# Proposta de uma linguagem de alto nível básica para a codificação de softwares automodificáveis

(21 Janeiro 2010)

S. R. B. da Silva, J. J. Neto

***Resumo* — Este trabalho propõe uma linguagem simples de alto nível, dotada de recursos que facilitam a codificação de programas automodificáveis. Para isso, estendeu-se uma linguagem de programação convencional com comandos e declarações especiais voltados à especificação de alterações dinâmicas do código. A linguagem resultante permite, dessa forma, que o programador indique as alterações desejadas para o seu código, modificações essas que se efetuam no momento da execução do programa. Como resultado, torna-se disponível uma linguagem de programação apropriada para o desenvolvimento de aplicações adaptativas. Neste texto procura-se descrever os principais aspectos do projeto e da implementação de uma linguagem dessa natureza, e um pequeno exemplo de aplicação é apresentado para ilustrar sua utilização.**

***Palavras chave* — Adaptatividade, linguagem de programação, código automodificável, tecnologia adaptativa.**

## I. INTRODUÇÃO

EM meados do século passado, quando os computadores careciam de recursos de armazenamento, sempre foi prática corrente entre os programadores procurar reaproveitar a escassa memória física então disponível, à medida que seus programas cresciam além da capacidade máxima dessa memória. Esta prática se manifestava pela automodificação de código – neste artigo, entende-se como automodificação a propriedade de um programa que tem a capacidade de se automodificar enquanto em execução, sem a intervenção de um agente externo ao processo – e era facilitada pelo uso extensivo de programação em linguagens de baixo nível e, também, pelo uso limitado de metodologias de programação, então incipientes [1].

Com o advento da Engenharia de Software e o surgimento de metodologias de programação e novos paradigmas de desenvolvimento de programas, a prática da automodificação de código teve seu uso reduzido drasticamente, uma vez que um código de programa armazenado em memória é considerado imutável. [2]

Recentemente, algumas aplicações especiais voltaram fazer uso dessa técnica conforme pode ser visto em [3], [4], [5], entre várias outras.

De acordo com [6], a adaptatividade, por sua vez, fundamenta seus métodos na evolução dinâmica do conjunto das regras que definem o comportamento do fenômeno adaptativo, podendo-se considerar natural o uso de código automodificável na implementação de programas que realizam procedimentos adaptativos.

O presente texto propõe-se a proporcionar aos programadores de softwares adaptativos, uma linguagem de programação simples e de alto nível, com recursos para especificação de operações de automodificação nos seus programas.

Tratando-se de uma linguagem experimental, através da qual se possam explorar as possibilidades de linguagens dessa natureza, optou-se por implementar uma extensão sintática, que possa ser aplicada a uma linguagem imperativa existente. Com essa extensão, o programador pode especificar as automodificações que deseja imprimir ao programa, cuja realização se efetiva em tempo de execução.

A linguagem resultante deve incorporar, dessa maneira, recursos que permitam ao programador de aplicações adaptativas exprimir seus programas de uma forma natural, apesar da presença de automodificações no mesmo.

## II. FUNDAMENTAÇÃO CONCEITUAL

Um programa adaptativo pode ser entendido como sendo um código adaptativo, ou seja, uma sequência automodificável de instruções. À luz da formulação geral apresentada em [7], programas dessa classe podem ser formalmente considerados como sendo dispositivos adaptativos especiais, cujo dispositivo subjacente não-adaptativo seria um programa convencional (este, interpretado como um conjunto de regras).

Analisando-se programas escritos em diversos estilos e paradigmas, constata-se que é viável converter programas escritos em um dado estilo para qualquer outro estilo, preservando-se seu significado. Assim, por exemplo, durante muito tempo foi usual o emprego de práticas baseadas na conversão de programas escritos em estilo recursivo para a forma de programas iterativos e vice versa. [9], [9]. Igualmente, programas desenvolvidos em paradigma funcional e lógico, fortemente baseados no estilo recursivo, costumam fazer uso de recursos de macros, conforme se pode ver em [10].

Considerando que, no estilo adaptativo, a aplicação de automodificações muitas vezes reflete a substituição de grupos de regras por outros grupos de regras, pode-se considerar que

S. R. B. da Silva é professor da Universidade do Estado do Amazonas e cursa mestrado em Engenharia Elétrica pela Universidade de São Paulo (sramosbs@usp.br).

J. J. Neto é o responsável pelo LTA – Laboratório de Linguagens e Técnicas Adaptativas, vinculado ao Departamento de Engenharia e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo – Av. Prof. Luciano Gualberto, trav. 3, No. 158 – Cidade Universitária – São Paulo – SP – Brasil (joao.jose@poli.usp.br) fone 55(11)3591 5402

os formalismos baseados em cálculo lambda [11], que dão apoio à utilização de funções e de macros, possam ser analogamente empregados na formalização do estilo adaptativo.

Macro [10], é um recurso de programação que permite ao programador especificar um procedimento apenas uma vez e usá-lo quantas vezes sejam necessárias dentro do programa, através de uma chamada a essa macro, como se fosse um subprograma (função/procedimento). A principal diferença entre o uso de macro e subprograma reside no fato de que, em tempo de compilação, a macro é acrescentada ao código. Uma vantagem do uso de macro é um aumento de performance do programa, uma vez que esta aumenta a velocidade de execução do código, pois evita o gerenciamento de chamada de função. Todavia, o uso extensivo de macros tende a causar aumentos não desprezíveis no tamanho do código, especialmente no caso em que a macro for grande.

