbds ]#
    }
    #type_declarations[ #type_declarations tds1 ]#
}#
...
/*** Declaração do template ChooseMethod ***/
template #class_body_declarations ChooseMethod #{
        #class_body_declarations[ #class_body_declarations cbds1 ]#
        #method_header mh #method_body mb
        #class_body_declarations[ #class_body_declarations cbds2 ]#
}#
...
```

O exemplo acima mostra um arquivo contendo declarações de dois *templates*: `ChooseClass` e `ChooseMethod`. Observe que inicialmente é declarado o pacote `metaj.examples.basicTemplates`. Em seguida, a linguagem objeto (*plug-in* "java") é especificada. Finalmente os dois *templates* são declarados. O primeiro, `ChooseClass`, define um padrão do tipo `#compilation_unit` que representa programas Java com pelo menos uma declaração de classe. No programa representado, a declaração de pacote e importações são opcionais. O segundo, `ChooseMethod`, encapsula um padrão do tipo `#class_body_declarations` que representa uma seqüência de declarações de corpo de classe onde ocorre, obrigatoriamente, uma definição de método.

Para se utilizar os *templates* declarados, eles precisam ser primeiramente compilados. MetaJ possui um *compilador de templates* que traduz declarações de *templates* em classes Java, permitindo assim que eles sejam instanciados dentro de qualquer programa Java. Depois de compilados, *templates* podem ser utilizados como qualquer classe Java. Veja o exemplo abaixo.

Exemplo 3.21 :

Depois de compilados, os *templates* declarados no Exemplo 3.20 podem ser utilizados da seguinte maneira.

```
package teste;
import metaj.examples.basicTemplates.*;
...// outras importações
public class Main {
     public static void main (String args[])throws Exception{
         // criação de uma instância do template ChooseClass
```

```
            ChooseClass cc = new ChooseClass ();
            PReference cid = cc.getClassId();
            cid.set("MinhaClasse");
            if(cc.matchFile(args[0])){
               if(!cc.getCbds().isNull()){
                   // criação de uma instância do template ChooseMethod
                    ChooseMethod cm = new ChooseMethod ();
                    if(cm.match(cc.getCbds())){
                        System.out.println(cm.getMh());
                    }else{
                        System.out.println("Metodo não encontrado.");
                    }
                }
            }
            ...
            // criação de uma nova instância do template ChooseClass
            cc = new ChooseClass();
            ...
        }
    }
```

Neste exemplo pode ser observada a criação de instâncias dos *templates* `ChooseClass` e `ChooseMethod`. Depois de criadas, os métodos definidos pelas classes resultantes da compilação destes *templates* podem ser utilizados.

Por serem utilizados como classes Java, instâncias diferentes de um mesmo *template* podem ser criadas e manipuladas de maneira independente sem que seja necessário reescrevê-lo.

Todo *template*, independente de como foi declarado, disponibiliza os métodos `match` e `toString` com semânticas equivalentes àquelas definidas pelos métodos de mesmo nome disponibilizados pelas referências-p.

- `boolean match (...)`: verifica se o código recebido como parâmetro está de acordo com o padrão de programa encapsulado pelo *template*. Este método define valores para as meta-variáveis declaradas no padrão de programa. Meta-variáveis com valores previamente definidos não são modificadas e representam apenas trechos de código sintaticamente equivalentes àqueles armazenados por elas.

  Assim como em referências, `match` é polimórfico e possui quatro versões: `match (String s)`, para código objeto representado na forma textual; `match (PReferencer)`, para manipular código que foi previamente armazenado em uma referência-p; `matchFile(String fileName)`, para verificar o código objeto armazenado no arquivo de nome `fileName` e `match (InputStream in)`, para verificar o código recebido pelo *stream* de entrada `in`.

- `String toString()`: converte o padrão de programa para sua forma textual substituindo cada meta-variável utilizada pelo valor que foi ligado a ela anteriormente. As versões `void toFile(String fileName)` e `PReferencetoPReference()` deste método realizam a mesma operação, com a diferença de que `toFile` grava o resultado no arquivo de nome `fileName` e `toReference` retorna o resultado da operação como uma referência-p para o código construído.

Templates disponibilizam métodos para acessar o valor de cada meta-variável do padrão de programa encapsulado. O valor de cada meta-variável é retornado como uma referência-p que pode ser recuperada através do método `PReference getXXX ()`, onde `XXX` é o nome da meta-variável cujo valor se deseja consultar. Assim, para o *template* `ChooseMethod` apresentado no Exemplo 3.20, os métodos `getMh`, `getMb`, `getCbds1` e `getCbds2` são disponibilizados. Observe no Exemplo 3.21 a utilização do método `getMh`. Depois de recuperadas, qualquer método da interface `PReference` poderá ser invocado, possivelmente modificando ou testando o valor da referência-p.

3.5.4 Regras para a construção gramática de padrões a partir da gramática da linguagem objeto

Na Seção 3.5.2 foram definidas, de maneira bastante informal, regras para se inserir meta-anotações em padrões de programas. Estas regras são definidas formalmente pela **gramática de padrões** da linguagem objeto para a qual se deseja descrever um padrão de programa. Esta gramática corresponde à gramática original desta linguagem acrescida de produções que permitem declarar meta-anotações em pontos específicos do código.

Esta seção tratará de detalhes sobre as regras para construção da gramática de padrões a partir da gramática da linguagem objeto. Devido à necessidade de exatidão destas regras, serão tratados detalhes que só são necessários para o desenvolvedor interessado em construir gramáticas de padrões, i.e., construir *plug-ins*.

Para se construir a gramática de padrões $G' = (N', T', R', P')$ a partir da gramática $G = (N, T, R, P)$ da linguagem objeto é necessário, primeiramente, classificar os não-terminais de G de acordo com as regras apresentadas na Seção 3.1, página 45.

O segundo passo é escolher um marcador de *anti-quotation*. Nesta seção será utilizado o símbolo "`#`". Também devem ser escolhidos os marcadores de trecho opcional. Assim como já foi feito anteriormente, serão utilizados os "`[`" e "`]#`".

Dados estes marcadores, o algoritmo abaixo descreve como se construir a gramática de padrões G' a partir da gramática original da linguagem objeto G.

[1] crie $G' = (N', T', R', P')$, com $N' = N$, $T' = T$, $R' = R$ e $P' = P$.
[2] adicione em T' um terminal ID para nomes de meta-variáveis
[3] para todo $A \in N$ faça
[4] Se A é **Simples**
[5] crie um terminal **#A** $\notin T'$ resultante da concatenação do marcador de *anti-quotation* escolhido com o nome do não-terminal A.
[6] adicione a R' as produções: (i) $A \rightarrow$ **#A** ID e (ii) $P' \rightarrow$ **#A** A

[7] Se A é **Opcional**
[8] crie um terminal **#A[** $\notin T'$ resultante da concatenação de **#A** com o marcador de abertura de trecho opcional escolhido.
[9] adicione a R' a produção $A \rightarrow$ **#A[** A **]#**

[10] Se A é **Lista**
[11] crie um terminal **#A** $\notin T'$ para representar o tipo sintático **#A**
[12] crie um terminal **#A[** $\notin T'$ resultante da concatenação de **#A** com o marcador de abertura de trecho opcional escolhido.
[13] adicione a R' as produções (i) $A \rightarrow$ **#A** ID e (ii) $P' \rightarrow$ **#A** A

[14] Se A é **Lista Uniforme** então
[15] adicione em R' a produção $A \rightarrow$ **#A[** A **]#**
[16] Se A é exclusivamente recursivo a esquerda (não o é à direita)
[17] adicione a R' as produções $A \rightarrow A$ **#A[** A **]#** e $A \rightarrow A$ **#A** ID
[18] Se A é exclusivamente recursivo a direita (não o é à esquerda)
[19] adicione a R' as produções $A \rightarrow$ **#A[** A **]#** A e $A \rightarrow$ **#A** ID A
[20] Se A é **Lista não Uniforme** então
[21] crie um não-terminal $A' \notin N'$
[22] Se A é exclusivamente recursivo a esquerda (não o é à direita)
[23] adicione a R' as produções: (i) $A \rightarrow A$ **#A** ID; (ii) $A \rightarrow$ **#A[** A **]#**; (iii) $A \rightarrow A$ **#A[** A' **]#**
[24] para toda regra da forma $A \rightarrow A\ Z$ ($Z \in (T \cup N)^*$), faça
[25] adicione a R' as produções $A' \rightarrow Z$ e $A' \rightarrow A'\ Z$

[26] Se A é exclusivamente recursivo a direita (não o é à esquerda)
[27] adicione a R' a produção $A \rightarrow$ **#A[** A' **]#** A
[28] para toda regra da forma $A \rightarrow A\ Z$ ($Z \in (T \cup N)^*$), faça
[29] adicione a R' as produções $A' \rightarrow Z$ e $A' \rightarrow A'\ Z$

[30] adicione a R' as seguintes produções: (i) $A' \rightarrow$ **#A** ID; (ii) $A' \rightarrow A'$ **#A** ID; (iii) $A' \rightarrow$ **#A[** A' **]#**; (iv) $A' \rightarrow A$ **#A[** A' **]#**;

Cada passo do algoritmo acima será descrito mais detalhadamente abaixo e nas seções que se seguem:

- **linhas 1 e 2:** a nova gramática G' é criada com os mesmos terminais, não-terminais, regras e símbolo inicial de G. Um terminal ID para representar identificadores de meta-variáveis é inserido em T', o conjunto dos terminais. Neste passo, é possível reaproveitar as regras léxicas para construções de identificadores da própria linguagem objeto, mas isso pode gerar conflitos na tabela de parser. É recomendado criar um identificador específico para meta-variáveis pois, desta forma, é garantida a ausência de conflitos.
- **linhas 3 a 30:** o *loop* percorre todo o conjunto N de não-terminais inserindo novas regras de meta-anotações baseadas na classificação de cada um deles. Para cada classe de não-terminal, i.e., simples, opcional, lista uniforme e não uniforme, uma ação é realizada.

As ações tomadas nas linhas 3 a 30, serão descritas nas próximas seções.

Simples (linhas 4 a 6)

Se A é simples, então um novo terminal **#A** $\notin T'$ é criado. **#A** será utilizado como a palavra-chave que marca o início de uma meta-anotação do tipo A, i.e., uma meta-anotação colocada no lugar de uma estrutura sintática A. O *token* representado por **#A** é exatamente aquele obtido pela concatenação do marcador de *anti-quotation* com o nome do não-terminal A, o *string* "`#A`".

Neste caso, além de criar o terminal **#A**, uma nova regra é adicionada a R', $A \rightarrow$ **#A**ID para permitir declarações de meta-variáveis. Veja um exemplo abaixo.

Exemplo 3.22 :

O seguinte trecho da gramática de Java define como uma declaração de pacote deve ser construída:

```
package_declaration → PACKAGE name SEMICOLON ;
```

Como esta é a única produção que possui o não-terminal `package_declaration` no lado esquerdo da regra, ele é classificado como simples. Por isso, na gramática de padrões Java, a seguinte produção também será definida:

```
package_declaration → #package_declaration ID;
```

Desta forma, para o não-terminal `package_declaration` as seguinte produções serão farão parte da gramática de padrões de Java.

```
package_declaration → PACKAGE name SEMICOLON ;
package_declaration → #package_declaration ID;
```

Com estas produções, são permitidas declarações de meta-variáveis do tipo **#package_declaration** dentro de códigos Java, veja um padrão abaixo:

```
#package_declaration pd
import java.util.*;
class Teste {
    ... // corpo da classe
}
```

Além disso, para permitir que as construções sintáticas do tipo **#A** sejam aceitas a partir do não-terminal inicial P', a regra $P' \rightarrow$ **#A** A é acrescentada. Observe que, o terminal **#A** é utilizado para escolher entre as estruturas sintáticas da linguagem objeto, evitando assim a ambigüidade.

Opcional (linhas 7 a 9)

Se A é opcional, então um novo terminal **#A[** $\notin T'$ é criado. Este terminal faz dois papeis em uma definição de trecho opcional: palavra-chave que marca o início de uma meta-anotação do tipo A (no caso, marcações opcionais colocados no lugar de uma estrutura sintática A) e também de marcador abertura de trecho opcional.

A produção $A \rightarrow$ **#A[** A **]#** é adicionada ao conjunto de regras R'. Esta nova produção permitirá a utilização de trechos opcionais no lugar de estruturas sintáticas do tipo A. Veja um exemplo abaixo.

Exemplo 3.23 :

Uma unidade de compilação Java é definida pelas seguintes regras:

```
compilation_unit → package_declaration_opt
                   import_declarations_opt
                   type_declarations_opt;
package_declaration_opt → package_declaration | λ;
package_declaration → PACKAGE name SEMICOLON ;
... // outras regras
```

Sem entrar em questões sobre os outros não-terminais, pode ser observado que o `package_declaration_opt` é um não-terminal opcional. Por isso, na gramática de padrões Java, a seguinte produção será definida:

```
package_declaration_opt → #package_declaration_opt[ package_declaration_opt ]#;
```

Como `compilation_unit`, `package_declaration` e `package_declaration_opt` são classificadas como não-terminais simples, regras para declaração de meta-variáveis destes tipos são também inseridas. A gramática resultante é a seguinte.

```
compilation_unit → package_declaration_opt
                   import_declarations_opt
                   type_declarations_opt;
compilation_unit → #compilation_unit ID;
package_declaration_opt → package_declaration | λ
        | #package_declaration_opt ID
        | #package_declaration_opt[ package_declaration_opt ]#;

package_declaration → PACKAGE name SEMICOLON;
                          #package_declaration ID;
... // outras regras
```

Com estas produções, é permitida a declaração de meta-variáveis do tipo **#package_declaration**, **#package_declaration_opt** e **#compilation_unit**, bem como trechos opcionais do tipo **#package_declaration_opt**, dentro de códigos Java, veja um exemplo de padrão abaixo:

```
#package_declaration[ package myPack; ]#
import java.util.*;
class Teste {
    ... // corpo da classe
}
```

Ele representa um programa Java onde pode ou não ocorrer a declaração do pacote `myPack`. No programa representado deve ser importado o pacote `java.util` e ser declarada a classe `Teste`.

Lista (linhas 10 a 31)

Se o não-terminal A for uma lista (uniforme ou não), serão criados os não-terminais **#A** $\notin T'$ e **#A[** $\notin T'$, assim como foi feito nos dois casos anteriores. A produção $A \rightarrow$ **#A** ID é adicionada ao conjunto de regras R'. Esta nova produção permitirá a declaração de uma meta-variável no lugar da lista de estruturas definida pelo não-terminal A. Esta produção permite não só que toda a lista seja substituída por uma meta-variável, mas também que a parte inicial ou final da lista (dependendo se a mesma é recursiva a direita ou a esquerda) seja substituída. A produção é uma base para o passo recursivo da definição da lista, veja abaixo um exemplo.

Exemplo 3.24 :

Uma unidade de compilação Java é definida pelas seguintes regras:

```
compilation_unit → package_declaration_opt
                   import_declarations_opt
                   type_declarations_opt;
import_declarations_opt → import_declarations | λ;
import_declarations → import_declarations import_declaration
                      | import_declaration;
... // outras regras
```

O não-terminal `import_declarations` é uma lista (uniforme, no caso) recursiva à esquerda. Ao acrescentar a produção para declarar meta-variáveis, este trecho da gramática fica da seguinte maneira:

```
compilation_unit → package_declaration_opt
                   import_declarations_opt
                   type_declarations_opt;
import_declarations_opt → import_declarations | λ;
import_declarations → import_declarations import_declaration
                      | import_declaration
                      | #import_declarations ID;
... // outras regras
```

Assim, o seguinte padrão passa a ser permitido.

```
package myPack;
#import_declarations imps
class Teste {
    ... // corpo da classe
}
```

Observe que a meta-variável `imps` representa a ocorrência de importações na unidade de compilação acima.

Da mesma maneira, o seguinte padrão também é permitido:

```
package myPack;
#import_declarations imps
import java.util.*;
import java.io.*;
class Teste {
    ... // corpo da classe
}
```

Como as produções do não-terminal `import_declarations` o definem como uma lista recursiva à esquerda, uma meta-variável do tipo **#import_declarations** poderá ser declarada com o objetivo de capturar o inicio de uma lista de importações. Observe que, se a lista fosse recursiva à direita, declarações de meta-variáveis poderiam ser definidas para representar o final da lista.

Assim como foi feito para os não-terminais simples, para permitir que construções sintáticas do tipo **#A** sejam aceitas a partir do não-terminal inicial P', a regra $P' \rightarrow$ **#A** A é acrescentada.

As produções citadas anteriormente são inseridas independentemente se a lista é uniforme ou não. As próximas ações serão guiadas pelo tipo de lista.

- **Uniforme (linhas 14 a 19):** se o não-terminal representa uma lista uniforme, a produção $A \rightarrow$ **#A[** A **]#** é inserida, permitindo que a lista uniforme seja delimitada por marcações de padrão opcional. As próximas ações são tomadas levando em consideração o fato da lista ser recursiva à esquerda ou à direita:
  - **recursiva à esquerda**: são acrescentadas as produções (i) $A \rightarrow A$ **#A[** A **]#** e (ii) $A \rightarrow A$ **#A** ID
  - **recursiva à direita**: são acrescentadas as produções (i) $A \rightarrow$ **#A[** A **]#** A e (ii) $A \rightarrow$ **#A** IDA

  O objetivo das produções marcadas com (i) é permitir a ocorrência de marcações de padrão opcional intercaladas em uma lista. As produções marcadas com (ii) permitem que meta-variáveis apareçam intercaladas em uma lista. Estas meta-variáveis representam seqüências de estruturas sintáticas que compõem a lista onde ela ocorre.

Exemplo 3.25 :

Seja o trecho da gramática de Java apresentado no Exemplo 3.24. O não-terminal `import_declarations` é uma lista uniforme. Ao se acrescentar a produção para declarar meta-variáveis e para permitir ocorrência de marcadores de padrão opcional, o resultado é:

```
compilation_unit → package_declaration_opt
                   import_declarations_opt
                   type_declarations_opt;
import_declarations_opt → import_declarations | λ
      | #import_declarations_opt ID
      | #import_declarations_opt[ import_declarations_opt ]#;
import_declarations → import_declarations import_declaration
      | import_declaration
      | #import_declarations ID
      | import_declarations #import_declarations ID
```

```
        | #import_declarations[ import_declarations ]#
        | import_declarations #import_declarations[ import_declarations ]#;
... // outras regras
```

As modificações realizadas permitem o seguinte padrão:

```
package myPack;
import java.util.*;
#import_declarations[
    #import_declarations imps0
]#
import java.awt.event.*;

class Teste { ... // corpo da classe }
```

Ele representa um programa Java em que as importações devem se iniciar com `import java.util.*;`. Em seguida pode ocorrer uma seqüência de importações (capturada pela variável `imps0`). A lista de importações deve terminar com a importação `import java.awt.event.*`.

Outro exemplo de padrão é apresentado abaixo:

```
package myPack;
import java.util.*;
#import_declarations[
    import java.io.*;
    import java.swing.*; ]#
import java.awt.event.*;
class Teste { ... // corpo da classe }
```

Este é muito parecido com o anterior, exceto pelo fato de que a lista opcional que se segue a importação `import java.util.*` deverá importar exatamente os pacotes `java.io.*` e `java.swing.*`, nesta ordem.

Com estas produções, em qualquer posição de uma lista uniforme poderão ocorrer declarações de meta-variáveis, colocadas como coringas para representar trechos desconhecidos da lista, e também marcadores de padrão opcional, delimitando seqüências de itens que poderão ocorrer na lista.

- **Não Uniforme (linhas 20 a 31):** pela definição de listas não uniformes (Seção 3.1, pág 45), é possível que as suas regras de derivação possuam o o item da base da recursão diferente dos itens que ocorrem no passo recursivo. Veja abaixo produções que definem uma lista não uniforme.

```
X → Y | X Z;
Z → "z";
Y → "y";
```

O não-terminal `X` é uma lista não uniforme. Observe que ele define uma lista de `z`s iniciada por um `y`. Devido à possibilidade de ocorrência deste tipo de definição, as regras de meta-anotações para listas não uniformes são diferentes das regras para listas uniformes.

Inicialmente deve-se criar um não-terminal $A' \notin N'$ que será utilizado nas regras de meta-anotações. O não-terminal A' será definido como uma lista de itens que ocorrem no passo recursivo da definição de A. No caso da gramática descrita anteriormente, A' seria uma lista de `Z`s. A utilização de A' ficará mais clara quando as produções para meta-anotações forem definidas.

As próximas ações são tomadas levando-se em consideração o fato da lista ser recursiva à esquerda ou à direita:

- **recursiva à esquerda:** as seguintes regras são adicionadas a R': (i) $A \rightarrow A$ **#A** ID, para permitir a ocorrência de declarações de meta-variáveis em qualquer ponto da lista; (ii) $A \rightarrow$ **#A[** A **]#**, para permitir a ocorrência de trechos opcionais na base da recursão, i.e., no início da lista; (iii) $A \rightarrow A$ **#A[** A' **]#**, para permitir a ocorrência de trechos opcionais em qualquer ponto da lista. Observe que, neste último caso o não-terminal A' foi utilizado.

  Em seguida, para cada regra da forma $A \rightarrow A\ Z$, onde $Z \in (T \cup N)^*$ as seguintes regras são adicionadas: $A' \rightarrow Z$ e $A' \rightarrow A'\ Z$. Estas regras definem A' como uma lista de itens que ocorrerem no passo recursivo da lista.

- **recursiva à direita:** a seguinte regra é adicionada: $A \rightarrow$ **#A[** A' **]#** A, para permitir a ocorrência de trechos opcionais em qualquer ponto da lista.

  Em seguida, para cada regra da forma $A \rightarrow Z\ A$, onde $Z \in (T \cup N)^*$ as seguintes regras são adicionadas: $A' \rightarrow Z$ e $A' \rightarrow A'\ Z$. Estas regras definem A' como uma lista de itens que ocorrerem no passo recursivo da lista.

  Estruturas definidas como listas não uniformes recursivas à direita são limitadas pois declarações de meta-variáveis só poderão ocorrer no final da lista e trechos opcionais só poderão ocorrer no corpo da mesma, nunca no final.

Finalmente, são acrescentados a R' as seguintes produções para completar as definições do não-terminal A':

- $A' \rightarrow$ **#A** ID e $A' \rightarrow A'$ **#A** ID: estas regras permitem que na lista de itens do passo recursivo ocorram declarações de meta-variáveis;

– $A' \rightarrow$ **#A[** A' **]#** e $A' \rightarrow A'$ **#A[** A' **]#**: estas regras permitem que na lista de itens do passo recursivo ocorram trechos de padrão opcional;

Tomando a gramática de ys e zs, sua gramática de padrões é a seguinte:

```
X → Y | X Z;
    | #X ID | X #X ID
    | #X[ X ]# | X #X[ X' ]#;
X' → Z | X' Z
    | #X ID  | X' #X ID
    | #X[ X' ]# | X' #X[ X' ]#;
Z → "z" | #Z ID;
Y → "y" | #Y ID;
```

Ela permite padrões como:

– `yzz #X zs`: lista com mais de 2 zs;

– `yzz #X[ #X zs ]#`: lista com, no mínimo 2 zs;

– `y #X[ zz ]# z`: lista com 1 ou 3 zs.

As regras acrescentadas à gramática da linguagem objeto para permitir meta-anotações em estruturas sintáticas classificadas como listas permitem a definição de *padrões estranhos* como `#X[ yz ]# zzzz` e `#X[ yzz ]#` para o caso da gramática de ys e zs. O primeiro fato perturbador é que o segundo padrão representa também listas vazias de ys e zs, o que não é possível derivar a partir da gramática da linguagem objeto. Além disso, nestes padrões o trecho de código delimitado pelos marcadores sempre deverão ocorrer nos programas representados pois uma lista não uniforme X sempre deve se iniciar com y e, por definição, padrões representam programas sintaticamente corretos.

Eliminar este tipo de construção utilizando-se apenas regras da gramática da linguagem objeto tornaria o processo de construção da gramática de padrões ainda mais complexo. Em virtude disso, o *plug-in* pode configurar o ambiente MetaJ de forma que ele faça verificações para alertar o usuário sobre a ocorrência deste tipo de construção. Veja na Seção 3.6.

3.6 *Plug-ins*

O último elemento de MetaJ a ser apresentado são os *plug-ins*. Estes são responsáveis por armazenar e gerenciar características inerentes à linguagem objeto. A possibilidade de se anexar ao ambiente diferentes *plug-ins* para diferentes linguagens objetos faz com que MetaJ atenda ao requisito de possibilitar a manipulação de diferentes linguagens objeto.

Durante toda a apresentação do ambiente, tomamos como verdadeira a existência de um *plug-in* de nome "`java`". Nesta seção, descreveremos quais as informações mantidas por este elemento e também como construí-lo. Um *plug-in* é composto, basicamente por 4 elementos:

- **o parser para padrões de programas**: responsável por construir a árvore sintática dos padrões de programas e de programas da linguagem objeto;
- **o verificador de compatibilidade de tipos**: responsável por verificar a propriedade de compatibilidade entre os tipos sintáticos;
- **o conversor de tipos**: componente responsável por encapsular as regras de conversão de tipos sintáticos;
- **o classificador de tipos**: componente capaz de classificar cada tipo sintático como simples, lista uniforme, lista não uniforme ou opcional;

Plug-ins são instalados no ambiente através da sua especificação no arquivo de nome *plugin.spec*, cujo diretório deve estar definido na variável de ambiente *classpath*. Este arquivo pode ocorrer várias declarações de *plug-ins*. Cada declaração deve especificar o nome do *plug-in* e as suas propriedades. A sintaxe para declarar um *plug-in* é a seguinte:

