SALVADOR RAMOS BERNARDINO DA SILVA

SOFTWARE ADAPTATIVO:
MÉTODO DE PROJETO, REPRESENTAÇÃO GRÁFICA E IMPLEMENTAÇÃO DE LINGUAGEM DE PROGRAMAÇÃO

São Paulo
2011

SALVADOR RAMOS BERNARDINO DA SILVA

SOFTWARE ADAPTATIVO:
MÉTODO DE PROJETO, REPRESENTAÇÃO GRÁFICA E IMPLEMENTAÇÃO DE LINGUAGEM DE PROGRAMAÇÃO

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia Elétrica.

São Paulo
2011

SALVADOR RAMOS BERNARDINO DA SILVA

SOFTWARE ADAPTATIVO:
MÉTODO DE PROJETO, REPRESENTAÇÃO GRÁFICA E IMPLEMENTAÇÃO DE LINGUAGEM DE PROGRAMAÇÃO

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. João José Neto

São Paulo
2010

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, de novembro de 2011.

Assinatura do autor 

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Silva, Salvador Ramos Bernardino da
Software adaptativo: método de projeto, representação gráfica e implementação de linguagem de programação / S.R.B. da Silva. -- ed.rev. -- São Paulo, 2011.
113 p.

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

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

DEDICATÓRIA

Ao meu pai (*in memoriam*), pela base que proporcionou tudo o mais.

À minha mãe, incansável lutadora pelo sucesso dos seus filhos.

AGRADECIMENTOS

Agradeço ao Deus do meu coração, por mais essa existência terrena e pelas oportunidades que me estão sendo concedidas.

À minha esposa Suely, pelo apoio incondicional, incentivo nos momentos menos favoráveis, tolerância pelas minhas ausências do convívio familiar e, sobretudo, pelo amor inconteste.

Aos meus filhos, pela compreensão e pelo carinho.

Ao Prof. Dr. João José Neto, pela inestimável orientação ao longo de todo o trabalho, amizade, incentivo, profícuas conversas do cafezinho.

Aos Professores Dr. Ricardo Luis de Azevedo Rocha e André Riyuiti Hirakawa, pelas orientações na banca de qualificação.

À Suframa, pelo patrocínio financeiro do Mestrado Interinstitucional – MINTER; à CAPES, pela criação e regulamentação do MINTER; à Universidade do Estado do Amazonas – UEA, por propiciar a oportunidade de cursar o programa de Pós Graduação da Escola Politécnica da Universidade de São Paulo – EPUSP em Manaus.

Aos colegas de docência da UEA, pelo apoio.

Aos colegas de turma, pelo incentivo, pelos grupos de estudo e pelas horas de agradável convivência.

A todos que, de alguma forma, contribuíram para o sucesso desta empreitada.

RESUMO

Linguagem para programação adaptativa se apresenta como um tema relativo às tecnologias adaptativas que suscita muitas pesquisas, até que venha a se constituir em mais uma ferramenta disponível ao programador. Nessa perspectiva, esta dissertação apresenta uma linguagem de alto nível, básica, que permita a geração de código adaptativo, bem como os requisitos e características de uma linguagem dessa natureza, recomendações sobre como projetar um software nessa linguagem e a representação gráfica do mesmo, o respectivo compilador e o ambiente de *run-time* no qual os programas serão executados. Apresenta, ainda, um exemplo de um programa fonte escrito na linguagem proposta.

Palavras chave: Adaptatividade. Linguagem de programação. Programação adaptativa.

ABSTRACT

Programming Languages for coding adaptive programs constitute a very rich research subject in the field of adaptive technology. It aims to offer a tool for programmers to express adaptive programs in a user-friendly, high-level, machine-readable, abstract notation. This MSc thesis presents the specification of requirements and desirable attributes for high-level languages allowing the automatic generation of adaptive code, and recommends a design methodology for conceiving and implementing adaptive programs using such languages as well. A graphical notation is suggested for that purpose, a compiler and a run-time environment are also considered and described. An example is given of the design and semi-formal specification of an adaptive software with the help of the suggested graphical notation, and finally the corresponding source program is encoded in the proposed language and executed.

Keywords: Adaptivity. Programming Languages. Adaptive Programs.

SUMÁRIO

1 INTRODUÇÃO 9

1.1 MOTIVAÇÃO 10
1.2 HISTÓRICO 13
1.3 OBJETIVOS 18
1.4 ORGANIZAÇÃO DA DISSERTAÇÃO 19

2. REFERENCIAL 21

2.1 ASPECTOS TEÓRICOS 21
2.1.1 Dispositivos guiados por regras 21
2.1.2 Autômato de pilha estruturado 23
2.1.3 Funções, recursividade, macros, iteração e adaptatividade 25
2.2 LINGUAGENS DE PROGRAMAÇÃO 28
2.2.1 Paradigmas de programação 30
2.2.2 Extensibilidade 35
2.2.2.1 Extensibilidade léxica 36
2.2.2.2 Extensibilidade funcional 37
2.2.2.3 Extensibilidade sintática 37
2.3 TECNOLOGIA 38
2.3.1Especificação de sintaxe 39
2.3.2 Especificação de semântica 40
2.3.3 Ambiente de execução 44

3. PROGRAMAÇÃO ADAPTATIVA 46

3.1 REPRESENTAÇÃO GRÁFICA PARA SOFTWARE ADAPTATIVO 48
3.2 RECOMENDAÇÕES PARA O PROJETO DE PROGRAMAS ADAPTATIVOS 53
3.3 PROPOSTA DE LINGUAGEM PARA PROGRAMAÇÃO ADAPTATIVA 56
3.4 IMPLEMENTAÇÃO DA LINGUAGEM PROPOSTA 60

4 APLICAÇÃO 65

5 CONSIDERAÇÕES FINAIS 76

5.1 CONTRIBUIÇÕES 76
5.2 TRABALHOS FUTUROS 77
5.3 CONCLUSÃO 78

REFERÊNCIAS 80

APÊNDICE A – NOTAÇÃO DE WIRTH 85

APÊNDICE B – ESTRUTURA SINTÁTICA DA LINGUAGEM PROPOSTA 86

APÊNDICE C – SEMÂNTICA OPERACIONAL DA LINGUAGEM 91

APÊNDICE D – CÓDIGO EXECUTÁVEL DO EXEMPLO A 107

APÊNDICE E – CÓDIGO EXECUTÁVEL DO EXEMPLO B 109

1 INTRODUÇÃO

Software adaptativo é um termo que será utilizado neste trabalho para designar programas cujo código pode sofrer automodificações em consequência de decisões tomadas pelo programa representado por esse mesmo código.

A presente dissertação tem como objetivo tornar mais metódico o desenvolvimento de software adaptativo, com a ajuda das seguintes contribuições principais: (i) recomendações para o projeto de softwares adaptativos; (ii) uma proposta de simbologia gráfica, através da qual se possam representar, de maneira expressiva e significativa, programas automodificáveis; (iii) uma linguagem de programação de alto nível para a codificação de programas automodificáveis; (iv) o projeto e implementação de um compilador para tal linguagem; (v) um exemplo ilustrativo, detalhado, destinado a servir como guia para desenvolvimentos dessa natureza, bem como base para futuras evoluções nessa área.

O desenvolvimento de linguagens de programação, nas últimas décadas, ocorreu, possivelmente, por fatores como o rápido aumento do poder computacional, ao mesmo tempo em que surgem novas aplicações computacionais demandadas por uma comunidade bastante diversificada, tais como as de usuários de inteligência artificial, educação, ciência e engenharia, sistemas de informação, sistemas de redes e os serviços disponibilizados pela rede mundial de computadores. (TUCKER; NOONAN, 2009).

A seguir, apresentam-se os principais aspectos motivadores da escolha do tema desta pesquisa. São descritos alguns aspectos históricos relativos às linguagens de programação, relevantes para o texto em pauta, bem como os objetivos que se pretende atingir.

1.1 MOTIVAÇÃO

O espectro de possibilidades de aplicação de software automodificável é bastante amplo. Entre essas, encontram-se aplicações: para proteção de programas contra engenharia reversa ou ação de pirataria (KANZAKI et al., 2003; MADOU et al., 2006); na construção de compiladores (CAI; SHAO; VAYNBERG, 2007), para melhorar o desempenho de sistemas computacionais, balancear a carga de sistemas operacionais, encriptação dinâmica de código (CAI; SHAO; VAYNBERG (2007).

Técnicas adaptativas são empregadas para a especificação e aceitação de linguagens sensíveis ao contexto (IWAI, 2000), em jogos eletrônicos, no ensino por computador, na engenharia de software, programação evolutiva, inteligência artificial, manipulação e representação do conhecimento, formulação, análise e processamento de linguagem natural, no processamento de sinais, reconhecimento de padrões, diagnóstico, na inferência, tomada de decisão, classificação e muitos outros (NETO, 2007; NETO, 2011).

É crescente o interesse pelo desenvolvimento de código adaptativo, também conhecido como código automodificável. Devido à limitação criada pelos aspectos de projeto dos computadores atuais, baseados na arquitetura de von Neumann (PELEGRINI, 2009), alguns pesquisadores recorrem ao recurso da extensão de linguagens de programação funcional (FREITAS, 2008), outros à sua extensão sintática (CASTRO JÚNIOR, 2009), outros ainda, à construção de ambientes virtuais (PELEGRINI, 2009), como forma de suplantar tais restrições na produção de código que possa se automodificar dinamicamente, em tempo de execução.Na presente dissertação prossegue-se com a busca de uma linguagem de alto nível, com a qual se possa gerar códigos automodificáveis.

Embora seja possível encontrar vários trabalhos relacionados a códigos automodificáveis, como os citados acima, observa-se, no entanto, até mesmo no cenário internacional, a raridade de pesquisas relacionadas a uma linguagem de programação com a qual se possa produzir tais aplicações.

Em nosso país, em especial no Laboratório de Linguagens e Técnicas Adaptativas – LTA (LTA, 2010), pesquisas já desenvolvidas buscam fornecer as bases para a elaboração de uma linguagem que permita programar códigos automodificáveis. Dentre essas pesquisas, destacam-se as que se seguem.

Freitas (2008) mostra a viabilidade de construção de uma linguagem para essa finalidade, e faz um levantamento de especificações com o fim de subsidiar o desenvolvimento de linguagens para a programação de software adaptativo. Entre as diretrizes levantadas estão:

a) quando a linguagem para programação adaptativa for obtida estendendo-se uma linguagem sem essa característica, a linguagem resultante deve continuar apresentando recursos para que sejam realizadas todas as operações originalmente disponíveis na linguagem base (sem as características de adaptatividade);

b) recursos expressivos que permitam que partes do programa em execução possam ser referenciados e modificados de forma dinâmica;

c) em relação à estrutura, é desejável que a linguagem para programação adaptativa esteja 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.

Freitas (2008) faz, ainda, outras recomendações sobre os recursos que uma linguagem desse tipo pode ter: recursos para editar e manusear arquivos, operações de rastreamento, operações de depuração, recursos de auxílio à escrita de programas adaptativos, acesso à camada adaptativa e monitoração do estado do programa em tempo de execução.

Embora a arquitetura da quase totalidade dos computadores atuais não impeça o acesso à memória de código, muitas restrições nesse sentido são impostas pelos sistemas operacionais nelas executados. Essa é uma das dificuldades enfrentadas nos dias atuais para o desenvolvimento de código automodificável. Uma maneira prática de superar essa dificuldade é recorrer ao uso da virtualização do conjunto de instruções da máquina de execução. Esse foi o

caminho seguido por Pelegrini (2009), que construiu um ambiente de execução aderente às necessidades dinâmicas do processamento de código adaptativo.

Foi desenvolvida por esse autor uma linguagem de baixo nível, a linguagem AdaptCode, que tomou como base a linguagem de montagem associada à arquitetura de 32 bits, por ser esta utilizada em muitos computadores domésticos. Dessa linguagem de montagem foi aproveitado um conjunto essencial de comandos básicos, aos quais foram acrescentados os comandos adaptativos. Esse novo conjunto forma uma linguagem que oferece suporte ao processamento de código adaptativo. Foi adotado um conjunto reduzido de comandos visando diminuir a complexidade da proposta. (PELEGRINI, 2009).

Castro Júnior (2009) desenvolveu uma linguagem de programação imperativa, mínima, denominada MINLA (*Minimal Language*), cuja definição foi formalizada e denotada, usando-se para isso, como dispositivo formal, o autômato adaptativo. Foi utilizada a ferramenta AdapTools (JESUS et al. 2007) como ambiente de execução para processar os programas escritos na linguagem proposta. Esta serviu de linguagem hospedeira para uma linguagem extensível, também proposta na mesma tese.

Programas em linguagem de montagem não são tão fáceis de escrever e compreender quanto aqueles escritos em linguagem de alto nível. A criação de uma linguagem de alto nível com a qual se possa escrever programas, a partir de cuja compilação se gere um código que possa ser processado pelo ambiente de execução acima mencionado, torna-se uma alternativa a mais para o programador.. A linguagem desenvolvida permite que programas adaptativos sejam escritos de forma mais fácil, abstraindo muitos dos detalhes formais usualmente necessários à descrição de um fenômeno adaptativo. Para que isso fosse possível, construiu-se um compilador específico, que é o responsável por converter o programa adaptativo escrito em linguagem de alto nível em um código correspondente na linguagem de baixo nível AdaptCode.

A linguagem desenvolvida é constituída por um conjunto de comandos básicos, com semântica semelhante à dos comandos tipicamente encontrados em diversas linguagens imperativas, aos quais se incorporam os comandos adaptativos, que permitem especificar comportamentos dinâmicos de automodificação. Essa

linguagem segue um esquema compatível com o paradigma imperativo estruturado e comporta, também, características particulares, obtidas com o auxílio dos comandos adaptativos.

A subseção seguinte faz um breve retrospecto da produção de código automodificável e do atual interesse pelo emprego desse tipo de código em situações diversas.

1.2 HISTÓRICO

As primeiras linguagens de programação surgiram em meados do século XX, inicialmente as de baixo nível e, posteriormente, as precursoras das atuais linguagens de alto nível. Os recursos computacionais das máquinas de então eram escassos, em particular a área disponível na memória principal. Uma técnica de programação empregada nessa época, visando otimizar o uso de tais recursos, consistia em dotar os programas com a capacidade de se automodificarem enquanto em execução (McKINLEY et al, 2004). Essa prática era facilitada pelo uso extensivo de linguagens de baixo nível e, também, pelo uso limitado, se não ausente, de metodologias de programação, então incipientes.

Por outro lado, códigos desenvolvidos em linguagem de programação de baixo nível geralmente mostram-se difíceis de serem criados, compreendidos, mantidos e depurados. Essa característica fez com que, gradativamente, seu uso fosse se restringindo. Atualmente, têm seu emprego em aplicações muito específicas, que apresentam exigências especiais de desempenho, levando o programador a trabalhar em um nível de granularidade muito fino de programação (ANCKAERT; MADOU; BOSSCHERE, 2007; PELEGRINI, 2009).

O conceito de adaptatividade adotado no presente texto é o que está definido em Neto (2007), o qual classifica como adaptativos os dispositivos capazes de se automodificarem enquanto em execução, motivados por algum estímulo normal de entrada, sem a intervenção de qualquer outro agente externo. Um caso particular, de interesse para este trabalho, corresponde aos programas cujo código pode ser

alterado durante sua execução. Dessa forma, programas dinamicamente alteráveis também se classificam como programas adaptativos, e seus códigos, como códigos adaptativos. Naturalmente, embora se automodifiquem dinamicamente, não se enquadram nesta categoria os softwares cuja alteração, em tempo de execução, não seja intencional e previsível, como é o caso de programas infectados por vírus, ou então daqueles programas cujo código se altere acidentalmente.

Publicações recentes mostram que a automodificação está voltando a ser utilizada, criando a possibilidade de aplicações diversas no desenvolvimento de programas para computador com código adaptativo. Dentre essas publicações, as mais significativas são mencionadas a seguir.

Kanzaki et al. (2003) apresentaram uma forma de realizar a proteção de código contra atos ilegais de pirataria. Eles procuram dificultar a compreensão do programa acrescentando um mecanismo de automodificação de código ao programa original. São selecionados comandos do programa, que são sobrescritos com comandos falsos. Rotinas de restauração e de ocultação dos comandos corretos também são inseridas no programa. Durante a execução, as rotinas de restauração trocam os comandos falsos pelos corretos, estes são executados e a seguir as rotinas de ocultação escondem os comandos corretos pela sobreposição dos comandos falsos. Dessa forma, as instruções corretas somente estão presentes no código durante o instante entre a execução das rotinas de restauração e ocultação desses comandos. Segundo os autores, o método por eles proposto permite produzir programas com baixo grau de vulnerabilidade, sem o uso de *hardware* especial.

Conforme citam Giffin; Christodorescu; Kruger (2005), quando dados são transmitidos através de uma rede, escritos em disco ou quando sofrem alguma outra manipulação, podem ocorrer erros, como a gravação ou a leitura incorreta de um bit, por exemplo. Um erro dessa natureza pode afetar sensivelmente os dados, inclusive torná-los inúteis.

Para a prevenção desse tipo de erro, um processo de verificação de integridade de dados costuma ser realizado antes e depois de determinada operação com os mesmos. Isso pode ser feito utilizando-se a técnica do *checksum*, que consiste em inserir redundâncias no código, de forma que estas, calculadas a

partir do texto original, possam ser recalculadas no momento da verificação, testando-se então se a redundância gravada juntamente com os dados e aquela recalculada coincidem. Caso isso não ocorra, constata-se uma falha, e os dados são considerados não confiáveis. A não ocorrência de tal detecção, no entanto, não assegura que os dados estejam corretos, portanto, embora muito prática, essa técnica deve ser usada com atenção.

Muitas técnicas de *checksum* eficientes já foram desenvolvidas, dentre as quais se destacam os programa de auto-verificação para detecção de modificação maliciosa de código, provocada por vírus, por exemplo. Giffin; Christodorescu; Kruger (2005) apresentam um mecanismo baseado na automodificação de código, que concede maior robustez aos atuais algoritmos de *self-checksumming*. Os programas, atualmente, são construídos partindo-se do pressuposto de que serão executados em uma máquina com arquitetura de von Neumann. Através da criação de uma arquitetura virtual de outro tipo, um ataque não autorizado pode ser bem sucedido, sem ser detectado por funções de *cheksumming*. O principal objetivo atingido pelo trabalho de Giffin; Christodorescu; Kruger (2005), usando a automodificação de código, foi a detecção de violação da hipótese subjacente, ou seja, quando um ataque acontece pela emulação de uma arquitetura virtual de gerenciamento de memória, denominada arquitetura de *Harvard*, na qual dados e instruções em memória são acessados por barramentos distintos. Não foi desenvolvido um novo algoritmo para detecção de ataque por replicação de página de memória empregando automodificação de código. Esta técnica foi empregada para melhorar os algoritmos já existentes de *self-checksumming*. Os autores consideraram que o método seja eficiente, uma vez que, observando testes realizados em três famílias de processadores, foi constatado que acrescenta menos que um microssegundo ao tempo de computação de cada verificação. Eles acrescentam que a detecção de arquitetura de memória pode ser combinada com qualquer técnica de autoverificação, em qualquer plataforma de hardware, o que torna o método possível de ser aplicado em uma grande variedade de situações em que seja necessário assegurar a resistência a modificações.

Tschudin e Yamamoto (2006) ressaltam que nenhum sistema ou programa é perfeito e alguma de suas partes pode falhar. Mesmo os sistemas de missão crítica, que sejam robustos e que tenham sido projetados utilizando técnicas avançadas de

tolerância a falhas, ainda carecem de soluções de programas autônomos que tenham a capacidade de moldar-se a condições anormais e recuperar-se delas e que sejam capazes de sobreviver sem supervisão, por longo tempo, em uma ampla gama de situações adversas.

Outra situação apresentada por Tschudin e Yamamoto (2006), que requer a existência de sistemas autônomos distribuídos, são, por exemplo, as grandes redes, compostas de pequenos componentes, que devem se manter em funcionamento sem supervisão por longos períodos de tempo, ou então veículos espaciais enviados a locais distantes do espaço, onde se torna impraticável qualquer intervenção externa para alguma manutenção que se faça necessária.

Tschudin e Yamamoto (2006) apontam a inexistência de soluções através de programas que considerem memória, comunicação e execução de erros com capacidades de auto recuperação. Eles afirmam que a automodificação de código terá um papel importante nessa área, apesar de, atualmente, ter seu uso restringido pelas dificuldades que apresenta para depuração e correção (ANCKAERT; MADOU; BOSSCHERE, 2007).

Para Anckaert; Madou; Bosschere (2007), um gráfico de controle de fluxo é uma boa forma de representar um programa tradicional (que não se modifica), uma vez que essa representação permite analisar todas as possibilidades do comportamento (estático) de um programa, em tempo de execução, além de ser um mecanismo sobre cuja construção, modificação, compilação e execução já existe uma boa compreensão. No entanto, um gráfico de controle de fluxo tradicional não é adequado para representar códigos automodificáveis. Assim, Anckaert; Madou; Bosschere (2007) introduziram extensões para superar essas limitações, das quais resultou uma forma de representação do código que se mostra mais aderente à sua característica automodificável. Os autores mostram que o esforço desenvolvido por quem tenta atacar um código automodificável é significativamente maior do que quando o código é estático.

Cai; Shao; Vaynberg (2007a) apresentaram um *framework*, denominado GCAP, para verificação de código genérico de máquina de von Neumann com manipulação de código em tempo de execução. É o primeiro *framework* formal capaz de certificar uma forma de manipulação de código em tempo de execução.

Programa certificado, nesse contexto, é um código executável acompanhado de uma prova rigorosa completa de que o software é livre de erro em relação a uma especificação particular, conforme definido em (CAI; SHAO; VAYNBERG, 2007b).

Nessa publicação, os autores usam o dispositivo por eles denominado *General Target Machine* – GTM, para modelar máquinas baseadas no modelo de von Neumann. A GTM é genérica e pode ser usada para modelar arquiteturas modernas de computador, tais como x86, MIPS ou PowerPC. Desenvolveram um sistema baseado na lógica de Hoare para a GTM, o qual analisa o código em memória.

