MARCELO ARAÚJO

LANGUAGE FEATURES LIBRARY
UMA BIBLIOTECA DE CLASSES PARA SEMÂNTICA DE AÇÕES ORIENTADAS A OBJETOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Martin A. Musicante
Co-orientador: André Luiz Pires Guedes

CURITIBA
2004

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Marcelo Araújo, avaliamos o trabalho intitulado, "*LANGUAGE FEATURES LIBRARY - UMA BIBLIOTECA DE CLASSES PARA SEMÂTICA DE AÇÕES ORIENTADA A OBJETO*", cuja defesa foi realizada no dia 30 de abril de 2004, às dez horas, no Anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 30 de abril de 2004.

Prof. Dr. Martin Alejandro Musicante
DINF/UFPR – Orientador

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR – Co-orientador

Profª. Dra. Mírian Halfeld Ferrari Alves
Universidade François Rabelais Blois-Tours-Chinon/França
Membro Externo

Prof. Dr. Alexandre Ibrahim Direne
DINF/UFPR – Membro Interno

AGRADECIMENTOS

Primeiramente agradeço a **Deus** por conceder esta oportunidade e manter-me com muita saúde e serenidade neste momento importante na minha vida.

Aos meus pais Osmar, Mércia e irmã Ana Cláudia, por terem me apoiado e ajudado em todos estes anos.

Obrigado professor André Luiz Pires Guedes pelo trabalho como co-orientador.

Ao professor Martin A. Musicante, muito obrigado pela aceitação no mestrado e pela sua excelente orientação em todas as fases desta dissertação e artigos.

A minha namorada, Marcia Corrêa, pelos incentivos e companheirismo durante este trabalho.

Pelo apoio de todos os meus familiares e colegas de mestrado.

SUMÁRIO

LISTA DE FIGURAS vii

RESUMO viii

ABSTRACT ix

1 INTRODUÇÃO 1

1.1 Definição do Problema 2

1.2 Objetivos 3

1.3 Estrutura desta Dissertação 4

2 SEMÂNTICA DE AÇÕES 5

2.1 Introdução 5

2.2 Notação de Ações 5

2.2.1 Notação de Dados 6

2.3 Classificação das Informações 7

2.3.1 *Transient* 8

2.3.2 *Binding (Scoped)* 8

2.3.3 *Storage (Stable)* 9

2.4 Entidades Semânticas 9

2.5 Facetas 12

2.5.1 Faceta Básica 12

2.5.2 Faceta Funcional 14

2.5.3 Faceta Declarativa 17

2.5.4 Faceta Imperativa 21

2.5.5 Faceta Reflexiva 22

2.6 Considerações Finais 24

3 ORIENTAÇÃO A OBJETOS **25**

3.1 Princípios da Orientação a Objetos 26

3.1.1 Modularização por Classe 26

3.1.2 Conceitos sobre Objetos 28

3.1.3 Abstração 30

3.1.4 Polimorfismo 30

3.2 Técnicas de Orientação a Objetos 31

3.2.1 Envio de Mensagem 31

3.2.2 Herança 31

3.2.3 Encapsulamento 33

3.3 Considerações Finais 33

4 SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS **35**

4.1 Aplicação da Orientação a Objetos 36

4.2 Sintaxe da Notação 38

4.3 Semântica da Notação 40

4.4 A Classe *State* 41

4.4.1 Atributos 41

4.4.2 Operações 41

5 BIBLIOTECA DE CLASSES PARA A SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS **43**

5.1 Introdução 43

5.2 Estrutura Organizacional 44

5.2.1 Organização das Classes nos Nodos *Syntax* e *Entity* 46

5.2.1.1 Nodo *Syntax* 47

5.2.1.2 Nodo *Entity* 47

5.2.2 Nodo *Semantics* 48

5.2.3 Hierarquia básica da LFL 48

5.3 Sintaxe da Notação 49

5.3.1 Notação de Classes 50
5.3.2 Definição do Corpo das Classes 53
5.3.3 Notação de Ações 54
5.4 Classes que compõem a LFL 55
5.4.1 Classes do Nodo *Declaration* 55
5.4.1.1 Nodo Imper 56
5.4.1.2 Nodo Shared 58
5.4.2 Classes do Nodo *Command* 59
5.4.2.1 Nodo Imper 59
5.4.2.2 Nodo Shared 61
5.4.3 Classes do Nodo *Expression* 63
5.4.3.1 Nodo Shared 64
5.5 Considerações Finais 68

6 ESPECIFICAÇÃO DA LINGUAGEM μPASCAL 69

6.1 Sintaxe Abstrata da Linguagem 69
6.2 Entidades Semânticas 71
6.3 Classes (Funções Semânticas) 72
6.3.1 Classes Auxiliares 72
6.3.2 Declarações 73
6.3.3 Comandos 75
6.3.4 Expressões 79
6.3.5 A Classe que Representa a Linguagem μPascal 85
6.4 Introduzindo Novas Características à Linguagem μPascal 85
6.4.1 Extensão da Sintaxe Abastrata 86
6.4.2 Classes (Extensão das Funções Semânticas) 87
6.4.2.1 Declarações 87
6.4.2.2 Comandos 89
6.4.2.3 Expressões 90
6.5 Considerações Finais 91

7 FORMAÇÃO DA LINGUAGEM JOOS **92**

7.1 Sintaxe Abstrata 92

7.2 Classes (Funções Semânticas) 95

7.2.1 Classes Auxiliares 95

7.2.2 Declarações 95

7.2.3 Comandos 97

7.2.4 Expressões 100

7.3 Considerações Finais 103

8 CONCLUSÕES E TRABALHOS FUTUROS **104**

REFERÊNCIAS BIBLIOGRAFIAS **108**

ANEXO A **109**

A.1 Notação de Classes 109

A.2 Definição do Corpo das Classes 109

A.3 Notação de Ações 110

ANEXO B **111**

B.1 Declaration 111

B.2 Command 115

B.3 Expression 118

ANEXO C **124**

C.1 Classes Auxiliares 124

C.2 Declarações 124

C.3 Comandos 126

C.4 Expressões 128

C.4.1 Representação da Linguagem μPascal 131

ANEXO D **132**

D.1 Abstract Syntax 132

D.2 Semantic Entities 134
D.3 Semantic Functions 137

LISTA DE FIGURAS

2.1 Combinador A_1 *and* A_2. 15
2.2 Combinador A_1 *then* A_2. 15
2.3 Combinador A_1 *and then* A_2. 16
2.4 Combinador A_1 *or* A_2. 16
2.5 Combinadores (A_1 *and* A_2) e (A_1 *and then* A_2). 18
2.6 Combinador A_1 *then* A_2. 18
2.7 Combinador A_1 *moreover* A_2. 19
2.8 Combinador A_1 *hence* A_2. 19
2.9 Combinador A_1 *before* A_2. 20

3.1 Modelagem de Classe. 27
3.2 Modelagem de Classe com *Herança*. 33

5.1 Nodos base da LFL. 44
5.2 Nodo Syntax. 47
5.3 Nodo Entity. 47
5.4 Nodo Semantics. 48
5.5 Estrutura organizacional aplicada à LFL. 49
5.6 Classes contidas no nodo *Declaration*. 56
5.7 Classes contidas no nodo *Command*. 59
5.8 Classes contidas no nodo *Expression*. 63

RESUMO

Neste trabalho é proposta a LFL (*Language Features Library*), uma biblioteca para armazenar classes genéricas especificadas utilizando a Semântica de Ações Orientada a Objetos. A Semântica de Ações Orientada a Objetos é uma nova abordagem à especificação de linguagens de programação que propõe soluções a alguns problemas relacionados ao reuso e extensão de especificações em Semântica de Ações. É proposto neste trabalho a inclusão de algumas diretivas na Semântica de Ações Orientada a Objetos, de forma a propiciar o agrupamento e utilização das classes genéricas contidas na LFL, as quais serão usadas para a formação de várias linguagens de programação, inclusive linguagens de paradigmas distintos. Para exemplificar a utilização da LFL são apresentados dois estudos de caso, um especificando uma linguagem do paradigma imperativo e outro, especificando uma linguagem do paradigma orientado a objetos.

ABSTRACT

We demonstrate the use of LFL (*Language Features Library*), a library composed by generic specification classes written in Object-Oriented Action Semantics. Oriented Object Action Semantics is a new approach to Action Semantics in which reuse and extension of specifications are treated in an object-oriented way. We propose the inclusion of some new directives in Object Oriented Action Semantics, to deal with the grouping and use of generic classes, which are built into the LFL. These classes can be used for the description of programming languages, including languages of different paradigms.

CAPÍTULO 1

INTRODUÇÃO

A utilização de métodos formais para especificar linguagens de programação é de vital importância no projeto e na utilização de linguagens [10] [20]. A notação utilizada nestes métodos formais conduz os projetistas e implementadores a um melhor conhecimento da linguagem, permitindo também a eliminação de características indesejadas em um estágio inicial no processo de projeto da linguagem. Atualmente existem vários modelos (*frameworks*) de notação formal que são utilizados para especificar linguagens de programação. Dentre estes modelos, o *framework Action Semantic* (Semântica de Ações) [10] [12] [17] possui boas propriedades para especificar formalmente uma linguagem de programação [10] [11] [15] [18].

A Semântica de Ações, como característica, disponibiliza uma notação formal e apropriada a ser usada na descrição de linguagens. Esta semântica formal possui propriedades de reusabilidade e estensibilidade [4] [5]. Entretanto, esta notação não possui uma sustentação sintática para a definição de bibliotecas. Tentando solucionar este problema, foi proposto por *Carvilhe* [1] [2] o uso de conceitos de orientação a objetos para a definição de uma nova extensão da Semântica de Ações. Como extensão da Semântica de Ações, a Semântica de Ações Orientada a Objetos tenta resolver alguns aspectos problemáticos de reutilização de código em Semântica de Ações. Esta nova abordagem, como característica, utiliza a notação de ações padrão [10] para as especificações, combinada com a criação de classes para prover uma composição de especificações orientadas a objetos.

Uma especificação em Semântica de Ações Orientada a Objetos consiste na definição de uma hierarquia de objetos e classes, sendo que os métodos e os atributos são definidos por meio das funções semânticas e dos dados da Semântica de Ações. A hierarquia em uma determinada linguagem é derivada de sua estrutura sintática. Cada frase da linguagem é representada por um ou mais objetos.

1.1 Definição do Problema

Na formação de uma nova linguagem de programação são aplicados alguns conceitos e estruturas presentes em linguagens já existentes, tais como: declaração de variáveis, definição de comandos e expressões. Desta forma, constata-se que pode haver um reaproveitamento de partes de especificação de linguagens na criação de novos projetos. Para que isto seja possível, devem-se prover mecanismos que permitam a divisão da especificação em partes que possam ser reutilizadas e também estendidas.

Em Semântica de Ações Orientada a Objetos foi proposta a aplicação de conceitos de orientação a objetos em Semântica de Ações, assim foram definidos conceitos para a formação de classes e métodos. Estas classes são constituídas por diretivas (destinadas a definir hierarquias e relacionamentos entre classes) e por equações semânticas (destinadas a definir o comportamento de frases da linguagem). Em um projeto que utiliza a Semântica de Ações Orientada a Objetos para especificar uma linguagem, novas classes são criadas e algumas são reutilizadas de outros projetos. Desta maneira, há um reaproveitamento de classes que pertencem a paradigma distintos, por exemplo, uma classe do paradigma imperativo pode ser utilizada em um projeto do paradigma orientado a objeto.

Esta reutilização é uma característica relevante em Semântica de Ações Orientada a Objetos. Atualmente o reaproveitamento destas classes é aplicado pela cópia da mesma no ambiente do projeto atual. Esta forma de reutilização, conforme os conceitos de orientação a objetos [8] [16] [19], não é considerada ideal, pois: (1) a localização de uma determinada classe fica inviável quando se tem vários projetos; (2) as alterações que são feitas na classe X que está no projeto A, não refletirá na mesma classe X do projeto B, vice-versa; (3) e o esforço de depuração está distribuído entre os vários projetos. Desta forma, o reaproveitamento de classes fica comprometido por não haver uma biblioteca que acomode classes que são consideradas de uso genérico. Na próxima seção será proposta uma solução para este problema.

1.2 Objetivos

Em uma especificação de linguagens de programação, várias classes são criadas e reutilizadas. Com o crescimento na quantidade destas classes, tem-se a necessidade de adequá-las a uma estrutura organizacional, que facilitará sua localização e uso. Está sendo proposto neste trabalho, um modelo para criação de uma biblioteca de classes, a qual será denominada por **LFL** - ***Language Features Library***. A função da LFL é reunir e organizar as classes genéricas (especificação similar) em uma estrutura. Funcionará como um repositório de classes, disponibilizando formas de acesso a estas classes.

A disposição das classes genéricas na LFL facilita o agrupamento de novas classes, bem como a utilização das mesmas. Quanto à manutenção destas classes, a mesma é facilitada, pois modificando uma única classe, as alterações serão refletidas em todas as especificações que fazem uso da classe. Será adotada uma estrutura de árvore como forma para organizar as classes na LFL, onde os nodos desta árvore representam os níveis de hierarquia organizacional e as folhas são as classes.

Com o intuito de validar os conceitos definidos para a LFL, serão estabelecidos dois estudos de caso que consistem na especificação em Semântica de Ações Orientada a Objetos de uma linguagem imperativa e outra, uma linguagem orientada a objetos. Nestas especificações, serão utilizadas algumas classes que estão agrupadas na LFL. Também serão incluídas e alteradas outras classes. Serão especificadas as linguagens μPascal e JOOS [18]. A primeira, é uma linguagem imperativa que contem declarações de variáveis e constantes, comandos e expressões. A outra linguagem é orientada a objetos, contendo os principais conceitos do paradigma, como classe, herança, métodos e construtores [8].

Além disso, está sendo proposta uma extensão da linguagem μPascal, visando adicionar procedimentos e funções à estrutura da linguagem original. Com a extensão desta linguagem, pode-se analisar o comportamento de inclusão de novas classes à linguagem e na LFL.

1.3 Estrutura desta Dissertação

Este trabalho é composto por 8 capítulos. No capítulo 2 é descrita a Semântica de Ações. O capítulo 3 faz uma breve introdução aos conceitos de orientação a objetos. No capítulo 4 será apresentada a Semântica de Ações Orientada a Objetos, a qual é o formalismo base utilizado neste trabalho. A LFL - *Language Features Library* está apresentada no capítulo 5. Como estudo de caso, o capítulo 6 mostra a especificação da linguagem μPascal e sua extensão para adicionar procedimentos e funções. No capítulo 7 a linguagem JOOS, é especificada. As conclusões e trabalhos futuros estão indicados no capítulo 8.

CAPÍTULO 2

SEMÂNTICA DE AÇÕES

2.1 Introdução

Métodos formais têm sido utilizados para prover especificações de linguagens de programação. A especificação formal de linguagens de programação auxilia na elaboração de programas corretos, evita ambigüidades e falhas no entendimento por parte dos projetistas, implementadores e usuários da linguagem.

Criada por Peter Mosses com a colaboração de David Watt, a Semântica de Ações [10] surgiu a partir da Semântica Denotacional [14] [17]. A Semântica de Ações é um formalismo utilizado para descrever a semântica de linguagens de programação, representando os significados de um programa através de uma entidade matemática denominada *ação*. Quando as *ações* são executadas, as informações são processadas gradualmente.

Uma especificação em Semântica de Ações é composta por: *sintaxe abstrata*, que representa a formação das frases da linguagem (declarações, comandos, expressões); e *semântica*, que representa o significado das frases (números, procedimentos).

A Notação de Ações (*Action Notation*) expressa a semântica de um programa pela utilização de um conjunto de possíveis ações, que são representadas por palavras da língua inglesa, p.ex. `check it and then complete`.

As entidades semânticas definem os tipos de dados usados pela linguagem e suas operações. Elas são classificadas em três tipos: as ações (*actions*), os dados (*data*) e os produtores (*yielders*).

2.2 Notação de Ações

A notação de ações (*action notation*) é formada por símbolos que descrevem gramaticalmente a Semântica de Ações. Estes símbolos são palavras da língua inglesa, os quais

permitem uma fácil leitura, sem comprometer o formalismo aplicado.

Para descrever uma linguagem de programação é necessário utilizar estes símbolos e também formar frases. Existem diferenças verbais na formação das frases para cada entidade semântica, as quais estão descritas abaixo [10]:

- Entidade *Action*: as frases estão no verbo imperativo, envolvendo conjunções e advérbios;

  Exemplo:
  ```
  check it and then escape
  ```

- Entidades *Data* e *Yielders*: são formadas por frases substantivas;

  Exemplo:
  ```
  the items of the given.
  ```

As frases formadas no artigo definido e indefinido possuem uso comum pelas entidades semânticas.

Exemplo:
```
choose a cell then reserve the given cell.
```

2.2.1 Notação de Dados

A Notação de Dados (*Data Notation*), em adição à notação de ações, é usada para descrever as informações que são processadas pelas ações. Estas informações são classificadas em uma coleção de tipos de dados abstratos, incluindo números, caracteres, valores lógicos, string, tuplas, listas, árvores, conjuntos e mapeamentos[1]. Para cada tipo de dado há uma notação padrão, que auxiliará o especificador na descrição da semântica de um programa. A descrição completa para cada tipo está apresentada no capítulo 5 de Mosses [10]. A seguir constam algumas definições:

[1]Mapeamento é o relacionamento de um conjunto de itens de dados a outros, por exemplo, a associação de um identificador à uma célula em memória.

- Números
  - **natural**: conjunto dos números naturais;
  - **integer**: conjunto dos números inteiros;
  - **sum** x y: retorna o somatório de x y;
  - **successor** x: retorna $x + 1$.

- Valores Lógicos
  - **not** x: retorna a negação lógica de x;
  - **both** x y: retorna o resultado da operação lógica *"and"* entre x e y.

- Strings
  - **uppercase** x: retorna a string de x convertendo as letras em maiúsculo;
  - **lowercase** x: retorna a string de x convertendo as letras em minúsculo.

2.3 Classificação das Informações

As informações que são manipuladas em uma linguagem de programação, necessitam ser especificadas na definição semântica da linguagem. Dados como números naturais, números inteiros, caracteres, listas, strings, tuplas, mapeamentos, dentre outros, são classificados como *sorts*. Em Semântica de Ações, *sorts* podem ser definidos como um conjunto de elementos com operações relacionadas, sendo que a união desses conjuntos forma a Notação de Dados (*Data Notation*).

Cada *sort* inclui automaticamente um valor especial denominado *nothing*, o qual representa o *sort* vazio. Um *sort* com valor *nothing*, pode representar o resultado de alguma operação ou ação que não terminou de forma normal [15] ou simplesmente um valor indefinido. Quando um *sort* é um *sub-sort* de outro, todos os elementos do primeiro também são elementos do segundo.

Exemplo:

- O número 6 é um sub-sort do *sort integer.*

  Pode ser representado por 6 $\leq$ inteiro;

- O valor *true* é um sub-sort do *sort truth-value.*

  Representa-se escrevendo *true* $\leq$ *truth-value.*

Quando uma ação é executada, informações podem ser disponibilizadas a outras ações. Estas informações são classificadas como *transients*, *bindings* e *storage*, as quais estão detalhadas a seguir.

2.3.1 *Transient*

A informação transitória (*transient*) representa os dados que são passados de uma ação para a outra, podem ser utilizados imediatamente ou simplesmente serem perdidos, dependendo do contexto aplicado. As informações recebidas pela ação, serão manipuladas e disponibilizadas para uma outra ação.

Exemplo:	*transient*	*binding*	*storage*
`| give 10`	(10)	( )	( )
`and`			
`| give successor(the given integer)`	(11)	( )	( )

A ação `give 10` produz o dado *transient* de valor 10, que será usado na próxima ação. As ações `give` e `sucessor(the given integer)` alteram a informação *transient* para seu sucessor, neste caso será 11. Neste exemplo, apenas existe o fluxo de informação *transients*. Ações e combinadores são detalhados na seção 2.5.

2.3.2 *Binding (Scoped)*

Os *bindings* são associações de identificadores a dados com a finalidade de vincular um nome àquele dado. Esses dados podem ser propagados durante a execução de uma ação ou tornarem, temporariamente, ocultos em uma sub-ação.

Exemplo:	*transient*	*binding*	*storage*
`| bind "x" to 1`	( )	(x→1)	( )
`before`			
`| give the given integer bound to "x"`	(1)	( )	( )

A ação `bind "x" to 1` mapeia o identificador `x` ao dado 1. Desta forma, é produzido um *binding*. Na ação `give the given integer bound to "x"` é produzido o dado *transient*, sendo o valor mapeado ao identificador `"x"`, no caso o inteiro 1. Ações e combinadores são detalhados na seção 2.5.

2.3.3 *Storage (Stable)*

São os valores em memória, como módulos de dados, que ficam armazenados em células para a definição da especificação de linguagem. A mudança no armazenamento ocorre durante a execução da ação. Para usar inicialmente um espaço em memória, é necessário efetuar sua alocação [10].

Exemplo:	*transient*	*binding*	*storage*
`| give true`	(true)	( )	( )
`and`			
`| allocate a cell`	(true, cell)	( )	(cell→uninitialized)

A ação `give true` produz o dado *transient* de valor `true`, que será disponibilizado à próxima ação. A ação híbrida [10] `allocate a cell`, aloca uma posição de memória livre e retorna esta posição como dado transitório (*transient*). Para exemplo, o valor da posição alocada é o dado transitório `cell`, que será disponibilizado juntamente com o anterior. Esta ação também gera uma alocação de memória (*storage*) com valor *uninitialized*. Isto significa que existe um espaço de memória alocado, porém não inicializado.

2.4 Entidades Semânticas

As entidades semânticas definem os tipos de dados usados pela linguagem e suas operações. Elas analisam e, se necessário, aplicam regras previamente definidas modelando os dados iniciais. A Semântica de Ações utiliza apenas três classes de entidades [10], as quais estão descritas abaixo:

• ***Actions*** (Ações): são entidades dinâmicas capazes de modelar o comportamento dos programas, atuando em estruturas de controle e no fluxo de dados. Quando uma *Ação* é executada, ocorre o recebimento de informações que são processadas e geram novos dados para o ambiente, refletindo em uma mudança no estado computacional. Ações, quando executadas, podem produzir os seguintes resultados:

- **complete**: terminação normal. Houve sucesso na execução;
- **escape**: corresponde ao término anormal. Parte da ação não é executada;
- **fail**: corresponde ao abandono da execução da ação. Todas as possibilidade de executar o procedimento correto falham;
- **diverge**: a execução da ação não finalizou.

Exemplo:	*transient*	*binding*	*storage*
`| allocate a cell`	(cell)	( )	(cell→uninitialized)
`then`			
`| bind "x" to it`	()	(x→cell)	(cell→uninitialized)

No exemplo acima, o resultado final é ***completing***, pois todas ações envolvidas tiveram sucesso em suas execuções.

As ações podem ser classificadas como *primitivas* e *compostas*. As execuções de ações primitivas, produzem resultados previamente conhecidos, como **complete**, **fail** e **diverge**. As ações compostas são constituídas pelo uso de combinadores de ações (*action combinator*). Estas, podem ser seqüenciadas, intercaladas ou alternativas. Uma ação seqüenciada, corresponde a uma sub-ação ser executada antes de outra, ou seja, só ocorrerá uma nova ação quando uma anterior tiver sido executada. Em ações compostas, interlacadas, suas sub-ações são executadas em ordem aleatória. Também pode ocorrer a exclusividade de execução, onde apenas uma das sub-ações é escolhida para a execução [10].

Quando uma ação é executada, ela pode receber e produzir dados de diversos tipos. Estes dados podem ser classificados como: *transients*, são passados imediatamente entre as ações; *bindings*, disponíveis a uma ação podem ser repassados às sub-ações; e *stable*, que podem ser lidas e modificadas.

Ações e combinadores são classificados de acordo com a informação processada, como as seguintes facetas denotam:

- *Faceta Básica*: trata do fluxo de controle;
- *Faceta Funcional*: corresponde as ações que processam dados transitórios (*transients*);
- *Faceta Declarativa*: corresponde as ações que geram ou recebem *bindings*;
- *Faceta Imperativa*: ações que consultam ou modificam informações em memória (*stable*).
- *Faceta Reflexiva*: ações que estão relacionadas com o encapsulamento de ações;

Na seção (2.5) é detalhado o funcionamento de cada faceta.

• ***Data*** (Dados): são entidades estáticas. Os Dados são compostos por vários itens que estão contidos nas informações processadas pelas entidades Ações. Estes itens são organizados em estruturas que regulam seu acesso. Dados podem ser entidades matemáticas como: valores lógicos, números, caracteres, strings, mapeamento e listas. Também podem ser formados por entidades como tokens[2] e mensagens.

• ***Yielders*** (Produtores): são entidades que podem produzir dados durante a execução de uma ação. Estes dados são denominados *yielded data* (dados produzidos).

Durante a execução de uma ação, certa informação corrente é mantida implícita e seu tipo pode ser avaliado por consistência. Na avaliação por consistência é verificado o tipo da informação atual com o tipo esperado. O dado não pode sofrer qualquer tipo de alteração.

Abaixo constam os tipos de informações que são mantidos implícitos[3].

- Os dados transitórios (*transients*) correspondem ao resultado intermediário entre ações;
- Os *bindings* de uma ação consistem no mapeamento dos dados com seus respectivos identificadores; e

[2]Token é a representação de um identificador.

[3]Detalhes sobre tipos de informações pode ser encontrado na página 7.

– O dado que está atualmente armazenado em memória.

2.5 Facetas

O comportamento das ações é classificado em facetas, de acordo com os tipos de informações por elas processadas. A junção de ações e *combinadores*[4] cria facetas com ações mais complexas, além de mecanismos para manipular e gerar dados a partir de um tipo específico de informação. As facetas têm uma grande relevância contextual, pois modelam o comportamento das linguagens de programação, impondo caminhos para a aplicação de regras que manipulam a informação.

Nas próximas seções é descrito brevemente o funcionamento das principais facetas das ações, baseadas em [10] [15] [17]. A descrição completa está no livro de Mosses [10].

As principais facetas são:

- **Faceta Básica**: trata o fluxo de controle;
- **Faceta Funcional**: ações que processam dados transitórios (*transients*);
- **Faceta Declarativa**: ações que geram ou recebem *bindings*(*scoped*);
- **Faceta Imperativa**: ações que manipulam informações em memória (*stable*);
- **Faceta Reflexiva**: ações que estão relacionadas com o encapsulamento de ações;

2.5.1 Faceta Básica

A faceta básica está relacionada com as ações que controlam o fluxo em sua execução. A execução de uma ação acarreta em algum tipo de comportamento, sendo a ação uma seqüência de passos atômicos que devem ser ordenados. Esta faceta, trata os vários tipos de terminações e confirmações que modelam o comportamento da ação.

[4]Os combinadores de ações determinam o controle e fluxo de dados entre ações. Na ação A_1 *and* A_2, o combinador é *and*.

A seguir estão relacionados os principais tipos de terminações que podem ocorrer a partir da execução de ações:

- ***complete***: termina normalmente, houve sucesso na execução.
- ***escape***: indica término anormal.
- ***fail***: quando a ação falha, término anormal da execução.
- ***diverge***: é uma ação que nunca termina.
- ***commit***: efetua a alteração do valor estável ou permanente processado pela ação. Terminação normal.
- ***unfold***: é uma ação usada conjuntamente com *unfolding* para desviar o fluxo de controle ao início deste combinador.
- ***unfolding*** A: a ação A substituirá o combinador *unfold* quando este for encontrado.
- ***indivisibly*** A: a ação A é executada em apenas um único passo. Esta ação não está intercalada com outras ações que são executadas pelo mesmo agente, assegurando também que a ação A não possa ser interrompida pelas ações *escape* ou *fail* ocorridas fora da ação.

Logo abaixo, estão descritos os combinadores que fazem parte da Faceta Básica:

- A_1 ***or*** A_2: estabelece uma escolha não determinística de uma das ações para ser executada. A escolha para execução da ação é aleatória, sendo que a outra ação só é executada quando a escolhida falha. No caso da ação escolhida efetuar um *commit* e seu resultado for *fail*, a outra ação não será executada. A ação composta falha se ambas falharem e completa se uma delas completar.
- A_1 ***and*** A_2: a execução das ações ocorre de forma intercalada, ou seja, a escolha de execução não é determinística. Os dados transitórios e os *bindings* são distribuídos de forma paralela para ambas as ações e as informações de resultado são associadas formando um novo conjunto de dados transitórios e *bindings*.

- A_1 ***and then*** A_2: corresponde a execução seqüencial das ações. A ação A_2 só é executada quando a ação A_1 terminar. Os dados transitórios e os *bindings* são distribuídos de forma paralela para ambas as ações e as informações de resultado são associadas formando um novo conjunto de dados transitórios e *bindings*.

- A_1 ***trap*** A_2: este combinador permite a execução da ação A_2 quando a ação A_1 escapar (*escape*).

2.5.2 Faceta Funcional

A faceta funcional trabalha com ações que envolvem dados transitórios. Dados transitórios representam os valores intermediários processados durante a execução de uma ação, podendo ou não serem transferidos a outra ação.

As principais ações e produtores da faceta funcional são:

- ***give*** _ : resulta em informação transitória com os dados fornecidos pelo produtor.

- ***escape with*** _ : sai da atual ação, retornando como informação transitória os dados fornecidos pelo produtor. Esta ação é a abreviação de ***give*** *Y* ***then escape***. Terminação excepcional.

- ***regive*** _ : propaga toda informação transitória recebida. É a abreviação de ***give the given data***.

- ***choose*** _ : esta ação efetua uma escolha não determinística de um elemento do *sort*, retornando esta referência como informação transitória.

- ***check*** _ : verifica se o dado fornecido pelo produtor é verdadeiro ou falso.

Na faceta funcional existem os combinadores *and*, *then*, *and then*, *or* e *trap* [10]. A seguir é descrito o funcionamento destes combinadores, exceto o *trap* que será descrito na seção 2.5.1.

Combinador A_1 ***and*** A_2 - os *transients* são distribuídos de forma paralela, sendo associados uniformemente ao término da execução. A escolha da ordem de execução da ação não é determinística. A figura 2.1 ilustra o fluxo das informações transitórias.

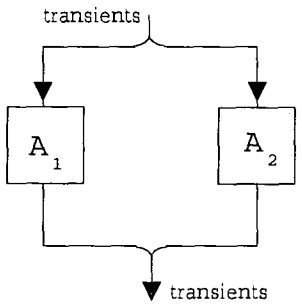


Figura 2.1: Combinador A_1 *and* A_2.

Combinador A_1 ***then*** A_2 - a primeira ação é executada usando os dados transitórios recebidos, que posteriormente serão passados à segunda ação. Os dados transitórios e as ações fluem de forma seqüencial e determinística. A informação transitória final será a produzida pela segunda ação.

A figura 2.2 ilustra o fluxo das informações transitórias.