Subprogramas, conforme [11], por sua vez, são procedimentos ditos fechados, cujo código é único no programa, de forma que não acontecem ocorrências múltiplas do mesmo causadas por seu uso em diversos pontos do programa.

No entanto, do ponto de vista formal, pode-se reconhecer que programas em que há algum tipo de repetição podem ser expressos indiferentemente, quer usando macros, quer subprogramas, ou, ainda, técnicas adaptativas, nas quais as chamadas dessas abstrações são expandidas somente quando o comando correspondente for, de fato, acionado em tempo de execução.

Os primeiros computadores foram projetados para processamento de aplicações científicas, as quais utilizavam estruturas de dados simples e elevada quantidade de cálculos em ponto flutuante, de acordo com [12].

A arquitetura básica desses computadores era constituída de uma memória e uma unidade central de processamento – UCP. Os dados e instruções de programa ficam na mesma memória. Os dados são canalizados para a UCP, processados e o resultado retorna para a memória. É a arquitetura de von Neuman. A maioria das linguagens de programação foi projetada em função dessa arquitetura. São as chamadas linguagens imperativas. Essa arquitetura leva os programas a terem as variáveis como recurso central. As instruções são armazenadas em posições contíguas de memória, o que torna o processamento eficiente, diz [12].

Com o passar do tempo, os computadores passaram a ser empregados para outras finalidades e começaram a surgir linguagens de programação com características distintas.

De acordo com as suas características, as linguagens de programação podem ser classificadas em genéricas e de propósito específico, de acordo com 0.

Linguagens genéricas, ou de propósito geral, são as que podem ser utilizadas para o desenvolvimento de aplicações em geral. São dotadas de recursos que permitem empregos na solução de diversos problemas.

Bibliotecas dedicadas podem ser acrescentadas a uma linguagem genérica, o que a torna bem mais poderosa e flexível. Essas bibliotecas são chamadas de extensões funcionais e permitem o desenvolvimento de aplicações de maneira mais eficiente. [14]

A necessidade de implementar determinados tipos de modelos com frequência levou ao desenvolvimento de linguagens com propósitos específicos. São linguagens que possuem interfaces que facilitam o trabalho do programador.

O desenvolvimento de aplicações usando uma linguagem de propósito geral, como foi o caso do FORTRAN, no passado, ou C, C++ e Java, entre outras, nos dias atuais, requer do programador um maior esforço de programação e domínio da linguagem, para o desenvolvimento de solução para problemas específicos, a não ser que se utilize algum bom repositório de bibliotecas disponível. Naturalamente, quando estão disponíveis tais bibliotecas, sua utilização pode até dispensar a de linguagens de propósito específico para essa finalidade.

Um exemplo de linguagem para propósito específico são as linguagens de simulação, projetadas com o fim determinado de permitir o desenvolvimento de simulações, como é o caso das linguagens SIMAN, SIMULA e SIMSCRIPT, como em 0.

Outro tipo de linguagem para fins específicos são as linguagens de marcação. São assim classificadas por utilizarem marcadores, que são palavras chave com função de delimitar blocos de dados e podem conter parâmetros. Como exemplo dessas linguagens temos XML, que foi desenvolvida pela W3C (Word Wide Web Consortium) e é definida como um formato universal de dados estruturados na WEB. [15]

Existem várias outras linguagens de desenvolvimento para Web, entre as quais podemos citar, ainda, OWL, para descrever semântica, RuleML, uma linguagem canônica para regras na Web, SWRL, que propõe combinar RuleML e OWL, como se pode ver em 0.

Linguagens de programação também podem ser classificadas de acordo com o paradigma de programação segundo o qual foram projetadas.

### A. *A. Paradigmas*

Problemas podem ser resolvidos de acordo com um padrão de resolução associado a um gênero de linguagem de programação. A esse padrão de resolução de problemas, dá-se o nome de paradigma de programação. Existem diversos paradigmas de programação. De acordo com [11], a partir dos anos 70 quatro desses paradigmas de programação evoluíram: programação imperativa, orientada a objetos, funcional e lógica. Serão descritas, a seguir, sempre com fundamento em [11], as características principais desses quatro paradigmas de programação:

#### *1) Programação Imperativa*

O paradigma imperativo tem por base teórica a Máquina de Turing, e como lastro tecnológico, a arquitetura de von Neumann. Nesse paradigma, a idéia central é o uso de efeitos colaterais para a alteração do estado de um programa, e essa função é desempenhada principalmente pelos comandos de atribuição – que consistem em alterar o estado de um programa através da substituição de um valor, contido em uma posição de memória, por algum outro valor. Nesse paradigma,

a atribuição é uma idéia central.

Quando uma linguagem é capaz de fornecer uma base que permita a implementação de qualquer algoritmo possível de ser projetado, essa linguagem é Turing-Completa. Dessa forma, uma linguagem de programação imperativa que disponibilize variáveis e valores inteiros, as operações aritméticas básicas, comandos de atribuição, comandos condicionais e iterativos é considerada Turing-completa.

