Sobre Semântica de Ações

Semântica de Ações é um formalismo criado com o intuito de tornar a especificação formal de linguagens de programação e sistemas computacionais em geral mais fácil de ser realizada e entendida. Descrições realizadas por meio de Semântica de Ações, de um modo geral, são claras, modulares e facilmente extensíveis.

A principal vantagem da Semântica de Ações sobre outros formalismos criados para descrever a semântica de linguagens de programação como por exemplo, a Semântica Denotacional e a Semântica Estrutural, é a sua legibilidade. A notação utilizada pela semântica de ações muito se assemelha com a linguagem natural sem, contudo, perder nada do seu formalismo.

Várias linguagens já foram especificadas em Semântica de Ações com o intuito de mostrar seu valor para trabalhos realistas. Entre elas podemos mencionar Pascal, Java e Standard ML.

Capítulo 1

Descrição Informal

JaTS (acrônimo para Java Transformation System) é uma linguagem criada com o objetivo de especificar transformações para programas na linguagem Java. Antes de entrar em maiores detalhes sobre como uma transformação é realizada, é importante explicar alguns conceitos necessários para entender o funcionamento do sistema.

Transformações em JaTS são construídas numa linguagem que consiste de Java acrescido de construções JaTS e estas têm por objetivo possibilitar o casamento de programas. A construção JaTS mais comum é a variável JaTS e, para o JaTS, uma variável é um string qualquer precedido pelo caractere "$".

Uma transformação JaTS é composta por duas partes: Um lado esquerdo e um lado direito. O lado esquerdo é casado com o programa fonte em Java que se deseja transformar, de modo que ambos têm que ter a mesma estrutura sintática. O lado direito é o esqueleto do programa que será o resultado da transformação. A aplicação de uma transformação escrita em JaTS a um programa Java consiste de três etapas: casamento, execução e substituição.

A primeira etapa da aplicação de uma transformação é o casamento de programas. Durante esta etapa, é realizado um casamento de padrões entre as árvores sintáticas do lado esquerdo da transformação e do programa fonte. As duas árvores são percorridas com o fim de verificar se elas são compatíveis. Por compatíveis, entende-se que, para cada nó da árvore do programa fonte, haverá ou um nó idêntico ou um nó representado uma variável na árvore do lado esquerdo da transformação. O lado esquerdo de uma transformação JaTS não pode possuir nenhuma estrutura executável.

Na etapa de execução, o lado direito da transformação é varrido à procura de estruturas executáveis. Por “executáveis”, entede-se que elas podem ser executadas como um programa Java comum. Cada vez que uma dessas estruturas é executada, o resultado obtido é colocado em seu lugar na árvore sintática do programa correspondente ao lado direito da transformação, de modo que, ao fim da etapa de execução, esse programa não possui mais nenhuma estrutura executável.

A segunda etapa da aplicação de uma transformação é a subsituição. Esta etapa recebe como entrada o lado direito da transformação e o conjunto resultado do casamento.

Durante a subsituição, o lado direito da transformação, sem nenhuma estrutura executável, é percorrido. Cada vez que uma variável é encontrada, é verificado se esta

variável está presente no conjunto resultado do casamento. Se estiver, a variável é subsituída por este valor.

Como a sintaxe de JaTS é um super-conjunto da de Java, não julgamos pertinente descrever cada uma das construções da mesma forma que elas estariam descritas num texto sobre o Java. Ao invés disso, descreveremos os aspectos da sintaxe de JaTS que têm a ver com casamento, execução e substituição. Para uma descrição de sintaxe e da semântica de Java, ver **[2]**.

1.1 Identificadores, Nomes, Variáveis e Expressões Executáveis

Sintaxe

Identifier = Word | Variable | ExecutableExpression.

Word = [[alpha | (alpha | digit)*]].

Variable = '$' [[alpha | (alpha | digit)*]].

ExecutableExpression = [[("[[" MethodCall "]]") | ("[[" Variable ":" MethodCall "]]")]].

Name = Identifier | [[Name "." Identifier]] | ExecutableExpression | Variable.

Names = < Name < "," Name >* > | [["list" Variable]] | [["list" ExecutableExpression]].

Semântica

Casamento:

- O casamento entre dois identificadores pode resultar simplesmente num sucesso (os dois identificadores são iguais) ou no *binding* de uma variável ao string correspondente a um identificador. Na nossa modelagem, quase todos os *bindings* são feitos de uma variável a um string ou conjunto de strings.
- Expressões executáveis não podem ser casadas.
- O casamento entre dois nomes consiste em casar um a um e na mesma ordem em que aparecem nos nomes, os identificadores que os compõem. Um casamento entre dois nomes também pode ser entre um nome e uma variável. Neste caso o resultado é o *binding* da variável para o string correspondente ao nome.
- O casamento entre duas listas de nomes funciona de forma idêntica ao casamento entre dois nomes, exceto que, ao invés de termos nomes compostos por identificadores, temos listas de nomes compostas por nomes. A única diferença está na sintaxe que descreve uma variável a ser casada com uma lista de nomes. A variável é precedida pela palavra chave "list".
- O casamento entre duas variáveis não é permitido.

Execução:

- Executar uma palavra resulta na própria palavra sendo *given*.
- Executar uma variável resulta na própria variável sendo *given*.
- A execução de uma estrutura executável consiste em executar as invocações de métodos presentes dentro dos delimitadores de estrutura executável (“[[“ “]]”) como elas seriam executadas por um interpretador Java. Existem dois tipos de estrutura executável. A primeira realiza uma única invocação de método e retorna o resultado dessa invocação. No segundo tipo de estrutura executável, várias invocações seqüenciais de métodos são possíveis e o resultado retornado é o bindable que estiver mapeado à variável que aparece na declaração (logo depois do colchete duplo).
- Executar um nome resulta em executar cada um dos identificadores que o compõem.
- Executar uma lista de nomes resulta em executar cada um dos nomes que o compõem.

Substituição:

- A substituição em qualquer estrutura que não seja uma variável resulta num string representando a própria estrutura sendo retornado. A saída da etapa de substituição de um programa JaTS consiste de um string representando todo o resultado da substituição.
- A substituição de uma variável resulta sempre no string que estiver mapeado a ela, caso haja algum, ou na própria variável, caso ela não esteja mapeada a nada. Há ocasiões em que a variável pode estar mapeada a um conjunto de strings. Nestes casos, o resultado da substituição de uma variável corresponde ao string resultante da concatenação de todos os strings que fazem parte do conjunto que está a ela mapeado.
- A substituição de um nome resulta no string correspondente à concatenção dos strings resultantes da substituição dos identificadores que o compõem.
- Listas de nomes são substituídas da mesma maneira que nomes, só que ao invés de serem compostas por identificadores são compostas por nomes.

Exemplos

Casamento:

- O casamento dos identificadores “ClassBody” e “ClassBody” resulta em sucesso e nenhum *bind* gerado.
- O casamento entre os nomes “$a.bad.cat” e “air.bad.cat” resulta no bind $a→“air” ,onde “air” é um string.
- O casamento entre “list $l” e “Runnable, Clonnable” resulta no bind $l→“Runnable,Clonable”, onde “Runnable,Clonable” é um string.

Execução:

- A execução de “Relógio” resulta em “Relógio”, onde “Relógio” é uma árvore sintática.

- A execução de “$a” resulta em “$a”, onde “$a” é uma árvore sintática.
- A execução de [[$m.getName()]] tem como resultado o objeto retornado pela invocação do método getName() no objeto que estiver mapeado à variável $m.

Substituição:

- A substituição de uma variável qualquer $a consiste em fazer com que a string mapeada a $a ocupe seu lugar no programa gerado pela etapa de substituição.
- A substituição do identificador “ClassName” fornece como resultado o string desse identificador.

1.2 Tipos

Sintaxe

- Type = PrimitiveType | Name | Variable | ExecutableExpression.
- PrimitiveType = “boolean” | “char” | “byte” | “short” | “int” | “long” | “float” | “double”.

Semântica

Casamento:

- Para que dois tipos casem, ou os dois são tipos primitivos, ou os dois são tipos representados por classes (denotados por Name), ou um é uma variável e o outro é de qualquer um dos dois tipos.
- No primeiro caso , o casamento consiste simplesmente de ver se os dois tipos representam o mesmo tipo primitivo (se os dois são “boolean” ou se os dois são “char”, etc).
- No segundo, vale o que já foi mencionado para o casamento entre dois nomes.
- No terceiro caso, o resultado do casamento é um *binding* da variável para o string correspondente ao tipo que está sendo mapeado.

Execução:

- As regras de execução para tipos são as mesmas já mencionadas para identificadores, nomes e listas de nomes.

Substituição:

- As regras de substituição para tipos também são as mesmas já mencionadas para identificadores, nomes e listas de nomes.

Exemplos

Casamento:

- O casamento entre os tipos primitivos “int” e “int” resulta em sucesso e nenhum *bind* gerado.

- O casamento entre o tipo primtivo “char” e o classe “String” resulta em *escape.*
- O casamento entre a variável $i e o tipo primitivo “long” fornece como resultado um *bind* $i → “long”, onde “long” é um string.

Execução e Substituição:

- Os exemplos para execução e substituição apresentados na seção anterior descrevem um comportamento que também é o manifestado no caso de execução e substituição de tipos. Por conseqüência, nenhum novo exemplo será apresentado nesta secção. A partir de agora, sempre que esse for o caso, omitiremos o subtítulo correspondente ao tipo do exemplo (Casamento, Substituição ou Execução).

1.3 Expressões e Invocação de Métodos

Sintaxe

- MethodCall = [[Name “(“ Arguments$^{?}$ “)”]] |
  [[Expression “.” Identifier “(“ Arguments$^{?}$ “)”]].
- Arguments = < Expression <Expression “,” Expression >*>.
- Expression = MethodCall | Literal.
- Literal = StringLiteral | IntegerLiteral.
- StringLiteral = [[‘ “ ’ InputCharacter ‘ ” ’]].
- IntegerLiteral = [[(‘0’ | non-zero-digit digit^{+})]] .

Semântica

Casamento:

- Não há semântica definida ainda em JaTS para casamento entre expressões ou entre invocações de métodos.

Execução:

- Até o presente momento, expressões e invocações de métodos só aparecem em JaTS dentro de blocos executáveis. Invocações de métodos são, até o presente momento, o único tipo de construção Java que pode ser executado em JaTS. Sua semântica é a mesma da execução de um método da linguagem Java.

Substituição:

- Não existe ainda em JaTS uma semântica definida para substituição de expressões ou invocações de métodos. Embora uma espécie de substituição seja realizada dentro de ExecutableExpression´s, ela não funciona da mesma maneira que a substituição que descreveremos ao longo do documento.

Exemplos

Execução:

- .[[$a : $a.setFieldName(“método”)]] : A execução dessa expressão invocará, no objeto mapeado à variável $a, o método setFieldName, passando como argumento o literal string “método”. O método é invocado da mesma forma que seria se o estivéssemos fazendo usando a linguagem Java.

1.4 Declarações

Sintaxe

- Modifiers = [[Modifier*]] | “mods” Variable.
- Modifier = “public” | “private” | “protected” | “static” | “abstract” | “final” | “native” | “synchronized” | “transient” | “volatile”.
- ClassDeclaration = [[Modifiers “class” Identifier “extends” Name “implements” Names ClassBody]].
- ClassBody = [[“{” ClassBodyDeclaration* “}”]].
- ClassBodyDeclaration = FieldDeclaration | MethodDeclaration | ConstructorDeclaration | ForallDeclaration | FieldsVarDeclaration | MethodsVarDeclaration | ConstructorsVarDeclaration.
- MethodDeclaration = [[Modifiers ResultType Identifier “(” FormalParameters? “)” (“throws” Names)? MethodBody]].
- ResultType = Type | “void”.
- MethodBody = Block | “;”.
- Block = .
- MethodsVarDeclaration = “methods” Variable “;”.
- FormalParameters =< FormalParameter < “,” FormalParameter >*> | [[“list” Variable]] | [[“list” ExecutableExpression]] .
- FormalParameter = [[Type Identifier]] | Variable | ExecutableExpression.
- ConstructorDeclaration = [[Modifiers Identifier “(” FormalParameters? “)” (“throws” Names)? ConstructorBody]].
- ConstructorBody = [[“{“ ExplicitConstructorCall? BlockStatement*]].
- ExplicitConstructorCall = .
- BlockStatement = .
- ConstructorsVarDeclaration = “constructors” Variable “;”.
- FieldDeclaration = [[Modifiers Type FieldDeclarators]].