No Brasil, o LTA – Laboratório de Linguagens e Técnicas Adaptativas (LTA, 2010) é pioneiro em pesquisas sobre métodos e técnicas adaptativas. Dentre os diversos trabalhos desenvolvidos no LTA, alguns produziram ferramentas que permitem desenvolver e observar o comportamento dinâmico de dispositivos adaptativos diversos. Destacam-se:

a) RSW, uma ferramenta que permite o reconhecimento e a geração de reconhecedores sintáticos baseados em autômatos de pilha estruturados e em autômatos adaptativos. Aceita, ainda, a descrição formal e promove a simulação de autômatos adaptativos. Esse aplicativo também foi utilizado para fins didáticos, no ensino e no estudo de linguagens de programação e compiladores. Entre suas funcionalidades estão a edição de código-fonte, compilação, detecção e tratamento de erros de compilação, execução e otimização de reconhecedores (PEREIRA; NETO, 1997; PEREIRA, 1999);

b) Adaptools, uma máquina virtual, dotada de interface gráfica, com a qual se pode projetar, implementar e testar autômatos adaptativos. Já sofreu diversas atualizações e aperfeiçoamentos desde sua implementação original. Na versão 2.0, apresenta uma integração com um compilador de autômatos adaptativos, o AdapMap, bem como dá a possibilidade de alteração externa de rotinas semânticas, sem necessidade de recompilar a ferramenta. O AdapMap, através de análises sintática e semântica e buscando reduzir ao mínimo os possíveis problemas na geração de um código-fonte, mostra ao usuário possíveis erros e construções

indesejáveis, produzindo um código-objeto sem erros. Outro recurso disponível é a possibilidade de executar várias máquinas virtuais simultaneamente, as quais podem estabelecer comunicação entre si. Essa faculdade pode ser de grande utilidade para a construção de compiladores que tenham por base a tecnologia adaptativa (CASTRO JÚNIOR, 2009; JESUS et al., 2007; PISTORI; NETO, 2003);

Os exemplos acima mencionados mostram apenas algumas das muitas possíveis aplicações de dispositivos adaptativos.

A próxima seção resume os objetivos da pesquisa desenvolvida nesta dissertação.

1.3 OBJETIVOS

Este trabalho tem como meta principal apresentar uma linguagem de programação de alto nível simples, que permita desenvolver aplicações que incorporem atividades de automodificação em tempo de execução.

Embora ainda não haja um método consagrado para o desenvolvimento de programas automodificáveis na linguagem em questão, é apresentada aqui uma série de recomendações com o objetivo de guiar o programador na construção de programas aderentes à sintaxe da linguagem. É apresentada, também, uma forma gráfica de representar tais programas através de uma representação estratificada. Embora não constituam um método completo, essas recomendações poderão servir como ponto de partida para o estabelecimento de um processo rigoroso e metódico a ser seguido na elaboração de programas adaptativos em geral.

Um programa assim especificado necessita de uma plataforma na qual o correspondente código seja executado podendo promover, durante essa execução, as automodificações dinâmicas previstas. Para isso, foi adotado o ambiente de execução de código adaptativo proposto por Pelegrini (2009), conforme foi mencionado na seção 1.1.

O ambiente de execução em questão é constituído de uma máquina virtual e exibe características específicas, entre as quais se destaca a possibilidade de um código realizar automodificações em tempo de execução. Por conseguinte, o código executável, correspondente ao programa escrito na linguagem de alto nível não pode ser um código convencional, mas necessita ser gerado de modo que possa ser aderente a tais características. Assim, a construção de um compilador que automatize a geração de códigos com essas características se torna mais um objetivo a ser contemplado nesta pesquisa.

Por fim, com a finalidade ilustrativa e de guiar o leitor para que compreenda e domine a técnica é apresentado um exemplo de programa adaptativo, que mostram o emprego da linguagem proposta, o uso da notação gráfica, sua utilização para a construção de representações expressivas dos programas adaptativos desejados, e seu emprego sistemático na fase de projeto desses programas.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação é constituída de cinco capítulos, os quais apresentam os seguintes conteúdos:

O capítulo dois define a terminologia aqui utilizada e apresenta os conceitos necessários para a leitura e o entendimento do restante do texto. São apresentadas ou, ao menos, comentadas as teorias, conceitos e tecnologias relativas às linguagens de programação.

O capítulo três trata da programação adaptativa. São feitas considerações sobre um modelo de programação e sobre um método de desenvolvimento de programas adaptativos, a proposta da linguagem, onde são apresentados, em detalhes, os comandos da linguagem e discutidos seus aspectos sintáticos e semânticos, bem como a sua implementação, sendo, também, feitas considerações sobre o ambiente de execução e a geração de código.

O capítulo quatro traz um exemplo ilustrativo de problema, codificado na linguagem proposta através do qual se procura mostrar a aplicação dessa linguagem

no desenvolvimento de programas diversos, ao mesmo tempo em que vão sendo sugeridos certos aspectos metodológicos da concepção, especificação, representação e elaboração desse tipo de software.

No capítulo cinco apresentam-se as considerações finais, bem como são mostradas as contribuições do presente trabalho para o estudo das técnicas adaptativas. São levantadas, também, diversas possibilidades de trabalhos futuros. Nas conclusões é feito, finalmente, um balanço dos mais importantes proveitos proporcionados pela realização da presente pesquisa.

2. REFERENCIAL

Neste capítulo são discutidas sumariamente as teorias e os conceitos relativos a linguagens de programação e à adaptatividade.

Inicialmente, apresentam-se os conceitos relativos aos dispositivos adaptativos. A seguir, procura-se mostrar os principais conceitos de linguagens de programação, em particular os relativos aos paradigmas de programação, comentando-se, ainda, a influência da extensibilidade das linguagens de programação sobre a atividade de programação. Por fim, são citados aspectos tecnológicos da adaptatividade.

2.1 ASPECTOS TEÓRICOS

Esta seção apresenta o conceito de dispositivos guiados por regras, que é um formalismo geral para dispositivos adaptativos (NETO, 2011), apresenta o conceito de autômatos de pilha estruturados e, ainda, tecem-se considerações acerca de algumas relações que existem entre iteratividade, recursividade, macros e adaptatividade.

2.1.1 Dispositivos guiados por regras

Um dispositivo guiado por regras, de acordo com Neto (2001), é composto por um conjunto finito de configurações e um conjunto finito de regras, as quais mapeiam o comportamento do dispositivo, desde a sua configuração inicial até alguma configuração final, em função de uma sequência de estímulos recebidos. O dispositivo é dito determinístico se, e somente se, para cada par configuração-estímulo, o conjunto de regras determina univocamente a próxima configuração. Será não-determinístico caso contrário.

No caso de ser um dispositivo reconhecedor, após processar toda a cadeia de entrada, o dispositivo atingirá uma configuração, na qual aceitará ou rejeitará a entrada.

Quando o conjunto de regras que compõem um dispositivo é estático, ou seja, inalterável para qualquer possível entrada, diz-se que tal dispositivo é não-adaptativo.

Ao dotar um dispositivo guiado por regras com a capacidade de alterar dinamicamente seu próprio conjunto de regras, obtém-se um dispositivo adaptativo guiado por regras. Essas alterações do conjunto de regras são promovidas por ações adaptativas, as quais são aplicadas a cada passo de operação do dispositivo, podendo ocorrer antes ou depois do processamento do estímulo que provocou tal passo de operação.

Em um dispositivo adaptativo, formalmente constituído, é possível distinguir duas componentes: um dispositivo subjacente, não adaptativo e o mecanismo adaptativo, que confere ao dispositivo subjacente a capacidade de realizar automodificações (NETO, 2007; NETO, 2011).

Um exemplo de dispositivo guiado por regras é o autômato adaptativo, cuja formalização e uma explicação detalhada do seu funcionamento são encontradas em Neto (1993), onde são apresentadas soluções para diversos problemas da área da construção de compiladores para linguagens de programação, empregando esse dispositivo formal de implementação.

Ainda em relação a dispositivos guiados por regras, os comandos e declarações que compõem uma linguagem de programação podem ser considerados como regras, as quais, quando dispostas na forma de um programa e executadas, determinam o comportamento desse programa, o qual, partindo de uma situação inicial e recebendo algum estímulo, na forma de dados de entrada, vai evoluindo e assumindo sucessivas configurações até atingir uma configuração final. Desse ponto de vista, um programa escrito em alguma linguagem de programação, pode ser considerado também um dispositivo guiado por regras.

Acrescentando-se a uma linguagem de programação mecanismos que permitam escrever programas capazes de realizar alterações dinâmicas do próprio

código, obtém-se uma linguagem para programação de códigos adaptativos e os programas nela denotados constituirão, potencialmente, programas com código adaptativo.

Um programa adaptativo pode ser entendido como uma especificação de uma sequência automodificável de instruções, que representa um código dinamicamente alterável. À luz da formulação geral apresentada em Neto (2001), programas dessa classe podem ser formalmente considerados como sendo dispositivos adaptativos especiais, cujo dispositivo subjacente não-adaptativo seria um programa estático (este, combinando um conjunto de regras).

Em um programa adaptativo, as ações adaptativas podem remover ou acrescentar linhas de código a esse programa, antes ou depois de processar um estímulo, constituindo, dessa forma, ações adaptativas anteriores ou posteriores, respectivamente.

Uma gramática, como dispositivo gerador de uma linguagem de programação usual, caracteriza-se por apresentar um conjunto de terminais, um de não terminais e um conjunto estático de regras de produção. Associando-se ações adaptativas às regras de produção de uma gramática, esta adquire a capacidade de alterar o seu conjunto de não terminais e de regras de produção, durante a geração de uma sentença, o que torna a gramática adaptativa (IWAI, 2000), com a possibilidade de acionar ações adaptativas que promovam consultas, remoções ou adições de regras de produção.

O mais antigo dispositivo guiado por regras que inspirou a concepção de um dispositivo adaptativo foi uma máquina de estados com pilha, denominada autômato de pilha estruturado, apresentado a seguir.

2.1.2 Autômato de pilha estruturado

O autômato de pilha estruturado pode ser visto como um conjunto de autômatos finitos chamados submáquinas, onde cada qual responsável por realizar o reconhecimento de cada uma das partes da cadeia de entrada com características

que as tornam uma subclasse dessa cadeia (NETO, 1993). Isso significa que cada grupo de estruturas com características semelhantes pode ser reconhecido por uma submáquina diferente.

O comportamento de um autômato de pilha estruturado é, basicamente, o mesmo de um autômato finito, exceto no que se refere à chamada e retorno de submáquina e à operação da pilha. A chamada e retorno de submáquina é representada por transições especiais, e se processa de forma similar ao que ocorre nas chamadas e retornos de subprogramas, em grande parte das linguagens de programação. Uma chamada de submáquina determina o empilhamento do estado de retorno, da mesma forma que um retorno de submáquina efetua o desempilhamento do estado para o qual a execução do autômato será desviada. O emprego desse mecanismo de divisão do programa confere a esse tipo de reconhecedor uma alta eficiência no processo de reconhecimento de uma sentença. Dessa forma, autômatos de pilha estruturados podem ser empregados eficientemente na construção de compiladores.

Uma linguagem de programação é constituída por componentes léxicos e sintáticos. Uma das tarefas de um compilador é verificar se um programa fonte, redigido em uma determinada linguagem de programação, está escrito de acordo com as especificações sintáticas da linguagem. Essa verificação é feita através da análise léxica e da análise sintática. O reconhecimento das componentes léxicas de um programa fonte é uma tarefa relativamente simples, pois pode ser sempre modelada na forma de uma máquina de estados finita determinística. No entanto, a tarefa de analisar a correção de um programa fonte do ponto de vista da sintaxe envolve um grau de complexidade maior que a análise léxica, pois não se trata apenas de verificar se cada componente léxico está sintaticamente colocado, formando sentenças sintaticamente aderentes à especificação sintática dada pela gramática da linguagem. O autômato de pilha estruturado pode ser empregado para modelar uma aproximação livre de contexto da linguagem de programação, nessa fase da compilação. Neste, estruturas de tipos diferentes podem ser reconhecidas por submáquinas distintas. Uma expressão aritmética, por exemplo, é tratada por uma submáquina, enquanto uma declaração de variáveis é reconhecida por outra. Aninhamentos podem ser tratados facilmente com o uso de chamadas e retornos de submáquinas.

Programas podem ser expressos através de diversos recursos sintáticos básicos comuns nas linguagens usuais de programação, tais como iteração, recursão, na forma de funções e de macros. A próxima seção procura mostrar a possível equivalência existente entre esses recursos de programação, bem como a possível relação deles com a adaptatividade.

2.1.3 Funções, recursividade, macros, iteração e adaptatividade

Esta seção mostra os princípios que regem o funcionamento dos programas desenvolvidos segundo as técnicas de programação iterativa, de programação recursiva, da programação usando macros e de programação adaptativa, procurando estabelecer uma relação entre tais técnicas.

Um processo iterativo é caracterizado por um conjunto de variáveis de estado, um conjunto fixo de regras que decide a forma como essas variáveis são atualizadas e como o processo muda de um estado para outro, e um teste condicional que determina quando o processo deve ser finalizado. As variáveis de estado indicam, a cada instante, a situação da execução de um processo iterativo. No caso de este ser interrompido, por exemplo, basta realizar um exame de suas variáveis para se ter conhecimento da posição do programa no instante em que ocorreu tal parada. Em outras palavras, para prosseguir a execução de um processo interrompido é suficiente conhecer o valor das variáveis de estado para que o processo possa prosseguir a partir do ponto em que havia parado (ABELSON; SUSSMAN, 1996).

A recursão é bastante empregada em programação para a resolução de problemas. A ideia básica da recursão é a de dividir o processamento do problema inicial em um conjunto de subproblemas menores, porém muito similares ao problema original até que seja obtido um subproblema tão simples, que sua solução seja conhecida *a prioiri*, e que é denominado caso-base. A partir dessa solução trivial do caso-base, é possível obter, sem dificuldade, a solução do problema imediatamente mais complexo, a qual, por sua vez, pode ser usada com facilidade para obter a seguinte, e assim por diante. O processo segue retroagindo e obtendo a

solução de subproblemas cada vez mais distantes do caso-base até que se chegue, finalmente, à solução do problema original (SKLIAROWA; SKLYAROV, 2009).

Em algumas situações, é necessário resolver todos os subproblemas para, só então, chegar à solução do problema inicial. Um exemplo é o cálculo de um termo da sequência de Fibonacci, cuja solução recursiva gera uma árvore, na qual podem ser identificadas redundâncias, o que torna ineficiente a solução. Em outras situações isso não ocorre, sendo necessário resolver apenas uma parte dos subproblemas para que a solução seja atingida, como acontece na resolução do problema de busca binária.

Os processos recursivos contêm informações a respeito do estado do programa que não estão presentes em variáveis de programa, como acontece em processos iterativos, de forma que, havendo uma parada, não é possível determinar o ponto de execução em que se encontrava o processo quando foi interrompido. (ABELSON; SUSSMAN, 1996).

O controle de um procedimento recursivo é realizado com o auxílio de uma pilha, que varia em função do número de chamadas em andamento dos procedimentos do programa. Quando tal número é grande, aumenta o *overhead* causado pelo uso dessa área de armazenamento, podendo até, num caso extremo, esgotar a memória disponível para isso. No entanto, existem muitos bons argumentos em favor do uso da recursão. Por um lado, o seu emprego pode tornar mais claros programas complexos, bem como mais eficientes, de forma que, em tais casos, se pode ignorar o *overhead*. Em outros casos, soluções recursivas podem requerer uma movimentação maior de dados deixando o programa menos claro e mais lento (SKLIAROWA; SKLYAROV, 2009).

É conhecido o fato de que haja uma equivalência entre programas recursivos e programas iterativos. A conversão de programas escritos em estilo recursivo para a forma de programas iterativos e vice versa foi empregada com frequência no passado (BIRD, 1979; LIU; STOLLER, 2000) e continua a suscitar pesquisas nos dias atuais (SKLIAROWA; SKLYAROV, 2009).

A transformação de um procedimento recursivo em iterativo elimina a pilha e, consequentemente, não apenas o espaço por ela consumido durante a execução do

programa como também o tempo empregado para essa finalidade. Liu; Stoller (2000) apresentaram um método de transformação de recursão em iteração, que pode ser automatizado.

Através do uso de macros, um procedimento (trecho de programa) pode ser especificado apenas uma vez em um programa para ser repetidamente utilizado no mesmo, tantas vezes quantas forem necessárias, através de chamadas de macros. A exemplo do que ocorre com procedimentos do tipo função ou sub-rotina, sempre que conveniente, uma maior versatilidade pode ser obtida para as macros utilizando-se parâmetros na sua especificação e utilização.

A utilização de macros se caracteriza por permitir a denotação do programa por meio de chamadas compactas das macros, as quais são expandidas em tempo de compilação, sendo, então, substituídas pelo texto que tais chamadas abreviam. Isso pode proporcionar ganhos no tempo de execução do programa em relação ao caso equivalente que faça uso de subprogramas, uma vez que, nestes últimos, a execução dos mecanismos das chamadas, retornos e, eventualmente, passagem de parâmetros operam em tempo de execução, o que os podem tornar mais lentos.

Por outro lado, o uso indiscriminado de macros costuma causar um aumento considerável no tamanho do código do programa, em decorrência da excessiva repetição de código das macros no mesmo, especialmente quando o código associado à macro for extenso.

Conforme Tucker; Noonam (2009), subprogramas, por sua vez, são procedimentos fechados, representados por um código único no programa, de forma que não provocam o surgimento de instâncias múltiplas do mesmo, a despeito de seu eventual uso em diversos pontos do programa.

No entanto, é possível utilizar indiferentemente macros, subprogramas ou técnicas adaptativas nas situações em que as correspondentes abstrações forem instanciadas diversas vezes, quer em tempo de compilação, quer em tempo de execução.

No caso dos subprogramas (funções ou subrotinas), as chamadas são inseridas em todos os pontos do programa em que a abstração for necessária, mas o código que a define terá uma única instância.

No caso das macros, suas chamadas, que ocorrem também ao longo do programa, são substituídas, em tempo de compilação, por instâncias que são cópias do código que as definem, cada qual personalizada conforme o valor dos argumentos passados em sua chamada.

No caso do uso de técnicas adaptativas, a instanciação dessas abstrações é feita somente se necessário, quando a execução do comando correspondente for, de fato, acionada em tempo de execução.

Na programação adaptativa é possível explorar, na prática, a automodificação dos programas com a finalidade de obter diversos efeitos na lógica do programa, incluindo aqueles que têm comportamento similar aos proporcionados usando-se iterações, recursões ou chamadas de macros.

Da mesma forma como ocorre com a programação funcional, que se apoia em modelos de computação baseados em funções recursivas ou no cálculo lambda (TUCKER; NOONAN, 2009), a operação dos programas desenvolvidos com base em técnicas adaptativas também pode usar a mesma fundamentação para apoiar métodos e técnicas de desenvolvimento de procedimentos com código adaptativo que desempenham atividades similares.

A seção a seguir aborda algumas das principais características das linguagens de programação.

2.2 LINGUAGENS DE PROGRAMAÇÃO

As linguagens de programação, assim como quaisquer outras linguagens, são concebidas para operarem como instrumentos de comunicação e, como tal, permitem a troca de informações entre entidades. Enquanto as linguagens naturais se prestam para a comunicação entre pessoas, as linguagens de programação são projetadas para a comunicação entre os humanos e o computador. Outro aspecto das linguagens de programação é o reduzido domínio de expressão, pois necessitam promover apenas a comunicação de um limitado conjunto de conceitos computacionais (SEBESTA, 2010).

Os primeiros computadores foram usados para o processamento de aplicações científicas, as quais, quase sempre, utilizam estruturas de dados relativamente simples e uma elevada quantidade de cálculos aritméticos em ponto flutuante.

Desde o seu surgimento, na década de 1950, até os dias atuais, esforços vêm sendo empreendidos no desenvolvimento de boas linguagens de programação de alto nível, motivados por diversos fatores. No início, buscava-se superar as dificuldades de utilização impostas pelas linguagens de baixo nível. Surgiu, então, a primeira linguagem de alto nível, denominada FORTRAN – *Formula Translation*. Esta linguagem evoluiu ao longo dos anos, recebendo aperfeiçoamentos em diversas versões e atualizações, e permanece em uso intenso até os dias atuais, encontrando aplicações, principalmente, em problemas de natureza científica.

A seguir surgiram outras linguagens, que foram projetadas para atender necessidades diversas. Entre essas devem ser destacas as linguagens COBOL, Lisp, Algol e BASIC, contemporâneas do FORTRAN e que, a seu exemplo, ainda hoje se utilizam e também linguagens mais recentes, tais como C, C++, Java e inúmeras outras linguagens para o desenvolvimento de programas de propósito geral (LÉVÉNEZ, 2011).

Posteriormente, foram aparecendo outras motivações para o surgimento de novas linguagens, entre elas a necessidade de modelagem de problemas oriundos de diversas áreas específicas de pesquisa, cujas características lhes são peculiares, e que não eram apoiadas por recursos de expressão nas linguagens então disponíveis, como é o caso da inteligência artificial. Essa área de conhecimento, inicialmente atendida pela linguagem LISP, passou a valer-se, mais recentemente, do paradigma da programação em lógica. Assim, com foco nos paradigmas funcional e lógico, foram criadas diversas linguagens de programação especializadas para o desenvolvimento de software para Inteligência Artificial, entre as quais se podem mencionar Prolog, Scheme, ML, Haskel, e muitas outras (LÉVÉNEZ, 2011).

A grande maioria dos projetos das máquinas das últimas décadas exibe a arquitetura do clássico modelo de von Neumann. Tal aspecto acarreta uma certa ineficiência na execução de programas escritos em linguagens não imperativas,

como as acima citadas, podendo dificultar a essas linguagens atingirem um nível de popularidade maior.

A seção a seguir abordará os paradigmas de programação que apresentam maior desenvolvimento nas últimas décadas.

2.2.1 Paradigmas de programação

Um paradigma de programação diz respeito a um padrão de resolução de problemas empregando determinado gênero de linguagens de programação (TUCKER; NOONAN, 2009).

Os diversos paradigmas de programação costumam induzir nas linguagens a eles aderentes características particulares que facilitam a expressão de ideias, conceitos e práticas de programação específicos, de forma que suas funcionalidades e objetivos possam ser atendidos mais adequadamente.

De acordo com Tucker; Noonan (2009), a partir dos anos 70 quatro desses paradigmas de programação ganharam destaque: programação imperativa, orientada a objetos, funcional e lógica. Serão descritas, a seguir, sempre com fundamento em Tucker; Noonan (2009), as características principais desses quatro paradigmas de programação:

a) Programação Imperativa

O paradigma imperativo apóia-se na base teórica proporcionada pela Máquina de Turing, tendo como principal lastro tecnológico, a arquitetura de von Neumann. Essa arquitetura leva os programas a terem como recurso central de armazenamento de informação valores armazenados em memória, em forma de estruturas de dados, ou seja, agrupamento de variáveis. As instruções do programa costumam também ser organizadas em posições lógicas contíguas de memória, o que pode tornar mais eficiente o processamento (PELEGRINI, 2009).

