Universidade Federal de Pernambuco – UFPE
Centro de Informática – CIn
Mestrado em Ciência da Computação

Victor Travassos Sarinho

Uma Biblioteca de Componentes Semânticos para Especificação de Linguagens de Programação

Dissertação apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Hermano Perrelli de Moura

Recife, março de 2003

Universidade Federal de Pernambuco

Centro de Informática

UMA BIBLIOTECA DE COMPONENTES SEMÂNTICOS PARA ESPECIFICAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO

Hermano Perrelli de Moura

Jorge Henrique Cabral Fernandes

André Luis de Medeiros Santos

Universidade Federal de Pernambuco

Av. Professor Luis Freire s/n

Cidade Universitária

50740-540 Recife - PE - Brasil

Tel: +55 81 3271.8430

Fax: +55 81 3271.8438

Título da dissertação:

Uma Biblioteca de Componentes Semânticos para Especificação de Linguagens de Programação

Mestrando: Victor Sarinho vts@cin.ufpe.br

Orientador: Hermano Moura hermano@cin.ufpe.br

RESUMO

Semântica de ações, um formalismo para especificação de linguagens de programação, define um conjunto padrão de operadores que descrevem conceitos comuns encontrados em linguagens de programação. Estes operadores facilitam a especificação de linguagens de programação porque eles liberam o projetista de linguagens da manipulação de definições complexas usadas para descrevê-las.

Entretanto, as especificações em semântica de ações não foram projetadas para serem reusadas ou estendidas. De fato, copiar/colar especificações é o único caminho para reutilizar semânticas de linguagens de programação, ou seja, é extremamente ineficiente e perigosa, no sentido de não produzir uma especificação totalmente confiável, seja por funções semânticas extremamente restritas à linguagem, seja por diferenças sintáticas mínimas que impedem uma integração adequada.

Também devemos considerar o fato de que a maioria das linguagens existentes apresentam uma grande semelhança conceitual do ponto de vista semântico, uma vez que o desenvolvimento de novas linguagens de programação geralmente é influenciado por linguagens previamente existente.

Portanto, o objetivo deste trabalho é, através do uso da semântica de ações baseada em componentes, uma técnica que permite melhorar a reutilização de especificações em semântica de ações, definir componentes semânticos para estruturas sintáticas abstratas, capazes de representar conceitos de linguagens de programação separados pelos diversos paradigmas de linguagens de programação existentes, e organizados de forma hierárquica garantindo assim um alto grau de reutilização semântica.

Como resultados, produzimos uma biblioteca de componentes semânticos, formada pelo agrupamento de componentes semânticos capazes de representar conceitos de linguagens de expressões, imperativas, funcionais e orientada a objetos; e projetamos uma linguagem multiparadigma denominada EIFOO, formada pelo agrupamento dos componentes semânticos especificados em cada um dos paradigmas abordados neste trabalho.

Palavras-chaves: Semântica de Ações, Paradigmas de Linguagens de Programação, Componentes Semânticos, Reusabilidade Semântica.

ABSTRACT

Action semantics, a formalism for programming language specification, defines a standard set of operators that describes common concepts found in programming languages. These operators ease the task of specifying programming languages, because they free the language designers from the need of complex definitions to describe them.

However, action semantics specifications are not designed to be reused or to be extended. In fact, copy/paste is the only way for programming language specification reusability in traditional, plain action semantics, which is extremely inefficient, dangerous, and does not make a totally acceptable specification (some semantic functions are restricted to few languages, minimal syntax differences can impact language integration, etc.).

Also we must consider the great conceptual equivalence among most existing languages from the semantic point of view, since the development of new programming languages generally is influenced by previously existing languages.

The objective of this work is, using component-based action semantics, a new technique for better reutilization of action semantics specifications, to define semantic components for abstract syntax structures, capable of represent programming languages concepts split in different programming language paradigms, and organized like a hierarchy which presents a great semantic reuse level.

As a result, we produce a semantic component library, composed by grouping the semantic components capable of representing imperative, functional and object oriented language concepts; and we design a new multiparadigm language called EIFOO, composed by grouping semantic components specified in each programming language paradigm mentioned in this work.

Key words: Action Semantics, Programming Language Paradigms, Semantic Components, Semantic Reusability.

Agradecimentos

Esta tese não começou a partir do meu ingresso na UFPE, muito menos no meu primeiro dia na escola. Esta tese teve início com um garoto de apenas 15 anos de idade, que, no ano de 1968, estudava à luz de poste a cópia de um livro escrito de próprio punho, procurando aquilo que nos acompanha até o fim de nossos dias, o conhecimento. A pessoa que eu acabo de descrever era meu pai, que faleceu quando eu tinha apenas 6 anos de idade. Ele não me deixou muitas lembranças, mas me deixou um exemplo de esforço e dedicação que sempre vai me acompanhar nos momentos mais difíceis da minha vida. Esta tese é o mínimo que eu posso fazer para demonstrar o orgulho que eu tenho por ele, que o seu esforço não foi em vão e que eu sou muito grato pela oportunidade na vida que ele me deixou.

Agradeço também as duas mulheres da minha vida: a minha mãe, que cuidou sozinha e de forma valente de seus três filhos, mostrando o caminho certo sempre que necessário; e a minha esposa, que durante esses anos de convivência tem demonstrado uma paciência única para com as minhas teimosias.

Gostaria também de agradecer à minha família, sempre paciente e compreensiva, que soube aceitar e compreender as minhas ausências nas alegrias e tristezas.

E finalmente, ao meu orientador Hermano Moura, que com muita paciência, motivação e conhecimento, conseguiu-me trilhar pelos caminhos da pesquisa e do desenvolvimento científico.

Gostaria de prestar também um agradecimento especial ao prof. Luis Menezes, que graças ao seu auxílio e suporte em relação a ferramenta ABACO, tornou viável a conclusão deste trabalho.

Obrigado UFPE !!

ÍNDICE

LISTA DE FIGURAS

Capítulo 1

Introdução

Semântica de ações [1], um formalismo para especificação de linguagens de programação, define um conjunto padrão de operadores que descrevem conceitos comuns encontrados em linguagens de programação. Estes operadores facilitam a especificação de linguagens de programação porque eles liberam o projetista de linguagens da manipulação de definições complexas usadas para descrevê-las.

As especificações em semântica de ações são similares à semântica denotacional [3] (sintaxe abstrata, funções semânticas e entidades semânticas), tornando-se difícil de ler porque a definição de conceitos simples da linguagem de programação é quebrada em várias partes (blocos sintáticos e semânticos de tamanhos distintos) e colocadas em diferentes pontos na especificação. Também, a estrutura da especificação não é projetada para ser reusada ou estendida.

De fato, copiar/colar especificações é o caminho usual para se reutilizar semânticas de linguagens de programação; ou seja, é uma técnica ineficiente e perigosa no sentido de dificultar a produção de uma especificação totalmente confiável, seja por funções semânticas extremamente restritas a uma determinada especificação de uma linguagem (nível de granularidade da função dentro da especificação como um todo), seja por diferenças sintáticas mínimas entre linguagens que impedem uma integração adequada entre as mesmas. Por conta disto, poucos trabalhos na literatura estão atualmente disponíveis mostrando o uso desta técnica na reutilização semântica de especificações de linguagens de programação como um todo.

Conseqüentemente, o reuso semântico pode ser considerado uma necessidade, principalmente se considerarmos o aumento da complexidade das especificações semânticas atualmente desenvolvidas, procurando cada vez mais se aproximar da realidade proposta pelas diversas linguagens de programação existentes.

Portanto, antes de procurarmos especificar novas semânticas para linguagens cada vez mais complexas e extensas, cujas especificações atingiram um nível de especialização que torna muito difícil o reaproveitamento de conceitos semânticos, faz-se necessário melhorar a reutilização semântica, haja vista que a maioria das linguagens de programação existentes apresentam uma grande semelhança conceitual.

Segundo [5] e [6], que propõem a mudança na estrutura das descrições em semântica de ações usando novos construtores, o projetista de linguagens pode, através do encapsulamento de definições sintáticas e semânticas em pequenos módulos independentes, especificar linguagens de modo que: possam ser facilmente modificadas para refletir as mudanças no seu design; e possam ser facilmente reusadas, através do uso de partes de especificações de linguagens já existentes para especificar conceitos similares.

Entretanto, além do reuso semântico, é necessário identificar trabalhos que mostrem, de forma hierárquica, a grande semelhança conceitual entre as diversas linguagens de programação disponíveis, principalmente aquelas que superficialmente parecem ser tão distintas.

1.1 Definição do Problema

Uma tentativa de aplicar o reuso semântico foi feita durante o desenvolvimento de uma semântica de ações para Delphi (Object Pascal) [16]. Sabíamos que era possível representar a semântica de Object Pascal a partir das semânticas de Pascal [12] e de Java [14], [15], resultando assim em uma linguagem híbrida, um Pascal com conceitos de orientação a objetos. Entretanto, apesar das especificações tomadas como base terem sido desenvolvidas pelo mesmo autor, elas apresentavam regras e estilos diferentes, gerando assim uma certa dificuldade durante a reutilização semântica, a qual levantou dúvidas sobre a real qualidade do trabalho final desenvolvido em [16].

Diante disto, precisávamos utilizar uma nova abordagem semântica, cujos conceitos pudessem ser reaproveitados na especificação semântica de antigas e novas linguagens de programação.

Entre as propostas apresentadas, surgiu a idéia de utilizarmos a semântica de ações baseada em componentes [5], [6], que consiste numa extensão da semântica de ações, a qual permite o encapsulamento e o reuso de especificações de uma forma extremamente sofisticada.

Com a capacidade de reuso disponível, restava saber o que realmente deveria ser encapsulado. Para isso utilizamos as notas de aula da disciplina Paradigma de Linguagens de

Programação, ministrada pelos professores Augusto Sampaio e Paulo Borba [20]. Nesta disciplina, os professores demonstraram uma arquitetura completa de classes e objetos capazes de representar conceitos diversos de paradigmas de linguagens de programação, de uma forma simples e reusável.

De fato, este trabalho serviu como um guia para a especificação dos nossos componentes semânticos, demonstrando quais conceitos de linguagens de programação podem ser reaproveitados e de que forma; apresentando uma possível hierarquia de elementos sintáticos reusável entre os diversos paradigmas; e finalmente mostrando a associação destes elementos sintáticos com os respectivos conceitos de linguagens de programação especificados.

1.2 Objetivos

Diante do que foi exposto até o momento, podemos dizer que esta dissertação possui dois principais objetivos:

1. Definir uma biblioteca inicial de componentes semânticos, utilizando partes de especificações já existentes e consagradas na literatura, garantindo assim um certo nível de "qualidade semântica";
2. Classificar os vários componentes semânticos de acordo com conceitos definidos em paradigmas de linguagens de programação, obtendo assim um alto nível de reuso já que estes conceitos são: bem definidos, isolados e reaproveitados pelos diversos paradigmas existentes.

Ao atingir estes objetivos poderemos definir linguagens de alto nível, partindo da reutilização de conceitos mais simples e comuns de outras linguagens já especificadas pela biblioteca de componentes semânticos, aumentando assim a velocidade no projeto de novas linguagens e melhorando a confiabilidade de antigas e novas especificações.

Também pretendemos animar a biblioteca de componentes semânticos através do uso da ferramenta ABACO (Algebraic Based Action COmpiler) [11], que combina orientação a objetos e semântica de ações para produzir implementações de linguagens de programação baseadas em especificações de semânticas de ações. Através do uso desta ferramenta, poderemos validar a biblioteca de componentes semânticos final produzida.

1.3 Organização da Dissertação

Esta dissertação está organizada em seis capítulos. No Capítulo 2 vamos falar sobre os principais métodos para a Especificação Modular de Semântica de Ações. O Capítulo 3 irá mostrar os conceitos iniciais envolvidos no Encapsulamento e Reuso de Linguagens de Programação. No Capítulo 4, apresentaremos a Biblioteca de Componentes Semânticos proposta. O Capítulo 5 descreve Uma Linguagem Multiparadigma, produzida a partir da integração dos componentes semânticos especificados para linguagens de expressões, imperativa, funcional e orientada a objetos, respectivamente. Conclusões e Trabalhos Futuros serão apresentados no Capítulo 6.

Capítulo 2

Especificação Modular em Semântica de Ações

Neste capítulo faremos uma breve introdução sobre a semântica de ações, descrevendo os seus principais elementos (a Notação de Ações e a Notação de Dados), e mostrando uma especificação em semântica de ações para expressões aritméticas. Também apresentaremos os principais métodos de especificação em semântica de ações modulares: Semântica de Ações Modular, Semântica de Ações Baseada em Componentes e Semântica de Ações Orientada a Objetos. No final, vamos falar um pouco sobre a ferramenta ABACO, descrevendo o seu projeto e as suas propriedades.

2.1 Semântica de Ações

Semântica de ações foi desenvolvida por Peter Mosses [1] em colaboração com David Watt [2], com o objetivo de fazer uma especificação semântica mais legível e, conseqüentemente, mais aceita, através do uso de notações especiais (Notação de Ações e Notação de Dados) em substituição a notação funcional utilizada na semântica denotacional [3].

De fato, através do uso de entidades semânticas pré-definidas os projetistas de linguagens não precisa, mais se preocupar com a representação de elementos básicos da linguagem (memória, abstrações, etc.), e o uso de uma notação bem intuitiva (baseada na língua inglesa) permite que várias pessoas consigam entender a especificação semântica, mesmo sem conhecer o formalismo.

Como resultado, semântica de ações tornou-se bastante utilizada na especificação de linguagens de programação (Pascal [32], Java [21], Lisp [28], Haskell [29], etc.), e na geração

automática de compiladores, onde em alguns casos conseguem ser comparáveis aos compiladores escritos manualmente [11].

2.1.1 Notação de Dados

Em Semântica de Ações, números naturais, números inteiros, caracteres, listas, strings, tuplas, mapeamentos, árvores, etc., são classificados como *sorts*. Podemos defini-los como conjuntos de elementos com *operações* relacionadas, sendo que a coleção destes conjuntos formam a Notação de Dados (*Data Notation*).

Exemplos da notação padrão de dados:

- **2, true**: correspondem a valores simples, a chamadas individuais em semântica de ações;
- **truth-value, integer**: *sorts* criados para representar valores verdade e valores inteiros;
- **sum(1,5), both(true,false)**: representam *operações* que resultam no inteiro **6** e no valor booleano **false**, respectivamente.

Dados podem ser produzidos durante a execução de uma ação por uma entidade especial denominada *yielder*. Baseada na informação corrente, esta entidade é avaliada produzindo uma nova informação.

A descrição completa da notação padrão de dados pode ser encontrada no Apêndice B de [1].

2.1.2 Notação de Ações

A Notação de Ações (*Action Notation*) é uma rica notação algébrica para expressar ações. Ações são essencialmente entidades computacionais, que recebem e propagam diversos valores de dados, e podem ser diretamente executadas acessando e/ou mudando a informação corrente. O resultado da execução de uma ação pode ser:

- **complete**, correspondendo a uma término normal;
- **escape**, correspondendo a uma término anormal;
- **fail**, correspondendo ao abandono da execução da ação; ou
- **diverge**, correspondendo à não terminação.

As ações podem ser classificadas como *primitivas* e *compostas*. As ações primitivas podem ser executadas atomicamente, produzindo os resultados previamente descritos.

Ações compostas são formadas pelo uso de ações primitivas em conjunto com combinadores de ações (*action combinators*). Estas podem ser: *seqüenciadas*, uma sub-ação é executada antes de outra; *intercaladas*, as suas sub-ações atômicas são executadas em ordem aleatória; *escolhidas exclusivamente*, onde apenas uma das sub-ações é escolhida para execução.

Exemplos de ações primitivas e compostas:

- **complete**;
- **give 4 and complete**;
- **complete then give the given integer**.

As ações podem manipular (receber e produzir) diversos tipos de dados estabelecendo um fluxo. Os dados transitórios (*transient information*) são passados imediatamente entre as ações. Os bindings (*scoped information*) disponíveis a uma ação podem ser repassados às sub-ações, e as informações disponíveis em células (*storable information*) podem ser lidas e modificadas.

Exemplos do fluxo de dados em semântica de ações:

- **{0 → 20, 1 → true, 2 → cell1}**: constituem dados transitórios, os quais são representados pelo mapeamento de números (*labels*) para dados;
- **{"x" → false, "y" → cell2, "z" → 12}**: representa os *bindings*, ou seja, um conjunto de mapeamentos de identificadores para dados (*bindables*);
- **{cell1 → 20, cell2 → false, cell3 → cell1}**: representa a pilha de alocação de memória, ou seja, o conjunto de mapeamentos entre células e dados específicos (*storables*).

A Notação de Ações é formada por várias partes independentes, as quais são denominadas de *facetas*. Cada faceta trabalha com um tipo de informação específico, separando assim as ações e os combinadores de ações em grupos bem definidos:

- Básica: para especificar o controle de fluxo das ações;
- Funcional: para especificar as ações que processam dados transitórios (recebem e fornecem dados);
- Declarativa: para especificar os *bindings* que são recebidos e produzidos pelas ações;
- Imperativa: para especificar ações que persistem ou consultam informações em memória.

Cada combinador determina o controle e o fluxo de informações entre as ações combinadas, permitindo assim o uso de ações de facetas distintas.

Além dos grupos básicos de facetas citados acima, podemos descrever: a faceta Reflexiva (para especificar abstrações de rotinas e aplicações), a faceta Comunicativa (para especificar a passagem de mensagens assíncronas), a faceta Híbrida (formada pela combinação de ações de facetas distintas), etc.

Maiores informações sobre Notação de Ações e suas facetas podem ser encontradas em [1] e [2].

2.1.3 Especificações em Semântica de Ações

Para mostrar o uso de semântica de ações para descrever linguagens de programação quaisquer, especificamos, no exemplo abaixo, uma semântica de ações para expressões aritméticas:

```
Num ::= [[ Digit+ ]].
Expr ::= Num | [[ Expr "+" Expr ]].

evaluate _ :: Expr → action .
(1) evaluate [[ (N : Num) ]] = give decimal string-of-characters N.
(2) evaluate [[ (E1 : Expr) "+" (E2 : Expr) ]] =
      (evaluate E1 and evaluate E2) then
      give the sum of (the given integer # 1, the given integer # 2).
```

Este exemplo define os seguintes elementos sintáticos: **Num**, representando numerais formados por uma seqüência de dígitos; e **Expr**, formado por numerais ou por uma operação de soma de expressões. A semântica de numerais e expressões é fornecida pela função semântica **evaluate _**, a qual produz: em **(1)**, um valor do *sort* **integer** como resultado da função semântica **decimal string-of-characters _**, que recebe como argumento o elemento sintático **Num**; e em **(2)**, um valor do *sort* **integer** como resultado da função semântica **the sum of _ _**, que recebe como argumentos os valores do *sort* **integer** produzidos pela avaliação semântica dos elementos sintáticos **E1 : Expr** e **E2 : Expr**.

Para representar conceitos simples, como a linguagem de expressões aritméticas apresentada acima, a semântica de ações não apresenta problemas, mas a partir do momento em que se começa a representar conceitos mais complexos (comandos, declaração de procedimentos, etc.), um número considerável de funções semânticas fortemente acopladas precisam ser especificadas. Conseqüentemente, a especificação final fica tão extensa e ilegível que precisa ser quebrada em várias partes, comprometendo o seu entendimento e a sua manutenção.

2.2 Semânticas Modulares

Atualmente, existem alguns métodos, baseados em semântica de ações que propõem melhorias na organização da especificação semântica como um todo. A semântica de ações é por definição modular, mas apenas no nível de ações. Esta modularidade não se apresenta no nível de especificações semânticas em geral.

Estes novos métodos: semântica de ações modular (Seção 2.2.1), semântica de ações baseada em componentes (Seção 2.2.2) e semântica de ações orientada a objetos (Seção 2.2.3), propõem que, através do encapsulamento de definições sintáticas e especificações semânticas em pequenos módulos independentes, poderemos projetar linguagens de modo que: possam ser facilmente modificadas para refletir as mudanças no seu design, e possam ser facilmente reusadas, através do uso de partes de especificações de linguagens já existentes, para especificar linguagens similares.

Faremos a seguir uma breve introdução sobre estes três métodos para o encapsulamento e reuso de especificações semânticas.

2.2.1 Semântica de Ações Modular

Trata-se de um método para composição de linguagens de programação pela combinação de módulos de semânticas de ações. Cada módulo é definido separadamente, e uma linguagem de programação é definida pela combinação de módulos existentes. Este método habilita o desenvolvedor da linguagem a gradualmente desenvolver uma linguagem através da definição, seleção e combinação de módulos específicos.

A estrutura modular resultante difere substancialmente da estrutura previamente empregada em descrições de semântica de ações. De fato, o tamanho de cada módulo é extremamente pequeno comparado a aqueles previamente empregados em descrições de semânticas de ações, e a sua organização como um todo é significativamente diferente: o estilo original procurava especificar uma sintaxe abstrata em um único módulo, e tinha um módulo separado para cada *sort* sintático onde se especificava as funções e equações semânticas para todas as construções sintáticas que estivessem em conjunto com este *sort*. O novo estilo de modularização proposto facilita bastante o reuso direto de módulos inteiros em semântica de ações.

A seqüência de exemplos a seguir mostra o uso deste formalismo para a criação de módulos para representar a semântica de expressões:

```
Expressions {
  syntax
    Expr.
```

```
   semantics
      datum >= expressible.
      (*) evaluate: Expr -> action [giving an expressible]
}
```

O módulo **Expressions** foi criado contendo o *sort* sintático **Expr**. Na seção semântica, iniciada pela palavra-chave **semantics**, o *sort* **datum** determina coleções de dados que podem ser utilizadas nas especificações. O operador **">="** introduz o *sort* **expressible** para ser utilizado nas expressões, definindo que este é *sub-sort* de **datum**. A função semântica **evaluate** é definida como sendo um mapeamento da entidade sintática **Expr** para uma ação que produz um **expressible**.

O módulo anterior serve de base para as seguintes definições:

```
Arithmetic Expressions {
   import Expressions.
syntax
   Expr ::= Num | [[ Expr "+" Expr ]].
   Num ::= [[ digit+ ]].
   E1, E2 : Expr; N : Num
semantics
   expressible >= natural.
   (1) evaluate N = give decimal string-of-characters N.
   (2) evaluate [[ E1 "+" E2 ]] =
      (evaluate E1 and evaluate E2) then
      give the sum of (the given natural#1, the given natural#2).
}
```

Neste segundo módulo as definições de **Expressions** são incorporadas pelo uso da palavra chave **import** que precede o nome do módulo em questão. O novo módulo **Arithmetic Expressions** é então definido por suas seções **syntax** e **semantics**. Na primeira seção determinamos que uma expressão pode ser um numeral **Num** ou a soma entre expressões **(Expr "+" Expr)**.

Na parte semântica deste módulo, definimos, pelo uso do operador **">="**, que **natural** é *sub-sort* de **expressible**, previamente definido em **Expressions**. Indicamos também duas equações semânticas: em **(1)** a entidade sintática representada por **N** está mapeada para um numero natural; em **(2)** definimos que as sub-expressões **E1** e **E2** resultam, quando avaliadas, em dois números naturais, e a seqüência de **([[E1 "+" E2]])** nada mais é que a soma destes números naturais correntes.

Da mesma maneira que definimos o módulo de **Arithmetic Expressions**, podemos definir outros módulos, como **Boolean Expressions** e **Relational Expressions**, para especificar expressões booleanas e relacionais, respectivamente.

Podemos então compor linguagens quaisquer pelo uso de módulos já definidos, como uma linguagem de expressões, por exemplo:

```
Expression Language {
   import Arithmetic Expressions,
        Boolean Expressions, Relational Expressions.
}
```

Esta definição, no entanto, é apenas ilustrativa e não serão detalhados os módulos correspondentes.

É importante ressaltar que, quando se combinam módulos quaisquer, podem ocorrer alguns conflitos na definição final do módulo combinado. De fato, a combinação de módulos é uma operação muito delicada, e para ser efetuada de forma correta devem ser observadas implicações importantes, as quais são detalhadas em [4].

2.2.2 Semântica de Ações baseada em Componentes

Semântica de ações baseada em componentes define um novo estilo de especificação de semânticas de ações, o qual pode ser usado para produzir descrições mais flexíveis e reusáveis de linguagens de programação [5], [6].

Assim como os módulos de [4], os componentes são constituídos de duas partes básicas: **syntax** e **semantics**, acrescido de um conjunto de novas operações, as quais serão usadas para descrever estruturas de diversas linguagens de programação.

Segue abaixo as principais operações criadas pela notação de componentes:

```
(1) syntax _ --> _ semantics _ :: syntax-tree, syntax-name, action -> language-description.
(2) _ and _ :: language-description, language-description -> language-description.
(3) semantics of _ :: Identifier -> action.
(4) compile _ _ _ :: language-description, syntax-name, syntax-tree -> action.
```

A operação semântica **(1)** produz o *sort* **language-description**, associando a uma **syntax-tree**: um **syntax-name**, representando o *namespace* do componente; e uma ação, representando a semântica do componente. A operação semântica **(2)** efetua a junção de duas **language-description**, produzindo no final uma linguagem única. Quando existe uma referência para um subcomponente, na **syntax-tree** de um outro componente, pode-se utilizar a operação semântica **(3)** para efetuar a avaliação semântica deste subcomponente. A operação semântica **(4)** irá transformar uma **syntax-tree** qualquer, usando uma **language-description** previamente definida, em um conjunto de ações, as quais fornecem um significado semântico para o código sintático fornecido.

A seqüência de exemplos a seguir mostra o uso deste formalismo para a criação de componentes semânticos para representar a semântica de expressões:

```
NumericConstant :: -> language-description.
NumericConstant =
   | syntax
```

```
|    [ n # number ] --> * Exp
| semantics
|    give * n.

SumOfExpressions :: -> language-description.
SumOfExpressions =
   | syntax
   |   [ x # * Exp "+" y # * Exp ] --> * Exp
   |  semantics
   |   | semantics of x and then semantics of y
   |   then
   |   | give sum them.
```

A operação **NumericConstant** produz uma **language-description** usando a operação **(1)**. Ela define um componente semântico com as seguintes características: a sintaxe associa o identificador **n** a um **number** qualquer, o *namespace* é o identificador **Exp**, e a semântica tem como objetivo produzir o valor de **n**.