Além de atribuição, as linguagens de programação imperativas costumam disponibilizar ao programador: declaração de variáveis, expressões, comandos condicionais, comandos iterativos e abstração procedural.

Segundo [12], abstrair é empregar apenas as informações relevantes sobre um problema para representá-lo dentro de uma determinada perspectiva. Destina-se a simplificar o raciocínio e o processo de programação.

A abstração procedimental dá ao programador a possibilidade de atentar para as relações existentes entre um procedimento e a operação que ele realiza (em particular, entre uma função e o cálculo que ela executa) sem preocupações acerca da maneira como essas operações são realizadas.

O refinamento gradual é uma maneira sistemática de desenvolver programas. Usando abstração procedimental, um programador pode particionar uma função, idealizada de forma macroscópica, em um grupo de funções mais simples e/ou mais específicas.

Com o fim de simplificar o desenvolvimento de algoritmos complexos, as linguagens imperativas modernas possuem suporte a matrizes e estruturas de registro, além de bibliotecas extensíveis de funções, que evitam que operações comuns necessitem ser reprogramadas, como é o caso de operações de entrada e saída de dados, gerenciamento de memória, manipulação de cadeias e de estruturas de dados clássicas, e tantas outras.

São exemplos de linguagens imperativas FORTRAN e C.

Como parte do processo de evolução da programação imperativa, buscou-se estender a abstração procedimental para incluir tipos de dados abstratos. Isto se deu através da criação e incorporação do conceito de encapsulamento de tipos, ainda no final da década de 60. Este foi um passo importante para o desenvolvimento do paradigma de programação orientada a objetos, que será comentado a seguir.

*2) Programação Orientada a Objetos*

Encapsular é agrupar constantes logicamente relacionadas, tipos, variáveis, métodos e outros em uma nova entidade.

A abstração de dados encapsula os tipos de dados e as respectivas funções em um único bloco ou pacote. Mas o bloco (ou pacote), não tem mecanismo de inicialização e finalização de um valor, nem uma maneira simples de acrescentar novas operações. Essas duas situações foram solucionadas pela idéia de classe, um conceito fundamental para a orientação a objetos.

A orientação a objetos envolve a utilização de conceitos, tais como o de classes, herança e polimorfismo, devendo-se notar, no entanto, que nem todas as características teóricas que a definem, estão obrigatoriamente presentes em todas as linguagens consideradas aderentes a esse paradigma.

As classes existem dentro de uma hierarquia. Uma classe pode ser subclasse de outra e esta será, neste caso, considerada sua superclasse. Assim, uma subclasse poderá herdar de sua superclasse variáveis e métodos. A herança é uma forma de reutilização de código.

Quando uma classe herda, de mais de uma superclasse, variáveis e métodos, identifica-se aí o conceito de herança múltipla.

Quando, da chamada de um método, resultarem chamadas a mais de uma forma de implementação de tal método, observa-se uma instância do conceito de polimorfismo.

As linguagens Smalltalk e Java são, basicamente, orientadas a objetos.

*3) Programação funcional*

O paradigma funcional tem como fundamentação teórica o cálculo Lambda. Devido ao uso extensivo da recursão, apóia-se, tecnologicamente, em arquiteturas nas quais o programador possa, de forma relativamente natural, fazer uso da recursão para exprimir sua lógica e executar seus programas.

O paradigma funcional baseia-se, fundamentalmente, no conceito de função, e tem como uma de suas principais características o conceito de transparência referencial, que se traduz, na prática, na ausência de efeitos colaterais.

Para que isso seja possível, diferentemente do que ocorre em linguagens imperativas, uma linguagem concebida estritamente de acordo com o conceito do paradigma funcional não utiliza variáveis nem comandos de atribuição.

As estruturas típicas encontradas em uma linguagem funcional são:

- Um conjunto de dados, que são representados por estruturas de alto nível, baseadas em listas.
- Um conjunto de definições de funções primitivas préconstruídas, destinadas, principalmente, à manipulação dos dados.
- Especificações de aplicações das funções, que podem ser formas funcionais para construção de novas funções.

A ausência de variáveis impossibilita a construção de programas baseados em lógica iterativa, uma vez que estes exigem variáveis de controle.

Tarefas repetitivas devem, portanto, ser expressas por algum outro tipo equivalente de estrutura, e nas linguagens funcionais costumam ser realizadas por meio de procedimentos e de funções recursivas.

A linguagem LISP foi a primeira linguagem de programação funcional. No entanto, nos dias atuais, incorpora características de outros paradigmas. Outro exemplo de linguagem funcional é a linguagem ML.

*4) Programação Lógica*

O paradigma lógico apóia-se na fundamentação teórica proporcionada pela Lógica de Primeira Ordem. A recursão também é extensivamente empregada na programação em lógica, a qual se apóia, tecnologicamente, nas arquiteturas que reduzam para o programador as dificuldades impostas pelo uso da recursão.

A programação em lógica tem por base a programação declarativa, na qual se procura privilegiar a especificação da tarefa específica que se deseja que o computador execute, evitando que o programador especifique de que forma o computador deverá proceder para realizá-la.

Dessa forma, enquanto nos demais paradigmas a meta do programador é de informar à máquina a trajetória exata a ser percorrida pelo programa, na programação lógica o programador deve limitar-se a especificar as premissas e os alvos a serem atingidos, deixando para a máquina a determinação do caminho a ser percorrido para solucionar o problema.