- FieldDeclarators = < FieldDeclarator < "," FieldDeclarator > .
- FieldDeclarator = [[Identifier < "=" VariableInitializer>?]].
- VariableInitializer = .
- FieldsVarDeclaration = "fields" Variable ";".
- ForallDeclaration = "forall" Variable "in" Variable "{" ForallBlockDeclaration* "}".
- ForallBlockDeclaration = MethodDeclaration | FieldDeclaration | ConstructorDeclaration | ExecutableExpression.
- CompilationUnit = [[PackageDeclaration? ImportDeclaration* TypeDeclaration*]].
- PackageDeclaration = [["package" Name ";"]]
- ImportDeclaration = [["import" Name < "." "*" >? ";"]]
- TypeDeclaration = ClassDeclaration | InterfaceDeclaration | [[";"]].
- InterfaceDeclaration = .

Semântica

Casamento:

- Todas formas de declaração que foram herdadas da linguagem Java funcionam da mesma forma no que concerne a casamento. O resultado do casamento entre duas declarações quaisquer que sejam de mesmo tipo (ex: Uma MethodDeclaration e outra MethodDeclaration) resulta no casamento das estruturas que às compõem. Por exemplo, o resultado do casamento entre duas declarações de pacote (PackageDeclaration) "package a.b.c" e "package a.b.$c" é o resultado do casamento entre os Name´s "a..b.c" e "a.b.$c".
- Declarações do tipo ForallDeclaration não podem ser casadas. Elas são declarações executáveis e nenhum programa JaTS destinado a ser casado pode conter declarações executáveis.
- As estruturas MethodsVarDeclaration, FieldsVarDeclaration e ConstructorsVarDeclaration apresentam as características de casamento mais interessantes da linguagem JaTS. O casamento das declarações de métodos entre um programa fonte em Java chamado *Source* e o lado esquerdo de uma transformação JaTS chamado *Trans*, que possui uma declaração MethodsVarDeclaration chamada $mds no seu corpo, ocorre da seguinte maneira:

  1 Tenta-se casar todas as declarações de métodos existentes em *Trans* com as declarações de métodos existentes em *Source*. Se isso acontecer, a execução procede. Se não for possível, para algum método de *Trans*, encontrar um método em *Source* com o qual seja possível casar, a execução é interrompida (*escape*). É importante ressaltar que só é obrigatório que sejam

casados individualmente todos os métodos de *Trans*. É permitido que vários métodos "sobrem" no corpo de *Source* **portanto que haja uma declaração MethodsVarDeclaration em *Trans***. Se "sobrarem" métodos no corpo do programa fonte mas não houver uma declaração MethodsVarDeclaration no corpo do lado esquerdo da transformação, a execução também é interrompida.

2. Uma vez cumprida a primeira etapa, varre-se o corpo de *Source* de modo a encontrar todos os métodos que "sobraram" sem ser casados. Esses métodos são colocados dentro de um conjunto. Vale ressaltar que o que é colocado dentro do conjunto são **strings** correspondentes às declarações dos métodos.
3. Por fim, é criado um *bind* da variável $mds para o conjunto de strings resultado do passo 2.

- Declarações FieldsVarDeclaration e ConstructorsVarDeclaration funcionam da mesma maneira que as declarações MethodsVarDeclaration, só que aplicadas a atributos e construtores, respectivamente.

Execução:

- O mesmo que foi dito para casamento se aplica à execução de declarações. Ela funciona de forma composicional da mesma maneira como já foi visto para nomes e listas de nomes.
- Numa declaração "forall" $m "in" $ms "{" ForallBlockDeclaration* "}", $m é o nome de uma variável que será usada como *placeholder* no decorrer da execução. $ms é o nome de uma variável que está mapeada a um conjunto de declarações, seja elas métodos, atributos ou construtores (como no caso de uma declaração MethodsVarDeclaration). Uma ForallBlockDeclaration pode ser uma declaração de método, atributo, construtor ou estrutura executável.
- Uma ForallDeclaration funciona da seguinte maneira: Para cada método/atributo/construtor existente em $ms, $m é usado para fazer referência ao método/construtor/atributo que estiver sendo alvo da execução num determinado momento. A execução propriamente dita consiste em executar cada uma das ForallBlockDeclaration´s da ForallDeclaration. Uma vez que todas as ForallBlockDeclaration´s tenham sido devidamente executadas (não possuam mais qualquer estrutura executável), elas são colocadas no corpo da classe à qual a ForallDeclaration pertence e esta última é então retirada do corpo dessa classe.

Substituição:

- A substituição é a etapa mais uniforme entre as três que compõem a aplicação de uma transformação JaTS. Uniforme no aspecto de que não há muita diferença entre a substituição num identificador ou numa declaração de método.

Exemplos

Casamento:

- O casamento do programa fonte

```
class ProgramExemplo {

  private int attr1;
```

```
    private boolean flag;

    public int meth1() { }
    public void meth2() { }
    public char meth3() throws IOException{ }

  }
```

com o lado esquerdo

```
  class ProgramExemplo {

    private $t attr1;
    fields $fds;

    public $t meth1() { }
    methods $mds;

  }
```

no fornece o seguinte conjunto resultado: `{ $t → "int", $fds → {"private boolean flag"}, $mds → { "public void meth2() {}", "public char meth3() throws IOException { }" } }`

Seguindo o exemplo passo a passo, primeiro o nome da classe do programa fonte ("ProgramaExemplo") é casado com o nome da classe do lado esquerdo da transformação, já que ambos são iguais (nenhum *bind* é gerado). Depois os modificadores da declaração de atributo "`private int attr1;` " são casados com os modificadores do primeiro atributo do lado esquerdo da transformação, já que os dois atributos têm como modificador apenas "`private`" (vale ressaltar que o fato do primeiro atributo do lado esquerdo casar justamente com o primeiro do lado direito é apenas uma questão de implementação). Depois disso, tenta-se casar os tipos dos dois atributos. Isso gera o *bind* `$t → int`, já que `$t` é uma variável. O casamento procede dessa maneira até que todas as declarações de atributos, métodos e construtores existentes no lado esquerdo da transformação tenham sido casados com algum atributo/método/construtor no programa fonte. Uma vez que isso tenha sido assegurado, o lado esquerdo da transformação é varrido a procura de declarações dos tipos FieldsVarDeclaration, MethodsVarDeclaration ou ConstructorsVarDeclaration. Se uma declaração do tipo FieldsVarDeclarationfor encontrada, ela é casada com os atributos do programa fonte que não foram casados a nenhum atributo no lado esquerdo da transformação. O mesmo se aplica aos outros dois tipos de declaração. O resultado dessa etapa do casamento aplicado ao nosso exemplo gera dois *binds*:

- `$fds → {"private boolean flag"}`
- `$mds → { "public void meth2() {}", "public char meth3() throws IOException { }" }`

É importante notar que as variáveis estão mapeadas a **conjuntos**, e não a elementos como, por exemplo, a variável `$t` mencionada anteriormente.

Execução:

- A execução da transformação cujo lado direito é

```
  class TransDir {
```

```
  private int attr1;
  fields $fds;

  public int attr1() { }

  forall $f in $fds {
    public [[ $f.getFieldType() ]] [[ $f.getFieldName() ]]() {}

  }

}
```

onde `$fds` está mapeada ao conjunto `{ “private int attr2”, “private String attr3” }` resulta no seguinte programa:

```
class TransDir {

  private int attr1;
  private int attr2
  private String attr3;

  public int attr1() { }
  public int attr2() { }
  public String attr3() { }
}
```

Como deve ter sido possível notar, o processo funciona como se estivéssemos usando a característica de reflexão da linguagem Java (interpretadores para JaTS escritos em Java de fato usam essa característica). No exemplo anterior, para cada atributo dentro do conjunto, são invocados dois métodos (`getFieldType()` e `getFieldName()`) que fazem parte da classe que implementa atributos num interpretador JaTS qualquer. Essa é a maneira mais comum de se empregar essa característica da linguagem. Nada nos impede, porém, de invocar um método qualquer classe padrão de Java ou de bibliotecas de terceiros.

Substituição:

- Dado o conjunto resultado da etapa de casamento `{ $n → “Exemplo”, $t → TipoDoExemplo, $p → “int par1”, $g → “ErradaException”}`, a substituição do programa

```
class $n {
  private $t attr3;
  public $t attr1($p) throws $g { }
  public $n() { }
}
```

resulta em

```
class Exemplo {
  private TipoDoExemplo attr3;
  public TipoDoExemplo attr1(int par) throws ErradaException { }
  public Exemplo() { }
}.
```

- A substituição do programa

```
class Classe {

  fields $fds;

  public Classe() { }
}
```

onde o resultado da etapa de casamento foi `$fds` → `{ “private int attr2”, “private String attr3” }` fornece como resultado o programa

```
class Classe {

  private int attr2;
  private String attr3;

  public Classe() { }
}
```

Capítulo 2

Sintaxe Abstrata

needs: **[Mosses 1992]/Data Notation/Characters, Java Action Semantics.**

grammar:

2.1.1 Identificadores, Nomes, Variáveis e Expressões Executáveis

- Identifier = Word | Variable | ExecutableExpression.
- Word = [[alpha | (alpha | digit)*]].
- Variable = '$' [[alpha | (alpha | digit)*]].
- ExecutableExpression = [[("[[" MethodCall "]]") | ("[[" Variable ":" MethodCall "]]")]].
- Name = Identifier | [[Name "." Identifier]] | ExecutableExpression | Variable.
- Names = < Name < "," Name >* > | [["list" Variable]] | [["list" ExecutableExpression]].

2.1.2 Tipos

needs: **Identifiers and Names**

- Type = PrimitiveType | Name | Variable | ExecutableExpression.
- PrimitiveType = "boolean" | "char" | "byte" | "short" | "int" | "long" | "float" | "double".

2.1.3 Expressões e Invocação de Métodos

needs: **Identifiers and Names, Types.**

- MethodCall = [[Name “(“ Arguments$^{?}$ “)”]] |
  [[Expression “.” Identifier “(“ Arguments$^{?}$ “)”]].
- Arguments = < Expression <Expression “,” Expression >*>.
- Expression = MethodCall | Literal.
- Literal = StringLiteral | IntegerLiteral.
- StringLiteral = [[‘ “ ’ InputCharacter ‘ ” ’]].

É necessário definir InputCharacter = *qualquer coisa que não seja aspas duplas ou barra invertida*.

- IntegerLiteral = [[(‘0’ | non-zero-digit digit^{+})]] .

Como foi possível notar, expressões ainda não recebem um tratamento adequado em JaTS. Até o presente momento elas são usadas apenas para realizar invocações de métodos dentro de expressões executáveis.

2.1.4 Declarações

needs: **Identifiers and Names, Types.**

- Modifiers = [[Modifier*]] | “mods” Variable.
- Modifier = “public” | “private” | “protected” | “static” | “abstract” | “final” |
  “native” | “synchronized” | “transient” | “volatile”.

2.1.4.1 Declarações de Classes

- ClassDeclaration = [[Modifiers “class” Identifier “extends” Name “implements”
  Names ClassBody]].
- ClassBody = [[“{” ClassBodyDeclaration* “}”]].
- ClassBodyDeclaration = FieldDeclaration | MethodDeclaration |
  ConstructorDeclaration | ForallDeclaration | FieldsVarDeclaration |
  MethodsVarDeclaration | ConstructorsVarDeclaration.

2.1.4.2 Declarações de Métodos

- MethodDeclaration = [[Modifiers ResultType Identifier “(” FormalParameters$^{?}$ “)” (“throws” Names)$^{?}$ MethodBody]].
- ResultType = Type | “void”.
- MethodBody = Block | “;”.
- Block = .