A operação **SumOfExpressions** produz também uma **language-description**. Ela define um componente semântico com as seguintes características: a sintaxe é formada por dois subcomponentes **Exp** (**x** e **y**) separados pelo símbolo **"+"**, o *namespace* também é o identificador **Exp**, e a sua semântica consiste na soma do resultado das avaliações das semânticas dos dois subcomponentes **Exp**. Esta avaliação recursiva só pode ser efetuada pelo uso da operação **(3)**.

Para se definir uma linguagem de expressões aritméticas, basta combinar os dois componentes semânticos previamente definidos, através do uso da operação **(2)**, como é demonstrado a seguir:

```
ArithmeticExpressions :: -> language-description.
ArithmeticExpressions = SumOfExpressions and NumericConstant.
```

E, finalmente, para produzir as ações a serem avaliadas, utilizamos a operação **(4)**, onde informamos a **language-description**, o *namespace* inicial a ser avaliado, e o código sintático que receberá um significado semântico.

Como exemplo, criamos a operação **run test**, que irá produzir uma ação, gerada a partir da avaliação dos componentes semânticos para representar expressões aritméticas associados a **syntax-tree** fornecida, cujo resultado será **complete**, seguida pela valor transitório **35**, resultante da avaliação da soma do componente **[5]** com o resultado da avaliação do subcomponente **[[10] "+" [20]]**:

```
run test :: -> action .
run test =
compile (ArithmeticExpressions) (* Exp)
   [
    [  [ 10 ] "+" [ 20 ] ]
    "+"
    [ 5 ]
   ].
```

Desta forma, podemos efetuar a combinação de diversos componentes semânticos, e caso ocorra algum conflito sintático no mesmo *namespace* de uma linguagem, a operação **(4)** irá gerar as ações dos possíveis componentes semânticos combinadas pelo combinador de ações **or**.

2.2.3 Semântica de Ações Orientada a Objetos

A semântica de ações orientada a objetos foi desenvolvida com o objetivo de melhorar o encapsulamento de especificações em módulos ou componentes em semântica de ações, adicionando algumas facilidades advindas da área de orientação a objetos [7].

Suas especificações definem um conjunto finito de classes, cujo relacionamento é especificado por meio de diretivas, que indicam os objetos instanciados pela classe atual, assim como a hierarquia à qual esta pertence. Assim como em [4], [5] e [6], uma classe em semântica de ações orientada a objetos deve ser composta de duas partes básicas: **syntax** e **semantics**.

O reuso de especificações é efetuado pela instanciação de objetos que representam as partes da linguagem. A própria definição de objetos sugere a divisão da descrição da linguagem em partes. Cada objeto é definido e uma interface é estabelecida entre eles, permitindo seu uso em dado instante. Conseqüentemente, a definição de uma hierarquia de classes simplificaria a especificação de novas linguagens.

A seqüência de exemplos a seguir mostra o uso deste formalismo para a criação de classes e objetos para representar a semântica de expressões:

```
Class Expression
  syntax:
    Exp
  semantics:
    evaluate _ : Exp -> Action
End Class
```

Na seção de sintaxe (**syntax**) do exemplo acima, definimos o *sort* sintático **Exp**, o qual representa a raiz da árvore sintática de expressões. Uma função semântica '**evaluate _**' foi criada na seção de semântica (**semantics**), indicando que um *sort* **Exp** está mapeado para uma ação (**Action**).

Em semântica de ações orientada a objetos, as funções e equações semânticas têm o mesmo papel dos *métodos* em Orientação a Objetos, portanto, a função '**evaluate _**' pode ser vista como um método de **Expression**, podendo ser redefinida (*sobrecarregada*) por suas sub-classes.

```
Class SumExp
  extending Expression
```

```
using E1:Expression, E2:Expression
syntax:
   Exp ::= E1 "+" E2
semantics:
   execute [[ E1 "+" E2 ]] =
            evaluate E1
         and
            evaluate E2
      then
         give sum(the given integer # 1, the given integer # 2)
End Class
```

Com base na diretiva **extending** indicamos que a classe **SumExp** é uma extensão de **Expression**. Dois objetos (**E1:Expression** e **E2:Expression**) são instanciados pelo uso da diretiva **using** para serem posteriormente utilizados.

Na seção de sintaxe, acrescentamos a descrição da estrutura de uma nova expressão, redefinindo a árvore sintática. Um método '**evaluate _**' é definido para esta expressão e sua semântica é especificada.

Em nossa abordagem, a ação representada por **evaluate E1**, por exemplo, é obtida pela chamada do método **evaluate** à instância do objeto **Expressions**. Desta maneira, interfaces devem ser disponibilizadas pelo objeto para expressar sua semântica.

Espera-se que, com a crescente utilização das técnicas de orientação a objetos, este formalismo ou outros semelhantes, baseados na mesma idéia, possam ser empregados na representação formal de linguagens de programação.

2.3 Animando Especificações com o ABACO

O ABACO (Algebraic Based Action COmpiler)[1] é um ambiente de desenvolvimento que contém ferramentas que ajudam no desenvolvimento de especificações em semântica de ações. Uma vez escrita, esta especificação pode ser usada para fornecer a semântica de um programa-fonte válido [11].

Segue abaixo algumas ferramentas do ABACO que ajudarão no desenvolvimento de especificações em semântica de ações:

- Compilador de especificações – usado pelo sistema para animar especificações em semântica de ações. Ele recebe uma especificação fonte, escrita no formalismo algébrico usado pelo ABACO, processa-a para encontrar erros sintáticos, e produz interpretadores para a especificação. O interpretador produzido é capaz de reconhecer termos válidos e avaliá-los. Avaliar um termo significa reduzi-lo pela aplicação das equações de reescrita contidas na especificação fonte;

- Biblioteca de Ações – formada pelos seguintes componentes: as especificações da notação de ações escritas no formalismo algébrico usado pelo ABACO; e o interpretador de ações que é responsável por animar a performace das ações escritas na notação de ações;
- Interface do Sistema – A interface do ABACO é formada por uma janela principal, a qual é mostrada na **Figura 2.1** abaixo:

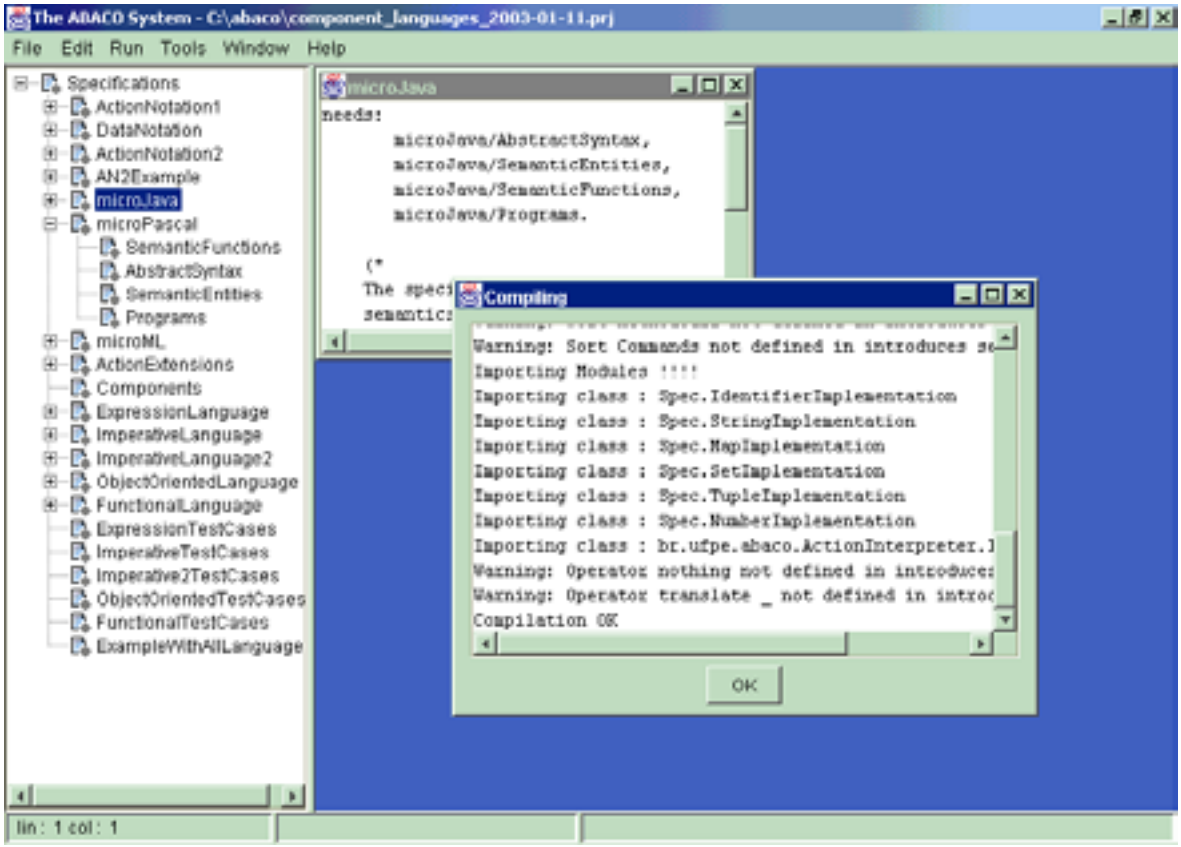

Figura 2.1 – Sistema de interface do ABACO

Foram necessárias algumas modificações na ferramenta ABACO durante a especificação da biblioteca de componentes semânticos, gerando assim uma versão exclusiva desta ferramenta para esta dissertação. Entre as modificações aplicadas, foram corrigidos pequenos bugs relacionados a animação de ações, e foram implementadas novas ações para suportar a semântica de ações baseada em componentes (Seção 2.2.2).

Capítulo 3

Encapsulamento e Reuso de Linguagens de Programação

Neste capítulo, vamos apresentar alguns conceitos iniciais sobre paradigmas linguagens de programação, seguido de uma breve explanação sobre a representação de uma hierarquia de conceitos de linguagens de programação. Também apresentaremos as principais razões envolvidas no uso de componentes semânticos, para a construção de uma biblioteca de conceitos de linguagens de programação, ao invés de módulos ou objetos semânticos. No final, mostraremos uma biblioteca inicial de componentes semânticos para representação de uma linguagem de expressões.

3.1 Paradigmas de Linguagens de Programação

As primeiras linguagens de programação de alto nível surgiram durante a década de 50 [10]. Desde então, as linguagens de programação têm sido uma fascinante e produtiva área de estudo. Programadores promovem intensos e intermináveis debates sobre os relativos méritos das suas linguagens favoritas de programação, algumas vezes de forma religiosa. Já em um nível mais acadêmico, cientistas da computação procuram desenvolver linguagens que combinem poder de expressão com simplicidade e eficiência.

Atualmente, existem diversas linguagens de programação, e algumas delas são tão complexas e irregulares que se tornaria quase impossível a tarefa de conhecer todas as características de todas as linguagens de programação existentes [10].

De qualquer forma, na medida em que foram criadas diversas linguagens, por qualquer que seja o motivo (propósitos diferentes, avanços tecnológicos, interesses comercias, cultura e background científico, etc.), surgiram diversos conceitos diferentes, mas que sempre se

mostravam presentes em uma linguagem ou outra. Geralmente, o design de uma linguagem nova era (e ainda é até hoje) influenciado pelo desenvolvimento de linguagens anteriores.

Devido a esta influência, começava-se a perceber que havia uma grande semelhança conceitual, um reuso de conceitos entre estas linguagens de programação, principalmente aquelas que superficialmente pareciam ser tão distintas, formadas por mecanismos como: nomes simbólicos, transferência de controle, estruturação de dados, definição de procedimentos, etc. Em qualquer linguagem, estes mecanismos eram governados pelos mesmos conceitos gerais, e entendê-los tornariam mais fácil o uso, a descrição, a comparação, a implementação e o design das linguagens de programação existentes.

Estes conceitos individuais começaram a ser agrupados, dando origem a uma área de pesquisa na computação denominada de *Paradigmas de Linguagens de Programação* [8], [9], [10], a qual se preocupa com o modelo, padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns. A classificação de linguagens em paradigmas é uma conseqüência de decisões de projeto que impactam radicalmente a forma na qual uma aplicação real é modelada do ponto de vista computacional

3.2 Definindo uma Hierarquia de Conceitos de Linguagens de Programação

Nesta dissertação, iremos analisar: os conceitos gerais envolvidos no desenvolvimento de novas linguagens de programação (valores, armazenamento, associações, abstrações, encapsulamento, sistema de tipos, etc.); os paradigmas que essas linguagens suportam: imperativo (Seções 4.1 e 4.2), orientado a objetos (Seção 4.4), funcional (Seção 4.3), etc.; e a maneira como todos estes elementos podem ser colocados juntos para formar uma linguagem de programação completa (Capítulo 5).

Para que possamos efetuar essa junção, precisamos definir uma hierarquia de conceitos de linguagem de programação, a qual delimita significados computacionais a determinadas construções sintáticas de determinados paradigmas. De fato, a partir do momento em que nós tivermos as estruturas sintáticas definidas, teremos condições de garantir uma implementação incremental de construções de linguagens nos diversos paradigmas, partindo do princípio de que para cada estrutura sintática delimitada haverá uma semântica especificada.

Um bom exemplo de definição de hierarquia de conceitos de linguagens de programação pode ser encontrado em [20], cujo objetivo era apresentar uma análise crítica de paradigmas de linguagens de programação, destacando: uma visão geral dos paradigmas imperativo, orientado

a objetos e funcional; discussões de alternativas de projeto de linguagens; e estudo de linguagens através de ambientes de execução (interpretação). De fato, [20] estuda com certa profundidade alguns conceitos centrais do projeto de linguagens, desenvolvendo uma visão crítica das mesmas e consolidando conceitos como abstração de dados, modularidade e reusabilidade.

A metodologia de ensino usada em [20] se baseava em conceitos, onde a linguagem Java [21], [33] era utilizada como uma ferramenta prática de apoio para ilustrar o paradigma OO, e como meta-linguagem para construir interpretadores. Os conceitos de linguagens de programação eram sedimentados através da implementação incremental de construções de linguagens dos diversos paradigmas.

Seguindo a mesma linha de raciocínio que as implementações de classes em Java, capazes de dar um significado computacional a determinadas construções sintáticas de determinados paradigmas de linguagens, vamos especificar componentes semânticos para dar um significado computacional a determinadas construções sintáticas de determinados paradigmas de linguagens, utilizando [20] como um guia para determinar quais estruturas sintáticas e semânticas serão relevantes para uma hierarquia de conceitos de linguagens de programação.

A definição desta hierarquia será observada durante a especificação de cada componente semântico, definido um comportamento semântico específico, para cada estrutura sintática associada a um conceito distinto de cada paradigma abordado.

3.3 Especificando Componentes Semânticos

Toda linguagem possui uma sintaxe e uma semântica. A sintaxe de uma linguagem de programação está concentrada na formação de frases em uma linguagem, isto é, em como as expressões, comandos, declarações, etc. são colocados juntos para formar um programa. Uma sintaxe de uma linguagem influencia na maneira como os programas são escritos pelo programador, lidos pelos outros programadores, e analisados pelo computador.

Cada linguagem possui regras que definem como as frases são compostas por símbolos e por outras frases. Estas regras constituem a gramática de uma linguagem, a qual define as relações hierárquicas de frases e subfrases de uma linguagem [2].

Pode-se classificar a sintaxe de uma linguagem em dois grupos: concreta e abstrata. A sintaxe abstrata se concentra apenas na estrutura composicional de frases de programas, ignorando como estas estruturas foram determinadas. Já a sintaxe concreta se preocupa não

apenas com a estrutura composicional, mas também com o reconhecimento de programas em textos e a análise imparcial de símbolos particulares e frases sintáticas.

A semântica de uma linguagem de programação está concentrada no significado do programa, definindo regras semânticas que podem ser verificadas antes (semântica estática) e durante (semântica dinâmica) a execução dos mesmos. De fato, uma semântica de uma linguagem determina a maneira como os programas são compostos pelo programador, entendidos por outros programadores, e interpretados pelo computador. Sintaxe é importante, mas semântica é mais importante ainda [10].

Existem vários formalismos para especificar a semântica de linguagens de programação, tais como: algébrica [2], denotacional [3], operacional [24], etc., entretanto preferimos utilizar a semântica de ações por questões já apresentadas na Seção 2.1.

Infelizmente, mesmo com o uso de semântica de ações, ainda é difícil especificar a semântica de linguagens reais e complexas. A forma como uma especificação em semântica de ações é organizada (em três grandes blocos: sintaxe abstrata, entidades semânticas e funções semânticas) torna difícil a leitura porque a definição de conceitos simples da linguagem é quebrada em várias partes (sintaxe e semântica) e colocadas em diferentes pontos na especificação, mesmo sendo fortemente acopladas.

Portanto, antes de procurarmos especificar novas semânticas para linguagens cada vez mais complexas e extensas, faz-se necessário melhorar a organização da especificação semântica, procurando obter uma maior reutilização semântica, haja vista que a maioria das linguagens existentes apresentam uma grande semelhança conceitual. Atualmente, existem alguns métodos, baseados em semântica de ações que propõem melhorias na organização da especificação semântica como um todo. A semântica de ações é por definição modular, mas apenas no nível de ações, esta modularidade não se apresenta no nível de especificações semânticas em geral.

Esses novos métodos (semântica de ações modular, semântica de ações baseada em componentes e semântica de ações orientada a objetos), definidos no Capítulo 2, propõem que, através do encapsulamento de definições sintáticas e semânticas em pequenos módulos independentes, poderemos especificar linguagens de modo que: possam ser facilmente modificadas para refletir as mudanças no seu design; e possam ser facilmente reusadas, através do uso de partes de uma definição de linguagem já existente, para especificar linguagens similares.

Cada um destes métodos apresentam vantagens e desvantagens, as quais serão apresentadas com o objetivo de justificar a nossa escolha pela semântica de ações baseada em

componentes para definir a biblioteca de componentes semânticos, que é o objetivo final desta dissertação.

A semântica de ações modular, apesar de ser o método mais antigo e experimentado entre os três citados, apresenta alguns problemas, principalmente no que se refere a dificuldade de combinar módulos quaisquer, podendo ocorrer alguns conflitos na definição final do módulo combinado. De fato, a combinação de módulos é uma operação muito delicada, e para ser efetuada de forma correta devem ser observadas algumas sugestões detalhadas em [4].

A semântica de ações orientada a objetos, por ser uma melhoria dos métodos modular e baseado em componentes, seria a mais indicada para especificar uma biblioteca de componentes semânticos, principalmente por causa das facilidades advindas da área de orientação a objetos tais como: controle de acesso, definição de interfaces, construção de hierarquias de classes semânticas, etc. Entretanto, este método ainda não é suportado por nenhuma ferramenta de processamento de semântica de ações (OASIS, ABACO, etc.), é mais complexo que a semântica de ações baseada em componentes, e, por se tratar de uma técnica muito recente, ela pode sofrer ajustes durante o seu ciclo de vida, como ocorreu com a semântica de ações baseada em componentes [6].

A semântica de ações baseada em componentes, na sua concepção inicial [5], era complicada, apresentava definições/operações ambíguas e desnecessárias, e definia duas novas notações (componentes e linguagens) que precisariam ser suportadas pelas devidas ferramentas de processamento de semântica de ações. Com as modificações propostas em [6], apenas uma notação final foi proposta, e as ambigüidades foram removidas, de modo que restaram apenas quatro operações fundamentais (descritas na Seção 2.2.2), as quais são suficientes para especificar toda uma biblioteca de componentes semânticos. Outra característica muito importante é o processamento desta nova notação pela ferramenta ABACO [11], o que garante uma melhor corretude na sintaxe e na semântica da biblioteca de componentes semânticos final produzida.

Portanto, para atingir o objetivo final desta dissertação, usaremos a semântica de ações baseada em componentes [5], [6] para dar um significado semântico para estruturas sintáticas abstratas (as quais fornecem uma ligação bastante conveniente entre a sintaxe e a semântica de uma linguagem). Estas estruturas sintáticas serão capazes de representar conceitos de linguagens de programação reutilizáveis, modularizados pelos diversos estilos de programação existentes, uma vez que serão definidas pela hierarquia de conceitos de linguagens de programação descrita na Seção 3.2.

3.4 Uma Biblioteca de Componentes Semânticos para uma Linguagem de Expressões

Para iniciar a especificação da nossa biblioteca de componentes semânticos, vamos definir os componentes necessários para dar significado a uma linguagem de expressões, a qual utiliza conceitos de linguagens de programação como valores e expressões. Vale salientar que não existe um paradigma de linguagens de expressões, mas sim um conjunto de operações básicas executadas em valores de diversos tipos, e repetidas nos demais paradigmas os quais serão abordados logo mais nesta dissertação.

3.4.1 Valores e Expressões

Um valor representa um dado, que é o principal "material" de computação existente. De certo modo, dados são mais importantes do que programas. Um grande volume de dados, como um dicionário ou uma lista telefônica, é relativamente mais importante, em valores econômicos, do que o programa que irá manipulá-los [10]. Todo valor está associado a um tipo, logo pode-se concluir que um tipo é formado por um conjunto de valores. Entretanto, é importante frisar que todos os valores de um tipo devem exibir um comportamento uniforme em relação as operações associadas ao mesmo [10].

Expressões são combinações ordenadas de valores literais (ou constantes), variáveis, operadores, parênteses e chamadas de métodos, que permitem realizar cálculos aritméticos, concatenar strings, comparar valores, realizar operações lógicas e manipular objetos. Sem expressões, uma linguagem de programação seria praticamente inútil. O resultado da avaliação de uma expressão é em geral um valor compatível com os tipos dos dados que foram operados [8].

Na biblioteca de componentes semânticos, a representação semântica de valores literais (tais como "**3**", "**true**") foi feita a partir da produção de componentes semânticos que dão um significado semântico a constantes literais. Já a representação de expressões se dá a partir do uso de componentes que interpretam operações unárias e binárias, de uma forma independente dos tipos de dados dos operandos envolvidos.

Segue abaixo as operações semânticas que geram componentes para representar constantes literais (**exp-literal-integer** e **exp-literal-boolean**) e expressões unárias e binárias (**exp-operations**):

```
exp-literal-integer :: -> language-description.
exp-literal-integer =
   | syntax
```

```
|    [ i # integer-value ] --> * Expression
| semantics
|    give * i.

exp-literal-boolean :: -> language-description.
exp-literal-boolean =
   | syntax
   |    [ b # boolean-value ] --> * Expression
   | semantics
   |    give * b.

exp-operations :: -> language-description.
exp-operations =
   | syntax
   |   [ e1 # * Expression op # * Operator e2 # * Expression ] --> * Expression
   | semantics
   |   | | semantics of op
   |   | and then
   |   | | semantics of e1
   |   | and then
   |   | | semantics of e2
   |   then
   |   | binary-operation-of (the given string # 1) (the given value # 2) (the given value # 3)
   and
   | syntax
   |   [ op # * Operator e # * Expression ] --> * Expression
   | semantics
   |   | | semantics of op
   |   | and then
   |   | | semantics of e
   |   then
   |   | unary-operation-of (the given string # 1) (the given value # 2).
```

A operação **exp-literal-integer** gera um componente que recebe um elemento sintático compatível com o *sort* **integer-value**, e semanticamente produz o valor associado a este elemento. O *sort* **integer-value** é um *subsort* de **primitive-value**, e é equivalente ao *sort* **number** (especificado previamente na notação de dados da ferramenta ABACO [11]), o qual está diretamente relacionado com a especificação de valores numéricos, portanto trata-se de um componente semântico para valores do tipo inteiro.

A operação **exp-literal-boolean** gera um componente semelhante à operação **exp-literal-integer**, mas sendo este compatível com o *sort* **boolean-value**, o qual é um *subsort* de **primitive-value** e é equivalente ao *sort* **truth-value**. Trata-se de um componente semântico que especifica valores do tipo booleano.

A operação **exp-operations** produz dois componentes, os quais especificam operações binárias e unárias, respectivamente. Para representar as operações binárias, foram utilizados subcomponentes, os quais, quando da avaliação de suas semânticas, obtêm-se os valores (**e # * Expression**) e o identificador da operação (**o # * Operation**) desejados. Estes valores transitórios serão utilizados pela operação semântica **binary-operation-of _ _ _**, de modo que, para cada tipo

de valor e identificador de operação transitório, existirá uma expressão semântica para os mesmos.

As operações unárias se assemelham as operações binárias, mas possuem apenas um valor (**e # * Expression**) e o identificador da operação (**o # * Operation**), e utilizam a operação semântica **unary-operation-of _ _**.

Segue abaixo alguns exemplos de redefinições semânticas para as funções semânticas **binary-operation-of _ _ _** e **unary-operation-of _ _**:

```
binary-operation-of "AND" (b1 : boolean-value) (b2 : boolean-value) =
   give both (b1, b2).
binary-operation-of "OR" (b1 : boolean-value) (b2 : boolean-value) =
   give either (b1, b2).

unary-operation-of "NOT" (b : boolean-value) =
   give not b.
```

É importante salientar que se pode criar representações para valores de qualquer tipo, através da definição de um novo componente. Ele irá trabalhar com um elemento sintático compatível com o *sort* associado ao novo tipo, e especificará as expressões semânticas relacionadas ao tipo do novo componente.

Os identificadores de qualquer operação, unária ou binária, são produzidos pela operação **exp-operators**, a qual comporta componentes que têm como objetivo produzir *strings* específicas para serem avaliadas posteriormente pelas funções semânticas **binary-operation-of _ _ _** e **unary-operation-of _ _**. Como nada impede que um mesmo identificador de operação possa ser usada por operações distintas, temos a capacidade de representar sobrecarga de operadores.

Segue abaixo uma versão resumida da operação **exp-operators**, produzindo identificadores de operação para os operadores "**+**" e "**-**":

```
exp-operators :: -> language-description.
exp-operators =
   | syntax
   |    [ "+" ] --> * Operator
   | semantics
   |    give "PLUS"
   and
   | syntax
   |    [ "-" ] --> * Operator
   | semantics
   |    give "MINUS".
```

3.4.2 Declarações

Além da utilização de valores literais, as expressões também trabalham com valores associados a um identificador qualquer. Este processo de associação representará uma *declaração de constante*, na qual um programador consegue associar um identificador a um valor obtido de uma expressão e esse identificador pode ser consultado em qualquer expressão do programa.

