Aparecido Valdemir de Freitas

Considerações sobre o Desenvolvimento de Linguagens Adaptativas de Programação

São Paulo
2008

Aparecido Valdemir de Freitas

Considerações sobre o Desenvolvimento de Linguagens Adaptativas de Programação

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Engenharia Elétrica

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr. João José Neto

São Paulo
2008

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 02 de junho de 2008.

FICHA CATALOGRÁFICA

Freitas, Aparecido Valdemir de
Considerações sobre o desenvolvimento de linguagens adaptativas de programação / A.V. de Freitas. -- São Paulo, 2008.
122 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Computação adaptativa 2. Linguagem de programação I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

“Assim diz Deus, o Senhor,
que criou os céus e os estendeu,
formou a terra e a tudo quanto produz;
que dá fôlego de vida
ao povo que nela está
e o espírito aos que andam nela.
Guiarei os cegos por um caminho que não conhecem,
fá-los-ei andar por veredas desconhecidas;
tornarei as trevas em luz perante eles
e os caminhos escabrosos, planos.
Estas coisas lhes farei
e jamais os desampararei.”

Isaías 42:5,16

Dedico este trabalho à Maria Helena, Mariana e Camila.

AGRADECIMENTOS

Agradeço a Deus, por ter me concedido a oportunidade de desenvolver este trabalho.

À minha esposa Maria Helena, pelo amor que sempre me tem dedicado, pelo companheirismo, paciência e apoio.

Às minhas filhas Mariana e Camila, por todos os momentos felizes que diariamente delas recebo.

Ao Prof. Dr. João José Neto, por toda a atenção, ajuda, amizade e orientação dada no desenvolvimento deste trabalho.

Ao Prof. Dr. Paulo Sérgio Muniz Silva e ao Prof. Dr. Ricardo Luis de Azevedo da Rocha pelas valiosas contribuições e sugestões apresentadas na banca de qualificação.

Aos meus pais, pelo esforço empreendido para me proporcionarem a educação que recebi.

Aos meus colegas professores da Universidade Municipal de São Caetano do Sul, Prof. Carlos Eduardo Bognar, Prof. Edson Pinheiro Pimentel e Prof. Luiz Carlos Rozante, pelos momentos de incentivo e pelo feliz convívio nas atividades da Coordenaria de Computação do Imes.

Ao meu amigo Prof. Mário Eugênio Longato pelo exemplo de profissionalismo, pela amizade e pelos votos de confiança e otimismo.

RESUMO

Um dispositivo adaptativo é constituído por um dispositivo subjacente (geralmente não adaptativo), por exemplo, um autômato, uma gramática, etc., no qual se adiciona um mecanismo adaptativo que é responsável pela automodificação autônoma que caracteriza os dispositivos adaptativos. As linguagens adaptativas são instâncias de dispositivos adaptativos, cujo formalismo subjacente é uma linguagem de programação. A tese tem como objetivo conceituar linguagens adaptativas de programação, descrever seus pontos essenciais e considerar os aspectos e particularidades que afetem seu desenvolvimento. A concepção de linguagens adaptativas motiva um novo estilo de programação, uma vez que a aplicação da tecnologia adaptativa sugere uma nova forma de pensar. O estilo adaptativo de programação poderá tornar-se uma alternativa viável para se obter códigos aderentes às aplicações de códigos automodificáveis.

ABSTRACT

Adaptive devices comprehend a subjacent (usually non-adaptive) device, e.g. an automaton, a grammar, etc., to which an adaptive mechanism is added which performs the self-modification operations usual in adaptive devices. Adaptive languages are special adaptive devices whose subjacent formalism is a programming language. This work defines adaptive programming languages, describes their main concepts and discusses their particular development issues. Adaptive programming languages suggest a new programming style, since applying adaptive technology induces a somewhat new way to think about programs. Such adaptive style may become an alternate way to obtain adequate code in self-modifying applications.

LISTA DE ILUSTRAÇÕES

1. Um programa Lisp na forma de árvore 21
2. Extensibilidade em Linguagens de Programação 25
3. Código para otimização 25
4. Código alterado para otimização 26
5. Otimização com automodificação 26
6. Processamento de Linguagem Básica (sem extensão e sem adaptatividade) . . 32
7. Processamento de Linguagem Extensível e sem Adaptatividade 32
8. Geração de Programas com Adaptatividade a partir de Ling. Extensível 33
9. Programas Adaptativos gerados a partir de Ling. Genuin. Adaptativas. . 33
10. Identificação de nós de árvore 36
11. O mesmo programa da Figura 11, em sua representação de árvore 37
12. Adaptação da Árvore – Ação de Remoção 38
13. Adaptação da Árvore após Ação de Remoção 39
14. Adaptação da Árvore – Ação de Remoção de Sub-árvore e marca 40
15. Adaptação da Árvore – Ação de Inserção de Sub-árvore 40
16. Operação Conceitual de uma Linguagem Adaptativa 41
17. Classificação de Programas Adaptativos 44
18. Geração de Programa na Linguagem-Base (Não-adaptativa) 44
19. Geração de Programa Não-adaptativo por Extensão 45
20. Geração de Programa Adaptativo 45
21. Árvore representando o Programa Lisp (a (b d e) (c f g)) 47
22. Identificação de nó para Adaptatividade no programa (a (b d e) (c f g)) . . 47
23. Árvore com nós marcados pela primitiva mark 48
24. Árvore com sub-árvore a ser removida 50
25. Árvore resultante da aplicação da primitiva rem 50
26. Árvore resultante da aplicação da primitiva rem-tree 51
27. Árvore resultante da aplicação da primitiva rem-mark 52
28. Árvore com nós referenciados pela primitiva ins 53
29. Árvore resultante após a operação da primitiva ins 54
30. Árvore com nó referenciado pela primitiva upd 55
31. Árvore resultante após a operação da primitiva upd 55

32. Exemplo de um programa adaptativo 57
33. Exemplo de uma Função Adaptativa com ação elementar de update . . . 57
34. Árvore do Programa antes da execução da Função Adaptativa 58
35. Árvore do Programa após a execução da Função Adaptativa 58
36. Conjunto das 28 peças do Dominó Adaptativo 59
37. Uma possível distribução das peças entre os quatro jogadores 60
38. Uma possível distribução das peças entre os quatro jogadores 61
39. Configuração após o lance-1 62
40. Configuração após o lance-2 63
41. Configuração após o lance-3 64
42. Configuração após o lance-4 65
43. Configuração após o lance-5 66
44. Configuração após o lance-6 67
45. Configuração após o lance-7 68
46. Configuração após o lance-8 69
47. Dominó Adaptativo – Pseudocódigo 69
48. Estrutura do conjunto de regras sem considerar adaptatividade 72
49. Estrutura do conj. de regras considerando primitivas de adaptatividade . 73
50. Ambiente de Execução de Programas Adaptativos 75
51. Diretrizes p/Desenv. de Linguagens Adaptativas c/Extensibilidade. . . . 77
52. Diretrizes p/Desenvolvimento de Linguagens Nativamente Adaptativas .78
53. Autômato Adaptativo para reconhecer cadeias expressas por $a^n b^n c^n$ . . . 92

SUMÁRIO

1. INTRODUÇÃO 13

1.1 Motivação 14
1.2 Objetivo 16
1.3 Organização da Tese 17

2. CONCEITOS 18

2.1 Dispositivos guiados por regras. 19
2.2 Linguagens Imperativas e Funcionais 22
2.3 Linguagens Extensíveis 23
2.4 Automodificação de código 25
2.5 Linguagens Adaptativas 30

3. CONCEPÇÃO DE LINGUAGENS ADAPTATIVAS 34

3.1 O Dispositivo Subjacente. 34
3.2 Adaptatividade em Linguagens de Programação 35
3.3 As Funções Adaptativas 38
3.4 Operação do Programa Adaptativo 41
3.5 A Camada Adaptativa 42
3.6 Programas Adaptativos 43

4. IMPLEMENTAÇÃO DA CAMADA ADAPTATIVA 46

4.1 Endereçamento de Porções Modificáveis de Código 46
4.2 Primitiva para remoção de código 49
4.3 Primitiva para inserção de código 52
4.4 Primitiva para atualização de código 54
4.5 Primitiva para consulta 56
4.6 Primitiva para geração automática de nomes de marcas 56
4.7 Exemplo ilustrativo das primitivas da camada adaptativa 57

5. EXEMPLO DE APLICAÇÃO 59

5.1 Um jogo de Dominó Adaptativo 59

5.2 Simulação de uma partida 61
5.3 Observações sobre estilos de implementação do jogo 70

6. CONSIDERAÇÕES FINAIS 74

6.1 Diretrizes para o desenvimento de Linguagens Adaptativas 74
6.2 Contribuições 78
6.3 Trabalhos Futuros 79
6.4 Conclusão 80

7. REFERÊNCIAS 82

8. BIBLIOGRAFIA COMPLEMENTAR 88

APÊNDICES 92

A Autômatos Adaptativos 92
B Códigos Automodificáveis 101
C Dominó Adaptativo – Implementação Adaptativa 104

GLOSSÁRIO 116

1. Introdução

A adaptatividade considera o uso de dispositivos que apresentam a característica de se modificarem dinamicamente em resposta a estímulos de entrada, sem interferência de agentes externos. Estas propriedades de automodificação, características dos dispositivos adaptativos, lhes dão naturalidade para expressarem e manusearem diversos aspectos de sistemas automodificáveis. Dispositivos adaptativos devem, portanto, ser capazes de detectar possíveis necessidades de modificação de comportamento para, em seguida, reagirem a elas executando as correspondentes modificações (NETO,2001).

A idéia principal que permeia a adaptatividade reside na habilidade que um dispositivo tem de executar operações de automodificação, denominadas ações adaptativas, as quais podem ser vistas como atividades internas ao dispositivo e que são executadas em resposta ou reação à detecção das situações que exigem alterações comportamentais.

Historicamente, dispositivos adaptativos foram inicialmente estudados e aplicados na área de construção de compiladores para a obtenção de mecanismos puramente sintáticos destinados a ampliar a capacidade de expressão dos autômatos de pilha estruturados (NETO,1993).

Um dispositivo adaptativo é constituído por um dispositivo subjacente (geralmente não adaptativo), por exemplo, um autômato, uma gramática, uma árvore de decisão, etc, ao qual se acrescenta um mecanismo adaptativo, responsável por permitir que a estrutura do dispositivo subjacente seja dinamicamente modificada. Por exemplo, ao se acrescentar um mecanismo adaptativo a um autômato de estados finitos, este adquire a possibilidade de efetuar inclusões ou remoções de transições de estados, durante o processamento da cadeia de entrada, o que aumenta seu poder de expressão.

A adaptatividade tem, portanto, como aplicação básica estender formalismos consolidados, aumentando seu poder de expressão (PISTORI,2003).

Posteriormente aos autômatos, outros dispositivos adaptativos foram concebidos a partir de diferentes dispositivos subjacentes, como por exemplo, statecharts (SANTOS,1997,RADY,1995), redes de Markov (BASSETO;NETO,1999), gramáticas (IWAI,2000), tabelas de decisão (NETO,2001), árvores de decisão (PISTORI,2003), etc. Aplicações surgiram em diversas áreas, tais como: aprendizagem computacional (ROCHA;NETO,2000), processamento de linguagens naturais (MENEZES, 2000), ambientes multilinguagens (FREITAS,2000), robótica (NETO; SOUZA; HIRAKAWA, 2004), etc.

Dentre os dispositivos adaptativos já estudados, destacam-se:

- Dispositivos de reconhecimento, da classe dos autômatos, baseado na sucessão de mudanças de estado;
- Dispositivos para a representação de sistemas assíncronos, tais como statecharts, que incorporam mecanismos responsáveis pela representação dos fenômenos de sincronização;
- Dispositivos estocásticos, como as redes de Markov, capazes de representar fenômenos de caráter aleatório;
- Dispositivos de auxílio à tomada de decisões, representados principalmente pelas tabelas de decisão e pelas árvores de decisão.

Todas essas contribuições incorporaram para a adaptatividade um conjunto de poderosos recursos, de natureza não apenas conceitual e teórica, mas também de caráter aplicativo, na forma de métodos, técnicas e ferramentas, que foram sendo desenvolvidos ao longo dessa história (NETO,2007).

1.1 Motivação

Anteriores ao surgimento de trabalhos explicitamente relacionados à adaptatividade (NETO,2007), pode-se considerar que a extensibilidade de linguagens de

programação seja uma das mais importantes (embora rudimentares) manifestações do fenômeno da adaptatividade em linguagens de programação.

O poder de atuação da extensibilidade se limita, entretanto, às componentes da linguagem de programação que podem ser tratatadas em tempo de compilação, e se refere, portanto, às alterações que o programador pode efetuar sobre a linguagem, de forma que possa adequá-la aos particulares propósitos dos seus programas.

Dessa forma, cada usuário pode construir sua própria versão da linguagem, que lhe seja especificamente personalizada, podendo então exprimir seus programas de forma confortável, e na notação mais aderente à sua própria aplicação.

Entretanto, isso ainda não atende aqueles usuários que necessitem que seus programas possam exprimir atividades, a serem executadas em tempo de execução, de alteração na estrutura de seus próprios algoritmos.

Para isso, tais programas necessitariam incorporar recursos de adaptatividade e, portanto, deveriam dispor de meios para, sem intervenção externa, determinar e realizar dinamicamente as automodificações estruturais necessárias.

Linguagens adaptativas podem ser vistas como dispositivos adaptativos, cujo formalismo subjacente é uma linguagem de programação. Sendo possível nessas linguagens a aplicação dinâmica de ações adaptativas, torna-se natural expressar, com relativa facilidade, fenômenos de automodificação nos programas nelas codificados.

A principal motivação da presente tese está associada ao levantamento dos requisitos para o desenvolvimento de uma linguagem de programação que atenda a tais requisitos, e para isso pode valer-se das propriedades dos formalismos adaptativos, que lhes são aderentes, como conseqüência. Disso podem resultar linguagens adaptativas de programação. Neste trabalho é mostrada a viabilidade de implementação de tais linguagens adaptativas.

Programas automodificáveis codificados usando linguagens adaptativas tendem a ser mais robustos, visto que já na época de sua compilação desses programas, é possível fazer algumas verificações com a finalidade de garantir que as automodificações

efetuadas sejam relativamente seguras, evitando problemas decorrentes de uma excessiva liberdade nas operações de automodificação presentes nos programas.

Isso pode ser conseguido forçando os seus usuários a seguirem uma disciplina, devidamente estabelecida, e respeitando algum método que os oriente a bem utilizar os recursos adaptativos de programação.

Nesta tese, adota-se, como formalismo subjacente para a linguagem adaptativa proposta, um núcleo básico funcional, representado pela linguagem Lisp. Sobre ele, é construída uma camada adaptativa, que habilita o programador a codificar programas com características de automodificação.

1.2 Objetivo

Esta tese tem como objetivo geral efetuar um levantamento de requisitos, em relação a diversos aspectos de projeto e implementação de linguagens adaptativas.

Para o atendimento deste objetivo geral, os seguintes objetivos específicos foram considerados:

- Estudo da viabilidade do emprego de uma linguagem de programação como dispositivo subjacente de um dispositivo adaptativo;
- Definição dos pontos essenciais que uma linguagem de programação necessita ter para ser considerada uma linguagem adaptativa de programação;
- Definição das extensões (funcionais) que devem ser incorporadas a uma linguagem de programação para que ela apresente características de automodificação;
- Consideração dos aspectos e particularidades que afetem o desenvolvimento de uma linguagem adaptativa de programação;
- Emprego de uma linguagem adaptativa à resolução de uma aplicação com características adaptativas;
- Implementação de uma linguagem adaptativa completa, obtida por meio da extensão funcional de uma linguagem hospedeira.

1.3 Organização da Tese

No capítulo 1 é apresentada a motivação e os objetivos da tese, bem como uma visão geral da organização da mesma.

No capítulo 2 apresenta-se a revisão bibliográfica que dará o embasamento conceitual para o desenvolvimento da tese. São discutidos os tópicos relacionados à tecnologia adaptativa e ao estilo adaptativo de programação.

No capítulo 3 apresenta-se a conceituação da linguagem adaptativa proposta, tecendo-se observações relativas às suas características básicas, bem como do esquema adotado para que a linguagem proposta apresente comportamento automodificável.

No capítulo 4 apresenta-se o projeto da linguagem adaptativa proposta, com os detalhes da definição da camada adaptativa e do núcleo funcional básico, bem como do procedimento adotado para o endereçamento dos elementos do programa. São apresentados os aspectos de implementação da camada a serem incorporados à linguagem hospedeira (em nosso caso funcional) empregada como dispositivo subjacente da linguagem adaptativa proposta.

No capítulo 5 apresentam-se algumas observações relativas à aplicação-exemplo realizada.

No capítulo 6 apresentam-se as considerações finais da tese, suas contribuições e sugestões de trabalhos futuros.

Por fim, no capítulo 7 apresentam-se as referências bibliográficas empregadas como subsídio para o desenvolvimento da tese.

2. Conceitos

Este capítulo apresenta uma pequena revisão conceitual sobre linguagens de programação e sua relação com adaptatividade.

Dentre as várias maneiras de se exteriorizar uma forma de pensar, podemos citar o efeito da linguagem, das atitudes, da mímica, do comportamento cultural, da vestimenta, etc. Dentre estas, talvez a linguagem desempenhe uma das mais importantes formas de se expressar um pensamento (FREITAS;NETO,2000).

Lingüistas têm enfocado que a linguagem na qual um pensamento é expresso reflete de forma fundamental a natureza de uma idéia (BUDD,1995). Tomada como uma atividade cognitiva, a linguagem tem a função de refletir os processos mentais humanos. Daí a conexão estreita entre pensamento e linguagem (BORBA,1987).

A associação entre pensamento e linguagem ainda se torna mais crítica se estendermos o conceito para o domínio das linguagens de programação. Assim, a linguagem utilizada por um programador na resolução de um problema pode estar fortemente relacionada à sua maneira de pensar, ou à sua forma de implementar a sua solução (BUDD,1995).

Uma linguagem de programação pode vista como uma notação para a descrição de um algoritmo (TENNENT,1981).

Ao programar em uma dada linguagem, pode parecer trivial que um programa equivalente possa escrito em outra linguagem. Definida a lógica de um programa, a teoria nos garante que tal lógica pode ser expressa indiferentemente em qualquer máquina universal, por exemplo, Máquina de Turing, Cálculo λ, funções μ-recursivas e outras (LEWIS,1981).

A escolha da linguagem a ser empregada na codificação de um programa está, usualmente, associada à familiaridade do programador e à facilidade de uso. Embora não se questione a potencialidade de execução de tarefas computacionais de linguagens de montagem e da Máquina de Turing, devido ao seu baixo nível de abstração elas se mostram pouco confortáveis para utilização e depuração de um programa, dificultando a programação e trazendo, como conseqüência, baixa produtividade.

Por outro lado, linguagens pertencentes a paradigmas aderentes às aplicações se mostram mais adequadas à codificação de programas voltados a essas aplicações.

Programadores que atuam há muitos anos com uma determinada linguagem de programação, costumam ter o correspondente paradigma arraigado à sua forma de pensar, e geralmente modelam suas soluções a partir dos recursos oferecidos pelas linguagens que utilizam. Tais programadores, quase sempre, têm dificuldades em quebrar o paradigma a que estão acostumados em favor de outros paradigmas de programação.

A operação de um computador (baseado no modelo de Von Neumann) corresponde à execução seqüencial de instruções, presentes em algum meio de armazenamento. Esse modelo de computação tem influenciado o desenvolvimento das linguagens de programação, condicionando em geral a visão de um programa a uma seqüência de instruções convencionais de máquina, armazenadas em posições sucessivas de memória (HARRISON,1988).

As linguagens de máquina, em geral, oferecem acesso direto a todos os recursos do hardware. Conseqüentemente, programas escritos nestas linguagens não costumam ser diretamente transportáveis para outras máquinas e, em geral, torna-se difícil ler ou escrever programas em tais linguagens (TENENNT,1981).

Assim, linguagens de máquina foram sendo, gradualmente, substituídas por linguagens de alto nível, que costumam ser classificadas de acordo com os respectivos paradigmas de programação (HUDAK,1989).

2.1 Dispositivos guiados por regras

O conceito da adaptatividade (NETO,2007) pode ser aplicado a qualquer dispositivo cuja operação seja definida por um conjunto de regras. Na construção de tais dispositivos, conceitualmente é possível separar de forma clara dois importantes

componentes: um dispositivo subjacente, tipicamente não-adaptativo, e um mecanismo adaptativo, responsável pela incorporação da adaptatividade (NETO,2007).

Nessa formulação, a característica principal de um dispositivo adaptativo é sua capacidade de realizar ações adaptativas, que se traduzem na execução de operações de automodificação ao serem identificadas determinadas situações específicas (NETO, 2001).

Da aplicação da adaptatividade a um sistema, torna-se possível que, em função dos estímulos recebidos, instâncias idênticas de uma mesma configuração evoluam para comportamentos finais totalmente diferentes, uma vez que esta capacidade de automodificação está associada à diversidade de eventos a que o sistema adaptativo é submetido durante sua respectiva utilização.

Partindo de uma configuração inicial, estando em uma dada configuração, e recebendo um determinado estímulo, o dispositivo, através da aplicação de alguma das regras que definem seu comportamento, assume uma nova configuração, de acordo com a regra adequada àquela situação.

O dispositivo é dito determinístico se e somente se, para qualquer configuração dada e qualquer estímulo de entrada, seu conjunto de regras determina uma, e somente uma, próxima configuração. No caso não-determinístico, havendo mais de uma regra aderente à situação corrente do dispositivo, as diversas possíveis alternativas são tratadas em paralelo [CASTRO;NETO;PISTORI,2007].

Neste trabalho o dispositivo subjacente é um programa Lisp, que pode ser formalizado (NETO,2001) da seguinte maneira:

<dispositivo>	→	<programa-Lisp-completo>
<programa-Lisp-completo>	→	<lista-de-um-nível>
<lista-de-um-nível>	→	(identificação-do-nó <seqüência>)
<seqüência>	→	ε
	→	<lista-de-um-nível>
	→	<seqüência> <lista-de-um-nível>

Dado um programa Lisp com tal estrutura, pode-se representá-lo como uma seqüência de regras:

1ª regra: Lista de um nível que representa o programa completo (representando toda a árvore).

Outras Regras: Listas de um nível que compõem a seqüência que define a lista de um nível de hierarquia imediatamente superior (representando alguma sub-árvore do programa)

Por exemplo, consideremos um programa Lisp que é representado pela árvore da Figura 1.

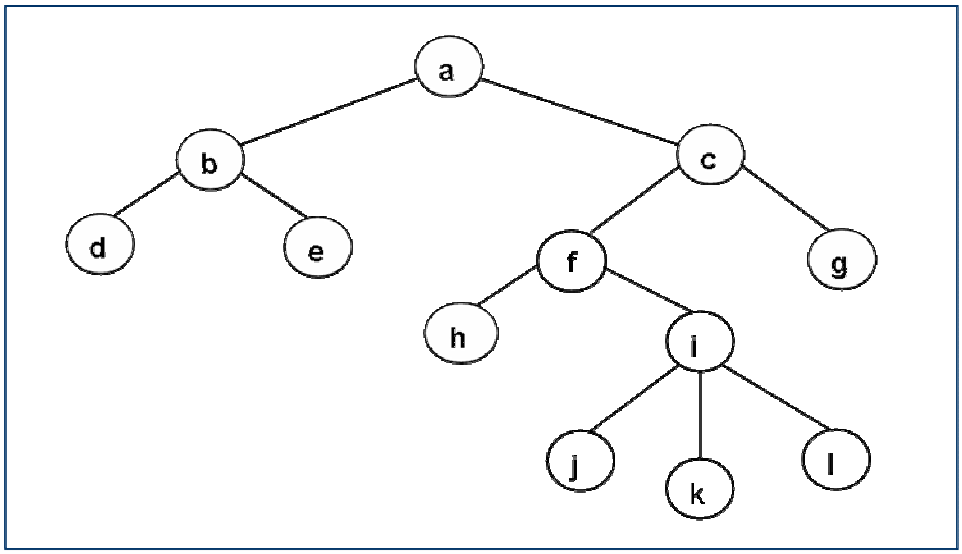


Figura 1 – Um programa Lisp na forma de árvore

Para este programa, teremos o seguinte conjunto de regras:

{ (a,b,c), (b,d,e), (b,d), (b,e), (d), (e), (c,f,g), (c.f), (c,g), (f,h,i), (f,h), (f,i), (g), (h), (i,j,k,l), (i,j), (i,k), (i,l), (j),(k),(l) }

Para o caso de programas escritos em linguagens imperativas, usualmente o programa apresenta áreas de dados e trechos seqüenciais de código acrescidos de estruturas de blocos hierárquicos e estruturas de controle, destinadas à alteração do fluxo (quebra de sequência). Nesta situação, a representação do programa pode ser formalizada como uma seqüência de elementos (que representam instruções executáveis). As estruturas de controle e as de bloco podem ser representadas de forma análoga à apresentada acima para programas Lisp. Os trechos correspondentes a áreas de dados e as partes seqüenciais do programa poderiam ser representadas como se fossem listas de um nível, cuja seqüência corresponderia aos elementos de dados ou às instruções básicas da linguagem. Este assunto não será detalhado aqui pois não está no escopo deste trabalho. Há outros trabalhos em andamento no LTA - Laboratório de Linguagens e Técnicas Adaptativas da Escola Politécnica da USP.

2.2 Linguagens Imperativas e Funcionais

Linguagens imperativas de programação têm foco no conceito de estado (modelado por variáveis) e ações (comandos) que efetivam alterações nestes estados. O vocábulo *imperativo* é de origem latina (*imperare*) o qual significa comandar, ordenar. Em função do estreito relacionamento com arquiteturas de máquina, pelo menos em princípio, as linguagens imperativas podem ser implementadas de forma eficiente em máquinas com arquitetura de Von Neumann (WATT,1990).

Nesse modelo, o computador opera como um manipulador de dados, seguindo um padrão de instruções determinado pelo código de seus programas. Cada instrução contida neste código pode extrair conteúdos (valores) da memória e transformá-los, podendo em seguida armazenar os resultados em localizações especificadas de memória. Quando do término da computação, os valores que restarem na memória representam o estado final do programa, o qual, de uma certa forma, representa a solução desejada (GHEZZI; JAZAYERI,1998).

Tendo em vista que a programação imperativa tem como base a mudança de estados dos programas, podemos verificar que a simples leitura do texto estático que descreve um programa é em si insuficiente para que se possa perceber exatamente os fenônemos que ocorrem durante a execução do mesmo (SETHI,1996).

Embora, na maioria das vezes, as ações executadas por um programa atuem sobre os conteúdos de sua área de dados, pode, às vezes, ser permitido ao programa que mesmo sua área de código seja modificada por força de execução de suas próprias instruções. Neste caso, caracterizam-se programas automodificáveis.

Nesta tese, discutem-se as linguagens adaptativas de programação, que permitem uma alteração organizada do próprio código dos programas escritos em tais linguagens, que assim se comportam como programas automodificáveis.

Em linguagens funcionais (TENNENT,1981), a computação se faz unicamente por meio da avaliação de funções. Em contraste com as linguagens imperativas, inexiste um estado implícito, baseando-se sua operação na computação por aplicação sucessiva de funções recursivas (HUDAK,1998).

Na programação funcional, o mapeamento dos dados de entrada em valores de saída é obtido de forma direta. O programa se reduz a uma função (ou uma composição de funções) que, tipicamente, referencia funções mais simples, e funções primitivas da linguagem. Os valores produzidos pelas funções se obtêm por recursão, em lugar da iteração utilizada preferencialmente por programas imperativos.

Uma conseqüência dessa maneira de proceder é o fato de que os programas funcionais se tornam atemporais. Assim, a restrição de que uma função somente pode ser aplicada quando os valores de seus argumentos estiverem disponíveis, faz com que a ordem da execução das funções não influa no seu resultado final (BUDD,1995).

Dentre as diversas linguagens funcionais de programação existentes, a linguagem *Lisp* (McCARTHY,1959) tem uma importância significativa uma vez que foi a pioneira dentre as funcionais de programação. O seu nome é derivado de *LISt Processing*, tendo tanto dados quanto código representados por listas, o que permite que a linguagem manipule o próprio código dos seus programas.

2.3 Linguagens Extensíveis

Historicamente, os primeiros passos no desenvolvimento de linguagens com capacidade de extensão foram dados ao surgir o conceito de macros.

Dentre as diversas formas existentes de se exercitar a extensibilidade em linguagens de programação, podemos destacar o exemplo clássico das macro-instruções, no caso em que a linguagem-base é uma linguagem de montagem.

Define-se tipicamente uma macro-instrução através de um agrupamento de instruções, ao qual se atribui um nome e se associam argumentos, nome este que pode ser utilizado posteriormente em uma chamada da macro, na forma usualmente utilizada na linguagem de montagem.

Este conceito de macro evoluiu posteriormente para incorporar alterações na sintaxe, através de macros sintáticas. Mais tarde, a extensibilidade teve um declínio, restando atualmente nas linguagens modernas de programação apenas extensões restritas (definições de novos tipos de dados e extensões funcionais) (SEBESTA,2007).

Intuitivamente podemos conceituar extensibilidade, de modo amplo, como sendo a capacidade que um sistema – em particular, uma linguagem de programação - tem de se expandir por meio da adição de novos componentes (no caso, novas instruções).

Assim, a extensibilidade de uma linguagem de programação se refere à sua capacidade de incorporar incrementalmente novas funcionalidades, novos recursos, e até nova sintaxe, por iniciativa do programador, o qual deve fornecer as definições desejadas, para as extensões, expressas como uma combinação de outras, então disponíveis (NETO,2007).

Com a extensibilidade, torna-se possível ao programador modificar, enquanto codifica seu programa, os recursos da sua linguagem de programação, adequando-a a seus propósitos particulares (NETO,2007).

Partindo-se de uma linguagem-base, e com o uso de mecanismos para a definição de novos recursos, o programador de uma linguagem extensível pode criar novas notações, novos tipos de dados, novas operações, novas estruturas de controle (STANDISH,1975).

A forma mais comumente empregada para se promover a extensibilidade de linguagens de programação ocorre através de paráfrase, através da qual peculiaridades de linguagens são definidas a partir de outras características já existentes na mesma linguagem (ZINGARO,2007).

Outra ocorrência do fenômeno da extensibilidade de linguagens pode ser encontrado em muitas implementações de linguagens atuais, e costumam ser localizadas em pré-processadores das mesmas, os quais incluem mecanismos para a definição de novas macros, a serem referenciadas no corpo do texto. Através do pré-processamento, essas definições são memorizadas, e depois, expandidas em comandos da linguagem-base, quando do tratamento de suas chamadas, de modo que, uma vez expandidas, possam ser convenientemente tratadas pelo compilador já existente da linguagem-base.

A Figura 2 apresenta uma classificação geral das linguagens de programação no tocante à extensibilidade.

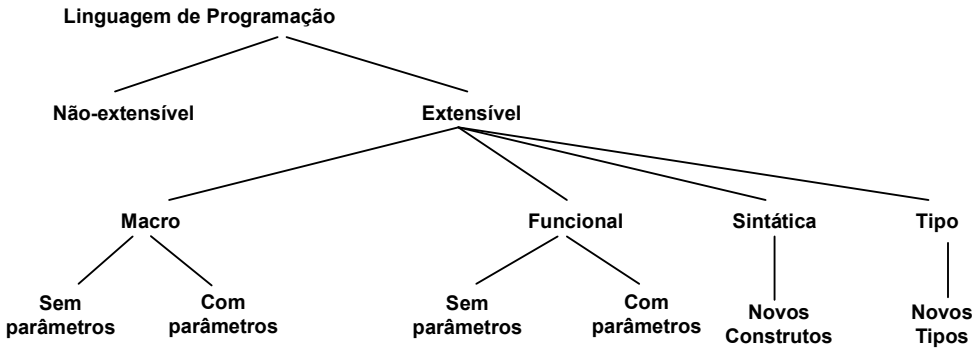


Figura 2 - Extensibilidade em Linguagens de Programação

2.4 Automodificação de Código

Programas automodificáveis foram usados de forma expressiva no início da Computação, motivados pela carência de memória nas máquinas precursoras (ANCKAERT;MADOU;BOSSCHERE,2007). Códigos automodificáveis têm, portanto, uma longa história e foram utilizados para reaproveitar áreas de memória, e também para mascarar o código dos programas.