A linguagem JaTS, em seu atual estado, ignora o que está contido no corpo de métodos e construtores.

2.1.4.3 Variáveis Casando com um Conjunto de Métodos

- MethodsVarDeclaration = “methods” Variable “;”.

2.1.4.4 Parâmetros Formais

- FormalParameters =< FormalParameter < “,” FormalParameter >*> | [[“list” Variable]] | [[“list” ExecutableExpression]] .
- FormalParameter = [[Type Identifier]] | Variable | ExecutableExpression.

2.1.4.5 Declarações de Construtores

- ConstructorDeclaration = [[Modifiers Identifier “(” FormalParameters$^{?}$ “)” (“throws” Names)$^{?}$ ConstructorBody]].
- ConstructorBody = [[“{“ ExplicitConstructorCall$^{?}$ BlockStatement*]].
- ExplicitConstructorCall = .
- BlockStatement = .

2.1.4.6 Variáveis Casando com um Conjunto de Construtores

- ConstructorsVarDeclaration = “constructors” Variable “;”.

2.1.4.7 Declarações de Atributos

- FieldDeclaration = [[Modifiers Type FieldDeclarators]].
- FieldDeclarators = < FieldDeclarator < “,” FieldDeclarator > .
- FieldDeclarator = [[Identifier < “=” VariableInitializer>$^{?}$]].

- VariableInitializer = .

2.1.4.8 Variáveis Casando com um Conjunto de Atributos

- FieldsVarDeclaration = “fields” Variable “;”.

2.1.4.9 Declarações “Para todo”

- ForallDeclaration = “forall” Variable “in” Variable “{” ForallBlockDeclaration* “}”.
- ForallBlockDeclaration = MethodDeclaration | FieldDeclaration |
  ConstructorDeclaration | ExecutableExpression.

2.1.4.10 Unidades de Compilação (Programas)

- CompilationUnit = [[PackageDeclaration? ImportDeclaration*
  TypeDeclaration*]].
- PackageDeclaration = [[“package” Name “;”]]
- ImportDeclaration = [[“import” Name < “.” “*” >? “;”]]
- TypeDeclaration = ClassDeclaration | InterfaceDeclaration | [[“;”]].
- InterfaceDeclaration = .

Capítulo 3

Funções Semânticas

needs: **Java Action Semantics [1], Action Notation.**

3.1 Programas

3.1.1 Execução de um Programa

needs: **Java Action Semantics [1].**
introduces: run _, _, _ , has executable expressions, isJava.

- run _,_,_ :: CompilationUnit → CompilationUnit → CompilationUnit → action
  [escaping | binding | giving a string |completing].

(1) run [[Left:CompilationUnit, JavaSource:JavaCompilationUnit,
Right:CompilationUnit]] =
|||| check (has executable expressions(Left)) or check not (is java(Source))
|| then
||| escape
| or
||| check not (has executable expressions(Left) and is java(Source))
|| and then
||| match (Left, JavaSource)
hence
||| execute Right
|| then
||||| check (has executable expressions (the given syntax-tree))
|||| then
||||| escape
||| or
||||| check not (has executable expressions (the given syntax-tree))
|||| and then
|||||| replace the given syntax-tree
||||| then
|||||| give the given string

```
    | and
    || rebind.
```

- has executable expressions _ :: CompilationUnit → yielder [of a truth-value].

(2) has executable expressions C:CompilationUnit = .

A função has executable expressions _ recebe a árvore sintática de um programa JaTS como entrada e verifica se, dentro deste programa, existem estruturas executáveis (ExecutableExpression ou ForallDeclaration. Serão especificadas posteriormente) e, em caso positivo, retorna true. Caso contrário, false.

Não definimos a função porque, além de acreditarmos que seu comportamento é facilmente entendível, vemos comportamento similar, embora com fins diferentes, em várias das funções que serão descritas posteriormente, como execute e replace. Quisemos evitar uma repetição desnecessária.

- is java _ :: CompilationUnit → yielder [of a truth-value].

(3) is java = .

Essa função é necessária para garantir que não estamos tentando casar um programa JaTS com outro programa JaTS. O leitor deve pensar nela como um *parser* para a gramática da linguagem Java que retorna verdadeiro que se o programa estiver correto (onde correto também implica não possuir qualquer tipo de característica JaTS, pois tudo que é Java é também JaTS mas nem tudo que é JaTS é Java).

3.2 Matchings

needs: Ações Auxiliares para Matching

3.2.1 Identificadores e Nomes

introduces: match identifiers _,_, match names _,_ , match nameslists _,_ .

- match identifiers _,_ :: Identifier → Identifier → action [escaping | binding |completing] [using current binds].

(1) match identifiers I_1:Word, I_2:Word =
```
    | check token of I1 is token of I2 then rebind
    or
    | escape.
```

(2) match identifiers V_1:Variable, I_1:Word =
```
    || check (already-matched(token of V))
    | and then
    ||| give the structure mapped to the Variable V
    || then
    ||| furthermore (match identifiers (the given syntax-tree, I1))
```

or
| furthermore (bind token of V to token of I).

- match names _,_ :: Name → Name → action[escaping | binding | completing]
  [using current binds].

(3) match names I_1:Word, I_2:Word =
furthermore (match identifiers (I_1, I_2)).

(4) match names [[I_1:Identifier "." N_1:Name, I_2:Identifier "." N_2:Name]] =
| furthermore (match identifiers(I_1, I_2))
hence
| furthermore (match names(N_1, N_2)).

(5) match names [[⟨ ⟩, N_1:Name]]= escape.

(6) match names [[N_1:Name, ⟨ ⟩]]= escape.

(7) match names ⟨ ⟩, ⟨ ⟩ = complete.

(8) match names [[V_1:Variable, N:Name]]=
|| check (already-matched(token of V))
| and then
||| give the structure mapped to the variable V
|| then
||| furthermore (match names (the given syntax-tree, N))
or
| furthermore (bind token of V to the string of N).

- match nameslists _,_ :: Names → Names → action [escaping | binding |
  completing] [using current binds].

(9) match namelists [[N_1:Name, N_2:Name]] =
furthermore (match names (N_1, N_2)).

(10) match [[N_1:Name "," NS_1:Names, N_2:Name "," NS_2:Names]] =
| furthermore (match names (N_1, N_2))
hence
| furthermore (match nameslists (NS_1, NS_2)).

(11) match nameslists [[⟨ ⟩, NS_1:Names]] = escape.

(12) match namelists [[NS_1:Names, ⟨ ⟩]] = escape.

(13) match namelists [[⟨ ⟩, ⟨ ⟩]] = complete.

(14) match namelists [["list" V_1:Variable, NS:Names]] =
|| check (already-matched(token of V))
| and then
||| give the structure mapped to the variable V
|| then

||| furthermore (match namelists (the given syntax-tree, NS))
or
| furthermore (bind token of V to the string of names NS).

3.2.2 Tipos

needs: **Identifiers and Names**
introduces: match types _,_ .

- match types _,_ :: Type → Type → action [escaping | binding |completing] [using current binds].

(1) match types [[T_1:PrimitiveType, T_2:PrimitiveType]] =
| check token of T_1 is token of T_2 then complete and rebind
or
| escape.

(2) match types [[T_1:PrimitiveType, N_1:Name]] = escape.

(3) match types [[N_1:Name, T_1:PrimitiveType]] = escape.

(4) match types [[V:Variable, T:Type]] =
|| check (already-matched(token of V))
| and then
||| give the structure mapped to the variable V
|| then
||| furthermore (match types (the given syntax-tree, T))
or
| furthermore (bind token of V to the string of T).

(5) match types [[N_1:Name, N_2:Name]] =
furthermore match names(N_1, N_2).

3.2.3 Declarações

needs: **Identifiers and Names, Types.**
introduces: match modifierlists_,_ , match modifiers_,_ .

- match modifierlists _,_:: Modifiers → Modifiers → action [escaping | binding |completing] [using current binds].

(1) match modifierlists [[M_1:Modifier, M_2:Modifier]] =
furthermore match modifiers (M_1, M_2).

(2) match modifierlists [[M_1:Modifier M_2:Modifiers, M_3:Modifier M_4:Modifiers]] =
|| furthermore (match modifiers(M_1, M_3)
| hence
|| furthermore (match modifierlists(M_2, M_4)).

(3) match modifierlists [[⟨ ⟩, M_1:Modifiers]] = escape.

(4) match modifierlists [[M_1:Modifiers, ⟨ ⟩]] = escape.

(5) match modifierlists ⟨ ⟩, ⟨ ⟩ = complete.

(6) match modifierlists [["mods" V:Variable, MS:Modifiers]] =
|| check (already-matched(token of V))
| and then
||| give the structure mapped to the variable V
|| then
||| furthermore (match modifierlists(the given syntax-tree, MS))
or
| furthermore (bind token of V to the string of MS).

- match modifiers _,_:: Modifier → Modifier → action [escaping | binding |completing] [using current binds].

(7) match modifiers [[M_1:Modifier, M_2:Modifier]] =
| check token of M_1 is token of M_2 then rebind
or
| escape.

3.2.3.1 Declarações de Classes

introduces: match class declarations _,_ .

- match class declarations _,_:: ClassDeclaration → ClassDeclaration → action [escaping | binding | completing] [using current binds].

(1) match class declarations [[MS_1:Modifiers "class" I_1:Identifier "extends" SC_1:Name "implements" IC_1:Names CB_1:ClassBody, MS_2:Modifiers "class" I_2:Identifier SC_2:Name "implements" IC_2:Names CB_2:ClassBody]] =
||| furthermore match modifierlists (MS_1, MS_2)
|| hence
||| furthermore match identifiers (I_1, I_2)
| hence
|| furthermore match names (SC_1, SC_2)
hence
|| furthermore match namelists (IC_1, IC_2)
| hence
|| furthermore match class bodies (CB_1, CB_2).

3.2.3.2 Declarações no Corpo de uma Classe

introduces: match class bodies _,_ , fully match _,_ , match remaining declarations _,_ using _, get remaining methods in _ using _,_ , get remaining fields in _ using _,_ , get remaining constructors in _ using _,_ , search and match

methods var in _ using _ , search and match fields var in _ using _ , search and match constructors var in _ using _.

- match class bodies _,_:: ClassBody → ClassBody → action [escaping | binding | completing] [using current binds].

(1) match class bodies [["{" C_1:ClassBodyDeclaration* "}", "{" C_2:ClassBodyDeclaration* "}"]] =
|| give an empty-set
|then
|| furthermore (fully match (C_1, C_2))
then
| furthermore (match remaining declarations C_1, C_2 using the given result-set).

- fully match _,_ :: ClassBodyDeclaration* → ClassBodyDeclaration* → action [escaping | binding | completing | giving a result-set] [using current binds | the given result-set].

(2) fully match < C_1:ClassBodyDeclaration CB_1:ClassBodyDeclaration*, C_2:ClassBodyDeclaration CB_2:ClassBodyDeclaration+ > =
furthermore
|| ((fully match (C_1, C_2))
| trap
|| (fully match (C_1, CB_2)))
then
| furthermore (fully match (CB_1, C_2 CB_2))).

(3) fully match < ⟨ ⟩, C_1:ClassBodyDeclaration+ > = regive and rebind.

(4) fully match < C_1:ClassBodyDeclaration+, ⟨ ⟩ > = escape.

(5) fully match ⟨ ⟩, ⟨ ⟩ = rebind and regive.

(6) fully match [[M_1:MethodDeclaration, M_2:MethodDeclaration]] =
|| regive
| and
|| check not
|||| the string of M_2 in the given result-set
||| or
|||| the string of M_1 in the given result-set
then
||| (furthermore (match method declarations (M_1, M_2))) and regive
|| then
||| give union(the given result-set, set of(the string of M_1),set of(the string of M_2))
| trap
|| regive.

(7) fully match [[F_1:FieldDeclaration, F_2:FieldDeclaration]] =
|| regive

| and
|| check not
|||| the string of F_2 in the given result-set
||| or
|||| the string of F_1 in the given result-set
then
||| (furthermore (match field declarations (F_1, F_2))) and regive
|| then
||| give union(the given result-set, set of(the string of F_1), set of(the string of F_2))
| trap
|| regive.