```
<nome do plug-in> {
    (<nome da propriedade> = <valor da propriedade>;)*
}
```

Primeiramente é especificado o nome do *plug-in* seguido por uma seção delimitada por chaves ("`{ ... }`"), na qual deverá ser descrita uma lista de propriedades do *plug-in*. Estas propriedades são informações que poderão ser acessadas pelos meta-programas ou pelo ambiente. São obrigatórias as propriedades: *compatibilityChecker* especificando a classe que implementa o verificador de compatibilidade de tipos; *parser* especificando a classe do *parser*para a linguagem de padrões de programas; *typeInfo* para especificar a classe que implementa o classificador de tipos e *typeConverter* definindo a classe do conversor de tipos. Podem também serem especificadas as propriedades *prettyPrinter*, a classe que deve ser invocada para formatar o código convertido para o formato textual, e *verifyCompleteOptionalList*, que indica se o ambiente deve verificar cada padrão de programa da linguagem objeto para alertar sobre padrões estranhos (ver Seção 3.5.4, pág 80). Estas duas últimas propriedades são opcionais.

Outras propriedades também podem ser especificadas para que posteriormente possam ser consultadas por algum componente do usuário. O ambiente não é influenciado pelo valor de nenhuma destas, mas permitirá que meta-programas acessem tais valores através da interface `PlugIn`. Esta interface define os métodos `Object getProperty (String propName)` e `public void setProperty (String p, Object v)` para recuperar e definir o valor de uma propriedade, respectivamente. O objeto que representa um *plug-in* pode ser recuperado através de seu nome pela utilização do método `Plugin getPlugin(String pnome)` da classe `MetaJSystem`.

A sintaxe para descrição do arquivo *plugins.spec* é definida pela gramática que se encontra no Anexo B. Veja um exemplo de declaração de *plug-in* abaixo.

Exemplo 3.26 :

```
java{
    compatibilityChecker = metaj.plugins.java.JavaCompatibilityChecker;
    parser = metaj.plugins.java.JavaParserAdapter;
    typeInfo = metaj.plugins.java.JavaTypeInfo;
    typeConverter = metaj.plugins.java.JavaTypeConverter;
    prettyPrinter =  metaj.plugins.java.JavaPrettyPrinter;
    verifyCompleteOptionalList = true;
}
```

Como a classificação de tipos sintáticos, as regras de compatibilidade e conversão de tipos podem ser capturadas automaticamente a partir da gramática da linguagem objeto, MetaJ disponibiliza o processador de gramáticas Cup, uma ferramenta capaz de produzir os quatro elementos do *plug-in* a partir da especificação de gramática livre do contexto da linguagem objeto.

3.6.1 O processador de gramáticas Cup

Cup [Hud99] é um gerador de analisadores léxicos LALR para Java. Esta ferramenta possui uma linguagem para especificação de gramáticas livres do contexto cuja sintaxe é muito próxima da BNF. A partir de uma especificação livre de contexto, o gerador Cup é capaz de construir um analisador sintático para a linguagem especificada.

A idéia do processador de gramáticas Cup é produzir semi-automaticamente o *plug-in* para uma linguagem objeto a partir da especificação de sua gramática livre de contexto.

Basicamente, o processador encapsula as regras de geração da gramática de padrões que foram apresentadas na Seção 3.5.4 e o processo de geração dos 4 componentes do *plug-in*. Ele age sobre a gramática Cup da linguagem objeto recebida como entrada da seguinte maneira:

1. extrai os tipos sintáticos definidos;
2. classifica-os segundo as regras de classificação de tipos sintáticos apresentadas na Seção 3.1, pág 45;
3. gera os seguintes elementos:
   (a) uma especificação Cup para a linguagem de padrões de programa seguindo as regras de geração da linguagem de padrões apresentadas na Seção 3.5.4,
   (b) cada um dos componentes do *plug-in*, exceto o *parser*;
   (c) a declaração do *plug-in* que deverá ser acrescentada pelo usuário ao arquivo *plugins.spec*;
   (d) as regras léxicas para reconhecer os novos terminais acrescentados à linguagem objeto de acordo com o processo de geração da linguagem de padrões de programa.

4. invoca o gerador de analisadores sintáticos Cup, que deve estar previamente instalado na máquina, para produzir o analisador sintático para a linguagem de padrões de programa gerada no item 3a.

Como o processador de gramáticas Cup não define um analisador léxico específico a ser utilizado, o usuário deverá, após o processo de geração, acrescentar ao analisador léxico da linguagem objeto as regras léxicas geradas pela ferramenta no passo 3(d). Se o usuário estiver utilizando um gerador de analisadores léxicos como o Lex [LS90] ou Flex [Pax95], esta tarefa é muito simples: basta acrescentá-las à especificação léxica da linguagem objeto e utilizar o gerador para produzir o novo analisador léxico. Além disso, o usuário também deve ajustar a comunicação entre este componente e o analisador sintático gerado.

A Figura 3.2 mostra a tela principal do processador de gramáticas Cup. Para iniciar o processo de geração deve-se informar:

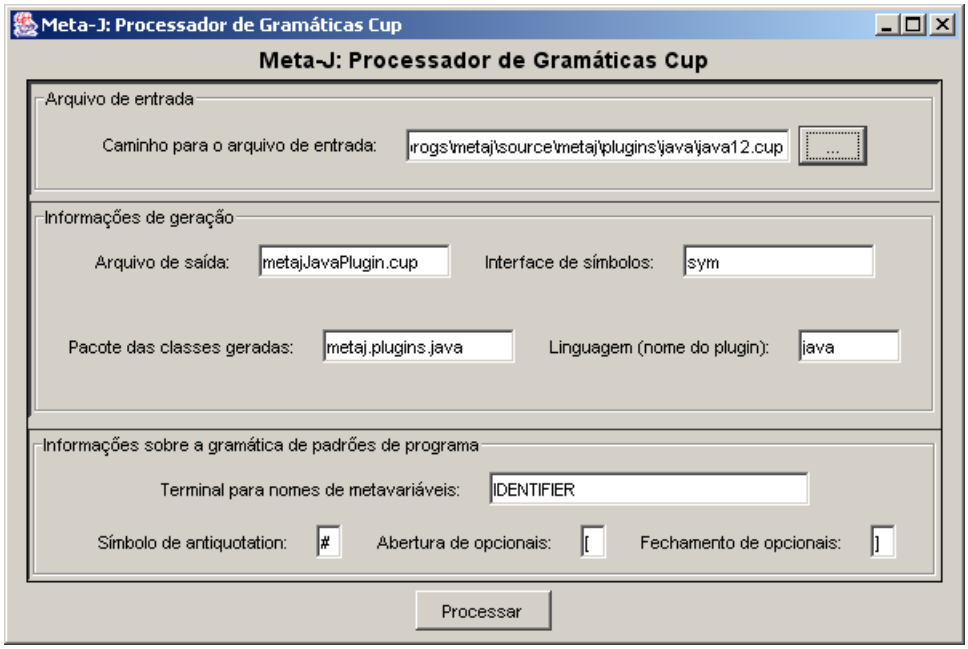


Figura 3.2: Tela do processador de gramáticas Cup

- *caminho para o arquivo de entrada*: o arquivo que contém a especificação da gramática da linguagem objeto;
- *nome do arquivo de saída*: nome do arquivo onde será gravada a gramática de padrões gerada;
- *interface de símbolos*: nome da interface que será utilizada pelo Cup para armazenar os nomes dos terminais utilizados na descrição da gramática. Esta interface é o meio de comunicação entre o analisador sintático gerado pelo Cup e o que analisador léxico utilizado;

- *pacote das classes geradas*: nome do pacote ao qual as classes geradas pertencerão;
- *linguagem (nome do* plug-in*)*: nome do *plug-in* que está sendo gerado;
- *terminal para nomes de variáveis*: terminal utilizado nas regras de declaração de meta-variáveis como representante do identificador declarado (veja Seção 3.5.4, pág 73);
- *símbolo de anti-quotation*: símbolo utilizado como marcador de início de meta-anotações (veja Seção 3.5.1);
- *abertura de opcionais*: símbolo que será utilizado para marcar abertura de trecho de padrão opcional (veja Seção 3.5.1, pág 65);
- *fechamento de opcionais*: símbolo que será utilizado para marcar fechamento de trecho de padrão opcional (veja Seção 3.5.1, pág 65). A este símbolo será automaticamente concatenado o símbolo de *anti-quotation*.

3.6.2 A geração do *plug-in* para Java

Nesta seção apresentaremos como foi gerado o *plug-in* para a linguagem Java.

O primeiro passo foi adquirir a gramática Cup da linguagem Java (veja esta gramática no Apêndice C) e a especificação léxica de Java para o gerador de analisadores léxicos escolhido, Flex [Pax95].

Depois disso, foi acrescentado ao arquivo que contém a especificação da gramática de Java as diretivas (definidas pelo gerador de *parsers* Cup) para conectar o analisador sintático que será gerado pelo Cup com o analisador léxico que será gerado pelo Flex. A diretiva acrescentada foi a seguinte:

```
scan with {: return lexer.next_token(); :};
```

Esta é uma diretiva que indica que o analisador sintático deverá requisitar o próximo *token* da entrada pela chamada ao método `next_token()` do objeto `lexer`. O objeto `lexer` foi declarado dentro do corpo do parser da seguinte maneira:

```
parser code  {: Scanner lexer; ... :}
```

Assim, o analisador léxico que será gerado pelo Flex é conectado ao analisador sintático que será gerado pelo Cup. Observe que a conexão entre os dois é realizada antes de ambos serem gerados. Isso não é um problema pois o processador de gramáticas Cup não modifica a interface do *parser* gerado pelo Cup. O programador poderá realizar a conexão entre os dois componentes seguindo os passos especificados no manual do Cup [Hud99], ignorando o fato de que o processador de gramáticas atuará sobre a especificação da gramática da linguagem objeto.

Em seguida, a gramática de Java descrita no Apêndice C foi processada utilizando-se o processador de gramáticas Cup. Os parâmetros passados para ele foram:

arquivo com a gramática de Java	`java12.cup`
nome do arquivo de saída	`metajJavaPlugin.cup`
nome da interface de símbolos	`sym`
pacote das classes geradas	`metaj.plugins.java`
linguagem (nome do *plug-in*)	`java`
terminal para nomes de meta-variáveis	`IDENTIFIER`. (Foi possível reutilizar a regra léxica de definição identificadores Java para definir identificadores para meta-variáveis).
símbolo de anti-quotation	"`#`"
abertura de opcionais	"`[`"
fechamento de opcionais	"`]`"

A partir desta informação, o processador gerou nove arquivos:

`JavaParser.java`	*parser* construído pelo gerador de *parsers* Cup
`sym.java`	interface de símbolos construída pelo gerador de *parsers* Cup.
`JavaParserAdapter.java`	classe construída pelo processador de gramáticas que faz a adaptação da interface gerada pelo Cup e para aquela exigida por MetaJ
`JavaCompatibilityChecker.java`	verificador de compatibilidade de tipos
`MetaJActionClass.java`	classe que define os métodos que encapsulam as ações para construção da árvore sintática do programa objeto processado.
`JavaTypeConverter.java`	conversor de tipos
`JavaTypeInfo.java`	classe que armazena as classificações dos tipos sintáticos da linguagem objeto
`metajJavaPlugin.cup.lex`	arquivo contendo as definições léxicas a serem acrescentadas no analisador léxico de Java
`metajJavaPlugin.cup`	arquivo que contém a gramática de padrões da linguagem objeto processada
`java_plugin_specification.spec`	arquivo com a declaração do *plug-in* de nome java

Em seguida, foram acrescentadas à especificação léxica da linguagem Java as regras léxicas geradas pelo processador de gramáticas Cup (arquivo `metajJavaPlugin.cup.lex`). O gerador de analisadores léxicos Flex [Pax95] foi utilizado para produzir o analisador desejado.

Finalmente, a declaração do *plug-in*, gerada no arquivo `java_plugin_specification.spec`, foi acrescentada ao arquivo `plugins.spec` que acompanha o ambiente MetaJ e todas as classes Java foram compiladas.

3.7 A implementação da refatoração *change_variable_name* para variáveis locais

Refatorações são reestruturações que preservam o comportamento observável do programa transformado [Opd92]. Refatorações podem ser utilizadas para extrair elementos reutilizáveis de

um sistema, melhorar a consistência entre os seus componentes e dar suporte ao desenvolvimento iterativo de *frameworks* orientados a objetos [Opd92].

Nesta seção será apresentada a implementação da refatoração *change_ variable_ name* proposta por Opdyke [Opd92]. A escolha de uma refatoração como exemplo permitirá ilustrar a utilização de MetaJ para analisar e gerar programas ao mesmo tempo.

O objetivo de *change_ variable_ name* é:

> *Modificar o nome de uma variável. A modificação do nome é refletida por todo seu escopo. A variável pode ser global, local, membro de classe ou argumento de uma função. As regras da linguagem objeto define onde a variável é visível. [Opd92]*

Tal refatoração será implementa apenas para o caso de variáveis locais em métodos. Para realizar esta transformação são necessárias as seguintes informações: (i) o nome da variável local (n), (ii) o novo nome desta variável (s), (iii) a assinatura do método onde tal variável foi declarada (am), (iv) o nome da classe que declara este método (c), (v) nome do arquivo que contém tal classe (a).

A mecânica da refatoração é a seguinte:

1. encontrar a classe c dentro do arquivo a;
2. encontrar o método de assinatura am no corpo de c;
3. verificar a existência de alguma declaração de variável local de nome s. Caso isso aconteça, um erro será lançado;
4. prefixar todos os acessos a campos de nome s com "`this`" para evitar a captura de nomes após a aplicação da refatoração;
5. modificar todas as ocorrências de n por s;
6. modificar a declaração de n por s.

A refatoração foi encapsulada no método `run` da classe `metaj.examples.refactorings.-renameLocalVariable.RenameLocalVariable`.

```
...
public class RenameLocalVariable {
    ...
    /* outros métodos auxiliares */
    private void run(String a, String c, String am,
                                 String n, String s)throws Exception{
        ChooseMethodInClass procMet = new ChooseMethodInClass();
        procMet.getClassId().set(c);
        procMet.getMh().set(am);
```

```
        if(procMet.matchFile(a)){
            PReference bs = procMet.getBs();
            // Verificar a existência de uma variável com nome s
            if(procurarDeclaracao(s, bs)){
                System.out.println("Erro!...");
                return;
            }else{
                //Adicionar this aos acessos a campos
                adicionarThisAosAcessosCampos(s, bs);
            }
            //Atualizar acessos e declaração da variável local
            atualizarVariavelLocal(n, s, bs);
            //Reescrevendo o arquivo de saída
            procurarMetodo.toFile(file);
            return;
        }
        System.out.println("Erro!...");
    }
}
```

Primeiramente é utilizado o *template* `ChooseMethodInClass` para localizar a class `c` e o método de assinatura `am` no arquivo `a`. Caso eles não sejam encontrados, um erro é reportado. Veja abaixo a declaração de `ChooseMethodInClass`:

```
template #compilation_unit ChooseMethodInClass #{
    #package_declaration_opt[ package #name pack; ]#
    #import_declarations_opt[ #import_declarations imps ]#

    #type_declarations[ #type_declarations tds ]#

    #modifiers_opt[ #modifiers m ]# class #identifier classId
                                  #super_opt[ extends #name superClassName ]#
                                             #interfaces_opt[#interfaces is]# {
        #class_body_declarations[ #class_body_declarations cbds1 ]#
        #method_header mh { #block_statements bs }
        #class_body_declarations[ #class_body_declarations cbds2 ]#
    }

    #type_declarations[ #type_declarations tds1 ]#
}#
```

Através da meta-variável `bs` deste *template* é possível recuperar o corpo do método de assinatura `mh`. Observe que o valor de `a` foi atribuído à meta-variável `mh` e o valor de `c` foi atribuído

à `classId`. Através da utilização da operação `match` do *template* escolhe-se o método desejado. Pela chamada ao método `procurarDeclaracao` é verificada a existência de uma declaração de variável de nome `s` em `bs`. Se esta declaração for encontrada, um erro é reportado. Caso contrário, os acessos aos campos de nome `s` são prefixados com "`this`". Isso é feito pela chamada ao método `adicionarThisAosAcessosCampos`. Finalmente, através de uma chamada a `atualizarNomesVariavelLocal`, os acessos à variável de nome `n` e sua declaração são modificados para `s`.

O comportamento dos métodos `procurarDeclaracao`, `adicionarThisAosAcessosCampos` e `atualizarNomesVariavelLocal` é apresentado abaixo:

- `procurarDeclaracao(String vid, Reference c)`: este método cria um iterador a partir da referência-p `c` recebida como parâmetro e utiliza-o para procurar um `#variable_declarator_id` (identificador de declaração de variável) com nome `vid`. Abaixo pode ser visto o corpo deste método.

```
public boolean procurarDeclaracao(String vid, Reference c){
     Iterator it = c.getIterator();
     while(it.hasNextIn()){
         it.nextIn();
         Reference r = it.get();
         if(r.hasType("java.#variable_declarator_id") &&
          r.match(vir)){
             return true;
         }
     }
     return false;
 }
```

  Observe a utilização do teste de tipos para alcançar as declarações de variáveis.

- `adicionarThisAosAcessosCampos(String campo, Reference r)`: este método percorre o conteúdo da referência `r` recebida como parâmetro em busca de termos de expressões que representem acessos ao campo de nome `campo`. Isso é feito pela busca a todas as construções do tipo `#postfix_expression` que são prefixadas com `campo`.

```
private void adicionarThisAosAcessosCampo(String campo, Reference contexto){
     substituirPrefixo(campo, "this." + campo, contexto.getIterator());
}

private void substituirPrefixo(String pref, String nPref, Iterator it){
     while(it.hasNextIn()){
         it.nextIn();
```

```
            Reference r = it.get();
            String rv = r.toString();
            if(r.hasType("java.#postfix_expression")
                    && (rv.startsWith(pref+".") || rv.equals(campo))){
                r.replace(nPref + r.toString().substring(pref.length()));
            }
        }
    }
```

O `adicionarThisAosAcessosCampo` prefixa os acessos ao campo de nome `campo` através da chamada ao método `substituirPrefixo`. Este percorre todo o iterador recebido como último parâmetro em busca de construções `#postfix_expression` prefixadas com `pref`, seu primeiro parâmetro. Quando encontrada, o prefixo de tal expressão é substituído por `nPref`;

- `atualizarVariavelLocal(String nomeVar, String novoNome, Reference bs)`: este método é um pouco mais complexo que os anteriores. Primeiramente, ele posiciona um iterador na declaração de variável de nome `nomeVar` através da chamada ao método `posicionarDeclaracao`. Este método retorna um objeto `Pair` que encapsula um iterador e uma variável booleana que indica se a declaração foi encontrada ou não. Se a declaração for encontrada, o iterador retornado estará posicionado exatamente nesta estrutura. Após a chamada ao método `posicionarDeclaracao`, o `atualizarNomesVariavelLocal` substitui o nome `nomeVar` utilizado na declaração na qual o iterador está posicionado por `novoNome`. Em seguida, todas as estruturas que se seguem a esta declaração são percorridas em busca de utilizações da variável renomeada. Esta tarefa é realizada pelo método `substituirPrefixo` já apresentado anteriormente.

```
private void atualizarVariavelLocal (String nomeVar,
                                     String novoNome, Reference bs){
    Pair p = posicionarDeclaracao(nomeVar, bs.getIterator());
    Iterator it = p.it;
    Reference declId = null;
    if(p.b) declId = it.get();
    else{
        System.out.println("Não existe variável declarada!");
        return;
    }
    declId.replace(novoNome);
    substituirPrefixo(nomeVar, novoNome, it);
}
```

O iterador retornado por `posicionarDeclaracao` só percorre o escopo da declaração encontrada, i.e., o bloco onde ela foi declarada. Veja abaixo como este método foi implementado.