Para tanto, um programa lógico costuma efetuar processamento simbólico, não numérico. Programas lógicos costumam usar como linguagem de programação alguma implementação da lógica simbólica.

A exemplo do que acontece com programas escritos no paradigma funcional, os programas lógicos costumam utilizar a mesma notação formal para representar programas e dados, não fazendo distinção entre argumentos de entrada e saída.

Assim como os programas escritos em paradigma funcional, as linguagens voltadas à programação em lógica não costumam dispor de comandos de controle para especificar repetições, os quais são extensivamente substituídos por formas recursivas equivalentes para esta finalidade.

Um programa em linguagem lógica é tipicamente constituído de cláusulas, que podem ser de dois tipos: fatos, que são considerados verdadeiros por definição, e regras, a serem aplicadas sobre os fatos e outras regras, e cuja avaliação pode resultar verdadeira ou falsa.

Prolog, Planner e Oz são linguagens para programação lógica.

Embora algumas linguagens de programação originalmente tenham sido projetadas estritamente de acordo com um dos paradigmas acima, a elas posteriormente foram incorporadas características de outros paradigmas, de modo que raramente se encontra uma linguagem que possa, atualmente, ser classificada como sendo autenticamente pertencente a um ou outro desses paradigmas.

Existem linguagens, denominadas extensíveis, que permitem ao programador definir extensões a uma linguagem básica inicial, facilitando, assim, a criação de características específicas para a solução de classes particulares de problemas de seu interesse.

### *B. B. Extensibilidade*

As primeiras publicações científicas sobre a extensibilidade de linguagens de programação ocorreram ainda na década de 60 e prosseguiram nos anos 70, inclusive com a realização de simpósios sobre linguagens extensíveis nos anos de 1969 e 1971, diz [17].

De acordo com [18], a extensibilidade possibilita ao programador modificar os recursos de uma linguagem de programação, tornando-a adequada a propósitos específicos.

A extensibilidade de linguagens de programação pode ocorrer de três formas:

Extensibilidade léxica, segundo a qual criam-se abreviaturas, paramétricas ou não, para trechos específicos de texto, de tal modo que, toda vez que tal abreviatura for encontrada, o trecho de texto a elas associado é usado em substituição à abreviatura. O mesmo é feito em relação aos argumentos em substituição aos correspondentes parâmetros.

A Linguagem C costuma ser implementada como linguagem lexicamente extensível, uma vez que permite a criação de abreviaturas textuais (DEFINE).

A extensibilidade funcional consiste em encapsular novas funcionalidades a funções definidas e tradicionalmente usadas de uma linguagem de programação. [19].

LISP é um exemplo de linguagem funcionalmente extensível.

A extensibilidade sintática diz respeito à capacidade de uma linguagem de programação permitir a adição de novos constructos sintáticos, sendo que estes podem ser combinados com os já existentes, ainda conforme [18].

Para a solução de problemas de naturezas diversas, vários estudos vêm sendo desenvolvidos empregando programação adaptativa.

### *C. C. Programação adaptativa*

Um fenômeno adaptativo é caracterizado por um comportamento dinâmico, que altera seu funcionamento em função de ter atingido alguma situação particular e, nesta situação, ter recebido um determinado estímulo. Uma maneira possível de representar um comportamento adaptativo através de uma linguagem de programação é utilizar uma linguagem, cujo código executável tenha a capacidade de modificar-se ao acrescentar ou remover porções de seu código a si próprio durante a execução. E muitas das mais importantes linguagens atuais não apenas não foram projetadas para apresentarem essa característica como também incluem propriedades que dificultam ao usuário a execução de tais operações em seus programas.

Linguagens de programação devem, por sua natureza e propriedades, ser classificadas, de acordo com a hierarquia de Chomsky, como linguagens dependentes de contexto (ou sensíveis ao contexto), e, portanto, geradas por gramáticas sensíveis ao contexto. Considerações de ordem prática, entretanto, fazem ser preferível representá-las não em sua forma autêntica, mas aproximada, por variantes livres de contexto, com a vantagem de disponibilizar para elas todo um ferramental disponível para este tipo de linguagens menos complexas. As lacunas introduzidas por tal simplificação, todavia, devem ser compensadas por meio do acréscimo de procedimentos auxiliares que efetuem as operações necessárias ao tratamento das dependências de contexto, e que não são supridas pelo formalismo livre de contexto que define a linguagem simplificada.

Uma linguagem para programação adaptativa constitui um caso particular de linguagens de programação, diferenciando-se das não-adaptativas pela capacidade que possui de permitir a geração de código automodificável.

A próxima seção analisa os requisitos de uma linguagem para programação adaptativa, comenta sobre as exigências do

ambiente de execução e apresenta uma proposta de uma linguagem dessa natureza.

## III. PROPOSTA DA LINGUAGEM

A pesquisa acerca do desenvolvimento de uma linguagem para programação adaptativa vem se desenvolvendo há alguns anos, conforme se pode ver em [1], [20][21].

Buscando contribuir com mais um passo nessa direção, propõe-se uma linguagem básica de alto nível, em estilo imperativo, que permite escrever códigos fonte que originarão programas adaptativos, de acordo com o que foi sugerido anteriormente.