Para que o programador possa trabalhar com mais de uma constante em uma expressão, se faz necessário o uso de declarações compostas; e, quando o programador trabalha com expressões em blocos, mas não quer efetuar nenhuma declaração local ao bloco, se faz o uso de uma declaração vazia.

Segue abaixo a operação semântica (**exp-declarations**) capaz de gerar componentes para representar declaração de constante, declaração composta e declaração vazia:

```
exp-declarations :: -> language-description.
exp-declarations =
   | syntax
   |    [ "const" i # Identifier "=" e # * Expression ] --> * Declaration
   | semantics
   |    | semantics of e
   |    then
   |    | bind token of * i to the given value
   and
   | syntax
   |    [ d1 # * Declaration ";" d2 # * Declaration ] --> * Declaration
   | semantics
   |    | semantics of d1
   |    before
   |    | semantics of d2
   and
   | syntax
   |    [ ] --> * Declaration
   | semantics
   |    complete.
```

A partir do momento em que declaramos uma constante com um determinado valor, precisamos criar meios para que possamos consultá-la. Como uma constante é uma expressão, definimos a operação semântica **exp-constants**, que produz um componente capaz de fazer a avaliação sintática de um identificador em uma expressão e produzir o valor associado ao respectivo **Identifier**.

Segue abaixo a operação semântica **exp-constants**, que gera o componente responsável pela avaliação de constantes:

```
exp-constants :: -> language-description.
exp-constants =
   | syntax
   |    [ i # Identifier ] --> * Expression
   | semantics
```

```
|    give the value bound to token of * i.
```

3.4.3 Bloco de Expressões

Com a declaração uma constante, cria-se um ciclo de vida para a mesma, de modo que não apenas outras declarações, mas também as demais expressões compartilham a sua existência. Com essa afirmação entramos no conceito de bloco, o qual define uma área de escopo para o uso das declarações de constantes dentro das expressões.

A operação semântica **exp-block** é responsável pela produção do componente de bloco de expressões. Além dos elementos sintáticos "**let**" e "**in**", este componente trabalha com os subcomponentes: **d # * Declaration** e **e # * Expression**, os quais irão produzir as constantes e as expressões que deverão trabalhar com essas constantes locais. Como um bloco é uma expressão, o subcomponente **e # * Expression** pode ser um novo bloco, representando assim o conceito de blocos aninhados, onde a declaração de uma constante local irá sobrepor qualquer declaração de constante global com o mesmo identificador.

Segue abaixo a operação semântica **exp-block**, que gera o componente responsável pela construção de blocos de expressões:

```
exp-block :: -> language-description.
exp-block =
   | syntax
   |    [ "let" d # * Declaration "in" e # * Expression ] --> * Expression
   | semantics
   |    | furthermore semantics of d
   |    hence
   |    | semantics of e.
```

3.4.4 Agrupando Componentes Semânticos de Expressões

Para finalizar a nossa biblioteca de componentes semânticos para expressões, criamos a operação semântica **exp-program**, a qual produz um indivíduo do *sort* **language-description**, formada pelo agrupamento de todos os componentes semânticos de expressões necessários para a animação de programas baseados em expressões:

```
exp-program :: -> language-description.
exp-program =
   | exp-literal-integer
   and
   | exp-literal-boolean
   and
   | exp-constants
   and
   | exp-operations
   and
   | exp-operators
```

```
and
| exp-declarations
and
| exp-block.
```

Todo agrupamento de componentes semânticos pode ser considerado como uma biblioteca de componentes semânticos. Neste momento, como especificamos apenas componentes semânticos para uma linguagem de expressões, nada mais justo do que denominar o agrupamento de componentes semânticos, produzidos pela operação semântica **exp-program**, de biblioteca de componentes semânticos para expressões. No Capítulo 4, quando for mostrado os componentes semânticos especificados para os diversos paradigmas (imperativo, funcional e orientado a objetos), também encontraremos definições como: biblioteca de componentes semânticos imperativos, biblioteca de componentes semânticos funcionais, etc.

Na compilação de programas baseados em expressões, usamos a operação semântica **compile**, a qual recebe como argumentos: uma **language-description**, neste caso **exp-program**; um *namespace* (**Expression**) definindo o enquadramento do valor final produzido; e a árvore sintática de expressões, representando o código do programa propriamente dito.

Exemplo 3.1: Declarando Expressões Aninhadas

Como exemplo, criamos um programa baseado em expressões (**Figuras 3.1** e **3.2**) que declara as constantes **k** e **j** em um bloco, cujos valores são **10** e **30** respectivamente. A expressão do bloco é formada por um sub-bloco que redeclara a constante **j** com o valor **20**, e cuja expressão é a soma das constantes **k** e **j**.

```
let
   const k = 10;
   const l = 30
in
   let
      const i = 20
   in
      k + i
```

Figura 3.1 – Linguagem concreta do programa Exemplo 3.1.

```
compile (exp-program) (* Expression)
   [
    "let"
    [
     [ "const" k "=" [ 10 ] ]
     ","
     [ "const" j "=" [ 30 ] ]
    ]
    "in"
    [
     "let" [ "const" j "=" [ 20 ] ] "in" [ [ k ] [ "+" ] [ j ] ]
    ]
   ]
```

Figura 3.2 – Linguagem abstrata do programa Exemplo 3.1 para uso da função semântica compile _ _ _.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 3.3** abaixo) que, quando executadas produzem: um valor transitório igual a **30**, e as associações **{k → 10, j → 20}**.

```
| furthermore
| | | | give 10
| | | then
| | | | bind "k" to  (the given value)
| | before
| | | | give 30
| | | then
| | | | bind "j" to  (the given value)
before
| | furthermore
| | | | give 20
| | | then
| | | | bind "j" to  (the given value)
| before
| | | | give "PLUS"
| | | and then
| | | | | give  (the value bound to "k")
| | | | and then
| | | | | give  (the value bound to "j")
| | then
| | | binary-operation-of  (the given string # 1)   (the given value # 2)   (the given value # 3)
```

Figura 3.3 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 3.1.

□

Capítulo 4

Biblioteca de Componentes Semânticos

Neste capítulo, vamos mostrar os componentes semânticos finais definidos nesta dissertação, os quais estão agrupados pelos seus respectivos paradigmas. Vamos associar cada componente especificado ao seu respectivo conceito em um determinado paradigma de linguagem de programação, e ainda explicar a presença/ausência de conceitos específicos em um determinado paradigma. Serão abordados três paradigmas de linguagens de programação, os quais são considerados de extrema importância no desenvolvimento de sistemas diversos: imperativo, funcional e orientado a objetos. Vamos mostrar também os componentes semânticos especificados para representar estruturas avançadas (records, arrays, ponteiros, etc.) do paradigma imperativo.

4.1 Componentes Imperativos

O modelo imperativo de programação, considerado o mais antigo de todos, baseia-se no modo de funcionamento do computador, ou seja, em comandos e atualização de variáveis. Isto é refletido na execução seqüencial de comandos e no armazenamento de dados alteráveis, ou seja, na maneira como os computadores executam programas a nível de linguagem de máquina. O paradigma imperativo foi predominante nas linguagens de programação, pois tais linguagens são mais fáceis de se traduzir numa forma adequada para execução na máquina, o que torna os ambientes de execução das linguagens imperativas bastante eficientes. Linguagens imperativas também têm sido chamadas de "baseadas em comandos" ou "orientada a atribuições", devido às suas características [9].

Um bom exemplo de linguagem imperativa é a linguagem de programação Pascal [32], originalmente desenvolvida por Niklaus Wirth, em 1971. Ela foi desenvolvida para ensinar técnicas e tópicos de programação em instituições de ensino.

4.1.1 Variáveis e Tipos

O primeiro conceito do paradigma imperativo que iremos explanar é o conceito de variável [10]. Basicamente, uma variável é uma entidade que contém um valor, o qual pode ser inspecionado e atualizado sempre que for desejado. Variáveis são usadas para modelar entidades do mundo real que possuem um estado, tais como a população do mundo, a data de hoje, o clima atual, ou índices econômicos de um país [10].

De forma semelhante aos valores, as variáveis geralmente possuem um tipo [8], [9], [10]. O tipo de uma variável determina que valores podem ser armazenados nela, e também determinam a organização da memória quando da alocação/liberação de espaço da mesma para um tipo qualquer [10].

Na biblioteca de componentes semânticos, a representação semântica de variáveis se dá a partir da definição de *sorts* tais como **integer-variable**, **boolean-variable**, etc. Estes *sorts* serão utilizados na declaração de variáveis, cuja semântica é especificada pelo componente semântico produzido pela operação **imp-declarations** descrita abaixo:

```
imp-declarations :: -> language-description.
imp-declarations =
   | syntax
   |    [ "var" i # Identifier ":" t # * Type ] --> * Declaration
   | semantics
   |   | semantics of t
   |   then
   |   | | allocate variable of (the given type)
   |   | then
   |   | | bind (token of * i) to (the given variable).
```

Podemos observar, na seção **syntax**, que este componente semântico estende o *namespace* **Declaration**, definido pela operação **exp-declarations** na biblioteca de componentes semânticos para expressões (Seção 3.4.2); e que foi definido um novo *namespace* denominado **Type**, no qual subentendesse que todo componente semântico deste *namespace* deve produzir um valor do *sort* **type**. Na seção **semantics**, podemos observar a avaliação semântica do subcomponente **Type**; seguida pela alocação de memória da nova variável, utilizando como modelo o valor transitório do *sort* **type** produzido pela ação anterior; e no final a associação do valor do elemento sintático **i # Identifier** com o valor transitório do *sort* **variable** produzido pela função semântica **allocate variable of _**.

Os primeiros componentes semânticos que estendem o *namespace* **Type** são os que produzem valores do *sort* **integer-type** e **boolean-type**, ou seja, os tipos primitivos básicos para representação de valores numéricos e booleanos:

```
imp-type-integer :: -> language-description.
imp-type-integer =
   | syntax
   |    [ "integer" ] --> * Type
   | semantics
   |    give integer-type.

imp-type-boolean :: -> language-description.
imp-type-boolean =
   | syntax
   |    [ "boolean" ] --> * Type
   | semantics
   |    give boolean-type.
```

Podemos observar que ambas as operações semânticas, **imp-type-integer** e **imp-type-boolean**, produzem apenas os valores dos *sorts* **integer-type** e **boolean-type**, respectivamente.

A função semântica **allocate variable of _**, utilizada no componente semântico para declaração de variáveis, assim como **attribute _ to _** e **the value attributed to _** são funções semânticas exclusivas para a manipulação de variáveis, ou melhor, valores do *sort* **variable**. Na biblioteca de componentes semânticos imperativos definimos como *subsort* de **variable**, o *sort* **primitive-variable**, que também possui como *subsorts* **integer-variable** e **boolean-variable**.

Segue abaixo as definições das funções semânticas para manipulação de variáveis de tipos simples:

```
allocate variable of _ :: yielder -> action.
attribute _ to _ :: yielder, yielder -> action.
the value attributed to _ :: yielder -> value.
```

Tanto para valores inteiros e booleanos, a forma de alocação de memória é a mesma, já que se tratam de variáveis primitivas, diferentes apenas no valor default atribuídas às mesmas quando da sua criação. Sendo assim, segue abaixo a definição das funções semânticas para o tratamento de variáveis primitivas:

```
allocate variable of (~t : primitive-type) =
   | | give ~t
   | and then
   | | allocate a cell
   then
   | | give the primitive-variable of (the given cell # 2) (the given primitive-type # 1)
   | and
   | | store (the default-value of (the given primitive-type # 1)) in (the given cell # 2).

attribute (~v1 : primitive-value) to (~v2 : primitive-variable) =
   store ~v1 in (the cell of ~v2).

the value attributed to (~v : primitive-variable) =
```

```
the primitive-value stored in (the cell of ~v).
```

A partir da observação destas funções semânticas, podemos concluir que uma variável primitiva nada mais é do que uma tupla, formada por uma nova célula alocada (armazenando um valor default dependente do tipo), e pelo tipo primitivo fornecido como parâmetro da função. As demais funções atribuem um novo valor para a célula e consultam o valor armazenado na célula.

Também podemos concluir que as funções semânticas estão sendo redefinidas para que possam trabalhar com *sorts* distintos, ou seja, dependendo do tipo da variável, seja simples ou composta, seja inteira ou real, estas funções precisam ser redefinidas para que possam efetuar ações particulares ao tipo da variável.

4.1.2 Trabalhando com Variáveis

Além da declaração de variáveis, é necessário especificar os componentes semânticos para consulta/atualização de variáveis. Como uma variável, numa visão de mais alto nível, é uma associação de um identificador a uma célula de memória, tudo que temos a fazer é trabalhar com o respectivo identificador. De fato, se uma expressão pode consultar o valor de uma constante, associado a um identificador qualquer, porque não consultar também o valor de uma variável qualquer.

Sendo assim, especificamos a operação semântica **imp-expressions** que produz um componente semântico que estende o *namespace* **Expression**, permitindo a consulta de valores armazenados em variáveis sempre que for possível o uso de expressões:

```
imp-expressions :: -> language-description.
imp-expressions =
   | syntax
   |    [ i # * Ident ] --> * Expression
   | semantics
   |   | semantics of i
   |   then
   |   | give the value attributed to (the given variable).
```

Podemos observar, na seção **syntax**, a definição do novo *namespace* **Ident**, o qual subentendesse que irá produzir, quando da sua avaliação, valores do *sort* **variable**. Conseqüentemente, a avaliação semântica do componente irá gerar o valor atribuído a variável produzida pela avaliação semântica do componente do *namespace* **Ident**.

Para produzir o componente do *namespace* **Ident**, especificamos a operação semântica **imp-idents**, cuja avaliação semântica gera um valor do *sort* **variable** associado ao valor do elemento sintático **i # Identifier**:

```
imp-idents :: -> language-description.
imp-idents =
   | syntax
   |    [ i # Identifier ] --> * Ident
   | semantics
   |    give the variable bound to token of * i.
```

Para efetuar a atualização de valores armazenados em variáveis, é necessário especificar a semântica do comando de atribuição. Para tanto, especificamos a operação semântica **imp-commands** que produz, além do componente semântico para o comando de atribuição, os demais componentes semânticos para representação de comandos no paradigma imperativo (Seção 4.1.3). Entretanto, como a especificação desta operação semântica é um pouco extensa, iremos apresentá-la em partes, facilitando assim a explicação de cada componente semântico.

Portanto, segue abaixo o trecho da operação semântica **imp-commands**, mostrando a especificação do componente semântico responsável pela atualização de valores nas variáveis:

```
imp-commands :: -> language-description.
imp-commands =
   | syntax
   |    [ i # * Ident "=" e # * Expression ] --> * Command
   | semantics
   |    | | semantics of i
   |    | and
   |    | | semantics of e
   |    then
   |    | attribute (the given value # 2) to (the given variable # 1)
   and
   ...
```

Na seção **syntax**, vemos a definição de mais um novo *namespace* denominado **Command**, e na seção **semantics** temos: a avaliação do subcomponente **Ident**, produzindo um **variable**; a avaliação do subcomponente **Expression**, produzindo um **value**; e finalmente a atribuição do **value** no **variable** respectivamente, através do uso da função semântica **attribute _ to _**.

4.1.3 Comandos

Antes de continuarmos com o tratamento de comandos, é interessante que façamos uma breve explanação do que vem a ser um comando, e como este conceito será abordado nesta dissertação.

Um comando é uma frase do programa que irá ser executada com o objetivo de manipular variáveis. Podem ser simples ou compostos por comandos simples, e podem ser divididos em: nulo, atribuição, chamada de procedimentos, seqüencial, condicional e iterativo [10].

O primeiro comando já analisado foi o comando de atribuição, seguiremos analisando as especificações semânticas para os comandos: nulo, condicional, iterativo e composto. Chamada de procedimento será analisada na biblioteca imperativa complexa (Seção 4.2), e comandos seqüenciais não foram especificados.

O comando nulo como o próprio nome já diz não faz nada [10]. Serve para complementar estruturas sintáticas que exigem a presença de um comando, mesmo quando não é desejado na semântica do programa em si:

```
imp-commands =
  ...
  and
  | syntax
  |    [ ] --> * Command
  | semantics
  |   complete
  and
  ...
```

Podemos observar que, na seção **syntax**, o componente não espera nenhuma estrutura sintática; e na seção **semantics**, o componente executa apenas a ação **complete**, repassando o processamento da informação corrente para a próxima ação.

4.1.4 Comandos Condicionais

Comandos condicionais são formados por uma determinada quantidade de subcomandos, e entre os quais irá escolher quem será executado [10].

Entre os comandos condicionais [8], [9], [10], especificamos o comando **if** com e sem a condição **else**. Sem a condição **else**, temos um componente semântico cuja seção **syntax** é formada por um subcomponente **Expression** e um subcomponente **Command**; e cuja seção **semantics** consiste na avaliação semântica do subcomponente **Command**, quando o resultado da avaliação semântica do subcomponente **Expression** for igual ao valor booleano **true**. Com a condição **else**, teremos a mesma estrutura do componente semântico anterior, acrescentando: na seção **syntax** mais um subcomponente **Command**; e, na seção **semantics**, a avaliação semântica deste subcomponente extra, quando a avaliação semântica do subcomponente **Expression** resultar no valor booleano **false**:

```
imp-commands =
  ...
  and
  | syntax
  |    [ "if" "(" e # * Expression ")" "then" c # * Command ] --> * Command
  | semantics
  |   | semantics of e
  |   then
  |   | semantics of c
```

```
   and
   | syntax
   |    [ "if" "(" e # * Expression ")" "then" c1 # * Command "else" c2 # * Command ] --> *
Command
   | semantics
   |    | semantics of e
   |    then
   |    | | semantics of c1
   |    | else
   |    | | semantics of c2
   and
   ...
```

4.1.5 Comandos Iterativos

Um comando iterativo é formado por um subcomando que irá ser executado repetidamente, e usualmente algum tipo de condição determina quando a iteração irá terminar [10].

Entre os comandos iterativos [8], [9], [10], especificamos o comando **while**, cuja seção **syntax** é composta pelos subcomponentes **Expression** e **Command**. Na seção **semantics**, teremos uma avaliação recursiva, onde enquanto a avaliação semântica do subcomponente **Expression** resultar num valor booleano **true**, teremos a avaliação semântica do subcomponente **Command** seguido da execução da ação **unfold**. Esta última ação irá repetir o processo até que a avaliação semântica do subcomponente **Expression** resulte num valor booleano **false**, ocasionando a execução da ação **complete** e o fim do processo recursivo:

```
imp-commands =
   ...
   and
   | syntax
   |    [ "while" "(" e # * Expression ")" c # * Command ] --> * Command
   | semantics
   |    unfolding
   |    | | semantics of e
   |    | then
   |    | | | semantics of c
   |    | | and then
   |    | | | unfold
   |    | else
   |    | | complete
   and
   ...
```

4.1.6 Comandos Compostos

Além dos comandos simples, temos no paradigma imperativo os comandos compostos, os quais são compostos também por comandos compostos ou por comandos simples [8], [9], [10]. No componente especificado para representar comandos compostos temos, na seção

syntax, dois subcomponentes **Command**, e na seção **semantics** temos a avaliação semântica do primeiro subcomponente **Command** seguido da avaliação semântica do segundo subcomponente **Command**:

```
imp-commands =
   ...
   and
   | syntax
   |    [ c1 # * Command ";" c2 # * Command ] --> * Command
   | semantics
   |    | semantics of c1
   |    and then
   |    | semantics of c2
   and
   ...
```

4.1.7 Bloco de Comandos

Da mesma forma que na biblioteca de componentes semânticos para expressões (Seção 3.4), a biblioteca de componentes semânticos imperativos possui também o conceito de bloco, definindo uma área de escopo para o uso das declarações de variáveis dentro dos comandos [10].

A operação semântica **imp-block** é responsável pela produção do componente semântico associado ao conceito de bloco de comandos, cuja seção **syntax** é formada pelos subcomponentes **Declaration** e **Command**, os quais irão produzir as variáveis e os comandos que irão trabalhar com as novas variáveis locais a serem produzidas. Como um bloco de comandos é um comando, o subcomponente **Command** pode ser um novo bloco, representando assim o conceito de blocos aninhados, onde a declaração de uma variável local irá sobrepor qualquer declaração de variável global com o mesmo identificador.

Segue abaixo a operação semântica **imp-block**, que gera o componente semântico responsável pela construção de blocos de comandos:

```
imp-block :: -> language-description.
imp-block =
   | syntax
   |    [ d # * Declaration "begin" c # * Command "end" ] --> * Command
   | semantics
   |    | furthermore semantics of d
   |    hence
   |    | semantics of c.
```

Vale salientar que uma declaração de variáveis possui as mesmas regras de escopo e ciclo de vida de uma declaração de constante, definida na biblioteca de componentes semânticos de expressões (Seção 3.4.2), o que muda é apenas a forma de representação e persistência dos elementos declarados.

4.1.8 Agrupando Componentes Imperativos

Para finalizar a nossa biblioteca de componentes semânticos imperativos, criamos as operações semânticas **imp-local-program** e **imp-program**, as quais produzem a **language-description** formada pelo agrupamento de todos os componentes semânticos imperativos. A diferença é que a **imp-program** também agrupa os componentes semânticos produzidos pela operação semântica **exp-program** (Seção 3.4.4), os quais são necessários para efeitos de animação da especificação com a ferramenta ABACO [11]:

```
imp-program :: -> language-description.
imp-program =
   | exp-program
   and
   | imp-local-program.

imp-local-program :: -> language-description.
imp-local-program =
   | imp-idents
   and
   | imp-expressions
   and
   | imp-declarations
   and
   | imp-commands
   and
   | imp-block
   and
   | imp-type-integer
   and
   | imp-type-boolean.
```

Na compilação de programas imperativos, a operação semântica **compile _ _ _**, especificada na Seção 2.2.2, irá receber como argumentos: uma **language-description**, neste caso **imp-program**; um *namespace* (**Command**) definindo que tipo de componente semântico deve ser inicialmente processado; e a árvore sintática de comandos, representando o código do programa propriamente dito.

Exemplo 4.1: Loop com Incremento de Variável

Como exemplo, criamos um programa imperativo (**Figuras 4.1** e **4.2**) que: declara a variável **x** do tipo inteiro; atribui a mesma um valor inicial igual a **0**; e executa um *loop*, de modo que, enquanto a variável **x** for diferente de **3**, será efetuado o incremento da variável **x** pelo valor **1**, através do uso de um comando de atribuição.

```
begin
   var x : integer
   begin
      x = 0;

      while ( x <> 3 )
         x = x + 1
   end
end
```

Figura 4.1 – Linguagem concreta do programa Exemplo 4.1.

```
compile (imp-program) (* Command)
[
 [ "var" x ":" [ "integer" ] ]
 "begin"
 [
  [ [ x ] "=" [ 0 ] ]
  ";"
  [
   "while" "(" [ [ [ x ] ] [ "<>" ] [ 3 ] ] ")"
   [
    [ x ] "=" [ [ [ x ] ] [ "+" ] [ 1 ] ]
   ]
  ]
 ]
 "end"
]
```

Figura 4.2 – Linguagem abstrata do programa Exemplo 4.1 para uso da função semântica compile _ _ _.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 4.3** abaixo) que, quando executadas produzem como resultado final: uma associação **{x → cell0}**, e uma célula de memória **{cell0 → 3}**.

```
| furthermore
| | | give integer-type
| | then
| | | | allocate variable of  (the given type)
| | | then
| | | | bind "x" to  (the given variable)
hence
```

Figura 4.3 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.1.

```
| | | | give  (the variable bound to "x")
| | | and
| | | | give 0
| | then
| | | attribute  (the given value # 2)  to  (the given variable # 1)
| and then
| | unfolding
| | | | | | give "DIFF"
| | | | | and then
| | | | | | | | give  (the variable bound to "x")
| | | | | | | then
| | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | and then
| | | | | | | give 3
| | | | then
| | | | | binary-operation-of  (the given string # 1)   (the given value # 2)   (the given value # 3)
| | | then
| | | | | | | | give  (the variable bound to "x")
| | | | | | | and
| | | | | | | | | | give "PLUS"
| | | | | | | | | and then
| | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | then
| | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | and then
| | | | | | | | | | give 1
| | | | | | | | then
| | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)   (the given value # 3)
| | | | | | then
| | | | | | | attribute  (the given value # 2)  to  (the given variable # 1)
| | | | | and then
| | | | | | unfold
| | | | else
| | | | | complete
```

Figura 4.3 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.1.

Não serão produzidos valores transitórios, mas apenas associações e células de memória, uma vez que comandos trabalham apenas com variáveis. Os componentes especificados na

biblioteca de componentes semânticos imperativos não realizam liberação de células alocadas, por isso sempre teremos células de memória alocadas no resultado final da avaliação semântica, mesmo quando alocadas para ambientes locais, tais como: blocos, procedimentos e funções, e objetos, por exemplo.

□

4.2 Componentes Imperativos Complexos

O paradigma imperativo possui, além das suas características básicas (alocação de memória e execução de comandos, por exemplo), características complexas, tais como: abstração, tipos compostos, ponteiros, tipos dinâmicos, etc. [9]. Para representá-las, separamos os respectivos componentes semânticos em uma biblioteca de componentes extra, denominada de imperativa complexa. Esta separação de conceitos é interessante porque demonstra a reusabilidade dos componentes mais simples, previamente especificados para representar linguagens de expressões e imperativas (Seções 3.4 e 4.1, respectivamente), na representação de conceitos avançados do paradigma imperativo.

A estrutura da biblioteca de componentes imperativa complexa pode ser definida pela representação dos seguintes conceitos: declaração e chamada de procedimentos, definição de parâmetros atuais e formais, representação dos tipos compostos array e record, e manipulação de ponteiros e tipos dinâmicos, respectivamente. Portanto, vamos começar a descrever a biblioteca de componentes imperativos complexos com a declaração e chamada de procedimentos, mas antes de começarmos a explicar o que vem a ser um procedimento, precisamos entender o que é uma abstração.

4.2.1 Abstração e Procedimento

Abstração é um modo de pensamento pelo qual se concentra em idéias genéricas ao invés de manifestações específicas dessas idéias. Abstração é o princípio básico de filosofia e matemática, tendo um grande aproveitamento em muitas outras disciplinas, incluindo todos os ramos da ciência da computação [10].

