Amaury Antônio de Castro Junior

Aspectos de Projeto e Implementação de Linguagens para Codificação de Programas Adaptativos

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

São Paulo
2009

Amaury Antônio de Castro Junior

Aspectos de Projeto e Implementação de Linguagens para Codificação de Programas Adaptativos

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

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. João José Neto

São Paulo
2009

“(...) Deus me dá aos poucos, em partes, dia a dia, em fragmentos.

Eu Dele me recebo, assim como o girassol se recebe do sol, porque não pode sobreviver sem sua luz. A flor condensa, ainda que de forma limitada, porque é criatura, o todo de sua natureza que o sol potencializa.

O mesmo é comigo. O mesmo é com você. Deus é nosso sol, e nós não poderíamos chegar a ser quem somos, em essência, se Nele não colocarmos a direção dos nossos olhos. (...) ”

Pe. Fábio de Melo

À minha esposa, Caroline.

Agradecimentos

Agradeço a Deus pela sua constante presença em minha vida e, acima de tudo, por ter colocado tantas pessoas maravilhosas em minha vida.

À minha esposa, Caroline, a quem também dedico esta tese, pelo companheirismo, pelo respeito, pela amizade, pelo incentivo, pela paciência e, acima de tudo, pelo seu amor que me fez suportar todos os períodos longe de casa. Amo você...por toda a eternidade e com toda a força do meu ser!

Aos meus filhos, Pedro e Mariana, cuja inocência e alegria de criança serviram de inspiração para que eu pudesse alcançar essa vitória, sem questionar o futuro. Vocês são o presente mais lindo que Deus me deu!

Aos meus pais, Amaury e Teresa, e aos meus irmãos, Katiuscia e André, pelo carinho, pelo apoio e pela torcida. Nenhuma árvore resistiria aos vendavais se não fossem as suas sólidas raízes!

Ao meu orientador e amigo, João José Neto, pela paciência, pela atenção, pela orientação, pelos cafés e, acima de tudo, pela amizade. O carinho e a alegria com os quais sempre me recebia foram fudamentais para esta conquista. O seu profundo conhecimento técnico influenciaram o meu desenvolvimento profissional e acadêmico. As suas palavras e seus exemplos promoveram grandes transformações em minha vida.

Ao amigo e professor, Ricardo Rocha, pelos valiosos conselhos e animadas conversas no café.

A todos os meus familiares, pelas orações, pela confiança, pelas palavras de incentivo e pelo carinho.

Aos amigos, Fábio, Leonardo, Said e Rafael pela amizade, pelos saudosos momentos que passamos na república e pelas inúmeras acolhidas em SP. Aos “amigos-irmãos” Peralta e Marcelo Cintra pela torcida, pelo incentivo e pelas vibrações positivas.

Aos amigos, Jeferson, Gonda, Daniele e Leonardo pelo companheirismo e amizade durante o período dos créditos de doutorado.

Aos amigos, Hemerson, Aparecido e Almir pela participação na minha banca e, acima de tudo, pelas palavras de incentivo e valiosas contribuições para a melhoria do

texto.

Aos meus queridos alunos e ex-alunos, com os quais muito tenho aprendido durante todos esses anos de exercício da docência.

Aos amigos e colegas de trabalho da UCDB, CPCX/UFMS, CPPP/UFMS, POLI/USP e DCT/UFMS pelo apoio e amizade.

Finalmente, agradeço a todos os amigos que me acompanharam e me apoiaram durante o desenvolvimento deste trabalho mas que, por descuido, não foram citados. No entanto, certamente, por serem verdadeiros amigos, partilham da minha alegria pela conquista de mais esta vitória.

Resumo

Este trabalho apresenta um conjunto de contribuições teóricas e metodológicas para o projeto e a implementação de linguagens de programação, utilizando o autômato adaptativo como dispositivo formal para sua definição. A especificação completa de uma linguagem de programação envolve desde a compreensão adequada de princípios e fundamentos comuns entre todas as linguagens de programação, transparentes ao programador, até as suas formas e características externas. Embora muitos modelos e notações possam ser utilizados na formalização de diferentes aspectos envolvidos no projeto e na implementação das linguagens de programação, o autômato adaptativo demonstra alta aplicabilidade e adequação para uma definição completa da linguagem, sem a necessidade do uso de diferentes notações. Demonstra-se como os autômatos adaptativos podem ser utilizados como uma metalinguagem unificada para especificar todas as componentes relevantes da definição formal da linguagem de programação, tais como: análise léxica, reconhecimento da sintaxe livre de contexto e manipulação de alguns aspectos dependentes de contexto da linguagem - declaração e uso de nomes simbólicos, semântica estática, declaração e expansão de macros, entre outros. São apresentados os conceitos relacionados, e descrito os aspectos mais importantes da formalização proposta. Para isso, utiliza-se uma linguagem imperativa simplificada, sobre a qual é acoplado um mecanismo de extensão para torná-la extensível.

Abstract

This work presents a set of theoretical and methodological contributions to the design and implementation of programming languages, using the adaptive automaton as device for its formal definition. The complete specification of a programming language involves proper understanding of principles and common ground between all the programming languages, transparent to the programmer, and forms and external characteristics. Although many models and notations can be used to formalize different aspects involved in the design and implementation of programming languages, the adaptive automaton shows high applicability and suitability to full definition of the language, without the need to use distincts notations. It is shown how the adaptive automata can be used as a unified metalanguage to specify all the relevant components of the formal definition of programming language, such as lexical analysis, sintax context-free recognition and handling of context-dependent aspects of language - declaration and use of symbolic names, static semantics, definition and expansion of macros, and others. Concepts are shown and the most important aspects are described of the this formal proposal. A simple imperative language is used, on which is attached an extension mechanism to make it extensible.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

1.1 Motivação e Objetivos

Até a primeira metade da década de 50, a tarefa de programação de computadores era realizada através da linguagem de máquina, que consistia de longas sequências de números binários, correspondentes às instruções de máquina que codificavam as soluções de problemas práticos em computadores. Com a invenção dos mnemônicos, que consistiam de cadeias de caracteres que representavam os códigos de operação das instruções de máquina, a programação de computadores passou a ser realizada em linguagem de montagem, que eram convertidas no código de máquina equivalente através dos montadores. Entretanto, a tarefa de programação continuava complexa, devido aos problemas e limitações para expressão e compreensão de algoritmos codificados na forma de linguagem de montagem. Devido a esses problemas e limitações, o interesse pelo desenvolvimento de novos mecanismos de programação, que pudessem simplificar a tarefa de programação, ganhou grande atenção nos anos seguintes (MARCOTTY; LEDGARD, 1986), culminando com o desenvolvimento do FORTRAN e de tantas outras linguagens e paradigmas conhecidos atualmente (SEBESTA, 2006).

Neste contexto, pode-se considerar o estudo das linguagens de programação como uma das mais importantes manifestações de pesquisa na área de computação e informática. Todo o processo de projeto e implementação de uma linguagem de programação envolve desde a compreensão adequada de princípios e fundamentos comuns a todas as linguagens de programação (transparentes ao programador), até as suas formas e características externas (visíveis ao programador).

Além disso, a concepção dos autômatos adaptativos (NETO, 1993) estimulou a criação e o desenvolvimento da área das tecnologias adaptativas e seus diversos segmentos. Com a formalização genérica para os dispositivos adaptativos guiados por regras (NETO, 2001), surgiram diversas propostas de trabalho com foco na resolução de problemas práticos através da aplicação de modelos baseados em tais dispositivos. Fundamentalmente, a tecnologia adaptativa estuda o conjunto de propriedades que um sistema ou um dispositivo apresenta de modificar suas próprias regras de funciona-

mento, em função de seu histórico, sem interferência externa.

Motivado por tais aspectos e potencialidades das áreas de projeto de linguagens de programação e tecnologia adaptativa, este documento descreve um conjunto de avanços obtidos do estudo e da pesquisa que envolve o projeto e a implementação de uma linguagem para codificação de programas adaptativos, passando pela implementação de uma linguagem base, aderente ao modelo dos autômatos adaptativos, sobre a qual pode-se implementar novas construções sintáticas capazes de incorporar a extensibilidade de linguagens e a adaptatividade sobre programas em execução. Um dos objetivos deste trabalho é o de validar resultados práticos e conceituais previamente obtidos, contribuindo com o desenvolvimento dos segmentos teóricos, metodológicos e paradigmáticos desta área. Para isso, serão explorados os aspectos de aplicação da tecnologia adaptativa ao processo de projeto e implementação de uma linguagem de programação com capacidade de incorporar novas funcionalidades de forma incremental, por solicitação do usuário, mediante a definição de extensões, ou através de alterações dinâmicas na lógica de suas instruções, em tempo de execução. Isso obviamente exige, da parte do ambiente em que a linguagem de programação é processada, a disponibilidade de recursos que permitam a tal programa adaptativo, sempre que necessário, automodificar-se de forma consistente.

1.2 Histórico e Revisão da Literatura

É fato que o uso adequado dos computadores para a solução de problemas complexos exige um profundo conhecimento teórico, bem como habilidade para uso extensivo e aplicação eficiente dos avanços da teoria. No entanto, o que se observa é que a solução destes problemas nem sempre faz uso adequado das descobertas teóricas e dos avanços tecnológicos obtidos (NETO, 2000). Além disso, entre as linguagens de programação disponíveis, não se conhece uma única que seja aceita como universal e que possua todos os recursos que proporcionem recursos para a obtenção fácil e direta de soluções práticas e eficientes para os diversos problemas propostos. Como resultado disso, são produzidos programas de baixa confiabilidade, de difícil compreensão, manutenção e fraco desempenho.

Neste contexto, muitas pesquisas vêm buscando desenvolver um modelo simples, intuitivo e eficiente, preferencialmente único, que possa abranger todas as classes de linguagens e que possa ser utilizado na forma mais simples que a linguagem desejada permita (TURBA, 1982; NOTKIN; GRISWOLD, 1988; CHRISTIANSEN, 1990; STEELE, 1999; VAN WYK, 2000; KOLBLY, 2002). Para isso, partindo das máquinas de estados finitos, largamente dominadas pelos estudantes de computação e engenharia, pode-se

incorporar, gradativamente, os recursos mínimos necessários à obtenção da potência suficiente para o tratamento de linguagens de classes sucessivamente mais complexas. Para tanto, existem os modelos de autômato finito (LEWIS; PAPADIMITRIOU, 1997), de autômato de pilha estruturado (NETO, 1987) e de autômato adaptativo (NETO, 1993; RAMOS; NETO; VEGA, 2009).

Os autômatos adaptativos são dispositivos formais capazes de coletar informação a partir dos seus dados de entrada, ao longo do seu funcionamento, e, com base em tal informação, modificar programadamente a sua própria estrutura, tornando-se dessa maneira um novo dispositivo, com comportamento diferente do original. Se utilizados como modelo formal para o projeto de linguagens de programação, as características dos autômatos adaptativos podem favorecer e simplificar a implementação de mecanismos para o tratamento adequado de aspectos de extensibilidade e de auto-modificação de código.

Os aspectos de extensibilidade estão limitados às componentes da linguagem de programação que podem ser tratadas em tempo de compilação, e correspondem, portanto, às alterações que o usuário pode efetuar sobre a linguagem, adequando-a aos seus particulares propósitos. Isso permite a personalização da linguagem, tornando a tarefa de codificação de programas mais confortável, e numa notação que seja aderente a aplicações para fins específicos (CHEATHAM JR., 1969; DUBY, 1971).

Os primeiros trabalhos científicos explorando o conceito de extensibilidade de linguagens de programação foram produzidos na década de 60. Um dos primeiros trabalhos nesta área, descreve o uso de instruções de macro em linguagens de programação de alto-nível (MCILROY, 1960). Outra abordagem clássica para tratar a extensibilidade de linguagens de programação foi apresentada por (BROOKER; MORRIS, 1962).

No final de década de 60 e início da década de 70, as pesquisas sobre extensibilidade atingiram o seu auge com a realização de dois simpósios sobre linguagens extensíveis nos anos de 1969 (CHRISTENSEN; SHAW, 1969) e 1971 (SCHUMAN, 1971). Uma introdução às linguagens extensíveis foi apresentada em (PERLIS, 1969). No mesmo ano, foram discutidos alguns aspectos e características de extensibilidade sobre as linguagens de programação Algol68 (MAILLOUX; PECK, 1969) e PPL (STANDISH, 1969b). Outros trabalhos apresentaram pontos de vista sobre conceito, motivação, alternativas e técnicas para o estudo e implementação de linguagens extensíveis (WEGNER, 1969; CHEATHAM JR., 1969; MCILROY, 1969; STANDISH, 1969a).

Em 1971, o evento contou com trabalhos relatando experimentos com linguagens extensíveis (STANDISH, 1971; CHANDLER, 1971), perspectivas da área (CHEATHAM JR., 1971; DUBY, 1971) e formas de implementação de extensões com macros

sintáticas (VAN GILS, 1971; WODON, 1971). Entretanto, apesar de alguns esforços isolados para tentar manter a comunidade científica motivada (HUBBARD JR., 1976), o movimento científico sobre o tema ficou reduzido a alguns trabalhos de análise dos resultados previamente obtidos (VAN GILS, 1972; STANDISH, 1975; METZNER, 1979). Um dos principais motivos utilizados para justificar a redução do interesse da comunidade científica pelo tema foi a necessidade de aumentar os níveis de abstração das linguagens de programação, visando omitir do usuário os detalhes de baixo nível de implementação de linguagens (STANDISH, 1975). Em outras palavras, a programação de sucessivas extensões de uma linguagem exigia um conhecimento prévio das linguagens dos níveis anteriores, antes da inclusão das novas construções e definições. Com isso, o movimento científico sobre as linguagens extensíveis perdeu espaço para a preocupação com a busca por um nível de abstração cada vez mais alto para a programação de computadores, sem a preocupação com a manutenção da capacidade de extensão das linguagens.

Entretanto, o surgimento de novas técnicas e modelos para a implementação de linguagens provocou um impacto positivo no desenvolvimento e aperfeiçoamento dos conceitos de programação. Com isso, surgem alternativas para o tratamento das características de dependência de contexto das linguagens extensíveis (STEELE, 1999; KOLBLY, 2002; BRABRAND; SCHWARTZBACH, 2002). Um formalismo muito promissor para a implementação de linguagens extensíveis é o autômato adaptativo (NETO, 1994). Neste contexto, o autômato adaptativo se apresenta como um modelo capaz de expressar e manipular, naturalmente, a dependência de contexto existente na descrição da linguagem de um mecanismo de extensão, que pode ser definido como um componente independente da linguagem base que, quando acoplado à mesma, torna-a extensível.

Observe-se, no entanto, que os aspectos de extensibilidade não são suficientes para tratar as necessidades daqueles usuários cujos programas necessitem efetuar alterações dinâmicas, durante a execução, na lógica de seus procedimentos. Para isso, tais programas necessitariam incorporar recursos de adaptatividade e, portanto, dispor de meios para, sem intervenção externa, definir e realizar alguma auto-modificação estrutural necessária.

Quando surgiram os primeiros computadores, o uso de programas com capacidade de se auto-modificar sem a intervenção externa era muito comum, devido à escassez de memória. Dessa forma, o uso e o aproveitamento da memória se tornavam mais eficientes, amenizando os efeitos provocados pelo espaço reduzido. No entanto, tais programas foram caindo em desuso devido às dificuldades de compreensão de seu código, de desenvolvimento da sua lógica, da verificação de consistência e de depuração e

manutenção de seu código. Isso coincidiu com o advento da Engenharia de Software em resposta à famosa Crise do Software, ocorrida na década de 70 (ANCKAERT; MADOU; DE BOSSCHERE, 2007).