```
private Pair posicionarDeclaracao (String nomeVar, Iterator it){
    while(it.hasNextIn()){
        it.nextIn();
        Reference r = it.get();
        if(r.hasType("java.#block")){
            Iterator tmp = it.get().getIterator();
            tmp.nextIn();
            Pair p = posicionarDeclaracao(nomeVar, tmp);
            if(p.b) return p;
            if(!it.hasNextOver()){
                return new Pair(false, it);
            }
            it.nextOver();
            r = it.get();
        }
        if(r.hasType("java.#variable_declarator_id") && r.match(nomeVar)){
            return new Pair(true, it);
        }
    }
    return new Pair(false, it);
}
```

Utilizando-se o iterador recebido como parâmetro, o método procura construções `#variable_declarator_id` que casem com o nome recebido como parâmetro. Quando encontrada, o método retorna o iterador atual e um flag indicando que a declaração foi encontrada.

Para toda construções do tipo `#block` alcançada, é criado um iterador para explorá-la exclusivamente. Para isso é realizada uma chamada recursiva a este mesmo método. Se o par retornado por esta chamada indicar que a declaração da variável `nomeVar` foi encontrada, então este mesmo par é retornado. Caso contrário, o bloco é ignorado (chamando-se `nextOver`) e continua-se o caminhamento pelo código

3.7.1 Conclusões

A implementação da refatoração ilustrou a utilização da ferramenta para aplicações um pouco mais complexas que simplesmente análise ou geração de pequenos trechos de código.

A possibilidade de se trabalhar com códigos fonte no seu formato textual, representados como literais *string* ou lidos a partir do conteúdo de arquivos, facilitou bastante a especificação de trechos do programa objeto. A utilização da sintaxe concreta da linguagem objeto para descrever padrões de programa facilitou a localização de elementos do código objeto. A possibilidade de utilização do teste de tipo (`hasType`) juntamente com iteradores permitiu encontrar facilmente

ocorrências de uma determinada estrutura sintática da linguagem objeto. Além disso, a possibilidade de se utilizar os recursos básicos de Java (classes de I/O) facilitou a interação com o usuário. Estes foram utilizados para alertar sobre violações das pré-condições de aplicação da refatoração.

Apesar destas facilidades, a reestruturação de programas utilizando-se MetaJ, mais especificamente buscas por trechos específicos do programa objeto, demandou o tratamento de muitos detalhes operacionais devido principalmente à utilização intensa de iteradores.

No Capítulo 5 será apresentada uma extensão de MetaJ que permite especificar consultas em código de maneira mais declarativa. Vale destacar que esta extensão não exclui a utilização de iteradores.

3.8 Comentários finais

Neste capítulo foi apresentado MetaJ, um ambiente para meta-programação com as seguintes características: (i) se baseia no paradigma orientado a objetos, (ii) disponibiliza padrões de programas *by example*, (iii) foi projetado como um conjunto de abstrações que foram implementadas como classes Java e que encapsulam a essência da meta-programação. Como conseqüência do projeto do ambiente, meta-programas em MetaJ nada mais são que programas Java que utilizam estas abstrações. Outra conseqüência é que estes meta-programas podem utilizar livremente os recursos básicos de Java, como I/O, GUI, *multi-threading*, *networking* e tratamento de exceções. Desta maneira, MetaJ apresenta uma solução para as principais críticas às ferramentas de meta-programação apresentadas no Capítulo 2.

As abstrações definidas por MetaJ são: referências-p, *templates*, iteradores e *plug-ins*.

Referências-p são abstrações que encapsulam trechos de programas objeto. As principais características destes elementos são: disponibilização de métodos que permitem a manipulação de código de maneira bastante simples, garantindo a consistência sintática do programa objeto. As principais operações são `set`, que permite definir o valor de uma referência-p, `toString`, que converte seu valor para uma *string* de código e `match`, que permite a comparação estrutural de códigos. Uma facilidade oferecida por esta abstração é a possibilidade de se atribuir a elas *strings* de código, que são convertidas automaticamente para uma representação interna. Assim como JaTS, referências-p disponibilizam também métodos especiais para tratar estruturas sintáticas classificadas como listas, permitindo que seus itens sejam consultados, removidos ou mesmo que novos itens sejam adicionados. Referências-p podem ser criadas em qualquer parte do meta-programa.

A partir de uma referência-p é possível criar iteradores para caminhar pelo código armazenado por ela. Iteradores encapsulam um caminhamento *top-down* pelo programa objeto oferecendo controle total sobre os passos do caminhamento. A cada passo é possível decidir por explorar a estrutura atual, ignorá-la, substituí-la ou mesmo interromper o processo de caminhamento.

Templates encapsulam padrões de programas e permitem extrair informações do código através de casamento de padrões. Estes elementos são declarados em módulos especiais e compilados pelo *compilador de templates*. Este compilador traduz as declarações de *templates* em declarações de classes Java, o que permite a utilização destes elementos em qualquer meta-programa. Esta característica permite que *templates* sejam reutilizados em outras oportunidades. Nas ferramentas apresentadas no Capítulo 2, padrões de programas apareciam descritos dentro do próprio meta-programa que o utilizava, isso impedia sua reutilização. Além disso, por serem utilizados como classes Java, instâncias diferentes de um mesmo *template* podem ser criadas e manipuladas de maneira independente sem que seja necessário reescrevê-lo.

Templates disponibilizam os métodos: `match`, que verifica se um determinado trecho de programa está de acordo com o padrão encapsulado, `getXXX` (onde `XXX` é o nome de uma meta-variável do padrão de programa), para recuperar o valor da meta-variável `XXX` e `toString`, que converte o padrão em um *string* representando o trecho de programa da linguagem objeto.

Plug-ins armazenam informações sobre a linguagem objeto: seus tipos sintáticos (inclusive as propriedades convertibilidade e compatibilidade) e o parser para programas e padrões de programas da linguagem objeto. A possibilidade de se anexar diferentes *plug-ins* para diferentes linguagens objeto permite a MetaJ atender ao requisito de manipulação de múltiplas linguagens objeto.

Neste capítulo foi apresentado também o processador de gramáticas Cup, uma ferramenta capaz de gerar o *plug-in* da linguagem objeto a partir de sua gramática livre do contexto.

Além de pequenos exemplos apresentados durante a descrição de cada uma destas abstrações, foi também apresentada a implementação da refatoração *change_ variable_ name* para variáveis locais. Com esta, foi mostrado que aplicações realmente complexas podem ser implementadas com relativa facilidade utilizando-se a ferramenta. Este exemplo mostrou a utilização efetiva de cada um dos recursos de MetaJ. No entanto, esta implementação também apresentou uma falha da ferramenta: buscas por trechos específicos do programa objeto demandaram o tratamento de muitos detalhes operacionais devido principalmente à utilização intensa de iteradores. No Capítulo 5 é apresentada uma possível solução para este problema: SCQL, uma extensão de MetaJ para possibilitar a realização de consultas declarativas em programas objeto.

Capítulo 4

A Implementação de MetaJ

4.1 Introdução

MetaJ foi implementado em 3 grandes componentes: (i) um *framework* para meta-programação, onde as abstrações descritas nas seções 3.3 a 3.6 foram definidas, (ii) o compilador de *templates* e (iii) o processador de gramáticas Cup. Nas seções que se seguem, os detalhes sobre a implementação de cada um deles serão apresentados.

4.2 O *Framework*

O *framework* para meta-programação é composto por interfaces, classes concretas e abstratas que implementam os recursos do ambiente. Alguns destes recursos foram realizados como componentes *black-box*, i.e., são utilizados através do mecanismo de composição. São eles as referências-p, os iteradores e as classes exceções e erros do ambiente. Outros foram realizados como componentes *white-box*, i.e., componentes utilizados através do mecanismo de herança. São eles os *templates* e os *plug-ins*. No primeiro caso, um *template* do usuário é uma extensão da classe `AbstractTemplate` definida por MetaJ. No caso dos *plug-ins*, cada um de seus componentes são implementações das interfaces definidas pelo *framework*. Mais detalhes sobre a implementação dos elementos do *framework* serão tratados nas seções 4.2.2 e 4.2.3

O *framework* para meta-programação foi implementado no pacote `metaj.framework`. Neste, apenas uma classe é implementada: `MetaJSystem`. Ela disponibiliza métodos para se criar referências-p (`createPReference`), como foi exemplificado na Seção 3.3, e acessar os *plug-ins* (`getPlugIn`), como apresentado na Seção 3.6. Os sub-pacotes de `metaj.framework` são:

- `metaj.framework.abstractions`: neste são implementadas cada uma das abstrações de meta-programação: referências-p, iteradores e *templates*. As seções 4.2.2 e 4.2.3 apresentarão cada uma delas mais detalhadamente;
- `metaj.framework.exceptions`: neste pacote estão implementadas as classes que represen-

tam erros e exceções do sistema. Dentre elas vale destacar `MetaJError` e `MetaJException`, superclasses comuns para todos os erros e exceções do ambiente, respectivamente;

- `metaj.framework.tree`: neste pacote são implementadas classes utilizadas na construção de árvores sintáticas para representar internamente programas objeto e padrões de programas. A Seção 4.2.1 apresentará este pacote mais detalhadamente;

- `metaj.framework.operations`: neste pacote são implementadas as operações sobre programas (árvores sintáticas): casamento (`match`), conversão para formato textual (`toString`) e duplicação (`duplicate`);

- `metaj.framework.parser`, `metaj.framework.typeSystem` e `metaj.framework.plugInsLoader`: os dois primeiros pacotes definem interfaces e classes abstratas que deverão ser implementadas pelos elementos que compõem um *plug-in*. O último, `metaj.framework.plugInsLoader`, implementa o gerenciador de *plug-ins*, componente responsável por carregar e gerenciar os *plug-ins* registrados no ambiente. Maiores detalhes serão tratados na Seção 4.2.3.

4.2.1 Representação interna de programas objeto e padrões de programa

Programas objeto e padrões de programas são representados internamente como árvores de sintaxe. No pacote `metaj.frameworks.tree` são definidas interfaces e classes que implementam os nodos utilizados na construção da estrutura, são elas:

- `Node`: interface comum para todos os nodos da árvore sintática. Ela define os métodos: (i) `get`, para recuperar um nodo filho, (ii) `getNumChildren`, para recuperar o número de nodos filhos, (iii) `add`, para adicionar um nodo filho ao nodo atual, e (iv) `setType`, para definir o tipo de construção sintática representada pelo nodo;

- `SimpleNode`: nodo utilizado para representar as ocorrências de estruturas sintáticas da linguagem objeto em programas ou padrões de programas;

- `TokenNode`: nodo utilizado para representar e armazenar os *tokens* utilizados no programa objeto;

- `MetaVariable`: nodo utilizado para representar a ocorrência de uma declaração de meta-variável em um padrão de programa. Eles armazenam o tipo da meta-variável e o seu nome. Este tipo de nodo nunca ocorre em árvores sintáticas que representam programas da linguagem objeto;

- `OptionalNode`: utilizado para marcar uma sub-árvore opcional, i.e., a ocorrência de um trecho de padrão de programa opcional. Assim como nodos `MetaVariable`, `OptionalNode`'s só podem ocorrer em padrões de programa;

Estas classes são utilizadas pelos *parsers* definidos pelos *plug-ins* para construir a árvore sintática de programas e de padrões de programa. No primeiro caso, são utilizados apenas nodos do tipo `SimpleNode` e `TokenNode`. No segundo, além destes, podem também serem utilizados nodos do tipo `MetaVariable` e `OptionalNode` para representar ocorrências de meta-anotações.

O diagrama de classes da Figura 4.2.1 resume o relacionamento entre as classes citadas anteriormente.

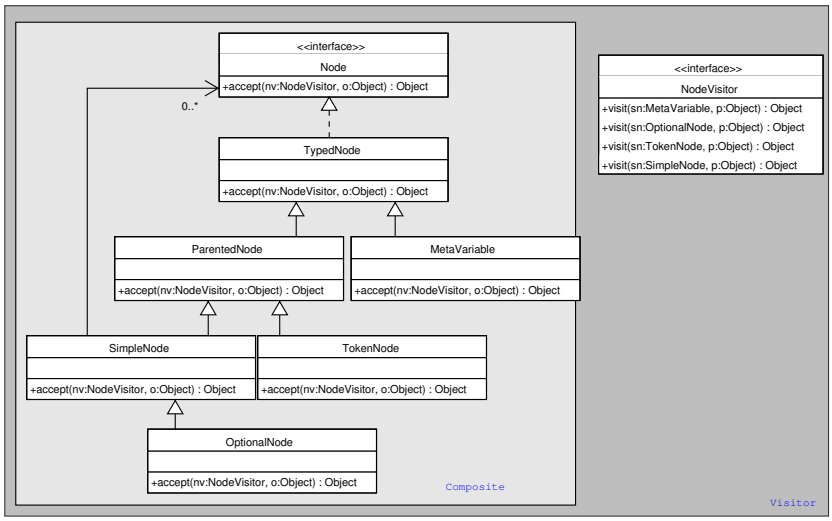


Figura 4.1: Classes do pacote `metaj.framework.tree`.

Os nodos sintáticos foram projetados de acordo o padrão *Composite* [GHJV94]. O objetivo deste padrão é:

> *Compor objetos em estruturas de árvore para representarem hierarquias parte-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composição de objetos.* [GHJV94].

Neste padrão, `SimpleNode` e `OptionalNode` fazem o papel de *composite nodes*, nodos que são composições de outros. `MetaVariable` e `TokenNode` fazem o papel de *leaf nodes*, nodos folha. `Node` é a interface comum para todos os componentes.

Outro padrão utilizado foi o *Visitor* [GHJV94]. A intenção deste padrão é:

> *Representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.* [GHJV94].

Fazem parte deste padrão os métodos `accept` definidos em cada um dos tipos de nodos e a interface `NodeVisitor`. Esta interface permitiu uma maior modularização do sistema, facilitando a implementação de algumas operações sobre árvores sintáticas (veja próxima seção). Este padrão permite que novas operações sejam definidas sem que seja necessário modificar as classes que representam nodos da árvore sintática.

As operações do pacote `metaj.framework.operations`

Operações de casamento de padrões, conversão de árvores sintáticas para o formato textual e duplicação de código são algumas das operações disponibilizadas pelas referências-p e *templates*.

No projeto do ambiente foram implementados componentes responsáveis por aplicar tais operações sobre árvores. Eles podem ser encontrados no pacote `metaj.framework.operations`. A maioria deles foram realizados como *Visitors*, i.e., objetos que caminham sobre árvores aplicando operações sobre seus nodos.

A operação de casamento é implementada pela classe `MatchOperation` do pacote `metaj.framework.operations.match`. Devido à sua complexidade, esta foi a única das operações que não foi implementada como um *Visitor*. Ela recebe como parâmetro duas árvores a_1 e a_2 para serem comparadas estruturalmente. O primeiro parâmetro a_1 pode ser tanto uma árvore que representa um padrão de programa quanto uma árvore que representa um programa da linguagem objeto. Por outro lado, a_2 não pode ser um padrão de programa. Para se realizar a operação também é necessário um objeto do tipo `Environment` que liga referências-p às meta-variáveis que aparecem na árvore a_1. Os trechos de programas que foram casados com cada uma das meta-variáveis são atribuídos às suas referências-p relacionadas pelo objeto `Environment`. Referências-p com valor predefinido não são modificadas. Neste caso, o seu valor deve ser estruturalmente equivalente àquele encontrado na árvore a_2.

A operação de conversão para forma textual, é implementada pela classe `PrintVistor` do pacote `metaj.framework.operations.print`. Esta operação recebe dois parâmetros: uma árvore a (que pode ser um padrão ou um programa da linguagem objeto) a ser convertida e um objeto `Environment` que liga referências-p às meta-variáveis que aparecem na árvore a. Estas referências-p definem quais são os trechos de programa que substituirão as ocorrências de meta-variáveis do padrão representado por a. Esta operação é invocada pelos métodos `toFile` e `toString` das abstrações referência-p e *template*.

A operação de reconstrução é equivalente à implementada por `PrintVistor`, exceto pelo fato de que o seu resultado é uma nova árvore sintática, e não um *string* de caracteres. Implementada pela classe `BuildVisitor`, do pacote `metaj.framework.operations.build`, ela é invocada pelos métodos `toReference` de *templates* e `duplicate` de referências-p.

A operação `isComposedBy` disponibilizada pelas referências-p é implementada pela classe `CompositionVisitor` pertencente ao pacote `metaj.framework.operations.composition`. Esta operação recebe como parâmetro duas árvores sintáticas a_1 e a_2. Nenhuma delas pode ser um padrão de programa. A operação verifica se a_2 ocorre em algum ponto da estrutura de a_1.

4.2.2 A implementação das abstrações Referências-p, Iteradores e *Templates*

A abstração referência-p é implementada, principalmente, no pacote `metaj.framework.abstractions.pReference`. A interface `PReference` define cada um dos métodos apresentados na Seção 3.3. `PReferenceImpl` é a implementação desta interface. Um objeto `PReferenceImpl`

encapsula um objeto `ValueList` utilizado, para armazenar o valor atribuído a ela, e um `String`, que armazena o tipo da referência-p. O objeto `ValueList` é uma implementação da interface `Node`. Isso permite que as operações do pacote `metaj.framework.operations` possam ser aplicados diretamente a partir dele.

A verificação sobre compatibilidade e convertibilidade de tipos realizada nos métodos das referências-p são apenas chamadas aos objetos componentes de *plug-ins*. Referências-p recuperam informações sobre a linguagem objeto utilizando seu tipo. Através do método `getPlugIn` da classe `metaj.framework.MetaJSystem` é possível recuperar o *plug-in*, através do qual será possível consultar as informações gerenciadas por ele.

As operações, `match`, `toString`, `duplicate` e `isComposedBy` são apenas chamadas a objetos cujas classes foram definidas no pacote `metaj.framework.operations`. As outras, `equals`, `getSize`, `getIterator`, `add`, `set`, `replace` e `remove`, são implementados diretamente no corpo da classe `PReferenceImpl`. O método `getIterator` retorna objetos da classe `IteratorImpl` que implementa a interface `metaj.framework.abstractions.iterator.Iterator`. A implementação de iteradores depende de informações encapsuladas pelas referências-p. Para permitir acesso privilegiado a estas informações, a classe `IteratorImpl` (que realiza a interface `Iterator`) se encontra no pacote `metaj.framework.abstractions.reference`.

A classe abstrata `AbstractTemplate`, implementada no pacote `metaj.framework.abstractions.template`, é a superclasse comum para todos os *templates* gerados pelo compilador de *templates* (ver Seção 3.20). Ela implementa os métodos comuns a qualquer *template* (métodos `match`, `toString` e `toPReference`) e também possuem um objeto `Environment`, que liga uma referência-p a cada de meta-variável do padrão encapsulado pelo *template*. Este objeto será passado como argumento para as chamadas às operações de casamento, conversão para o formato textual e reconstrução definidas no pacote `metaj.framework.operations`. É de responsabilidade do *template* gerado definir e disponibilizar o texto do padrão encapsulado, implementar os métodos `getXXX` para cada meta-variável `XXX` declarada por ele e registrar cada uma delas no objeto `Environment` definido pela classe `AbstractTemplate`.

4.2.3 *Plug-ins*

Como a implementação dos elementos dos *plug-ins* é diferente para cada linguagem objeto, MetaJ define apenas as interfaces que deverão ser implementadas por cada um deles. São elas:

- `metaj.framework.parser.Parser`: que define as operações `setType (String t)` e `ParseData parse(InputStream is)`. A primeira é um método que deve ser invocado para definir qual o tipo de construção sintática será fornecida como entrada para o método `parse`. Este é responsável por iniciar o processo de reconhecimento do código (programa objeto ou padrão de programa) recebido como parâmetro e pela construção da sua árvore sintática. O método `parse` deverá retornar um objeto do tipo `metaj.framework.ParseData` que encapsula a árvore sintática construída e também um objeto `Environment` com o nome de

cada uma das meta-variáveis que ocorreram no padrão de programa ligadas a uma nova instância de `Reference`. Se o código fornecido para o método `parse` for um programa, então este `Environment` é retornado vazio;

- `metaj.framework.typeSystem.CompatibilityChecker`: esta interface define, basicamente, dois métodos: `boolean verifyAdditionCompatibility (String t1, String t2)` e `boolean verifyUpdateCompatibility (String t1, String t2)` para verificar, respectivamente, a compatibilidade de adição e substituição para os tipos sintáticos `t1` e `t2`;

- `metaj.framework.typeSystem.TypeConverter`: esta interface define o método `Node convertTo(Node n, String t)`. Ele converte o valor `n` recebido como parâmetro para o tipo `t`. O resultado da conversão é retornado. O método retorna `null` caso não seja possível converter `n` para o tipo `t`;

- `metaj.framework.typeSystem.TypeInfo`: esta interface define operações para verificar a classificação de um determinado tipo (ex.: `isSimple`, `isOptional`, etc.) e também operações que retornam um *array* com todos os tipos sintáticos de uma determinada classe (ex.: `getSimples`, `getOptionals`, `getLists`, etc.).

A classe `metaj.framework.pluginLoader.PluginLoader` é responsável por carregar e gerenciar os *plug-ins* registrados no ambiente. Objetos desta classe são capazes de carregar sob demanda objetos do tipo `metaj.framework.pluginLoader.PlugIn` para representar cada *plug-in* registrado. No pacote `metaj.framework.pluginLoader.pluginDeclarationsParser` foi implementado o *parser* para reconhecer o arquivo de especificação de *plug-ins* (ver Seção 3.6, pág. 81).

Para facilitar a utilização das operações definidas pelos vários elementos do plug-in foi implementada a classe `PluginAccessObject`. Esta classe foi implementada segundo o padrão de projeto *Façade* [GHJV94] cujo objetivo é:

> *Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado. [GHJV94]*

Isso permite que, a partir do objeto `PluginAccessObject`, a verificação de compatibilidade, conversão de tipos, construção de árvores sintáticas e recuperação de informações sobre tipos – serviços disponibilizados por diferentes componentes do *plug-in* – possam ser realizadas a partir de um único objeto. Uma instância de `PluginAccessObject` mantém uma instância de `PluginLoader` para carregar, quando necessário, os *plug-ins*.

A classe `PluginAccessObject` também implementa o padrão de projeto *Singleton* [GHJV94], cujo objetivo é:

Garantir que uma classe tenha apenas uma instância e fornecer um ponto global de acesso a ela. [GHJV94]

O ponto global de acesso é o método `getPlugInAccessObject` que retorna a única instância desta classe. A partir do objeto `PluginAccessObject` é possível acessar um *plug-in* específico através da chamada ao método `getPlugin(String nomePlugin)`. Este é equivalente ao método de mesma assinatura da classe `MetaJSystem`.

A Figura 4.2 apresenta um diagrama de classes que descreve o relacionamento entre as principais classes descritas acima. Observe que a classe `PluginAccessObject` depende de cada um dos elementos dos *plug-ins* e também de um `PluginLoader`.

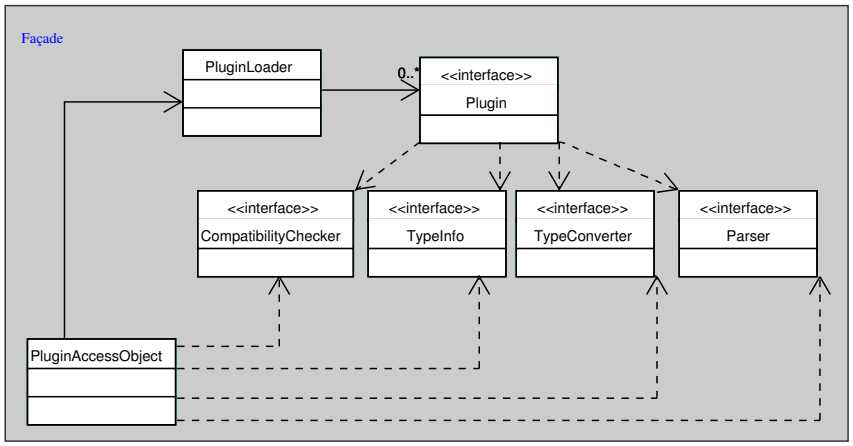


Figura 4.2: Interfaces para os componentes de *plug-ins* e as classes `PluginLoader` e `PluginAccessObject`

4.3 O processador de gramáticas Cup

O processador de gramáticas Cup (Seção 3.6.1) foi implementado separadamente de MetaJ, no pacote `cupProcessor`. Os principais componentes desta ferramenta são:

- `cupProcessor.parser`: esta é a classe que implementa o *parser* para reconhecer arquivos contendo especificações Cup. Este componente constrói uma árvore sintática para representar o arquivo utilizando-se de objetos das classes definidas no pacote `cupProcessor.tree`. Estas classes foram projetadas segundo o padrão de projeto *Composite* [GHJV94] e *Visitor* [GHJV94], de maneira equivalente àquela adotada no pacote `metaj.framework.tree`. A interface `CupNode` é comum para todos os tipos de nodos sintáticos. A interface `CupNodeVisitor` define os métodos que deverão ser implementados pelos *Visitors* definidos em outros pacotes;

- `cupProcessor.productionsCollection.ProductionsCollectionVisitor`: este componente: (i) analisa a árvore sintática construída pelo *parser*, (ii) constrói uma representação interna para as produções especificadas, (iii) classifica os não-terminais da gramática e também (iv) coleta informações sobre compatibilidade e convertibilidade de tipos sintáticos. Esta classe implementa a interface `cupProcessor.tree.CupNodeVisitor`;

- `cupProcessor.printer.CupNodePrintVisitor`: responsável por reimprimir a gramática Cup acrescentando as produções para meta-anotações (ver Seção 3.5.4). Esta classe também implementa a interface `cupProcessor.tree.CupNodeVisitor`;

A classe que inicia o processamento do arquivo contendo especificação de gramática Cup é a `cupProcessor.Main`. Ela faz chamadas aos métodos dos três componentes apresentados acima e, utilizando-se das informações coletadas por um objeto da classe `ProductionsCollectionVisitor`, gera o verificador de compatibilidade de tipos, o conversor de tipos e o classificador de tipos para o *plug-in*. Para finalizar o processamento de uma gramática, o gerador de *parser* Cup é invocado para processar a gramática gerada pelo `CupNodePrintVisitor`, produzindo, desta maneira, o analisador sintático para a linguagem de padrões.

4.4 O compilador de declarações de *Templates*

O compilador de declarações de templates (Seção 3.20, pág. 69) é implementado no pacote `templateCompiler` pela classe `TemplateCompiler`. Esta classe utiliza os recursos de MetaJ para fazer análise de declarações de *templates* e geração da classe Java referente ao *template* declarado. Por esta razão, para utilizá-lo é necessário que MetaJ esteja instalada, com, no mínimo, dois *plug-ins* registrados: o de nome "`java`", citado e utilizado nos exemplos apresentados nas seções anteriores, e outro de nome "`templates`" gerado a partir da gramática que especifica as regras para declaração de *templates* apresentada no Apêndice A.

A classe `TemplateCompiler` utiliza referências-p e *templates* implementados manualmente para facilitar o processo de análise de declarações de *templates* e geração de classes Java.

Capítulo 5

A Linguagem de Consultas

5.1 Introdução

A principal crítica a MetaJ é o fato de que buscas por trechos específicos do programa objeto demandam o tratamento muitos detalhes operacionais. Esta deficiência está relacionada diretamente a utilização intensa de iteradores. Devido ao fato de que, na construção de meta-programas, buscas por determinado tipo de estrutura sintática são bastante freqüentes, é necessário oferecer uma forma de especificar em alto nível consultas sobre um trecho de código. Uma consulta, em sua essência, é uma seleção de construções sintáticas de um determinado tipo que ocorrem dentro de um programa objeto.

Neste capítulo será apresentada SCQL – ***S**ource **C**ode **Q**uery **L**anguage* – uma linguagem declarativa para expressar consultas sobre o código de um programa objeto. A linguagem é apresentada como uma extensão de MetaJ. Por esta razão, os conceitos de tipos sintáticos (ver Seção 3.2, pág. 44), classificação de tipos sintáticos (ver Seção 3.2.1, pág. 45) e referências-p (ver Seção 3.3) também são utilizados. O resultado de uma consulta em SCQL nada mais é que um conjunto de referências-p para fragmentos do programa objeto sobre o qual a consulta foi aplicada. SCQL pode ser dividida em duas partes: a linguagem de consultas em si e a API para executar consultas a partir de meta-programas MetaJ.

5.2 A API SCQL

Os principais elementos da API são as *fábricas de consultas*, representadas pela classe `scql.QueryFactory`, e os objetos que armazenam o resultado das consultas, `scql.ResultSet`. Fábricas de consultas são objetos utilizados pelo meta-programa para criar consultas. Além disso, elas são ambientes para execução de consultas no qual podem ser anexados valores que serão posteriormente acessados por elas. Fábricas de consultas são criadas através do método `createQueryFactory(String nomePlugIn)` da classe `scql.SCQLSystem`. Este método recebe como parâmetro o nome do *plug-in* da linguagem objeto em que o programa objeto que será

consultado foi escrito.

Pela utilização do método `add(String id, Object value)`, valores, que posteriormente serão acessados pelas consultas, podem ser inseridos no ambiente definido pela fábrica de consultas. O primeiro parâmetro deste método `id` é o identificador que consultas deverão utilizar para acessar o valor passado como segundo parâmetro.

Objetos `Query` representam consultas. Eles são criados através do método `Query createQuery(String query)` de alguma fábrica de consultas criada previamente. Como pode ser observado na assinatura do método `createQuery`, uma consulta deve ser expressa como um *string*. A partir do objeto `Query` retornado, utilizando-se o método `ResultSet getResultSet()`, é possível criar `ResultSets`, objetos que armazenam o resultado de uma consulta. Objetos `ResultSet` permitem navegar pelo resultado de uma consulta através da utilização do método `boolean next()`. Veja abaixo um pequeno meta-programa que utiliza todos estes recursos:

Exemplo 5.1 :