(8) fully match [[C_1:ConstructorDeclaration, C_2:ConstructorDeclaration]] =
|| regive
| and
|| check not
|||| the string of C_2 in the given result-set
||| or
|||| the string of C_1 in the given result-set
then
||| (furthermore (match constructor declarations (C_1, C_2))) and regive
|| then
||| give union(the given result-set, set of(the string of C_1), set of(the string of C_2))
| trap
|| regive.

(9) fully match [[C:ConstructorDeclaration, M:MethodDeclaration]] = escape.

(10) fully match [[M:MethodDeclaration, C:ConstructorDeclaration]] = escape.

(11) fully match [[C:ConstructorDeclaration, F:FieldDeclaration]] = escape.

(12) fully match [[F:FieldDeclaration, C:ConstructorDeclaration]] = escape.

(13) fully match [[M:MethodDeclaration, F:FieldDeclaration]] = escape.

(14) fully match [[F:FieldDeclaration, M:MethodDeclaration]] = eacape.

(15) fully match [[TFV: TFieldsVarDeclaration, C:ClassBodyDeclaration]] = escape.

(16) fully match [[TCV: TConstructorsVarDeclaration, C:ClassBodyDeclaration]] = escape.

(17) fully match [[TMV: TMethodsVarDeclaration, C:ClassBodyDeclaration]] = escape.

- match remaining declarations _,_ using _:: ClassBodyDeclaration* → ClassBodyDeclaration* → result-set → action [escaping | binding | completing] [using current binds | the given result-set].

(18) match remaining declarations [[C_1:ClassBodyDeclaration*,
C_2ClassBodyDeclaration*]] using S: result-set =
||| give an empty-set and label it #1
|| and
||| give an empty-set and label it #2
| and
|| give an empty-set and label it #3
then
||| get remaining methods in C_2 using S, the given result-set #1
|| then
||| furthermore search and match methods var in C_1 using the given result-set
| and
|||| get remaining fields in C_2 using S, the given result-set #2
||| then
|||| furthermore search and match fields var in C_1 using the given result-set
|| and
|||| get remaining constructors in C_2 using S, the given result-set #3
||| then
|||| furthermore search and match constructors var in C_1 using the given result-set.

- get remaining methods in _ using _,_:: ClassBodyDeclaration* → result-set → result-set → action [escaping | completing | giving a result-set] .

(19) get remaining methods [[M: MethodDeclaration C:ClassBodyDeclaration*]]
using S:result-set, D:result-set =
||| check not ((the string of M) in S)
|| then
||| give union(D, set of (the string of M))
| or
||| check ((the string of M) in S)
|| then
||| give D
then
| get remaining methods (C, S, the given result-set).

(20) get remaining methods [[F: FieldDeclaration C:ClassBodyDeclaration*]] using S:result-set, D:result-set =
get remaining methods (C, S, D).

(21) get remaining methods [[CD: ConstructorDeclaration C:ClassBodyDeclaration*]]
using S:result-set, D:result-set =
get remaining methods (C, S, D).

(22) get remaining methods [[FV: FieldsVarDeclaration C:ClassBodyDeclaration*]]
using S: result-set, D:result-set =
get remaining methods (C, S, D).

(23) get remaining methods [[MV: MethodsVarDeclaration C:ClassBodyDeclaration*]]
using S: result-set, D:result-set =
get remaining methods (C, S, D).

(24) get remaining methods [[CV: ConstructorsVarDeclaration
C:ClassBodyDeclaration*]] using S: result-set, D:result-set =
get remaining methods (C, S, D).

(25) get remaining methods ⟨ ⟩ using S:result-set, D:result-set = give D.

- get remaining fields in _ using _,_:: ClassBodyDeclaration* → result-set → result-set → action [escaping | completing | giving a result-set] .

(26) get remaining fields [[F: FieldDeclaration C:ClassBodyDeclaration*]] using
S:result-set, D:result-set =
||| check not ((the string of F) in S)
|| then
||| give union(D, set of (the string of F))
| or
||| check ((the string of F) in S)
|| then
||| give D
then
| get remaining fields (C, S, the given result-set).

(27) get remaining fields [[M: MethodDeclaration C:ClassBodyDeclaration*]] using
S:result-set, D:result-set =
get remaining fields (C, S, D).

(28) get remaining fields [[CD: ConstructorDeclaration C:ClassBodyDeclaration*]]
using S:result-set, D:result-set =
get remaining fields (C, S, D).

(29) get remaining fields [[FV: FieldsVarDeclaration C:ClassBodyDeclaration*]] using
S: result-set, D:result-set =
get remaining fields (C, S, D).

(30) get remaining fields [[MV: MethodsVarDeclaration C:ClassBodyDeclaration*]]
using S: result-set, D:result-set =
get remaining fields (C, S, D).

(31) get remaining fields [[CV: ConstructorsVarDeclaration C:ClassBodyDeclaration*]]
using S: result-set, D:result-set =
get remaining fields (C, S, D).

(32) get remaining fields ⟨ ⟩ using S:result-set, D:result-set = give D.

- get remaining constructors in _ using _,_:: ClassBodyDeclaration* → result-set → result-set → action [escaping | completing | giving a result-set] .

(33) get remaining constructors [[C_1: ConstructorDeclaration
C_2:ClassBodyDeclaration*]] using S:result-set, D:result-set =
||| check not ((the string of C_1) in S)
|| then
||| give union(D, set of (the string of C_1))

| or
||| check ((the string of C_1) in S)
|| then
||| give D
then
| get remaining constructors (C_2, S, the given result-set).

(34) get remaining constructors [[F: FieldDeclaration C:ClassBodyDeclaration*]]
using S:result-set, D:result-set =
get remaining constructors (C, S, D).

(35) get remaining constructors [[M: MethodDeclaration C:ClassBodyDeclaration*]]
using S:result-set, D:result-set =
get remaining constructors (C, S, D).

(36) get remaining constructors [[FV: FieldsVarDeclaration C:ClassBodyDeclaration*]]
using S: result-set, D:result-set =
get remaining constructors (C, S, D).

(37) get remaining constructors [[MV: MethodsVarDeclaration
C:ClassBodyDeclaration*]] using S: result-set, D:result-set =
get remaining constructors (C, S, D).

(38) get remaining constructors [[CV: ConstructorsVarDeclaration
C:ClassBodyDeclaration*]] using S: result-set, D:result-set =
get remaining constructors (C, S, D).

(39) get remaining constructors ⟨ ⟩ using S:result-set, D:result-set = give D.

- search and match methods var in _ using _ :: ClassBodyDeclaration* → result-set → action [escaping | completing | binding]
[using current bindings].

(40) search and match methods var in [[M: MethodDeclaration
C:ClassBodyDeclaration*]] using D =
furthermore search and match methods var in C using D.

(41) search and match methods var in [[F: FieldDeclaration
C:ClassBodyDeclaration*]] using D =
furthermore search and match methods var in C using D.

(42) search and match methods var in [[C_1: ConstructorDeclaration
C_2:ClassBodyDeclaration*]] using D =
furthermore search and match methods var in C_2 using D.

(43) search and match methods var in [[FV: FieldsVarDeclaration
C:ClassBodyDeclaration*]] using D =
furthermore search and match methods var in C using D.

(44) search and match methods var in [[CV: ConstructorsVarDeclaration
C:ClassBodyDeclaration*]] using D =
furthermore search and match methods var in C using D.

(45) search and match methods var in [[“methods” V:Variable “;”
C_2:ClassBodyDeclaration*]] using D =
|| check not (already-matched(token of V))
|then
|| furthermore (bind token of V to D)
or
| check (already-matched(token of V)) then escape.

(46) search and match methods var in ⟨ ⟩ using D =
| check (D is empty -set) then complete
or
| check not (D is empty-set) then escape.

- search and match fields var in _ using _ :: ClassBodyDeclaration* → result-set →
action [escaping | completing | binding]
[using current bindings].

- search and match constructors var in _ using _ :: ClassBodyDeclaration* →
result-set → action [escaping | completing | binding]
[using current bindings].

Considerando a semelhança das funções semânticas search and match methods var in _ using _ , search and match fields var in _ using _ e search and match constructors var in _ using _ , iremos nos abster de definir explicitamente essas duas últimas.

3.2.3.3 Declarações de Métodos

introduces: match method declarations _,_ , match method bodies _,_ , match result types _,_ .

- match _ , _ :: MethodDeclaration → MethodDeclaration → action [escaping |
binding | completing]
[using current binds].

(1) match [[M_1:Modifiers R_1:ResultType I_1:Identifier
“(“ F_1:FormalParameters? “)” (“throws” N_1:Names)? MB_1:MethodBody,
M_2:Modifiers R_2:ResultType I_2:Identifier “(“ F_2:FormalParameters? “)”
(“throws” N_2:Names)? MB_2:MethodBody]] =
||||| furthermore match modifierlists (M_1, M_2)
|||| hence
||||| furthermore match result types(R_1, R_2)
||| hence
|||| furthermore match identifiers (I_1, I_2)
|| hence
||| furthermore match formal parameter lists(F_1, F_2)
|hence
|| furthermore match namelists (N_1, N_2)

hence
| furthermore match method bodies (MB_1, MB_2).

- match result types _ , _ :: ResultType → ResultType → action [escaping | binding | completing]
  [using current binds].

(2) match result types [[“void”, “void”]] = rebind.

(3) match result types [[“void”, T:Type]] = escape.

(4) match result types [[T_1: Type, T_2: Type]] = furthermore match types (T_1, T_2).

- match method bodies _ , _ :: MethodBody → MethodBody → action [escaping | binding | completing] [using current binds].

(5) match method bodies [[B_1:Block , B_2:Block]] = .

(6) match method bodies [[“;” , B:Block]] = escape.

(7) match method bodies [[B:Block, “;”]] = escape.

(8) match method bodies [[“;” , “;”]] = complete.

3.2.3.4 Declarações de Construtores

introduces: match constructor bodies _,_ , match constructor declarations _,_ .

- match constructor declarations _ , _ :: MethodDeclaration → MethodDeclaration → action [escaping | binding | completing]
  [using current binds].

(1) match constructor declarations [[M_1:Modifiers I_1:Identifier
“(“ F_1:FormalParameters$^?$ “)” (“throws” N_1:Names)$^?$
CB_1:ConstructorBody, M_2:Modifiers I_2:Identifier “(“ F_2:FormalParameters$^?$
“)” (“throws” N_2:Names)$^?$ CB_2:ConstructorBodyBody]] =
|||| furthermore match modifierlists (M_1, M_2)
||| hence
|||| furthermore match identifiers (I_1, I_2)
|| hence
||| furthermore match formal parameter lists(F_1, F_2)
|hence
|| furthermore match namelists (N_1, N_2)
hence
| furthermore match constructor bodies (CB_1, CB_2).

- match constructor bodies _ , _ :: ConstructorBody → ConstructorBody → action [escaping | binding | completing] [using current binds].

(2) match constructor bodies [[“{“ E_1:ExplicitConstructorCall$^?$ B_1:BlockStatement*
“}”, “{“E_2:ExplicitConstructorCall$^?$ B_2:BlockStatement* “}”]] = .

3.2.3.5 Parâmetros Formais

introduces: match formal parameter lists _,_ , match formal parameters _,_ .

- match formal parameter lists _ , _ :: FormalParameters → FormalParameters →
  action [escaping | binding | completing]
  [using current binds].

(1) match formal parameter lists [[F_1:FormalParameter, F_2:FormalParameter]] =
  furthermore (match formal parameters (F_1, F_2)) .

(2) match formal parameter lists [[F_1:FormalParameter "," FS_1:FormalParameters,
  F_2:FormalParameter "," FS_2:FormalParameters]] =
  | furthermore (match formal parameters (F_1, F_2))
  hence
  | furthermore (match formal parameter lists (FS_1, FS_2)).

(3) match formal parameter lists [[⟨ ⟩, F_1:FormalParameters]] = escape.

(4) match formal parameter lists [[F_1:FormalParameters, ⟨ ⟩]] = escape.

(5) match formal parameter lists ⟨ ⟩, ⟨ ⟩ = complete.