Depois de um longo período de estagnação, o código auto-modificável reapareceu em algumas aplicações (TSCHUDIN; YAMAMOTO, 2005; GIFFIN; CHRISTODORESCU; KRUGER, 2005). Entre as principais aplicações em que se vem utilizando esse recurso, destacam-se: proteção de programas, compressão de código, engenharia reversa indesejada, otimização de código, sistemas de computação evolucionária, entre outros. Além disso, outros trabalhos vêm sendo desenvolvido na busca pela definição adequada de linguagens e ambientes que sejam capazes de tratar, de forma adequada, os aspectos de auto-modificação de código (PELEGRINI; NETO, 2008; FREITAS; NETO, 2007b; FREITAS, 2008). No entanto, ainda existe a necessidade de um modelo capaz de tratar de forma completa, conveniente e confortável os aspectos de extensibilidade sintática e de auto-modificação de código em tempo de execução.

1.3 Organização do Texto

Este documento está dividido em cinco capítulos. O presente capítulo introduz a proposta, destacando o contexto de pesquisa, a motivação para a realização da mesma e seus os objetivos. Além disso, foi traçado um perfil histórico, através de uma breve revisão da literatura, citando os trabalhos de pesquisa correlatos, cujos resultados contribuíram para o desenvolvimento deste documento.

O Capítulo 2 apresenta os princípios e conceitos relacionados com o desenvolvimento desta pesquisa, discutindo os fundamentos relacionados com a formalização, representação e concepção de linguagens. Além disso, este capítulo apresenta os autômatos adaptativos e realiza uma breve discussão sobre as suas propriedades e modo de operação, em especial sobre os aspectos relacionadas com a sua execução e seus aspectos de determinismo e não-determinismo.

O Capítulo 3 discute o processo de concepção de uma linguagem de programação imperativa, usando o autômato adaptativo como dispositivo formal para sua definição e descrição. Neste capítulo, são definidos as componentes léxica e sintática de uma linguagem base simplificada, denominada MINLA (da contração *MINimal LAnguage*). Também é apresentada a proposta de um conjunto de primitivas que modelam a representação dos comandos desta linguagem na forma de construtos adaptativos.

O Capítulo 4 faz uma revisão da literatura a respeito da extensibilidade em linguagens de programação e discute os aspectos de formalização de uma linguagem ex-

tensível usando o autômato adaptativo, denominada EXTLA (da contração *EXTensible LAnguage*). É apresentada a proposta de uma gramática que representa um mecanismo de extensão, capaz de definir novas construções sintáticas sobre a linguagem MINLA, através de construções sintáticas existentes ou previamente definidas. Exemplifica-se, conceitualmente, o ambiente de execução para dar suporte ao mecanismo de extensão proposto, usando o formalismo adaptativo como base para o tratamento dos aspectos sintáticos e semânticos envolvidos em uma linguagem extensível de programação.

O Capítulo 5 apresenta uma breve discussão sobre a implementação de ambientes de interpretação e execução de programas adaptativos. É apresentado um ambiente descrito na linguagem do AdapTools (PISTORI; NETO, 2003a; JESUS et al., 2007), definido na Seção 3.2, que implementa uma máquina de execução para códigos intermediários, gerados na forma de autômatos adaptativos. Este ambiente é capaz de tratar, de forma elementar, as ações adaptativas associadas a instruções de uma linguagem de programação, permitindo o uso de recursos para a implementação de programas de código auto-modificável.

Por fim, o Capítulo 6 apresenta algumas considerações sobre as principais contribuições deste trabalho, descrevendo alguns detalhes de implementação e apresentando uma breve descrição sobre algumas propostas de trabalhos futuros, que podem, em princípio, servir de base para a ampliação e a melhoria dos resultados obtidos nesta pesquisa.

2 Fundamentação Teórica

O autômato adaptativo se mostra como instrumento prático, eficiente, compacto e elegante quando empregado na solução de problemas complexos (NETO, 2000). Dessa forma, a melhoria no processo de implementação e uso efetivo deste dispositivo exige uma compreensão mais profunda das leis e dos processos da computação envolvidos. O projeto de um ambiente aderente a esta teoria pode facilitar, incentivar e estimular o desenvolvimento de novos trabalhos de pesquisa, explorando o autômato adaptativo como paradigma de programação e como modelo de computação. No entanto, para a concepção de um ambiente de execução totalmente baseado no autômato adaptativo, desde a sua formulação gramatical, incluindo os aspectos dependentes de contexto, até os aspectos semânticos de seu comportamento dinâmico se faz necessária a compreensão de alguns conceitos fundamentais, aqui relacionados.

Nas seções seguintes serão apresentadas as principais definições para a compreensão do funcionamento e dos conceitos fundamentais envolvidos no desenvolvimento deste trabalho de pesquisa.

2.1 Representação de Linguagens

Por definição, um compilador pode ser entendido como sendo um programa de computador capaz de aceitar como entrada um outro programa de computador, geralmente escrito em uma linguagem de alto nível (linguagem-fonte), com o objetivo de traduzi-lo para uma linguagem simbólica ou de máquina (linguagem-objeto de baixo nível).

Uma linguagem pode ser definida a partir de um conjunto de elementos (símbolos) e um conjunto de métodos (regras) para combinar estes elementos, usado e entendido por uma determinada comunidade. As linguagens formais são mais restritas e contemplam os mecanismos formais para representação e especificação de linguagens. Tais representações podem ser feitas com o auxílio de reconhecedores e geradores. Os reconhecedores correspondem aos dispositivos formais que servem para verificar se uma sentença pertence ou não a uma determinada linguagem. Os sistemas geradores são

dispositivos formais que permitem a geração sistemática de todas as sentenças de uma linguagem.

Considerando que a construção de ferramentas que manipulam linguagens devem trabalhar sobretudo, se preocupando com os problemas sintáticos e semânticos das linguagens, vale ressaltar que a sintaxe trata das propriedades livres da linguagem como, por exemplo, a verificação gramatical de programas, enquanto a semântica objetiva dar uma interpretação para a linguagem, como por exemplo, um significado ou valor para um determinado programa. Isto nos indica então que a sintaxe simplesmente manipula símbolos e não os seus significados. Portanto, para a etapa de processamento sintático de linguagens, podem ser utilizados alguns modelos computacionais mais simples.

Uma das formas de representação de linguagens é a utilização de gramáticas:

Definição 1 *Uma gramática G é um dispositivo de geração de sentenças, que pode ser definido como uma quádrupla ordenada* $G = (V, T, P, S)$*, onde:*

- V*: representa o vocabulário da gramática, ou seja, o conjunto de símbolos terminais e não terminais da gramática G.*
- T*: representa o conjunto de símbolos terminais de* V*, ou seja, os símbolos que são utilizados para construir as sentenças da gramática G. Os demais símbolos de* V *são denominados não-terminais e pode-se definir N seu conjunto, ou seja,* $N = V - T$*.*
- *P: é o conjunto das leis de formação da gramática G. Cada uma dessas regras é denominada produção e possui o formato* $\alpha \rightarrow \beta$*, onde* α *é composto de um símbolo não-terminal e* β *é composto de terminais e não-terminais, e pode ser vazio.*
- *S: é o símbolo inicial da gramática. S necessariamente é um símbolo não-terminal.*

Uma forma sentencial é sequência de terminais e/ou não-terminais produzida a partir do símbolo inicial dessa gramática pela aplicação de algum elemento do conjunto de regras de produção da gramática. O conjunto de regras de produção é utilizado para possíveis substituições de símbolos não-terminais por símbolos terminais. Dessa forma, a eliminação de símbolos não-terminais através de aplicações sucessivas de substituição definidas pelas produções da gramática, resultam em uma sentença da gramática. Mais formalmente, uma sentença F é uma forma sentencial cujos componentes são apenas terminais, ou seja, $F \in T^*$.

Define-se como uma derivação, a aplicação de uma ou mais substituições utilizando as regras de produção definida na linguagem. Para representar uma derivação da gramática *G*, pode-se utilizar a seguinte notação $\Rightarrow_G$ ou simplesmente $\Rightarrow$, quando não há dúvidas sobre qual gramática está sendo referenciada. Uma sequência de zero ou mais derivações também é denotada por $\Rightarrow^*$. Pode-se afirmar ainda que, ao conjunto de todas as possíveis sentenças geradas através de derivações a partir do símbolo inicial *S* de uma gramática *G*, denomina-se linguagem gerada por *G*. Podemos denotar a linguagem gerada por *G* como $L(G)$ e representá-la da seguinte forma:

$$L(G) = \{F \in T^* : S \Rightarrow_G^* F\}$$

De acordo com o formato das regras de produção as gramáticas, a Hierarquia de Chomsky, classifica as gramáticas em: irrestritas ou do tipo 0, sensíveis ao contexto ou do tipo 1, livres de contexto ou tipo 2 e lineares ou do tipo 3. Cada uma dessas classes é brevemente descrita a seguir:

- Gramáticas Irrestritas: nestas gramáticas não há imposição de nenhuma limitação. Dessa forma, todas as linguagens que podem ser definidas através de gramáticas são o conjunto e linguagens geradas por esse tipo de gramática. Essa classe de gramática é conhecida também como gramática do tipo 0. A forma das produções deste tipo de gramática é: $\alpha \rightarrow \beta, \alpha \in V^*NV^*$, $\beta \in V^*$.

- Gramáticas Sensíveis ao Contexto: nestas gramáticas não podem existir regras de produção que reduzam o comprimento da forma sentencial à qual a substituição é aplicada. Estas gramáticas são denominadas também gramáticas do tipo 1. As produções deste tipo de gramática possuem a forma: $\alpha \rightarrow \beta, \alpha \in V^*NV^*$, $\beta \in V^*$, com $|\alpha| \leq |\beta|$.

- Gramáticas Livres de Contexto: neste tipo de gramática há também a restrição de que o símbolo do lado esquerdo da produção deve ser necessariamente um símbolo não-terminal isolado. Estas gramáticas também são denominadas de gramáticas do tipo 2 e suas produções possuem a forma $A \rightarrow \beta$, $A \in N$, $\beta \in V^*$.

- Gramáticas Regulares: neste tipo de gramática, as restrições dependem se as gramáticas são lineares à esquerda ou lineares à direita. As produções das gramáticas lineares à direita especificam a substituição de um não-terminal por uma cadeia de terminais, terminada por no máximo um não-terminal. As produções das gramáticas lineares à esquerda especificam a substituição de um não-terminal por uma cadeia de terminais precedida por no máximo não-terminal. As

produções das gramáticas lineares são da seguinte forma: $A \rightarrow \alpha B$ ou $A \rightarrow \alpha$ (lineares à direita); $A \rightarrow B\alpha$ ou $A \rightarrow B$ (lineares à esquerda), com $A, B \in N, \alpha \in T^*$.

Essa classificação é importante porque nem sempre todas as aplicações necessitam de todas as generalidades que as gramáticas gerais apresentam. Mais especificamente, as linguagens de programação, podem ser representadas por gramáticas livres de contexto (Definição 1).

Para a representação de linguagens e, conseqüentemente, de gramáticas que as representam, podem-se utilizar diversas metalinguagens. Neste trabalho, será utilizada a notação de Wirth, definida na próxima seção.

2.1.1 Notação

A notação de *Wirth* é uma variação da Notação BNF (NETO, 1993) e, por isso, também é conhecida como notação BNF estendida.

As gramáticas descritas nesta notação obedecem ao seguinte formato:

X representa um símbolo não-terminal da gramática, ou seja, *X* é uma cadeia qualquer de caracteres que identifica uma classe de construções sintáticas da linguagem.

"y" denota um símbolo terminal da gramática, a cadeia de caracteres entre aspas pertence ao conjunto dos símbolos terminais (átomos) que compõem as sentenças da linguagem.

[*z*] o uso de colchetes indica que uma dada construção sintática *z* é opcional. Isto é equivalente a $z \mid \varepsilon$, onde *z* é um conjunto de opções de cadeias constituídas de terminais e/ou não-terminais.

(*z*) o uso de parênteses denota somente o agrupamento de construções sintáticas, sendo equivalente à forma *z*.

{ ***z*** } o uso de chaves indica um número de iterações arbitrário de *z*.

z*|*t indica que as construções sintáticas representadas pelas cadeias *z* e *t* são alternativas válidas no contexto em que aparecem e podem ser representadas por quaisquer das construções definidas anteriormente ou por concatenações de outras construções.

zt denota a concatenação entre as construções sintáticas *z* e *t*, nesta ordem. Por sua vez, tanto *z* como *t* podem ser quaisquer das construções definidas anteriormente.

= é o símbolo utilizado para separar o não-terminal (lado esquerdo), isolado, do conjunto de cadeias que representam uma ou mais opções sintáticas válidas de substituições para o não-terminal correspondente.

. é o símbolo que indica o final de uma regra desta notação.

A notação de Wirth descreve um dispositivo gerador (gramática), que permite a especificação de gramáticas do tipo 2 (Seção 2.1). Com extensões adequadas, gramáticas dos demais tipos também podem ser representadas, em uma notação estendida, obedecidas as respectivas convenções. Portanto, se tivermos como ponto de partida um arquivo de entrada com a especificação de uma gramática na notação de Wirth, os mecanismos do compilador poderão ser utilizados no sentido de gerar um programa de computador que seja capaz de atuar como reconhecedor para a linguagem descrita pelo arquivo de entrada.

Recentemente, alguns trabalhos de pesquisa tratam da aplicação do autômato adaptativo (NETO, 1993; NETO, 1994) na representação e implementação de linguagens de programação (FREITAS; NETO, 2005; FREITAS; NETO, 2006b; FREITAS; NETO, 2006a; FREITAS; NETO, 2006c; FREITAS; NETO, 2007b; FREITAS; NETO, 2007a; FREITAS, 2008; PELEGRINI; NETO, 2008). Este formalismo possui características mais sofisticadas, capaz de representar, em muitos casos, aspectos complexos de uma linguagem, de forma natural e expressiva. As características e potencialidades deste formalismo são apresentadas na próxima seção.

2.2 Autômatos Adaptativos

Os formalismos adaptativos são descritos como uma dupla composta por uma componente que define um dispositivo guiado por regras não-adaptativo (ou dispositivo subjacente) e uma componente conhecida como camada adaptativa que define, através das ações adaptativas, as possíveis transformações sobre o dispositivo subjacente (NETO, 2001).

Em cada fase de sua operação desses formalismos, eles assumem alguma configuração, que é definida como sendo o conjunto de todos os elementos que definem o estado atual do autômato. Dessa forma, o autômato opera sucessivamente, movendo-se de uma configuração para outra, em resposta a estímulos recebidos como entrada. Sem perda de generalidade, pode-se afirmar que tais dispositivos partem de alguma configuração inicial, seguindo um conjunto de regras pré-definido e, depois de ter processado todos os estímulos de entrada, o autômato atinge uma certa configuração final que pode ser classificada como configuração de aceitação ou de rejeição.

Nesse contexto, para definir o modo de operação do autômato adaptativo, será utilizada a seguinte terminologia:

- **Passo subjacente**: refere-se à mudança de configuração do dispositivo subjacente (não-adaptativo) correspondente.
- **Passo adaptativo**: refere-se à mudança de configuração do dispositivo adaptativo. Neste caso, a aplicação de uma regra acompanha a execução de ações adaptativas que, por sua vez, podem provocar alterações no comportamento subseqüente do dispositivo subjacente.

