Capítulo 1

Conceitos e Paradigmas de Programação via Projeto de Interpretadores

Augusto Sampaio e Antônio Maranhão

Material em preparação para o JAI-2008 (Jornadas de atualização em Informática – Sociedade Brasileira de Computação) – ACESSO EXCLUSIVO PARA A DISCIPLINA

Abstract

In this course we present a uniform, interpreter-based, approach to design programming language concepts and paradigms. Particularly, we show how object-oriented techniques and design patterns can be used to develop an incremental and modular design of language concepts from different paradigms. The proposed style has proved well suited for teaching programming concepts. We start with a simple expression language, and progressively cover functional, imperative and object-oriented programming; logic programming, concurrency and more recent paradigms, like agents and aspects, are only briefly discussed. In addition to designing languages of individual paradigms, the presented framework can also be useful to design multiparadigm languages, as well as programming tools like theorem provers.

Resumo

Neste curso, apresentamos uma abordagem uniforme, baseada em interpretadores, para projetar conceitos e paradigmas de programação. Particularmente, mostramos como conceitos de orientação a objetos e padrões de projeto podem ser usados para desenvolver um projeto modular e incremental de conceitos de linguagens de diferentes paradigmas. O estilo proposto tem se mostrado adequado ao ensino de linguagens. Iniciamos com uma linguagem simples de expressões e, progressivamente, cobrimos programação funcional, imperativa e orientada a objetos; programação lógica, concorrente e paradigmas mais recentes como agentes e aspectos são brevemente considerados. Em adição ao projeto de linguagens de paradigmas individuais, o framework apresentado pode ser útil no projeto de linguagens multiparadigmas, bem como de ferramentas como provadores de teoremas.

1.1. Introdução

Uma linguagem de programação é caracterizada em termos de uma gramática e um significado bem definidos. Além disso, a linguagem deve ser implementável (executável) com eficiência aceitável e deve ser *universal*, no sentido que deve ser possível expressar todo problema computável nesta linguagem; recursão, comandos de iteração (como *while loops*) ou mesmo comandos de desvio (como *go to*) são suficientes para garantir universalidade a uma linguagem de programação. Esta característica distingue uma linguagem de programação de, por exemplo, uma linguagem de consulta a banco de dados (*query languages*). Alguns autores [Watt 1990 & 2004] consideram ainda que uma linguagem deve permitir uma expressão natural de problemas em um certo domínio de aplicação, apesar de este ser um critério subjetivo.

A gramática de uma linguagem é usualmente apresentada em termos de regras de formação que definem uma *sintaxe* precisa, estabelecendo que programas são bem formados na linguagem, com relação a aspectos estáticos. A notação padrão para se definir a sintaxe de uma linguagem é a BNF (*Backus-Naur form*) [Backus 1978], conforme será extensivamente ilustrado neste texto. Uma análise mais ampla da validade de um programa inclui, adicionalmente, aspectos dinâmicos, como a definição de um sistema que permita verificação do uso coerente dos tipos de dados da linguagem.

Com relação ao comportamento, existem três estilos principais para definir a semântica de linguagens de programação. No estilo operacional [Plotkin 1981], um modelo matemático da máquina destino é definido e o significado de um programa é dado em termos de uma execução passo a passo do programa no modelo. Isto permite, por exemplo, analisar se os construtores propostos para a linguagem são passíveis de implementação.

Um outro estilo clássico é o denotacional [Schmidt 1986], que mapeia cada construção da linguagem para um valor (denotação) em algum domínio matemático conveniente e independente de uma máquina destino particular. O uso de um modelo matemático explícito permite assegurar consistência do mapeamento semântico. Como exemplos de modelos matemáticos usualmente utilizados podemos citar conjuntos, relações, funções e predicados lógicos.

Finalmente, o estilo axiomático [Hoare 1969] é baseado em postular propriedades gerais dos operadores da linguagem. Dentro do estilo axiomático, o algébrico [Hoare 1987] é o mais usual, onde as propriedades são capturadas por equações que relacionam os construtores da linguagem. Contrariamente aos dois estilos anteriores, um modelo matemático explícito não é construído em uma apresentação algébrica.

Desde o surgimento da primeira linguagem de programação de alto nível, Fortran, na década de 1950, uma grande variedade de linguagens de programação tem sido proposta, como conseqüência de domínios de aplicação distintos, avanços tecnológicos e interesses comerciais, dentre outros aspectos. Algumas linguagens compartilham características em comum e são ditas pertencerem a um mesmo paradigma: um modelo, padrão ou estilo de programação. 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.

Os quatro paradigmas mais estabelecidos são o imperativo [Watt 1990], o lógico [Kowalski 1985], o funcional [Bird 1998] e o orientado a objetos [Meyer 1988]. Cada um possui conceitos e características que influenciam um estilo próprio de programação. Nos paradigmas imperativo e funcional, o modelo computacional é o de uma função que, portanto, mapeia entradas em saídas de forma determinística. Entretanto, enquanto em um programa imperativo este mapeamento é direto, através de funções definidas como na matemática, em um programa imperativo tal mapeamento é indireto, via estados explicitamente modelados por variáveis globais e locais do programa em execução. A Figuras 1 ilustra o modelo computacional dos dois paradigmas.

(a) Paradigma imperativo **(b) Paradigma funcional**

Figura 1. Modelos computacionais dos paradigmas imperativo e funcional

O modelo computacional do paradigma orientado a objetos também é uma função; como no paradigma imperativo, a computação envolve estados explícitos. Cada objeto possui seu estado próprio, formado pela valoração dos respectivos atributos durante a execução do programa. Portanto, estritamente, este paradigma pode ser considerado imperativo [Watt 2004]. Por outro lado, orientação a objetos oferece uma série de mecanismos de estruturação (classes, herança, polimorfismo, mecanismos de visibilidade, ligação dinâmica - *dynamic binding* - entre outras características) que induz a uma forma particular de modelar as aplicações computacionais. Em particular, o estado de uma aplicação, ao invés de monolítico como em um programa imperativo típico, é particionado entre os objetos ativos durante a execução. Os mecanismos de visibilidade (essencialmente os atributos privados) permitem um controle de acesso e atualização de cada partição. A Figura 2 ilustra o modelo computacional de um programa orientado a objetos. Note que ações (comandos) do programa podem envolver invocação de métodos que alteram o estado de um objeto específico, ao invés de um estado único da aplicação. O estado de cada objeto pode, obviamente, envolver outros objetos, criando uma estrutura hierárquica com complexidade arbitrária.

No paradigma lógico, há uma mudança mais substancial do modelo computacional. Ao invés de uma função, a execução é baseada em relação entre entrada e saída. Em princípio, isto permite uma execução nos dois sentidos. Por exemplo, um compilador modelado como um programa lógico que relaciona código fonte e destino pode ser utilizado tanto como um compilador quanto como um decompilador, gerando um programa fonte dado um programa destino [Bowen 1993].

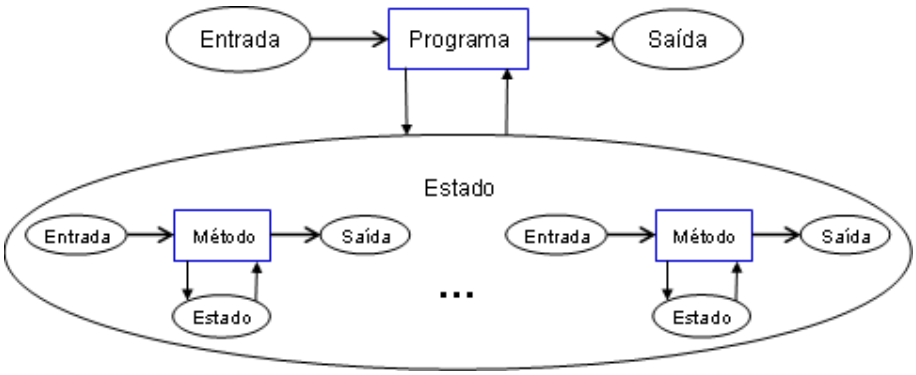

Figura 2. Modelo computacional do paradigma orientado a objetos

A Figura 3 ilustra o modelo computacional do paradigma lógico.

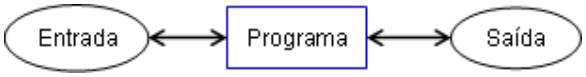

Figura 3. Modelo computacional do paradigma lógico

Uma outra questão interessante, como discutido em [Watt 2004], é se concorrência deve ser considerado um paradigma. Enquanto orientação a objetos (pelo menos do ponto de vista seqüencial) enfatiza a distribuição do estado de um programa, concorrência está relacionada com a distribuição do fluxo de controle da aplicação. Portanto, programas imperativos, orientados a objetos e lógicos podem ser seqüenciais ou concorrentes. Em particular, o recente paradigma de agentes [Shoham 1993] inclui elementos autônomos (agentes) que podem ser considerados objetos ativos (com fluxo de controle independente). Neste paradigma, dados e controle são distribuídos, como ilustrado na Figura 4. O operador || representa a execução paralela dos diversos objetos ativos.

De forma mais ampla, a classificação de programas como seqüenciais ou concorrentes pode ser considerada ortogonal à classificação em paradigmas. Aqui nós adotamos esta visão e o foco é em programas seqüenciais; concorrência (agentes em particular) é brevemente considerada na Seção 1.7.

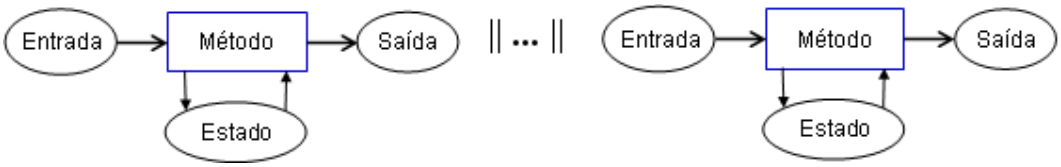

Figura 4. Modelo computacional do paradigma de agentes

Um outro paradigma recente é o de aspectos. A grosso modo, uma aplicação é estruturada em módulos (aspectos) que agrupam pontos de interceptação de código (*pointcuts*) que afetam outros módulos (como classes) ou outros aspectos, definindo novo comportamento (*advice*). Aspectos podem ser estendidos e/ou usados como tipos.

Ensinar conceitos e paradigmas de programação é um grande desafio. Além da eventual complexidade de cada conceito ou paradigma isoladamente, considerar mais de um paradigma usualmente envolve adotar mais de uma linguagem e modelos computacionais variados, como ilustrado acima, o que implica em notações e ambientes de execução distintos, constituindo um empecilho adicional ao aprendizado.

Após algumas tentativas de ensinar paradigmas de programação, percebemos que apenas alunos já com conhecimento de paradigmas isolados conseguiam um bom rendimento no curso. Mesmo assim, os alunos demonstravam certa dificuldade de análise e síntese ao relacionar os conceitos e paradigmas de programação. Isto serviu de motivação para o modelo deste curso, baseado no uso de técnicas de orientação a objetos e padrões de projeto [Gamma, Richard, Johnson and Vlissides 1994]. O objetivo é a construção de um *framework* que possibilita o projeto e implementação, incremental e modular, de uma família de linguagens. As linguagens são criadas através da composição e extensão de componentes que representam os conceitos de programação. O foco é na construção de interpretadores para as linguagens. *Parsers* e verificadores de tipo são também projetados e implementados, mas são considerados aqui apenas superficialmente.

A estrutura geral das linguagens é apresentada na Figura 5. Iniciamos com o projeto de uma linguagem simples de expressões, que oferece os conceitos de valor, expressão, associação (*binding*) entre identificadores e valores, e escopo (Linguagens de Expressão 1 e 2). Mesmo em um contexto tão simples, os alunos já podem começar a exercitar alternativas de decisões de projeto, como escopo (estático ou dinâmico) e declarações (seqüencial e colateral [Watt 2004]). A linguagem de expressões fornece a base necessária para o paradigma funcional. Consideramos primeiro a introdução de funções de primeira odem (Funcional 1) e, em seguida, funções de alta ordem (Funcional 2). Após explorar o paradigma funcional, estendemos a linguagem de expressões na direção de uma linguagem imperativa, primeiro considerando comandos (Imperativa 1) e, em seguida, procedimentos (Imperativa 2). Esta linguagem é então estendida com conceitos de orientação a objetos como classes, criação dinâmica de objetos e invocação de métodos (Orientada a Objetos 1); herança é implementada na Linguagem Orientada a Objetos 2 e, juntamente com. programação lógica, agentes e aspectos são brevemente considerados nas conclusões.

Um outro objetivo é explorar a construção de linguagens multiparadigmas, através da integração de linguagens que implementam paradigmas isolados. Enquanto alguns conceitos são apresentados para os estudantes, vários outros são propostos como exercícios, dando oportunidade aos estudantes de estenderem o *framework* em várias direções. A abordagem tem sido continuamente avaliada e melhorada, através de alguns anos de ensino de linguagens para alunos de pós-graduação. A implementação completa das linguagens pode ser encontrada em [PLP 2008].