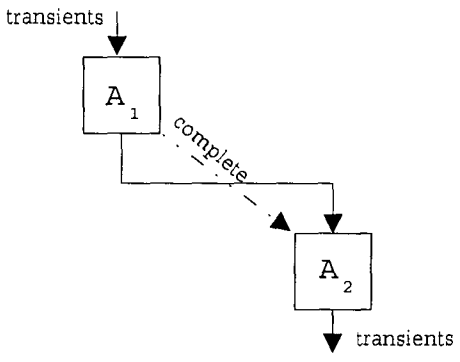


Figura 2.2: Combinador A_1 *then* A_2.

Combinador A_1 ***and then*** A_2 - os *transients* são distribuídos de forma paralela para ambas ações. Ao final do processo, os *transients* são mesclados e determinados pela ação

combinada. A execução da ação é seqüencial, sendo que, A_2 é executado apenas quando A_1 terminar em processo normal (*complete*).

A figura 2.3 ilustra o fluxo das informações transitórias.

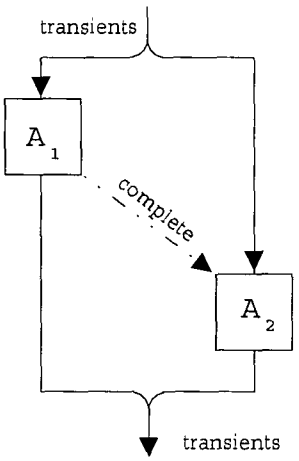


Figura 2.3: Combinador A_1 *and then* A_2.

Combinador básico A_1 ***or*** A_2 - os *transients* são distribuídos de forma paralela para ambas as ações. Uma das ações A_1 ou A_2 é escolhida ao acaso e somente se esta ação falhar a outra será executada. A informação transitória final será a produzida pela última ação executada.

A figura 2.4 ilustra o fluxo das informações transitórias.

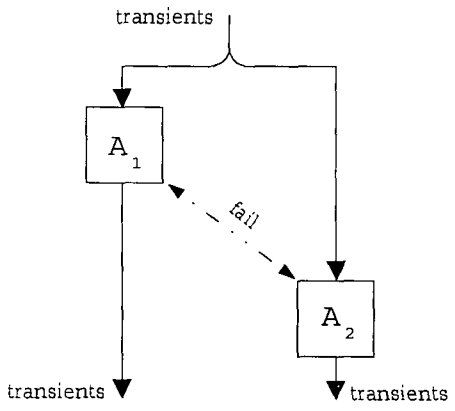


Figura 2.4: Combinador A_1 *or* A_2.

A seguir é mostrado em um exemplo, a aplicação de algumas ações da faceta funcional.

Exemplo:

A ação a seguir calcula a soma de duas expressões. As informações contidas nas expressões E_1 e E_2 são avaliadas produzindo dados rotulados por 1 e 2, respectivamente. Esses dados são passados à próxima ação que produz a soma.

```
evaluate ⟦  E1 "+" E2 ⟧ =
  | | evaluate E1
  | and then
  | | evaluate E2
  then
  | give sum(the given integer♯1, the given integer♯2)
```

2.5.3 Faceta Declarativa

A faceta declarativa trabalha com ações relacionadas ao processamento de informações com escopo, que geralmente são propagadas além das informações transitórias. A informação com escopo representa a associação (*binding*) de identificadores a dados. Estes dados, são valores que estão armazenados em memória. Os identificadores possuem o mesmo comportamento daqueles usados em programas, sendo representados por *strings* de caracteres. As ações possibilitam manipular as informações armazenadas em memória. A seguir estão descritas as principais ações e produtores:

- ***bind*** T ***to*** Y: a informação resultante da avaliação do dado produtor Y associada ao token T.

- ***rebind***: representa a propagação de todos os *bindings* para o próximo escopo. É a abreviação de ***produce the current bindings***.

- ***unbind*** T: desfaz a associação relativa ao idenficicador T. É a abreviação de ***bind*** T ***to unknown***.

- ***furthermore*** A: produz os mesmos *bindings* de A, atualizados com os *bindings* da ação. É a abreviação de ***rebind moreover*** A.

Na faceta declarativa, além dos combinadores *and*, *then* e *and then* [15], existem os combinadores *moreover*, *hence* e *before* [10].

O fluxo dos *bindings* para os combinadores *and* e *and then* é distribuído de forma paralela, sendo associados uniformemente ao término da execução. A diferença entre ambos combinadores é que o *and* flui de forma paralela e *and then* seqüencial. A figura 2.5 ilustra o fluxo dos dados transitórios e *bindings*.

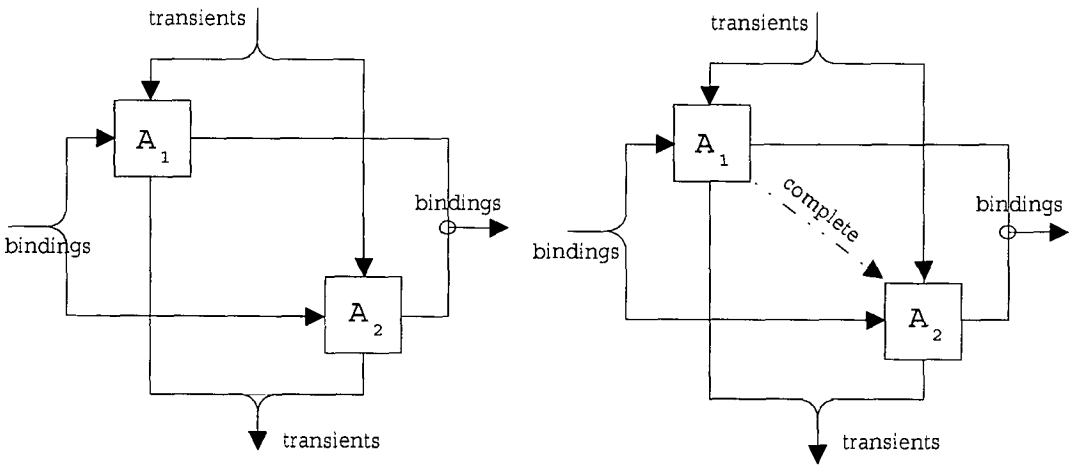


Figura 2.5: Combinadores (A_1 *and* A_2) e (A_1 *and then* A_2).

O combinador de ação *then* possui o mesmo comportamento, com relação aos *bindings*, do combinador *and then*. A figura 2.6 ilustra o fluxo dos dados transitórios e *bindings*.

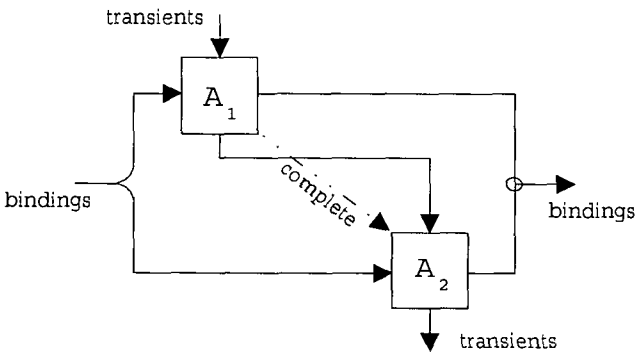


Figura 2.6: Combinador A_1 *then* A_2.

Combinador A_1 ***moreover*** A_2 - os *bindings* e dados transitórios são passados paralelamente para ambas ações, sendo estas executadas de modo intercalado. Ao término da execução, os *bindings* produzidos pelas ações são unidos, prevalecendo os gerados pela segunda ação. A figura 2.7 ilustra o fluxo dos dados transitórios e *bindings*.

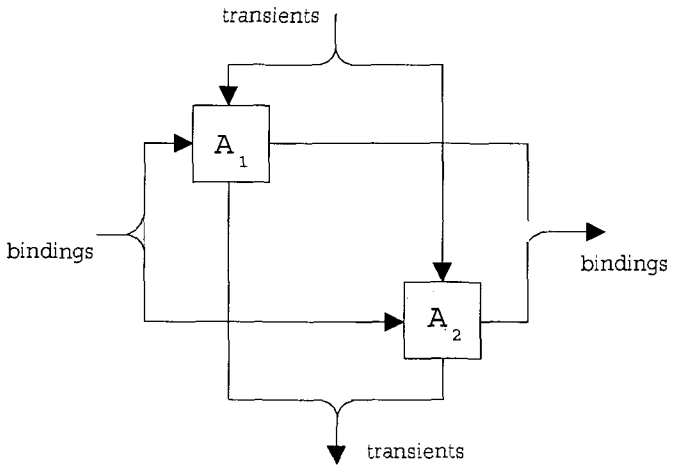


Figura 2.7: Combinador A_1 *moreover* A_2.

Combinador básico A_1 ***hence*** A_2 - as ações são executadas sequencialmente, sendo que da mesma maneira fluem os *bindings*. A ação A_1 propaga os *bindings* para A_2 que produz a informação final. Os dados transitórios fluem paralelamente havendo uma concatenação ao término do processo. A figura 2.8 ilustra os dados transitórios e *bindings*.

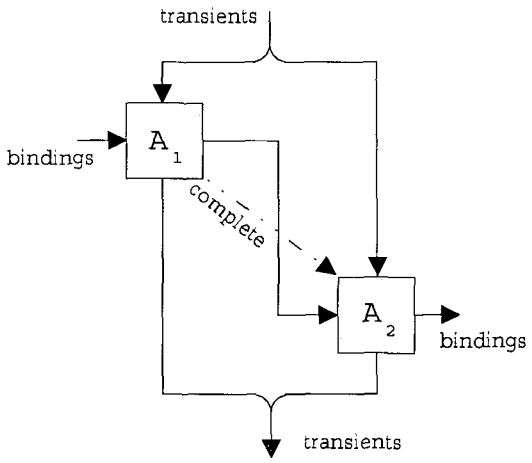


Figura 2.8: Combinador A_1 *hence* A_2.

Combinador A_1 ***before*** A_2 - as ações são executadas sequencialmente, sendo que A_2 é executado apenas quando A_1 terminar em processo normal (*complete*).

Os *bindings* produzidos pela ação A_1 são propagados para a ação A_2, mesclando com os *bindings* recebidos do ambiente (*bindings* originais). Ao término da execução, os *bindings* produzidos pelas ações são unidos, prevalecendo os gerados pela segunda ação.

Os *transients* são distribuídos de forma paralela para ambas ações. Ao final do processo, os *transients* são mesclados e determinados pela ação combinada. A figura 2.9 ilustra o fluxo dos dados transitórios e *bindings*.

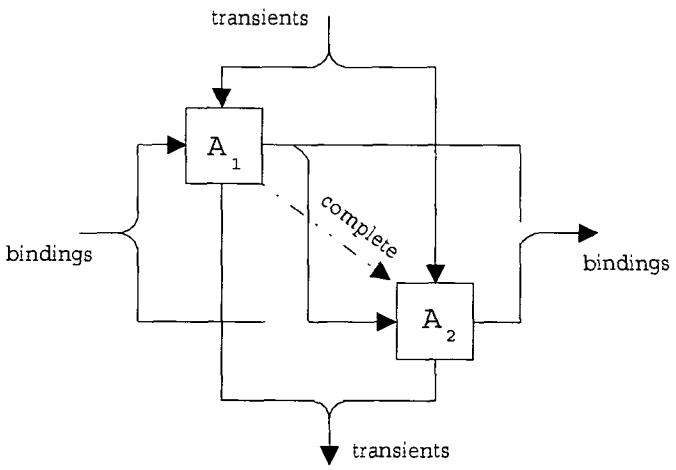


Figura 2.9: Combinador A_1 *before* A_2.

A seguir é mostrado em um exemplo, a aplicação de algumas ações da faceta declarativa.

Exemplo:

O exemplo abaixo mostra a especificação formal para a declaração de uma constante. A ação `evaluate` avalia a expressão *E* e produz o dado associado a expressão. Em um novo ambiente, através da ação `bind`, o identificador é mapeado ao dado.

```
elabore [[ "const" E : Expression ]] =
  | | evaluate E
  | then
  | | bind E to the value♯1
```

2.5.4 Faceta Imperativa

As ações da faceta imperativa interagem com a memória (*storage*), podendo efetuar operações de leitura, modificações por meio da alocação e liberação de células e, pelo armazenamento de informações.

As células são independentes umas das outras, logo, qualquer operação atinge apenas a célula escolhida. As atualizações realizadas são definitivas, irreversíveis, portanto para obter o valor anterior, antes de efetuar a modificação, é necessário criar uma cópia.

A memória é formada por um conjunto potencialmente infinito de células, que são classificadas como elementos do tipo *cell*. O valor contido em uma célula é definido formalmente como elemento do tipo *storage*, que representa o mapeamento da célula para o valor armazenado.

Uma célula pode estar no estado alocado ou desalocado. Quando alocado pode armazenar dados. Um outro valor que pode ser vinculado a uma célula é o *uninitialized*, que indica a ausência de informação armazenado na célula.

Em seguida, constam ações e produtores responsáveis pela manipulação de informações estáveis (*storage*):

- ***allocate a cell***: escolhe uma célula dentre as células não alocadas e faz sua reserva. Esta célula reservada é passada para ação como dado transitório. Esta ação é uma abreviação da seguinte ação híbrida [10]:

```
indivisibly
  | | choose a cell [not in the mapped-set of the current storage]
  | then
  | | reserve the given cell and give it
```

- ***desallocate*** *C*: esta ação irá desalocar a célula de memória representado por *C*. Caso a memória não esteja alocada, a ação falha.

- ***store*** *E* ***in*** *C*: o valor de *E* é avaliado produzindo um dado simples que é armazenado na célula de memória *C*. A ação efetua a verificação do valor existente na expressão *E*, que deve fazer parte do sorte *storable*. Também é verificado se a célula esta alocada. Se uma dessas condições não forem satisfeitas, a ação falha.

- ***unstore*** *C*: remove os dados armazenados na célula. Se a célula não estiver alocada, a ação falha.

- ***reserve*** *Y*: reserva a célula fornecida pelo produtor *Y*, desde que esta célula não esteja reservada.

- ***unreserve*** *Y*: desaloca a célula fornecida pelo produtor *Y*, desde que esta célula esteja reservada.

- ***current storage***: resulta no mapeamento do atual armazenamento recebido pela ação.

A seguir é mostrado em um exemplo, a aplicação de algumas ações da faceta imperativa.

Exemplo:

O exemplo especifica a declaração de uma variável com armazenamento de um dado. Inicialmente é efetuada uma alocação de memória. A seguir, é criado um novo ambiente contendo o *binding* do identificador *I* à célula. Por fim, o valor contido em *N* é armazenado na célula de memória referenciada pelo identificador *I*.

```
elabore ⟦ "var"  I:Identifier := N:Integer ⟧ =
    | allocate a cell
    then
    | | bind the given cell to I
    | and
    | | store valuation of N in the given cell
```

2.5.5 Faceta Reflexiva

As ações da faceta reflexiva consiste em tratar conceitos abstratos como se fossem reais ou objetivos. Estão relacionadas ao encapsulamento de ações como dados, desta forma

geram abstrações. Uma abstração é um item de dado o qual encapsula uma ação, que pode ser implementada como uma seqüência de instruções ou como um apontador para tal seqüência. As abstrações são usadas para representar construtores como os procedimentos e funções em linguagem de programação. A ação encapsuladada na abstração é geralmente executada em um contexto diferente daquele em que a própria abstração acontece [10].

A seguir estão descritas as principais ações e produtores:

- ***enact*** Y: ativa o encapsulamento na abstração produzida por Y. Em tempo de execução, quando os dados transitórios e os *bindings* não são incorporados, *closure* transita com valores vazio.

- ***abstraction of*** A: a partir da encapsulação da ação A é fornecido como resultado um dado.

- ***application*** A ***to*** Y: resulta em um dado a partir do encapsulamento da ação A, juntamente com o dado transitório proveniente como resultado fornecido pelo produtor Y.

- ***closure*** Y: resulta em um dado a partir do encapsulamento da ação, juntamente com a informação com escopo recebida.

É mostrado em um exemplo, a aplicação de algumas ações da faceta reflexiva.

Exemplo:

O exemplo especifica a construção de um procedimento (*procedure*). Inicialmente o identificador passado ao procedimento é mapeado, sendo incorporado a *procedure* quando esta entrar em execução. Um novo ambiente é criado e a abstração associada em I incorpora os dados transitórios e os *bindings* corrente, os quais são assegurados pelo uso de `closure of abstraction`. Quando o bloco `B` for executado, ele recebe os mesmos *bindings*, como aqueles da declaração. A abstração estará completa quando o bloco de instruções terminar sua execução.

```
elabore ⟦ "procedure" I:Identifier "is" B:Block ";"  ⟧ =
  | bind the token of I to
  | | parameterless procedure of closure of abstraction of
  | | | execute B
  | | | trap check there is given a procedure-return
```

2.6 Considerações Finais

A Semântica de Ações é um formalismo que pode ser utilizado para descrever a semântica de linguagens de programação. Uma especificação em Semântica de Ações é composta por *sintaxe abstrata* e *semântica*, representando respectivamente a formação das frases da linguagem e seu significado. A notação usada para a especificação é denominada *action notation*, a qual é formada por palavras da língua inglesa.

O significado de um programa é representado através de entidades matemáticas especiais denominadas *ações*. O resultado da execução de uma ação pode ser: *complete*, *escape*, *fail* ou *diverge*.

As funções e entidades semânticas representam o comportamento de cada parte do programa. Existem três classes de entidades na semântica de ações usadas em entidades semânticas: *actions*, *data* e *yielders*. A informação processada por uma ação, pode ser classificada como *transient*, *binding* e *stable*.

O comportamento das ações é classificado em facetas, de acordo com os tipos de informação por elas processadas. As facetas modelam o comportamento das linguagens de programação, definindo a aplicação de regras que manipulam a informação. As principais facetas são: **Básica**, trata do fluxo de controle; **Funcional**, processam os *transients*; **Declarativa**, gera ou recebe *bindings*; **Imperativa**, manipula informações do tipo *stable*; e **Reflexiva**, está relacionada com o encapsulamento de ações.

CAPÍTULO 3

ORIENTAÇÃO A OBJETOS

Da busca constante de melhorias nas linguagens e técnicas para o desenvolvimento de software, decorrem as transformações e evoluções de novos paradigmas.

Dentre várias evoluções, está o paradigma Orientado a Objetos. O conceito deste paradigma é aplicado em vários segmentos da informática, como ambientes de desenvolvimento, interfaces, linguagens de programação, banco de dados, etc.

A aplicação dos conceitos de Orientação a Objetos não é uma tarefa fácil, pois a interpretação de uma frase, em uma descrição textual de um sistema, depende da forma que é analisada. Como exemplo, a frase *"O caminhão estaciona no depósito e descarrega sua carga"*. Esta frase deve ser analisada da seguinte maneira: primeiro pensa-se nos objetos caminhão, depósito e carga, imaginando como eles seriam e procurando definir seu comportamento. Após isto, pensa-se no relacionamento entre estes objetos, como o caminhão se relaciona com o depósito e a carga, e como o depósito se relaciona com a carga. De modo geral, pode-se dizer que a Orientação a Objetos apresenta uma ênfase aos substantivos da frase.

Atualmente, vem crescendo o uso da Orientação a Objetos em vários segmentos da informática, devido à sensível redução no custo de manutenção e ao aumento na reutilização de código.

A redução no custo de manutenção, decorre de certas características, como a herança e o encapsulamento[1], que permitem efetuar modificações apenas no objeto desejado, sem alterar outros objetos. Desta forma, o processamento de cada objeto se torna independente, porém necessitam estar relacionados entre si para que ocorra troca de mensagens entre os processos (objetos).

A reutilização de código reduz o trabalho repetitivo, pois pode-se usar a mesma codi-

[1]Herança e Encapsulamento estão detalhados nas seções 3.2.2 e 3.2.3, respectivamente

ficação para atender várias requisições de objetos distintos.

Nas próximas seções serão abordados os princípios básicos do paradigma de Orientação a Objetos, sendo descritos suas definições conceituais, como podem ser aplicados, quando devem ser utilizados e principalmente por que devemos usar estes conceitos e quais são suas vantagens e desvantagens.

Os princípios básicos são: Classes, Atributos, Métodos, Objetos, Mensagens, Herança, Generalização, Especialização, Abstração, Polimorfismo e Encapsulamento.

3.1 Princípios da Orientação a Objetos

3.1.1 Modularização por Classe

Uma Classe é um termo técnico utilizado conceitualmente em Orientação a Objetos para representar um conjunto de dados estruturados que são caracterizados por propriedades comuns. É constituída como uma estrutura modular completa que descreve as propriedades estáticas e dinâmicas dos elementos manipulados por um programa [7] [13]. Também, pode-se definir classes como a descrição de um grupo de objetos por meio de um conjunto uniforme de atributos e serviços. Uma classe é um conjunto de objetos que compartilham as mesmas operações.

Adicionalmente, agrega-se a classe conceitos de *dependência* e *coesão*. A *dependência* indica a ocorrência de um relacionamento semântico entre dois ou mais elementos do modelo onde uma classe depende de alguns serviços de outra classe. O intuito é minimizar o uso de dependência para evitar impactos indesejáveis nas classes subjacentes, ou seja, evitar que uma classe seja erroneamente modificada. A *coesão* é uma medida de integridade conceitual de uma classe [7]. A maximização da coesão, neste caso, é essencial para garantir o agrupamento de operações existentes na classe, diminuindo assim grandes esforços em manutenção.

Uma classe é estática, sendo sua estrutura e comportamento comuns a todos os objetos que são relacionados. Um objeto possui uma identidade e suas características são definidas na classe. Esta é definida por:

- **Nome:** o nome de uma classe é sua identidade. Ele deve ser único dentro do contexto que está sendo usado. Em um mesmo projeto podem haver classes com o mesmo nome, porém estão criadas em contextos diferentes;
- **Atributos:** um atributo é um dado para o qual cada objeto tem seu próprio valor. Atributos são, basicamente, a estrutura de dados que vai representar a classe; e
- **Métodos:** métodos são declarados dentro de uma classe para representar as operações que os objetos pertencentes a esta classe podem executar. Um método é a implementação de uma rotina. Pode ser comparado a um procedimento ou a uma função das linguagens imperativas.

Exemplo:

O exemplo ilustra a modelagem de uma classe, que posteriormente poderá ser codificada em uma linguagem de programação. O nome atribuido à esta classe é PESSOA. Para a classe PESSOA, foram construídos os atributos: Codigo, que é do tipo *integer*; Nome, que do tipo *string* e DtNascimento, que é do tipo *date*. Estes atributos alocam dados de acordo com sua tipagem. Também foram criados vários métodos, que podem retornar valor ou não. O método getCodigo() não requerer parâmetro, sua função é retornar o valor que está associado ao atributo Codigo. O método setNome(), requer um parâmetro do tipo *string*, sendo sua função alocar ao atributo Nome o valor do parâmetro passado. O método calculaIdade(), requerer um parâmetro do tipo *date* e retorna um valor do tipo *integer*. Este método tem como finalidade calcular o número de anos a partir da data informada.

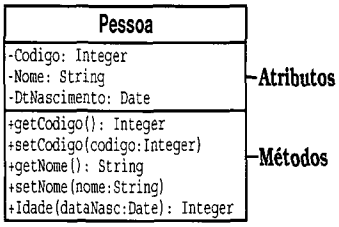


Figura 3.1: Modelagem de Classe.

3.1.2 Conceitos sobre Objetos

O que caracteriza a Orientação a Objetos são os objetos que podem representar algo que seja do mundo real ou abstrato, aparecendo como uma entidade autônoma que combina a representação da informação e sua manipulação. No mundo real os objetos podem ser algo como: carro, bicicleta, lápis, livro, etc.

Devido a aproximação que Orientação a Objetos tem do mundo real, existem facilidades na modelagem e programação de um sistema como um todo, mas pode haver exceções, visto que às vezes podem ocorrer problemas que são extremamente abstratos, como "o estilo de uma casa". Em problemas abstratos é difícil representar a estrutura lógica em torno de objetos, com isso, os objetos não são facilmente identificados.

Um Objeto é uma instância de uma classe que possui os atributos e métodos descritos pela classe. O instanciamento da classe cria um objeto que utiliza a estrutura (atributos e métodos) montada pela classe. Este objeto possui características como:

- **Identidade:** trata-se da propriedade que distingue os objetos um dos outros, determinando uma entidade única;
- **Estado:** são informações contidas no objeto, que representam o estado atual conforme a variação dos atributos; e
- **Comportamento:** é a forma em que o objeto age e reage devido a sua mudança de estado. Representa o método em uma classe.

Exemplo:

Para demonstrar a criação de um objeto, foi selecionado um trecho do código de uma linguagem de programação orientada a objetos. Na linha (1), a variável **data** é referenciada para o tipo *Date*, que é uma classe. Neste instante, ainda não há a criação do objeto. Em (2), a variável **data** é um objeto para a classe *Date*, pois a instrução *new Date()* instância o objeto. O método *getToday()*, na linha (3), a partir do objeto, irá retornar a data atual.

```
...
Date data;                                          (1)
data = new Date();                                  (2)
System.out.println(data.getToday());                (3)
...
```

Dentro de um objeto, alguns métodos e atributos podem ser privados ao objeto, sendo inacessíveis diretamente por qualquer elemento fora dele. Para um objeto referenciar seus elementos privados ele deve possuir métodos específicos que façam a operação desejada, ou seja, ele pode acessar seus atributos privados somente por meio de seus métodos. Isso consiste no encapsulamento de dados que está descrito na seção 3.2.3.

Assim, pode-se diferenciar claramente uma classe de objeto. A classe é apenas um tipo de dado, que somente representa características comuns a determinados objetos, podendo ser comparada com uma estrutura, com apenas uma forma definida, mas nenhuma variável que a manipula. Para manipulá-la é preciso definir uma variável. Esta variável do tipo da classe é que se chama objeto.

Os objetos de uma determinada classe não são iguais. Por exemplo, podemos ter os objetos João e Pedro da classe Aluno, cada um terá um nome, telefone, número, notas, e uma posição na memória. A sua similaridade consiste apenas no fato de que possuem propriedades idênticas.

Algo importante de um objeto é seu ciclo de vida, que engloba o momento em que é criado até sua eliminação. A partir de sua criação, o objeto está pronto para ser usado. Sua disponibilidade é verdadeira até sua eliminação, a qual pode ser de duas formas:

- Primeira forma: eliminando o objeto no final do programa se ele for global, no final de um módulo se for local, e no final de um bloco se for criado dentro deste.

- Segunda forma: através da *Garbage Collection*. Esta maneira de eliminação não é implementada em todas as linguagens e não é uma característica somente de orientação a objetos. O *Garbage Collection* consiste em eliminar o objeto após sua última utilização. A partir do momento em que o objeto não é mais referenciado, ele deixa de existir [16].

Para aplicar os conceitos de Orientação a Objetos, primeiramente devem ser identificados os objetos contidos no problema. Posteriormente devem ser criados seus atributos e métodos.

3.1.3 Abstração

A abstração pode ser aplicada a uma classe ou a um método. O uso da abstração é um artifício empregado quando a classe ou o método possui representação diferente dentro de um contexto homogêneo. Por exemplo, a palavra substantiva "comida", representa uma abstração, pois as pessoas não comem comida, por que o que pode ser comido são instâncias tais como maçã, chocolate, pão, mamão, etc. A *Comida* representa o conceito abstrato de coisas que todas as pessoas podem comer. Não faz sentido existir uma instância de comida.

Uma classe abstrata é a modelagem de um conceito abstrato sem poder criar uma instância disto. O uso desta classe só é empregado pela criação de uma subclasse, ou seja, sempre uma classe abstrata será superclasse.

Um método abstrato pode ser modelado dentro de uma classe não abstrata, porém suas subclasses herdam este método. O método abstrato, na classe pai, é apenas declarado não havendo implementação, o qual poderá ser modelada nas subclasses.

Geralmente, a abstração é usada quando ainda não existe uma definição plena do que deseja-se construir. Isto torna a aplicação bastante volátil, pois cada subclasse pode implementar um código diferente à um mesmo assunto.

3.1.4 Polimorfismo

Polimorfismo é a propriedade de uma ou mais classes responderem a mesma mensagem, cada uma de uma forma diferente [16]. Ele pode também representar vários comportamentos que uma mesma operação pode assumir [13].

Em algumas linguagens de programação, como o Java [6], o polimorfismo é implementado pelo uso de sobrecarga dos métodos. Dois ou mais métodos podem compartilhar o mesmo nome, contanto que as seus parâmetros sejam diferentes. A sobrecarga de métodos

é importante pelo fato de ajudar a simplificar o entendimento do software.

O operador "+" é um exemplo de polimorfismo implementado mediante a sobrecarga do operador. Ele pode somar números inteiros, matrizes, números decimais, concatenar strings, etc., sem a necessidade de mudar o seu nome. Desta forma, se torna mais fácil usar o método "+" para efetuar a mesma operação com informações diferentes.

3.2 Técnicas de Orientação a Objetos

3.2.1 Envio de Mensagem

Uma mensagem é criada pela comunicação entre objetos. Esta comunicação é constituída por uma requisição enviada de um objeto a outro solicitando algum serviço, o qual será executado por um método[2]. No objeto receptor da mensagem, os métodos serão executados baseados nos dados disponibilizados, seguindo a hierarquia de classe. Após o método ter processado a informação, é gerada uma mensagem contendo o resultado que é enviada ao objeto solicitante.

Os métodos podem ter parâmetros de entrada e saída com tipos determinados. Esta característica, juntamente com o seu nome, definem a assinatura de uma mensagem.

3.2.2 Herança

O mecanismo de herança permite a uma nova classe utilizar atributos e métodos de uma classe existente, permitindo assim criar classes complexas, proporcionando a reusabilidade e reimplementação destes atributos e métodos. A nova classe simplesmente herda em seu nível básico as características de um antepassado na hierarquia de classe, sendo que a profundidade desta hierarquia será definida conforme a necessidade. Uma subclasse[3] herda também todas as dependências de relacionamento que a superclasse[4] poderia ter com outras classes.

Os atributos e métodos são herdados por todas as classes dos níveis mais baixos.

[2]A execução de um método pode envolver mais de um objeto.
[3]Subclasse é a classe que está, hierarquicamente, abaixo de outra.
[4]Superclasse é a classe que está, hierarquicamente, acima de outra.

Quanto mais baixo uma classe estiver na hierarquia, mais especializado é seu comportamento. O termo superclasse refere-se à classe mais generalizada, também conhecida como classe base ou classe pai. Subclasse representa a classe mais especializada, também pode ser denominada classe filha ou classe derivada.

O conceito de herança é um dos aspectos importante da modelagem de objetos, pois provê uma forma de construir componentes que poderão ser reutilizados, objetivando simplificar a definição das classes com características semelhantes.

Partindo do conceito de herança, surgem as classes abstratas[5], as quais são construídas como modelo para serem utilizadas por subclasses. As subclasses podem implementar estas classes abstratas adicionando atributos e métodos. O uso de classes abstratas, permite especificar atributos e operações a várias classes de uma única vez, possibilitando modificações de acordo com a necessidade.

A herança múltipla ocorre quando há uma relação de hierarquia de uma classe com duas ou mais classes, ou seja, uma classe (subclasse) herda atributos e/ou operações de duas ou mais classes (superclasses). Quando as características são herdadas apenas de uma superclasse é denominado por herança simples.