Para se ilustrar o uso de códigos automodificáveis, considera-se um código gerado automaticamente por um compilador, com a finalidade de otimizá-lo. Assume-se nessa situação, conforme Figura 3, que esse código tenha uma estrutura de controle com um bloco *if-then-else* interno a uma estrutura de repetição.

Em um programa como esse, o teste *if* deverá ser executado em cada iteração do *loop*.

```
;código original
Loop
        Bloco-A
        If condicao then
                Bloco-B
        Else
                Bloco-C
        End-if
        Bloco-D
End-Loop
```

Figura 3 - Código para Otimização

Para programas em que o resultado do teste é sempre o mesmo, o fluxo de controle do programa sempre será guiado para o mesmo desvio do *if-then-else*.

A execução repetida desse teste, para o caso em que já se sabe o resultado, torna a execução ineficiente.

Nessa situação, pode-se otimizar a execução do código, invertendo a ordem dos componentes e incluindo uma cópia do bloco de repetição em cada desvio do bloco *if-then-else*, conforme Figura 4.

Esta nova versão, no entanto, exibe um aumento nas exigências de memória, uma vez que inclui uma duplicação dos blocos A e D. Todavia, a execução é mais rápida, que a versão anterior, pois o teste *if* é executado apenas uma vez ao invés de ser executado em toda iteração do loop no código original.

```
;código modificado-1
If condicao then
        Loop
                Bloco-A
                Bloco-B
                Bloco-D
        End-loop
Else
        Loop
                Bloco-A
                Bloco-C
                Bloco-D
        End-Loop
End-if
```

Figura 4 - Código alterado para otimização

Nesse caso, os códigos presentes nos blocos A e D não devem modificar a condição do teste, pois se assim o fizerem, invalidarão a técnica de otimização apresentada.

Para se otimizar tanto o tamanho do código quanto seu tempo de execução, pode-se empregar uma terceira técnica, baseada em código automodificável, a qual considera uma área de memória, com tamanho suficiente de bytes, para acomodar o maior entre os blocos B ou C. Nessa área, será copiado o bloco apropriado em função do teste *if* executado, conforme Figura 5.

```
;código automodificável
If condicao then
        Cópia do código do Bloco B para area
Else
        Cópia do código do Bloco C para area
End-if
Loop
        Bloco-A
        Bloco-area
        Bloco-D
End-loop
```

Figura 5 - Otimização com automodificação

Essa técnica foi amplamente utilizada, em épocas passadas, para a alocação de memória de programas e se chama "*overlay*".

Na literatura recente pode-se encontrar um número significativo de publicações que indicam uma retomada do interesse do emprego de códigos automodificáveis para, dentre outras aplicações, proteção de código contra viroses, compressão de código, otimização de código, prevenção de engenharia reversa indesejada, etc. (ANCKAERT et al., 2007).

No apêndice B, mostra-se o funcionamento de um código automodificável desenvolvido para um processador hipotético.

Apresenta-se a seguir uma breve descrição de algumas áreas recentes que estão sendo beneficiadas com o emprego da técnica de código automodificável.

A atividade de *cracking*[1] tem apresentado um sério problema para a área de proteção de software. Um cenário típico ocorre quando se analisa uma rotina de proteção de cópia de código para, em seguida, modificar-se o programa com a finalidade de neutralizar a ação dessa rotina de proteção (MATSUMOTO; KANZAKI; MONDEN; NAKAMURA, 2003).

Um outro cenário ocorre quando uma empresa rival, pela análise de código, obtém de forma ilícita o conhecimento inerente ao programa original.

Ao se considerar a ação de um *cracker*[2] com a tarefa de anular uma rotina de verificação de senhas, pode-se resumidamente listar as seguintes atividades: Primeiro, decompõem-se o programa binário em um programa escrito em linguagem de montagem. Em seguida, para se reduzir o custo de compreensão do código, freqüentemente, delimita-se o espaço de código no qual será feita a análise. Por fim, um fragmento de código contendo a rotina de tratamento de senha é identificado e analisado. Como resultado, esta é neutralizada ou cancelada, com a conseqüente liberação ilegal do programa (MATSUMOTO; KANZAKI; MONDEN; NAKAMURA, 2003).

Existe, portanto, no conjunto de atividades desenvolvidas, uma fase na qual o programa deve ser compreendido, para que se obtenham indícios que levem ao programador a arquitetar a forma pela qual um possível *cracking* será idealizado.

[1] Ações para burlar segurança de sistemas, por meio de invasões e explanação intencional de falhas na segurança de computadores.
[2] Programador que desenvolve a atividade de cracking.

Uma solução efetiva para dificultar a ação do programador é o de aumentar os custos para a compreensão de um programa.

Muitos métodos de proteção de programas têm sido propostos na literatura, tais como: ofuscação de código, encriptação e fragmentação de programas. Todos esses métodos procuram aumentar os custos para a compreensão dos programas.

A ofuscação de programas consiste em se aplicar sucessivas transformações no programa original, gerando um programa equivalente, mais difícil de ser compreendido (MONDEN;TAKADA;TORII,1997).

Criptografia é uma técnica que emprega ferramentas de encriptação, para a obtenção de programas criptografados formados por duas partes: o código criptografado e módulos (não-encriptados) que, em tempo de execução, efetuam a decodificação do código (MATSUMOTO et al.,2003).

Apesar da dificuldade de compreensão de um código criptografado, há sempre o risco de a proteção ser cancelada pela análise e modificação dos módulos de encriptação. Métodos de criptografia estão descritos em (MORSE,1984; SAITO,2000).

Outra técnica empregada para proteção de software é a fragmentação de código. Esta basicamente consiste em dividir o programa em diversos fragmentos e controlar a reseqüênciação dos mesmos (MATSUMOTO et al., 2003).

Ferramentas de fragmentação geram programas compostos por duas partes: fragmentos do programa original e módulos que controlam, em tempo de execução, a seqüência de ativação.

Nesse caso, há o risco de se anular a proteção de software por meio de *cracking* no módulo de controle. Métodos que descrevem esta técnica estão apresentados em (AUCSMITH,1996).

Engenharia reversa é um conjunto de técnicas que permitem recriar um programa, ou seu modelo, tomando-se por base a análise de um código do produto final. Sua aplicação, na maioria das vezes, ocorre visando a melhoria de sistemas, tais como, otimizações, detecção de erros, remoção de vírus, recuperação de códigos-fontes, melhoria no processo de manutenção, etc. (ANCKAERT et al. ,2007).

As primeiras ferramentas de engenharia reversa empregadas para auxiliar o processo de depuração de software foram decompiladores, os quais ainda são úteis nos

processos de desenvolvimento de software, embora tenham sido empregados também para a obtenção de informações relativas ao licenciamento de software, ou então de detalhes sobre a operação e estruturas de dados de aplicações dos programas de computadores (HUSSAIN,2005). Isso justifica a construção de códigos executáveis que dificultem a ação da engenharia reversa (ANCKAERT et al., 2007).

Uma das alternativas para se implementar a proteção de software se dá através da técnica conhecida como ofuscação de código, consiste em se disponibilizar um algoritmo que, ao ser aplicado a um determinado programa, garanta que as seguintes propriedades sejam asseguradas: manutenção da funcionalidade, da eficiência e aumento da dificuldade de compreensão (ANCKAERT et al., 2007), (THOMBORSON; COLLBERG, 2002).

No entanto, em virtude da natureza das plataformas mais atuais, tais como, *Java* e *.Net*, a distribuição de código não-ofuscado, conforme (HUSSAIN,2005), é quase que equivalente à distribuição de código fonte aberto.

Para manter independência de plataforma, essas arquiteturas geram programas cujos códigos apresentam facilidade de decompilação aumentando, assim, o risco de serem atacados por engenharia reversa maliciosa (THOMBORSON; COLLBERG; LOW,1998).

Dentre os diversos modos de se implementar a ofuscação de código (TYAGI,2005), destaca-se a técnica de ofuscação por meio de código automodificável, a qual tem se mostrado efetiva em virtude da dificuldade de compreensão do código ofuscado (VAYNBERG;CAI;SHAO,2007).

A idéia-chave do método de proteção de software por meio de automodificação de código corresponde a adicionar um mecanismo de automodificação ao programa original para aumentar o custo de entendimento do programa original. No mecanismo de automodificação, uma instrução A do programa altera uma outra instrução B no mesmo programa por uma diferente instrução C, em tempo de execução (KANZAKI;MONDEN; NAKAMURA; MATSUMOTO, 2003).

De acordo com (KANZAKI,MONDEN;NAKAMURA;MATSUMOTO,2003), o mecanismo de automodificação de código usado em conjunto com as técnicas já conhecidas de proteção de código tais como, ofuscação de código, encriptação e fragmentação, podem aumentar a eficácia da proteção de software.

2.5 Linguagens Adaptativas

Dispositivos adaptativos guiados por regras iniciam a operação em alguma configuração inicial pré-estabelecida e evoluem sucessivamente para novas configurações até que seja atingida alguma configuração de parada (aceitação ou rejeição).

Caso o dispositivo nunca atinja tal situação, ocorre algo similar ao problema da parada da máquina de Turing, ou seja, o dispositivo entra em "loop" e não consegue decidir a situação.

Linguagens de programação podem ser vistas como um caso particular de dispositivos dirigidos por regras, em que as regras correspondem às diversas sub-árvores da representação do programa, conforme item 2.1 desta tese.

Seguindo o mesmo esquema de formalização empregado para os dispositivos adaptativos guiados por regras, define-se a seguir as linguagens de programação adaptativas.

Não será feita aqui uma formalização completa, explícita, das linguagens adaptativas consideradas. Entretanto, são apresentados os pontos principais dos quais podem resultar formalizações adequadas:

a) Dispositivo Subjacente: é a linguagem (não-adaptativa) que servirá de suporte para as ações adaptativas responsáveis por sua automodificação. O conjunto de regras que o definem têm o formato e o significado descritos no item 2.1 desta tese.

b) Camada Adaptativa: Compreende a definição de um conjunto de funções adaptativas, como combinações das funções elementares (? , + , -) clássicas (NETO,1994), e o acoplamento de suas chamadas parametrizadas às regras básicas do dispositivo subjacente.

c) Programa Adaptativo: Descrito na forma de um conjunto de regras adaptativas, representa o código cuja execução efetua o trabalho desejado.

Programas adaptativos exigem automodificação em tempo de execução. Linguagens adaptativas devem fornecer meios para que o programador expresse tais

automodificações em seus programas. É possível construir-se linguagens que realizem estas operações de forma nativa, e com sintaxe própria, por meio de bibliotecas de operações adaptativas em um ambiente de execução para fornecer o suporte ao compilador.

Alternativamente, é possível partir-se de linguagens existentes, estendendo-as para se incorporar adaptatividade. Há duas formas de se tratar a extensibilidade: extensão funcional e sintática. A extensibilidade funcional corresponde à criação de uma biblioteca própria para tratamento de adaptatividade, enquanto que a sintática permite se obter uma linguagem parecida com as genuinamente adaptativas, a partir da extensão sintática de uma linguagem extensível.

Ao empregarmos o conceito de extensibilidade, não estamos na verdade aplicando adaptatividade, uma vez que esta última ocorre em tempo de execução, enquanto que extensibilidade é tratada em tempo de compilação (NETO,2007).

Embora se possa considerar que a extensibilidade de linguagens de programação seja uma das mais significativas manifestações de pesquisa na área, seu escopo se limita, entretanto, aos componentes de linguagem que são tratados em tempo de compilação e corresponde, portanto, às alterações que o usuário pode fazer sobre a linguagem, de modo a torná-la mais adequada a seus particulares objetivos.

Esta limitação não contempla, portanto, as necessidades daqueles usuários que necessitem que seus programas possam efetuar dinamicamente, ou seja, em tempo de execução, alterações na estrutura de seus próprios algoritmos (NETO,2007).

Para isso, tais programas necessitariam incorporar recursos de adaptatividade, e para tanto deveriam apresentar recursos para que eles próprios, sem intervenção externa, pudessem optar pela efetivação de alguma automodificação necessária, para em seguida, realizá-la (NETO,2007).

A Figura 6 ilustra a situação de um programa (não-adaptativo) em uma linguagem-base desprovida de quaisquer mecanismos de extensão e de adaptatividade. Nesse caso, o programa gerado a partir dessa linguagem-base terá código fixo (estático) e sem quaisquer adições de recursos na linguagem-base, uma vez que esta não é extensível.

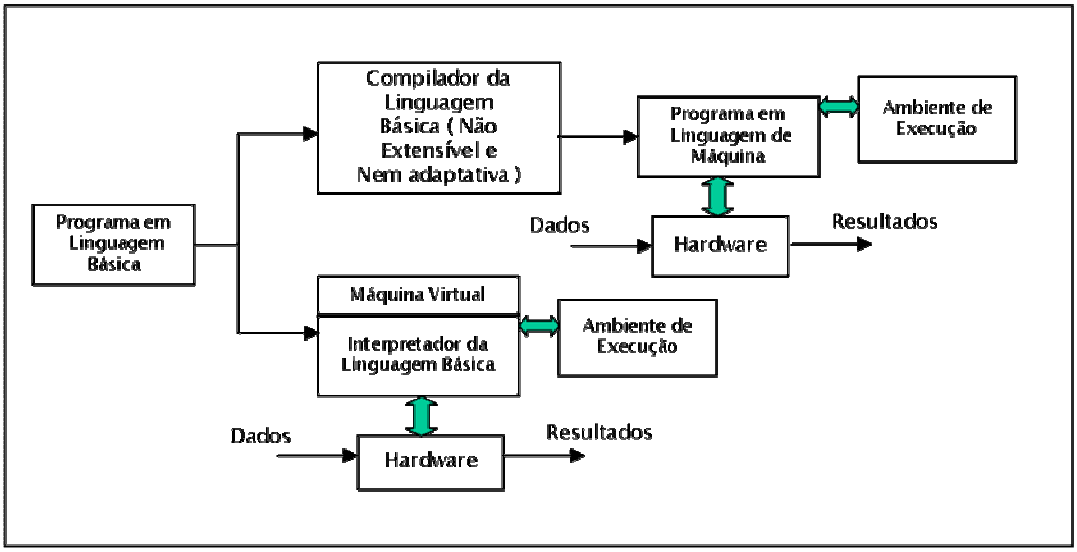


Figura 6 - Processamento de um programa em Linguagem Básica (sem extensão e sem adaptatividade)

Em seguida considera-se o caso em que à linguagem-base acrescenta-se um mecanismo de extensibilidade, conforme ilustrado na Figura 7.

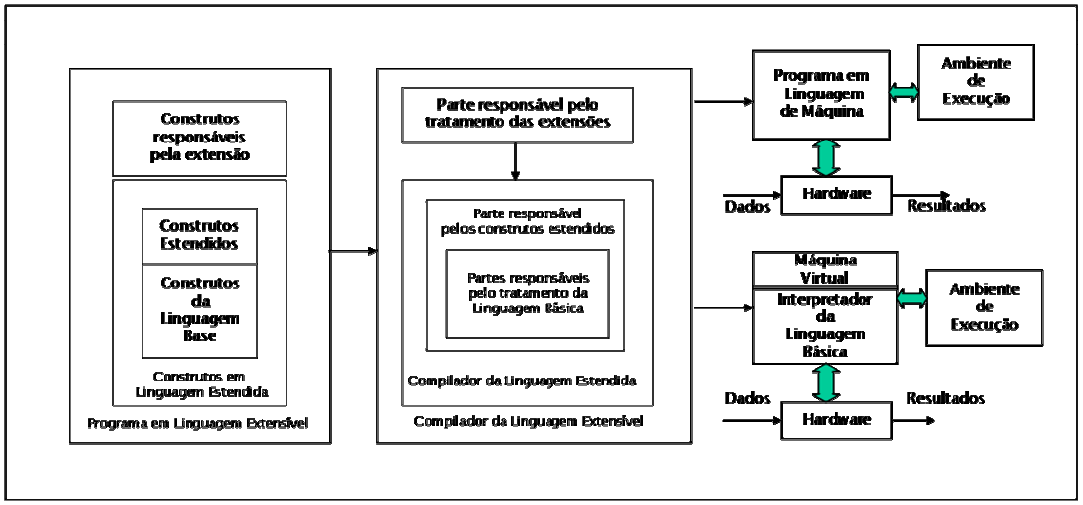


Figura 7 - Processamento de um programa em uma Linguagem Extensível e sem Adaptatividade

Na situação da Figura 7, o programa gerado a partir da linguagem estendida, também terá código fixo, ou seja, suas instruções terão, em tempo de execução, um comportamento estático, uma vez que não se agregou ao ambiente qualquer mecanismo de adaptatividade. No entanto, o programa gerado foi obtido com uma linguagem diferente da linguagem-base, uma vez que existe um mecanismo de extensibilidade de linguagem presente nessa situação.

A Figura 8 ilustra o caso em que se acrescenta um mecanismo adaptativo à linguagem-base.

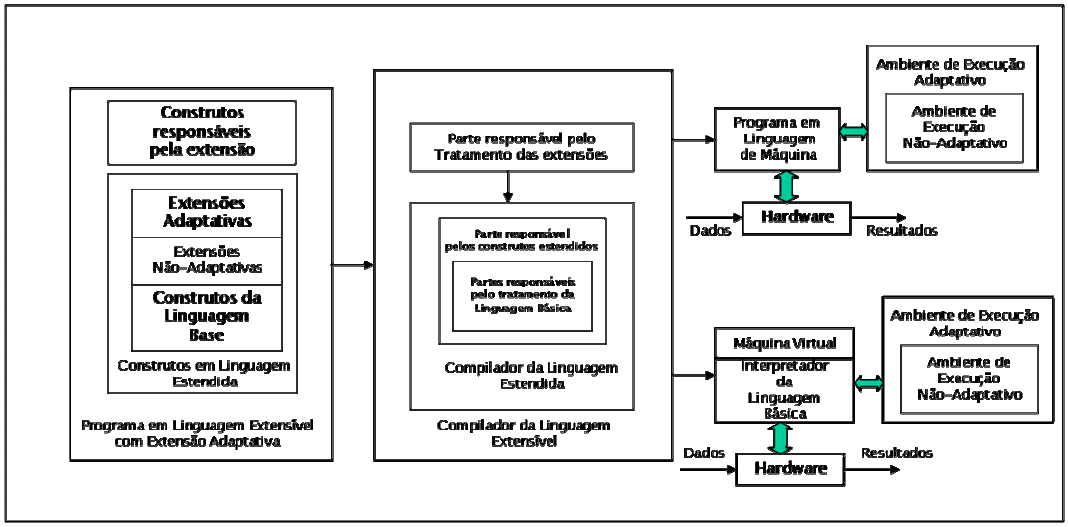


Figura 8 - Geração de Programas com Adaptatividade a partir de Linguagem Extensível

Na situação da Figura 8, o programa gerado pode ter comportamento automodificável caso apresente características adaptativas. Nesse caso, o programa foi obtido a partir de uma linguagem adaptativa, que por sua vez, foi desenvolvida com os recursos de extensibilidade disponíveis na linguagem-base.

Por fim, a Figura 9 ilustra a geração de programas adaptativos a partir de uma linguagem genuinamente adaptativa.

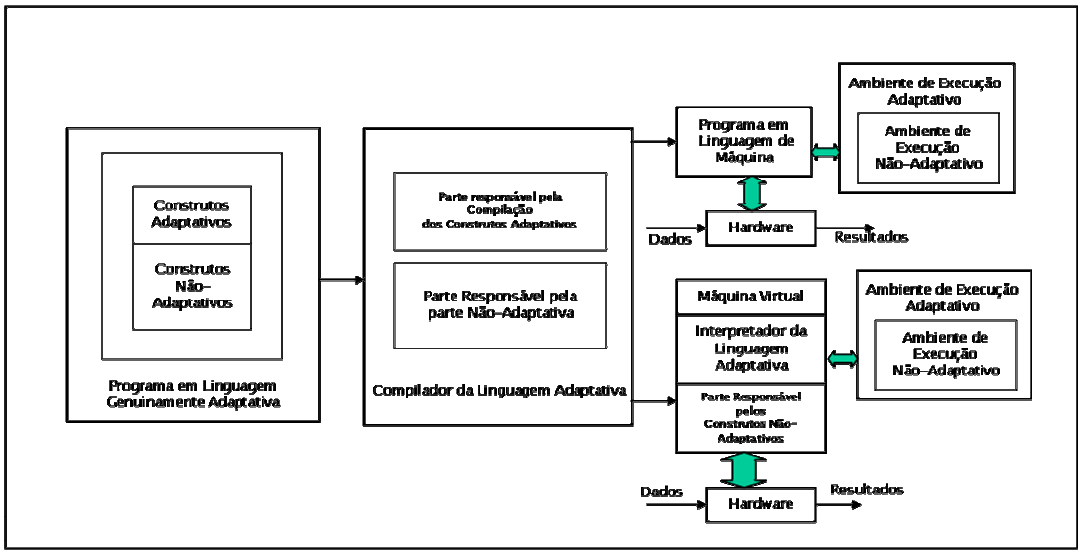


Figura 9 - Programas Adaptativos gerados a partir de Linguagem Genuinamente Adaptativa

3. Concepção de Linguagens Adaptativas

Com base nos conceitos de adaptatividade e em ensaios realizados com linguagens, chegou-se a algumas conclusões sobre a forma como uma linguagem adaptativa poderia ser descrita.

3.1 O Dispositivo Subjacente

Para se evitar os problemas mencionados no capítulo 2, sem impedir o uso da adaptatividade, propõe-se neste trabalho que as automodificações sejam restritas apenas àquelas especificadas por ações adaptativas, nas quais se confinariam todas as automodificações utilizadas, e por meio de restrições do uso podem ser impostas disciplinas que assegurem as propriedades desejáveis para o programa desenvolvido.

Nesta tese, considerou-se que o mais adequado seria usar uma linguagem subjacente de alto nível que apresentasse propriedades que facilitassem a inclusão de características adaptativas. Optou-se por um dialeto Lisp (McCARTHY,1960), por causa da sua aderência ao conceito de extensibilidade funcional, e pela facilidade com que permite que áreas de dados possam ser executadas como se fossem instruções[3], o que na maioria das linguagens de programação, não é uma operação trivial.

Uma linguagem adaptativa baseada em linguagem funcionalmente extensível deve disponibilizar operadores (funções) não-adaptativos e adaptativos (expressos na linguagem subjacente). Os programas dela oriundos terão comportamento automodificável em tempo de execução e o processamento dos mesmos será efetuado com o auxílio do interpretador da linguagem subjacente.

Assim, torna-se relativamente simples criar e executar dinamicamente partes do programa que não estavam presentes ao início de execução do programa.

[3] Em particular, a linguagem *Lisp* apresenta a função nativa "*eval*" que permite avaliar uma cadeia como se fosse uma expressão, e retorna um resultado. *Lisp* foi a primeira linguagem a fazer uso de funções dessa natureza, principalmente na construção de interpretadores.

Deve-se observar que a linguagem *Lisp,* originalmente funcional, não foi escolhida devido a essa característica, e sim, principalmente à sua facilidade de extensão funcional e de incorporação de características adaptativas.

O presente trabalho mostra a viabilidade prática de construir uma linguagem de programação adaptativa, embora não tenha como meta disponibilizar um produto acabado, mas sim levantar algumas especificações que possam nortear o futuro projeto de linguagens genuinamente adaptativas.

De forma análoga ao que ocorre com linguagens de alto nível, cuja sintaxe se baseia em abstrações de nível elevado, as linguagens adaptativas são mapeadas em construções sintáticas de nível inferior de abstração, representadas por uma linguagem hospedeira e por chamadas de funções pertencentes à camada adaptativa a ela acrescentada por extensão funcional.

3.2 Adaptatividade em Linguagens de Programação

Propõe-se aqui uma arquitetura para a camada adaptativa da linguagem adaptativa desenvolvida neste trabalho. Um programa, escrito na linguagem adaptativa, tem, no início de sua execução, um comportamento semelhante àquele que não incorpora adaptatividade, podendo ser, portanto, processado pelo intepretador da linguagem *Lisp* subjacente.

Sempre que ocorrer a execução de uma ação adaptativa, tudo se passa, portanto, como se o programa fosse, temporariamente, suspenso e as modificações, efetuadas. Ao final do processamento de cada ação adaptativa, uma nova instância do programa surge como resultado, e a execução é retornada ao interpretador *Lisp* subjacente que se encarrega de prosseguir sua execução.

Dessa forma, um programa, expresso em linguagem adaptativa, partindo da configuração inicial de código CF_1, evolui para sucessivas configurações CF_2, CF_3,.., CF_n , pela aplicação de ações adaptativas, à medida que sua execução for progredindo.

Note-se que essa seqüência de configurações depende fortemente dos dados, e, portanto, só pode ser determinada durante a execução do programa, sendo que execuções de um mesmo programa, com dados diferentes, podem produzir seqüências diferentes de configurações.

Tendo em vista que as funções adaptativas devem expressar todas as possibilidades de automodificação do programa, torna-se conveniente disponibilizar algum recurso que permita ao programador referenciar, univocamente, as porções de código que serão objeto de tal automodificação. É natural que, especialmente no caso de linguagens funcionais, a introdução de tais mecanismos de referenciação afaste o programa do paradigma funcional. Entretanto, sua utilização pode simplificar os procedimentos necessários à implementação da camada adaptativa, razão pela qual foi mantido neste projeto.

Preferencialmente, a linguagem subjacente deve oferecer meios para o programador fazer essa referenciação. Caso a linguagem subjacente não ofereça formas nativas para tal referenciação, torna-se necessário que algum recurso extra seja disponibilizado ao usuário, por exemplo, na forma de algum tipo de extensão, adicionado à linguagem subjacente.

No caso particular da linguagem *Lisp,* os programas podem ser representados como árvores. Dessa forma, sua automodificação pode ser implementada através da alteração estrutural da árvore que o representa.

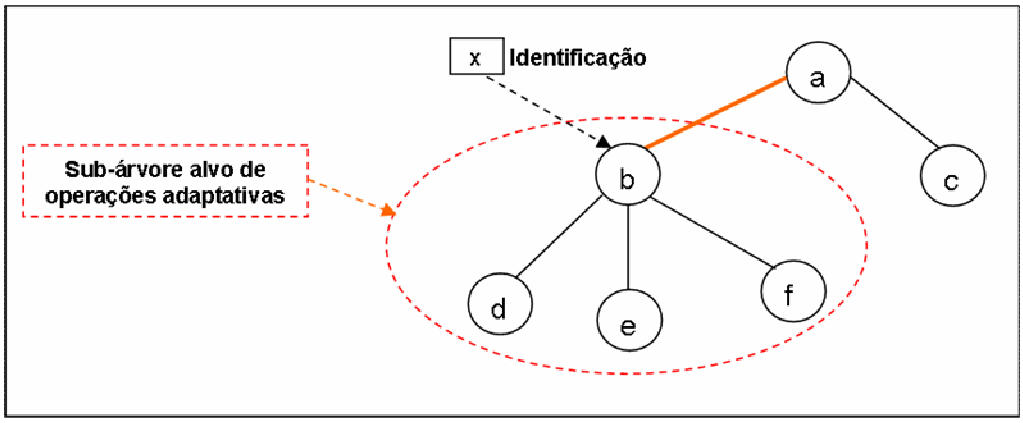


Figura 10 - Identificação de Nós de árvore: x identifica a sub-árvore cuja raiz é o nó b

Conforme esboçado na Figura 10, a identificação da região a ser modificada corresponde a um nome simbólico associado à raiz da sub-árvore que a representa.

Considerando que, no dialeto Lisp escolhido para este trabalho, não há funções nativas que enderecem os nós internos da árvore que representa o programa, torna-se necessário que se disponibilize, na camada adaptativa alguma operação que permita ao usuário refernciar uma sub-árvore.

Assim, propõe-se como primitiva da camada adaptativa, uma função, denominada *mark*, que terá como parâmetros um nome (que define um identificador para ser usado como referência) e uma porção de código (que define a sub-árvore que será alvo da adaptatividade).

Para ilustrar a operação desta primitiva, considere-se o programa, esboçado na forma de árvore, na Figura 10.

Na Figura 11, mostra-se a árvore que representa o programa, devidamente marcado pela primitiva *mark*, com a identificação 'x'.

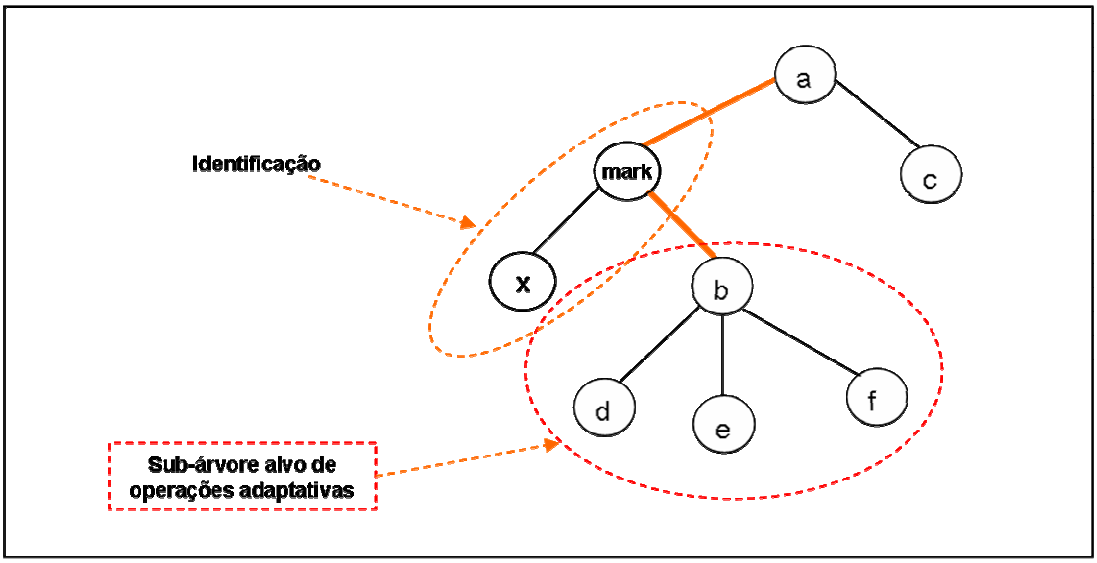


Figura 11 - O mesmo programa da Figura 10, em sua representação de árvore, com nó identificado pela primitiva mark.

O capítulo 4 dessa tese apresenta os detalhes da implementação da primitiva *mark* presente na camada adaptativa.

Estabelecida a forma de vinculação de identificações às porções correspondentes de código automodificável do programa, podem-se construir funções adaptativas que se responsabilizem pelas operações de automodificação do código do programa adaptativo.

Optando-se por não utilizar identificações explícitas, as funções adaptativas deverão ativar funções equivalentes da biblioteca do Lisp para localizar e manipular as sub-estruturas desejadas.

3.3 As Funções Adaptativas

As funções adaptativas, aplicadas a um nó específico da árvore que representa o programa, são codificadas a partir da combinação de um conjunto de ações elementares de alteração, inclusão e exclusão de nós, que, em tempo de execução, afetarão a forma da árvore que representa o programa.

Por exemplo, uma função qualquer $\mathcal{F}$ pode ativar funções primitivas de alteração, inclusão e remoção de sub-árvores.

Essas primitivas de alteração, inclusão e remoção de porções de código devem ser disponibilizadas pela camada adaptativa para serem utilizadas na programação das funções adaptativas.