(6) match formal parameter lists [["list" V:Variable, FS:FormalParameters]] =
  || check (already-matched(token of V))
  | and then
  ||| give the structure mapped to the variable V
  || then
  ||| furthermore (match formal parameter lists (the given syntax-tree, FS))
  or
  | furthermore (bind token of V to the string of formal parameters list FS).

- match formal parameters _ , _ :: FormalParameter → FormalParameter → action
  [escaping | binding | completing] [using current binds].

(7) match formal parameters [[T_1:Type I_1:Identifier, T_2:Type I_2:Identifier]] =
  | furthermore (match types T_1, T_2)
  hence
  | furthermore (match identifiers I_1, I_2).

(8) match formal parameters [[V:Variable, F:FormalParameter]] =
  || check (already-matched(token of V))
  | and then
  ||| give the structure mapped to the variable V
  || then
  ||| furthermore (match formal parameters (the given syntax-tree, F))
  or
  | furthermore (bind token of V to the string of F).

3.2.3.6 Declarações de Atributos

introduces: match field declarations _,_ , match field declarator lists _,_ , match field declarators _,_ , match variable initializers _,_ .

- match field declarations _ , _ :: FieldDeclaration → FieldDeclaration → action
  [escaping | binding | completing] [using current binds].

(1) match field declarations [[M_1:Modifiers T_1:Type F_1:FieldDeclarators ";",
  M_2:Modifiers T_2: Type F_2:FieldDeclarators ";"]] =
  || furthermore match modifiers (M_1, M_2)
  | hence
  || furthermore match types(T_1, T_2)
  hence
  | furthermore match field declarator lists (F_1, F_2).

- match field declarator lists _,_ :: FieldDeclarators → FieldDeclarators → action
  [escaping | binding | completing] [using current binds].

(2) match field declarator lists [[F_1:FieldDeclarator ("," FD_1:FieldDeclarators)$^?$,
  F_2:FieldDeclarator ("," FD_2:FieldDeclarators)$^?$]] =
  | furthermore match field declarators (F_1, F_2)
  hence
  | furthermore match field declarator lists (FD_1, FD_2).

(3) match field declarator lists [[⟨ ⟩, F_1:FieldDeclarators]] = escape.

(4) match field declarator lists [[F_1:FieldDeclarators, ⟨ ⟩]] = escape.

(5) match field declarator lists ⟨ ⟩, ⟨ ⟩ = complete.

- match field declarators _,_ :: FieldDeclarator → FieldDeclarator → action
  [escaping | binding | completing] [using current binds].

(6) match field declarators [[I_1:Identifier ("="V_1:VariableInitializer)$^?$, I_2:Identifier
  (V_2:VariableInitializer)$^?$]] =
  | furthermore match identifiers (I_1, I_2)
  hence
  | furthermore match variable intializers (V_1, V_2).

- match variable initializers _,_ :: VariableInitializer → VariableInitializer → action
  [escaping | binding | completing] [using current binds].

(7) match variable initializers [[V_1:VariableInitializer, V_2:VariableInitializer]] = .

3.2.3.7 Unidades de Compilação

introduces: match _,_ , match package declarations _,_ , match import declarations _,_ , match type declarations _,_ .

- match _ , _ :: CompilationUnit → CompilationUnit → action [escaping | binding | completing] [using current binds].

(1) match [[P_1:PackageDeclaration$^?$, I_1:ImportDeclaration* T_1:TypeDeclaration, P_2:PackageDeclaration$^?$, I_2:ImportDeclaration* T_2:TypeDeclaration]] =
|| furthermore match package declarations (P_1, P_2)
| hence
|| furthermore match import declarations (I_1, I_2)
hence
| furthermore match type declarations (T_1, T_2) .

- match package declarations _,_ :: PackageDeclaration → PackageDeclaration → action [escaping | binding | completing] [using current binds].

(2) match package declarations[["package" N_1:Name ";", "package" N_2:Name ";"]]=
furthermore match names (N_1, N_2).

(3) match package declarations [[⟨ ⟩, P_1:PackageDeclaration]] = escape.

(4) match package declarations [[P_1:PackageDeclaration, ⟨ ⟩]] = escape.

(5) match package declarations ⟨ ⟩, ⟨ ⟩ = complete.

- match import declarations _,_ :: ImportDeclaration* → ImportDeclaration* → action [escaping | binding | completing] [using current binds].

(6) match import declarations < I_1:ImportDeclaration I_2:ImportDeclaration$^+$, I_3:ImportDeclaration I_4:ImportDeclaration$^+$ >=
| furthermore match import declarations (I_1, I_3)
hence
| furthermore match import declarations (I_2, I_4).

(6) match import declarations [[⟨ ⟩, I_1:ImportDeclaration]] = escape.

(7) match import declarations [[I_1:ImportDeclaration, ⟨ ⟩]] = escape.

(8) match import declarations ⟨ ⟩, ⟨ ⟩ = complete.

(9) match import declarations [["import" N_1:Name "." "*" ";", "import" N_2:Name "." "*" ";"]] =
furthermore match names (N_1, N_2).

(10) match import declarations [["import" N_1:Name ";", "import" N_2:Name ";"]] =
furthermore match names (N_1, N_2).

(11) match import declarations[["import" N_1:Name ";", "import" N_2:Name "." "*" ";"]]
= escape.

(12) match import declarations[[“import” N_1:Name “.” “*” “;”, “import” N_2:Name “;”]]
= escape.

- match type declarations _ , _ :: TypeDeclaration → TypeDeclaration → action [escaping | binding | completing] [using current binds].

(13) match type declarations [[C_1:ClassDeclaration , C_2:ClassDeclaration]] =
furthermore match class declarations (C_1, C_2).

(14) match type declarations [[I_1:InterfaceDeclaration , I_2:InterfaceDeclaration]]= .

3.3 Execução

3.3.1 Identificadores e Nomes

needs: Parsing de Strings.
introduces: execute _ , execute identifier _ , execute name _ , execute names _ .

- execute identifier _ :: Identifier → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute identifier [[W:Word]] = give W.

(2) execute identifier [[V:Variable]] = give V.

(3) execute identifier [[E:ExecutableExpression]] = execute E.

- execute name _ :: Name → action [escaping | completing | giving a syntax-tree] [using current binds].

(4) execute name [[W:Word]] = give W.

(5) execute name [[I:Identifier “.” N:Name]] =
| execute identifier I and execute name N
then
| give < the given syntax-tree #1, the given syntax-tree #2 >.

(6) execute name [[E:ExecutableExpression]] = execute E.

(7) execute name [[V:Variable]] = give V.

- execute names _ :: Names → action [escaping | completing | giving a syntax-tree] [using current binds].

(8) execute names [[N:Name]] = give N.

(9) execute names [[N:Name “,” NS:Names]] =
| execute name N and execute names NS
then
| give < the given syntax-tree #1, the given syntax-tree #2 >.

(10) execute names [[“list” E:ExecutableExpression]] = execute E.

(11) execute names [[“list” V:Variable]] =
| parse (“list ”)
then
| give < the given syntax-tree, V >.

3.3.2 Tipos

needs: **Identifiers and Names**.
introduces: execute type _ .

- execute type _ :: Type → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute type [[P:PrimitiveType]] = give P.

(2) execute name [[I:Identifier “.” N:Name]] =
| execute identifier I and execute name N
then
| give < the given syntax-tree #1, the given syntax-tree #2 >

(3) execute name [[E:ExecutableExpression]] = execute E.

3.3.3 Declarações

needs: **Identifiers and Names, Types, Parsing de Strings**.
introduces: execute modifiers _ .

- execute modifiers _ :: Modifiers → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute modifiers [[M:Modifier^{+}]] = give M.

(2) execute modifiers [[E:ExecutableExpression]] = execute E.

(3) execute modifiers ⟨ ⟩ = give parse(“”).

3.3.3.1 Declarações de Classes

introduces: execute _ , execute class body _ , execute class body declaration _ .

- execute _ :: ClassDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[MS:Modifiers “class” I:Identifier (“extends” SC:Name)$^{?}$ (“implements” IC:Names)$^{?}$ CB:ClassBody]] =
||||| execute modifiers MS and parse (“ class ”) and execute identifier I

```
|||| then
||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the
      given string #3 >
||| then
|||||| check is empty string(SC)
||||| and then
|||||| regive
|||| or
|||||| check not (is empty string (SC))
||||| and then
|||||| regive and parse(“ extends ”) and execute name SC
|||||| then
||||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the
      given syntax-tree #3 >
|| then
||||| check is empty string(IC)
|||| and then
||||| regive
||| or
||||| check not (is empty string (IC))
|||| and then
|||||| regive and parse(“ implements ”) and execute names IC
||||| then
|||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the
      given string #3 >
| then
||| regive and execute class body CB
|| then
||| give < the given syntax-tree #1, the given syntax-tree #2 >
and
| rebind.
```

O operador binário apresentado acima (< _ , _ >) realiza concatenação de duas árvores sintáticas e faz parte da notação padrão de dados da Semântica de Ações.

- execute class body _ :: ClassBody → action [escaping | completing | giving a syntax-tree] [using current binds].

(2) execute class body [[“{” C:ClassBodyDeclaration* “}”]] =
```
    | execute class body declaration C
    then
    | give the given syntax tree.
```

- execute class body declaration _ :: ClassBodyDeclaration* → action [escaping | completing | giving a syntax-tree] [using current binds].

(4) execute class body declaration ⟨ ⟩ = regive.

(5) execute class body declaration [[C:ClassBodyDeclaration]] = execute C .

(6) execute class body declaration < C:ClassBodyDeclaration

CS:ClassBodyDeclaration+ > =
| execute C and execute class body declaration CS
then
| give < the given syntax-tree #1, the given syntax-tree #2 >.

3.3.3.2 Declarações de Métodos

introduces: execute _ , execute result type _ , execute method body _ .

- execute _ :: MethodDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[MS:Modifiers R:ResultType I:Identifier "(" F:FormalParameters? ")" ("throws" EX:Names)? MB:MethodBody]] =
|||||execute modifiers MS and execute result type R and execute identifier I
|||| then
||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the given string #3 >
||| then
|||||| check is empty string(F)
||||| and then
|||||| (regive and parse ("( )")) then give <the given syntax-tree #1, the given syntax-tree #2>
|||| or
|||||| check not (is empty string (F))
||||| and then
||||||| regive and parse("(") and execute formal parameters F and parse(")")
|||||| then
||||||| give <<< the given syntax-tree #1, the given syntax-tree #2 >, the given syntax-tree #3 >, the given syntax-tree #4>
|| then
||||| check is empty string(EX)
|||| and then
||||| regive
||| or
||||| check not (is empty string (EX))
|||| and then
|||||| regive and parse(" throws ") and execute names EX
||||| then
|||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the given string #3 >
| then
||| regive and execute method body MB
|| then
||| give < the given syntax-tree #1, the given syntax-tree #2 >
and
| rebind.

- execute method body _ :: MethodBody → action [escaping | completing | giving a syntax-tree] [using current binds].

(2) execute method body [[B:Block]] = parse (“{ }”) .

(3) execute method body [[“;”]] = parse (“;”) then give then given syntax tree.

- execute result type _ :: ResultType → action [escaping | completing | giving a syntax-tree] [using current binds].

(4) execute result type [[“void”]] = parse(“void”).

(5) execute result type [[T:Type]] = execute type T.

3.3.3.3 Parâmetros Formais

introduces: execute formal parameters _ , execute formal parameter _ .

- execute formal parameters _ :: FormalParameters → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute formal parameters [[F:FormalParameter]] =
execute formal parameter F.

(2) execute formal parameters [[F:FormalParameter “,” FS:FormalParameters]] =
| execute formal parameter F and and parse (“,”) and execute formal parameters FS
then
| give <<the given syntax-tree #1, the given syntax-tree #2>, the given syntax-tree #3>.

(3) execute formal parameters ⟨ ⟩ = parse (“”).

(4) execute formal parameters [[“list” V:Variable]] = give < parse(“list”) , V >.

(5) execute formal parameters [[“list” E:ExecutableExpression]] = execute E.