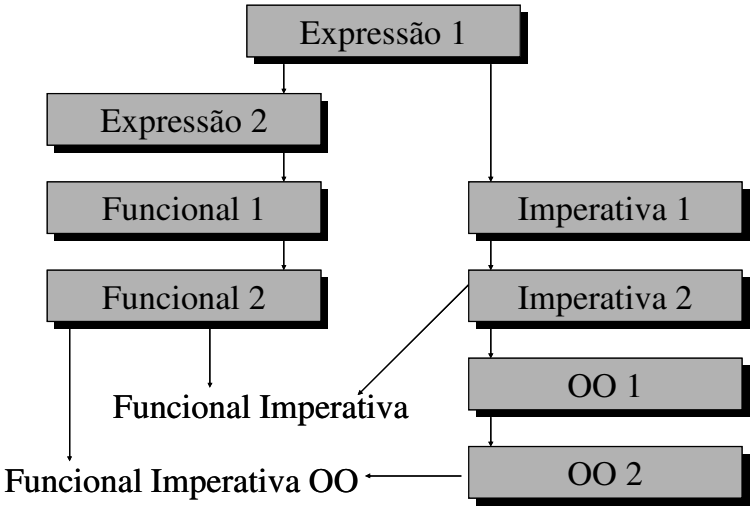

Figura 5. Família de linguagens projetadas

As seções seguintes estão organizadas da seguinte forma. A Seção 1.2 apresenta uma visão geral da abordagem, com foco na apresentação de dois padrões de projeto (Interpreter e Visitor [Gamma, Richard, Johnson & Vlissides 1994]) e uma análise das vantagens de utilizar cada um dos padrões para estruturar o projeto das linguagens. As seções seguintes (de 1.3 a 1.6) apresentam as linguagens da Figura 5, iniciando com expressões e cobrindo os demais paradigmas. Finalmente, a Seção 1.7 apresenta algumas considerações finais, incluindo uma discussão sobre o projeto de outros paradigmas (lógico, aspectos e agentes), bem como o projeto de ferramentas adicionais como provadores de teoremas.

1.2. Abordagem baseada em interpretadores

Há uma série de linguagens que integram conceitos de diferentes paradigmas. Como vimos, orientação a objetos pode ser considerado uma extensão do paradigma imperativo, que por sua vez tem sido combinado com outros paradigmas, como o funcional [Paulson 1996]. Outros exemplos de integração de paradigmas são: funcional e orientado a objetos [Xi 2002, Qian & Moulah 2000], lógico e orientado a objetos [Liu 2001], lógico e funcional [Hanus 1994], e orientado a objetos, funcional e lógico [Ait-Kaci 1991, Budd 1995, van Roy & Haridi 1999].

O problema associado ao projeto destas linguagens reside precisamente na estratégia de integração de conceitos de paradigmas distintos. Em [Ng & Luk 1995], é reportado que a maioria das linguagens multiparadigmas adota uma das seguintes técnicas de integração:

- Construção de uma interface entre dois ou mais linguagens.
- Extração de características úteis dos diferentes paradigmas e inclusão destes no paradigma dominante.

- Redefinição de uma linguagem existente no contexto de novos objetivos e conceitos teóricos.
- Projeto inteiramente novo de uma família de linguagens, com uma base consistente.

Nossa estratégia está baseada nesta última técnica, através da criação de um *framework* que possibilita reutilizar componentes para projetar novas linguagens a partir das já existentes, eventualmente gerando linguagens multiparadigmas. Contudo, ao invés de adotarmos uma semântica formal como base, seguimos uma abordagem baseada na construção de interpretadores, que pode ser considerada um estilo de semântica operacional. Para atingir o objetivo de um modularidade e extensibilidade, centramos nosso modelo em técnicas de orientação a objetos e padrões de projeto. O papel de cada conceito e técnica no projeto fica evidenciado a medida que prosseguimos com o projeto incremental da família de linguagens.

Java [Flanagan 1999] tem se mostrado uma linguagem adequada para desenvolver o projeto, pois é largamente utilizada e inclui os conceitos de orientação a objetos necessários: interfaces, classes (abstratas), exceções e herança, dentre outros. Em particular, Java 1.5 [McLaughlin & Flanagan 2004] oferece conceitos adicionais bastante convenientes, como parametrização de classes (*generics*), conversão entre alguns tipos primitivos como inteiros e objetos que encapsulam os mesmos (*autoboxing*), e tipos enumerados. Assim, Java é utilizada como uma meta-linguagem para implementação da nossa família de linguagens. Entretanto, outras linguagens como C# [Schildt 2006] também são adequadas à implementação de nosso *framework*.

Dois padrões de projeto clássicos [Gamma, Richard, Johnson & Vlissides 1994] são particularmente interessantes para estruturar o projeto de uma família de linguagens: os padrões *Interpreter* e o *Visitor*. O primeiro, apresentado na Figura 6, define uma estrutura na qual o elemento de mais alto nível é uma interface que inclui as assinaturas dos métodos que devem ser implementados por classes concretas que representam elementos terminais da linguagem. Elementos não terminais implementam o significado de algum operador da linguagem e contém um agregado de instâncias que representam os respectivos argumentos, permitindo representar uma estrutura em árvore da sintaxe da linguagem. Assim, as regras para interpretar a linguagem estão dispersas através das construções da linguagem. Cada elemento sabe como se interpretar e recebe uma instância da classe *Context*, representando o ambiente de execução, que serve como um meio comum de propagação dos efeitos gerados pela interpretação de cada elemento.

A principal vantagem deste mecanismo de estruturação está na simplicidade de modificar ou estender as construções das linguagens. Herança pode ser usada para criar novas expressões como variações das já existentes. Além disso, o interpretador para a nova construção tende a ser um incremento ortogonal, desde que este não provoque mudança no contexto nem nas assinaturas dos métodos da interface. Uma das desvantagens do padrão é que adicionar novos métodos ou modificar a assinatura de métodos existentes na interface requer a atualização de todos os elementos da linguagem.

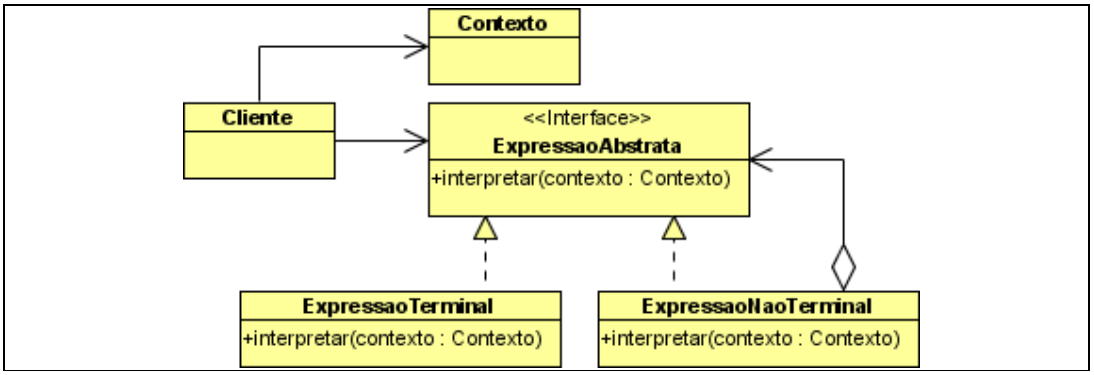

Figura 6. O padrão *Interpreter*

O padrão *Visitor*, modelado na Figura 7, separa a estrutura (sintaxe) do comportamento (semântica). Neste padrão, o código relacionado a uma funcionalidade é agrupado em uma mesma classe (ConcreteVisitor). Com esta estrutura, a tarefa de adicionar ou modificar uma funcionalidade é localizada, desde que é necessário apenas criar um novo ConcreteVisitor que concetra a implementação da nova funcionalidade para todos os elementos da linguagem. Entretanto, a adição de novos elementos sintáticos gera um maior impacto do que no padrão *Interpreter*, pois um novo elemento resulta na inserção de um novo método na interface Visitor e, portanto, este novo método deve ser implementado em todo ConcreteVisitor.

Em nosso caso, nenhum dos dois padrões oferece uma solução ótima. A decisão de qual padrão adotar deve considerar o custo benefício de cada um dos mecanismos de estruturação para a implementação das linguagens em questão. Escolhemos o padrão *Interpreter* porque, na estratégia incremental adotada, é mais freqüente a inclusão de novos elementos do que modificação de contexto ou da assinatura dos métodos da interface. Além disso, consideramos o padrão *Interpreter* mais intuitivo e simples de contextualizar do que o *Visitor*. A estrutura da implementação das linguagens em Java é muito semelhante à estrutura da gramática de cada linguagem, como apresentada pela respectiva BNF. Os elementos da BNF são implementados por classes, interfaces e classes abstratas de Java. Usualmente, um elemento não-terminal é implementado por uma interface ou uma classe abstrata, enquanto um elemento terminal é implementado por uma classe concreta que implementa uma interface ou herda de uma classe abstrata que modela um elemento não-terminal. Além disso, herança tem seu papel usual quando é necessário adicionar ou especializar um comportamento de um elemento existente. A utilização de outros padrões de projeto é brevemente discutida na seção de conclusões.

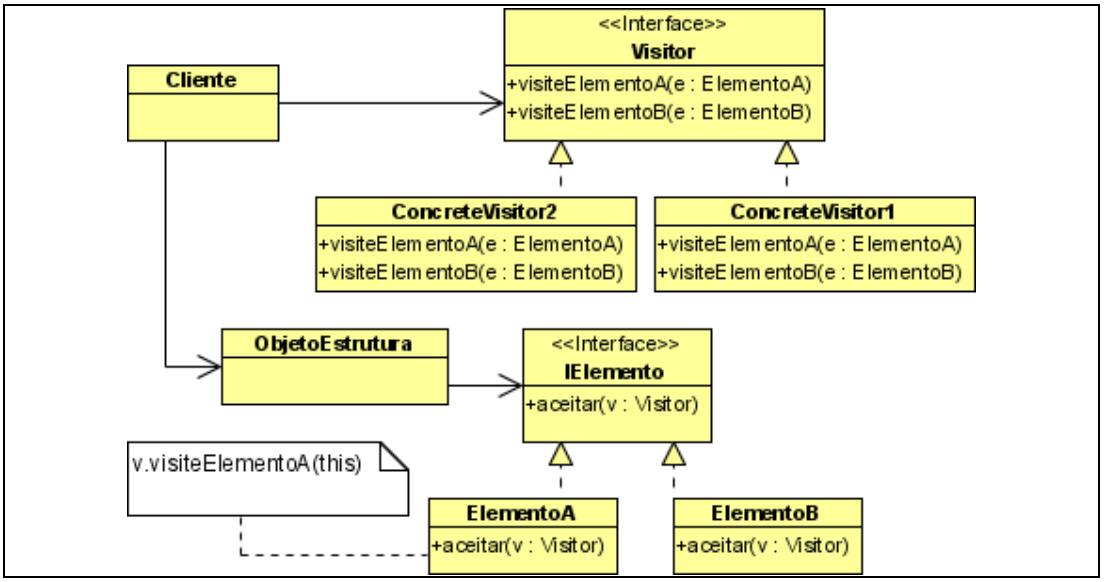

Figura 7. O padrão *Visitor*

Com base nestas decisões de projeto, nas seções seguintes apresentamos um projeto incremental de conceitos de programação, iniciando com uma linguagem de expressões e progressivamente adicionando características dos paradigmas funcional, imperativo e orientado a objetos, como sumarizado na Figura 5. O projeto das linguagens inclui os seguintes aspectos:

- Um *parser* realiza análise léxica e sintática e efetua a redução (conversão da sintaxe concreta em uma sintaxe abstrata modelada por classes em Java).
- Um método, checaTipo, é utilizado em cada unidade (interface, classe ou classe abstrata) que implementa algum construtor da linguagem, com o objetivo de efetuar análise semântica (verificação de tipos)
- Similarmente, um outro método (denominado avaliar no caso de linguagens de expressões e funcionais, e executar para imperativas e orientadas a objetos) é utilizado para avaliação e execução de programas escritos nas linguagens.

O foco da apresentação é na estrutura geral das implementações e no código do método de avaliação e execução de programas. Cada seção a seguir inicia com uma explicação informal seguida da BNF da linguagem. Alguns exemplos de programas e partes da implementação são apresentados. Finalmente, conceitos de programação relacionados às construções são explorados.

1.3. Linguagem de Expressão

Como exemplo do projeto de linguagens nesta abordagem, iniciamos com uma linguagem simples de expressões. Em seguida, consideramos os conceitos de declaração e escopo, mas ainda no contexto de uma linguagem de expressão.

1.3.1 Expressões constantes

Uma expressão é uma construção que é avaliada para produzir um resultado. A primeira linguagem, a Linguagem de Expressão 1, apresentada é composta essencialmente por expressões. Portanto, nesta linguagem, um programa é uma expressão e, assim, sua execução (avaliação) produz um valor como resultado. Esta linguagem também envolve conceitos de tipo, valores constantes e operadores sobre expressões. A Figura 8 apresenta a BNF da linguagem.

```
Programa ::= Expressão
Expressao ::= Valor | ExpUnaria | ExpBinaria
Valor ::= ValorConcreto
ValorConcreto ::= ValorInteiro | ValorBooleano | ValorString
ExpUnaria ::= "-" Expressao | "not" Expressao | "length" Expressao
ExpBinaria ::= Expressao "+" Expressao |  Expressao "-" Expressao |
                Expressao "and" Expressao | Expressao "or" Expressao |
                Expressao "==" Expressao | Expression "++" Expression
```

Figure 8. BNF para a Linguagem de Expressão 1

De acordo com a BNF da linguagem, um programa é uma expressão que pode ser um valor, uma expressão unária ou binária. Nesta linguagem, consideramos apenas valores concretos como inteiros, booleanos e *string* (que admitem comparação sintática por igualdade). Em linguagens mais elaboradas como a funcional de alta ordem (considerada na próxima seção), funções são valores que devem ser diferenciados de valores concretos, pois a igualdade de duas funções não pode ser determinada por comparação sintática; categorizamos valores do tipo função e instância de classe (objeto) como abstratos. Como exemplos de expressão unária, consideramos a troca de sinais de inteiros, a negação de um booleano e a operação que retorna o tamanho de um *string*. As expressões binárias envolvem soma e subtração sobre inteiros, conjunção e disjunção lógica, igualdade sobre todos os tipos de expressão e concatenação de *strings*.