A Herança atribuída entre classes (objetos), as classificam como genéricas ou específicas. Uma classe herdeira é em geral uma **especialização** do seu ancestral, que por conseqüência será uma **generalização** de seu sucessor. Uma estrutura de hierarquias entre classes pode ser construída, baseada na relação de generalização e especialização. O resultado é que as classes mais ancestrais são mais genéricas ou abrangentes, e suas sucessoras se tornam cada vez mais específicas, à medida que aprofunda-se na estrutura.

Exemplo:

A figura 3.2 ilustra a modelagem completa de uma *classe*, a qual denomina-se por superclasse. As subclasses, Terrestre e Aquático recebem os métodos que existem na superclasse.

[5]A Classe Abstrata apenas declara alguns atributos e métodos, mas não os implementam. Maiores detalhes ver seção 3.1.3.

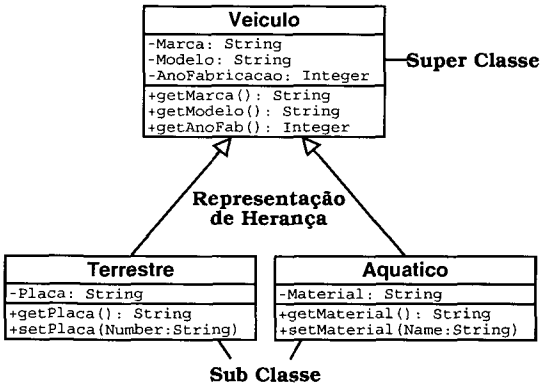


Figura 3.2: Modelagem de Classe com *Herança.*

3.2.3 Encapsulamento

O conceito de encapsulamento é decorrente da combinação dos atributos e métodos em um único objeto. Ele garante que a única forma de acesso a estes atributos é através de métodos disponíveis ao usuário, os quais são chamados de *métodos públicos.* Os demais métodos e os atributos da classe ficam sendo privados, ou seja, apenas funções da própria classe podem acessar diretamente estes atributos.

O uso do encapsulamento permite modificações mais seguras, sendo possível preservar a interface de um objeto quando houver mudanças em sua estrutura. Além desse aspecto, ainda há a proteção dos dados. Restringir o acesso dos atributos aos métodos da classe garante que nenhum dado será alterado por engano ou de forma descontrolada.

A tarefa de projetar uma classe envolve, entre outras atividades, definir da melhor maneira possível quais métodos de acesso ao objeto serão disponibilizados ao usuário. Um bom projeto inicial evita a necessidade de se redefinir uma classe já implementada.

3.3 Considerações Finais

Atualmente existe um consenso de que as metodologias orientadas a objetos são mais eficazes em lidar com a complexidade que emerge do projeto de *software* complexos e de grande porte. Isso acontece porque o processo e os dados possuem a mesma importância.

Os Objetos são utilizados para combinar os dados com os procedimentos que os operam. Dentre as vantagens no uso de objetos, pode-se observar que eles possuem tanto a abstração como o encapsulamento.

A modularização existente na orientação a objetos, permite a reutilização de código em uma nova formação de programas e modelagem de sistemas. A utilização de classes, facilita a criação de programas e minimiza grandes impactos no sistema quando houver a necessidade de manutenção.

CAPÍTULO 4

SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

Este capítulo faz uma introdução sobre uma proposta de extensão da Semântica de Ações: o formalismo da Semântica de Ações Orientada a Objetos [1] [2], aplica conceitos de orientação a objetos em Semântica de Ações.

Dentre várias aplicações, a Semântica de Ações é um formalismo utilizado para especificar linguagens de programação. Durante a especificação de linguagens de programação, inclusive linguagens de paradigmas distintos, nota-se que existem partes de código (especificação) que são comuns a outros já existentes, ou seja, existe uma semelhança estrutural entre estas partes. A replicação de código na especificação, acarreta em trabalho excessivo tanto para a construção quanto para a manutenção da especificação da linguagem. Esta replicação de código, também dificulta a leitura e o entendimento da especificação.

Com o intuito de minimizar a replicação de código, *Doh* [5] e *Menezes* [9] apresentaram uma proposta para reduzir a replicação de código e, conseqüentemente, facilitar a utilização da Semântica de Ações.

Utilizando o formalismo proposto por *Doh*, é possível evidenciar cada módulo e compor um linguagem a partir deles. A linguagem projetada em módulos, define cada um separadamente, sendo que um novo módulo é criado a partir da extensão ou combinação de outros já existentes. Geralmente, estes módulos são bem pequenos. Segundo *Carvilhe* [1] [2], existem fatores positivos e negativos para o uso de módulos, tais como: o corpo do módulo importado é inteiramente incluído no módulo que está importando, é um ponto negativo. A existência de um ferramental para a combinação de módulos, pela inclusão de outros na especificação atual, é um fator positivo.

A Semântica de Ações Baseada em Componentes, apresentada por *Menezes*, possui aspectos de reutilização e adequação de partes de linguagens em novas especificações. Nesta semântica, foram introduzidas duas notações sendo a notação de componentes que per-

mite a definição de estruturas reutilizáveis e a notação das linguagens de programação que fornece suporte à criação, uso e extensão da especificação em linguagens de programação. Em ambas notações existem pontos positivos e negativos, segundo *Carvilhe* [1] [2]. Ele considera como negativo, a dificuldade de usar as funções e operadores que foram definidos. Como ponto positivo, ele considera que os componentes disponibilizados são no geral genéricos. A reutilização é bem empregada, pois não apresenta repetição da parte de especificação, mas sim a reutilização de um componente ao trabalho atual.

Uma nova abordagem foi apresentada para a aplicação e o uso de módulos e componentes. Foi proposto tratá-los como objetos, usufruindo das características que o conceito de Orientação a Objetos oferece. Nas próximas seções é descrito o funcionamento da Semântica de Ações Orientada a Objetos.

4.1 Aplicação da Orientação a Objetos

A proposta introduzida por *Carvilhe*, visa solucionar alguns problemas existentes em módulos e componentes. A aplicação dos conceitos de orientação a objetos em Semântica de Ações, ocasiona algumas modificações e adaptações, as quais estão descritas a seguir.

Em Semântica de Ações Orientada a Objetos, uma classe é composta por sintaxe e semântica. O uso de objetos requer que algumas modificações sejam efetuadas na escrita da especificação.

A estrutura sintática de uma linguagem, servirá como base para a definição da hierarquia de classes. Em Semântica de Ações Orientada a Objetos, o objeto representa cada frase (declaração, comando, expressão) da linguagem, podendo existir mais de uma representação de objetos para a mesma frase. Os métodos da Semântica de Ações Orientada a Objetos possuem as mesmas funcionalidades das funções e equações semânticas existentes na Semântica de Ações, ou seja, os conceitos de funções e equações semânticas da Semântica de Ações são utilizados como métodos em Semântica de Ações Orientada a Objetos.

Existem algumas implicações no tratamento de Módulos como Objetos que devem ser observadas. Podemos considerar que a sintaxe e semântica é encapsulada por uma

instância de uma classe, logo tem-se um objeto. A aplicação dos conceitos de Orientação a Objetos deve estar claramente evidenciada quanto a criação, manutenção e uso de classes e objetos, destacando-se o Encapsulamento, Mensagem e Abstração [7] [13].

A seguir é exemplificado o uso do conceito de Orientação a Objetos para a definição da classe `IFCommand`. Esta classe especifica formalmente o comando "*if*" de linguagens de programação. A estrutura "*if*" é utilizada para realizar controle de fluxo baseado em expressões condicionais:

```
Class IFCommand
    extending Command
    using E:Expression, C:Command
    syntax:
        Com::= "if" E "then" C "endif"
    semantics:
        execute[[ "if" E "then" C "endif"]] =
                evaluate E
            then
                check(the given TruthValue is true) and then
                    execute C
                and then
                    complete
End Class
```

A diretiva `extending` indica que a classe `IFCommand` é uma subclasse de `Command`. Como base para os diversos comandos, a classe `Command` representa os comandos da linguagem. A diretiva `using` instância os objetos `E:Expression` e `C:Command`.

Uma redefinição da árvore sintática ocorre na seção de sintaxe, o qual é representado pela diretiva `syntax`. Nesta seção será definido a estrutura de um novo comando. Na seção `semantics` é definida o conjunto de instruções para cada comando declarado na seção de sintaxe. Neste contexto, o método `execute` é definido e sua semântica especificada. Também, ocorre a chamada do método `execute` que pertencente ao objeto `Command`, representando a ação `execute C`.

A notação de ações, definida por *Mosses* [10], é mantida no trabalho de *Carvilhe* [2], a qual representa um sub-conjunto da notação original baseado no trabalho de *Moura* [3]. A diferença está na interpretação das funções definidas pelo usuário, sendo que, em Semântica de Ações, a ação correspondente a `evaluate E` está determinada por equações

semânticas sobre a sintaxe representada por `E`. Na abordagem proposta por *Carvilhe*, `evaluate` é um método sobre o objeto `E`, estruturado exatamente como uma equação semântica, que produz a mesma ação.

4.2 Sintaxe da Notação

A relação entre as classes é caracterizada por diretivas, que indicam o instanciamento dos objetos. Desta forma, fica evidenciada a hierarquia do objeto e sua respectiva classe. A estrutura das frases de uma linguagem é dada pela especificação de sua árvore sintática. A semântica, por sua vez, é expressa pela notação de ações [10]. A definição da notação usada na Semântica de Ações Orientada a Objetos é definida por *Carvilhe* [2] e transcrita a seguir em um forma similar a BNF:

```
Class-Molule =
 [[ "Class" Class-Name Class-Body "End-Class" ]]

Class-Body =
 [[ ⟨ "extending" Base-Class-Name ⟩?
   ⟨ "using" Objects-Declaration ⟩?
   ⟨ Class-Definition ⟩? ]].

Base-Class-Name =
 [[ Class-Name | Class-Name "::"  Base-Class-Name ]].

Objects-Declaration =
 [[ Object-Declaration ]] | [[ Object-Declaration "," Objects-Declaration ]].

Object-Declaration =
 [[ Identifier ":" Class-Name ]].
```

O token **Class** representa a abertura de um bloco para a formação da classe. Ele é seguido de um identificador, o qual representa o nome da classe. Em seguida, há um conjunto de instruções que caracteriza o corpo da classe e, o token **End-Class** determina o fechamento da declaração.

No corpo da classe, opcionalmente, são declarados a *Base-Class-Name* (classe-base) e os objetos por ela utilizados. As seguintes diretivas podem ser empregradas:

- **extending:** informa que a classe atual está estendendo uma classe-base, sendo assim criada uma hierarquia. A utilização do operador "::" possibilita o acesso de uma determinada classe dentro da hierarquia.

- **using:** informa que serão instanciados objetos na classe atual. O identificador do objeto deve ser único dentro do escopo da classe atual. Cada objeto instanciado é representado por um identificador, seguido de ":" e da classe correspondente.

Exemplo: (Declaração de um comando).

```
Class DoCommand
    extending Command
    using E:Expression, C:Command
    ...
End Class
```

A classe `DoCommand` representa a declaração de uma classe do tipo **comando**, de uma linguagem de programação. `DoCommand` é uma extensão da classe `Command`, que corresponde à classe base para quaisquer formas de comando. Os identificadores `E` e `C`, representam instâncias dos objetos `Expression` e `Command`, respectivamente. Estes objetos são usados na classe atual.

A notação para a definição de classes é apresentada a seguir :

Class-Definition =
⟦ "syntax" ":" Syntatic-Part "semantic" Semantic-Part ⟧.

Syntatic-Part =
⟦ Token-Name | Token "::=" syntax-tree ⟧.

Semantic-Part =
⟦ ⟨ Semantic-Functions ⟩$^{?}$ ⟨ Semantic-Equations ⟩$^{?}$ ⟧.

Semantic-Functions =
⟦ Semantic-Function ⟧ | ⟦ Semantic-Function Semantic-Functions ⟧.

Semantic-Function =
⟦ Function-Name "_ " * Tokens "→" "Action" | data ⟧.

Tokens =
⟦ TokenName ⟧ | ⟦ TokenName "," Tokens ⟧.

Semantic-Equations =
⟦ Semantic-Equation ⟧ | ⟦ Semantic-Equation Semantic-Equations ⟧.

Semantic-Equation =
⟦ Function-Name "[[" syntax-tree "]]" "=" Action ⟧.

A seção **syntax** define a sintaxe abstrata da frase corrente, podendo ser um nome de token ou mesmo seguido de uma árvore sintática, como em [10]. Na seção **semantics**, as

funções e equações semânticas são tratados como métodos, definindo assim a semântica das frases. Ambas seções fazem parte da definição de uma classe.

A seguir é apresentada a sintaxe da notação de ações que é usada para definir o corpo dos métodos (semântica das frases). Aqui estão descritos as Ações (*Actions*), os Produtores (*Yielderes*) e os *Sorts* que são utilizados.

```
Action =
    [[ "complete" ]] | [[ "fail" ]] | [[ "unfold" ]] |
    [[ Action "and" Action ]] | [[ Action "and then" Action ]] |
    [[ "unfolding" Action ]] | [[ "give" Yielder "label" "#" n ]] |
    [[ "check" Yielder ]] | [[ Action "then" Action ]] |
    [[ "bind" Yielder "to" Yielder ]] |
    [[ "furthermore" Action ]] | [[ Action "hence" Action ]] |
    [[ Action "moreover" Action ]] | [[ Action "before" Action ]] |
    [[ "store" Yielder "in" Yielder ]] | [[ "deallocate" Yielder ]] |
    [[ "enact" Yielder ]] | [[ Action "else" Action ]] |
    [[ "recursively" "bind" Yielder "to" Yielder ]] |
    [[ "allocate a" Sort ]]

Yielder =
    [[ "the" Sort "#" n ]] |
    [[ "the" Sort "bound" "to" k ]] |
    [[ "the" Sort "stored" "in" Yielder ]] |
    [[ Yielder "with" Yielder ]] |
    [[ "closure" Yielder ]] |
    [[ "abstration of" Action ]]

Sort =
    [[ "bindable" ]] | [[ "cell" ]] | [[ "cell" "[" Sort "]" ]] |
    [[ "storable" ]] | [[ "abstraction" ]] | [[ "datum" | ]]
    [[ "integer" | "value" ]] | [[ "truth-value" ]] |
    [[ Sort | Sort ]]
```

Na próxima seção está descrita a semântica de notação usada em Semântica de Ações Orientada a Objetos.

4.3 Semântica da Notação

A aplicação dos conceitos de Orientação a Objetos possibilita a existência de hierarquias entre as classes. A superclasse (classe pai) sempre vai ser mais genérica que a

subclasse (classe filha). Partindo deste princípio, existe uma classe-base que é a mais genérica dentre todas, esta por sua vez, é donominada *State*. Esta super-classe é base para a geração de quaisquer outras classes.

A super-classe *State*, possui atributos genéricos que correspondem aos dados transitórios, *bindings* e armazenamento. Disponibiliza também operações que possibilitam interagir com estes atributos. As sub-classes herdam os atributos e operações declarados na super-classe. Nas próximas seções é descrito, resumidamente, o comportamento da classe *State*. A descrição completa está no capítulo 5 de [2].

4.4 A Classe *State*

Em Semântica de Ações Orientada a Objetos, a semântica de uma linguagem de programação é caracterizada pela criação de uma estrutura de Classes. Estas classes são responsáveis por representar a Semântica de Ações original [10] e fazem uso dos conceitos relativos aos Objetos, os quais permitem a reutilização e extensão das especificações.

Os atributos e operações sobre *State* são descritos nas seções seguintes.

4.4.1 Atributos

Os atributos são responsáveis pela estrutura das informações na especificação de uma linguagem. A classe *State* possui atributos como dados transitórios (*transient*), mapeamentos (*binding*) e armazenamento (*storage*). A manipulação destes atributos é disponibilizada pelas operações.

4.4.2 Operações

Operações são funções sobre os atributos de *State* [2]. Estas operações, dentro do contexto da Semântica de Ações Orientada a Objetos, correspondem exatamente às ações da Semântica de Ações, definida por [10]. Os termos utilizados para as operações são os mesmos descritos em Semântica de Ações. Em ordem de classificação de ações, as operações são dividas em **básicas** (controle de fluxo), **funcionais** (*transients*), **impera-**

tivas (*storage*), **declarativas** (*bindings*), **reflexivas** (abstrações) e **híbridas** (comportamento múltiplo). A descrição completa de cada operação e um exemplo completo sobre a aplicação da Semântica de Ações, estão descritos na dissertação de *Carvilhe* [2].

CAPÍTULO 5

BIBLIOTECA DE CLASSES PARA A SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

Este capítulo aborda a criação de uma biblioteca de classes em Semântica de Ações Orientada a Objetos. Esta biblioteca, denominada por LFL - *Language Features Library*, descreve conceitos comuns a várias linguagens de programação. As seções a seguir, apresentam uma introdução sobre a LFL, assim como sua estrutura organizacional e a sintaxe da notação de ações. Por fim, um estudo de caso baseado em uma pequena linguagem imperativa é apresentado.

5.1 Introdução

O uso de especificações formais para descrever linguagens de programação, proporciona uma visão antecipada das estruturas que deverão ser usadas na construção de uma linguagem, facilitando a descrição da linguagem.

Quando surgiram as primeiras especificações formais, estas eram complicadas de ser criadas, modificadas e estendidas. Com o passar dos anos, estas especificações tiveram grandes mudanças, sempre com o intuito de facilitar sua leitura e escrita [15]. A partir da Semântica Denotacional [14], surgiu a Semântica de Ações [10], a qual foi estendida a fim de idealizar a Semântica de Ações Orientada a Objetos. A proposta define a aplicação dos conceitos de Orientação a Objetos em Semântica de Ações, originando a Semântica de Ações Orientada a Objetos, que oferece uma especificação com a reutilização e extensão de código.

No processo de especificação de uma linguagem de programação, percebe-se que certas partes da especificação possuem funcionalidades semelhantes a outras já existentes. Isto motiva a reutilizar estas partes, porém alguns ajustes são necessários para adequá-las a uma estrutura comum. Esta estruturação pode ser feita usando o conceito de Classe.

Em uma especificação de linguagens de programação, inúmeras classes são criadas e reutilizadas. Com o crescimento na quantidade destas classes, tem-se a necessidade de adequá-las a uma estrutura organizacional, que facilitará sua localização e uso.

A nossa proposta é apresentar um modelo para criação de uma biblioteca de classes, a qual denomina-se por **LFL - *Languange Features Library***. A função da LFL é reunir e organizar as classes genéricas (especificação similar) em uma estrutura. Funcionará como um repositório de classes, disponibilizando formas de acesso à estas classes. A seção 5.2, descreve e exemplifica a estrutura organizacional da LFL.

5.2 Estrutura Organizacional

A forma adotada para organizar as Classes na LFL tem estrutura árvore. Os nodos desta árvore representam os níveis de hierarquia organizacional e as folhas são as classes. Algumas figuras são utilizadas para ilustrar a estrutura da LFL. Nestas figuras, existem retângulos com astes arredondadas ou quadradas, representando os nodos e folhas da árvore, respectivamente. A LFL é um conjunto de classes da Semântica de Ações Orientada a Objetos. Hierarquicamente, encontra-se diretamente ligada à classe *State* (capítulo 3, pág. 41), como mostrado na figura 5.1.

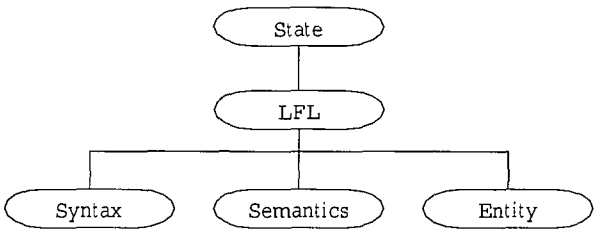


Figura 5.1: Nodos base da LFL.

A LFL está composta por três estruturas principais, dedicadas, respectivamente, a agregar conceitos relacionados às três partes principais da especificação em Semântica de Ações. Estas partes são: *Abstract Syntax*, *Semantics* e *Semantic Entities*. A partir destas três partes principais, estão sendo propostos a criação dos nodos *Syntax*, *Semantics* e *Entity*, respectivamente (figura 5.1).

O trabalho concentra-se na elaboração de uma versão inicial da árvore a partir do nodo *Semantics*. A definição das classes dos nodos *Syntax* e *Entity* é deixada para uma segunda versão da biblioteca, fugindo ao escopo desta dissertação.

A seguir serão apresentadas algumas classes pertencentes a LFL. Estas classes foram modificadas com o intuito de adequa-las à biblioteca. As modificações efetuadas nas classes agrupadas nos nodos *Syntax* e *Entity*, estão apenas sendo sugeridas, pois estas não fazem parte de nossos estudos. Os ajustes efetuados às classes pertencentes ao nodo *Semantics*, são efetivamente aqueles os quais foram desenvolvidos como estudo principal.

Syntax (*Abstract Syntax*): como sugestão, as classes pertencentes a este nodo possuem definições relativas à sintaxe das linguagens. A seguir é mostrada em um exemplo como poderia ser a definição sintática da classe `Identifier`.

Exemplo: 5.1

```
Class Identifier
  locating LFL.Syntax
  syntax:
    Id ::= letter [ letter | digit ]*
End Class
```

No exemplo acima é definido que cada elemento da classe `Identifier` é formado por `letter` e `digit`. A localização da classe na LFL é indicada através da diretiva `locating`.

Entity (*Semantic Entities*): como sugestão, as classes pertencentes a este nodo referem-se àquelas que não se enquadram nas categorias dos nodos *Syntax* e *Semantics*. Nestas classes, pode ser especificada a sintaxe, semântica ou ambas. No exemplo abaixo é especificado a classe `Types`.

Exemplo: 5.2

```
Class Types
  syntax:
    Type ::= booleanType | integerType | referenceType
  semantics:
    default value of _ :  Type → Value
End Class
```

A classe `Types`, define na sua parte sintática, alguns tipos de dados representados pelo *sort* `Type`. Na semântica, a função '`default value of`' recebe um elemento do tipo `Type` e devolve um valor.

Semantics (*Semantic Functions*): as classes pertencentes a este nodo, definem conceitos relacionados à semântica das linguagens, por exemplo, a especificação semântica da classe `Constant` é mostrada a seguir:

Exemplo: 5.3

```
(1)   Class Constant ≪
              Identifier,
              Expression implementing <evaluate _ :  Expression → Action>  ≫
(2)     locating LFL.Semantics.Declaration.Paradigm.Imper
        using E:Expression, I:Identifier
        semantics:
(3)           evaluate-constant( E, I ) =
                evaluate ⟦ E ⟧
              then
                bind I to the value
      End Class
```

As classes da LFL têm algumas diferenças na sua notação em relação a seu equivalente na Semântica de Ações Orientada a Objetos original. Algumas destas diferenças podem ser notadas no exemplo 5.3. Na linha (1) há uma nova definição de classe, sendo possível a passagem de parâmetros. A diretiva `locating`, na linha (2), indica em qual nodo a classe está agrupada. Na linha (3), há uma nova definição de método contendo parâmetros mais gerais que o emprego de árvores sintáticas proposta em [2]. A notação da Semântica de Ações Orientada a Objetos alterada para adequar as classes na LFL, encontra-se descrita na seção 5.3.

5.2.1 Organização das Classes nos Nodos *Syntax* e *Entity*

Nas seções 5.2.1.1 e 5.2.1.2 estão sendo sugeridos modelos para estruturar organizacionalmente as classes subordinadas aos nodos *Syntax* e *Entity*. Conforme descrito na seção 5.2 [pág. 45], ambos nodos não são objetos de nosso estudo, apenas estão sendo sugeridos para trabalhos futuros.

5.2.1.1 Nodo *Syntax*

A sintaxe de uma linguagem fornece regras para a formação das frases da linguagem, como declarações, comandos e expressões. O nodo *Syntax* contém classes de biblioteca para a definição de elementos sintáticos de linguagens de programação.

A figura 5.2 ilustra hierarquicamente o nodo *Syntax* com duas classes. A especificação das classes *Identifier* e *Numeral* possui apenas a definição sintática dos elementos de linguagem de programação.

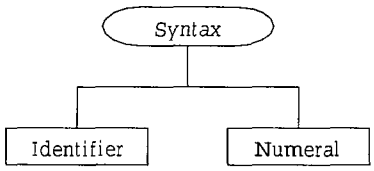


Figura 5.2: Nodo Syntax.

5.2.1.2 Nodo *Entity*

As entidades semânticas em Semântica de Ações definem os tipos de dados e operações auxiliares de outras partes da especificação formal de linguagens de programação. O nodo *Entity* possui as mesmas características que as entidade semânticas e sua ramificação é definida conforme os tipos das entidades que são usadas pela Semântica de Ações. As entidades usadas são: *Action*, *Data* e *Yielder* [10][seção 1.5.2]. A figura 5.3 ilustra hierarquicamente o nodo *Entity* com suas ramificações.

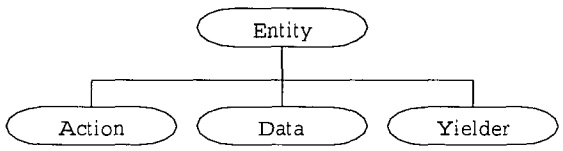


Figura 5.3: Nodo Entity.

5.2.2 Nodo *Semantics*

Na sua apresentação tradicional em Semântica de Ações, as funções semânticas representam o significado das frases de uma linguagem. O nodo *Semantics* contém classes que definem o comportamento (semântica) de elementos presentes em linguagens de programação. Este nodo é subdivido em: *Declaration*, *Command* e *Expression* (figura 5.4). Esta subdivisão (ramificação) é definida em função da estrutura sintática de linguagens de programação. Cada conceito presente nesta parte da LFL, estará disponível para seu uso na especificação de linguagens de programação e pode ser estendida conforme a inserção de classes para representar novos conceitos na LFL.

Em cada um dos nodos *Declaration*, *Command* e *Expression* há duas subdivisões denominadas por *Paradigm* e *Shared*. A primeira agrupará as classes por tipos de paradigma de linguagem de programação e a segunda conterá as classes que são comuns a vários paradigmas de linguagens. Como proposta inicial, o nodo *Paradigm* está sendo subdividido em: *OO*, *Func*, *Logical* e *Imper*, representando as linguagens Orientadas a Objetos, Funcionais, Lógicas e Imperativas, respectivamente. A figura 5.4 ilustra hierarquicamente o nodo *Semantics* com suas ramificações na versão inicial proposta da LFL.

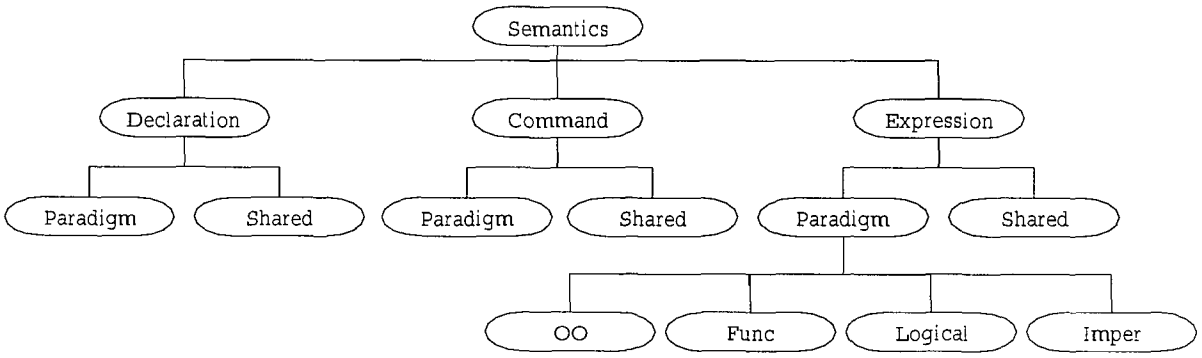


Figura 5.4: Nodo Semantics.

5.2.3 Hierarquia básica da LFL

Nas seções anteriores (5.2.1.1, 5.2.1.2 e 5.2.2) foram apresentados os nodos que fazem parte da base estrutural da LFL. Dentre os três principais, o nodo *Semantics* é o alvo

principal de nosso estudo. Como parte da proposta, vamos concentrar os esforços em definir a organização e formas de acesso para as classes deste nodo.

A figura 5.5 ilustra, em uma estrutura de árvore, a organização das classes na LFL. Os nodos que não fazem parte de nosso estudo, porém estão sendo sugeridos, aparecem na figura em linhas pontilhadas. Nas próximas seções é apresentada, com modificações, a sintaxe da notação da Semântica de Ações Orientada a Objetos. Uma pequena linguagem de programação, será utilizada para exemplificar o uso da LFL.

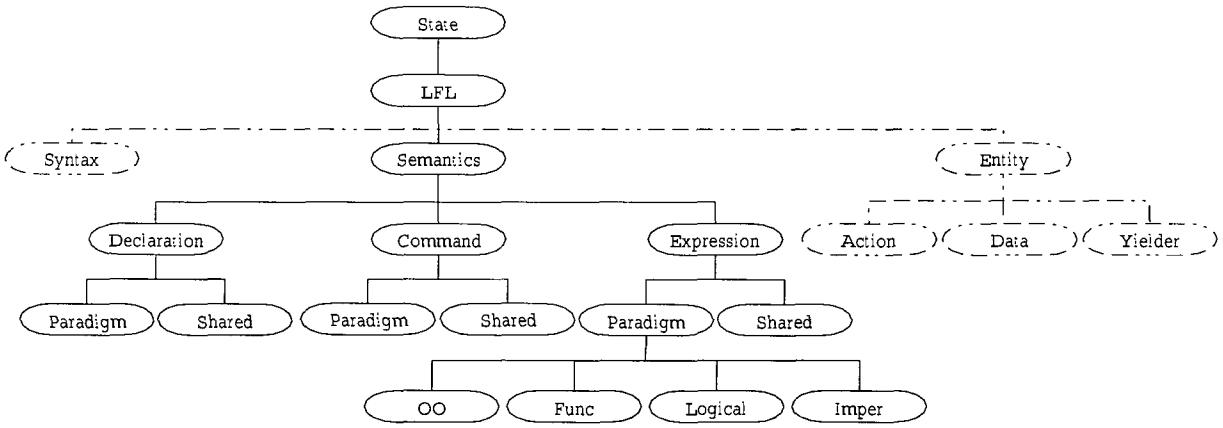


Figura 5.5: Estrutura organizacional aplicada à LFL.

5.3 Sintaxe da Notação

O relacionamento entre classes é caracterizado por meio de diretivas. Estas diretivas possuem funcionalidades distintas, podendo expressar ações como a criação de objetos ou a definição da estrutura organizacional à qual a classe pertence, bem como indicar a hierarquia de classes. A estrutura das frases de uma linguagem é dada usando árvores sintáticas, sendo que a semântica é expressa pela notação de ações [10]. A partir da aplicação de conceitos de orientação a objetos em Semântica de Ações, foi definida uma notação para a Semântica de Ações Orientada a Objetos [1] [2]. Alterações tiveram que ser efetuadas na notação, visando fazer ajustes necessários para adequar as classes na LFL. A seguir estão descritas as modificações que foram aplicadas na notação da Semântica de Ações Orientada a Objetos, sendo que a notação original pode ser encontrada na seção

4.2 [pág. 38].