Faz-se mister definir, neste ponto, os termos código automodificável e linguagem para programação adaptativa, de acordo com [22].

Neste trabalho, adota-se o termo "código adaptativo" para significar um código executável cujas propriedades permitem que o mesmo se automodifique autonomamente, de forma dinâmica, por acréscimo e/ou supressão de comandos do programa.

Um programa escrito em alguma linguagem de alto nível convencional pode ser considerado estático, na medida em que não pode ser executado diretamente por uma máquina, necessitando, para tal, ser antes interpretado, ou então transformado em código executável por um compilador.

Mesmo assim, se essa linguagem de alto nível permite a produção de código fonte que apresenta comportamento dinâmico, pode-se considerar que se trata de um código adaptativo, pela idéia que traduz, estendendo, dessa forma, o significado dessa expressão ao código-fonte.

Trabalhos anteriores relativos às bases para o estabelecimento de uma linguagem que permita escrever programas com as características acima citadas, como em [20], adotavam o termo "linguagem adaptativa" para significar uma linguagem que permita escrever códigos adaptativos.

Vista como um dispositivo guiado por regras [7], a linguagem não pode se alterar, ainda que as regras gramaticais se modifiquem durante a utilização da gramática. No entanto, o conjunto de sentenças compreendido na linguagem não pode se alterar, ou a linguagem não mais seria aquela.

Assim, é fácil intuir que o termo "linguagem adaptativa" não se aplica e, portanto, passa-se a adotar a denominação "linguagem para programação adaptativa", para especificar uma linguagem de alto nível que venha a permitir o desenvolvimento de códigos adaptativos.

Para [20], é desejável que uma linguagem para programação adaptativa deva ter, como requisitos, principalmente:

- maneiras de endereçar a porção de código que poderá ser modificada, como parte de sua sintaxe, bem como que tipo de modificação será efetuada.

- Estar associada a um ambiente de execução, que possibilite ao compilador referenciar, no código objeto gerado, as operações adaptativas definidas para esse ambiente de execução.

- Um conjunto de operadores adaptativos, que expressem de forma clara e intuitiva as operações adaptativas disponíveis.

Considerando que a lógica de um programa com código automodificável pode ser desenvolvida na forma de um programa convencional, de código estático, pode-se argumentar que a adaptatividade seja supérflua, e dado que o código necessário para redigir programas automodificáveis tende a ser mais extenso e mais complexo que um programa convencional, a programação adaptativa pode exigir práticas nem sempre alinhadas com as usualmente recomendadas para a construção de programas, atualmente considerados de boa qualidade.

Em atenção a posicionamentos como esses, os sistemas operacionais atuais dispõem de mecanismos de segurança que procuram impedir que os códigos executáveis em memória sofram alterações (ou mesmo se imponham automodificações). Esse aspecto dificulta muito, até mesmo, eventualmente, impedindo que um código adaptativo possa ser executado em memória, diretamente sob controle do sistema operacional.

Dentro das limitações de espaço disponível neste trabalho, procuramos apresentar as características mais relevantes dessa linguagem, que passamos a descrever.

a) A implementação da adaptatividade nos programas desenvolvidos nesta linguagem é feita mediante o acréscimo, por extensão sintática, de uma camada adaptativa a um código subjacente não-adaptativo, desenvolvido em alguma linguagem hospedeira disponível e compatível.

b) A linguagem resultante deverá ter sua própria característica, ditada essencialmente pelas extensões acrescentadas à linguagem hospedeira, e que conferem a adaptatividade ao código resultante, mas os programas automodificáveis nela desenvolvidos conterão, como elementos de composição, um conjunto de trechos estáticos, escritos puramente na linguagem hospedeira.

c) Cada trecho estático deve ser escrito de tal forma que seu código esteja em conformidade ao que está especificado em [1]: uma só entrada, uma só saída. Neste artigo, por simplicidade, estão omitidos todos os detalhes da linguagem hospedeira, pois o foco são apenas os recursos sintáticos que se referem à definição dos mecanismos adaptativos no código.

d) A interligação entre os trechos estáticos deve estar, também, em conformidade com [1] para garantir a coerência dos grafos que representam os programas assim construídos. Para isso, convencionou-se, na linguagem proposta, que a declaração dessas interligações se faça sempre a partir da saída de trechos estáticos do programa, e que seja indicada a condição.

e) Convencionou-se, também, que cada trecho estático seja responsável pelo cálculo de um código numérico, que represente univocamente a condição segundo a qual, após a execução do trecho estático corrente, deva ocorrer a ativação de um outro trecho estático do programa.

f) A associação de ações adaptativas às conexões existentes entre os trechos estáticos é que proporciona a adaptatividade aos programas assim desenvolvidos, e isso deve ser feito também de acordo com o especificado em [1].

g) Logo, na camada adaptativa de uma linguagem que deva proporcionar esses recursos ao programador, devem estar

presentes, no mínimo, os elementos sintáticos que permitam:

- Delimitar trechos de código, e identificar tais trechos, estática e dinamicamente, de modo que possam ser referenciados em outra parte do programa quando necessário. Exemplos: declaração de nome, início e final de um trecho de programa.
- Estabelecer conexões entre trechos iniciais do programa, previamente delimitados e identificados, com a finalidade de montar, a partir desses trechos, um grafo inicial subjacente de fluxo para o programa.
- Associar a cada conexão existente, se necessário, algum tipo de atividade de automodificação para o programa através de chamadas de funções adaptativas. Na linguagem aqui proposta, tratam-se de chamadas de funções adaptativas anteriores apenas, não estando disponíveis funções adaptativas posteriores. Os principais casos de atividades de automodificação contemplados nesta proposta são: indicação do trecho a ser processado a seguir, acompanhado da condição de saída do trecho identificado anterior que causa a ativação da presente conexão; indicação da aplicação de funções adaptativas sobre o programa.
- Mecanismos sintáticos para a declaração de funções adaptativas e suas chamadas.
- Mecanismos sintáticos de acesso à ativação de operações adaptativas elementares, tais como consultas, inclusões e exclusões: de trechos, de conexões entre trechos, e da associação entre conexões e chamadas de funções adaptativas.
- Mecanismos sintáticos para a criação dinâmica de novos trechos delimitados e identificados de código, conexões e associações de chamadas de funções adaptativas.
- Mecanismos sintáticos que permitam a especificação da geração de novos identificadores inéditos para trechos recém-criados, sempre que necessário.
- Mecanismos sintáticos de consulta e de memorização de resultados de buscas.

h) A maior independência possível da linguagem proposta em relação à linguagem hospedeira.

Será apresentada, a seguir, no Quadro I, a gramática simplificada da linguagem proposta neste trabalho, a qual foi denominada de Basic Adaptive Language – BADAL. No entanto, trata-se de uma gramática ainda incompleta, cuja linguagem encontra-se em evolução. Dessa forma, nem todas as características acima enunciadas estão presentes na linguagem, como por exemplo, a criação dinâmica de trechos de programas delimitados e identificados.

QUADRO I
GRAMÁTICA SIMPLIFICADA DA LINGUAGEM BADAL

```
programa-adaptativo = "{{" "Start" "at" nome "where"
                          declarações-adaptativas "}}" .

declarações-adaptativas =
          decl-trechos-identificados ";"
          decl-conexões-adaptativas ";"
          decl-funções-adaptativas.

decl-trechos-identificados =
          { decl-trecho-identificado ";" } .

decl-trecho-identificado =
          "Code" nome "is" ":"
          "{" chamada-de-procedimento-hospedeira "}" .

chamada-de-procedimento-hospedeira =
     << chamada de subrotina ou similar na
         linguagem hospedeira >> .

decl-conexões-adaptativas = { conexão-adaptativa ";" }
.

conexão-adaptativa =
     "Conect" "output" "(" nome ")" ":"
                              "//" {   número ":" Link nome
"(" "to" nome ")"
                                                    ["," "Call"
nome
                    "(" [ nome { "," nome } ] ")" ] ";"
                }
                                                  "else" ":"
"Link" nome "(" "to" nome ")"

          ["," "Call" nome
                    "(" [ nome { "," nome } ] ")" ] ";"
                              "//" .
decl-funções-adaptativas =
        { decl-função-adaptativa ";" } .

decl-função-adaptativa =
     "Adapt" nome "(" [ nome { "," nome } ] ")" ";"
                                   "Variable" nome { ","
nome } ";" }
                                   "Generator" nome {
"," nome } ";" }
     "is" ":"
                                   [ "Call" nome
                    "(" [ nome { "," nome } ] ")" ";" ]
                                   "//" {pesquisas}
{remoções} {inserções} "//"

pesquisas =
   { "Search" "Link" nome "from" nome "to" nome ";" } .

remoções  =
   { "Remove" "Link" nome ";" } .

inserções =
   { "Add" (
        "[" "Code" nome "is" ":"
```

Com o fim de demonstrar o uso da linguagem ora proposta, considerou-se um simulador de pilha, apresentado por [24], o qual foi descrito como se segue:

Produções iniciais:

$( 1 , "\beta" ) : \rightarrow 4$

$( 2 , "\beta" ) : \rightarrow 3$

$( 3 , ")" ) : \rightarrow 4$

$( 1 , "(" ) : \rightarrow 2 , \mathcal{A} ( 2 , 3 , 1 )$

Função adaptativa

$\mathcal{A} ( i , j , n ) = \{ k^* , m^* :$

$+ [ ( k , "\beta" ) : \rightarrow m ]$

$+ [ ( m ,$

")" ) : → j ]

+ [ ( i , "(" ) : → k , $\mathcal{A}$ ( k , m , i ) ]

- [ ( n , "(" ) : → i , $\mathcal{A}$ ( i , j , n ) ]

+ [ ( n , "(" ) : → i ] }

A representação gráfica desse simulador de pilha consta da Fig. 1.

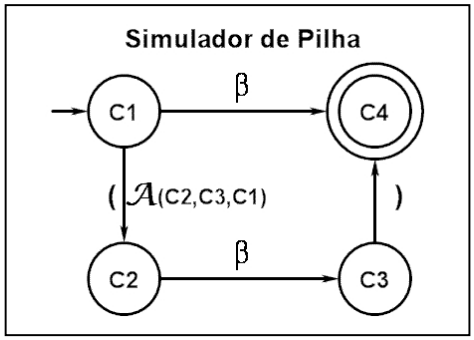


Fig. 1 Simulador de Pilha

A função adaptativa $\mathcal{A}$, realiza as seguintes ações adaptativas:

- Ao ser lido "(", na cadeia de entrada, a transição associada a $\mathcal{A}$ é removida e outra transição sem a função adaptativa associada é adicionada;
- Mais dois estados são criados;
- Três transições são adicionadas: uma associada à função adaptativa $\mathcal{A}$, outra relativa ao símbolo "β" e mais outra correspondente ao símbolo ")".