Nesse paradigma, a ideia central é o conceito de estado de um programa, materializado na configuração da memória do programa e dos seus dados, e que é

alterado durante a execução do mesmo através de sucessivas modificações dos valores das variáveis, impostas pelas instruções do programa sobre o seu estado.

Essa função é desempenhada, principalmente, pelos comandos de atribuição, que alteraram o estado de um programa através da substituição de um valor, contido em uma posição de memória, por algum outro valor.

Quando uma linguagem é capaz de fornecer recursos adequados que permitam a implementação de qualquer algoritmo que possa ser projetado, essa linguagem se diz Turing-Completa. Dessa forma, uma linguagem de programação imperativa que disponibilize variáveis e valores inteiros, as operações aritméticas básicas, comandos de atribuição, comandos condicionais e iterativos é considerada Turing-completa (SEBESTA, 2010).

Além dos comandos de atribuição, as linguagens de programação imperativas costumam disponibilizar ao programador: declarações de variáveis, expressões, comandos condicionais, comandos iterativos e abstrações procedimentais.

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

O refinamento gradual de abstrações é uma maneira sistemática muito empregada no desenvolvimento de programas e de muitas outras modalidades de sistemas computacionais (técnica dos refinamentos sucessivos). Usando abstração procedimental, um programador pode particionar a lógica de uma função, idealizada de forma macroscópica, em um grupo de funções mutuamente dependentes, mais simples e/ou mais específicas.

Com o fim de simplificar o desenvolvimento de algoritmos complexos, as linguagens imperativas modernas oferecem ao programador suporte a matrizes e estruturas de registro, além de bibliotecas extensíveis de funções, que, facilitando o reaproveitamento de partes já desenvolvidas de programas, evitam que operações comuns necessitem ser reprogramadas, como é o caso de operações de entrada e saída de dados, gerenciamento de memória, manipulação de cadeias, estruturas de dados clássicas, etc.

São exemplos de linguagens imperativas, entre inúmeras outras, as linguagens FORTRAN e C.

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

b) Programação Orientada a Objetos

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

A abstração de dados encapsula os tipos de dados e as respectivas funções em um único bloco ou pacote. Mas, embora eficaz e potente, o encapsulamento não proporciona ao bloco (ou pacote), um mecanismo prático de inicialização e finalização de um valor, nem uma forma simples de incorporação de operações adicionais.

A orientação a objetos envolve a utilização de conceitos, tais como o de classes, herança e polimorfismo, devendo-se notar, no entanto, que nem todas as características teóricas que a definem, são obrigatoriamente incorporadas às implementações existentes das linguagens que seguem esse paradigma.

Em programas orientados a objetos, as classes existem dentro de uma hierarquia. Uma classe pode ser subclasse de outra e esta será, neste caso, considerada sua superclasse. Assim, uma subclasse poderá herdar de sua superclasse variáveis e métodos.

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

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

As linguagens Smalltalk e Java são, basicamente, orientadas a objetos. A linguagem C++, embora classificada por muitos como linguagem orientada a objetos, é uma linguagem de paradigma híbrido, uma vez que possibilita o uso simultâneo, em um mesmo programa, de técnicas imperativas e orientadas a objetos.

c) Programação funcional

O paradigma funcional tem como fundamentação teórica o cálculo Lambda, que neste caso desempenha um papel similar ao das máquinas de Turing no paradigma imperativo. Devido ao uso extensivo da recursão em programas funcionais, esta técnica se apoia, tecnologicamente, em arquiteturas reais ou virtuais, nas quais o programador possa, de forma relativamente natural, explorar a recursão usada para exprimir e executar a lógica de seus programas.

O paradigma funcional baseia-se, fundamentalmente, no uso do conceito de função, para exprimir a lógica dos programas explorando, como uma de suas principais características, o conceito de transparência referencial, que se traduz, na prática, pela ausência de efeitos colaterais.

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

As estruturas típicas encontradas em uma linguagem funcional são:

- Um conjunto de dados, que são representados por estruturas de alto nível, por exemplo, através de listas, como ocorre na linguagem LISP.

- Um conjunto de definições de funções primitivas preconstruidas, destinadas, principalmente, à manipulação dos dados.

- Especificações de aplicações das funções, que podem ser formas funcionais para construção de novas funções.

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

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

A linguagem LISP foi a primeira linguagem de programação concebida integralmente com base teórica fundamentada no cálculo lambda. Nos dias de hoje, os inúmeros dialetos dessa linguagem incorporam características de outros paradigmas. Outro exemplo de linguagem funcional é a linguagem ML.

d) Programação Lógica

Este paradigma tem sua fundamentação teórica na Lógica de Primeira Ordem (SEBESTA, 2010). A recursão também é extensivamente empregada na programação em lógica, cuja implementação exige, tecnologicamente, arquiteturas (reais ou virtuais) que implementem eficientemente esse recurso de programação.

Do ponto de vista lógico, o paradigma lógico se baseia na programação declarativa, a qual privilegia a especificação da particular tarefa que se deseja que o computador execute, evitando que o programador seja obrigado a indicar a forma como o computador deverá proceder para realizá-la, ao contrário do que ocorre na programação imperativa.

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

Para tanto, um programa lógico costuma efetuar processamento simbólico, não numérico. Programas escritos no paradigma lógico costumam usar como linguagem de programação alguma implementação tecnológica da lógica simbólica.

A exemplo do que acontece com programas expressos em paradigma funcional, programas lógicos costumam denotar formalmente de maneira idêntica programas e dados, não fazendo distinção entre argumentos de entrada e de saída.

Assim como os programas escritos em paradigma funcional, as linguagens voltadas à programação em lógica não costumam oferecer ao programador

comandos de controle para especificar repetições, os quais são extensivamente substituídos por formas recursivas equivalentes.

Um programa, escrito em alguma linguagem de programação de paradigma lógico, tal como Prolog, Planner e Oz, é tipicamente constituído de cláusulas, que podem ser de dois tipos: fatos, que são considerados verdadeiros por definição, e regras, a serem aplicadas sobre fatos e sobre outras regras, e cuja avaliação pode resultar verdadeira ou falsa.

2.2.2 Extensibilidade

Linguagens extensíveis permitem ao programador utilizar um conjunto inicial de instruções e um mecanismo de extensão, de forma que, com isso, seja possível ao programador utilizar as instruções disponíveis e também criar novas instruções a partir dessas, facilitando, assim, a inserção, na linguagem de programação, de características específicas para facilitar a solução de classes particulares de problemas de seu interesse, sem a necessidade de criar um novo compilador (ALGHAMDI, URBAN 1973).

As primeiras publicações científicas sobre a extensibilidade de linguagens de programação (STANDISH, 1975), ocorreram ainda na década de 60 e prosseguiram nos anos 70, inclusive com a realização de simpósios sobre linguagens extensíveis nos anos de 1969 e 1971.

Na ocasião desse simpósio, os pesquisadores foram tomados por uma certa euforia em relação às possibilidade de um usuário poder utilizar a extensibilidade de linguagens de forma rápida e a baixo custo, em aplicações em diversas áreas. Nos anos seguintes foram reveladas algumas dificuldades com a extensibilidade de linguagens, como por exemplo, derivar da filosofia do “faça você mesmo”, que não interessa a alguns usuários pelo grande trabalho que pode dar, ou por resultar, com certa freqüência, em soluções longas, deselegantes e ineficientes (STANDISH, 1975).

Nos últimos anos tem-se renovado o interesse pela extensibilidade de linguagens (WILSON, 2004). Uma linguagem extensível é formada por duas componentes essenciais (SCHUMAN, 1970):

a) uma linguagem base, constituída por um conjunto de comandos organizados de maneira a formarem uma linguagem coerente;

b) um conjunto de mecanismos de extensão, através dos quais novas construções linguísticas possam ser definidas a partir das já existentes.

A extensibilidade de linguagens de programação pode ocorrer de três formas: extensibilidade léxica, sintática e funcional.

2.2.2.1 Extensibilidade léxica

Extensibilidade léxica, segundo a qual criam-se abreviaturas, paramétricas ou não, para trechos específicos de texto, de tal modo que, toda vez que tal abreviatura for encontrada, seja usado, em substituição à abreviatura, o trecho de texto a elas associado, devidamente personalizado pelo programador através dos argumentos que instanciam os parâmetros em cada chamada dessa abreviatura. O mesmo mecanismo é empregado para tal personalização, que se faz pela substituição dos parâmetros pelos argumentos, reinstanciados pelo programador a cada chamada dessas abreviaturas.

A ação desse tipo de extensibilidade se dá sobre o programa, em tempo de compilação.

A Linguagem de programação C costuma ser implementada como linguagem lexicamente extensível, uma vez que permite a criação de abreviaturas textuais (DEFINE). A esse mecanismo de declaração e uso subsequente dessas abreviaturas costuma-se dar o nome de mecanismo de declaração e chamada de macros (léxicas).

2.2.2.2 Extensibilidade funcional

Uma linguagem de programação pode ser estendida através da incorporação de novos verbos que representem abstrações inexistentes originalmente na linguagem-base. A esse tipo de ampliação dá-se o nome de extensibilidade funcional, presente em todas as linguagens de programação minimamente expressivas.

A forma mais simples de extensibilidade funcional corresponde à possibilidade, proporcionada pela linguagem ao programador, de definir e utilizar abstrações na forma de procedimentos, fechados (sub-rotinas e funções, eventualmente paramétricas) ou abertos (macros léxicas, eventualmente paramétricas), como ocorre nas linguagens Fortaran, Lisp, C, entre outras.

Na década de 60, a linguagem Simula 67 introduziu, através das primeiras realizações de um grande conjunto de importantes conceitos, então inéditos, ligados à abstração em programas, que viriam a manifestar-se mais tarde através dos conceitos de classes, encapsulamento de tipos, heranças, polimorfismo e outros conceitos que hoje podem ser considerados como característicos da programação orientadas a objetos, e que caracterizam as modernas linguagens orientadas a objetos (DAHL; NYGAARD, 1966).

Uma das formas de extensibilidade funcional consiste em encapsular novas funcionalidades a funções definidas e tradicionalmente usadas de uma linguagem de programação, adicionando-lhe novas funcionalidades (RUDIGER, 1999).

LISP é um exemplo de linguagem funcionalmente extensível.

2.2.2.3 Extensibilidade sintática

A extensibilidade sintática diz respeito à capacidade de uma linguagem de programação permitir a adição de novos constructos sintáticos, sendo que estes

geralmente podem ser combinados com os já existentes (ALGHAMDI; URBAN, 1973).

As linguagens que não permitem esse tipo de recurso costumam ser definidas através de gramáticas essencialmente estáticas, de forma que o programador não precisa se preocupar com essa gramática, embora seja obrigado a escrever seus programas observando as regras sintáticas por ela definidas. Entretanto, se uma linguagem for sintaticamente extensível, essa situação muda. Essa propriedade costuma ser implementada com a ajuda de um conjunto de funções adequadamente embutidas na linguagem, cujas ações alteram a sintaxe e, portanto, a gramática dessa linguagem. Dessa forma, um usuário da linguagem tem a capacidade de especificar alterações dinâmicas em sua sintaxe na ocasião da codificação de um programa em tal linguagem (SCHUMAN, 1970).

A definição de macro sintática difere de outro tipo de declaração na medida em que ela é uma função que atua diretamente sobre a gramática, ou seja, ela permite modificar a especificação sintática da linguagem. Ao reconhecer uma macro sintática, o *parser* da linguagem de programação expande a chamada das macros sintáticas em estruturas sintáticas definidas na linguagem de programação original (RIEHL, 2008).

2.3 TECNOLOGIA

A especificação de linguagens de programação é feita, principalmente, contemplando os seus aspectos sintáticos e semânticos. Para que possa produzir seus efeitos em um ambiente de execução, um programa necessita ser interpretado ou compilado. Esta seção descreve, conceitualmente, tais aspectos.

2.3.1Especificação de sintaxe

O estudo da sintaxe das linguagens de programação diz respeito aos sistemas formais utilizados para definir a lei de formação que rege a construção da sequência de símbolos que compõem as suas sentenças sem considerar os seus significados. Os dois principais grupos de formalismos usados para essa finalidade são os dispositivos reconhecedores (como as máquinas de estados) e os geradores (gramáticas, expressões regulares, etc.) (SEBESTA, 2010; SLONNEGER, KURTZ, 1995).

Em uma linguagem definida por meio de reconhecedores, uma cadeia de entrada é submetida para ser reconhecida ou rejeitada pelo formalismo reconhecedor. Caso a cadeia de entrada tenha sido escrita totalmente de acordo com as regras sintáticas que definem a linguagem, ela será aceita. Caso o reconhecedor constate a presença de alguma parte dessa cadeia que não esteja em conformidade com as leis de formação da linguagem, a cadeia será rejeitada. Este é um dos fenômenos que ocorrem ao submeter-se um programa fonte ao compilador da respectiva linguagem (SEBESTA, 2010).

Ao definir uma linguagem através de dispositivos geradores, estes se comportam como um mecanismo através do qual é possível construir sentenças pela aplicação de alguma sequência adequada de suas regras. Formalismos de geração que assim operam, que encontram largo emprego na formalização de linguagens, denominam-se gramáticas.

Da mesma maneira que acontece com as linguagens naturais, uma gramática de uma linguagem artificial de programação pode ser usada para disciplinar a maneira como os átomos da linguagem podem ser combinados para formar as respectivas sentenças.

Os elementos significativos da linguagem, conhecidos, por simplicidade, como átomos (*tokens*), normalmente costumam ser descritos em formalização própria por intermédio de uma especificação léxica, à parte da descrição sintática.

Embora a apresentação da sintaxe das linguagens de programação, na década de 1970, tomasse como base a notação BNF (BAKUS, et AL, 1963), frequentemente as novas linguagens eram apresentadas em diferentes variantes da notação BNF, fato que motivou Wirth (1977) a apresentar uma notação equivalente, simples, que, segundo ele, mostrava-se satisfatória.

A sintaxe da meta linguagem apresentada por Wirth (1977), descrita por ela própria, é mostrada no Apêndice A.

Essa notação é adotada neste trabalho para a formalização da linguagem para programação adaptativa, apresentada no capítulo três.

2.3.2 Especificação de semântica

Por terem poder de expressão para definirem linguagens livres de contexto, algumas características, embora genuinamente sintáticas, das linguagens de programação não podem ser expressas em BNF nem na notação de Wirth.

Dessa forma, tais características não podem ser verificadas diretamente pela análise sintática, embora possam ser resolvidas adequadamente em tempo de compilação. São as chamadas dependências de contexto, também denominadas regras de semântica estática. Um exemplo dessas regras é a verificação de tipos de uma linguagem imperativa (SEBESTA, 2010).

Adicionalmente, o termo semântica, também usado para denominar a semântica dinâmica, está relacionado com o significado das construções sintáticas a que se refere, e não pode ser verificada em tempo de compilação, já que o comportamento das construções sintáticas da linguagem depende do valor dos dados no momento da sua execução.

A análise semântica é, portanto, realizada em dois momentos: durante o processo de compilação, quando não depende de uma entrada, (a semântica estática) e em tempo de execução (a semântica dinâmica).

A semântica estática busca descrever as restrições estruturais da linguagem que, embora não sejam abrangidas pela representação livre de contexto, podem, entretanto, ser verificadas em tempo de compilação.

A semântica dinâmica diz respeito aos significados das cadeias sintaticamente válidas dessa linguagem, seu comportamento em tempo de execução, seus relacionamentos mútuos e suas aplicações nos programas escritos nessas linguagens. Através dessa semântica dinâmica, é possível descrever o comportamento esperado de um computador ao executar um programa nessa linguagem (SLONNEGER, KURTZ, 1995).

Um programa, escrito em uma linguagem de programação, necessita ser traduzido para uma forma que possa ser processada pelo ambiente de execução, para produzir os seus efeitos. Essa tradução pode ser feita por um interpretador ou por um compilador. Um interpretador captura e verifica uma sentença, executando-a a seguir. Um compilador, por sua vez, após analisar todo o programa, produz uma versão do mesmo traduzida para a linguagem reconhecida pelo ambiente de execução, entre os quais estão as máquinas virtuais, que emulam o comportamento da máquina ideal de execução para a linguagem.

Um compilador classicamente executa, sobre um programa fonte, análises léxica, sintática, semântica e a geração de código. Essas fases podem ser executadas em sequência, uma após a outra, com o programa fonte sendo analisado desde o início, em cada uma delas. Em muitos compiladores, a arquitetura dos mesmos em *pipeline* permite que resultados de uma fase prévia sirvam de entrada para a fase subsequente.

Compiladores implementados segundo essa orientação são ditos compiladores de múltiplos passos. Já compiladores de passo único passam uma informação por vez diretamente para seus diversos elementos, não necessitando gravar resultados intermediários do processo, produzidos em cada fase (WATSON, 1989).

Decompor a tarefa de compilação em suas fases distintas torna-se vantajoso: simplifica a escrita, depuração e alteração, além de, potencialmente, reduzir os requisitos de recursos computacionais. Outra importante vantagem é tornar

independente de máquina o processo de análise, confinando tal dependência apenas ao módulo de geração do código (WATSON, 1989).

Apresenta-se, a seguir, uma breve visão de cada uma das fases clássicas da compilação.

a) Análise léxica

Um exemplo de sentença de uma linguagem natural, da língua portuguesa por exemplo, com sentido completo, pode ser formada por um sujeito e um predicado. O predicado tem, obrigatoriamente, um verbo e, se necessário, também pode ter complementos.

A análise léxica de uma sentença na língua portuguesa se encarrega de identificar as palavras e verificar se as mesmas pertencem ao léxico da linguagem e determinar a que classe de palavras pertence. De forma análoga, é isso que faz o analisador léxico de um compilador: identifica e classifica os átomos do programa fonte.

Os átomos significativos são denominados lexemas. Durante a análise léxica, estes são classificados em categorias, formando os ***tokens***. Um ***token*** é composto por um nome e um atributo, da forma:

<nome, atributo>,

onde nome indica a classe de ***token***, por exemplo “identificador” e o atributo indica um valor associado ao ***token***. Normalmente, por questão de uniformidade, a uma palavra reservada ou a sinais de pontuação, e a outros elementos simples e únicos da linguagem associa-se um ***token*** sem atributo, ou um ***token*** cujo atributo é igual à respectiva classe.

A construção de um analisador léxico é bastante simplificada pela natureza das cadeias de entrada que necessita reconhecer. Após a análise léxica, é procedida a análise sintática.

b) Análise sintática

Retomando a analogia anterior, a análise sintática de uma língua natural classificaria cada parte da sentença em sujeito, predicado, objeto, predicativo,

adjunto adnominal, e assim por diante, e verificaria se a ordem desses termos está de acordo com a gramática da língua.

Em se tratando de um analisador sintático de uma linguagem de programação, de forma semelhante, é realizada uma verificação que busca determinar se a cadeia de *tokens*, extraída da sentença pelo analisador léxico, está construída sintaticamente de acordo com a respectiva gramática.

Diversas técnicas são empregadas na obtenção de analisadores sintáticos. Alguns dessas técnicas se baseiam na construção de uma árvore de derivação em que a sequência de *tokens*, recebida da análise prévia, pode ser gerada pela aplicação de alguma sequência de regras extraída da gramática. Métodos de análise sintática baseados em árvore sintática podem empregar análise descendente, que constroem a árvore sintática da raiz para as folhas ou análise ascendente, que adotam o processo inverso. Outras técnicas podem não implementar a árvore sintática.

Uma contribuição importante para o projeto e implementação de compiladores foi dada por (NETO, 1993), ao introduzir a construção de reconhecedores sintáticos empregando autômatos de pilha estruturados, que procura associar as vantagens dos métodos ascendentes à eficiência dos reconhecedores baseados em autômatos finitos. Esse formalismo foi definido na seção 2.1.2.

Pela simplicidade e facilidade de modificação, torna-se particularmente indicado para reconhecer linguagens experimentais, ainda não consolidadas, como a apresentada neste trabalho. Desse modo, o analisador léxico do compilador que implementa essa linguagem será modelado por esse formalismo.

c) Análise semântica

Identifica os erros não detectados na análise sintática, tais como variáveis referenciadas sem terem sido declaradas, inconsistência de tipos de variáveis em expressões, múltiplas declarações de um identificador, expressões de tipo incompatível com o comando.

Devido a não existir um padrão para representar a semântica estática de uma linguagem, bem como pela quantidade e tipos de análise variarem de uma linguagem para outra, implementar a análise semântica torna-se uma tarefa não muito bem definida.

Um recurso utilizado para representar a semântica de uma linguagem são as gramáticas de atributos. Uma gramática de atributos é composta por equações de atributos e regras semânticas, que descrevem a relação entre os atributos semânticos e as regras gramaticais da linguagem (TUCKER; NOONAN, 2009).

Outra análise, classificada como semântica, diz respeito à que é feita sobre o código gerado, visando sua otimização.

Após compilado, um programa será executado em um ambiente de execução, que será abordado pela subseção a seguir.

2.3.3 Ambiente de execução

Ambientes de execução compreendem todo o software que realiza a infraestrutura necessária para que o código-objeto gerado pelo compilador possa ser finalmente executado. Uma de suas mais importantes manifestações nos compiladores atuais ocorre na forma de um simulador da máquina virtual para a qual o compilador gera código.

Outra componente importante de um ambiente de execução corresponde ao conjunto de funções de biblioteca utilizadas pelo programa compilado, que são chamadas, em tempo de execução, pelo código-objeto gerado pelo compilador. Outro, ainda, refere-se ao software que proporciona outras funções essenciais à execução do programa compilado, tais como operações básicas de entrada e saída, acessos ao sistema de arquivos, ao gerenciamento de recursos do computador, especialmente de memória, e a outros serviços do sistema operacional subjacente.

O ambiente de execução é a estrutura de memória e de registros de memória, usada no gerenciamento da memória durante a execução de um programa.

Um programa que possui características adaptativas não pode ser executado em ambientes de execução que impeçam o mesmo de promover as auto-modificações nele especificadas.

Uma máquina virtual é uma aplicação computacional para criar um ambiente virtual. Dentre os diferentes tipos de máquinas virtuais existentes, o termo é mais comumente empregado para designar um ambiente virtual de hardware, ou monitor de máquina virtual (*virtual machine monitor*). Um monitor de máquina virtual tem um funcionamento idêntico ao da máquina real que lhe dá suporte, ou seja, um software que executa em uma máquina sem virtualização, também executará no ambiente virtual (BUZEN, GAGLIARD, 1973).