Um autômato adaptativo é um formalismo adaptativo cuja camada subjacente consiste de um autômato de pilha estruturado (NETO, 1987), definido na Seção 2.2.1, e a camada adaptativa, definida na Seção 2.2.2, contém as ações adaptativas (NETO, 1994; PISTORI, 2003), que correspondem a chamadas das respectivas funções adaptativas, responsáveis pelas mudanças estruturais do dispositivo subjacente em questão. Portanto, um autômato adaptativo pode ser definido como uma dupla $AA = (DS_0, CA)$, onde DS_0 corresponde à estrutura inicial do autômato de pilha estruturado subjacente e CA à camada adaptativa correspondente, que contém as funções adaptativas que implementam as possíveis mudanças estruturais sobre o dispositivo subjacente. As seções seguintes descrevem cada um destes componentes com mais detalhes.

2.2.1 Autômato de Pilha Estruturado

No estudo de linguagens formais, o autômato de pilha (LEWIS; PAPADIMITRIOU, 1997) possui uma pilha adicional e é utilizado para o tratamento das propriedades das linguagens livres de contexto e de suas implementações. Porém, o mapeamento de uma gramática livre de contexto para a forma de um reconhecedor baseado em tais autômatos não é direta.

Neste contexto, foi apresentada uma idéia levemente modificada desse formalismo, denominada autômato de pilha estruturado (APE). O APE pode ser visto como um conjunto de máquinas de estados finitos com transições internas e externas, denominadas submáquinas. As transições internas são responsáveis pelo consumo dos símbolos de entrada e pela mudança de estados dentro de uma submáquina específica. As transições externas fornecem mecanismos para a chamada e retorno de submáquinas. Portanto, no APE, a pilha adicional é utilizada para armazenar os estados de retorno, de forma similar ao que acontece com os endereços de retorno durante as operações de chamada e retorno de procedimentos em programas de computador (NETO, 1993).

Definição 2 *Um autômato de pilha estruturado (APE) é uma 9-upla*

$$M = (S, Q, \mu, \Sigma, Q_0, s_0, \delta, \Delta, F)$$

Onde:

S *é um conjunto finito e não vazio de identificadores das submáquinas (autômatos) de* M.

Q *é um conjunto finito e não vazio de estados de* M.

$\mu : Q \to S$ *é uma função que identifica a qual submáquina pertence cada um dos estados de* M. *Dessa forma, dado* $q \in Q$, $\mu(q) \in S$ *é a submáquina da qual o estado* q *faz parte.*

Σ *é o alfabeto de entrada do APE.*

$Q_0 \subseteq Q$ *é um conjunto de estados iniciais em que,* $\forall s \in S$, $|\mu'(s) \cap Q_0| = 1$, *ou seja, cada submáquina possui um, e apenas um, estado inicial.*

$s_0 \in S$ *é a submáquina inicial do APE e, portanto,* $\mu'(s_0) \cap Q_0$ *é o estado inicial do APE.*

δ *é o conjunto de* transições internas *de* M, *definido pela relação sobre* $\mu'(s) \times (\Sigma \cup \{\varepsilon\}) \times \mu'(s)$, *com* $s \in S$. *As transições elementares são representadas pela tripla* $(q, \sigma, q') \in \delta$, *com* $\mu(q) = \mu(q')$, *e funcionam do modo análogo às transições de um autômato de estados finitos, provocando a simples mudança de estado, sem alterar o conteúdo da pilha de estados de retorno.*

Δ *é o conjunto de* transições de chamada de submáquina *de* M, *definido pela relação sobre* $\mu'(s) \times S \times \mu'(s)$, *com* $s \in S$. *As transições de chamada de submáquina alteram o conteúdo da pilha de estados de retorno e são representadas pela tripla* $(q, \theta, q') \in \Delta$, *com* $\mu(q) = \mu(q')$. *Nesta situação,* $\theta \in S$ *indica a submáquina de destino e* q' *o "estado de retorno da chamada" que é empilhado. O controle é transferido para o estado inicial da submáquina* θ, *identificado pelo elemento unitário do conjunto* $\mu'(\theta) \cap Q_0$[1].

$F \subseteq Q$ *é o conjunto de estados finais do autômato. Quando um estado final é atingido, executa-se uma transição de retorno de submáquina para o estado armazenado no topo da pilha. Portanto, quando executada, esta operação provoca a*

[1] Observe que a complexidade da operação de identificação do estado inicial da máquina de destino, para o qual será transferido o controle, pode ser reduzida com o acréscimo de uma componente que define uma função $\psi : Q_0 \to S$, que funciona de forma semelhante à função μ, proposta em (PISTORI, 2003).

alteração no conteúdo da pilha de estados de retorno. O controle é transferido para a submáquina $\mu(t)$, onde $t \in Q$ corresponde ao estado de destino, que é desempilhado. Quando o estado final atingido pertence à submáquina principal e a pilha está vazia, isso corresponde ao reconhecimento da cadeia de entrada.

Observe, ainda, que a função μ determina, indiretamente, o conjunto de estados de cada uma das submáquinas de M. Para isso, pode-se utilizar o conjunto potência de Q (2^Q) como contradomínio, obtendo-se uma função inversa para μ, $\mu' : S \rightarrow 2^Q$, que mapeia cada elemento de S ao seu conjunto de estados. Portanto, dado um $s \in S$, $\mu'(s) \subseteq Q$ é o conjunto de estados de s.

Tendo formalizado a estrutura básica de um APE, pode-se utilizá-lo como um mecanismo subjacente de um dispositivo adaptativo, adotando-se a definição mais geral de um dispositivo guiado por regras não-adaptativo (NETO, 2001):

Definição 3 *Um APE M pode ser definido como um dispositivo guiado por regras pela 6-upla*

$$M = (C, R, \Sigma, c_0, A, \Pi)$$

onde:

C *é o conjunto de todas as possíveis configurações de M, definido por uma relação sobre* $Q \times \Sigma^* \times F^*$. *Portanto, uma configuração* $c_i \in C$ *de um APE é uma tripla* (q, x, z), *onde:*

- $q \in Q$ *corresponde ao estado corrente;*
- $x \in \Sigma^*$ *é a parte da cadeia de entrada a ser consumida, e;*
- $z \in F^*$ *representa o conteúdo da pilha de estados de retorno de submáquina.*

Σ *é o alfabeto de entrada do APE.*

c_0 *é a configuração inicial, definida pela tripla* $c_0 = (Q \cap Q_0, w, \varepsilon)$, *onde* $w \in \Sigma^*$ *corresponde à cadeia de entrada.*

A *é o subconjunto de todas as configurações de aceitação de M:*

$$A = \{c_i \in C | c_i = (q, \varepsilon, \varepsilon), q \in F\}$$

Observe que, nesta situação, um estado final é atingido, a cadeia de entrada foi totalmente processada e a pilha de estados de retorno de submáquina está vazia.

Π *é um conjunto finito de todos os possíveis símbolos de saída que podem ser gerados como efeitos colaterais da execução das regras de M. No caso do APE,* $\Pi = \{\}$.

R *é uma relação sobre* $C \times (\Sigma \cup \{\varepsilon\}) \times C \times (\Pi \cup \{\varepsilon\})$ *que corresponde ao conjunto de regras que define M. Uma regra* $r \in R$ *é representada pela 4-upla* (c_i, σ, c_j, π)*, indicando que, em resposta a algum* $\sigma \in (\Sigma \cup \{\varepsilon\})$ *da cadeia de entrada, a regra r altera a configuração corrente de* c_i *para* c_j*, em M, consumindo* σ *e produzindo* $\pi \in \Pi$ *como saída.*

Observe que, cada elemento de R indica a sequência de *configurações* que representam o aspecto do APE em momentos sucessivos de sua operação para uma dada cadeia de entrada $w \in \Sigma^*$. Portanto, um *passo* do APE M corresponde à aplicação de uma regra $r \in R$ e é denotado por $c_i \vdash_M c_j$. Esta relação binária se aplica a um par de configurações $c_i, c_j \in C$ se e somente se a regra $r = (c_i, \sigma, c_j, \pi) \in R$ e o símbolo de entrada seja compatível com σ.

Definição 4 *Sejam* $\sigma \in (\Sigma \cup \{\varepsilon\})$, $\theta \in S$ *e* $c_i = (q, x, z)$ *e* $c_j = (q', x', z')$ *duas configurações consecutivas de um APE M. Um* passo *de M pode ocorrer se e somente se, uma das três seguintes situações se aplica:*

1. $(q, \sigma, q') \in \delta$, $z' = z$, $x = \sigma x'$ *e* $\mu(q) = \mu(q')$;
2. $(q, \theta, t) \in \Delta$, $z' = tz$, $x = x'$ *e* $q' \in (Q_0 \cap \mu'(\theta))$;
3. $q \in F$, $z = q'z'$ *e* $x = x'$.

Uma sequência de passos é denotada por $c_i \vdash_M^* c_j$. Uma cadeia $w \in \Sigma^*$ é *aceita* pelo APE M se e somente se existir uma configuração $c_k = (q, \varepsilon, \varepsilon) \in C$, tal que $c_0 \vdash_M^* c_k$ e $q \in F$.

2.2.2 Camada Adaptativa

Na seção anterior, foi definido o autômato de pilha estruturado, que corresponde ao dispositivo subjacente utilizado pelos autômatos adaptativos, que também incorporam uma camada adaptativa como componente genérica responsável pelas mudanças na topologia do dispositivo subjacente.

Definição 5 *A camada adaptativa é definida por uma 2-upla* $CA = (\mathscr{R}, \mathscr{A})$*, onde:*

$\mathscr{R}$ *é um conjunto de ações adaptativas (*$\varepsilon \in \mathscr{R}$ *indica uma ação nula).*

$\mathscr{A} : R \rightarrow \mathscr{R}^2$ *é uma função que mapeia cada possível regra do dispositivo subjacente em um par ordenado de ações adaptativas. As duas ações adaptativas deste par ordenado devem ser acionadas, respectivamente, antes e depois da execução da regra à qual elas estão associadas. Por isto, elas são também denominadas, respectivamente,* ação anterior *e* ação posterior.

Portanto, a camada adaptativa é composta por ações adaptativas que corresponde a chamadas paramétricas (ou seja, isntanciando argumentos) de uma função adaptativa. As funções adaptativas são definidas na próxima seção.

2.2.2.1 Funções Adaptativas

Uma função adaptativa é representada por um par ordenado (F,P) que a identifica, onde F é o nome de uma função adaptativa que implementa a ação correspondente, e P é a lista $(p_0, p_1, ..., p_k)$ de argumentos a serem passados para F. Uma função adaptativa é definida da seguinte forma (NETO, 1994):

Definição 6 *Uma função adaptativa é uma 6-upla* $FA = (F,P,\Omega,\Gamma,S,E)$ *na qual:*

- F *é o nome da função adaptativa.*
- P *é a lista* $(r_0, r_1, ..., r_m)$ *de parâmetros formais.*
- Ω *é a lista* $(\omega_0, \omega_1, ..., \omega_n)$ *de identificadores de variáveis ou geradores.*
- Γ *é uma lista de ações elementares de consulta, remoção ou inserção, denotadas respectivamente pelos prefixos* $?$, $-$ *e* $+$ *e cujo comportamento é definido da seguinte forma:*
  - *Ações elementares de consulta possibilitam a busca de padrões definidos de acordo com o formato das regras do dispositivo subjacente.*
  - *Ações elementares de remoção determinam as regras a serem removidas do conjunto de regras corrente do dispositivo subjacente.*
  - *Ações elementares de inserção determinam as regras a serem inseridas no conjunto de regras corrente do dispositivo subjacente.*
- S *é uma* ação adaptativa inicial *a ser executada antes de qualquer ação elementar de* F.
- E *é uma* ação adaptativa final *a ser executada depois da execução de todas as ações elementares de* F.

O funcionamento das funções adaptativas é análogo ao das funções paramétricas em um programa de computador. Os parâmetros formais são associados ao argumento correspondente à sua posição, no momento em que a função adaptativa é chamada.

O corpo de uma função adaptativa consiste de uma lista de *ações adaptativas elementares* (Γ) e das ações adaptativas inicial (S) e final (E), que correspondem a duas ações adaptativas (opcionais) que são executadas, respectivamente, antes e depois da execução da lista de ações elementares especificada. A lista das ações adaptativas elementares (ou, simplesmente, ações elementares) é organizada em seqüências de ações elementares *de consulta*, de ações elementares de *remoção* e de ações elementares de *inserção*. Os prefixos ?, $-$ e $+$ são utilizados para denotarem as ações elementares de consulta, remoção e inserção, respectivamente. Portanto, as ações elementares possuem o seguinte formato:

$$(?|-|+)\,[\,\textit{padrão}\,]$$

onde *padrão* corresponde à representação de uma regra do dispositivo adaptativo.

Uma regra padrão relaciona duas configurações consecutivas de um APE e é denotada por $\vdash$. A Seção 2.3 define cada um dos elementos desse padrão e discute outros aspectos e propriedades do modo de operação de um dispositivo adaptativo.

A lista de identificadores de variáveis ou geradores (Ω) consiste de uma lista de nomes que identificam variáveis ou geradores dentro de uma função adaptativa. As variáveis e geradores podem ocupar as posições correspondentes a elementos de uma regra do dispositivo adaptativo.

As variáveis têm os seus valores atribuídos durante a execução das ações elementares de consulta ou de remoção (consulta implícita). A atribuição de valor à variável se dá através de um mecanismo de busca de padrões que é responsável por atribuir os valores, tendo como base o conjunto de regras do dispositivo adaptativo. Quando o mecanismo de busca de padrões não encontra qualquer regra do dispositivo adaptativo que seja compatível com o formato definido na ação elementar de consulta, as variáveis são marcadas como indefinidas e, todas as ações elementares de remoção e inserção, que delas se utilizam, são desconsideradas, ou seja, não são executadas. Um gerador opera como uma espécie de constante, cujo valor é determinado no início da execução da ação adaptativa, e se mantém inalterado enquanto durar sua execução. O valor de um gerador é único e diferente de qualquer símbolo em uso no dispositivo adaptativo. Uma vez atribuídos, os valores de cada variável e de cada gerador não se alteram durante a execução da função adaptativa. Os prefixos ? e $*$ são utilizados para

denotar os identificadores do tipo variável e gerador, respectivamente.

Considerando que o *padrão* possui uma estrutura dependente do formato das regras do dispositivo adaptativo, uma função adaptativa possui o seguinte formato:

01 $F(r_1,...,r_m)$ {
02 $?v_1,?v_2,...,?v_n,*g_1,*g_2,...,*g_p$:
03 $A(p_1,...,p_k)$
04 $?[padrão]$
05 $?[padrão]$
06 $\cdots$
07 $-[padrão]$
08 $-[padrão]$
09 $\cdots$
10 $+[padrão]$
11 $+[padrão]$
12 $\cdots$
13 $B(p_1,...,p_j)$
14 }

A linha 01 representa o cabeçalho da função adaptativa, contendo do nome e a lista de parâmetros formais. A linha 02 corresponde à lista de identificadores de variáveis e geradores, denotados pelos prefixos ? e $*$, respectivamente. As linhas 03 e 13 correspondem, respectivamente, às chamadas das ações inicial e final. As linhas de 04 a 12 representam a lista das ações elementares de consulta (?), remoção ($-$) e inserção ($+$).