Em análise de sistemas, abstração se concentra nos aspectos essenciais do problema que tem em mãos, ignorando todos os aspectos estranhos. Por exemplo, em um sistema de tráfego aéreo existem numerosos detalhes, tais como cores e marcas externas de cada aeronave, roupa de passageiros, etc., que são irrelevantes para o funcionamento do sistema. O analista deve se abstrair sempre desses detalhes e se concentrar apenas nos pontos essenciais do problema, como o tipo de cada aeronave ou os sinais de chamada de vôo, por exemplo [10].

Em programação, abstração faz alusão a distinção entre: o que um pedaço de programa faz e como isto é implementado. Uma linguagem de programação é formada por construtores que são abstrações de código de máquina. Todas as vezes que se faz uma referência a um procedimento está se fazendo referência à uma abstração. Quando se chama um procedimento, pode-se concentrar apenas no que o procedimento faz. Apenas o autor do procedimento é que deve se preocupar em como o procedimento é implementado [10].

Através da implementação de procedimentos de alto nível em termos de procedimentos de baixo nível, os programadores conseguem atingir vários níveis de abstração. Este tipo de hierarquia é uma ótima ferramenta para a construção de grandes programas [10].

Na biblioteca de componentes semânticos imperativos complexos, a declaração de procedimentos é especificada por dois componentes semânticos, ambos produzidos pela operação semântica **imp2-declarations**. Por ser uma declaração, ambos os procedimentos irão estender o *namespace* **Declaration**, já especificado em componentes semânticos anteriores (Seções 3.4.2 e 4.1.1):

```
imp2-declarations :: -> language-description.
imp2-declarations =
   | syntax
   |    [ "procedure" i # Identifier "()" c # * Command ] --> * Declaration
   | semantics
   |     recursively bind token of * i to procedure of closure abstraction of
   |     | semantics of c
   and
   | syntax
   |    [ "procedure" i # Identifier "(" fp # * FormalParameter ")" c # * Command ] --> *
Declaration
   | semantics
   |     recursively bind token of * i to procedure of closure abstraction of
   |     | | furthermore semantics of fp
   |     | hence
   |     | | semantics of c.
```

O primeiro componente semântico para declaração de procedimentos, na sua seção **syntax**, é formado por um subcomponente **Command** e um elemento sintático **i # Identifier**; e, na sua seção **semantics**, procura associar o novo procedimento ao valor do elemento sintático **i # Identifier**, cuja abstração procura avaliar a semântica do subcomponente **Command**.

O segundo componente semântico para declaração de procedimentos utiliza a mesma estrutura do componente semântico anterior, acrescentando, na sua seção **syntax**, o subcomponente **FormalParameter**; e, na sua seção **semantics**, a avaliação semântica deste novo subcomponente, gerando parâmetros formais, antes da avaliação semântica do subcomponente **Command**.

O conceito de parâmetros formais será tratado mais adiante, mas podemos adiantar que se trata da declaração de variáveis responsáveis pela manipulação dos valores passados como argumentos do procedimento [20].

4.2.2 Chamada de Procedimentos

Toda chamada de procedimento é um comando [10], e como tal procura atualizar variáveis locais e globais. A chamada de um procedimento ocasiona a execução de um procedimento qualquer, e pode ser acompanhada de argumentos ou não, denominados parâmetros atuais [10].

Portanto, segue abaixo a especificação da operação semântica **imp2-commands**, responsável pela produção de componentes semânticos que representam o conceito de chamada de procedimentos:

```
imp2-commands :: -> language-description.
imp2-commands =
   | syntax
   |    [ i # Identifier "(" ")" ] --> * Command
   | semantics
   |    | enact procedure-body of (the procedure bound to token of * i)
   and
   | syntax
   |    [ i # Identifier "(" ap # * ActualParameter ")" ] --> * Command
   | semantics
   |    | | semantics of ap
   |    | then
   |    | | enact application procedure-body of (the procedure bound to token of * i)
   |    | | to (the given data).
```

Podemos observar que o primeiro componente semântico, na seção **syntax**, é formado pelo elemento sintático **i # Identifier**; e, na sua seção **semantics**, procura executar a abstração do procedimento associado ao valor do elemento sintático **i # Identifier**. Já o segundo componente semântico utiliza a mesma estrutura do componente semântico anterior, acrescentando, na sua seção **syntax**, o subcomponente **ActualParameter**; e, na sua seção **semantics**, a avaliação semântica deste novo subcomponente antes da execução da abstração do procedimento associado ao valor do elemento sintático **i # Identifier**.

Vamos agora abordar o conceito de parâmetros atuais, os quais já foram ligeiramente citados na especificação de componentes para a chamada de procedimentos.

4.2.3 Parâmetros Atuais

Os parâmetros atuais são expressões passadas como parâmetros na chamada dos procedimentos, e a avaliação destas expressões geram argumentos a serem usados durante a execução do procedimento em si [10].

Os componentes semânticos que representam o conceito de parâmetro atual são produzidos pela operação semântica **imp2-actual-parameter**, cuja especificação segue abaixo:

```
imp2-actual-parameter :: -> language-description.
imp2-actual-parameter =
   | syntax
   |    [ i # * Ident ] --> * ActualParameter
   | semantics
   |    semantics of i
   and
   | syntax
   |    [ e # * Expression ] --> * ActualParameter
   | semantics
   |    semantics of e
   and
   | syntax
   |    [ ap1 # * ActualParameter "," ap2 # * ActualParameter ] --> * ActualParameter
   | semantics
   |    | | semantics of ap1
   |    | and
   |    | | semantics of ap2.
```

Podemos observar, nos dois primeiros componentes semânticos, nas seções **syntax**, a criação de um novo *namespace* denominado **ActualParameter**, e a presença dos subcomponentes **Ident** e **Expression** para cada um dos novos componentes semânticos; e, nas seções **semantics**, a avaliação de cada um dos subcomponentes, produzindo valores do *sort* **variable** e do *sort* **value**, respectivamente.

O terceiro componente semântico foi especificado apenas para permitir o processamento de mais de um parâmetro atual quando necessário.

4.2.4 Parâmetros Formais

Parâmetros formais foram ligeiramente citados na especificação de componentes semânticos para a declaração de procedimentos (Seção 4.2.1). Os parâmetros formais definem a forma de declaração das variáveis que irão manipular os argumentos passados na chamada do procedimento [10].

De fato, a passagem de parâmetros é um conceito largamente estudado em paradigmas de linguagens, definindo o relacionamento entre o parâmetro formal declarado e o argumento produzido pela avaliação do parâmetro atual na chamada de uma função. Por exemplo, se for declarado um parâmetro formal com uma passagem de parâmetros por *valor*, no mecanismo

de declaração [10], somente serão aceitos parâmetros atuais que produzam valores durante a sua avaliação. Outro exemplo, se for declarado um parâmetro formal com uma passagem de parâmetros por *referência*, no mecanismo de declaração [10], somente serão aceitos parâmetros atuais que produzam variáveis durante a sua avaliação.

Existem outros tipos de passagem de parâmetros, mas os dois exemplos acima mostrados são os mais utilizados pelas linguagens de programação imperativas, portanto especificamos componentes semânticos apenas para esses dois tipos de passagens de parâmetros, os quais serão produzidos pela operação semântica **imp2-formal-parameter**, cuja especificação segue abaixo:

```
imp2-formal-parameter :: -> language-description.
imp2-formal-parameter =
   | syntax
   |    [ i # Identifier ":" t # * Type ] --> * FormalParameter
   | semantics
   |    | | give first them
   |    | then
   |    | | | | | semantics of t
   |    | | | | then
   |    | | | | | allocate variable of (the given type # 1)
   |    | | | then
   |    | | | | | bind token of * i to the given variable # 1
   |    | | before
   |    | | | attribute the given value # 1 to (the variable bound to token of * i)
   |    and then
   |    | give rest them
   and
   | syntax
   |    [ "var" i # Identifier ":" t # * Type ] --> * FormalParameter
   | semantics
   |    | | give first them
   |    | then
   |    | | bind token of * i to the given variable # 1
   |    and then
   |    | give rest them
   and
   | syntax
   |    [ fp1 # * FormalParameter ";" fp2 # * FormalParameter ] --> * FormalParameter
   | semantics
   |    | semantics of fp1
   |    then
   |    | semantics of fp2.
```

O primeiro componente semântico especifica a passagem de parâmetros por *valor*, onde, na seção **syntax**, são fornecidos um elemento sintático **i # Identifier** e um subcomponente **Type**; e, na seção **semantics**, ocorre a alocação de uma nova variável, seguida da atribuição de um valor, passado como argumento na chamada do procedimento, para esta nova variável alocada.

O segundo componente semântico especifica a passagem de parâmetros por *referência*, onde, na seção **syntax**, são fornecidos um elemento sintático **i # Identifier** e um subcomponente

Type; e, na seção **semantics**, ocorre apenas a associação do elemento sintático **i # Identifier** à variável passada como argumento na chamada do procedimento.

Podemos observar também, nos dois primeiros componentes semânticos, nas seções **syntax** dos mesmos, a criação de um novo *namespace* denominado **FormalParameter**; e, nas seções **semantics**, o consumo do primeiro argumento durante a avaliação semântica de cada componente semântico, e após este processo, a propagação dos demais argumentos para serem novamente consumidos pelo próximo parâmetro formal, no caso de existir mais de um.

O terceiro componente semântico foi especificado apenas para permitir o processamento de mais de um parâmetro formal quando necessário.

Com isso terminamos a parte de abstrações e seguiremos agora para a parte de tipos compostos. O tema abstrações (Seção 4.2.1) será novamente abordado na biblioteca de componentes semânticos funcionais (Seção 4.3), quando falarmos sobre funções.

4.2.5 Tipos Compostos

Um tipo composto (ou tipo de dado estruturado) é um tipo cujos valores são compostos ou estruturados por valores simples [10]. Linguagens de programação suportam uma grande variedade de tipos de dados: tuplas, records, variantes, uniões, arrays, conjuntos, strings, listas, árvores, arquivos, relações, etc. A variedade as vezes assusta, mas de fato esses tipos podem ser entendidos em termos de um pequeno número de conceitos de estruturas. Estes conceitos são: produtos cartesianos (tuplas e records), uniões disjuntas (variantes e uniões), mapeamento (arrays e funções), powersets (conjuntos) e tipos recursivos (estruturas dinâmicas de dados), os quais são diretamente representados em algumas notações semânticas [10].

De fato, semântica de ações traz em sua notação de dados a capacidade de representar essas estruturas [1], [2], no entanto, tipos compostos são formados por tipos simples, os quais já foram especificados previamente na biblioteca de componentes semânticos imperativos (Seção 4.1), portanto temos que nos ater para o processo de manipulação destes tipos simples, uma vez que os mesmos processos serão ligeiramente diferentes para tipos compostos.

4.2.6 Arrays

O primeiro tipo composto a ser especificado é o *array*, o qual pode ser definido como uma estrutura de dados constituída de pares (índice, valor), cujas operações básicas são armazenamento e recuperação de valores [9], [22].

Para que se possa efetuar uma declaração de variável *array*, é necessário estender o *namespace* **Type** com a especificação de um novo componente semântico para representar o

tipo *array*. Sendo assim, especificamos a operação semântica **imp2-array-types**, que produz o componente semântico responsável pela geração de valores do *sort* **array-type** (*subsort* de **type**):

```
imp2-array-types :: -> language-description.
imp2-array-types =
   | syntax
   |    [ "array" "[" e # * Expression "]" "of" t # * Type ] --> * Type
   | semantics
   |    | | semantics of e
   |    | and
   |    | | semantics of t
   |    then
   |    | give the array-type of (the given type # 2) (the given integer-value # 1).
```

Podemos observar que, na seção **syntax**, é composta pelos subcomponentes **Expression** e **Type**; e, na seção **semantics**, efetua-se a avaliação semântica dos respectivos subcomponentes, produzindo: o número total de índices do *array* e o tipo das variáveis que formarão o *array*. No final da seção **semantics**, temos a produção de um valor do *sort* **array-type**, composto pelos valores dos *sorts* **integer-value** e **type**, produzidos pelos respectivos subcomponentes.

Como um *array* é uma variável, vamos precisar redefinir as funções semânticas utilizadas na manipulação de variáveis (**allocate variable of_**, **the value attributed to _** e **attribute _ to _**), as quais serão mostradas abaixo:

```
allocate variable of (~t : array-type) =
   | | | give ~t
   | | then
   | | | | regive
   | | | and
   | | | | | maketuple  (the component-type of the given type) (the index-limit of the given type)
   | | | | then
   | | | | | respectively allocate variables of them
   | then
   | | give the array-variable of (rest them) (first them).

the value attributed to (~a : array-variable) =
   the array-value of (the values respectively attributed to (the components of ~a)).

attribute (~a1 : array-value) to (~a2 : array-variable) =
   respectively attribute (the components of ~a1) to (the components of ~a2).
```

Podemos observar que a função semântica de alocação de variável, após a criação de uma tupla de tipos, produzida pela função semântica **maketuple _ _**, que recebe o tipo e o número de elementos do valor do *sort* **array-type**, ocorre a criação de uma tupla de variáveis. Essa tupla de variáveis é produzida pela função semântica **respectively allocate variables of _**, a qual posteriormente entraremos em maiores detalhes. No final, é produzido um valor do *sort* **array-variable**, utilizando as informações transitórias: tupla de variáveis e **array-type** (valor fornecido como argumento da função semântica).

Na consulta ao valor de um *array*, podemos observar, como resultado final, um valor do *sort* **array-value**, produzido a partir da tupla de valores gerada pela função semântica **the values respectively attributed to _**, a qual posteriormente entraremos em maiores detalhes.

E finalmente, na atribuição de um valor do *sort* **array-value** num valor do *sort* **array-variable**, podemos observar o uso da função **respectively attribute _ to _**, a qual posteriormente entraremos em maiores detalhes, recebendo como argumentos as tuplas de valores e variáveis que fazem parte dos valores dos *sorts* **array-value** e **array-variable**, respectivamente.

Para controlar o acesso a variáveis *array*, foi especificado um componente semântico, produzido pela operação semântica **imp2-array-idents**, que estende o *namespace* **Ident**, garantindo assim um acesso comum tanto para variáveis simples (Seção 4.1.2) como para variáveis *array*:

```
imp2-array-idents :: -> language-description.
imp2-array-idents =
   | syntax
   |    [ i # * Ident "[" e # * Expression "]" ] --> * Ident
   | semantics
   |    | | semantics of i
   |    | and
   |    | | semantics of e
   |    then
   |    | give the component indexed by
   |    | (the given integer-value # 2) in (the given array-variable # 1).
```

Podemos observar que o componente semântico, na seção **syntax**, é formado pelos subcomponentes **Ident** e **Expression**. Na seção **semantics**, temos: a avaliação semântica do subcomponente **Ident** produzindo um valor do *sort* **array-variable**; a avaliação semântica do subcomponente **Expression** produzindo um índice **N** que indicará o elemento do *array* a ser consultado; e finalmente a produção da variável desejada, contida no **array-variable** na posição **N**, produzida pela função semântica **the component indexed by _ in _**.

Por uma questão de conveniência, as principais especificações imperativas, usando semântica de ações [12], [13], representam *arrays* usando tuplas ao invés de mapeamentos para índices ordinais [10]. De certa forma, a notação de dados para tuplas possui a ação **component # _** que retorna o elemento da tupla na posição desejada, tornando-se desnecessário a utilização de mapeamentos. A ação **component # _** é utilizada pela função semântica **the component indexed by _ in _** para acessar posições específicas de uma variável *array*.

Com isso apresentamos as principais especificações semânticas para o tipo composto *array*. Vamos agora mostrar as funções semânticas responsáveis pela manipulação de memória de variáveis compostas, as quais foram citadas durante a especificação semântica de *array*, mas não foram detalhadas.

4.2.7 Trabalhando com Variáveis Compostas

Basicamente, existem três funções semânticas que efetuam a manipulação de variáveis compostas: **respectively allocate variables of _**, **respectively attribute _ to _** e **the values respectively attributed to _**. O principal objetivo da especificação destas funções semânticas se encontra na manipulação de alocação, consulta e atualização de tuplas de variáveis.

Na função semântica **respectively allocate variables of _**, quando o argumento da função semântica é uma tupla, teremos uma avaliação recursiva, de modo que: será feita a alocação de uma variável simples para o primeiro elemento da tupla, usando a função semântica **allocate variable of _** (Seção 4.1.1); e, em seguida, a alocação do resto dos elementos da tupla usando a própria função semântica **respectively allocate variables of _**:

```
respectively allocate variables of _ :: yielder -> action.
respectively allocate variables of () = give ().
respectively allocate variables of (~t : type) = allocate variable of ~t.
respectively allocate variables of (~t : tuple) =
   | allocate variable of (first ~t)
   and
   | respectively allocate variables of (rest ~t).
```

Na função semântica **respectively attribute _ to _**, quando os argumentos da função semântica são tuplas de valores e variáveis, teremos uma avaliação recursiva, de modo que: será feita a atribuição de um valor (do primeiro elemento da tupla de valores) para uma variável simples (do primeiro elemento da tupla de variáveis), usando a função semântica **attribute _ to _** (Seção 4.1.1); e, em seguida, a atribuição do resto dos elementos da tupla de valores para o resto dos elementos da tupla de variáveis, usando a própria função semântica **respectively attribute _ to _**:

```
respectively attribute _ to _ :: yielder, yielder -> action.
respectively attribute (~v1 : value) to (~v2 : variable) =
   | check both (~v1 is (), ~v2 is ())
   or
   | attribute ~v1 to ~v2.
respectively attribute (~t1 : tuple) to (~t2 : tuple) =
   | check both (~t1 is (), ~t2 is ())
   or
   | | attribute (first ~t1) to (first ~t2)
   | and
   | | respectively attribute (rest ~t1) to (rest ~t2).
```

Na função semântica **the values respectively attributed to _**, quando o argumento da função semântica é uma tupla, teremos uma avaliação recursiva, de modo que: será feita a consulta do valor de uma variável simples para o primeiro elemento da tupla, usando a função semântica **the value attributed to _** (Seção 4.1.1); e, em seguida, a consulta aos valores do resto

dos elementos da tupla usando a própria função semântica **the values respectively attributed to _**:

```
the values respectively attributed to _ :: yielder -> yielder.
the values respectively attributed to () = complete.
the values respectively attributed to (~v : variable) = the value attributed to ~v.
the values respectively attributed to (~t : tuple) =
   (the value attributed to (first ~t), the values respectively attributed to (rest ~t)).
```

4.2.8 Records

Iremos agora tratar o tipo composto *record*, o qual pode ser definido como uma estrutura de dados constituída de elementos agregados heterogêneos. Cada elemento de um *record* geralmente é denominado de campo, e, antes de começarmos a mostrar especificações semânticas para *records*, iremos apresentar o conceito de campo (*field*) [9], [22].

Na biblioteca de componentes semânticos, um campo é uma tupla, composta pelo nome do campo e por um componente de campo qualquer, desde que seja um *subsort* de **value**, **type** ou **variable**. As equações semânticas abaixo mostram a especificação semântica para o conceito de campo:

```
field-component >= value.
field-component >= type.
field-component >= variable.
the field of _ _ :: string, field-component -> field.
```

A principal operação semântica para manipulação de campos é **the field-identified by _ in _**, a qual recebe como argumentos o nome do campo desejado, e a tupla de campos a ser pesquisada. Como resultado ela produz o primeiro campo que encontrar na tupla de campos cujo nome seja igual ao nome pesquisado:

```
the field-identified by _ in _ :: yielder, yielder -> action.
the field-identified by ~i in () = give ().
the field-identified by ~i in (~f : field) =
   | check (~i is the identifier of ~f)
   | then
   | | | give ~f
   or
   | check not (~i is the identifier of ~f)
   | then
   | | | give ().
the field-identified by ~i in (~t : tuple) =
   | check (~i is the identifier of (first ~t))
   | then
   | | give (first ~t)
   or
   | check not (~i is the identifier of (first ~t))
   | then
   | | the field-identified by ~i in (rest ~t).
```

A utilização do conceito de campos na especificação semântica de *records* é feita pelos *sorts*: **record-type**, **record-value** e **record-variable**, os quais são formados por campos ou por tuplas de campos. De fato, podemos observar isso nos componente semânticos produzidos pela operação semântica **imp2-record-types**, os quais produzem um valor do *sort* **record-type** (*subsort* de **type**), necessário para a declaração de variáveis *record*:

```
imp2-record-types :: -> language-description.
imp2-record-types =
   | syntax
   |    [ "record" frt # * FieldRecordType "end" ] --> * Type
   | semantics
   |    | semantics of frt
   |    then
   |    | give the record-type of them
   and
   | syntax
   |    [ i # Identifier ":" t # * Type  ] --> * FieldRecordType
   | semantics
   |    | semantics of t
   |    then
   |    | give the field of token of * i (the given type)
   and
   | syntax
   |    [ frt1 # * FieldRecordType ";" frt2 # * FieldRecordType  ] --> * FieldRecordType
   | semantics
   |    | | semantics of frt1
   |    | and
   |    | | semantics of frt2.
```

No primeiro componente semântico, podemos observar a produção de um valor do *sort* **record-type**, partindo da tupla de campos produzida pela avaliação semântica do subcomponente **FieldRecordType**. No segundo componente semântico, o qual estende o *namespace* **FieldRecordType**, temos a produção de um campo, cujo nome é o valor do elemento sintático **i # Identifier**, e cujo componente do campo é o valor do *sort* **type** produzido pela avaliação semântica do subcomponente **Type**. O terceiro componente semântico estende o *namespace* **FieldRecordType**, permitindo a produção de tuplas de campos.

A utilização de campos também pode ser observada nas funções semânticas para manipulação de variáveis, tanto para o *sort* **record-variable** como para o *sort* **record-value**:

```
allocate variable of (~t : record-type) =
   | | | give ~t
   | | and then
   | | | respectively allocate variables of (the field-components of (the fixed-part of ~t))
   | then
   | | give the record-variable of
   | | (the fields of (the identifiers of the fixed-part of ~t) (rest them)) (first them).

attribute (~r1 : record-value) to (~r2 : record-variable) =
   respectively attribute
   (the field-components of the fixed-part of ~r1) to
   (the field-components of the fixed-part of ~r2).
```

```
the value attributed to (~r : record-variable) =
   the record-value of the fields of (the identifiers of the fixed-part of ~r)
   (the values respectively attributed to (the field-components of (the fixed-part of ~r))).
```

Podemos observar que a função semântica de alocação de variável (Seção 4.1.1), após a declaração das variáveis usando como tipos base os componentes dos campos do valor do *sort* **record-type**, produz um valor do *sort* **record-variable** utilizando: uma tupla de campos, formada pela tupla de identificadores dos campos de **record-type** e pela tupla de variáveis declaradas previamente pela função semântica **respectively allocate variables of _** (Seção 4.2.7); e o valor do *sort* **record-type**, recebido como argumento da função semântica.

Na atribuição de um valor do *sort* **record-value** num valor do *sort* **record-variable**, podemos observar o uso da função **respectively attribute _ to _** (Seção 4.2.7), recebendo como argumentos as tuplas de valores e variáveis produzidas pelos componentes dos campos do valor dos *sorts* **record-value** e **record-variable**, respectivamente.

E finalmente, na consulta ao valor de um *record*, podemos observar, como resultado final, um valor do *sort* **record-value**, produzido a partir da tupla de campos, formada pela tupla de identificadores do valor do *sort* **record-variable**, e pela tupla de valores, gerada pela função semântica **the values respectively attributed to _** (Seção 4.2.7) recebendo como argumento a tupla de componentes de campos do valor do *sort* **record-variable**.

Para controlar o acesso a variáveis *records*, foi especificado um componente semântico, produzido pela operação semântica **imp2-record-idents**, que estende o *namespace* **Ident**, garantindo assim um acesso comum tanto para variáveis simples como para variáveis *record*:

```
imp2-record-idents :: -> language-description.
imp2-record-idents =
   | syntax
   |    [ i1 # * Ident "." i2 # Identifier ] --> * Ident
   | semantics
   |    | semantics of i1
   |    then
   |    | the component-designated by token of * i2 in (the given record-variable).
```

Podemos observar que o componente semântico, na seção **syntax**, é formado pelo subcomponente **Ident** e pelo elemento sintático **i2 # Identifier**. Na seção **semantics**, temos: a avaliação semântica do subcomponente **Ident** produzindo um valor do *sort* **record-variable**; seguido da produção da variável desejada, contida no **record-variable** e identificada pelo elemento sintático **i2 # Identifier**, através do uso da função semântica **the component-designated by _**, a qual utiliza a função semântica **the field-identified by _ in _** especificada previamente no início desta Seção.

Com isso apresentamos as principais especificações semânticas para o tipo composto *record*. Vamos tratar agora de um conceito muito poderoso e ao mesmo tempo perigoso nas linguagens imperativas, os ponteiros.

4.2.9 Ponteiros

O conceito de endereço de memória é um conceito de baixo-nível e difere de um computador para outro. Uma abstração de alto-nível para este conceito se chama ponteiro. Na realidade, um ponteiro não é um tipo de dados, mas sim um construtor de tipos. Cada tipo ponteiro é ligado a outro tipo de dados chamado de tipo base e o conjunto de valores de um tipo ponteiro é o conjunto de endereços de memória de objetos do tipo base [30], [31], [32].

Na biblioteca de componentes semânticos imperativos complexos, foram especificadas três operações semânticas para a produção de componentes semânticos associados ao conceitos de ponteiros: **imp2-pointer-expressions**, **imp2-pointer-idents** e **imp2-pointer-types**.

Portanto, segue abaixo a especificação da operação semântica **imp2-pointer-expressions**, cujos componentes semânticos estendem o *namespace* **Expression**, e abordam conceitos como: valor nulo, endereço de uma variável e alocação dinâmica de variáveis, nessa mesma ordem:

```
imp2-pointer-expressions :: -> language-description.
imp2-pointer-expressions =
   | syntax
   |    [ "null" ] --> * Expression
   | semantics
   |    give null-pointer
   and
   | syntax
   |    [ "@" i # * Ident ] --> * Expression
   | semantics
   |    | semantics of i
   |    then
   |    | give pointer to the given variable
   and
   | syntax
   |    [ "new" t # * Type ] --> * Expression
   | semantics
   |    | semantics of t
   |    then
   |    | | allocate variable of (the given type)
   |    | then
   |    | | give pointer to (the given variable).
```

O primeiro componente semântico, na seção **syntax**, é formado apenas pelo elemento sintático "**null**"; e, na seção **semantics**, produz um valor do *sort* **pointer-value**, neste caso **null-pointer**.