Funcionando como uma cópia de um sistema físico, mas de forma isolada e eficiente, permite que um programa seja executado sem estar sujeito às mesmas restrições impostas à execução direta na máquina real.

Justifica-se, assim, recorrer-se a um ambiente virtual, especialmente construído com a finalidade de permitir que um programa possa realizar alterações em seu código, seja pela supressão, seja pelo acréscimo de instruções.

Um ambiente com essas características foi desenvolvido por Pelegrini (2009), conforme citado na seção 1.1, no qual serão executados os programas desenvolvidos na linguagem apresentada no próximo capítulo.

3. PROGRAMAÇÃO ADAPTATIVA

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

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

Em Freitas (2008), afirma-se que, em particular, é desejável que uma linguagem para programação adaptativa atenda, no mínimo, os seguintes requisitos:

a) Oferecer, como parte de seus recursos de sintaxe, notações práticas para referenciar as porções de código passíveis de modificação, os tipos de modificação a realizar, bem como especificar exatamente as alterações a serem efetuadas sobre o código.

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

c) Oferecer ao programador um conjunto de operadores adaptativos, que expressem, de forma clara e intuitiva, as operações de automodificação disponíveis ao usuário.

Em adição aos recursos usuais encontrados em linguagens não voltadas à especificação de programas automodificáveis, um conjunto de recursos próprios de uma camada adaptativa deve ser incluído, de forma que possa proporcionar ao programador a possibilidade de indicar o funcionamento da parte dinâmica de seu código. Para isso, linguagens voltadas para a codificação de programas adaptativos devem, no mínimo, disponibilizar ao programador elementos sintáticos que proporcionem a possibilidade de especificação de alguns recursos específicos, tais como a criação e remoção de trechos de código, e de conexões entre eles.

Com essas considerações em mente, passamos a descrever as características mais relevantes da linguagem proposta.

A seção seguinte apresenta a estrutura de um código genérico descrito em uma linguagem de programação composta por uma linguagem hospedeira e uma camada adaptativa, que permite construir programas compostos de trechos de código escritos na linguagem hospedeira, que implementam a parte não-adaptativa subjacente do código adaptativo, e um conjunto de ligações entre esses trechos, às quais são atreladas ações adaptativas, responsáveis pelas automodificações. Na linguagem proposta, essa ligação entre trechos não adaptativos é feita através de conectores, sendo a lógica de conexão controlada por operações de escolha múltipla, definida por decisores destinados a escolher dinamicamente as conexões apropriadas entre trechos de programa, a serem usadas em cada particular situação.

3.1 REPRESENTAÇÃO GRÁFICA PARA SOFTWARE ADAPTATIVO

Uma forma organizada de representação de um programa é a divisão de seus elementos em camadas, cada qual responsável por abrigar elementos de um único tipo. Apresenta-se, assim, uma maneira gráfica de representar programas construídos segundo essa estratificação em camadas.

a) Dispositivo subjacente

A implementação do dispositivo subjacente não-adaptativo é realizada pela compilação de um programa escrito em uma linguagem de alto nível hospedeira disponível, cujo paradigma e cuja sintaxe devem ser compatíveis com esta linguagem. A partir dele, uma camada adaptativa é adicionada para prover os mecanismos de automodificação que proporcionem ao código a automodificação desejada. A Figura 3 apresenta o esquema da arquitetura de um programa projetado como um dispositivo guiado por regras, onde é possível observar a representação da camada de código, formada por blocos básicos escritos em linguagem hospedeira (camada 1), os quais são interligados por conexões, constantes da camada 3. O valor de saída de um bloco básico é utilizado por um decisor (camada 2), o qual, em função desse valor, conecta a saída do bloco básico corrente à entrada de algum dos blocos básicos do programa.

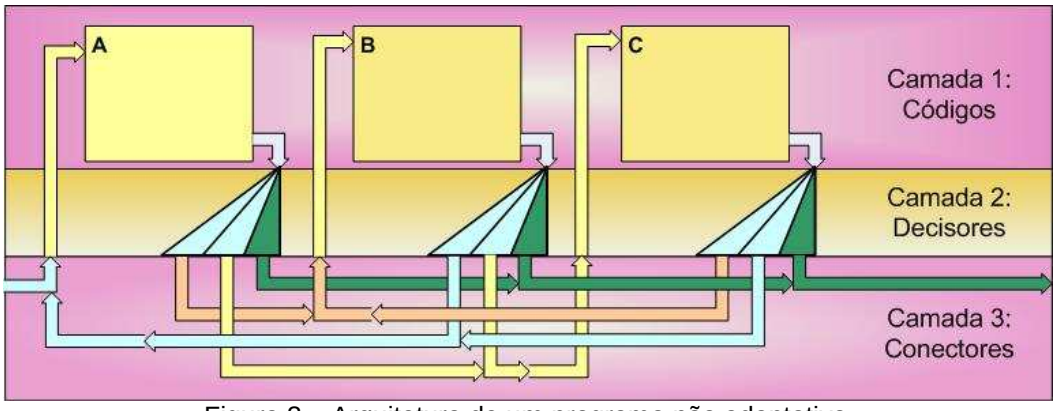

Figura 3 – Arquitetura de um programa não adaptativo
na forma de um dispositivo guiado por regras.

b) Componentes construtivos

Define-se um bloco básico como sendo uma parcela do programa, expressa na forma de uma sequência de comandos da linguagem hospedeira, e que deve ser descrito de tal modo que apresente uma só entrada e uma só saída. Uma representação esquemática de um bloco básico pode ser observada na Figura 4.

Para facilitar a especificação da lógica do fluxo de execução de um programa, a saída de cada bloco básico deve ser sempre associada a algum valor, que exprima, de alguma forma coerentemente convencionada, uma condição referente ao resultado da sua execução (ou seja, uma espécie de classificação dos resultados da execução do bloco básico). Tal valor deve ser calculado internamente ao respectivo bloco básico, de alguma forma arbitrária, e sob total responsabilidade do programador.

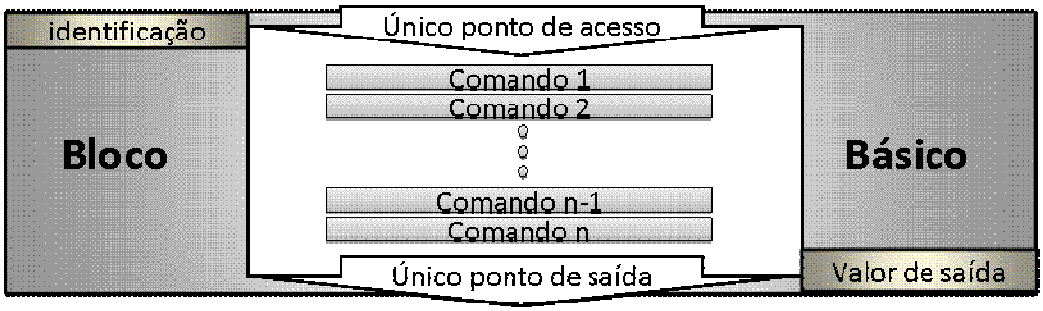

Figura 4 – Estrutura de um bloco básico genérico.

Os programas automodificáveis a serem descritos pelo usuário final na linguagem de alto nível aqui proposta contêm, como elementos construtivos iniciais, um conjunto de blocos básicos, escritos puramente na linguagem hospedeira.

Para assegurar a coerência estrutural dos programas assim construídos, é preciso que se garanta a conectividade entre os blocos básicos constituintes do programa, e, para isso, essas conexões devem também atender as especificações apresentadas em (PELEGRINI, 2009). Cabe ao programador projetar adequadamente o seu programa para atender a esse requisito, e ao compilador, efetuar os devidos testes de consistência que assegurem que o requisito tenha sido devidamente cumprido.

Assim sendo, a cada saída de bloco básico deve estar associado a um valor, de forma similar ao que ocorre com os valores calculados por uma função. A lógica da interconexão se especifica associando-se a cada conexão um valor diferente, avaliado no bloco básico onde a conexão se origina, condição essa que deve ser atendida como pré-requisito para que a mesma seja percorrida, ao ser executado o programa.

Cabe ao programador determinar os possíveis valores de saída do bloco básico. Caso nenhum valor seja especificado, a cláusula OTHERWISE, obrigatória em todas as conexões, garante que sempre haverá algum destino para o fluxo do programa após o término da execução de um bloco básico. Na Figura 5 esse destino padrão está representado como único apenas para simplificar tal representação, ou seja, não necessariamente a saída padrão de um bloco básico será para um único destino em um dado programa.

Convencionou-se, na linguagem proposta, que seja especificada cada uma dessas conexões partindo sempre da saída de algum dos blocos básicos do programa, tendo sempre como destino a entrada de algum desses trechos estáticos de código.

Naturalmente, uma entrada de bloco básico pode receber mais de uma conexão, conforme exemplificado na Figura 3. Por outro lado, cada valor numérico de saída de um bloco básico deve estar associado a uma única conexão e, portanto, a um único bloco básico de destino. Isto significa que um mesmo bloco básico pode estar conectado a mais de um bloco básico de destino, desde que cada uma dessas conexões esteja associada a um valor de saída diferente do bloco básico.

Essa é uma forma de particionar um programa em unidades de tamanho máximo, com uma entrada e uma saída.

Neste ponto completa-se a especificação da parte da linguagem responsável pela representação do dispositivo subjacente, ou seja, da lógica do programa em cada instante.

Dessa especificação, resulta claramente que um grafo conexo orientado pode ser usado como uma representação natural para o dispositivo subjacente.

Cada um dos vértices desse grafo especifica um algoritmo, com uma entrada e uma saída, que implementa um bloco básico de código associado ao vértice.

As conexões do grafo que interligam esses vértices são todas condicionadas ao valor de saída do bloco básico, e tais valores representam as diversas situações em que, durante a sua execução, o programa pode ou não percorrer cada um dos caminhos alternativos existentes na sua trajetória, no grafo que representa o programa.

Cada um dos algoritmos associados aos vértices do grafo é responsável, portanto, pela materialização de um bloco básico de código, a partir do qual é construída a lógica do programa.

c) Camada adaptativa

Para obter o dispositivo adaptativo desejado, ou seja, um programa completo operante, com código auto-modificável que opere de acordo com especificado no programa-fonte desenvolvido na linguagem proposta, resta acrescentar ao dispositivo subjacente, representado pelo conjunto de blocos básicos e conexões, mencionados anteriormente, uma camada adaptativa.

Esta se responsabiliza pela realização da parte dinâmica do código que esse programa simboliza, ou seja, um conjunto de definições de funções adaptativas, e de suas chamadas atreladas às conexões condicionais estabelecidas entre os blocos básicos. Em termos de representação gráfica, isso corresponde a se associar ações adaptativas às arestas existentes entre os vértices do grafo que descreve a topologia do programa, responsáveis pela interligação dos blocos básicos.

Adicionando dessa forma a camada adaptativa à arquitetura esboçada na Figura 1, obtém-se, como resultado, a representação mostrada na Figura 5, que permite esquematizar integralmente um programa adaptativo.

Na camada 3 figuram todas as funções adaptativas cuja utilização esteja prevista na lógica do programa, e essas funções adaptativas são associadas aos conectores da camada 4, incluindo a informação da ocasião da sua execução (anterior ou posterior) em relação ao momento em que se dá a transferência de controle de processamento entre os blocos básicos envolvidos nessa conexão.

Assim, ações adaptativas anteriores devem ser empregadas quando se deseja que sua execução ocorra imediatamente antes da transferência de controle para o bloco básico de destino. Ações posteriores devem ser utilizadas no caso oposto.

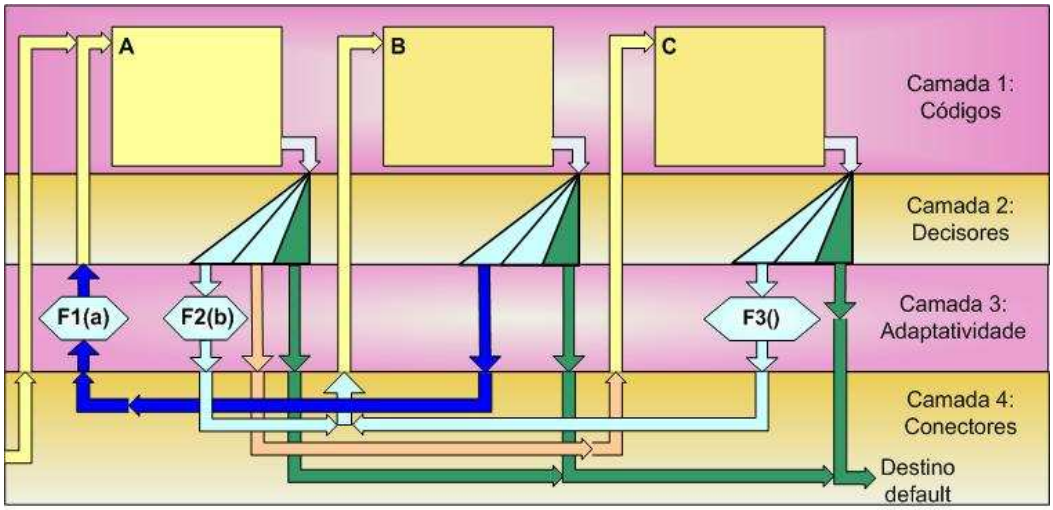

Figura 5 – Arquitetura de um programa adaptativo.

As ações adaptativas são expressas, como se sabe, na forma de chamadas de funções adaptativas, devendo estas ser, naturalmente, declaradas em alguma parte do corpo do programa, especificada na definição gramatical da sintaxe da linguagem.

A declaração de uma função adaptativa resume-se a indicar as ações de modificação do programa adaptativo, a serem efetuadas em tempo de execução, nas ocasiões em que a função for ativada.

Isso ocorrerá repetidamente, todas as vezes que o programa percorrer alguma conexão à qual a função adaptativa esteja associada, produzindo como efeito, de cada vez, a aplicação de todas as alterações especificadas pela função.

Na linguagem aqui proposta, as funções adaptativas se restringem a executar ações de inserção ou de remoção, tanto de blocos básicos, como de conexões já estabelecida entre eles.

A referência a um bloco básico deve ser feita por nome, enquanto a referência a uma conexão deve especificar o respectivo bloco básico de origem, bem como o valor de saída desse bloco básico que o seleciona.

3.2 RECOMENDAÇÕES PARA O PROJETO DE PROGRAMAS ADAPTATIVOS

Nesta seção propõe-se uma série de recomendações, aplicáveis ao desenvolvimento de programas adaptativos. Para isso pressupõe-se que o programador tenha alguma familiaridade com técnicas adaptativas como meio de atingir seus objetivos. Convém ainda que conheça os conceitos básicos disponíveis na linguagem proposta, bem como a notação gráfica que os representa.

Pode-se buscar inspiração para um procedimento sistemático de programação adaptativa, observando-se algumas práticas que foram adotadas durante a concepção da linguagem cujas características foram anteriormente especificadas.

Assim, observou-se que o desenvolvimento de um programa automodificável costuma originar-se de uma análise prévia da lógica do seu código não-adaptativo subjacente, inicialmente descrito de forma sequencial, ignoradas as alterações dinâmicas nesse primeiro momento.

Concluída essa fase, identifica-se as regiões desse programa sobre as quais se espera que operem as funções adaptativas encarregadas de efetuarem alterações dinâmicas no código. Tais alterações, no caso, podem ser especificadas por operações de inserção ou de remoção de código, conforme o caso.

A seguir, determinam-se e identificam-se, no programa, todos os pontos nos quais se possa prever a inserção de novas instruções, bem como os pontos que delimitam as regiões de código que poderão ser removidas, seccionando-se, então, o programa nesses pontos.

Em seguida, delimitam-se grupos de instruções, de comprimento máximo, que apresentem um só ponto de acesso e um só ponto de saída, e que não contenham nenhum dos pontos em que o programa foi seccionado anteriormente.

Caso seja seccionada alguma estrutura de controle da linguagem hospedeira, tais como comandos iterativos, essa estrutura deve ser reescrita para que utilize estritamente estruturas mais simples, tais como testes e desvios condicionais.

Resulta, da aplicação desse procedimento ao programa em desenvolvimento, um conjunto de blocos básicos correspondentes a trechos inalteráveis de código, cada qual com uma única entrada e uma única saída, e interligados por uma lógica de conexão baseada na seleção de caminhos com base nos valores de saída associados a os blocos básicos, e comandada por estruturas de múltipla seleção.

Cada um desses blocos básicos, devidamente identificado através de um nome, é então codificado através de uma declaração apropriada (CODE).

Cada um desses blocos básicos deve portanto calcular um valor de saída, de forma análoga ao que ocorre com uma função. cada um dos possíveis diferentes valores de saída determina o desvio, com origem no (único) ponto de saída de um bloco básico, para algum dos diferentes blocos básicos associados a tais valores de saída, no momento da execução.

Como conexões entre blocos básicos são implementáveis através de uma escolha múltipla, talvez utilizando desvios condicionais, à saída de cada bloco básico o compilador deve acoplar um decisor, estrutura de desvios condicionais múltiplos através da qual conexões diversas podem ser associadas a essa saída, promovendo sua ligação à entrada de outros blocos básicos, regida dinamicamente pelo valor de saída calculado pelo bloco básico.

As conexões elementares, assim acrescentadas com a ajuda dos decisores, interligam condicionalmente os blocos básicos de acordo com a lógica de cada programa, disso resultando uma topologia que deve refletir com exatidão a versão inicial (única) do programa não-adaptativo subjacente.

Para completar a construção do programa automodificável, resta apenas acrescentar às conexões entre blocos básicos as devidas ações adaptativas, na forma de chamadas paramétricas de funções adaptativas, que tenham sido devidamente declaradas no texto do programa.

As ações adaptativas, de acordo com o modelo usualmente empregado (NETO, 2001), são acopladas ao programa, sempre como parte das conexões

existentes entre blocos básicos, podendo-se especificá-las como anteriores ou posteriores à transferência do controle de um bloco básico para outro, de acordo com a ocasião que se mostre mais adequada para sua execução.

Os grupos de instruções que deverão ser inseridos dinamicamente no código durante a execução, constituirão blocos básicos que serão declarados nas respectivas funções adaptativas.

As ações adaptativas são indicadas na descrição das conexões que as suportam, e sempre que sua execução provocar a remoção ou adição de qualquer conexão, o programador deve tomar as precauções necessárias para garantir que seja mantida a conectividade do grafo que representa o programa resultante. Ao compilador cabe verificar se tal coerência acontece na ocasião do início da execução do programa, e ao ambiente de execução, garantir que tal coerência seja mantida durante toda a execução do programa, a despeito de quais sejam as auto-modificações que ele venha a sofrer.

Assim, em tempo de execução, ao ser percorrida uma conexão contendo uma ação adaptativa posterior, o controle do programa é passado ao bloco básico de destino antes de aplicadas as ações adaptativas elementares que compõem a função adaptativa chamada, garantindo, assim, que esse bloco básico seja sempre atingido independentemente de uma eventual remoção da conexão corrente pela auto-modificação executada pela ação adaptativa.

No caso de uma ação adaptativa anterior, que é especificada imediatamente à saída de um bloco básico, é executada a aplicação de todas as auto-modificações especificadas na função adaptativa antes que seja percorrida a conexão à qual ela estiver associada.

A codificação do programa se encerra com a declaração das funções adaptativas, cada qual especificando uma lista das remoções e das inserções desejadas, de blocos básicos ou de conexões a serem removidas ou inseridas no programa, sempre que for percorrida a conexão à qual uma chamada dessa função estiver associada (SILVA, NETO, 2010).

3.3 PROPOSTA DE LINGUAGEM PARA PROGRAMAÇÃO ADAPTATIVA

A pesquisa acerca do desenvolvimento de uma linguagem para programação adaptativa vem se desenvolvendo há alguns anos, conforme as referências feitas nos parágrafos anteriores aos principais trabalhos desenvolvidos nesse sentido (FREITAS, 2008; PELEGRINI, 2009; CASTRO JÚNIOR, 2009; PISTORI, 2003).

Buscando com a presente dissertação dar mais uma contribuição que represente outro passo na direção do estabelecimento de uma linguagem para uso prático, no desenvolvimento de software adaptativo, propõe-se, neste capítulo, uma linguagem simples de alto nível, em estilo imperativo, dotada de recursos que permitam escrever códigos-fonte que especifiquem, de forma completa e consistente, programas que exibam a capacidade de se auto-modificarem durante a sua própria execução.

Partiu-se, para tanto, de um conjunto muito simples de comandos básicos, comuns a qualquer linguagem de programação imperativa típica, estruturada, e a tal conjunto mínimo foram acrescentadas extensões incluindo comandos específicos, os quais incorporam à linguagem de programação a capacidade que permite escrever, através da mesma, códigos adaptativos. Procurou-se, ainda, respeitar uma certa independência entre a linguagem hospedeira e a adaptativa, com o objetivo de simplificar uma futura substituição dessa linguagem-base subjacente por alguma outra que se considerar mais conveniente.

Considerando que a lógica de qualquer programa com código automodificável pode, também, ser expressa convencionalmente na forma de um programa equivalente, de código estático, é possível argumentar que a adaptatividade seja desnecessária.

Adicionalmente, considerando que o código necessário para exprimir programas automodificáveis tende a ser mais extenso e mais complexo que o de um programa não-adaptativo, a programação adaptativa pode exigir práticas de programação nem sempre alinhadas àquelas geralmente recomendadas para a construção de programas considerados adequados segundo os critérios correntemente mais aceitos.

Influenciados por posicionamentos como esses, os sistemas operacionais atuais incorporam mecanismos de segurança que procuram impedir que os códigos executáveis sofram alterações, tanto externas como espontâneas.

A seguir, detalha-se a presente proposta, comentando, inicialmente, as características desejáveis da linguagem, modelando o código que ela representa através de um dispositivo adaptativo guiado por regras, e apresentando, a seguir, uma gramática da sua camada adaptativa, comentando uma possível metodologia para sua utilização e, por fim, apresentando um exemplo completo de sua utilização.

A gramática apresentada no Quadro 1 define, na notação de Wirth, a sintaxe livre de contexto da linguagem, que representa a camada adaptativa da linguagem para a codificação de programas adaptativos, estruturados segundo o modelo apresentado na seção 3.1, apresentada neste trabalho. No Quadro 2 esta gramática é complementada pela gramática da linguagem-base simples adotada para a experimentação aqui desenvolvida. Juntas, essas duas gramáticas se complementam, representando a gramática da linguagem proposta.