5.3.1 Notação de Classes

A seguir é apresentada, parcialmente, a sintaxe da notação para a Semântica de Ações Orientada a Objetos. Serão apresentados os elementos da notação que tiveram alterações, bem como aqueles que foram inclusos. Para facilitar a explicação do funcionamento de cada elemento da notação, foram inseridos números de referência. Estes números estão localizados à esquerda de cada elemento.

```
     Class-Molule =
(1)  [[ "Class" Class-Name "≪" Parameter-Class "≫" Class-Body "End-Class" ]].
```

O elemento na linha (1) define a declaração para uma classe. Esta classe (*Class-Name*) recebe argumentos como parâmetro, o qual é classificado, dentro do escopo atual, como uma classe parametrizada (*Parameter-Class*). Juntamente com esta classe parametrizada, pode ser informado algum método. A classe e o método recebidos como parâmetro, serão usados na semântica da classe que está sendo definida (*Class-Name*).

A declaração de uma Classe (1) é definida pelo token **Class** seguido de seu nome, por um token "≪" seguido por parâmetros da classe (5) e o token "≫". Logo após, o corpo da classe é inserido seguido de **End Class**.

```
     Class-Body =
(2)  [[ ⟨ "locating" Structure-Locate ⟩? ]] |
(3)  [[ ⟨ "including" Structure-Locate ⟩? ]].
```

No corpo da classe (2) (3) pode ser atribuído várias diretivas, como o endereço da classe e sua localização na estrutura organizacional da LFL. As seguintes diretivas podem ser empregadas:

- **locating**: determina na classe atual sua localização em uma estrutura organizacional. Portanto, a aplicação desta diretiva está relacionada com as classes que pertencem a LFL. Logo abaixo, no número de referência (6), está definida a forma de acesso a estrutura organizacional da LFL.

- **including**: torna disponível para a classe atual os métodos das classes contidas em um determinado nodo da estrutura organizacional (número de referência (6)).

(4) Object-Declaration =
⟦ Identifier ":" ⟨ Structure-Locate "." ⟩? Class-Name ⟧ |
⟦ Identifier ":" ⟨ Structure-Locate "." ⟩? Class-Name "≪" Paremeter-Class "≫" ⟧.

Uma classe ao ser instanciada, tem sua referência atribuída a um objeto. A declaração de um objeto (4) pode ser definida por duas formas:

1. Define-se o nome do identificador do objeto seguido pelo token ":". Em seguida, opcionalmente, pode-se indicar qual estrutura a classe que está sendo instanciada pertence (6), juntamente com o token ".". Por fim, indica-se o nome da classe que será instanciada.

2. A outra forma de declaração, é semelhante à primeira. Porém, adicionam-se argumentos à classe que será instanciada. Estes argumentos são nomes de classes (opcionalmente incluindo nomes de métodos, conforme (5)) passados como parâmetros, os quais devem estar dispostos entre "≪" e "≫".

(5) Parameter-Class =
⟦ Class-Name ⟧ |
⟦ Class-Name "implementing" "<" Method-Name ">" ⟧ |
⟦ Parameter-Class "," Parameter-Class ⟧.

Uma classe quando declarada, pode requerer argumentos como parâmetros. A passagem de argumentos à classe pode ser efetuada de três maneiras. São elas:

1. O nome de uma classe é declarado como parâmetro.

2. Uma classe incluindo o nome de um método, são requeridos como parâmetro. A diretiva ***implementing*** é empregada para declarar o método que também será recebido como parâmetro.

   A declaração do parâmetro é constituído pelo nome da classe seguido pelo token *implementing* e a declaração de seu respectivo método. Este método deve estar disposto entre "<" e ">".

3. Uma seqüência de parâmetros.

(6) Structure-Locate =
[[Node-Name]] | [[Node-Name "." Structure-Locate]].

Para acessar as classes que estão agrupadas em algum nodo da árvore, a qual representa a estrutura organizacional da LFL, é necessário apontar o caminho completo, ou seja, indicar todos os nodos (6). Estes nodos devem ser interlacados pelo token ".". Para acessar o nodo raiz da árvore, basta indicar o nome LFL.

A especificação completa da notação de classes para a Semântica de Ações Orientada a Objetos, está no Anexo A.

A seguir são mostrados exemplos sobre a notação de classes da Semântica de Ações Orientada a Objetos:

Exemplo: 5.4 (Uma Classe da LFL que avalia um identificador)

```
(1)   Class IdentifierEvaluate ≪ Identifier ≫
(2)     locating LFL.Semantics.Expression.Shared
(3)     using I:Identifier
        semantics:
           evaluate-identifier( I ) =
              give the value bound to I
      End Class
```

A classe `IdentifierEvaluate`, pertencente à LFL, representa a avaliação de um argumento. Esta classe, em sua declaração, recebe como parâmetro um argumento (1). Este parâmetro será visto no escopo de `IdentifierEvaluate` como uma classe, a qual está denominada por `Identifier` (3). A diretiva `locating` (2), permite indicar a qual nodo da estrutura organizacional na LFL a classe está alocada.

Exemplo: 5.5 (Utilizando a classe `IdentifierEvaluate` da LFL)

```
      Class Identificador
(1)     including LFL.Semantics.Expression.Shared
(2)     using objId:IdentifierEvaluate ≪ Identificador ≫
        syntax:
(3)        Id ::= letter [ letter | digit ]*
        semantics:
           avaliacao "[[" Id "]]" =
(4)           objId.evaluate-identifier( Id )
      End Class
```

O exemplo 5.5 mostra o uso de uma classe pertencente à LFL. A classe a ser utilizada é a `IdentifierEvaluate`, a qual está declarada no exemplo 5.4. As diretivas empregadas para o uso da classe `IdentifierEvaluate`, são apresentadas a seguir:

A diretiva `including` (1) disponibiliza dentro do escopo da classe atual `Identificador` os métodos das classes contidas na estrutura `LFL.Semantics.Expression.Shared`. Note que apenas as classes alocadas em `Shared` estarão disponibilizando seus métodos. Para a utilização das classes pertencentes a outros níveis da estrutura, como o nodo `Expression`, é necessário efetuar uma nova declaração da diretiva `including`. Por exemplo: `including LFL.Semantics.Expression`.

A criação de objetos a partir das classes disponibilizadas na declaração de `including`, torna-se possível através da diretiva `using` (2). Ao instanciar a classe `IdentifierEvaluate`, esta por sua vez requer um argumento como parâmetro (2), para o qual está sendo passada a classe `Identificador`. Note-se que as classes da LFL usam os elementos sintáticos que são definidos por classes externa à LFL.

Os métodos das classes contidas na LFL, podem ser usados mediante o instanciamento dessas classes em objetos.

A definição de como os métodos devem ser acessados, está descrita na próxima seção.

5.3.2 Definição do Corpo das Classes

A seguir é apresentada, parcialmente, a notação que define a sintaxe do corpo das classes para a Semântica de Ações Orientada a Objetos. Com relação a notação original [2], um novo elemento foi inserido na notação. Logo abaixo, este novo elemento será

apresentado. Para facilitar a explicação, foi inserido o número (1), o qual está localizado a esquerda do novo elemento.

```
Class-Definition =
 [[ "syntax" ":" Syntatic-Part "semantic" Semantic-Part ]].

Syntatic-Part =
 [[ Token-Name | Token "::=" syntax-tree ]].

Semantic-Part =
 [[ ⟨ Semantic-Functions ⟩? ⟨ Semantic-Equations ⟩? ]].

Semantic-Functions =
 [[ Semantic-Function ]] | [[ Semantic-Function Semantic-Functions ]].

Semantic-Function =
 [[ Function-Name "_ " * Tokens "→" "Action" | data ]].

Tokens =
 [[ TokenName ]] | [[ TokenName "," Tokens ]].

Semantic-Equations =
 [[ Semantic-Equation ]] | [[ Semantic-Equation Semantic-Equations ]].

Semantic-Equation =
 [[ Function-Name "[[" syntax-tree "]]" "=" Action ]] |
(1) [[ Function-Name "(" ⟨ data ⟩? ")" "=" Action ]]
```

Um novo método é definido na linha (1). Este método pode receber como parâmetro um determinado dado (***data***) que será empregado na semântica definida pelo método. Um *Action* sempre será gerado para o método. A definição de um método é composto por um nome seguido pelos tokens "(" e ")". No caso de haver parâmetro, este deve estar disposto entre os tokens citados. Por fim, uma ação deve ser definida à este método.

A especificação completa da notação que define a sintaxe do corpo das classes para a Semântica de Ações Orientada a Objetos, está no Anexo A.

5.3.3 Notação de Ações

A sintaxe da notação de ações será utilizada para definir o corpo dos métodos (semântica das frases). A seguir é apresentada a única modificação para *Action*, a qual está dada pela adição da chamada a métodos externos.

```
Action =
    ... |
(1)     [[ Object-Name "." Method-Name "(" ⟨ data ⟩? ")" ]]
```

O elemento incluso na notação de ações (1), define como acessar o método de uma classe. Para utilizar este método, é necessário associar o nome do objeto (**Obejct-Name**) ao método (**Method-Name**) correspondente. Esta associação é feita através do token ".". Logo após o nome do método, devem ser inseridos os tokens "(" e ")". Entre estes tokens, pode ser empregado algum tipo de dado (*data*) a ser passado como parâmetro ao método.

Um exemplo da utilização de método, pode ser visto na seção 5.5.

A especificação completa da notação de ações para Semântica de Ações Orientada a Objetos, está no Anexo A.

5.4 Classes que compõem a LFL

Nesta seção, serão apresentadas as classes que pertencem à LFL. Inicialmente, foram introduzidas algumas classes do paradigma Imperativo e as que são comuns a várias linguagens de programação. Lembrando que o alvo principal de nosso estudo, concentra-se no nodo *Semantics* (seção 5.2.3), portando as classes aqui apresentadas estão abaixo deste nodo.

Para facilitar a visualização das classes na LFL, algumas figuras são usadas como ilustração. Nestas figuras, existem retângulos com astes arredondadas ou quadradas, representando os nodos e folhas da árvore, respectivamente. Serão apresentadas as classes referentes às declarações (nodo *Declaration*), em seguida aquelas que se referem aos comandos (nodo *Command*) e por fim, as classes referentes às expressões (nodo *Expression*).

5.4.1 Classes do Nodo *Declaration*

A LFL possui classes que permitem a criação de constantes, variáveis e a seqüenciação de variáveis. Estas classes estão agrupadas no nodo *Declaration*. A estrutura de localização destas classes pode ser vista na figura 5.6.

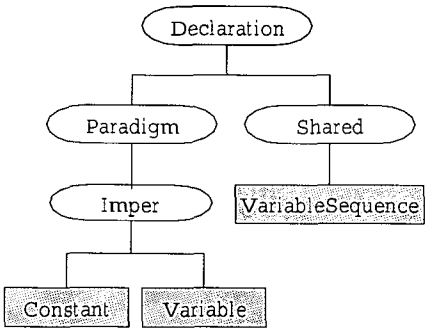


Figura 5.6: Classes contidas no nodo *Declaration*.

5.4.1.1 Nodo Imper

Este nodo associa as classes que são consideradas do paradigma Imperativo em linguagens de programação. A seguir são apresentadas as classes `Constant`, `Variable` e `VariableDec` .

Classe: 5.1 (Declaração de Constante)

```
Class Constant ≪
      Identifier,
      Expression
         implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Declaration.Paradigm.Imper
  using E:Expression, I:Identifier
  semantics:
      elaborate-constant( I , E ) =
         evaluate ⟦ E ⟧
          then
         bind I to the value
End Class
```

A classe `Constant` especifica uma declaração de constante. Esta classe recebe como parâmetro dois argumentos, `Identifier` e `Expression` juntamente com seu método `evaluate`. A diretiva `locating` indica que a classe pertence ao nodo `Imper`. Os argumentos recebidos como parâmetro serão tratados como classes, sendo criados os objetos `E:Expression` e `I:Identifier`. A semântica de declaração de constante consiste em efetuar um *binding* do identificador `I` ao valor produzido como resultado à chamada do método `evaluate` do objeto `E`.

Classe: 5.2 (Declaração de Variável com Atribuição)

```
Class Variable ≪
     Identifier,
     Type,
     Expression
        implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Declaration.Paradigm.Imper
  using E:Expression, I:Identifier, T:Type
  semantics:
     elaborate-variable( I, T, E ) =
            evaluate ⟦ E ⟧ then give the value label#1
          and
            allocate-for-type T then give the cell label#2
       then
            bind I to the cell#2
          and
            store the value #1 in the cell#2
End Class
```

A classe `Variable` especifica uma declaração de variável com atribuição de uma expressão. Esta classe requer três argumentos como parâmetro, `Identifier`, `Type` e `Expression` (Esta última, com a definição do método `evaluate`). Estes parâmetros são tratados como classes dentro do escopo da classe atual. A diretiva `using` cria os objetos `I`, `T` e `E`, os quais são instâncias das classes `Identifier`, `Type` e `Expression`, respectivamente. A diretiva `locating` indica que a classe atual está associada ao nodo `Imper`.

Na semântica da classe, ocorre a declaração de variáveis com atribuição do valor inicial para a mesma. Um *binding* é efetuado entre o identificador `I` e uma célula de memória, sendo que o método `allocate-for-type` do objeto `T` reserva o espaço em memória para este fim. Em seguida, um valor é armazenado na célula de memória reservada, resultante da aplicação do método `evaluate` do objeto `E`.

Classe: 5.3 (Declaração de Variável sem Atribuição)

```
Class VariableDec ≪  Identifier, Type ≫
  locating LFL.Semantics.Declaration.Paradigm.Imper
  using I:Identifier, T:Type
  semantics:
     elaborate-variable( I , T ) =
       allocate-for-type T then give the cell label#1
       then
       bind token I to the cell#1
End Class
```

A classe `VariableDec` especifica uma declaração de variável sem a atribuição de uma expressão. Esta classe requer como parâmetro os argumentos `Identifier` e `Type`. A diretiva `using` cria os objetos `I` e `T`, os quais são instâncias das classes `Identifier` e `Type`, respectivamente. A diretiva `locating` indica que a classe atual está associada ao nodo `Imper`.

Na semântica da classe, é efetuada a declaração de variáveis sem considerar a especificação do valor inicial para a mesma. Um *binding* é formado entre o identificador `I` e uma célula de memória, sendo que o método `allocate-for-type` do objeto `T` reserva o espaço em memória para este fim.

5.4.1.2 Nodo Shared

As classes contidas neste nodo, são aquelas consideradas genéricas a vários paradigmas de linguagens de programação. A seguir é apresentada a classe *VariableSequence*.

Classe: 5.4 (Seqüenciação de Declaração de Variáveis)

```
Class VariableSequence ≪
      VariableDeclaration
         implementing <elaborate _ :  VariableDeclaration → Action> ≫
  locating LFL.Semantics.Declaration.Shared
  using VD1:VariableDeclaration, VD2:VariableDeclaration
  semantics:
      elaborate-var-sequence( VD1, VD2) =
            elaborate ⟦ VD1 ⟧
         before
            elaborate ⟦ VD2 ⟧
End Class
```

A seqüenciação de declaração de variáveis é expressa pela classe `VariableSequence`, a qual recebe como parâmetro o argumento `VariableDeclaration` com a implementação do método `elaborate`. Os objetos VD_1 e VD_2 são instâncias formadas a partir da classe `VariableDeclaration`. A indicação que a classe atual (*VariableSequence*) pertence ao nodo `Shared`, está sendo feita ao usar a diretiva `locating`.

5.4.2 Classes do Nodo *Command*

Este nodo agrupa classes que definem os comandos para uma linguagem de programação. Estes comandos possuem características comuns, mas cada um apresenta um formato e um significado diferente. A estrutura de localização destas classes pode ser vista na figura 5.7.

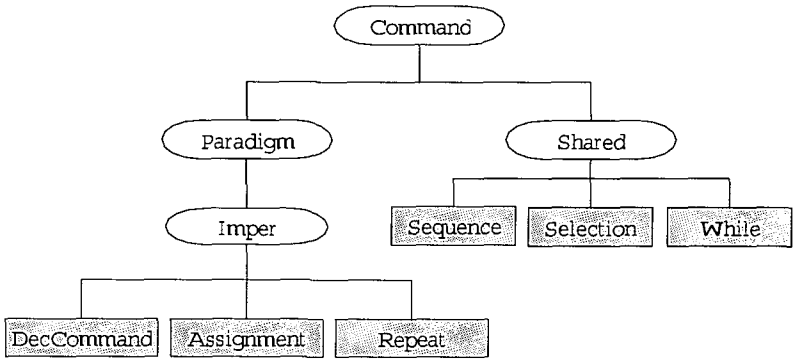


Figura 5.7: Classes contidas no nodo *Command*.

5.4.2.1 Nodo Imper

Este nodo representa as classes que são consideradas do paradigma Imperativo em linguagens de programação. A seguir são apresentadas as classes `DecCommand`, `Assignment` e `Repeat`.

Classe: 5.5 (Declaração de um Comando)

```
Class DecCommand ≪
     Declaration
        implementing < elaborate _ :  Declaration → Action >,
     Command
        implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Command.Paradigm.Imper
  using D:Declaration, C:Command
  semantics:
     execute-dec-command( D, C ) =
           furthermore elaborate ⟦ D ⟧
        hence
           execute ⟦ C ⟧
End Class
```

A classe `DecCommand` especifica semanticamente a elaboração de uma declaração e execução de um comando. Os parâmetros recebidos por esta classe são: `Declaration` com o método `elaborate` e `Command` com a definição do método `execute`. A partir destas classes recebidas como parâmetro, são criados os objetos `D:Declaration` e `C:Command`. A indicação de localização na LFL é apresentada pela diretiva `locating`. Os objetos `D` e `C` serão aplicados aos métodos `elaborate` e `execute`, respectivamente.

Classe: 5.6 (Comando de Atribuição)

```
Class Assignment ≪
     Identifier,
     Expression
           implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Command.Paradigm.Imper
  using I:Identifier, E:Expression
  semantics:
    execute-assignment( I, E ) =
              evaluate ⟦ E ⟧
            then
              store the value int the cell bound to I
End Class
```

O comando de atribuição é representado pela classe `Assignment`. Esta classe recebe como parâmetro os argumentos `Identifier` e `Expression` com o método `evaluate`. Estes argumentos são considerados classes no escopo de `Assignment`, desta forma são instanciados os objetos `I:Identifier` e `E:Expression`. O método `evaluate` denota uma atribuição, ou seja, a ação utilizada neste método efetua o armazenamento de um valor na célula de memória correspondente a `I`.

Classe: 5.7 (Comando de Repetição)

```
Class Repeat ≪
     Command
       implementing < execute _ :  Command → Action >,
     Expression
       implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Command.Paradigm.Imper
  using E:Expression, C:Command
  semantics:
    execute-repeat( E , C ) =
      unfolding
           execute ⟦ C ⟧
```

```
          and then
            evaluate [[ E ]]
              then
                   complete
                 else
                   unfold
End Class
```

A classe Repeat representa uma estrutura de repetição de um comando. Ao instanciar esta classe, dois parâmetros são necessários serem atribuídos: O parâmetro Command com seu respectivo método execute e Expression, juntamente com o método evaluate. Estes parâmetros são considerados classes dentro do escopo atual, assim os objetos E:Expression e C:Command são instanciados. A semântica da estrutura de repetição consiste na re-execução do comando C até que a expressão E produza valor verdadeiro.

5.4.2.2 Nodo Shared

O nodo *Shared* agrupa classes que são consideradas genéricas á vários paradigmas de linguagens de programação. A seguir são apresentadas as classes Sequence, Selection e While.

Classe: 5.8 (Seqüenciação de Comandos)

```
Class Sequence ≪
     Command
       implementing < execute _ :  Command → Action >  ≫
  locating LFL.Semantics.Command.Shared
  using C1:Command, C2:Command
  semantics:
    execute-sequence( C1, C2 ) =
          execute [[ C1 ]]
       and then
          execute [[ C2 ]]
End Class
```

A seqüenciação de comandos é expressa pela classe Sequence. Esta classe ao ser instanciada requer o parâmetro Command seguido do método execute. Os objetos C_1 e C_2 são criados a partir da classe Command. A indicação que a classe atual (VariableSequence) pertence ao nodo Shared, está sendo feita ao usar a diretiva locating.

Classe: 5.9 (Comando de Seleção)

```
Class Selection ≪
     Command
        implementing < execute _ :  Command → Action >,
     Expression
        implementing < evaluate _ :  Expression → Action >  ≫
   locating LFL.Semantics.Command.Shared
   using E:Expression, C1:Command, C2:Command
   semantics:
     execute-if-then( E , C1 ) =
            evaluate ⟦ E ⟧
        then
            check(the given TruthValue is true) and then
              execute ⟦ C1 ⟧
            and then
            complete

     execute-if-then-else( E, C1, C2 ) =
              evaluate ⟦ E ⟧
            then
              check(the given TruthValue is true) and then
                  execute ⟦ C1 ⟧
        or
              check(the given TruthValue is false) and then
                  execute ⟦ C2 ⟧
            and then
              complete
End Class
```

Na definição da classe `Selection`, dois parâmetros são necessários: `Command` com o método `execute` e `Expression` com o método `evaluate`. Utilizando estes parâmetros como classes, são instanciados três objetos: `E:Expression`, C_1`:Command` e C_2`:Command`. A indicação que a classe atual (`Selection`) pertence ao nodo `Shared`, está sendo feita ao empregar a diretiva `locating`.

São definidos dois métodos para a classe `Selection`. Em ambos a expressão representada pelo objeto `E` produz um valor-verdade mediante a chamada do método `evaluate`. No método `execute-if-then`, existe apenas a possibilidade de execução do comando C_1. No método `execute-if-then-else`, em caso positivo o primeiro comando (C_1) será executado, e caso contrário, o segundo comando (C_2) é executado.

Classe: 5.10 (Repetição Condicional)

```
Class While ≪
      Expression
         implementing < evaluate _ :  Expression → Action >,
      Command
         implementing < execute _ :  Command → Action > ≫
   locating LFL.Semantics.Command.Shared
   using E:Expression, C:Command
   semantics:
      execute-while ( E, C ) =
         unfolding
           evaluate ⟦ E ⟧ then
           infalibly select
             given true then execute ⟦ C ⟧ then unfold
           or
             given false then skip
End Class
```

A classe `While` representa uma estrutura de repetição de um comando. Ao instanciar esta classe, dois parâmetros são necessários serem atribuídos: `Command` com o método `execute` e `Expression` juntamente com o método `evaluate`. Estes parâmetros são considerados classes no escopo de `While`, desta forma são instanciados os objetos `C:Command` e `E:Expression`. A semântica da estrutura de repetição consiste na produção de um valor verdadeiro pela expressão representada pelo objeto `E`, assim o comando, representado pelo objeto `C`, pode ser (re)executado.

5.4.3 Classes do Nodo *Expression*

As classes que definem as expressões de uma linguagem de programação, estão agrupadas no nodo `Expression`. A hierarquia de localização dessas classes está ilustrada na figura 5.8.

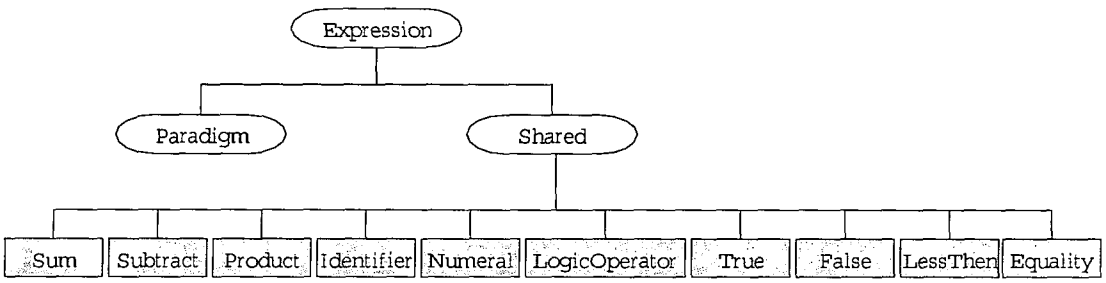


Figura 5.8: Classes contidas no nodo *Expression*.

5.4.3.1 Nodo Shared

As classes que são genéricas a vários paradigmas de linguagens de programação estão agrupadas no nodo Shared. A seguir são apresentadas as classes `Sum`, `Subtract`, `Product`, `Identifier`, `Numeral`, `LogicOperator`, `True`, `False`, `LassThen` e `Equality`.

Classe: 5.11 (Soma de Expressões)

```
Class Sum ≪
      Expression
         implementing < evaluate _ :  Expression → Action >  ≫
   locating LFL.Semantics.Expression.Shared
   using E1:Expression, E2:Expression
   semantics:
      evaluate-sum( E1 , E2 ) =
            evaluate ⟦ E1 ⟧
         and then
            evaluate ⟦ E2 ⟧
      then
         give sum(the give integer#1, the give integer#2)
End Class
```

Classe: 5.12 (Subtração de Expressões)

```
Class Subtract ≪
      Expression
         implementing < evaluate _ :  Expression → Action >  ≫
   locating LFL.Semantics.Expression.Shared
   using E1:Expression, E2:Expression
   semantics:
      evaluate-sub( E1 , E2 ) =
            evaluate ⟦ E1 ⟧
         and then
            evaluate ⟦ E2 ⟧
      then
         give subtraction(the give integer#1, the give integer#2)
End Class
```

Classe: 5.13 (Multiplicação de Expressões)

```
Class Product ≪
      Expression
         implementing < evaluate _ :  Expression → Action >  ≫
   locating LFL.Semantics.Expression.Shared
   using E1:Expression, E2:Expression
   semantics:
      evaluate-prod( E1 , E2 ) =
            evaluate ⟦ E1 ⟧
         and then
            evaluate ⟦ E2 ⟧
      then
         give product(the give integer#1, the give integer#2)
End Class
```

Pode-se notar que há uma estrutura comum na definição das classes `Sum` (5.11), `Subtract` (5.12) e `Product` (5.13), onde estas representam a soma, subtração e multiplicação respectivamente, entre expressões. Para qualquer uma dessas classes, o parâmetro `Expression` com o método `evaluate` deve ser informado, assim como os objetos E_1 e E_2 são instanciados. O resultado final em cada classe é definido pelo uso das funções auxiliares `sum( _ , _ )`, `subtraction( _ , _ )` ou `product( _ , _ )` que efetuam as operações da soma, subtração ou multiplicação baseado nos valores das expressões. Estas funções estão definidas em [10].

Classe: 5.14 (Avaliação de Identificador)

```
Class Identifier ≪ Ident ≫
  locating LFL.Semantics.Expression.Shared
  using I:Ident
  semantics:
     evaluate-identifier( I ) =
        give the value bound to I
     or
        give the value stored in the cell bound to I
End Class
```

A classe `Identifier` retorna o valor associado a um identificador. Esta classe requer o argumento `Ident` como parâmetro, o qual será instanciado no objeto `I`. O parâmetro recebido pode ser uma variável ou constante, pois a classe `Identifier` possui a especificação para ambas possibilidades. O emprego da diretiva `locating` determina a localização estrutural da classe na LFL.

Classe: 5.15 (Numeral)

```
Class Numeral ≪
      Number implementing < valuation _ :  Number → Integer > ≫
  locating LFL.Semantics.Expression.Shared
  using N:Number
  semantics:
     evaluate-numeral( N ) =
          give valuation N
End Class
```

A classe `Numeral` produz um número inteiro que representa o numeral. Esta classe ao ser instanciada requer o parâmetro `Number` seguido do método `valuation`. O emprego

da diretiva `locating` determina a localização estrutural da classe na LFL. O objeto `N` é instanciado, e seu método `valuation` produz um número inteiro que representa o numeral.

Classe: 5.16 (Operadores Lógicos)

```
Class LogicOperator ≪
      Expression implementing < evaluate _ :  Expression → Action > ≫
   locating LFL.Semantics.Expression.Shared
   using E1:Expression, E2:Expression
   semantics:
      evaluate-or( E1, E2 ) =
           evaluate ⟦ E1 ⟧ then
                  check (the given value is true)
              then
                  give true
            or
                  check (the given value is false)
              then
                  evaluate ⟦ E2 ⟧

      evaluate-and( E1, E2 ) =
           evaluate ⟦ E1 ⟧ then
                  check (the given value is true)
              then
                  evaluate ⟦ E2 ⟧
            or
                  check (the given value is false)
              then
                  give false
End Class
```

A classe `LogicOperator` (5.16) representa os operadores lógicos em linguagens de programação. Esta classe recebe o argumento `Expression` com o método `evaluate` como parâmetro. Este parâmetro é tratado como classe dentro do escopo atual, assim são instanciados os objetos E_1 e E_2. A localização da classe na LFL é atribuída pela diretiva `locating`. Em `LogicOperator` existem dois métodos que implementam os operadores "*or*" e "*and*".

O método `evaluate-or` realiza a avaliação dos objetos analisando da esquerda para direita, ou seja, o objeto E_1 será avaliado primeiro que E_2. A avaliação inicia-se pelo objeto E_1, a qual produz um valor-verdade referente ao resultado da ação. Caso o valor produzido neste resultado não seja verdadeiro, o objeto E_2 é avaliado.

A forma de avaliação dos objetos no método `evaluate-and` segue da esquerda para direita, ou seja, o objeto E_2 será avaliado mediante a produção de um valor verdadeiro decorrente da avaliação do objeto E_1.

Classe: 5.17 (Valor Verdade)

```
Class True
  locating LFL.Semantics.Expression.Shared
  semantics:
     evaluate-true( ) =
        give true
End Class
```

Classe: 5.18 (Valor Falso)

```
Class False
  locating LFL.Semantics.Expression.Shered
  semantics:
     evaluate-false( ) =
        give false
End Class
```

As classes `True` (5.17) e `False` (5.18) disponibilizam como *transients* os valores verdadeiro e falso, respectivamente. A localização da classe na LFL é instituída pela diretiva `locating`.

Classe: 5.19 (Avalia a Expressão Menor)

```
Class LessThan ≪
      Expression
         implementing < evaluate _ :  Expression → Action >  ≫
  locating LFL.Semantics.Expression.Shered
  using E1:Expression, E2:Expression
  semantics:
     evaluate-less-than( E1 , E2 ) =
           evaluate ⟦ E1 ⟧
        and
           evaluate ⟦ E2 ⟧
     then
        give less(the give integer#1, the give integer#2)
End Class
```

Classe: 5.20 (Igualdade entre Expressões)

```
Class Equality ≪
      Expression
         implementing < evaluate _ :. Expression → Action >  ≫
   locating LFL.Semantics.Expression.Shered
   using E1:Expression, E2:Expression
   semantics:
      evaluate-equality( E1 , E2 ) =
            evaluate ⟦ E1 ⟧
         and
            evaluate ⟦ E2 ⟧
      then
         give equal(the give integer#1, the give integer#2)
End Class
```

As classes `LessThan` (5.19), `Equality` (5.12) possuem uma estrutura comum entre si. Elas representam a escolha da expressão com menor valor e a igualdade entre expressões, respectivamente. Para qualquer uma dessas classes, o parâmetro `Expression` com o método `evaluate` devem ser informados, assim os objetos E_1 e E_2 são instanciados. O resultado final em cada classe é definido pelo uso das funções auxiliares `less( _ , _ )` ou `equal( _ , _ )` que efetuam as operações de menor valor e igualdade entre expressões.