Consta da Fig. 2. uma ilustração da configuração assumida pelo autômato após ter sido lido o símbolo "(" na cadeia de entrada e a função adaptativa ter sido executada.

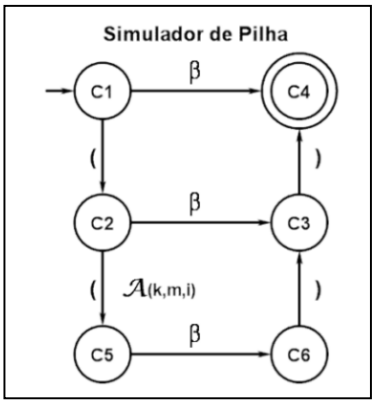


Fig. 2. Simulador de pilha após a execução da função $\mathcal{A}$

Um exemplo de um programa adaptativo, escrito na linguagem BADAL, que representa o simulador de pilha acima é apresentado no Quadro II. C1 é o nome do programa.

QUADRO II
EXEMPLO DE PROGRAMA ADAPTATIVO

```
% programa-adaptativo
% C1 é o nome do trecho do código em que o programa
% tem início

{{ Start at C1 where
```

QUADRO II (CONTINUAÇÃO)

```
% trechos identificados de código
                Code C0 is : { erro ( ) } ;
                Code C1 is : { } ;
                Code C2 is : { } ;
                Code C3 is : { } ;
                Code C4 is : { stop ( ) } ;
% conexões entre os trechos identificados
% (incluindo as adaptativas)

% códigos de saída convencionados:
%               1 = "(" ,    2 = beta ,    3 = ")"

                Conect output (C1) :
                        // 1     : Link L12 ( to C2 ) ,
                Call FA ( C2, C3, C1 ) ;
                         2       : Link L14 ( to C4 ) ;
                           Else : Link L10 ( to C0 ) ;
//
                Conect output (C2) :
                        // 2     : Link L23 ( to C3 ) ;
                           Else : Link L20 ( to C0 ) ;
//
                Conect output (C3) :
                        // 3     : Link L34 ( to C4 ) ;
                           Else : Link L30 ( to C0 ) ;
//
% funções-adaptativas

                  Adapt FA ( PI, PJ, PN ) ;
              Variable lv;
              Generator lk, lm, lkm, k, m;

         is :     //       % remove o Link que usa "(" e
chama FA
                                  Search Link lv from PN
to PI ;
                                  Remove Link lv ;

                % restaura o Link removido,
     % eliminando a chamada de FA
                                  Add [ 1 : Link lv ( to
PI ) From PN ] ;

                                  % cria dois novos
códigos - k e m
                                  Add [ Code k is : { }
] ;
                                  Add [ Code m is : { }
] ;

                                  % cria novo Link lk,
de PI para k,
          % com chamada de FA
                                  Add [ Case 1 : Link lk
```

## IV. CONSIDERAÇÕES FINAIS

Tem sido observado nas publicações científicas um interesse pela retomada do uso da automodificação de código para a solução de problemas de diversas naturezas [4], [23].

Esses trabalhos apresentam código automodificável desenvolvido em linguagem de baixo nível [4], [5], ou por extensões a uma linguagem de alto nível [20].

Ao mesmo tempo, despontam, também, publicações sobre os fundamentos de uma linguagem de alto nível que permita a programação de código capaz de se automodificar [1], [20]. Buscando contribuir com essa perspectiva, este artigo propõe uma linguagem básica, que permite desenvolver programas em alto nível com características adaptativas.

A linguagem ainda está em desenvolvimento. Entretanto, após a conclusão do trabalho, espera-se disponibilizar uma

linguagem através da qual seja possível programar códigos adaptativos.

Espera-se, ainda, que o presente trabalho seja motivador de novos estudos, nos quais se possa avaliar os custos computacionais de tempo e de espaço ocupado por repetidas adições de código, durante a execução, que não foram contemplados neste estudo.

## V. REFERÊNCIAS


[1] É. J. Pelegrini, "Códigos Adaptativos e linguagem para programação adaptativa: conceitos e tecnologia." Dissertação de Mestrado, apresentada à Escola Politécnica da USP. São Paulo. 2009

[2] C. Hongxu; S. Zhong; V. Alexander. "Certified self-modifying code". SIGPLAN Notices. Vol. 42(6), 2007. pp. 66-77.

[3] B. Anckert; M. Madou, K. D. Bosschere. "A Model for Self-Modifying Code". Lecture Notes in Computer Science, Springer Berlin / Heidelberg, ISSN 0302-9743, Volume 4437/2007, Information Hiding, ISBN 978-3-540-74123-7, Pages 232-248, September 14, 2007.

[4] J. T. Giffin, M. Christodorescu, L. Kruger. "Strenghtening software self-checksumming via self-modifying code". Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005).