O segundo componente semântico, na seção **syntax**, possui um subcomponente **Ident**; e, na seção **semantics**, produz um valor do *sort* **pointer-value** para a variável produzida pela avaliação semântica do subcomponente **Ident**.

O terceiro componente semântico, na seção **syntax**, possui um subcomponente **Type**; e, na seção **semantics**, aloca uma variável usando o valor do *sort* **type** produzido pela avaliação semântica do subcomponente **Type**, em seguida produz um valor do *sort* **pointer-value** para a nova variável alocada.

Na operação semântica **imp2-pointer-idents**, é produzido apenas um componente semântico, o qual estende o *namespace* **Ident**, que trata da dereferência de uma variável ponteiro, ou seja, da obtenção de uma variável apontada por um ponteiro [31]:

```
imp2-pointer-idents :: -> language-description.
imp2-pointer-idents =
   | syntax
   |    [ "*" i # * Ident ] --> * Ident
   | semantics
   |    | semantics of i
   |    then
   |    | give target-variable of (the value attributed to (the given pointer-variable)).
```

Podemos observar que o componente semântico, na seção **syntax**, possui um subcomponente **Ident**; e na seção **semantics**, é produzido um valor do *sort* **variable**, apontado pelo valor do *sort* **pointer-value** atribuído ao valor do *sort* **pointer-variable**, gerado na avaliação semântica do subcomponente **Ident**.

E para finalizar o tratamento de ponteiros, vamos analisar a operação semântica **imp2-pointer-types**, a qual produz o componente semântico responsável pela produção do tipo ponteiro:

```
imp2-pointer-types :: -> language-description.
imp2-pointer-types =
   | syntax
   |    [ "*" t # * Type ] --> * Type
   | semantics
   |    | semantics of t
   |    then
   |    | give pointer-type of (the given type).
```

De fato, na seção **syntax**, temos o subcomponente **Type**; e, na seção **semantics**, temos a produção de um valor do *sort* **pointer-type** a partir de um valor do *sort* **type**, gerado pela avaliação semântica do subcomponente **Type**.

Vamos agora falar um pouco sobre tipos dinâmicos e explicar porque os mesmos não foram especificados nesta dissertação.

4.2.10 Tipos Dinâmicos

Há muitas situações computacionais nas quais o tamanho exato ou a organização de uma coleção de dados não pode ser conhecido antecipadamente, uma vez que ela pode variar consideravelmente de uma execução de programa para outra. Em tais situações, *arrays* e *records* não têm flexibilidade suficiente para serem utilizados devido ao fato de suas estruturas e tamanhos serem pré-fixados. Entretanto, se for relaxada a exigência de que os elementos estejam em posições consecutivas de memória, é possível permitir o crescimento de uma estrutura sempre que houver espaço disponível em memória. Estas estruturas seriam denominadas de dinâmicas [30], [31], [32].

Para representar estruturas dinâmicas, se faz necessária a presença de dois conceitos imperativos: ponteiros, que já foi especificado na Seção 4.2.9; e declaração de tipos recursivos, que consiste na declaração de um tipo composto por ele mesmo. Infelizmente, devido a limitações do ABACO [11], o qual não suporta a animação da faceta diretiva em semântica de ações [1], [2], não pudemos especificar componentes semânticos para tipos recursivos, mas sim, apenas componentes semânticos para a declaração de novos tipos.

De fato, os componentes semânticos para declaração e uso de novos tipos são produzidos pelas operações semânticas **imp2-type-declarations** e **imp2-type-idents**, respectivamente:

```
imp2-type-declarations :: -> language-description.
imp2-type-declarations =
   | syntax
   |    [ i # Identifier "=" t # * Type ] --> * Declaration
   | semantics
   |    | semantics of t
   |    then
   |    | bind token of * i to the given type.


imp2-type-idents :: -> language-description.
imp2-type-idents =
   | syntax
   |    [ i # Identifier ] --> * Type
   | semantics
   |    give the type bound to token of * i.
```

Podemos observar que, o componente semântico produzido pela operação semântica **imp2-type-declarations**, na seção **syntax**, possui um elemento sintático **i # Identifier** e um subcomponente **Type**; e, na seção **semantics**, associa o valor do *sort* **type**, produzido na avaliação semântica do subcomponente **Type**, com o valor do elemento sintático **i # Identifier**.

Já o componente semântico produzido pela operação semântica **imp2-type-idents**, na seção **syntax**, possui apenas o elemento sintático **i # Identifier**; e, na seção **semantics**, produz o valor do *sort* **type** associado ao valor do elemento sintático **i # Identifier**.

4.2.11 Agrupando Componentes Imperativos Complexos

Para finalizar a nossa biblioteca de componentes semânticos imperativos complexos, criamos as operações semânticas **imp2-local-program** e **imp2-program**, as quais produzem a **language-description** formada pelo agrupamento de todos os componentes semânticos da biblioteca imperativa complexa. A diferença é que a **imp2-program** também agrupa os componentes semânticos produzidos pela operação semântica **imp-program** (Seção 4.1), os quais são necessários para efeitos de animação da especificação com a ferramenta ABACO [11]:

```
imp2-program :: -> language-description.
imp2-program =
   | imp-program
   and
   | imp2-local-program.

imp2-local-program :: -> language-description.
imp2-local-program =
   | imp2-declarations
   and
   | imp2-commands
   and
   | imp2-array-types
   and
   | imp2-array-idents
   and
   | imp2-record-types
   and
   | imp2-record-idents
   and
   | imp2-actual-parameter
   and
   | imp2-formal-parameter
   and
   | imp2-type-declarations
   and
   | imp2-type-idents
   and
   | imp2-pointer-expressions
   and
   | imp2-pointer-idents
   and
   | imp2-pointer-types.
```

Na compilação de programas imperativos complexos, a operação semântica **compile** (Seção 2.2.2) irá receber como argumentos: uma **language-description**, neste caso **imp2-program**; um *namespace* (**Command**) definindo que tipo de componente semântico deve ser

inicialmente processado; e a árvore sintática de comandos, representando o código do programa propriamente dito.

Exemplo 4.2: Atualização/Consulta de Records e Subrecords

Como exemplo, criamos um programa baseado nas estruturas imperativas complexas (**Figuras 4.4** e **4.5**), que declara uma variável **y** do tipo inteiro e uma variável **x** do tipo *record*, contendo: uma variável **a** do tipo inteiro e uma variável **b** novamente do tipo *record*, contendo também uma variável **a** do tipo inteiro. O programa tem basicamente dois comandos: o primeiro atribui ao campo **a** da variável **x** o valor inteiro **5**, e o segundo comando atribui a variável **y** o valor do campo **a** da variável **x**.

```
begin
   var x : record
             a : integer;
             b : record
                   a : integer
                 end
         end;

   var y : integer

   begin
      x.a = 5;
      y = x.a
   end
end
```

Figura 4.4 – Linguagem concreta do programa Exemplo 4.2.

Trata-se de um programa que testa o sistema de atualização/consulta de records, o controle de escopo das variáveis *record*, e a declaração de *records* compostos por *subrecords*.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 4.6** a seguir) que, quando executadas produzem como resultado final: as associações **{"x" → the record-variable of …, "y" → the primitive-variable of cell2 integer-type}**, e as células de memória **{cell0 → 5, cell1 → 0, cell2 → 5}**.

```
compile (imp2-program) (* Command)
[
  [
    [ "var" x ":"
      [ "record"
        [
          [ a ":" [ "integer" ] ]
          ","
          [ b ":" [ "record" [ a ":" [ "integer" ] ]  "end" ] ]
        ]
        "end"
      ]
   ]
    ","
    [ "var" y ":" [ "integer" ] ]
  ]
  "begin"
  [
    [ [ [ x ] "." a ] "=" [ 5 ] ]
    ","
    [ [ y ] "=" [ [ [ x ] "." a ] ] ]
  ]
  "end"
]
```

Figura 4.5 – Linguagem abstrata do programa Exemplo 4.2 para uso da função semântica compile _ _ _.

```
| furthermore
| | | | | | give integer-type
| | | | | then
| | | | | | give  (the field of "a"  (the given type) )
| | | | and
| | | | | | | | give integer-type
| | | | | | | then
| | | | | | | | give  (the field of "a"  (the given type) )
| | | | | | then
| | | | | | | give  (the record-type of  (the given data) )
| | | | | then
| | | | | | give  (the field of "b"  (the given type) )
| | | then
| | | | give  (the record-type of  (the given data) )
| | then
| | | | allocate variable of  (the given type)
| | | then
| | | | bind "x" to  (the given variable)
| | before
| | | give integer-type
| | then
| | | | allocate variable of  (the given type)
| | | then
| | | | bind "y" to  (the given variable)
hence
```

Figura 4.6 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.2.

```
| | | | | give  (the variable bound to "x")
| | | | then
| | | | | | the field-identified by "a" in  (the fixed-part of  (the given record-variable) )
| | | | | then
| | | | | | give  (the field-component of  (the given datum) )
| | | and
| | | | give 5
| | then
| | | attribute  (the given value # 2)  to  (the given variable # 1)
| and then
| | | | give  (the variable bound to "y")
| | | and
| | | | | | give  (the variable bound to "x")
| | | | | then
| | | | | | | the field-identified by "a" in  (the fixed-part of  (the given record-variable) )
| | | | | | then
| | | | | | | give  (the field-component of  (the given datum) )
| | | | then
| | | | | give  (the value attributed to  (the given variable) )
| | then
| | | attribute  (the given value # 2)  to  (the given variable # 1)
```

Figura 4.6 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.2.

Vale lembrar que, por ser um programa imperativo, não teremos valores transitórios, uma vez que comandos trabalham apenas com variáveis, neste caso associações e células de memória.

□

4.3 Componentes Funcionais

O paradigma funcional surgiu com o desenvolvimento da linguagem Lisp [28] (List Processing) por John McCarthy em 1958. Lisp foi projetada (numa época em que só existia processamento numérico) para atender aos interesses dos grupos de Inteligência Artificial no processamento de dados simbólicos. Conseqüentemente, surgiu como uma nova base para o desenvolvimento de sistemas, através do uso de funções matemáticas e composição de funções, introduzindo assim um novo modelo para representação de problemas a serem resolvido pela máquina.

Um outro bom exemplo de linguagem funcional é a linguagem de programação Haskell [29], originalmente desenvolvida em 1987. Ela foi desenvolvida com o intuito de se criar uma linguagem puramente funcional, com avaliação lazy, baseada em lambda calculus e polimorficamente tipada.

Segundo o paradigma funcional, programar significa definir funções, aplicar funções e conhecer o comportamento de funções na máquina; os mecanismos de controle, no programa, passam de iterativos a recursivos. Assim, representar a solução de um problema para ser resolvido num ambiente funcional passa a necessitar de uma abordagem completamente diferente dos métodos usados em linguagens imperativas [8], [9], [10].

De fato, o paradigma funcional não usa o conceito de atribuição, uma vez que todo o programa é composto por definições de funções. Uma função pode chamar outra e o resultado de uma pode ser usado como o argumento de outra. Mas não é essa omissão de recursos, como a atribuição, que torna esse paradigma mais ou menos poderoso. O que torna o paradigma funcional bastante atraente é a facilidade de modularidade, reusabilidade e extensibilidade. Além disso não existe a possibilidade de uma chamada a uma função causar efeitos colaterais uma vez que tudo o que ela pode fazer é retornar o seu valor. Isso torna o programa mais fácil de ser entendido evitando erros que de outra forma poderia ser bastante difíceis de serem detectados e corrigidos [8], [9], [10].

Portanto, vamos começar com a especificação de uma biblioteca de componentes semânticos funcionais, a partir da definição de conceitos básicos do paradigma funcional, tais como: declaração de variáveis e funções, chamada de funções, parâmetros atuais e formais, e expressões-if.

4.3.1 Declarações Funcionais

Na biblioteca de componentes semânticos funcionais, a declaração de variáveis e funções é especificada por dois componentes semânticos, ambos produzidos pela operação semântica **fnc-declarations**. Por ser uma declaração, ambos os componentes irão estender o *namespace* **Declaration**, especificado na biblioteca de componentes semânticos para expressões (Seção 3.4.2):

```
fnc-declarations :: -> language-description.
fnc-declarations =
   | syntax
   |    [ "val" i # Identifier "==" e # * Expression ] --> * Declaration
   | semantics
   |    | semantics of e
   |    then
   |    | bind token of * i to the given value
   and
   | syntax
   |    [ "val" "rec" i # Identifier "==" "fn" fp # * FormalParameter "==>" e # * Expression ] --> *
Declaration
   | semantics
   |    recursively bind token of * i to function of closure abstraction of
   |    | | furthermore semantics of fp
   |    | hence
   |    | | semantics of e.
```

O primeiro componente semântico para declaração de variáveis, na sua seção **syntax**, é formado por um subcomponente **Expression** e um elemento sintático **i # Identifier**; e, na sua seção **semantics**, procura associar o valor do *sort* **value**, produzido pela avaliação semântica do subcomponente **Expression**, ao valor do elemento sintático **i # Identifier**. Se compararmos o componente semântico para declaração de variáveis funcionais com o componente semântico para declaração de constantes (Seção 3.4.2), veremos que o comportamento semântico é o mesmo, ou seja, ambos procuram associar um valor produzido por uma expressão a um identificador qualquer.

O segundo componente semântico para declaração de funções, na sua seção **syntax**, é formado pelos subcomponentes **FormalParameter** e **Expression**, e um elemento sintático **i # Identifier**; e, na sua seção **semantics**, procura associar a nova função ao valor do elemento sintático **i # Identifier**, cuja abstração procura avaliar a semântica do subcomponente **FormalParameter**, gerando parâmetros formais, seguida da avaliação da semântica do subcomponente **Expression**, gerando um valor de retorno.

O conceito de parâmetros formais, trabalhado pelo subcomponente **FormalParameter**, já foi tratado pelos componentes do paradigma imperativo (Seção 4.2.4), entretanto, existem algumas diferenças com relação ao paradigma funcional, as quais serão comentadas mais adiante.

4.3.2 Chamada de Funções

Toda chamada de função é uma expressão, e como tal produz um valor final que pode ser usado por outras funções e expressões. A chamada de uma função ocasiona a execução de uma função qualquer, e pode ser acompanhada de argumentos ou não, produzidos pela avaliação dos parâmetros atuais.

Portanto, segue abaixo a especificação da operação semântica **fnc-expressions**, responsável pela produção dos componentes semânticos que representam o conceito de chamada de funções:

```
fnc-expressions :: -> language-description.
fnc-expressions =
   | syntax
   |    [ e # * Expression ap # * ActualParameter ] --> * Expression
   | semantics
   |    | | semantics of e
   |    | and
   |    | | semantics of ap
   |    then
   |    | enact application (function-body of the given function # 1) to
   |    | (rest (the given data))
```

```
and
...
```

Podemos observar que o componente semântico produzido, na seção **syntax**, é formado pelo elemento sintático **i # Identifier** e pelo o subcomponente **ActualParameter**; e, na sua seção **semantics**, procura executar a abstração do procedimento associado ao valor do elemento sintático **i # Identifier**, seguida pela avaliação semântica do subcomponente **ActualParameter**, antes da execução da abstração do procedimento associado ao valor do elemento sintático **i # Identifier**.

4.3.3 Expressões Funcionais

A operação semântica **fnc-expressions** também produz, além do componente semântico para chamada de funções (Seção 4.3.2), um componente semântico para a produção de uma função associada a um identificador qualquer, o que é totalmente válido, já que uma função de alta ordem também é um valor, e conseqüentemente uma expressão [10]:

```
fnc-expressions =
   ...
   and
   | syntax
   |    [ i # Identifier ] --> * Expression
   | semantics
   |    give the function bound to token of * i
   and
   ...
```

Este novo componente semântico contém, na seção **syntax**, um elemento **sintático i # Identifier**; e efetua, na seção **semantics**, a produção da função associada ao valor do elemento sintático **i # Identifier**.

Um outro componente semântico produzido pela operação semântica **fnc-expressions** procura declarar e produzir uma nova função como uma expressão. Esta nova função não tem um identificador associado, sendo portanto indicada para representar funções que só serão avaliadas uma única vez:

```
fnc-expressions =
   ...
   and
   | syntax
   |    [ "fn" fp # * FormalParameter "==>" e # * Expression ] --> * Expression
   | semantics
   |    give function of closure abstraction of
   |    | | furthermore semantics of fp
   |    | hence
   |    | | semantics of e
   and
   ...
```

Podemos observar que o componente semântico produzido, na sua seção **syntax**, é formado pelos subcomponentes **FormalParameter** e **Expression**; e, na sua seção **semantics**, procura produzir uma abstração, cuja semântica procura avaliar a semântica do subcomponente **FormalParameter**, gerando parâmetros formais, e em seguida processa a semântica do subcomponente **Expression**, gerando um valor de retorno.

4.3.4 Parâmetros Atuais

O conceito de parâmetros atuais, já abordado na biblioteca de componentes semânticos imperativos complexos (Seção 4.2.3), além dos procedimentos, também se aplicam as funções, entretanto, no caso do paradigma funcional, apenas funções e valores serão passados como parâmetros, devido ao uso do mecanismo de declaração constante na passagem de parâmetros [10], [20].

Os componentes semânticos para representação do conceito de parâmetros atuais na biblioteca de componentes semânticos funcionais são produzidos pela operação semântica **fnc-actual-parameter**, cuja especificação segue abaixo:

```
fnc-actual-parameter :: -> language-description.
fnc-actual-parameter =
   | syntax
   |    [ e # * Expression ] --> * ActualParameter
   | semantics
   |    semantics of e
   and
   | syntax
   |    [ ap1 # * ActualParameter "," ap2 # * ActualParameter ] --> * ActualParameter
   | semantics
   |    | | semantics of ap1
   |    | and
   |    | | semantics of ap2.
```

Como funções e valores podem ser produzidos por expressões [10], só é preciso especificar um componente semântico, neste caso o primeiro, para produzir o resultado de uma expressão, dentro do *namespace* **ActualParameter**. O segundo componente semântico permite o processamento de mais de um parâmetro atual quando necessário.

4.3.5 Parâmetros Formais

O conceito de parâmetros formais, já abordado na biblioteca de componentes semânticos imperativos complexos (Seção 4.2.4), além dos procedimentos, também se aplicam as funções. Quanto a passagem de parâmetros, esta será por *constante* devido ao uso do

mecanismo de declaração constante, de forma que apenas valores, e conseqüentemente funções, serão passados como parâmetros [10], [20].

Os componentes semânticos para representação de parâmetros formais serão produzidos pela operação semântica **fnc-formal-parameter**, cuja especificação segue abaixo:

```
fnc-formal-parameter :: -> language-description.
fnc-formal-parameter =
   | syntax
   |    [ i # Identifier ] --> * FormalParameter
   | semantics
   |    | | give first them
   |    | then
   |    | | bind token of * i to the given value
   |    and then
   |    | give rest them
   and
   | syntax
   |    [ fp1 # * FormalParameter ";" fp2 # * FormalParameter ] --> * FormalParameter
   | semantics
   |    | semantics of fp1
   |    then
   |    | semantics of fp2.
```

O primeiro componente semântico, na seção **syntax**, possui um elemento sintático **i # Identifier**; e, na seção **semantics**, ocorre apenas a associação do elemento sintático **i # Identifier** ao valor do *sort* **value** passado como argumento na chamada da função. Podemos observar também, a criação de um novo *namespace* denominado **FormalParameter**; e, na seção **semantics**, o consumo do primeiro argumento durante a sua avaliação semântica, e após este processo, a propagação dos demais argumentos para serem novamente consumidos pelo próximo parâmetro formal, no caso de existir mais de um (Seção 4.2.4).

O segundo componente semântico permite o processamento de mais de um parâmetro formal quando necessário.

4.3.6 Expressão Condicional

Para se efetuar um controle condicional sobre qual expressão deve ser avaliada, o paradigma funcional utiliza o conceito de expressão **if** [10]. Esta expressão será representada pelo componente semântico produzido pela operação semântica **fnc-expressions**, responsável também pela produção de outros componentes semânticos já demonstrados:

```
fnc-expressions =
   ...
   and
   | syntax
   |    [ "if" e1 # * Expression "then" e2 # * Expression "else" e3 # * Expression ] --> *
Expression
   | semantics
   |    | semantics of e1
```

```
|   then
|   | | check it is true and then semantics of e2
|   | or
|   | | check it is false and then semantics of e3.
```

Podemos observar que o componente semântico produzido, na sua seção **syntax**, é formado por três subcomponentes **Expression**; e, na sua seção **semantics**, efetua-se inicialmente a avaliação semântica do primeiro subcomponente **Expression**. Caso o resultado seja igual a **true**, o segundo subcomponente **Expression** será avaliado. Caso o resultado seja igual a **false**, o terceiro subcomponente **Expression** será avaliado.

4.3.7 Agrupando Componentes Semânticos Funcionais

Para finalizar a nossa biblioteca de componentes semânticos funcionais, criamos as operações semânticas **fnc-local-program** e **fnc-program**, as quais produzem a **language-description** formada pelo agrupamento de todos os componentes semânticos da biblioteca funcional. A diferença é que a **fnc-program** também agrupa os componentes semânticos produzidos pela operação semântica **exp-program** (Seção 3.4.4), os quais são necessários para efeitos de animação da especificação com a ferramenta ABACO [11]:

```
fnc-program :: -> language-description.
fnc-program =
   | exp-program
   and
   | fnc-local-program.

fnc-local-program :: -> language-description.
fnc-local-program =
   | fnc-declarations
   and
   | fnc-expressions
   and
   | fnc-actual-parameter
   and
   | fnc-formal-parameter.
```

Na compilação de programas funcionais, a operação semântica **compile** (Seção 2.2.2) irá receber como argumentos: uma **language-description**, neste caso **fnc-program**; um *namespace* (**Expression**) definindo que tipo de componente semântico deve ser inicialmente processado; e a árvore sintática de expressões, representando o código do programa propriamente dito.

Exemplo 4.3: Declarando uma Função Fatorial

Como exemplo, criamos um programa funcional (**Figuras 4.7** e **4.8**), que declara uma função **fatorial**, que calcula o fatorial de um número qualquer, e uma variável **y** cujo valor é o resultado da chamada função **fatorial** passando o valor **3** como argumento. O *namespace* definido para a operação semântica **compile** é o *namespace* **Declaration** porque o programa exemplo está apenas declarando elementos, ou seja, produzindo associações ao invés de valores.

```
begin
   val rec fatorial == fn x ==>
      begin
         if ( x <> 1 )
         then
            x * fatorial ( x - 1)
         else 1
       end;

   val y == fatorial( 3 )
end
```

Figura 4.7 – Linguagem concreta do programa Exemplo 4.3.

```
compile (fnc-program) (* Declaration)
[
  [
    "val" "rec" fatorial "==" "fn" [ x ] "==>"
    [
      "if" [ [ x ] [ "<>" ] [ 1 ] ]
      "then"
      [
        [ x ] [ "*" ] [ [ fatorial ] [ [ [ x ] [ "-" ] [ 1 ] ] ] ]
      ]
      "else" [ 1 ]
    ]
  ]
  "."
  [
    "val" y "=="
    [
      [ fatorial ]  [ [ 3 ] ]
    ]
  ]
]
```

Figura 4.8 – Linguagem abstrata do programa Exemplo 4.3 para uso da função semântica compile _ _ _.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 4.9** abaixo) que, quando executadas produzem como resultado final: as associações **{"fatorial" --> function of (...), "y" --> 6}**. Valores transitórios não foram produzidos.

```
| recursively bind "fatorial" to (function of (closure abstraction of
| | | furthermore
| | | | | | give (first (the given data) )
| | | | | then
| | | | | | bind "x" to (the given value)
| | | | and then
| | | | | give (rest (the given data) )
| | hence
| | | | | | give "DIFF"
| | | | | and then
| | | | | | | | give (the value bound to "x")
| | | | | | | or
| | | | | | | | give (the function bound to "x")
| | | | | | and then
| | | | | | | give 1
| | | | then
| | | | | binary-operation-of (the given string # 1) (the given value # 2) (the given value # 3)
| | | then
| | | | | | check ( (the given datum) is true)
| | | | | and then
| | | | | | | | give "MULT"
| | | | | | | and then
| | | | | | | | | | give (the value bound to "x")
| | | | | | | | | or
| | | | | | | | | | give (the function bound to "x")
| | | | | | | | and then
| | | | | | | | | | | | give (the value bound to "fatorial")
| | | | | | | | | | | or
| | | | | | | | | | | | give (the function bound to "fatorial")
| | | | | | | | | | and
| | | | | | | | | | | | | | give "MINUS"
| | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give (the value bound to "x")
| | | | | | | | | | | | | | | or
| | | | | | | | | | | | | | | | give (the function bound to "x")
| | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | then
| | | | | | | | | | | | binary-operation-of (the given string # 1) (the given value # 2) (the given
value # 3)
| | | | | | | | | then
| | | | | | | | | | enact (application (function-body of (the given function # 1) ) to (rest (the given
data) ) )
| | | | | | then
| | | | | | | binary-operation-of (the given string # 1) (the given value # 2) (the given value # 3)
```

Figura 4.9 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.3.

```
| | | | or
| | | | | | check  ( (the given datum)  is false)
| | | | | and then
| | | | | | give 1
| ) )
before
| | | | | give  (the value bound to "fatorial")
| | | | or
| | | | | give  (the function bound to "fatorial")
| | | and
| | | | give 3
| | then
| | | enact  (application  (function-body of  (the given function # 1) )  to  (rest  (the given data) ) )
| then
| | bind "y" to  (the given value)
```

Figura 4.9 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.3.

Vale ressaltar que, como o paradigma funcional não trabalha com alocação de memória, nenhuma célula de memória foi produzida.