5.5 Considerações Finais

Neste capítulo foi apresentada a LFL, a qual permite agrupar classes genéricas que poderão ser utilizadas para especificar linguagens de programação, incluindo linguagens de paradigmas distintos. A disposição das classes genéricas na LFL facilita o agrupamento de novas classes, bem como a utilização das mesmas. Quanto à manutenção destas classes, a mesma se torna fácil, pois modificando uma única classe, seu resultado se refletirá em todas as especificações que fazem uso dela.

CAPÍTULO 6

ESPECIFICAÇÃO DA LINGUAGEM μPASCAL

Neste capítulo será mostrada a especificação da μPascal, uma linguagem com características de programação imperativa similar a SPECIMEN [3] e IMP [17] [20]. Esta linguagem visa apresentar a utilização das classes contidas na LFL. Para a efetuar a especificação completa da linguagem, nas próximas seções serão apresentadas a sintaxe, as entidades semânticas e as classes da linguagem μPascal. Para facilitar a explicação dos elementos da linguagem, foram inseridos números de referência, os quais estão localizados à esquerda de cada elemento.

6.1 Sintaxe Abstrata da Linguagem

```
(1)   Programa =
          [[ "declare" Declaração "use-em" Comando ]].

(2)   Declaração =
          [[ "var" Identificador ":" Tipo ⟨ "=" Expressão ⟩? ]] |
          [[ "const" Identificador ":" Tipo "=" Expressão ]].

(3)   Tipo =
          "boolean" | "integer"
```

A definição de um programa é indicada pelo uso do texto `declare` seguido de uma declaração, sendo que esta declaração é válida dentro do bloco de comandos definido sob `use-em`. Em `Declaração` (2) são definidas as variáveis e as constantes, indicando seu tipo (3). Uma expressão define um valor inicial para variáveis e constantes, sendo que para as variáveis esta definição é opcional. Os tipos que podem ser usados nesta linguagem estão definidos na equação (3).

```
(4)  Comando =
        [[ "declare" Declaração "use-em" Comando ]] |
        {[ Identificador ":=" Expressão ]} |
        [[ "se" Expressão "então" Comando ⟨ "senão" Comando ⟩? fim-se ]] |
        {[ "repete" Comando "ate" Expressão ]} |
        [[ Comando ";" Comando ]].
```

A linguagem μPascal possui alguns comandos, conforme está apresentado em (4). Segue abaixo, respectivamente, a descrição de cada comando.

- Declaração seguida de Comandos: Trata-se da declaração de identificadores locais a um comando (`Comando`) da linguagem. Estas declarações são definidas dentro do escopo do comando após `use-em`;
- Atribuição: O resultado de uma expressão é atribuído a uma variável definida por `Identificador`;
- Seleção: Verifica se o valor resultante da `Expressão` é verdadeiro, então executa o primeiro `Comando`. Caso contrário será executado o segundo `Comando`. O bloco `"senão"` é opcional;
- Repetição de Comando: Representa uma repetição de `Comando` até que o valor resultante da `Expressão` seja verdadeiro, ou seja, o comando será executado enquanto o valor da expressão for falso;
- Seqüenciação: Efetua uma seqüência de execuções de comando.

As expressões que estão presentes em comando, são representadas pelo símbolo não-terminal `Expressão`, estão definidas a seguir (5).

```
(5)  Expressão =
        [[ "true" ]] |
        {[ "false" ]} |
        [[ Algarismo ]] |
        {[ Identificador ]} |
        [[ Expressão "+" Expressão ]] |
        {[ Expressão "-" Expressão ]} |
        [[ Expressão "*" Expressão ]] |
        {[ Expressão ">" Expressão ]} |
        [[ Expressão "maior que" Expressão ]].
        {[ Expressão "=" Expressão ]}.
```

```
(6)   Numeral =
          [[ Digit ⟨ Digit )* ]].

(7)   Identificador =
          [[ Letter ⟨ Letter | Digit )* ]].
```

As expressões a serem usadas na linguagem μPascal estão definidas em (5). Os tokens `"true"` e `"false"` definem os valores verdadeiro e falso em expressões lógicas. Uma expressão pode ser representada por um número inteiro ou por um identificador. Operações aritméticas também estão definidas em (5), respectivamente, representam a adição, subtração, multiplicação, comparação e igualdade entre expressões. Note que a gramática define duas formas sintáticas para a operação de comparação "maior que" e ">". Esta característica da linguagem será usada para melhor exemplificar o uso da LFL.

Em (6) o *sort* `Numeral` é definido. Os elementos deste *sort* poderão ser usados na linguagem para escrever números inteiros. O `Identificador`, em (7), define os elementos que podem ser identificadores.

6.2 Entidades Semânticas

Durante a especificação da linguagem μPascal, alguns *sorts* serão referenciados pelas classes. A seguir, estes *sorts* serão definidos.

```
(8)   value = integer | truth-value.
(9)   token = value.
(10)  bindable = value | cell.
(11)  cell = cell[truth-value] | cell[integer].
(12)  storable = value.
```

O *sorts* definidos em (8), (9) e (12), indicam que os valores manipulados nas classes serão inteiros ou valores-verdade. O *sort* `bindable` (10) define que seus elementos poderão ser valores ou células a serem mapeadas a identificadores que representam constantes e variáveis, respectivamente. As constantes utilizam o `bindable value` e as variáveis `cell`. Em (11) é definido que uma célula pode armazenar valores-verdade e números inteiros.

6.3 Classes (Funções Semânticas)

Nesta seção será apresentada a utilização da LFL para especificar a linguagem μPascal. Esta linguagem será mostrada em uma hierarquia de classes.

A ordem de apresentação das classes que formam a linguagem μPascal, será instituída a partir das classes auxiliares, posteriomente serão apresentadas as declarações, os comandos e as expressões. Na hierarquia atribuída às classes da linguagem, as classes auxiliares estão definidas em mesmo nível que as classes que especificam as Declarações, os Comandos e as Expressões. Estas classes auxiliares serão utilizadas (instanciadas) pelas classes mais genéricas da linguagem. A seguir serão apresentadas as classes auxiliares da linguagem μPascal.

6.3.1 Classes Auxiliares

Serão apresentadas as classes auxiliares `Identificador`, `Algarismo` e `Tipo`.

```
Class Identificador
  syntax:
     uId ::= letter[ letter  |  digit ]*
End Class
```

A classe `Identificador` representa os nomes de constantes e variáveis na linguagem. Nesta classe está definida apenas a sintaxe, sendo estabelecida a estrutura dos nomes na definição de `uId`.

```
Class Algarismo
  syntax:
     uA ::= digit+
  semantics:
     avaliação _ :  uN → integer
End Class
```

A classe `Algarismo` possui definições sintáticas (seção `syntax`) e semânticas (seção `semantics`). Na seção sintática, através do símbolo não-terminal `uA`, é definida a estrutura dos números inteiros. A seção semântica apresenta o método `avaliação`, o qual especifica o significado de um numeral. Este método está declarado conforme a definição de [17].

```
Class Tipo
  syntax:
     uT ::= "boolean" | "integer"
  semantics:
     allocate-for-type _ :  uT → Action

     allocate-for-type[[ boolean ]]  =
        allocate a cell[ truth-value ]

     allocate-for-type[[ integer ]]  =
        allocate a cell[ integer ]
End Class
```

A sintaxe e semântica dos tipos de dados de μPascal estão definidas na classe `Tipo`. Esta classe apresenta, em sua parte sintática, os tipos `"boolean"` e `"integer"`, os quais podem ser utilizados pela linguagem. A classe `Tipo` é utilizada nas declarações de variáveis, sendo necessário alocar espaço em memória. Esta alocação de memória é efetuada pelos métodos definidos na parte semântica da classe, onde estes métodos reservam uma célula de memória de acordo com o tipo definido. O método `allocate-for-type` estabelece duas ações possíveis, podendo reservar um valor-verdade ou um valor inteiro.

As classes que representam as declarações da linguagem μPascal estão descritas na próxima seção.

6.3.2 Declarações

As declarações na linguagem μPascal definem constantes e variáveis. A seguir são apresentadas classes `Declaração`, `Constante` e `Variavel`.

```
Class Declaração
  syntax:
     uDec
  semantics:
     elaborar _ :  uDec → Action
End Class
```

A classe `Declaração` é a mais genérica das declarações. O símbolo não-terminal `uDec` é definido na parte sintática e pode ser redefinido pelas sub-classes. Na seção semântica, apenas a *signature* do método `elaborar` é definido, indicando que o uso das declarações mapeia em ações.

```
Class Constante
   extending Declaração
   including LFL.Semantics.Declaration.Paradigm.Imper
   using uE:Expressão, uT:Tipo, uI:Identificador,
         objCst:Constant ≪ Identificador, Expressão<avaliar> ≫
   syntax:
         uDec ::= "const" uI ":" uT "=" uE
   semantics:
         elaborar[[ "const" uI ":" uT "=" uE ]]  =
            objCst.elaborate-constant( uI, uE )
End Class
```

A classe `Constante` especifica uma declaração de constante na linguagem μPascal. Esta classe foi estendida a partir da classe `Declaração`. A diretiva `including` é empregada para disponibilizar os métodos de todas as classes que estão associadas ao nodo indicado, ou seja, as classes do nodo `LFL.Semantics.Declaration.Paradigm.Imper` serão disponibilizadas dentro do escopo de `Constante`.

Os objetos `uI`, `uT` e `uE` são instâncias das classes `Identificador`, `Tipo` e `Expressão`, respectivamente. O objeto `objCst` é instanciado a partir da classe genérica ***Constant***, a qual pertence à LFL. A classe ***Constant*** requer argumentos como parâmetro, sendo passadas as classes `Identificador` e `Expressão`, esta última com o método `avaliar`.

Na parte sintática da classe, o símbolo não terminal `uDec` é redeclarado definindo a estrutura da declaração de constante. Na seção semântica, esta estrutura é elaborada pelo método `elaborar`, o qual pertence à classe pai (`Declaração`). O comportamento deste método é atribuído pelo método ***elaborate-constant***, o qual pertence a classe ***Constant*** (está classe está definida na seção 5.4.1.1 [pág. 56]).

```
Class Variavel
   extending Declaração
   including LFL.Semantics.Declaration.Paradigm.Imper
   using uE:Expressão, uT:Tipo, uI:Identificador,
         objVar:Variable ≪ Identificador, Tipo, Expressão<avaliar> ≫
         objVdc:VariableDec ≪ Identificador, Tipo ≫
   syntax:
         uDec ::= "var" uI ":" uT [ "=" uE ]
   semantics:
         elaborar[[ "var" uI ":" uT ]]  =
            objVdc.elaborate-variable( uI, uT )

         elaborar[[ "var" uI ":" uT "=" uE ]]  =
            objVar.elaborate-variable( uI, uT, uE )
End Class
```

Uma declaração de variável na linguagem μPascal é definida pela classe `Variavel`. Esta classe foi estendida a partir da classe `Declaração`. Para a classe atual acessar os métodos das classes que estão associadas em um determinado nodo da LFL, utiliza-se a diretiva `including`.

Os objetos `uI`, `uT` e `uE` são instâncias das classes `Identificador`, `Tipo` e `Expressão`, respectivamente. Os objetos `objVar` e `objVdc` são instanciados a partir das classes genéricas *`Variable`* e *`VariableDec`*, respectivamente. Ambas as classes genéricas pertencem à LFL. A classe *`Variable`* requer argumentos como parâmetro, os quais são passadas as classes `Identificador`, `Tipo` e `Expressão`, esta última com o método `avaliar`. O objeto `objVdc` é uma instância da classe *`VariableDec`*, a qual requer como parâmetros as classes `Identificador` e `Tipo`.

Na parte sintática da classe, uma estrutura de declaração de variável é redefinida em `uDec`. Na seção semântica, duas possibilidades de declaração de variáveis são disponibilizadas através da sobrecarga do método `elaborar`. A primeira contempla a declaração de variáveis, sem considerar a especificação de valor inicial para a mesma. A segunda forma de declaração é semelhante a primeira, porém é atribuído um valor inicial para a variável. Em ambas possibilidades de declaração, o comportamento de cada uma é disponibilizado pelo método *`elaborate-variable`*, sendo disponibilizado pelas classes *`Variable`* e *`VariableDec`*, respectivamente (estas classes estão definidas na seção 5.4.1.1 [pág. 56]).

6.3.3 Comandos

As classes que definem os comandos na linguagem μPascal são: `Comando`, `DeclareComando`, `Atribuição`, `Seleção`, `Repetição` e `SequenciaComando`. A seguir estas classes serão apresentadas.

```
Class Comando
  syntax:
     uCom
  semantics:
     executar _ :  uCom → Action
End Class
```

A classe Comando é a mais genérica dos comandos. O símbolo não-terminal uCom é definido na parte sintática e pode ser redefinido pelas sub-classes. Na seção semântica, o método executar estabelece o mapeamento da estrutura sintática uCom para uma ação correspondente.

```
Class DeclareComando
  extending Comando
  including LFL.Semantics.Command.Paradigm.Imper
  using uD:Declaração, uC:Comando,
        objDec:DecCommand ≪ Declaração<elaborar>, Comando<executar> ≫
  syntax:
        uCom ::= "declare" uD "use-em" uC
  semantics:
        executar[[ "declare" uD "use-em" uC ]]  =
           objDec.execute-dec-command( uD, uC )
End Class
```

A classe DeclareComando, estendida a partir da classe Comando, especifica declarações seguidas de comandos. A diretiva including é empregada para disponibilizar, dentro do escopo da classe atual, os métodos das classes que estão associadas ao nodo Imper.

São instanciados três objetos na classe DeclareComando: os objetos uD e uC são instâncias das classes Declaração e Comando, respectivamente. O terceiro objeto objDec é instanciado a partir da classe *DecCommand*, a qual pertence à LFL. Para instanciar esta classe, parâmetros devem ser informados. Estão sendo passados à *DecCommand*, as classes Declaração com seu respectivo método elaborar e Comando juntamente com o método executar.

O símbolo não terminal uCom é redeclarado na seção sintática da classe. Na seção semântica, a estrutura de declarações seguidas de comandos é executada pelo método executar, o qual está definido na classe pai (Comando). O método *execute-dec-command* pertencente a classe *DecCommand*, determina o comportamento para o método executar.

```
Class Atribuição
  extending Comando
  using uI:Identificador, uE:Expressão,
        objAtr:LFL.Semantics.Command.Paradigm.Imper.Assignment ≪
           Identificador, Expressão<avaliar> ≫
  syntax:
        uCom ::= uI ":=" uE
  semantics:
        executar[[ uI ":=" uE ]]  =
           objAtr.execute-assignment( uI, uE )
End Class
```

A classe `Atribuição` especifica a atribuição de um valor a um determinado identificador. Hierarquicamente, esta classe é uma especialização da classe `Comando`. Os objetos `uI` e `uE` são instanciados a partir das classes `Identificador` e `Expressão`. A partir da classe ***Assignment***, pertencente à LFL, o objeto `objAtr` é instanciado. As classes `Identificador` e `Expressão` juntamente com o método `avaliar`, são passadas como parâmetros para a classe ***Assignment***.

Na parte sintática da classe, o símbolo não terminal `uCom` define a estrutura de atribuição, a qual será mapeada, na seção semântica, pelo método `executar`. O comportamento deste método é estabelecido pelo método ***execute-assignment***, o qual pertence a classe ***Assignment*** da LFL (esta classe está definida na seção 5.6 [pág. 60]).

```
Class Seleção
  extending Comando
  using uC1:Comando, uC2:Comando, uE:Expressão,
        objSel:LFL.Semantics.Command.Shared.Selection ≪
           Comando<executar>, Expressão<avaliar> ≫
  syntax:
        uCom ::= "se" uE "então" uC1 [ "senão" uC2 ] "fim-se"
  semantics:
        executar[[ "se" uE "então" uC1 "fim-se" ]]  =
           objSel.execute-if-then( uE, uC1 )

        executar[[ "se" uE "então" uC1 "senão" uC2 "fim-se" ]]  =
           objSel.execute-if-then-else( uE, uC1, uC2 )
End Class
```

A execução de um comando de seleção na linguagem μPascal é definida pela classe `Seleção`, a qual é estendida partir da classe `Declaração`. Os objetos uC_1, uC_2 e `uE`, sendo os dois primeiros, instâncias das classes `Comando` e o último da classe `Expressão`.

A classe ***Selection***, pertencente à LFL, é instanciada no objeto `objSel`. As classes `Comando` com seu respectivo método `executar` e `Expressão` com o método `avaliar`, são passadas como parâmetros para ***Selection***.

Na parte sintática da classe `Seleção` é definido o formato da estrutura de seleção, informando a possibilidade da omissão do teste para posterior execução do segundo comando (["senão" uC_2]). Por este motivo, na seção semântica são definidas duas possibilidades para o método `executar`. Na primeira é possível executar apenas o primeiro comando (uC_1). Na segunda, em caso positivo o primeiro comando (uC_1) é executado, caso contrário, o segundo é executado (uC_2). O comportamento para cada possibilidade é estabelecido pelos métodos ***execute-if-then*** e ***execute-if-then-else***, respectivamente. Estes métodos pertencem a classe ***Selection*** da LFL (esta classe está definida na seção 5.9 [pág. 62]).

```
Class Repetição
   extending Comando
   using uC:Comando, uE:Expressão,
         objRep:LFL.Semantics.Command.Paradigm.Imper.Repeat ≪
            Comando<executar>, Expressão<avaliar> ≫
   syntax:
         uCom ::= "repita" uC "ate" uE
   semantics:
         executar[[ "repita" uC "ate" uE ]]  =
            objRep.execute-repeat( uC, uE )
End Class
```

A classe `Repetição`, estendida a partir da classe `Comando`, define uma estrutura de repetição de comandos para a linguagem μPascal. As classes `Comando` e `Expressão` são instanciadas pelos objetos `uC` e `uE`. O objeto `objRep` é uma instância da classe ***Repeat***, a qual pertence a LFL. Esta classe recebe como parâmetros as classes `Comando` juntamente com o método `executar` e `Expressão` com o método `avaliar`.

Na seção sintática da classe, o símbolo não terminal `uCom` é redefinido para a estrutura do comando de repetição. O método `executar`, na parte semântica, mapeia a estrutura definida em `uCom` para um ação. O comportamento deste método é atribuído pela chamada e passagem de parâmetros ao método ***execute-repeat***.

```
Class SequenciaComando
  extending Comando
  using uC1:Comando, uC2:Comando,
        objCmd:LFL.Semantics.Command.Shared.Sequence ≪ Comando<executar> ≫
  syntax:
        uCom ::= "sequencie" uC1 ";" uC2
  semantics:
        executar[[ "sequencie" uC1 ";" uC2 ]]  =
            objCmd.execute-sequence( uC1, uC2 )
End Class
```

A seqüência de comandos é expressa pela classe `SequenciaComando`, a qual é uma especialização da classe `Comando`. Os objetos uC_1 e uC_2 são instâncias da classe `Comando`. O objeto `objCmd` é uma instância da classe ***Sequence***, a qual pertence à LFL. Ao instanciar a classe ***Sequence***, é necessário passar como parâmetro a classe `Comando`.

A sintaxe da classe `SequenciaComando` define uma estrutura para seqüenciação de comandos. Na seção semântica, esta estrutura será mapeada para uma ação através do método `executar`, o qual terá seu comportamento definido pela utilização do método ***execute-sequence*** pertencente a classe ***Sequence***.

6.3.4 Expressões

Na linguagem μPascal são definidas algumas expressões aritméticas e lógicas. As classes `Expressão`, `ValorVerdade`, `ValorFalso`, `IdentificadorExp`, `Soma`, `Subtração`, `Multiplicação`, `VerificaMaiorQue`, `SimboloMaiorQue`, `PalavraMaiorQue` e `Igualdade` são definidas a seguir:

```
Class Expressão
  syntax:
     uExp
  semantics:
     avaliar _ :  uExp → Action
End Class
```

A classe `Expressão` é a mais genérica das expressões. O símbolo não-terminal `uExp` é definido na parte sintática e pode ser redefinido pelas sub-classes. Na seção semântica, o método `avaliar` estabelece o mapeamento da estrutura sintática `uExp` para uma ação correspondente.

```
Class ValorVerdade
  extending Expressão
  using objVvd:LFL.Semantics.Expression.Shared.True
  syntax:
        uExp ::= "true"
  semantics:
        avaliar[[ "true" ]]  =
            objVvd.evaluate-true( )
End Class
```

A classe `ValorVerdade` define como *transient* o valor *true*. Esta classe é estendida a partir da classe `Expressão`. O objeto `objVvd` é uma instância da classe ***True***, a qual pertence à LFL. Na semântica da classe `ValorVerdade`, o método `avaliar` é redeclarado, sendo seu comportamento definido pelo método ***evaluate-true***.

```
Class ValorFalso
  extending Expressão
  using objVfs:LFL.Semantics.Expression.Shared.False
  syntax:
        uExp ::= "false"
  semantics:
        avaliar[[ "false" ]]  =
            objVfs.evaluate-false( )
End Class
```

A classe `ValorFalso` define como *transient* o valor *false*. Esta classe é estendida a partir da classe `Expressão`. O objeto `objVfs` é uma instância da classe ***False***, a qual pertence à LFL. Na semântica da classe `ValorFalso`, o método `avaliar` é redeclarado, sendo seu comportamento definido pelo método ***evaluate-false***.

```
Class AlgarismoExp
  extending Expressão
  using A:Algarismo,
        objNum:LFL.Semantics.Expression.Shared.Numeral ≪ Algarismo<avaliação> ≫
  syntax:
        uExp ::= A
  semantics:
        avaliar[[ A ]]  =
            objNum.evaluate-numeral( A )
End Class
```

A classe `AlgarismoExp` representa a semântica de um numeral em expressões. Os objetos `A` e `objNum` são instâncias das classes `Algarismo` e ***Numeral***, respectivamente. Esta última classe, pertencente a LFL, requer argumento como parâmetro, o qual está

sendo passado a classe `Algarismo` juntamente com seu método `avaliação`. A semântica do método `avaliar` é determinada pelo método *evaluate-numeral*.

```
Class IdentificadorExp
  extending Expressão
  using Id:Identificador,
        objId:LFL.Semantics.Expression.Shared.Identifier ≪ Identificador ≫
  syntax:
        uExp ::= Id
  semantics:
        avaliar[[ Id ]]  =
            objId.evaluate-identifier( Id )
End Class
```

Um identificador na linguagem μPascal pode ser uma variável ou constante. A classe `IdentificadorExp` avalia o identificador e retorna o valor associado a ele. Nesta classe são usados os objetos `Id` e `objId`, os quais são instâncias das classes `Identificador` e *Identifier*, respectivamente. Esta última classe pertence à LFL, a qual recebe como parâmetro a classe `Identificador`[1].

Na sintaxe da classe `IdentificadorExp`, o não terminal `uExp`, é redeclarado com a estrutura sintática de um identificador. Esta estrutura é atribuída a `uExp` através do objeto `Id` (classe `Identificador`). O método `avaliar` expressa a semântica da avaliação do identificador que produz o valor relacionado a este. O comportamento desta avaliação é definido pelo método *evaluate-identifier*, o qual pertence a classe *Identifier* que está agrupada na LFL.

```
Class Soma
  extending Expressão
  using uE1:Expressão, uE2:Expressão,
        objSum:LFL.Semantics.Expression.Shared.Sum ≪ Expressão<avaliar> ≫
  syntax:
        uExp ::= uE1 " + " uE2
  semantics:
        avaliar[[ uE1 "+" uE2 ]]  =
            objSum.evaluate-sum( uE1, uE2 )
End Class
```

[1] A classe `Identificador` está definida na seção 6.3.1.

```
Class Subtração
   extending Expressão
   using uE1:Expressão, uE2:Expressão,
         objSub:LFL.Semantics.Expression.Shared.Subtract ≪ Expressão<avaliar> ≫
   syntax:
         uExp ::= uE1 " - " uE2
   semantics:
         avaliar[[ uE1 " - " uE2 ]]  =
            objSub.evaluate-sub( uE1, uE2 )
End Class
```

```
Class Multiplicação
   extending Expressão
   using uE1:Expressão, uE2:Expressão,
         objPrd:LFL.Semantics.Expression.Shared.Product ≪ Expressão<avaliar> ≫
   syntax:
         uExp ::= uE1 " * " uE2
   semantics:
         avaliar[[ uE1 " * " uE2 ]]  =
            objPrd.evaluate-prod( uE1, uE2 )
End Class
```

Uma estrutura comum na definição das classes `Soma`, `Subtração` e `Multiplicação` pode ser verificada nas três classes acima. Estas três classes são estendidas a partir da `Expressão`, também utilizam os objetos uE_1 e uE_2, os quais são instâncias da classe `Expressão`.

Na seção sintática, a diferença entre estas classes está na definição do operador binário "+", "-" e "*" para a representação da soma, subtração e multiplicação, respectivamente, entre expressões. Na parte semântica, a definição do comportamento para o método `avaliar`, o qual está contido nas classes `Soma`, `Subtração` e `Multiplicação`, é atribuída pelos objetos `objSum`, `objSub` e `objPrd`, respectivamente, sendo estes objetos instâncias das classes ***Sum***, ***Subtract*** e ***Product***, todas pertencentes à LFL. Respectivamente, estas classes da LFL especificam os métodos ***evaluate-sum***, ***evaluate-sub*** e ***evaluate-prod***, os quais estão apresentados na seção 5.4.3.1 [pág. 64].

```
Class VerificaMaiorExpressão
  using uE1:Expressão, uE2:Expressão
  semantics:
     avalie-maior-que( uE1, uE2 ) =
             avaliar ⟦ E1 ⟧ then give the value label#1
          and
             avaliar ⟦ E2 ⟧ then give the value label#2
        then
          give greater than(the give integer#1, the give integer#2)
End Class
```

A LFL, em sua primeira versão, agrupa classes que são compostas apenas pela especificação semântica (seção 5.2). A classe `VerificaMaiorExpressão` da linguagem μPascal possui, em sua definição, apenas a especificação semântica, pois esta classe será utilizada por outras dentro do escopo de μPascal. `VerificaMaiorExpressão` foi criada exclusivamente para esta linguagem, pois não é de uso comum a outras especificações. Por este motivo ela não será agregada à LFL. Desta maneira, pode-se notar que é possível criar uma biblioteca na própria especificação da linguagem.

A classe `VerificaMaiorExpressão` avalia duas expressões e indica, como resultado, a expressão que representa o maior valor. Esta classe utiliza os objetos uE_1 e uE_2, os quais são instâncias da classe `Expressão`. Na parte semântica, o método `avalie-maior-que` requer dois objetos do tipo `Expressão` como parâmetros. Na definição deste método ambos os objetos são avaliados. Em seguida, a função auxiliar `greater than( _ , _ )`, contida no método `avalie-maior-que`, produz um valor verdadeiro se o valor resultante da avaliação da expressão uE_1 for maior que o resultado de uE_2 e falso em caso contrário.

```
Class SimboloMaiorQue
  extending Expressão
  using uE1:Expressão, uE2:Expressão, objVme:VerificaMaiorExpressão
  syntax:
     uExp ::= uE1 ">"  uE2
  semantics:
     avaliar[[ uE1 ">" uE2 ]]  =
        objVme.avalie-maior-que( uE1, uE2 )
End Class
```

```
Class PalavraMaiorQue
  extending Expressão
  using uE1:Expressão, uE2:Expressão, objVme:VerificaMaiorExpressão
  syntax:
     uExp ::= uE1 "maior que" uE2
  semantics:
     avaliar[[ uE1 "maior que" uE2 ]]  =
        objVme.avalie-maior-que( uE1, uE2 )
End Class
```

As classes SimboloMaiorQue e PalavraMaiorQue definem entre duas expressões a que possui maior valor. A diferença entre estas classes está na definição sintática. A primeira classe define o símbolo ">" como operador lógico. A segunda, define as palavras "maior que" como operador lógico. Na semântica, comum em ambas as classes, são criados os objetos uE_1, uE_2 e objVme, sendo que os dois primeiros são instâncias da Expressão e o outro da classe VerificaMaiorExpressão. O mapeamento das avaliações em ações é produzido método avaliar. A especificação semântica deste método é referenciada pelo método avalie-maior-que, o qual pertence a classe VerificaMaiorExpressão.

A reutilização de partes da especificação definidas dentro da própria especificação da linguagem μPascal, pode ser vista nas classes SimboloMaiorQue e PalavraMaiorQue, pois o método avalie-maior-que implementa a semântica de uma determinada operação, a qual está sendo agregada à duas definições sintáticas.

```
Class Igualdade
  extending expressão
  using uE1:Expressão, uE2:Expressão,
        objEqy:LFL.Semantics.Expression.Shared.Equality ≪ Expressão<avaliar> ≫
  syntax:
        uExp ::= uE1 " = " uE2
  semantics:
        avaliar[[ uE1 " = " uE2 ]]  =
           objEqy.evaluate-equality( uE1, uE2 )
End Class
```

A classe Igualdade define uma operação lógica utilizando o simbolo "=" como operador binário. Esta classe é estendida a partir da classe Expressão. São utilizados nesta classe os objetos uE_1, uE_2 e objEqy, sendo os dois primeiros instâncias da classe Expressão e o outro instância da classe *Equality*, a qual pertence a LFL. Na seção

sintática, o símbolo não terminal `uExp` é redefinido para uma estrutura lógica que verifica a igualdade entre duas expressões. Na parte semântica, o comportamento do método `avaliar` é determinado pelo método *evaluate-equality*.

6.3.5 A Classe que Representa a Linguagem μPascal

Nas seções anteriores deste capítulo, foram definidas algumas classes referentes às declarações, comandos e expressões para a linguagem μPascal. Nesta seção será apresentada a classe que define a linguagem μPascal.

```
Class Linguagem$\mu$Pascal
  using uD:Declaração, uC:Comando
  syntax:
        uPrg ::= "declare" uD "use-em" uC
  semantics:
        execute _ :  uPrg → Action

        execute[[ "declare" uD "use-em" uC ]]  =
              elaborar ⟦ D ⟧
            hence
              executar ⟦ C ⟧
End Class
```

A classe `Linguagem$\mu$Pascal` define uma estrutura geral de um programa na linguagem μPascal. Os objetos `uD` e `uC` são instâncias das classes `Declaração` e `Comando`, respectivamente. A estrutura de um programa, é definida na parte sintática da classe pelo token `uPrg`. Na seção semântica é definido o método `execute`, o qual mapeia uma programa para uma determinada ação.

Com a especificação da classe `Linguagem$\mu$Pascal`, está completa a definição da linguagem de programação μPascal. Nas seções seguintes, esta linguagem será estendida com a introdução de novas características. Serão apresentadas as classes que definem as funções e procedimentos, bem como as classes que auxiliam neste processo.

6.4 Introduzindo Novas Características à Linguagem μPascal

Será apresentada nesta seção a extensão da linguagem μPascal, sendo incorporada duas estruturas fundamentais em linguagens de programação, serão definidas Funções e

Procedimentos.