```
package scqlTestes;
import scql.*;
public abstract class Main {
    public static void main (String args[]){
        QueryFactory qf = SCQLSystem.createQueryFactory("java");
        PReference r = MetaJSystem.createPReference("java",
                                            "#compilation_unit");
        r.setFile(args[0]);
        qf.add("meuArq", r);
        Query q =  qf.createQuery("VIEW TREE meuArq AS TABLE ...");
        ResultSet rs = q.getResultSet();
        while(rs.next()){
            ... //explorar cada item retornado pela consulta
        }
    }
}
```

No exemplo acima é criado um objeto `QueryFactory` para manipular códigos Java. Em seguida, uma referência-p `r` é criada. É atribuído a ela o conteúdo do arquivo cujo nome foi recebido como parâmetro pelo programa (`args[0]`). A referência-p `r` é acrescentada ao ambiente definido pela fábrica de consultas `qf`. Conseqüentemente, consultas criadas a partir desta fábrica poderão acessar o valor de `r`. Uma consulta é criada e o seu resultado é recuperado através de um objeto `ResultSet` capturado por `rs`. Finalmente, o resultado da consulta é explorado no corpo do **while**.

Como pode ser percebido no Exemplo 5.1 e também na assinatura do método `createQuery`, consultas são expressas como um *string* que deve ser descrito de acordo com as regras sintáticas da linguagem de consultas.

5.3 A linguagem de consultas SCQL

O projeto de SCQL foi inspirado na forma de gerenciamento de dados adotada pelos sistemas de gerenciamento de banco de dados relacionais. Estes sistemas organizam seus dados em tabelas, um conceito equivalente ao conceito matemático de relações. Relações são sub-conjuntos de tuplas resultantes do produto cartesiano de uma lista de domínios [SKS01]. Um exemplo é a relação $Contas = \{(\text{“}Maria\text{”}, 100.0), (\text{“}Jo\tilde{a}o\text{”}, 50.43), (\text{“}Paula\text{”}, 1000.15)\}$. Observe que $Contas$ é um subconjunto do produto cartesiano $(String \times Real)$. Uma tabela é uma relação onde cada elemento das tuplas que a compõem pode ser endereçado por um nome. Para a relação $Contas$, por exemplo, pode-se nomear os primeiros elementos das tuplas com “nome” e os segundos com “saldo”. Seus dados podem, então, ser visualizados da seguinte maneira:

Contas	
Nome	**Saldo**
“Maria”	100.0
“João”	50.43
“Paula”	1000.15

A tabela $Contas$ pode ser utilizada para representar a situação das contas de clientes de um banco. Cada elemento da tabela, i.e., cada conta do banco, é chamado de linha, registro ou mesmo tupla. Cada coluna da tabela é chamada de campo ou atributo. Os sistemas de gerenciamento de banco de dados relacionais permitem criar, atualizar (incluindo inserção e remoção de dados) e expressar consultas sobre tabelas. Para isso eles disponibilizam uma linguagem de consultas, SQL – ***S****tructured* ***Q****uery* ***L****anguage* – na qual são oferecidos os comandos: `CREATE` para criar tabelas, `UPDATE`, `INSERT` e `DELETE`, para atualizar os dados e `SELECT`, para consultar. É possível, por exemplo, pedir para que o sistema encontre todos os registros da tabela $Contas$ cujo atributo *saldo* possua um valor maior ou igual a 90.50. O resultado desta consulta é uma outra tabela:

Resultado da Consulta	
Nome	**Saldo**
“Maria”	100.0
“Paula”	1000.15

A idéia de SCQL é visualizar o código de um programa objeto como uma tabela de forma a permitir que consultas sejam realizadas sobre ela. Para isso, a linguagem disponibiliza dois comandos `VIEW TREE`, para visualizar um programa como uma tabela, e `SELECT`, para combinar duas ou mais tabelas (ou resultados de consultas). Fazendo um paralelo com SQL, o comando `VIEW TREE` de SCQL faz o mesmo papel do `CREATE` de SQL, já que aquele especifica como a tabela deve ser montada a partir do programa. O `SELECT` de SCQL tem exatamente o mesmo papel do comando de mesmo nome em SQL, combinar várias tabelas e realizar seleção de registros baseados em condições (predicados) especificadas pelo usuário. Em ambos os casos, o resultado de um `SELECT` é uma tabela que poderá também ser alvo de outras consultas. A versão atual de SCQL não possui comandos para atualizar os dados de uma tabela (ou programa). As próximas seções explicarão a sintaxe e a semântica de cada comando de SCQL.

5.3.1 VIEW TREE

A operação `VIEW TREE` é utilizada para "visualizar" um programa armazenado em uma referência-p como uma tabela. Este comando realiza a conversão baseando-se nas seguintes informações passadas pelo usuário:

- o programa que será visualizado, o nome de uma referência-p que armazena o trecho de programa a ser consultado. Como já foi citado anteriormente, toda consulta é executada no ambiente definido pela fábrica de consultas que a criou. A referência-p que armazena o programa a ser consultado será procurada neste ambiente;
- operadores de corte, que serão aplicados sobre o programa com o objetivo de escolher as construções sintáticas a serem exploradas (operador `FOCUSED ON`) e excluir outras que deverão ser evitadas (operador `EXCLUDING`). Estes operadores são aplicados no programa objeto antes da consulta ser realizada. Se nenhum operador for especificado, todas as construções sintáticas do programa serão exploradas durante a consulta;
- o esquema da tabela, informação que corresponde a uma seqüência de especificações de campos da tabela que está sendo construída. Para cada campo deve ser especificado seu nome e o tipo de construção sintática que deverá ser armazenada nele. Para cada campo, o usuário poderá especificar um filtro, i.e., uma expressão que descreve a condição que deve ser satisfeita pelos valores a serem inseridos nele;
- condição de seleção de registros, uma expressão booleana que, utilizando os nomes dos campos, realiza verificações sobre o código armazenado por eles.

Assim, uma instância deste comando pode ser vista como uma quadrupla $T = (p, c, e, w)$, onde p é o programa a ser visualizado, c os operadores de corte a serem aplicados, e um esquema que descreve o formato da tabela a ser construída a partir de p e w a condição para seleção de registros a serem inseridos na tabela após sua construção. O esquema e é uma lista de especificações de campos da tabela, i.e., $e = [e_1, e_2, \ldots e_n]$. Cada especificação de campo $e_i = (c_i, t_i, f_i)$ é uma tupla onde o primeiro elemento c_i é o nome do campo especificado, t_i o tipo do campo e f_i é um filtro. A execução de T ocorre da seguinte maneira:

1. os operadores de corte c são aplicados ao programa p a ser visualizado resultando em p_c;
2. é criada uma tabela t de acordo com o esquema e;
3. seja n o número de campos especificados por e, para cada especificação de campo $e_i = (c_i, t_i, f_i)$ é construído um conjunto v_i de construções sintáticas de tipo t_i extraídas de p_c. O filtro f_i é aplicado sobre os valores de v_i para reduzir a quantidade de valores a serem inseridos na tabela. O resultado deste passo é uma lista $vs = [v_1, v_2, \ldots, v_n]$;

4. a tabela t é preenchida com as tuplas (registros) resultantes do produto cartesiano dos conjuntos $v_i \in vs$, $1 < i \leq n$;

5. finalmente, a expressão de seleção de registros w é utilizada para escolher, dentre os registros inseridos em t, aqueles que farão parte do resultado da "visualização".

A regra sintática para descrição deste comando é dada abaixo:

```
view → VIEW TREE tree focus_opt excluding_opt
       AS TABLE LPAREN typed_fields RPAREN
       where_opt;
typed_fields → typed_fields COMMA typed_field | typed_field;
typed_field → MJTYPE IDENTIFIER filter_opt;
filter_opt → FILTERED BY expression | λ;
excluding_opt → EXCLUDING exc_focus_list | λ;
focus_opt → FOCUSED ON exc_focus_list | λ;
where_opt → WHERE expression | λ;
```

Um `VIEW TREE` se inicia pelas palavras reservadas `VIEW` e `TREE` seguidas do nome de uma referência-p do ambiente onde a consulta está sendo executada. Dois possíveis operadores de corte podem ser utilizados sobre o programa que está sendo visualizado: `FOCUSED ON` e `EXCLUDING`. Em seguida, na seção `AS TABLE`, o esquema da tabela deve ser especificado. Neste, o tipo e o nome de cada campo da tabela devem ser especificados. Opcionalmente poderão ser especificados filtros para cada campo. Finalmente, na seção `WHERE`, uma expressão indicando a condição de seleção de valores para a tabela pode ser especificada. Veja abaixo uma consulta de visualização de código, na sua forma mais simples.

Exemplo 5.2:

```
VIEW TREE prog          } programa a ser visualizado
AS TABLE (#identifier id) } esquema da tabela
```

Esta consulta converte o programa `prog` em uma tabela de um único campo, `id`. Neste campo serão colocados todos os identificadores (`#identifier`) encontrados em `prog`.

Aplicada sobre o trecho de programa Java "`x = y + z.max() - 2;`", o resultado será a seguinte tabela:

id:#identifier
x
y
z
max

Especificado na seção `AS TABLE` de uma consulta, o esquema da tabela é uma seqüência definições de campos onde são definidos: identificadores para campos c_i, seus tipos t_i e, opcionalmente, um filtro f_i. f_i é uma expressão Java booleana que define uma condição para que os valores de c_i sejam selecionados. A expressão de um filtro pode se referir ao campo que está sendo definido através do identificador c_i.

A especificação de condições para selecionar entre os registros da tabela é realizada na cláusula `WHERE`. Nesta, qualquer expressão booleana Java pode ser especificada. Qualquer dos campos definidos no esquema da tabela pode ser acessado nesta expressão pela utilização dos identificadores declarados na seção `AS TABLE`.

Nestas expressões, o acesso aos valores dos campos da tabela é feito através de referências-p, i.e., através da interface `PReference`. Identificadores prefixados com "`outer.`" se referem a valores que se encontram no ambiente onde a consulta está sendo executada (ambiente definido pela fábrica de consultas que a criou). Veja abaixo um exemplo que utiliza tais recursos.

Exemplo 5.3 :

Seja `outer.tipo = ''int''`, então a seguinte consulta pode ser definida.

```
VIEW TREE prog                          } programa a ser visualizado
  AS TABLE                                                          }
      (#variable_declaration vd,                                    }
      #type t FILTERED BY t.match(outer.tipo),                      } esquema da tabela
      #variable_declarator_id vid)                                  }
WHERE vd.isComposedBy(t) && vd.isComposedBy(vid) } condição de seleção registros
```

A consulta constrói uma tabela com três campos: o primeiro, `vd`, é uma declaração de variável, o segundo, `t`, uma especificação de tipo filtrada pela expressão booleana especificada em `FILTERED BY`, o terceiro, `vid`, é o identificador declarado em `vd`. Vale destacar a utlização do valor `outer.tipo` especificado no ambiente da consulta. Esta consulta liga cada identificador declarado no programa objeto ao seu tipo. Esta consulta pode ser muito útil já que Java permite declarações simultâneas. Seja o programa Java:

```
int x = 0, k;
int y = 90;
MeuTeste z;
x = y + z.max() - 2 + x;
```

Veja abaixo resultado da aplicação da consulta a este trecho de programa:

vd:#variable_declaration	t:#type	vid:#variable_declarator_id
int x = 0, k;	int	x
int x = 0, k;	int	k
int y = 90;	int	y

É possível também especificar operadores de corte: `FOCUSED ON` e `EXCLUDING`. O primeiro operador define sobre quais tipos de construções sintáticas a consulta será realizada. O segundo especifica quais os tipos de construções devem ser evitadas. O exemplo abaixo utiliza os operadores de corte para evitar que certos trechos de programa sejam explorados.

Exemplo 5.4:

```
VIEW TREE prog                          } programa a ser visualizado
       FOCUSED ON #local_variable_declaration  }
       EXCLUDING #block                        } operadores de corte
  AS TABLE                                     }
      (#variable_declaration vd,               }
      #type t FILTERED BY t.match("int"),      } esquema da tabela
      #variable_declarator_id vid)             }
  WHERE vd.isComposedBy(t) && vd.isComposedBy(vid)  } condição de seleção registros
```

Esta consulta é realizada apenas sobre declarações de variáveis locais `#local_variable_declaration` e também evita que blocos aninhados (`#block`) sejam visitados.

5.3.2 SELECT

A operação `SELECT` é utilizada para selecionar valores a partir de tabelas ou combinações de tabelas. Um `SELECT` faz a seleção baseado nas informações passadas pelo usuário, são elas:

- a especificação dos campos das tabelas que serão selecionados (π);
- a lista de tabelas a serem combinadas (ts);
- uma condição de seleção de registros (ρ).

O comportamento do `SELECT` de SCQL é muito parecido com o de SQL, i.e., ele realiza o produto cartesiano das tabelas ts, aplica a condição de seleção de registros ρ para eliminar valores que não estarão no resultado da consulta e, finalmente, seleciona apenas os campos escolhidos em π.

O comando `SELECT` é mais simples que o `VIEW TREE`. Sua sintaxe é apresentada abaixo:

```
selection → SELECT fields FROM LPAREN resultset_list RPAREN where_opt
          | SELECT * FROM LPAREN resultset_list RPAREN where_opt;
fields → fields COMMA field  | field;
field → IDENTIFIER AS IDENTIFIER | IDENTIFIER;
resultset_list → resultset_list COMMA resultset | resultset;
resultset → selection | view;
```

Uma seleção inicia-se pela palavra reservada `SELECT` seguida, imediatamente pela lista de campos selecionados π, ou "*" para selecionar todos os campos. Em seguida, na seção `FROM` deve ser especificada uma lista de tabelas a serem combinadas. Finalmente, a condição de seleção de registros é descrita na cláusula `WHERE`, assim como foi feito para o comando `VIEW TREE`. A expressão definida nesta cláusula pode acessar os valores de todos os campos selecionados em π. De maneira equivalente ao comando `VIEW TREE` estes valores são manipulados pela utilização da interface `PReference`. Veja abaixo um exemplo de consulta.

Exemplo 5.5 :

```
SELECT (td, id, lvd, t) FROM (                  } seleção de campos
                VIEW TREE p1 AS TABLE                          }
                    (#local_variable_declaration lvd,          } Consulta 1
                                                #type t)       }
                WHERE lvd.isComposedBy(t),                     }
                VIEW TREE p2 AS TABLE                          }
                    (#type_declaration td,                     } Consulta 2
                                        #identifier id)        }
                WHERE outer.verifier.isTypeName(td, id))       }
WHERE t.match(id)                          } condição para seleção de registros
```

A consulta cruza as declarações de variáveis locais `lvd` que ocorrem no programa `p1` com as declarações de tipo `td` do programa `p2`. Vale destacar o acesso ao método `isTypeName (PReference td, PReference id)` do objeto `outer.verifier` colocado no ambiente onde a consulta é executada. Este objeto é uma instância da classe definida abaixo:

```
public class Verificador{
    public boolean isTypeName(PReference tdecl,  PReference ident) { ... }
}
```

O objetivo do método `isTypeName` é verificar se o identificador `id` é aquele declarado como nome do tipo `td`. Este método pode ser facilmente realizado utilizando-se *templates* e referências-p.

Opcionalmente, o nome de um campo escolhido pode ser redefinido assim como mostra o exemplo abaixo.

Exemplo 5.6 :

```
SELECT (td, id AS idTipo, lvd, t AS tipo) FROM (
            ... // Assim como foi feito no Exemplo 5.5 )
WHERE tipo.match(idTipo)
```

Nesta consulta, o nome dos campos `id` e `t` foram renomeados para `idTipo` e `tipo`, respectivamente. Agora, estes nomes deverão ser utilizados na cláusula `WHERE` do `SELECT`, ao invés de `id` e `t`.

Caso haja conflitos entre os nomes de campos definidos pelas consultas que estão sendo combinadas é possível prefixá-los com `$n$`, onde n é o número de ordem da consulta onde tal campo foi definido. As consultas são numeradas na ordem em que elas aparecem, i.e., a primeira recebe o número 1, a segunda o 2, e assim por diante. Veja o exemplo abaixo.

Exemplo 5.7 :

```
SELECT (td, $2$t AS idTipo, lvd, $1$t) FROM (
                VIEW TREE p1 AS TABLE                          } Consulta 1
                    (#local_variable_declaration lvd,          }
                                          #type t)             }
                WHERE lvd.isComposedBy(t),                     }
                VIEW TREE p2 AS TABLE                          } Consulta 2
                    (#type_declaration td,                     }
                                     #identifier t)            }
                WHERE outer.verifier.isTypeName(td, t))        }
               )
WHERE $1$t.match(idTipo)
```

Neste exemplo, ocorre conflito entre os nomes de campos `t` especificados pelas consultas 1 e 2. Prefixando-se o identificador `t` com 1 ou 2 é possível acessar, sem ambigüidade, os valores desejados. Esta consulta é a mesma apresentada no Exemplo 5.5, exceto pela diferença de nome entre os campos da consulta 2.

5.4 Detalhes de implementação

A Figura 5.1 mostra o relacionamento entre os principais componentes de SCQL. As fábricas

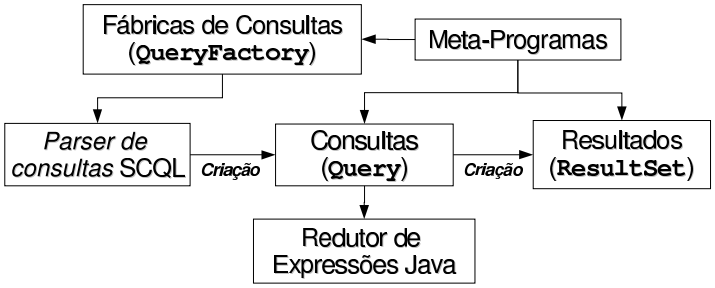


Figura 5.1: Arquitetura geral de SCQL

de consultas (`QueryFactory`) criam, utilizando-se do *parser* SCQL, consultas que são retornadas

para o meta-programa que, a partir dela, poderá criar `ResultSet`s. No processo de construção do resultado da consulta, objetos `Query` utilizam o redutor de expressões Java para avaliar as expressões especificadas nas cláusulas `WHERE` ou em filtros (nas cláusulas `FILTERED BY`).

Nas próximas seções serão apresentados os detalhes sobre a implementação de SCQL. Primeiramente serão apresentadas as classes `ResultSet`, `Query` e `QueryFactory`. Em seguida, os detalhes sobre o parser da linguagem de consultas serão apresentados. Finalmente, será apresentada a implementação de uma das principais classes do pacote `scql`, a `CartesianProduct`, juntamente com detalhes sobre o processo de interpretação das expressões Java que aparecem em cláusulas `WHERE` ou `FILTERED BY`.

5.4.1 `ResultSet`s, `Query`s e `QueryFactory`s

Como já foi citado na Seção 5.1, um objeto `QueryFactory` é uma fábrica de consultas a partir da qual é possível criar objetos `Query`, que representam consultas. Estes objeto retornam como resultado da sua execução `ResultSet`'s. Todas estas classes foram definidas no pacote `scql`. A Figura 5.2 mostra o relacionamento entre estas e outras classes deste mesmo pacote.

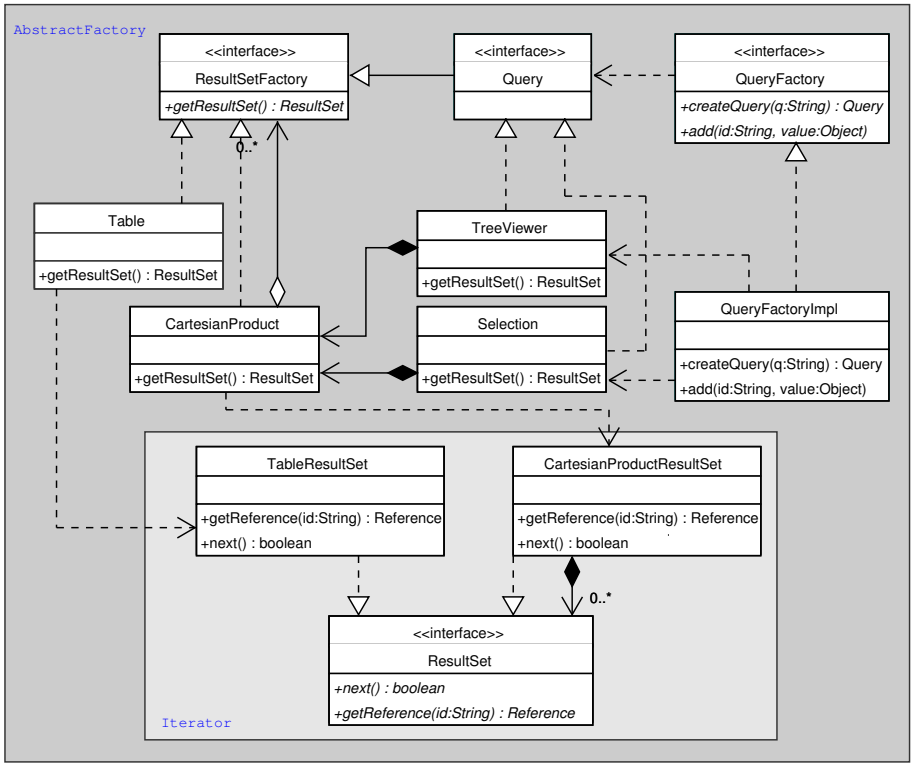


Figura 5.2: `ResultSet`s e `ResultSetFactory`s

As classes `TreeViewer` e `Selection` implementam respectivamente as operações `VIEW TREE` e `SELECT` disponibilizadas por SCQL. Elas são implementações da interface `Query`. Estas classes

combinam objetos `CartesianProduct`s e `Table`s para definir o comportamento das operações representadas por elas:

- `TreeViewer`: a partir do valor de uma referência-p (`PReference`), esta classe constrói uma tabela (`Table`) para cada tipo de construção sintática escolhida na cláusula `AS TABLE` do comando `VIEW TREE`. Cada uma destas tabelas é preenchida com as ocorrências da respectiva estrutura sintática. Finalmente, é criado um objeto `CartesianProduct` para realizar a combinação (produto cartesiano) destas tabelas.
- `Selection`: constrói um objeto `CartesianProduct` para combinar os vários `ResultSetFactory`s especificados na cláusula `FROM` da operação `SELECT` de uma consulta SCQL.

As classes `CartesianProduct`, `Table`, `TreeViewer` e `Selection` foram projetadas segundo o padrão de projeto *AbstractFactory* [GHJV94]. A intenção deste padrão é:

> *Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.*

Estas classes fazem o papel de *ConcreteFactories*. A interface `ResultSetFactory` é uma *AbstractFactory*, cujo objetivo é definir as operações que deverão ser implementadas pelas fábricas concretas. A interface *ResultSet* faz o papel de *AbstractProduct*, define as operações que serão fornecidas pelos tipos de objetos produzidos pelas fábricas. `CartesianProductResultSet` e `TableResultSet` são os *ConcreteProducts*. Objetos destas duas últimas classes são produzidos por `CartesianProduct` e `Table`. As classes `TreeViewer` e `Selection` são, basicamente, combinações das classes `CartesianProduct` e `Table`. Os objetos produzidos por aquelas são apenas combinações dos objetos produzidos por estas.

O padrão de projeto *Iterator* foi utilizado nas classes `ResultSet`, `CartesianProduct-ResultSet` e `TableResultSet` que encapsulam o resultado de uma consulta. A classe `ResultSet` faz o papel de *Iterator*. `CartesianProductResultSet` e `TableResultSet` são *ConcreteIterators*. Estas permitem explorar, de maneira iterativa, os resultados de uma consulta. As vantagens da utilização do padrão *Iterator* são que iteradores (`ResultSet`s) escondem a representação interna do resultado da consulta e também permitem que "caminhamentos" simultâneos e independentes possam ser realizados sobre uma mesma consulta. Isso pode ser realizado pela criação de diferentes `ResultSet`s a partir de uma mesma consulta (`Query`).

5.4.2 O *parser* de consultas e os nodos da árvore sintática de expressões

O *parser* para a linguagem de consultas foi construído utilizando-se a ferramenta de geração de parser Cup [Hud99]. Na especificação da gramática de *scql* foram acrescentadas rotinas semânticas para a instanciação de objetos `TreeView` e `Selection`, que encapsulam o processo de consulta. Foram acrescentadas também rotinas para a montagem da árvore sintática da

expressão definida na cláusula `WHERE` e em filtros (`FILTERED BY`). Esta árvore é passada para os objetos `TreeView` ou `Selection` criados pelo *parser*.

Os nodos utilizados para construção da árvore sintática foram implementados no pacote `scql.expressionReduction.tree`, seguindo os padrões de projeto *Composite* e *Visitor* [GHJV94]. Veja o esquema da Figura 5.3.

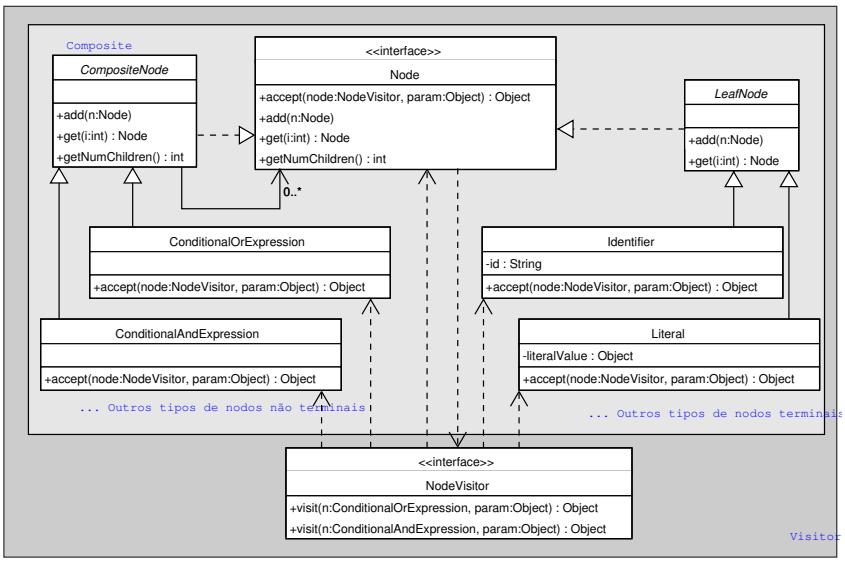


Figura 5.3: Principais classes utilizadas na construção da árvore sintática de expressões

A interface `Node` declara os métodos `get`, `add` e `getNumChildren`. Eles são utilizados para recuperar, adicionar e recuperar o número de filhos de um nodo, respectivamente. As classes abstratas `CompositeNode` e `LeafNode` fazem implementações padrão destes métodos para nodos que possuam filhos e para nodos que não possuam filhos, respectivamente. No esquema da Figura 5.3 foram apresentados apenas os nodos `ConditionalOrExpression`, `ConditionalAndExpression` `Identifier`, e `Literal`, mas existem vários outros, um para cada símbolo não-terminal da gramática de expressões. No padrão de projeto *Composite* [GHJV94], utilizado no projeto destas classses, `Node` faz papel de *Component*. As subclasses de `LeafNode` e as de `CompositeNode` são *Leaf*s e *Composite*s, respectivamente.

Um nodo que também é implementado no pacote `scql.expressionReduction.tree` é o `ReducedValue`. Este não é utilizado pelo *parser* para construir a árvore sintática da expressão, mas é utilizado pelo redutor de expressões Java para marcar um trecho já avaliado.

A interface `NodeVisitor` e os métodos `accept`, definidos em cada um dos nodos, fazem parte da implementação do padrão de projeto *Visitor* [GHJV94]. Esta interface facilitou a implementação do redutor de expressões Java.

5.4.3 A implementação de `CartesianProduct`

A operação `SELECT` recebe como parâmetro uma lista de tabelas cujos valores armazenados devem ser combinados. Esta combinação é realizada pela aplicação da operação produto cartesiano. De maneira equivalente, o comando `VIEW TREE` combina os conjuntos de estruturas sintáticas coletadas no código do programa consultado através desta mesma operação. Depois de aplicar esta operação, estes comandos verificam quais valores satisfazem a condição descrita na cláusula `WHERE`.

É apresentado abaixo um algoritmo que calcula o produto cartesiano $P' = A_i \times B_j$ de dois conjuntos indexado indexados A_i e B_j ($i, j \in \mathbb{N}$, $|A_i| = m$, $|B_i| = n$) e seleciona, dentre os valores calculados, aqueles que satisfazem uma condição w, resultando em um conjunto $P \subseteq A_i \times B_j$.