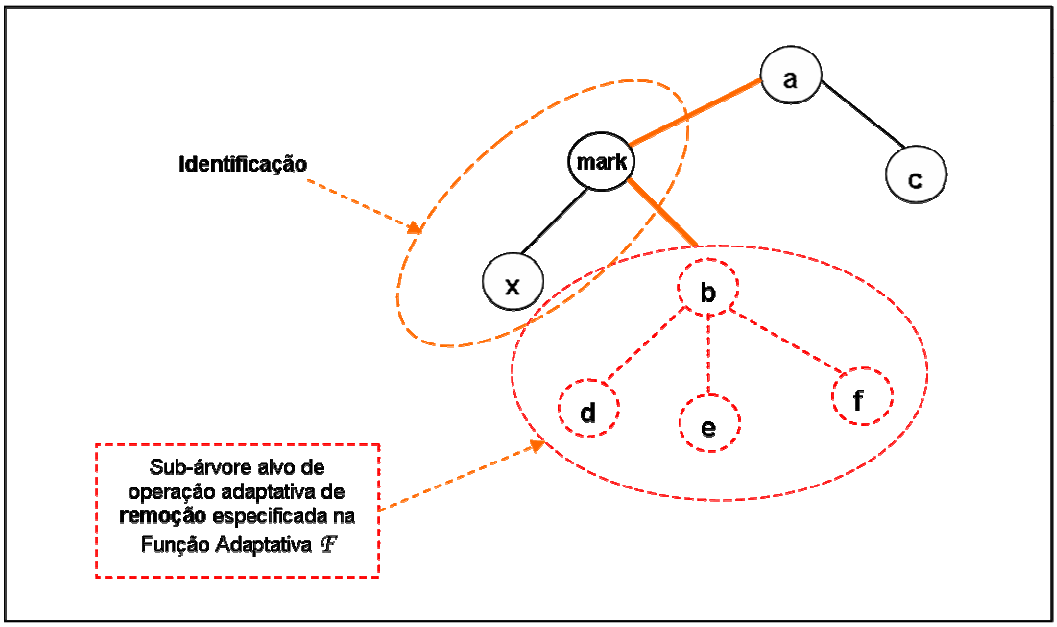


Figura 12 - Adaptação da Árvore – Ação de Remoção

Durante a aplicação da função adaptativa $\mathcal{F}$ mencionada acima, poderá ocorrer, por exemplo, a remoção de uma sub-árvore, em um ponto especificado da árvore que representa o programa, conforme esboçado na Figura 12.

Após a execução de uma ação de remoção especificada na função adaptativa $\mathcal{F}$ mencionada anteriormente, a árvore que representa o programa terá a forma esboçada na Figura 13.

No caso em questão, está se admitindo que a ação de remoção é aplicada na sub-árvore apontada pelo endereçamento definido pela função *mark*. Dessa forma, o endereçamento continua a existir na estrutura do programa e a sub-árvore alvo de adaptatividade é substituída por um nó representando *nil*.

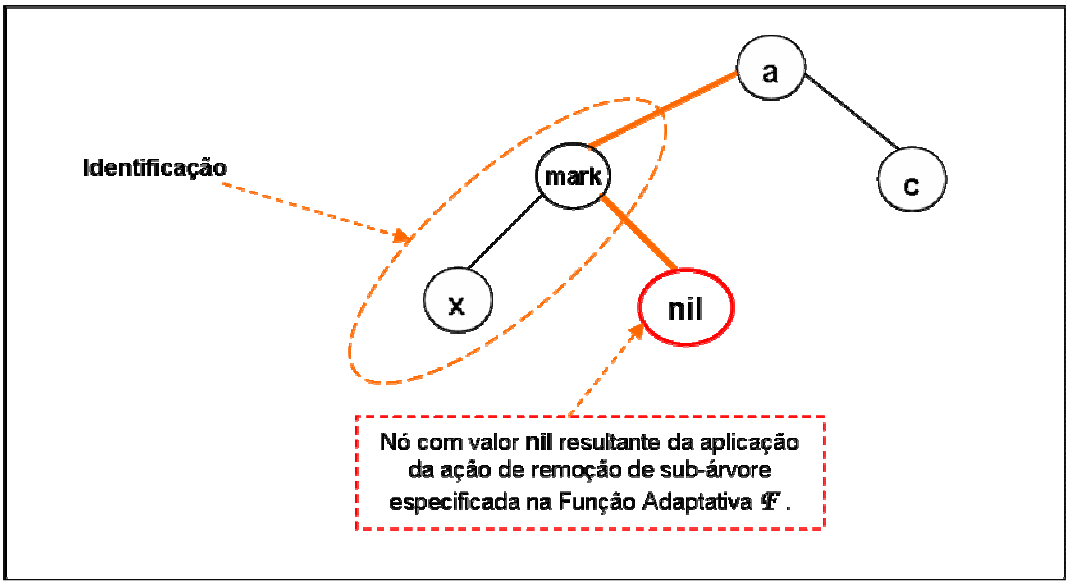


Figura 13 - Adaptação da Árvore após Ação de Remoção

Em caso de se desejar remover, da árvore que representa o programa, uma determinada sub-árvore juntamente com a marca a ela associada, será necessário que seja disponibilizada, na camada adaptativa, outra primitiva para tal propósito. Esse caso é esboçado na Figura 14.

No caso de se optar pelo uso de primitivas que não utilizam identificadores explícitos, pode não ser preciso identificar a sub-árvore modificada, mas deve-se tomar o cuidado de, se necessário, preservar-se ao menos a topologia da árvore ancestral, para evitar perda de integridade do programa representado. Caso seja necessário prever manipulação futura do resultado dessas operações, a inclusão de marcas pode ser feita, eventualmente, por uma função adequada e a marca poderá, em outra ocasião, servir como referência.

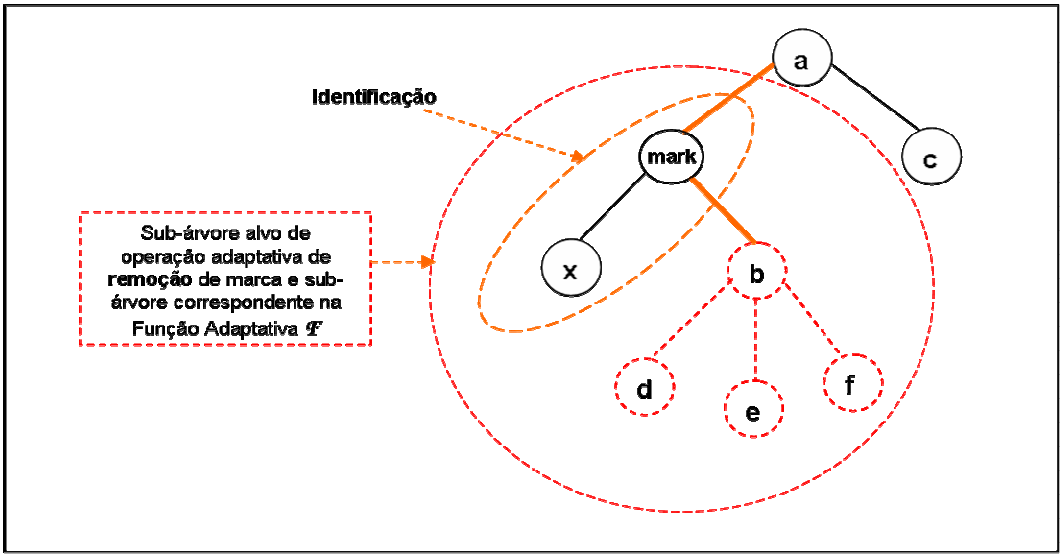


Figura 14 - Adaptação da Árvore - Ação de Remoção de Sub-árvore e marca correspondente

Da mesma forma, $\mathcal{F}$ pode aplicar à árvore correspondente ao programa, uma ação adaptativa elementar de inclusão, e neste caso a árvore também tomaria uma nova forma, na qual uma sub-árvore (passada como parâmetro na ação adaptativa) seria inserida na posição especificada (também como parâmetro) da árvore original. Esta alteração está esboçada na Figura 15, a partir da configuração apresentada na Figura 13.

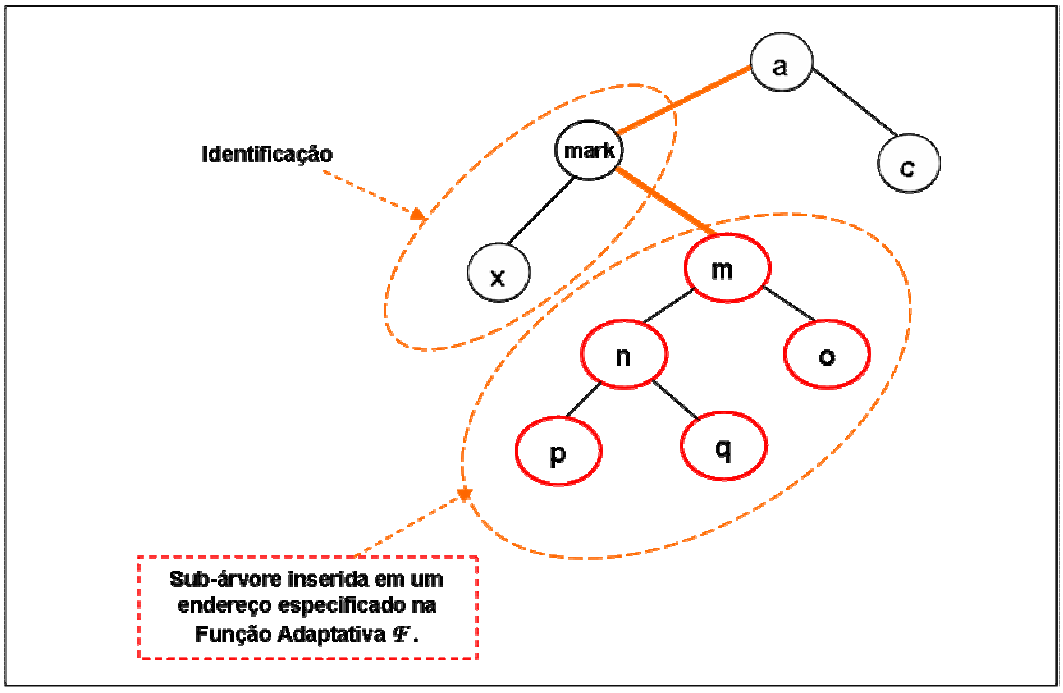


Figura 15 - Adaptação da Árvore - Ação de Inserção de Sub-árvore

A operação seqüencial das ações adaptivas elementares de remoção e inserção pode ser equivalentemente operacionalizada por uma outra primitiva de atualização de sub-árvores.

Em caso de operação de consulta, a ação elementar terá como parâmetro um nome que corresponde a sub-árvore alvo da consulta. A sub-árvore correspondente a este nome será retornará pela ação elementar.

O capítulo 4 dessa tese exibe detalhes a respeito da implementação das ações elementares aqui apresentadas. Apresentam-se a seguir algumas considerações a respeito da operação do programa adaptativo resultante de sua utilização.

3.4 Operação do Programa Adaptativo

Para que o programa, obtido a partir da linguagem adaptativa proposta nessa tese, possa ser convenientemente processado, necessita-se de um ambiente de execução adequado, que é composto pelo interpretador da linguagem subjacente e pela camada adaptativa, que terá como responsabilidade avaliar chamadas adaptativas e administrar a execução do programa.

Conforme a Figura 16, os seguintes eventos ocorrem durante o processamento de programas escritos na linguagem adaptativa:

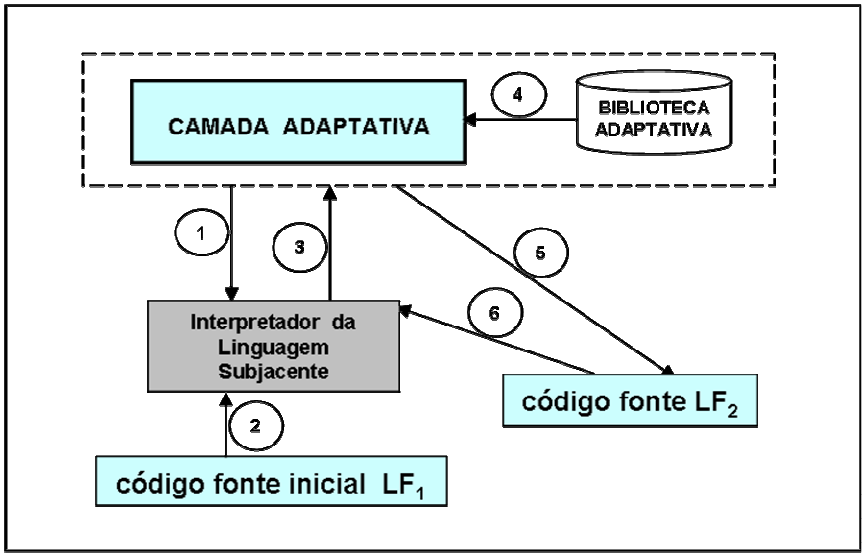


Figura 16 - Operação Conceitual de uma Linguagem Adaptativa

1. A camada adaptativa efetua a chamada do interpretador da linguagem subjacente, passando-lhe o código-fonte inicial fornecido pelo usuário.
2. O interpretador da linguagem subjacente inicia a avaliação das funções de forma usual, até que ocorra alguma chamada de função adaptativa. Caso não haja chamadas adaptativas, o avaliador atuará tal qual um interpretador não-adaptativo.
3. O controle é retornado para a camada adaptativa, que irá providenciar o tratamento da chamada adaptativa.
4. Considerando-se que as ações adaptativas são compostas por ações elementares, a camada adaptativa utilizará funções da biblioteca adaptativa.
5. Como resultado da execução das ações adaptativas, um novo código-fonte é gerado.
6. Este novo código-fonte é passado ao interpretador da linguagem subjacente, que se encarregará da continuidade da execução do programa adaptativo.

3.5 A Camada Adaptativa

O programa adaptativo será processado em uma camada adaptativa, especificamente preparada para comportar operações adaptativas, a qual terá como responsabilidade dar todo o suporte necessário para que a adaptatividade seja devidamente validada no programa, em tempo de execução.

A camada adaptativa basicamente será composta por um procedimento do tipo "*read-eval-printloop"* (FLATT et al,1999; NORVIG, 1990). Nesse procedimento, "*read*" efetua a leitura do programa, "*eval*" efetua o tratamento das funções adaptativas responsáveis pela automodificação e posterior avaliação e "*print*" exibe os resultados.

Assim, basicamente teremos na camada adaptativa, as seguintes operações:

1) Carga da instância inicial do programa fonte do usuário.
2) Gravação do código adaptativo em uma área de memória auxiliar.
3) Avaliação do código adaptativo com o auxílio da função "*eval*" da linguagem hospedeira.

4) Avaliação das funções adaptativas chamadas pelo código adaptativo.

Considerando-se que a camada adaptativa faz uso do interpretador da linguagem subjacente, em caso do programa a ser carregado não apresentar chamadas adaptativas, haverá um processamento tal qual a execução de um código *Lisp* usual.

Apresenta-se a seguir o pseudocódigo da camada adaptativa proposta nessa tese.

```
(enquanto houver programa adaptativo para execução
        (carga do programa adaptativo)
        (salvamento do código em área do ambiente)
        (avaliação do programa adaptativo)
                (avaliação de chamadas adaptativas)
        (apresentação dos resultados)
)
```

O capítulo 4 dessa tese apresenta maiores detalhes a respeito da implementação da camada adaptativa proposta.

3.6 Programas Adaptativos

Programas automodificáveis irrestritos permitem alterar arbitrariamente seu código. Isso dá origem a uma série de problemas, como já foi dito no capítulo 2. Para evitar tais fenômenos indesejáveis, é possível impor restrições às modificações operadas por tais programas. Nos programas adaptativos, toda ação de modificação fica encapsulada nos comandos pertencentes à camada adaptativa do programa. Assim sendo, é aí a região do programa mais adequada para a inclusão das restrições que o impedem de sofrer alterações consideradas indesejáveis.

Programas adaptativos são gerados a partir de uma linguagem de programação adaptativa e que podem, por sua vez, ser classificadas de acordo com a forma pela qual funções adaptativas são projetadas.

Assim, conforme a Figura 17, a programação adaptativa permite a obtenção de dois tipos de programas automodificáveis: aqueles cuja disciplina é imposta pela aplicação de ações adaptativas arbitrárias e aqueles em que, de um modo mais rígido, as ações adaptativas satisfazem restrições convenientes para cada particular problema em questão.

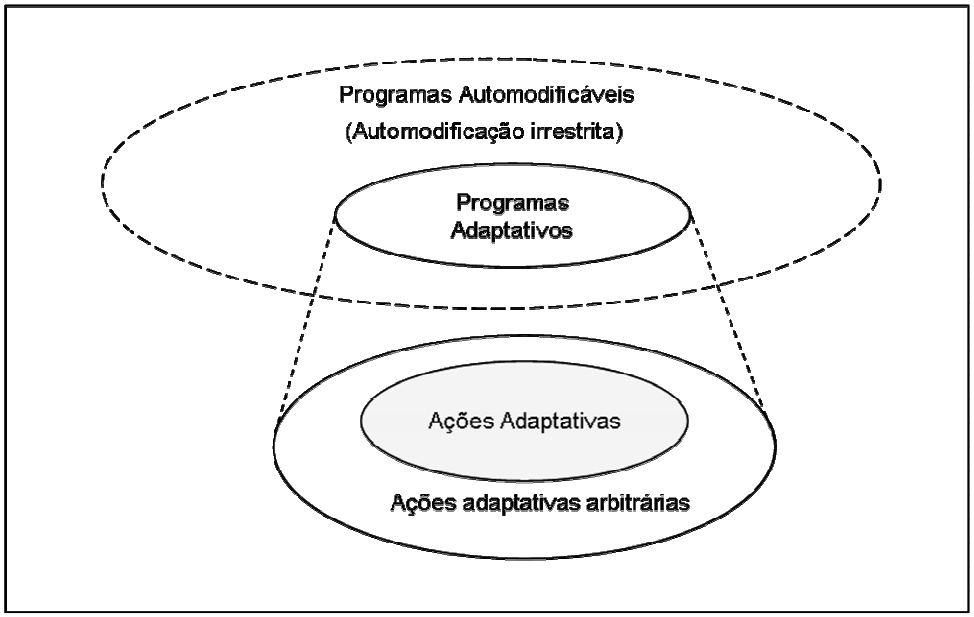


Figura 17 – Classificação de Programas Adaptativos

Durante o projeto das funções adaptativas, é fundamental que o programador tenha controle total sobre suas automodificações, de modo que tais funções sejam escritas seguindo as restrições mais adequadas para o problema particular em questão.

Tais programas adaptativos são preferíveis em relação a outros que se automodificam irrestritamente.

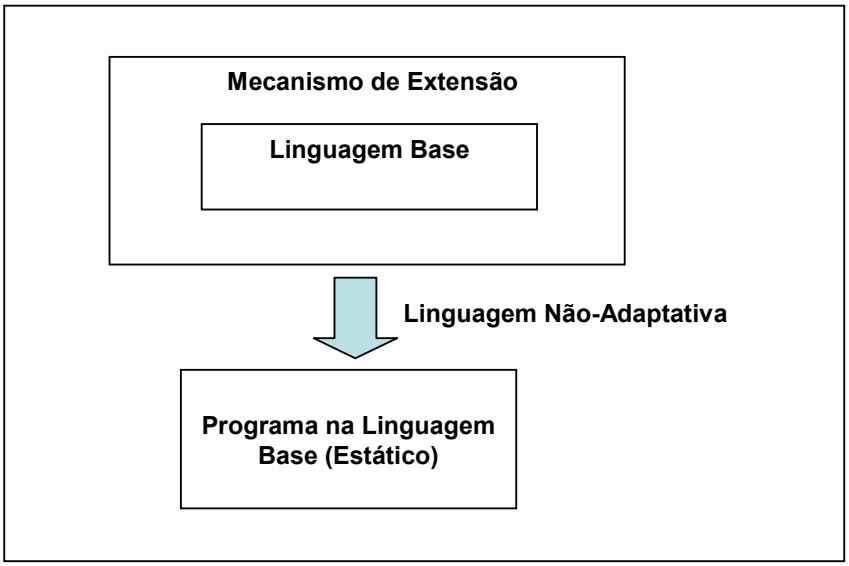


Figura 18 – Geração de Programa na Linguagem-Base (Não-adaptativa)

A Figura 18 esboça a geração de um programa não-adaptativo a partir de uma linguagem extensível na qual, embora haja um mecanismo de extensão, o programa gerado possui apenas construtos da linguagem-base.

A Figura 19 esboça um programa oriundo de uma linguagem estendida e sem adaptatividade.

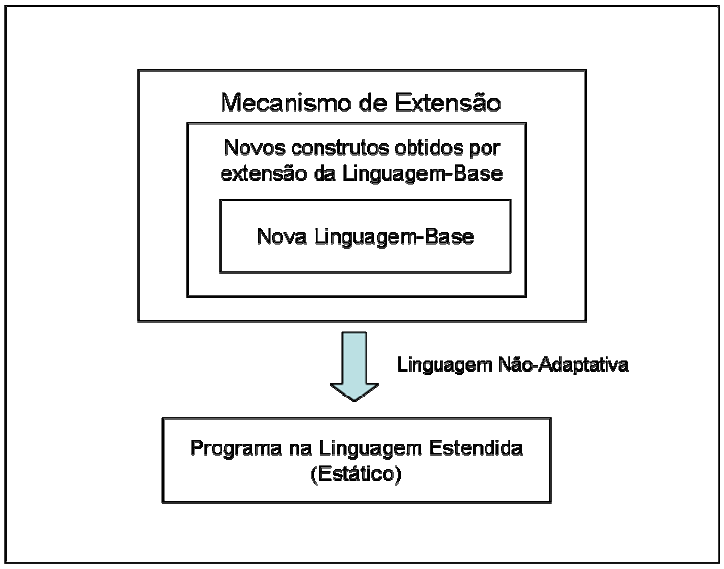


Figura 19 – Geração de Programa Não-adaptativo por Extensão

A Figura 20 retrata o instante em que o programa adaptativo é gerado em tempo de compilação.

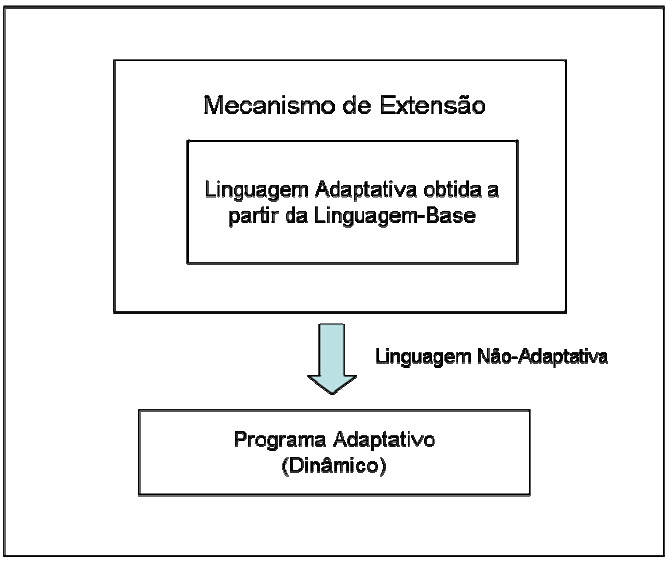


Figura 20 – Geração de Programa Adaptativo

4. Implementação da Camada Adaptativa

Neste capítulo, apresentam-se algumas considerações a respeito da implementação da camada adaptativa. São relatadas as principais primitivas dessa camada e a forma pela qual as mesmas foram definidas nesta tese. Para os experimentos, emprega-se aqui a implementação *Newlisp* (MUELLER,2007). Detalhes desta implementação, podem ser obtidos em www.newlisp.org.

4.1 Endereçamento de Porções Modificáveis do Código

Conforme explanado no Capítulo 3, as funções adaptativas devem expressar todas as possíveis operações utilizadas nas modificações do programa adaptativo. Para que o programador possa expressar, durante a escrita das funções adaptativas, a forma pela qual estas promovem a modificação de código, torna-se necessário disponibilizar algum meio que lhe permita endereçar, univocamente, cada uma das porções de código que sejam passíveis de modificação.

Caso esse mecanismo de endereçamento esteja disponível na linguagem subjacente, basta lançar mão desse mecanismo para a escrita das funções adaptativas.

Considerando que, nas implementações usuais da linguagem *Lisp*, não há funções nativas que endereçem nós internos da árvore que representa o programa, torna-se necessário que esteja disponível, alguma primitiva que permita ao usuário endereçar ao menos os nós que correspondam às partes modificáveis de código. Optou-se por incorporar tal recurso na camada adaptativa.

Assim, propõe-se como primeira primitiva da camada adaptativa, a função *mark*, que recebe como primeiro parâmetro um identificador que vincula um nome à porção de código que se deseja tornar alvo da adaptatividade, e, como segundo parâmetro, a sub-árvore propriamente dita.

A definição da primitiva *mark* pode ser feita da seguinte maneira:

```
(define

        (mark identificador subarvore)
        (set identificador subarvore)
)
```

A vinculação de um identificador a uma sub-árvore que representa uma porção de código modificável pode ser feita da seguinte forma:

```
(mark identificador  subarvore)
```

Para ilustrar o emprego da primitiva *mark*, a Figura 21 apresenta uma estrutura em árvore (que corresponde a um programa *Lisp),* representada pela lista:

```
(a   (b   d   e ) ( c   f   g ) )
```

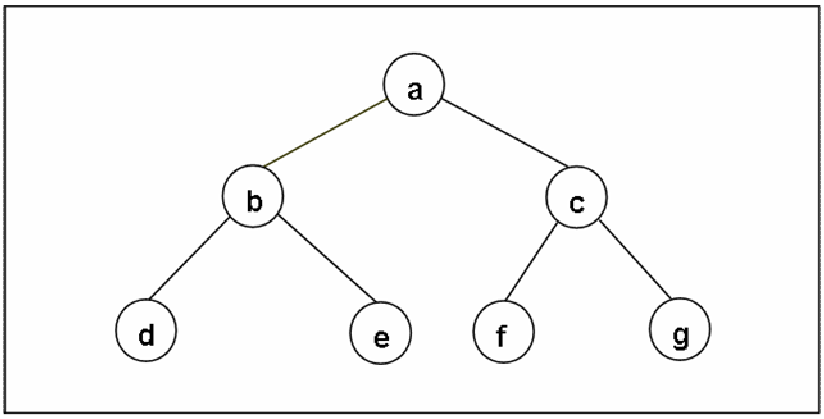


Figura 21 - Árvore representando o programa Lisp (a (b d e) (c f g))

A Figura 22 ilustra a situação em que se deseja promover a automodificação de código na sub-árvore cuja raiz é o nó c.

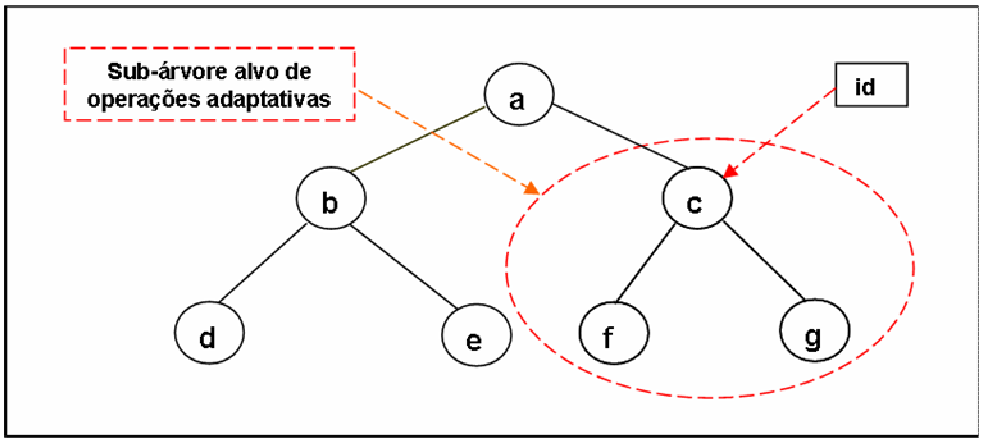


Figura 22 – Identificação do nó para Adaptividade no programa (a (b d e) (c f g))

Para que alguma função adaptativa possa endereçar a sub-árvore da Figura 22, deve-se efetuar a chamada da primitiva *mark* com a seguinte assinatura:

```
(mark 'id ( c f g ) )
```

A primitiva *mark* não altera a avaliação da sub-árvore passada como parâmetro, uma vez que não se efetuam alterações nessa sub-árvore.

Uma vez definida a associação entre o identificador e sub-árvore correspondente, este vínculo fica disponível na camada adaptativa, para futuras referências por parte de funções adaptativas presentes no programa.

Com a chamada da primitiva *mark*, a árvore que representa o programa (esboçada na Figura 22) modifica-se, assumindo a forma esboçada na Figura 23.

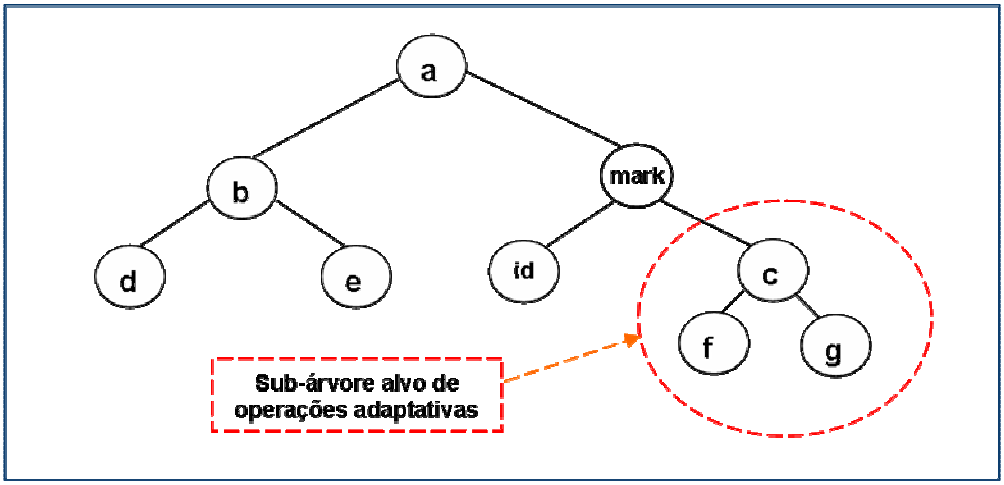


Figura 23 – Árvore com nós marcados pela primitiva mark

Para ilustrar o emprego da primitiva *mark*, considera-se abaixo a execução de um pequeno programa, com o seguinte código *Lisp*:

```
> (define ( pgm x ) (+ x x ) )
> (pgm 3)
6
```

Por meio da primitiva *mark*, atribui-se ao nó-raiz da árvore que representa o programa, um identificador, por exemplo, com o valor 'id. Para que este vínculo seja estabelecido, escreve-se:

```
> (define ( pgm x ) (mark 'id (+ x x ) ) )
> (pgm 3)
6
```

4.2 Primitiva para remoção de código

Neste item apresentam-se detalhes de implementação da primitiva *rem* que permite a remoção de uma porção de código resultando, como conseqüência, uma modificação controlada da árvore que representa o programa *Lisp* em execução.

Para que esta primitiva possa operar, usa-se a primitiva *mark* apresentada no item anterior.

A primitiva *rem* recebe como primeiro parâmetro a sub-árvore que constitui o trecho de código a ser modificado e, como segundo parâmetro, um identificador que seleciona a sub-árvore a ser removida da sub-árvore passada no primeiro parâmetro.

A definição da primitiva *rem* pode ser feita da seguinte maneira:

```
(define
        (rem arvore identificador )
                (busca  arvore identificador )
                (remove subarvore)
                arvore
        )
)
```

A primitiva *rem* tem como rotinas elementares a função *busca,* que tem como responsabilidade efetuar uma busca em profundidade para verificar e identificar a ocorrência da marca passada como parâmetro, e assim, definir o nó da árvore que representa a raiz da sub-árvore a ser removida.

Para ilustrar o emprego da primitiva *rem*, a Figura 24 apresenta uma estrutura em árvore representada pela lista:

```
(a (b  d  e) (c (mark  'id  (g  h  i)) f))
```

Na situação esboçada na Figura 24, a primitiva *mark* endereça a sub-árvore (g h i) que é alvo de uma ação adaptativa de remoção.

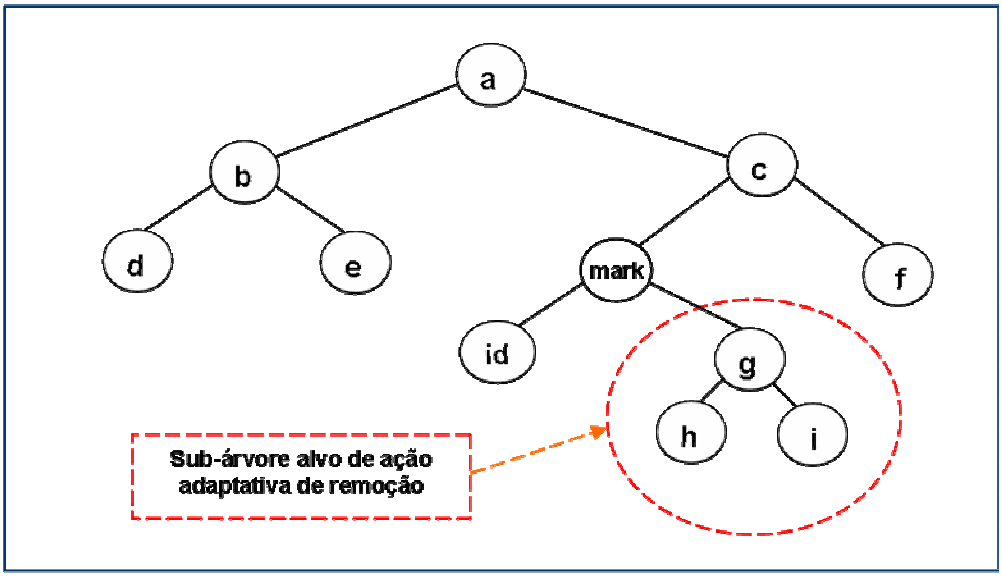


Figura 24 – Árvore com sub-árvore a ser removida

A figura 25 esboça a árvore gerada pela aplicação da primitiva *rem* à estrutura da Figura 24.