Um *Applet* foi implementado com o objetivo de permitir ao usuário selecionar uma das linguagens disponíveis, entrar com um programa fonte (no caso, uma expressão que obedeça à gramática da Figura 8), e executar o programa, conforme ilustrado na Figura 9. Exemplos de programas válidos nesta linguagem envolvem apenas constantes, como em length("abc") + 3, true and false, "curso" ++ " de " ++ " paradigmas", que, quando executados, retornam 6, false e "curso de paradigmas", como esperado. Expressões como 1 + true resultam em erro de tipo.

A estrutura do projeto desta linguagem, através de um diagrama de classes UML, pode ser vista na Figura 10. No projeto orientado a objetos das linguagens, cada item da BNF é uma classe, interface ou classe abstrata de Java. A hierarquia da BNF é modelada através de herança e subtipos (extends e implements). Classes adicionais são introduzidas para modelar estruturas internas do interpretador, como o contexto. Como

mencionado anteriormente, elementos não terminais são usualmente mapeados em interfaces ou em classes abstratas. Por exemplo, o item Expressao é modelado como uma interface. Já ExpressaoUnaria e ExpressaoBinaria são modeladas como classes abstratas. A razão é que não há qualquer elemento comum entre os diversos tipos de expressão que possam ser compartilhados, a não ser a assinatura dos métodos; já no caso de expressões unárias, por exemplo, todas possuem um operador e um argumento (que também é uma expressão), de forma que há a possibilidade de um compartilhamento estrutural (atributos); o mesmo ocorre no caso de expressões binárias, que possuem 2 argumentos (expressões) e um operador.

Figura 9. Applet para execução dos programas

O elemento não terminal Valor também é claramente capturado por uma interface, já que valores concretos e abstratos não compartilham estrutura. Em alguns casos, parametrização de classes fornece poder de expressão adicional. Em Java 1.4, ValorConcreto também é representado como uma interface, pois não há como expressar compartilhamento entre os diversos tipo de valor concreto, já que cada um é representado por um atributo de tipo distinto (inteiro, booleano e *string*). Entretanto, em Java 1.5, é possível representar ValorConcreto como uma classe abstrata parametrizada (usando o conceito de *generics*) com o tipo do atributo encapsulado; então, elementos terminais como ValorInteiro, ValorBooleano e ValorString são instâncias desta classe

parametrizada abstrata. Os elementos da BNF que implementam cada um dos operadores unários e binários também são implementados por classes concretas.

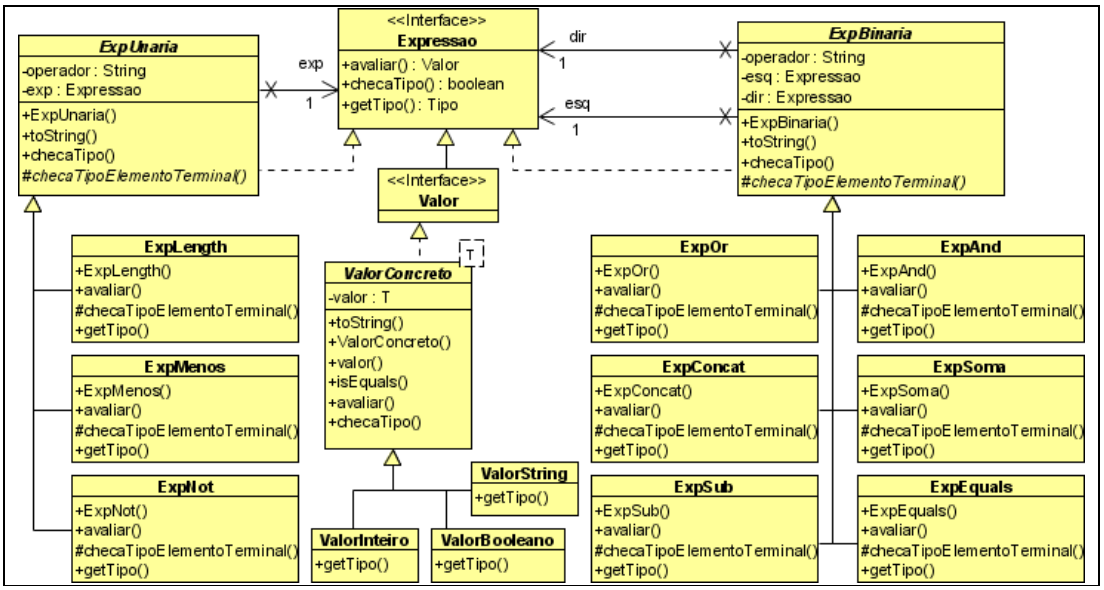

Figura 10 Diagrama de classes para a Linguagem de Expressão 1

Com relação ao código, a interface Expressao inclui as assinaturas dos métodos avaliar, checaTipo e getTipo. O primeiro retorna o valor da expressão avaliada. O método checaTipo retorna um valor booleano indicando se a expressão é ou não bem tipada. O último método retorna o tipo da expressão. A classe que implementa um Programa contém um atributo (a expressão a ser avaliada) e invoca estes métodos para verificar se a expressão é bem tipada e, caso seja, avaliá-la e retornar o resultado produzido, conforme já ilustrado na Figura 9. A interface Valor estende Expressao mas não adiciona qualquer assinatura de método. Já ValorConcreto é implementado por uma classe parametrizada, cujo código é apresentado na Figura 11. Note que a classe é parametrizada pelo tipo T, que pode então ser instanciado com classes representando inteiros, boleanos e *string*, para criar valores dos respectivos tipos, como ilustrado pelo classe ValorInteiro, na mesma Figura. O atributo valor representa o valor encapsulado, que pode ser consultado usando o método valor(). O método isEquals compara dois valores e implementa o operador ==. A avaliação de um valor retorna o próprio e um valor está sempre bem tipado. Construtores e alguns métodos responsáveis pela impressão de um valor são omitidos no código apresentado. A classe ValorInteiro herda e instancia a classe ValorConcreto, adicionando a implementação do método que retorna o tipo especifico da constante. O código de ValorBooleano e ValorString é análogo.

```
public abstract class ValorConcreto<T> implements Valor {
   private T valor;
   public T valor() {return valor;}
   public boolean isEquals(ValorConcreto<T> obj) {return valor().equals(obj.valor()); }
   public Valor avaliar() {return this;}
   public boolean checaTipo() {return true;}
}

public class ValorInteiro extends ValorConcreto<Integer> {
   public Tipo getTipo() {return Tipo.TIPO_INTEIRO;}
}
```

Figura 11. Código das classes ValorConcreto **e** ValorInteiro

Como um exemplo adicional, a Figura 12 apresenta o código da classe ExpSoma, que implementa a operação binária de soma de duas expressões inteiras.

```
public class ExpSoma extends ExpBinaria {
   public Valor avaliar() {
       return new ValorInteiro(
         ((ValorInteiro) getEsq().avaliar()).valor() + ((ValorInteiro) getDir().avaliar()).valor() );
   }
   public boolean checaTipo() {
       return (getEsq().getTipo().eInteiro() && getDir().getTipo().eInteiro());
   }
   public Tipo getTipo() {return Tipo.TIPO_INTEIRO;}
}
```

Figura 12. Código da classe ExpSoma

A avaliação de uma expressão binária envolve a avaliação de cada uma das subexpressões. Os valores parciais são somados e o resultado final é encapsulado como uma objeto do tipo ValorInteiro. Tal expressão é bem tipada se e somente se ambos os argumentos forem de fato do tipo inteiro. O tipo da expressão como um todo é, obviamente, do tipo inteiro.

Os conceitos de programação relevantes que podemos extrair desta linguagem simples de expressão são: valor e tipo. Um valor é algo que pode ser avaliado, armazenado, incorporado em estruturas de dados, passado como parâmetro, retornado como resultado. Ou seja, um valor é uma entidade que existe durante uma computação. Decidir que construções da linguagem têm o status de valor pode impactar radicalmente

o poder de expressão da linguagem. Por exemplo, considerar que funções são valores resulta em uma linguagem funcional de alta ordem, como explorado na Seção 1.4.

É conveniente agrupar valores em tipos, que são são conjuntos de valores e operações associadas. Valores de um tipo devem exibir comportamento uniforme em relação às operações sobre o tipo. O conceito de tipos permite a definição de restrições contextuais que eliminam vários erros já durante uma análise estática (compilação). A linguagem apresentada nesta seção agrupa valores em tipos e a verificação de tipos foi implementada via inferência, desde que não há declaração explícita de tipos.

Alguns exercícios são sugeridos a seguir.

1. Estender a LE1 com ValorChar (caracter) e uma operação que retorna a ordem (ascii) do caracter.
2. Implementar ValorReal e as operações aritméticas usuais sobre os reais.
3. Implementar operadores relacionais como *menor que*, *maior que*, ..., para valores inteiros.

1.3.2. Expressões com declarações

A Linguagem de Expressão 2 estende a anterior com identificadores (variáveis) que possuem um valor (constante). Durante a interpretação e verificação de tipos, surge a necessidade de *contextos*: um mapeamento entre identificadores e valores, e um outro mapeamento entre identificadores e tipos. Na avaliação de uma expressão, a ocorrência de um identificador é substituída pelo valor associado ao identificador. Durante a verificação de tipos, o tipo de um identificador que ocorre em uma expressão precisa ser conhecido para que a verificação da expressão seja analisada. Um programa continua sendo uma expressão.

A BNF apresentada na Figura 13 estende a linguagem anterior (Figura 8) relação com a inclusão das expressões ExpDeclaracao e Id, que parecem destacados no figura; O restante da gramática é exatamente como na linguagem anterior. Uma expressão do tipo ExpDeclaracao permite a introdução de um bloco let ... in ... que declara uma variável local que pode ser utilizada na expressão após o in, como no exemplo let var x = 1 in x + 1, cujo resultado é 2, já que o valor de x nesta expressão é 1. A expressão x + 1 é válida nesta linguagem porque identificadores (Id) são expressões, conforme definido pela BNF.

```
Programa ::= Expressão
Expressao ::= Valor | ExpUnaria | ExpBinaria | ExpDeclaracao | Id
ExpDeclaracao ::= "let" DecVariavel "in" Expressao
DecVariavel ::= "var" Id "=" Expressao | DecVariavel "," DecVariavel
Valor ::= ...
ExpUnaria ::= ...
ExpBinaria ::= ...
```

Figura 13. BNF para a Linguagem de Expressão 2

Blocos de declaração podem ser arbitrariamente aninhados. Inclusive, uma mesmo identificador pode ser declarado mais de uma vez em blocos aninhados. Na avaliação de uma expressão, o valor de um identificador é o do bloco let mais interno onde o identificador foi declarado. Por exemplo, o valor da expressão

```
let var x = 1 in
   let var x = 2 in x + 1
```

é 3. Um bloco let pode declarar várias variáveis simultaneamente, como em:

```
let var str = "abc", var x = 3 in x + length(str)
```

Uma decisão de projeto de linguagens é se uma variável pode ser referenciada na expressão associada à outra na parte de declaração de um mesmo bloco. Por exemplo, a expressão let var str = "abc", var x = length(str) in x + x é válida? Uma seqüência de declaração pode ser classificada como *colateral* ou *seqüencial* [Watt 2004]. Uma declaração colateral segue a idéia de um conjunto de declarações, onde a ordem é irrelevante; portanto, a expressão acima seria inválida, já que a declaração de x não poderia referenciar str, a menos que houvesse uma outra declaração de str em um bloco mais externo. Já uma declaração seqüencial a ordem é relevante e a ocorrência de str na declaração de x se refere ao identificador declarado no mesmo bloco. Portanto, a expressão é válida e produz o valor 6 quando avaliada. A implementação corrente da linguagem adota o conceito de declaração colateral.

Como mencionado anteriormente, declarações geram a necessidade de contextos tanto com relação a uma análise estática (tempo de compilação) quanto dinâmica (execução) do programa. Portanto, além da implementação dos elementos da BNF da linguagem, são necessários componentes que implementem os contextos. É importante lembrar que o padrão *Interpreter* (Figura 6) inclui um objeto para representar o contexto. A Figura 14 apresenta o diagrama de classes destes componentes auxiliares.

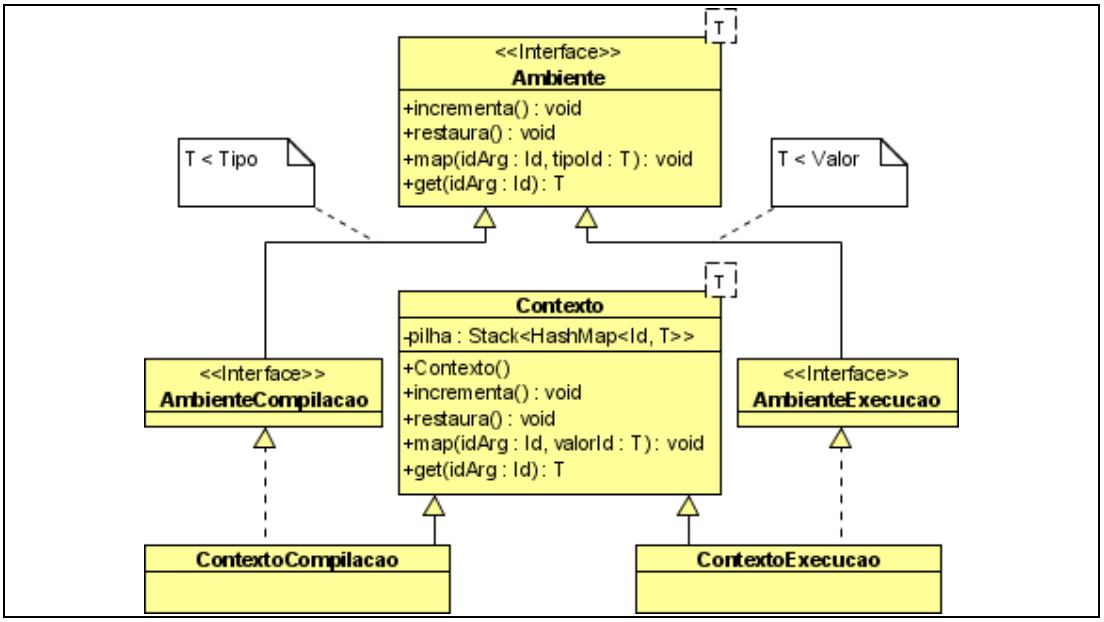

Figura 14. Estrutura do contexto de declarações

Cada bloco de declarações dá origem a um mapeamento de identificadores em tipos e outro de identificadores em valores. Como a linguagem admite estrutura de blocos aninhados, é necessária uma pilha de mapeamentos que capture a estrutura de blocos. Na Figura 14, a interface Ambiente inclui quatro métodos:o método incrementa acrescenta no topo um novo mapeamento; restaura elimina o mapeamento no topo; map inclui um novo par no mapeamento no topo da pilha; e get retorna o elemento associado a um dado identificador. Esta interface é parametrizada pelo tipo dos elementos na imagem do mapeamento, de forma a permitir o reuso.