Um programa adaptativo descreve o código auto-modificável que se deseja implementar. Inicialmente, são identificados o NAME (nome do programa), e são declarados o nome do bloco básico de início e de final do processamento do programa como um todo. A seguir, declaram-se todos os blocos básicos (`decl-bloco-básico`) nomeando cada um dos blocos básicos, e indicando o respectivo conteúdo. Depois, são declaradas todas as conexões entre eles, através das `decl-conexões-adaptativas`, por meio das quais são especificados, para cada valor de saída dos vários blocos básicos, as correspondentes conexões individuais entre o bloco básico de origem e o correspondente bloco básico de destino. Vários casos (representados por vários valores numéricos) e os respectivos blocos básicos de destino são associados nessa ocasião.

Esta operação promove a ponte necessária entre a programação não adaptativa e a adaptativa, permitindo ao programador, essencialmente: delimitar trechos de código e identificar tais trechos, estática (identificação inicial, feita pelo programador no momento da codificação) e dinamicamente (feita pelo próprio programa, pela identificação de um trecho do código durante a execução do programa, para que possa daí em diante ser referenciado). Os trechos assim

definidos constituirão os blocos básicos de que se compõe o programa (definido no Quadro 1 em `decl-bloco-básico`). Exemplo: ao ser declarado um bloco básico, dá-se-lhe um nome, e esse trecho do programa poderá passar a ser referenciado por esse nome.

```
programa-adaptativo =
     “ADAPTIVE” “MAIN” “[” “NAME” “=” nome “,” “ENTRY” “=” nome “,”
     “EXIT” “=” nome “]” “IS” declarações-adaptativas “END” “MAIN” “.” .
declarações-adaptativas =
     decl-blocos-básicos “;” decl-conexões-adaptativas “;”
     decl-funções-adaptativas .

decl-blocos-básicos =
     decl-bloco-básico { “;” decl-bloco-básico }
decl-bloco-básico =
     “CODE” nome “:” “<” código-hospedeira “>” .

decl-conexões-adaptativas =
    [ conexão-adaptativa { “;” conexão-adaptativa } ] .
conexão-adaptativa =
    “CONNECTION” “FROM” nome
    “(” conexões-condicionais “OTHERWISE” conexão-incondicional “)” .
conexões-condicionais =
    [ conexão-condicional { “,” conexão-condicional }  ] .
conexão-condicional = “CASE” número “:” conexão-incondicional .
conexão-incondicional =
    “TO” nome
    [ “PERFORM” nome “(” lista-de-nomes “)” “BEFORE” ]
    [ “PERFORM” nome “(” lista-de-nomes “)” “AFTER” ] .
lista-de-nomes = [ nome { “,” nome} ] .

decl-funções-adaptativas =
    [ decl-função-adaptativa { “;” decl-função-adaptativa } ] .
decl-função-adaptativa =
    “ADAPTIVE” “FUNCTION” nome “(” lista-de-nomes “)”
    [ “[” “GENERATORS” nome { “,” nome } “]” ]  “{”
    { “REMOVE” “[” ( “CODE” nome | “CONNECTION” número “FROM” nome ) “]” “;”
    | “INSERT” “[” ( decl-bloco-básico | conexão-adaptativa ) “]” “;” } “}” .
```

Quadro 1 – Sintaxe da camada adaptativa da linguagem proposta

As declarações que especificam conexões entre blocos básicos (CONNECTIONs) realizam o elemento conceitual “decisor”, mencionado anteriormente, através das suas cláusulas CASE. Os blocos básicos de destino são especificados na cláusula TO.

Caso haja alguma chamada de função adaptativa (anterior e/ou posterior) associada a alguma dessas conexões, isso deve ser declarado em cláusulas PERFORM correspondentes, usando o atributo BEFORE/AFTER para designar que as correspondentes ações adaptativas são anteriores/posteriores, respectivamente.

A declaração de conexão é encerrada por uma cláusula OTHERWISE, que especifica um bloco básico de destino, para onde o processamento deverá ser desviado quando o valor de saída do bloco básico não estiver contemplado em suas cláusulas CASE. A cláusula OTHERWISE sempre ocorre em uma declaração de conexão, ao passo que as cláusulas CASE são opcionais.

A chamada da função adaptativa é denotada indicando-se o nome da função, acompanhada da respectiva lista de argumentos, em correspondência posicional com os parâmetros indicados na declaração da função adaptativa.

```
código-hospedeira =
[ declaração-de-variáveis ] { declaração-de-procedimento } sequência-de-comandos “;”
.
declaração-de-variáveis = decl-de-variáveis {“;” decl-de-variáveis } “;” .
decl-de-variáveis = { “INT” lista-de-nomes |
                      “CHAR” lista-de-nomes } .
decl-de-procedimento =
       “PROC” nome “(” lista-de-nomes “)” sequência-de-comandos “END” “;” .
lista-de-nomes = nome { “,” nome} .
sequência-de-comandos   = comando { “;” comando } .
comando        = [ nome “:” ]
                 [ atribuição | leitura | impressão | decisão | chamada | desvio ] .
atribuição = nome “:=” expressão.
expressão  = termo { ( “+” | “-” ) termo } .
termo      = fator { ( “*” | “/” ) fator } .
fator      = nome | número | “(” expressão “)” .
leitura    = “READ” [ lista-de-nomes ] .
impressão  = “PRINT” lista-de-expressões .
lista-de-expressões = [ expressão { “,” expressão } ] .
decisão    = “IF” comparação “THEN” comando “ELSE” comando .
comparação = expressão operador-de-comparação expressão .
operador-de-comparação = “>” | “=” | “<” .
chamada = “CALL” nome “(”lista-de-expressões “)” .
desvio = “GO” “TO” nome .
nome = letra { letra | dígito } .
número     = dígito { dígito } .
letra =  “a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m”
| “n” | “o” | “p” | “q” | “r” | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z” | “A” |
“B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N” | “O” |
“P” | “Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z” .
dígito =  “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” .
```

Quadro 2 – Sintaxe da linguagem hospedeira

Encerra-se a descrição do programa adaptativo com uma sequência de declarações das funções adaptativas utilizadas nas conexões. Para cada uma delas, um cabeçalho define seu nome, sua lista de parâmetros, além de uma lista de variáveis e outra de geradores, ambas opcionais. A seguir, uma sequência de comandos de inserção (INSERT) e de remoção (REMOVE) de blocos básicos e de conexões é apresentada, completando a especificação das ações de automodificação associadas à chamada da função em questão. Para evitar

dificuldades com o controle da eliminação de partes do código, só são permitidas eliminações de blocos básicos que, no momento da execução da ação adaptativa que os elimina, estiverem totalmente desconectados do restante do programa. Caso não estejam, o programador deverá especificar a remoção prévia das conexões a ele acopladas.

O não terminal `código-hospedeira`, constante da gramática do Quadro 2, define o formato do código subjacente, escrito na linguagem hospedeira, cuja sintaxe está especificada pela gramática livre de contexto apresentada no Quadro 2, acima, completando a linguagem mínima proposta, que é denominada **B**asic **Ada**ptive **L**anguage - BADAL.

3.4 IMPLEMENTAÇÃO DA LINGUAGEM PROPOSTA

A implementação da linguagem proposta deve ser feita de tal forma que programas escritos nessa notação sejam devidamente convertidos para um formato tal que sua execução seja viabilizada. Para tanto, foi construído um compilador, que produz um código automodificável, que é processado em um ambiente de execução preexistente (PELEGRINI, 2009).

A implementação de um compilador para a linguagem proposta não constitui em si um problema complexo, dada a simplicidade da linguagem, em cujo projeto se procurou evitar a inclusão de quaisquer recursos que não os estritamente necessários.

O método de construção adotado foi o da conversão direta da gramática, expressa na notação de Wirth, que define a sintaxe livre de contexto da linguagem, em um reconhecedor sintático expresso na forma de um autômato de pilha estruturado, obtendo-se assim, com muita facilidade, um reconhecedor sintático muito eficiente que constitui o núcleo desejado de um compilador guiado pela sintaxe para a linguagem proposta.

O método de obtenção, que foi aqui utilizado para construir um reconhecedor desse tipo para a linguagem proposta, se apóia na forte correspondência existente entre a estrutura das regras gramaticais da Notação de Wirth (WIRTH, 1977), na qual está denotada a gramática, e a de um autômato de pilha estruturado equivalente, que representa o reconhecedor correspondente. Detalhes desse método podem ser encontrados em (NETO, 1987; NETO, LEONARDI, PARIENTE,1998).

Tome-se como exemplo o comando de decisão da linguagem proposta, para demonstrar o uso do mecanismo de modelagem do compilador através de autômatos de pilha estruturados. O mesmo é definido pelas regras gramaticais da Figura 6.

```
decisão    = “IF” comparação “THEN” comando “ELSE” comando .
comparação = expressão operador-de-comparação expressão .
operador-de-comparação = “>” | “=” | “<” .
```

Figura 6 – Definição do comando de decisão na linguagem proposta

O não terminal `expressão` está sintaticamente definido como sendo constituído de um número, um nome (variável) ou expressão aritmética, e o não-terminal `comando`, como sendo qualquer dos comandos disponíveis na linguagem descrita. A representação correspondente, através de autômato de pilha estruturado, é mostrada na Figura 7.

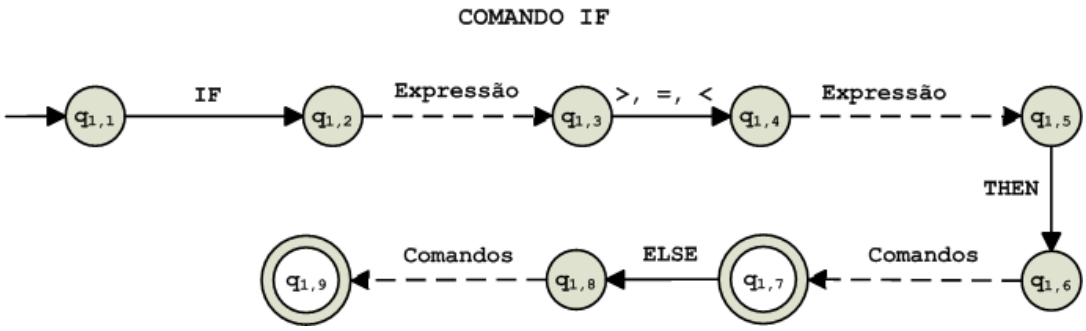

Figura 7 – Exemplo de autômato de pilha estruturado

Para simplificar o exemplo, suponhamos que esse autômato de pilha estruturado seja constituído por duas sub-máquinas apenas: Comandos, da qual foi aproveitado apenas o comando de decisão, com função de transição P1 e Expressão, que teria função de transição P2. Será apresentada apenas a função P1.

P1 = {
$p_{1,1}$:$q_{1,1}$ IF → $q_{1,2}$
$p_{1,2}$:$q_{1,2}$ Expressao → Empilha $q_{1,3}$
$p_{1,3}$:$q_{1,3}$ >, =, < → $q_{1,4}$
$p_{1,4}$:$q_{1,4}$ Expressao → Empilha $q_{1,5}$
$p_{1,5}$:$q_{1,5}$ THEN → $q_{1,6}$
$p_{1,6}$:$q_{1,6}$ Comandos → Empilha $q_{1,7}$
$p_{1,7}$:$q_{1,7}$ ELSE → $q_{1,8}$
$p_{1,8}$:$q_{1,8}$ Comandos → Empilha $q_{1,9}$
$p_{1,9}$:$q_{1,91}$ ε → Retorna }

Após transitar do estado $q_{1,1}$ para o estado $q_{1,2}$, a regra $p_{1,2}$, especifica uma transição de chamada de submáquina, no caso a submáquina Expressão e é guardado na pilha do autômato o estado de retorno $q_{1,3}$.

Após o término dessa execução, é desempilhado o topo da pilha do autômato que contém a informação para qual submáquina e respectivo estado deverá ocorrer o retorno. O mesmo ocorre no estado $q_{1,4}$. Nos estados $q_{1,6}$ e $q_{1,8}$ a chamada se dá para a outra submáquina do autômato de forma similar.

Pode-se observar, então, que o reconhecimento sintático por esse método é mais simples que através da construção de uma árvore sintática, adotada por muitos construtores de compiladores.

O método adotado para a construção do núcleo léxico-sintático deste compilador se mostrou prático, tendo propiciado a elaboração de um reconhecedor bastante eficiente.

O autômato de pilha estruturado que representa a estrutura léxico-sintática do compilador, pode ser encontrado no Apêndice B.

É constituído por nove submáquinas, cada qual responsável pelo reconhecimento de partes distintas da linguagem.

3.5 MÁQUINA VIRTUAL DE EXECUÇÃO DE CÓDIGO ADAPTATIVO

O código-objeto gerado como saída pelo compilador é expresso em uma linguagem de baixo nível, que se executa em uma máquina virtual especialmente desenvolvida para a execução de programas automodificáveis (PELEGRINI, 2009).

O conjunto de instruções dessa máquina virtual, empregado pelo compilador para a tradução do programa-fonte adaptativo para uma forma executável, é mostrado no Quadro 3.

A tradução do código fonte para um código na linguagem da máquina virtual de execução é realizada conforme a especificação de semântica apresentada no apêndice C.

COMANDO	SIGNIFICADO
adapt X,Y	Substitui uma sequência de comandos X por outra Y
add X,Y	Soma os inteiros X e Y e armazena o resultado em X
addcmd X,Y	Concatena a cadeia Y à variável X
call X	Chamada de subprograma X
cmp X,Y	Comparação entre X e Y
code	Delimita o início ....
data	Inicia seção declaração de variáveis globais.
div X,Y	Divisão inteira de X por Y armazenando o resultado em X.
emark X	Rotula o final de trecho adaptativo X.
end	Final físico de programa
endp	Final de ma sequência de instruções
X endproc	Finaliza o procedimento X
exit	Finaliza a execução do programa
inputcmd Y	Entrada de string para a variável Y
inputint X	Entrada de inteiro para a variável X
int a,b,c,…,n	Declaração de variável tipo inteiro
je X	Se os valores comparados são iguais desvia para o label X
jge X	Se o 1º valor é maior ou igual ao 2º desvia para o label X
jle X	Se o 1º valor é menor ou igual ao 2º desvia para o label X
jmp X	Desvio incondicional
jne X	Se os valores comparados são diferentes, desvia para o label X
label X	Associa um rótulo X à posição corrente
main	Início da função principal
mark X	Marca início do trecho adaptativo X
mov X,Y	Atribui valor Y à variável X
mul X,Y	Multiplica X por Y e atribui o resultado à variável X
popint X	Desempilha valor inteiro no topo da pilha para a variável X
popstr X	Desempilha valor cadeia no topo da pilha para a variável X
printint X	Imprime valor inteiro X
printstring X	Imprime cadeia X
X proc	Rotula início do procedimento X
pushint X	Empilha valor inteiro no topo da pilha

pushstr X	Empilha cadeia de caracteres no topo da pilha
resetstr X	Inicia variável X, tipo cadeia, com valor nulo.
ret	Ponto de retorno de subrotina.
start	Rótulo que marca o ponto de início de execução do programa.
string X,Y, …	Declaração de variável tipo cadeia
sub X,Y	Subtrai Y de X e armazena o resultado em X.

Quadro 3 – Conjunto de instruções da máquina virtual

4 APLICAÇÃO

Para ilustrar a utilização da linguagem, são apresentados dois exemplos de programas, seguindo as recomendações de projeto de programa constantes da seção 3.2.

a) Exemplo A

Considerando o problema que consiste em calcular o n-ésimo termo da sequência matemática

1, 5, 11, 19, 29, 31, ...

Uma definição recorrente para representar tal sequência é dada por:

S(1) = 1

S(n) = S(n – 1) + 2n, para n ≥ 2

Inicialmente, faz-se a análise da lógica de uma solução não adaptativa. Dado um valor de n maior que dois, faz-se necessário repetir o passo de indução até que o n-ésimo valor da sequência seja encontrado. Essas repetições podem ser realizadas através de iteração, recursão ou macro. Adotando uma solução iterativa chega-se a um algoritmo como no quadro 4, a seguir:

```
algoritmo sequencia
     inteiro n,k,sn
     leia n
     k  = 1
     sn = k
     enquanto (n>k)
          k  = k + 1
          sn = sn + 2 * k
     fimenquanto
     imprima sn
fimalgoritmo
```

Quadro 4 – Algoritmo sequencia

Observa-se que a leitura do valor do número de termos e a atribuição de valores iniciais às variáveis, bem como a impressão do resultado, são executadas apenas uma vez. Esses trechos do programa podem ser identificados, portanto, como blocos básicos. Secciona-se, então essa solução nesses pontos.

A componente iterativa do processo está materializada no comando de repetição e constituirá uma função adaptativa, ou seja, as repetições dessa computação serão realizadas pela inserção desse trecho código em tempo de execução, até que a condição-limite que controla o número de repetições seja satisfeita.

Dessa forma, o algoritmo é seccionado nas seguintes partes:

1) leitura do valor de entrada e atribuição de valores iniciais às variáveis:

```
inteiro n,k,sn
leia n
k  = 1
sn = k
```

2) trecho de comandos cuja execução deve ser repetida em função do valor das variáveis n e k

```
enquanto (n>k)
     k  = k + 1
     sn = sn + 2 * k
fimenquanto
```

3) impressão do resultado:

```
imprima sn
```

Após seccionar o programa conforme descrito, passa-se à codificação do mesmo em BADAL

Deve-se nomear os blocos básicos. O primeiro será chamado `início` e o último `imprime`.

A codificação do programa é iniciada pelo cabeçalho do programa, no qual são especificados o nome do programa e os blocos básicos de início e de término de execução.

Neste particular exemplo, o programa é iniciado executando o bloco básico `início` e termina pela execução do bloco básico `imprime`.

Chamando o programa de `sequencia`, a primeira linha do programa é, então, codificada da seguinte forma:

```
adaptive main [name = sequencia, entry = bloco1, exit = imprime ] is
```

Cada bloco básico deve gerar sempre um valor de saída, a menos que seja o último bloco a ser processado. Assim, faz-se necessário acrescentar ao primeiro bloco básico um valor de saída, arbitrário, e um decisor, que direcionará o fluxo de execução em função de tal valor de saida. O elemento decisor é implementado por um comando de decisão. O valor de entrada, digitado pelo usuário, significa o número ordem do termo da sequência que ele está interessado em obter. Por definição, o primeiro termo é 1 e não precisa ser calculado, apenas será impresso. Serão calculados pelo programa os termos a partir do segundo. A variável k guarda o número de ordem do termo cujo valor foi computado e armazenado na variável sn. A comparação do valor da variável k com o da variável n permite saber se o termo requerido pelo usuário já foi encontrado. Essa condição está implementada no decisor. Por convenção, neste caso particular, o valor de saída será 1 se o termo de ordem n ainda não foi atingido e 2 caso contrário. É esse valor que será testado pela conexão para determinar qual o próximo bloco básico a ser processado. Os dois blocos básicos codificados em BADAL têm a representação a seguir.

Bloco básico `bloco1`:

```
code bloco1 : <
     int n,k,sn;
     print "valor de n";
     read n;
     k := 1;
     sn := k;
     if n>k then bloco1 := 1 else bloco1 := 2;
     >;
```

Bloco básico `imprime`:

```
code imprime : <
     print sn;
     >;
```

Observar que esse segundo bloco básico encerra o processamento (cláusula `exit`, do cabeçalho) e, por isso, neste caso particular, não necessita de um decisor.

Nessa linguagem, primeiro devem ser declarados os blocos básicos. feito isso, eles precisam ser interligados para que se estabeleça o fluxo de execução. Essa interligação é realizada pelos conectores, que constam na segunda seção do programa. É nos conectores que se especificam as chamadas de função adaptativa, quando for o caso.

A representação gráfica dos blocos básicos acima é mostrada na Figura 8.

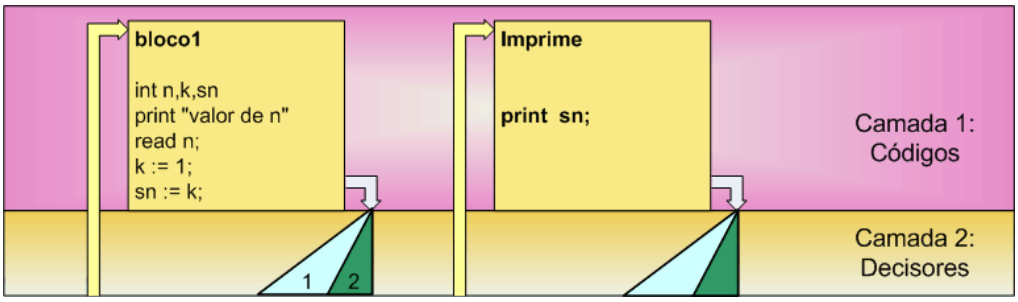

Figura 8 – Blocos básicos

O valor de saída igual a 1, do bloco básico `bloco1`, indica que o valor de entrada informado pelo usuário é maior que 1, necessitando, portanto, computar os termos seguintes da sequência, até o de ordem *n*, ou seja, primeiro é necessário obter esse termo para depois imprimi-lo. Assim, a conexão que interliga esse bloco básico ao bloco básico `imprime` necessita realizar a chamada da função adaptativa que calculará o termo de ordem n antes transitar para a função `imprime`. A chamada de função antes de atingir o bloco básico de destino caracteriza uma função adaptativa anterior.

Essa conexão, em BADAL, que liga o bloco básico `bloco1` ao bloco básico `imprime`, chamando a função adaptativa anterior c, é a seguinte:

```
connection from bloco1 ( case 1 : to imprime perform
c(bloco1) before otherwise to imprime);
```

Por fim, codifica-se a função adaptativa. O agrupamento de comandos (correspondente àquele que, na versão iterativa, constituía o corpo do comando iterativo) será instanciado como um novo bloco básico a cada vez que a função adaptativa for executada.

A condição de parada do laço de repetição, que figura na versão iterativa do programa, aqui é realizada na forma de um valor de saída do bloco básico de saída do código adaptativo, atuando sobre o decisor associado a essa saída. Em outras palavras, enquanto essa condição permanecer verdadeira será feita nova chamada (tipicamente com novos parâmetros) a essa mesma função adaptativa. Quando se tornar falsa, haverá a chamada ao bloco básico que imprime o resultado da computação e o processamento será encerrado. Assim, o bloco básico `imprime` não necessita gerar um valor de saída.

A função adaptativa assim codificada é apresentada abaixo.

```
adaptive function c(i)
     [ generators g ]
{
     insert [ code g : <
          k := k + 1;
          sn := sn + 2 * k;
          if n>k then g := 1 else g := 2;
          >; ];
          insert [ connection from g (case 1 : to imprime
      perform c(g) before otherwise to imprime) ]; }
```

A Figura 9 apresenta o programa completo.