□

4.4 Componentes Orientados a Objetos

O paradigma orientada a objetos surgiu em paralelo com a criação de uma linguagem de programação orientada a objetos chamada de SIMULA-67 [27], desenvolvida na Suécia em 1967. A idéia básica do paradigma orientado a objetos é imaginar que programas simulam o mundo real, um mundo povoado de objetos. Dessa maneira, linguagens baseadas nos conceitos de simulação do mundo real devem incluir um modelo de objetos que possam enviar e receber mensagens e reagir a mensagens recebidas. Esse conceito é baseado na idéia de que no mundo real freqüentemente usamos objetos sem precisarmos conhecer como eles realmente funcionam. Assim, programação orientada a objetos fornece um ambiente onde múltiplos objetos podem coexistir e trocar mensagens entre si [20], [21].

Um outro bom exemplo de linguagem de programação orientada a objetos é a linguagem Java [33]. Ela foi desenvolvida com o objetivo de apresentar poucas dependências de implementação, permitindo aos desenvolvedores de aplicações escreverem programas somente uma vez e serem capazes de rodar o mesmo programa em qualquer máquina [33].

De fato, cada um dos objetos é instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamentos de herança. Existem três aspectos importantes nesta definição de linguagem orientada a objetos: usa objetos e não funções como

seu bloco lógico fundamental; cada objeto é instância de alguma classe; e classes estão relacionadas umas com as outras via relacionamentos de herança. Caso um destes elementos não esteja presente, a linguagem pode até parecer, mas não será considerado orientado a objetos [20], [21], [32].

Do ponto de vista de programação, objetos são coleções de operações que compartilham um estado. As operações determinam a que mensagens (ativações de operações) o objeto pode responder, enquanto o estado compartilhado é escondido do mundo exterior. Apenas as operações têm acesso a ele. Variáveis representando o estado interno de um objeto são chamadas de *variáveis de instância* e suas operações são chamadas de *métodos* [20], [21].

Fazendo uma analogia dos termos do paradigma orientado a objetos com os termos tradicionais de programação, podemos afirmar que objetos correspondem a valores, enquanto que classes estão associadas a tipos. Dizer que um objeto é instância de uma classe é semelhante a dizer que um valor é de um determinado tipo. Já o conceito de herança não tem correspondente nas linguagens de programação tradicionais, sendo a contribuição original do paradigma orientado a objetos [9].

Portanto, vamos começar com a especificação de uma biblioteca de componentes semânticos orientado a objetos, a partir da definição de conceitos básicos do paradigma orientado a objetos, tais como: declaração de classes, variáveis de instância, construtores e métodos; manipulação de objetos e referências; operações com classes e objetos; e chamada de métodos e superconstrutores.

4.4.1 Declaração de Classes

Na biblioteca de componentes semânticos orientado a objetos, a declaração de classes é especificada por dois componentes semânticos, ambos produzidos pela operação semântica **obj-class-types**. Como o conceito de classe pode ser associado ao conceito de tipo, ambos os componentes irão estender o *namespace* **Type**, especificado previamente nas bibliotecas de componentes semânticos anteriores (Seções 4.1, 4.2):

```
obj-class-types :: -> language-description.
obj-class-types =
   | syntax
   |    [ "class" "private" fct # * FieldClassType
   |      "public" cd # * ConstructorDec md # * MethodDec "end" ] --> * Type
   | semantics
   |    | | semantics of fct
   |    | and then
   |    | | semantics of cd
   |    | and then
   |    | | semantics of md
   |    then
   |    | give class-of (the given type-bindings # 1) (the given method-bindings # 3)
```

```
|    |            (the given constructor # 2) empty-class
and
| syntax
|    [ "class" t # * Type "private" fct # * FieldClassType
|      "public" cd # * ConstructorDec md # * MethodDec "end" ] --> * Type
| semantics
|    | | semantics of t
|    | and then
|    | | semantics of fct
|    | and then
|    | | semantics of cd
|    | and then
|    | | semantics of md
|    then
|    | give class-of (the given type-bindings # 2) (the given method-bindings # 4)
|    |            (the given constructor # 3) (the given class # 1)
and
...
```

Podemos observar que o primeiro componente semântico produzido, na sua seção **syntax**, é formado pelos subcomponentes **FieldClassType**, **ConstructorDec** e **MethodDec**, respectivamente, cujos *namespaces* serão produzidos posteriormente (Seções 4.4.2, 4.4.4, 4.4.3 respectivamente); e, na sua seção **semantics**, efetua-se a avaliação semântica destes subcomponentes, produzindo valores dos *sorts* **type-bindings**, **constructor** e **methos-bindings**, respectivamente. No final, o componente semântico produz um valor do *sort* **class**, através do uso da função semântica **class-of _ _ _ _**, que recebe como argumentos os valores dos *sorts* produzidos pela avaliação semântica dos subcomponentes anteriores mais o valor **empty-class** do *sort* **class**, indicando que a nova classe produzida é filha da classe default da linguagem.

O segundo componente semântico possui a mesma estrutura do componente semântico anterior, acrescentando, na seção **syntax**, o subcomponente **Type**; e, na seção **semantics**, a avaliação semântica deste novo subcomponente, produzindo um valor do *sort* **class** que será usado no lugar do valor **empty-class** para a produção de um novo valor do *sort* **class**, através do uso da função semântica **class-of _ _ _ _**.

4.4.2 Declaração de Variáveis de Instâncias

Infelizmente, as especificações em semântica de ações para linguagens orientadas a objetos, até o presente momento, não fornecem um controle de escopo para variáveis de instância e métodos nos objetos de uma classe [14], [15]. Sendo assim, variáveis de instância serão sempre privadas e métodos serão sempre públicos.

Portanto, vamos agora especificar componentes semânticos para representar a declaração de variáveis de instância, os quais declaram o *namespace* **FieldClassType** utilizado durante a declaração de uma classe. Estes componentes semânticos são produzidos pela

operação semântica **obj-class-types**, utilizada previamente na produção dos componentes semânticos para declaração de classes (Seção 4.4.1), portanto segue abaixo apenas o trecho da especificação referente a variáveis de instância:

```
obj-class-types =
   ...
   and
   | syntax
   |    [ ] --> * FieldClassType
   |  semantics
   |    give empty-type-bindings
   and
   | syntax
   |     [ i # Identifier ":" t # * Type  ] --> * FieldClassType
   | semantics
   |     | semantics of t
   |     then
   |     | give type-bindings of (map of token of * i to (the given type))
   and
   | syntax
   |     [ fct1 # * FieldClassType ";" fct2 # * FieldClassType ] --> * FieldClassType
   | semantics
   |     | | semantics of fct1
   |     | and
   |     | | semantics of fct2
   |     then
   |     | give type-bindings of (disjoint union (map-type of the given type-bindings # 1) (map-
type of the given type-bindings # 2)).
```

O primeiro componente semântico, na sua seção **syntax**, não possui elementos sintáticos; e, na seção **semantics**, produz o valor **empty-type-bindings** do *sort* **type-bindings**, o que significa que a classe não terá variáveis de instância.

O segundo componente semântico, na sua seção **syntax**, é formado pelo elemento sintático **i # Identifier** e pelo subcomponente **Type**; e, na seção **semantics**, produz um valor do *sort* **type-bindings**, a partir do mapeamento formado entre valor do elemento sintático **i # Identifier** e o valor do *sort* **type**, gerado durante a avaliação semântica do subcomponente **Type**.

O terceiro componente semântico foi especificado para garantir que uma classe possa ter mais de uma variável de instância.

4.4.3 Declaração de Métodos

Vamos agora especificar componentes semânticos para representar a declaração de métodos, os quais declaram o *namespace* **MethodDec** utilizado durante a declaração de uma classe (Seção 4.4.1). Estes componentes semânticos são produzidos pela operação semântica **obj-class-method**, cuja especificação segue abaixo:

```
obj-class-method :: -> language-description.
obj-class-method =
   | syntax
```

```
|   [ ] --> * MethodDec
| semantics
|   give empty-method-bindings
and
| syntax
|   [ "procedure" i # Identifier "()" c # * Command ] --> * MethodDec
| semantics
|   give method-bindings of (map of token of * i to method of
|   closure abstraction of
|     | | furthermore
|     | | | | generate the variable-bindings of (the given object-variable # 1)
|     | | | and
|     | | | | bind "self" to (the given object-variable # 1)
|     | hence
|     | | semantics of c
|   )
and
| syntax
|   [ "procedure" i # Identifier "(" fp # * FormalParameter ")" c # * Command ] --> *
MethodDec
| semantics
|   give method-bindings of (map of token of * i to method of
|   closure abstraction of
|     | | furthermore
|     | | | | generate the variable-bindings of (the given object-variable # 1)
|     | | | and
|     | | | | bind "self" to (the given object-variable # 1)
|     | hence
|     | | | | give rest the given data
|     | | | then
|     | | | | furthermore semantics of fp
|     | | before
|     | | | semantics of c
|   )
and
| syntax
|   [ md1 # * MethodDec ";" md2 # * MethodDec ] --> * MethodDec
| semantics
|   | | semantics of md1
|   | and
|   | | semantics of md2
|   then
|   | give method-bindings of (disjoint union (the given method-bindings # 1 , the given
method-bindings # 2)).
```

O primeiro componente semântico, na sua seção **syntax**, não possui elementos sintáticos; e, na seção **semantics**, produz o valor **empty-method-bindings** do *sort* **method-bindings**, o que significa que a classe não terá métodos.

O segundo componente semântico, na sua seção **syntax**, é formado pelo elemento sintático **i # Identifier** e pelo subcomponente **Command**; e, na seção **semantics**, produz um valor do *sort* **method-bindings**, a partir do mapeamento formado entre valor do elemento sintático **i # Identifier** e o valor do *sort* **method**, cuja abstração é formada pela geração das associações das variáveis de instância da variável objeto atual usando a função semântica **generate _**, pela

associação do elemento sintático "**self**" a variável objeto atual, e pela avaliação semântica do subcomponente **Command**, respectivamente.

O terceiro componente semântico possui a mesma estrutura do componente semântico anterior, acrescentando, na seção **syntax**, o subcomponente **FormalParameter**; e, na seção **semantics**, a avaliação semântica do novo subcomponente (gerando parâmetros formais), antes da avaliação semântica do subcomponente **Command**.

O quarto componente semântico foi especificado para garantir que uma classe possa ter mais de uma declaração de método.

4.4.4 Declaração de Construtor de Classe

Um construtor é um método especial da classe que serve para alocar memória e inicializar variáveis quando necessário [30], [32]. Portanto, vamos agora especificar componentes semânticos para representar a declaração de construtores, os quais declaram o *namespace* **ConstructorDec** utilizado durante a declaração de uma classe (Seção 4.4.1). Estes componentes semânticos são produzidos pela operação semântica **obj-class-constructor**, cuja especificação segue abaixo:

```
obj-class-constructor :: -> language-description.
obj-class-constructor =
   | syntax
   |    [ ] --> * ConstructorDec
   |  semantics
   |    give constructor of (abstraction of complete)
   and
   | syntax
   |    [ "constructor" "()" c # * Command ] --> * ConstructorDec
   | semantics
   |    give constructor of (closure abstraction of
   |      | | furthermore
   |      | | | | bind "self" to (the given object-variable # 1)
   |      | | | and
   |      | | | | generate the variable-bindings of (the given object-variable # 1)
   |      | hence
   |      | | semantics of c
   |      )
   and
   | syntax
   |    [ "constructor" "(" fp # * FormalParameter ")" c # * Command ] --> * ConstructorDec
   | semantics
   |    give constructor of (closure abstraction of
   |      | | furthermore
   |      | | | | bind "self" to (the given object-variable # 1)
   |      | | | and
   |      | | | | generate the variable-bindings of (the given object-variable # 1)
   |      | hence
   |      | | | | give rest them
   |      | | | then
   |      | | | | furthermore semantics of fp
   |      | | before
   |      | | | semantics of c
```

```
|     | then
|     | | complete
|     ).
```

O primeiro componente semântico, na sua seção **syntax**, não possui elementos sintáticos; e, na seção **semantics**, produz um valor do *sort* **constructor** cuja abstração é formada pela ação **complete**, o que significa que a execução do construtor não afetará nenhuma informação corrente.

O segundo componente semântico, na sua seção **syntax**, é formado pelo subcomponente **Command**; e, na seção **semantics**, produz um valor do *sort* **constructor** cuja abstração é formada pela associação do elemento sintático "**self**" a variável objeto atual, pela geração das associações das variáveis de instância da variável objeto atual usando a função semântica **generate _**, e pela avaliação semântica do subcomponente **Command**, respectivamente.

O terceiro componente semântico possui a mesma estrutura do componente semântico anterior, acrescentando, na seção **syntax**, o subcomponente **FormalParameter**; e, na seção **semantics**, a avaliação semântica do novo subcomponente, gerando parâmetros formais (Seção 4.2.4), antes da avaliação semântica do subcomponente **Command**.

4.4.5 Trabalhando com Objetos

Vamos agora tratar da manipulação de objetos. Basicamente, existem três funções semânticas que são responsáveis pela manipulação de objetos: **instantiate object-variable of _,** que gera a variável objeto em si; **instantiate _**, que aloca memória para as variáveis de instância de uma classe qualquer; e **generate _**, que produz as associações das variáveis de instância de uma classe qualquer.

Podemos observar que, na função **instantiate object-variable of _**, cuja especificação segue abaixo, ocorre: a alocação de memória das variáveis de instância de um valor do *sort* **class** qualquer, através da chamada da função semântica **instantiate _**; seguida da produção de um valor do *sort* **object-variable**, utilizando os valores dos *sorts* **class** (recebido como argumento) e **variable-bindings** (produzido pela função semântica **instantiate _**):

```
instantiate object-variable of _ :: yielder -> action.
instantiate object-variable of (~c : class) =
   | | give ~c
   | and then
   | | instantiate the field-type-bindings of ~c
   then
   | give object-variable of (the given class # 1) (the given variable-bindings # 2).
```

Na função **instantiate _**, podemos observar que existe um processamento recursivo, ou seja, a função semântica: recebe como argumento um valor do *sort* **type-bindings**; aloca uma

variável para o valor do *sort* **type** apontado pelo identificador do primeiro elemento contido no mapeamento do valor do *sort* **type-bindings**; e chama novamente a função semântica **instantiate _** com o mesmo valor do *sort* **type-bindings** omitindo o identificador do primeiro elemento usado anteriormente. No final é produzido um valor do *sort* **variable-bindings**, através da junção: do mapeamento produzido entre o identificador do primeiro elemento usado anteriormente e o valor do *sort* **variable** produzido na alocação de variável, e do valor do *sort* **variable-bindings** produzido pela chamada recursiva de **instantiate _**:

```
instantiate _ :: yielder -> action.
instantiate empty-type-bindings = give empty-variable-bindings.
re instantiate (~t : type-bindings) =
   | | check (map-type of ~t) is empty-map
   | then
   | | instantiate (empty-type-bindings)
   or
   | | check not ((map-type of ~t) is empty-map)
   | then
   | | | | give (map-type of ~t)
   | | | and then
   | | | | give first elements mapped-set (map-type of ~t)
   | | then
   | | | | allocate variable of ((the given map # 1) at (the given string # 2))
   | | | and then
   | | | instantiate type-bindings of ((the given map # 1) omitting (set of (the given string # 2)))
   | | | and then
   | | | | give (the given string # 2)
   | then
   | | give variable-bindings of disjoint union
   | |  (map-variable of (the given variable-bindings # 2))
   | |  (map of (the given string # 3) to (the given variable # 1)).
```

Na função **generate _**, podemos observar que também existe um processamento recursivo, ou seja, a função semântica: recebe como argumento um valor do *sort* **variable-bindings**; produz uma associação entre o identificador do primeiro elemento contido no mapeamento do valor do *sort* **variable-bindings** e o valor do *sort* **variable** apontado pelo mesmo identificador; e chama novamente a função semântica **generate _** com o mesmo valor do *sort* **variable-bindings** omitindo o identificador do primeiro elemento usado anteriormente, gerando assim as demais associações para todos os elementos do valor do *sort* **variable-bindings**:

```
generate _ :: yielder -> action.
generate empty-variable-bindings = complete.
generate (~v : variable-bindings) =
   | | | give (map-variable of ~v)
   | | and then
   | | | give first elements mapped-set (map-variable of ~v)
   | then
   | | bind (the given string # 2) to ((the given map # 1) at (the given string # 2) )
   before
   | generate variable-bindings of ((the given map # 1) omitting (set of (the given string # 2))).
```

Vamos agora tratar da manipulação de variáveis referência, ponto chave na junção entre o paradigma imperativo e o orientado a objetos, no que diz respeito a declaração de variáveis.

4.4.6 Variáveis Referência

Uma variável referência é uma variável que aponta para uma variável objeto (Seção 4.4.5), ou seja, quando o programador declara uma variável de um tipo classe qualquer (Seção 4.4.1), na verdade ele está declarando uma variável referência, a qual irá apontar para uma variável objeto qualquer que lhe for atribuída, desde que seja compatível com a classe da variável referência [21], [30].

Para manipulação de variáveis referência, foi apenas especificado um componente semântico o qual é produzido pela operação **obj-reference-expressions** cuja especificação segue abaixo:

```
obj-reference-expressions :: -> language-description.
obj-reference-expressions =
   | syntax
   |    [ "nil" ] --> * Expression
   | semantics
   |    give null-reference.
```

Podemos observar que o componente semântico, na seção **syntax**, é formado pelo elemento sintático "**nil**"; e, na seção **semantics**, produz o valor **null-reference** do *sort* **reference-value**. Apesar da variável referência apontar para uma variável objeto, assim como uma variável ponteiro aponta para uma variável qualquer (Seção 4.2.9), ela não é compatível com ponteiros [32], de modo que, se o elemento sintático para indicar uma referência nula fosse "**null**" (Seção 4.2.9) ao invés de "**nil**" teríamos um problema de compatibilidade semântica.

Entre as funções semânticas para manipulação de variáveis referência temos: a redefinição das funções semânticas de manipulação de variáveis (**alocate variable of _**, **the value attributed to _** e **attribute _ to _**) (Seção 4.1.1), e a função semântica **_ is a instance of _**.

A função semântica **_ is a instance of _** é usada na verificação de compatibilidade entre uma variável referência e uma variável objeto que irá ser apontada pela mesma. São efetuados três testes de compatibilidade: um valor referência para uma classe, uma referência nula para uma classe (sempre é **false**), e um valor referência para uma variável referência:

```
_ is a instance of _ :: yielder, yielder -> yielder.
(~r : reference-value) is a instance of (~c : class) =
   ~c is in the superclasses of (the class-ref of ~r).
null-reference is a instance of (~c : class) = false.
(~r1 : reference-value) is a instance of (~r2 : reference-variable) =
   (the class-ref of (the type of ~r2)) is in the superclasses of (the class-ref of ~r1).
```

Podemos observar que os testes de compatibilidade verificam se o tipo da variável que vai receber a nova referência está entre as superclasses (incluindo a própria classe) desta referência, já que variável de uma superclasse pode receber uma referência de uma subclasse. Entretanto, o processo inverso não é correto, já que uma referência de uma superclasse não pode representar todas as variáveis de instância de uma subclasse [20], [22].

Quanto às funções semânticas para manipulação de variáveis do tipo referência, podemos observar que temos duas funções semânticas para alocação de variáveis: uma para o tratamento do tipo classe, e outra para o tratamento de uma valor do *sort* **reference-type**:

```
allocate variable of (~c : class) = allocate variable of (the reference-type of ~c).

allocate variable of (~t : reference-type) =
   | | give ~t
   | and then
   | | allocate variable of pointer-type
   then
   | give the reference-variable of (the given pointer-variable # 2) (the given reference-type #
1).
```

A primeira redefinição de função semântica **allocate variable of _** (Seção 4.1.1) repassa a alocação de variável, através de uma chamada recursiva da mesma função semântica, mas agora para alocar um valor do *sort* **reference-type**, produzido pela função semântica **the reference-type of _** que recebe um valor do *sort* **class** como argumento.

A segunda redefinição de função semântica **allocate variable of _** (Seção 4.1.1) irá alocar uma variável para o valor do *sort* **reference-type**, gerando um valor do *sort* **reference-variable**. Primeiro aloca-se uma variável ponteiro (Seção 4.2.9), gerando um valor do *sort* **pointer-variable**, em seguida produz-se um valor do *sort* **reference-variable** através da função semântica **the reference-variable of _**, que recebe como argumentos o valor do *sort* **pointer-variable** gerando e o valor do *sort* **reference-type** (argumento da função semântica **allocate variable of _**).

Quanto as demais funções semânticas para manipulação de variáveis do tipo referência, temos: a consulta do valor atribuído a uma variável referência e a atribuição de um valor referência a uma variável referência:

```
the value attributed to (~r : reference-variable) =
   | the reference-value of (the value attributed to (the object-ref of ~r))
   | (the class-ref of (the type of ~r)).

attribute (~r1 : reference-value) to (~r2 : reference-variable) =
   | check not (~r1 is null-reference) and then
   | | check (~r1 is a instance of ~r2) and then
   | | | attribute (the identity of ~r1) to (the object-ref of ~r2)
   or
   | check (~r1 is null-reference) and then
   | | attribute null-reference to (the object-ref of ~r2).
```

Na consulta do valor atribuído a uma variável referência, temos a produção de um valor do *sort* **reference-value** a partir da função semântica **the reference-value of _** (Seção 4.1.1), que recebe como argumentos: um valor do *sort* **pointer-value**, que aponta para uma variável objeto qualquer; e um valor do *sort* **class**. Ambos são obtidos do valor do *sort* **reference-variable** fornecido como argumento da função **the value attributed to _**.

Na atribuição de um valor referência a uma variável referência, através do uso da função semântica **attribute _ to _** (Seção 4.1.1), a principal preocupação é garantir que será atribuído um valor referência compatível com a variável referência, através da utilização da função semântica **_ is a instance of _**. Após a garantia da compatibilidade, é atribuído o valor do *sort* **pointer-value** ao valor do *sort* **pointer-variable**, ambos obtidos dos valores dos *sorts* **reference-value** e **reference-variable**, respectivamente.

4.4.7 Novas Expressões

Entre os componentes semânticos orientados a objetos especificados, existem algumas novas expressões que são: a produção do valor "**self**" e a operação "**instance-of**". O valor **self** é declarado apenas no construtor e nos métodos de uma classe, e indica a instância atual da variável objeto que está sendo executada [30]. A operação **instance-of** indica se um determinado objeto é uma instância de uma classe qualquer [32], ela também utiliza a função semântica **_ is a instance of _** especificada anteriormente.

Portanto, segue abaixo a especificação da operação semântica **obj-expressions**, responsável pela produção de componentes semânticos orientados a objetos que estendem o *namespace* **Expression**:

```
obj-expressions :: -> language-description.
obj-expressions =
   | syntax
   |    [ "self" ] --> * Expression
   | semantics
   |    | give (the object-variable bound to "self")
   |    then
   |    | give the reference-value of (pointer to (the given object-variable))
   |    | (the class of (the given object-variable))
   and
   | syntax
   |    [ e # * Expression "instanceof" t # * Type ] --> * Expression
   | semantics
   |    | | semantics of e
   |    | and then
   |    | | semantics of t
   |    then
   |    | give (the given reference-value # 1) is a instance of (the given class # 2)
   and
   | syntax
   |    [ e # * Expression "instanceof" i # * Ident ] --> * Expression
   | semantics
```

```
|    | | semantics of e
|    | and then
|    | | semantics of i
|    | then
|    | give (the given reference-value # 1) is a instance of (the given reference-variable # 2)
and
...
```

O primeiro componente semântico, na seção **syntax**, possui apenas o elemento sintático "**self**"; e, na seção **semantics**, produz um valor do *sort* **reference-value**, gerado a partir da função semântica **the reference-value of _** que recebe os *sorts* **pointer-value** (ponteiro para o valor do *sort* **object-variable** associado ao elemento sintático "**self**") e **class** como argumentos.

O segundo componente semântico, na seção **syntax**, é formado pelos subcomponentes **Expression** e **Type**; e, na seção **semantics**, efetua a avaliação semântica dos subcomponentes, gerando valores dos *sorts* **reference-value** e **class**, os quais serão usados pela função semântica **_ is a instance of _** especificada previamente.

O terceiro componente semântico, na seção **syntax**, é formado pelos subcomponentes **Expression** e **Ident**; e, na seção **semantics**, efetua a avaliação semântica dos subcomponentes, gerando valores dos *sorts* **reference-value** e **reference-variable**, os quais serão usados pela função semântica **_ is a instance of _** especificada previamente.

Existem outros dois componentes semânticos produzidos pela operação semântica **obj-expressions**, os quais são responsáveis pela criação de objetos dinamicamente. De fato, quando se declara uma variável de um tipo classe qualquer, implicitamente se declara uma variável referência, que neste momento inicial não aponta para nenhuma instância de objeto. Só após a atribuição de uma instância de objeto é que essa variável referência irá apontar para uma variável objeto qualquer [30], [32].

Portanto, segue abaixo um trecho da especificação da operação semântica **obj-expressions**, produzindo os componentes semânticos responsáveis pela produção de instâncias de objetos (contidas em valores do *sort* **reference-value**):

```
obj-expressions =
  ...
  and
  | syntax
  |    [ "new" t # * Type "()" ] --> * Expression
  | semantics
  |    | | semantics of t
  |    | then
  |    | | | regive
  |    | | and then
  |    | | | instantiate object-variable of the given class # 1
  |    | then
  |    | | enact application constructor-body of
  |    | | (the constructor of (the given class # 1)) to (the given object-variable # 2)
  |    | hence
```

```
|    | | give the reference-value of
|    | | (pointer to (the given object-variable # 2)) (the given class # 1)
and
| syntax
|    [ "new" t # * Type "(" ap # * ActualParameter ")" ] --> * Expression
| semantics
|    | | semantics of t
|    | then
|    | | | regive
|    | | and then
|    | | | instantiate object-variable of the given class # 1
|    | and then
|    | | semantics of ap
|    then
|    | | enact application constructor-body of
|    | | (the constructor of (the given class # 1)) to (rest them)
|    | hence
|    | | give the reference-value of
|    | | (pointer to (the given object-variable # 2)) (the given class # 1).
```

O primeiro componente semântico, na seção **syntax**, é formado pelo subcomponente **Type**; e, na seção **semantics**, efetua a avaliação semântica do subcomponente **Type**, seguido pela geração de um valor do *sort* **object-variable** (instância de um objeto) a partir da função semântica **instantiate object-variable of _**, especificada previamente. Em seguida é feita a chamada do construtor do valor do *sort* **class**, produzido na avaliação semântica do subcomponente **Type**; e finalmente a produção do valor do *sort* **reference-value**, gerado pela função semântica **the reference-value of _** que recebe os *sorts* **pointer-value** (ponteiro para o valor do *sort* **object-variable** produzido pela função semântica **instantiate object-variable of _**) e **class** (produzido na avaliação semântica do subcomponente **Type**) como argumentos.