- execute formal parameter _ :: FormalParameter → action [escaping | completing | giving a syntax-tree] [using current binds].

(6) execute formal parameter [[T:Type I:Identifier]] =
| execute type T and execute identifier I
then
| give <the given syntax-tree #1, the given syntax-tree #2>.

(7) execute formal parameter [[V:Variable]] = give V.

(8) execute formal parameter [[E:ExecutableExpression]] = execute E.

3.3.3.4 Declarações de Construtores

introduces: execute _ , execute constructor body _ .

- execute _ :: ConstructorDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[MS:Modifiers I:Identifier "(" F:FormalParameters? ")" ("throws" EX:Names)? CB:ConstructorBody]] =
|||||execute modifiers MS and execute identifier I
|||| then
||||| give <the given syntax-tree #1, the given syntax-tree #2 >
||| then
|||||| check is empty string(F)
||||| and then
|||||| (regive and parse ("( )")) then give <the given syntax-tree #1, the given syntax-tree #2>
|||| or
|||||| check not (is empty string (F))
||||| and then
||||||| regive and parse("(") and execute formal parameters F and parse(")")
|||||| then
||||||| give <<< the given syntax-tree #1, the given syntax-tree #2 >, the given syntax-tree #3 >, the given syntax-tree #4>
|| then
||||| check is empty string(EX)
|||| and then
||||| regive
||| or
||||| check not (is empty string (EX))
|||| and then
|||||| regive and parse(" throws ") and execute names EX
||||| then
|||||| give << the given syntax-tree #1, the given syntax-tree #2 >, the given string #3 >
| then
||| regive and execute constructor body CB
|| then
||| give < the given syntax-tree #1, the given syntax-tree #2 >
and
| rebind.

- execute constructor body _ :: ConstructorBody → action [escaping | completing | giving a syntax tree] [using current binds].

(2) execute constructor body [["{" E_1:ExplicitConstructorCall? B_1:BlockStatement* "}"]] = parse("{}").

3.3.3.5 Declarações de Atributos

introduces: execute _ , execute field declarators _ , execute field declarator _ .

- execute _ :: FieldDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[MS:Modifiers T:Type V:FieldDeclarators ";"]] =
|||execute modifiers MS and execute type T
|| then
||| give <the given syntax-tree #1, the given syntax-tree #2 >
| then
||| regive and execute field declarators V
then
| give <<the given syntax-tree #1, the given syntax-tree #2>, parse(";")>.

- execute field declarators _ :: FieldDeclarators → action [escaping | completing | giving a syntax tree] [using current binds].

(2) execute field declarators [[V:FieldDeclarator]] =
execute field declarator V.

(3) execute field declarators [[V:FieldDeclarator "," VS:FieldDeclarators]] =
| execute field declarator V and parse(",") and execute field declarators VS
then
| give <<the given syntax-tree #1, the given syntax-tree #2>, the given syntax tree #3>.

- execute field declarator _ :: FieldDeclarator → action [escaping | completing | giving a syntax tree] [using current binds].

(4) execute field declarator [[I:Identifier ("=" V:VariableInitializer)$^{?}$]] =
| execute identifier I and execute variable initializer V
then
| give <the given syntax-tree #1, the given syntax-tree #2>.

- execute variable initializer _ :: VariableInitializer → action [escaping | completing | giving a syntax tree] [using current binds].

(5) execute field declarator [[V:VariableInitializer]] = .

3.3.3.6 Declarações "Para todo"

needs: **Java Action Semantics [1].**
introduces: execute _ , execute block for all _ in _ using _ , respectively execute block for all declarations in _ using _ and _.

- execute _ :: ForallDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [["forall" V_1:Variable "in" V_2:Variable "{"D:ForallBlockDeclaration*"}"]] =
||| check (already-matched V_2) and check not (already-matched V_1)
|| then

||| give the result-set bound to V_2
| then
|| respectively execute block for all declarations in (the given result-set) using V_1 and D
or
|| check not (already-matched V_2) or check (already-matched V_1)
| then
|| escape.

- respectively execute block for all declarations in _ using _ and _ :: result-set → Variable → ForallBlockDeclaration* → action [escaping | completing | giving a syntax-tree | binding] [using current binds].

(2) respectively execute block for all declarations in S:result-set using V:Variable and [[D:ForallBlockDeclaration*]] =
||| choose S
|| then
||| bind token of V to the given string and regive
| then
|| execute block for all V in D and regive
then
||| check (difference(S, result-set of(the string bound to token of V)) is empty-set)
|| and then
||| (give the given syntax-tree) and unbind token of V
| or
||| check not (difference(S, set of(the string bound to token of V)) is empty-set)
|| and then
||||| unbind V
|||| hence
||||| regive and (respectively execute block for all declarations in difference(S, set of(the given element)) using V and D)
||| then
|||| give <the given syntax-tree #1, the given syntax-tree #2>.

- execute block for all _ in _:: Variable → ForallBlockDeclaration* → action [escaping | completing | giving a syntax-tree] [using current binds].

(3) execute block for all V:Variable in < D_1: ForallBlockDeclaration D_2:ForallBlockDeclaration+ > =
|| execute block for all V in D_1 using C
| and
|| execute block for all V in D_2 using C
then
| give <the given syntax-tree #1, the given syntax-tree #2>.

(4) execute block for all V:Variable in [[D: ForallBlockDeclaration]] using S:result-set =
| replace in for all block declaration D
then
|| replace and execute the executable expressions in D using V

| then
|| give the given syntax-tree.

(5) execute block for all V:Variable in ⟨ ⟩ using S:result-set = parse("").

- replace and execute the executable expressions in _ using _:: ----------
  ForallBlockDeclaration → Variable → action [escaping | completing | giving a syntax-tree] [using current binds].

(5) replace and execute the executable expressions in [[D:ForallBlockDeclaration]] using V:Variable = .

A função replace and execute the executable expressions in _ using _ não foi definida por dois fatores:

1 – Ela envolve substituições com um grau de complexidade considerável, pois tem que levar em consideração a maneira como um objeto a partir do qual um método pode ser invocado é representado.

2 – Ela lida com a execução de um ou mais métodos a partir da modelagem de um objeto e este documento não especifica esse tipo de comportamento (está fora de seu escopo).

3.3.3.7 Unidades de Compilação

introduces: execute _ .

- execute _ :: CompilationUnit → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[P:PackageDeclaration$^{?}$ I:ImportDeclaration* T:TypeDeclaration*]] =
|||| check is empty string(P)
|||| and then
|||| regive
||| or
|||| check not (is empty string (P))
|||| and then
|||| execute P
|| then
|||| check is empty string(I)
|||| and then
|||| regive
||| or
|||| check not (is empty string (I))
|||| and then
|||||| regive and execute I
|||| then
|||||| give <the given syntax-tree #1, the given syntax-tree #2>
| then
||| regive and execute T
|| then

||| give <the given syntax-tree #1, the given syntax-tree #2>
and
| rebind.

- execute _ :: PackageDeclaration → action [escaping | completing | giving a syntax-tree] [using current binds].

(2) execute [[“package” N:Name “;”]] =
| execute name N
then
| give <<parse(“package ”), the given syntax-tree> , parse(“;”)>.

- execute _ :: ImportDeclaration* → action [escaping | completing | giving a syntax-tree] [using current binds].

(3) execute < I_1:ImportDeclaration I_2:ImportDeclaration+ > =
| execute I_1 and execute I_2
then
| give <the given syntax-tree #1, the given syntax-tree #2>.

(4) execute [[“import” N:Name “;”]] =
| execute name N
then
| give <<parse(“import ”), the given syntax-tree> , parse(“;”)>.

(5) execute [[“import” N:Name “.” “*” “;”]] =
| execute name N
then
| give <<parse(“import ”), (the given syntax-tree)>, parse(“.*;”)>.

- execute type declaration _ :: TypeDeclaration* → action [escaping | completing | giving a syntax-tree] [using current binds].

(6) execute type declaration < T_1:TypeDeclaration T_2:TypeDeclaration+ > =
| (execute type declaration T_1) and (execute type declaration T_2)
then
| give <the given syntax-tree #1, the given syntax-tree #2>.

(7) execute type declaration [[C:ClassDeclaration]] = execute C.

(8) execute type declaration [[I:InterfaceDeclaration]] = .

3.3.4 Expressões Executáveis

needs: **Java Action Semantics [1], Identifiers and Names, Types, Parsing de Strings**.
introduces: execute _ .
privately introduces: execute single _ , execute multiple _

- execute _ :: ExecutableExpression → action [escaping | completing | giving a syntax-tree] [using current binds].

(1) execute [[“[[” P:MethodCall “]]”]] = execute single P.

(2) execute < “[[” V:Variable “:” P:MethodCall^{+} “]]” > =
execute multiple P.

- execute single _ :: ExecutableExpression → action [escaping | completing | giving a syntax-tree] [using current binds].

(3) execute single [[“[[” P:MethodCall “]]”]] = .

- execute multiple _ :: ExecutableExpression → action [escaping | completing | giving a syntax-tree] [using current binds].

(4) execute multiple [[V:Variable “:” P:MethodCall]] = .

3.4 Substituição

3.4.1 Identificadores e Nomes

introduces: replace _ , is empty string _ , replace name_ , replace names _.

- is empty string _ :: Name → yielder [of a truth-value].

(1) is empty string [[N:Name]] = true.

(2) is empty string ⟨ ⟩ = false.

- is empty string _ :: Names → yielder [of a truth-value].

(3) is empty string [[N:Names]] = true.

(4) is empty string ⟨ ⟩ = false.

- replace _ :: Identifier → action [escaping | completing | giving a string] [using current binds].

(5) replace [[W:Word]] = give token of W.

(6) replace [[V:Variable]] =
|check already-matched(token of V) then
|| verify-identifier-compatibility(the string bound to token of V)
|and then
|| give (the string bound to token of V) and rebind
or
|| check not already-matched(token of V)
| then
|| give(token of V ^ “ ”).

(7) replace [[E::ExecutableExpression]] = give the string of E.

- replace name _ :: Name → action [escaping | completing | giving a string]
  [using current binds].

(8) replace name [[V:Variable]] =
  | check already-matched(token of V) then
  || verify-name-compatibility(the string bound to token of V)
  |and then
  || give (the string bound to token of V) and rebind
  or
  || check not already-matched(token of V)
  | then
  || give(token of V ^ “ ”).

(9) replace name [[W:Word]] = give token of W.

(10) replace name [[I:Identifier “.” N:Name]] =
  || replace (I)
  |then
  ||| regive and replace name (N)
  || then
  ||| give (the given string #1) ^ “.” ^ the given string #2)
  and
  | rebind.

(11) replace name [[E::ExecutableExpression]] = give the string of E.

- replace names _ :: Names → action [escaping | completing | giving a string]
  [using current binds].

(12) replace names [[N:Name]] = replace name N.

(13) replace names [[N:Name “,” NS:Names]] =
  || replace name (N)
  |then
  ||| regive and replace names (NS)
  || then
  ||| give (the given string #1) ^ “,” ^ then given string #2)
  and
  | rebind.

(14) replace names [[“list” V:Variable]] =
  |check already-matched(token of V) then
  || verify-names-compatibility(the string bound to token of V)
  |and then
  || give (the string bound to token of V) and rebind.
  or
  || check not already-matched(token of V)
  | then
  || give(“list” ^ token of V ^ “ ”).

(15) replace names [[“list” E::ExecutableExpression]] = give “list ” ^ the string of E.

3.4.2 Tipos

needs: **Identifiers and Names**.

introduces: replace type _ .

- replace type _ :: Type → action [escaping | completing | giving a string] [using current binds].

(1) replace type [[P:PrimitiveType]] =
give (token of P) and rebind.

(2) replace type [[V:Variable]] =
|check already-matched(token of V) then
||| verify-type-compatibility(the string bound to token of V)
|| and then
||| give (the string bound to token of V) and rebind.
| or
|| check not already-matched(token of V) then give(token of V ^ " ").

(3) replace type [[E::ExecutableExpression]] = give the string of E.

3.4.3 Declarações

needs: **Identifiers and Names, Types**.

introduces: replace _ .