A interface AmbienteExecucao estende a interface Ambiente simplesmente instanciando o parâmetro com a classe Valor. A interface AmbienteCompilacao estende Ambiente de forma análoga, só que o parâmetro é instanciado com a classe que representa o tipo de uma expressão. A classe Contexto implementa estes métodos de forma genérica (parametrizada). As classes ContextoExecucao e ContextoCompilacao instanciam Contexto de forma análoga à instanciação da interface Ambiente pelas interfaces específicas para execução e compilação.

Com o entendimento do conceito de contexto, é possível explorar partes das implementações das classes que implementam elementos da BNF da linguagem. A classe Id, que implementa um identificador é bastante simples, como mostra a Figura 15. A avaliação de um identificador simplesmente retorna o respectivo valor armazenado no ambiente de execução; A checagem de tipos usa o ambiente de compilação para verificar se o identificador foi declarado; o método get retorna uma exceção caso não esteja.

```
public class Id implements Expressao {
   private String  idName;
   public Valor avaliar(AmbienteExecucao ambiente)  throws VariavelNaoDeclarada {
       return ambiente.get(this);
   }
   public boolean checaTipo(AmbienteCompilacao amb) throws VariavelNaoDeclarada {
       Tipo t = amb.get(this); // se estiver no ambiente, entao esta ok.
       return true;
   }
}
```

Figura 15. Código da classe Id

A classe que implementa um bloco de declaração é mais elaborada. A Figura 16 apresenta os atributos da classe e o método que avalia um bloco. A classe tem dois atributos: a seqüência de declarações introduzidas pelo bloco e a expressão a ser avaliada considerando estas declarações em escopo. A avaliação inicia incrementando o contexto (criando espaço no topo da pilha para armazenar as declarações do bloco). Em seguida, a seqüência de declarações é avaliada, resultando em um mapeamento dos identificadores declarados nos respectivos valores. Só após todas as declarações serem avaliadas, o mapeamento é inserido (no topo) da pilha que representa o ambiente de execução. A outra alternativa seria inserir cada declaração no ambiente a medida que fosse avaliada, mas isto implementaria declaração seqüencial, ao invés de colateral, pois uma declaração já estaria em escopo na avaliação de uma declaração subseqüente. Finalmente, após as declarações serem inseridas no ambiente, a expressão do bloco é avaliada, o ambiente é restaurado, eliminando as declarações referentes ao bloco processado, e o valor resultante da avaliação é retornado. O código dos métodos invocados na avaliação são omitidos por questão de simplicidade.

```
public class ExpDeclaracao implements Expressao{
  List<DecVariavel> seqdecVariavel;
  Expressao expressao;

  public Valor avaliar(AmbienteExecucao ambiente) {
      ambiente.incrementa();
      Map<Id,Valor> resolvedValues = resolveValueBindings(ambiente);
      includeValueBindings(ambiente, resolvedValues);
      Valor result = expressao.avaliar(ambiente);
      ambiente.restaura();
      return result;
  }
}
```

Figura 16. Avaliação de um bloco de declarações

Como enfatizado, na Linguagem de Expressão 2 surge a necessidade de armazenar os valores dos identificadores (variáveis). Em geral, um *binding* é uma associação entre um identificador e uma entidade (constante, variável, função, procedimento, tipo, etc.). Um ambiente (*environment*) ou contexto é um conjunto de associações. Portanto, um contexto é um mapeamento de identificadores em entidades. Na linguagem apresentada, os únicos tipos de associação possíveis são entre identificador e tipo, e identificador e valor. Uma expressão é interpretada (seja para fins de verificação de tipos ou de avaliação) em um dado contexto; todo identificador que aparece na expressão tem que ter uma associação no contexto. Uma mesma expressão ocorrendo em partes diferentes do programa pode produzir resultados distintos (quando os contextos forem diferentes). Outros tipos de associação vão aparecer à medida que estudarmos as outras linguagens.

Seguindo [Watt 2004], as entidades que podem ser associadas a identificadores são denominadas ligáveis (*bindable*). As entidades ligáveis de uma linguagem imperativa típica como Pascal são: valores primitivos, *strings*, referências a variáveis, funções, procedimentos e tipos. Entretanto, valores do tipo registro ou vetores (*arrays*) não são ligáveis. Já a linguagem funcional ML [Paulson 1996] é mais uniforme neste aspecto, pois todos os tipos de valores são ligáveis.

Escopo é a porção do texto do programa na qual uma dada declaração (ou *binding* resultante) é visível. Em uma linguagem monolítica (sem estrutura de bloco), o escopo de uma declaração é todo o programa, como na linguagem Cobol. Há a estrutura de blocos linear (ou *flat*) onde um programa é particionado em blocos (tipicamente procedimentos ou subrotinas), como em Fortran. Neste caso, variáveis declaradas no programa e os nomes das rotinas possuem escopo global; variáveis declaradas na subrotina possuem escopo local. A estrutura de blocos aninhados (*nested*) é a mais geral e permite que um bloco seja introduzido dentro de qualquer outro bloco. Neste caso, há vários níveis de escopo e surge o conceito

de “buraco” no escopo de um identificador; isto ocorre quando em uma parte do texto do programa o identificador não é visível porque um outro (com mesmo nome) foi introduzido. Como vimos, a Linguagem de Expressão 2 permite estrutura de blocos.

Uma declaração é uma construção do programa a partir da qual associações são definidas. Alguns tipos de declaração são: definições, declarações colaterais, declarações seqüenciais, declarações recursivas. A linguagem apresentada permite definições (declaração de constantes) e declarações colaterais, como explicado. Declarações recursivas são exploradas na próxima seção.

Como exercício, Modifique o interpretador de Linguagem de Expressão 2 de forma que as declarações em um bloco let sejam seqüenciais ao invés de colaterais.

1.4. Programação Funcional

Como mencionado anteriormente, no paradigma funcional um programa estabelece uma relação clara e explícita entre entrada e saída. Neste paradigma, um programa consiste essencialmente de funções. Modularidade é suportada através da possibilidade de decompor uma função complexa em termos de outras mais simples; reuso é igualmente apoiado através da composição de funções existentes na construção de uma tarefa mais elaborada.

Em linguagens funcionais ditas *puras*, não há a variáveis, apenas associações fixas entre identificadores e valores constantes. Assim, não há o conceito de estado como em programas imperativos, nem tampouco efeito colateral (*side effects*). Programas funcionais podem ser escritos e manipulados (transformados) como uma expressão matemática. Em particular, a transformação de programas é facilitada por transparência referencial (*referential transparency*), que permite que uma expressão possa sempre ser substituída por outra de mesmo valor, já que, sem efeitos colaterais, não há a possibilidade de uma expressão ter seu valor alterado. Por esta razão, costuma-se dizer que programação funcional apresenta um estilo declarativo, em oposição ao estilo imperativo explorado na próxima seção.

Apesar de alguns conceitos de programação não serem inerentes ao paradigma funcional, historicamente estão bastante associados a este, pois foram introduzidos e consolidados em linguagens funcionais. Alguns exemplos são polimorfismo paramétrico (ou real), funções de alta ordem e avaliação sob demanda, entre outros. A subseção 1.4.1 explora o conceito de funções recursivas e a seguinte explora funções de alta ordem.

1.4.1. Linguagem Funcional de Primeira Ordem

A sintaxe da Linguagem Funcional 1 é apresentada na Figura 17. A linguagem estende a Linguagem de Expressão 2 com funções parametrizadas e recursivas (de primeira ordem, funções não são valores). O corpo de uma função é uma expressão e a aplicação da função a um argumento retorna um valor (aplicação também é uma expressão). Conforme as linguagens anteriores, um programa nesta linguagem também é uma expressão, que pode envolver invocação (aplicação) de função.

Além da introdução de definição e aplicação de função, a linguagem inclui uma expressão condicional (if ... then ... else). Apesar de ser mais usual como um comando, o comportamento aqui é semelhante: se a avaliação da primeira expressão (que deve necessariamente ser boolena) resultar em true, a segunda expressão é selecionada; caso contrário, a terceira expressão é selecionada. O uso de expressões condicionais é essencial na definição de funções recursivas que tipicamente incluem um caso base e o caso geral.

Declaração de função ocorre em blocos de declaração. Na Linguagem de Expressão 2, um bloco pode conter apenas declaração de variáveis; a construção é estendida aqui para permitir também declaração de funções. Um mesmo bloco pode conter os dois tipos de declaração. A declaração de uma função inclui uma lista de identificadores (os parâmetros da função) e um corpo, constituído da expressão a ser avaliada quando a função é invocada. A aplicação de uma função envolve o identificador da função invocada e uma lista de expressões, que deve obedecer a ordem e os tipos dos parâmetros.

Programa ::= Expressao

Expressao ::= Valor | ExpUnaria | ExpBinaria | **ExpDeclaracao** | Id |
Aplicacao | IfThenElse

ExpDeclaracao ::= "let" DeclaracaoFuncional "in" Expressao

DeclaracaoFuncional ::= DecVariavel | DecFuncao |
DeclaracaoFuncional "," DeclaracaoFuncional

DecFuncao ::= "fun" ListId "=" Expressao

Aplicacao:= Id"(" ListExp ")"

IfThenElse ::= "if" Expressao "then" Expressao "else" Expressao

DecVariavel ::= ...

Valor ::= ...

ExpUnaria ::= ...

ExpBinaria ::= ...

Figura 17. BNF para a Linguagem de Funcional 1

Um exemplo de programa nesta linguagem é o bloco

```
let fun prod x y = if y == 0 then 0 else x + prod(x, y-1)  in prod(3,4)
```

que introduz uma definição recursiva do produto de dois inteiros usando soma. Na definição, o produto de x e y é obtido pela soma do valor de x um número de vezes dado pelo valor de y. Os blocos podem ser aninhados, como na Linguagem de Expressão 2, só que agora podem incluir declarações de variáveis e funções, como no exemplo

```
let fun prod x y = if y == 0 then 0 else x + prod(x, y-1)  in
    let var k = 5, fun fat n = if n == 0 then 1 else prod(n, fat(n-1)) in fat(k)
```

que computa o fatorial de k (com valor 5), usando a função prod. O resultado da avaliação do programa inteiro é 120, como esperado.

O tipo de uma função é inferido automaticamente a partir das declarações e expressões em do programa. O tipo inferido é o mais geral possível para os parâmetros e resultado da função. Um exemplo simples é a função identidade, que é polimórfica já que não há qualquer restrição no argumento e resultado da função.

```
let fun id x = x in id(2)
```

No caso, a função é aplicada a um argumento inteiro, mas argumentos de quaisquer tipos podem ser utilizados.

Como esta linguagem oferece duas formas de declaração (variáveis e funções), os ambientes de execução e de compilação precisam incluir mapeamentos que representem armazenem as respectivas informações. O ambiente de execução inclui dois componentes: um mapeamento de identificadores em valores (como na Linguagem de Expressão 2) e um mapeamento de identificadores (nomes de função) em definições de função. O ambiente de compilação possui um mapeamento de identificadores em tipo, mas o conceito de tipo é estendido para funções, que é representado pelo produto cartesiano dos tipos dos parâmetros e o tipo do resultado produzido pela função. A interface que modela o ambiente de execução é apresentada na Figura 18. Analogamente aos métodos para mapear uma declaração e retornar o valor de um identificador, a interface estendida adiciona métodos para mapear uma declaração de função e retornar a definição de uma função associada a um dado identificador.

```
public interface AmbienteExecucaoFuncional extends AmbienteExecucao {
    public void mapFuncao(Id idArg, ValorFuncao funcao) throws VariavelJaDeclaradaException;
    public ValorFuncao getFuncao(Id idArg) throws VariavelNaoDeclaradaException;
}
```

Figura 18. Interface do ambiente de execução funcional

Para esta linguagem e as seguintes, focamos nos métodos de avaliação; a implementação da verificação de tipos pode ser encontrada em [PLP 2008]. A classe mais elaborada da Linguagem Funcional 1 é a que implementa aplicação de função, conforme código na Figura 19, que destaca os atributos da classe e o método responsável pela avaliação. O primeiro atributo representa o identificador da função invocada. O segundo é uma lista de expressões que representa os argumentos da

invocação. A implementação da avaliação da aplicação é semelhante à avaliação de um bloco de declarações let. (contrastar com a Figura 16).

```
public class Aplicacao implements Expressão {
    private Id func;
    private List<Expressao> argsExpressao;
    public Valor avaliar(AmbienteExecucao ambiente) {
        AmbienteExecucaoFuncional ambienteFuncional =(AmbienteExecucaoFuncional) ambiente;
        ValorFuncao funcao = ambienteFuncional.getFuncao(func);
        Map<Id,Valor> mapIdValor = resolveParametersBindings(ambiente, funcao);
        ambiente.incrementa();
        includeValueBindings(ambiente, mapIdValor);
        Valor vresult = funcao.getExp().avaliar(ambiente);
        ambiente.restaura();
        return vresult;
    }
}
```

Figura 19. Código da classe Aplicação

A primeira linha do corpo do método avaliar efetua um *cast* no ambiente de execução (parâmetro do método) requerendo que o ambiente seja de fato um ambiente de execução funcional. A idéia de manter o parâmetro de avaliar com tipo AmbienteExecucao e realizar o *cast* é que isto permite a reutilização das interfaces da linguagem modelada na seção anterior. A segunda linha do método recupera a definição da função, no ambiente, associada ao identificador da função invocada. A linha seguinte cria um mapeamento que representa a avaliação das expressões passadas como argumento para a função. Em seguida, o ambiente é incrementado com este mapeamento, a expressão que representa o corpo da função é avaliada neste contexto, o ambiente é restaurado e o valor resultante da avaliação é retornado. Uma alternativa de avaliação seria transformar uma invocação de função em um bloco de declaração e reutilizar a avaliação de um bloco. Os métodos auxiliares chamados por avaliar são omitidos por simplicidade.