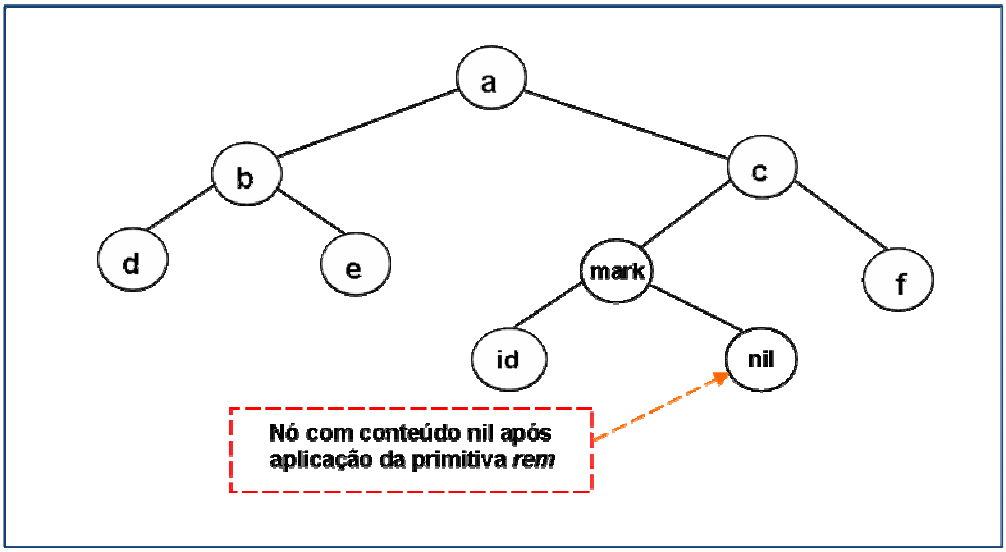


Figura 25 – Árvore resultante da aplicação da primitiva rem

Conforme esboçado na Figura 25, a primitiva *rem* remove a sub-árvore que lhe é passada como parâmetro e a substitui por um nó que corresponde à ausência de código. Tal nó em *Lisp* representa uma lista vazia.

Recomenda-se ao programador que, ao utilizar a primitiva *rem*, substitua uma sub-árvore por outra cujo resultado seja da mesma natureza, para evitar inconsistências semânticas.

Como isso interfere com a liberdade do programador, tal verificação não é realizada pelas primitivas implementadas.

Observa-se, adicionalmente, que a primitiva *rem*, embora remova a sub-árvore que lhe é passada como parâmetro, mantém na árvore resultante o nó correspondente à marca de endereçamento.

Para que seja possível à função adaptativa promover a remoção de código juntamente com os nós correspondentes à marca de endereçamento de sub-árvore, disponibiliza-se também a primitiva *rem-tree* com a mesma assinatura da primitiva *rem*.

A definição da primitiva *rem-tree* pode ser feita da seguinte maneira:

```
(define
        (rem_tree arvore identificador )
                (busca  arvore  identificador )
                (remove subarvore )
                (remove identificador )
                arvore
        )
)
```

A Figura 26 esboça a árvore gerada pela aplicação da primitiva *rem-tree* à estrutura da Figura 24.

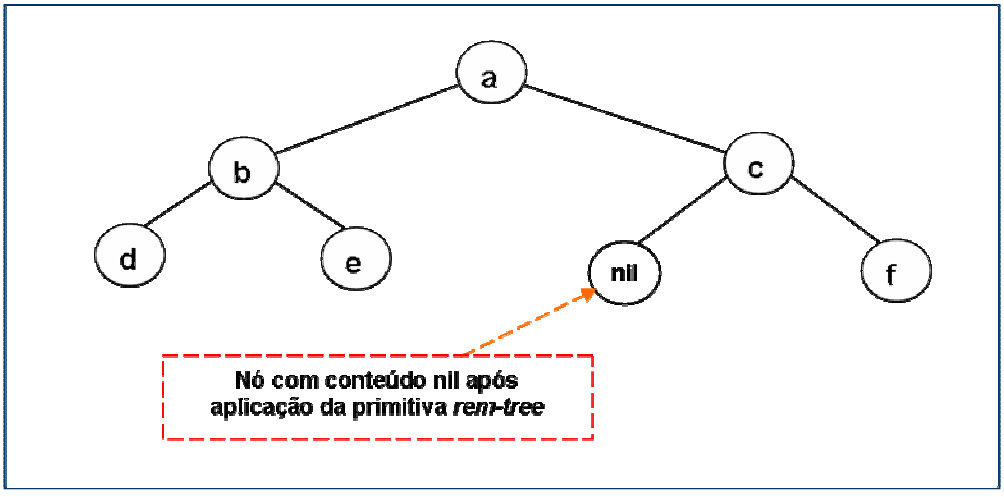


Figura 26 – Árvore resultante da aplicação da primitiva rem-tree

Em caso do programador necessitar da operação de remoção da marca previamente incorporada a alguma sub-árvore, mas sem removê-la, disponibiliza-se na camada adaptativa a primitiva *rem-mark* para tal finalidade.

A definição da primitiva *rem-mark* pode ser feita da seguinte maneira:

```
(define
        (rem_mark   arvore  identificador )
                    (busca  arvore   identificador )
                    (remove identificador )
                    arvore
         )
)
```

A Figura 27 esboça a árvore gerada pela aplicação da primitiva *rem-mark* à estrutura da Figura 24.

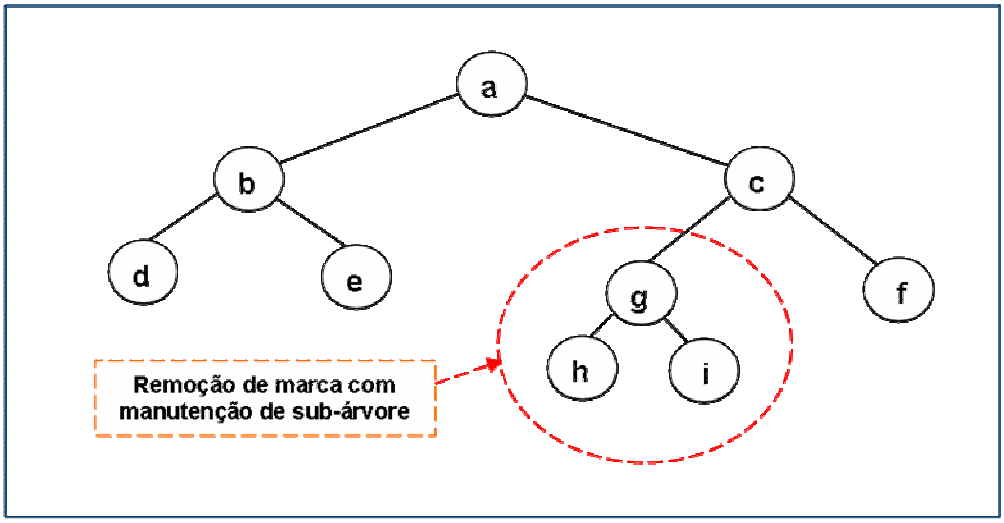


Figura 27 – Árvore resultante da aplicação da primitiva rem-mark

4.3 Primitiva para inserção de código

Ao se escrever uma função adaptativa, pode-se necessitar da inserção de trechos de código à árvore que representa o programa em execução. Para isso, disponibiliza-se na camada adaptativa da linguagem adaptativa a primitiva *ins*.

A primitiva *ins* recebe como primeiro parâmetro o próprio código a ser modificado; como segundo parâmetro, um identificador que endereça o nó interno a esse código, que será alvo da inserção de uma nova sub-árvore, e, por último, a própria sub-árvore a ser inserida.

Assim, diferentemente da primitiva *rem*, a primitiva *ins* tem uma assinatura definida por três parâmetros. A definição da primitiva *ins* pode ser feita da seguinte maneira:

```
(define
        (ins arvore identificador subarvore)
          (busca arvore identificador)
          (insere arvore subarvore)
        arvore
)
```

A primitiva *ins*, também se utiliza de rotinas elementares, tais como as funções *busca* e *insere*.

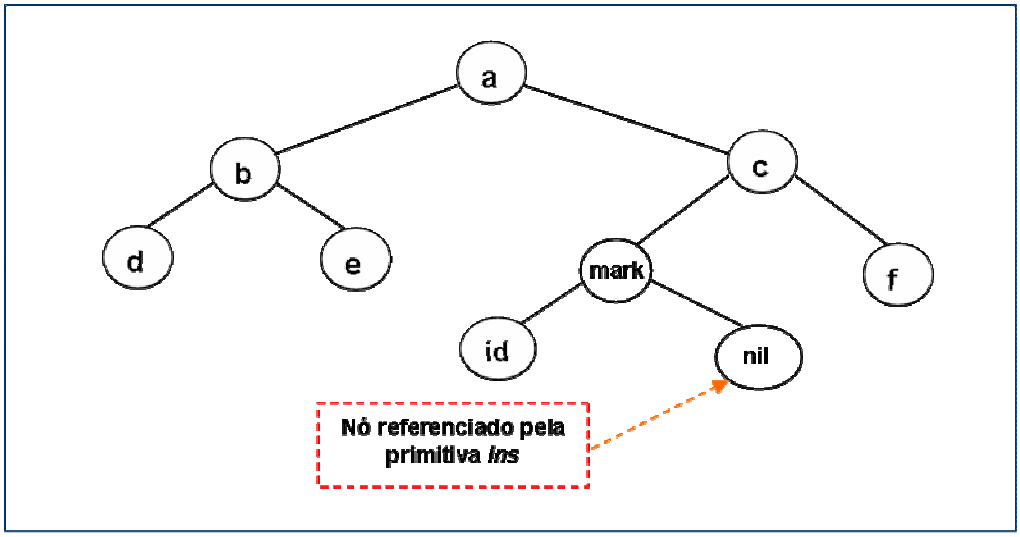


Figura 28 – Árvore com nós referenciados pela primitiva ins

Para ilustrar o emprego da primitiva *ins*, a Figura 28 apresenta a árvore representada pela lista:

```
( a   (b   d   e ) ( c  (mark   id   nil  ) f ) )
```

Na situação esboçada pela Figura 28, a primitiva *mark* endereça um nó (com valor nil) que será a referência da primitiva *ins*.

A Figura 29 esboça a árvore resultante da aplicação da primitiva *ins* com a seguinte chamada:

```
(ins arvore 'id  '( g h i j ) )
```

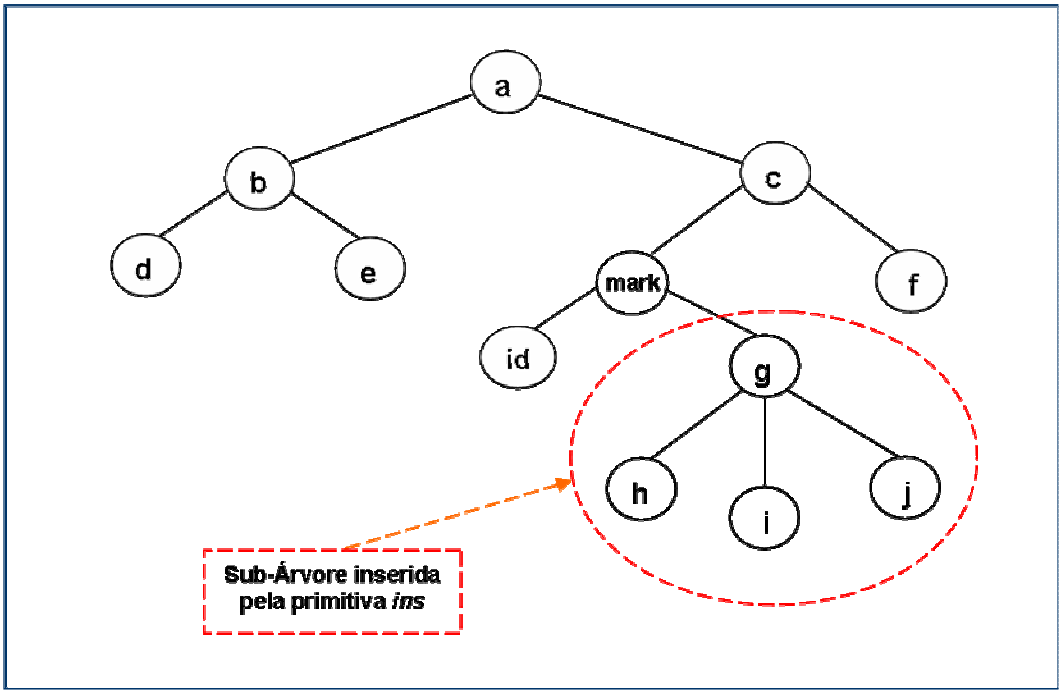


Figura 29 – Árvore resultante após a operação da primitiva ins

4.4 Primitiva para atualização de código

A primitiva *upd* recebe como primeiro parâmetro o trecho de código a ser modificado; como segundo parâmetro, recebe um identificador que endereça o nó que será alvo de atualização da sub-árvore e, por último, o terceiro parâmetro correspondente à nova sub-árvore a ser substituída.

Assim, sua assinatura é definida por três parâmetros: a árvore alvo da auto-modificação, o nó de endereçamento e o novo código definido para a sub-árvore a ser modificada.

A definição da primitiva *upd* foi feita da seguinte maneira:

```
(define
        (upd        arvore  identificador  subarvore)
        (busca      arvore  identificador)
        (atualiza   arvore  subarvore)
        arvore
)
```

A primitiva *upd*, também se utiliza de rotinas elementares, tais como a função busca, e, adicionalmente, também se utiliza da rotina elementar atualiza que terá como

responsabilidade substituir a sub-árvore corrente pela nova sub-árvore passada como parâmetro.

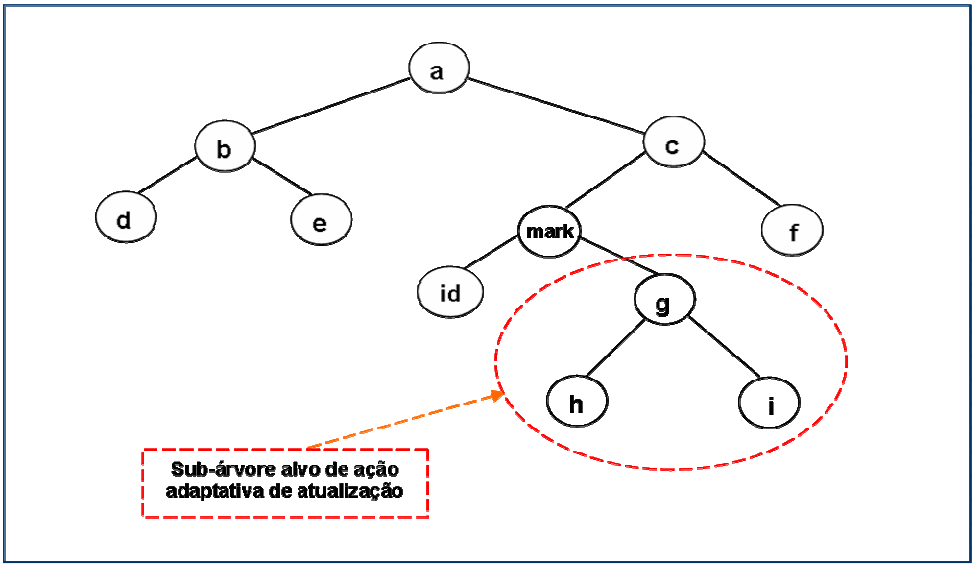


Figura 30 – Árvore com nó referenciado pela primitiva upd

Para ilustrar o emprego da primitiva *upd*, a Figura 30 apresenta uma estrutura em árvore representada pela lista: `(a (b d e) (c (mark id (g h i) f))`

Na situação esboçada pela Figura 30, a primitiva *mark* endereça um nó (que é raiz da sub-árvore corrente) que será a referência da primitiva *upd*.

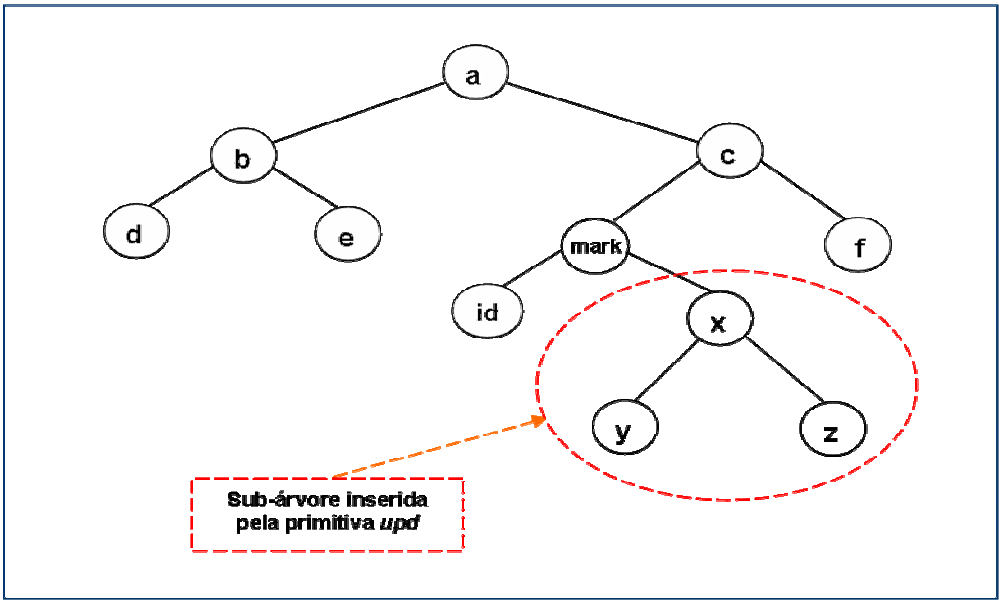


Figura 31 – Árvore resultante após a operação da primitiva upd

A Figura 31 esboça a árvore resultante da aplicação da primitiva *upd* com a seguinte chamada: `(upd arvore 'id '(x y z))`

4.4 Primitiva para consulta

A primitiva *cons* recebe como parâmetro um identificador que identifica o nó correspondente à sub-árvore a ser consultada e a retorna. Assim, sua assinatura é definida por apenas um parâmetro. Sua definição foi feita da seguinte maneira:

```
(define
        (cons       identificador  subarvore)
        (busca      identificador)
        subarvore
)
```

4.5 Primitiva para geração automática de nomes de marcas

Conforme apresentado no início deste capítulo, a primitiva *mark* recebe como primeiro parâmetro um identificador que vincula um nome à porção de código que se deseja tornar alvo da adaptatividade. Este identificador é de livre escolha do programador durante a escrita das funções adaptativas.

No entanto, pode ser conveniente ao programador o emprego de um procedimento automático para a geração de marcas às porções automodificáveis de código. Para tanto, disponibiliza-se na camada adaptativa da linguagem, uma primitiva denominada *next* que terá como responsabilidade gerar marcas (identificadores) a partir de um determinado valor inicial. Assume-se nessa tese que a marca inicial gerada pela primitiva *next* é a cadeia "1". À medida em que a primitiva for sendo executada, as marcas subseqüentes serão representadas pelas cadeias "2", "3", e assim por diante. Sua definição foi feita da seguinte maneira:

```
(define
        (next   w )
        (set    w       Numero_Serie)
        (+      Numero_Serie   1 )
        Numero_Serie
)
```

A primitiva *next* é usualmente empregada em conjunto com a primitiva mark, como por exemplo:

```
(mark  (next  x ) ( c f g ) )
```

4.6 Exemplo ilustrativo das primitivas da camada adaptativa

Nesse item, para ilustrar o emprego das primitivas da camada adaptativa apresenta-se um simples programa que apresenta uma decisão que se baseia na chamada de uma função adaptativa. No exemplo, caso os parâmetros passados ao programa sejam diferentes o programa executa uma função adaptativa que, em tempo de execução, acrescenta um código alternativo para a decisão. O exemplo mostra assim a situação de um "*if adaptativo*".

A Figura 32 apresenta o programa em *Lisp* e a Figura 33 o código da função adaptativa.

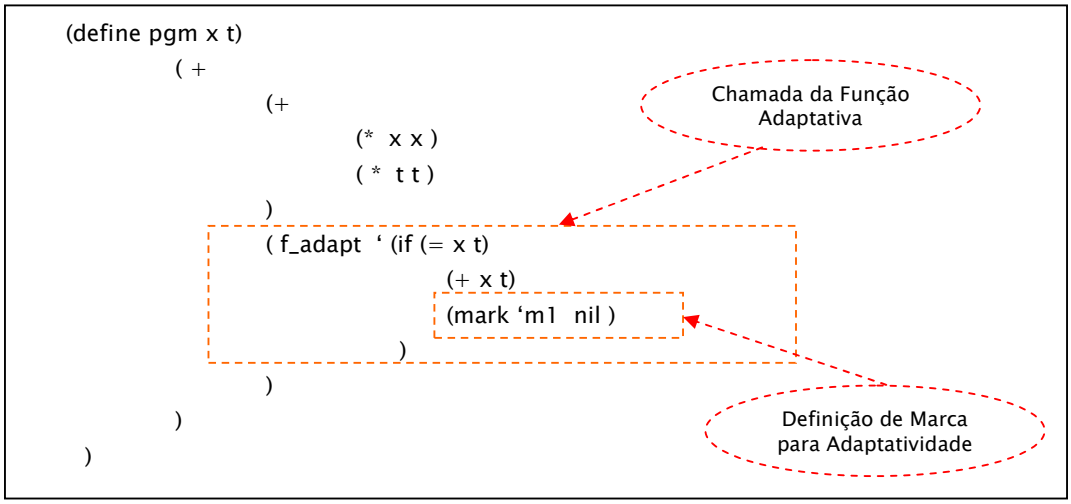


Figura 32 – Exemplo de Um programa Adaptativo

```
(define f_adapt  arvore)
                 ( upd  arvore 'm1  '(+ 2 x)
                 ( eval arvore)
 )
```

Figura 33 – Exemplo de uma Função Adaptativa com ação elementar de update

As Figuras 34 e 35 esboçam as árvores correspondentes à execução do programa da Figura 32 antes e após a aplicação da função adaptativa apresentada na Figura 33.

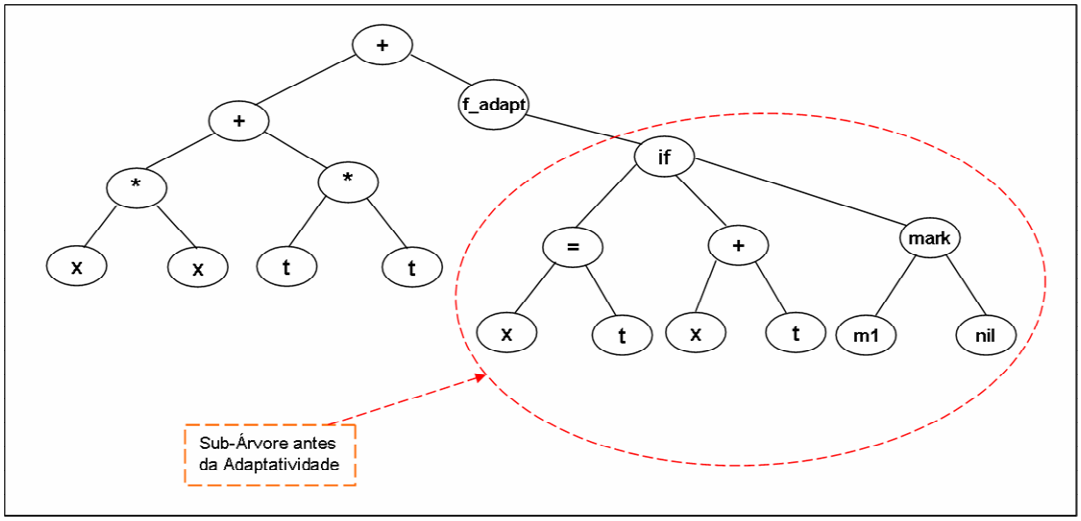


Figura 34 – Árvore do Programa antes da execução da Função Adaptativa

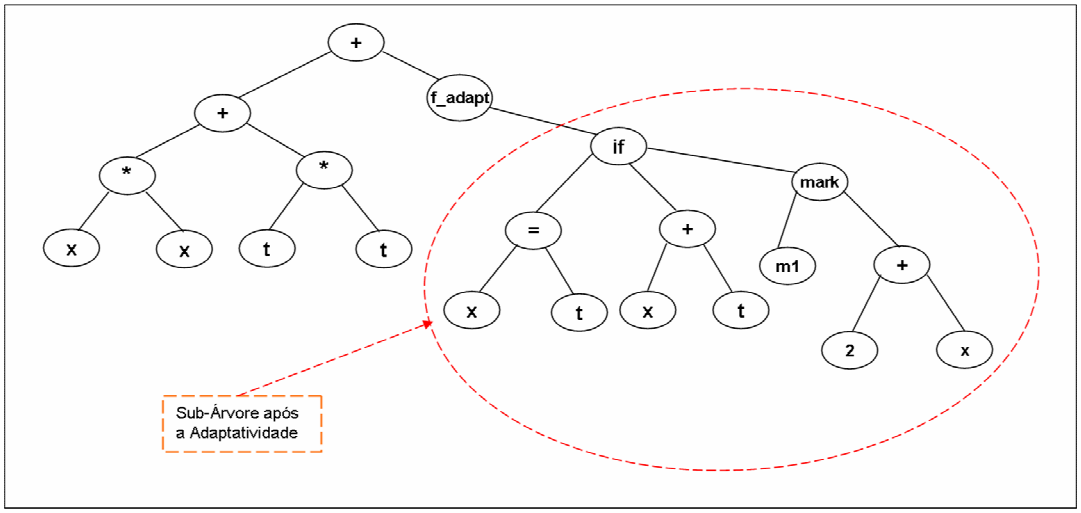


Figura 35 – Árvore do Programa após a execução da Função Adaptativa

5. Exemplo de aplicação

Neste capítulo, apresenta-se o desenvolvimento de uma aplicação escrita com a linguagem adaptativa proposta nesta tese. A aplicação consiste em uma versão eletrônica do popular jogo de dominó, porém com a incorporação de adaptatividade em suas regras e em sua implementação.

5.1 Um jogo de Dominó Adaptativo

A Figura 36 apresenta o conjunto de 28 peças do dominó adaptativo antes do sorteio entre os jogadores.

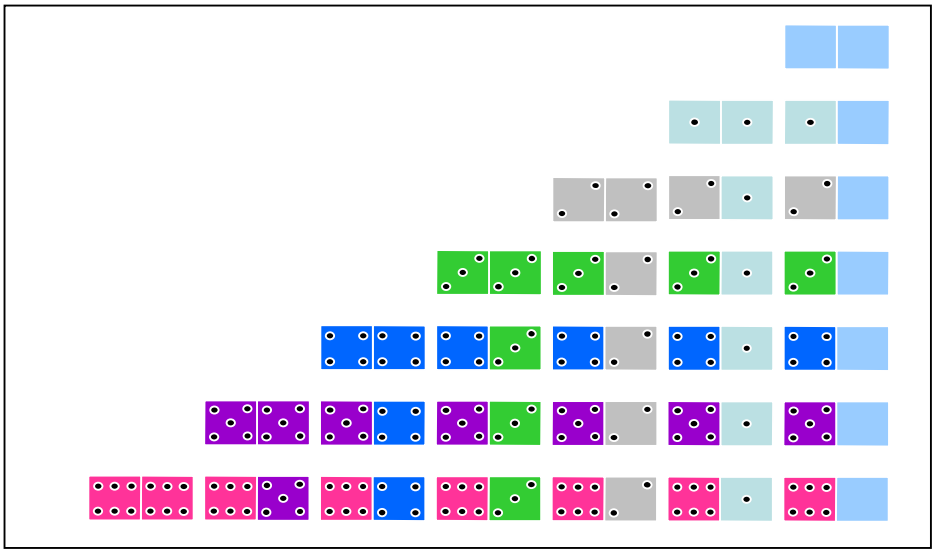


Figura 36 – Conjunto das 28 peças do Dominó Adaptativo

Embora o jogo de dominó possa ser utilizado por dois ou três jogadores, usualmente uma partida de dominó é jogada por quatro jogadores. A Figura 37 apresenta uma possível seqüência de peças distribuída entre os quatro jogadores.

O dominó adaptativo segue as regras usuais do dominó tradicional, substituindo-se as regras de anexação de peças às extremidades da mesa por um conjunto de regras adaptativas, que especificam como determinar as peças a serem incorporadas. Inicialmente tal conjunto é o mesmo utilizado no dominó tradicional, mas no decorrer do jogo ele sofre alterações.

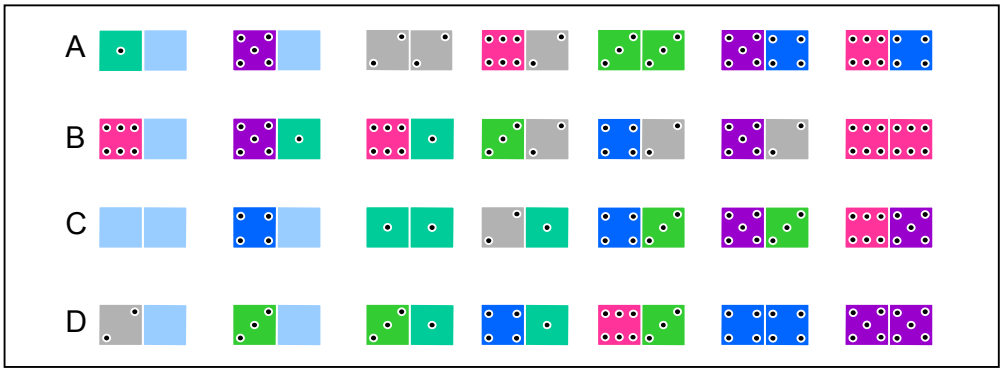


Figura 37 – Uma possível distribuição das peças entre os quatro jogadores

Para a distribuição apresentada na Figura 37, os quatro jogadores terão os seguintes conjuntos de peças:

A={ (1,0), (5,0), (2,2), (6,2), (3,3), (5,4), (6,4) }

B={ (6,0), (5,1), (6,1), (3,2), (4,2), (5,2), (6,6) }

C={ (0,0), (4,0), (1,1), (2,1), (4,3), (5,3), (6,5) }

D={ (2,0), (3,0), (3,1), (4,1), (6,3), (4,4), (5,5) }

O jogo se inicia com o jogador que tiver a peça (6,6), o que pela Figura 37 corresponde ao jogador B.

Indicaremos por J o conjunto de regras do jogo. Inicialmente este conjunto tem a seguinte configuração:

J_0 = { (0,0,–),(1,1,–),(2,2,–),(3,3,–),(4,4,–),(5,5,–),(6,6,*) }

No conjunto J acima descrito, emprega-se a seguinte notação: Uma regra qualquer é representada pela terno (M,N,C), sendo M o primeiro número que indica a extremidade na mesa, N é o segundo número que representa o número a ele anexável e C pode ter como valores o caractere '-' que indica que essa regra é não-adaptativa, ou então o caractere '*' que caracteriza uma regra adaptativa.

A título de ilustração, seja J_i o conjunto correspondente à configuração em um dado instante i, durante uma partida.

J_i = {(0,0,–),(1,1,–),(2,3,–),(3,6,–),(4,2,–),(5,5,–),(6,4,*) }

Na configuração acima, as duas primeiras e a última regra continuam correspondendo às do dominó tradicional, porém a regra (2,3,-) indica que a ocorrência de um 2 na mesa aceita a anexação de uma peça contendo um 3. Situações similares ocorrem com as regras (3,6,-) , (4,2,-) e (6,4,*).