2.3 Propriedades do Modo de Operação do Autômato Adaptativo

Nas subseções seguintes, discutem-se aspetos adicionais sobre o modo de operação do autômato adaptativo, estendendo o conceito de *passo* para esse formalismo e fazendo considerações sobre os aspectos de determinismo em formalismos adaptativos.

2.3.1 Considerações sobre Passo de Execução

Na Seção 2.2 foi definido o conceito de passo em um dispositivo adaptativo qualquer. Portanto, na operação do autômato adaptativo, o conceito de *passo* também deve ser estendido para que se possam incorporar as ocasionais mudanças na topologia do APE

subjacente. Com isso, pode-se distinguir duas situações possíveis na operação dos autômatos adaptativos:

- **Passo subjacente**: corresponde à execução de um passo na operação análogo ao passo de operação do APE subjacente. Nesta situação, não ocorre nenhuma alteração no conjunto de regras que define o APE.
- **Passo adaptativo**: corresponde à execução de um passo de operação associado a alguma ação adaptativa não nula. Nesta situação, as componentes que definem o conjunto de regras do autômato adaptativo e, por decorrência também do APE subjacente, são modificadas pelos efeitos da ação adaptativa correspondente e, conseqüentemente, a sua topologia deve refletir essas alterações.

Os termos *passo subjacente* e *passo adaptativo* serão utilizados para identificar, respectivamente, uma mudança de configuração sem e com a chamada de influência da camada adaptativa. Para a especificação formal do modo de operação do autômato adaptativo faz-se necessária a definição de um formato padrão para o seu conjunto de regras. Tais regras relacionam duas configurações sucessivas do dispositivo, partindo sempre do estado inicial q_0. Essa relação é denotada por $\vdash$ e é definida por:

$$(q,\gamma),\mathscr{B} \vdash (q',\beta),\mathscr{A}$$

Os componentes (q,γ) e (q',β) do padrão correspondem a parte de operação do APE subjacente, com $q,q' \in Q$, $\gamma \in (\Sigma^* \cup \{\varepsilon\}) \cup S$ e $\beta \in \Sigma^* \cup \emptyset$ (vide Definição 2). É importante explicar que o símbolo $\gamma \in (\Sigma^* \cup \{\varepsilon\})$ quando a regra trata o consumo de algum símbolo da cadeia de entrada e que $\gamma \in S$ quando a regra corresponde a uma chamada de submáquina. Os componentes $\mathscr{B}$ e $\mathscr{A}$ correspondem a chamadas de funções adaptativas e estão associadas com a camada adaptativa do APE subjacente.

Dessa forma, quando usamos um APE como dispositivo subjacente de um autômato adaptativo, a operação de *passo* em um autômato adaptativo pode ser definida da seguinte forma:

Definição 7 *Seja um APE M, o dispositivo subjacente de um autômato adaptativo, cuja camada adaptativa é definida pela 2-upla* $CA = (\mathscr{R},\mathscr{A})$. *Sejam* $B,A \in \mathscr{R}$ *duas ações adaptativas,* $r \in R$ *uma regra do APE subjacente e* $r_{\mathscr{A}} \in \mathscr{A}$ *uma regra do autômato adaptativo correspondente, onde* $r_{\mathscr{A}}$ *associa a regra subjacente r com as ações adaptativas* (B,A). *Considere ainda que* M_i *e* M_{i+1} *são duas instâncias consecutivas do APE subjacente.*

1. *Um* passo subjacente *de M ocorre, se e somente se a regra $r_{\mathscr{A}}$ é aplicada com $B = A = \varepsilon$. Neste caso, $M_i = M_{i+1}$.*

2. *Um* passo adaptativo *ocorre quando $B \neq \varepsilon$ ou $A \neq \varepsilon$. Neste caso, uma das duas situações podem acontecer:*

   - *O autômato adaptativo modifica a sua topologia e os efeitos permanecem na máquina resultante. Com isso, $M_i \neq M_{i+1}$.*
   - *O autômato adaptativo modifica a sua topologia apenas para a execução da regra $r_{\mathscr{A}}$ e depois retorna ao estado original. Neste caso, $M_i = M_{i+1}$ e pode-se dizer que as ações adaptativa B e A se anulam.*

No caso dos formalismos adaptativos, além de estender o conceito de passo de execução, existe uma séria dificuldade relacionada com os seus aspectos de determinismo: sua natureza dinâmica não nos permite determinar a priori, com facilidade, se o dispositivo é e permanece (ou não) determinístico. Nesse contexto, optou-se pelo estudo de tais aspectos sobre uma classe de dispositivos adaptativos que é particularmente importante devido à sua relativa simplicidade, a dos autômatos de estados finitos adaptativos (PISTORI, 2003). Os autômatos de estados finitos, além de apresentarem uma formalização mais simples do que os APEs, são mais conhecidos e difundidos na área de computação, simplificando a apresentação e assimilação dos aspectos explorados neste estudo inicial. Dessa forma, tal formalismo foi utilizado para ilustrar, definir e apresentar alguns aspectos relacionados com o determinismo em dispositivos adaptativos.

Partindo do estudo deste formalismo específico, além da definição de uma subclasse de dispositivos adaptativos determinísticos, foi relacionado um conjunto de restrições que devem ser obedecidas para a obtenção de autômatos de estados finitos adaptativos determinísticos e que podem, como uma sugestão interessante de trabalhos futuros, ser estendidas para tratar os aspectos de determinismo sobre formalismos adaptativos quaisquer.

2.3.2 Determinismo em Dispositivos Adaptativos

O determinismo é uma das características que devem ser observadas durante o projeto de dispositivos formais, quando se busca bom desempenho. No caso dos formalismos adaptativos, existe uma séria dificuldade relacionada com a manutenção do determinismo pois, devido ao seu comportamento auto-modificável, é muito difícil garantir que o determinismo será mantido durante toda a operação do dispositivo. Nesta

seção, apresenta-se formalmente o conceito de determinismo para uma classe de dispositivos adaptativos que é particularmente importante devido à sua relativa simplicidade, a dos autômatos de estados finitos adaptativos. Partindo do estudo deste formalismo específico, além da definição de uma subclasse de dispositivos adaptativos determinísticos, foi relacionado um conjunto de restrições que devem ser obedecidas para a obtenção de dispositivos adaptativos determinísticos.

Alguns trabalhos discutem a aplicação dos formalismos adaptativos em problemas complexos da teoria da computação (NETO, 2000), construção de compiladores (NETO; IWAI, 1998), inteligência artificial (PISTORI; NETO, 2003b; PISTORI; MARTINS; CASTRO JR., 2005), processamento de linguagem natural (NETO; DE MORAES, 2003), robótica (SOUSA; HIRAKAWA, 2005), entre outros. Além disso, os autômatos adaptativos têm sofrido mudanças, com o objetivo de simplificar sua formulação original e torná-lo um dispositivo mais prático e eficiente (PISTORI, 2003).

O determinismo, por exemplo, possui relação direta com o bom desempenho das aplicações que fazem uso dos dispositivos formais projetados. Em (HROMKOVIC et al., 2000) foi realizado um estudo comparativo sobre a complexidade de autômatos de estados finitos determinísticos e não-determinísticos. Em (STEARNS, 2003) foi destacada a importância da compreensão do relacionamento existente entre os tempos de computação determinística e não-determinística, uma vez que os algoritmos desenvolvidos para muitos problemas práticos, apesar de serem inerentemente não-determinísticos, são executados em máquinas que atuam de forma determinística. Além disso, a maior parte das soluções encontradas para o processamento de linguagens naturais, se apresentam com características de não-determinismo ou de ambigüidade, as quais são resolvidas através da simulação seqüencial do paralelismo inerente aos processos em questão, em tempo exponencial (NETO, 2000).

Considerando a definição mais geral para a classe de dispositivos guiados por regras (NETO, 2001), tem-se:

Definição 8 *Um dispositivo adaptativo é dito* determinístico *se e somente se a cada passo adaptativo de sua operação, o mecanismo subjacente permanece determinístico, qualquer que seja o passo realizado.*

Apesar de parecer simples e direta, no caso de formalismos como os autômatos de estados finitos adaptativos, existem situações nas quais, dependendo da cadeia de entrada apresentada ao autômato, não-determinismos nunca se manifestem durante a operação do dispositivo, embora estejam presentes (Figura 2.1(b)).

Um exemplo que ilustra este aspecto do determinismo em tais dispositivos é o da

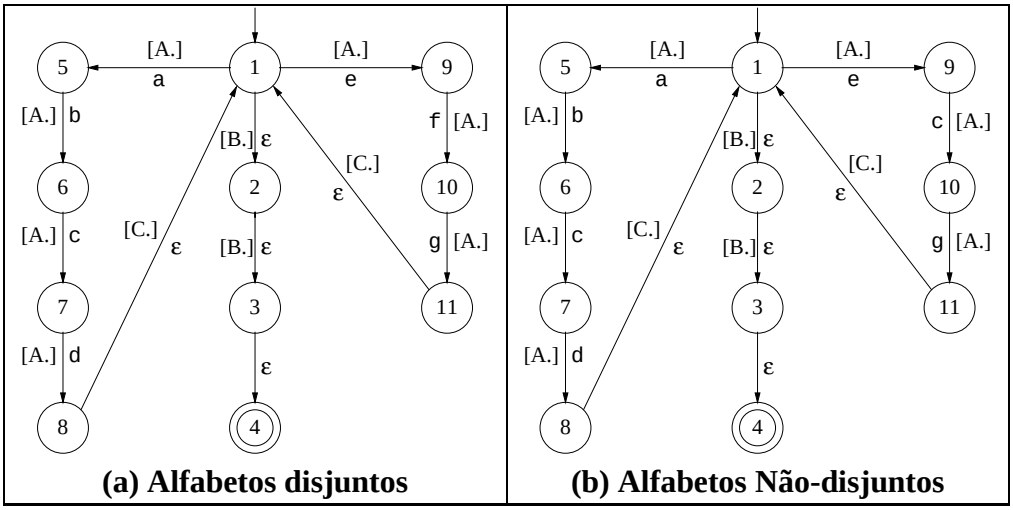


Figura 2.1: Dois autômatos de estados finitos adaptativos que aceitam o *merging* de duas cadeias curtas. Em (a), o autômato adaptativo sempre opera deterministicamente, seja qual for a cadeia de entrada. Em (b), algum não-determinismo pode, eventualmente, se manifestar. Por exemplo, para a cadeia `aecbgcd`, a operação do autômato será determinística, o que não ocorre com a cadeia `aebcgcd`.

solução compacta e eficiente, baseada em autômatos adaptativos, proposta em (NETO, 2000), para reconhecer uma linguagem formada a partir de um número finito de cadeias sobre alfabetos disjuntos, pela intercalação arbitrária (*merge*) dos símbolos das cadeias participantes do "merge".

As Figuras 2.1(a) e 2.1(b) mostram, respectivamente, um autômato de estados finitos adaptativo que reconhece o *merging* de duas seqüências curtas com alfabetos disjuntos (NETO, 2000), e um autômato de estados finitos adaptativo que faz o mesmo, mas usando seqüências com alfabetos não-disjuntos. Ambos operam da seguinte forma: todos os símbolos da cadeia de entrada são consumidos em transições que partem do estado inicial. A cada transição que consome um símbolo de uma das seqüências, está associada a ação adaptativa $A()$ que, quando executada, exclui a transição associada e conecta o estado inicial à próxima transição correspondente àquela seqüência. Quando todos os símbolos de uma das seqüências tiverem sido consumidos, a ação adaptativa $C()$ é executada, removendo do autômato uma transição contendo uma chamada da função adaptativa $B()$. Portanto, se os símbolos de todas as seqüências forem consumidos, todas as transições associadas à ação adaptativa $B()$ terão sido removidas e, dessa forma, o autômato alcançará o estado final, reconhecendo assim a cadeia de entrada. No entanto, se a ação adaptativa $B()$ chegar a ser executada durante a operação, a transição que levaria o autômato ao estado final será removida e, portanto, a cadeia de entrada não tem como ser aceitado, logo será rejeitada. Vale ressaltar que, a ação $B()$ só será executada se não houver símbolo algum (de nenhuma das cadeias envolvidas na operação de *merging*) que possa ser consumido a partir do estado inicial do

autômato nessa ocasião.

Na Figura 2.1(a), a operação do autômato adaptativo exemplificado é totalmente determinística, já que os alfabetos são disjuntos, impossibilitando que surja algum não-determinismo devido à operação de "merge"implementado por esse método. Na Figura 2.1(b), o não-determinismo do autômato só se manifesta quando, para casos particulares de cadeias de entrada, houver um conflito entre dois símbolos iguais, provenientes de cadeias diferentes. Nestes casos, o autômato não consegue determinar, diretamente, se o símbolo conflitante proveio de uma das cadeias ou da outra.

O exemplo anterior ilustra uma situação na qual, apesar de estar presente, o não-determinismo pode não se manifestar, para uma dada cadeia de entrada. Dessa forma, o autômato pode operar de forma determinística, mesmo sendo potencialmente não-determinístico.

Seja M um autômato finito adaptativo determinístico que executa uma ação adaptativa durante a sua operação. Neste caso, para garantir que o determinismo seja mantido, algumas restrições devem ser observadas durante a definição e a operação das ações elementares de consulta, remoção e inserção.

Definição 9 *Um autômato de estados finitos adaptativo é dito* **fortemente determinístico** *quando seu autômato de estados finitos subjacente permanece determinístico durante toda a sua operação, sejam quais forem as ações adaptativas executadas para qualquer cadeia de entrada* $w \in \Sigma^*$.

Observe-se que as ações adaptativas elementares de consulta e remoção não introduzem não-determinismos de forma direta. No entanto, os efeitos das consultas realizadas podem, ocasionalmente, introduzir não-determinismos através de ações de remoção e inserção que façam uso de variáveis utilizadas nas consultas. As seções seguintes estudam, separadamente, as situações em que as ações elementares, presentes nos autômatos de estados finitos adaptativos, podem influenciar no determinismo do autômato.

2.3.2.1 Ações Elementares de Consulta

Em um autômato de estados finitos adaptativo, seja AF uma função adaptativa descrita conforme a definição apresentada em (PISTORI, 2003). Uma ação elementar de consulta $\varphi \in \Gamma$ tem a seguinte forma geral:

$$?[(q, s\alpha), \mathscr{B} \vdash (q', \alpha)]$$

Tais ações elementares executam consultas, buscando no atual conjunto de regras, um padrão similar ao que foi especificado na forma geral. A instância encontrada desse padrão é utilizada para determinar os valores das variáveis referenciadas correspondentes. Dessa forma, não-determinismos podem ser introduzidos quando estas variáveis, devidamente preenchidas, são utilizadas em ações elementares de remoção e inserção. Nas ações elementares Em tais casos, para manter o determinismo, as seguintes condições devem ser observadas:

1. Se $q \in \Omega$. Quando o estado de origem não é conhecido, uma forma de evitar a introdução de potenciais não-determinismos, quando a variável q associada é usada em ações elementares de inserção, é garantir que q não seja preenchida com múltiplos valores. Embora seja muito restritiva, uma forma simples e direta é fazer com que s e q' sejam previamente conhecidos. Caso isto não seja possível, como técnica para evitar o não-determinismo, deve-se eliminar o uso destas variáveis em ações elementares de inserção na função adaptativa correspondente.

2. Se $s \in \Omega$. Quando s não é conhecido, a única restrição sobre as ações elementares de inserção é que o valor de q' seja único. Com isso, mesmo que a variável s seja preenchida com múltiplos valores, garante-se que não haverá risco da introdução de potenciais não-determinismos, pois todas as transições de saída que serão associadas (inseridas) ao estado q' consumem símbolos distintos.