- replace _ :: Modifiers → action [escaping | completing | giving a string] [using current binds].

(1) replace [[M:Modifiers]] =
give (the string of M) and rebind.

(2) replace [["mods" V:Variable]] =
|check already-matched(token of V) then
||| verify-modifier-compatibility(the string bound to token of V)
|| and then
||| give (the string bound to token of V) and rebind.
or
|| check not already-matched(token of V)
| then
|| give("mods " ^ token of V ^ " ").

(3) replace [["mods" E::ExecutableExpression]] = give "mods " ^ (the string of E).

3.4.3.1 Declarações de Classes

- replace _ :: ClassDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[MS:Modifiers “class” I:Identifier (“extends” SC:Name)? (“implements” IC:Names)? CB:ClassBody]]=
|||| replace MS and replace I
|||| then
|||| give (the given string #1 ^ “ class ” ^ the given string #2))
||| then
|||||| check is empty string(SC)
||||| and then
|||||| regive
|||| or
|||||| check not (is empty string (SC))
||||| and then
||||||| regive and replace name SC
|||||| then
||||||| give (the given string #1) ^ “extends” ^ (the given string #2)
|| then
||||| check is empty string(IC)
|||| and then
||||| regive
||| or
||||| check not (is empty string (IC))
|||| and then
|||||| regive and replace names IC
||||| then
|||||| give (the given string #1) ^ “implements” ^ (the given string #2)
| then
||| regive and replace CB
|| then
||| give (the given string #1) ^ (the given string #2)
and
| rebind.

- replace _ :: ClassBody → action [escaping | completing | giving a string] [using current binds].

(2) replace [[“{” C:ClassBodyDeclaration* “}”]] =
| replace C
then
| give “{” ^ the given string ^ “}”.

- replace _ :: ClassBodyDeclaration* → action [escaping | completing | giving a string] [using current binds].

(3) replace [[C:ClassBodyDeclaration]] =
replace C.

(4) replace < C_1:ClassBodyDeclaration C_2:ClassBodyDeclaration+ > =

|| replace C_1
|and
|| replace C_2
then
| give (the given string #1) ^ (the given string #2).

(5) replace ⟨ ⟩ = give "".

3.4.3.2 Declarações de Métodos

introduces: replace result type _ , replace method body .

- replace _ :: MethodDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[MS:Modifiers R:ReturnType I:Identifier "(" F:FormalParameters ")" ("throws" EX:Names)? MB:MethodBody]]=
|||||| replace MS and replace result type R
||||| then
|||||| give (the given string #1 ^ " " ^ the given string #2))
|||| then
||||||| regive and replace I
|||||| then
||||||| give (the given string #1) ^ " " ^ (the given string #2)
||| then
|||||| check is empty string(F)
||||| and then
|||||| give (the given string #1) ^ "(" ^ ")"
|||| or
|||||| check not (is empty string (F))
||||| and then
|||||| regive and replace formal parameters F
||||| then
|||||| give (the given string #1) ^ "(" ^ (the given string #2) ^ ")"
|| then
||||| check is empty string(EX)
|||| and then
||||| regive
||| or
||||| check not (is empty string (EX))
|||| and then
|||||| regive and replace names EX
||||| then
|||||| give (the given string #1) ^ "throws" ^ (the given string #2)
| then
||| regive and replace method body MB
|| then
||| give (the given string #1) ^ (the given string #2)
and
| rebind.

- replace result type _ :: ResultType → action [escaping | completing | giving a string] [using current binds].

(2) replace result type [[“void”]] = give “void”.

(3) replace result type [[T:Type]] = replace T.

- replace method body _ :: MethodBody → action [escaping | completing | giving a string] [using current binds].

(4) replace method body[[“;”]] = give “;”.

(5) replace method body [[“{” B:Block “}”]] = give “{ }”.

3.4.3.3 Casamentos com Múltiplos Métodos

needs: **Substituição de Casamentos um-para-muitos (MethodsVar, FieldsVar, ConstructorsVar).**

- replace _ :: MethodsVarDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[“methods ” V:Variable “;”]] =
|||| check (already-matched (token of V))
||then
||| give the result-set bound to V then (replace using (the given result-set))
| or
|| check not (already-matched (token of V))
| then
|| give “methods” ^ (token of V)
and
| rebind.

3.4.3.4 Parâmetros Formais

introduces: replace formal parameters _ , replace formal parameter _ , is empty string _.

- is empty string _ :: FormalParameters → yielder [of a truth-value].

(1) is empty string [[F:FormalParameters]] = true.

(2) is empty string ⟨ ⟩ = false.

- replace formal parameters _ :: FormalParameters → action [escaping | completing | giving a string] [using current binds].

(3) replace formal parameters [[F: FormalParameter]] = replace F.

(4) replace formal parameters [[F_1: FormalParameter "," F_2:FormalParameters]] =
| replace formal parameter F_1 and replace formal parameters replace F_2
then
| give (the given string #1) ^ (the given string #2).

(5) replace formal parameters [["list" V:Variable]] =
|check already-matched(token of V) then
||| verify-formalparameters-compatibility(the string bound to token of V)
|| and then
||| give (the string bound to token of V) and rebind.
or
|| check not already-matched(token of V)
| then
|| give("list " ^ token of V).

(6) replace formal parameters ⟨ ⟩ = "".

(7) replace [["list" E::ExecutableExpression]] = give "list" ^ (the string of E).

- replace formal parameter _ :: FormalParameter → action [escaping | completing | giving a string] [using current binds].

(7) replace formal parameter [[T: Type I: Identifier]] =
| replace type T and replace I
then
| give (the given string #1) ^ (the given string #2).

(8) replace formal parameter [[V:Variable]] =
|check already-matched(token of V) then
||| verify-formalparameter-compatibility(the string bound to token of V)
|| and then
||| give (the string bound to token of V) and rebind.
or
|| check not already-matched(token of V)
| then
|| give (token of V).

(9) replace [[E::ExecutableExpression]] = give the string of E.

3.4.3.5 Declarações de Construtores

- replace _ :: ConstructorDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[MS:Modifiers I:Identifier "(" F:FormalParameters ")" ("throws" EX:Names)? CB:MethodBody]]=
||||| replace MS and replace I
|||| then
||||| give (the given string #1 ^ " " ^ the given string #2))

```
||| then
|||||| check is empty string(F)
||||| and then
|||||| give the given string #1 ^ “( )”
|||| or
|||||| check not (is empty string (F))
||||| and then
||||||| regive  and replace formal parameters F
|||||| then
||||||| give (the given string #1) ^ “(” ^ (the given string #2) ^ “)”
|| then
||||| check is empty string(EX)
|||| and then
||||| regive
||| or
||||| check not (is empty string (EX))
|||| and then
|||||| regive and replace names EX
||||| then
|||||| give (the given string #1) ^ “throws” ^ (the given string #2)
| then
||| regive and replace CB
|| then
||| give (the given string #1) ^ (the given string #2)
and
| rebind.
```

- replace _ :: ConstructorBody → action [escaping | completing | giving a string] [using current binds].

(4) replace [[“{“ E_1:ExplicitConstructorCall$^?$ B_1:BlockStatement* “}”]] = “{ }” .

3.4.3.6 Casamentos com Múltiplos Construtores

needs: **Substituição de Casamentos um-para-muitos (MethodsVar, FieldsVar, ConstructorsVar).**

- replace _ :: ConstructorsVarDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[“constructors ” V:Variable “;”]] =

```
||| check (already-matched (token of V))
||then
||| give the result-set bound to V then (replace using (the given result-set))
| or
|| check not (already-matched (token of V))
| then
|| give “constructors ” ^ (token of V)
and
| rebind.
```

3.4.3.7 Declarações de Atributos

introduces: replace field declarators _ , replace field declarator _ .

- replace _ :: FieldDeclaration → action [escaping | completing | giving a string] [using current binds].

(1) replace [[MS:Modifiers T: Type V:FieldDeclarators ";"]]=
||| replace MS and replace type T
|| then
||| give (the given string #1 ^ " " ^ the given string #2))
| then
||| regive and replace field declarators V
|| then
||| give (the given string #1) ^ " " ^ (the given string #2) ^ ";"
and
| rebind.

- replace field declarators _ :: FieldDeclarators → action [escaping | completing | giving a string] [using current binds].

(2) replace field declarators [[V:FieldDeclarator]] =
replace field declarator V.

(3) replace < V_1:FieldDeclarator "," V_2:FieldDeclarator$^+$> =
| replace field declarator V_1 and replace field declarators V_2
then
| give (the given string #1) ^ "," ^ (the given string #2).

- replace field declarator _ :: FieldDeclarator → action [escaping | completing | giving a string] [using current binds].

(4) replace field declarator [[I:Identifier ("=" V:VariableInitializer)$^?$]] =
| replace I and replace V
then
| give (the given string #1) ^ " = " ^ (the given string #2).

- replace _ :: VariableInitilizer → action [escaping | giving a string | completing] [using current binds].

(5) replace [[V_1:VariableInitializer]] = .

3.4.3.8 Casamentos com Múltiplos Atributos

needs: **Substituição de Casamentos um-para-muitos (MethodsVar, FieldsVar, ConstructorsVar).**

- replace _ :: ConstructorsVarDeclaration → action [escaping | completing | giving

a string] [using current binds].

```
(1)  replace [[ “fields ” V:Variable “;”]] =
        ||| check (already-matched (token of V))
        ||then
        ||| give the result-set bound to V then (replace using (the given result-set))
        | or
        || check not (already-matched (token of V))
        | then
        || give “fields ” ^ (token of V)
        and
        | rebind.
```

3.4.3.9 Declarações “Para todo”

introduces: replace in for all block declaration _ .

- replace in for all block declaration _ :: ForallBlockDeclaration → action [escaping | completing | giving a string] [using current binds].

```
(1)  replace in for all block declaration [[ M:MethodDeclaration ]] =
        replace M.

(2)  replace in for all block declaration [[ F:FieldDeclaration ]] =
        replace F.

(3)  replace in for all block declaration [[ C:ConstructorDeclaration ]] =
        replace C.

(4)  replace in for all block declaration [[ E:ExecutableExpression ]] =
        replace E.
```

Assim como no caso da substituição de declarações executáveis, a substituição em declarções “forall” é um caso especial de substituição. Isso ocorre porque ela não é realizada na etapa de substituição propriamente dita, e sim na de execução.

3.4.3.10 Unidades de Compilação

introduces: is empty string _ , replace type declaration _ .

- is empty string _ :: PackageDeclaration → yielder [of a truth-value].

```
(1)  is empty string [[ P:PackageDeclaration ]] = true.

(2)  is empty string ⟨ ⟩ = false.
```

- is empty string _ :: ImportDeclaration → yielder [of a truth-value].

(3) is empty string [[I:ImportDeclaration]] = true.

(4) is empty string ⟨ ⟩ = false.

- is empty string _ :: TypeDeclaration → yielder [of a truth-value].

(5) is empty string [[T:TypeDeclaration]] = true.

(6) is empty string ⟨ ⟩ = false.

- replace _ :: CompilationUnit → action [escaping | completing | giving a string] [using current binds].

(7) replace [[P:PackageDeclaration? I:ImportDeclaration* T:TypeDeclaration*]] =
|||| check is empty string(P)
|||| and then
|||| regive
||| or
|||| check not (is empty string (P))
|||| and then
|||| replace P
|| then
|||| check is empty string(I)
|||| and then
|||| regive
||| or
|||| check not (is empty string (I))
|||| and then
|||||| regive and replace I
|||| then
|||||| give (the given string #1) ^ (the given string #2)
| then
||| regive and replace CB
|| then
||| give (the given string #1) ^ (the given string #2)
and
| rebind.

- replace _ :: PackageDeclaration → action [escaping | completing | giving a string] [using current binds].

(8) replace [["package" N:Name]] =
| replace N
then
| give "package " ^ (the given string) ^ ";".

- replace _ :: ImportDeclaration* → action [escaping | completing | giving a string] [using current binds].

(9) replace < I_1:ImportDeclaration I_2:ImportDeclaration+ > =

| replace I_1 and replace I_2
then
| give (the given string #1) ^ (the given string #2).