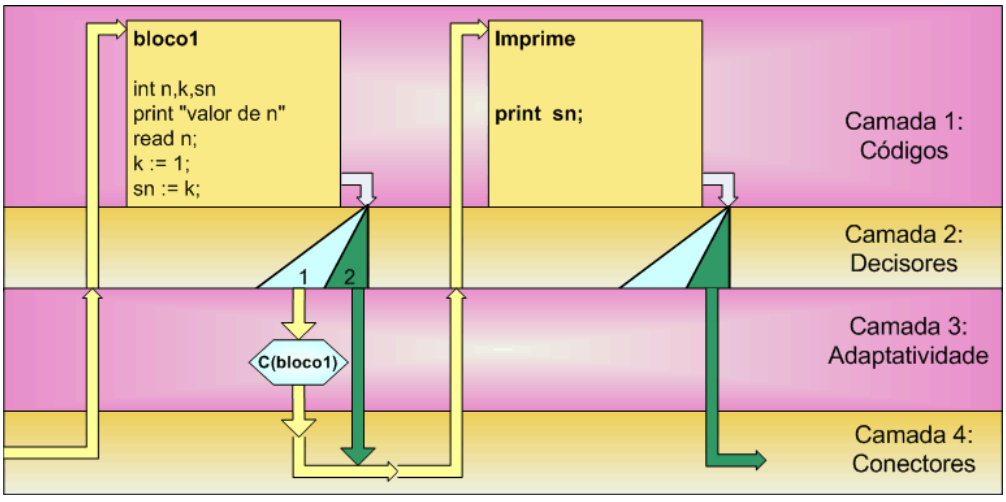

Figura 9 – Representação gráfica do programa.

A cada chamada da função adaptativa são instanciados são instanciados e inseridos no programa adaptativo um novo bloco básico e as conexões correspondentes. Quando o valor de saída do bloco básico corrente for igual a 2 o

respectivo decisor, através da cláusula `otherwise`, selecionando a conexão adequada, desviará o fluxo de execução para o bloco básico `imprime`, que será processado, encerrando, a seguir, a execução do programa. Diferentemente do que ocorre em uma solução recursiva, não é usada uma pilha para a execução de um programa assim construído. Em vez disso, há uma forte semelhança entre o que aqui acontece em tempo de execução com as operações de expansão, em tempo de compilação, de programas-fonte escritos utilizando macros.

O código completo do programa pode ser visto no Quadro 5.

```
adaptive main [ name = sequencia, entry = bloco1,
                exit = imprime ] is

code bloco1 : <
     int n,k,sn;
     print "valor de n";
     read n;
     k  := 1;
     sn := k;
     if n>k then bloco1 := 1 else bloco1 := 2;
     >;
code imprime : <
     print sn;
     >;

connection from bloco1 ( case 1 : to imprime perform
c(bloco1)
      before otherwise to imprime);

adaptive function c(i)
     [ generators g ]
{
     insert [ code g : <
          k  := k + 1;
          sn := sn + 2 * k;
          if n>k then g := 1 else g := 2;
          >; ];

     insert [ connection from g (case 1 : to imprime
perform c(g) before otherwise to imprime) ];
}
end main
```

Quadro 5 – Programa-fonte relativo ao exemplo A.

O código executável pela máquina virtual referenciada na seção 3.5, gerado após a compilação desse programa-fonte, é mostrado no Apêndice D.

b) Exemplo B

Busca-se mostrar, nesse segundo exemplo, aspectos da linguagem não requeridos pelo exemplo A, tais como, ação adaptativa anterior e a existência de mais de uma função adaptativa.

O problema proposto em Neto (1993) e também discutido em Neto (2001), denominado coletor de nomes, implementa um autômato adaptativo que reconhece e memoriza cadeias de caracteres iniciadas por uma letra (λ), seguida (ou não) por letras ou dígitos numéricos (λ ou δ), será empregado neste exemplo B.

A função de transição que define esse autômato consta no Quadro 6 (adaptado de Neto (2001)).

Foi convencionado que uma cadeia de caracteres será encerrada pelo símbolo #, ou seja, esse símbolo deve estar presente no final de toda entrada, mas nunca será considerado um componente desta.

```
Função de transição do autômato básico
    (Start,λ)  : → I1, X(I,λ)•
    (I1,λ)     : → I1
    (I1,δ)     : → I1
    (I1,#)     : → L, • Y(J)
Função de transição da função adaptativa X
    X(p1,p2) = { g* :
      +[(p1, p2) : → g ]
      +[(g, λ)   : → I1, X(g, λ) • ]
      +[(g, δ)   : → I1, X(g, δ) • ]
      +[(g, #)   : → L, • Y(g)]
      –[(p1, p2) : → I1, X(p1, p2) • ] }
Função de transição da função adaptativa Y
    Y(q1) = {
      – [(q1,#) : → L, • Y(q1)]
      + [(q1,#) : → K ]}
```

Quadro 6 – Função de transição do coletor de nomes

A representação gráfica da topologia inicial deste autômato adaptativo consta da Figura 10.

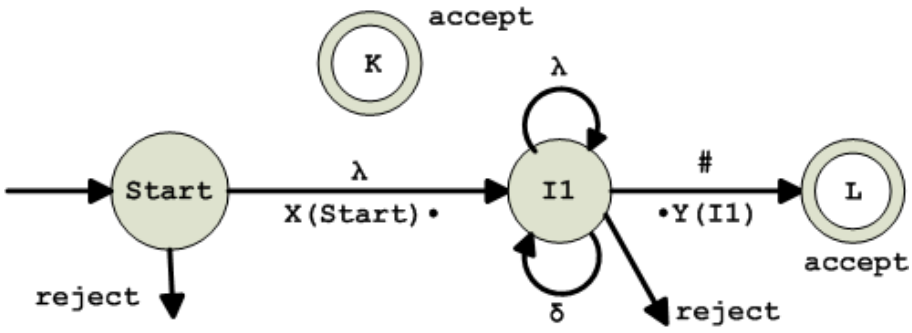

Figura 10. Autômato adaptativo “coletor de nomes”.

Devido às limitações do ambiente de execução, a entrada da cadeia a ser reconhecida será feita um símbolo por vez, através do seu respectivo código ASCII. São considerados caracteres alfabéticos (maiúsculos ou minúsculos) aqueles cujo código ASCII estiver na faixa de 65 a 122 e caracteres alfanuméricos na faixa de 48 a 122, sendo desconsiderados, por simplificação os caracteres não alfabéticos e não numéricos situados nessas duas faixas de valores. O caractere #, que encerra uma cadeia de entrada, tem código ASCII igual a 35.

Os estados `Start` e `I1`, do autômato da Figura 10, verificam o símbolo corrente da cadeia de entrada, rejeitando-o quando for inválido. Os estados K e L do mesmo autômato são estados de aceitação. Cada um deles constituirá um bloco básico do programa adaptativo. Por questões operacionais, foram acrescentados, na codificação do programa, dois blocos básicos, não presentes na Figura 10. Um, denominado `error`, que representa o estado de rejeição e outro denominado `out`, que encerra a execução do programa. O nome do estado `Start` foi renomeado para `cstart` no programa, por motivo da palavra *start* ser reservada do ambiente de execução, conforme Quadro 3.

Foram convencionados como valores de saída dos blocos básicos: 1 – quando for lido, na cadeia de entrada, λ ou δ, 2 – para valores inválidos, 4 – para símbolo #, que representa o final de uma cadeia de símbolos válidos, 8 – quando for

solicitado processar novos dados de entrada e 6 – para encerrar a execução do programa.

A representação gráfica da configuração inicial do programa assim projetado é mostrada na Figura 11.

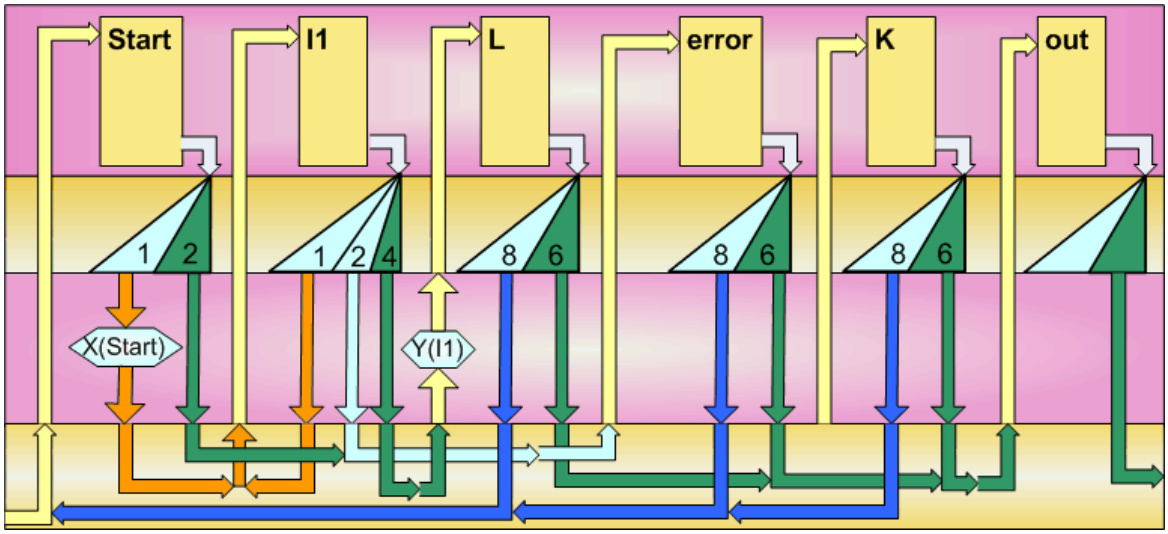

Figura 11 – Representação gráfica do programa do coletor de nomes

Pela definição do problema, somente haverá uma transição para o bloco básico K após a execução da função adaptativa Y e esta só será executada quando for encontrado o símbolo de final de cadeia de entrada. A ausência dessa transição para o estado K no autômato da Figura 10 está, assim, representada na Figura 11 pela ausência de uma conexão para o bloco básico K.

O programa codificado na linguagem proposta é apresentado a seguir, no Quadro 7.

```
ADAPTIVE MAIN [ NAME=coletor, ENTRY=cstart, EXIT=out ] IS
/* codes */
  CODE cstart : <
      int v;
      read v;
      if v>122 then cstart := 2 else cstart :=1 ;
      if v<65  then cstart := 2;   >;
CODE I1    : <
        read v;
      if v>122 then I1 := 2 else I1 := 1 ;
      if v<48  then I1 := 2;
        if v = 35 then I1 := 4 ;   >;
```

```
CODE L : <
       int c;
       print "aceita.";
       print "digite 1 para continuar outro para encerrar";
       read c;
       if c = 1 then L:=8 else L:=6 >;
  CODE error : < print "rejeita.";
       print "digite 1 para continuar outro para encerrar";
       read c;
        if c = 1 then error:=8 else error:=6 >;
  CODE K     : < print "aceita.";
       print "digite 1 para continuar outro para encerrar";
       read c;
      if c = 1 then K:=8 else K:=6 >;
  CODE out   : < print "saindo..." >;

/* conexoes entre CODEs */
  CONNECTION FROM cstart   (
     CASE 1 : TO I1 PERFORM X(cstart) BEFORE
     OTHERWISE TO error     ) ;
  CONNECTION FROM I1 (
     CASE 1 : TO I1,
     CASE 2 : TO error
     OTHERWISE TO L PERFORM Y(I1) AFTER ) ;
  CONNECTION FROM L  (
     CASE 6 : TO out
     OTHERWISE TO cstart    ) ;
  CONNECTION FROM error   (
     CASE 6 : TO out
     OTHERWISE TO cstart   ) ;
  CONNECTION FROM K  (
     CASE 6 : TO out
     OTHERWISE TO cstart   ) ;

/* funcoes adaptativas */
ADAPTIVE FUNCTION X ( p1 ) [ GENERATORS G ]
  {
    INSERT [ CODE g : <
     read v;
     if v > 65 then if v<130 then g := 1
          else g := 2;
     if v = 55 then g := 4;    >;];
  INSERT   [ CONNECTION FROM g
  (   CASE 1 : TO I1 PERFORM X ( g ) BEFORE ,
     CASE 2 : TO error
     OTHERWISE TO L PERFORM Y(g) AFTER ) ] ;
 }
ADAPTIVE FUNCTION Y ( pJ )
 {
    INSERT      [ CONNECTION FROM pJ (
    CASE 4 : TO K OTHERWISE TO out ) ] ; }
/* Final do programa */
END MAIN
```

Quadro 7 – Programa adaptativo do coletor de nomes

O código executável correspondente ao programa do Quadro 7 é apresentado no Apêndice E

5 CONSIDERAÇÕES FINAIS

Este capítulo apresenta os resultados das pesquisas realizadas e registradas nesta dissertação, as contribuições para a pesquisa das técnicas adaptativas, as sugestões de trabalhos futuros que poderão complementar, ou estender, as pesquisas realizadas e as conclusões que podem ser tiradas da execução deste trabalho.

5.1 CONTRIBUIÇÕES

Esta seção apresenta as contribuições do presente trabalho para as pesquisas relacionadas com as técnicas adaptativas.

Foram levantados requisitos e apresentados critérios e técnicas de projeto e de implementação para uma linguagem de programação com características especiais, apropriadas para uma particular classe de aplicações.

A linguagem básica operante disponibilizada, para programação adaptativa, completa, dessa forma, um ciclo de pesquisas sobre o assunto, e deixa por fazer a criação de uma linguagem completa de grande porte, para o desenvolvimento de aplicações adaptativas que sejam executadas eficientemente em qualquer máquina e sob qualquer sistema operacional.

A criação de uma camada adaptativa padrão, utilizável com outras linguagens-base imperativas de alto nível mediante pouquíssimos ajustes, constitui-se em outra contribuição do presente trabalho.

A formulação e proposta de uma linguagem de pequeno porte, de alto nível, imperativa, com a qual seja possível desenvolver programação adaptativa através da instanciação dos programas auto-modificáveis como casos particulares de dispositivos adaptativos guiados por regras.

Foram discutidas as técnicas adaptatividade, recursividade, macros e iteração, todas observadas do ponto de vista da expressão de fenômenos computacionais repetitivos, evidenciando a relação existente entre elas.

Apresentação de uma série de recomendações para o desenvolvimento organizado de programas auto-modificáveis.

Proposta de uma notação gráfica para representar e documentar a parte adaptativa da arquitetura de programas auto-modificáveis.

Proposta de um esquema aderente às recomendações e à notação sugeridas, para a documentação e codificação de programas adaptativos.

Apresentação de um exemplo, a título de ilustração da utilização das técnicas e notações propostas nesta dissertação .

5.2 TRABALHOS FUTUROS

Sugere-se, como trabalho futuro, o uso de uma linguagem-base mais potente, preferencialmente uma das utilizadas comercialmente.

Outra possibilidade que poderá ser buscada no futuro é a apresentação de uma solução similar usando linguagem-base orientada a objetos.

Desenvolver outras formas de implementação de programas adaptativos com o auxílio de linguagens de programação adequadas, aderentes aos tipos de problemas que se deseja resolver, também se apresenta como possibilidade de futuros trabalhos.

Outro trabalho futuro pode ser a criação de um ambiente, baseado na camada adaptativa proposta, capaz de dar suporte ao desenvolvimento de outras linguagens para programação adaptativa.

A criação de uma ferramenta de auxílio ao desenvolvimento de compiladores para esse tipo de linguagens poderá ser objeto de outros trabalhos no futuro.

Fundir em uma ferramenta as técnicas aqui propostas com o método de projeto apresentado por Wagner et al (2006), que realiza a modelagem de projetos através de máquinas de estado, também poderá se constituir em trabalho futuro.

A partir de situações-problema, investigar quando é mais apropriado desenvolver uma solução usando recursão, iteração, macro ou adaptatividade.

Fazer uma verificação formal da relação entre iteração, recursividade, macro e adaptatividade.

5.3 CONCLUSÃO

Esta dissertação apresenta uma linguagem de alto nível, que permite programar códigos adaptativos. Apesar de ser uma linguagem bastante limitada, proporciona os recursos mínimos, com os quais é possível produzir códigos adaptativos, através de recursos que se mostram capazes de solucionar problemas diversos de forma adaptativa.

Resumiu-se, inicialmente, o estudo dos objetivos, mostrando-se uma possível forma de levantamento das principais características desejadas para a linguagem resultante.

Mostrou-se, também, a necessidade de estabelecimento de um modelo abstrato para os programas a serem desenvolvidos nessa linguagem, realçando-se a forma como a escolha desse modelo pode interferir e auxiliar na concepção da própria linguagem.

A linguagem apresentada neste trabalho vem completar um ciclo de pesquisas, que passou por diversas fases, entre as quais o levantamento de requisitos e a construção de um ambiente de execução aderente ao estilo adaptativo.

O ambiente de execução, diversas vezes referenciado ao longo do trabalho, como é constituído por linguagem de baixo nível, sem um gerador automático de código em sua linguagem, tem seu uso não muito prático, da mesma forma que acontece com as linguagens de montagem, também referenciadas nesta pesquisa. A existência de uma linguagem de alto nível e o respectivo compilador, para geração de código executável, proporciona um melhor aproveitamento desse ambiente de execução, da mesma forma que acontece com as linguagens de programação já consagradas pelo seu uso.

Espera-se que as recomendações de projeto de programa apresentadas, partindo da especificação do problema, possam servir como ponto de partida para a definição de um método com essa finalidade.

Espera-se que, em breve, sejam construídas linguagens mais poderosas e mais expressivas, que deverão ser utilizadas de maneira confortável para a criação de soluções adaptativas para problemas práticos complexos.

Há muito a fazer ainda nessa área, antes que se possa constatar uma plena disponibilidade de recursos tecnológicos adequados e aderentes ao estilo adaptativo de programação.

Isso deverá acontecer quando se dispuser de uma base matemática mais completa e bem fundamentada, de linguagens expressivas e com recursos diversificados e poderosos, acompanhadas de metodologias e ambientes adequados de projeto e implementação de aplicativos, além de ferramentas para apoio à concepção, documentação, projeto, ensaio, depuração e validação desses programas adaptativos.

REFERÊNCIAS

ABELSON, H.; SUSSMAN, G. J. **Structure and Interpretation of computer program**. MIT Press. 1996. Versão eletrônica disponível em http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-2.html, acesso em 22/06/2010.

ALGHAMDI, J.; URBAN, J. **Comparing and Assessing Programming Languages: Basis for a Qualitative Methodology**. Proceedings of ACM/SIGAPP, pag. 222-229. 1993.

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

BAKUS, J. W. et al. **Revised Report on the Algorithmic Language ALGOL 60**. Communications of the ACM, Vol. 6 Issue 1, pp. 1-17, Jan. 1963.

BIRD, R. S. **Recursion elimination with variable parameters**. The Computer Journal 1979 22(2):151-154; doi:10.1093/comjnl/22.2.151 by British Computer Society. 1979.

BUZEN, J. P., GAGLIARDI, U. O**. The evolution of virtual machine architecture** ACM, proceedings of the National Computer Conference and Exposition, New York, doi: 10.1145/1499586.1499667. 1973.

CAI, H., SHAO, Z., VAYNBERG, A. **Certified self-modifying code**. LDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. ACM 978-1-59593-633-2. 2007a.

_____. **Certified self-modifying code** (extended version & coq implementation). Technical Report YALEU/DCS/TR-1379, Yale Univ., Dept. of Computer Science, Mar. 2007b. Disponível em http://flint.cs.yale.edu/publications/smc.html. Acesso em 28/10/2009.

CASTRO JÚNIOR, Amaury A. de. **Aspectos de projeto e implementação de linguagens para codificação de programas adaptativos**. Tese de Doutorado, apresentada à Escola Politécnica da Universidade de São Paulo. 2009.

DAHL, O. J., NYGAARD, K. SIMULA – an **Algol-Based Simulation Language**. Communicationa of the ACM, volume 9, número 9. Pag. 671-678. 1966.

FREITAS, Aparecido V. de. **Considerações sobre o desenvolvimento de linguagens adaptativas de programação**. Tese de Doutorado, apresentada à Escola Politécnica da Universidade de São Paulo. 2008.

GIFFIN, J. T., CHRISTODORESCU, M., KRUGER, L. **Strenghtening software self-checksumming via self-modifying code**. Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC) 2005.

IWAI, Margarete K. **Um formalismo gramatical adaptativo para linguagens dependentes de contexto**. Tese de doutorado. Escola Politécnica da UPS. São Paulo. 2000.

JESUS, L.; SANTOS, D. G.; CASTRO JR., A. A.; PISTORI, H. **AdapTools 2.0: Aspectos de Implementação e Utilização.** Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. (p. 527-532).

KANZAKI, Y. et al. **Exploiting selfmodication mechanism for program protection**. In Proc. of the 27th Annual International Computer Software and Applications Conference, pages 170-181, 2003.

LÉVÉNEZ, E. **Computer Languages History**, disponível em http://www.levenez.com/lang/, acesso em 05/06/2011.

LIU, Y. A., STOLLER, S. C. Stoller. **From Recursion to iteration**. **What are the optimization**. In Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Programming Manipulation (PEPM ’00), pages 73-82. ACM Press, Jan. 2000.

LTA – Laboratório de Linguagens e Técnicas Adaptativas, da Escola Politécnica da Universidade de São Paulo, disponível em http://www.pcs.usp.br/~lta/, acesso em 25/01/2010.

MADOU, M. et al. **Software Protection Through Dynamic Code Mutation**. Wisa 2005. LNCS 3786, pag. 194-206. 2006.

McKINLEY, Philip. K. et al. **A Taxonomy of Compositional Adaptation**. Technical Report MSU-CSU-04-17. Michigan State University, July, 2004.

NETO, J. J. **Introdução à compilação**. LTC: São Paulo. 1987.

_____**. Contribuições à metodologia de construção de compiladores**. Tese de livre docência, apresentada à Escola Politécnica da USP. São Paulo. 1993.

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

_____. **Um levantamento da evolução da adaptatividade e da tecnologia adaptativa**. Revista IEEE América Latina. Vol. 5, Num. 7, ISSN: 1548-0992, Novembro 2007. (p. 496-505).

_____. **Um levantamento da pesquisa em técnicas adaptativas na EPUSP**. Revista de Sistemas e Computação, Salvador, v. 1, n. 1, p. 23-47, jan./jun. 2011

NETO, J. J.; LEONARDI, F. ; PARIENTE, C. A. B. . **Compiler construction** - a Pedagogical approach. In: V Congreso Internacional en Ingeniería en Informática, 1999, Buenos Aires. Anales del V Congreso Internacional en Ingeniería en Informática, 1998.