O segundo componente semântico, na seção **syntax**, é formado pelos subcomponentes **Type** e **ActualParameter**; e, na seção **semantics**, efetua a avaliação semântica do subcomponente **Type**, seguido pela geração de um valor do *sort* **object-variable** (instância de um objeto formada a partir da função semântica **instantiate object-variable of _** especificada previamente na Seção 4.4.5), e pela avaliação semântica do subcomponente **ActualParameter**, gerando os parâmetros atuais a serem usados pelo construtor (Seção 4.2.3). Em seguida é feita a chamada do construtor do valor do *sort* **class**, produzido na avaliação semântica do subcomponente **Type**; e finalmente a produção do valor do *sort* **reference-value**, gerado pela função semântica **the reference-value of _** que recebe os *sorts* **pointer-value** (ponteiro para o valor do *sort* **object-variable** produzido pela função semântica **instantiate object-variable of _** especificada previamente na Seção 4.4.5) e **class** (produzido na avaliação semântica do subcomponente **Type**) como argumentos.

4.4.8 Novos Comandos

Entre os componentes semânticos orientados a objetos especificados, existem alguns novos comandos que são: a chamada de métodos e a chamada de superconstrutores. Chamada de métodos, de forma semelhante a chamada de procedimentos (Seção 4.2.2), procura executar a abstração associada ao mesmo, e quando necessário, efetua-se a produção de argumentos a partir da avaliação de parâmetros atuais (Seção 4.2.3). A chamada de superconstrutores só pode ser efetuada se o mesmo for chamado no início do construtor ou do método, e serve basicamente para chamar um construtor de uma superclasse da classe do objeto que está sendo processado [21].

Portanto, segue abaixo a especificação da operação semântica **obj-commands**, responsável pela produção de componentes semânticos orientados a objetos que estendem o *namespace* **Command**:

```
obj-commands :: -> language-description.
obj-commands =
   | syntax
   |    [ i1 # * Ident "." i2 # Identifier "()" ] --> * Command
   | semantics
   |    | | semantics of i1
   |    | then
   |    | | give target-variable of (the identity of
   |    | | (the value attributed to (the given reference-variable)))
   |    then
   |    | enact application method-body of
   |    | (method token of * i2 of (the class of the given object-variable)) to
   |    | (the given object-variable)
   and
   | syntax
   |    [ i1 # * Ident "." i2 # Identifier "(" ap # * ActualParameter ")" ] --> * Command
   | semantics
   |    | | | semantics of i1
   |    | | then
   |    | | | give target-variable of (the identity of
   |    | | | (the value attributed to (the given reference-variable)))
   |    | and then
   |    | | semantics of ap
   |    then
   |    | enact application method-body of
   |    | (method token of * i2 of
   |    | (the class of (the given object-variable # 1))) to them
   and
   | syntax
   |    [ "super" "()" ] --> * Command
   | semantics
   |    enact application constructor-body of
   |    (the constructor of (the superclass of (the class of (the given object-variable # 1)))) to
   |    (the given object-variable # 1)
   and
   | syntax
   |    [ "super" "(" ap # * ActualParameter ")" ] --> * Command
   | semantics
   |    | | regive
```

```
|    | and then
|    | | semantics of ap
|    | then
|    | enact application constructor-body of
|    | (the constructor of (the superclass of (the class of (the given object-variable # 1)))) to
|    | (them).
```

O primeiro componente semântico, na seção **syntax**, é formado pelo subcomponente **Ident** e pelo elemento sintático **i2 # Identifier**; e, na seção **semantics**, efetua a avaliação semântica do subcomponente **Ident**, gerando um valor do *sort* **reference-variable**, o qual será usado para identificar o método a ser executado, e que será executado logo em seguida.

O segundo componente semântico possui a mesma estrutura do componente semântico anterior, acrescentando, na seção **syntax**, o subcomponente **ActualParameter**; e, na seção **semantics**, a avaliação semântica deste novo subcomponente antes da execução do método.

O terceiro componente semântico, na seção **syntax**, possui apenas o elemento sintático "**super**"; e, na seção **semantics**, executa o construtor da superclasse da classe da variável objeto fornecida quando da execução de um método ou construtor qualquer.

E o quarto componente semântico possui a mesma estrutura do componente anterior, acrescentando, na seção **syntax**, o subcomponente **ActualParameter**; e, na seção **semantics**, a avaliação semântica deste novo subcomponente antes da execução do construtor.

4.4.9 Agrupando Componentes Semânticos Orientados a Objetos

Para finalizar a nossa biblioteca de componentes semânticos orientado a objetos, criamos as operações semânticas **obj-local-program** e **obj-program**, as quais produzem a **language-description** formada pelo agrupamento de todos os componentes semânticos da biblioteca orientada a objetos. A diferença é que a **obj-program** também agrupa os componentes semânticos produzidos pela operação semântica **imp2-program** (Seção 4.2.11), os quais são necessários para efeitos de animação da especificação com a ferramenta ABACO [11]:

```
obj-program :: -> language-description.
obj-program =
   | imp2-program
   and
   | obj-local-program.

obj-local-program :: -> language-description.
obj-local-program =
   | obj-expressions
   and
   | obj-commands
   and
```

```
| obj-reference-expressions
and
| obj-class-types
and
| obj-class-constructor
and
| obj-class-method.
```

Na compilação de programas orientado a objetos, a operação semântica **compile** (Seção 2.2.2) irá receber como argumentos: uma **language-description**, neste caso **obj-program**; um *namespace* (**Command**) definindo que tipo de componente semântico deve ser inicialmente processado; e a árvore sintática de comandos, representando o código do programa propriamente dito.

Exemplo 4.4: Instanciando Objetos usando Superconstrutores

Como exemplo, criamos um programa orientado a objetos (**Figuras 4.10** e **4.11**), que declara: um tipo classe **t1**, formado pela variável de instância **b** do tipo inteiro e por um construtor que recebe como argumento um valor **x** que será atribuído diretamente para a variável de instância **b**; um tipo classe **t2**, que herda a classe **t1** e define um novo construtor o qual chama o construtor da superclasse e incrementa o valor da variável de instância **b**; e uma variável **x** do tipo classe **t2**.

```
t1 = class
        private
          b : integer
        public
           constructor( x : integer )
           begin
              b = x
           end
        end
    end;
t2 = class( t1 )
        private
        public
           constructor( x : integer )
           begin
              super( x );
              b = b + 1
           end
        end;
var x : t2;
begin
   x = new t2( 4 )
end
```

Figura 4.10 – Linguagem concreta do programa exemplo 4.4.

A execução do programa em si consiste apenas na declaração de uma instância de objeto da classe **t2** passando como argumento o valor **4**, e da atribuição do novo objeto alocado na variável **x** cujo tipo é compatível.

```
compile (obj-program) (* Command)
[
 [
  [ t1 "="
   [ "class"
    "private"
    [ b ":" [ "integer" ] ]
    "public"
    [
     "constructor" "(" [ x ":" [ "integer" ] ] ")"
     [
      [ ]
      "begin"
       [ [ b ] "=" [ [ x ] ] ]
      "end"
     ]
    ]
    [ ]
    "end"
   ]
  ] ";"
  [
   [ t2 "="
    [ "class" [ t1 ]
     "private"
     [ ]
     "public"
     [
      "constructor" "(" [ x ":" [ "integer" ] ] ")"
      [
       [ ]
       "begin"
       [
        [ "super" "(" [ [ [ x ] ] ] ")" ]
        ";"
        [ [ b ] "=" [ [ [ b ] ] [ "+" ] [ 1 ] ] ]
       ]
       "end"
      ]
     ]
     [ ]
     "end"
    ]
   ] ";"
   [ "var" x ":" [ t2 ] ]
  ]
 ]
 "begin"
  [ [ x ] "=" [ "new" [ t2 ] "(" [ [ 4 ] ]  ")" ] ]
 "end"
]
```

Figura 4.11 – Linguagem abstrata do programa exemplo 4.4 para uso da função semântica compile _ _ _.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 4.12** abaixo) que, quando executadas produzem como resultado final: as associações **{x → cell0, b → cell1, b → cell2, b → cell3}**, e as células de memória **{cell0 --> pointer to (object-variable of …), cell1 --> 5, cell2 --> 4, cell3 --> 4}**.

```
| furthermore
| | | | | | | give integer-type
| | | | | | then
| | | | | | | give (type-bindings of (map of "b" to (the given type) ) )
| | | | | and then
| | | | | | | give (constructor of (closure abstraction of
| | | | | | | | | furthermore
| | | | | | | | | | | bind "self" to (the given object-variable # 1)
| | | | | | | | | and
| | | | | | | | | | | generate (the variable-bindings of (the given object-variable # 1) )
| | | | | | | hence
| | | | | | | | | | | give (rest (the given data) )
| | | | | | | | | then
| | | | | | | | | | | furthermore
| | | | | | | | | | | | | | | give (first (the given data) )
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | give integer-type
| | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | allocate variable of (the given type # 1)
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | bind "x" to (the given variable # 1)
| | | | | | | | | | | | | | | before
| | | | | | | | | | | | | | | | | attribute (the given value # 1) to (the variable bound to "x")
| | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | give (rest (the given data) )
| | | | | | | | | before
| | | | | | | | | | | furthermore
| | | | | | | | | | | complete
| | | | | | | | | hence
| | | | | | | | | | | | | | give (the variable bound to "b")
| | | | | | | | | | | | | and
| | | | | | | | | | | | | | give (the variable bound to "x")
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | give (the value attributed to (the given variable) )
| | | | | | | | | | | | then
| | | | | | | | | | | | | attribute (the given value # 2) to (the given variable # 1)
| | | | | | | ) )
| | | | | | and then
| | | | | | | give empty-method-bindings
| | | | then
| | | | | give (class-of (the given type-bindings # 1) (the given method-bindings # 3) (the
given constructor # 2) empty-class)
| | | then
| | | | bind "t1" to (the given type)
| | before
| | | | | | | give (the type bound to "t1")
| | | | | | and then
| | | | | | | | give empty-type-bindings
```

Figura 4.12 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.4.

```
| | | | | | | and then
| | | | | | | | | give  (constructor of  (closure abstraction of
| | | | | | | | | | furthermore
| | | | | | | | | | | | bind "self" to  (the given object-variable # 1)
| | | | | | | | | | and
| | | | | | | | | | | | generate  (the variable-bindings of  (the given object-variable # 1) )
| | | | | | | | | hence
| | | | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | | | then
| | | | | | | | | | | | furthermore
| | | | | | | | | | | | | | | give  (first  (the given data) )
| | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | give integer-type
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | allocate variable of  (the given type # 1)
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | bind "x" to  (the given variable # 1)
| | | | | | | | | | | | | | | | before
| | | | | | | | | | | | | | | | | attribute  (the given value # 1)  to  (the variable bound to "x")
| | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | | | before
| | | | | | | | | | | | furthermore
| | | | | | | | | | | | | complete
| | | | | | | | | | hence
| | | | | | | | | | | | | | | | regive
| | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | enact  (application  (constructor-body of  (the constructor of  (the
superclass of  (the class of  (the given object-variable # 1) ) ) ) )  to  (the given data) )
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the variable bound to "b")
| | | | | | | | | | | | | | | and
| | | | | | | | | | | | | | | | | give "PLUS"
| | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | give  (the variable bound to "b")
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)  (the given value # 2)  (the
given value # 3)
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | attribute  (the given value # 2)  to  (the given variable # 1)
| | | | | | | | | ) )
| | | | | | | | and then
| | | | | | | | | give empty-method-bindings
| | | | | then
| | | | | | give  (class-of  (the given type-bindings # 2)  (the given method-bindings # 4)  (the
given constructor # 3)  (the given class # 1) )
| | | | then
| | | | | bind "t2" to  (the given type)
```

Figura 4.12 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.4.

```
| | | before
| | | | | give  (the type bound to "t2")
| | | | then
| | | | | | allocate variable of  (the given type)
| | | | | then
| | | | | | bind "x" to  (the given variable)
hence
| | | give  (the variable bound to "x")
| | and
| | | | | give  (the type bound to "t2")
| | | | then
| | | | | | | | regive
| | | | | | and then
| | | | | | | | instantiate object-variable of  (the given class # 1)
| | | | | and then
| | | | | | give 4
| | | then
| | | | | enact  (application  (constructor-body of  (the constructor of  (the given class # 1) ) )  to
(rest  (the given data) ) )
| | | | hence
| | | | | give  (the reference-value of  (pointer to  (the given object-variable # 2) )  (the given
class # 1) )
| then
| | attribute  (the given value # 2)  to  (the given variable # 1)
```

Figura 4.12 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 4.4.

As células de memória cell2 e cell3 são resíduos das chamadas dos construtores, os quais alocam as variáveis de instância locais ao escopo do construtor mas não conseguem liberá-las ao final da execução dos mesmos. Este problema será resolvido com a extensão da biblioteca de componentes semânticos, através da inclusão de novos componentes semânticos com capacidade de liberação de memória ou um componente semântico para representar o conceito de garbage collector.

□

Capítulo 5

Uma Linguagem Multiparadigma

Neste capítulo apresentaremos uma linguagem multiparadigma, denominada EIFOO, formada pela junção dos diversos componentes semânticos, especificados na Seção 3.4 e no Capítulo 4, representando conceitos de linguagens de expressões, imperativas, funcionais e orientada a objetos. Faremos também uma breve descrição sobre o conceito de integração de paradigmas, e apresentaremos as dificuldades envolvidas na integração dos componentes semânticos especificados para cada paradigma. No final mostraremos alguns exemplos de programas-fonte para a linguagem EIFOO final produzida.

5.1 Integração de Paradigmas

Linguagens multiparadigmas procuram fornecer uma determinada variedade de estilos de linguagens de programação, permitindo assim uma mistura de construções de diversos paradigmas distintos [23]. A principal vantagem da integração de vários paradigmas em uma única linguagem de programação se encontra na combinação das facilidades dos mesmos, aumentando assim o domínio de aplicação da linguagem final resultante [20].

Diferentes tipos de tarefas podem ser melhor implementadas em diferentes paradigmas. Por exemplo, o paradigma de programação lógico é particularmente bem formatado para implementar sistemas especialistas, enquanto que muitas operações em listas podem ser elegantemente descritas no paradigma de programação funcional. Programação multiparadigma permite que cada parte de um sistema possa ser implementado no paradigma melhor formatado ao problema [23].

Entre os problemas encontrados para atingir uma melhor formatação temos: acomodação de diferentes notações sintáticas, acomodação de diferentes modelos de execução, suporte a diferentes estratégias de implementação, habilidade no uso de ferramentas existentes, e mistura e casamento de paradigmas arbitrários.

Dois bons exemplos de linguagens multiparadigmas são as linguagens LIFE e LEDA. LIFE (Logic, Inheritance, Functions, Equations) é uma linguagem de programação experimental com uma poderosa facilidade de representação de herança de tipos estruturados [25]. Leda é uma linguagem de programação multiparadigma, cujas técnicas suportadas incluem programação imperativa, abordagem orientada a objetos, programação lógica e funcional [26].

Hoje em dia, existe uma forte tendência para a integração de paradigmas de linguagens de programação. Por conta disso, várias ferramentas de programação já foram e continuam sendo desenvolvidas, com o objetivo de fornecer subsídios, ao programador/desenvolvedor comum, para se trabalhar com essas novas linguagens, garantindo assim uma gama de possibilidades no desenvolvimento de novos sistemas. Alguns bons exemplos de ferramentas que já trabalham com linguagens multiparadigmas são: compiladores C++ [31] (gcc, C++ Builder, etc.) e a ferramenta Delphi [22], [30], ambas combinando os conceitos dos paradigmas imperativo e OO.

A integração de paradigmas deve ser conduzida com muita cautela, para que não se viole os princípios básicos de cada um deles. Entretanto, com o uso da biblioteca de componentes semânticos, especificada nesta dissertação, não teremos maiores problemas de integração, uma vez que estes conceitos estão bem definidos e encapsulados.

5.2 Integração de Bibliotecas

Diferentemente da semântica de ações modular [4], a semântica de ações baseada em componentes [5], [6] não apresenta dificuldades na composição dos seus elementos. Prova disso é o nível de integração e dependência alcançado entre os componentes semânticos especificados para cada paradigma. Um bom exemplo para esta afirmação são os componentes semânticos do paradigma orientado a objetos (Seção 4.4), os quais se apresentam como uma extensão do paradigma imperativo (Seções 4.1 e 4.2), e de expressões (Seção 3.4), cujos componentes semânticos também foram especificados nesta dissertação.

Como o nosso objetivo é criar uma linguagem multiparadigma, precisamos incorporar todos os paradigmas previamente especificados. Os componentes semânticos orientados a objetos (Seção 4.4) já incorporam os componentes semânticos imperativos e de expressões, falta agora incorporar os componentes semânticos funcionais, gerando assim a linguagem híbrida EIFOO. Entretanto, como os componentes semânticos orientados a objetos e funcionais já incorporam os componentes semânticos de expressões, teríamos uma

redundância de especificações, o que nos obrigou a criar operações semânticas que produzem componentes semânticos restritos a cada paradigma.

De fato, se analisarmos os componentes semânticos de cada paradigma (Capítulo 4), veremos que cada um produz duas linguagens: a primeira representando apenas os componentes semânticos do respectivo paradigma (**imp-local-program**, na seção 4.1, por exemplo), e a segunda representando os componentes semânticos do próprio paradigma e o do paradigma antecessor (**imp-program**, na seção 4.1, por exemplo). Esta última operação semântica serviria apenas para fins de animação com o ABACO [11], através da produção de componentes semânticos básicos necessários para uma mínima capacidade de execução de programas no paradigma propriamente dito. Como os componentes semânticos de expressões (Seção 3.4) representam apenas conceitos básicos, eles serão produzidos apenas por uma única operação semântica (**exp-program**), já que não possuem dependência com outros paradigmas.

Seguindo os conceitos mostrados previamente (nas Seções 3.4, 4.1, 4.2, 4.3 e 4.4), criamos a operação semântica **all-paradigms,** que produz um indivíduo do *sort* **language-description** formado pela combinação dos resultados das operações semânticas produtoras de componentes semânticos (**exp-program**, **imp-local-program**, **imp2-local-program**, **fnc-local-program** e **obj-local-program**) restritos à cada paradigma:

```
all-paradigms :: -> language-description.
all-paradigms =
   | exp-program
   and
   | imp-local-program
   and
   | imp2-local-program
   and
   | fnc-local-program
   and
   | obj-local-program.
```

Será através desta operação semântica que iremos produzir o agrupamento de todos os componentes semânticos necessários para a animação de programas-fonte baseados na linguagem EIFOO.

5.3 A Linguagem EIFOO

Como mostrado anteriormente, a operação semântica **all-paradigms** produz um indivíduo do *sort* **language-description** capaz de representar todos os conceitos básicos envolvidos pelos paradigmas especificados nesta dissertação. Vamos chamar essa nova linguagem de EIFOO, a qual incorpora os conceitos de Expressões (valores, operações unárias e binárias, etc.); os conceitos Imperativos (declaração de variáveis, alocação de

memória, comandos, etc.) e imperativos complexos (arrays, records, ponteiros, etc.); os conceitos Funcionais (funções, valor função, etc.); e, finalmente, os conceitos Orientado a Objetos (classes, objetos, referência a objetos, etc.).

De fato, a operação semântica **all-paradigms** produz o agrupamento de todos os componentes semânticos, incluindo componentes semânticos que podem não ser desejados pelo projetista de linguagens em um determinado momento.

Portanto, na medida em que se estende a biblioteca de componentes semânticos, torna-se cada vez mais interessante a produção de componentes semânticos por operações semânticas individuais, ao invés de operações semânticas produtoras de grandes agrupamentos de componentes semânticos. Conseqüentemente, o projetista de linguagens poderá indicar, de uma maneira mais simples, que componentes semânticos serão realmente interessantes para a sua linguagem customizada.

Vamos mostrar agora alguns exemplos de programas-fonte baseado na linguagem EIFOO:

Exemplo 5.1: Utilizando Funções na Instanciação de Objetos

Como exemplo, criamos um programa baseado na linguagem EIFOO (**Figuras 5.1** e **5.2**), que declara: a função **fatorial**, para representar o algoritmo recursivo de cálculo de fatorial; o tipo classe **t**, contendo um construtor e um método de incremento para as suas respectivas variáveis de instância; e a variável de referência **x**, a qual irá apontar para uma instância do tipo **t**.

Na seção de comandos temos primeiramente a execução de uma atribuição, a qual procura atualizar o valor da variável **x**, declarada previamente, com o valor de uma nova referência. Esta nova referência aponta para uma variável objeto, a qual foi instanciada dinamicamente pela operação **new**, que recebe como parâmetro o tipo classe **t**. Em seguida o programa executa uma chamada do método de incremento, passando como parâmetro a função **fatorial** com um argumento igual a **3**. Como uma função é um valor, esta operação é totalmente válida.

```
begin
   val rec fatorial ==  fn x ==>
   begin
      if ( x <> 1)
      then
         x * fatorial ( x - 1)
      else 1
   end;
   t = class
      private
         a : integer
      public
         constructor()
         begin
            a = 1
         end
         procedure incrementa( x : integer )
         begin
            a = a + x
         end
       end;
    var x : t
   begin
      x = new t();
      x.incrementa(fatorial 3)
   end
end
```

Figura 5.1 – Linguagem concreta do programa exemplo 5.1.

```
compile (all-paradigms) (* Command)
[
  [
    [
      "val" "rec" fatorial "==" "fn" [ x ] "==>"
      [
        "if" [ [ x ] [ "<>" ] [ 1 ] ]  "then"
        [
          [ x ] [ "*" ] [ [ fatorial ] [ [ [ x ] [ "-" ] [ 1 ] ] ] ]
        ]
        "else" [ 1 ]
      ]
    ]  ";"
```

Figura 5.2 – Linguagem abstrata do programa exemplo 5.1 para uso da função semântica compile _ _ _.

```
  [
   [ t "="
    [ "class"
     "private"
     [ a ":" [ "integer" ] ]
     "public"
     [
      "constructor" "()"
      [
       [ ]
       "begin"
        [ [ a ] "=" [ 1 ] ]
       "end"
      ]
     ]
     [
      "procedure" incrementa "(" [ x ":" [ "integer" ] ] ")"
      [
       [ ]
       "begin"
        [ [ a ] "=" [ [ [ a ] ] [ "+" ] [ [ x ] ] ] ]
       "end"
      ]
     ]
     "end"
    ]
   ]
   ";"
   [ "var" x ":" [ t ] ]
  ]
 ]
 "begin"
 [
  [ [ x ] "=" [ "new" [ t ] "()" ] ]
  ";"
  [ [ x ] "." incrementa "(" [ [ [ fatorial ] [ [ 3 ] ] ] ] ")" ]
 ]
 "end"
]
```

Figura 5.2 (cont.) – Linguagem abstrata do programa exemplo 5.1 para uso da função semântica compile _ _ _.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 5.3** abaixo) que, quando executadas produzem: uma célula de memória denominada **c1**, armazenando uma referência para uma variável objeto, composta pela definição de uma classe e por um mapeamento do identificador **a** para a célula de memória **c2**; outra célula de memória denominada **c2**, armazenando o valor **7**, resultante da avaliação da função fatorial com o argumento **3** acrescido de **1** pelo método incrementa; e as associações **{fatorial → function of ... , t → class-of ... , x → the reference-variable of ... }**. Valores transitórios não são produzidos.

```
| furthermore
| | | recursively bind "fatorial" to  (function of  (closure abstraction of
| | | | | furthermore
| | | | | | | | give  (first  (the given data) )
| | | | | | | then
| | | | | | | | bind "x" to  (the given value)
| | | | | | and then
| | | | | | | give  (rest  (the given data) )
| | | | hence
| | | | | | | | give "DIFF"
| | | | | | and then
| | | | | | | | | give  (the value bound to "x")
| | | | | | | | or
| | | | | | | | | give  (the function bound to "x")
| | | | | | | and then
| | | | | | | | give 1
| | | | | then
| | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)   (the given
value # 3)
| | | | then
| | | | | | | check  ( (the given datum)  is true)
| | | | | | and then
| | | | | | | | | give "MULT"
| | | | | | | | and then
| | | | | | | | | | | give  (the value bound to "x")
| | | | | | | | | | or
| | | | | | | | | | | give  (the function bound to "x")
| | | | | | | | | and then
| | | | | | | | | | | | | give  (the value bound to "fatorial")
| | | | | | | | | | | or
| | | | | | | | | | | | | give  (the function bound to "fatorial")
| | | | | | | | | | and
| | | | | | | | | | | | | | | | | give "MINUS"
| | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | give  (the value bound to "x")
| | | | | | | | | | | | | | | | | or
| | | | | | | | | | | | | | | | | | give  (the function bound to "x")
| | | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)
(the given value # 3)
| | | | | | | | | | | | | or
| | | | | | | | | | | | | | give "MINUS"
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the value bound to "x")
| | | | | | | | | | | | | | | or
| | | | | | | | | | | | | | | | give  (the function bound to "x")
| | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)
(the given value # 3)
| | | | | | | | | | then
| | | | | | | | | | | enact  (application  (function-body of  (the given function # 1) )  to  (rest
(the given data) ) )
```