3. Se $q' \in \Omega$. Neste caso, a presença de um não-determinismo é imediatamente constatada quando $\exists[q,s]$, tal que à variável q' são atribuídos múltiplos valores. Caso contrário, q' terá sempre um valor único.

Se todas as restrições acima forem satisfeitas, pode-se garantir que os efeitos da execução das ações elementares de consulta nunca introduzirão não-determinismos. Na seção seguinte, são analisadas as situações ocasionadas exclusivamente pela execução das ações elementares de inserção.

2.3.2.2 Ações Elementares de Inserção

Ações elementares de inserção possuem a seguinte forma geral:

$$+[(q,s\alpha),\mathscr{B} \vdash (q',\alpha)] \tag{2.1}$$

no momento de sua aplicação, para cada transição na forma

$$(q'',s''\alpha),\mathscr{B}' \vdash (q''',\alpha) \tag{2.2}$$

as seguintes situações devem ser verificadas:

1. $q \notin Q$. Neste caso, q está representado por um gerador, pois o respectivo estado não pertence ao conjunto de estados do autômato e será gerado. Com isso, não há conflitos e o determinismo será mantido.

2. $q \in Q$. Neste caso, poderá haver conflitos entre as regras de transição correntes e a regra incluída, resultando na introdução de não-determinismos nas seguintes situações: (1) se, como efeito de uma ação de consulta, o valor de q' não é único; (2) se q' possui valor único mas, no momento da aplicação da ação elementar de inserção, para algum q'', o autômato possui regras na forma (ocasionalmente, $\mathscr{B}' = \varepsilon$): $[(q, s\alpha), \mathscr{B}' \vdash (q'', \alpha)]$

3. $q \in Q$ e $s \neq s''$. Se $\nexists[q, s, q'']\forall q''$, então o autômato transita com um símbolo s que não é consumido pelas regras existentes. Portanto, o determinismo será mantido, pois estão sendo incluídas transições que especificam valores distintos daqueles utilizados em transições existentes.

4. $q \in Q$ e $s = s''$. Nesta situação, pode haver a inclusão de não-determinismo caso o autômato possua transições que utilizem o mesmo símbolo s que é consumido por alguma regra já existente, uma vez que na configuração corrente do autômato, uma das transições existentes satisfaz o lado esquerdo daquela que está sendo inserida. Nesse caso, as condições devem ser satisfeitas:

   (a) $\mathscr{B} = \mathscr{B}'$. Para uma dada configuração, se as ações adaptativas de duas transições igualmente aplicáveis não forem idênticas, então haverá a introdução de um não-determinismo, caso o autômato venha a conter ambas as transições. Isto se deve ao perigo potencial de que as duas ações adaptativas gerem efeitos diferentes.

   (b) $q' = q'''$. Nessa situação, se $q' \neq q'''$, estaremos inserindo uma transição que consome o mesmo símbolo e alcança um estado distinto do especificado em transição já existente no autômato. Isso caracteriza um não-determinismo, exceto quando ambas as transições (a corrente e a que está sendo inserida) forem iguais (portanto, não-determinismos não serão incluídos).

Exceto para os casos (1) e (3), que inserem novas regras com segurança, se cumpridas as demais condições, além da impossibilidade de introdução de não-determinismos por parte das ações adaptativas elementares de inclusão, a regra a ser incluída já estará fazendo parte do conjunto de transições do autômato e, portanto, não afetará o seu determinismo.

A importância dos aspectos práticos do determinismo tem recebido atenção especial, visto que o desempenho está diretamente relacionado com tais aspectos (ALLAUZEN; MOHRI, 2003; ALLAUZEN; MOHRI, 2002; STEARNS, 2003; HROMKOVIC et al., 2000). Este capítulo registra e ilustra os recentes avanços obtidos nesta área, visando a definição de uma metodologia de projeto para usuários dos dispositivos adaptativos. A inserção dinâmica de não-determinismos através do mecanismo adaptativo do dispositivo foi discutida e algumas regras para o projeto de funções adaptativas foram propostas, visando garantir a permanente execução determinística do autômato de estados finitos adaptativo. Como sugestão de trabalhos futuros, os resultados obtidos podem ser estendidos e generalizados para a superclasse dos dispositivos adaptativos.

Além disso, os resultados parciais do estudo descrito nesta seção pode motivar e facilitar estudos futuros sobre a análise de complexidade e sobre os aspectos de determinismo em outras instâncias de dispositivos adaptativos e podem inspirar o desenvolvimento e a implementação de algoritmos, técnicas e ferramentas de software para detecção de não-determinismo e manutenção de determinismo em dispositivos adaptativos.

2.4 Considerações Sobre a Formalização de Linguagens

A especificação completa de uma linguagem de programação exige a especificação de cada fase de processamento de um programa escrito nesta respectiva linguagem. Estas fases são conhecidas por análise léxica, análise sintática e análise semântica. Apesar de já existir um consenso para a especificação formal das duas primeiras fases, o tema ainda é controverso quando se discute a fase de análise semântica (NETO, 1987; NETO, 1993; NETO; PARIENTE; LEONARDI, 1999).

A componente léxica define o conjunto de símbolos válidos para uma linguagem de programação e pode ser facilmente descrita através de gramáticas regulares. A componente sintática determina as seqüências válidas destes símbolos e pode ser descrita através de gramáticas livres de contexto. Por fim, a componente semântica descreve o significado e, dessa forma, confronta-se com freqüência à sintaxe, caso em que a primeira se ocupa do que algo significa, enquanto a segunda se concentra sobre as estruturas ou padrões formais do modo como esse algo é expresso.

Neste contexto, percebe-se que as componentes léxica e sintática são as de formalização mais simples e mais uniforme, enquanto a componente semântica é, usualmente, definida de modo informal, como uma implementação na mesma linguagem que está sendo especificada (PFLÚCKER; NETO, 2007). Conceitualmente, ainda não há um con-

senso entre os autores sobre a separação entre os aspectos sintáticos e semânticos de uma linguagem. Entretanto, neste trabalho, adota-se a definição de (PAGAN, 1981):

- **Semântica Estática:** todos os aspectos de uma linguagem de programação que são manipulados em tempo de compilação e não são livres de contexto;
- **Semântica Dinâmica:** todos os aspectos de uma linguagem de programação que são manipulados em tempo de execução.

Portanto, toda formalização semântica deve, de alguma forma, associar significado aos programas e construções de uma linguagem objeto. Entretanto, a ausência de um consenso para a formalização da semântica motivou o desenvolvimento de uma variedade de modelos distintos para a definição desta etapa, que podem ser classificados em três grupos: operacional, denotacional e axiomática. As características de cada um desses grupos são as seguintes (NIELSON; NIELSON, 1999):

- **Semântica operacional**: O significado de uma construção da linguagem é especificado pela computação que ela induz quando executada em alguma máquina. Isto significa que o que interessa mais é a forma como o efeito da computação é produzido;
- **Semântica denotacional**: Os significados são modelados por objetos matemáticos, geralmente funções semânticas definidas composicionalmente, que representam o efeito de executar uma estrutura. Isto significa que o efeito interessa mais do que a forma como é produzida a computação;
- **Semântica axiomática**: Especificam-se propriedades do efeito da execução das estruturas como asserções, que são sentenças da lógica de predicados. Estas sentenças são usualmente chamadas axiomas, e daí o nome desta técnica. Alguns aspectos da execução são ignorados.

Na semântica operacional se enquadra um dos modelos mais utilizados, a gramática de atributos (PAAKKI, 1995). Este modelo foi citado, juntamente com os autômatos de pilha estruturados, como um dos fundamentos teóricos para a definição de uma linguagem utilizada em um metacompilador, denominado GAMA-NW (PFLÚCKER; NETO, 2007). A principal vantagem da GAMA-NW é que ela possui um potencial de reutilização e uniformiza a notação utilizada na definição dos aspectos léxicos, sintáticos e semânticos de uma linguagem de programação. Esta uniformização foi obtida a partir do conceito de textos paralelos (PFLÚCKER; NETO, 2007). Neste trabalho, no entanto, vamos utilizar os recursos implementados pela ferramenta AdapTools.

Entretanto, como trabalhos futuros, sugere-se um estudo mais detalhado e profundo sobre a GAMA-NW e textos paralelos e a sua aplicação para a formalização de linguagens.

Nos capítulos seguintes, os conceitos aqui expostos serão utilizados para o desenvolvimento de uma proposta de concepção de uma linguagem de programação imperativa, usando o autômato adaptativo como dispositivo formal para definição e descrição da mesma.

3 Concepção de Linguagens de Programação Imperativas Aderentes ao Modelo Adaptativo

Neste capítulo, realiza-se uma discussão sobre o processo de concepção de uma linguagem de programação imperativa, usando o autômato adaptativo como dispositivo formal para a definição e descrição de uma linguagem-base simplificada, bem como o projeto e a implementação do seu mecanismo de interpretação e compilação. O objetivo é demonstrar o uso de tais formalismos na especificação desta classe de linguagens de programação, servindo de base para a implementação de uma linguagem extensível e, posteriomente, de uma linguagem para codificação de programas adaptativos. Uma das ferramentas que aparece como alternativa experimental para a implementação desta linguagem é o AdaptTools (PISTORI; NETO, 2003a; PISTORI, 2003). Esta ferramenta foi concebida, inicialmente, com fins pedagógicos, para ser utilizada como um ambiente computacional onde os autômatos adaptativos podem ser projetados, implementados e testados, incluindo recursos de depuração e mecanismos de animação gráfica, controle de projetos e comunicação entre máquinas.

Por razões de simplicidade, optou-se pela definição de uma linguagem mínima, denominada MINLA (contração da expressão, em inglês, *MINimal LAnguage*), que será combinada posteriomente com um mecanismo de extensão, cuja estrutura será definida no capítulo seguinte. A seção seguinte descreve, de forma geral, todos os passos realizados para a obtenção de um ambiente capaz de tratar os programas escritos em MINLA. Posteriormente, este mesmo ambiente será utilizado para incorporar outras características e funcionalidades a programas ecritos nesta linguagem.

3.1 Visão Geral da Proposta

A linguagem *MINLA* é utilizada como linguagem-base imperativa, contendo todos os recursos mínimos necessários para a definição do núcleo mais primitivo da presente proposta, composto pelas componentes léxica, sintática e semântica da linguagem base

e pelas regras de mapeamento e representação, de cada um dos objetos elementares que a compõem, para um conjunto de primitivas que possa ser facilmente mapeada em um autômato adaptativo. Os reconhecedores para as componentes desta linguagem foram obtidos com a aplicação da ferramenta Wirth2APE (NETO; PARIENTE; LEONARDI, 1999; PISTORI, 2003). As Figuras 3.1 e 3.2 representam, respectivamente, o diagrama resumido de cada uma das etapas descritas acima.

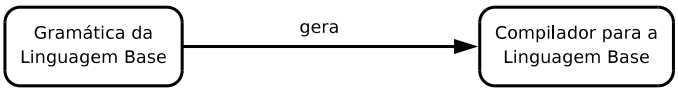


Figura 3.1: Obtenção do compilador para L_{bas} a partir de sua gramática.

É importante observar que no esquema resumido da Figura 3.1, a gramática da linguagem-base é descrita em notação de Wirth (Seção 2.1.1) e que, para a obtenção do compilador propriamente dito, existe uma etapa intermediária que gera o reconhecedor sintático para L_{bas} e define a biblioteca de ações semânticas que serão associadas às regras do respectivo reconhecedor. A ferramenta utilizada para gerar o reconhecedor é o Wirth2APE.

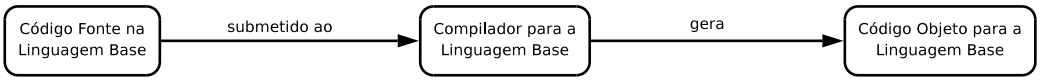


Figura 3.2: Obtenção do código-objeto para programas escritos em L_{bas}.

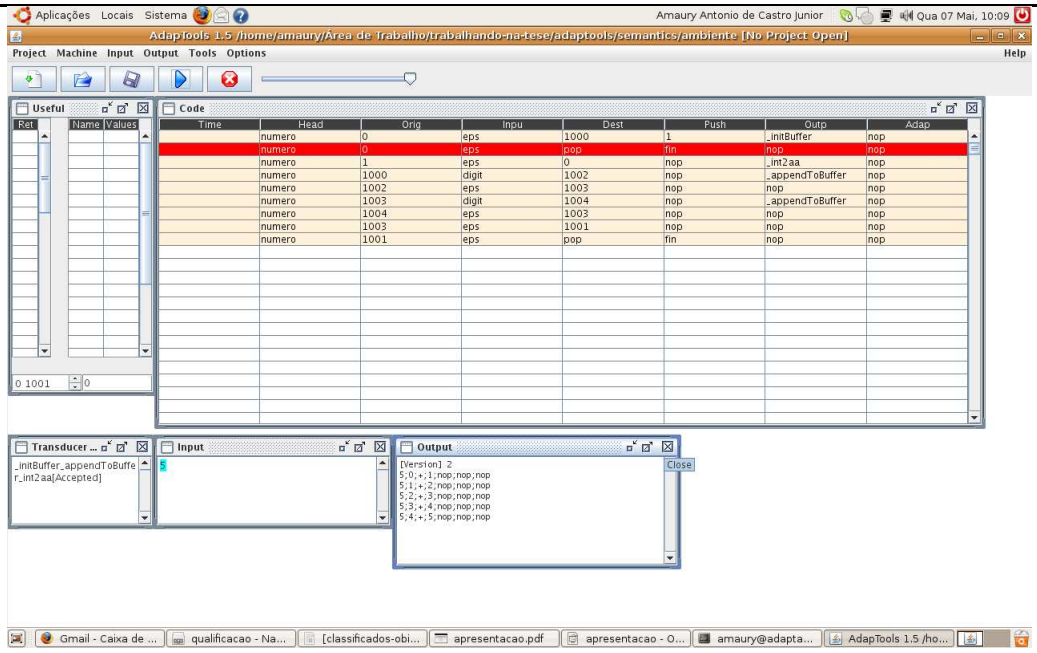
Para o funcionamento adequado do esquema acima descrito, foi proposto um ambiente operacional com os recursos necessários para a execução do código-objeto obtido a partir dos programas escritos na linguagem-base. Considerando a opção pelo modelo do autômato adaptativo, torna-se primordial a implementação das bibliotecas capazes de realizar as operações básicas de entrada e saída de objetos da linguagem, bem como das primitivas de operadores básicos, gerando um conjunto de funções que possam executar todas as operações básica da linguagem-base sobre o ambiente de execução proposto. O mesmo deve ser realizado para as primitivas de controle do fluxo de execução.

Com a definição rigorosa de todas as regras e convenções mínimas para a representação dos elementos da linguagem-base, pode-se utilizá-la como base para a linguagem extensível L_{ext} proposta, através do projeto de um mecanismo de extensão que será utilizado para incorporar as extensões definidas à linguagem-base L_{bas}. Esta etapa é descrita em detalhes no Capítulo 4.

A seção seguinte apresenta a ferramenta AdapTools e o Wirth2APE, implementado no AdapTools, que foram utilizados como base para a implementação dessa proposta.