PELEGRINI, Éder J. **Códigos Adaptativos e linguagem para programação adaptativa: conceitos e tecnologia**. Dissertação de Mestrado, apresentada à Escola Politécnica da USP. 2009.

PEREIRA, J. C. D. e NETO, J. J. **Um Ambiente de Desenvolvimento de Reconhecedores Sintáticos Baseado em Autômatos Adaptativos**. II Simpósio Brasileiro de Linguagens de Programação - SBLP97. pp. 139-150, Campinas, 1997.

PEREIRA, J. C. D. **Ambiente Integrado de Desenvolvimento de Reconhecedores Sintáticos, baseado em autômatos adaptativos**. Dissertação de mestrado, apresentada à Escola Politécnica da Universidade de São Paulo. 1999.

PISTORI, H. e NETO, J. J. **AdapTools: Aspectos de Implementação e Utilização.** Boletim Técnico PCS, Escola Politécnica, São Paulo, 2003.

PISTORI, Hemerson. **Tecnologia adaptativa em Engenharia de Computação: estado da arte a aplicações**. Tese de Doutorado apresentada à Escola Politécnica da USP. São Paulo. 2003.

RIEHL, J. **Reflective Techniques in extensible languages**. Tese de PhD, submetida à Faculdade da Divisão de Ciências Físicas da Universidade de Chicago, Illinois, EUA. 2008.

RUDIGER, R. **Extensible programming in Oberon. A tutorial**. Technical Report. FH Braunchweig /Wolfenbuttel – University of Applied Science. 1999.

SCHUMAN, S. **Definition mechanisms in extensible programming languages**. AFIPS Joint Computer Conferences. ACM Proceedings of the November 17-19, 1970, fall joint computer conference. Pages: 9-20. 1970.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. 9ª. Ed. São Paulo: Bookman, 2010.

SILVA, S.R.B. ; NETO, J. J. . **Um método para a programação de software adaptativo**. In: CACIC 2010 - Congreso Argentino de Ciencias de la Computación, 2010, Buenos Aires. Memorias del CACIC 2010 - Congreso Argentino de Ciencias de la Computación, 2010. p. 1-10.

SKLIAROWA, I.; SKLYAROV, V. **Recursion in Reconfigurable Computing: a survey of implementation approaches**. IEEE, Digital Object Identifier 978-1-4244-3892-1. Pages 224 – 229. 2009.

SLONNEGER, K; KURTZ, B. L. **Formal Sintax and Semantics of Programming Languages**, a laboratory approach. United States of America: Addison Wesley Publisher, 1995.

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

TSCHUDIN, Christian, YAMAMOTO, Lidia. **Harnessing Self-Modifying Code for Resilient Software**. Proc 2nd IEEE Workshop on Radical Agent Concepts (WRAC), NASA Goddard Space Flight Center Visitor's Center, Greenbelt, MD, USA, September 2005. In: Innovative Concepts for Autonomic and Agent-Based Systems,Springer LNCS 3825. Pages: 197-204. 2006.

TUCKER, B. A.; NOONAN, R. E. **Linguagens de programação**, princípios e práticas. São Paulo: MacGraw Hill, 2009.

WAGNER, F. et al. **Modeling Software with Finite State Machines**. Crc Pres: new York, USA, 2006.

WATSON, D. **High-Level Languages and Their Compilers**. Addison-Wesley Publishers Limited. Great Britain, 1989.

WILSON, G. V. **Extensible Programming for the 21st Century**. ACMQUEUE, 2004. Disponível em http://queue.acm.org/detail.cfm?id=1039534, acessado em abril de 2011.

WIRTH, N. **What Can We do about the Unnecessary Diversity of Notation for Syntatic Definitions?** Communications of the ACM , Volume 20 Issue 11, Nov. 1977, pag. 822-823.

APÊNDICE A – Notação de Wirth

Anotação de Wirth, definida por ela própria é apresentada abaixo.

syntax = {produção}.

produção = identificador “=” expressão “.”.

esxpressão = termo {“|” termo}.

termo = fator {fator}.

fator = identificador | literal | “(“ expressão “)” | “[“ expressão “]”

| “{“ expressão “}”.

literal = “””” caractere {caractere} “””””.

Nessa definição, a palavra *identificador* é usada para denotar *símbolo não terminal* e *literal* para *símbolos terminais*. Para reduzir a descrição dessa metalinguagem, *identificador* e *caractere* não são definidos em detalhes.

De acordo com a notação de Wirth, expressões entre chaves indicam que podem ocorrer *n* vezes, com *n* igual a zero ou mais ocorrências, ou seja, {z} significa que o símbolo z pode não ocorrer, ocorrer uma vez ou se repetir um número arbitrário de vezes. Parênteses são empregados para agrupar e uma barra vertical | separa entre si um conjunto de expressões que podem ser instanciadas arbitrariamente. Colchetes agrupam expressões opcionais, ou seja, que podem ser instanciadas zero ou uma vez.

Assim, [(a|b)c]d significa acd|bcd, ou d, isto é, se ocorre a ou b, necessariamente, ocorre cd em seguida, e em caso contrário ocorre apenas d. Símbolos terminais são colocados entre aspas.

APÊNDICE B – Estrutura sintática da linguagem proposta

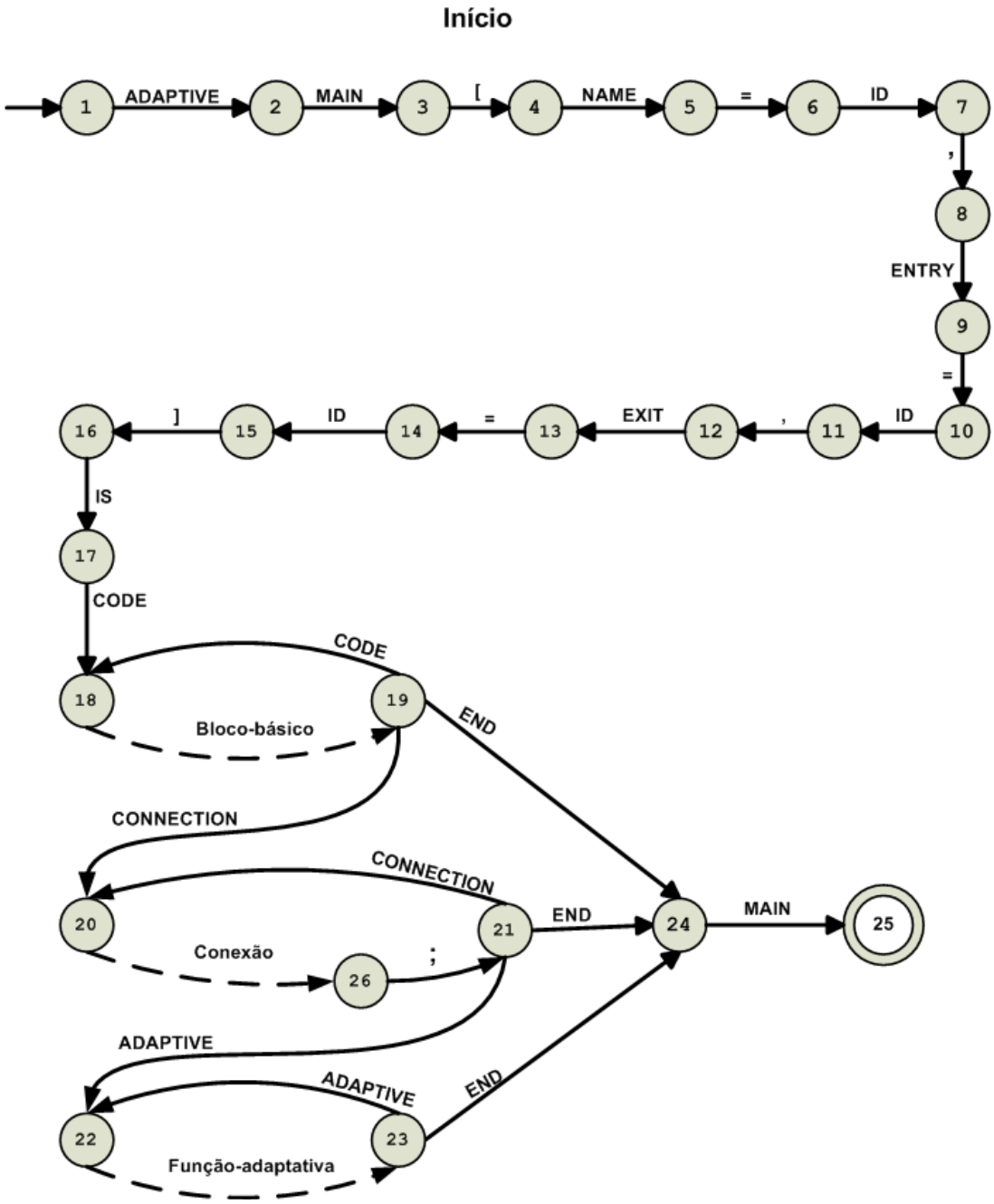

Figura 8 – Submáquina Início

A submáquina Inicio, constante da Figura 8, reconhece a declaração inicial do programa, onde são especificados o NAME, o nome do bloco básico (ENTRY = <nome>) pelo qual é iniciada a execução do programa e o nome do bloco básico (EXIT = <nome>) que finaliza a execução do mesmo.

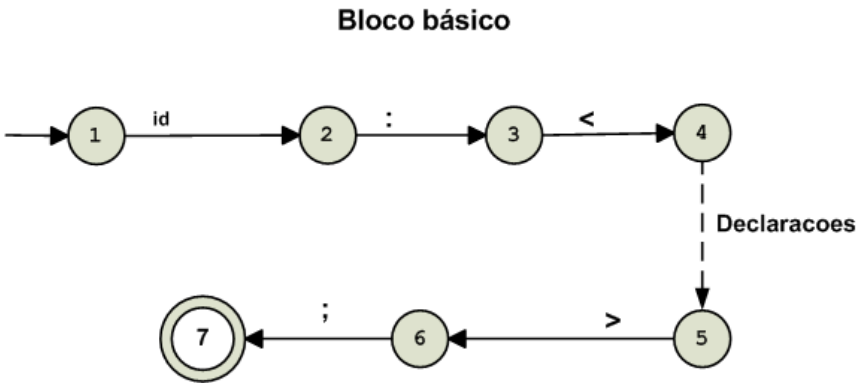

Figura 9 – Submáquina Bloco Básico

A submáquina Bloco Básico, apresentada na Figura 9, reconhece o início e final de um bloco básico. As declarações e comandos contidos em cada bloco básico serão reconhecidos a partir da chamada da submáquina Declarações.

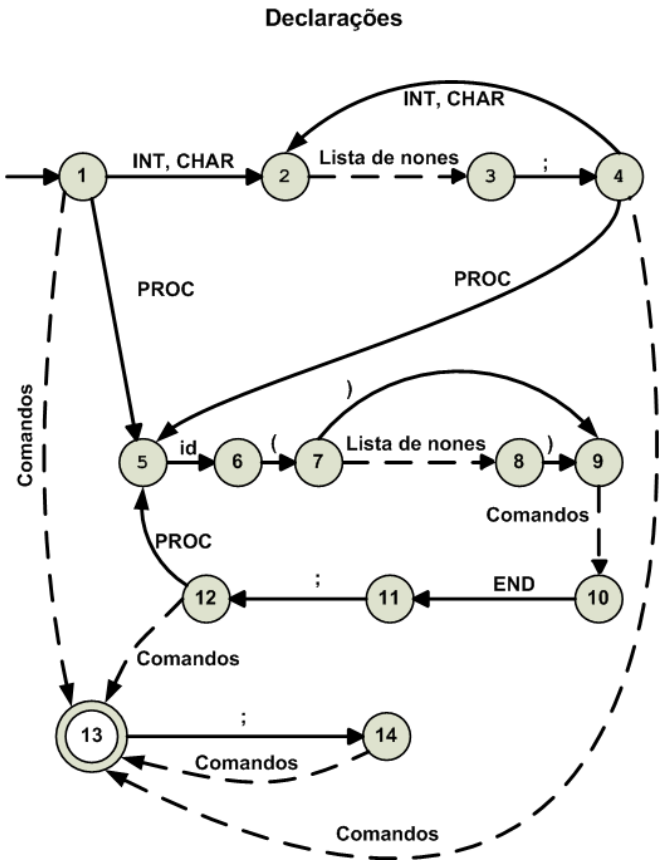

Figura 10 – Submáquina declarações

A submáquina Declarações realiza o reconhecimento de declarações de variáveis e de procedimentos, conforme a Figura 10.

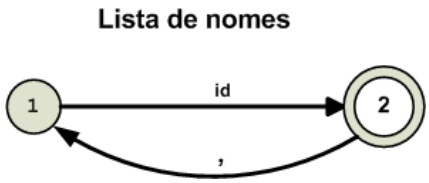

Figura 11 – Submáquina Lista de Nomes

Uma lista de identificadores separados por vírgula pode ser reconhecida pela submáquina Lista de nomes, apresentada na Figura 11.

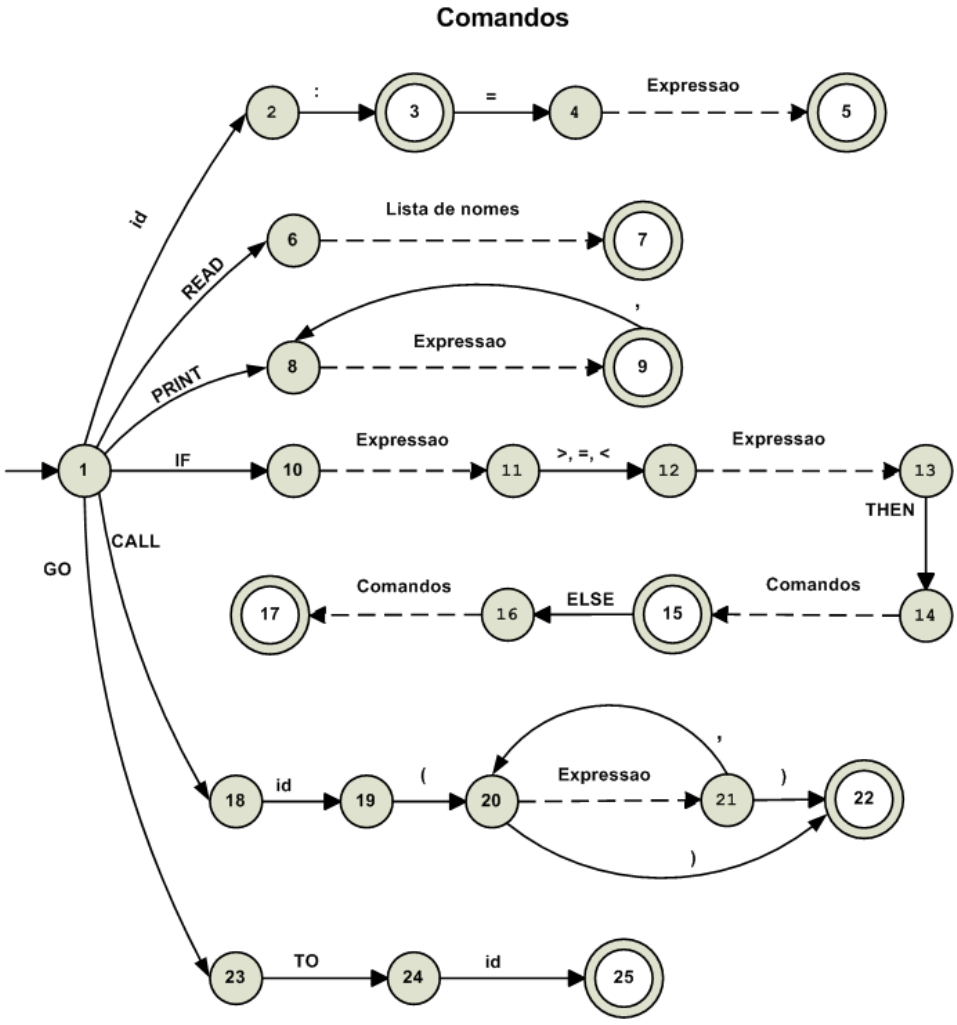

Figura 12 – Submáquina Comandos

A submáquina Comandos, representada na Figura 12, realiza o reconhecimento dos componentes disponíveis na linguagem hospedeira, à exceção da declaração de variáveis e das expressões, que são analisadas por submáquinas específicas.

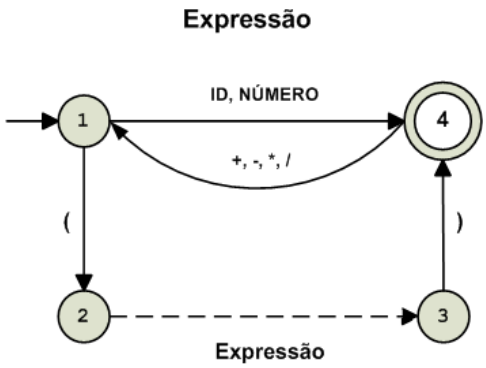

Figura 13 – Submáquina Expressão.

Expressões aritméticas de soma, subtração, multiplicação e divisão, entre parênteses ou não, têm o seu reconhecimento efetuado pela submáquina Expressão, constante da Figura 13.

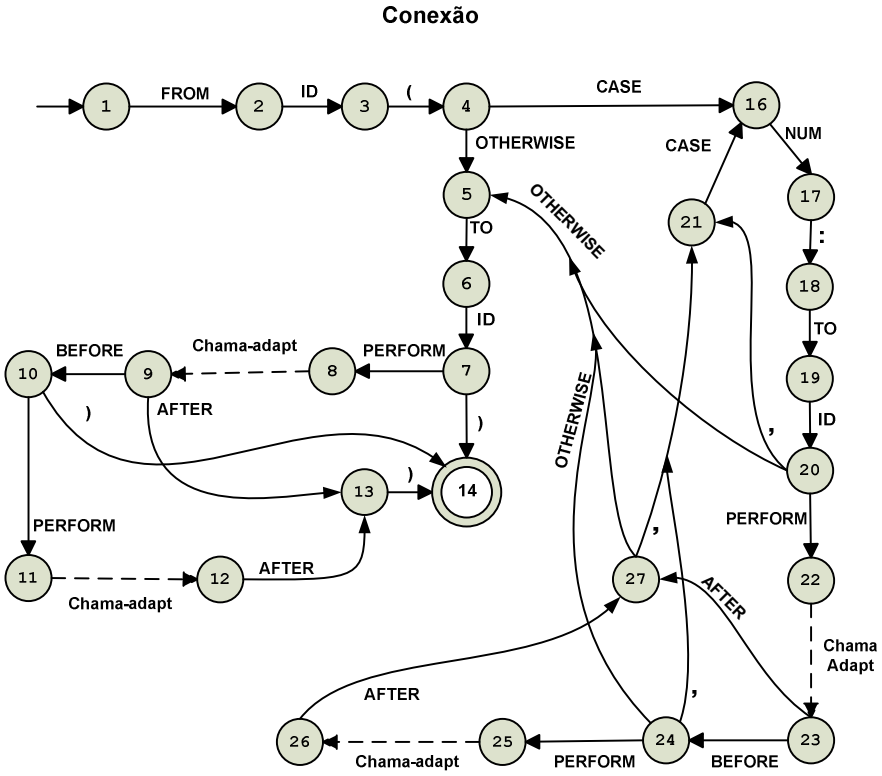

Figura 14 – Submáquina conexão

A Figura 14 representa a submáquina Conexão, que realiza o reconhecimento do elemento conexão. Uma conexão interliga blocos básicos e, opcionalmente,

poderá executar a chamada a uma função adaptativa, sendo esta chamada reconhecida pela submáquina Chama-adapt, apresentada na Figura 15.

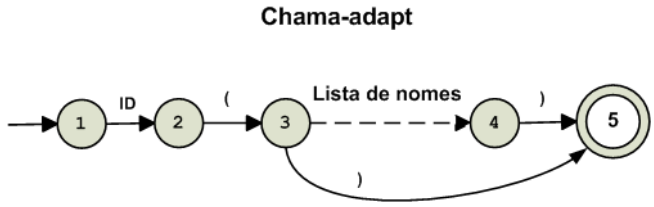

Figura 15 – Submáquina Chama-adapt

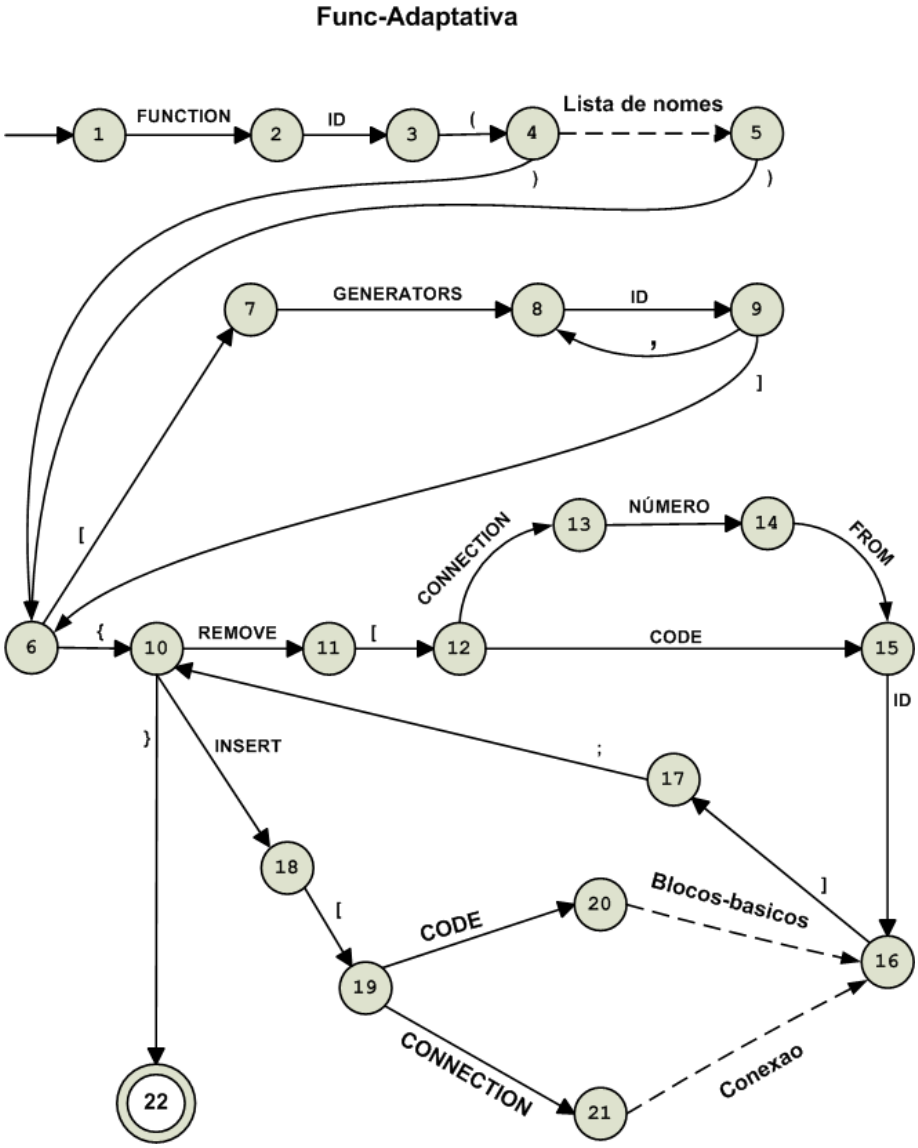

Figura 16 – Submáquina Func-adaptativa

Finalmente, a estrutura de uma função adaptativa pode ser reconhecida pela submáquina Func-adaptativa, apresentada na Figura 16.