Se um jogador usar uma regra marcada com *, ganha o direito de alterar J (embora isso seja facultativo), ou permutando o segundo número entre duas regras, ou deslocando o * para outra regra. No próximo item, apresenta-se um exemplo, passo a passo, de alguns momentos de uma partida.

5.2 Simulação de uma partida

Neste item, apresenta-se passo a passo, uma simulação dos primeiros lances de uma partida.

A Figura 38 esboça a configuração inicial da partida que utiliza a configuração apresentada na Figura 37.

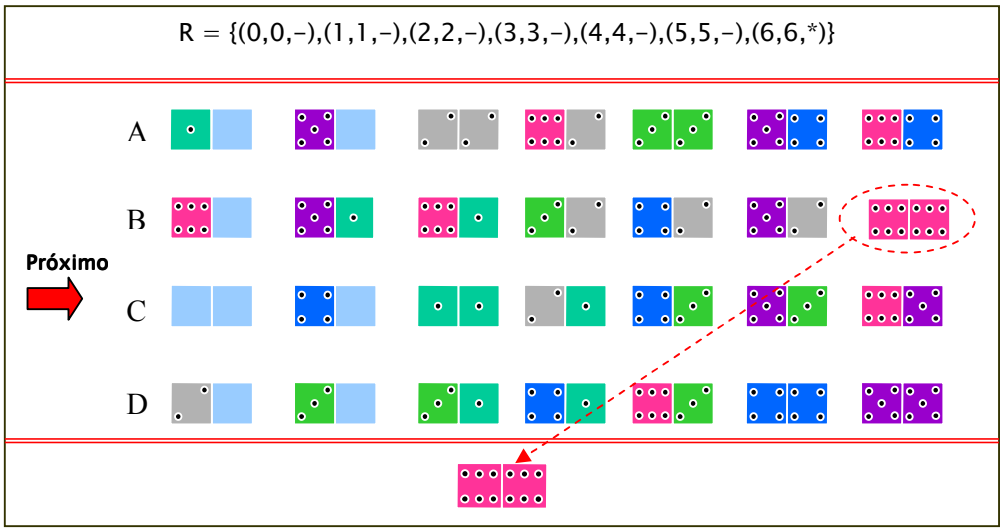


Figura 38 – Uma possível distribuição das peças entre os quatro jogadores

Para a partida esboçada na Figura 38, os quatro jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (2,2), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (6,1), (3,2), (4,2), (5,2), (6,6) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3), (5,3), (6,5) }

D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4), (5,5) }

O conjunto R de regras adaptativas é do dominó tradicional. No entanto, cabe ressaltar que a última regra (marcada com '*') é adaptativa.

R = { (0,0,–),(1,1,–),(2,2,–),(3,3,–),(4,4,–),(5,5,–),(6,6,*) }

Na configuração inicial, a peça (6,6) é colocada na mesa de jogo e o próximo jogador a efetuar o lance é C.

No Lance-1 da partida, o jogador C tem "6" na mesa e, assim, a única peça possível é (6,5), correspondendo à regra (6,6,*) que é adaptativa. De acordo com a convenção, o primeiro número da trinca corresponde à peça na mesa, o segundo número a peça do jogador a ser substituída e o caractere '*' indica que a regra é adaptativa. Admite-se, neste lance, que o jogador C escolheu mudar o asterisco entre as duas últimas regras de R.

A Figura- 39 esboça a configuração após o Lance-1 da partida.

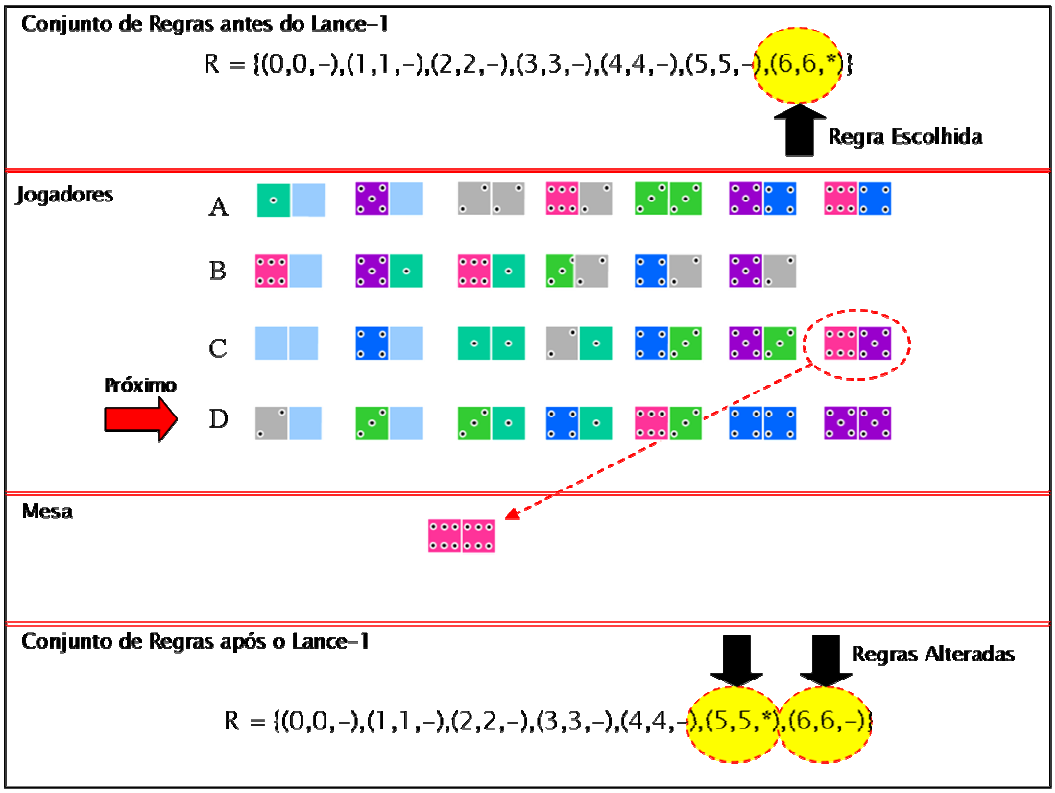


Figura 39 – Configuração após o Lance-1

Após o Lance-1, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (2,2), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (6,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3), (5,3) }

D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4), (5,5) }

Após o Lance-1, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,2,–),(3,3,–),(4,4,–),(5,5,*),(6,6,–) }

No Lance-2 da partida, o jogador D tem "6" e "5" na mesa e, assim, as peças possíveis são (6,3) e (5,5) que correspondem às regras (6,6,–) e (5,5,*). Admite-se que o jogador D escolhe usar a sua peça (5,5) e aplicar a regra (5,5,*) que é adaptativa. O jogador D vai alterar as regras (2,2,–) para (2,6,–) e (6,6,–) para (6,2,–).

Após o Lance-2, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (2,2), (6,2), (3,3), (5,4), (6,4) }
B = { (6,0), (5,1), (6,1), (3,2), (4,2), (5,2) }
C = { (0,0), (4,0), (1,1), (2,1), (4,3), (5,3) }
D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4) }

Após o Lance-2, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,–),(4,4,–),(5,5,*),(6,2,–) }

A Figura 40 esboça a configuração da partida após o Lance-2.

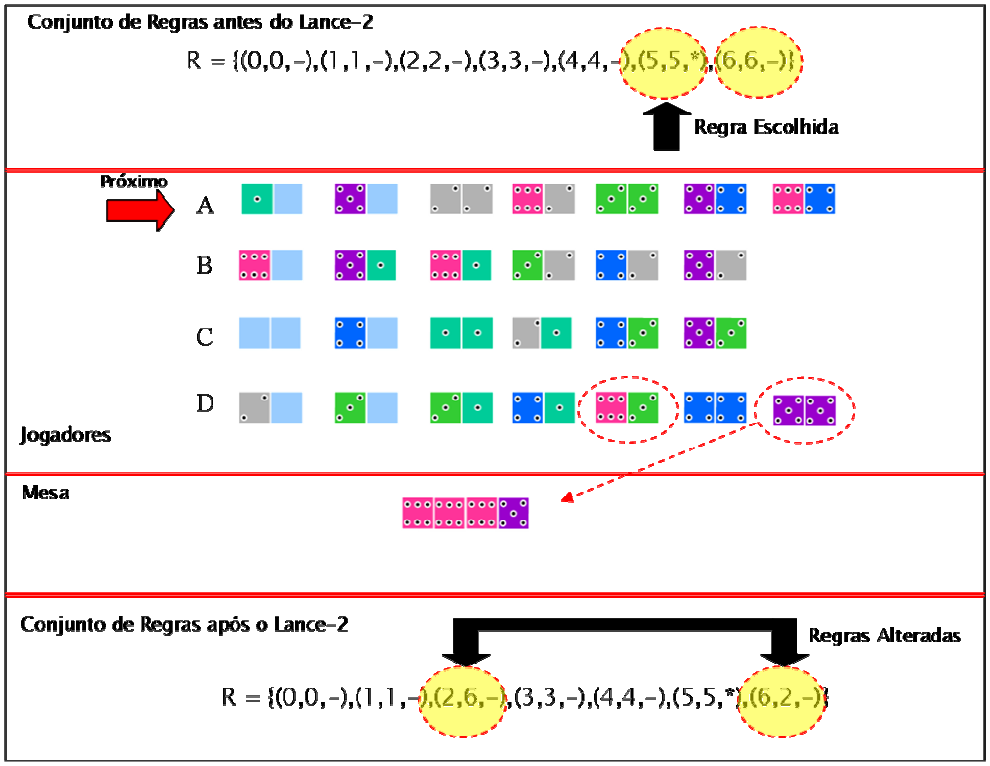


Figura 40 – Configuração após o Lance-2

No Lance-3 da partida, o jogador A tem "6" e "5" na mesa e, assim, as peças possíveis são (2,2), (6,2) com o emprego da regra (6,2,–) ou (5,0), (5,4) com o emprego da regra (5,5,*). Admite-se que o jogador A escolhe usar a sua peça (2,2) e

aplicar a regra (6,2,–) que não é adaptativa. Portanto, o conjunto de regras nesse lance não é alterado.

Após o Lance-3, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (6,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3), (5,3) }

D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4) }

Após o Lance-3, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,–),(4,4,–),(5,5,*),(6,2,–) }

A Figura 41 esboça a configuração da partida após o Lance-3.

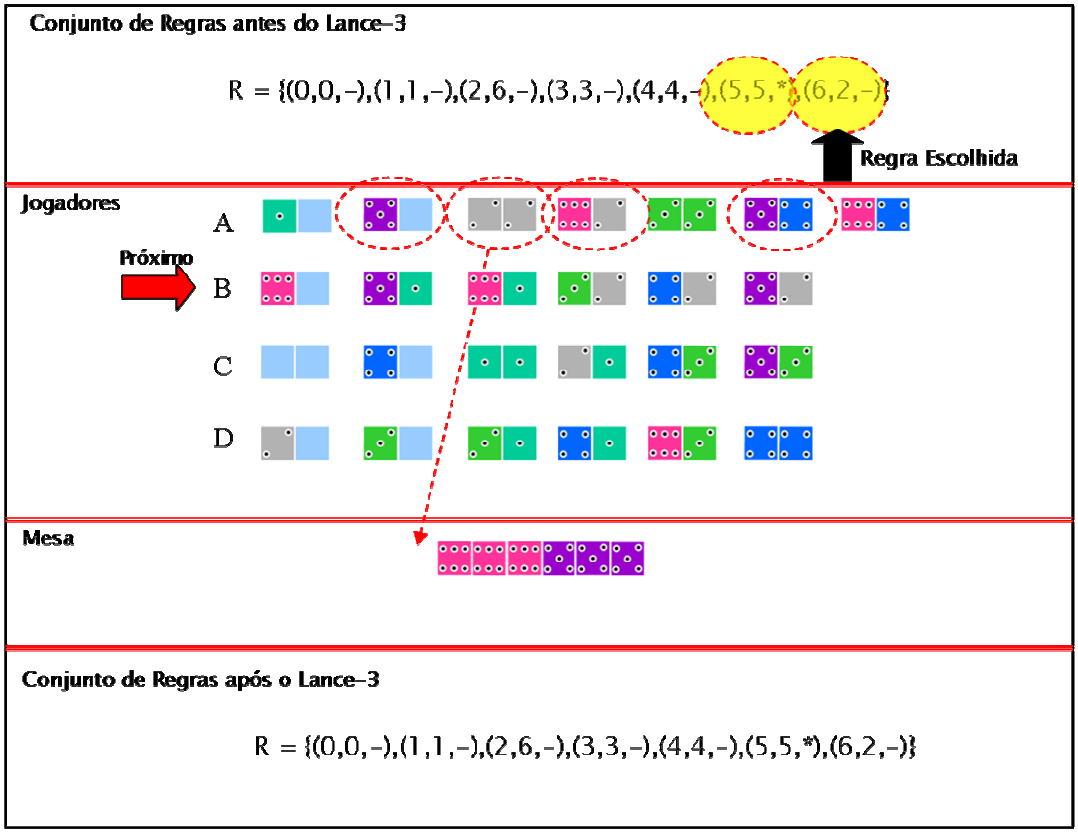


Figura 41 – Configuração após o Lance-3

No Lance-4 da partida, o jogador B tem "2" e "5" na mesa e, assim, as peças possíveis são (6,0), (6,1) com o emprego da regra (2,6,–) ou (5,1), (5,2) com o emprego da regra (5,5,*). Admite-se que o jogador B escolhe usar a sua peça (6,1) e aplicar a regra (2,6,–) que não é adaptativa. Portanto, o conjunto de regras nesse lance não é alterado.

Após o Lance-4, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3), (5,3) }

D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4) }

Após o Lance-4, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,–),(4,4,–),(5,5,*),(6,2,–) }

A Figura 42 esboça a configuração da partida após o Lance-4.

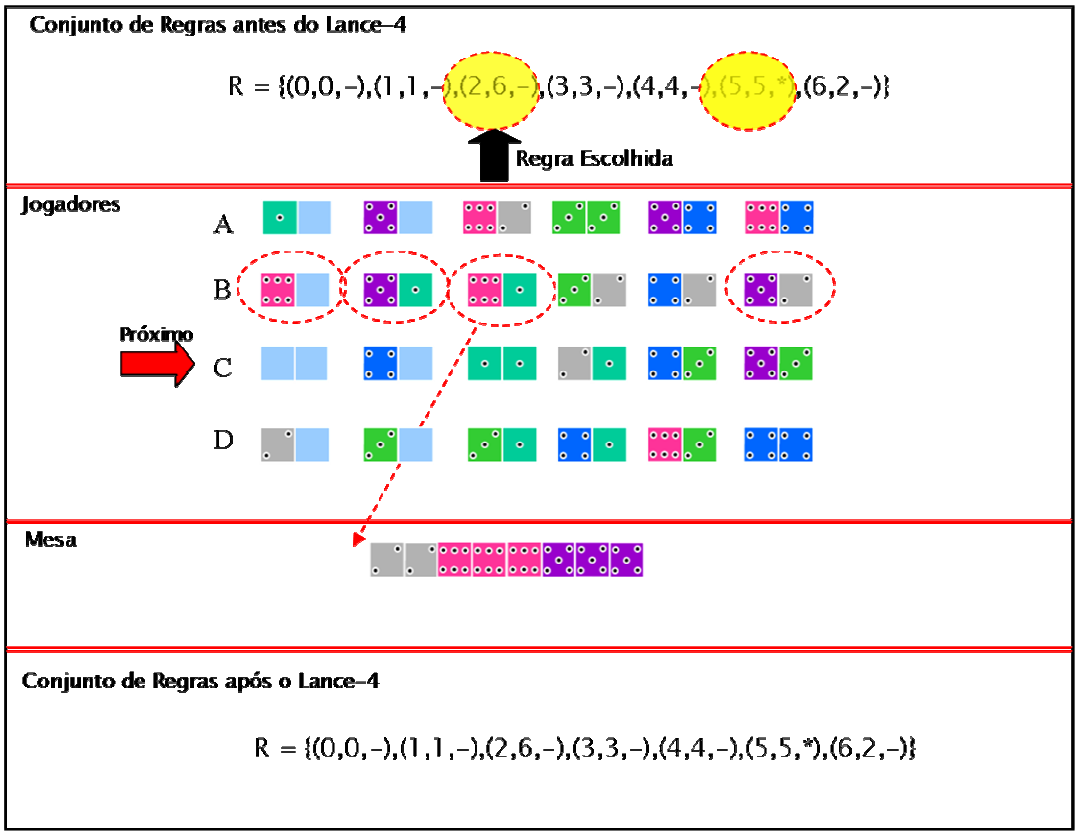


Figura 42 – Configuração após o Lance-4

No Lance-5 da partida, o jogador C tem "1" e "5" na mesa e, assim, as peças possíveis são (1,1), (2,1) com o emprego da regra (1,1,–) ou (5,3) com o emprego da regra (5,5,*). Admite-se que o jogador C escolhe usar a sua peça (5,3) e aplica a regra (5,5,*) que é adaptativa. O jogador C vai alterar as regras (3,3,–) para (3,3,*) e (5,5,*) para (5,5,–).

Após o Lance-5, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3) }

D = { (2,0), (3,0), (3,1), (4,1), (6,3), (4,4) }

Após o Lance-5, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,*),(4,4,–),(5,5,–),(6,2,–) }

A Figura 43 esboça a configuração da partida após o Lance-5.

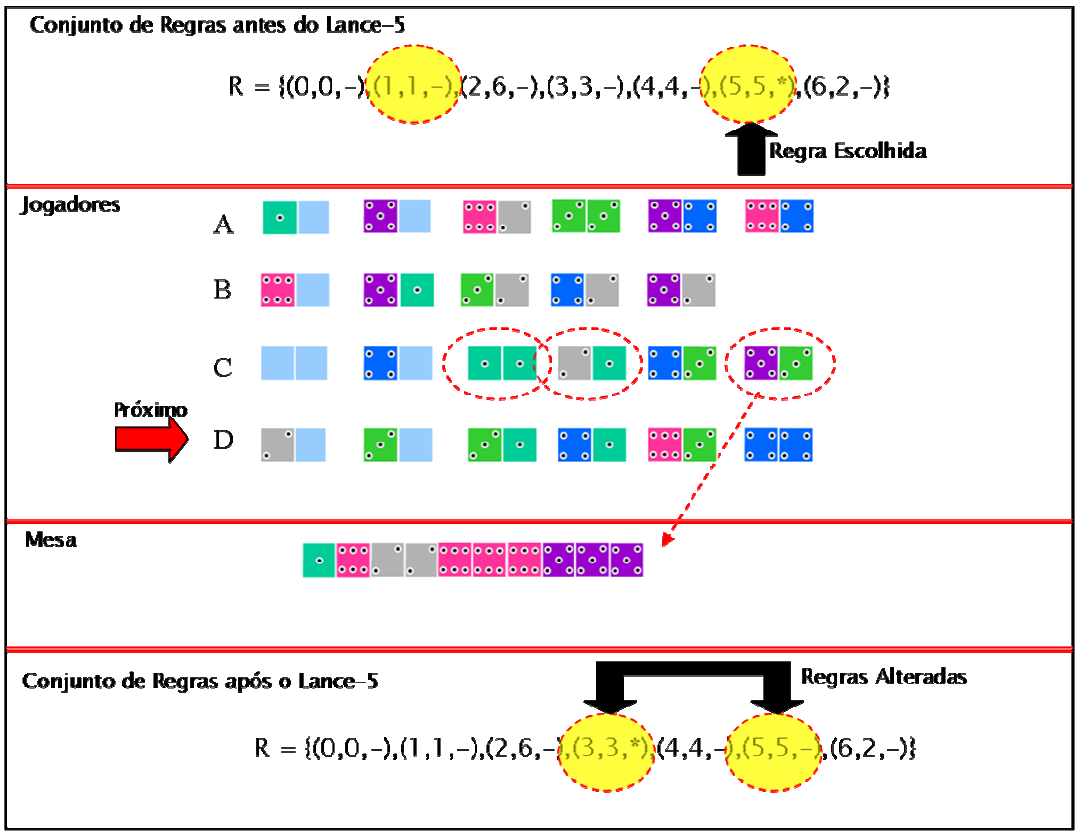


Figura 43 – Configuração após o Lance-5

No Lance-6 da partida, o jogador D tem "1" e "3" na mesa e, assim, as peças possíveis são (3,1), (4,1) com o emprego da regra (1,1,–) ou (3,0),(3,1) com o emprego da regra (3,3,*). Admite-se que o jogador D escolhe usar a sua peça (3,1) e aplica a regra (3,3,*) que é adaptativa. O jogador D decidiu não alterar as regras de adaptatividade.

Após o Lance-6, os jogadores estão com os seguintes conjuntos de peças:

A = { (1,0), (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3) }

D = { (2,0), (3,0), (4,1), (6,3), (4,4) }

Após o Lance-6, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,*),(4,4,–),(5,5,–),(6,2,–) }

A Figura 44 esboça a configuração da partida após o Lance-6.

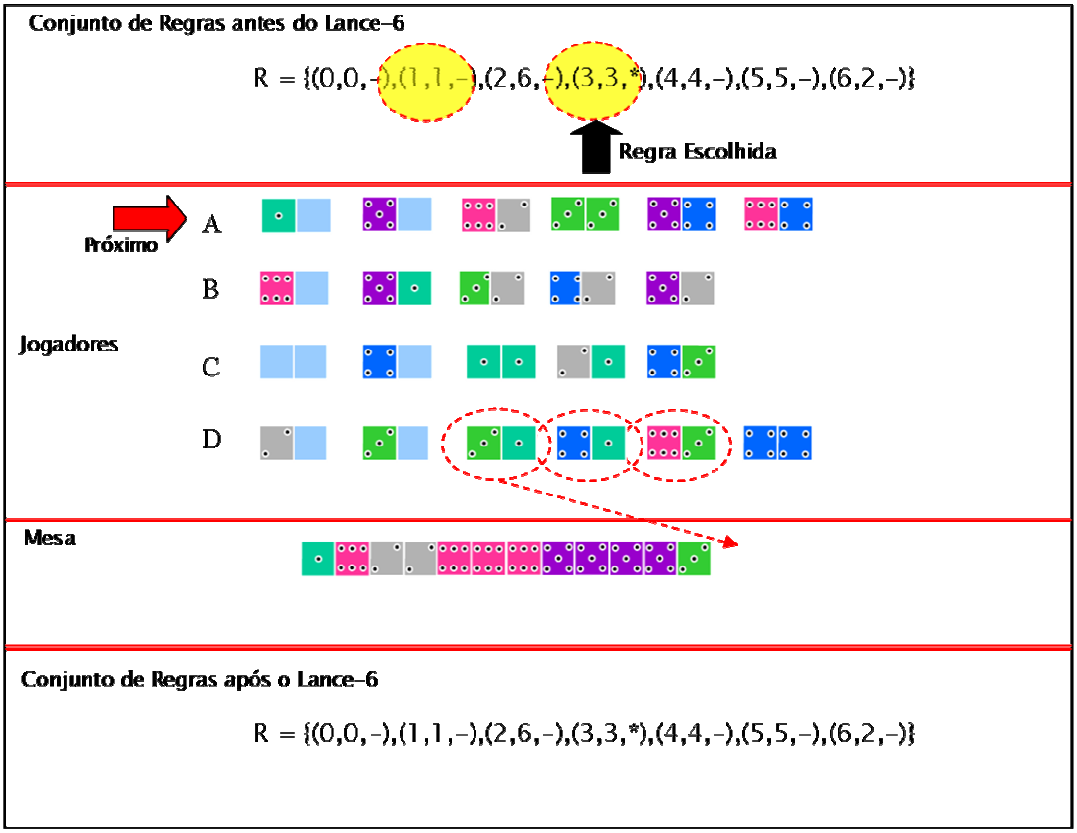


Figura 44 – Configuração após o Lance-6

No Lance-7 da partida, o jogador A tem "1" e "1" na mesa e, assim, a única peça possível é (1,0) com o emprego da regra (1,1,–). Sendo esta regra não-adaptativa, o conjunto de regras não se altera.

Após o Lance-7 os jogadores estão com os seguintes conjuntos de peças:

A = { (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (6,0), (5,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3) }

D = { (2,0), (3,0), (4,1), (6,3), (4,4) }

Após o Lance-7, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,*),(4,4,–),(5,5,–),(6,2,–) }

A Figura 45 esboça a configuração da partida após o Lance-7.

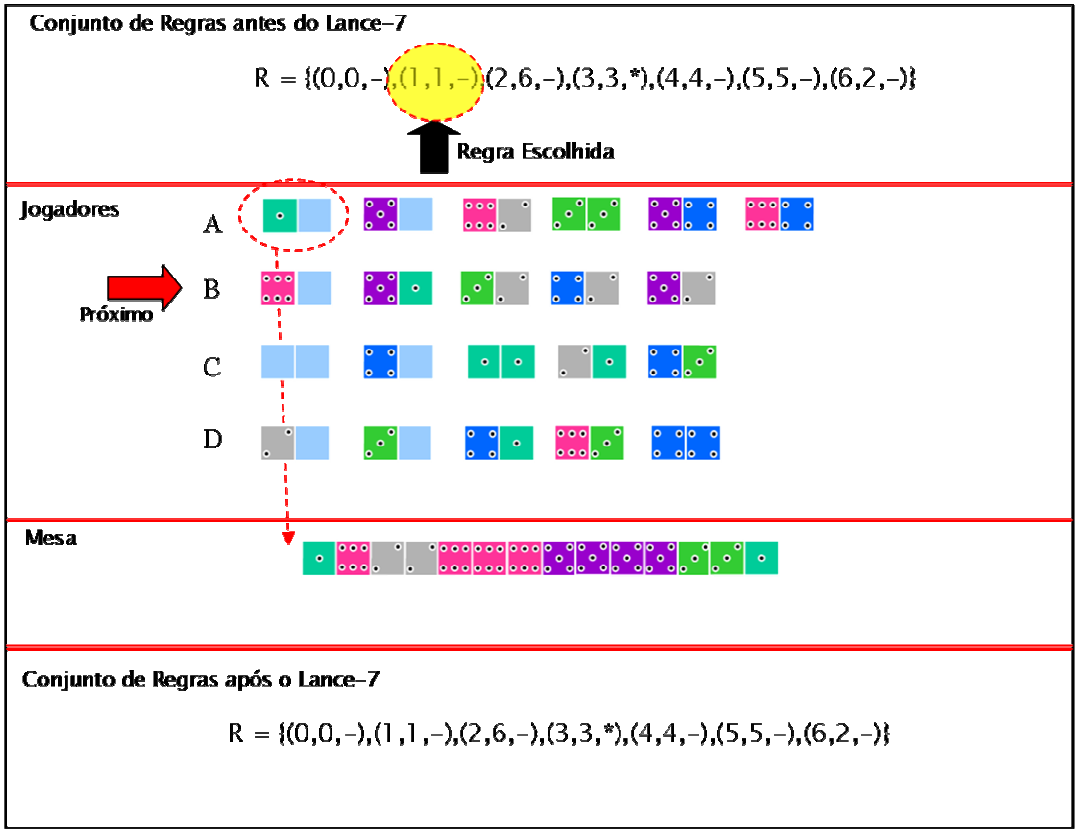


Figura 45 – Configuração após o Lance-7

No Lance-8 da partida, o jogador B tem "0" e "1" na mesa e, assim, as peças possíveis são (6,0) com o emprego da regra (0,0,–) ou (5,1) com o emprego da regra (1,1,–). Admite-se que o jogador B escolhe usar a sua peça (6,0) e aplica a regra (0,0,–). Sendo esta regra não-adaptativa, o conjunto de regras não se altera.

Após o Lance-8 os jogadores estão com os seguintes conjuntos de peças:

A = { (5,0), (6,2), (3,3), (5,4), (6,4) }

B = { (5,1), (3,2), (4,2), (5,2) }

C = { (0,0), (4,0), (1,1), (2,1), (4,3) }

D = { (2,0), (3,0), (4,1), (6,3), (4,4) }

Após o Lance-8, o conjunto de regras é definido por:

R = { (0,0,–),(1,1,–),(2,6,–),(3,3,*),(4,4,–),(5,5,–),(6,2,–) }

A Figura 46 esboça a configuração da partida após o Lance-8.

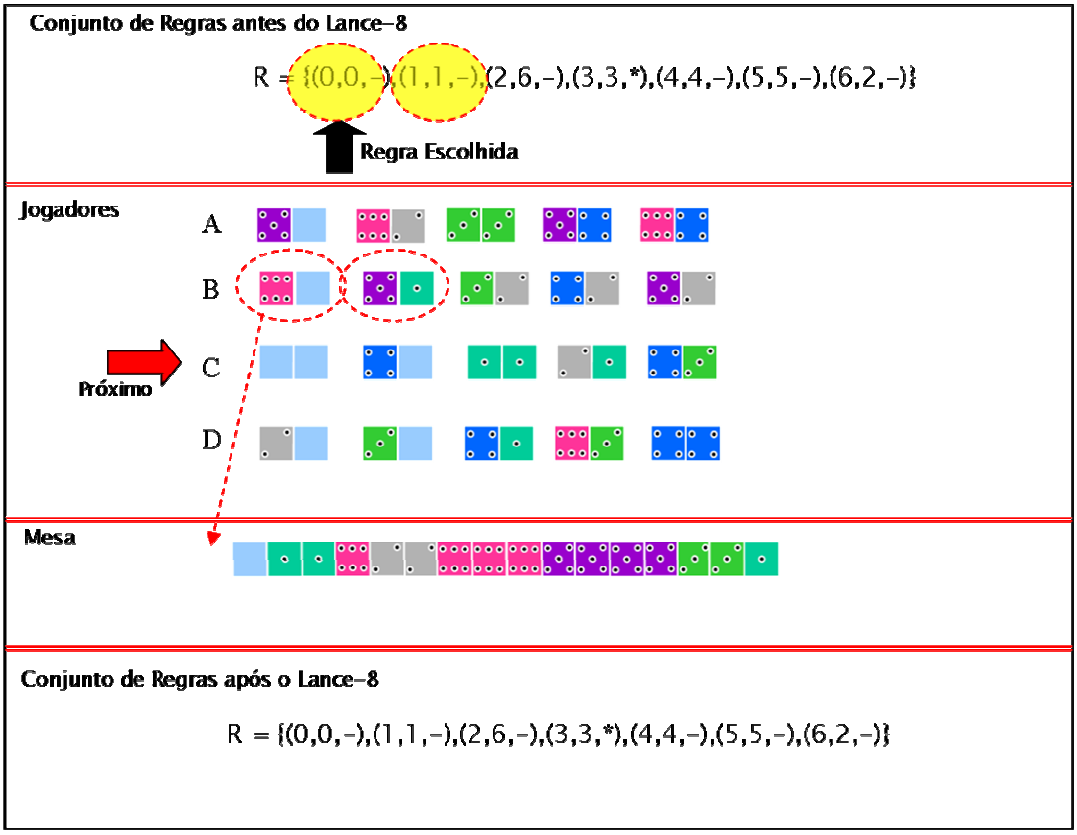


Figura 46 – Configuração após o Lance-8

A Figura 47 apresenta o pseudo-código da aplicação.

```
Inicialização
Se Usuário tiver alguma jogada
        Processamento de Alteração de Regras caso o usuário assim o desejar
        Processamento da jogada do usuário
        Se for última peça, usuário ganha a partida
        Fim-Se
Caso contrário
        Se empate
                Cálculo da Somatória de Pontos
                Envio de mensagem para ganhador
        Fim-Se
Fim-Se
Se Máquina tiver jogada
        Processamento de Alteração de Regras caso o usuário assim o desejar
        Processamento da jogada da máquina
        Se for última peça, máquina ganha a partida
        Fim-se
Caso contrário
        Se empate
                Cálculo da Somatória de Pontos
                Envio de mensagem para ganhador
        Fim-Se
Fim-Se
```

Figura 47 – Dominó Adaptativo - Pseudocódigo

5.3 Observações sobre estilos de implementação do jogo

Quando se emprega o estilo imperativo de programação, todas as estruturas de dados podem ser definidas por meio de variáveis que representam listas e são referenciadas explicitamente pelas diversas funções do programa. Os algoritmos empregados nesta versão correspondem a blocos de iteração, típicos da programação imperativa. Assim, o conjunto de regras da aplicação pode ser explicitamente expresso pela seguinte lista:

```
'(      (0 0 -)                 ;Regra 0
        (1 1 -)                 ;Regra 1
        (2 2 -)                 ;Regra 2
        (3 3 -)                 ;Regra 3
        (4 4 -)                 ;Regra 4
        (5 5 -)                 ;Regra 5
        (6 6 *)                 ;Regra 6
)
```

Da mesma forma, o conjunto de peças do jogo, do usuário e da máquina podem ser também representados por meio de estruturas de dados do tipo lista, conforme abaixo representado:

```
'(      (0 0)                   ;Peça 0
        (0 1)                   ;Peça 1
        (0 2)                   ;Peça 2
        ...
        (6 6)                   ;Peça 27
)
```

Nesta alternativa de implementação, embora se esteja utilizando uma linguagem funcional, a solução é tipicamente imperativa, uma vez que todas as estruturas de dados são alteradas por meio de efeitos colaterais decorrentes das sucessivas aplicações de funções escritas no código com tratamento iterativo.