3.2 AdapTools

O AdapTools é um software livre, escrito em JAVA, que foi originalmente desenvolvido com o objetivo de representar e simular autômatos adaptativos, bem como suas especializações, tais como os autômatos de pilha estruturados e os autômatos de estados finitos (PISTORI, 2003). O núcleo deste sistema é composto por uma máquina virtual que executa uma versão levemente modificada de um autômato adaptativo. Essas pequenas modificações no formalismo original simplificam e uniformizam o formato de especificação de transições internas, transições externas e das ações adaptativas elementares. Com isso, todos os elementos do dispositivo passam a poder ser apresentados através de uma única tabela (PISTORI; NETO, 2003c).



(a) Interface gráfica do AdapTools.

Time	Head	Orig	Inpu	Dest	Push	Outp	Adap

(b) Detalhe da tabela que descreve as máquinas representadas no AdapTools.

Figura 3.3: (a) Interface gráfica e (b) detalhe da tabela do AdapTools - versão 1.5.

O formato tabular escolhido apresenta 7 colunas: (1) cabeçalho (*head*), (2) estado de origem (*orig*), (3) símbolo de entrada (*inpu*), (4) estado de destino (*dest*), (5) empilha (*push*), (6) símbolo de saída (*outp*) e (7) ação adaptativa (*adap*). Dessa forma, no AdapTools, cada autômato (máquina de estados) é definido como uma tabela da ferramenta. Mais detalhes sobre o formato da tabela e das regras (linhas) pode ser obtido

em (PISTORI; NETO, 2003a).

Recentemente, o AdapTools passou por um processo de aperfeiçoamento, possibilitando a inclusão de novas funcionalidades e extensões (JESUS et al., 2007). O objetivo foi o de melhorar o suporte para o desenvolvimento de outras aplicações que façam uso de tecnologia adaptativa, uma vez que o AdapTools pode ser visto como um ambiente de execução para dispositivos baseados nesta tecnologia. A versão atual possui recursos para tratamento de não-determinismos e inclusão simplificada de novas ações semânticas que podem ser associadas às regras do dispositivo. Uma rotina semântica consiste de uma classe pré-definida em JAVA, que contém uma biblioteca de métodos associados à máquina em execução. Tais métodos são executados sempre que forem referenciados (pelo nome) na coluna correspondente à saída, dentro da tabela que define a respectiva máquina. Nesta situação, são gerados os eventos necessários, conforme a aplicação, para o tratamento adequado de cada uma das regras. No caso da implementação de um compilador dirigido por sintaxe, as rotinas semânticas podem ser associadas com os métodos que geram o código objeto correspondente. As rotinas semânticas disponíveis na ferramenta AdapTools podem ser visualizadas, simplificando a seleção do conjunto de ações semânticas a serem utilizadas pelo respectivo autômato.

Além das rotinas semânticas, um outro recurso extremamente útil para a construção de compiladores baseados na tecnologia adaptativa é a possibilidade de comunicação entre máquinas distintas. Este recurso permite que a saída de uma rotina semântica associada a uma máquina *A* possa ser redirecionada, como entrada, para uma outra máquina *B*.

3.2.1 Metacompilador Wirth para AdapTools

Algumas aplicações com finalidade pedagógica foram apresentadas por (PISTORI, 2003). Uma delas é o metacompilador que recebe, como entrada, uma especificação de uma linguagem livre de contexto, em notação de Wirth, e produz um reconhecedor para tal linguagem. Esse reconhecedor é compatível com a máquina virtual do AdapTools, podendo, portanto, ser diretamente executado após ter sido gerado. A implementação desta ferramenta foi realizada com base no trabalho descrito em (NETO; PARIENTE; LEONARDI, 1999).

Além dos objetivos pedagógicos, a implementação desse metacompilador também permitiu demonstrar o uso de um outro recurso poderoso do AdapTools, que é a execução concorrente de múltiplas máquinas e a possibilidade de comunicação entre elas. Com isso, as etapas de análise sintática e léxica são implementadas através de

duas máquinas distintas (autômatos de pilha estruturados) às quais se associam ações semânticas para a geração de código compatível com a própria ferramenta AdapTools. As máquinas que representam as componentes léxica e sintática podem ser executadas concorrentemente. Essas duas máquinas são definidas, na linguagem do AdapTools, no Anexo B.

Apesar de não haver uma rigidez do ponto de vista da implementação, conceitualmente um compilador pode ser decomposto em fases seqüencialmente organizadas, de modo que, uma vez realizada a leitura do arquivo com o programa-fonte, o produto de uma determinada fase seja a entrada da fase subseqüente até que, por fim, o programa-objeto seja criado. Segundo Neto (NETO, 1987), em função da enorme gama de opções para projeto e implementação de compiladores, e por questões de simplicidade, as fases de projeto de um compilador podem ser reunidas em três fases macroscópicas, envolvendo: análise léxica, análise sintática e a geração de código.

3.3 A Linguagem MINLA (MINimal LAnguage)

A linguagem MINLA possui um subconjunto de elementos comuns à maioria das linguagens imperativas, tais como: números inteiros, variáveis simples, expressões aritméticas simples (soma e subtração), comando de atribuição, comando de desvio incondicional (*goto*), estrutura condicional simples (*if-then*) e comandos simples para entrada e saída. As subseções seguintes definem as componentes léxica, sintática e semântica para a linguagem MINLA.

Para a representação de linguagens e, conseqüentemente, de gramáticas que as representam, existem diversas notações que podem ser utilizadas. Neste trabalho, foi utilizada a notação de *Wirth*, definida na Seção 2.1.1.

3.3.1 Componente Léxica

A componente léxica é a mais simples de ser definida, já que é composta por expressões regulares e, dessa forma, podem ser mapeadas diretamente em autômatos de estados finitos. A formalização da etapa de análise léxica envolve a descrição de todos os possíveis átomos da linguagem, incluindo nomes de variáveis, números, palavras reservadas, operadores, entre outros. Além disso, a análise léxica também é responsável por identificar e desconsiderar os caracteres especiais que não têm influência sobre o funcionamento da linguagem, tais como espaços em branco e comentários.

Exemplo 1 *Definição da componente léxica para a linguagem MINLA:*

tokenlexico	=	*palavrachave* | *identificador* | *numero* | *especial* .
palavrachave	=	*"E" "N" "D"* | *"G" "O" "T" "O"* | *"I" "F"* |
		"L" "E" "T" | *"P" "R" "I" "N" "T"* |
		"R" "E" "A" "D" | *"T" "H" "E" "N"* .
identificador	=	**letter** { **letter** | **digit** } .
numero	=	**digit** { **digit** } .
especial	=	**special** .

As expressões **letter**, **digit** *e* **special** *são utilizadas no AdapTools e identificam, respectivamente, letras (a...z e A...Z), dígitos (*0...9*) e qualquer símbolo que não seja nem letra nem dígito*[1]*, tais como os operadores aritméticos e de comparação.*

Observe que, da forma como foi definida, cada uma das regras da componente léxica é responsável pela extração e classificação (identificação) dos átomos e pela eliminação de partes não significativas das sentenças da linguagem. O resultado a ser produzido pela componente léxica é uma sequência de átomos que servirá como entrada para a componente sintática.

Em tempo de compilação, a componente léxica, além de extrair e classificar os átomos da linguagem, cria os construtos que representam cada um dos elementos léxicos da linguagem. Inicialmente, estes construtos correspondem a autômatos que contêm apenas as transições que representam os caracteres que formam cada uma das palavras reservadas da linguagem e ações adaptativas capazes de incorporar novos identificadores e associá-los às suas respectivas classificações. Para tanto, pode-se utilizar uma versão levemente modificada do coletor de nomes, proposto em (NETO, 1993), de tal forma que cada um dos identificadores possui um estado final que o identifique univocamente. Com isso, em tempo de execução pode-se, facilmente, percorrer o autômato construído até que o estado final seja atingido e, a partir dele, obter as informações sobre o identificador encontrado.

3.3.2 Componente Sintática

A componente sintática é usualmente descrita por uma gramática livre de contexto (Seção 2.1) que define a estrutura sintática livre de contexto da linguagem. Essa estrutura sintática é responsável pelo reconhecimento da seqüência de átomos gerada pela

[1] Por conveniência, os exemplos poderão utilizar algumas convenções próprias da ferramenta AdapTools. A intenção é deixar indicadas algumas situações que podem ser tratadas pela ferramenta através da implementação de ações semânticas em JAVA, facilitando a simulação e o ensaio de alguns aspectos definidos formalmente.

componente léxica, e é realizado por um processo muito semelhante ao dessa componente.

Dessa forma, a componente sintática é utilizada em conjunto com a componente léxica para gerar um reconhecedor sintático para uma dada linguagem. Os autômatos de pilha estruturados (Seção 2.2.1) podem ser utilizados como dispositivo formal para descrever a componente sintática da linguagem (NETO, 1987), resultando na especificação formal de um reconhecedor sintático responsável pela verificação da adequação da cadeia de átomos de entrada (proveniente da componente léxica) às regras de formação da linguagem (definidas pela componente sintática).

Exemplo 2 *Definição da componente sintática para a linguagem MINLA:*

programa = *comando* { *";" comando* } *"END"* .
comando = { **identificador** *":"* }
(*"LET"* **identificador** *":" "=" exparit*
| *"GOTO"* **identificador**
| *"READ"* (**identificador** { *","* **identificador** })
| *"PRINT"* (*exparit* { *","* *exparit* })
| *"IF" expcomp "THEN" comando*
| ε) .
expcomp = *exparit* (*"<"* | *"="* | *">"*) *exparit*
exparit = (**identificador** | **numero**)
{ (*"+"* | *"-"*) (**identificador** | **numero**) } .

As expressões **identificador** *e* **numero** *identificam átomos da linguagem que são gerados pela máquina que trata a componente léxica e tomados como entrada pela máquina que trata a componente sintática da linguagem. A linguagem MINLA inclui apenas variáveis simples, números inteiros, expressões aritméticas simples (soma e subtração) sem parênteses, expressões de comparação simples (>, < e =), comando de atribuição, declaração de rótulos, comando "if-then" (a cláusula "else" não existe nesta linguagem), comando de desvio incondicional e comandos simples de entrada e saída.*

Observe-se que, até o momento, apenas os aspectos regulares e livres de contexto da linguagem foram tratados. Entretanto, para uma descrição sintática completa de uma linguagem de programação imperativa, existe uma componente dependente de contexto. Esses aspectos sintáticos ainda não definidos, para a linguagem MINLA, devem ser incluídos no mecanismo de processamento dessa linguagem pela componente semântica.

3.3.3 Aspectos de Implementação

A operação dos dispositivos adaptativos poder ser descrita de forma incremental, e seu comportamento, programado para se alterar dinamicamente em resposta aos estímulos de entrada recebidos (NETO; DE MORAES, 2003; PISTORI, 2003). Desde os autômatos adaptativos, uma série de dispositivos formais adaptativos já foram desenvolvidos e utilizados no desenvolvimento de soluções para problemas complexos em áreas como processamento de linguagens naturais, inteligência artificial, engenharia de software, computação natural, processamento e reconhecimento de padrões, visão computacional, aprendizagem de máquina, entre outras (NETO; ROCHA, 2008). No caso das linguagens de programação, a proposta é a de utilizar a ferramenta AdapTools (PISTORI; NETO, 2003c) como suporte para a implementação dos aspectos adaptativos presentes no projeto de linguagens que possam ser por ela simulados.

Durante o desenvolvimento deste trabalho, estes recursos de simulação implementados pela ferramenta foram utilizados para gerar as máquinas que realizam a análise léxica e sintática da linguagem MINLA. Ambas as máquinas poderão receber novas ações semânticas que possam ser utilizadas para gerarem os construtos adaptativos correspondentes a cada um dos componentes da linguagem. Além disso, pode ser projetada uma biblioteca de funções adaptativas que realizem as modificações necessárias nas máquinas que representam as componentes léxica e sintática, respectivamente, de acordo com cada caso.

Existem ainda duas atividades que merecem destaque dentro da estrutura de um compilador por estarem diretamente ligadas a todas as fases que o compõem. Além do mecanismo de recuperação de erros e otimização de código, que também podem ser tratados adaptativamente (PISTORI, 2003; LUZ, 2004), outra atividade de grande relevância é o gerenciamento da tabela de símbolos, que é a estrutura de dados responsável pela manutenção das informações relativas aos atributos de cada identificador(AHO; SETHI; ULLMAN, 1995). O tratamento adaptativo da tabela de símbolos está descrito detalhadamente em (NETO, 1993).

Foi adotada esta visão macroscópica, proposta por Neto(NETO, 1987), como forma de guiar o processo de construção da linguagem. A linguagem e os conceitos apresentados aqui podem ser utilizados para a apresentação dos detalhes relacionados ao projeto da linguagem extensível e, a partir dela, de uma linguagem para programação adaptativa, sendo definidas como linguagens de programação que possibilitam a escrita de códigos que se modificam por meio das funções adaptativas, ou seja, que permitam a escrita de códigos adaptativos (FREITAS, 2008).

Neste contexto, partindo das gramáticas que definem as componentes léxica e sintática da linguagem MINLA, o metacompilador Wirth para AdapTools foi utilizado para gerar o código AdapTools para as componentes léxica e sintática da linguagem MINLA. O Anexo D mostra a máquina obtida deste processo.

3.3.4 Componente Semântica

Uma vez definidas as componente léxica, sintática, pode-se especificar um conjunto de primitivas para o ambiente de execução e os demais aspectos semânticos da linguagem. Conceitualmente, muitos autores divergem quanto ao limite existente entre os conceitos de sintaxe e semântica. Neste trabalho, optou-se por utilizar o conceito de sintaxe como o conjunto de aspectos livres de contexto e como semântica o conjunto de aspectos dependentes de contexto.

Os autômatos adaptativos serão adotados para a representação integral da linguagem. Esta seção descreve, conceitualmente, a interpretação que pode ser atribuída a todo o conjunto de sentenças da linguagem de forma aderente a um ambiente de execução baseado nos autômatos adaptativos e apresenta algumas estratégias para a geração de código aderente ao formalismo adaptativo para a linguagem MINLA, definida previamente. Nesta etapa, o foco é definir o mapeamento de cada uma das componentes da linguagem MINLA em uma representação equivalente no modelo dos autômatos adaptativos. Para isso, foi criado um conjunto de primitivas baseada no modo de operação desse formalismo, relacionadas no Anexo A. Cada uma das primitivas possui um nome e recebe um conjunto de parâmetros que correspondem a um identificador de um objeto representado na forma de um autômato adaptativo, denimonado **construto**.

As seções seguintes definem o mapemaneto para cada um dos casos a serem tratados.

3.3.4.1 Representação de Variáveis Simples

A linguagem MINLA não possui tratamento de tipos e possui escopo único. A intenção, neste momento, é simplificar o processo de representação dos nomes de variáveis utilizados por um programa em MINLA na forma de um autômato adaptativo. Para isso, pode-se utilizar um autômato com estrutura similar ao de uma simples árvore de busca.

Na prática, uma variável representa uma posição de memória onde será armazenado algum valor. Na representação adaptativa, uma variável é representada como um

autômato construído de tal forma que, partindo-se do estado inicial desse autômato, percorre-se uma seqüência de transições responsáveis pelo consumo sucessivo dos caracteres que compõem o nome da variável. Por fim, atinge-se um estado final, específico para o identificador reconhecido. Esse será o estado diferenciado que identifica univocamente a variável associada ao identificador em questão. O autômato que representa cada um dos identificadores reconhecidos pela linguagem pode ser construído através da operação de um coletor de nomes (NETO, 1993; NETO, 2001), que é uma estrutura similar a uma tabela de símbolos, representada na forma de um autômato adaptativo. O coletor de nomes pode ser executado a partir de uma seção de declaração de variáveis. A primitiva *def(?o,n)* é responsável pela criação do construto adaptativo correspondente ao identificador *n* encontrado e pela sua associação com o construto ?*o*, que pode ser considerado como um elemento da representação que identifica, univocamente, o respectivo identificador. Caso o construto já tenha sido definido previamente, a primitiva *def* apenas o identifica.