```
Faça P = ∅
Faça P' = ∅
Para cada 0 ≤ k < m, onde k ∈ ℕ, faça
    Para cada 0 ≤ z < n, onde z ∈ ℕ, faça
        Adicione em P' a tupla (A(k), B(z))
Para cada elemento p ∈ P'
    se p satisfaz w, então insira p em P
```

O resultado desta operação é um conjunto de tuplas (a, b), onde $a \in A_i$ e $b \in B_j$. Este algoritmo pode ser generalizado para realizar o produto cartesiano de uma quantidade qualquer de conjuntos indexados.

Uma crítica a este algoritmo é que o cálculo do produto cartesiano pode resultar em uma quantidade muito grande de valores a serem avaliados pela expressão w, o que pode implicar em uma perda de eficiência da linguagem de consultas. Por causa disso, são necessárias otimizações neste processo. Uma possibilidade é implementação do processo de geração de tuplas juntamente com a avaliação da cláusula `WHERE`. Veja abaixo como isso pode ser feito tomando os conjuntos A_i e B_j e uma condição w (observe que este processo pode ser generalizado para um produto cartesiano de qualquer número de conjuntos):

```
Faça P = ∅
Para cada 0 ≤ k < m, onde k ∈ ℕ, faça
    Se A(k) não reduz a condição w para false então
        Para cada 0 ≤ z < n, onde z ∈ ℕ, faça
            Se (A(k), B(z)) reduz a condição w para true então
                Adicione em P a tupla (A(k), B(z))
```

O algoritmo acima tenta reduzir a expressão w utilizando apenas o primeiro elemento da tupla. Se esta redução resultar em `false`, o próximo elemento de A_i é testado. Neste caso, $|B_j| = n$ valores do conjunto resultante P são descartados. Caso contrário, a expressão será avaliada novamente quando um valor de B_j for escolhido.

Para otimizar ainda mais este processo, pode-se, quando w não for completamente reduzida para `false`, aproveitar o resultado de sua redução parcial para avaliações posteriores, evitando assim que uma sub-expressão seja avaliada várias vezes desnecessariamente. Abaixo é apresentado o algoritmo para geração das tuplas do conjunto $P = A_i \times B_i$ tomando w como uma expressão de seleção.

```
Faça P = ∅
Para cada 0 ≤ k < m, onde k ∈ ℕ, faça
        // reaproveitando o resultado da avaliação de w
        w' = expressão reduzida resultante da avaliação de w com A(k)
        Se w' != false então
            Para cada 0 ≤ z < n, onde z ∈ ℕ, faça
               Se A(k) e B(z)) reduzem a condição w' para true então
                   Adicione em P a tupla (A(k), B(z))
```

Vale destacar o aproveitamento, através de w', da expressão w parcialmente avaliada. Para implementar tal otimização é necessário um avaliador parcial de expressões: um componente capaz de reduzir apenas os termos da expressão que possuem todas as suas variáveis ligadas. Este avaliador será apresentado na próxima seção, *"A avaliação de expressões Java"*.

O algoritmo acima, generalizado para uma quantidade qualquer de conjuntos indexados, é encapsulado pelas classes `CartesianProduct` e `CartesianProductResultSet`. Um `CartesianProduct` encapsula um vetor de `ResultSetFactory`s, que são os "conjuntos" que serão alvo da operação produto cartesiano, e uma condição de seleção (a expressão da cláusula `WHERE`). Apesar de armazenar todas estas informações, o processo de cálculo do produto cartesiano é na realidade implementado pela classe `CartesianProductResultSet`.

Seja *cp* um objeto da classe `CartesianProduct`, ao se chamar o método `cp.getResultSet()`, uma instância *cprs* de `CartesianProductResultSet` é criada. Quando o método `next` de *cprs* é invocado, o primeiro valor da consulta é calculado. Se chamado novamente, o segundo valor é calculado, e assim sucessivamente. Este objeto gera resultados sob demanda, evitando uma perda de tempo calculando resultados que poderão nem mesmo serem explorados .

Uma crítica a esta decisão de projeto é que cada instância de `CartesianProductResultSet` calcula, independentemente, o resultado da consulta, i.e., sem se importar se alguma outra instância já o calculou. Apesar da implementação deste recurso ser simples, ela não foi realizada. Esta decisão de projeto é justificada pela imensa quantidade de tuplas geradas, mesmo em consultas bastante simples.

A avaliação de expressões Java

Abaixo está apresentada uma gramática de expressões Java simplificada[1] que será utilizada para demostrar o funcionamento do processo de avaliação parcial de expressões:

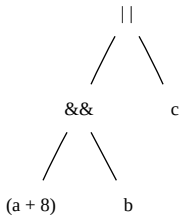
[1] A gramática que define a sintaxe das expressões pode ser encontrada no Apêndice E

```
expression → conditional_or_expression;

conditional_or_expression → conditional_and_expression
        | conditional_or_expression "||" conditional_and_expression;

conditional_and_expression → java_term_or_aritmetic_expression
        | conditional_and_expression "&&" java_term_or_aritmetic_expression;

java_term_or_aritmetic_expression → ...;
```

A partir desta gramática, o parser SCQL constrói a árvore sintática da expressão a ser reduzida. O redutor de expressões age sobre esta árvore. O exemplo abaixo lista algumas expressões e suas árvores sintáticas.

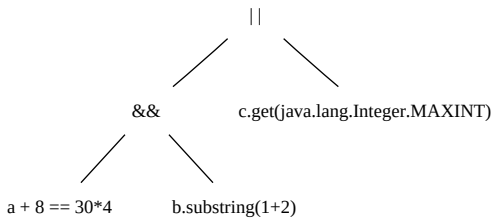
Exemplo 5.8 :

1. `a + 8 && b || c`



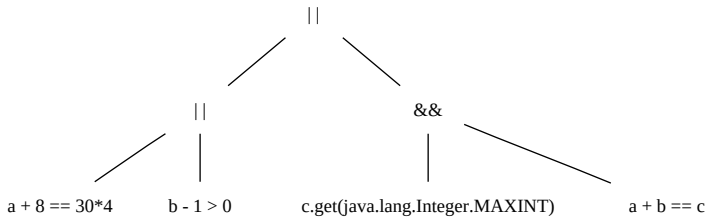
2. [2]

   `a + 8 == 30*4 && b.substring(1+2) || c.get(java.lang.Integer.MAX_VALUE)`



3. `a + 8 == 30*4 || b - 1 > 0 || c.get(java.lang.Integer.MAX_VALUE)&& a + b==c`

[2] Por motivo de simplificação, as sub-expressões 1+2 e `java.lang.Integer.MAX_VALUE` não foram representadas como árvores sintáticas, mas, na implementação real de SCQL, todo termo é representado como uma árvore sintática.

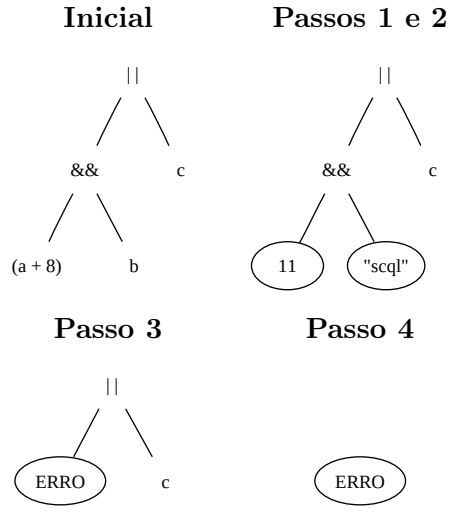


O processo de avaliação de uma expressão é realizado caminhando-se em pós-ordem pelos nodos da árvore e substituindo-se cada sub-árvore que não possua variáveis livres por um nodo especial (`ReducedValue`) que representa o resultado da avaliação daquela sub-expressão. Abaixo são mostrados os passos da avaliação das expressões do Exemplo 5.8:

Exemplo 5.9 :

Avaliação da Expressão 1: Contexto de avaliação: `a = 3`, `b = ‘‘scql’’` e `c` é uma variável livre.

Passos de avaliação:
Inicial) ⇒ `a + 8 && b || c`
1 e 2) ⇒ `11 && ‘‘scql’’ || c`
3) ⇒ `ERRO || c`
4) ⇒ `ERRO`



Nos dois primeiros passos da avaliação, a soma `a + 8` é reduzida para `11` e o acesso à variável `b` é reduzido para `‘‘scql’’`. No terceiro, a conjunção `11 && ‘‘scql’’` gera um erro pois o operador && não pode ser aplicado sobre valores não booleanos. Com isso, o resultado da redução desta expressão é um erro. Observe que, apesar de ser livre, a variável `c` não influenciou no processo de avaliação já que ela não foi necessária.

Avaliação da Expressão 2: Contexto de avaliação: `a = 10`. `b` e `c` são livres.

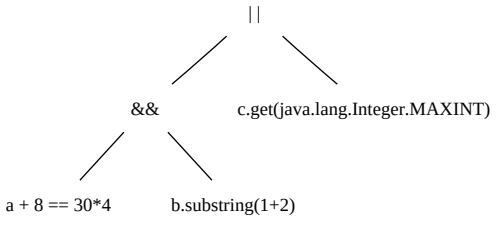
Passos de avaliação:

Inicial) `a + 8 == 30*4 && b.substring(1+2) || c.get(java.lang.Integer.MAX_VALUE)`

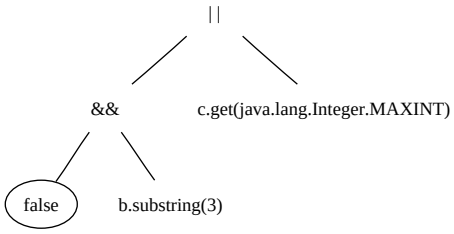
1 e 2) `false  && b.substring(3)|| c.get(java.lang.Integer.MAX_VALUE)`

3 e 4) `false || c.get(2147483647)`

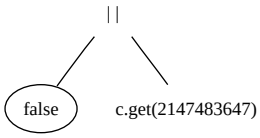
Inicial



Passos 1 e 2



Passos 3 e 4



Diferentemente da expressão apresentada no item 1, esta expressão não é reduzida completamente. Nos dois primeiros passos as expressões `a + 8 == 30*4` e `1+2` são reduzidas. No terceiro, a conjunção é reduzida para `false` já que `a + 8 == 30*4` reduziu para `false`. Observe que `b` não influenciou nesta redução. No quarto `java.lang.Integer.MAX_VALUE` é reduzido, mas o processo não prossegue pois `c` é uma variável livre.

Avaliação da Expressão 3: Contexto de avaliação: `a = 24` e `b = -10`. `c` é uma variável livre.

Passos de avaliação:

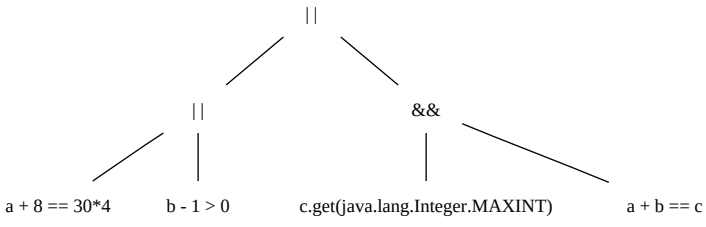
Inicial) `a + 8 == 30*4 || b - 1 > 0 || c.get(java.lang.Integer.MAX_VALUE)&& a+b==c`

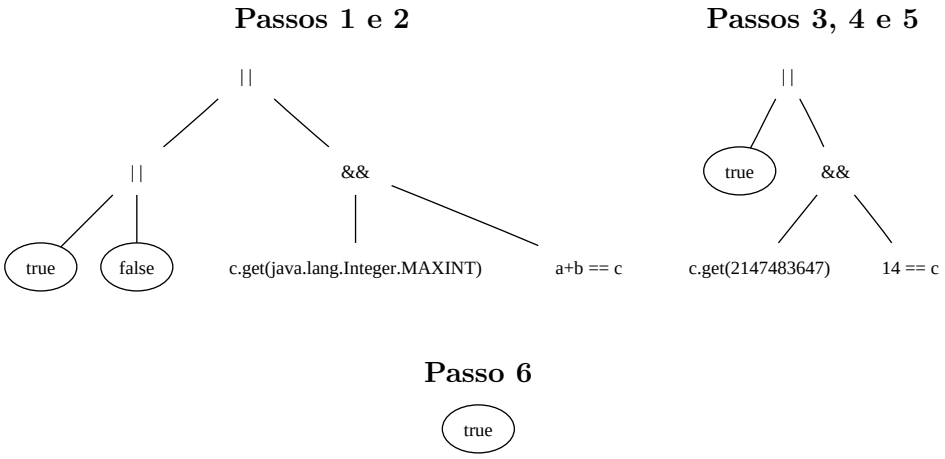
1) `true  || false || c.get(2147483647) && 14 == c`

2) `true || c.get(2147483647) && 14 == c`

3) `true`

Inicial





Nos dois primeiros passos da avaliação da expressão, `a + 8 == 30*4` e `b - 1 > 0` são reduzidos para `true` e `false`, respectivamente. No terceiro passo a disjunção mais à esquerda é reduzida para `true`. No quarto e quinto passos `java.lang.Integer.MAX_VALUE` e `a + b` são reduzidos. Finalmente toda a expressão é reduzida para `true`. Observe que, apesar de existirem termos com variáveis livres, eles não influenciaram na avaliação e a expressão foi completamente reduzida.

Seguindo o padrão de projeto *Visitor* [GHJV94], a classe `ExpressionReducer` realiza a interface `NodeVisitor` e implementa, em seus métodos, o processo de redução de expressões apresentado. Neste processo, ao invés de reduzir a árvore original substituindo as sub-árvores avaliadas, o redutor caminha pela árvore gerando uma nova árvore reduzida em que as sub-árvores reduzidas foram substituídas por nodos `ReducedValue`. Desta maneira, é possível avaliar uma expressão sem perder a expressão original. As classes relacionadas com o processo de redução de expressões podem ser encontradas no pacote `scql.expressionReduction`.

5.5 A refatoração *change_ variable_ name* para variáveis locais

Nesta seção será descrita a implementação da refatoração *change_ variable_ name*, apresentada anteriormente na Seção 3.7, utilizando-se os recursos de SCQL. Ela foi implementada no método `run` da classe `scql.examples.refactorings.renameLocalVariable.-RenameLocalVariable`.

A utilização da linguagem de consulta não modificou a mecânica da refatoração, mas alterou bastante a forma como cada passo é realizado. Abaixo mostramos o código da classe `RenameLocalVariable`.

```
...
public class RenameLocalVariable {
    QueryFactory qf = SCQLSystem.createQueryFactory("java");
    ...
```

```
    /* outros métodos auxiliares */
    private void run(String a, String c, String am,
                                    String n, String s)throws Exception{
        qf.add("varName", n);
        qf.add("varNewName", s);
        qf.add("verifier", new Verifier ());
        ChooseMethodInClass procMet = new ChooseMethodInClass();
        procMet.getClassId().set(c);
        procMet.getMh().set(am);
        if(procMet.matchFile(a)){
            PReference mh = procMet.getMh(), bs = procMet.getBs();
            qf.add("context", bs);
            // Verificar a existência de uma variável com nome s
            if(verificarDeclaracaoNovoNome()){
                System.out.println("Erro!...");
                return;
            }else{
                // Adicionando this aos acessos a campos
                adicionarThisAosAcessosCampos();
            }
            // Atualizando utilizações da variável local
            atualizarNomesVariavelLocal(s);
            // Reescrevendo o arquivo de saída
            procMet.toFile(a + ".refactoring.result");
            return;
        }
        System.out.println("Classe não encontrada!");
    }
}
```

Obseve que a estrutura do método `run` é equivalente à daquele de mesmo nome apresentado na Seção 3.7, exceto pelo fato de que os parâmetros `n` e `s` são inseridos no ambiente definido pela fábrica de consultas `qf`. A utilização de SCQL influenciou bastante a implementação dos métodos auxiliares. Abaixo é apresentado o comportamento de cada um deles:

- `verificarDeclaracaoNovoNome`: o objetivo deste método é verificar se existe alguma declaração de variável de nome `s`. Veja abaixo o corpo deste método:

```
String findVarDecl = "VIEW TREE context AS TABLE " +
                     " (#variable_declarator_id vid " +
                        " FILTERED BY vid.match(outer.varNewName))";

private boolean verificarDeclaracaoNovoNome()throws Exception{
    return qf.createQuery(findVarDecl).getResultSet().next();
```

```
}
```

Este método simplesmente executa a consulta armazenada na variável `findVarDecl` e verifica se o `ResultSet` retornado possui ao menos um elemento. A consulta recolhe todas as declarações de variáveis que ocorrem em `context`. Como foi definido no método `run`, `context` é o corpo do método onde a variável que se deseja renomear foi declarada.

- `adicionarThisAosAcessosCampo`: o objetivo deste método é adicionar "`this`" aos acessos ao campo que possui o nome `s`;

```
String findFieldUsage = "VIEW TREE context AS TABLE "+
                        " (#postfix_expression pe FILTERED BY "
                          " pe.toString().startsWith(outer.varNewName+"."))"
                          " || pe.toString().equals(outer.varNewName)";

private void adicionarThisAosAcessosCampos (String fn)throws Exception{
    ResultSet rs = qf.createQuery(findFieldUsage).getResultSet();
    while(rs.next()){
        Reference r = rs.getReference("pe");
        r.replace("this"+ fn+ r.toString().substring(fn.length()));
    }
}
```

Através da consulta armazenada em `findFieldUsage`, o método encontra todos os acessos à variável cujo nome está armazenado em `outer.varNewName`. Para finalizar, o método percorre o resultado da consulta substituindo o prefixo de cada elemento.

- `atualizarNomesVariavelLocal`: este método é um pouco mais complexo que os anteriormente apresentados, veja seu corpo abaixo:

```
private void atualizarNomesVariavelLocal (String n, String s)throws Exception{
        Query q = qf.createQuery(findVarDeclarationAndOccurrences);
        ResultSet rs = q.getResultSet();
        Vector vdecls = new Vector();
        while(rs.next()){
           Reference decl = rs.getReference("vid");
           if(!vdecls.contains(decl)) vdecls.add(decl);
           Reference pe = rs.getReference("pe");
           decl.replace(s);
           r.replace(s+ r.toString().substring(n.length()));
        }
}
```

Ele executa a consulta armazenada em `findVarDeclarationAndOccurrences` que serve para encontrar todas as declarações variáveis e suas ocorrências. Veja esta consulta abaixo:

```
VIEW TREE context
AS TABLE
    (#variable_declarator_id vid
          FILTERED BY vid.match(outer.varName),
     #block_statement blcks,
     #postfix_expression pe
          FILTERED BY pe.toString().equals(outer.varNewName) ||
                      pe.toString().startsWith(outer.varNewName+".")),
     #block_statement blcks2 , #block_statements bs)
WHERE blcks.isComposedBy(vid) && blcks2.isComposedBy(pe) &&
      outer.verifier.after(blcks2, blcks, bs) &&
      outer.verifier.depthLeqThan(bs, vid, "java.#block", 1);
```

A consulta constrói tuplas (`vid`, `blcks`, `pe`, `blcks2`, `bs`) onde:

- `vid` é uma declaração de variável cujo nome é definido pela variável externa `outer.varName`. A cláusula `FILTERED BY` define qual o nome da variável;
- `blcks` o *statement* (`#block_statement`) onde `vid` ocorre. A sub-expressão `blcks.isComposedBy(vid)` da expressão definida na cláusula `WHERE` define tal relação entre `vid` e `blcks`;
- `pe` é uma termo de expressão (`#postfix_expression`) cujo prefixo é definido pela variável externa `outer.varName`. A cláusula `FILTERED BY` define qual deve ser este prefixo;
- `blcks2` é o *statement* onde `pe` ocorre. A sub-expressão `blcks2.isComposedBy(pe)` da expressão definida na cláusula `WHERE` define tal relação entre `vid` e `blcks`;
- `bs` a lista de *statements* onde `blcks` e `blcks2` ocorrem.

Os dois últimos termos da expressão definida na cláusula `WHERE` utilizam as funções `after` e `depth` do objeto externo `outer.verifier` para verificar a relação entre `blcks2` e `blcks`. O objeto `verifier` é uma instância da classe `Verifier`. Veja abaixo a definição das funções utilizadas na consulta:

- `after (Reference a, Reference b, Reference contexto)`: percorre os valores armazenados na lista de estruturas sintáticas `contexto` verificando se `a` ocorre após a ocorrência de `b`.

```
public boolean after (Reference a, Reference b, Reference context){
        Enumeration enum = context.values();
```

```
            while(enum.hasMoreElements() &&
                !((Reference) enum.nextElement()).equals(b));
            while(enum.hasMoreElements()){
                Reference r = (Reference) enum.nextElement();
                if(r.equals(a)) return true;
            }
            return false;
    }
```

O método percorre os valores da lista em busca do trecho de programa armazenado por `b`. Se este for encontrado, então ele continuará em busca de `a`. Se alguma das duas buscas falhar, o método retornará `false`

– `depthLeqThan(Reference context, Reference v, String t, int d)`: este método verifica se a profundidade do trecho de programa `v` dentro de `context` é menor ou igual a `d`. A profundidade é medida em relação ao tipo sintático cujo nome foi recebido pelo parâmetro `t`.