APÊNDICE C – Semântica operacional da linguagem

Hipótese: a geração é de código-fonte para a linguagem da máquina virtual de execução. Adotou-se, por convenção, que o código será gerado em duas áreas: a de declarações e a de topologia. A última se destina a guardar o programa principal da linguagem da máquina virtual.

1. Estrutura geral do programa

programa-adaptativo =
“ADAPTIVE” “MAIN” “[” “NAME” “=” nome “,” “ENTRY” “=” nome “,” “EXIT” “=” nome “]” “IS”
declarações-adaptativas “END” “MAIN” “.” .

Gramática	Comentário	Semântica
“ADAPTIVE” “MAIN”	Inicio físico da declaração do programa	**Gerar abertura de programa principal na área de topologia. Por exemplo: “main proc” .**
“[”	Inicia as 3 informações do programa	**Não gera nada.**
“NAME” “=” nome “,”	Nome do programa	**Memorizar o <nome> para batizar o arquivo de saída do compilador.**
“ENTRY” “=” nome “,”	Nome do trecho identificado onde o programa inicia sua execução	**Gerar desvio incondicional para um rótulo start<nome>, associado ao “code” indicado. Por exemplo, (sendo start o código em questão):** **“jmp start<nome> ”**
“EXIT” “=” nome	Nome do trecho identificado onde o programa termina sua execução	**Memorizar o <nome> para dar o devido tratamento à saída do “code” que termina a execução do programa.**
“]”	Finaliza as 3 informações	**Não gera nada.**

	do programa	
“IS”		**Não gera nada.**
declarações-adaptativas	Corpo do programa (detalhado no item 2)	**... ver item 2 ...**
“END” “MAIN” “.” .	Final físico da declaração do programa	**Gerar final de programa principal. Por exemplo: “main endproc”**

2. Corpo do programa (declarações-adaptativas)

```
declarações-adaptativas = decl-trechos-identificados “;”
                          decl-conexões-adaptativas “;”
                          decl-funções-adaptativas .
```

Gramática	Comentário	Semântica
decl-trechos-identificados “;”	Em primeiro lugar, figuram as declarações dos trechos identificados (“codes”)	**... ver item 3 ...**
decl-conexões-adaptativas “;”	Em seguida, são declaradas as conexões condicionais entre os “codes”	**... ver item 5 ...**
decl-funções-adaptativas .	Finalmente, devem ser definidas as funções adaptativas utilizadas no programa	**... ver item 9 ...**

3. Declaração dos Trechos identificados usados pelo programa (lista de declarações de “codes”)

decl-trechos-identificados = [decl-trecho-identificado { “;” decl-trecho-identificado }] .

Gramática	Comentário	Semântica
[	Esta forma sintática, como um todo, é opcional, podendo ser omitida.	**Não gerar nada**
decl-trecho-identificado	Se houver declarações de “codes”, esta é a primeira instância (Detalhada no item 4).	**... ver item 4 ...**
{	Se houver outras declarações de “codes”, figuram a seguir,...	**Não gerar nada**
“;”	... separadas por ponto-e-vírgulas.	**Não gerar nada**
decl-trecho-identificado	Igual à anterior (Detalhada no item 4).	**... ver item 4 ...**
}	Final do loop de instâncias.	**Não gerar nada**
] .	Final da forma sintática opcional.	**Não gerar nada**

4. Sintaxe de cada Trecho identificado (declarações de cada “code”)

decl-trecho-identificado = “CODE” nome “:” “<” código-hospedeira “>” .
código-hospedeira = << qualquer código c/ só 1 entrada e 1 saída na linguagem hospedeira >> .

Gramática	Comentário	Semântica
“CODE” nome “:”	Nome atribuído ao trecho identificado.	**Iniciar a geração do novo bloco básico <nome>. Memorizar <nome>, para atribuir, logo adiante, um valor-default de saída à função que irá implementar este “code”. Gerar, na área de declarações uma declaração do tipo inteiro para <nome>.** **Exemplo:** **“label start <nome> ” na área de topologia** **“int <nome>” na área de declarações**
“<”	Delimitador esquerdo.	**Não gerar nada.**
código-hospedeira		
INT <n1>,<n2>...	Declaração de variáveis	**Declaração de uma lista de variáveis tipo inteiro.** **Exemplo: int <n1>,<n2>...**
CHAR <n1>,<n2>...	Declaração de variáveis	**Declaração de uma lista de variáveis tipo cadeia.** **Exemplo: string <n1>,<n2>...**
READ <nome>	Comando de leitura	**Faz a leitura para uma variável <nome> tipo inteiro** **Exemplo: inputint <nome>**

PRINT <expressão>	Comando de saída	**imprime o conteúdo de uma variável inteira <nome>, resultante do cálculo de <expressão>** **Exemplo: printint <nome>**
IF <comparação> **THEN <comando>** **ELSE <comando>**	Desvio condicional	**Caso <comparação> resulte em valor lógico verdadeiro executa <comando> da cláusula THEN, caso seja falso, executa <comando> da cláusula ELSE** **Exemplo: IF a>b THEN <COMANDo1> ELSE <comando2>** **cmp a,b** **jle lbl1** **<COMANDO1>** **jmp label2** **label lbl1** **<comando2>** **label lbl2**
nome ":=" **expressão**	Comando de atribuição	**Atribui a <nome> o resultado de uma <expressão>.** **Exemplo: c = a;** **mov c,a**
<expressão>	Pode ser uma variável, um número ou uma expressão aritmética	**O cálculo de uma expressão será feito transformando-a em notação pós-fixada e resolvendo-a com a criação de variáveis auxiliares. Não haverá otimização de código.** **Exemplo: a = b + c;** **bc+** **mov aux1,b → atribui o valor da variável b à variável aux1.**

“GO” “TO” nome **<nome>:** **a + b** **a – b** **a * b** **a / b**	Desvio incondicional Label <nome> Operação de soma Operação de subtração Operação de multiplicação Operação de divisão	**mov aux2,c** → **atribui o valor da variável c à variável aux2.** **add aux1,aux2** → **soma o valor das variáveis aux1 e aux2 e atribui o resultado à variável aux1.** **mov a,aux1** → **atribui o valor da variável aux1 à variável a.** **Desvia o fluxo de execução para o label <nome>** **Exemplo: jmp <nome>** **Não realiza nenhuma ação semântica.** **label <nome>** **add a,b** **sub a,b** **mul a,b** **div a,b**
“>” .	Delimitador direito.	**Gerar, na área das declarações, o delimitador direito do corpo do bloco básico. Por exemplo: “end <nome>”**

5. Sintaxe das conexões adaptativas (declarações de cada “code”)

decl-conexões-adaptativas = [conexão-adaptativa { “;” conexão-adaptativa }] .

Gramática	Comentário	Semântica
[	Esta forma sintática, como um todo, é opcional, podendo ser omitida.	**Não gerar nada**
conexão-adaptativa	Se houver declarações de conexões, esta é a primeira delas (Detalhada no item 6).	**... ver item 6 ...**
{	Se houver outras conexões adaptativas declaradas, elas são listadas a seguir,...	**Não gerar nada**
“;”	... separadas por ponto-e-vírgulas.	**Não gerar nada**
conexão-adaptativa	A semântica dessas demais conexões é similar à da primeira (Detalhada no item 6).	**... ver item 6 ...**
}	Final do loop de instâncias.	**Não gerar nada**
] .	Final da forma sintática opcional.	**Não gerar nada**

6. Sintaxe da declaração de um grupo de conexões adaptativas (declarações de “connections”)

conexão-adaptativa = “CONNECTIONS” “FROM” nome “(” conexões “)” .

Gramática	Comentário	Semântica
“CONNECTIONS” “FROM” nome	Indica o nome do “code” a partir de cuja saída partem as conexões declaradas no conjunto de conexões listado a seguir	**Guardar o <nome> para comparar o valor armazenado, resultado do processamento do CODE de mesmo nome, com os valores das cláusulas CASE.**
“(”		**Não gerar nada**
conexões	conjunto de conexões que partem da saída do “code” <nome> (Detalhada no item 7).	**... ver item 7 ...**
“)” .		**Não gerar nada**

7. Sintaxe de uma lista de conexões adaptativas

conexões = conexão { “,” conexão } .

Gramática	Comentário	Semântica
conexão	Se houver várias conexões, esta é a primeira delas (Detalhada no item 8).	**... ver item 8 ...**
{	Se houver mais declarações de outras de conexões, elas são listadas a seguir,...	**Não gerar nada**
“,”	... separadas por vírgulas.	**Não gerar nada**
conexão	A semântica de cada conexão aqui listada é idêntica à da primeira (Detalhada no item 8).	**... ver item 8 ...**
} .	Final do loop de instâncias de conexões.	**Não gerar nada**

8. Sintaxe de cada conexão adaptativa individual

```
conexão = [ “CASE” número “:” ]
          “TO” nome
          [ “PERFORM” nome “(” [ nome { “,” nome } ] “)” “BEFORE” ]
          [ “PERFORM” nome “(” [ nome { “,” nome } ] “)” “AFTER” ] .
```

Gramática	Comentário	Semântica
[	A cláusula CASE é opcional.	**Não gerar nada**
“CASE” número “:”	Se esta cláusula estiver presente, o número serve como seletor das conexões.	**Gerar, na área de topologia, um comando de comparação do valor da variável <nome>, correspondente ao CODE de partida da conexão (“FROM” nome), com <número>.** **Exemplo: cmp <nome>,<numero>** **Caso resulte em valor lógico verdadeiro, realizar o desvio para o CODE da cláusula TO, a seguir.**
	Omitindo-se a cláusula CASE, a (única) conexão torna-se incondicional.	
“TO” nome	Indica o “code” cuja entrada é o destino da conexão	**Gerar na área de topologia o comando que complementa**
[	A cláusula PERFORM ... BEFORE a seguir é opcional, e corresponde à chamada de uma função adaptativa anterior.	**Não gerar nada**
“PERFORM” nome	Indica o nome da função adaptativa que se deseja chamar antes de	**Gerar, na área de topologia, a sequência de comandos que realizam a inserção dos comandos da função adaptativa <nome> indicada. Por exemplo**

	ser percorrida esta conexão.	**adapt <nome1>,step1** **mark <nome2>** **mark <nome1>** **emark <nome1>** **emark <nome2>** **Obs.:** **1) O compilador gerará step1 para a primeira função adaptativa, step2 para a segunda função adaptativa, etc.** **2) O compilador acrescenta os numerais 1 e 2 ao <nome> da função adaptativa para gerar os comandos mark e emark.**
"("	O abre-parênteses é obrigatório	**Não gerar nada**
[**nome** **{ "," nome }** **]**	Pode não haver parâmetros	**Neste caso, não gerar nada, na área de topologia.**
	Havendo parâmetros, seus nomes são separados por vírgulas quando mais de um.	**Neste caso, gerar, na área de declarações, a declaração das variáveis para a lista de <nome>s que representam os parâmetros. Por exemplo:** **"int PAR1"** **ou então, para mais de um** **"int PAR1 , PAR2, PAR3".**
")"	O fecha-parênteses é obrigatório	**Não gerar nada**
"BEFORE"	BEFORE indica que se trata de ação adaptativa anterior.	
]		**Não gerar nada**
[	A cláusula PERFORM ... AFTER a seguir	**Não gerar nada, na área de topologia.**

	corresponde à chamada de uma função adaptativa posterior, e é opcional.	
“PERFORM” nome	Indica o nome da função adaptativa que se deseja chamar depois de ser percorrida esta conexão.	**Idem acima**
“(”	O par de parênteses é obrigatório	**Não gerar nada**
[**nome** **{ “,” nome }** **]**	Pode não haver parâmetros	**Neste caso, não gerar nada, na área de topologia.**
	Havendo parâmetros, seus nomes são separados por vírgulas quando mais de um.	**Idem acima**
“)”	O fecha-parênteses é obrigatório	**Não gerar nada**
“AFTER”	AFTER indica que se trata de ação adaptativa posterior.	**Não gerar nada, na área de topologia.**
]		**Não gerar nada, na área de topologia.**

Declaração de um conjunto de funções adaptativas

decl-funções-adaptativas = [decl-função-adaptativa { “;” decl-função-adaptativa }] .

Gramática	Comentário	Semântica
[	Esta forma sintática, como um todo, é opcional, podendo ser omitida.	**Não gerar nada**
decl-função-adaptativa	Se houver declarações de um conjunto de funções adaptativas, esta será a primeira delas .	**... ver item 10 ...**
{	Se houver outras declarações de funções adaptativas, suas definições são listadas a seguir,...	**Não gerar nada**
“;”	... separadas por ponto-e-vírgulas.	**Não gerar nada**
decl-função-adaptativa	Igual à anterior (Detalhada no item 10).	**... ver item 10 ...**
}	Final do loop de instâncias.	**Não gerar nada**
] .	Final da forma sintática opcional.	**Não gerar nada**

9. Declaração de uma função adaptativa individual

```
decl-função-adaptativa =
    “ADAPTIVE” “FUNCTION” nome “(” [ nome { “,” nome } ] “)”
                              [ “[” “GENERATORS” nome { “,” nome } “]” ] “:”
            “{” {  “REMOVE” “[” ( “CODE” nome | conexão-adaptativa ) “]” “;”
                 | “INSERT” “[” ( decl-trecho-identificado | conexão-adaptativa ) “]”
“;”
                } “}”  .
```

Gramática	Comentário	Semântica
“ADAPTIVE” “FUNCTION” nome	Define inicialmente o nome da função adaptativa	**Gera a declaração do bloco de comandos que permitem a inserção adaptativa de outros comandos. Exemplo:** **addcmd base1, “mark <nome2>”** **addcmd base1, “mark <nome1>”** **addcmd base1, “emark <nome1>”** **addcmd base1, “emark <nome2>”**
“(”	O par de parênteses é obrigatório.	**Não gerar nada**
[nome { “,” nome }]	Mas a lista de parâmetros, separados por vírgulas quando mais de um, é opcional.	**Gerar, na área de topologia, os comandos de atribuição de valor que realizam a recepção dos valores passados por parâmetros.**
“)”		**Não gerar nada**
[“[” “GENERATORS”	Declaração de geradores, quando existe ...	**Não gerar nada**
nome	...depois o primeiro gerador...	
{ “,” nome }	... seguido pelos demais,	**Gerar, como antes, um comando de**

	separados por vírgulas, ...	**atribuição, ao <nome> declarado como gerador.**
“]”]	... terminando com fecha-colchetes.	**Não gerar nada**
“:”	Dois-pontos introduz o corpo da função adaptativa:	**Não gerar nada**
“{”	Iniciada por chave,	**Não gerar nada**
{	Seguida por um conjunto opcional de ações adaptativas	**Não gerar nada**
“REMOVE” “[”	De remoção...	**Gerar o comando que inicia o bloco de instruções a serem removidas**
(“CODE” nome	... de um “code” cujo nome é aqui especificado...	**Não gerar nada**
| “CONNECTION” número “FROM” nome)	... ou a conexão aptativa cujo número e nome do “code” de origem estão indicados	**Não gerar nada**
“]” “;”		**Gerar o comando que finaliza o bloco de instruções a serem removidas.**
| “INSERT” “[”	Ou de inserção...	**Todos os comandos de inserção dinâmica serão gerados iniciando por addcmd step1, “<comando>”**
(decl-trecho-identificado	... de um “code” (detalhada em 4)	**Não gerar nada**
| conexão-adaptativa)	... ou de uma conexão adaptativa (detalhada em 8)	**Não gerar nada**
“]” “;”		**Não gerar nada**
}		**Não gerar nada**
“}” .	Terminando a declaração da função adaptativa com chave	**Não gerar nada**

APÊNDICE D – Código executável do exemplo A

```
.data

int BLOCO1
int N, K, SN
int IMPRIME
int I
int G
int temp1
int temp2

string stepC, baseC

.code
start:
   call main
exit
main proc

resetstr stepC
resetstr baseC
addcmd baseC,"mark C2"
addcmd baseC,"mark C1"
addcmd baseC,"emark C1"
addcmd baseC,"emark C2"
addcmd stepC,"mov temp1,K"
addcmd stepC,"add temp1,1"
addcmd stepC,"mov K,temp1"
addcmd stepC,"mov temp1,2"
addcmd stepC,"mul temp1,K"
addcmd stepC,"mov temp2,SN"
addcmd stepC,"add temp2,temp1"
addcmd stepC,"mov SN,temp2"
addcmd stepC,"cmp N,K"
addcmd stepC,"jle LB06"
addcmd stepC,"mov G,1"
addcmd stepC,"jmp LB07"
addcmd stepC,"label LB06"
addcmd stepC,"mov G,2"
addcmd stepC,"label LB07"
addcmd stepC,"cmp G,1"
addcmd stepC,"jne LB08"
addcmd stepC,"adapt C1,stepC"
addcmd stepC,"mark C1"
addcmd stepC,"emark C1"
addcmd stepC,"jmp startIMPRIME"
addcmd stepC,"label LB08"
```

```
addcmd stepC,"jmp startIMPRIME"
addcmd stepC,"endp"

jmp startBLOCO1

label startBLOCO1
printstring "valor de n"
inputint N
mov K,1
mov SN,K
cmp N,K
jle LB02
mov BLOCO1,1
jmp LB03
label LB02
mov BLOCO1,2
label LB03
label endBLOCO1
cmp BLOCO1,1
jne LB04
adapt C1,stepC
mark C2
mark C1
emark C1
emark C2
jmp startIMPRIME
label LB04
jmp startIMPRIME

label startIMPRIME
printint SN
label endIMPRIME

ret
main endproc

end start
```

APÊNDICE E – Código executável do exemplo B

```
.data

int CSTART
int V
int I1
int L
int C
int ERROR
int K
int OUT
int P1
int G
int PJ

string stepX, baseX
string stepY, baseY

.code
start:
   call main
exit
main proc

resetstr stepX
resetstr baseX
addcmd baseX,"mark X2"
addcmd baseX,"mark X1"
addcmd baseX,"emark X1"
addcmd baseX,"emark X2"
addcmd stepX,"inputint V"
addcmd stepX,"cmp V,55"
addcmd stepX,"jne LB026"
addcmd stepX,"mov G,4"
addcmd stepX,"label LB026"
addcmd stepX,"cmp V,96"
addcmd stepX,"jle LB027"
addcmd stepX,"cmp V,130"
addcmd stepX,"jge LB028"
addcmd stepX,"mov G,1"
addcmd stepX,"jmp LB029"
addcmd stepX,"label LB028"
addcmd stepX,"mov G,2"
addcmd stepX,"label LB029"
addcmd stepX,"label LB027"
addcmd stepX,"cmp G,1"
addcmd stepX,"jne LB030"
addcmd stepX,"adapt X1,stepX"
```

```
addcmd stepX,"mark X1"
addcmd stepX,"emark X1"
addcmd stepX,"jmp startI1"
addcmd stepX,"label LB030"
addcmd stepX,"cmp G,2"
addcmd stepX,"jne LB031"
addcmd stepX,"jmp startERROR"
addcmd stepX,"label LB031"
addcmd stepX,"jmp startL"
addcmd stepX,"adapt Y1,stepY"
addcmd stepX,"mark Y1"
addcmd stepX,"emark Y1"
addcmd stepX,"endp"
resetstr stepY
resetstr baseY
addcmd baseY,"mark Y2"
addcmd baseY,"mark Y1"
addcmd baseY,"emark Y1"
addcmd baseY,"emark Y2"
addcmd stepY,"cmp PJ,4"
addcmd stepY,"jne LB032"
addcmd stepY,"jmp startK"
addcmd stepY,"label LB032"
addcmd stepY,"jmp startOUT"
addcmd stepY,"endp"


jmp startCSTART

label startCSTART
inputint V
cmp V,65
jge LB02
mov CSTART,2
label LB02
cmp V,122
jle LB03
mov CSTART,2
jmp LB04
label LB03
mov CSTART,1
label LB04
label endCSTART
cmp CSTART,1
jne LB05
adapt X1,stepX
mark X2
mark X1
emark X1
emark X2
jmp startI1
```

```
label LB05
jmp startERROR

label startI1
inputint V
cmp V,48
jge LB07
mov I1,2
label LB07
cmp V,122
jle LB08
mov I1,2
jmp LB09
label LB08
mov I1,1
label LB09
cmp V,35
jne LB010
mov I1,4
label LB010
label endI1
cmp I1,1
jne LB011
jmp startI1
label LB011
cmp I1,2
jne LB012
jmp startERROR
label LB012
jmp startL
adapt Y1,stepY
mark Y2
mark Y1
emark Y1
emark Y2

label startL
printstring "aceita."
printstring "digite 1 para continuar outro para encerrar"
inputint C
cmp C,1
jne LB014
mov L,8
jmp LB015
label LB014
mov L,6
label LB015
label endL
cmp L,6
jne LB016
jmp startOUT
```

```
label LB016
jmp startCSTART

label startERROR
printstring "rejeita."
printstring "digite 1 para continuar outro para encerrar"
inputint C
cmp C,1
jne LB018
mov ERROR,8
jmp LB019
label LB018
mov ERROR,6
label LB019
label endERROR
cmp ERROR,6
jne LB020
jmp startOUT
label LB020
jmp startCSTART

label startK
printstring "aceita."
printstring "digite 1 para continuar outro para encerrar"
inputint C
cmp C,1
jne LB022
mov K,8
jmp LB023
label LB022
mov K,6
label LB023
label endK
cmp K,6
jne LB024
jmp startOUT
label LB024
jmp startCSTART

label startOUT
printstring "saindo..."
label endOUT

ret
main endproc

end start
```