Dessa forma, a adaptatividade resultante da alteração do conjunto de regras pode ser feita de maneira trivial, uma vez que os dados desta estrutura são acessados diretamente pelas diversas funções do programa, sem, portanto, haver necessidade da chamada de funções da camada adaptativa.

Diferentemente do estilo imperativo, ao se empregar o estilo funcional não há declarações explícitas de estruturas de dados e os algoritmos das funções que compõem o programa são recursivos. Nesse estilo, o programa trabalha exclusivamente com funções que recebem os dados via passagem de parâmetros, efetuam alterações nesses dados e os retornam para as funções que as chamaram. A seguir, apresenta-se uma possível chamada da função inicial do programa.

```
( domino  '(
                    ()          ;lista do usuário
                    ()          ;lista da máquina
                    ( (6 6)     ;lista da mesa
                    )
                    (
                              Lista de regras da aplicação
                              (0 0 -)    ;Regra 0
                              (1 1 -)              ;Regra 1
                              (2 2 -)              ;Regra 2
                              (3 3 -)              ;Regra 3
                              (4 4 -)              ;Regra 4
                              (5 5 -)              ;Regra 5
                              (6 6 *)              ;Regra 6
                    )
                    ()          ;lista total de jogadas
                    ()          ;lista jogada escolhida
                    (           ; lista de peças do jogo
                              (0 0)(0 1)(0 2)(0 3)(0 4)(0 6)(1 1)(1 2)(1 3)(1 4)(1 5)
                              (1 6)(2 2)(2 3)(2 4)(2 5)(2 6)(3 3)(3 4)(3 5)(3 6)(4 4)
                              (4 5)(4 6)(5 5)(5 6)
                    )
                    ()
                    ()
                    ()
                    i 6 6 6 6 n 0 0
          )
)
```

Nesta alternativa de implementação a adaptatividade resultante da alteração do conjunto de regras pode também ser feita por meio de código não-adaptativo, uma vez que os dados desta estrutura são acessados através de parâmetros que são passados às diversas funções do programa, sem, portanto, haver necessidade da chamada de funções da camada adaptativa.

Ao se empregar o estilo adaptativo para a aplicação em questão, pode-se empregar uma implementação semelhante à apresentada no estilo funcional, sem haver, portanto, declarações explícitas de estruturas de dados. Analogamente, os algoritmos das funções que compõem o programa são recursivos.

No entanto, na versão adaptativa o conjunto de regras pode não ser definido por meio de estruturas de dados em lista, e sim por uma função especificamente construída para se responsabilizar pelo mapeamento das regras da aplicação.

Com isso, na versão adaptativa, para que se obtenha a modificação do conjunto de regras da aplicação, e conseqüente adaptatividade, torna-se indispensável o emprego de um programa adaptativo capaz de alterar o código correspondente a essas regras, em tempo de execução.

Se considerarmos a não existência de adaptatividade na aplicação, pode-se substituir a estrutura de dados correspondente ao conjunto de regras, pela seguinte função:

```
(define (regras P)
  (if (= P 0 ) '(0 -)
     (if (= P 1 ) '(1 -)
       . . .
)
```

No caso da função regras acima esboçada, o parâmetro passado à função *define*, dentre as várias alternativas, qual a regra a ser considerada. Por exemplo, se o parâmetro tiver o valor ‘1’, a função regras retorna o valor ‘(1 -) que corresponde a uma regra não-adaptativa associando um ‘1’ da mesa com um ‘1’ existente em alguma peça do jogador.

A Figura 48 exibe, na forma de árvore, o conjunto inicial de regras da aplicação.

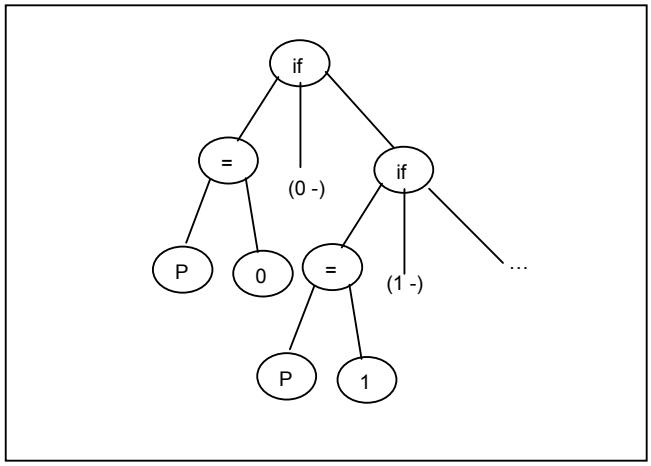


Figura 48 – Estrutura do conjunto de regras sem considerar adaptatividade

Tendo em vista que a aplicação em questão é adaptativa, o conjunto de regras pode variar em tempo de execução e, para tanto, necessita-se de incluir no código acima as primitivas aqui desenvolvidas. A seguir mostra-se o código anterior com as primitivas da linguagem adaptativa.

```
(define (init)
(context 'ADAPT )
(set 'ADAPT:PGM
   '( if (= P 0 ) (mark 'm0 '(0 -) )
      (if (= P 1 ) (mark 'm1 '(1 -) )
      … )
```

Pode-se perceber no código acima, que a estrutura correspondente ao conjunto de regras é inicialmente salva no contexto do ambiente adaptativo (materializado pela função *context*).

Assim, a área PGM na camada adaptativa contém o código inicial da aplicação que será, em tempo de execução, modificado pela função adaptativa projetada para esse fim.

A Figura 49 exibe, na forma de árvore, o conjunto inicial de regras da aplicação com a inclusão das primitivas de endereçamento de código da linguagem adaptativa.

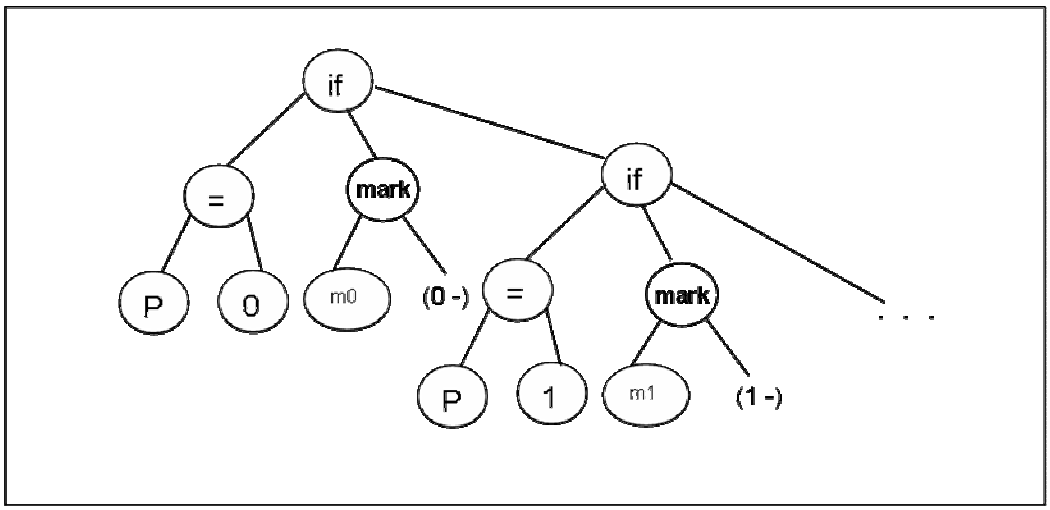


Figura 49 – Estrutura do conjunto de regras considerando primitivas de adaptatividade

Nessa situação, uma possível implementação do conjunto de regras da aplicação pode ser feita por meio do código:

```
(define (f_adapt P NOVAREGRA arvore)
        (println "Executando função adaptativa...")
        (if (= P 0 ) (upd arvore 'm0 NOVAREGRA) )
         (if (= P 1 ) (upd arvore 'm1 NOVAREGRA) )
          . . .
        (eval arvore)
)
```

Na implementação empregada para a função adaptativa, destaca-se o emprego da primitiva *upd* que recebe como parâmetros: o elemento que define a regra a ser modificada, a nova regra a ser considerada e a árvore correspondente ao código alvo da adaptatividade. A função adaptativa efetiva a alteração na sub-árvore correspondente por meio do endereçamento providenciado pela primitiva *mark*.

Com base nos ensaios experimentados nesta aplicação, apresenta-se a seguir algumas diretrizes para o desenvolvimento das linguagens adaptativas de programação e as considerações finais deste trabalho.

6. Considerações Finais

Neste capítulo descrevem-se as diretrizes para o desenvolvimento de linguagens adaptativas e as contribuições advindas desta pesquisa, destacando-se os resultados obtidos e sua aplicação como subsídio para prosseguimento da pesquisa.

6.1 Diretrizes para o desenvolvimento de Linguagens Adaptativas

Com base nas experimentações e estudos desenvolvidos nesta tese, pode-se apresentar um conjunto de diretrizes que permitem definir os requisitos de uma linguagem de programação adaptativa.

Em caso de a linguagem adaptativa ter como base uma linguagem não-adaptativa e um mecanismo de extensão, pode-se citar como requisito de funcionalidade que a linguagem adaptativa deve ter condições de efetuar todas as operações presentes na correspondente linguagem-base não-adaptativa. Essa característica é obtida por meio de algum mecanismo de extensão presente na linguagem-base.

Outro requisito de funcionalidade de uma linguagem adaptativa, nativa ou baseada em linguagem não-adaptativa, é a existência de algum recurso que permita, de forma dinâmica, referenciar e modificar partes do próprio programa em execução.

Nesta tese, a forma de endereçamento, utilizada para se referenciar e modificar partes do programa se deu por meio do desenvolvimento e disponibilização de uma primitiva de rotulação presente na camada adaptativa.

No entanto, em uma linguagem genuinamente adaptativa tal recurso deveria estar incorporado na sintaxe e no ambiente de execução da linguagem, sem a necessidade, portanto, de extensões ou de chamadas de biblioteca para tal propósito.

Como uma característica estrutural desejável, uma linguagem de programação adaptativa, baseada ou não na extensão de uma linguagem não-adaptativa, deve estar associada a um ambiente de execução que dê ao compilador a possibilidade de referenciar no código objeto gerado, as operações adaptativas presentes na biblioteca ou no ambiente de execução. No primeiro caso, o ambiente deve ser visível ao programador enquanto que no segundo, deverá, de preferência, ser implícito e transparente ao programador.

Dentre os diversos recursos que podem fazer parte do ambiente de desenvolvimento e de execução para uma linguagem adaptativa, pode-se citar: recursos de edição, manuseio de arquivos, operações de rastreamento, operações de depuração, auxílio à escrita de programas adaptativos, acesso à camada adaptativa, monitoração do estado do programa (em caso de linguagens em estilo imperativo, incluir variáveis e valores).

Este ambiente deve, preferencialmente, ter uma interface gráfica, para melhor interação com o usuário. A Figura 50 esboça os componentes desejáveis de um ambiente de execução para uma linguagem adaptativa.

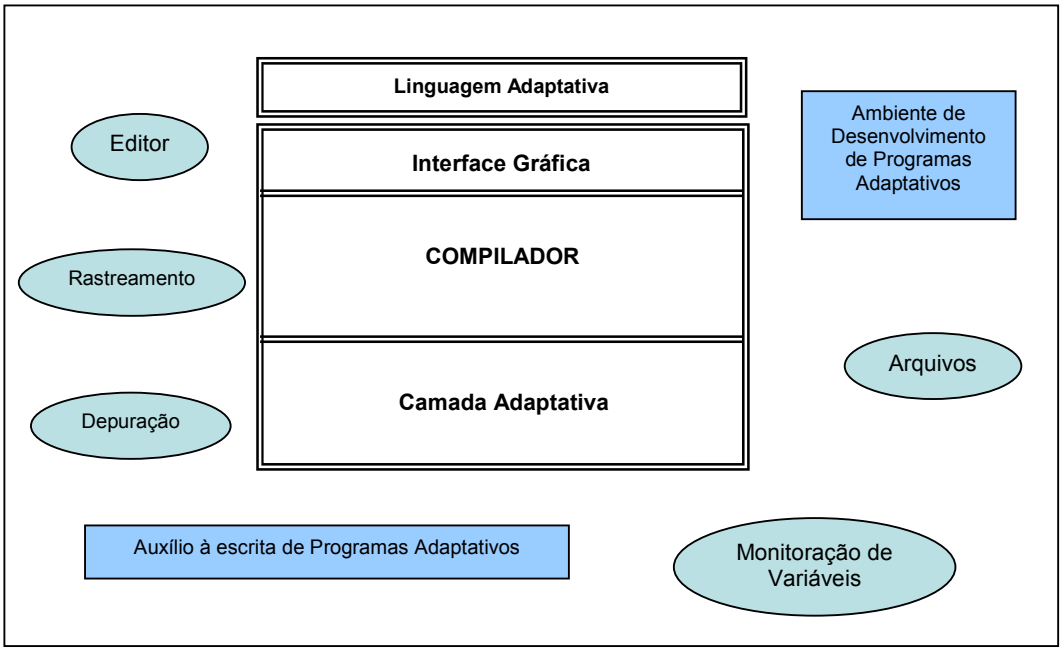


Figura 50 – Ambiente de Execução de Programas Adaptativos

Nesta tese, a biblioteca de operações adaptativas disponibiliza primitivas que podem ser empregadas, de forma manual, pelo programador durante a escrita de um programa em linguagem adaptativa.

O mecanismo de extensão presente na linguagem-base permite ao programador que esta seja estendida com a inserção de construtos adaptativos. Assim, com o auxílio de

uma biblioteca de operações adaptativas, de um ambiente de execução e de recursos de extensibilidade, pode-se ensaiar construtos adequados para a implementação de uma linguagem genuinamente adaptativa. Este assunto deverá ser alvo de pesquisas futuras, estando fora do escopo da presente tese.

A seguir descrevem-se algumas recomendações adicionais que se tornam desejáveis para a implementação de linguagens adaptativas, no tocante às características sintáticas e que serão objeto de futuras investigações.

Do ponto de vista sintático, no caso de a linguagem adaptativa ser baseada em uma linguagem não-adaptativa, todas as extensões sintáticas devem ser projetadas de forma aderente ao estilo sintático da linguagem-base. Tal recomendação trará ao programador maior facilidade no emprego da linguagem adaptativa, uma vez que a manutenção do estilo sintático para os recursos de adaptatividade pode proporcionar um aprendizado mais natural dos novos construtos.

Para que esta característica seja consolidada, recomenda-se também que o projeto da linguagem adaptativa mantenha uniforme a granularidadade da linguagem-base, de forma que não sejam expostos detalhes da linguagem estendida que sejam próprios de níveis de abstração inferiores aos da linguagem-base, nem tampouco do seu ambiente de execução, os quais devem se manter, preferenciamente, encapsulados na biblioteca da linguagem ou então totalmente transparentes ao usuário.

Uma questão, que deve ser considerada no desenvolvimento de uma linguagem adaptativa, se refere à forma pela qual se definem as partes do programa que são alvo da adaptatividade. Essas partes do programa compreendem regiões contíguas que delimitam o escopo das alterações adaptativas.

Alternativamente, é necessário que se defina, também através de referências, o ponto exato do programa onde será processada alguma operação adaptativa. Na sintaxe de uma linguagem genuinamente adaptativa recomenda-se que sejam definidos os construtos próprios para essas formas de referência, bem como, no mínimo, para operações adaptativas básicas de inserção e remoção de partes do programa.

Embora não seja regra, a forma adotada neste trabalho para estabelecer pontos referenciáveis se dá através do emprego de rótulos. Para delimitar regiões contíguas referenciáveis pode-se empregar o mesmo mecanismo de rótulos utilizado para blocos de instruções do programa. Alternativamente, mecanismos similares à delimitação de

regiões (escopos) utilizados em outras linguagens permitem obter efeitos semelhantes sem o emprego de nomes.

Outra questão aderente ao mecanismo de endereçamento de porções do programa se refere à possibilidade de o programador, ao projetar suas funções adaptativas, definir meta-rótulos, ou seja, dar um nome a uma família de rótulos a ser instanciada diversas vezes, sempre com nomes diferentes, permitindo assim que as referências aos componentes instanciados possam ser conjuntamente especificadas, de maneira uniforme.

Assim, quando o programador, numa função adaptativa, define um rótulo fixo, estabelece uma referência direta e explícita a um ponto referenciável do programa. No entanto, com o emprego de parâmetros, variáveis e geradores (no estilo clássico da teoria dos dispositivos adaptativos), as referências se tornam instanciáveis.

As Figuras 51 e 52 apresentam uma sumarização das diretrizes apontadas nesta tese, para auxiliar no projeto e implementação das linguagens adaptativas de programação.

- Linguagem Adaptativa (LPA) = Linguagem Não-Adaptativa (LNA) + Extensibilidade
- LPA é superset de LNA
- LPA – LNA é obtido por meio de Extensibilidade
- Mecanismo Adaptativo introduzido por extensão e por bibliotecas
- Ambiente de Execução visível ao programador
- Necessita de mecanismo de referência para modificação dinâmica
- Primitivas de Identificação e de Modificação de Código
- Estilo sintático aderente à Linguagem-Base (LNA)
- Facilidade aprendizagem

Figura 51 –Diretrizes para Desenvolvimento de Linguagens Adaptativas obtidas por Extensibilidade

- Necessita de mecanismo de referência para modificação dinâmica
- Primitivas de Rotulação e de Modificação de Código
- Estilo sintático aderente à Linguagem-Base
- Facilidade de aprendizagem
- Ambiente de Execução implícito ao programador
- Ambiente de Desenvolvimento c/edição, *trace*, *debug*, auxílio à escrita de programas, acesso à camada adaptativa, monitoração de estado, interface gráfica.
- Encapsulamento do ambiente de execução
- O projeto da linguagem adaptativa deve manter uniforme a granularidade da Linguagem-Base
- Construtos próprios para operações de adaptatividade
- Definição de meta-rótulos
- Definição do Escopo de Adaptatividade

Figura 52 –Diretrizes para Desenvolvimento de Linguagens Nativamente Adaptativas

6.2 Contribuições

Dentre as contribuições desta tese, pode-se destacar a conceituação de linguagens adaptativas de programação e consideração das diversas características que uma linguagem de programação necessita ter para ser considerada uma linguagem adaptativa.

Nesta tese mostrou-se a viabilidade de implementação de linguagens adaptativas por meio do emprego de linguagens funcionalmente extensíveis de programação.

A implementação usada nesta tese considerou uma linguagem funcional de programação à qual se acrescentou uma camada adaptativa responsável pela automodificação de código.

Para se exercitar a linguagem adaptativa desenvolvida nesta tese, desenvolveu-se uma aplicação, com características adaptativas, na qual foram ensaiados e validados os diversos recursos presentes na camada adaptativa da linguagem.

Esses recursos se limitam, entretanto, aos componentes de linguagem que são tratados em tempo de compilação e correspondem, portanto, às alterações que o usuário pode fazer sobre a linguagem, de modo a torná-la mais adequada a seus particulares objetivos.

A partir da disponibilização da linguagem adaptativa desenvolvida nesta tese estão sendo contempladas as necessidades daqueles usuários que necessitem que seus programas possam dinamicamente (ou seja, em tempo de processamento) efetuar alterações no fluxo de controle de execução.

Com base nas experimentações e estudos desenvolvidos nesta tese, pode-se também citar como contribuição a apresentação de um conjunto de diretrizes que permitem definir os requisitos de uma linguagem de programação adaptativa.

Ainda como contribuição desta tese pode-se citar, como decorrência da disponibilização das linguagens adaptativas de programação, a possibilidade do emprego destas em inúmeras áreas emergentes de aplicações de códigos automodificáveis tais como, proteção de software, ofuscação de código, engenharia reversa, etc.

6.3 Trabalhos Futuros

Tomando-se por base os ensaios e estudos desenvolvidos nesta tese, apresentam-se neste item alguns temas que podem fazer parte de futuros trabalhos envolvendo a área de linguagens de programação adaptativa.

Durante o desenvolvimento da aplicação com a linguagem adaptativa desenvolvida nesta tese, constatou-se que ao se empregar tal linguagem, uma forma própria de pensar ou de se raciocinar deve ser exercitada, de maneira natural e espontânea, como conseqüência das ações adaptativas presentes nos programas dela gerados.

Isso exige uma forma própria de raciocínio ao programar, uma vez que o comportamento do programa em execução está diretamente relacionado à prévia execução de ações adaptativas, cuja operação define o comportamento dinâmico do programa. Em outras palavras, exige uma disciplina de programação que antecipe os efeitos das ações adaptativas sobre o comportamento do dispositivo.

Como trabalho futuro, espera-se que seja desenvolvido um método que oriente o programador a bem utilizar a adaptatividade na sua programação. Nesta tese, como conseqüência dos experimentos realizados, observou-se que um primeiro passo na escrita de código adaptativo é a cuidadosa definição da primeira instância do programa, uma vez que, a partir desta instância, são aplicadas, ações de automodificação do código.

Esse método, para auxílio ao desenvolvimento do estilo adaptativo de programação, deve contemplar uma forma de avaliação da taxa de crescimento de código, tanto no espaço como no tempo, em conseqüência da sucessiva aplicação de ações adaptativas pela execução do programa.

Os ensaios desenvolvidos nesta tese mostraram que, como trabalho futuro, pode ser viável a construção de linguagens genuinamente adaptativas, nas quais a extensibilidade seria dispensável, uma vez que a nova linguagem pode incorporar todas as operações adaptativas que forem necessárias para a geração de programas adaptativos. Por exemplo, nesta tese, a forma de referenciação proposta, utilizada para se identificar partes do programa, se fez por meio do desenvolvimento e disponibilização de uma primitiva de identificação presente na camada adaptativa.

No entanto, em uma linguagem genuinamente adaptativa, tal recurso poderia estar incorporado à sintaxe e ao seu ambiente de execução, sem a necessidade, portanto, de extensões ou de chamadas explícitas de biblioteca para tal propósito.

Para propiciar ao usuário maior facilidade no emprego de linguagens adaptativas de programação é desejável também que, como trabalho futuro, se desenvolva um ambiente de execução que propicie outros recursos de desenvolvimento para programação adaptativa, tais como: recursos de edição, manuseio de arquivos, operações de rastreamento da execução de ações adaptativas e suas conseqüências, rastreamento das instâncias de código, monitoração de variáveis, geradores e outros elementos da camada adaptativa, etc.

Como trabalhos futuros, pode-se vislumbrar a conveniência do desenvolvimento de linguagens adaptativas de programação apoiadas em uma variedade de linguagens subjacentes.

6.4 Conclusão

Apesar da pesquisa envolvendo a tecnologia adaptativa ter contribuído de forma significativa em diversas áreas, poucos foram os trabalhos desenvolvidos com o foco de se explorar a questão da adaptatividade em linguagens de programação.

Assim, esta tese representa um ponto de partida para a pesquisa da área de linguagens adaptativas de programação, e mostra a viabilidade do uso de uma linguagem

de programação como dispositivo subjacente de um dispositivo adaptativo, não tendo a pretensão de apresentar uma proposta de linguagem completa para fins práticos. Além disso, extrai dos resultados de seus ensaios algumas diretrizes que possam nortear o projeto de linguagens genuinamente adaptativas.

Diversos outros temas de pesquisa que podem advir deste trabalho, os quais podem se apresentar na forma de projetos de conclusão de curso, projetos isolados, dissertações de mestrado ou teses de doutorado. Seguem abaixo algumas sugestões de temas:

- Projeto e implementação de Linguagens Genuinamente Adaptativas;
- Método para apoio ao Estilo Adaptativo de Programação;
- Projeto e Implementação de Linguagens Adaptativas aderentes ao Paradigma Orientado a Objetos;
- Projeto e Implementação de Linguagens Adaptativas aderentes ao Paradigma Imperativo;
- Ambientes de Execução para o apoio à construção de Linguagens Adaptativas;
- Linguagens Adaptativas aplicadas à Proteção de Software;
- Ambientes Multilinguagens com inclusão de Linguagens Adaptativas.

Agradeço aos professores que participaram das bancas de qualificação e defesa, pelas valiosas sugestões apresentadas e que contribuiram, de forma relevante, para a revisão deste trabalho.

7. Referências

ANCKAERT, B; MADOU, M; BOSSCHERE, K. D. **A Model for Self-Modifyind 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.

AUCSMITH, D. **Tamper resistant software: an implementation**. Lecture Notes in Computer Science, Springler Berlin / Heidelberg, ISSN 0302-9743, Volume 1174/1996, Information Hiding, ISBN 978-3-540-61996-3, Pages 317-333, January 20, 1996.

BASSETO, B. A.; NETO, J.J. **A stochastic musical composer based on adaptative algorithms**. Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação, SBC-99, PUC-Rio, Rio de Janeiro, Brasil, 1999.

BORBA, F. S. **Introdução aos Estudos Lingüísticos**. Companhia Editora Nacional, 9ª edição, São Paulo, SP, 1987.

BUDD, T. A. **Multiparadigm Programming in LEDA**. Addison-Wesley Publishing Company, ISBN: 0-201-82080-3, 1995.

CASTRO JR., A.A; NETO, J. J.; PISTORI, H. **Determininsmo em Autômatos de Estados Finitos Adaptativos**. Revista IEEE América Latina. Vol. 5, Número 7, ISSN: 1548-0992, pág. 515-521, Novembro 2007.

CHRISTODORESCU, M. ; JHA, S. **Static analysis of executables to detect malicious patterns**. USENIX Security Symposium, Proceedings of the 12th conference on USENIX Security Symposium, Volume 12, Washington, DC, Pages 169-186, 2003.

CIFUENTES, C.; GOUCH, K. **Decompilation of binary programs**. Software - Practice & Experience, Volume 25, Issue 7, Pages 811- 829, ISSN: 0038-0644, July, 1995.

DUX, B.; IYER, A.; DEBRAY, S.; FORRESTER, D.; KOBOUROV, S. **Visualizing the Behavior of Dynamically Modifiable Code**. IWPC, Proceedings of the 13th International Workshop on Program Comprehension, pages 337-340, ISBN: 1092-8138, ISSN: 0-7695-2254, 2005.

EBCIOGLU, K.; ALTMAN, E.; GSCHWIND, M.; SATHAYE, S. **Dynamic binary translation and optimization**. IEEE Transactions on Computers, Volume 50, Issue 6, Pages: 529-548, ISSN: 0018-9340, 2001.

FLATT, M.; FINDLER, R. B.; KRISHNAMURTHI S.; FELLEISEN M. **Programming Languages as Operating Systems**. International Conference on Functional Programming, Proceedings of the fourth ACM SIGPLAN International conference on Functional Programming, Pages 138-147, ISBN: 1-58113-111-9, Paris, France, 1999.

FREITAS, A. V. ; NETO, J. J. **Aspectos do Projeto e Implementação de Ambientes Multiparadigmas de programação**. Proceedings of ICIE Y2K - International Congress on Informatics Engineering, Buenos Aires, Argentina, 2000.

GIFFIN, J.T.;CHRISTODEVESCU M: Kruger L. **Strengthening Software Self-Checksumming via Self-Modifying Code**. ASCAC, Proceedings of 21st Annual Computer Security Applications Conference, Pages 23-32, ISBN: 1063-9527, ISSN: 0-7695-2461-3, 2005.

GHEZZI, C.; JAZAYERI, M. **Programming Language Concepts**. Third Edition, John Wiley & Sons, ISBN: 0-471-10426-4, 1998.

HARRISON, P. G.; FIELD, A. G. **Functional Programming**. Addison-Wesley Publishing Company, International Computer Science Series, Imperial College of Science and Technology, University of London, ISBN: 0-201-19249-7, 1988.

HENDERSON, P. **Functional Programming Application and Implementation** Prentice Hall, ISBN: 0-13-331579-7, 1980.

HUDAK, P. **Conception, Evolution, and Application of Functional Programming Languages**. ACM Computing Surveys (CSUR), Volume 21, Issue 3, Pages: 359-411, ISSN: 0360-0300, September, 1989.

HUSSAIN, S. **Code Obfuscation**. Master´s program in Security Engineering - Blekinge Institute of Technology, 2005.

IWAI, M. K. **Um formalismo gramatical adaptativo para Linguagens dependentes de Contexto**. Tese (Doutorado), Escola Politécnica da Universidade de São Paulo, São Paulo, Brasil, 2000.

KANZAKI, Y.; MONDEN, A.; NAKAMURA, M.; MATSUMOTO, K. **Exploiting self-modification mechanism for program protection**. COMPSAC, Proceedings of the 27th Annual International Computer Software and Applications Conference, Pages:

LINN, C. ; DEBRAY, S. **Obsfucation of executable code to improve resistance to static disassembly**. Conference on Computer and Communications Security, Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington D.C., USA, SESSION: Emerging applications, Pages: 290-299, ISSN: 1-58113-738-9, 2003.

LEWIS, H. R.;PAPADIMITRIOU, C. H. **Elements of the Theory of Computation**. Second Edition. Prentice-Hall Inc. 1998, ISBN: 0-13-262478-8, 1998.

MACLENNAN, B. J. **Functional Programming Practice and Theory**. Addison-Wesley Publishing Company, ISBN: 0-201-13744-5, 1990.

MADOU, M.; ANCKAERT, B.; MOSELEY, P.; DEBRAY, S.; SUTTER, B.D.; BOSSCHERE, K. **Software Protection through Dynamic Code Mutation**. Conference International Workshop on Information Security Applications – WISA, Pages: 194-206, 2005.

MAEBE, J. ; BOSSCHERE, K. D. **Instrumenting self-modifying code**. Proceedings of the Fifth International Workshop on Automated Debugging, September, Ghent. Computer Researching Repository, 2003.

MATSUMOTO, K.; KANZAKI, Y.; MONDEN, A.; NAKAMURA, M. **Exploiting Self-Modification Mechanism for Program Protection**. COMPSAC, Proceedings of the 27th Annual International Conference on Computer Sofware and Applications, 2003.

McCARTHY, J. **Recursive Functions of Symbolic Expressions and Their Computation by Machine**. Technical Report: AIM-8, Massachusetts Institute of Technology, Cambridge, MA, USA, 1959.

MCKINLEY, P. K.; MASOUD, S.; KASTEN, E. P.; CHENG, B. H. C. **Composing Adaptive Software**. Computer, Volume 37, Issue 7, Pages: 56-64, ISSN: 0018-9162, IEEE Computer Society Press, July, 2004.

MENEZES, C.E.D. **Um método para a construção de Analizadores Morfológicos, Aplicados à Lingua Portuguesa baseado em Autômatos Adaptativos**. Dissertação (Mestrado), Escola Politécnica da Universidade de São Paulo, São Paulo, Brasil, Julho 2000.

MONDEN, A.; TAKADA, Y.; TORII, K. **Methods for Scrambling Programs Containing Loops.** Transactions of the IEICE, The Institute of Electronics, Information and Communication Engineers, Volume J80-D-I, No. 7, Pages: 644-652, July,1997.

MORSE, S. P.; ALBERT, D. J. **Combating software piracy by encryption and key management**. IEEE Computer, Pages: 68-73, April, 1984.

MUELLER, L. **NewLISP User manual and reference**. v.9.2.0, 2007.

NETO, J. J. **Contribuição à metodologia de construção de compiladores**. São Paulo, 272p. Tese (Livre-Docência), Escola Politécnica, Universidade de São Paulo, 1993.

NETO, J. J. **Adaptive Automata for Context -Sensitive Languages**. SIGPLAN NOTICES, Volume 29, Issue 9, Pages: 115-124, September, 1994.

NETO, J. J. ; SOUZA, M. A. A.; HIRAKAWA, A. R. **Adaptive Automata for Mobile Robotic Mapping**. SBRN, Proceedings of VIII Brazilian Symposium on Neural Networks, São Luís/MA, Brazil, September 29 - October 1, 2004.