```
public boolean depthLeqThan (Reference context, Reference v, String t, int d){
    int dpt = depth(context, v, t);
    return dpt > 0 && dpt <= d;
}
public int depth(Reference context, Reference value, String type){
    int d = -1;
    Iterator it = context.getIterator();
    it.nextIn();
    while(it.hasNextIn()){
       it.nextIn();
       Reference r = it.get();
       if(r.equals(value)) return -1*d;
       if(r.getType().equals(type)){
           int dtmp = depth (r, value, type);
           if(dtmp > 0) return dtmp+1;
           if(it.hasNextOver()) it.nextOver();
           else return d;
       }
    }
    return d;
}
```

O método `depthLeqThan` faz uma chamada ao `depth` para calcular a profundidade de `v` em `context` levando em consideração o tipo `t`. Caso a profundidade seja maior que 0 e menor ou igual a `d`, o método retornará verdadeiro. O método `depth` retorna um valor negativo caso `v` não seja encontrado em `context`.

Utilizando-se de iteradores, o `depth(Reference context, Reference value, String type)` percorre o código de `context` em busca de `value`. Cada construção sintática do tipo `type` é explorada independentemente através de uma chamada recursiva a `depth`. Se o resultado desta chamada for menor que 0, então ignora-se a estrutura explorada e continua-se o caminhamento. Caso contrário, é retornado o resultado da chamada recursiva, incrementado de uma unidade.

5.5.1 Conclusões

Esta implementação da refatoração *change_ variable_ name* utiliza, além de SCQL, os recursos básicos de MetaJ. Por isso, todos os benefícios trazidos por MetaJ para a implementação apresentada na Seção 3.7 também são válidos para a desta seção.

Além destes, a utilização de consultas SCQL facilitou bastante a localização de trechos de código, veja que consultas simplificaram ainda mais a implementação dos métodos `adicionarThisAosAcessosCampo` e `adicionarThisAosAcessosCampo`. A possibilidade de se anexar objetos para realizar verificações que serão invocadas na cláusula `WHERE` e em filtros aumenta consideravelmente o poder das consultas. Observe que, com a utilização destes, o relacionamento entre declarações e usos de variáveis pôde ser expresso por uma só consulta.

Por outro lado, a utilização de SCQL pode aumentar a quantidade de passagens pelo código para se realizar uma determinada verificação. Além disso, o custo da operação produto cartesiano pode ser muito alto quando o código a ser analisado for muito grande. Por isso, é recomendado que o programador equilibre a utilização de iteradores e consultas SCQL.

5.6 Comentários finais

Este capítulo apresentou SCQL, uma extensão de MetaJ para permitir consultas declarativas sobre programas objeto. Inspirada na forma em que bancos de dados relacionais tratam seus dados, SCQL permite visualizar o código objeto como tabelas sobre as quais seleções podem ser executadas.

SCQL é composta por uma API para execução de consultas dentro de meta-programas MetaJ e da linguagem de especificação de consultas.

A linguagem de consultas permite a especificação de dois tipos de comandos: `VIEW TREE` e `SELECT`. O primeiro é utilizado para converter códigos objeto para o formato tabular. Neste comando deve ser especificado o formato da tabela a ser construída e também expressões para determinar quais trechos de código deverão preenchê-la. Sobre o programa que será convertido para o formato tabular podem também ser aplicadas operações de corte. Estas permitem reduzir o escopo da consulta tornando-a mais eficiente e precisa. O comando `SELECT` é utilizado para combinar resultados de consultas. Para este comando devem ser especificadas tabelas (ou resultados de consultas anteriores) para serem combinadas.

Por ser construída a partir de MetaJ, conceitos como referências-p e tipagem de códigos objeto são

também válidos em SCQL.

Os comandos da linguagem permitem especificar seleções de trechos de código objeto em alto nível, de maneira mais declarativa que aquela proposta por iteradores.

Além de trazer benefícios para a meta-programação, ela também mostra a possibilidade de se definir ambientes de meta-programação a partir dos conceitos definidos por MetaJ.

Por outro lado, a utilização de consultas pode implicar em uma maior quantidade de passagens pelo código a ser reestruturado, o que pode resultar em uma perda de eficiência do meta-programa. É de responsabilidade do meta-programador equilibrar a utilização de SCQL e os recursos de nível mais baixo definidos por MetaJ.

Capítulo 6

A Implementação da Refatoração *Self Encapsulate Field*

6.1 Introdução: refatorações

Como pode ser percebido na refatoração apresentada na Seção 3.7, implementar tal tipo de reestruturação de programas não é uma tarefa muito simples. Opdyke destaca que:

> *A aplicação de refatorações é uma tarefa complexa. Muito desta complexidade é atribuída aos relacionamentos entre as classes. [Opd92]*

Em sistemas desenvolvidos em Java, estes relacionamentos podem ocorrer em vários níveis do código:

Nível 1 (*Relacionamento em nível de Classe ou Interface*): Neste nível, os relacionamentos ocorrem como resultado de: (i) declarações de campos e métodos, (ii) declarações de parâmetros formais, (iii) declaração de exceções lançadas, (iv) declaração de super classe e (v) declaração de interfaces implementadas;

Nível 2 (*Relacionamento em nível de bloco*): Neste nível, os relacionamentos ocorrem como resultado de: (i) declarações de variáveis locais, (ii) declarações variáveis locais a uma repetição `for` e (iii) declarações de exceções capturadas;

Nível 3 (*Relacionamento em nível de expressão*): Neste nível, os relacionamentos ocorrem como resultado de: (i) acesso a variáveis locais, (ii) acesso a campos de objetos e de classes[1] e (iii) chamada de métodos de objetos e de classes[1].

Na aplicação de refatorações é muito comum a necessidade de observação de todos estes níveis. Veja o exemplo do *change_ variable_ name* para variáveis membros de classes[2] proposto

[1] Incluindo a classe e objetos da classe onde a expressão se encontra.

[2] Diferentemente da versão de *change_ variable_ name* apresentada anteriormente, em que se modificava o nome de uma variável local, esta modificará o nome de uma variável membro de classe.

por Opdyke [Opd92]: para modificar o nome de uma variável pública v membro de uma classe c, o nível 1 de c deve ser percorrido a fim de renomear a declaração de v. Em seguida, o nível 3 de todas as classes que utilizam c deve ser percorrido em busca de utilizações de variáveis declaradas com tipo c. Para coletar o tipo de cara variável, o nível 2 deve ser observado. Outros *refactorings* que afetam todos os níveis são: *Self Encapsulate Field* [FBB+99], *RenameMethod* [FBB+99], *change_class_name*[Opd92][3].

Algumas refatorações são mais simples e afetam menos níveis. Dentre elas podemos citar algumas implementadas pela ferramenta *JRefactory*[jre] e que afetam apenas o nível 1: *Add Abstract Parent Class*, *Add Child Class*, *Move a class between packages* e *Push Up Abstract Method*. Outro exemplo é o *Convert local variable to field*, disponibilizado pela ferramenta Eclipse [ecl], que afeta apenas os níveis 1 e 2. A versão de *change_variable_name* para variáveis locais a métodos implementada nas Seções 3.7 e 5.5 também afeta apenas os níveis 1 e 2.

Refatorações que afetam apenas o nível 1 são, normalmente, simples e podem ser implementadas sem grandes esforços utilizando-se apenas os recursos básicos de MetaJ (sem recorrer à SCQL). Geralmente, refatorações que afetam vários níveis ao mesmo tempo podem ser construídas também utilizando-se apenas os recursos básicos de MetaJ, mas a utilização de SCQL pode tornar sua implementação mais simples, veja a implementação de *change_variable_name* apresentado nas Seções 3.7 e 5.5.

O *Self Encapsulate Field*, como já foi dito, é uma refatoração complexa. Seu objetivo é: *encapsular um campo que foi declarado como público e adicionar métodos* get *e* set *para recuperar e definir seu valor, respectivamente.*

A mecânica da refatoração é a seguinte:

1. criar métodos de leitura e gravação para o campo que será encapsulado;
2. encontrar todas as referências cruzadas e atualizá-las com chamadas a um dos dois métodos acrescentados no item anterior;
3. tornar o campo privado.

Vale destacar que atualizar referências cruzadas não é uma tarefa simples. Por razão desta complexidade, foram desenvolvidos componentes para controlar de escopo de nomes de variáveis, para calcular o tipo de expressões e para representar internamente classes e interfaces definidas no programa objeto (ver Seção 6.2). Para implementar tal refatoração utilizaremos o mesmo *plug-in* Java que foi utilizado nos exemplos das outras seções. Vale lembrar que a gramática utilizada para gerá-lo se encontra no Apêndice C

[3]Afeta o nível 3 para atualizar *type castings*.

6.2 O pacote *scql.examples.javaModel* – extraindo informações de tipo e escopo

As classes e interfaces definidas neste pacote podem ser divididas em três grupos:

1. as responsáveis pela representação interna de programas objeto;
2. as responsáveis pela implementação do cálculo do tipo de expressões;
3. as responsáveis pela implementação das regras de escopo.

6.2.1 A representação interna de programas Java

A representação interna de programas objeto é responsabilidade das classes definidas no pacote `scql.examples.types`, exceto a `TypeAnalyser` cuja responsabilidade é implementar as regras de escopo de nomes de variáveis e de tipagem de expressões. A Figura 6.1 mostra a organização das classes responsáveis pela representação interna de programas objeto.

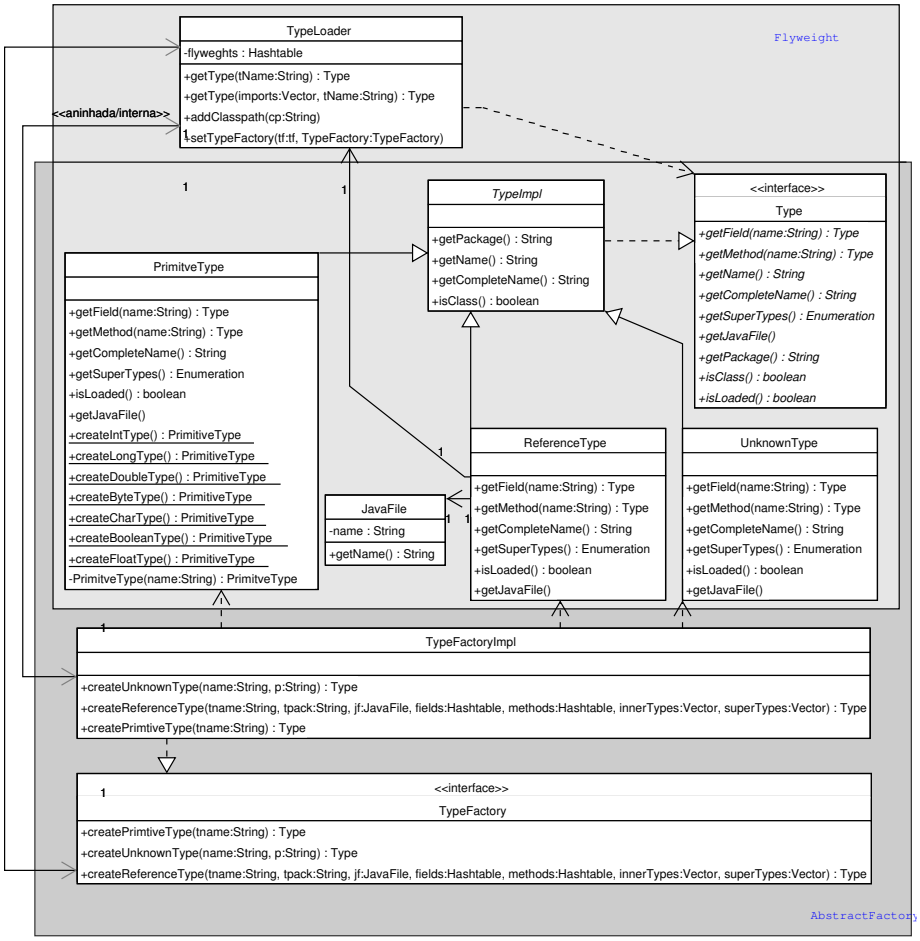


Figura 6.1: Classes utilizadas na representação interna de programas.

A interface `Type` é comum a todas as classes que representam um tipo Java (classe ou interface) definido ou utilizado pelo programa objeto. Os tipos primitivos de Java (`int`, `boolean`, `long`, `char`, `byte`, `float` e `double`) são representados por instâncias únicas da classe `PrimitiveType`. Para garantir a unicidade destas instâncias e para evitar que objetos de `PrimitiveType` sejam utilizados para representar tipos não primitivos, o padrão de projeto *Singleton* [GHJV94] foi utilizado. Na Figura 6.1 podem ser vistos métodos estáticos (sublinhados) para criação de objetos para representar cada tipo primitivo de Java (`createXXXType`, onde XXX é um tipo primitivo de Java). Também é possível perceber que o construtor da classe `PrimitiveType` foi declarado como privado, fazendo com que os métodos estáticos sejam a única forma de criação de objetos `PrimitiveType`.

Para representar tipos de dados declarados no programa objeto são utilizadas instâncias da classe `ReferenceType`. Estas instâncias carregam consigo a listagem dos métodos e dos campos[4] que foram definidos no corpo do tipo de dado representado por ela, bem como o tipo de cada um destes elementos. Por fim, a classe `UnknownType` representa um tipo que, apesar de ser utilizado pelo programa objeto, não foi possível de ser carregado. Esta classe foi definida para tornar possível a aplicação de refatorações mesmo sem se conhecer todos os tipos utilizados pelo programa objeto[5].

Para controlar a criação e a manutenção de objetos `Type`, evitando redundância e ambigüidade entre informações, bem como desperdício de memória, foi implementado o padrão de projeto *Flyweight* [GHJV94]. O objetivo deste padrão é:

> *Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina. [GHJV94]*

Neste padrão, um *FlyweightFactory* gerencia a criação de objetos *Flyweight* (na verdade, *ConcreteFlyweights*) evitando cópias desnecessárias, i.e., *"garantindo que os flyweights sejam compartilhados apropriadamente"* [GHJV94]. Dentre as classes utilizadas para representar internamente programas objeto, a `TypeLoader` faz o papel de *FlyweightFactory*. `Type` é um *Flyweight*, e `ReferenceType`, `PrimitiveType` e `UnknownType` são *ConcreteFlyweight*s. Seguindo ainda este padrão de projeto, a classe `TypeLoader` cria *ConcreteFlyweight*s sob demanda, i.e., assim que eles se tornam necessários. As informações armazenadas em cada *ConcreteFlyweight* são extraídas do código fonte do programa objeto. Para encontrar a declaração de um determinado tipo de dado, objetos `TypeLoader` percorrem os arquivos dos diretórios que foram especificados através do método `addClasspath`. Depois de encontrar a declaração de tipo, as informações são extraídas utilizando-se consultas SCQL e *templates*. Para cada tipo consultado, um objeto `ReferenceType` é criado utilizando-se um objeto da classe `TypeFactory`. As principais consultas realizadas sobre definições de tipos Java são:

[4] Classes aninhadas/internas não foram tratadas

[5] Na versão atual do pacote `scql.examples.javaMode.types` os tipos definidos nos pacotes que acompanham o *Java Development Kit* não são carregados, o que faz eles sejam carregados como desconhecidos (`UnknownType`).

```
VIEW TREE typeBody
         FOCUSED ON #field_declaration
         EXCLUDING #class_body, #interface_body
AS TABLE (#field_declaration fd, #variable_declarator_id vid)
WHERE fd.isComposedBy(vid)

VIEW TREE typeBody
         FOCUSED ON #method_header
         EXCLUDING #class_body, #interface_body
AS TABLE (#method_header mh)
```

A primeira é utilizada para encontrar todas as declarações de campos que ocorrem no corpo do tipo (`typeBody`), a segunda para encontrar as declarações de métodos (assinaturas dos métodos). Elas utilizam operadores de corte para evitar que classes ou interfaces aninhadas/internas sejam exploradas.

`TypeFactory` e `TypeFactoryImpl` foram projetadas utilizando-se o padrão de projeto *AbstractFactory*. O objetivo deste padrão é:

> *Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. [GHJV94]*

A classe `TypeFactoryImpl`, que faz o papel de *ConcreteFactory*, é aninhada à classe `TypeLoader`. Ela encapsula todo o processo de criação de objetos `Type`. O `TypeLoader` permite, através do método `setTypeFactory`, que sua fábrica de tipos seja substituída.

6.2.2 Implementação do calculo do tipo de expressões Java

A classe `TypeAnalyser` implementa rotinas capazes de retornarem o tipo de uma expressão Java. A implementação atual é limitada no sentido de que ela só é capaz de analisar nomes (trechos de código derivados a partir da produção `name` da gramática de Java, i.e., apenas referências-p de tipo `#name`), expressões pós-fixadas (referências-p de tipo `#postfix_expression`) e expressões primárias (referências-p de tipo `#primary`) [6]. A Figura 6.2 exibe a classe `TypeAnalyser` e seus métodos. Pode ser observado um método `getXXXType` para cada um dos tipos de expressão possível de ser analisada.

[6] Para o *Self Encapsulate Field* só são importantes as ocorrências de chamadas de métodos e acesso a campos de classes, por isso apenas as expressões pós-fixadas, as primárias e os nomes precisam ser analisados. Veja no Apêndice C que chamadas de métodos e acesso a campos só ocorrem nas produções `postfix_expression`, `primary` e `name`

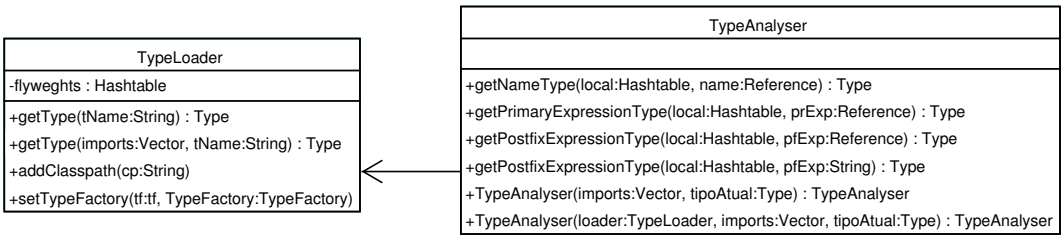


Figura 6.2: `TypeAnalyser`

Para se criar um `TypeAnalyser` deve-se especificar um `TypeLoader` (opcional, quando não especificado é criado uma nova instância), as importações válidas no contexto de avaliação da expressão e também a classe onde a expressão foi definida. Para calcular o tipo de uma expressão, através da chamada aos métodos `getXXXType`, deve ser fornecido um ambiente local (objeto que liga um tipo a cada variável local). Para todo tipo de expressão, o algoritmo de análise é o mesmo, exceto pelos detalhes sintáticos:

Seja e a expressão a ser analisada, l o ambiente local e t a classe onde a expressão foi definida, faça

Se e é uma expressão indivisível (identificador), então
    Se existir ligação de e a algum tipo te em l então retorne te,
    senão retorne o tipo de e no ambiente definido pela ligação dos nomes declarados em t a seus respectivos tipos
senão
    divida e em suas sub-expressões $e_1,\ e_2,\ e_3,\ \ldots,\ e_n$
    calcule o tipo t_i de cada expressão e_i
    combine os t_i's para calcular o tipo de e

Os passos sublinhados no algoritmo acima são dependentes da estrutura sintática da expressão que está sendo avaliada. Por isso, *templates* foram muito utilizados neste processo permitindo que o acesso às sub-expressões fosse bastante simplificado. Abaixo são mostrados dois exemplos destes *templates*:

```
template #primary PrimaryMethodInvocationRec #{
    #primary p . #identifier id (#argument_list_opt[ #argument_list al ]#)
}#

template #primary PrimaryFieldAccessRec #{
    #primary p . #identifier id
}#
```

O primeiro é utilizado para decompor invocações de métodos. O segundo para decompor acessos a campos.

6.2.3 Implementação das regras de escopo de nomes de variáveis

A Figura 6.3 mostra as classes utilizadas na implementação das regras de escopo de nomes de variáveis e seus relacionamentos. A classe `ExpressionExplorer` é capaz de percorrer corpos de

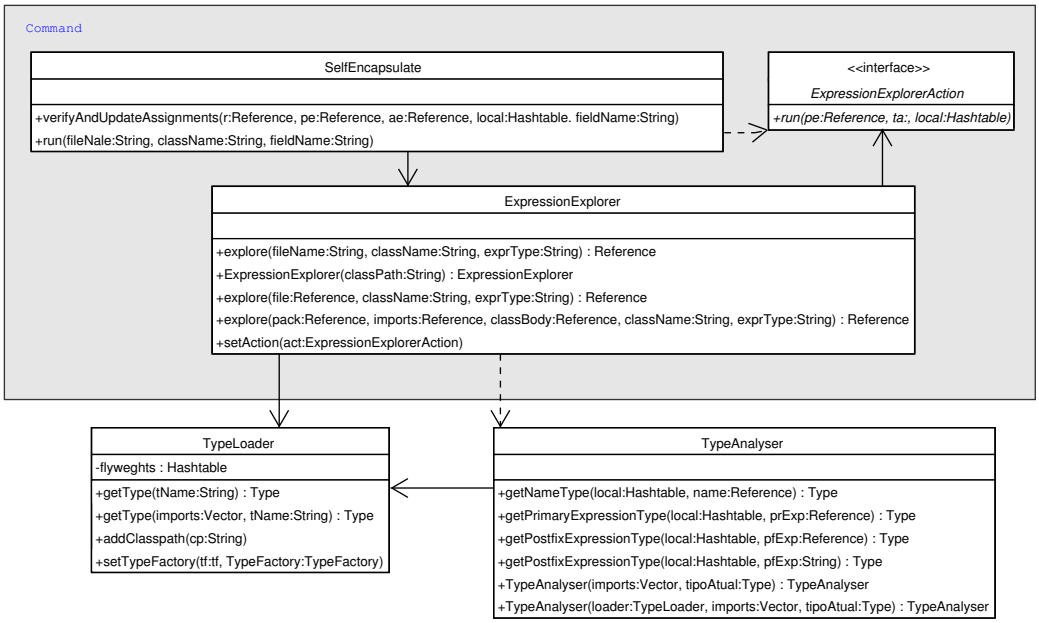


Figura 6.3: `ExpressionExplorer` e `ExplorerAction`

classes em busca de expressões de tipos específicos. Para cada expressão encontrada, o método `run` da ação (`ExpressionExplorerAction`) cadastrada através do método `setAction` é executada. O método `run` da ação cadastrada recebe como parâmetro: (i) a expressão alcançada, (ii) uma instância do analisador de tipos (`TypeAnalyser`), que permite à ação verificar o tipo da expressão, e (iii) uma tabela (uma instância de `Hashtable`) ligando cada um dos nomes válidos no escopo onde a expressão recebida como primeiro parâmetro foi definida ao seu respectivo tipo. A tabela recebida como parâmetro possui apenas os nomes válidos no escopo da expressão.

Para projetar tais classes foi utilizado o padrão *Command* [GHJV94]. O objetivo deste é:

> *Encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações (. . .). [GHJV94]*

Neste padrão `ExpressionExplorerAction` é um *Command*, `SelfEncapsulate` faz o papel de *Client*. Ela utiliza a classe `ExpressionExplorer`, passando a ela as ações (*Concrete Commands*) a serem executados para cada expressão encontrada. `SelfEncapsulte` encapsula a implementação da refatoração. Ela define várias classes internas que implementam ações

(`ExpressionExplorerAction`) de atualização de expressões. Estas classes são responsáveis por converter cada acesso ao campo que está sendo encapsulado para uma chamada a um método `get` ou `set`.

6.3 A classe `SelfEncapsulate`

O método `run` desta classe implementa o comportamento da refatoração. Primeiramente, ele acrescenta os métodos `get` e `set` para o campo a ser encapsulado. Em seguida, ele percorre todas as expressões para atualizar as referências cruzadas. Neste último passo, ele deve substituir todas as atribuições por chamadas ao método `set` e todos os acessos ao campo por chamadas ao `get`

Analisando-se a gramática de Java (Anexo C), é possível perceber sobre quais estruturas sintáticas a refatoração `SelfEncapsulate` deve agir para substituir as atribuições, são elas: `#statement_expressions`, para atualizar atribuições como `x = x + 1;` e `#assignment_expressions`, para atualizar atribuições como `if((x = readInt()) == 10){ ... }`. Para modificar os acessos aos campos, deve se agir sobre `#postfix_expression`'s, para substituir acessos como `x + 1`, e `#primary`, para casos com `this.x + 1`. Estes quatro tipos de expressões podem ocorrer de duas formas possíveis: acesso direto da classe ao seu campo (ex. `x = x+1`) ou acesso do campo através de um outro objeto (ex. `obj.x = obj.x+1`). Veja abaixo a estrutura básica do método `run` que encapsula o comportamento da refatoração:

```
public void run (String fileName, String className,  final String fieldName)...{
    ... // Adicionar métodos get e set
    // Ação para atualizar #assignment_expressions (acesso direto ao campo)
    pee.setAction(new ExpressionExplorerAction(){ ... });
    pee.explore(file, className, "#assignment_expression");

    // Ação para atualizar #statement_expressions (acesso direto ao campo)
    pee.setAction(new ExpressionExplorerAction(){ ... });
    pee.explore(file, className, "#statement_expression");
    ...
    // Ação para atualizar #postfix_expressions (acesso direto ao campo)
    pee.setAction(new ExpressionExplorerAction(){ ... });
    pee.explore(file, className, "#postfix_expression");
    ...
    // Ação para atualizar #assignment_expressions (acesso indireto ao campo)
    pee.setAction(new ExpressionExplorerAction(){ ... });
    pee.explore(file, className, "#assignment_expression");

    // Ação para atualizar #statement_expressions (acesso indireto ao campo)
    pee.setAction(new ExpressionExplorerAction(){ ... });
    pee.explore(file, className, "#statement_expression");
    ...
```

```
        // Ação para atualizar #primarys (acesso indireto ao campo)
        pee.setAction(new ExpressionExplorerAction(){ ... });
        pee.explore(file, className, "#primary");