Figura 5.3 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.1.

```
| | | | | | | | then
| | | | | | | | | binary-operation-of  (the given string # 1)  (the given value # 2)  (the given
value # 3)
| | | | | | or
| | | | | | | | check  ( (the given datum)  is false)
| | | | | | | and then
| | | | | | | | give 1
| | | ) )
| | before
| | | | | | | give integer-type
| | | | | | then
| | | | | | | give  (type-bindings of  (map of "a" to  (the given type) ) )
| | | | | and then
| | | | | | | give  (constructor of  (closure abstraction of
| | | | | | | | | furthermore
| | | | | | | | | | | bind "self" to  (the given object-variable # 1)
| | | | | | | | | and
| | | | | | | | | | | generate  (the variable-bindings of  (the given object-variable # 1) )
| | | | | | | | hence
| | | | | | | | | furthermore
| | | | | | | | | | complete
| | | | | | | | | before
| | | | | | | | | | | give  (the variable bound to "a")
| | | | | | | | | | and
| | | | | | | | | | | give 1
| | | | | | | | | then
| | | | | | | | | | | attribute  (the given value # 2)  to  (the given variable # 1)
| | | | | | | | ) )
| | | | | | | and then
| | | | | | | | give  (method-bindings of  (map of "incrementa" to  (method of  (closure
abstraction of
| | | | | | | | | furthermore
| | | | | | | | | | generate  (the variable-bindings of  (the given object-variable # 1) )
| | | | | | | | | and
| | | | | | | | | | bind "self" to  (the given object-variable # 1)
| | | | | | | | hence
| | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | | then
| | | | | | | | | | furthermore
| | | | | | | | | | | | give  (first  (the given data) )
| | | | | | | | | | | | then
| | | | | | | | | | | | | | give integer-type
| | | | | | | | | | | | | | then
| | | | | | | | | | | | | | allocate variable of  (the given type # 1)
| | | | | | | | | | | | | | then
| | | | | | | | | | | | | | bind "x" to  (the given variable # 1)
| | | | | | | | | | | | | | before
| | | | | | | | | | | | | | attribute  (the given value # 1)  to  (the variable bound to "x")
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | | | | | before
| | | | | | | | | | | | | furthermore
| | | | | | | | | | | | | complete
| | | | | | | | | | | | before
| | | | | | | | | | | | | | give  (the variable bound to "a")
```

Figura 5.3 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.1.

```
| | | | | | | | | | | | | and
| | | | | | | | | | | | | | | give "PLUS"
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the variable bound to "a")
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)
(the given value # 3)
| | | | | | | | | | | then
| | | | | | | | | | | | attribute  (the given value # 2)  to  (the given variable # 1)
| | | | | | | ) ) ) )
| | | | | then
| | | | | give  (class-of  (the given type-bindings # 1)   (the given method-bindings # 3)
(the given constructor # 2)  empty-class)
| | | | then
| | | | | bind "t" to  (the given type)
| | | before
| | | | | give  (the type bound to "t")
| | | | then
| | | | | | allocate variable of  (the given type)
| | | | | then
| | | | | | bind "x" to  (the given variable)
before
| | | | give  (the variable bound to "x")
| | | and
| | | | | | give  (the type bound to "t")
| | | | | then
| | | | | | | regive
| | | | | | and then
| | | | | | | instantiate object-variable of  (the given class # 1)
| | | | then
| | | | | | enact  (application  (constructor-body of  (the constructor of  (the given class #
1) ) )  to  (the given object-variable # 2) )
| | | | | hence
| | | | | | give  (the reference-value of  (pointer to  (the given object-variable # 2) )   (the
given class # 1) )
| | then
| | | attribute  (the given value # 2)  to  (the given variable # 1)
| and then
| | | | | give  (the variable bound to "x")
| | | | then
| | | | | give  (target-variable of  (the identity of  (the value attributed to  (the given
reference-variable) ) ) )
| | | and then
| | | | | | | | give  (the value bound to "fatorial")
| | | | | | | or
| | | | | | | | give  (the function bound to "fatorial")
| | | | | | and
| | | | | | | | give 3
| | | | | | | or
| | | | | | | | give 3
```

Figura 5.3 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.1.

```
| | | | | then
| | | | | | enact (application (function-body of (the given function # 1) ) to (rest (the
given data) ) )
| | | | or
| | | | | | | | give (the value bound to "fatorial")
| | | | | | | or
| | | | | | | | give (the function bound to "fatorial")
| | | | | | and
| | | | | | | | give 3
| | | | | | | or
| | | | | | | | give 3
| | | | | then
| | | | | | enact (application (function-body of (the given function # 1) ) to (rest (the
given data) ) )
| | then
| | | enact (application (method-body of (method "incrementa" of (the class of (the
given object-variable # 1) ) ) ) to (the given data) )
```

Figura 5.3 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.1.

Podemos observar que, no resultado da compilação do programa, existem ações repetidas, agregadas pelo combinador de ações **or**. Trata-se de um conflito de sintaxe, entre os componentes especificados para a representação de parâmetros atuais e formais, nas bibliotecas funcional e imperativa complexa. Este problema não afeta o processamento do programa, e pode ser facilmente resolvido através de uma melhor combinação de componentes semânticos realmente necessários para o processamento da linguagem customizada.

□

Vamos agora criar um programa exemplo (**Figura 5.4**) baseado na linguagem EIFOO, que mostre a definição de métodos funcionais dentro de objetos. Para isso, é necessário estender o *namespace* **MethodDec** da biblioteca de componentes semânticos orientados a objetos (Seção 4.4.3). Portando, especificamos a operação semântica **all-paradigms2**, que produz os componentes semânticos da operação semântica **all-paradigms** acrescentando os componentes semânticos que representam a declaração e a chamada de métodos função:

```
all-paradigms2 :: -> language-description.
all-paradigms2 =
   | all-paradigms
   and
   | syntax
   |    [ "function" i # Identifier "(" fp # * FormalParameter ")" e # * Expression ] --> *
MethodDec
   | semantics
   |    give method-bindings of (map of token of * i to method of
```

```
|    closure abstraction of
|      | | furthermore
|      | | | | generate the variable-bindings of (the given object-variable # 1)
|      | | | and
|      | | | | bind "self" to (the given object-variable # 1)
|      | hence
|      | | | | give rest the given data
|      | | | then
|      | | | | furthermore semantics of fp
|      | | before
|      | | | semantics of e
|    )
and
| syntax
|    [ e # * Expression "." i # Identifier "(" ap # * ActualParameter ")" ] --> * Expression
| semantics
|    | | | semantics of e
|    | | then
|    | | | give target-variable of (the identity of (the given reference-value))
|    | and then
|    | | semantics of ap
|    then
|    | enact application method-body of
|    | (method token of * i of
|    | (the class of (the given object-variable # 1))) to them.
```

O primeiro componente semântico, na seção **syntax**, é formado pelos subcomponentes **FormalParameter** e **Expression** e pelo elemento sintático **i # Identifier**; e, na seção **semantics**, produz um valor do *sort* **method-bindings**, a partir do mapeamento formado entre valor do elemento sintático **i # Identifier** e o valor do *sort* **method**, cuja abstração é formada: pela geração das associações das variáveis de instância da variável objeto atual usando a função semântica **generate _**; pela associação do elemento sintático "**self**" a variável objeto atual; pela avaliação semântica do subcomponente **FormalParameter**, gerando parâmetros formais (Seção 4.2.4); e pela avaliação semântica do subcomponente **Expression**, respectivamente.

O segundo componente semântico, na seção **syntax**, é formado pelos subcomponentes **Expression** e **ActualParameter** e pelo elemento sintático **i # Identifier**; e, na seção **semantics**, efetua a avaliação semântica do subcomponente **Expression**, gerando um valor do *sort* **reference-value**, o qual será usado para identificar o método a ser executado (através do uso do elemento sintático **i # Identifier**) logo após a avaliação semântica do subcomponente **ActualParameter**, gerando parâmetros atuais (Seção 4.2.3).

Exemplo 5.2: Utilizando Funções como Métodos de Objetos

Como exemplo, criamos um programa baseado na linguagem EIFOO (**Figuras 5.4** e **5.5**), que declara: o tipo classe **t**, contendo um construtor e um método função denominado

fatorial2, o qual representa o algoritmo de cálculo de fatorial; e a variável de referência **x**, a qual irá apontar para uma instância do tipo **t**.

```
begin
      t = class
         private

         public
            constructor()

            function fatorial2 (x : integer)
               if (x <> 1)
               then
                  x * self.fatorial2( x - 1 )
               else 1
         end;

      var x : t
   begin
      x = new t() ;
      x.fatorial2( 4 )
   end
end
```

Figura 5.4 – Linguagem concreta do programa exemplo 5.2.

```
compile (all-paradigms2) (* Command)
[ [
   [ t "="
     [ "class"
       "private"
       [ ]
       "public"
       [  "constructor" "()"  [ ] ]
       [
         "function" fatorial2 "(" [ x ":" [ "integer" ] ] ")"
         [
           "if" [ [ [ x ] ] [ "<>" ] [ 1 ] ]  "then"
           [
             [ [ x ] ]  [ "*" ]  [  [ "self" ] "." fatorial2 "(" [ [ [ [ x ] ] [ "-" ] [ 1 ] ] ] ")" ]
           ]
           "else" [ 1 ]
         ]
       ]
       "end"
     ]
   ] ";" [ "var" x ":" [ t ] ]
 ]
 "begin"  [
   [ [ x ] "=" [ "new" [ t ] "()" ] ]
   ";"
   [ [ x ] "." fatorial2 "(" [ [ 4 ] ] ")" ]
 ]
 "end"
]
```

Figura 5.5 – Linguagem abstrata do programa exemplo 5.2 para uso da função semântica compile _ _ _.

Na seção de comandos temos primeiramente a execução de uma atribuição, a qual procura atualizar o valor da variável **x**, declarada previamente, com o valor de uma nova referência. Esta nova referência aponta para uma variável objeto, a qual foi instanciada dinamicamente pela operação **new**, que recebe como parâmetro o tipo classe **t**. Em seguida o programa executa uma chamada do método de função **fatorial2**, passando como parâmetro um argumento igual a **4**.

O resultado da avaliação desta expressão semântica pela ferramenta ABACO [11] é um conjunto de ações (as quais podem ser vistas na **Figura 5.6**) que, quando executadas produzem: um valor transitório **24**, representando o valor resultante da chamada do método função; as células de memória **{cell0 → pointer to (object-variable of ...), cell1 → 4, cell2 → 3, cell3 → 2, cell4 → 1}**; e a associação **{"t" → class-of ...}**.

```
| furthermore
| | | | | give empty-type-bindings
| | | | and then
| | | | | | give  (constructor of  (closure abstraction of
| | | | | | | | furthermore
| | | | | | | | | | bind "self" to  (the given object-variable # 1)
| | | | | | | | and
| | | | | | | | | | generate  (the variable-bindings of  (the given object-variable # 1) )
| | | | | | | hence
| | | | | | | | complete
| | | | | | | ) )
| | | | | and then
| | | | | | give  (method-bindings of  (map of "fatorial2" to  (method of  (closure
abstraction of
| | | | | | | | furthermore
| | | | | | | | | | generate  (the variable-bindings of  (the given object-variable # 1) )
| | | | | | | | and
| | | | | | | | | | bind "self" to  (the given object-variable # 1)
| | | | | | | hence
| | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | then
| | | | | | | | | | furthermore
| | | | | | | | | | | | | | give  (first  (the given data) )
| | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give integer-type
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | allocate variable of  (the given type # 1)
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | bind "x" to  (the given variable # 1)
| | | | | | | | | | | | | | | | before
| | | | | | | | | | | | | | | | attribute  (the given value # 1)  to  (the variable bound to "x")
| | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (rest  (the given data) )
| | | | | | | | | | | | before
| | | | | | | | | | | | | | give "DIFF"
```

Figura 5.6 – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.2.

```
| | | | | | | | | | | | and then
| | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | then
| | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | and then
| | | | | | | | | | | | give 1
| | | | | | | | | | | then
| | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)
(the given value # 3)
| | | | | | | | | | then
| | | | | | | | | | | | check  ( (the given datum)  is true)
| | | | | | | | | | | and then
| | | | | | | | | | | | | give "MULT"
| | | | | | | | | | | | and then
| | | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give  (the object-variable bound to "self")
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give  (the reference-value of  (pointer to  (the given object-
variable) )   (the class of  (the given object-variable) ) )
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give  (target-variable of  (the identity of  (the given
reference-value) ) )
| | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | give "MINUS"
| | | | | | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given
value # 2)   (the given value # 3)
| | | | | | | | | | | | | | | | or
| | | | | | | | | | | | | | | | | | | | | | give "MINUS"
| | | | | | | | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | | | | | give  (the variable bound to "x")
| | | | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | | | | give  (the value attributed to  (the given variable) )
| | | | | | | | | | | | | | | | | | | | | | and then
| | | | | | | | | | | | | | | | | | | | | | | | give 1
| | | | | | | | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given
value # 2)   (the given value # 3)
| | | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | enact  (application  (method-body of  (method "fatorial2" of
(the class of  (the given object-variable # 1) ) ) )  to  (the given data) )
| | | | | | | | | | | | then
| | | | | | | | | | | | binary-operation-of  (the given string # 1)   (the given value # 2)
(the given value # 3)
| | | | | | | | | | | or
| | | | | | | | | | | | check  ( (the given datum)  is false)
```

Figura 5.6 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.2.

```
| | | | | | | | | | | | and then
| | | | | | | | | | | | | give 1
| | | | | | | ) ) ) )
| | | | then
| | | | | give  (class-of  (the given type-bindings # 1)   (the given method-bindings #
3)  (the given constructor # 2)  empty-class)
| | | then
| | | | bind "t" to  (the given type)
| | before
| | | | give  (the type bound to "t")
| | | then
| | | | | allocate variable of  (the given type)
| | | | then
| | | | | bind "x" to  (the given variable)
before
| | | | give  (the variable bound to "x")
| | | and
| | | | | | give  (the type bound to "t")
| | | | | then
| | | | | | | regive
| | | | | | and then
| | | | | | | instantiate object-variable of  (the given class # 1)
| | | | then
| | | | | | enact  (application  (constructor-body of  (the constructor of  (the given
class # 1) ) )  to  (the given object-variable # 2) )
| | | | | hence
| | | | | | give  (the reference-value of  (pointer to  (the given object-variable # 2) )
(the given class # 1) )
| | then
| | | attribute  (the given value # 2)  to  (the given variable # 1)
| and then
| | | | | give  (the variable bound to "x")
| | | | then
| | | | | give  (target-variable of  (the identity of  (the value attributed to  (the given
reference-variable) ) ) )
| | | and then
| | | | | give 4
| | | | or
| | | | | give 4
| | then
| | | enact  (application  (method-body of  (method "fatorial2" of  (the class of  (the
given object-variable # 1) ) ) )  to  (the given data) )
```

Figura 5.6 (cont.) – Resultado da avaliação da função semântica compile _ _ _ recebendo como argumento o programa Exemplo 5.2.

Vale salientar que as células de memória **cell1**, **cell2**, **cell3** e **cell4** são alocadas pelas chamadas recursivas da função **fatorial2**, na declaração de parâmetros formais, as quais não são liberadas no final da execução do bloco funcional.

□

Capítulo 6

Conclusões e Trabalhos Futuros

Diante do que foi apresentado, concluímos que a especificação de linguagens de programação usando componentes semânticos é extremamente viável, seja para a representação de conceitos simples (expressões e atribuições), seja para a representação de conceitos complexos (classes e objetos).

Conseguimos desenvolver uma biblioteca de componentes semânticos capaz de representar os vários conceitos de linguagens de programação, mas não todos, existentes nos paradigmas citados nesta dissertação; e mostramos também que não há mais necessidade de se especificar semânticas de ações para novas linguagens de programação a partir do zero, pois os componentes semânticos produzidos podem ser perfeitamente reutilizados na definição destas novas linguagens.

Também conseguimos não apenas representar os vários conceitos de linguagens de programação existentes nos paradigmas citados nesta dissertação, mas também representá-los de forma hierárquica, com um nível muito alto de reutilização semântica, o que servirá como um guia para futuras especificações de novos módulos na semântica de ações modular e de novos objetos na semântica de ações orientada a objetos.

6.1 Contribuições

Podemos sumarizar nossa contribuição a partir dos seguintes itens:

- O ganho no projeto de novas linguagens de programação é visível, tanto no nível de leitura/manutenção (componentes legíveis e de fácil entendimento), como também no nível de velocidade e qualidade na produção de novas especificações (benefícios da reusabilidade e componentes bem definidos para o seu domínio de aplicação);
- A simplificação da representação de conceitos de linguagens de programação, usando trechos já definidos em especificações anteriores, garante um bom nível de

"corretude" semântica, uma vez que esses trechos foram descritos em especificações previamente consolidadas na literatura [11], [12], [13], [14], [15], [17], [18];

- Os componentes semânticos produzidos são totalmente extensíveis, não apenas devido aos princípios da semântica de ações baseada em componentes (que garantem uma descrição menos acoplada, mais independente e legível), mas também devido à representação dos conceitos de linguagens de programação de uma forma hierárquica e reutilizável;
- Mesmo sem a capacidade de representar um comportamento semelhante a uma hierarquia de classes e objetos, a biblioteca de componentes semânticos se mostrou capaz de representar, de forma bastante satisfatória, a hierarquia de conceitos de linguagens de programação proposta em [20], provando assim que se trata de um meio bastante viável para a representação encapsulada de semânticas de ações;
- Validamos a ferramenta ABACO, provando que se trata de um sistema confiável e extremamente útil no desenvolvimento e na animação de especificações de linguagens de programação, tanto na semântica de ações tradicional como na semântica de ações baseada em componentes. De fato, a biblioteca de componentes semânticos final produzida nesta dissertação pode ser considerada a maior especificação em semântica de ações já produzida na ferramenta ABACO;
- Os níveis de reusabilidade atingidos na especificação de cada paradigma são gratificantes. Considerando apenas a reutilização de componentes semânticos (num total de: 19 componentes de expressões, 12 componentes imperativos, 23 componentes imperativos complexo, 10 componentes funcionais e 22 componentes orientado a objetos), e ignorando o reaproveitamento de funções e entidades semânticas, temos as seguintes porcentagens de reuso de sistemas: imperativo (61,23%), imperativo complexo (57,41%), funcional (65,52%) e orientado a objetos (71,05%). Estes índices foram obtidos através da fórmula para o cálculo de reusabilidade de sistemas orientado a objetos: $R_{lev} = OBJ_{reused} / OBJ_{built}$, definida em [35], onde R_{lev} representa.o nível de reuso, OBJ_{reused} representa o número de objetos reusado no sistema, e OBJ_{built} representa o número de objetos construídos no sistema;
- Vários casos de teste foram criados para cada paradigma, e animados pela ferramenta ABACO, com o objetivo de validar a biblioteca de componentes semânticos final produzida. Foram criados 8 testes para componentes de expressões, 5 testes para componentes imperativos, 23 testes para componentes imperativos

complexos, 5 testes para componentes funcionais e 9 testes para componentes orientados a objetos. Em todos eles, apenas 1 caso de teste não funcionou devido a limitações já conhecidas da biblioteca de componentes semânticos, mais precisamente na representação de variáveis de instância pelos componentes orientados a objetos.

6.2 Trabalhos Futuros

Entre as futuras atividades possíveis de continuação desta pesquisa, podemos destacar: (1) a extensão da biblioteca de componentes semânticos; (2) a criação de uma biblioteca de componentes semânticos para verificação de tipos nas linguagens em tempo de compilação; (3) a especificação de componentes semânticos para novos paradigmas; e (4) a especificação de uma semântica de ações para UML, conforme detalhes:

1. Infelizmente nem todos os diversos conceitos de linguagens de programação existentes foram especificados, como por exemplo: procedimentos com escopo não-aninhado, registros variantes, modificadores de acesso a classes, etc., porque foi definida apenas uma biblioteca inicial de componentes semânticos. Portanto é necessário estender essa biblioteca, seja através de novas teses e artigos, seja através da aplicação da mesma em projetos de sala de aula (projetos finais de disciplinas, provas, etc.);
2. Atualmente, a verificação de tipos nas linguagens de programação, usando a biblioteca de componentes semânticos, ocorre apenas em tempo de execução, de forma que quando uma ação falha, este problema pode ser relacionado a uma incompatibilidade de tipos ou não. Para evitar esse problema, já que fica muito difícil para o programador encontrar o possível defeito, é necessário desenvolver uma biblioteca de componentes semânticos que seja capaz de efetuar uma checagem de tipos no código desejado de uma linguagem de programação específica, e esta biblioteca de tipos seria avaliada antes da biblioteca de execução do código propriamente dita, ou seja, em tempo de compilação;
3. Além de estender a biblioteca de componentes semânticos na representação de novos conceitos para os paradigmas trabalhados nesta dissertação, é necessário especificar componentes semânticos para representar conceitos de novos paradigmas tais como o paradigma lógico, o paradigma concorrente, etc., aumentando assim o domínio de aplicação da biblioteca de componentes semânticos como um todo;

4. Uma das pretensões iniciais desta dissertação era especificar uma semântica de ações para UML (Unified Modeling Language) [34] que procura unificar os principais métodos envolvidos na modelagem de sistemas. Entretanto, por razões discutidas na Seção 2.2, era necessário melhorar a reusabilidade semântica para especificar a mesma, já que: não haveria tempo suficiente para representá-la na forma usual (2 anos de mestrado); e o reuso semântico seria quase que uma obrigação, uma vez que já existe uma semântica semi-formal para UML. Com a finalização desta dissertação, torna-se viável especificar uma semântica de ações para uma linguagem como UML. Uma vez que podemos agrupar componentes semânticos específicos, produzindo uma nova linguagem com conceitos de linguagens de programação totalmente diversos e desejados pelo desenvolvedor UML, restaria apenas: **(i)** transformar os diagramas UML em um código-texto legível para os componentes semânticos; e **(ii)** definir uma nova linguagem, usando componentes semânticos previamente definidos, para dar um significado ao código gerado pela transformação.

REFERÊNCIAS BIBLIOGRÁFICAS

[1]. Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

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

[3]. D. A. Schmidt. *Denotational Semantics*. Allyn & Bacon, 1986.

[4]. Kyung-Goo Doh and Peter D. Mosses. *Composing programming languages by combining action semantics modules*. First Workshop on Language Descriptions, Tools and Applications. Elsevier Science, April 2001.

[5]. Luis Menezes and Hermano Moura, *Component-based Action Semantics: A new Approach for Programming Language Specifications*, V Simpósio Brasileiro de Linguagens de Programação, pages 152-163, Federal University of Paraná, 2001.

[6]. Luis Menezes, *Presentation - Modular Language Definitions for Action Semantics*, Center of Informatics, Federal University of Pernambuco, 2002.

[7]. Cláudio Carvilhe and Martin Musicante, *Semântica de Ações Orientada a Objetos*, Federal University of Paraná, 2002.

[8]. R.D. Tennet, *Principles of Programming Languages*, Ed. Prentice-Hall, 1981.

[9]. Robert W. Sebesta, *Concepts of Programming Languages 4º Edition*, Ed. Addison-Wesley, 1998.

[10]. David A. Watt, *Programming Language Concepts and Paradigms*, Ed. Prentice-Hall, 1991.

[11]. Luis C. S. Menezes and Hermano P. Moura, *Uso de Orientação a Objetos na Prototipação de Semântica de Ações*, Center of Informatics, Federal University of Pernambuco, 1998. Available at www.cin.ufpe.br/~rat.

[12]. Peter D. Mosses and David A. Watt, *Pascal Action Semantics*, version 0.6. Available by FTP as ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWatt93DRAFT/pas-0.6.ps.Z, March 1993.

[13]. Hermano Moura, *Action Semantics of Specimen*, Center of Informatics, Federal University of Pernambuco, October 10, 1996.

[14]. David A. Watt, *JOOS Action Semantics*, Department of Computing Science, University of Glasgow. Available at www.dcs.gla.ac.uk/_daw/-publications/JOOS.ps, 1997.

[15]. David A. Watt and Deryck F. Brown, *Java Action Semantics*, Department of Computing Science, University of Glasgow, 1998.

[16]. Victor T. Sarinho, *Uma Semântica de Ações para Delphi*, Center of Informatics, Federal University of Pernambuco, 2002.

[17]. Bruno Monteiro, *Uma Semântica de Ações para Haskell*, Center of Informatics, Federal University of Pernambuco, 2000.

[18]. David A. Watt, *Standard ML Action Semantics*, Department of Computing Science, University of Glasgow, 1997.

[19]. *Recife Action Tools*, available at http://www.cin.ufpe.br/~rat, last visit, 2003-01-22.

[20]. Augusto Sampaio and Paulo Borba, *Notas de Aulas – Paradigmas de Linguages de Programação*, Center of Informatics, Federal University of Pernambuco. Available at http://www.cin.ufpe.br/~in1007/transparencias/apresentacao.html, last visit, 2003-01-22.

[21]. Cay Horstmann and Gary Cornell, *Core Java 2 Volume 1 Fundamentos*, Ed. Prentice Hall, 2002.

[22]. Marco Cantú, *Dominando o Delphi 4*, Ed. Makron, 1998 .

[23]. Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach, *An Object Model for Multiparadigm Programming*, Department of Computing - Imperial College of Science, Technology and Medicine, 1994.

[24]. Hanne Riis Nielson and Flemming Nielson, *Semantics With Applications A Formal Introduction*, Ed. Jonh Wiley & Sons, available at www.daimi.au.dk/~hrn, 1992.

[25]. Hassan Ait-Kaci , *An Overview of LIFE*, Digital Equipment Corporation, Paris Research Laboratory, 1990.

[26]. Timothy A. Budd, *Multiparadigm Programming in Leda*, Addison Wesley, Massachusetts, 1995.

[27]. Jan Rune Holmevik, *The History of Simula*, Center for Technology and Society, University of Trondheim, 1995.

[28]. *An Introduction and Tutorial for Common Lisp*, available at http://www.apl.jhu.edu/~hall/lisp.html, last visit, 2003-03-11.

[29]. *Haskell, A Purely Functional Language*, available at http://haskell.org/, last visit, 2003-03-11.

[30]. Inprise Corporation, *Object Pascal Language Guide*, 1998. Available at www.ee.ic.ac.uk/bdoc/d4/OPLG.PDF.

[31]. Bjarne Stroustrup, *The C++ Programming Language Special Edition*, Ed. Addison-Wesley, 2001.

[32]. Marco Cantú, *Mastering Delphi 3*, Ed. Sybex, 1997.

[33]. Sun Microsystems, *The Java Language Specification Second Edition*, 2000. Available at http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[34]. Grady Booch, Jame RumBaugh e Ivar Jacobson, *UML - Guia do Usuário*, Ed. Campus, 2000.

[35]. Pressman S. Roger, *Software Engineering A Practitioner's Approach Fourth Edition*, Ed. McGrawHill, 1997.