Proveniente desta extensão, novas classes serão constituidas na linguagem μPascal. Em algumas destas classes, a especificação semântica será atribuída pelos métodos das classes pertencentes à LFL. Na apresentação da LFL, capítulo 5, as classes referentes a funções e procedimentos não foram apresentadas, entretanto estão descritas no Anexo B. Na seção a seguir, serão mostradas as mudanças na sintaxe da linguagem.

6.4.1 Extensão da Sintaxe Abastrata

A incorporação de procedimentos e funções na linguagem μPascal, requer a inclusão de novos elementos sintáticos na definição da linguagem[2]. A seguir estes elementos são apresentados. Foram atribuídas reticências (...) para indicar que a sintaxe da linguagem está sendo estendida.

```
(1)   Declaration =
            ... |
          [[ "proc" Identificador "(" Formal-Par ")" "=" "Comando" ]]
          [[ "func" Identificador "(" Formal-Par ")" "=" "Expressão" ]].

(2)   Formal-Par =
          [[ "var" Identificador ":" Tipo ]].
```

Nas declarações da linguagem μPascal, são inseridas duas regras sintáticas (1). Estas regras serão usadas para definir procedimentos e funções, respectivamente. Utilizando uma estrutura similar à definida em [17], é estabelecido que um procedimento executa um comando e uma função retorna o valor resultante da avaliação de uma expressão.

Na μPascal estendida, procedimentos e funções admitem um único parâmetro, o qual está definido na equação (2). A definição do parâmetro tem a mesma estrutura estabelecida para a declaração de variáveis (seção 6.1 [pág. 69]).

```
(3)   Comando =
            ... |
          [[ "chamar" Identificador "(" ⟨ Atual-Par ⟩ ")" ]].
```

[2] A definição original da sintaxe abastrata da linguagem μPascal está apresentada na seção 6.1 [pág. 69].

```
(4)   Expressão =
          ... |
        [[ Identificador "(" ⟨ Atual-Par ⟩ ")" ]].

(5)   Atual-Par =
        [[ Expressão ]].
```

Para utilizar os procedimentos e funções é necessário efetuar uma chamada dos mesmos. Está chamada está definida nas equações sintáticas (3) e (4), respectivamente. A regra para chamar um procedimento é adicionada às definições de `Comando`. A chamada de funções é especificada modificando as definições de `Expressão`. Tanto para procedimentos e funções apenas um único parâmetro deve ser informado. A definição desse parâmetro está especificada conforme a equação (5).

Na extensão da linguagem μPascal não houveram mudanças nas entidades semânticas. A seção seguinte apresenta a especificação das classes que definem os procedimentos e as funções.

6.4.2 Classes (Extensão das Funções Semânticas)

Nesta seção serão apresentadas as classes que especificam os procedimentos e as funções da linguagem μPascal. A ordem de apresentação destas classes será a mesma que foi adotada na especificação de classes original (seção 6.3 [pág. 72]).

6.4.2.1 Declarações

Inicialmente será definida a classe `FormalPar` que descreve os parâmetros formais para os procedimento e as funções. Em seguida serão apresentadas as classes `ProcedimentoDec` e `FunçãoDec`.

```
Class FormalPar
  extending Declaração
  including LFL.Semantics.Declaration.Paradigm.Imper
  using uI:Identificador, uT:Tipo,
        objVar:Variable ≪ Identificador, Tipo, Expressão<avaliar> ≫
  syntax:
        uFP ::= "var" uI ":" uT
```

```
  semantics:
        elaborarFP _ :  uFP → Action

        elaborarFP[[ "var" uI ":" uT ]]  =
           objVar.elaborate-variable( uI, uT )
End Class
```

A semântica dos parâmetros no momento da declaração dos procedimentos e funções é expressa pela classe `FormalPar`. Esta classe foi estendida a partir da classe `Declaração`. Para a classe atual acessar os métodos das classes que estão associadas um determinado nodo da LFL, utiliza-se a diretiva `including`.

Os objetos `uI` e `uT` são instâncias das classes `Identificador` e `Tipo`, respectivamente. O objeto `objVar` é instanciado a partir da classe genérica ***`Variable`***, a qual pertence à LFL. A classe ***`Variable`*** requer argumentos como parâmetro, os quais são passadas as classes `Identificador`, `Tipo` e `Expressão`, esta última com o método `avaliar`.

A sintaxe para declaração do parâmetro está definida em `uFP`. Na seção semântica, o mapeamento dos parâmetros formais em ações é definido pelo método `elaborarFP`. O comportamento atribuído à estrutura que está definida em `uFP` é especificado pelo método ***`elaborate-variable`***, o qual pertence a classe ***`Variable`*** da LFL [3].

```
Class ProcedimentoDec
  extending Declaração
  including LFL.Semantics.Declaration.Paradigm.Imper
  using uI:Identificador, uC:Comando, uFP:FormalPar,
       objPrd:Procedure ≪ Identificador, FormalPar<elaborarFP>,
                          Comando<executar> ≫
  syntax:
        uDec ::= "proc" uI "(" uFP ")" ":" uC
  semantics:
        elaborar[[ "proc" uI "(" uFP ")" ":" uC ]]  =
           objPrd.elaborate-proc-with-par( uI, uFP, uC )
End Class
```

A classe `ProcedimentoDec` elabora parâmetros formais e executa comandos. Os objetos `uI`, `uC` e `uFP` são instâncias das classes `Identificador`, `Comando` e `FormalPar`, respectivamente. A classe ***`Procedure`*** é instanciada no objeto `objPrd`. Esta classe pertence a LFL e está associada ao nodo `Imper`. A indicação de qual nodo está sendo disponibilizado à classe atual (`ProcedimentoDec`) é definida pela diretiva `including`.

[3] A classe ***`Variable`*** está definida na seção 5.4.1.1 [pág. 56].

Uma nova sintaxe de procedimento é definida em uDec. Na seção semântica, a estrutura de declaração de procedimentos é elaborada pelo método elaborar, o qual possui seu comportamento modelado pelo método *elaborate-proc-with-par*, que pertence a classe *Procedure*.

```
Class FunçãoDec
  extending Declaração
  including LFL.Semantics.Declaration.Paradigm.Imper
  using uI:Identificador, uFP:FormalPar, uE:Expressão,
       objFnd:Function ≪ Identificador, FormalPar<elaborarFP>,
                         Expressão<avaliar> ≫
  syntax:
        uDec ::= "func" uI "(" uFP ")" ":" uE
  semantics:
        executar[[ "proc" uI "(" uFP ")" ":" uC ]]  =
           objFnd.elaborate-func-with-par( uI, uFP, uE )
End Class
```

A declaração de funções, definida pela classe FunçãoDec, segue a mesma estrutura da classe anterior. Entretanto, o objeto objFnd é uma instância da classe *Function*, a qual pertence à LFL.

Em uDec, uma nova sintaxe é definida para a declaração de funções. O comportamento do método elaborar é definido pelo método *elaborate-func-with-par* da classe *Function*.

6.4.2.2 Comandos

A seguir será apresentada a classe ProcedimentoCom, a qual efetua a chamada de um procedimento. Na especificação desta classe, será utilizada a classe AtualPar, a qual está descrita na próxima seção.

```
Class ProcedimentoCom
  extending Comando
  including LFL.Semantics.Command.Paradigm.Imper
  using uI:Identificador, uAP:AtualPar,
        objPrc:Procedure ≪ Identificador, AtualPar<avaliarAP> ≫
  syntax:
        uCom ::= "chamar" uI "(" uAP ")"
  semantics:
        executar[[ "chamar" uI "(" uAP ")" ]]  =
           objPrc.execute-proc-without-par( uI, uAP )
End Class
```

Na classe ProcedimentoCom os objetos uI, uAP e objPrc são instâncias das classes Identificador, AtualPar e *Procedure*, respectivamente. Esta última classe, pertencente a LFL, é disponibilizada para o escopo atual através da diretiva including. Uma nova sintaxe para chamada de procedimento é redefinida em uCom. A semântica da classe ProcedimentoCom é especificada pela redefinição do método executar, declarado inicialmente em Comando. O comportamento deste método é descrito pelo método execute-proc-without-par, o qual pertence a classe *Procedure*.

Pode-se notar que a classe *Procedure* está sendo utilizada na especificação das classes ProcedimentoDec e ProcedimentoCom. Todavia, trata-se de classes distintas, pois estão agrupadas em nodos diferentes. Isto pode ser constatado pelo emprego da diretiva including.

6.4.2.3 Expressões

A seguir será definida a classe AtualPar que especifica os parâmetros no momento da chamada de procedimentos ou aplicações de funções. Posteriormente, a classe FunçãoExp será apresentada.

```
Class AtualPar
  extending Expressão
  using uE:Expressão
  syntax:
        uAP ::= uE
  semantics:
        avaliarAP _ :  uAP → Value

        avaliarAP[[ uE ]]  =
           avaliar ⟦ uE ⟧
End Class
```

A semântica dos parâmetros no momento da chamada de procedimentos ou aplicação de funções é expressa pela classe AtualPar. A especificação da sintaxe é tão somente uma expressão, conforme definido em uAP. Na seção semântica, o mapeamento dos parâmetros reais (expressões) em ações é definido pelo avaliarAP. A semântica deste método é provida pelo método avaliar, definido na classe Expressão (seção 6.3.4 [pág. 79]).

```
Class FunçãoExp
   extending Expressão
   including LFL.Semantics.Expression.Paradigm.Imper
   using uI:Identificador, uAP:AtualPar,
         objFnc:Procedure ≪ Identificador, AtualPar<avaliarAP> ≫
   syntax:
         uExp ::= uI "(" uAP ")"
   semantics:
         avaliar[[ uI "(" uAP ")" ]] =
            objFnc.evaluate-func-without-par( uI, uAP )
End Class
```

Na especificação da classe `FunçãoExp`, os objetos `uI`, `uAP` e `objPrc` são instâncias das classes `Identificador`, `AtualPar` e ***Function***, respectivamente. Esta última classe, pertencente a LFL, é disponibilizada para o escopo atual através da diretiva `including`. Um nova sintaxe para a aplicação de função é redefinida em `uExp`. A semântica da classe `FunçãoExp` é especificada pela redefinição do método `avaliar`, declarado inicialmente em `Expressão`. O comportamento deste método é descrito pela utilização do método `evaluate-proc-without-par`, o qual pertence a classe ***Function***.

Note que a classe ***Function*** está sendo utilizada na especificação das classes `FunçãoDec` e `FunçãoExp`. Todavia, trata-se de classes distintas, pois estão agrupadas em nodos diferentes. Isto pode ser constatado pelo emprego da diretiva `including`.

6.5 Considerações Finais

A longo deste capítulo foi apresentada a especificação da linguagem μPascal e uma extensão da mesma. Para a definição da linguagem μPascal, foram utilizadas várias classes que estão agrupadas na LFL. Com a utilização destas classes, verificou-se que a LFL possui, como características, facilidades de acesso às classes que já estão associadas, bem como a agregação de novas classes. Também notou-se que a forma de disponibilização dos métodos pelas classes da LFL, agiliza o processo de criação de classes na linguagem. Procedimentos e Funções foram adicionados à especificação da linguagem μPascal; este processo de extensão da linguagem utilizando classes da LFL, demonstrou facilidades na agregação de novas classes tanto na linguagem μPascal, quanto na LFL.

CAPÍTULO 7

FORMAÇÃO DA LINGUAGEM JOOS

Neste capítulo será especificada mais uma linguagem de programação utilizando a Semântica de Ações Orientada a Objetos. JOOS é uma linguagem orientada a objetos baseada em Java [8] [19]. Esta linguagem possui os principais conceitos de orientação a objetos, como classe, herança, métodos e construtores [8]. A especificação de JOOS apresentada neste capítulo foi definida a partir da especificação da linguagem em [18].

O intuito da apresentação da linguagem JOOS é demonstrar a capacidade de adequação das classes de paradigmas diferentes na LFL, bem como a reutilização de código referente a paradigmas distintos. Com a criação desta linguagem, novas classes serão agrupadas à LFL, ou seja, a biblioteca receberá classes do paradigma Orientado a Objetos (OO). A seguir serão apresentadas algumas partes da especificação de JOOS, sendo que a descrição completa está definida no Anexo D.

7.1 Sintaxe Abstrata

Nesta seção será apresentada a definição de alguns elementos da sintaxe abastrata da linguagem JOOS. Apenas os elementos que serão apresentados neste capítulo foram incluídos na gramática abaixo. A especificação completa está descrita no Anexo D. Foram inseridas reticências (...) para indicar o local onde existem outros elementos na sintaxe da linguagem.

```
(1)   Program =
        ⟦ "declare" Declaration "used-in" Statement ⟧.

(2)   Declaration =
        ... |
        ⟦ VariableDeclaration ⟧ | ⟦ FieldDeclaration ⟧

(3)   VariableDeclaration =
          ⟦ TypeDenoter Identifier ";" ⟧
```

```
(4)   FieldDeclaration =
            [[ "private" TypeDenoter Identifier  ]]

(5)   TypeDenoter =
        "int" | "boolean" | Identifier
```

Na equação (1) a definição de um programa é indicada pelo uso do texto `declare` seguido de uma declaração (`Declaration`), sendo que esta declaração é válida dentro do bloco de comandos definido sob `used-in`. Vários elementos são definidos em `Declaration` (2), dentre eles estão descritos o elemento `VariableDeclaration` e `FieldDeclaration`. Uma declaração de variável é expressa pela equação (3), onde deve ser informado o tipo (`TypeDenoter`) da variável antes do identificador da mesma. `FieldDeclaration` (4) mapeia um identificador a um tipo, porém há uma indicação do tipo de acesso a que este identificador está submetido. A aplicação do não-terminal `"private"` indica uma limitação de acesso ao identificador, ou seja, o identificador poderá ser acessado apenas dentro da classe que o declarou. Os tipos que poderão ser utilizados na linguagem JOOS são definidos em `TypeDenoter` (5).

```
(6)   Statement =
        ... |
        [[ "declare" Declaration "used-in" Statement ]] |
        [[ "return" ";" ]] |
        [[ "return" Expression ";" ]] |
        [[ "if" "(" Expression ")" Statement "else" Statement  ]]
```

A linguagem JOOS possui vários comandos. Dentre estes comandos, alguns dos mesmos são apresentados em `Statement` (equação 6 acima). Segue abaixo, respectivamente, a descrição de cada comando apresentado na equação (6).

- Declaração seguida de Comandos: Trata-se da declaração de identificadores locais a um comando (`Comando`) da linguagem. Estas declarações são definidas dentro do escopo do comando após `used-in`;
- Retorno Simples: Este comando quando avaliado retorna um valor vazio (*void*);
- Retorno com valor: Este comando efetua a avaliação de uma expressão e retorna, como dado *transient*, o resultado desta avaliação;

- *Seleção*: Verifica se o valor resultante de `Expression` é verdadeiro, então executa o primeiro `Statement`. Caso contrário será executado o segundo `Statement`.

```
(7)   Expression =
        ... |
        [[ Identifier ]] |
        [[ PrefixOperator Expression ]] |
        [[ Expression InfixOperator Expression ]] |
        [[ StatementExpression ";" ]]

(8)   Identifier =
        [[ Letter ( Letter | Digit )* ]]

(9)   PrefixOperator =
        [[ "!" | "-" ]]

(10)  InfixOperator =
        [[ "!=" | "<" | ">" | "<=" | ">=" |
           "==" | "+" | "-" | "*" | "/" | "%" ]]

(11)  StatementExpression =
        ... |
        [[ Identifier "=" Expression ]]
```

Algumas das expressões a serem usadas na linguagem JOOS estão definidas em (7). A linguagem define que uma expressão pode ser representada por um identificador. Também é permitido que sejam efetuadas operações lógicas e aritméticas entre expressões. Em `StatementExpression` são definidos vários elementos, como atribuição, criação de objetos, etc. Na equação (8) é definido o formato para os identificadores. Nas equações (9) e (10) são estabelecidos os formatos dos elementos para `PrefixOperator` e `InfixOperator`, respectivamente. Na equação (11) existem várias expressões definidas. Dentre estas expressões, é apresentada a frase que representa a atribuição. A atribuição é constituída pelo resultado de uma expressão, o qual é atribuído a uma variável definida por `Identifier`.

As **Equações Semânticas** da linguagem JOOS estão descritas no Anexo D. Elas não serão apresentadas neste capítulo por não serem o foco principal de nossos estudos. A seguir serão apresentadas as classes correspondentes aos elementos definidos na sintaxe abstrata.

7.2 Classes (Funções Semânticas)

Serão apresentadas nesta seção algumas classes que compõem a linguagem JOOS. A seguir estão descritas apenas as classes correspondentes às regras de sintaxe abstrata apresentadas na seção 7.1.

7.2.1 Classes Auxiliares

Como classe auxiliar, será apresentada a classe `Identifier`.

Classe: 7.1 (Declaração de Identificador)

```
Class Identifier
  syntax:
     uId ::= Letter ( Letter | Digit )*
End Class
```

A classe `Identifier` representa o nome de várias categorias de elementos (variáveis, objetos, classes, tipos, etc.) da linguagem. Nesta classe está definida apenas a sintaxe, sendo estabelecida a estrutura dos nomes na definição de `uId`. A seguir serão apresentadas as classes que compõem as declarações da linguagem.

7.2.2 Declarações

Algumas das classes que definem as declarações na linguagem JOOS são apresentadas nesta seção. As classes `VariableDeclaration`, `FieldDeclaration` e `TypeDenoter` estão descritas a seguir.

Classe: 7.2 (Declaração de Variável)

```
Class VariableDeclaration
    extending Declaration
    using TD:TypeDenoter, I:Identifier
    syntax:
         Dec ::= TD I ";"
```

```
    semantics:
        elaborate _ :  VariableDec* → Action

        elaborate ⟦ TD I ";" ⟧ =
            give default value of type denoted by ⟦ TD ⟧ then
            allocate a variable then
            bind I to the variable
End Class
```

Uma declaração de variável com atribuição de valor inicial (*default*) na linguagem JOOS é expressa pela classe `VariableDeclaration`, a qual foi estendida a partir da classe `Declaration`. Nesta classe são criados os objetos `I` e, `TD`, os quais são instâncias das classes `Identifier` e `TypeDenoter`, respectivamente.

Na parte sintática da classe, uma estrutura de declaração de variável é redefinida em `Dec`. Na seção semântica, através da sobrecarga do método `elaborate`, o identificador é mapeado à uma variável alocada em memória, sendo um valor inicial (*default*) associado a esta variável.

Classe: 7.3 (Declaração de um Campo Privado)

```
Class FieldDeclaration
    extending Declaration
    using I:Identifier, TD:TypeDenoter,
    syntax:
        Dec ::= "private" TD I ";"
    semantics:
        type bindings of ⟦ "private" TD I ";" ⟧ =
            the map of I
            to the type denoted by ⟦ TD ⟧
End Class
```

Com limitação ao escopo da classe, o mapeamento de um identificador a um tipo é definido pela classe `FieldDeclaration`. Esta classe foi estendida a partir da classe `Declaration`. Os objetos `I` e `TD` são instâncias das classes `Identifier` e `TypeDenoter`, respectivamente.

Na parte sintática da classe é redefinido, em `Dec`, uma estrutura que efetua o mapeamento de um identificador a um tipo. Os tipos que podem ser mapeados estão definidos na classe `TypeDenoter`. A aplicação do não-terminal `"private"` indica uma limitação de acesso ao identificador, ou seja, o identificador poderá ser acessado apenas dentro da classe

que o declarou. Na seção semântica, através da sobrecarga do método `type bindings of` é contemplado o mapeamento de um identificador à um tipo definido na linguagem.

Classe: 7.4 (Definição de Tipos)

```
Class TypeDenoter
    extending Declaration
    using I:Identifier
    syntax:
         TypeDenoterDec ::= "boolean" | "int" | I
    semantics:
         type denoted by [[ "boolean" ]] =
              booleanType

         type denoted by [[ "int" ]] =
              integerType

         type denoted by [[ I ]] =
              referenceType
End Class
```

A classe `TypeDenoter` define os tipos que poderão ser utilizados na linguagem JOOS. Esta classe é uma especialização de `Declaration`. O objeto `I` é uma instância da classe `Identifier`. Na parte sintática da classe, são definidos os tipos que a linguagem JOOS poderá utilizar. Na semântica da classe, o método `type denoted by` será sobrecarregado para cada tipo declarado na sintaxe. Este método produzirá um valor relacionado ao tipo informado.

7.2.3 Comandos

Na linguagem JOOS existem várias classes que definem os comandos. Dentre estas classes, a seguir serão apresentadas as classes `DecStatement`, `Selection` e `Return`.

Classe: 7.5 (Declaração de Comandos)

```
Class DecStatement
   extending Statement
   including LFL.Semantics.Command.Paradigm.Imper
   using D:Declaration, S:Statement,
         objDec:DecCommand ≪ Declaration<elaborate>, Statement<execute> ≫
   syntax:
         Stmt ::= "declare" D "used-in" S
   semantics:
         executar[[ "declare" D "used-in" S ]]  =
            objDec.execute-dec-command( D, S )
End Class
```

A classe DecStatement, estendida a partir da classe Statement, especifica declarações seguidas de comandos. A diretiva including é empregada para disponibilizar, dentro do escopo da classe atual, os métodos das classes que estão associadas ao nodo LFL.Semantics.Declaration.Paradigm.Imper.

São instanciados três objetos na classe DecStatement: os objetos D e S são instâncias das classes Declaration e Statement, respectivamente. O terceiro objeto objDec é instanciado a partir da classe *DecCommand*, a qual pertence à LFL. Para instanciar esta classe, parâmetros devem ser informados. Como parâmetros, estão sendo passados à *DecCommand*, as classes Declaration com seu respectivo método elaborate e Statement juntamente com o método execute.

O símbolo não terminal Stmt é redeclarado na seção sintática da classe. Na seção semântica, a estrutura de declarações seguidas de comandos é executada pelo método execute, o qual está definido na classe pai (Statement). O método *execute-dec-command* pertencente a classe *DecCommand* da LFL, determina o comportamento para o método execute.

Classe: 7.6 (Comando de Seleção)

```
Class Selection
    extending Statement
    using S1:Statement, S2:Statement, E:Expression,
          objSel:LFL.Semantics.Command.Shared.Selection ≪
          Statement<execute>, Expression<evaluate> ≫
    syntax:
          Stmt ::= "if" "(" E ")" S1 "else" S2
    semantics:
          execute ⟦ "if" "(" E ")" S1 "else" S2 ⟧ =
               objSel.execute-if-then-else( E, S1, S2 )
End Class
```

A execução de um comando de seleção na linguagem JOOS é definida pela classe Selection, a qual é estendida partir da classe Statement. Os objetos S_1, S_2 e E são instanciados, sendo os dois primeiros, instâncias das classe Command e o último da classe Expression. A classe *Selection*, pertencente à LFL, é instanciada no objeto objSel. As classes Statement com seu respectivo método execute e Expression com o método evaluate, são passadas como parâmetros para *Selection*.

Na parte sintática da classe `Selection` é definido o formato da estrutura de seleção. Na seção semântica, a expressão representada pelo objeto `E` produz um valor-verdade mediante a chamada do método `evaluate`, que em caso positivo o primeiro comando (S_1) será executado, e caso contrário, o segundo comando (S_2) é executado. O comportamento do método `execute` é estabelecido pelo método ***execute-if-then-else***, pertencente a classe ***Selection*** da LFL (esta classe está definida na seção 5.9 [pág. 62]).

Note que na declaração das duas classes acima (`DecStatement` e `Selection`), estão sendo usadas classes da LFL que estão associadas a nodos que não fazem parte da estrutura destinada à especificação de conceitos de orientação a objetos na LFL. Desta forma, há uma reutilização efetiva de código das classes destinadas a outros paradigmas na LFL.

Classe: 7.7 (Declaração do Tipo de Retorno)

```
Class Return
    extending Statement
    including LFL.Semantics.Command.Paradigm.OO
    using E:Expression, objNvr:ReturnWithoutValue,
          objRet:ReturnWithValue ≪ Expression<evaluate> ≫
    syntax:
          Stmt ::= "return" ";"  |  "return" E ";"
    semantics:
          execute ⟦ "return" ";" ⟧ =
               objNvr.execute-return( )

          execute ⟦ "return" E ";" ⟧ =
               objRet.execute-return-exp( E )
End Class
```

Na linguagem JOOS existem dois tipos de retornos que podem ser utilizados. Esses retornos, geralmente, são utilizados em métodos para devolver algum tipo de valor ao ambiente. A classe `Return` é uma especialização de `Statement`. O objeto `E` é uma instância da classe `Expression`. Os objetos `objNvr` e `objRet` são instâncias das classes ***ReturnWithoutValue*** e ***ReturnWithValue***, respectivamente. Estas classes pertencem à LFL. Para o instanciamento da classe ***ReturnWithValue***, é necessário informar como parâmetro a classe `Expression`, juntamente com o método `evaluate`.

A sintaxe da classe `Return` define duas estruturas para retorno. A primeira (`"return" ";"`) não retorna valor algum (*void*). A segunda estrutura (`"return" E ";"`) retorna uma

expressão (valor). Na parte semântica, o método `execute` é sobrecarregado, tendo seu comportamento estabelecido pelos métodos *execute-return* e *execute-return-exp*. Estes métodos pertencem às classes da LFL *ReturnWithoutValue* e *ReturnWithValue*, respectivamente.

7.2.4 Expressões

Várias expressões são definidas para a linguagem JOOS. Nesta seção serão apresentadas as classes `Identifier`, `PrefixOperator`, `InfixOperator` e `StatementExpression`. Todas as classes referentes a expressão estão apresentadas no anexo D.

Classe: 7.8 (Identificadores)

```
Class Identifier
    extending Expression
    including LFL.Semantics.Expression.Shared
    using objId:Identifier ≪ Identifier ≫
    syntax:
         Id ::= Letter ⟨ Letter | Digit ⟩*
    semantics:
         evaluate ⟦ Id ⟧ =
              objId.evaluate-identifier( Id )
End Class
```

Um identificador na linguagem JOOS pode ser expresso por letras ou dígitos. A classe `Identifier` avalia o identificador e retorna o valor associado a ele. Nesta classe é usado o objeto `objId`, o qual é uma instância da classe *Identifier*. Esta classe pertence à LFL. Para instanciar a classe *Identifier* da LFL, é necessário passar como parâmetro a classe `Identifier` (esta classe é a atual, onde está sendo declarado o objeto `objId`).

Na sintaxe da classe `Identifier`, é definida a estrutura sintática de um identificador. O método `evaluate` expressa a semântica da avaliação do identificador que produz o valor relacionado a este. O comportamento desta avaliação é definido pelo método *evaluate-identifier*, o qual pertence a classe *Identifier* que está agrupada na LFL. (Esta classe está definida na seção 6.3.1).

Classe: 7.9 (Operadores Prefixos)

```
Class PrefixOperator
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using Exp:Expression,
          objPfxOp:PrefixOperator ≪ Expression<evaluate>,
                                    PrefixOperator<apply prefix> ≫
    syntax:
          PreOp := "!" | "-"
    semantics:
          apply prefix _ :  PreOp → Action

          apply prefix ⟦ "!" ⟧ =
               give not (the boolean)

          evaluate ⟦ PreOp, Exp ⟧ =
               objPfxOp.evaluate-prefix( PreOp, Exp )
End Class
```

A classe `PrefixOperator` aplica um operador a uma expressão, desta maneira o resultado obtido a partir da aplicação deste operador é diferente do valor original da expressão. Nesta classe são criados os objetos `E` e `objPfxOp`, os quais são instâncias das classes `Expression` e ***`PrefixOperator`***. Para esta última classe, a qual pertence à LFL, é passado como parâmetro a classe `Expression` com o método `evaluate`.

Na seção sintática da classe `PrefixOperator` são definidos os operadores que deverão anteceder à expressão. O comportamento de cada operador determina um resultado diferente ao valor original da expressão. Na parte semântica, é definido o comportamento de um operador através da sobrecarga do método `evaluate`, o qual avalia a expressão e aplica o método `apply prefix`.

Classe: 7.10 (Operadores Infixos)

```
Class InfixOperator
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using E1:Expression, E2:Expression,
          objIfxOp:InfixOperator ≪ Expression<evaluate>,
                                   InfixOperator<apply prefix> ≫
    syntax:
          InfOp := "!=" | "<" | ">" | "<=" | ">=" | "=="
                   "+" | "-" | "*" | "/" | "%"
```

```
    semantics:
          apply infix _ :  InfOp → Action

          evaluate ⟦ E1 InfOp E2  ⟧ =
                   objIfxOp.evaluate-infix( E1, InfOp, E2 ) =

          apply infix ⟦ "==" ⟧ =
                   select
                       when (the boolean #1 = the boolean #2 ) or
                       when (the integer #1 = the integer #2 ) or
                       when (the reference #1 is identical to the reference #2 )
                   then give true
                   otherwise give false

          apply infix ⟦ "+" ⟧ =
                   give (the integer #1 + the integer #2 )

          %(Os outros operadores infixos são similares)%
End Class
```

A classe `InfixOperator` especifica a utilização de operadores lógicos e aritméticos entre expressões. Nesta classe são criados os objetos E_1, E_2 e `objIfxOp`, sendo os dois primeiros instâncias da classe `Expression` e o último, instância da classe *PrefixOperator*. Para esta classe, a qual pertence a LFL, é passado como parâmetro à classe `Expression` com o método `evaluate`.

Na seção sintática da classe `InfixOperator` são definidos os operadores que deverão ser aplicados entre expressões. Na parte semântica, é definido o comportamento de um operador através da sobrecarga do método `evaluate`, o qual avalia as expressões e aplica o método `apply infix`.

Classe: 7.11 (Comandos em Expressões)

```
Class StatementExpression
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, I:Identifier,
          objAtt:Attribution ≪ Identifier, Expression<evaluate> ≫,
          ...
    syntax:
          Exp ::= I "=" E  |  ...
    semantics:
          evaluate ⟦  I "=" E  ⟧ =
                objAtt.evaluate-attribution( I, E )

          ...
End Class
```

Alguns comandos são executados juntamente com avaliações de expressões. A classe `StatementExpression` contém alguns métodos que possuem esta característica. Nesta classe existem várias definições sintáticas distintas, dentre a quais tem a atribuição de valor a um identificador. As reticências (...) foram usadas para indicar que existem outros elementos na definição da classe `StatementExpression`.

Na seção sintática da classe, são definidas várias estruturas, dentre estas é definida uma estrutura de atribuição de um valor à um identificador. Na parte semântica, estas estruturas são avaliadas através da sobrecarga do método `evaluate`. Para a atribuição, conforme definido na classe `StatementExpression`, o comportamento do método `evaluate` é estabelecido pelo método *`evaluate-attribution`*, o qual pertence à classe *`Attribution`* da LFL.

7.3 Considerações Finais

Neste capítulo foi apresentado um estudo de caso envolvendo a criação da linguagem de programação JOOS [18]. A especificação desta linguagem foi importante para demonstrar a utilização das classes que estão agrupadas na LFL, bem como a agregação de novas classes genéricas. Algumas classes com características distintas à orientação a objetos foram empregadas na especificação de JOOS, ou seja, foram usadas classes genéricas de paradigmas diferentes ao paradigma orientado a objetos. A utilização destas classes genéricas na especificação de JOOS, demonstrou certas facilidades devido a estrutura a qual estas classes foram constituídas. O comportamento de uma determinada sintaxe da linguagem JOOS pode ser estabelecido através de métodos, os quais são disponibilizados por objetos provenientes das instâncias de classes genéricas.