        // Ação para atualizar #postfix_expressions (acesso indireto ao campo)
        pee.setAction(new ExpressionExplorerAction(){ ... });
        pee.explore(file, className, "#postfix_expression");
        ... // Atualizar declaração do campo
        // Imprimir o arquivo
        r.toFile(fileName);
    }
```

Primeiramente, adiciona-se métodos `get` e `set` à classe alvo da refatoração. Em seguida, para cada um dos tipos de expressão apresentados anteriormente, invoca-se o método `explore` de um objeto `ExpressionExplorer` (`pee`) previamente criado para se iniciar o processo de coleta e substituição dos acessos ao campo encapsulado. Este método recebe como parâmetro o corpo da classe a ser explorado, o nome da classe, e o tipo de expressão a ser procurada. Para definir a ação a ser executada a cada expressão encontrada, são implementadas classes internas que definem o comportamento a ser tomado para cada um dos tipos de expressão. Finalmente, a declaração do campo é atualizada e o arquivo é impresso novamente.

6.4 Avaliação do processo de implementação

Assim como foi citado para a implementação da refatoração *change_ variable_ name*, os recursos de MetaJ facilitaram muito a manipulação (analise e geração) de trechos de programa. *Templates*, por exemplo, foram muito importantes para se decompor expressões. SCQL facilitou consideravelmente a busca por trechos específicos de código, utilizando-se duas consultas relativamente simples foi possível coletar todos os métodos e atributos de uma classe.

Por outro lado, como a refatoração *Self Encapsulate Field* é mais complexa do que as apresentadas anteriormente, ela exigiu a implementação de componentes para calcular o tipo de expressões (ou termos de expressões), o que aumentou consideravelmente a complexidade do sistema. Apesar disso, devido a forma em que foram implementado, é possível reutilizar tais compoenentes na implementação de outras refatorações.

MetaJ + SCQL facilitou a manipulação de trechos de programas, mas a implementação de refatorações continua sendo uma tarefa muito complexa que exige um grande conhecimento sobre os detalhes sintáticos da linguagem objeto. Por isso, a definição de um *plug-in* a partir de uma gramática projetada especialmente para transformação de programas pode facilitar bastante a implementação deste tipo de meta-programa. Isso pode reduzir a quantidade de estruturas sintáticas a serem modificadas. Veja que, na implementação apresentada neste capítulo, quatro tipos de estruturas sintáticas foram afetadas.

Capítulo 7

Avaliação da Ferramenta, Conclusões e Trabalhos Futuros

Meta-programas, apesar de parecerem raros, são bastante comuns e amplamente utilizados. A compilação talvez seja a aplicação mais conhecida. Além desta, também podem ser citadas: transformação de programas, compreensão de programas [Rug95], avaliação parcial [JGS93], mobilidade de código [TCM98], refatorações [FBB+99, Opd92], etc..

Devido a grande dificuldade em se construir meta-programas pela utilização de linguagens de programação de uso geral, até mesmo com o auxílio de geradores de parser, tornam-se necessárias ferramentas especialmente ajustadas para atender às necessidades da meta-programação.

Dentre as ferramentas disponíveis na literatura, estudamos: TXL, ASF+SDF, Stratego, JaTS, TAWK e SCRUPLE. Apesar de constituírem ferramentas com as quais várias aplicações já foram construídas, é possível detectar algumas deficiências:

- TXL, ASF+SDF e Stratego são ferramentas projetadas como sistemas de reescrita, o que, além de prejudicar o aprendizado da ferramenta (já que este não é um paradigma de programação muito popular). Estas ferramentas são também linguagens completamente novas, com suas próprias construções sintáticas e conceitos. Isso é outro fator que torna o aprendizado da ferramenta mais difícil;

- No caso da linguagem TXL, não foram encontradas referências sobre recursos que permitam aos seus meta-programas interagir com o usuário (como bibliotecas de I/O ou GUI), *multi-threading*, tratamento de exceções e comunicação entre processos. Além disso, a ferramenta utiliza um *parser full backtracking top-down* para reconhecer sentenças da linguagem objeto, o que pode tornar o tempo de processamento de programas muito alto ou mesmo impraticável para algumas gramáticas.

  Em ASF+SDF, o tratamento de exceções é um recurso precário;

- JaTS, TAWK e SCRUPLE apresentam a deficiência de não permitirem a flexibilidade de

se trabalhar com múltiplas linguagens objeto;

- Todas estas ferramentas (exceto SCRUPLE que só permite consultas em código objeto) apresentam uma estrutura de descrição de meta-programas muito rígida: pares padrão-ação ou pares padrão de casamento/padrão de reescrita.

Alguns pontos em comum podem ser percebidos nestas ferramentas: todas elas disponibilizam disponibilizam padrões de programas, variáveis para representar ou armazenar trechos de programas objeto, possuem ou permitem definir caminhamentos sobre o código objeto e possuem elementos responsáveis por armazenar as características específicas da linguagem objeto. Estes quatro elementos são essencias para estas ferramentas. A falta de algum deles diminuiria consideravelmente sua facilidade de utilização na descrição de meta-programas.

A proposta desta dissertação é MetaJ, uma ferramenta de meta-programação baseada no paradigma orientado a objetos e que permite padrões de programas *by example* reutilizáveis.

O ambiente também permite o acesso a todos os recursos básicos da linguagem Java – bibliotecas de I/O e GUI, *multi-threading*, tratamento de exceções e comunicação entre processos – e apresenta uma estrutura de descrição de meta-programas bastante flexível. Estes benefícios são alcançados devido a forma em que MetaJ foi projetado: ele é um conjunto de abstrações, realizadas como classes Java, que encapsulam os recursos essenciais de uma ferramenta de meta-programação. Por isso, um meta-programa MetaJ é um programa Java que utiliza estas abstrações. Desta maneira MetaJ é uma proposta de solução para as deficiências das ferramentas apresentadas anteriormente.

Abaixo apresentamos uma comparação dos recursos disponibilizados por MetaJ com aqueles encontrados nas ferramentas apresentadas anteriormente.

- Referências-p: Assim como as ferramentas apresentadas anteriormente, MetaJ permite a declaração de variáveis para representar (ou armazenar) trechos de programas objeto: as referências-p. Nas ferramentas apresentadas anteriormente, Em MetaJ, além poderem ser criados (ou declarados) dentro de padrões de programa pelo uso de meta-variáveis, referências-p podem também ser instanciadas em qualquer ponto do meta-programa. Esta abstração disponibiliza vários métodos que permitem comparar estruturalmente dois trechos de código objeto, realizar testes de tipos, verificar se uma estrutura é composta por outra, adicionar, remover e recuperar elementos de listas e também converter o valor de uma referência para seu formato textual.

  Por outro lado, em comparação com JaTS e TAWK, referências-p não oferecem nenhum tipo de operação que disponibilize informações mais específicas sobre a linguagem objeto. Em MetaJ estas informações devem ser codificadas pelo meta-programador pela utilização dos recursos do ambiente.

  Uma característica importante de referências-p é que todas as suas operações mantém a

consistência sintática do código armazenado, fazendo com que MetaJ atenda ao requisito de garantir a correção sintática de programas objeto (veja Seção 2.2).

- Iteradores: É uma abstração que encapsula um caminhamento por um trecho de programa objeto, e oferece total controle sobre os passos do caminhamento. A possibilidade de se criar várias instâncias de um mesmo iterador permite que se mantenham vários caminhamentos independentes sobre uma mesma estrutura sintática. Para permitir o controle sobre o processo de caminhamento esta abstração oferece operações que possibilitam ao meta-programa escolher entre explorar a próxima estrutura sintática, ignorá-la, substituí-la ou mesmo interromper o processo de caminhamento.

  Em comparação com outras ferramentas de meta-programação, em que o caminhamento é pré-definido ou declarado, iteradores oferecem bastante flexibilidade pois permitem controlar o processo de exploração do programa objeto passo a passo. Mas, por outro lado, a possibilidade de se adotar apenas uma ordem de caminhamento (*top-down*) é uma deficiência da abstração.

  Além disso, a intensa utilização de iteradores implica em um meta-programa carregado de detalhes operacionais. Uma proposta de solução para tal deficiência é SCQL.

- *Templates*: Esta abstração encapsula um padrão de programa descrito *by example*. A possibilidade de se invocar as operações de casamento, impressão e redefinição dos valores de meta-variáveis (através de referências-p) sem uma ordem pré-definida dá ao meta-programador bastante flexibilidade, em contraste com a estrutura fixa de descrição de meta-programas apresentada por outras ferramentas.

  Além disso, *templates* em MetaJ são compilados em módulos separados do meta-programa que os utiliza, permitindo assim a reutilização dos mesmos. Esta possibilidade não foi apresentada por nenhuma das ferramentas descritas anteriormente. Outro fator que contribui para a reutilização de padrões é o fato de permitir a descrição de trechos de padrões opcionais. Este recurso já havia sido implementado anteriormente por JaTS. Por outro lado, a utilização de trechos opcionais em padrões de programa torna sua descrição mais complexa.

- *Plug-ins*: Estes elementos encapsulam informações sobre a linguagem objeto. A possibilidade de adicionar e removê-los do ambiente livremente, faz com que MetaJ atenda ao requisito de manipulação de múltiplas linguagens objeto.

  Utilizando-se o processador de gramáticas Cup, é possível gerar o *plug-in* MetaJ de uma linguagem objeto a partir da especificação de sua gramática livre do contexto. Apesar da sua importância no ambiente, *plug-ins* só podem trabalhar com gramáticas LALR, o que é um ponto negativo. ASF+SDF e Stratego são mais flexíveis quanto a este ponto pois

elas utilizam técnicas GLR, o que permite a especificação de qualquer gramática livre do contexto.

Apesar de suprir as deficiências das ferramentas de meta-programação encontradas na literatura, meta-programas MetaJ se mostraram carregados de detalhes operacionais principalmente quando se torna necessário realizar buscas por trechos específicos do programa objeto. Esta deficiência é conseqüência da grande flexibilidade oferecida pela ferramenta e também da forma adotada para se controlar caminhamentos pelo programa objeto (iteradores).

SCQL é uma proposta de solução para este problema. Esta é uma linguagem desenvolvida como uma extensão de MetaJ que permite descrever consultas sobre programas objeto. SCQL pode ser dividida em duas partes, a API, que permite a meta-programas MetaJ realizarem consultas sobre código objeto, e a linguagem de consultas em si. O projeto da linguagem foi inspirado na forma de gerenciamento de dados adotada pelos sistemas de gerenciamento de banco de dados relacionais. A idéia de SCQL é visualizar o código de um programa como uma tabela de forma a permitir que consultas sejam realizadas sobre ela. Para isso, a linguagem disponibiliza dois comandos `VIEW TREE`, para visualizar um programa como uma tabela, e `SELECT`, para combinar duas ou mais tabelas (ou resultados de consultas). Na descrição de uma consulta é possível especificar operadores de corte, o que pode tornar consultas muito mais eficientes.

Apesar de oferecer um nível de abstração mais elevado, a utilização dos recursos SCQL aumenta a quantidade de passagens pelo código objeto, o que pode acarretar em uma perda na eficiência dos meta-programas. Para comprovar esta perda é necessário realizar medidas de tempo de meta-programas construídos com SCQL e compará-las com medidas de implementações equivalentes em MetaJ. Também é preciso comparar SCQL com outras linguagens de consultas.

7.1 Conclusões

MetaJ é um ambiente de meta-programação flexível que propõem algumas soluções para as deficiências das ferramentas encontradas na literatura. A principal vantagem deste ambiente é seu enquadramento em um paradigma de programação popular e a sua construção sobre uma linguagem amplamente utilizada, Java. Tais características contribuem para a aceitação da ferramenta pois sistemas de grande porte são desenvolvidos em linguagens orientados a objeto ou imperativas. Além disso, MetaJ também é simples de ser integrado em sistemas desenvolvidos em Java, basta que seus módulos sejam importados.

Uma contribuição importante do trabalho de desenvolvimento de MetaJ foi a identificação de quatro recursos essenciais de meta-programação. Além disso, o projeto do ambiente, guiado pela implementação de abstrações para representar estes elementos, é também possível de ser repetido para se estender outras linguagens de uso geral para atender aos requisitos da meta-programação.

SCQL é uma proposta de solução para o principal problema de MetaJ: a grande quantidade de detalhes operacionais encontrados em seus meta-programas. Com SCQL é possível descrever

consultas em código objeto para expressar as relacões entre estruturas sintáticas da linguagem objeto (ex.: declarações e uso de variáveis, declarações e uso de tipos, etc.). Além de propor uma visão mais declarativa da meta-programação, SCQL mostra também a possibilidade de construção de ambientes a partir dos recursos e conceitos definidos por MetaJ.

Apesar das contribuições apresentadas acima, MetaJ e SCQL possuem algumas deficiências que são temas para trabalhos futuros.

7.2 Trabalhos futuros

Abaixo apresentamos uma lista de trabalhos futuros juntamente com uma breve descrição de cada um deles.

- **Implementação de outros ambientes de meta-programação.** A implementação de ambientes de meta-programação seguindo a mesma idéia de MetaJ, mas como extensão de outras linguagens de programação. Por exemplo, MetaC++, MetaObjectPascal, etc..

- **Construção de bibliotecas com características inerentes à linguagem objeto.** Meta-programas geralmente precisam de informações sobre as regras de escopo e do sistema de tipos da linguagem objeto. É necessário implementar estas regras em MetaJ+SCQL para as linguagens de programação atuais e disponibilizá-las como bibliotecas que poderão ser importadas pelo meta-programa.

- **Iteradores mais flexíveis.** Stratego é uma linguagem que permite grande flexibilidade em relação à forma de se caminhar por um código objeto. Iteradores, da forma em que se encontram implementados atualmente, só permitem uma ordem de caminhamento: *top-down*. É necessário estender tais elementos para permitir a exploração mais flexível do código. Uma possível solução é permitir que, a cada estrutura sintática alcançada, seja permitido escolher a forma em que ela deverá ser explorada (*top-down*, *bottom-up*, ...).

- ***Plug-ins* com *parser* GLR.** Uma das conclusões resultantes da implementação da refatoração *Self Encapsulate Field* (Capítulo 6) foi que a definição de um *plug-in* a partir de uma gramática projetada especialmente para meta-programação poderia facilitar bastante a implementação daquele tipo de meta-programa. O problema é que a redefinição de algumas estruturas sintáticas pode fazer com que a gramática da linguagem objeto deixe de pertencer à classe de GLCs LR(1). A solução apresentada por ASF+SDF e Stratego para este problema é a disponibilização de *parsers* GLR. A utilização deste tipo de ferramenta nos *plug-ins* poderá permitir a descrição de gramáticas mais adequadas para a meta-programação e também poderá permitir descrever padrões de programa de maneira mais flexível (provavelmente sem as restrições sobre estruturas classificadas como listas não uniformes).

- **Avaliação experimental de MetaJ.** É necessário avaliar experimentalmente MetaJ através da implementação de um sistema de grande porte. Esta avaliação deverá verificar se os requisitos de flexibilidade, legibilidade e expressividade da ferramenta, apresentados na Seção 2.2, foram alcançados.

- **Análise da eficiência e otimização de MetaJ.** Uma das questões que ficou em aberta neste trabalho foi à eficiência (tempo de processamento) de meta-programas desenvolvidos utilizando-se MetaJ em comparação com os mesmos implementados em outras ferramentas. Com estas medidas será possível detectar os pontos de maior ineficiência do ambiente sendo possível também propor **otimizações no código de MetaJ**.

- **Análise da eficiência de SCQL.** Assim como foi dito para MetaJ, é preciso analisar a eficiência (tempo de processamento de consultas) de SCQL em comparação com outras ferramentas. Neste trabalho não foi analisada a perda de eficiência causada pelo uso de SCQL. É necessário comparar os tempos de execução de meta-programas que utilizam SCQL com meta-programas que utilizam apenas MetaJ.

Bibliografia

[Bag03] O. S. Bagge. Codeboost: A framework for transforming C++ programs. Master's thesis, University of Bergen, March 2003.

[BDK+96] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[BHK89] J. A. Bergstra, Jan Heering, and Paul Klint. *Algebraic specification.* ACM Press, 1989.

[BKHV03] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[BMB+98] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick, and T. J. Mowbray. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns.* John Wiley & Sons Ltd, Chichester, England, 1996.

[CB01] Fernando Castor and Paulo Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programming Languages*, pages 236–251, May 2001.

[CC93] James R. Cordy and Ian H. Carmichael. The TXL programming language syntax and informal semantics. Technical report, Department of Computing and Information Science. Queens University at Kingston, Kinston, Canada, June 1993.

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[CHP88] James R. Cordy, C.D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proc. IEEE 1988 International Conference on Computer Languages*, pages 280–285, October 1988.

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):20–54, 1984.

[Cor04] James R. Cordy. TXL - a language for programming language tools and applications. In *Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications(LDTA 2004)*, Barcelona, April 2004.

[COS+01] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379, October 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.

[dH03] Robbert de Haan. Using ASF+SDF for the verification of annotated Java programs. Master's thesis, University of Amsterdam, March 2003.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[ecl] Eclipse Home Page. http://www.eclipse.org/.

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

[GA95] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.

[GAM96] W. Griswold, D. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the IEEE Fourth Workshop on Program Comprehension,* (Berlin, Germany; March 29-31, 1996). IEEE Computer Society Press, March 1996.

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

[Hud99] Scott E. Hudson. Cup LALR parser generator for Java – User's Manual. *Available at http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html*, July 1999.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[jre] JRefactory Home Page. http://jrefactory.sourceforge.net/csrefactory.html.

[Kli03] Paul Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[Lam87] David Alex Lamb. Idl: sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, 1987.

[LS90] M. E. Lesk and E. Schmidt. Lex: a lexical analyzer generator. *UNIX: research system (10th ed.)*, 2:375–387, 1990.

[MMW01] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[OBMB04] Ademir de Alvarenga Oliveira, Thiago H. Braga, Marcelo de Almeida Maia, and Roberto da Silva Bigonha. MetaJ: An Extensible Environment for Metaprogramming in Java. *Journal of Universal Computer Science*, 10(7):872–891, July 2004.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[OV02] Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

[Pax95] V. Paxson. Flex: Fast lexical analyzer generator. Technical report, Lawrence Berkeley Laboratory, 1995.

[PP94] S. Paul and A. Prakash. A framework for source code analysis using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, Abril 1996.

[Rug95] Spencer Rugaber. Program comprehension. In *Encyclopedia of Computer Science and Technology*, volume 35(20), pages 341–368. Marcel Dekker, Inc., New York, 1995.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. *Lecture Notes in Computer Science*, 2196:2–, 2001.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 2001.

[TCM98] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 135. IEEE Computer Society, 1998.

[vdBBK03] Mark van den Brand Brand and Paul Klint. ASF + SDF meta-environment user manual, September 2003. http://www.cwi.nl/projects/MetaEnv/.

[vdBKV97] Mark van den Brand, Paul Klint, and Chris Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM SIGSOFT Software Engineering Notes*, 22(1):57–68, 1997.

[vdBKV03] Mark van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[vdBSV98] Mark van den Brand, Alex Sellink, and Chris Verhoef. Current parsing techniques in software renovation considered harmful. *6th International Workshop on Program Comprehension*, pages 108–117, 1998.

[vdBSVV02] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Computational Complexity*, pages 143–158, 2002.

[vdBvDH+01] Mark van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF + SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370, 2001.

[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.

[Vis01] 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.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[Vis03] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, November 2003. (Draft; Accepted for publication).

[vW03] J. van Wijngaarden. Code generation from a domain specific language. designing and implementing complex program transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, July 2003.

[Zaa01] Hans Zaadnoordijk. Source code transformations using the new ASF+SDF meta-environment. Master's thesis, University of Amsterdam, March 2001.

Apêndice A

Gramática para a linguagem de declaração de templates

```
terminal PACKAGE, TEMPLATE, LANGUAGE, SEMICOLON, DOT, IDENTIFIER,
         TEMPLATEBODY, MJTYPE;

non terminal goal, compilation_unit, package_declaration_opt,
              package_declaration, template_declarations, template_declaration,
              language_declaration, name, simple_name, template_type,
              template_body, qualified_name;

compilation_unit → package_declaration_opt
                     language_declaration
                     template_declarations;
name → simple_name
         | name qualified_name;
qualified_name → DOT IDENTIFIER;
simple_name → IDENTIFIER;
package_declaration_opt → package_declaration | λ;
language_declaration → LANGUAGE simple_name SEMICOLON;
package_declaration → PACKAGE name SEMICOLON;
template_declarations → template_declarations template_declaration
                          | template_declaration;
template_declaration → TEMPLATE template_type simple_name template_body;
template_type → MJTYPE;
template_body → TEMPLATEBODY;
```

Apêndice B

Gramática para arquivos de declaração de *plug-ins*

```
terminal SEMICOLON, DOT, EQ, LBRACE, RBRACE, CLASS, BOOLEAN_LITERAL,
NULL_LITERAL, INTEGER_LITERAL, FLOATING_POINT_LITERAL,  STRING_LITERAL,
CHARACTER_LITERAL, IDENTIFIER;

non terminal goal, plugin_declarations_opt, plugin_declarations,
plugin_declaration, property_declarations_opt, property_declarations,
property_declaration, value, type, name, literal;

goal → plugin_declarations_opt;
plugin_declarations_opt → plugin_declarations | λ ;
plugin_declarations → plugin_declarations plugin_declaration
                      |   plugin_declaration;

plugin_declaration → IDENTIFIER LBRACE property_declarations_opt RBRACE;
property_declarations_opt →  property_declarations | λ;
property_declarations → property_declarations property_declaration
                      |   property_declaration;

property_declaration → IDENTIFIER EQ value SEMICOLON;
name → IDENTIFIER | name DOT IDENTIFIER;
value →  BOOLEAN_LITERAL | NULL_LITERAL
       | INTEGER_LITERAL | FLOATING_POINT_LITERAL
       | STRING_LITERAL  | CHARACTER_LITERAL
       | name | name DOT CLASS;
```

Apêndice C

Gramática do *plug-in* Java

```
terminal BOOLEAN, BYTE, SHORT, INT, LONG, CHAR, FLOAT, DOUBLE,
terminal LBRACK, RBRACK, IDENTIFIER, DOT;
terminal SEMICOLON, MULT, COMMA, LBRACE, RBRACE, EQ, LPAREN, RPAREN, COLON;
terminal PACKAGE, IMPORT, PUBLIC, PROTECTED, PRIVATE, STATIC;
terminal ABSTRACT, FINAL, NATIVE, SYNCHRONIZED, TRANSIENT, VOLATILE;
terminal CLASS, EXTENDS, IMPLEMENTS, VOID, THROWS, THIS, SUPER;
terminal INTERFACE, IF, ELSE, SWITCH, CASE, DEFAULT;
terminal DO, WHILE, FOR, BREAK, CONTINUE;
terminal RETURN, THROW, TRY, CATCH, FINALLY;
terminal NEW, PLUSPLUS, MINUSMINUS, PLUS, MINUS, COMP, NOT, DIV, MOD;
terminal LSHIFT, RSHIFT, URSHIFT, LT, GT, LTEQ, GTEQ, INSTANCEOF;
terminal EQEQ, NOTEQ, AND, XOR, OR, ANDAND, OROR, QUESTION;
terminal MULTEQ, DIVEQ, MODEQ, PLUSEQ, MINUSEQ;
terminal LSHIFTEQ, RSHIFTEQ, URSHIFTEQ, ANDEQ, XOREQ, OREQ;
terminal INTEGER_LITERAL, FLOATING_POINT_LITERAL, BOOLEAN_LITERAL;
terminal CHARACTER_LITERAL, STRING_LITERAL, NULL_LITERAL;
// Reserved but unused:
terminal CONST, GOTO;
// strictfp keyword, new in Java 1.2
terminal STRICTFP;
// lexer compatibility with Java 1.4
terminal ASSERT;
// lexer compatibility with Java 1.5
terminal ELLIPSIS;
terminal ENUM;

// 19.2) The Syntactic Grammar
non terminal goal;
non terminal literal;
// 19.4) Types, Values, and Variables
```

```
non terminal type, primitive_type, numeric_type, integral_type, floating_point_type;
non terminal reference_type, class_or_interface_type, class_type, interface_type;
non terminal array_type, var_init;
// 19.5) Names
non terminal name, qualified_name;
// 19.6) Packages
non terminal compilation_unit, package_declaration_opt, package_declaration;
non terminal import_declarations_opt, import_declarations, type_declaration;
non terminal type_declarations_opt, type_declarations, import_declaration;
non terminal single_type_import_declaration, type_import_on_demand_declaration;

// 19.7) Productions used only in the LALR(1) grammar
non terminal modifiers_opt, modifiers, modifier;
// 19.8.1) Class Declaration
non terminal class_declaration, super, super_opt;
non terminal interfaces, interfaces_opt, interface_type_list;
non terminal class_body, class_body_declarations, class_body_declarations_opt;
non terminal class_body_declaration, class_member_declaration;
// 19.8.2) Field Declarations
non terminal field_declaration, variable_declarators, variable_declarator;
non terminal variable_declarator_id, variable_initializer;
// 19.8.3) Method Declarations
non terminal method_declaration, method_header, method_declarator;
non terminal formal_parameter_list_opt, formal_parameter_list;
non terminal formal_parameter, throws_opt, throws, class_type_list, method_body;
// 19.8.4) Static Initializers
non terminal static_initializer;
// 19.8.5) Constructor Declarations
non terminal constructor_declaration, constructor_declarator;
non terminal constructor_body, explicit_constructor_invocation;
// 19.9.1) Interface Declarations
non terminal interface_declaration, extends_interfaces_opt, extends_interfaces;
non terminal interface_body, interface_member_declarations_opt;
non terminal interface_member_declaration, constant_declaration;
non terminal abstract_method_declaration, interface_member_declarations;
// 19.10) Arrays
non terminal array_initializer, variable_initializers;
// 19.11) Blocks and Statements
non terminal block, block_statements_opt, block_statements, block_statement;
non terminal local_variable_declaration_statement, local_variable_declaration;
non terminal statement, statement_no_short_if,statement_without_trailing_substatement;
non terminal empty_statement, labeled_statement, labeled_statement_no_short_if;
non terminal expression_statement, statement_expression, if_then_statement;
```

```
non terminal if_then_else_statement, if_then_else_statement_no_short_if;
non terminal switch_statement, switch_block, switch_block_statement_groups;
non terminal switch_block_statement_group, switch_labels, switch_label;
non terminal while_statement, while_statement_no_short_if, do_statement;
non terminal for_statement, for_statement_no_short_if, for_init_opt, for_init;
non terminal for_update_opt, for_update, statement_expression_list;
non terminal identifier_opt, break_statement, continue_statement;
non terminal return_statement, throw_statement, synchronized_statement, try_statement;
non terminal catches_opt, catches, catch_clause, finally;
// 19.12) Expressions
non terminal primary, primary_no_new_array, class_instance_creation_expression;
non terminal argument_list_opt, argument_list;
non terminal array_creation_init, array_creation_uninit;
non terminal dim_exprs, dim_expr, dims_opt, dims;
non terminal field_access, method_invocation, array_access;
non terminal postfix_expression, postincrement_expression, postdecrement_expression;
non terminal unary_expression, unary_expression_not_plus_minus;
non terminal preincrement_expression, predecrement_expression;
non terminal cast_expression, multiplicative_expression, additive_expression;
non terminal shift_expression, relational_expression, equality_expression;
non terminal and_expression, exclusive_or_expression, inclusive_or_expression;
non terminal conditional_and_expression, conditional_or_expression;
non terminal conditional_expression, assignment_expression;
non terminal assignment, assignment_operator, identifier;
non terminal expression_opt, expression,constant_expression, final_modifier_opt;

start with goal;

// 19.2) The Syntactic Grammar
goal → compilation_unit;
// 19.3) Lexical Structure.
literal → INTEGER_LITERAL | FLOATING_POINT_LITERAL | BOOLEAN_LITERAL
    | CHARACTER_LITERAL | STRING_LITERAL | NULL_LITERAL;
// 19.4) Types, Values, and Variables
type    → primitive_type | reference_type;
primitive_type → numeric_type | BOOLEAN;
numeric_type → integral_type | floating_point_type;
integral_type → BYTE | SHORT | INT | LONG | CHAR ;
floating_point_type → FLOAT | DOUBLE;
reference_type → class_or_interface_type | array_type;
class_or_interface_type → name;
class_type → class_or_interface_type;
interface_type → class_or_interface_type;
```

```
array_type → primitive_type dims | name dims ;