(10) replace [[“import” N:Name “;”]] =
| replace N
then
| give “import ” ^ (the given string) ^ “;”.

(11) replace [[“import” N:Name “.” “*” “;”]] =
| replace N
then
| give “import ” ^ (the given string) ^ “.” ^ “*” “;”.

- replace type declaration _ :: TypeDeclaration* → action [escaping | completing | giving a string] [using current binds].

(11) replace type declaration < T_1:TypeDeclaration T_2:TypeDeclaration+ > =
| (replace type declaration T_1) and (replace type declaration T_2)
then
| give (the given string #1) ^ (the given string #2).

(12) replace type declaration [[C:ClassDeclaration]] = replace C.

(13) replace type declaration [[I:InterfaceDeclaration]] = .

3.5 Criação de Strings Mapeáveis

3.5.1 Identificadores e Nomes

introduces: token _ , the string of _ .

- token of _ :: Identifier → yielder [of a string].

(1) token of W:Word = W.

- token of _ :: Variable → yielder [of a string].

(2) token of V:Variable = V.

- the string of _ :: Name → yielder [of a string].

(3) the string of [[I:Identifier]] = token of I.

(4) the string of [[I:Identifier “.” N:Name]] =
token of I ^ “.” ^ the string of N.

- the string of namelist _ :: Names → yielder [of a string].

(5) the string of namelist [[N:Name]] = the string of N.

(6) the string of namelist [[N:Name “,” NS:Names]] =
the string of N ^ “,” ^ the string of namelist NS.

3.5.2 Tipos

needs: Identifiers and Names.
introduces: the string of _ , token of _ .

- the string of _ :: Type → yielder [of a string].

(1) the string of [[M: PrimitiveType]] = token of P.

- token of _ :: PrimitiveType → yielder [of a string].

(2) token of “boolean” = “boolean”.

(3) token of “char” = “char”.

(4) token of “byte” = “byte”.

(5) token of “short” = “short”.

(6) token of “int” = “int”.

(7) token of “long” = “long”.

(8) token of “float” = “float”.

(9) token of “float” = “float”.

3.5.3 Declarações

needs: Identifiers and Names, Types.
introduces: the string of _ , token of _ .

- the string of _ :: Modifiers → yielder [of a string].

(1) the string of [[M_1:Modifier]] = token of M_1.

(2) the string of [[M_1:Modifier M_2:Modifier$^+$]] =
token of M_1. ^ “ ” ^ the string of M_2.

- token of _ :: Modifier → yielder [of a string].

(2) token of “public” = “public”.

(3) token of “private” = “private”.

(4) token of “protected” = “protected”.

(5) token of “static” = “static”.

(6) token of “abstract” = “abstract”.

(7) token of “final” = “final”.

(8) token of “native” = “native”.

(9) token of “synchronized” = “synchronized”.

(10) token of “transient” = “transient”.

(11) token of “volatile” = “volatile”.

3.5.3.1 Declarações de Métodos

introduces: the string of throws declaration _ , the string of result type _.

- the string of _ :: MethodDeclaration → yielder [of a string].

(1) the string of [[M: Modifiers R:ResultType I: Identifier “(” F:FormalParameters “)” (“throws” NS:Names)? CB: ClassBody]] =
(the string of M) ^ “ ” ^ (the string of result type R) ^ “ ” ^ (token of I) ^ “(” ^ (the string of formal parameters list F) ^ “) ” ^ (the string of throws declaration NS) ^ “ ” ^ (the string of CB).

- the string of throws declaration _ :: Names → yielder [of a string].

(2) the string of throws declaration ⟨ ⟩ = “”

(3) the string of throws declaration [[NS:Name]] = “ throws ” ^ the string of NS.

- the string of result type _ :: ResultType → yielder [of a string].

(4) the string of result type [[“void”]] = “void”.

(5) the string of result type [[T:Type]] = the string of T.

- the string of _ :: MethodBody → yielder [of a string].

(6) the string of [[B: Block]] = “{ }”.

(7) the string of [[“;”]] = “;”.

Como não estamos tratando com o corpo do método, apenas com sua declaração, julgamos adequado definir que a string gerada pelo corpo de um método consiste apenas de um par de chaves vazio.

3.5.3.2 Parâmetros Formais

introduces: the string of formal parameters list _ .

- the string of formal parameters list_ :: FormalParameters → yielder [of a string].

(1) the string of formal parameter list [[F:FormalParameter]] =
the string of F.

(2) the string of formal parameter list [[F:FormalParameter ","
FS:FormalParameter]] =
the string of F ^ ", " ^ the string of formal parameter list (FS).

(3) the string of formal parameter list ⟨ ⟩ = "".

- the string of _ :: FormalParameter → yielder [of a string].

(4) the string of [[T:Type I:Identifier]] =
the string of T ^ " " ^ token of I.

3.5.3.3 Declarações de Construtores

- the string of _ :: ConstructorDeclaration → yielder [of a string].

(1) the string of [[M: Modifiers I: Identifier "(" F:FormalParameters ")" ("throws"
NS:Names)? CB: ConstructorBody]] =
(the string of M) ^ " " ^ (token of I) ^ "(" ^ (the string of F) ^ ") " ^ (the string of throws declaration NS) ^ " " ^ (the string of CB).

- the string of _ :: ConstructorBody → yielder [of a string].

(2) the string of [[CB:ConstructorBody]] = "{ }".

3.5.3.4 Declarações de Atributos

introduces: the string of field declarators.

- the string of _ :: FieldDeclaration → yielder [of a string].

(1) the string of [[M: Modifiers T:Type V:FieldDeclarators]] =
(the string of M) ^ " " ^ (the string of T) ^ " " ^ (the string of field declarators V) ^ ";".

- the string of field declarators _ :: FieldDeclarators → yielder [of a string].

(2) the string of field declarators [[V:FieldDeclarator]] =
the string of V.

(3) the string of field declarators [[V:FieldDeclarator “, ” VS:FieldDeclarators]] =
the string of V ^ “, ” the string of field declarators VS.

- the string of _ :: FieldDeclarator → yielder [of a string].

(4) the string of [[I:Identifier “=” V:VariableInitializer]] =
the string of I ^ “=” the string of V.

(5) the string of [[I:Identifier]] =
the string of I.

- the string of _ :: VariableInitializer → yielder [of a string].

(6) the string of [[V:VariableInitializer]] = .

Capítulo 4

Entidades Semânticas

needs: **Data Notation, Matchings, Execution and Replacement**

4.1 Números

- digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘’ | ‘’ | ‘6’ | ‘7’ | ‘8’ | ‘9’.
- non-zero-digit = ‘1’ | ‘2’ | ‘3’ | ‘’ | ‘’ | ‘6’ | ‘7’ | ‘8’ | ‘9’.

4.2 ‘Strings e Bindables

- alpha = uppercase-alpha | lowercase-letter.
- string = string-of(alpha (digit|alpha)*).
- token = string.
- variable = string of (‘$’, alpha (alpha | digit)*).
- result-set = set of string.
- bindable = string | syntax-tree | result-set.

4.3 Parsing de Strings

introduces: parse _ .

- parse :: yielder [of a string] → yielder [of a syntax-tree].

(1) parse S:String = .

4.4 Casamentos

4.4.1 Ações Auxiliares para Matching

introduces: already-matched _ .

- already-matched :: yielder [of a string] → yielder [of a truth value].

```
(1)  already-matched V =
         | give the string bound to V
         then
         ||check not (the given string is "") then true
         | or
         || check (the given string is "") then false.
```

4.5 Substituição

4.5.1 Substituição de Casamentos um-para-muitos(MethodsVar, FieldsVar, ConstructorsVar).

introduces: replace using _ .

- replace using :: yielder [of a result-set] → action [escaping | completing | giving a string] [using current binds].

```
(1)  replace using S:result-set =
         choose S then
         ||| check (difference(S, set of(the given string)) is empty-set)
         || and then
         ||| give the given string
         | or
         ||| check not (difference(S, set of(the given string)) is empty-set)
         || and then
         ||| replace using difference(S, set of(the given string
         ||| then
         |||| give the given string #1 ^ the given string #2.
```

Capítulo 5

Um Pequeno Exemplo

Neste capítulo veremos como procede a aplicação de uma pequena transformação JaTS a um pequeno programa Java.

Para nosso exemplo, usaremos o seguinte programa Java:

```
class Pessoa {

  private String nome;

}
```

E a seguinte transformação:

Lado Esquerdo:

```
class $c {

  private $t1 nome;

 }
```

Lado Direito:

```
class Funcionario {

  private $c cad;
  private $t1 info;

}
```

5.1 Casamento

O primeiro passo será executar a função run passando como parâmetro o lado esquerdo da transformação, o programa fonte e o lado direito. Ao fazermos isso, a execução começará da seguinte maneira:

```
||| check (has executable expressions(Left))
|| then
||| escape
| or
||| check not (has executable expressions(Left))
|| and then
||| match (Left, JavaSource)
```

Como o lado esquerdo não tem nenhuma declaração executável e o programa fonte não possui nenhuma construção JaTS, entramos na etapa de casamento:

||| furthermore match modifierlists (MS_1, MS_2)
|| hence
||| furthermore match identifiers (I_1, I_2)

A partir disso temos:

match modifierlists ⟨ ⟩, ⟨ ⟩ = complete.

Em seguida:

furthermore match identifiers (“$c”, “`Pessoa`”) =

check (already-matched(token of “$c”))
(i)

Como “$c” não foi mapeada ainda, podemos seguir adiante.

furthermore (bind token of V to token of I). **(ii)**

Ao fim dessa ação temos o nosso conjunto resultado igual a { $c → “`Pessoa`” }.

(i) token of “$c” = “$c”
(ii) token of “`Pessoa`” = “`Pessoa`”

Seguindo com o casamento das duas classes:

| hence furthermore match names (⟨ ⟩, ⟨ ⟩) =
match names ⟨ ⟩, ⟨ ⟩ = complete.

E, similarmente:

hence furthermore match namelists (⟨ ⟩, ⟨ ⟩) =
match namelists ⟨ ⟩, ⟨ ⟩ = complete.

O último passo do casamento entre as duas classes é:

```
| hence furthermore match class bodies (“{
                    private $t1 nome; }}”,
                    “{
                      private String nome; }”).
=
|| give an empty-set
|then
|| furthermore (fully match (“private $t1 nome;”, ”private String
nome;”))
then
| furthermore (match remaining declarations C1, C2 using the given result-set).

=

fully match [[“private $t1 nome;”, ”private String nome;”]] =
```

|| regive
| and
|| check not
|||| the string of (“`private $t1 nome;`”) in the empty-set
||| or
|||| the string of (”`private String nome;`”) in the empty-set

Não é necessário calcular a aplicação da função the string of para saber que nada pode estar dentro do conjunto vazio.

A partir disso a seguinte instrução é executada:

furthermore (match field declarations (F_1, F_2))) and regive

O que resulta no seguinte:

|| furthermore match modifiers (“`private`”,“`private`”) **(iv)**
| hence
|| furthermore match types(“`$t1`” ,“`String`”) **(v)**
hence
| furthermore match field declarator lists (“`nome`”, “`nome`”). **(vi)**

(vi) furthermore match field declarators (“`nome`”, “`nome`”) =
furthermore match identifiers (“`nome`”, “`nome`”) =
check token of I_1 is token of I_2 then rebind =
check “`nome`” is “`nome`” = true

(v) furthermore match types(“`$t1`”, “`String`”) =
check (already-matched(token of `$t1`)) ----> `$t1` $\notin$ dom { `$c` → “`Pessoa`” }

Consequentemente:
| furthermore (bind token of “`$t1`” to the string of “`String`”),

O que resulta em { `$c` → “`Pessoa`”, `$t1` → “`String`” }

REFERÊNCIAS BIBLIOGRÁFICAS

[1] WATT, D.; BROWN, D. *Java Action Semantics*. 1998

[2] GOSLING, J. et al. *The Java Language Specification 2nd Ed*. 2000

[3] MOSSES, P. *A Tutorial on Action Semantics.* 1996

[4] MOURA, H. *Action Semantics of SPECIMEN.* 1992