Considerando que as variáveis correspondem a identificadores que representam valores, também existe a necessidade de definir como será realizada esta associação entre os construtos adaptativos que representam as variáveis e seus respectivos valores. A primitiva *val(?o, n)* é responsável pela associação do construto ?*o* com a constante cujo valor está armazenado na variável *n*. Ou seja, caso uma variável *soma* armazene o valor 5, a primitiva *val(?o, n)* associará a constante 5 ao construto ?*o*. A seção seguinte explica a representação de constantes inteiras neste modelo.

3.3.4.2 Representação de Constantes Inteiras

De modo análogo às variáveis, as constantes inteiras também pode ser representado por um autômato com um único estado de entrada e um único estado de saída. Estabelece-se, por convenção, que a constante 0(zero) é representada como um autômato com um só estado, sem qualquer transição, e que esse estado é um estado final. Para os demais valores decimais, o número de transições corresponde ao valor representado, estabelecendo-se uma notação unária para a representação dos mesmos. Esta representação unária possibilita a representação de números positivos e negativos. Pode-se, por exemplo, optar por representar as constantes positivas com transições que consomem o símbolo '$+$' e as constantes negativas com transições que consomem o símbolo '$-$'. A primitiva *eval(?o,v)* é responsável pela criação do construto adaptativo que representa cada um dos valores inteiros encontrados no programa e a sua associação ao construto ?*o*. A primitiva *neg(?o)* é responsável pela inversão do sinal do construto *o,* que representa alguma constante previamente definida. Esta primitiva pode ser utilizada após a primitiva *replic(?o,?*o_1*)*, que realiza uma cópia do cons-

truto $?o_1$ e o associa ao construto $?o$. Dessa forma, pode-se manter o valor original, realizando-se a troca de sinal apenas sobre a cópia do respectivo construto.

3.3.4.3 Expressões Aritméticas

Na linguagem MINLA, foram definidas apenas as operações de soma ($+$) e de subtração ($-$), cujos operandos são números inteiros ou variáveis, cuja representação adaptativa foi proposta nas Seções 3.3.4.1 e 3.3.4.2. No caso de variáveis, em sua representação adaptativa, os estados finais dos construtos que as representam identificam seus respectivos valores (e demais informações, se for o caso). Observe que os identificadores também podem estar associados a tipos, escopos ou outros elementos que não serão tratados neste trabalho, mas podem fazer parte da definição de uma linguagem de programação imperativa. Portanto, uma vez obtidos os construtos que representam os operandos, a operação soma produzirá um terceiro construto adaptativo que representa o resultado desta operação, onde o número de transições resultantes corresponde ao número de transições presentes na concatenação dos construtos que representam cada um dos operandos.

A primitiva *add(*$?o$*,* $?o_1$*,* $?o_2$*)* é responsável pela execução da operação de adição sobre os construtos $?o_1$ e $?o_2$, passados como parâmetros. Vale ressaltar que os construtos $?o_1$ e $?o_2$ podem ser obtidos pelas primitivas *val* ou *eval*, definidas anteriormente, e pela associação do resultado obtido ao construto o. Para isso, pode-se replicar os construtos e, sobre suas respectivas cópias, executa-se uma ação adaptativa que conecta o estado imediatamente anterior ao estado final do primeiro construto ao estado de entrada do segundo. O construto resultante corresponde à representação adaptativa do resultado. Observe que a conexão entre os dois construtos simula uma "fusão" entre os estados de saída e entrada dos construtos passados como parâmetro, tornando inatingível o estado de saída do primeiro.

No caso da operação de subtração também pode-se replicar (copiar) as suas respectivas representações e inverter o sinal das transições do segundo operando. A primitiva *neg(*$?o$*)*, definida anteriormente, realiza a troca do sinal do construto $?o$. Portanto, a operação de subtração entre dois valores numéricos v_1 e v_2 consiste da eliminação de pares de transições nas cópias realizadas que possuem sinais opostos, sendo uma do primeiro operando e outra do segundo, até que se esgotem as transições de um dos operandos, ficando o resultado representado no autômato restante e com o sinal do maior número, ou seja, com o sinal do construto com o maior número de transições.

3.3.4.4 Expressões de Comparação

O resultado das operações de comparação podem ser simuladas a partir de operações aritméticas de subtração sobre as representações de seus operandos, seguidas de um teste sobre o resultado da diferença. Para tanto, foram definidas as primitivas *compig*, *compil* e *compie*. A combinação destas primitivas permite a implementação das operações de comparação em MINLA. Por convenção, também se define a representação das constantes lógicas *true* e *false* como sendo, respectivamente, os construtos associados às constantes aritméticas 0(zero) e diferentes de 0(zero). Dessa maneira, as operações de comparação podem ser simuladas com facilidade a partir de operações aritméticas sobre as suas representações adaptativas, considerando a representação dos sinais para os números inteiros negativos. Neste caso, a verificação do sinal do construto que representa o resultado da expressão de comparação define a qual constante lógica será conectado o estado de saída do construto que implementa a operação de comparação. Depois disso, basta verificar o valor da constante associado para decidir entre as duas possibilidades:

1. Caso o construto que implementa a constante *true* seja o resultado, então acrescentam-se transições que conectam o estado de saída do construto que calcula o resultado da expressão de comparação ao construto que implementa a cláusula *then*;

2. Caso o construto que implementa a constante *false* seja o resultado, então acrescentam-se transições que conectam o estado de saída do construto que calcula o resultado da expressão de comparação ao construto que implementa a instrução seguinte do programa em MINLA.

A primitiva *compXX(?o,* $?o_1$*,* $?o_2$*)* realiza uma operação de subtração entre os valores numéricos representados pelos construtos $?o_1$ e $?o_2$, passados como parâmetros. A representação do valor do resultado será associada ao construto *o* e corresponderá a um valor lógico, definido de acordo com o sinal do resultado da operação de subtração, seguindo os critérios abaixo relacionados:

- Para a primitiva *compig*, que implementa o operador $>$, o resultado da comparação é *true* quando o sinal do resultado da subtração é positivo e *false*, caso contrário;

- Para a primitiva *compil*, que implementa o operador $<$, o resultado da comparação é *true* quando o sinal do resultado da subtração é negativo e *false*, caso contrário;

- Para a primitiva *compie*, que implementa o operador =, o resultado da comparação é *true* quando o resultado da subtração é 0(zero) e *false*, caso contrário;

3.3.4.5 Comando de Atribuição

O comando de atribuição consiste dos seguintes elementos: da palavra reservada "*LET*", um identificador (nome de uma variável), o operador de atribuição (":=") e uma expressão aritmética. Depois de consumir o átomo "*LET*" e criar a representação adaptativa do identificador do lado esquerdo do operador de atribuição, o estado de saída do autômato que representa o identificador deve se conectar ao estado de entrada do resultado obtido depois de processada a expressão aritmética do lado direito. Dessa forma, o comando de atribuição é responsável por modificar a transição que aponta a representação do valor associado ao identificador (caso exista algum) para passar a apontar o resultado da expressão aritmética correspondente.

A primitiva *conn(o_1, o_2)* realiza a conexão entre os estados de saída do construto o_1 e o estado de entrada do construto o_2. Vale ressaltar que a conexão realizada pela primitiva *conn* é diferente da conexão realizada pela primitiva *add*, pois esta última realiza uma "fusão" entre os estados de saída e de entrada dos construtos, enquanto a outra não.

3.3.4.6 Definição de Rótulos e Desvio Incondicional

Na linguagem MINLA, uma determinada seqüência de comandos pode estar associada a um ou mais rótulos, representados por identificadores seguidos de dois pontos, que são utilizados como referências a trechos de código, que podem ser executados através de desvios incondicionais. Portanto, considerando que um rótulo é simbolizado por um identificador, a sua representação adaptativa corresponde a um autômato que representa um nome (atribuído ao rótulo), cujo estado de saída se conecta ao estado de entrada do trecho de código referenciado pelo respectivo rótulo. A primitiva *defl(?o, n)* é responsável pela criação do construto adaptativo que representa o rótulo e de sua associação com o respectivo trecho de código.

A operação de desvio incondicional na linguagem MINLA (*goto*) pode ser representada por uma transição em vazio, que liga o estado de saída do autômato que modela o comando imediatamente anterior ao comando de desvio ao estado de entrada associado a algum rótulo definido dentro do programa, conforme descrito anteriormente. A primitiva *jump(?o)* é responsável pela criação da transição que liga o estado corrente do programa em execução ao estado associado ao construto ?*o*, cujo valor pode ser obtido pela primitiva *val(?o, n)*, onde *o* representa o construto e *n* corresponde ao nome

dado ao respectivo rótulo.

É importante ressaltar que, neste caso, o que diferencia a definição de uma variável da definição de um rótulo é que a primeira se associa a valores (dados), enquanto o segundo se associa a trechos de um programa (código). Esta informação também pode ser armazenada na forma de um autômato, acrescentando-se uma transição adicional à forma definida para a representação do nome. Esta transição liga os respectivos estados de saída do nome a uma descrição de seu tipo (variável ou rótulo).

3.3.4.7 Estrutura Condicional

As estruturas condicionais possuem expressões de comparação (lógicas) associadas e podem ser combinadas com o comando de desvio incondicional, possibilitando a implementação da capacidade de selecionar qual o comando a ser executado após a avaliação da respectiva expressão de comparação. A linguagem MINLA oferece apenas a forma mais simples de estrutura condicional: o comando *if-then*. Nesta estrutura, a decisão tomada define apenas se executa ou não um determinado comando, conforme o valor assumido por uma condição.

A representação adaptativa das estruturas condicionais consiste, primeiramente, na geração do autômato que resulta da execução de uma expressão de comparação e de dois rótulos que representam transições "indefinidas" do fluxo de execução. Por convenção, as transições indefinidas são representadas da mesma forma que um construto da linguagem, seguida de dois pontos e de uma seta para a direita (?o :$\rightarrow$) e se associam à instrução seguinte na sequência de código gerada. A escolha da transição a ser criada é definida pelos seguintes critérios:

- Caso o resultado da expressão de comparação seja *true*, cria-se uma transição para o autômato que representa o trecho de código associado à cláusula *then*;
- Caso o resultado da expressão de comparação seja *false*, cria-se uma transição para o autômato que representa trecho de código associado à instrução seguinte à cláusula *then*.

A primitiva *jcond(*?o_1, ?o_2, ?o_3*)* é responsável pela avaliação do resultado da expressão de comparação, representado pelo construto ?o_1 e pela decisão sobre qual o caminho a ser seguido, se o trecho representado pelo construto ?o_2 (quando ?$o_1 = true$) ou pelo trecho representado pelo construto ?o_3 (quando ?$o_1 = false$).

Na linguagem MINLA, esta estrutura pode ser combinada com a cláusula *goto* para a implementação de desvios condicionais.

3.3.4.8 Operações de Entrada e Saída

As operações de entrada e saída de valores são similares quanto ao modo de operação, ao comando de atribuição, com a diferença de que os valores são lidos de algum meio externo ao programa, como o teclado, por exemplo. Neste caso, pode-se implementar tais operações através de primitivas que realizam a leitura dos dados do dispositivo de entrada correspondente e constroem o autômato que representa o dado lido. A primitiva *read(*?*o)* é responsável por esta tarefa.

Em seguida, a exemplo do que acontece com a operação de atribuição, o estado inicial deste autômato deve ser apontado por uma transição em vazio partindo do estado final que identifica a variável que armazenará o valor lido. Neste caso, pode-se após a obtenção do construto que representa o valor lido, através da primitiva *read()*, pode-se utilizar a primitiva *conn* para realizar a associação entre o construto adaptativo correspondente ao dado lido e o construto ?*o* que representa uma variável que armazenará o valor lido.

A operação de conversão do formato binário de entrada para a forma de autômato que o representa deve fazer parte do repertório de primitivas disponíveis na máquina de execução, assim como outras operações primitivas que auxiliarão na execução das demais estruturas da linguagem.

No caso da escrita, a operação é inversa e consiste em promover a saída física dos dados representados no computador para o meio externo, por exemplo, o vídeo. Isso é feito enviando-se para o periférico um padrão de bits que represente o dado, no formato adequado, para o dispositivo de saída em questão. Esse padrão de bits é construído a partir do autômato que representa o valor do dado associado à variável referenciada pelo comando de saída. Essa conversão para formato binário de saída deve também fazer parte do conjunto de operações primitivas da máquina de execução. A primitiva *write(*?*o)* é responsável por esta tarefa, onde ?*o* representa o construto adaptativo a ser convertido.

3.4 O Ambiente de Interpretação e Execução para a Linguagem MINLA

Para o funcionamento adequado de qualquer linguagem de programação deve ser definido um ambiente de execução compatível e capaz de realizar e processar a semântica relativa a todas as construções sintáticas da linguagem definida. De forma prática, esse ambiente de execução exige, essencialmente, o projeto de um conjunto de ativi-

dades que possam ser executadas por algum ambiente de *run-time* com mecanismos de interpretação para os autômatos adaptativos. O AdapTools(PISTORI; NETO, 2003a; JESUS et al., 2007) possui essas características e é a ferramenta utilizada neste trabalho de pesquisa.

Uma possível proposta de um mecanismo de controle para o ambiente de execução pode ser pensada de forma similar ao reconhecedor simultâneo para um conjunto de linguagens, proposto em (NETO, 2000). Na proposta de reconhecimento de múltiplas linguagens, esta proposta utiliza-se da idéia de multiprogramação durante a execução do autômato. Entretanto, no caso do uso para simulação do mecanismo de controle de execução, o modelo proposto pode simplificar o entendimento e a definição de todos os elementos requeridos para o tratamento e a execução de um programa adaptativo. Dessa forma, pode-se propor um ambiente de controle e de execução totalmente baseado nos autômatos adaptativos, garantindo a possibilidade de utilização plena de todas as vantagens do modelo sem depender da maquina ou do sistema operacional em uso.

Exemplo 3 *O programa abaixo, escrito em MINLA, lê o valor de duas variáveis, calcula a soma dos dois valores lidos e escreve o resultado na tela:*

```
READ a;
READ b;
LET c := a + b;
PRINT c
END
```

O código gerado para o programa acima corresponde à execução das primitivas:

```
def (?a, "a")
read (?v_a)
conn (?a, ?v_a)
def (?b, "b")
read (?v_b)
conn (?b, ?v_b)
val (?v_a, "a")
repl (?vc_a, ?v_a)
val (?v_b, "b")
repl (?vc_b, ?v_b)
add (?v_c,?vc_a,?vc_b)
write (?v_c)
```

A definição das variáveis "*a*", "*b*" e "*c*" pode ser feita através da associação dos construtos adaptativos que representam estes identificadores a um estado que os identifica dentro do autômato. Tal estado compõe, juntamente com outros estados que podem identificar rótulos e outros símbolos utilizados pelo programa, o espaço correspondente à tabela de símbolos para o programa em execução. Os efeitos da execução do conjunto de primitivas da linguagem resultará no construto adaptativo que representa o programa escrito na linguagem MINLA.