// 19.5) Names
name → identifier | name qualified_name;
qualified_name → DOT identifier;

// 19.6) Packages
compilation_unit →
        package_declaration_opt
        import_declarations_opt
        type_declarations_opt ;
package_declaration_opt → package_declaration | λ;
import_declarations_opt → import_declarations | λ;
type_declarations_opt→ type_declarations | λ;
import_declarations → import_declaration
    | import_declarations import_declaration;
type_declarations → type_declaration
    | type_declarations type_declaration;
package_declaration → PACKAGE name SEMICOLON;
import_declaration →     single_type_import_declaration
    | type_import_on_demand_declaration;
single_type_import_declaration → IMPORT name SEMICOLON;
type_import_on_demand_declaration → IMPORT name DOT MULT SEMICOLON;
type_declaration → class_declaration | interface_declaration | SEMICOLON;

// 19.7) Productions used only in the LALR(1) grammar
modifiers_opt→ modifiers | λ;
modifiers → modifier | modifiers modifier;
modifier →  PUBLIC | PROTECTED | PRIVATE | STATIC
    | ABSTRACT | FINAL | NATIVE | SYNCHRONIZED | TRANSIENT | VOLATILE | STRICTFP;

// 19.8) Classes
// 19.8.1) Class Declaration:
class_declaration → modifiers_opt CLASS identifier super_opt interfaces_opt
                                  class_body ;
super → EXTENDS class_type;
super_opt → super | λ;
interfaces → IMPLEMENTS interface_type_list;
interfaces_opt→ interfaces | λ;
interface_type_list → interface_type | interface_type_list COMMA interface_type;
class_body → LBRACE class_body_declarations_opt RBRACE;
class_body_declarations_opt → class_body_declarations | λ;
class_body_declarations → class_body_declaration
```

```
    | class_body_declarations class_body_declaration;
class_body_declaration → static_initializer | constructor_declaration
    | block | class_member_declaration;
class_member_declaration → field_declaration | method_declaration
    | class_declaration  | interface_declaration | SEMICOLON;

// 19.8.2) Field Declarations
field_declaration → modifiers_opt type variable_declarators SEMICOLON;
variable_declarators → variable_declarator
            | variable_declarators COMMA variable_declarator;
variable_declarator → variable_declarator_id var_init;
var_init → EQ variable_initializer | λ;
variable_declarator_id → identifier | variable_declarator_id LBRACK RBRACK;
variable_initializer → expression  | array_initializer;

// 19.8.3) Method Declarations
method_declaration → method_header method_body ;
method_header → modifiers_opt type method_declarator throws_opt
    | modifiers_opt VOID method_declarator throws_opt;
method_declarator → identifier LPAREN formal_parameter_list_opt RPAREN
    | method_declarator LBRACK RBRACK; // deprecated
formal_parameter_list_opt → formal_parameter_list | λ;
formal_parameter_list → formal_parameter
    | formal_parameter_list COMMA formal_parameter;
formal_parameter → final_modifier_opt type variable_declarator_id;
final_modifier_opt → FINAL | λ;
throws_opt → throws | λ;
throws → THROWS class_type_list;
class_type_list → class_type | class_type_list COMMA class_type;
method_body → block | SEMICOLON;

19.8.4) Static Initializers
static_initializer → STATIC block;

// 19.8.5) Constructor Declarations
constructor_declaration → modifiers_opt constructor_declarator throws_opt
                                        constructor_body;
constructor_declarator → identifier LPAREN formal_parameter_list_opt RPAREN;
constructor_body → LBRACE explicit_constructor_invocation block_statements RBRACE
    | LBRACE explicit_constructor_invocation RBRACE | LBRACE block_statements RBRACE
    | LBRACE RBRACE;
explicit_constructor_invocation → THIS LPAREN argument_list_opt RPAREN SEMICOLON
    | SUPER LPAREN argument_list_opt RPAREN SEMICOLON
```

```
    | primary DOT SUPER LPAREN argument_list_opt RPAREN SEMICOLON
    | name DOT SUPER LPAREN argument_list_opt RPAREN SEMICOLON;

// 19.9) Interfaces
// 19.9.1) Interface Declarations
interface_declaration → modifiers_opt INTERFACE identifier extends_interfaces_opt
                                    interface_body;
extends_interfaces_opt → extends_interfaces | λ;
extends_interfaces → EXTENDS interface_type | extends_interfaces COMMA interface_type;
interface_body → LBRACE interface_member_declarations_opt RBRACE;
interface_member_declarations_opt → interface_member_declarations | λ;
interface_member_declarations → interface_member_declaration
    | interface_member_declarations interface_member_declaration;
interface_member_declaration → constant_declaration | abstract_method_declaration
    | class_declaration | interface_declaration | SEMICOLON;
constant_declaration → field_declaration;
abstract_method_declaration → method_header SEMICOLON;

// 19.10) Arrays
array_initializer → LBRACE variable_initializers COMMA RBRACE
    | LBRACE variable_initializers RBRACE | LBRACE COMMA RBRACE
    | LBRACE RBRACE;
variable_initializers → variable_initializer
    | variable_initializers COMMA variable_initializer;

// 19.11) Blocks and Statements
block → LBRACE block_statements_opt RBRACE;
block_statements_opt → block_statements | λ;
block_statements → block_statement | block_statements block_statement;
block_statement → local_variable_declaration_statement
    | statement | class_declaration | interface_declaration;
local_variable_declaration_statement → local_variable_declaration SEMICOLON;
local_variable_declaration → type variable_declarators
    | FINAL type variable_declarators;
statement → statement_without_trailing_substatement | labeled_statement
    | if_then_statement | if_then_else_statement | while_statement | for_statement;
statement_no_short_if → statement_without_trailing_substatement
    | labeled_statement_no_short_if | if_then_else_statement_no_short_if
    | while_statement_no_short_if | for_statement_no_short_if;
statement_without_trailing_substatement → block | empty_statement
    | expression_statement | switch_statement | do_statement | break_statement
    | continue_statement | return_statement
    | synchronized_statement | throw_statement | try_statement;
```

```
empty_statement → SEMICOLON;
labeled_statement → identifier COLON statement;
labeled_statement_no_short_if → identifier COLON statement_no_short_if;
expression_statement → statement_expression SEMICOLON;
statement_expression → postfix_expression | assignment;
if_then_statement → IF LPAREN expression RPAREN statement;
if_then_else_statement → IF LPAREN expression RPAREN statement_no_short_if
                                        ELSE statement;
if_then_else_statement_no_short_if →
            IF LPAREN expression RPAREN statement_no_short_if
            ELSE statement_no_short_if;
switch_statement → SWITCH LPAREN expression RPAREN switch_block;
switch_block → LBRACE switch_block_statement_groups switch_labels RBRACE
    | LBRACE switch_block_statement_groups RBRACE | LBRACE switch_labels RBRACE
    | LBRACE RBRACE;
switch_block_statement_groups → switch_block_statement_group
    | switch_block_statement_groups switch_block_statement_group;
switch_block_statement_group → switch_labels block_statements;
switch_labels → switch_label
    | switch_labels switch_label;
switch_label → CASE constant_expression COLON
    | DEFAULT COLON;
while_statement → WHILE LPAREN expression RPAREN statement;
while_statement_no_short_if → WHILE LPAREN expression RPAREN statement_no_short_if;
do_statement → DO statement WHILE LPAREN expression RPAREN SEMICOLON;
for_statement → FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
                                                    for_update_opt RPAREN statement;
for_statement_no_short_if → FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
                                            for_update_opt RPAREN statement_no_short_if;
for_init_opt → for_init | λ;
for_init → statement_expression_list | local_variable_declaration;
for_update_opt → for_update | λ;
for_update → statement_expression_list;
statement_expression_list → statement_expression
    | statement_expression_list COMMA statement_expression;
identifier_opt → identifier | λ;
break_statement →BREAK identifier_opt SEMICOLON;
continue_statement → CONTINUE identifier_opt SEMICOLON;
return_statement → RETURN expression_opt SEMICOLON;
throw_statement → THROW expression SEMICOLON;
synchronized_statement → SYNCHRONIZED LPAREN expression RPAREN block;
try_statement → TRY block catches | TRY block catches_opt finally;
catches_opt → catches | λ;
```

```
catches → catch_clause | catches catch_clause;
catch_clause → CATCH LPAREN formal_parameter RPAREN block;
finally → FINALLY block;

// 19.12) Expressions
primary →  primary_no_new_array | array_creation_init | array_creation_uninit;
primary_no_new_array →
      literal | THIS |  LPAREN expression RPAREN | class_instance_creation_expression
    | field_access | method_invocation | array_access | primitive_type DOT CLASS
    | VOID DOT CLASS | array_type DOT CLASS | name DOT CLASS | name DOT THIS;
class_instance_creation_expression →
      NEW class_or_interface_type LPAREN argument_list_opt RPAREN
    | NEW class_or_interface_type LPAREN argument_list_opt RPAREN class_body
    | primary DOT NEW identifier LPAREN argument_list_opt RPAREN
    | primary DOT NEW identifier LPAREN argument_list_opt RPAREN class_body
    | name DOT NEW identifier LPAREN argument_list_opt RPAREN
    | name DOT NEW identifier LPAREN argument_list_opt RPAREN class_body;
argument_list_opt → argument_list | λ;
argument_list → expression
    | argument_list COMMA expression;
array_creation_uninit →
      NEW primitive_type dim_exprs dims_opt
    | NEW class_or_interface_type dim_exprs dims_opt;
array_creation_init → NEW primitive_type dims array_initializer
    | NEW class_or_interface_type dims array_initializer;
dim_exprs → dim_expr | dim_exprs dim_expr;
dim_expr → LBRACK expression RBRACK;
dims_opt → dims | λ;
dims → LBRACK RBRACK | dims LBRACK RBRACK;
field_access → primary DOT identifier
    | SUPER DOT identifier | name DOT SUPER DOT identifier;
method_invocation → name LPAREN argument_list_opt RPAREN
    | primary DOT identifier LPAREN argument_list_opt RPAREN
    | SUPER DOT identifier LPAREN argument_list_opt RPAREN
    | name DOT SUPER DOT identifier LPAREN argument_list_opt RPAREN;
array_access → name LBRACK expression RBRACK
    | primary_no_new_array LBRACK expression RBRACK
    | array_creation_init LBRACK expression RBRACK;
postfix_expression → primary | name
    | postincrement_expression | postdecrement_expression;
postincrement_expression → postfix_expression PLUSPLUS;
postdecrement_expression → postfix_expression MINUSMINUS;
unary_expression → preincrement_expression
```

```
    | predecrement_expression | PLUS unary_expression
    | MINUS unary_expression | unary_expression_not_plus_minus;
preincrement_expression → PLUSPLUS unary_expression;
predecrement_expression → MINUSMINUS unary_expression;
unary_expression_not_plus_minus → postfix_expression
    | COMP unary_expression | NOT unary_expression | cast_expression;
cast_expression → LPAREN primitive_type dims_opt RPAREN unary_expression
    | LPAREN expression RPAREN unary_expression_not_plus_minus
    | LPAREN name dims RPAREN unary_expression_not_plus_minus;
multiplicative_expression → unary_expression
    | multiplicative_expression MULT unary_expression
    | multiplicative_expression DIV unary_expression
    | multiplicative_expression MOD unary_expression;
additive_expression → multiplicative_expression
    | additive_expression PLUS multiplicative_expression
    | additive_expression MINUS multiplicative_expression;
shift_expression →additive_expression
    | shift_expression LSHIFT additive_expression
    | shift_expression RSHIFT additive_expression
    | shift_expression URSHIFT additive_expression;
relational_expression → shift_expression
    | relational_expression LT shift_expression
    | relational_expression GT shift_expression
    | relational_expression LTEQ shift_expression
    | relational_expression GTEQ shift_expression
    | relational_expression INSTANCEOF reference_type;
equality_expression → relational_expression
    | equality_expression EQEQ relational_expression
    | equality_expression NOTEQ relational_expression;
and_expression → equality_expression
    | and_expression AND equality_expression;
exclusive_or_expression → and_expression
    | exclusive_or_expression XOR and_expression;
inclusive_or_expression → exclusive_or_expression
    | inclusive_or_expression OR exclusive_or_expression;
conditional_and_expression → inclusive_or_expression
    | conditional_and_expression ANDAND inclusive_or_expression;
conditional_or_expression → conditional_and_expression
    | conditional_or_expression OROR conditional_and_expression;
conditional_expression → conditional_or_expression
    | conditional_or_expression QUESTION expression COLON conditional_expression;
assignment_expression → conditional_expression | assignment ;
assignment →  postfix_expression assignment_operator assignment_expression;
```

```
assignment_operator → EQ | MULTEQ | DIVEQ |  MODEQ | PLUSEQ |    MINUSEQ
    | LSHIFTEQ | RSHIFTEQ | URSHIFTEQ | ANDEQ | XOREQ | OREQ;
expression_opt → expression | λ;
expression → assignment_expression;
constant_expression → expression ;
identifier → IDENTIFIER;
```

Apêndice D

Classificação dos não terminais da gramática de Java

Opcionais	#var_init	#for_update_opt
	#import_declarations_opt	#catches_opt
	#modifiers_opt	#super_opt
	#interfaces_opt	#dims_opt
	#formal_parameter_list_opt	#throws_opt
	#extends_interfaces_opt	#final_modifier_opt
	#block_statements_opt	#for_init_opt
	#package_declaration_opt	#identifier_opt
	#type_declarations_opt	#argument_list_opt
	#class_body_declarations_opt	#expression_opt
	#interface_member_declarations_opt	

Listas Uniformes	#import_declarations	#type_declarations
	#class_body_declarations	#modifiers
	#interface_member_declarations	#block_statements
	#switch_block_statement_groups	#switch_labels
	#catches	#dim_exprs

Listas não uniformes	#name	#interface_type_list
	#variable_declarators	#variable_declarator_id
	#method_declarator	#formal_parameter_list
	#class_type_list	#inclusive_or_expression
	#variable_initializers	#statement_expression_list
	#argument_list	#conditional_or_expression
	#unary_expression	#multiplicative_expression
	#additive_expression	#relational_expression
	#shift_expression	#equality_expression
	#and_expression	#exclusive_or_expression
	#extends_interfaces	#conditional_and_expression
	#dims	#conditional_expression

Simples	#goal	#literal
	#type	#primitive_type
	#numeric_type	#integral_type
	#floating_point_type	#reference_type
	#class_type	#class_or_interface_type
	#interface_type	#array_type
	#qualified_name	#compilation_unit
	#package_declaration	#single_type_import_declaration
	#import_declaration	#type_import_on_demand_declaration
	#type_declaration	#modifier
	#class_declaration	#constructor_declarator
	#interfaces	#class_body_declaration
	#class_body	#class_member_declaration
	#field_declaration	#variable_declarator
	#variable_initializer	#method_declaration
	#method_header	#formal_parameter
	#throws	#explicit_constructor_invocation
	#static_initializer	#constructor_declaration
	#super	#constructor_body
	#method_body	#interface_declaration
	#interface_body	#interface_member_declaration
	#constant_declaration	#abstract_method_declaration
	#array_initializer	#block
	#block_statement	#local_variable_declaration_statement
	#statement	#local_variable_declaration
	#statement_no_short_if	#statement_without_trailing_substatement
	#empty_statement	#labeled_statement_no_short_if
	#labeled_statement	#expression_statement
	#switch_label	#if_then_else_statement
	#if_then_statement	#if_then_else_statement_no_short_if
	#switch_statement	#switch_block_statement_group
	#switch_block	#statement_expression
	#while_statement	#while_statement_no_short_if
	#do_statement	#for_statement
	#for_init	#for_update
	#break_statement	#continue_statement
	#return_statement	#class_instance_creation_expression
	#try_statement	#synchronized_statement
	#catch_clause	#finally
	#primary	#primary_no_new_array
	#throw_statement	#array_creation_init
	#array_creation_uninit	#unary_expression_not_plus_minus
	#field_access	#method_invocation
	#array_access	#postincrement_expression
	#postfix_expression	#postdecrement_expression
	#dim_expr	#preincrement_expression
	#cast_expression	#predecrement_expression
	#assignment	#assignment_expression
	#expression	#assignment_operator
	#constant_expression	#for_statement_no_short_if
	#identifier	

Apêndice E

Gramática de SCQL

```
terminal SELECT, FROM, WHERE, AS, TABLE, VIEW;
terminal TREE, FILTERED, BY, FOCUSED, ON, EXCLUDING, OUTER, MJTYPE;
terminal BOOLEAN, BYTE, SHORT, INT, LONG, CHAR, FLOAT, DOUBLE;
terminal LBRACK, RBRACK, IDENTIFIER, DOT;
terminal SEMICOLON, MULT, COMMA, LBRACE, RBRACE, EQ, LPAREN, RPAREN, COLON;
terminal VOID, THIS, SUPER, CLASS, NEW, PLUSPLUS, MINUSMINUS;
terminal PLUS, MINUS, COMP, NOT, DIV, MOD, LSHIFT, RSHIFT, URSHIFT;
terminal LT, GT, LTEQ, GTEQ, INSTANCEOF, EQEQ, NOTEQ;
terminal AND, XOR, OR, ANDAND, OROR, QUESTION;
terminal MULTEQ, DIVEQ, MODEQ, PLUSEQ, MINUSEQ;
terminal LSHIFTEQ, RSHIFTEQ, URSHIFTEQ, ANDEQ, XOREQ, OREQ;
terminal INTEGER_LITERAL, FLOATING_POINT_LITERAL;
terminal DOUBLE_PRECISION_LITERAL, LONG_LITERAL;
terminal BOOLEAN_LITERAL, CHARACTER_LITERAL;
terminal STRING_LITERAL, NULL_LITERAL;

non terminal goal, literal, primitive_type, numeric_type, integral_type;
non terminal reference_type, floating_point_type, class_or_interface_type;
non terminal array_type, name, qualified_name;
non terminal variable_initializer, array_initializer;
non terminal VariableInitializers variable_initializers, primary, primary_no_new_array;
non terminal argument_list_opt, argument_list, array_creation_init, array_creation_uninit;
non terminal dim_exprs, dim_expr, dims_opt, dims, field_access, method_invocation;
non terminal array_access, postfix_expression, postincrement_expression;
non terminal postdecrement_expression, unary_expression, unary_expression_not_plus_minus;
non terminal preincrement_expression, predecrement_expression, cast_expression;
non terminal multiplicative_expression, additive_expression, shift_expression;
non terminal relational_expression, equality_expression, and_expression;
non terminal exclusive_or_expression, inclusive_or_expression, conditional_and_expression;
non terminal conditional_or_expression, conditional_expression, assignment_expression;
```

```
non terminal assignment, assignment_operator, expression, identifier;
non terminal class_instance_creation_expression, query, selection, view;
non terminal typed_fields, fields, resultset_list, typed_field, field;
non terminal resultset, where, tree, filter_opt, exc_focus_filter_opt;
non terminal excluding_opt, focus_opt, exc_focus_list;

start with goal;

goal → query | expression ;
query → selection| view ;
selection → SELECT fields:f FROM LPAREN resultset_list:rs RPAREN where
            | SELECT MULT FROM LPAREN resultset_list:rs RPAREN where
             | SELECT fields:f FROM LPAREN resultset_list:rs RPAREN
             | SELECT MULT FROM LPAREN resultset_list:rs RPAREN;
where → WHERE expression;
fields → fields COMMA field | field;
field → IDENTIFIER AS IDENTIFIER | IDENTIFIER;
typed_fields → typed_fields COMMA typed_field    | typed_field;
typed_field → MJTYPE IDENTIFIER filter_opt;
filter_opt → FILTERED BY expression | λ;
exc_focus_filter_opt → IDENTIFIER FILTERED BY expression:e |;
resultset_list → resultset_list COMMA resultset | resultset;
resultset → selection| view;
excluding_opt → EXCLUDING exc_focus_list| λ;
focus_opt → FOCUSED ON exc_focus_list | λ;
exc_focus_list →  MJTYPE exc_focus_filter_opt
    | exc_focus_listss COMMA MJTYPE exc_focus_filter_opt;
view → VIEW TREE tree focus_opt excluding_opt AS TABLE LPAREN
                               typed_fieldss RPAREN where
    | VIEW TREE tree focus_opt excluding_opt AS TABLE LPAREN typed_fieldss RPAREN;
tree → LBRACK STRING_LITERAL RBRACK | IDENTIFIER ;

// 19.3) Lexical Structure.
literal → LONG_LITERAL | INTEGER_LITERAL | FLOATING_POINT_LITERAL
    | DOUBLE_PRECISION_LITERAL | BOOLEAN_LITERAL
    | CHARACTER_LITERAL | STRING_LITERAL | NULL_LITERAL;

// 19.4) Types, Values, and Variables
primitive_type → numeric_type | BOOLEAN;
numeric_type → integral_type | floating_point_type;
integral_type → BYTE | SHORT | INT | LONG | CHAR;
floating_point_type → FLOAT | DOUBLE;
reference_type → class_or_interface_type
```

```
    | array_type;
class_or_interface_type → name;
array_type → primitive_type dims | name dims;

// 19.5) Names
name → identifier | name qualified_name;
qualified_name → DOT identifier;
variable_initializer → expression | array_initializer;

// 19.10) Arrays
array_initializer → LBRACE variable_initializers COMMA RBRACE
    | LBRACE variable_initializers RBRACE
    | LBRACE COMMA RBRACE | LBRACE RBRACE;
variable_initializers → variable_initializer
    | variable_initializers COMMA variable_initializer;
// 19.12) Expressions
primary →    primary_no_new_array | array_creation_init
    | array_creation_uninit;
primary_no_new_array → literal | OUTER | LPAREN expression RPAREN | field_access
    | method_invocation | array_access | class_instance_creation_expression
    | primitive_type DOT CLASS | VOID DOT CLASS | array_type DOT CLASS
    | name DOT CLASS | name DOT THIS;
class_instance_creation_expression →
      NEW class_or_interface_type LPAREN argument_list_opt RPAREN
    | primary DOT NEW identifier LPAREN argument_list_opt RPAREN;
argument_list_opt → argument_list | λ;
argument_list → expression | argument_list COMMA expression;
array_creation_uninit → NEW primitive_type dim_exprspr dims_opt
    | NEW class_or_interface_type dim_exprspr dims_opt;
array_creation_init → NEW primitive_type dims array_initializer
    | NEW class_or_interface_type dims array_initializer ;
dim_exprs → dim_expr | dim_exprs dim_expr;
dim_expr → LBRACK expression RBRACK;
dims_opt → dims | λ;
dims →  LBRACK RBRACK | dims LBRACK RBRACK;
field_access → primary DOT identifier | SUPER DOT identifier
    | name DOT SUPER DOT identifier;
method_invocation → name LPAREN argument_list_opt RPAREN
    | primary DOT identifier LPAREN argument_list_opt RPAREN
    | SUPER DOT identifier LPAREN argument_list_opt  RPAREN
    | name DOT SUPER DOT identifier LPAREN argument_list_opt RPAREN;
array_access → name LBRACK expression RBRACK
    | primary_no_new_array LBRACK expression RBRACK
```

```
    | array_creation_init LBRACK expression RBRACK;
postfix_expression → primary | name | postincrement_expression
    | postdecrement_expression;
postincrement_expression → postfix_expression PLUSPLUS;
postdecrement_expression → postfix_expression MINUSMINUS;
unary_expression → preincrement_expression | predecrement_expression
    | PLUS unary_expression | MINUS unary_expression
    | unary_expression_not_plus_minus;
preincrement_expression → PLUSPLUS unary_expression;
predecrement_expression → MINUSMINUS unary_expression;
unary_expression_not_plus_minus → postfix_expression
    | COMP unary_expression | NOT unary_expression | cast_expression;
cast_expression → LPAREN primitive_type dims_opt RPAREN unary_expression
    | LPAREN expression RPAREN unary_expression_not_plus_minus
    | LPAREN name dims RPAREN unary_expression_not_plus_minus;
multiplicative_expression → unary_expression
    | multiplicative_expression MULT unary_expression
    | multiplicative_expression DIV unary_expression
    | multiplicative_expression MOD unary_expression;
additive_expression → multiplicative_expression
    | additive_expression PLUS multiplicative_expression
    | additive_expression MINUS multiplicative_expression;
shift_expression → additive_expression
    | shift_expression LSHIFT additive_expression
    | shift_expression RSHIFT additive_expression
    | shift_expression URSHIFT additive_expression;
relational_expression → shift_expression
    | relational_expression LT shift_expression
    | relational_expression GT shift_expression
    | relational_expression LTEQ shift_expression
    | relational_expression GTEQ shift_expression
    | relational_expression INSTANCEOF reference_type;
equality_expression → relational_expression
    | equality_expression EQEQ relational_expression
    | equality_expression NOTEQ relational_expression;
and_expression → equality_expression
    | and_expression AND equality_expression;
exclusive_or_expression → and_expression
    | exclusive_or_expression XOR and_expression;
inclusive_or_expression → exclusive_or_expression
    | inclusive_or_expression OR exclusive_or_expression;
conditional_and_expression → inclusive_or_expression
    | conditional_and_expression ANDAND inclusive_or_expression;
```

```
conditional_or_expression → conditional_and_expression
    | conditional_or_expression OROR conditional_and_expression;
conditional_expression → conditional_or_expression
    | conditional_or_expression QUESTION expression conditional_expression;
assignment_expression → conditional_expression | assignment;
assignment →    postfix_expression assignment_operator assignment_expression;
assignment_operator →     EQ | MULTEQ | DIVEQ | MODEQ | PLUSEQ | MINUSEQ
    | LSHIFTEQ | RSHIFTEQ | URSHIFTEQ | ANDEQ | XOREQ | OREQ ;
expression → assignment_expression;
identifier → IDENTIFIER;
```