NETO, J. J. **Adaptive Rule-Driven Devices - General Formulation and Case Study.** Lecture Notes in Computer Science, Implementation and Application of Automata, 6th International Conference, CIAA 2001, Vol.2494, Pretoria, South Africa, July 23-25, Springer-Verlag, Pages: 234-250, 2001.

NETO, J. J. **Um levantamento da Evolução da Adaptatividade e da Tecnologia Adaptativa**. Revista IEEE América Latina. Volume 5, Número 7, ISSN: 1548-0992, Páginas: 496-505, Novembro, 2007.

NORVIG, P. **Self-Reproducing Programs in Common Lisp**. Computer Science Division, University of California, Berkeley, CA, 1990.

PEREIRA, J. C. **Ambiente Integrado de Desenvolvimento de Reconhecedores Sintáticos, baseado em Autômatos Adaptativos**. Dissertação de mestrado apresentada ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, São Paulo, 1999.

PISTORI, H. **Tecnologia em Engenharia da Computação: Estado da Arte e Aplicações**. Tese de Doutorado, Escola Politécnica da Universidade de São Paulo, 2003

RADY, J. A. J **Stad: uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos**. Tese de Doutorado, Escola Politécnica da Universidade de São Paulo, 1995.

RANGEL, J. L. ; CELES, W.; CERQUEIRA, R., **Introdução a Estrutura de Dados com Técnicas de Programação em C**. Editora Campos, ISBN: 85-352-1228-0, 2004.

ROCHA, R. L. A.; Neto, J.J. **Uma proposta de método adaptativo para a seleção automática de soluções**. Proceedings of ICIE Y2K, International Congress on Informatics Engineering, Buenos Aires, Argentina, 2000.

ROCHA, R. L. A.; NETO, J. J. **Uma proposta de linguagem de programação funcional com características adaptativas**. CACIC, IX Congreso Argentino de Ciencias de la Computación, 6-10 de Outubro, La Plata, Argentina, , 2003.

SAITO, I.; ISHIMA, H. ; KAMEI, K. ; SHIN, M. **Tamper Resistant Technology for Software.** Fuji Xerox Technical Report, N°. 13, Pages: 20-28, 2000.

SANTOS, J.M.N. **Um formalismo adaptativo com mecanismo de sincronização para aplicações concorrentes**. Dissertação (Mestrado), Escola Politécnica da Universidade de São Paulo, Brasil, 1997.

SEBESTA, R. W. **Concepts of Programming Languages**. 8th edition, Addison-Wesley Publishing Company, ISBN: 0321493621, 2007.

SETHI, R. **Programming Languages Concepts & Constructs**. Addison Wesley Publishing Company, 2th Edition, ISBN: 0-201-59065-4, 1996.

STANDISH, T. **Extensibility in programming language design**. ACM SIGPLAN Notices, Volume 10, Issue 7, Pages: 18–21, ISSN: 0362-1340, 1975.

TENNENT, R. D. **Principles of Programming Languages**. Prentice Hall PHI International, Inc., C.A.R. Hoare Series Editor, ISBN: 0-13-709873-1, 1981.

THOMBORSON, C.;COLLBERG, C. S. Watermarking, **Tamper-Proofing, and Obfuscation – Tools for Software Protection**. IEEE Transactions on Software Engineering, Volume 28, Issue 8, August, 2002.

THOMBORSON, C.; COLLBERG C. S.; LOW D. **Abstractions and Unstructuring Data Structures**. ICCL, Proceedings IEEE International Conference on Computer Languages Pages: 28-38, May, 1998.

TYAGI, A.; GE J.; CHAUDHURI S. **Control Flow Based Obfusction** ACM Workshop On Digital Rights Management, Proceedings of the 5th ACM Workshop on Digital rights management, SESSION: Software issues, Pages: 83-92, ISBN: 1-59593-230-5, Alexandria, Virginia, USA, 2005.

TSCHUDIN, C.; YAMAMOTO L. **Harnessing Self-Modifying Code for Resilient Software**. WRAC, Proceedings of the 2nd IEEE Workshop on Radical Concepts, NASA Goddard Space Flight Center Visitor´s Center, September 2005.

VAYNBERG, A. ; CAI H.; SHAO, Z. **Certified Self-Modifying Code**. ACM SIGPLAN Notices, Volume 42, Issue 6, Proceedings of the 2007 PLDI Conference, SESSION: Compiled correctly, Pages: 66-77, 2007.

WATT, D. A. **Programming Language Concepts and Paradigms**. Prentice Hall, C.A. R. Hoare Series Editor, Europe, ISBN: 0-13-728866-2, 1990.

ZINGARO, D. **Modern Extensible Languages**. McMaster University, Hamilton, Ontario, Canada, April, 2007.

8. Bibliografia Complementar

AHO, A.V; SETHI, R.; ULMANN, J.D. **Compilers, Techniques, and Tools**. Addison-Wesley, World Student Series Editon, ISBN: 0-201-10194-7, 1986.

APPEL A.W. **Modern Compiler Implementation in C**. Cambridge University Press, ISBN: 0-521-58653-4, 1997.

BIRD, R. **Introduction to Functional Programming using Haskell**. Second Edition, Prentice Hall Europe, ISBN: 0-13-484346-0, 1998.

BACKUS, J. **Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs.** Communications of the ACM, Volume 21, Issue 8 (August 1978), Pages 613-641, ISSN: 0001-0782, 1986.

BAKER, H. G. **Metacircular semantics for common Lisp Special Forms**. ACM SIGPLAN Lisp Pointers. Volume V, Issue 4 (October 1992), Pages 11-20, ISSN: 1045-3563, 1992.

BAL, H.E.; GRUNE D. **Programming Language Essentials**. Addison-Wesley Publishing Company, ISBN: 0-201-63179-2, 1994.

BARENDREGT, H.P. **The Lambda Calculus: its syntax and semantics**. 2nd ed., North-Holland, 1984.

BUDD, T. A. **Blending Imperative and Relational Programming**. IEEE Software, Volume 8, Issue 1 (January 1991) Pages: 58-65, ISSN: 0740-7459, 1991.

BUDD, T. A. **Data Structures in C++ using the Standard Template Library**. Second Edition, Addison-Wesley Longman, ISBN: 0-201-30879-7, 1998.

BURGE, W. H. **Recursive Programming Techniques**. Addison-Wesley Publishing Company, 1975.

CROWL, L. A.;HILZER, R. C.;LAWRENCE, A. J. **A Survey of Sequential and Parallel Implementation Techniques for Functional Programming Languages**. Departament of Computer Science Oregon State University, Corvallis, Oregon 97331-3202, Technical Report 95-60-05, May 1995.

DILLER, **A. Compiling Functional Languages**. John Wiley & Sons LTD, ISBN: 0 471 92027 4, 1988.

FARRER H.;BECKER C.G.;FARIA E.C.; MATOS H.F.;SANTOS M. A.;MAIA M.L. **Algoritmos Estruturados**, 3ª edição, LTC – Livros Técnicos e Científicos Editora S.A., ISBN: 85-216-1180-3, 1999.

FLOYD, R. W. **The Paradigms of Programming**. Communications of ACM, Volume 22, Issue 8, Pages 455-460, ISSN: 0001-0782, August, 1979.

FREITAS, A. V.; NETO, J.J. **Uma ferramenta para construção de aplicações multilinguagens de programação**. CACIC, Congreso Argentino de Ciencias de la Computación, El Calafate, Argentina, 2001.

FREITAS, A. V.; NETO, J.J. **Adaptive Device with underlying mechanism defined by a programming language**. Proceedings of the 4th International Conference on Information Security, Communications and Computers, December 16-18, Pages 423-428, Tenerife, Canary Islands, Spain, 2005.

FREITAS, A. V.; NETO, J.J. **Conception of Adaptive Programming Languages**. Proceedings of the 17h IASTED International Conference Modelling and Simulation, Pages: 453-458, May 24-26, Montreal, Canada, 2006.

FREITAS, A. V.; NETO, J.J. **Um ambiente para o processamento de Linguagens Adaptativas de Programação**. CACIC, Congreso Argentino de Ciencias de la Computación, San Luis, Argentina, 2006.

FREITAS, A. V.; NETO, J.J. **Adaptive Languages and a new Programming Style**. Proceedings of the 6th International Conference on Applied Computer Science, Pages: 220-225, Tenerife, Canary Islands, Spain, December 16-18, 2006.

FREITAS, A. V.; NETO, J. J. **Linguagens de Programação aderentes ao Paradigma Adaptativo.** Revista IEEE América Latina, Volume 5, Número 7, Páginas: 522-526, ISSN: 1548-0992, Novembro, 2007.

FREITAS, A. V.; NETO, J. J **Adaptivity in Programming Languages.** WSEAS TRANSACTIONS on INFORMATION SCIENCE & APPLICATIONS, Volume 4, Issue 4, ISSN: 1709-0832, Pages: 779-786, April, 2007.

FRIEDMAN, P. D.; MATHIAS, F. **The Little LISPer**. Macmillan Publishing Company, NY, 1989.

GLASER, H.; HANKIN, C. ; TILL D. **Principles of Functional Programming**. Prentice Hall, PHI International, ISBN: 0-13-709163-X, 1984.

GORDON, J. C. M. **The denotational Description of Programming Languages An Introducton**. Springer-Verlag, ISBN: 0-387-90433-6, 1979.

HUNTER, R. **The Design and Construction of Compilers**. John Wiley & Sons, ISBN: 0-471-28054-2, 1981.

JENKINS, M. A.; GLASGOW, J. I. **Programming Styles in Nial**. IEEE Software, Volume 3, Issue 1, Pages: 46-55, ISSN: 0740-7459, 1986.

KNOTT, G. **LispBook**. Civilized Software, Inc. 3310 Pendleton Drive Silver Spring. Maryland, 1997.

LANDIN, P.J. **The next 700 programming languages**. Communications of the ACM, Volume 9, Issue 3, Pages: 157-166, ISSN: 0001-0782, May,1966.

LANGSAM, Y.; AUGENSTEIN, M. J. ; TENENBAUM, A. M. **Data Structures using C and C++**. Second Edition, Prentice Hall, ISBN: 0-13-036997-7, 1990.

LOUDEN K.C. **Compiladores Princípios e Práticas**. Editora Thomson, ISBN: 85-221-0422-0, 2004.

MAURER W.D. **The Programmer's Introduction to Lisp**. MacDonald American Elsevier Computer Monographs, ISBN: 0-44419572-6, 1972.

MICHAELSON G.;TRINDER P.;LOIDL H. **Trends in Functional Programming**. Intellect Books, ISBN 1-84150-024-0. 2000.

MODY, R. P. **Functional programming is not Self-modifying Code**. ACM Sigplan Notices, Volume 27, Issue 11, Pages 13-14, ISSN: 0362-1340, November, 1992.

MÜLLER, M.; MÜLLER, T.; ROY, P. V. **Multiparadigm Programming in Oz**. German Research Center of Artificial Intelligence (DFKI), Stuhlsatzenhausweg, Saarbrücken, Germany, 1995.

PLACER, J. **Multiparadigm Research: A New Direction in Language Design**. ACM Sigplan Notices, Volume 26, Issue 3, Pages: 9-17, ISSN: 0362-1340, March,1991.

PRATA, S. **C++ Primer Plus –Third Edition** – SAMS, ISBN: 1-57169-162-6, 1998.

PRATT, T. W. ; ZELKOWITZ M. V. **Programming Languages, Design and Implementation**. Third Edition - Prentice Hall Inc., ISBN: 0-13-678012-1, 1996.

RANGEL, J. L. ; CELES, W.; CERQUEIRA, R., **Introdução a Estrutura de Dados com Técnicas de Programação em C**. Editora Campos, ISBN: 85-352-1228-0, 2004.

SIKLÓSSY, L. **Let´s talk Lisp**. Prentice Hall, Inc. Englewood Cliffs, ISBN: 0-13-532762-8, New Jersey , 1976.

STEELE, G. L. **Common Lisp, The Language**. 2nd Edition, ISBN: 1-55558-041-6, Digital Press, Bedfor, MA, 1990.

STEARNS, R.E.; LEWIS P.M.; ROSENKRANTZ, D. J. **Compiler Design Theory**, Addison Wesley Publishing Company, ISBN: 0-201-14455-7, 1976.

STROUSTRUP, B. **C++ Programming Language**. Third Edition, Addison Wesley Publishing Company, ISBN: 0-201-88954-4, 1997.

SZWARCFITER J.L.; MARKENZON L. **Estruturas de Dados e seus Algoritmos**, 2ª Edição Revista, Editora LTC, ISBN: 85-216-1014-9, 1994.

ZIVIANI N. **Projeto de Algoritmos com implementações em Pascal e C**. Editora Thomson, ISBN: 85-221-0174-4, 1993.

Apêndice A – Autômatos Adaptativos

Dentre os diversos dispositivos adaptativos estudados, os autômatos adaptativos têm um destaque especial, uma vez que marcaram o início da pesquisa na área. Nesta tese, as linguagens adaptativas de programação foram conceituadas tomando-se por base o estudo dos autômatos adaptativos. Assim, neste item apresentam-se algumas considerações a respeito de uma implementação utilizando a linguagem *Lisp* como linguagem subjacente.

Autômatos adaptativos são dispositivos adaptativos particulares cujo formalismo subjacente é o autômato de pilha estruturado. Sua operação corresponde à uma seqüência de evoluções sucessivas de um autômato de pilha estruturado inicial processadas por ações adaptativas. A cada ação adaptativa, surge um novo autômato, dando continuidade ao tratamento da cadeia de entrada (NETO,1994).

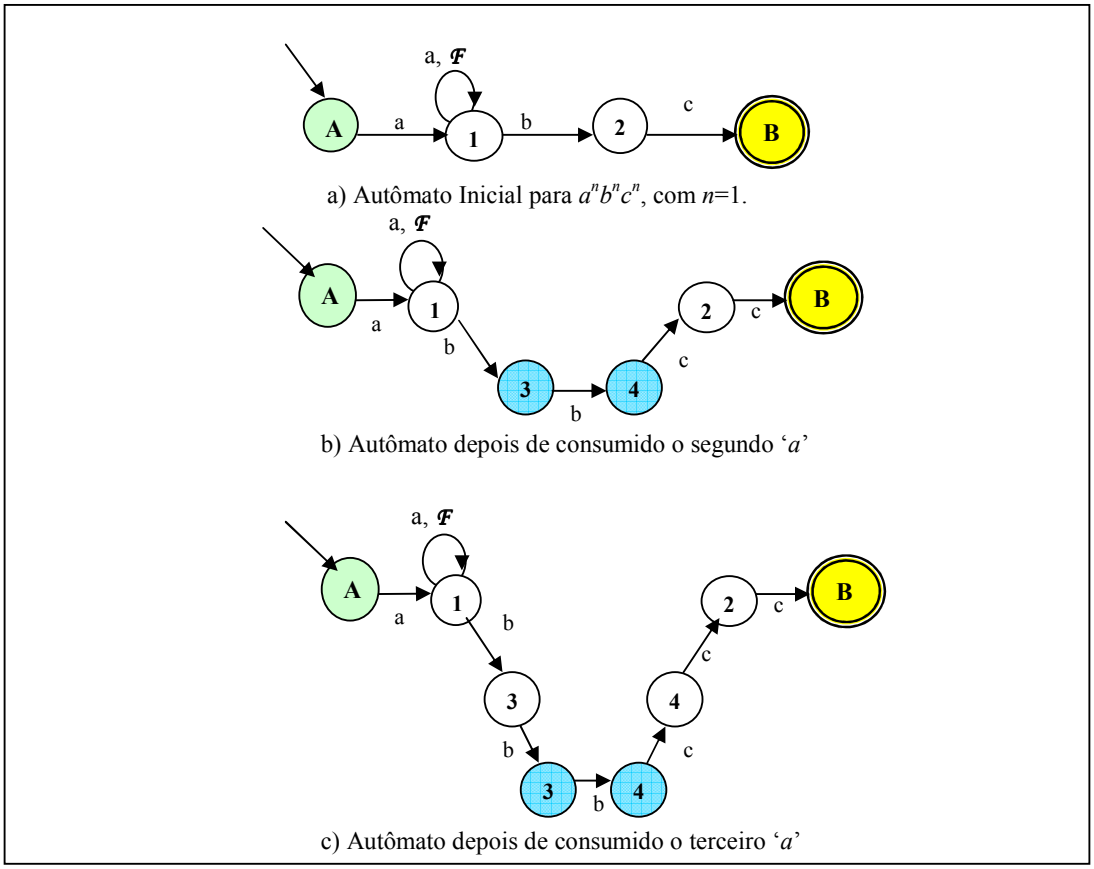


Figura 53 – Autômato Adaptativo para reconhecer cadeias expressas por $a^n b^n c^n$, com n>0

Para exemplificar a operação dos autômatos adaptativos, apresenta-se a seguir uma implementação do reconhecimento da linguagem $a^n b^n c^n$, com n>0, a qual poderá ser representada por um autômato que deverá reconhecer as cadeias "abc", "aabbcc", "aaabbbccc", e assim por diante, e rejeitar as demais. A Figura 53 mostra o esquema do autômato a ser modelado no problema.

Caso a linguagem a ser reconhecida fosse definida por $a^1 b^1 c^1$, o nosso autômato finito poderia ser representado pelos estados (A,1,2,B) sendo A o estado inicial, B o estado de aceitação e pelas transições ((A,'a',1), (1,'b',2), (2,'c',B)). Denotaremos este autômato finito inicial por AF_1 (Figura 53-a).

Analogamente, conforme Figura 53-b, para a linguagem definida por $a^2 b^2 c^2$, necessitaríamos de um outro autômato finito AF_2 representado por um novo conjunto de estados (A,1,2,3,4,B) e também por uma nova lista de transições ((A,'a',1), (1,'a',1), (1,'b',3), (3,'b',4), (4,'c',2), (2,'c',B)).

Para se generalizar o reconhecimento da linguagem $a^n b^n c^n$, pod-se estruturar um autômato adaptativo formado pelo espaço de autômatos finitos $AF_1, AF_2, \ldots, AF_n$, de tal modo que a partir do autômato finito inicial AF_1 , e por meio de chamadas de funções adaptativas, o dispositivo evolua para sucessivas configurações $AF_2, AF_3, .., AF_n$,à medida em que a cadeia de entrada fosse sendo processada.

Para se materializar estas modificações das configurações dos autômatos finitos, necessita-se do processamento de ações adaptativas, as quais terão a missão de promover a reconfiguração dinâmica do dispositivo e, portanto, sem a interferência de agentes externos (NETO,94).

Para a efetivação destes conceitos, inclui-se na transição (1,'a',1) do autômato inicial AF_1, uma chamada de função adaptativa, denotada por ***F***, que será executada antes do processamento da correspondente transição. Em formato de listas, denota-se a transição por ((***F***), (1,'a',1), ( )). A lista vazia, especificada a direita da transição, indica que não se tem neste caso, chamadas de funções adaptativas posteriores a execução da transição usual do autômato, enquanto que a lista (F) especificada à esquerda da transição indica que a função adaptativa ***F*** deverá ser executada anteriormente à execução da transição (1,'a', 1).

Conforme esquema apresentado na Figura 53, para cada átomo ‘a’ consumido no estado 1, adicionam-se novos estados em posições convenientes, novas transições são adicionadas e uma é eliminada, como resultado da chamada da função adaptativa ***F***.

Esta função adaptativa cria, inicialmente, dois novos estados (3 e 4), outra transição consumindo ‘b’ do estado 3 para o estado 4, mais uma transição consumindo ‘c’ do estado 4 para o estado 2 e retira a transição consumindo ‘b’ do estado 1 para o estado 2 (Figura 53-b).

Portanto, quando inicialmente chamada, a função adaptativa ***F*** é composta pelas seguintes ações elementares:

```
+ Transição ( ( ) , (1, 'b', 3 ), ( ) )
+ Transição ( ( ) , (3, 'b', 4 ), ( ) )
+ Transição ( ( ) , (4, 'c', 2 ), ( ) )
- Transição ( ( ) , (1, 'b', 2 ), ( ) )
```

Depois de providenciada a adição dos novos estados 3 e 4, a posição onde devem ser incluídos os próximos dois novos estados também deve ser atualizada. Na primeira vez que a função adaptativa é chamada, os estados 1 e 2 são passados como parâmetros. Na próxima evolução do autômato, os estados 3 e 4 são passados como parâmetros, e entre estes estados devem ser inseridos dois novos estados, por exemplo, 5 e 6.

Genericamente, a função adaptativa ***F*** poderá ser especificada da seguinte forma:

Sejam i , j estados quaisquer do autômato adaptativo, no qual a seguinte configuração ocorre: $(i,'b') \rightarrow j$ e $(j,'c') \rightarrow k$.

Por exemplo, na situação inicial os estados *i* e *j* correspondem aos estados 1 e 2. Na evolução seguinte, 3 e 4, na seguinte 5 e 6, e assim por diante.

Para propiciar ao autômato características de automodificação, a função adaptativa ***F*** deverá pesquisar o autômato adaptativo para determinar os estados i e j entre os quais serão acrescentadas novas transições e deverá ser composta por uma lista de ações adaptativas elementares, as quais efetuarão consultas e alterações explícitas na configuração corrente do autômato adaptativo.

Considerando-se que, com a evolução do autômato, haverá criação de novas transições, as ações adaptativas elementares irão manipular variáveis diferenciadas, que serão preenchidas automaticamente, sem solicitação explícita, com valores únicos ainda

não empregados na definição corrente do autômato. Estas variáveis serão denotadas por *m** e *n** e as chamaremos de geradores (NETO,1994).

Assim, para cada *i* , *j* estados quaisquer do autômato adaptativo, tem-se o mapeamento: (i,j) : m*,n*

Ou seja, tem-se com o decorrer da execução do autômato, os mapeamentos:

(1,2): m*=3,n*=4 ; (3,4): m*=5,n*=6 ; . . .

No exemplo, a função adaptativa ***F*** pode ser composta pelas seguintes ações elementares:

```
? [(i,'b') → j ]          ;;ação de consulta retornando transições (i, 'b', j)
? [(j,'c') → k ]          ;;ação de consulta retornando transição (j, 'c', k)
                          ;;i e j são encontrados...
- [(i,'b') → j ]          ;;deleção de transição (i,'b',j)
+ [(i,'b') → m* ]         ;;adição da transição (i,'b', m*)
+ [(m*,'b') → n*]         ;;adição da transição (m*,'b', n*)
+ [(n*,'c') → j ]         ;;adição da transição (n*,'c', j )
```

O conjunto inicial de produções do dispositivo adaptativo pode ser representado por:

```
'(
        (       ( )     (A 'a' 1)       ( )     )
        (       ( )     (2 'c' B)       ( )     )
        (       ( F )   (1 'a' 1)       ( )     )
        (       ( )     (1 b '2')( )    )
)
```

Após a primeira evolução do dispositivo, têm-se as seguintes produções:

```
'(
( ( )   (A 'a' 1)       ( )     )       mantida
( ( )   (2 'c' B)       ( )     )       mantida
( (F )  (1 'a' 1)       ( )     )       mantida
~~( ( )   (1 'b' 2)       ( )     )       eliminada~~
( ( )   (1 'b' 3 )      ( )     )       inserida
( ( )   (3 'b' 4 )      ( )     )       inserida
( ( )   (4 'c' 2 )      ( )     )       inserida
)
```

Na segunda evolução do dispositivo, têm-se as seguintes produções:

```
'(
   ( ( )  (A  'a'  1)        ( )      )      mantida
   ( ( )  (2  'c'  B)        ( )      )      mantida
   ( ( F ) (1  'a'  1)       ( )      )      mantida
   ( ( )  (1  'b'  3 )       ( )      )      mantida
   ( ( )  (4  'c'  2 )       ( )      )      mantida
   ( ( )  (3  'b'  4 )       ( )      )      eliminada
   ( ( )  (3  'b'  5 )       ( )      )      inserida
   ( ( )  (5  'b'  6 )       ( )      )      inserida
   ( ( )  (6  'c'  4 )       ( )      )      inserida
)
```

Um autômato finito é definido por uma quíntupla (Q, Σ, δ, q_o, F) [Lewis 1998] e sua representação em notação de listas pode ser feita da seguinte forma:

```
AUTOMATO:     (
                    Lista_de_Estados
                    Alfabeto
                    Lista_de_Transições
                    Estado_Inicial
                    Lista_de_Estados_de_Aceitação
              )
Lista_de_Transições:  ( Transição Transicao Transicao  . . . )
Transicao:  ( Estado_Origem        Átomo       Estado_Destino )
```

Para se definir o autômato adaptativo, necessita-se redefinir as transições para suportarem as ações adaptativas. Assim, cada transição do autômato pode ser redefinida por: Transicao: ((AA) (Estado_Origem Átomo Estado_Destino) (AP))

sendo: AA a ação adaptativa anterior e AP a ação adaptativa posterior.

Estas ações adaptativas (tanto anterior quanto posterior), podem ser definidas em notação Lisp, por meio da definição:

```
(define  (Nome_Funcao_Adaptativa   Lista_Parâmetros)
                variáveis
                geradores
                Lista_Ações_Elementares
        )
 )
```

Com estas considerações, pode-se representar o autômato adaptativo por meio da função *anbncn*, que recebe como parâmetros a cadeia de entrada, a lista inicial de estados do autômato, o estado inicial, a lista inicial de transições, o alfabeto e a lista de estados finais de aceitação:

```
(anbncn
        '(a a a b b b c c c )          ;; cadeia de entrada
        '( A 1 2 B )                   ;; lista inicial de estados
        '( A )                         ;; estado inicial
        '(
                ( ()  (A a 1) () )
                ( ()  (2 c B) () )
                ( (F) (1 a 1) () )
                ( ()  (1 b 2) () )
          )                            ;; lista inicial de transições
        '(a b c)                       ;; alfabeto
        '(B)                           ;;lista de estados de aceitação
)
```

A lista de parâmetros anteriormente descrita está coerente com o esquema adotado para a representação dos autômatos adaptativos, uma vez que na lista inicial de transições tem-se a ocorrência de uma transição adaptativa, representada pela lista ((**F**) (1 a 1) ( )).

O programa funcional pode ser representado por uma função anbncn que recebe como parâmetros os elementos que devem compor o autômato finito adaptativo:

```
(define
        (anbncn
                Cadeia_Entrada
                Lista_de_Estados
                Estado_Inicial
                Lista_de_Transicoes
                Alfabeto
                Lista_de_Estados_de_Aceitacao
        )
)
```

Essa função anbncn inicialmente entra em operação como se estivesse processando um autômato finito usual.

O programa inicialmente efetua o posicionamento da máquina no estado inicial, e a partir daí faz o processamento da cadeia de entrada.

Com o átomo lido e estado corrente definido, dá-se início à pesquisa das transições passadas como argumento. Caso a transição que atenda as condições de momento sejam não-adaptativas, o autômato evolui normalmente conforme um autômato finito usual.

Porém, se a transição que atenda as condições de pesquisa seja adaptativa, o programa efetua chamadas de funções adaptativas, que podem ocorrer antes ou após a execução da transição não-adaptativa. Neste exemplo, a função adaptativa é chamada antes da execução da transição adaptativa e será responsável pela reconfiguração do autômato.

Um possível procedimento para esta função pode ser apresentado da seguinte maneira:

```
;; posicionamento da máquina no estado inicial
;; leitura da cadeia de entrada
;; pesquisa transições a partir do estado corrente e átomo lido
;; pesquisa se na lista de transições há chamadas adaptativas
;; se houver chamadas adaptativas, executá-las
;; após chamada adaptativa, continuar com a
;; evolução usual do autômato
;; se houver chamadas adaptativas após a transição, executá-las
;; após processamento da cadeia de entrada
;; verificar se estado final é de aceitação
;; se sim então cadeia reconhecida ...
```

Para que o tratamento adaptativo seja materializado nesse exemplo, deve-se definir a função adaptativa necessária para que o dispositivo tenha a propriedade de automodificação.

Tendo em vista que a função adaptativa é composta por ações elementares de consulta, remoção ou de adição de transições, seria desejável que se tivesse à disposição um conjunto de rotinas, as quais poderiam ser convenientemente combinadas, para a implementação das funções adaptativas.

Este conjunto de rotinas define uma "camada adaptativa", a qual será responsável pelo encapsulamento das ações adaptativas elementares.

Denotam-se essas ações elementares por ***?adapt***, ***+adapt*** e ***–adapt***, respectivamente responsáveis pelas ações de consulta, adição e eliminação de transições da lista de transições corrente (FREITAS,2005).

Obviamente, para cada problema em particular, extraim-se da camada adaptativa, as ações elementares, na quantidade e seqüência convenientes para o dispositivo aderente ao problema em questão.

As ações de consulta também devem estar atreladas ao dispositivo em particular, uma vez que dependem de cada problema a ser tratado. Ou seja, para cada problema, deve-se especificar uma consulta diferente ou mais conveniente à lista de produções.

Para generalizar esta função de consulta, "queries", modeladas por "pattern matching", podem ser especificadas de modo a retornar a lista de produções que atendem a consulta.

Assim, nesse exemplo, ao se passar a lista de transições do dispositivo adaptativo juntamente com o padrão ((*) (8,*,*) (*)) à função ***?adapt*** da camada adaptativa, tem-se como retorno todas as transições (adaptativas ou não) partindo do estado 8.

Analogamente, ao se passar a lista de transições do dispositivo adaptativo juntamente com o padrão ((***F***) (*,'a',*) (*)) à função ***?adapt*** da camada adaptativa, obtem-se como retorno uma lista de todas as transições adaptativas que chamam a função adaptativa F antes de promover a transição não-adaptativa, quando um átomo 'a' for consumido.

Se passarmos a lista de transições do dispositivo adaptativo juntamente com o padrão ((*) (*,*,*) (***G***)) à função ***?adapt*** da camada adaptativa, obtem-se como retorno uma lista de todas as transições adaptativas que chamam a função adaptativa *G* após a execução de qualquer transição.

O padrão ((*) (*, *, *) (*)) passado juntamente com a lista de transições do dispositivo à função ***?adapt*** da camada adaptativa, retornará a lista de todas as produções correntes do dispositivo adaptativo.

Assim, genericamente, a função ***?adapt*** receberá como parâmetro um padrão de definição de "*query*" e a lista de transições do dispositivo, retornando uma lista de transições que atendem ao padrão passado, ou uma lista vazia em caso de não haver "matching".

Em nosso exemplo, as regras que compõem nossa função adaptativa podem ser implementadas a partir da camada adaptativa, em formato Lisp, da seguinte forma:

Regra:	Mapeamento com chamada da Camada Adaptativa
? [(i,'b') → j]	**(set 'L1 (?adapt (((*),(*,'b',*),(*)) , Lista_TR)))**
	// retorna L1 com transições (i, 'b', j) – vinculadas ao átomo 'b'
? [(j'c') → k]	**(set 'L2 (?adapt (((*),(j,'c',*),(*)) , L1)))**
	// retorna L2 com transição (j, 'c' , k) – determinação dos estados (i , j)
– [(i,'b') → j]	**(set 'Lista_TR (–adapt ((( ),(i,'b',j),( )) , Lista_TR)))**
	// retorna Lista de Transições c/ eliminação da transição (( ),(i,'b',j),( ))
+ [(i,'b') → m*]	**(set 'Lista_TR (+adapt ((( ),(i,'b', m*),( )) , Lista_TR)))**
	// retorna Lista de Transições c/adição da transição (( ),(i,'b', m*),( ))
+ [(m*,'b') → n*]	**(set 'Lista_TR (+adapt ((( ),(m* ,'b', n*),( )) ,Lista_TR)))**
	// retorna Lista de Transições c/adição da transição (( ),(m* ,'b', n*),( ))
+ [(n*,'c') → j]	**(set 'Lista_TR (+adapt ((( ),(n* ,'c', j),( )) , Lista_TR)))**
	// retorna Lista de Transições c/adição da transição (( ),(n* ,'c', j),( ))

Na implementação apresentada, mostrou-se como as funções adaptativas, por meio do emprego de funções elementares presentes na camada adaptativa, propiciam as alterações de configuração do dispositivo adaptativo.

Em caso da substituição do mecanismo subjacente de autômato para linguagem de programação, as funções adaptativas estarão alterando a própria linguagem, gerando como conseqüência, numa linguagem automodificável.

Apêndice B - Códigos automodificáveis

Define-se um código automodificável como sendo um código que tem, durante sua execução, a capacidade de alterar suas próprias instruções.

Tendo em vista que códigos automodificáveis são difíceis de serem compreendidos, mantidos e depurados, muitas ferramentas de análise de código assumem que o código existente é constante, ou seja, não-automodificável.