[5] Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto. "Exploiting selfmodication mechanism for program protection". In Proc. of the 27th Annual International Computer Software and Applications Conference, pages 170-181, 2003.

[6] J. J. Neto. "Um Levantamento da Evolução da Adaptatividade e da Tecnologia Adaptativa." Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. (p. 496-505).

[7] J. J. Neto. "Adaptive Rule-Driven Devices - General Formulation and Case Study." Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol.2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.

[8] R. S. Bier. "Recursion elimination with variable parameters". The Computer Journal 1979 22(2):151-154; doi:10.1093/comjnl/22.2.151 by British Computer Society. 1979.

[9] Y. A. Lui, S. C. Stoller. "From Recursion to iteration. What are the optimization?" In Proceedings of the ACM SIGPLAN Workshpo on Partial Evaluation and Semantics-Based Programming Manipulation (PEPM '00), pages 73-82. ACM Press, Jan. 2000.

[10] M. Flatt. Composable and Compilable Macros. Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. Pag. 72-83. 2002.

[11] A. B. Tucker e R. E. Noonan. *Linguagens de programação: princípios e paradignmas*. 2ª. ed. São Paulo: Mc Graw Hill. 2009. p. 362-366.

[12] R. W. Sebesta. *Conceitos de Linguagens de Programação*. 5ª. Ed. São Paulo: Bookman, 2002.

[13] P. J. da Rocha, MARQUES, Silva S. "Simulação de um sistema automático de logística interna para a indústria de calçados." Dissertação de mestrado em Automação, Instrumentação e Controle, apresentada à Universidade do Porto, Portugal. 2007.

[14] M. Bravenboer e E. Visser. "Designing Syntax Embeddings and Assimilations for Language Libraries". In *Models in Software Engineering: Workshops and Symposia at MoDELS 2007*, volume 5002 of LNCS, 2008.

[15] W3C. "Extensible Markup Language. (XML)", disponível em http://www.w3.org/XML/, acessado em 16/01/1010.

[16] A. Kamada. "Execução de serviços baseada em regras de negócios." Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação, da Universidade Estadual de Campinas. São Paulo. 2006.

[17] T. Standish. "Extensibility in programming language design". ACM SIGPLAN Notices, Volume 10, Issue 7, Pages: 18–21, ISSN: 0362-1340, 1975.

[18] J. Alghamdi, J. Ilrhaa**. "**Comparing and Assessing Programming Languages: Basis for A Qualitative Methodology". Proceedings of ACM/SIGAPP, pag. 222-229. 1993.

[19] R. Rüdiger. "Extensible programming in Oberon. A tutorial". Technical Report. FH Braunchweig /Wolfenbüttel – University of Applied Science. 1999.

[20] A. V. de Freitas. "Considerações sobre o desenvolvimento de linguagens adaptativas de programação." Tese de Doutorado, apresentada à Escola Politécnica da Universidade de São Paulo. 2008.

[21] A. A. de Castro Júnior. "Aspectos de projeto e implementação de linguagens para codificação de programas adaptativos." Tese de Doutorado apresentada à Escola Politécnica da Universidade de São Paulo. 2009.

[22] J. J. Neto. "Um glossário sobre adaptatividade". III Workshop de Tecnologias adaptativas. Universidade de São Paulo. 2009.

[23] C. Tschudin, L.Yamamoto. "Harnessing Self-Modifying Code for Resilient Software". Proc 2nd IEEE Workshop on Radical Agent Concepts (WRAC), NASA Goddard Space Flight Center Visitor's Center, September 2005. Springer LNCS/LNAI 3825, 2006.

[24] J. J. Neto. "Contribuições à metodologia de construção de compiladores". Tese de Livre Docência, apresentada à Escola Politécnica da Universidade de São Paulo. 1993.


**Salvador Ramos Bernardino da Silva** é graduado em Estatística (1986). Atualmente, é professor do Curso de Engenharia de Computação da Universidade do Estado do Amazonas – UEA e aluno do Curso de Mestrado em Engenharia Elétrica, do Departamento de Engenharia de Computação e Sistemas Digitais – PCS, da EPUSP. Suas áreas de interesse são, principalmente: programação de computadores, teoria da computação, compiladores. Está desenvolvendo pesquisa sobre dispositivos adaptativos, tecnologia adaptativa, autômatos adaptativos, e suas aplicações à Engenharia de Computação, particularmente sobre programação adaptativa.

**João José Neto** é graduado em Engenharia de Eletricidade (1971), mestre em Engenharia Elétrica (1975), doutor em Engenharia Elétrica (1980) e livre-docente (1993) pela Escola Politécnica da Universidade de São Paulo. Atualmente, é professor associado da Escola Politécnica da Universidade de São Paulo e coordena o LTA – Laboratório de Linguagens e Tecnologia Adaptativa do PCS – Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP. Tem experiência na área de Ciência da Computação, com ênfase nos Fundamentos da Engenharia da Computação, atuando principalmente nos seguintes temas: dispositivos adaptativos, tecnologia adaptativa, autômatos adaptativos, e em suas aplicações à Engenharia de Computação, particularmente em sistemas de tomada de decisão adaptativa, análise e processamento de linguagens naturais, construção de compiladores, robótica, ensino assistido por computador, modelagem de sistemas inteligentes, processos de aprendizagem automática e inferências baseadas em tecnologia adaptativa.