A apresentação da linguagem JOOS, foi essencial para prover mais um estudo de caso que utiliza a LFL como repositório de especificações em Semântica de Ações Orientada a Objetos. Nesta apresentação, pode-se notar certas facilidades para reutilização e inclusão de classes na LFL.

CAPÍTULO 8

CONCLUSÕES E TRABALHOS FUTUROS

A Semântica de Ações Orientada a Objetos [1] [2] é o resultado da aplicação de conceitos de orientação a objetos na definição de um formalismo baseado em Semântica de Ações. O formalismo inclui construtores sintáticos para a formação de classes e métodos. Em um projeto que utiliza a Semântica de Ações Orientada a Objetos para especificar uma linguagem, novas classes são criadas e algumas são reutilizadas de outros projetos. Este reaproveitamento de classes é pertinente à orientação a objetos, porém a estrutura organizacional estabelecida na versão inicial da Semântica de Ações Orientada a Objetos define que a reutilização de uma classe seja caracterizada por uma cópia da mesma no projeto atual. Desta forma, o reaproveitamento de classes se torna inadequado.

Com o intuito de sanar este problema, neste trabalho é proposta a ***Language Features Library*** - **LFL** que possui como função o agrupamento e disponibilização de classes genéricas a várias especificações de linguagens de programação. A disposição destas classes na LFL, facilita a asssociação de novas classes e a utilização das mesmas. Quanto à manutenção das classes, a mesma é facilitada, pois modificando uma única classe, as alterações serão refletidas em todas as especificações que fazem uso da classe.

Com o propósito de validar os conceitos definidos para a LFL, nos capítulos 6 e 7 foram apresentados estudos de caso. Nestes estudos, incialmente foram associadas várias classes genéricas à LFL. A partir disso, estas classes foram utilizadas para especificar dois projetos de linguagens de programação:

- μPascal é uma linguagem com características do paradigma imperativo.
- JOOS é uma linguagem do paradigma orientado a objetos.

O processo de especificação de ambas as linguagens, contribuiu ao desenvolvimento da LFL tanto no seu conteúdo como na sua organização. Neste processo de desenvolvimento, várias classes consideradas genéricas foram inseridas e modificadas na LFL. A

manipulação de classes na LFL, demonstrou ser eficiente, pois a maneira pela qual a LFL foi estruturada, resulta em pouco esforço para sua utilização.

Como contribuição neste trabalho pode-se sumarizar os seguintes itens:

- A criação da LFL seguindo uma hierarquia organizacional, visando evitar a duplicação de classes semelhantes em vários projetos. Desta forma, qualquer modificação nas classes da LFL refletirá em todos os projetos que fazem uso da mesma. O modelo de árvore adotado para organizar estruturalmente as classes genéricas na LFL, facilita a localização e utilização das mesmas. Este modelo também simplifica a extensão da LFL, devido a organização das classes na LFL seguirem uma hierarquia.

- Inclusão de diretivas na notação da Semântica de Ações Orientada a Objetos para permitir as operações de inclusão, alteração e exclusão de classes na LFL. Seguindo o mesmo padrão estabelecido em Semântica de Ações Orientada a Objetos, estas diretivas podem ser aplicadas na especificação de classes genéricas (classes pertencentes à LFL), bem como nas classes que são especificadas em um projeto de linguagem de programação.

Como trabalhos futuros, alguns itens apresentados nesta proposta podem ser expandidos. Estes itens estão descritos a seguir:

- Descrição de outras linguagens utilizando a LFL. Desta forma, podem ser verificadas mais a fundo as capacidades e limitações dos conceitos propostos para uma primeira versão da LFL.

- Continuar o processo de desenvolvimento da LFL, incluindo novas funcionalidades na hierarquia. Em particular, estabelecer a estruturação para as classes que pertencem à sintaxe (*Syntax*) e entidades semânticas (*Entity*).

- Com o intuito de prover facilidades durante o processo de especificação de uma linguagem de programação, seria necessário o fornecimento de um suporte computacional. Este suporte poderia ser definido em quatro módulos básicos:

- O auxílio para descrever a linguagem de programação proveria um editor de especificações. Neste editor estariam sendo disponibilizadas as estruturas necessárias para a especificação em Semântica de Ações Orientada a Objetos de uma linguagem.
- Um *browser* para a LFL, por exemplo, onde seriam definidos mecanismos para criação, alteração e exclusão de classes. Estes mecanismos, também poderiam estar presentes na especificação de linguagens de programação.
- Poderiam ser criados mecanismos para a verificação estática da especificação da linguagem, onde seria feita uma análise para identificação de falhas no projeto. Por exemplo, verificar se uma determinada classe da LFL está recebendo seus parâmetros corretamente quando está for instanciada, indentificar se os elementos que estão sendo utilizados na especificação estão definidos, ou seja, verificar o elemento descrito em uma função semântica.
- Construção de um "animador" de especificações, de forma a permitir interações reais entre o projeto especificado e o usuário. Assim, o usuário poderá atribuir valores reais a uma especificação.

BIBLIOGRAFIA

[1] C. Carvilhe e M.A. Musicante. Object-oriented action semantics specifications. *Journal of Universal Computer Science*, 9(8):910–934, August de 2003. http://www.jucs.org/jucs_9_8/object_oriented_action_semantics.

[2] Cláudio R. V. Carvilhe. *Semântica de Ações Orientada a Objetos.* Dissertação de Mestrado, Universidade Federal do Paraná, 2002.

[3] Hermano Perrelli de Moura. *Action Notation Transformations.* Tese de Doutorado, University of Glasgow, 1993.

[4] Luis Carlos de Sousa Menezes. Uso de orientação a objetos na prototipação de semântica de ações, 1998.

[5] Kyung-Goo Doh e Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, volume 47 - Issue 1, Abril de 2003.

[6] H. M. Deitel e P. J. Deitel. *Java, como programar.* Editora Bookman, Porto Alegre, RS, 2001.

[7] José Davi Furlan. *Modelagem de Objetos Através da UML - The Unified Modeling Language.* Makron Books do Brasil Editora Ltda, 1998.

[8] Cay S. Horstmann e Gary Cornell. *The Sun Microsystems Press Core Series. Core Java 2*, volume 1-Fundamentals. Prentice Hall, fifth edition, 2000.

[9] Luis Carlos Menezes e Hermano P. de Moura. Component-based action semantics: A new approach for programming language specifications. *SBLP 2001 - V Simpósio Brasileiro de Linguagens de Programação*, páginas 152–163, 2001. Paraná, Brasil. Universidade Federal do Paraná.

[10] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[11] Peter D. Mosses. Theory and practice of action semantics. Report series rs-96-53, BRICS, Departament of computer science. University of Aarhus, 1996.

[12] Peter D. Mosses. A modular sos for action notation. Technical report series rs-99-56, BRICS, Departament of computer science. University of Aarhus, 1999.

[13] Roger S. Pressman. *Software Engineering. A Practitioner's Aprroach.* McGraw Hill International Editions, fourth edition, 1997.

[14] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn Bacon, 1986.

[15] Kenneth Slonnerger e Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languagens - A laboratory based approach.* AddisonWesley Publishing Company, Inc., 1995.

[16] Daniel T. Xavier Tadao Takahashi, Hans K. E. Liesenberg. *Programação orientada a objetos - uma visão integrada do paradigma de objetos.* Editora IME-USP, 1990.

[17] David Watt. *Programming Language Syntax and Semantics.* Prentice Hall International, UK, 1991.

[18] David A. Watt. Joos action semantics, 2000. Draft version 2.0, http://www.dcs.gla.ac.uk/daw/publications/JOOS2.ps.

[19] David A. Watt e Deryck F. Brown. *Java Collections. An introduction to abstract data types, data structures and algorithms.* John Wiley and Sons, 2001.

[20] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction Foundations of Computing Series.* MIT Press, 1993.

ANEXO A

SINTAXE DA NOTAÇÃO DA SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

A.1 Notação de Classes

Class-Molule =
⟦ "Class" Class-Name Class-Body "End-Class" ⟧ |
⟦ "Class" Class-Name "≪" Parameter-Class "≫" Class-Body "End-Class" ⟧.

Class-Body =
⟦ ⟨ "extending" Base-Class-Name ⟩$^{?}$
⟨ "locating" Structure-Locate ⟩$^{?}$
⟨ "including" Structure-Locate ⟩$^{?}$
⟨ "using" Objects-Declaration ⟩$^{?}$
⟨ Class-Definition ⟩$^{?}$ ⟧.

Base-Class-Name =
⟦ Class-Name | Class-Name "::" Base-Class-Name ⟧.

Objects-Declaration =
⟦ Object-Declaration ⟧ | ⟦ Object-Declaration "," Objects-Declaration ⟧.

Object-Declaration =
⟦ Identifier ":" ⟨ Structure-Locate "." ⟩$^{?}$ Class-Name |
Identifier ":" ⟨ Structure-Locate "." ⟩$^{?}$ Class-Name "≪" Paremeter-Class "≫" ⟧.

Parameter-Class =
⟦ Class-Name |
Class-Name "implementing" "<" Method-Name ">" |
Parameter-Class "," Parameter-Class* ⟧.

Structure-Locate =
⟦ Node-Name | Node-Name "." Structure-Locate* ⟧.

A.2 Definição do Corpo das Classes

Class-Definition =
⟦ "syntax" ":" Syntatic-Part "semantic" Semantic-Part ⟧.

Syntatic-Part =
⟦ Token-Name | Token "::=" syntax-tree ⟧.

Semantic-Part =
⟦ ⟨ Semantic-Functions ⟩? ⟨ Semantic-Equations ⟩? ⟧.

Semantic-Functions =
⟦ Semantic-Function ⟧ | ⟦ Semantic-Function Semantic-Functions ⟧.

Semantic-Function =
⟦ Function-Name "_ " * Tokens "→" "Action" | data ⟧.

Tokens =
⟦ TokenName ⟧ | ⟦ TokenName "," Tokens ⟧.

Semantic-Equations =
⟦ Semantic-Equation ⟧ | ⟦ Semantic-Equation Semantic-Equations ⟧.

Semantic-Equation =
⟦ Function-Name "[[" syntax-tree "]]" "=" Action ⟧ |
⟦ Function-Name "(" ⟨ data ⟩? ")" "=" Action ⟧

A.3 Notação de Ações

Action =
⟦ "complete" ⟧ | ⟦ "fail" ⟧ | ⟦ "unfold" ⟧ |
⟦ Action "and" Action ⟧ | ⟦ Action "and then" Action ⟧ |
⟦ "unfolding" Action ⟧ | ⟦ "give" Yielder "label" "#" n ⟧ |
⟦ "check" Yielder ⟧ | ⟦ Action "then" Action ⟧ |
⟦ "bind" Yielder "to" Yielder ⟧ |
⟦ "furthermore" Action ⟧ | ⟦ Action "hence" Action ⟧ |
⟦ Action "moreover" Action ⟧ | ⟦ Action "before" Action ⟧ |
⟦ "store" Yielder "in" Yielder ⟧ | ⟦ "deallocate" Yielder ⟧ |
⟦ "enact" Yielder ⟧ | ⟦ Action "else" Action ⟧ |
⟦ "recursively" "bind" Yielder "to" Yielder ⟧ |
⟦ "allocate a" Sort ⟧ |
⟦ Object-Name "." Method-Name "(" ⟨ data ⟩? ")" ⟧

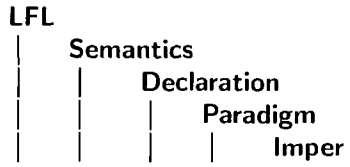
Yielder =
⟦ "the" Sort "#" n ⟧ |
⟦ "the" Sort "bound" "to" k ⟧ |
⟦ "the" Sort "stored" "in" Yielder ⟧ |
⟦ Yielder "with" Yielder ⟧ |
⟦ "closure" Yielder ⟧ |
⟦ "abstration of" Action ⟧

Sort =
⟦ "bindable" ⟧ | ⟦ "cell" ⟧ | ⟦ "cell" "[" Sort "]" ⟧ |
⟦ "storable" ⟧ | ⟦ "abstraction" ⟧ | ⟦ "datum" | "integer" | "value" ⟧ | ⟦ "truth-value" ⟧ |
⟦ Sort | Sort ⟧

ANEXO B

CLASSES DA LFL - *LANGUAGE FEATURES LIBRARY*

B.1 Declaration




```
Class Constant ≪
        Identifier,
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Declaration.Paradigm.Imper
    using E:Expression, I:Identifier
    semantics:
        elaborate-constant( I , E ) =
             evaluate ⟦ E ⟧
            then
             bind I to the value
End Class
```

```
Class Variable ≪
        Identifier,
        Type,
        Expression
            implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Declaration.Paradigm.Imper
    using E:Expression, I:Identifier, T:Type
    semantics:
        elaborate-variable( I, T, E ) =
                    evaluate ⟦ E ⟧ then give the value label#1
                and
                    allocate-for-type T then give the cell label#2
            then
                    bind I to the cell#2
                and
                    store the value #1 in the cell#2
End Class
```

```
Class VariableDec ≪  Identifier, Type ≫
    locating LFL.Semantics.Declaration.Paradigm.Imper
    using I:Identifier, T:Type
    semantics:
        elaborate-variable( I , T ) =
            allocate-for-type T then give the cell label#1
            then
            bind token I to the cell#1
End Class



Class Procedure ≪
         Identifier,
         FomalParameter
              implementing < elaborate _ :  FormalParameter → Action >,
         Command
              implementing < execute _ :  Command → Action >  ≫
    locating LFL.Semantics.Declaration.Paradigm.Imper
    using I:Identifier, FP:FormalParameter, C:Command
    semantics:
         elaborate-proc-with-par( I, FP, C ) =
              recursively bind I to
                   closure of abstraction of
                            furthermore elaborate ⟦ FP ⟧
                       hence
                            execute ⟦ C ⟧
End Class



Class Function ≪
         Identifier,
         FomalParameter
              implementing < elaborate _ :  FormalParameter → Action >,
         Expression
              implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Declaration.Paradigm.Imper
    using I:Identifier, FP:FormalParameter, E:Expression
    semantics:
         elaborate-func-with-par( I, FP, E ) =
              recursively bind I to
                  closure abstraction of
                       furthermore elaborate ⟦ FP ⟧
                  hence
                       evaluate ⟦ E ⟧ then give the value
End Class
```

LFL
Semantics
Declaration
Paradigm
OO

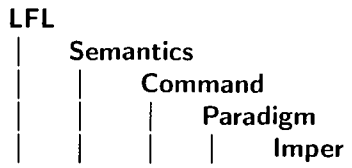
```
Class ClassDeclaration ≪
        Identifier,
        FieldDeclaration
            implementing < type bindings of _ :  FieldDeclaration → TypeBindings >,
        ConstructorDeclaration
            implementing < constructor of _ :  ConstructiorDeclaration → Yielder >,
        MethodDeclaration
            implementing < method binding of _ :  MethodDeclaration → Yielder > ≫
    locating LFL.Semantics.Declaration.Paradigm.OO
    using I1:Identifier, I2:Identifier, FD:FieldDeclaration,
        CD:ConstructorDeclaration, MD:MethodDeclaration
    semantics:
        elaborate-class-declaration( I1, I2, FD, CD, MD ) =
            recursively bind (I1, class (
                type bindings of ⟦ FD ⟧, constructor of ⟦ CD ⟧,
                method binding of ⟦ MD ⟧, the class bound to I2 ))
End Class

Class ConstructorDeclaration ≪
        Identifier,
        FormalParameters
            implementing < respectively formal bind _ :  FormalParameters → Action >,
        Argument
            implementing < respectively evaluate _ :  Argument → Action >,
        BlockStatements
            implementing < execute _ :  BlockStatements → Action > ≫
    locating LFL.Semantics.Declaration.Paradigm.OO
    using I:Identifier, FPs:FormalParameters, As:Argument, BS:BlockStatements
    semantics:
        elaborate-constructor( I, FPs, As, BS )
            closure
                furthermore
                        give field bindings of the object #1 moreover
                        bind ("this", the object #1) moreover
                        respectively formally bind ⟦ FPs ⟧ (rest (the data))
                hence
                                give the object bound to "this"and
                                respectively evaluate ⟦ As ⟧
                        then enact
                                 provide the data then
                                 constructor of superclass of the class bound to I
                    and then
                        execute ⟦ BS ⟧
                        excepsionally
                                give a return the skip otherwise raise
End Class
```

```
Class MethodDeclaration ≪  Identifier, MethodTypeDenoter,
         FormalParameters
              implementing < respectively formal bind _ :  FormalParameters → Action >,
         BlockStatements
              implementing < execute _ :  BlockStatements → Action > ≫
    locating LFL.Semantics.Declaration.Paradigm.OO
    using I:Identifier, FPs:FormalParameters, MTD:MethodTypeDenoter,
          BS:BlockStatements
    syntax:
         Dec ::= "public" MTD I "(" FPs ")" "{" BS "}"
    semantics:
         elaborate-method-declaration( MTD, I, FPs, BS ) =
              binding (I, closure
                   furthermore
                        give field bindings of the object #1 moreover
                        bind ("this", the object #1) moreover
                        repectively formally bind ⟦ FPs ⟧ (rest (the data))
                   hence
                        execute ⟦ BS ⟧
                        exceptionally
                             given a return the vie returned value of it otherwise raise
End Class
```

LFL
| **Semantics**
| | **Declaration**
| | | **Shared**

```
Class VariableSequence ≪
        VariableDeclaration
            implementing < elaborate _ :  VariableDeclaration → Action > ≫
    locating LFL.Semantics.Declaration.Shared
    using VD1:VariableDeclaration, VD2:VariableDeclaration
    semantics:
        elaborate-var-sequence( VD1, VD2) =
                elaborate ⟦ VD1 ⟧
            before
                elaborate ⟦ VD2 ⟧
End Class
```

B.2 Command




```
Class DecCommand ≪
        Declaration
             implementing < elaborate _ :  Declaration → Action >,
        Command
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Command.Paradigm.Imper
    using D:Declaration, C:Command
    semantics:
        execute-dec-command( D, C ) =
                  furthermore elaborate ⟦ D ⟧
             hence
                  execute ⟦ C ⟧
End Class

Class Assignment ≪
        Identifier,
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Command.Paradigm.Imper
    using I:Identifier, E:Expression
    semantics:
        execute-assignment( I, E ) =
                  evaluate ⟦ E ⟧
             then
                  store the value in the cell bound to I
End Class

Class Repeat ≪
        Command
             implementing < execute _ :  Command → Action >,
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Command.Paradigm.Imper
    using E:Expression, C:Command
    semantics:
        execute-repeat( E , C ) =
             unfolding
                       execute ⟦ C ⟧
                  and then
                       evaluate ⟦ E ⟧
                           then
                                    complete
                               else
                                    unfold
End Class
```

```
Class Procedure ≪
         Identifier,
         ActualParameter
              implementing < evaluate _ :  ActualParameter → Value> ≫
    locating LFL.Semantics.Command.Paradigm.Imper
    using I:Identifier, AP:ActualParameter
    semantics:
         execute-proc-without-par( I, AP ) =
                   evaluate ⟦ AP ⟧
              then
                   enact ( the procedure bound to I wih the value )
End Class
```

LFL
| Semantics
| | Command
| | | Paradigm
| | | | OO

```
Class ReturnWithoutValue
    locating LFL.Semantics.Command.Paradigm.OO
    semantics:
         execute-return(  ) =
              raise voidReturn
End Class

Class ReturnWithValue ≪
         Expression
              implementing < evaluate _ :  Expression → Action > ≫
    locating LFL.Semantics.Command.Paradigm.OO
    using E:Expression
    semantics:
         execute-return-exp( E ) =
                   evaluate ⟦ E ⟧
              then
                   raise return (the value)
End Class
```

LFL
| Semantics
| | Command
| | | Share

```
Class Sequence ≪
        Command
            implementing < execute _ :  Command → Action >  ≫
    locating LFL.Semantics.Command.Shared
    using C1:Command, C2:Command
    semantics:
        execute-sequence( C1, C2 ) =
                  execute ⟦ C1 ⟧
             and then
                  execute ⟦ C2 ⟧
End Class
```

```
Class Selection ≪
        Command
             implementing < execute _ :  Command → Action >,
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Command.Shared
    using E:Expression, C1:Command, C2:Command
    semantics:
        execute-if-then( E , C1 ) =
                  evaluate ⟦ E ⟧
             then
                  check(the given TruthValue is true) and then
                      execute ⟦ C1 ⟧
                and then
                  complete

        execute-if-then-else( E, C1, C2 ) =
                      evaluate ⟦ E ⟧
                  then
                      check(the given TruthValue is true) and then
                          execute ⟦ C1 ⟧
             or
                      check(the given TruthValue is false) and then
                          execute ⟦ C2 ⟧
                  and then
                      complete
End Class

Class While ≪
         Expression
              implementing < evaluate _ :  Expression → Action >,
         Command
              implementing < execute _ :  Command → Action > ≫
    locating LFL.Semantics.Command.Shared
    using E:Expression, C:Command
    semantics:
         execute-while ( E, C ) =
              unfolding
                  evaluate ⟦ E ⟧ then
                  infalibly select
                      given true then execute ⟦ S ⟧ then unfold
                  or
                      given false then skip
End Class
```

B.3 Expression

```
LFL
|   Semantics
|   |   Expression
|   |   |   Paradigm
|   |   |   |   Imper
```

% *Nesta versão da LFL não há classes agrupadas ao nodo acima.* %

```
LFL
|   Semantics
|   |   Expression
|   |   |   Paradigm
|   |   |   |   OO
```

```
Class PrefixOperator ≪
        Expression
            implementing < evaluate _ :  Expression → Action >,
        PrefixOp
            implementing < apply prefix _ :  PrefixOp → Action >  ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, PreOp:PrefixOp
    semantics:
        evaluate-prefix( PreOp, Exp ) =
                evaluate ⟦ Exp ⟧
            then
                apply prefix ⟦ PreOp ⟧
End Class
```

```
Class InfixOperator ≪
    Expression
        implementing < evaluate _ :  Expression → Action >,
    InfixOp
        implementing < apply infix _ :  InfixOp → Action >  ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using E1:Expression, E2:Expression, InfixOp:InfOp
    semantics:
        evaluate-infix( E1, InfOp, E2 ) =
                        evaluate ⟦ E1 ⟧
                    and
                        evaluate ⟦ E2 ⟧
                then
                    apply infix⟦ InfOp ⟧
End Class
```

```
Class LogicOperatorInstanceOf ≪
        Identifier,
        Expression
                implementing < evaluate _ :  Expression → Action > ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, I:Identifier
    semantics:
        evaluate-instance-of( E, "instance of", I ) =
                    evaluate ⟦ E ⟧
                then
                    when the reference is an instance of the class bound to I
                    then give true
                    otherwise give false
End Class



Class Attribution ≪
        Identifier,
        Expression
                implementing < evaluate _ :  Expression → Action > ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using I:Identifier, E:Expression
    semantics:
        evaluate-attribution( I, E ) =
                    evaluate ⟦ E ⟧
                then
                        store the given value in the variable bound to I
                    and
                        give the given value
End Class



Class NewObject ≪
        Identifier,
        Argument
                implementing < respectively evaluate _ :  Argument → Action > ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, I:Identifier, As:Argument
    semantics:
        evaluate-new( I, As ) =
                    allocate an object of the class bound to I and
                    repectively evaluate ⟦ As ⟧
                then
                    enact
                      provide the data then
                      constructor of the class bound to I
                    and give the object#1
End Class
```

```
Class ReferenceIdentifier ≪
        Identifier,
        Expression
                implementing < evaluate _ :  Expression → Action >,
        Argument
                implementing < respectively evaluate _ :  Argument → Action > ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, I:Identifier, As:Argument
    semantics:
        evaluate-exp-arg( E, I, As ) =
                    evaluate ⟦ E ⟧ and
                    repectively evaluate ⟦ As ⟧
                then infallibly select
                        given null #1 then raise nullReferenceException
                    or
                        given an object #1 then enact
                          provide the data then
                          select method( I, class of the object #1 )
End Class

Class ReferenceClassAbove ≪
        Identifier,
        Argument
                implementing < respectively evaluate _ :  Argument → Action > ≫
    locating LFL.Semantics.Expression.Paradigm.OO
    using I:Identifier, As:Argument
    semantics:
        evaluate-super( I, As ) =
                    give the object bound to "this"and
                    repectively evaluate ⟦ As ⟧
                then
                    enact
                      provide the data then
                      selected method(I, superclass of class of the object #1)
End Class
```

LFL
| **Semantics**
| | **Expression**
| | | **Shared**

```
Class Sum ≪
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Expression.Shared
    using E1:Expression, E2:Expression
    semantics:
        evaluate-sum( E1 , E2 ) =
                 evaluate ⟦ E1 ⟧
             and then
                 evaluate ⟦ E2 ⟧
        then
             give sum(the give integer#1, the give integer#2)
End Class

Class Subtract ≪
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Expression.Shared
    using E1:Expression, E2:Expression
    semantics:
        evaluate-sub( E1 , E2 ) =
                 evaluate ⟦ E1 ⟧
             and then
                 evaluate ⟦ E2 ⟧
        then
             give subtraction(the give integer#1, the give integer#2)
End Class

Class Product ≪
        Expression
             implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Expression.Shared
    using E1:Expression, E2:Expression
    semantics:
        evaluate-prod( E1 , E2 ) =
                 evaluate ⟦ E1 ⟧
             and then
                 evaluate ⟦ E2 ⟧
        then
             give product(the give integer#1, the give integer#2)
End Class

Class Identifier ≪  Ident  ≫
    locating LFL.Semantics.Expression.Shared
    using I:Ident
    semantics:
        evaluate-identifier( I ) =
             give the value bound to I
        or
             give the value stored in the cell bound to I
End Class
```

```
Class Numeral ≪
        Number implementing < valuation _ :  Number → Integer > ≫
    locating LFL.Semantics.Expression.Shared
    using N:Number
    semantics:
        evaluate-numeral( N ) =
            give valuation N
End Class



Class LogicOperator ≪
         Expression implementing < evaluate _ :  Expression → Action > ≫
    locating LFL.Semantics.Expression.Shared
    using E1:Expression, E2:Expression
    semantics:
         evaluate-or( E1, E2 ) =
              evaluate ⟦ E1 ⟧ then
                          check (the given value is true)
                     then
                          give true
                or
                          check (the given value is false)
                     then
                          evaluate ⟦ E2 ⟧

         evaluate-and( E1, E2 ) =
              evaluate ⟦ E1 ⟧ then
                          check (the given value is true)
                     then
                          evaluate ⟦ E2 ⟧
                or
                          check (the given value is false)
                     then
                          give false
End Class



Class True
    locating LFL.Semantics.Expression.Shared
    semantics:
         evaluate-true( ) =
              give true
End Class



Class False
    locating LFL.Semantics.Expression.Shered
    semantics:
         evaluate-false( ) =
              give false
End Class
```

```
Class LessThan ≪
        Expression
            implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Expression.Shered
    using E1:Expression, E2:Expression
    semantics:
        evaluate-less-than( E1 , E2 ) =
                evaluate ⟦ E1 ⟧
            and
                evaluate ⟦ E2 ⟧
        then
            give less(the give integer#1, the give integer#2)
End Class

Class Equality ≪
        Expression
            implementing < evaluate _ :  Expression → Action >  ≫
    locating LFL.Semantics.Expression.Shered
    using E1:Expression, E2:Expression
    semantics:
        evaluate-equality( E1 , E2 ) =
                evaluate ⟦ E1 ⟧
            and
                evaluate ⟦ E2 ⟧
        then
            give equal(the give integer#1, the give integer#2)
End Class

Class Function ≪
        Identifier,
        ActualParameter
            implementing < evaluate _ :  ActualParameter → Action > ≫
    locating LFL.Semantics.Expression.Shered
    using I:Identifier, AP:ActualParameter
    semantics:
        evaluate-func-without-par( I, AP ) =
                evaluate ⟦ AP ⟧
            then
                enact (the function bound to I with the value)
End Class
```

ANEXO C

ESPECIFICAÇÃO DA LINGUAGEM μPASCAL

C.1 Classes Auxiliares

```
Class Identificador
    syntax:
        uId ::= letter[ letter | digit ]*
End Class
```

```
Class Algarismo
    syntax:
        uA ::= digit+
    semantics:
        avaliação _ :  uN → integer
End Class
```

```
Class Tipo
    syntax:
        uT ::= "boolean" | "integer"
    semantics:
        allocate-for-type _ :  uT → Action

        allocate-for-type[[ boolean ]]  =
            allocate a cell[ truth-value ]

        allocate-for-type[[ integer ]]  =
            allocate a cell[ integer ]
End Class
```

C.2 Declarações

```
Class Declaração
    syntax:
        uDec
    semantics:
        elaborar _ :  uDec → Action
End Class
```

```
Class Constante
    extending Declaração
    including LFL.Semantics.Declaration.Paradigm.Imper
    using uE:Expressão, uT:Tipo, uI:Identificador,
          objCst:Constant ≪ Identificador, Expressão<avaliar> ≫
    syntax:
          uDec ::= "const" uI ":" uT "=" uE
    semantics:
          elaborar[[ "const" uI ":" uT "=" uE ]]  =
               objCst.elaborate-constant( uI, uE )
End Class



Class Variavel
    extending Declaração
    including LFL.Semantics.Declaration.Paradigm.Imper
    using uE:Expressão, uT:Tipo, uI:Identificador,
          objVar:Variable ≪ Identificador, Tipo, Expressão<avaliar> ≫
          objVdc:VariableDec ≪ Identificador, Tipo ≫
    syntax:
          uDec ::= "var" uI ":" uT [ "=" uE ]
    semantics:
          elaborar[[ "var" uI ":" uT ]]  =
               objVdc.elaborate-variable( uI, uT )

          elaborar[[ "var" uI ":" uT "=" uE ]]  =
               objVar.elaborate-variable( uI, uT, uE )
End Class



Class FormalPar
    extending Declaração
    including LFL.Semantics.Declaration.Paradigm.Imper
    using uI:Identificador, uT:Tipo,
          objVar:Variable ≪ Identificador, Tipo, Expressão<avaliar> ≫
    syntax:
          uFP ::= "var" uI ":" uT
    semantics:
          elaborarFP _ :  uFP → Action

          elaborarFP[[ "var" uI ":" uT ]]  =
               objVar.elaborate-variable( uI, uT )
End Class

Class ProcedimentoDec
    extending Declaração
    including LFL.Semantics.Declaration.Paradigm.Imper
    using uI:Identificador, uC:Comando, uFP:FormalPar,
          objPrd:Procedure ≪ Identificador, FormalPar<elaborarFP>, Comando<executar> ≫
    syntax:
          uDec ::= "proc" uI "(" uFP ")" ":" uC
    semantics:
          elaborar[[ "proc" uI "(" uFP ")" ":" uC ]]  =
               objPrd.elaborate-proc-with-par( uI, uFP, uC )
End Class
```

```
Class FunçãoDec
    extending Declaração
    including LFL.Semantics.Declaration.Paradigm.Imper
    using uI:Identificador, uFP:FormalPar, uE:Expressão,
          objFnd:Function ≪ Identificador, FormalPar<elaborarFP>,
                            Expressão<avaliar> ≫
    syntax:
          uDec ::= "func" uI "(" uFP ")" ":" uE
    semantics:
          executar[[ "proc" uI "(" uFP ")" ":" uC ]]  =
               objFnd.elaborate-func-with-par( uI, uFP, uE )
End Class
```