Das linguagens analisadas até o momento, a Linguagem Funcional 1 é a primeira que pode ser considerada uma linguagem de programação no sentido definido na Seção 1. A definição requer que a linguagem possua uma Sintaxe (que, no caso, é dada pela BNF que define regras para boa formação de valores, expressões e funções), uma semântica (o método de avaliação pode ser considerado uma semântica operacional) e uma implementação, que também é dada pelo código Java que realiza a avaliação de qualquer programa bem formado na linguagem. Finalmente, o critério de universalidade é atendido pela introdução de funções recursivas.

Com relação à semântica, a avaliação de um termo se dá através de sua redução a uma forma normal (ou canônica), através de reescrita de termos baseada na definição das funções e operadores que ocorrem em uma expressão. Como as linguagens

estudadas até o momento são declarativas, o processo de redução é composicional: o da expressão é obtido a partir dos valores dos subcomponentes, graças à propriedade de transparência referencial. Na forma como as linguagens foram implementadas, as expressões mais internas são avaliadas primeiro. A forma canônica de uma expressão é o valor resultante da avaliação. Este valor é o significado (semântica) da expressão.

Entretanto, nem sempre a avaliação de uma expressão é bem definida, pois algumas expressões não podem ser reduzidas; não denotam um valor. Um exemplo simples é a divisão de um número por zero, que é indefinida já que divisão é um operador parcial, não estando definido neste caso. Um outro exemplo é uma função recursiva cuja avaliação resulta em uma iteração infinita, como no exemplo

```
let fun f x = f x in f(2)
```

Portanto, formas normais nem sempre existem. É comum se introduzir um símbolo especial $\perp$ (lido como *bottom*), para representar um valor indefinido: portanto, considerando a definição acima de f, temos que $f(x) = \perp$, para todo x.

Em geral, há diferentes alternativas para se conduzir a avaliação de uma expressão, com relação à ordem de aplicação das funções. Por exemplo, considere a definição da função a seguir que retorna o quadrado de um inteiro:

```
let fun prod x y = if y == 0 then 0 else x + prod(x, y-1)  in
    let fun quad x = prod(x,x)
```

A função é definida em termos do produto de um inteiro por ele mesmo; a função prod já foi explicada anteriormente. Uma possível ordem de avaliação de quad(5 + 3) é capturada pela seqüência de reescrita de termos:

```
quad(5 + 3)  = quad (8)       [pela definição de soma]
             = prod(8,8)      [pela definição de quad]
             = 64             [pela definição de prod]
```

Uma outra ordem de avaliação é dada pela seqüência:

```
quad(5 + 3)  = prod((5 + 3),(5+3))      [pela definição de quad]
             = prod(8,(5+3))            [pela definição de soma]
             = prod(8,8)                [pela definição de soma]
             = 64                       [pela definição de prod]
```

A primeira estratégia de avaliação acima é conhecida como *eager evaluation*, onde os argumentos (em geral, expressões mais internas) são avaliados primeiro. A segunda é denominada de *normal-order evaluation*; a avaliação ocorre das expressões mais externas para as mais internas. Uma decisão de projeto de linguagens é qual estratégia de avaliação adotar. Primeiro é importante analisar se diferentes estratégias de avaliação produzem sempre o mesmo resultado. Considere a expressão

```
let fun f x = f x in let fun first y z = y in first(2,f(3))
```

Em uma estratégia *eager*, a tentativa de avaliar f(3) gera uma recursão infinita, o que resulta em ⊥, fazendo com que o bloco de declaração como um todo tenha um valor indefinido. Já em uma estratégia *normal-order* a avaliação de first(2,f(3)) resulta em 2 sem a necessidade de avaliar f(3). Portanto, nem sempre os resultados produzidos são o mesmo. Neste caso em particular, o processo de avaliação de um programa resulta em um valor que corresponde ao seu significado e no outro retorna um valor indefinido.

Entretanto, um importante resultado teórico, conhecido como a propriedade de Church-Rosser, garante que, se uma expressão (sem efeitos colaterais) possuir uma forma normal, esta será sempre obtida através de uma redução *normal-order*. Se a redução usando a estratégia *eager* (ou qualquer outra combinação de avaliação entre *eager* e *normal-order*) produzir um resultado, este será o mesmo produzido usando uma redução *normal-order*. A estratégia *normal-order* também é conhecida como *normal outermost reduction*. O mecanismo *innermost* também é conhecido como *applicative-order reduction*. Avaliação preguçosa, ou sob demanda, *lazy evaluation*, é uma especialização do mecanismo *normal-order* onde o argumento é avaliado apenas uma vez. Durante a execução de um programa, qualquer ocorrência subseqüente da expressão é substituída pelo valor que denota a expressão, sem a necessidade de uma reavaliação.

No caso de funções estritas, onde os argumentos precisam ser sempre avaliados durante a computação da invocação de uma função, o mecanismo *eager* é mais eficiente. Entretanto, o mecanismo *normal-order* (e avaliação preguiçosa, em particular) pode resultar em economia computacional para funções não estritas. Além disso, este mecanismo permite modelar estruturas potencialmente infinitas, como listas, favorecendo um estilo de programação onde o fluxo de controle é separado da computação propriamente dita [Watt 2004].

Uma outra decisão de projeto de linguagens, com relação a bloco de declarações envolvendo abstrações como funções ou procedimentos, é a interpretação do escopo dos identificadores declarados. Por exemplo, considere o programa:

```
let var z = 2, fun f(x) = x + 1
    in Let fun g(y) = f(y) + z
           in Let var z = 0, fun f(x) = x
              in g(1)
```

O resultado depende da associação de f e z usados na definição de g. Se as associações forem as obtidas a partir do bloco mais externo (z = 2, f(x) = x + 1) o resultado é 4, mas se forem as obtidas a partir do bloco mais interno (z = 0, f(x) = x) a o resultado é 1. No primeiro caso, diz-se que o escopo das variáveis é *léxico* ou *estático* e, no segundo, *dinâmico*. Escopo léxico é mais usual e é adotado em todas as linguagens de programação mais recentes. Escopo dinâmico foi adotado pela linguagem LISP. Os interpretadores atuais da Linguagem Funcional 1 implementa escopo dinâmico; isto é uma conseqüência de implementar o contexto usando a estrutura de pilhas.

Alguns exercícios são sugeridos a seguir:

1. Modifique a implementação da Linguagem Funcional 1 de forma que o escopo das variáveis seja estático.

2. Atualmente, a verificação de tipos é implementada através de um algoritmo de inferência, já que as variáveis são declaradas sem os tipos explícitos; inclua tipos explícitos nas declarações de variáveis e funções e reimplemente a verificação de tipos. Possível sintaxe: let var z : Int = 2, fun f : Int -> Int . f x = x + 1 in …

3. Implementar tipos livres, ou *datatypes*, comuns em programação funcional. O caso mais simples permite a definição de tipos enumerados, como em

   Type Cor ::= amarelo | azul | preto ...

   Uma facilidade mais elaborada é a união (disjunção de tipos), como em:

   Type IntBool ::= inteiro (Int) | booleano (Bool)

   Em geral, os tipos podem ser recursivos, como no exemplo a seguir.

   Type Lista ::= vazia | cons (Int x Lista)

4. Implementar listas como um tipo de dado pré-definido na linguagem. Com relação à sintaxe, usar **[]** para representar a lista vazia e **[e1,e2,...,en]** para uma lista de n elementos. Incluir as seguintes operações sobre listas:

   - **cons(e,L)** recebe um elemento (**e**), uma lista (**L**) e retorna uma lista com o elemento e acrescido no início de **L**
   - **head(L)** recebe uma lista **L** e retorna o elemento que está na cabeça (elemento mais à esquerda) de **L.**
   - **Tail(L)** recebe uma lista **L** e retorna uma lista com todos os elementos de **L** exceto o head.

5. Implementar casamento de padrão em LF1. A BNF deve ser estendida de forma a permitir que declarações de funções possam ser escritas da seguinte forma:

   fun f $padrao_1$ = exp_1,[if $cond_1$]

   | ...

   | f $padrao_n$ = exp_n, [if $cond_n$]

   onde $padrao_i$ é uma constante ou identificador, exp_i é uma expressão e $cond_i$ é uma expressão booleana.

1.4.2. Linguagem Funcional de Alta Ordem

No projeto anterior, as funções são elementos de primeira ordem, o que significa que as mesmas não podem ser passadas como parâmetro ou resultado da avaliação de uma expressão A possibilidade de definir funções de alta ordem, onde funções têm o status de valor, é uma característica poderosa encontrada na maioria das linguagens funcionais. Portanto, é interessante analisar o impacto de incorporar esta característica ao projeto da Linguagem Funcional 1.

A princípio, a introdução desta facilidade parece requerer esforço significativo, mas, surpreendentemente, a evolução do projeto para incorporar funções como valor é bastante simples e localizada. Graças ao projeto modular, é necessário apenas utilizar o conceito de herança para introduzir a definição de uma função como uma classe que implementa a interface Valor. Como explicado anteriormente, é recomendável distinguir

a categorização de valores concretos, como inteiros, booleanos e strings, de valores abstratos como funções, que não admitem, por exemplo, igualdade sintática. A Figura 20 apresenta as partes relevantes da BNF da Linguagem Funcional 2; os itens omitidos são idênticos à Linguagem Funcional 1.

```
Programa ::= Expressao
Expressao ::= ...
Valor ::= ValorConcreto | ValorAbstrato
ValorAbstrato ::= ValorFuncao
ValorFuncao ::= "fn" Id Id "." Expressao
ValorConcreto ::= ...
Aplicacao:= Expressao"(" ListExp ")"
```

Figura 20. BNF para a Linguagem de Funcional 2

Além da introdução de função como valor, uma outra modificação é que aplicação de função é generalizada para o primeiro argumento ser uma expressão, ao invés de estar restrita a ser um identificador de função, como na linguagem anterior. A razão é que, no contexto de uma linguagem de alta ordem, tal expressão, quando avaliada, pode resultar em uma função que, então, é aplicada aos argumentos.

Apesar do conceito de função de alta ordem ser relativamente elaborado, do ponto de vista da concepção de uma linguagem, a introdução do conceito torna-a mais uniforme. Em particular, o ambiente de execução torna-se uma simplificação do ambiente da linguagem anterior, pois como funções são valores, um único mapeamento é necessário para registrar as associações entre identificadores e valores. Na Linguagem Funcional 1 um mapeamento adicional fez-se necessário para associar identificadores de função às respectivas definições. Por outro lado, a verificação de tipos se torna mais complexa, devido à possibilidade de funções poderem ser argumentos e resultados de outras funções.

O exemplo a seguir define ilustra o fato de uma função poder ser o resultado de uma computação nesta linguagem.

```
let fun id x = x in id
```

A expressão acima define uma função identidade e, ao invés de aplicá-la a um argumento, retorna a própria função como resultado da computação. O próximo exemplo mostra que id pode ser aplicado a qualquer argumento, inclusive uma outra função, já que não há qualquer restrição no parâmetro x.

```
let fun suc x = x + 1 in
    let fun id x = x in id(suc)
```

Neste caso, a avaliação do programa retorna a função sucessor.

A composição de duas funções recebe ambas como parâmetro e aplica estas funções em seqüência, como definido a seguir..

```
let fun pred x = x -1, fun suc x = x + 1 in
    let  fun comp f g x = f(g(x)) in comp(pred,suc,1)
```

No caso particular, a composição está sendo aplicada às funções predecessor e sucessor, o que resulta na função identidade. Portanto, a avaliação do bloco acima resulta em 1. Na realidade, se omitirmos o último parâmetro da composição, o resultado da computação é a função composta propriamente dita.

```
let fun pred x = x -1, fun suc x = x + 1 in
    let  fun comp f g  = fn x . f(g(x)) in comp(pred,suc)
```

No programa acima, a notação `fn x . f(g(x))` representa uma função anônima (mais usualmente conhecida como *expressão lambda*) que tem parâmetro `x` e corpo `f(g(x))`. Portanto, a aplicação `comp(pred,suc)` retorna a função `fn x . pred(suc(x))` que, se simplificada, claramente equivale a `fn x . x`, que é a função identidade.