Para se ilustrar o conceito de códigos automodificáveis, apresenta-se a seguir um simples conjunto de instruções baseado num processador hipotético:

1. Gravação de uma informação em um determinado endereço de memória:

Código Assembly:	MOVBYTE VALOR,ENDEREÇO
Código de operação:	0x0F
Código Binário:	0x0F valor endereço
Exemplo:	0x0F0x0B0x04

//Move o dado B0 para o endereço 04

2. Desvio para um endereço absoluto de memória:

Código Assembly:	BRANCH ENDEREÇO
Código de operação:	0x0A
Código Binário:	0x0A endereço
Exemplo:	0x0A0x04

//Desvia para o endereço 04

3. Incrementa registrador:

Código Assembly:	INCREG REGISTRADOR
Código de operação:	0x0B
Código Binário:	0x0B registrador
Exemplo:	0x0B0x05

//Incrementa o registrador 5

4. Decrementa registrador:

Código Assembly:	DECREG REGISTRADOR
Código de operação:	0x0C
Código Binário:	0x0C registrador
Exemplo:	0x0B0x05

//Decrementa o registrador 5

5. PUSH registrador:

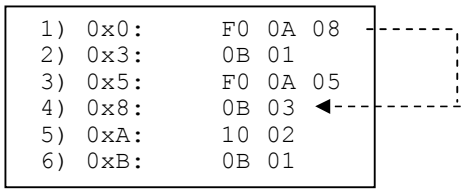
Código Assembly:	PUSH REGISTRADOR
Código de operação:	0x10
Código Binário:	0x10 registrador
Exemplo:	0x100x04

//Push do registrador 4 na pilha

Para se elucidar a operação de um código automodificável, consideraremos o seguinte trecho de código assembly:

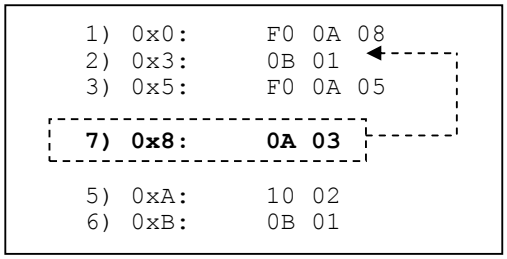
```
1) MOVBYTE  0xA, 0x8
2) INCREG         R1
3) MOVBYTE  0xA, 0x5
4) INCREG         R3
5) PUSH           R2
6) DECREG         R1
```

Para o código assembly acima, tem-se o seguinte código binário correspondente:

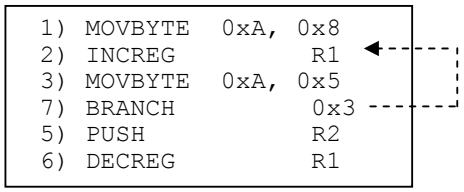


Ao se executar o código acima, logo na primeira instrução, observa-se que esta modifica o endereço correspondente ao código de operação da instrução 4.

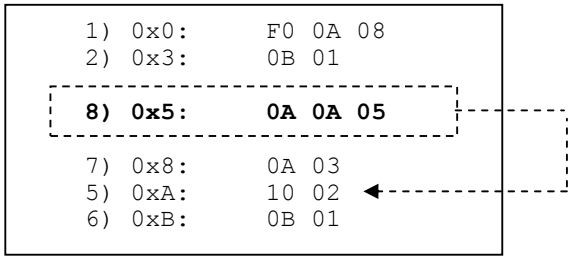
Desta forma após a execução da instrução 1, o código binário terá o seguinte conteúdo:



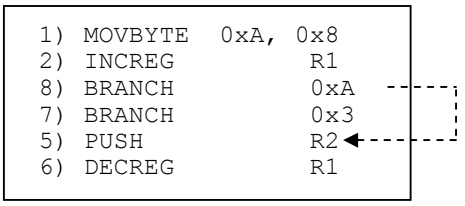
o que irá, por sua vez, corresponder ao seguinte código assembly:



A seguir a instrução 2 é executada, seguida pela instrução 3 que se automodifica. Após a execução da instrução 3, o código binário terá o seguinte conteúdo:



o que irá, por sua vez, corresponder ao seguinte código assembly:



Conforme explanação anterior pode-se constatar que o grafo de fluxo de controle para códigos automodificáveis se modifica com o correr da execução do programa. Assim, para códigos automodificáveis torna-se necessário representar no grafo de fluxo de controle um superconjunto de todas as possíveis execuções que podem ocorrer em tempo de execução (ANCKAERT,2006).

Apêndice C - Dominó Adaptativo

Implementação c/Linguagem Adaptativa

```
;-
(define (domino L)

 (if       ( =  ( L 10)  'i )

                        ( domino (init L ) ) )              ;; inicialização
(if       ( =  ( L 10)  'u )

                        ( domino (jogada_usuario L ) ) )        ;; vez do usuário

(if       (=  ( L 10)  'm )

                        ( domino (jogada_maquina L ) ) ) ;; vez da máquina

(if       (=  ( L 10)  'ug )

                        (silent (println "\n\nParabéns, você venceu...") ) )

(if       (=  ( L 10)  'mg )

                        (silent (println "\n\nSinto muito, a máquina venceu...")))

;-
(define (jogada_usuario L )

(imprime_configuracao L)

;; checa se usuário tem jogadas para realizar

(if (!= (length ( (gera_LJ L (L 0)  )  9) ) 0 )

;; sim, o usuário tem peças para jogar

(begin (nth-set ( L 5)

( ( (gera_LJ L (L 0)  )  9)

( - (mostra_opcoes ( (gera_LJ L (L 0) )  9) 0 )  1) ) )

(nth-set ( L 2) (atualiza_mesa (L 5) (L 2) ) )

(nth-set ( L 0 ) (atualiza_pecas (L 5) (L 0) '() ) )

;; usuário jogou uma determinada peça

;; usuário não passou

(nth-set (L 15) 'n)

;; teste para verificar se usuário ganhou a partida

(if (= (length ( L 0) )  0 )
```

```
(nth-set (L 10) 'ug )                ;; usuário ganhou

(begin  (nth-set (L 10) 'm )      ;; a próxima vez é da máquina

;; se usuário escolheu regra adaptativa, questiona se deseja mudar o conjunto de
regras...

(if (= ( (L 5) 3 ) 'a )

(begin (println "\n\nVocê escolheu uma jogada associada a uma regra adaptativa...")

(print "Deseja alterar o conjunto de regras ?  ==> (s/n)  ")

(if  (= (read-line) "s")

(begin (nth-set (L 3) (altera_regras (L 3) (L 3) '(nil nil) ) )

(println "\n\nNovas Regras:  " (L 3) ) )

(println "\nVoce não fez jogada associada

          a uma regra adaptativa ") ) ) ) ) ) )

;; usuário não tem jogada

;; checagem de eventual empate

(begin (if (= (L 15) 's)

(begin (silent (dotimes (x 80) (println " ") ) )

(println "_____________________________________________________")

(imprime_pecas_usuario ( L 0 ) )

(imprime_pecas_maquina ( L 1 ) )

(imprime_mesa (L 2) )

;; empate (deadlock...)

(println "\n\nIMPASSE.... ")

(println "\nGanha quem tiver menos pontos...\n\n")

(println "Total de Pontos do Usuário:  " (calcula_pontos (L 0) 0 )   )

(println "Total de Pontos da Máquina:  " (calcula_pontos (L 1) 0 )   )

(if (>= (calcula_pontos (L 0) 0 ) (calcula_pontos (L 1) 0 ) )

(nth-set (L 10) 'mg )

(nth-set (L 10) 'ug ) ) )

(begin (nth-set (L 15) 's )
```

```
(print "\n\n\n ==> Você não tem peças para jogar!!! " )

;; a próxima vez é da máquina

(nth-set (L 10) 'm ) ) ) ) ) L )

;-
(define (jogada_maquina L )

(println  " \n\n ==> Tecle enter para ver a jogada da máquina ... " )

(read-line)

;; checa se a máquina tem jogadas para realizar

(if (!= (length ( (gera_LJ L (L 1) )  9) ) 0 )

;; sim, a máquina tem peças para jogar

;limpa a console

(begin (silent (dotimes ( x 80) (println " " ) ) )

(nth-set ( L 5)  ( ( randomize  ( (gera_LJ L (L 1) )  9 ) ) 0 ) )

(println "A máquina escolheu a jogada [" (( L 5) 1) ":" ( (L 5) 2) "]" " a"

(if ( = ((L 5 )  0) 'd ) " direita" " esquerda" )  " da mesa!")

(println "_____________________________________________________________")

(nth-set ( L 2) (atualiza_mesa (L 5) (L 2) ) )

(nth-set (L 1 ) (atualiza_pecas (L 5) (L 1) '() ) )

;; máquina jogou uma determinada peça

;; máquina não passou

(nth-set (L 15) 'n)

;; teste para verificar se máquina ganhou a partida

(if (= (length ( L 1) )  0 )

(nth-set (L 10) 'mg )    ;; máquina ganhou

(nth-set (L 10) 'u ) ) )   ;; a próxima vez é do usuário

;; máquina não tem jogada ;; checagem de eventual empate

(begin (if (= (L 15) 's)

; empate (deadlock...)

(begin (silent (dotimes (x 80) (println " ") ) )

(println "_____________________________________________________________")
```

```
(imprime_pecas_usuario ( L 0 ) )

(imprime_pecas_maquina ( L 1 ) )

(imprime_mesa (L 2) )

(println "\n\nIMPASSE.... ")

(println "\nGanha quem tiver menos pontos...\n\n")

(println "Total de Pontos do Usuário:  " (calcula_pontos (L 0) 0 ) )

(println "Total de Pontos da Máquina:  " (calcula_pontos (L 1) 0 ) )

(if (>= (calcula_pontos (L 0) 0 ) (calcula_pontos (L 1) 0 ) )

(nth-set (L 10) 'mg )

(nth-set (L 10) 'ug ) ) )

(begin

(nth-set (L 15) 's )

(silent (dotimes (x 80) (println " ") ) )

(print "\n\n\n ==> A máquina não tem peças para jogar !!!  \n\n" )

(println "_____________________________________________________")

;; a próxima vez é do usuário

(nth-set (L 10) 'u )) ) ) ) L )

;--
(define (init L )

;; definicao da lista de dados do programa

;; índice        nomenclatura
;;       0       LU: lista do usuário
;;       1       LM: lista da máquina
;;       2       LT: lista da mesa
;;       3       LR: lista de regras
;;       4       LTJ: lista total de jogadas
;;       5       LJE: lista de jogada escolhida
;;       6       LP: lista das 27 pecas
;;       7       LJE: lista de jogadas a esquerda
;;       8       LJD: lista de jogadas a direita
;;       9       LJV: lista de jogadas válidas
;;       10      controle
;;               (u      -> usuario joga)
;;               (m      -> maquina joga)
;;               (ug     -> usuario ganha)
;;               (mg     -> maquina ganha)
;;               (e      -> empate)
;;               (i      -> inicializa)
```

```
;;
;;        11      pem - ponta esquerda da mesa
;;        12      pdm - ponta direita da mesa
;;        13      ne - nipe esquerdo
;;        14      np - nipe direito
;;        15      controle de empate
;;                (s      -> empate)
;;                (n      -> não empate)
;;        16      (somatoria de pontos do usuário)
;;        17      (somatória de pontos da máquina)
 (silent (dotimes ( x 80) (println " " ) ) )

(println "----------------------------------------")

(println " Bem vindo ao Dominó Adaptativo. V 1.0" )

(println "----------------------------------------")

;; pecas embaralhadas

(nth-set (L 6) (randomize (L 6) ) )

;Usuario com 7 peças

(nth-set (L 0) (sort (slice (L 6) 0 7) ) )

;máquina com 7 peças

(nth-set (L 1) (slice (L 6) 7 7 ) )

;vai iniciar o jogo com o usuário

(nth-set (L 10) 'u ) L )

;-
(define (imprime_configuracao L)

(imprime_regras ( L 3 ) )

(imprime_total_maquina  ( L 1 ) )

(imprime_pecas_usuario        ( L 0 ) )

(imprime_mesa ( L 2 ) ) )

;-
(define (sel_peca  L)

;imprime pecas do jogo

(print " " "["(L 0) ":" ( L 1)"]"))

;-
(define (imprime_mesa          L)

;imprime configuração da Mesa

(print "\n\n\n--------------")
```

```
(print "\nMesa ===> " )
(map sel_peca L)
(print "\n---------------")
(print "\n"))
;-
(define (imprime_pecas_usuario        L )
;imprime total de peças do usuário
(print "\nTotal de peças com o Usuário:  " (length L) "  ===> ")
(map sel_peca  L ))
;-
(define (imprime_pecas_maquina       L )
;imprime total de peças da máquina
(print "\nTotal de peças com a máquina:  " (length L) "  ===> ")
(map sel_peca  L ) )
;-
(define (sel_regra  L)
;imprime Regras
(print   "(" (L 0) "," ( L 1) "," ( L 2 ) ") " ))
;-
(define (imprime_regras L )
(print "\nRegras vigentes  ===> \t")
(map sel_regra L )
(println "\n\n\t\t+--------------------------+")
(imp_regra L)
(print "\n\t\t+--------------------------+"))
;-
(define (imp_regra L)
(if (!= (length L ) 0 )
(begin (if (=  ( (L 0 ) 2 ) '* )
(print "\t\t| A regra R" ( (L 0) 0) " é Adaptativa! |"  ) )
(imp_regra (rest L) ) )) )
```

```
;-
(define (imprime_total_maquina        L )

          (println "\n\nTotal de peças com a Máquina:  " (length L) )   )

;-
(define  (gera_LJ  L PECAS )

; atualiza ponta esquerda da mesa

(nth-set (L 11) ( (first ( L 2) ) 0 ) )

;;(println "Ponta esquerda da mesa = " ( L 11 ) )

; atualiza ponta direita da mesa

(nth-set (L 12)  (last (last ( L 2) ) ) )

;;(println "Ponta direita da mesa  = " ( L 12 ) )

; busca regras para as pontas da mesa

;; usuario pode se quizer alterar regras aqui....

; atualiza nipe_esquerdo a partir das regras

(nth-set (L 13)  (busca_nipe (L 3) (L 11) ) )

;;(println "Nipe esquerdo obtido de regras=" (L 13) )

(nth-set (L 14)  (busca_nipe (L 3) (L 12) ) )

;;(println "Nipe direito obtido de regras=" (L 14) )

;; gera total de jogadas válidas

(nth-set (L 4)  (combpecas  PECAS '() ) )

;; pesquisa jogadas válidas ****com nipe a esquerda***

(nth-set (L 7) ( bje  ( L 4) (L 13) '()  )  )

;; pesquisa jogadas válidas ****com nipe a direita****

(nth-set (L 8)  ( bjd  ( L 4) (L 14)  '()  )  )

;; constrói lista total de jogadas válidas

(nth-set (L 9 ) (cljv (L 8) 'd (L 3) (L 11) (L 12)

( cljv (L 7) 'e  (L 3) (L 11) (L 12) '() )  )  ) L )

;-
(define (busca_nipe R peca_mesa )

(if (!= (length R ) 0 )

(if (= ( ( R 0 ) 0 ) peca_mesa ) ( ( R 0 ) 1 )
```

```
          (busca_nipe (rest R) peca_mesa) ) ) )

;-
(define (combpecas U  LTJ  )

(if (!= (length U ) 0 )

(begin (push (first U) LTJ )

(push (list (last ( U 0) ) ( first ( U 0) ) ) LTJ )

(combpecas  (rest U ) LTJ ) )( unique LTJ )))

;-
(define (bje  L ne LJE )

(if (!= (length L) 0)

(begin  (if        (=  (last ( L 0 ) )  ne )   (push (L 0) LJE) )

(bje (rest L) ne LJE ) ) LJE ) )

;-
(define (bjd  L nd LJD )

(if (!= (length L) 0) (begin (if   (=  (first ( L 0 ) )  nd )

(push (L 0) LJD ) )

(bjd (rest L) nd LJD ) )  LJD  )  )

;-
(define (cljv L lado R pem pdm LJV )

; Constrói  lista de jogagas válidas no Formato:
;
;; ( 'e x y 'a ) ... ('e x y 'n ) )

;; ( 'd x y 'a ) ... ('d x y 'n ) )

;; x e y são os nipes da jogada

;; 'e significa que joga no lado esquerdo da mesa

;; 'd significa que joga no lado direito da mesa

;; 'a significa que é jogada c/regra adaptativa

;; 'n significa que é jogada c/regra não-adaptativa

(if (!= (length L) 0 )  (begin  (push

( list lado (first (L 0 ) ) (ast  (L 0 ) )

(if        (or (and  (= lado 'e) (=  (last ( R pem )  )  '* )

(and     (= lado 'd) (=  (last ( R pdm ) )  '* ) ) ) 'a          ;;regra adaptativa
```

```
'n        ;;regra não-adaptativa ) ) LJV            -1 )

(cljv (rest L) lado R pem pdm LJV )) LJV ) )

;-
(define (mostra_opcoes L indice )

(if (!=  (length L ) 0 )   (begin (print "\n\t")

(print (+ indice 1) )

(print " ====> ")

(print "[" )

(print ( (L 0) 1) )

(print ":")

(print ( (L 0) 2) )

(print "]" )

(if       (= ( ( L 0) 0 ) 'e ) (print "   ( Jogada a ESQUERDA da mesa ")

(print "   ( Jogada a DIREITA  da mesa ") )

(if       (= ( ( L 0) 3 ) 'a )  (print "ADAPTATIVA )  ")

(print "NÃO-ADAPTATIVA ) ")

(print "\n")

(mostra_opcoes (rest L) (+ indice 1) )  )

(begin (print "\n Escolha uma jogada entre 1 e " indice  ": => ")

(escolhe_opcao (integer (read-line) ) indice ) ) ) )

;-
(define (escolhe_opcao opcao_digitada indice)

(if       (or  >  opcao_digitada indice )         (<=  opcao_digitada 0 ) )

(begin (println "\n\t\t\t **** Opção Inválida **** ")

(print "\n Escolha uma jogada entre 1 e " indice  ": => ")

(escolhe_opcao (integer (read-line) ) indice ) ) opcao_digitada ) )

;-
(define ( atualiza_mesa    LJE  LT )

(if (= (LJE 0) 'e ) (push (list ( LJE 1) (LJE 2) )  LT) )

(if (= (LJE 0) 'd ) (push (list ( LJE 1) (LJE 2) )  LT -1) ) LT )
```

```
;-
(define  (atualiza_pecas J  U  P )

(if (!= (length U ) 0 )

(begin (if         (not     ( or      ( and    (= (J 1 ) ( ( U 0 ) 0 ) )

(= (J 2 ) ( ( U 0 ) 1 ) )    ) ( and (= (J 2 ) ( ( U 0 ) 0 ) )

(= (J 1 ) ( ( U 0 ) 1 ) )    ) ) )

(push (U 0) P ) )

(atualiza_pecas J (rest U) P  )   ) (reverse P ) ) )

;-
(define (calcula_pontos L TP)

(if (!= (length L ) 0 )

(calcula_pontos (rest L) (+ TP ( +   ( ( L 0 ) 0 )

( ( L 0 ) 1 )  ) ) ) TP ) )

;-
(define (altera_regras R T X)

(nth-set (X 0) (mostra_opcoes_regras R 0) )

(nth-set ( R ( ( X 0) 0 ) 1 )  ( (T ( ( X 0 ) 1) ) 1 ) )

(nth-set ( R ( ( X 0) 0 ) 2 )  ( (T ( ( X 0 ) 1) ) 2 ) )

(nth-set ( R ( ( X 0) 1 ) 1 )  ( (T ( ( X 0 ) 0) ) 1 ) )

(nth-set ( R ( ( X 0) 1 ) 2 )  ( (T ( ( X 0 ) 0) ) 2 ) ) R )

;-
(define (mostra_opcoes_regras L indice )

(if (!=  (length L ) 0 )

(begin (print "\n\t")

(print indice )

(print " -> ")

(print (L 0) )

(print "\n")

(mostra_opcoes_regras (rest L) (+ indice 1) ) )

(begin (println "\nInforme duas regras para alterar adaptatividade ..." )

(list (escolhe_regras L indice ) (escolhe_regras L indice) ) ) ) )
```

```
;-
(define ( escolhe_regras L indice)

(print "\n => ")

(escolhe_opcao_regra  (integer (read-line) ) ) )

;-
(define (escolhe_opcao_regra opcao_digitada)

(if       (or  (>  opcao_digitada 6 ) (<  opcao_digitada 0 ) )

(begin (println "\n\t\t\t **** Opção Inválida **** ")

(print "\n => ")

(escolhe_opcao_regra (integer (read-line) ) ) ) opcao_digitada ) )

;-
( domino

'(

()                ;lista do usuário

()                ;lista da máquina

( (6 6) )         ;lista da mesa

(
(0 0 -)  ;Regra 0
(1 1 -)  ;Regra 1
(2 2 -)  ;Regra 2
(3 3 -)  ;Regra 3
(4 4 -)  ;Regra 4
(5 5 -)  ;Regra 5
(6 6 *)  ;Regra 6
)
 ()               ;lista total de jogadas
 ()               ;lista jogada escolhida
 (
(0 0)    ;Peça 0
(0 1)    ;Peça 1
(0 2)    ;Peça 2
(0 3)    ;Peça 3
(0 4)    ;Peça 4
(0 5)    ;Peça 5
(0 6)    ;Peça 6
(1 1)    ;Peça 7
(1 2)    ;Peça 8
(1 3)    ;Peça 9
(1 4)    ;Peça 10
(1 5)    ;Peça 11
(1 6)    ;Peça 12
(2 2)    ;Peça 13
(2 3)    ;Peça 14
(2 4)    ;Peça 15
(2 5)    ;Peça 16
```

```
(2 6)    ;Peça 17
(3 3)    ;Peça 18
(3 4)    ;Peça 19
(3 5)    ;Peça 20
(3 6)    ;Peça 21
(4 4)    ;Peça 22
(4 5)    ;Peça 23
(4 6)    ;Peça 24
(5 5)    ;Peça 25
(5 6)    ;Peça 26
)
 ()      ;lista jogadas a esquerda
 ()      ;lista jogadas a direita
 ()      ;lista jogadas válidas
i        ;controle p/init
6        ;pem - ponta esquerda da mesa
6        ;pdm - ponta direita da mesa
6        ;ne - nipe esquerdo
6        ;np - nipe direito
n        ; empate = n
0        ; total de pontos do usuário
0        ; total de pontos da máquina
)
)
```

Glossário

Adaptatividade

Conceito apoiado na capacidade que um sistema tem de, sem a interferência de qualquer agente externo, tomar a decisão de modificar seu próprio comportamento, em resposta ao seu histórico de operação e aos dados de entrada.

Ambiente Multilinguagem

É um sistema integrado ou conjunto de primitivas que permite ao projetista desenvolver uma aplicação em mais de uma linguagem de programação. Caso as linguagens componentes da aplicação sejam oriundas de diferentes paradigmas, o ambiente também será dito multiparadigma.

Aprendizagem de Máquina

Estudo do desenvolvimento de algoritmos e técnicas de que possibilitem o aprendizado do computador, isto é, permitem que o computador aperfeiçoe seu desempenho durante a execução de alguma tarefa.

Árvore de Decisão

Corresponde à representação de uma tabela de decisão sob a forma de uma árvore.

Autômato Adaptativo

Dispositivos propostos por Neto (1993) como uma forma eficiente de se executar o reconhecimento da classe de linguagens sensíveis ao contexto. Por meio de funções adaptativas presentes no autômato, a configuração do autômato é modificada de acordo com o processamento da cadeia de entrada, onde transições podem ser inseridas ou eliminadas, de forma que este possa adquirir nova estrutura. No início do processamento tem-se uma versão inicial automodificadora do autômato que, conforme processa a cadeia de entrada, exibe configurações intermediárias, diferentes da sua configuração original, até que uma configuração final é alcançada, quando a cadeia de entrada é totalmente consumida.

Autômato de Pilha Estruturado

Dispositivos reconhecedores constituídos de uma família de submáquinas, cujas transições podem ser internas, como as de um autômato finito, ou então responsáveis pela movimentação entre submáquinas (chamadas e retornos). Neste caso, uma pilha é utilizada, estritamente para memorizar estados de retorno, os quais são empilhados sempre que uma submáquina é chamada, para memorizar o estado para o qual ela deve retornar ao final da operação da submáquina assim acionada. Ao final da operação da submáquina, tal estado de retorno é desempilhado e utilizado para promover o prosseguimento da operação interrompida da submáquina chamadora.

Cálculo Lambda

Formalismo matemático que modela aspectos computacionais das funções. O seu estudo nos auxilia a entender os elementos básicos da programação funcional e sua semântica, independentemente da consideração dos aspectos sintáticos de uma particular linguagem de programação. De uma forma bem simplista podemos considerar o cálculo lambda como o cálculo de funções anônimas. Expressões lambda basicamente consistem de um lambda (λ), um ou mais parâmetros formais seguido por um ponto, um corpo de função e um ou mais argumentos.

Cognição

Ação ou processo de conhecimento, que envolve atenção, percepção, memória, raciocínio, juízo, pensamento e linguagem. A psicologia cognitiva estuda os processos de aprendizagem e de aquisição de conhecimento.

Dispositivo Adaptativo

Dispositivos que podem ser modelados como abstrações matemáticas capazes de modificar-se dinamicamente, constituindo assim formalismos capazes de automodificar-se autonomamente.

Dispositivo Subjacente

Um dispositivo adaptativo é constituído de duas partes: a primeira é usualmente algum dispositivo usual, definido através de regras (em particular, algum autômato, gramática ou qualquer outro dispositivo descrito através de um conjunto finito de regras estáticas), denominado seu dispositivo subjacente (tipicamente não-adaptativo); a segunda é um mecanismo adaptativo, cuja conexão ao formalismo subjacente proporciona-lhe todos os recursos complementares necessários para a realização das propriedades responsáveis pela automodificação autônoma que caracteriza os dispositivos adaptativos.

Efeitos Colaterais

Correspondem a uma característica das linguagens imperativas que tornam difícil a manutenção de um programa. No entanto, podem ser usados para se estabelecer uma comunicação entre unidades de programa.

Escopo

Regras semânticas que disciplinam o acesso a objetos do programa de acordo com o seu escopo de declaração e execução. Quando estas regras dependem apenas do exame isolado do programa-fonte, são chamadas de regras de escopo estático, ou léxico. Quando dependem de considerações em tempo de execução, são chamadas de regras de escopo dinâmico.

Extensibilidade

Capacidade que um sistema tem de se expandir por meio da adição de novos componentes. Por meio da extensibilidade um sistema pode incorporar incrementalmente novas funcionalidades, ou novos recursos, e até nova sintaxe, por solicitação do usuário, mediante o fornecimento de definições, para as extensões desejadas, em função da combinação dos recursos já disponíveis. Com a extensibilidade, torna-se possível ao programador modificar, enquanto codifica seu programa, a potencialidade de sua linguagem de programação, adequando-a às suas necessidades.

Função

Declaração que associa um identificador e um enunciado. O identificador é o nome da função, sendo que ao nome da função se associa uma informação do tipo de valor retornado pela função após o término da sua execução. O enunciado é o corpo da função, sendo o enunciado formado por sentenças válidas da linguagem de programação.

Gramática

Formalismo empregado na descrição das sentenças válidas de uma linguagem a partir do seu alfabeto.

Linguagem de montagem

Representação simbólica das instruções executáveis pelo processador, para facilitar a leitura e a compreensão do código-objeto pelos humanos.

Lisp

Linguagem de programação criada por John McCarthy no qual se usa exclusivamente funções matemáticas como estruturas de dados elementares. O seu nome vem de LISt Processing. Em Lisp a lista é a estrutura de dados principal. Tanto dados como o próprio programa são representados por listas, o que permite que a linguagem manipule o código fonte como qualquer outro tipo de dados.

Macro-instrução

Corresponde a um agrupamento de instruções, ao qual se atribui um nome e se associam argumentos, agrupamento este que pode ser utilizado posteriormente como uma chamada da macro, na forma usualmente utilizada na linguagem de montagem. O uso de macros traz como benefício uma maior comodidade para se exprimir trechos repetidos de instruções.

Máquina de Turing

Formalismo concebido pelo britânico Alan Turing no qual se representa um modelo abstrato de um computador, restringindo-o apenas aos seus aspectos lógicos de funcionamento (memória, estados e transições). Consiste de um controle finito, uma fita e uma unidade de leitura/gravação nesta fita. Vista como uma forma de exprimir uma

linguagem de programação, a Máquina de Turing tem um simples conjunto de símbolos como estrutura de dados. As operações disponíveis permitem ao programa mover um cursor para a esquerda ou direita sobre uma cadeia de símbolos de entrada e efetuar leitura ou gravação na posição corrente do cursor

Máquina virtual

Modelo idealizado da máquina responsável pela execução do código-objeto. A máquina virtual combina instruções, ou seqüência de instruções, em alguma forma de representação intermediária e interpreta-as acionando rotinas de apoio do ambiente de execução que provocam a execução dos comandos do programa-fonte na máquina.

Otimização

Fase, opcional, do processo de compilação, que promove a geração de versões mais eficientes de código-objeto. Uma versão mais eficiente alcança os mesmos resultados que uma versão não otimizada, porém de forma mais rápida, ou usando menos memória, ou ambos. Algumas otimizações são independentes da arquitetura do computador e podem ser realizadas como transformações das formas intermediárias de representação do programa. Outras otimizações dependem da arquitetura do computador e devem ser realizadas como transformações do código-objeto. Assim, a otimização aparece como duas fases adicionais do processo de compilação, uma imediatamente após a fase de expansão de código, denominada de otimização global, e outra imediatamente após a geração do código-objeto, denominada de otimização local, ou do código-objeto.

Paradigma

É a representação do padrão de um modelo a ser seguido. Corresponde a um teoria ou um conhecimento que origina o estudo de um campo científico.

Procedimento

Declaração que associa um identificador a um enunciado. O identificador é o nome do procedimento e o enunciado é o corpo do procedimento, sendo o enunciado formado por sentenças válidas da linguagem de programação. Em muitas linguagens, os procedimentos que retornam valores são chamados de funções. Pode-se considerar um programa completo como um procedimento.

Processamento de Linguagem Natural

Corresponde a uma área da Inteligência Artificial que estuda os problemas de geração e compreensão automática das linguagens humanas naturais.

Programa automodificável

Código que tem, durante sua execução, a capacidade de alterar suas próprias instruções.

Programa-fonte

Consiste em um arquivo com instruções do programa, denotados em alguma linguagem de programação. Uma vez codificado, o programa é submetido ao compilador para geração do código-objeto ou ao interpretador para execução.

Recursividade

Forma de se criar algoritmos no qual uma função tem a capacidade de se chamar a si mesma.

Rede de Markov

Sistema de estados e transições, como no autômato finito. O sistema estocástico, uma vez em um dado estado, transita para outro estado em função das relações de probabilidade entre as transições que partem desse estado. O que caracteriza a rede de Markov, chamada de primeira ordem, é o fato de que a probabilidade de o sistema atingir um dado estado depende apenas do estado atual e da probabilidade da ocorrência de uma transição entre estes dois estados.

Robótica

É uma área da Tecnologia que estuda o desenvolvimento de sistemas compostos por máquinas e partes mecânicas automáticas e controladas por circuitos integrados.

Sentença

Uma seqüência finita de símbolos do alfabeto da linguagem e que obedece às regras que regem a estrutura das sentenças da linguagem.

Statechart

Formalismo utilizado para especificar o aspecto comportamental de sistemas reativos complexos, tendo sido desenvolvido por David Harel em Israel, em 1982. Baseia-se em Máquinas de Estados Finitos e diagramas de transição, com características adicionais que tornam mais clara sua representação gráfica (visual).

Tabela de Decisão

Corresponde a uma maneira de expressar, na forma de tabela, um conjunto de condições que devem ocorrer para que um determinado conjunto de ações deva ser executado. O conjunto de ações a serem tomadas, a partir de um conjunto de condições, se constitui em uma regra de decisão.

Tecnologia Adaptativa

Aplicação da Adaptatividade para fins práticos.

Transparência Referencial

Propriedade de uma linguagem de programação no qual se assegura que o resultado da aplicação de uma função será o mesmo, para um dado conjunto de argumentos, sem que se considere quando ou onde será avaliada. Uma das principais diferenças entre as notações matemáticas e programas imperativos é a noção de transparência referencial. A manipulação de fórmulas na álgebra, aritmética e lógica é apoiada no princípio da transparência referencial. As linguagens imperativas violam este princípio.