C.3 Comandos

```
Class Comando
    syntax:
         uCom
    semantics:
         executar _ :  uCom → Action
End Class
```

```
Class DeclareComando
    extending Comando
    including LFL.Semantics.Command.Paradigm.Imper
    using uD:Declaração, uC:Comando,
          objDec:DecCommand ≪ Declaração<elaborar>, Comando<executar> ≫
    syntax:
          uCom ::= "declare" uD "use-em" uC
    semantics:
          executar[[ "declare" uD "use-em" uC ]]  =
               objDec.execute-dec-command( uD, uC )
End Class
```

```
Class Atribuição
    extending Comando
    using uI:Identificador, uE:Expressão,
          objAtr:LFL.Semantics.Command.Paradigm.Imper.Assignment ≪
               Identificador, Expressão<avaliar> ≫
    syntax:
          uCom ::= uI ":=" uE
    semantics:
          executar[[ uI ":=" uE ]]  =
               objAtr.execute-assignment( uI, uE )
End Class
```

```
Class Seleção
    extending Comando
    using uC1:Comando, uC2:Comando, uE:Expressão,
          objSel:LFL.Semantics.Command.Shared.Selection ≪
               Comando<executar>, Expressão<avaliar> ≫
    syntax:
          uCom ::= "se" uE "então" uC1 [ "senão" uC2 ] "fim-se"
    semantics:
          executar[[ "se" uE "então" uC1 "fim-se" ]]  =
               objSel.execute-if-then( uE, uC1 )

          executar[[ "se" uE "então" uC1 "senão" uC2 "fim-se" ]]  =
               objSel.execute-if-then-else( uE, uC1, uC2 )
End Class



Class Repetição
    extending Comando
    using uC:Comando, uE:Expressão,
          objRep:LFL.Semantics.Command.Paradigm.Imper.Repeat ≪
               Comando<executar>, Expressão<avaliar> ≫
    syntax:
          uCom ::= "repita" uC "ate" uE
    semantics:
          executar[[ "repita" uC "ate" uE ]]  =
               objRep.execute-repeat( uC, uE )
End Class



Class SequenciaComando
    extending Comando
    using uC1:Comando, uC2:Comando,
          objCmd:LFL.Semantics.Command.Shared.Sequence ≪ Comando<executar> ≫
    syntax:
          uCom ::= "sequencie" uC1 ";" uC2
    semantics:
          executar[[ "sequencie" uC1 ";" uC2 ]]  =
               objCmd.execute-sequence( uC1, uC2 )
End Class



Class ProcedimentoCom
    extending Comando
    including LFL.Semantics.Command.Paradigm.Imper
    using uI:Identificador, uAP:AtualPar,
          objPrc:Procedure ≪ Identificador, AtualPar<avaliarAP> ≫
    syntax:
          uCom ::= "chamar" uI "(" uAP ")"
    semantics:
          executar[[ "chamar" uI "(" uAP ")" ]]  =
               objPrc.execute-proc-without-par( uI, uAP )
End Class
```

C.4 Expressões

```
Class Expressão
    syntax:
         uExp
    semantics:
         avaliar _ :  uExp → Action
End Class


Class ValorVerdade
    extending Expressão
    using objVvd:LFL.Semantics.Expression.Shared.True
    syntax:
          uExp ::= "true"
    semantics:
          avaliar[[ "true" ]]  =
               objVvd.evaluate-true( )
End Class


Class ValorFalso
    extending Expressão
    using objVfs:LFL.Semantics.Expression.Shared.False
    syntax:
          uExp ::= "false"
    semantics:
          avaliar[[ "false" ]]  =
               objVfs.evaluate-false( )
End Class


Class AlgarismoExp
    extending Expressão
    using A:Algarismo,
          objNum:LFL.Semantics.Expression.Shared.Numeral ≪ Algarismo<avaliação> ≫
    syntax:
          uExp ::= A
    semantics:
          avaliar[[ A ]]  =
               objNum.evaluate-numeral( A )
End Class


Class IdentificadorExp
    extending Expressão
    using Id:Identificador,
          objId:LFL.Semantics.Expression.Shared.Identifier ≪ Identificador ≫
    syntax:
          uExp ::= Id
    semantics:
          avaliacao[[ Id ]]  =
               objId.evaluate-identifier( Id )
End Class
```

```
Class Soma
    extending Expressão
    using uE1:Expressão, uE2:Expressão,
          objSum:LFL.Semantics.Expression.Shared.Sum ≪ Expressão<avaliar> ≫
    syntax:
          uExp ::= uE1 " + " uE2
    semantics:
          avaliar[[ uE1 "+" uE2 ]]  =
               objSum.evaluate-sum( uE1, uE2 )
End Class


Class Subtração
    extending Expressão
    using uE1:Expressão, uE2:Expressão,
          objSub:LFL.Semantics.Expression.Shared.Subtract ≪ Expressão<avaliar> ≫
    syntax:
          uExp ::= uE1 " - " uE2
    semantics:
          avaliar[[ uE1 " - " uE2 ]]  =
               objSub.evaluate-sub( uE1, uE2 )
End Class


Class Multiplicação
    extending Expressão
    using uE1:Expressão, uE2:Expressão,
          objPrd:LFL.Semantics.Expression.Shared.Product ≪ Expressão<avaliar> ≫
    syntax:
          uExp ::= uE1 " * " uE2
    semantics:
          avaliar[[ uE1 " * " uE2 ]]  =
               objPrd.evaluate-prod( uE1, uE2 )
End Class


Class VerificaMaiorExpressão
    using uE1:Expressão, uE2:Expressão
    semantics:
         avalie-maior-que( uE1, uE2 ) =
                    avaliar ⟦ E1 ⟧ then give the value label#1
               and
                    avaliar ⟦ E2 ⟧ then give the value label#2
            then
               give greater than(the give integer#1, the give integer#2)
End Class


Class SimboloMaiorQue
    extending Expressão
    using uE1:Expressão, uE2:Expressão, objVme:VerificaMaiorExpressão
    syntax:
         uExp ::= uE1 ">"  uE2
    semantics:
         avaliar[[ uE1 ">" uE2 ]]  =
              objVme.avalie-maior-que( uE1, uE2 )
End Class
```

```
Class PalavraMaiorQue
    extending Expressão
    using uE1:Expressão, uE2:Expressão, objVme:VerificaMaiorExpressão
    syntax:
         uExp ::= uE1 "maior que" uE2
    semantics:
         avaliar[[ uE1 "maior que" uE2 ]]  =
              objVme.avalie-maior-que( uE1, uE2 )
End Class


Class Igualdade
    extending expressão
    using uE1:Expressão, uE2:Expressão,
          objEqy:LFL.Semantics.Expression.Shared.Equality ≪ Expressão<avaliar> ≫
    syntax:
          uExp ::= uE1 " = " uE2
    semantics:
          avaliar[[ uE1 " = " uE2 ]]  =
               objEqy.evaluate-equality( uE1, uE2 )
End Class


Class AtualPar
    extending Expressão
    using uE:Expressão
    syntax:
          uAP ::= uE
    semantics:
          avaliarAP _ :  uAP → Value

          avaliarAP[[ uE ]]  =
               avaliar ⟦ uE ⟧
End Class


Class FunçãoExp
    extending Expressão
    including LFL.Semantics.Expression.Paradigm.Imper
    using uI:Identificador, uAP:AtualPar,
          objFnc:Procedure ≪ Identificador, AtualPar<avaliarAP> ≫
    syntax:
          uExp ::= uI "(" uAP ")"
    semantics:
          avaliar[[ uI "(" uAP ")" ]]  =
               objFnc.evaluate-func-without-par( uI, uAP )
End Class
```

C.4.1 Representação da Linguagem μPascal

```
Class Linguagem$\mu$Pascal
    using uD:Declaração, uC:Comando
    syntax:
          uPrg ::= "declare" uD "use-em" uC
    semantics:
          execute _ :  uPrg → Action

          execute[[ "declare" uD "use-em" uC ]]  =
                    elaborar ⟦ D ⟧
               hence
                    executar ⟦ C ⟧
End Class
```

ANEXO D

Semântica de Ações Orientada a Objetos da Linguagem JOOS

Neste anexo é utilizada a Semântica de Ações Orientada a Objetos para descrever formalmente a linguagem JOOS [18]. Primeiramente é definida a sintaxe abastrata da linguagem, posteriormente define-se as entidades semânticas e por fim, as funções semânticas da linguagem são definidas. Na linguagem JOOS, apresentada neste anexo, são utilizadas algumas classes que estão agrupadas na LFL. A descrição completa das classes da LFL está no anexo B.

D.1 Abstract Syntax

Declaration

ClassDeclaration ::=
⟦ "public" "final"? "class" Identifier "extends" Identifier
"{" FieldDeclaration* ConstructorDeclaration MethodDeclaration "}" ⟧.

FieldDeclaration ::=
⟦ "private" TypeDenoter Identifier ⟧.

ConstructorDeclaration ::=
⟦ "public" MethodTypeDenoter Identifier "(" FormalParameters ")".
"{" "super" "(" Argument ")" ";" BolckStatement "}" ⟧.

MethodDeclaration ::=
⟦ "public" MethodTypeDenoter Identifier
"(" FormalParameters ")" "{" BolckStatement "}" ⟧.

FormalParameters ::=
⟨ FormalParameter ⟨ "," FormalParameter ⟩* ⟩?.

FormalParameter ::=
⟦ TypeDenoter | Identifier ⟧.

VariableDeclaration ::=
⟦ TypeDenoter | Identifier ";" ⟧.

TypeDenoter ::=
⟦ "int" | "boolean" | Identifier ⟧.

MethodTypeDenoter ::=
⟦ "void" | TypeDenoter ⟧.

Statement

Statement ::=
⟦ ";" ⟧ | ⟦ "{" BlockStatements "}" ⟧ |
⟦ Program ⟧ | ⟦ StatementExpression ";" ⟧ |
⟦ "return" ";" ⟧ | ⟦ "return" Expression ";" ⟧ |
⟦ "if" "(" Expression ")" Statement "else" Statement ⟧ |
⟦ "while" "(" Expression ")" Statement ⟧.

BlockStatements ::=
VariableDeclaration* Statement*.

Program =
⟦ "declare" Declaration "used-in" Statement ⟧.

Expressions

Expression ::=
Literal | Identifier | "this" |
⟦ PrefixOperator Expression ⟧ |
⟦ Expression InfixOperator Expression ⟧ |
⟦ Expression ("||" | "&&") Expression ⟧ |
⟦ Expression "instanceof" Identifier ⟧ |
⟦ "(" TypeDenoter ")" Expression ⟧ |
⟦ "(" Expression ")" ⟧ | StatementExpression.

StatementExpression ::=
⟦ Identifier "=" Expression ⟧ |
⟦ "new" Identifier "(" Argument ")" ⟧ |
⟦ Exp"."Identifier "(" Argument ")" ⟧ |
⟦ "super."Identifier "(" Argument ")" ⟧.

Argument :=
⟨ Expression ⟨ "," Expression ⟩* ⟩?.

PrefixOperator ::=
"-" | "!".

InfixOperator ::=
"!=" | "<" | ">" | "<=" | ">=" | "=="
"+" | "-" | "*" | "/" | "%".

Literal ::=
BooleanLiteral | IntegerLiteral | "null".

BooleanLiteral ::=
"true" | "false".

IntegerLiteral ::=
⟦ digit+ ⟧.

Identifier ::=
⟦ letter "(" letter | digit+ ")" ⟧.

D.2 Semantic Entities

vars
A, A_1, A_2 : Action;
c : Class;
o, o_1, o_2 : Object;
tk : Token;
v : Value;
Y : Yielder

Actions

- respectively _ : Action
- respectively A =
  unfolding
  infilibly select
  given ( )
  or
  given a datum$^+$ then
  A (#1) and
  unfold (rest)

Data

- Datum ::= **sorts** Value, Variable, Type, Class, Method, Constructor, TypeBindings, VariableBindings, MethodBindings, Token, Reason
- Token ::= **sort** Identifier
- Bindable ::= **sorts** Class, Object, Variable
- Storable ::= **sort** Value | nothing

Values

- Value ::= **sorts** Boolean, Integer, Reference

Variables

- Variable ::= **sort** Cell
- allocate a variable : Action % { *given a value, giving a variable, storing*} %
- allocate a variable = create (the value)

Types

- Type ::= booleanType | integerType | referenceType
- default value of _ : Type → Value
- default value of booleanType = false;
- default value of integerType = 0;
- default value of referenceType = null

Classes

- Class ::= class(field type bindings of _ : Type Bindings;
  constructor of _ : Constructor;
  method bindings of _ : MethodBindings;
  superclass of _ : Opt[Class])

- Constructor $\leq$ Action
- Method $\leq$ Action
- TypeBindings ::= sort Map[Token, Type]
- MethodBindings ::= sort Map[Token, Method]

- inherited field type bindings of _ : Class $\rightarrow$ TypeBindings
- inherited field type bindings of c =
  else disjoint union (inherited field type bindings of superclass of c,
  field type bindings of c when superclass of c = ()
  field type bindings of c)

- selected method : Token $\times$ Class $\rightarrow$? Method
- selected method (tk, c) =
  bound (inherited method bindings of c, tk)

- superclasses of _ : Class $\rightarrow$ Set[Class]
- superclasses of c =
  set(c) when superclass of c = ( )
  else union (set (c), superclasses of superclass of c)

Objetcts and References

- Object ::= object (class of : Class;
  field bindings of : VariableBindings;
  identity of : Cell)
- Reference ::= null | **sort** Object
- VariableBindings ::= **sort** Map[Token,Variable]

- _ is identical to _ : Reference $\times$ Reference
- null is identical to null;
  o_1 is identical to $o_2 \Leftrightarrow$ identity of o_1 = identity of o_2;
  $\neg$ (null is identical to o);
  $\neg$ (o is identical to null)

- _ is an instance of _ : Reference $\times$ Class
- o is an instance of $c \Leftrightarrow$ c is in superclasses of class of o;
  $\neg$ (null is an instance of c)

- allocate an object : Action
- allocate an object =
  give the class and
  instantiate inherited field type bindings of the class and
  create (nothing)
  then give object (the class#1, the variableBindings#2, the cell#3)

- instantiate : Action
- instantiate =
  give tuple (graph (the typeBindings)) then
  respectively
  give the token (first (the pair)) and
  give default value of the type (second (the pair)) then
  allocate a variable
  then give binding (the token #1, the variable #2)
  then give disjoint union (the variableBindings*)

Reasons, Returns and Exceptions

- Reason ::= **sorts** Return, Exception
- Return ::= voidReturn | return (Value)
- Exception ::= nullReferenceException

- returned value of _ : Return → Opt[Value]
- returned value of voidReturn = ();
  returned value of return $(v) = v$

D.3 Semantic Functions

Declaration

```
Class Declaration
    syntax:
         Dec
    semantics:
         elaborate _ :  Dec → Action
         type bindings of _ :  FieldDeclaration → TypeBindings
         type denoted by _ :  TypeDenoterDec → Type

         elaborate ⟦   ⟧ =
              skip
End Class
```

```
Class ClassDeclaration
    extending Declaration
    including LFL.Semantics.Declaration.Paradigm.OO
    using I1:Identifier, I2:Identifier, FD:FieldDeclaration,
          CD:ConstructorDeclaration, MD:MethodDeclaration,
          objCls: ClassDeclaration ≪
                     Identifier,
                     FieldDeclaration<type bindings of>,
                     ConstructorDeclaration<constructor of>,
                     MethodDeclaration<method binding of> ≫
    syntax:
          Dec ::= "public" "final"? "class" I1 "extends" I2 "{" FD* CD MD* "}"
    semantics:
          elaborate ⟦ "public" "final"? "class" I1 "extends" I2 "{" FD* CD MD* "}"  ⟧ =
                 objCls.elaborate-class-declaration( I1, I2, FD*, CD, MD* ) =
End Class
```

```
Class ConstructorDeclaration
     extending Declaration
     including LFL.Semantics.Declaration.Paradigm.OO.ConstructorDeclaration
     using I:Identifier, FPs:FormalParameters, As:Argument, BS:BlockStatements,
           objCdc:ClassDeclaration ≪
                     Identifier,
                     FormalParameters<respectively formal bind>,
                     Argument<respectively evaluate>,
                     BlockStatements<execute> ≫
     syntax:
          ConstructorDec ::= "public" I "(" FPs ")" "{" "super" "(" As ")" ";" BS "}"
     semantics:
          constructor of _ :  ConstructorDec → Action

          elaborate ⟦ "public" I "(" FPs ")" "{" "super" "(" As ")" ";" BS "}" ⟧ =
                 objCdc.elaborate-constructor( I, FPs, As, BS )
End Class
```

```
Class MethodDeclaration
    extending Declaration
    including objMth.LFL.Semantics.Declaration.Paradigm.OO.MethodDeclaration
    using I:Identifier, FPs:FormalParameters,
          MTD:MethodTypeDenoter, BS:BlockStatements,
          objMth:MethodDeclaration ≪ Identifier, MethodTypeDenoter,
                      FormalParameters<respectively formal bind>,
                      BlockStatements<execute> ≫
    syntax:
          MethodDec ::= "public" MTD I "(" FPs ")" "{" BS "}"  |  MethodDec, MethodDec*
    semantics:
          method bindings of _ :  MethodDec* → Yielder

          elaborate method bindings of ⟦    ⟧ =
                 no bindings

          elaborate method bindings of ⟦ MethodDec, MethodDec* ⟧ =
                 disjoint union (method bindings of ⟦ MethodDec ⟧,
                      method bindings of ⟦ MethodDec* ⟧)

          method bindings of ⟦ "public" MTD I "(" FPs ")" "{" BS "}" ⟧ =
                 objMth.elaborate-method-declaration( MTD, I, FPs, BS )
End Class

Class RespectivelyFormally
    extending Declaration
    using FP:FormalParameter, FPs:FormalParamaters
    syntax:
          Dec ::= FP  |  FP, FPs
    semantics:
          respectively formally bind ⟦    ⟧ =
               skip

          respectively formally bind ⟦ FP ⟧ =
               formally bind ⟦ FP ⟧

          respectively formally bind ⟦ FP FPs ⟧ =
               formally bind ⟦ FP ⟧ (the value #1) and
               respectively formally bind ⟦ FPs ⟧ (rest(the data))
End Class

Class TypeBinding
    extending Declaration
    using FD:FieldDeclaration
    syntax:
          FieldDec ::= FD, FD*
    semantics:
          type bindings of ⟦    ⟧ =
               no bindings

          type bindings of ⟦ FD, FD* ⟧ =
          disjoint union (type bindings of ⟦ FD ⟧,
               type bindings of ⟦ FD* ⟧)
End Class
```

```
Class TypeDenoter
    extending Declaration
    using I:Identifier
    syntax:
        TypeDenoterDec ::= "boolean" | "int" | I
    semantics:
        type denoted by ⟦ "boolean" ⟧ =
            booleanType

        type denoted by ⟦ "int" ⟧ =
            integerType

        type denoted by ⟦ I ⟧ =
            referenceType
End Class

Class FormalParameter
    extending Declaration
    using TD:TypeDenoter, I:Identifier
    syntax:
        FormalParameterDec ::= TD I
    semantics:
        formally bind _ :  FormalParameterDec → Action

        formally bind ⟦ TD I ⟧ =
            give the value then
            allocate a variable then
            bind I to the variable
End Class

Class FormalParameters
    extending Declaration
    using FP:FormalParameter
    syntax:
        FormalParametersDec ::= ⟨ FP ⟨ "," FP ⟩* ⟩?
    semantics:
        respectively formally bind _ :  FormalParametersDec → Action
End Class

Class VariableDeclaration
    extending Declaration
    using TD:TypeDenoter, I:Identifier
    syntax:
        Dec ::= TD I ";"
    semantics:
        elaborate _ :  VariableDec* → Action

        elaborate ⟦ TD I ";" ⟧ =
            give default value of type denoted by ⟦ TD ⟧ then
            allocate a variable then
            bind I to the variable
End Class
```

```
Class FieldDeclaration
    extending Declaration
    using TD:TypeDenoter, I:Identifier
    syntax:
         Dec ::= "private" TD I ";"
    semantics:
         type bindings of ⟦ "private" TD I ";" ⟧ =
              binding (I, type denoter by ⟦ TD ⟧)
End Class

Class MethodTypeDenoter
    extending Declaration
    using TD:TypeDenoter
    syntax:
         Dec ::= "void"  |  TD
End Class

 Class VariableSequence
      extending Declaration
      using VD:VariableDeclaration,
            objSeq:LFL.Semantics.Command.Shared.VariableSequence ≪
                    VariableDeclaration<elaborate> ≫
      syntax:
            Dec ::= VD VD*
      semantics:
            elaborate ⟦ VD VD* ⟧ =
                    objSeq.elaborate-var-sequence( VD, VD* ) =
 End Class

Class DecStatement
    extending Statement
    including LFL.Semantics.Command.Paradigm.Imper
    using D:Declaration, S:Statement,
          objDec:DecCommand ≪ Declaration<elaborate>, Statement<execute> ≫
    syntax:
          Stmt ::= "declare" D "used-in" S
    semantics:
          executar[[ "declare" D "used-in" S ]]  =
                  objDec.execute-dec-command( D, S )
End Class
```

Statement

```
Class Statement
    syntax:
        Stmt
    semantics:
        execute _ :  Stmt* → Action

        execute ⟦   ⟧ =
            skip

        execute ⟦ ";" ⟧ =
            skip
End Class

Class SimpleStatement
    extending Statement
    using StmtExpE:StatementExpression
    syntax:
        Stmt ::= StmtExp ";"
    semantics:
        execute ⟦ StmtExp ";" ⟧ =
                evaluate ⟦ StmtExp ⟧
            then
                skip
End Class

Class BlockStatements
    extending Statement
    using VariableDec:VariableDeclaration, Stmt:Statement
    syntax:
        BlockStmt ::= VariableDec* Stmt*
    semantics:
        execute _ :  BlockStmt → Action

        execute ⟦ "{" BlockStmt "}" ⟧ =
            execute ⟦ BlockStmt ⟧
End Class

Class Return
    extending Statement
    including LFL.Semantics.Command.Paradigm.OO
    using E:Expression, objNvr:ReturnWithoutValue,
          objRet:ReturnWithValue ≪ Expression<evaluate> ≫
    syntax:
          Stmt ::= "return" ";"  |  "return" E ";"
    semantics:
          execute ⟦ "return" ";" ⟧ =
              objNvr.execute-return(  )

          execute ⟦ "return" E ";" ⟧ =
              objRet.execute-return-exp( E )
End Class
```

```
Class Selection
    extending Statement
    using E:Expression, S1:Statement, S2:Statement,
          objSel:LFL.Semantics.Command.Shared.Selection ≪
          Statement<execute>, Expression<evaluate> ≫
    syntax:
          Stmt ::= "if" "(" E ")" S1 "else" S2
    semantics:
          execute ⟦ "if" "(" E ")" S1 "else" S2 ⟧ =
               objSel.execute-if-then-else( E, S1, S2 )
End Class

Class While
    extending Statement
    using E:Expression, S:Statement,
    objRep:LFL.Semantics.Command.Shared.While ≪
          Expression<evaluate>, Statement<execute> ≫
    syntax:
          Stmt ::= "while" "(" E ")" S
    semantics:
          execute ⟦ "while" "(" E ")" S ⟧ =
               objRep.execute-while( E, S )
End Class

Class Sequence
    extending Statement
    using S1:Statement, S2:Statement,
          objSeq:LFL.Semantics.Command.Shared.Sequence ≪ Statement<execute> ≫
    syntax:
          Stmt ::= S1 ";" S2 *
    semantics:
          execute ⟦ S1 ";" S2* ⟧ =
               objSeq.execute-sequence( S1, S2* )
End Class

Class Variable
    extending Statement
    using VD:VariableDeclaration, S:Statement
    syntax:
         Stmt ::= VD* S*
    semantics:
         execute ⟦ VD* S* ⟧ =
                   furthermore elaborate ⟦ VD* ⟧
              hence
                   execute ⟦ S* ⟧
End Class

Class DecStatement
    extending Statement
    including LFL.Semantics.Command.Paradigm.Imper
    usingD:Declaration, S:Statement,
         objDec:DecCommand ≪ Declaration<elaborate>, Statement<execute> ≫
    syntax:
         Stmt ::= "declare" D "used-in" S
    semantics:
         executar[[ "declare" D "used-in" S ]]  =
              objDec.execute-dec-command( D, S )
End Class
```

Expression

```
Class Expression
    syntax:
         Exp
    semantics:
         evaluate _ :  Exp → Action

         evaluate ⟦ "(" Exp ")" ⟧ =
              evaluate ⟦ Exp ⟧
End Class

Class BooleanLiteral
    extending Expression
    syntax:
         Exp ::= "true" | "false"

         value of ⟦ "true" ⟧ =
              "true"

         value of ⟦ "false" ⟧ =
              "false"
End Class

Class IntegerLiteral
    extending Expression
    syntax:
         Exp ::= Digit+
End Class

Class Identifier
    extending Expression
    using objId:LFL.Semantics.Expression.Shared.Identifier ≪ Identifier ≫
    syntax:
         Id ::= Letter ⟨ Letter | Digit ⟩*
    semantics:
         evaluate ⟦ Id ⟧ =
              objId.evaluate-identifier( Id )
End Class

Class Literal
    extending Expression
    using BooleanLit:BooleanLiteral, integerLit:IntegerLiteral
    syntax:
         Lit ::= BooleanLit | IntegerLit | "null"
    semantics:
         value of _ :  Lit → Value

         evaluate ⟦ Lit ⟧ =
              give value of ⟦ Lit ⟧

         value of ⟦ "null" ⟧ =
              "null"
End Class
```

```
Class Argument
    extending Expression
    using Exp:Expression
    syntax:
         Args := ⟨ Exp ⟨ Exp ⟩* ⟩?
    semantics:
         respectively evaluate _ : Args → Action
End Class
```

```
Class PrefixOperator
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using Exp:Expression,
          objPfxOp:PrefixOperator ≪ Expression<evaluate>,
                                    PrefixOperator<apply prefix> ≫
    syntax:
          PreOp := "-" | "!"
    semantics:
          apply prefix _ : PreOp → Action

          apply prefix ⟦ "!" ⟧ =
               give not (the boolean)

          evaluate ⟦ PreOp, Exp ⟧ =
               objPfxOp.evaluate-prefix( PreOp, Exp )
End Class
```

```
Class InfixOperator
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using E1:Expression, E2:Expression,
          objIfxOp:InfixOperator ≪ Expression<evaluate>,
                                   InfixOperator<apply prefix> ≫
    syntax:
          InfOp := "!=" | "<" | ">" | "<=" | ">=" | "=="
                   "+" | "-" | "*" | "/" | "%"
    semantics:
          apply infix _ : InfOp → Action

          evaluate ⟦ E1 InfOp E2 ⟧ =
                   objIfxOp.evaluate-infix( E1, InfOp, E2 ) =

          apply infix ⟦ "==" ⟧ =
                   select
                       when (the boolean #1 = the boolean #2 ) or
                       when (the integer #1 = the integer #2 ) or
                       when (the reference #1 is identical to the reference #2 )
                   then give true
                   otherwise give false

          apply infix ⟦ "+" ⟧ =
                   give (the integer #1 + the integer #2 )
```

```
        %(Os outros operadores prefixo são similares)%
End Class



Class LogicOperator
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO.LogicOperatorInstanceOf
    including LFL.Semantics.Expression.Shared.LogicOperator
    using E1:Expression, E2:Expression, E:Expression, I:Identifier,
          objLogic:LogicOperator ≪ Expression<evaluate> ≫,
          objInstance:LogicOperatorInstanceOf ≪ Identifier, Expression<evaluate> ≫
    syntax:
          Exp ::= E1 "||" E2  |  E1 "&&" E2  |  E "instance of" I
    semantics:
          evaluate ⟦  E1 "||" E2  ⟧ =
               objLogic.evaluate-or( E1, "||", E2 )

          evaluate ⟦  E1 "&&" E2  ⟧ =
               objLogic.evaluate-and( E1, "&&", E2 )

          evaluate ⟦  E "instance of" I  ⟧ =
               objInstance.evaluate-instance-of( E, "instance of", I )
End Class



Class StatementExpression
    extending Expression
    including LFL.Semantics.Expression.Paradigm.OO
    using E:Expression, I:Identifier, As:Argument,
          objAtt:Attribution ≪ Identifier, Expression<evaluate> ≫,
          objNob:NewObject ≪ Identifier, Argument<respectively evaluate> ≫,
          objRid:ReferenceIdentifier ≪ Identifier, Expression<evaluate>,
                                       Argument<respectively evaluate> ≫,
          objRcl:ReferenceClassAbove ≪ Identifier, Argument<respectively evaluate> ≫
    syntax:
          Exp ::= I "=" E  |
                  "new" I "(" As ")"  |
                  E "." I "(" As ")"  |
                  "super" "." I "(" A ")"
    semantics:
          evaluate ⟦  I "=" E  ⟧ =
                  objAtt.evaluate-attribution( I, E )

          evaluate ⟦ "new" I "(" As ")" ⟧ =
                  objNob.evaluate-new( I, As )

          evaluate ⟦ E "." I "(" As ")" ⟧ =
                  objRid.evaluate-exp-arg( E, I, As )

          evaluate ⟦  "super" "." I "(" As ")"  ⟧ =
                  objRcl.evaluate-super( I, As )
End Class
```

```
Class ReferenceThis
    extending Expression
    syntax:
        Exp ::= "This"
    semantics:
        evaluate ⟦ Exp ⟧ =
            give the object bound to ⟦ Exp ⟧
End Class

Class Respectively
    extending Expression
    using E:Expression, As:Argument
    syntax:
        Exp ::= E | E, As
    semantics:
        respectively evaluate ⟦   ⟧ =
            give ( )

        respectively evaluate ⟦ E ⟧ =
            evaluate ⟦ E ⟧

        respectively evaluate ⟦ E, As ⟧ =
                evaluate ⟦ E ⟧
            and
                respectively evaluate ⟦ As ⟧
End Class

 Class TypeDenoterExp
     extending Expression
     using TD:TypeDenoter, E:Expression
     syntax:
         Exp ::= "(" TD ")" E
     semantics:
         evaluate ⟦ "(" TD ")" E ⟧ =
             evaluate ⟦ Exp ⟧
 End Class

 Class AssignmentExp
     extending Expression
     including LFL.Semantics.Expression.Paradigm.OO
     usingI:Identifier, E:Expression,
         objAtt:.Attribution ≪ Identifier, Expression<evaluate> ≫
     syntax:
         Exp ::= I "=" E
     semantics:
         evaluate ⟦ I "=" E ⟧ =
             objAtt.evaluate-attribution( I, E ) =
 End Class
```