Funções anônimas podem ser utilizadas para definir funções na forma *currificada ()*, onde os argumentos são fornecidos um por vez. Para ilustrar o conceito, considere a definição `fun soma x y = x + y`. Definida desta forma, uma invocação só é válida se aplicada a ambos os argumentos, como em `soma(1,3)`. Uma alternativa é definir a função como `fun soma x  = fn y . x + y`. Neste caso, a função tem apenas 1 argumento e, portanto, a aplicação `soma(1)` é válida, resultando na função `fn x . x + 1`, que é a função sucessor. Como esta definição de `soma` tem apenas um argumento, a aplicação `soma(1,3)` é inválida. Por outro lado, `(soma(1))(3)` é uma expressão válida, já que `soma(1)` retorna uma função que, em seguida, é aplicada ao argumento `3`. A idéia pode ser facilmente generalizada para um número arbitrário de argumentos.

Como vimos na subseção anterior, computação nas duas linguagens funcionais apresentadas nesta seção (e em programação funcional em geral) é baseada em reescrita de termos. A computação (avaliação) de uma expressão `e` resultar em um valor `v` é também a prova da equação `e = v`. Portanto, além de ser um mecanismo de computação, reescrita é um *método de dedução* (largamente difundido). Reescrita e Lógica Equacional é uma teoria com igualdade que permite a substituição do lado esquerdo pelo lado direito de uma equação, e vice-versa. Igualdade é uma relação de ordem: *reflexiva*, *simétrica* e *transitiva*. Uma propriedade fundamental desta teoria é substitutividade:

$$\forall f: T1 \rightarrow T2;\ x, y: T1 \bullet (x = y) \Rightarrow (f\ x = f\ y)$$

que garante o determinismo de uma função: argumentos iguais possuem a mesma imagem. Igualdade entre funções é definida por *extensionalidade*:

$$(f = g) \Leftrightarrow (\forall x \bullet f\ x = f\ y)$$

que permite determinar a igualdade de duas funções a partir da igualdade da imagem associada a cada elemento do domínio.

Como um exemplo simples de prova equacional, é possível provar que o bloco já apresentado anteriormente

```
let fun pred x = x -1, fun suc x = x + 1 in
    let  fun comp f g  = fn x . f(g(x)) in comp(pred,suc)
```

equivale formalmente á função predecessor. A prova é apresentada aseguir.

```
comp(pred,suc)(x) = pred(suc(x))     [Definição de comp]
                  = pred(x + 1)      [Definição de suc]
                  = x                [Definição de pred]
                  = id(x)            [Definição de id]
```

No caso de definições recursivas, a prova é mais elaborada e tipicamente baseada em indução.

Linguagens funcionais usualmente oferecem várias facilidades adicionais às apresentadas nesta seção. Estruturas como listas, compreensão de listas e padrões recursivos sobre listas e outras estruturas permitem um poder de expressão significativo; estes conceitos estão fora do escopo deste curso.

1.5. Programação Imperativa

O paradigma imperativo está baseado na idéia de um estado que sofre modificações durante a computação. O estado é representado por um conjunto de associações de identificadores (variáveis globais e locais ativas do programa) em valores. A diferença fundamental com relação ao paradigma funcional é que esta associação é dinâmica, sendo constantemente modificada durante a execução de um programa. As linguagens imperativas oferecem uma variedade de comandos cuja execução envolve aspectos de controle, mas essencialmente modificação do estado do programa.

O elemento central de um programa imperativo é o comando, que pode ser primitivo (como atribuição, leitura e escrita) ou composto, como seqüência, condicional e iteração. Um importante mecanismo de estruturação das linguagens imperativas é o procedimento, que, como uma função, pode ser parametrizado, mas tem como corpo um comando, ao invés de uma expressão. Em geral, um programa em uma linguagem imperativa deixa de ser uma expressão e passa a ser um comando. A próxima subseção apresenta a Linguagem Imperativa 1, que inclui alguns comandos usuais. A seção seguinte estende esta linguagem com procedimentos parametrizados e recursivos.

1.5.1 Comandos

Com o objetivo de explorar a implementação de características imperativas, estendemos a Linguagem de Expressão 1 com alguns comandos usuais. A razão de partir da linguagem mais simples de expressão, ao invés da Linguagem de Expressão 2 é que os blocos de declaração nesta linguagem introduzem constantes (associações fixas entre identificadores e valores), enquanto na linguagem imperativa estas associações são dinâmicas. A Figura 21 apresenta a BNF da Linguagem Imperativa 1. Os comandos primitivos são skip (que não tem qualquer efeito), atribuição, escrita e leitura. Os comandos compostos são condicional, iteração e bloco de declaração. As variáveis declaradas no bloco são semelhantes ao bloco let, introduzido anteriormente, mas, como

já explicado, as variáveis declaradas podem sofrer alteração de valor, através de comandos de atribuição e de leitura.

A seguir apresentamos um exemplo simples de programa nesta linguagem.

```
{ var a = 3; write(a);
    { var a = 2, var b = 5; write(a); write(b+a) };
    write(a)
}
```

O resultado produzido é a seqüência 3 2 7 3. No caso do segundo e do terceiro comandos de escrita, o identificador "a" em escopo é o mais interno, com valor 2. No caso do primeiro e do último comando, a declaração mais externa do identificador é a que está em escopo, com valor 3.

```
Programa ::= Comando
Comando ::= "skip" | Atribuicao | IO | Comando ";" Comando
            | IfThenElse  | While | ComandoDeclaracao
Atribuicao ::= Id ":=" Expressao
IO ::= "write" "(" Expressao ")" | "read" "(" Id ")"
Expressao ::= ... como na Linguagem de Expressão 1 (Figura 8) ...
IfThenElse ::= "if" Expressao "then" Comando "else" Comando
While ::= "while" Expressao "do" Comando
ComandoDeclaracao :: = "{" Declaracao ";" Comando "}"
Declaracao ::= DeclaracaoVariavel |  DeclaracaoComposta
DeclaracaoVariavel ::= "var" Id "=" Expressao
DeclaracaoComposta ::= Declaracao "," Declaracao
```

Figura 21. BNF para a Linguagem Imperativa 1

O exemplo próximo exemplo ilustra o uso do condicional e do comando de leitura.

```
{var n = 0, var m = 0;
    read(n); read(m);
    if (m == n) then
        write("valores de entrada iguais")
    else
        write("valores de entrada diferentes")
}
```

Neste caso, a execução bloqueia o programa até que valores sejam fornecidos de alguma entrada padrão. Por exemplo, para a seqüência de entrada 2 3, o programa produz como saída a mensagem valores de entrada diferentes.

Um exemplo simples do comando de iteração é apresentado a seguir.

```
{ var i = 0;
    while not (i == 3) do
    i := i + 1;
    write("Hello World")
}
```

Este programa imprime a mensagem Hello World 3 vezes, sem seqüência.

O ambiente de execução de uma linguagem imperativa é, em geral, mais complexo do que de uma linguagem declarativa. No caso de uma linguagem imperativa, além do mapeamento de identificadores em valores, que é dinâmico ao invés de estático, é necessário representar a entrada e a saída padrão, para permitir a execução dos comandos de entrada e saída. A interface do ambiente de execução da Linguagem Imperativa é apresentada na Figura 22.

```
public interface AmbienteExecucaoImperativa extends AmbienteExecucao {
    public void changeValor(Id idArg, Valor valorId)  throws VariavelNaoDeclarada;
    public Valor read() throws EntradaVazia;
    public void write(Valor v);
    public ListaValor getSaida();
}
```

Figura 22. Código da interface AmbienteExecucaoImperativa

A interface estende AmbienteExecucao com métodos que permitem modificar a associação de um identificador com o valor, leitura do próximo valor disponível na entrada padrão, escrita de um valor na saída padrão e o retorno da lista de saída corrente do programa.

A classe que implementa o comando de atribuição é apresentada na Figura 23. Apenas o código do método responsável pela execução é mostrado; a verificação de tipos pode ser encontrada na implementação completa das linguagens [PLP 2008].

```
public class Atribuicao implements Comando {
    private Id id;
    private Expressao expressao;
    public AmbienteExecucaoImperativa executar(AmbienteExecucaoImperativa ambiente) {
        ambiente.changeValor(id, expressao.avaliar(ambiente));
        return ambiente;
    }
}
```

Figura 23. Código da classe Atibuicao

O comportamento é capturado pela chamada do método .changeValor do ambiente, que altera o ambiente associando o valor resultante da avaliação da expressão (lado direito

da atribuição) ao identificador (lado esquerdo da atribuição); obviamente, o valor anterior associado ao identificador é perdido. O ambiente modificado é explicitamente retornado, como forma de enfatizar a alteração do estado do programa.

O comando de leitura tem um efeito similar à atribuição, só que o valor a ser atribuído é obtido a partir da entrada padrão, ao invés de uma expressão explícita que define o valor de uma atribuição. A execução de um comando de escrita avalia a expressão e inclui o valor resultante na lista de saída. A execução da composição seqüencial de comandos executa o primeiro, atualizando o ambiente, e, em seguida, executa o segundo comando já considerando o ambiente modificado pelo primeiro. Os comandos condicional e iteração são facilmente implementados em termos dos comandos correspondentes de Java. O comando mais elaborado é um bloco de declaração. A implementação é dada na Figura 24.

```
public class ComandoDeclaracao implements Comando {
   private Declaracao declaracao;
   private Comando comando;
   public AmbienteExecucaoImperativa executar(AmbienteExecucaoImperativa ambiente) {
      ambiente.incrementa();
      ambiente = comando.executar(declaracao.elabora(ambiente));
      ambiente.restaura();
      return ambiente;
   }
}
```

Figura 24. Código da classe ComandoDeclaracao

A execução de um comando declaração é implementada de forma semelhante à avaliação de um bloco na Linguagem de Expressão 2: o comando é executado considerando um ambiente que é incrementado para incluir um novo mapeamento para todas as variáveis introduzidas no bloco. O método elabora é responsável por gerar este mapeamento e incluí-lo no ambiente. Em seguida o ambiente é restaurado e o ambiente resultante da execução do comando é retornado.

Um conceito fundamental em uma linguagem imperativa é o de estado, que é representado por uma memória: uma coleção de células, onde cada célula tem status (alocada ou não) e conteúdo, um valor armazenável ou uma constante que representa valor indefinido (digamos, *undefined*). Uma variável denota um valor, o qual pode ser inspecionado e atualizado, e é implementada por uma ou mais células. Uma visão abstrata da memória é um mapeamento direto entre identificadores e valores, como implementamos nas linguagem Imperativa 1. Por exemplo:

{x→5, y→9, z→'a'}

Uma representação mais concreta mapeia identificadores em endereços e um outro mapeamento associa endereços a valores, como no exemplo a seguir.

{x→13, y→72, z→00} {00→'a',...,13→5, 72→9,...,99→*undefined*}

A avaliação de variáveis é feita de forma diferente de acordo com a sua posição no comando de atribuição. No caso de ocorrência de uma variável à esquerda do comando de atribuição (*L-value*) o endereço da célula de memória que implementa a variável (referência) é considerado. Para uma ocorrência de uma variável à direita da atribuição (*R-value*), o valor armazenado na célula de memória que implementa a variável (conteúdo) é adotado.

Apesar de termos considerado apenas variáveis simples, cujos valores são primitivos, as linguagens imperativas usualmente oferecem variáveis compostas, que são agrupamentos de variáveis componentes, como *record* de Pascal, *struct* de C. Estas são variáveis heterogêneas, pois agrupam elementos de diferentes tipos. Algumas variáveis compostas são homogêneas, tipicamente vetores (*arrays*). A atualização das variáveis compostas pode ser seletiva ou total.

A alocação de memória pode ser estática ou dinâmica. As variáveis criadas via declarações são alocadas estaticamente. Variáveis da *heap,* criadas por comandos, são alocadas dinamicamente. Estas variáveis podem ser criadas e destruídas em qualquer parte do programa, por comandos especiais; são anônimas e acessadas via ponteiros. O uso de ponteiros pode resultar em *aliasing* (dois identificadores com o mesmo endereço de memória) e no problema conhecido como referências pendentes (*dangling references*), como conseqüência da desalocação dinâmica de variáveis.

Na implementação da Linguagem Imperativa 1, modelamos apenas variáveis transientes. As linguagens imperativas normalmente oferecem facilidades para persistir valores, que sobrevivem à execução de um programa, como arquivos e bancos de dados.

Vários outros comandos, bem como variações dos comandos aqui considerados, existem em linguagens imperativas. Por exemplo, no caso de atribuição, há variações que permitem atribuições múltiplas, onde o valor de uma expressão é simultaneamente atribuído a vários identificadores, como em `x := y := 0`. Uma outra forma é a atribuição paralela, que associa uma lista de valores a uma lista de identificadores simultaneamente. Por exemplo, `x,y := 2,3` atribui `2` a `x` e `3` a `y`. Várias formas de condicional e iteração também estão disponíveis, bem como composição paralelas de comandos.

Um aspecto não considerado é a possibilidade de efeito colateral em avaliação de expressões. Isto pode ocorrer, por exemplo, se invocação de funções (o mesmo vale para procedimentos e métodos) em expressões atualizarem variáveis globais do programa, ou no caso de *aliasing*. A expressão `f(x) + f(x)` pode ser diferente de `2*f(x)`, caso a avaliação do corpo de `f` cause algum efeito inesperado, como a atualização de uma variável global; isto não faz sentido no contexto de linguagens imperativas puras, mas pode ocorrer em linguagens como Pascal, onde o corpo de uma função é um comando, ao invés de uma expressão. Em `f(x) + f(x)`, o efeito seria replicado, já que há suas invocações a `f`, enquanto que em `2 * f(x)` há apenas uma invocação.

Como exercício, estenda a Linguagem Imperativa 1 com os seguintes recursos:

1. declaração de variável em qualquer ponto de um bloco, como em Java;
2. comandos *continue* e *goto* de C;
3. *arrays* e *records* de Pascal;

4. ponteiros de Pascal; e
5. tratamento de exceções como em Java, mas passando para o *throw* um *string,* não um objeto.

1.5.2 Procedimentos

A parte relevante da BNF da Linguagem Imperativa 2 é apresentada na Figura 25. A linguagem estende a Linguagem de Imperativa 1 com procedimentos parametrizadas e recursivas (de primeira ordem). O corpo de um procedimento é um comando e a chamada de um procedimento também é um comando. A evolução da Linguagem Imperativa 1 para a 2 é análoga à introdução de funções na Linguagem de Expressão 2.

Declaração de procedimento ocorre em blocos de declaração. Na Linguagem Imperativa 1, um bloco pode conter apenas declaração de variáveis; a construção é estendida aqui para permitir também declaração de procedimentos. Um mesmo bloco pode conter os dois tipos de declaração. A declaração de um procedimento inclui uma lista de identificadores (os parâmetros do procedimento) com os respectivos tipos, e um corpo, constituído do comando a ser executado quando o procedimento é invocado. A chamada de um procedimento envolve o identificador do procedimento invocado e uma lista de expressões, que deve obedecer a ordem e os tipos dos parâmetros.

```
Programa ::= Comando
Comando ::= “skip” | Atribuicao | IO | Comando ";" Comando
            | IfThenElse  | While | ComandoDeclaracao
            | ChamadaProcedimento
ComandoDeclaracao :: = "{" Declaracao ";" Comando "}"
Declaracao ::= DeclaracaoVariavel  | DeclaracaoProcedimento
               | DeclaracaoComposta
DeclaracaoVariavel ::= ...
DeclaracaoComposta ::= ...
DeclaracaoProcedimento ::=
      "proc"  Id "(" [ ListaDeclaracaoParametro ] ")" "{" Comando "}"
ListaDeclaracaoParametro ::= Tipo Id | Tipo Id "," ListaDeclaracaoParametro
ChamadaProcedimento ::= "call" Id "(" [ ListaExpressao ] ")"
```

Figura 25. BNF para a Linguagem Imperativa 2

Um exemplo de um bloco que declara um procedimento sem parâmetros e incrementa uma variável global é dado a seguir.

```
{ var a  =  0, proc incA ()  {a := a + 1};
    call incA(); call incA(); write(a)
}
```

O valor produzido é 2, como conseqüência das duas chamadas ao procedimento.

O próximo exemplo ilustra um procedimento recursivo.

```
{ var b = 3,
  proc escreveRecursivo (int a) {
     if (not (a == 0)) then {
        var x = 0; x := a - 1;
        write("Ola");
        call escreveRecursivo(x)}
     else skip
  };
      call escreveRecursivo(b)
}
```

A primeira chamada ocorre como parte do corpo do procedimento e, portanto, é uma chamada recursiva. A segunda chamada é a invocação não recursiva do procedimento. O resultado produzido pelo programa é uma seqüência com três ocorrências de Ola.

A Linguagem Imperativa 2 implementa escopo dinâmico, como as linguagens funcionais apresentadas. O exemplo a seguir ilustra este fato.

```
{ var x = 0, proc p (int y) {x := x + y};
    { var x = 1; call p(3); write(x) };
    call p(4); write(x)
}
```

O procedimento é declarado no contexto de uma variável x global ao programa. A primeira chamada ao procedimento ocorre no contexto de uma outra declaração da variável. Como o escopo é dinâmico, a primeira chamada ao procedimento considera a declaração mais interna e, portanto, o primeiro comando de escrita imprime o valor 4. Na segunda chamada, apenas a declaração mais externa está em escopo e o valor produzido é o mesmo.

O ambiente de execução inclui quatro componentes: uma pilha de mapeamentos de identificadores em valores (memória principal); duas listas de valores (entrada e saída); uma pilha de mapeamentos de identificadores em procedimentos (declarações). Como na Linguagem Imperativa 1, o ambiente pode ser modificado por atribuição, leitura e por declarações de variáveis. A interface do ambiente é apresentada na Figura 26. Além dos métodos da interface do ambiente de execução da Linguagem Imperativa

1, são incluídos métodos para associar um identificador de procedimento à sua definição (envolvendo os parâmetros e o comando) e para retornar uma definição de procedimento dado um identificador.

```
public interface AmbienteExecucaoImperativa2 extends AmbienteExecucaoImperativa {
    public void mapProcedimento(Id idArg, Procedimento procedimentoId) throws ProcJaDeclarado;
    public Procedimento getProcedimento(Id idArg) throws ProcNaoDeclarado;
}
```

Figura 26. Código da interface AmbienteExecucaoImperativa2

A implementação de chamada de procedimento é semelhante à implementação de aplicação de função. O código é apresentado na Figura 27.

```
public class ChamadaProcedimento implements Comando {
    private Id nomeProcedimento;
    private ListaExpressao parametrosReais;
    public AmbienteExecucaoImperativa executar(AmbienteExecucaoImperativa amb) {
        AmbienteExecucaoImperativa2 ambiente = (AmbienteExecucaoImperativa2) amb;
        Procedimento procedimento = ambiente.getProcedimento(nomeProcedimento);
        ambiente.incrementa();
        ListaDeclaracaoParametro parametrosFormais = procedimento
        getParametrosFormais();
        AmbienteExecucaoImperativa2 aux = bindParameters(ambiente, parametrosFormais);
        aux = (AmbienteExecucaoImperativa2) procedimento.getComando().executar(aux);
        aux.restaura();
        return aux;
    }
}
```

Figura 27. Código da classe ChamadaProcedimento

Os atributos da classe são o identificador do procedimento a ser chamado e os parâmetros da chamada (uma lista de expressões). A primeira linha do método de execução requer que o ambiente seja de fato da Linguagem Imperativa 2; isto permite reusar as interfaces, como explicado anteriormente. Em seguida, a definição do procedimento a ser invocado é recuperado no ambiente, a partir do identificador. O ambiente é então incrementado e um mapeamento dos identificadores dos parâmetros formais nos respectivos argumentos é incluído no ambiente. O comando que define o corpo do procedimento é executado neste ambiente incrementado. O mapeamento dos parâmetros é eliminado e o ambiente (refletindo o efeito da execução do comando) é retornado.

Procedimentos e funções são importantes mecanismos de abstração. Enquanto funções são abstrações de expressões, procedimentos abstraem comandos. Como mecanismo computacional, ambos envolvem a idéia de uma invocação. No caso de uma função, a expressão é avaliada e o resultado retornado; no caso de um procedimento, o comando é executado, possivelmente afetando o ambiente, e valores também podem ser retornados via passagem de parâmetro.

Há vários mecanismos de passagem de parâmetro. Tanto as linguagens funcionais como imperativas apresentadas aqui implementam um mecanismo simples de passagem por valor: a expressão utilizada como parâmetro real é avaliada produzindo um valor denominado argumento, que então é associado ao parâmetro (formal) da função ou procedimento. Esta forma é um dos tipos de passagem de parâmetro do mecanismo de cópia; a denominação se dá porque o mecanismo pode ser explicado através de uma atribuição do argumento ao parâmetro formal. Um outro tipo de passagem de parâmetro que segue o mesmo princípio é a passagem por resultado. Neste caso, o valor inicial do parâmetro real na chamada é desconsiderado. O corpo do procedimento é executado e o valor final do parâmetro real é copiado no parâmetro real, que neste caso deve, obrigatoriamente, ser uma variável. A passagem por valor-resultado combina estes dois efeitos. Os mecanismos de cópia possuem uma semântica simples e favorecem boas práticas de programação, pois os valores das variáveis, ao invés de endereços de memória são copiados. Entretanto, a desvantagem é que pode ser ineficiente no caso de valores complexos, como objetos.

Um outro mecanismo de passagem de parâmetro é por referência. Neste caso, o endereço da variável usada como parâmetro real é associado ao parâmetro formal, formando um *aliasing*. Assim, qualquer efeito no parâmetro formal, durante a execução do corpo do procedimento, é imediatamente refletido no parâmetro real (já que ambos passam a referenciar a mesma posição de memória). Este tipo de passagem de parâmetro é extremamente eficiente, pois não há cópia de valores, apenas de referências. Entretanto, *aliasing* é uma das causas freqüentes de erros de programação difíceis de depurar. Esta forma é um dos tipos de passagem de parâmetro do mecanismo de declaração. A intuição para a denominação é que o parâmetro real é avaliado, resultando no argumento, o parâmetro formal denota o identificador de uma declaração local à abstração e o argumento denota o corpo da declaração. Outros exemplos do mecanismo de declaração são a passagem de constantes e de procedimentos.

Há ainda a passagem por nome, onde o parâmetro real não é avaliado antes do início da execução do objeto da abstração; não há argumento. O parâmetro formal é substituído pelo parâmetro real no objeto da abstração, trocando os nomes de variáveis locais para evitar conflitos de nomes. O objeto resultante é executado.

Como exercício, estenda a Linguagem Imperativa 2 com os seguintes recursos:

1. procedimentos com parâmetros opcionais;
2. procedimentos que retornam valores;
3. Procedimentos de alta ordem
4. blocos de declaração com escopo estático.

5. Passagem de parâmetros por resultado, valor-resultado, constante, referência, e nome.

1.6. Programação Orientada a Objetos

Como enfatizado anteriormente, no paradigma orientado a objetos, ao invés do estado monolítico de um programa imperativo, cada objeto tem o status de uma entidade independente, com seu estado próprio e as ações que podem manipular este estado. A Linguagem Orientada a Objetos estende a Imperativa 1 com declarações de classes, criação dinâmica de objetos, e chamada de métodos. Um programa é um comando precedido por declarações de classes. Atributos são similares às variáveis das Linguagens Imperativas 1 e 2. Métodos são similares aos procedimentos da Linguagem Imperativa 2; métodos não são valores e podem ser recursivos e parametrizados, mas só podem ocorrer dentro de declarações de classes. Uma versão simplificada da BNF para esta linguagem é apresentada na Figura 28.

```
Programa ::= "{" DecClasse ";" Comando "}"
Comando ::= “skip” | Atribuicao | IO | Comando ";" Comando
            | IfThenElse | While | ComandoDeclaracao
            | New | ChamadaMetodo
Atribuicao ::= LeftExpression ":=" Expressao
LeftExpression ::= Id | AcessoAtributo
AcessoAtributo ::= LeftExpression.Id | this.Id
Expressao ::= Valor | ExpUnaria | ExpBinaria | LeftExpression | this
Valor ::= ValorConcreto
ValorConcreto ::= ValorInteiro | ValorBooleano | ValorString | ValorNull
ComDeclaracao :: = "{" DecVariavel ";" Comando"}"
DecVariavel ::= Tipo Id "=" Expressao | DecVariavel "," DecVariavel
                | Tipo Id ":=" "new" Id
New ::= LeftExpression ":=" "new" Id
ChamadaMetodo ::= Expressao "." Id "(" ListaExpressao ")" | Expressao "." Id "("")"
DecClasse ::= "classe" Id "{" DecVariavel ";" DecProcedimento "}"
| DecClasse "," DecClasse
Tipo ::= TipoClasse | TipoPrimitivo
TipoClasse ::= Id
```

Figura 28. BNF para a Linguagem Orientada a Objetos

A principal construção nova é o elemento que define declaração de classes. Com relação aos comandos da linguagem, a atribuição a atributos da classe é baseada em acesso qualificado à variável (como em conta.saldo, assumindo que conta é uma instância da classe que tem saldo como atributo); acesso à variável pode ocorrer também em expressões. Como em Java, this representa o objeto corrente e, portanto, é

também uma expressão válida na linguagem. O valor de um identificador de classe antes da criação de um objeto é representado por ValorNull. Em um comando declaração, a declaração de variáveis agora envolve também um tipo explícito. No caso de declaração de identificadores de instâncias de classes, objetos são criados a partir da construção new seguido do nome da classe. A criação de objetos não é uma expressão na linguagem: a construção é limitada à inicialização de uma declaração ou ao comando (New) que define uma forma particular de atribuição para a criação de objetos. A chamada de métodos segue a mesma sintaxe da chamada de procedimentos, já explicada na seção anterior. Como de costume, a declaração de uma classe envolve o nome da classe, a declaração dos atributos e dos métodos; estes últimos seguem a sintaxe da declaração de procedimentos na Linguagem Imperativa 2.

Um exemplo de programa nesta linguagem é apresentado na Figura 29. O programa declara uma classe lista com atributos que representam o valor a ser armazenado e uma lista (a declaração da classe é recursiva). Dois métodos são definidos: um método recursivo que inclui um elemento na lista e um método que imprime os elementos presentes na lista. O comando principal cria ima instância da classe e invoca um método para inserir elementos na lista e depois imprimi-los.

```
{ classe LValor {
      int valor = -100,
      LValor prox = null;
      proc insere(int v) {
          if (this.valor == -100) then { this.valor := v; this.prox := new LValor }
          else {this.prox.insere(v)}
      },
      proc print() {
          write(this.valor);
          If (not(this.prox == null)) then {this.prox.print()} else {skip}
      }
  };
  { LValor lv := new LValor; lv.insere(3); lv.insere(4); lv.print()}
}
```

Figura 29. Exemplo de programa na Linguagem Orientada a Objetos

O ambiente de execução de um programa orientado a objetos é bem mais complexo do que o de um programa imperativo. Além dos métodos já considerados na Linguagem Imperativa 2, a interface do ambiente inclui métodos para mapear nomes de classes em definições de classes; retornar a definição de uma classe dada seu identificador; mapear identificadores, incluindo this, em valores, incluindo referências (este método é herdado e reutilizado da interface Ambiente); mapear referências em objetos; retornar um objeto associado a uma referência; e retornar a referência para a próxima célula a ser alocada. A Figura 30 mostra a interface do ambiente.

```
public interface AmbienteExecucaoOO1 extends Ambiente<Valor>{
    public void mapDefClasse(Id idArg, DefClasse defClasse) ;
    public DefClasse getDefClasse(Id idArg);
    public void mapObjeto(ValorRef valorRef, Objeto objeto);
    public Objeto getObjeto(ValorRef valorRef);
    public ValorRef getProxRef();
}
```

Figura 30. Código da interface AmbienteExecucaoOO1

Parte da implementação do comando New é apresentada na Figura 31.

```
public class New implements Comando{
   private LeftExpression av;
   private Id classe;
   public AmbienteExecucaoOO1 executar(AmbienteExecucaoOO1 ambiente) {
        DefClasse defClasse = ambiente.getDefClasse(classe);
        DecVariavel decVariavel = defClasse.getDecVariavel();
        AmbienteExecucaoOO1 aux =
                                    decVariavel.elabora(new ContextoExecucaoOO1(ambiente));
        Objeto objeto = new Objeto(classe, aux);
        ValorRef vr = ambiente.getProxRef();
        ambiente.mapObjeto(vr, objeto);
        ambiente = new Atribuicao(av,vr).executar(ambiente);
        return ambiente;
   }
}
```

Figura 31. Código da classe New

Os atributos são semelhantes ao da classe que implementa uma atribuição, só que o lado esquerdo pode ser um acesso à variável (explicado anteriormente) e o direito contém o identificador da classe do objeto a ser criado. O método executar recupera a definição da classe associada ao identificador, no ambiente. A partir da definição, os atributos da classe, com os respectivos tipos, são recuperados. Um mapeamento destes atributos em valores pré-definidos é introduzido em um ambiente auxiliar, a partir do qual o objeto é criado. Uma nova referência para o objeto é obtida e o objeto é mapeado no ambiente, associado a esta referência. Por sua vez, a referência é associada ao acesso à variável que forma o lado esquerdo do comando New. Note que este mapeamento é feito como em uma atribuição. Finalmente, o ambiente é retornado.

Originalmente, as noções de encapsulamento e controle de visibilidade (*information hiding*) surgiram com os tipos abstratos de dados, que existem hoje em linguagens de diversos paradigmas, como imperativas e funcionais. Tipos abstratos são unidades que definem tipos através das operações que operam sobre o tipo, não por valores; a representação dos elementos utilizados para implementar o tipo têm sua representação escondida. Potencialmente, isto favorece reuso e manutenção. Os tipos abstratos induzem a uma estruturação da aplicação baseada nas entidades (dados), ao invés de uma decomposição funcional. A experiência tem confirmado que uma decomposição baseada em dados tende a ser mais estável do que uma baseada em funções.

Classes (mais precisamente objetos) são consideradas uma extensão da noção de tipos abstratos com um estado (definido pelos atributos da classe). Uma outra visão é que uma classe é um tipo abstrato acrescido de uma variável do próprio tipo. Com relação às operações, um dos argumentos de cada operação do tipo é (implicitamente) a variável utilizada para consultar o estado ou armazenar o resultado de tal operação, através da modificação do estado do objeto. Conceitos adicionais como herança e associação dinâmica (*dynamic binding*) oferecem mecanismos adicionais de estruturação a uma aplicação organizada em classes. A linguagem orientada a objetos apresentada nesta seção implementa o conceito de classe, mas sem mecanismos de visibilidade, herança ou associação dinâmica.

Como exercício, estenda a Linguagem Orientada a Objetos 1 com os seguintes recursos:

1. remoção dinâmica de objetos;
2. verificação dinâmica de tipos, como o *instanceof* de Java;
3. construtores (inicializadores) como em Java;
4. Expressão *new* para criação de objetos; e
5. Mecanismos de visibilidade.

1.7. Conclusões

Combinando conceitos e técnicas de orientação a objetos e padrões de projeto, propomos uma estratégia modular e uniforme para o projeto e implementação incremental de construções de programação. O uso de herança e do padrão *Interpreter*, em particular, tornou possível evoluir a família de linguagens através de um processo de extensões conservativas, com poucas exceções. Mostramos como a estratégia pode ser utilizada para implementar alguns dos conceitos mais representativos encontrados nos paradigmas funcional, imperativo e orientado a objetos. .

O *framework* desenvolvido tem sido utilizado e continuamente avaliado e evoluído no contexto da disciplina Conceitos e Paradigmas de Programação, na Pós-Graduação no Centro de Informática da Universidade Federal de Pernambuco, bem como de uma versão compacta do curso apresentado para o Software Engineering Institute of the United Nations University (UNU-IIST). Nossa experiência tem mostrado que os alunos absorvem mais facilmente os conceitos de programação, dada a

oportunidade de acompanhar e implementar vários conceitos de forma progressiva. Alguma elementos do *framework* não foram considerados neste texto, como os *parsers* para as linguagens, que permitem aos estudantes programarem em uma sintaxe concreta e estenderem as linguagens analisando tanto o impacto na sintaxe como no comportamento capturado pelos interpretadores. O *website* do curso [PLP 2008] contém a BNF e os *parsers* das linguagens, bem como o código fonte, o javadoc, e os diagramas de classes dos interpretadores.

Haynes [Haynes 1998] discute as vantagens de usar uma abordagem baseada em interpretadores, bem como apresenta uma série de considerações para ensinar conceitos de programação de forma bem sucedida. Kamin [Kamin 1990] apresenta interpretadores para vários conceitos de programação, mas estas abordagens não possui uniformidade, uma vez que os autores utilizam linguagens distintas e não relacionadas na apresentação dos conceitos de programação. Ghezzi e Jazayeri [Ghezzi & Jazayeri 1997] utiliza uma máquina abstrata chamada SIMPLESM, cujo conjunto de instruções se assemelha ao de uma linguagem de montagem, forçando os estudantes a codificarem em uma linguagem de baixo nível. Uma estratégia mais recente [Christiansen 2002] propõe o uso de Prolog como meta-linguagem para descrever os conceitos de programação; tal descrição é considerada tanto uma especificação de alto nível como uma implementação de um interpretador para construções de programação. Entretanto, devido à ausência de conceitos como herança e outros conceitos, Prolog não contribui para a concepção de um *framework* modular como o apresentado aqui.

Nossa abordagem tem a vantagem de favorecer um projeto progressivo. Os elementos das linguagens são sempre reutilizados no projeto de conceitos mais elaborados. Apesar de não ter sido explorado neste texto, nós também consideramos a integração de linguagens de diferentes paradigmas, combinando implementações existentes. Em [Maranhão 2004], é enfatizado o uso de aspectos e de outros paradigmas (como *Strategy, Memento e Adapter*) para prover a integração de linguagens de maneira ortogonal, formando linguagens multiparadigmas. Programação lógica é também adicionada ao conjunto de paradigmas considerado.

De forma mais ampla, várias extensões têm sido propostas, pelos próprios estudantes, ao longo das edições do curso. Com relação ao paradigma funcional, tipos estruturados (tuplas, listas, conjuntos e tipos algébricos e abstratos), avaliação sob demanda, avaliação parcial foram implementados. Do lado imperativo, alguns conceitos implementados foram vetores, referências, registros, mecanismos de passagem de parâmetro, e procedimentos de alta ordem. Com relação à orientação a objetos, herança, associação dinâmica e exceções foram considerados. Outros paradigmas, além de programação lógica, mencionado acima, foram também incorporados: programação orientada a aspectos, concorrência e agentes.

A estrutura do *framework*, baseada no padrão *Interpreter*, favorece a implementação de outras aplicações, como editores e provadores de teoremas. Para cada uma destas aplicações, a idéia é implementar um método semelhante ao de avaliação de expressões e de execução de programas, para cada elemento da BNF da linguagem.

Neste trabalho, adotamos uma abordagem prática para o projeto e implementação de linguagens, baseada na interpretação de cada elemento da linguagem. Seria interessante investigar uma abordagem mais rigorosa, baseada em semântica

formal. Trabalhos como os baseados em semântica de ações [Mosses 1992] pode ser um ponto de partida viável para o nosso contexto, pois uma preocupação central em semântica de ações é permitir uma apresentação modular e incremental dos conceitos de programação. A adoção de uma semântica formal permite uma ênfase em prova e transformação de programas, bem como a geração automática de compiladores para as linguagens.

Agradecimentos

Toda a concepção desta estratégia de projeto e ensino de linguagens multiparadigmas foi desenvolvida conjuntamente com Paulo Borba, a quem os autores são especialmente gratos. O material de aulas preparado conjuntamente com Paulo Borba foi uma importante base para a elaboração deste texto. A implementação de uma família de linguagens, com um ambiente uniforme de execução, só foi possível com muita dedicação e iniciativa de vários alunos do curso de Paradigmas de Linguagens de Programação, da Pós-Graduação do CIn-UFPE. Em especial, agradecemos a Adeline Souza, Allan Araújo, André Furtado, Ângelo Ribeiro, Ayla Dantas, Eduardo Aranha, Eduardo Laureano, Eduardo Tavares, Gibeon Aquino, Gustavo Santos, Joabe Jesus, Leonardo Cole, Marcus Machado, Rafael Duarte, Rafael Oliveira, Rodrigo Ramos, Rohit Gheyi, Sérgio Soares, Thiago Massoni e Vander Alves.

Referências

Backus, J. (1978) "Can Programming be liberated from the von Neumann style?: a functional style and its algebra of programs", Communications of the ACM, Vol. 21, No. 8.

Bird, R. Introduction to Functional Programming Using Haskell, Prentice Hall, 1998.

Bowen, J. (1993) "From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation", Journal of Software Maintenance: Research and Practice, **5**(4):205-234..

Budd, T. A., "Multiparadigm Programming in LEDA", Addison-Wesley, 1995.

Christiansen, H. (2002) "Using Prolog as Metalanguage for Teaching Programming Language Concepts", In Kacprzyk J., M. Krawczak and S. Zadrozny (eds.) Issues in Information Technology, EXIT, Warszawa, p. 59-82.

Elrad, T., Filman, R. E., Bader, A. (2001) "Aspect-Oriented Programming: Introduction", Communications of ACM, Vol. 44, Issue 10, ACM Press, p. 29-32.

Flanagan, D., "Java in a Nutshell", O'Reilly & Associates, November 1999.

Fowler, M., Scott, K., "UML Distilled: A Brief Guide to the Standard Object Modeling Language", Addison Wesley, 2000.

Gamma, E., Richard, H., Johnson, R., Vlissides, J., "Design Patterns – Elements of Reusable Object-Oriented Software", Addison Wesley, 1994.

Ghezzi, C., Jazayeri M. "Programming Language Concepts", Wiley Text Books, 1997.

Gosling, J., Joy, W. and Steele,G., “The Java Language Specification”, Addison-Wesley, 1996.

Kamin, S. N., Programming Languages: An Interpreter-based Approach, Addison Wesley, 1990.

Hannemann, J. and Kiczales, G. (2002) “Design Pattern Implementation in Java and AspectJ”, In Proceddings OOPSLA, ACM SIGPLAN Notices, p. 161-173.

Hanus M. (1994) “The Integration of Functions into Logic Programming: From Theory to Practice”, Journal of Logic Programming, Vol. 19&20, p. 583-628.

Haynes, C. T. (1998), “Experience with an Analytic Approach to Teaching Programming Languages”. Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, ACM Press, p. 350-354.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM, 12:576–580, 1969. 115.

Hoare, C. A. R. *et al*. (1987) Laws of Programming. Communications of the ACM, 30(8):672–686, 1987.

Kowalski, R. A. (1985) “Directions for Logic Programming”, Proceedings of IEEE Symposium on Logic Programming.

Liu, M. (2001) “Pluto: An Object-Oriented Logic Programming Language”, 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems.McLaughlin, B., Flanagan, D. “Java 1.5 Tiger: A Developer's Notebook”, O’Reilly, 2004.

Maranhão, A. (2004) “Incremental Design of Programming Language Concepts”, Dissertação de Mestrado, Centro de Informática da Universidade Federal de Pernambuco.

Meyer, B. (1988) “Object-Oriented Software Construction”, Prentice Hall, 1988.

Mosses, P. (1992). “Action Semantics”, Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

Ng, K. W. and Luk, C. K. (1995), “A Survey of Languages Integrating Functional, Object-oriented and Logic Programming”. Journal of Systems Architecture (formerly Microprocessing and Microprogramming), ed.41, p. 5-36.

Paulson, L. C., “ML for the Working Programmer”, Cambridge University Press, 1996.

PLP (2008). Curso de Paradigmas de Linguagens de Programação. Centro de Informática da Universidade Federal de Pernambuco. www.cin.ufpe.br/~in1007.

Plotkin, G. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, AArhus University, Computer Science Department, 1981.

Qian, Z. and Moulah, B. A. (2000) “Combining Object-oriented and Functional Language Concepts”, Journal of Software, ed. 11, p. 8-22.

Robinson, J. A. (1965) “A Machine-Oriented Logic Based on the Resolution Principle”, Journal of the ACM, Vol. 12, No. 1, p. 23-41.

Schildt, H. C# 2.0: The Complete Reference (Complete Reference Series)., 2006.

Schmidt, D. Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Inc, 1986.

Shoham, Y. (1993). “Agent Oriented Programming.” *Artificial Intelligence* 60(1), 51-92.

Van Roy, P. and Haridi S. (1999) "Mozart: A Programming system for Agent Applications", International Workshop on Distributed and Internet Programming with Logic and Constraint Languages, Las Cruces NM.

Watt, D. Programming Language Design Concepts, John Wiley & Sons, 2004.

Watt, D. Programming Language Concepts and Paradigms, Prentice Hall, 1990.

Xi, H. (2002) “Unifying Object-Oriented Programming with Typed Functional Programming”, Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, p. 117-125.