Exemplo 4 *O programa abaixo, escrito em MINLA, lê o valor de uma variável e escreve o valor se ele for positivo:*

```
READ a;
IF a > 0 THEN PRINT a
END
```

O código gerado para o programa acima corresponde à execução das primitivas:

```
def (?a, "a")
read (?v_a)
conn (?a, ?v_a)
val (?v_a, "a")
eval (?z, 0)
cmpig (?r, ?l_1, ?l_2)
?l_1 :→
val (?v_a, "a")
write (?v_a)
?l_2 :→
```

O exemplo anterior mostra a seqüência de primitivas gerada para o comando condicional. Os rótulos criados, neste caso, são internos e devem possuir uma notação apropriada, que os diferencie dos rótulos definidos pelo usuário. Da mesma forma que no exemplo anterior, os efeitos da execução do conjunto de primitivas da linguagem resultará no construto adaptativo que representa o programa correspondente, escrito na linguagem MINLA.

Vale ressaltar que, em ambos os casos, a execução do construto adaptativo gerado depende da implementação de uma biblioteca de ações adaptativas ou semânticas que, quando associadas ao ambiente de execução, torna possível a interpretação deste construto e a produção de seus efeitos.

O objetivo deste capítulo foi o de oferecer uma base conceitual para o projeto e implementação do ambiente de interpretação e execução que possa tratar adequadamente uma linguagem de programação imperativa, que seja totalmente aderente ao modelo adaptativo. Entretanto, de forma prática, a implementação deste mecanismo exige a definição de estratégias de simulação dos aspectos teóricos envolvidos. Mais detalhes sobre este assunto constam no Capítulo 5. No capítulo seguinte, a mesma base conceitual discutido neste capítulo é discutida. Entretanto, o que se discute são os aspectos envolvidos no projeto de linguagens extensíveis de programação.

4 Concepção de Linguagens Extensíveis de Programação

Neste capítulo, apresenta-se um breve histórico sobre a extensibilidade de linguagens e discute-se todo o processo de concepção de uma linguagem extensível de programação. A linguagem utilizada como base para o tratamento das extensões é a MINLA, definida no capítulo anterior. O objetivo é descrever, conceitualmente, como utilizar o autômato adaptativo como dispositivo formal para a definição e descrição do ambiente de interpretação e execução que dê suporte ao mecanismo de extensão proposto. Com isso, pode-se demonstrar o uso de tais formalismos na especificação desta classe de linguagens de programação, servindo de base para a implementação de uma linguagem extensível e, futuramente, de uma linguagem para codificação de programas adaptativos.

4.1 Histórico

Os primeiros trabalhos científicos explorando o conceito de extensibilidade de linguagens de programação foram produzidos na década de 60. Um dos primeiros trabalhos nesta área descreve o uso de instruções de macros em linguagens de programação de alto nível (MCILROY, 1960). Outra técnica clássica para tratar a extensibilidade de linguagens de programação foi apresentada em (BROOKER; MORRIS, 1962).

No final de década de 60 e início da década de 70, as pesquisas sobre extensibilidade atingiram o seu auge com a realização de dois simpósios sobre linguagens extensíveis nos anos de 1969 (CHRISTENSEN; SHAW, 1969) e 1971 (SCHUMAN, 1971). Uma introdução às linguagens extensíveis foi apresentada em (PERLIS, 1969). No mesmo ano, foram discutidos alguns aspectos e características de extensibilidade sobre as linguagens de programação Algol68 (MAILLOUX; PECK, 1969) e PPL (STANDISH, 1969b). Outros trabalhos apresentaram pontos de vista sobre conceito, motivação, alternativas e técnicas para o estudo e implementação de linguagens extensíveis (WEGNER, 1969; CHEATHAM JR., 1969; MCILROY, 1969; STANDISH, 1969a).

Em 1971, o evento contou com trabalhos relatando experimentos com linguagens extensíveis (STANDISH, 1971; CHANDLER, 1971), perspectivas da área (CHEATHAM JR., 1971; DUBY, 1971) e formas de implementação de extensões com macros sintáticas (VAN GILS, 1971; WODON, 1971). Entretanto, apesar de alguns esforços isolados para tentar manter a comunidade científica motivada (HUBBARD JR., 1976), o movimento científico sobre o tema ficou reduzido a alguns trabalhos de análise dos resultados previamente obtidos (VAN GILS, 1972; STANDISH, 1975; METZNER, 1979). Um dos principais motivos utilizados para justificar a redução do interesse da comunidade científica pelo tema foi a necessidade de aumentar os níveis de abstração das linguagens de programação, visando omitir do usuário os detalhes de baixo nível de implementação de linguagens (STANDISH, 1975). Em outras palavras, a programação de sucessivas extensões de uma linguagem exigia um conhecimento prévio das linguagens dos níveis anteriores, antes da inclusão das novas construções e definições. Com isso, o movimento científico sobre as linguagens extensíveis perdeu espaço para a preocupação com a busca de um nível de abstração cada vez mais alto para a programação de computadores, sem a preocupação com a manutenção da capacidade de extensão das linguagens.

Entretanto, o surgimento mais recente de novas técnicas e modelos para a implementação de linguagens provocou um impacto positivo no desenvolvimento e aperfeiçoamento dos conceitos de programação. Com isso, surgiram alternativas para o tratamento das características de dependências de contexto das linguagens extensíveis (STEELE, 1999; KOLBLY, 2002; BRABRAND; SCHWARTZBACH, 2002). Um formalismo muito promissor para a implementação de linguagens extensíveis é o autômato adaptativo (NETO, 1994). Nesse contexto, o autômato adaptativo se apresenta como um modelo capaz de expressar e manipular, naturalmente, a dependência de contexto existente na descrição da linguagem de um mecanismo de extensão, definido como um componente independente da linguagem-base, mas que pode ser acoplado à mesma, tornando-a extensível.

Na seções seguintes, são discutidas algumas características importantes das linguagens extensíveis, e apresentadas algumas possíveis formas de implementação para elas. Em seguida, é discutido como os autômatos adaptativos podem ser usados como modelo para a descrever todo o processo de implementação e processamento das linguagens extensíveis, através de mecanismos estritamente sintáticos.

4.2 Visão Geral da Proposta

Com a definição de todos as regras e convenções mínimas para a representação dos elementos da linguagem-base, pode-se utilizá-la como linguagem hospedeira para a linguagem extensível L_{ext} proposta, através do projeto de um mecanismo de extensão que será utilizado para incorporar as extensões definidas à linguagem-base L_{bas}.

Para a obtenção do reconhecedor sintático das extensões propostas pelo usuário o metacompilador Wirth2APE também foi utilizado. A partir desta definição, foi gerado um autômato de pilha estruturado que reconhece a nova extensão definida (NETO; PARIENTE; LEONARDI, 1999) que pode ser incorporada à linguagem base a partir de uma biblioteca de ações adaptativas do ambiente de execução que estejam adequadamente acopladas ao código da máquina que representa o compilador para a linguagem L_{ext}. Tais ações adaptativas são associadas às regras do compilador obtido para gerar as auto-modificações obtidas a partir das extensões sintáticas definidas pelo usuário.

De forma prática, o compilador para L_{ext} pode ser obtido da mesma forma que para L_{bas}, excetuando-se as ações adaptativas, que devem ser projetadas separadamente. Neste caso, considera-se que a gramática da linguagem extensível é definida como a junção entre as gramáticas que definem o mecanismo de extensão M e a linguagem base L_{bas}, acoplando-se uma biblioteca de ações adaptativas para o tratamento dos aspectos de incorporação das extensões sintática definidas pelo usuário. As Figuras 4.1 e 4.2 representam o esquema descrito.

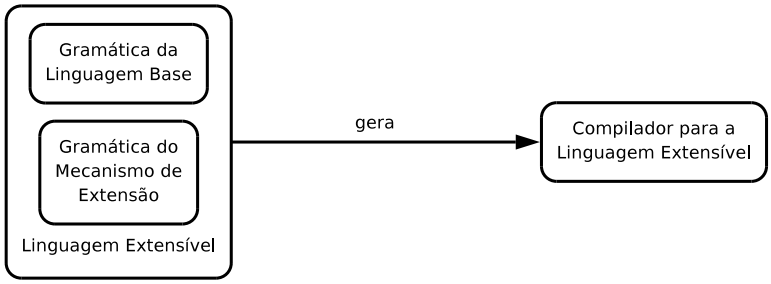


Figura 4.1: Obtenção do compilador para L_{ext} a partir das gramáticas de L_{bas} e da gramática do Mecanismo de Extensão.

É importante observar que no esquema resumido da Figura 4.1, a gramática da linguagem extensível é descrita em notação de Wirth (Seção 2.1.1) e que, para a obtenção do compilador da linguagem extensível, existe uma fase intermediária de construção do reconhecedor apropriado. Após esta etapa, deve ser implementada a biblioteca de ações semânticas que serão associadas às regras do respectivo reconhecedor gerado para o tratamento das extensões. A ferramenta utilizada para gerar o reconhecedor é o Wirth2APE. A biblioteca de ações semânticas para o tratamento da extensibilidade não

foi implementada neste trabalho. Entretanto, partindo desta proposta, pode-se sugerir esta implementação dessa biblioteca como trabalhos futuros.

Observe que, partindo da gramática original do mecanismo de extensão acoplado à linguagem base utilizada, pode-se obter a instância C_0 do reconhecedor sintático através do mesmo processo representado na Figuras 3.1. A diferença, entretanto, é que a gramática da linguagem-base é substituída pela gramática do mecanismo de extensão adicionado à gramática da linguagem-base utilizada.

Dessa forma, para tratar as extensões sintáticas do código em L_{ext}, projetam-se as seguintes componentes, que são acionadas em tempo de compilação:

- Uma biblioteca de ações adaptativas, para gerar as novas extensões em um formato que seja compatível com o ambiente de execução proposto para L_{bas}, que será o mesmo utilizado por L_{ext};
- Uma biblioteca de ações adaptativas, para incorporar as extensões obtidas à nova instância de L_{bas}.

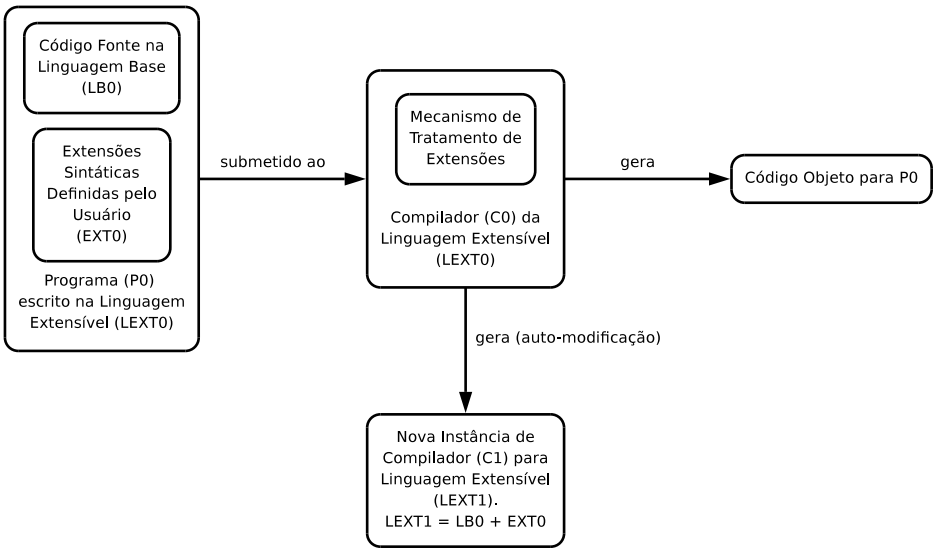


Figura 4.2: Obtenção do código objeto para programas escritos em L_{ext} e de uma nova instância do compilador de L_{ext}, incorporando as novas construções sintáticas definidas pelo usuário.

De posse da instância C_0 do compilador e das componentes acima descritas é possível gerar novas linguagens e, consequentemente, novas instâncias de compiladores que possam processar estas novas linguagens.

4.3 Formalização de Linguagens Extensíveis

A definição formal de linguagens extensíveis deve apresentar dois níveis de sintaxe:

- A sintaxe de uma linguagem-base;
- A linguagem utilizada pelo mecanismo de extensão.

De forma mais detalhada, partindo de uma linguagem-base e usando um mecanismo de extensão apropriado, um usuário de uma linguagem extensível pode criar notações, novas estruturas de dados, novas operações, estruturas de controle e regras de sintaxe. Dessa forma, o usuário pode modificar as características de uma linguagem para adaptá-la às suas necessidade e propósitos (STANDISH, 1975).

Analisando uma linguagem extensível, percebe-se a necessidade de que a linguagem da própria máquina de execução possa ser estendida de acordo com as novas definições fornecidas pelo usuário. Portanto, um formalismo extensível parece ser uma escolha lógica para a especificação de linguagens com sintaxe extensível. Nesse contexto, o autômato adaptativo apresenta-se como uma alternativa adequada, permitindo que a especificação formal da classe de linguagens extensíveis e de seus processadores seja realizada de maneira natural.

O Autômato Adaptativo, definido na Seção 2.2, pode ser usado como um dispositivo formal alternativo na descrição e implementação de analisadores sintáticos para linguagens extensíveis pois, a partir de uma configuração original, os autômatos adaptativos podem evoluir quando necessário, incorporando novas potencialidades quando necessário. Dessa forma, durante o consumo dos símbolos de entrada pode não haver nenhuma alteração na configuração original do autômato adaptativo e as alterações na topologia do autômato podem ser projetadas para que ocorram somente quando for definida pelo usuário uma forma sintática válida, mas ainda desconhecida pelo reconhecedor da linguagem extensível.

A aceitação de uma sentença, neste caso, pode ser interpretada como uma seqüência de reconhecimentos parciais, cada um deles manipulado por um autômato adaptativo em uma configuração intermediária diferente. Este modo de operação permite que, tanto a dependência de contexto (aspectos sintáticos conhecidos como semântica estática), como a sintaxe dinâmica (representada pelo mecanismo de extensão sintática), possam ser tratados de modo estritamente sintático, sem nenhuma ajuda de regras semânticas. Com isso, palavras reservadas, tabelas de símbolos, escopos, prototipagem, passagem de parâmetros, definição e expansão de macros, definição de novas construções sintáticas, e outros tipos de dependência de contexto podem ser

facilmente e naturalmente manipulados de forma sintática pelos autômatos adaptativos (NETO, 1993).

4.4 A Linguagem EXTLA (EXTensible LAnguage)

Nesta seção, apresenta-se a linguagem EXTLA (contração da expressão, em inglês, *EXTensible LAnguage*), sobre a qual pode ser definido um conjunto de novas construções sintáticas, utilizando a linguagem MINLA como base.

4.4.1 Mecanismo de Extensão

Existem duas formas muito populares e simples para que os usuários possam realizar modificações sobre uma linguagem de programação: (1) definição e chamada de funções e subrotinas e (2) definição e expansão de macros léxicas. Embora seus mecanismos de tratamento sejam distintos, ambas podem ser tratadas no nível léxico. A primeira consiste na definição e chamada de procedimentos e funções parametrizáveis e mutuamente dependentes a até recursivas. O segundo caso consiste apenas da manipulação de sucessivas substituições das abreviações simbólicas, encontradas no programa, pelas suas respectivas definições em seqüências textuais da própria linguagem. Em (NETO, 1993), pode-se encontrar uma explicação detalhada sobre a manipulação de definições e expansões de macros simples, parametrizadas e mutuamente dependentes através de um mecanismo de processamento de linguagens baseado nos autômatos adaptativos.

Entretanto, no caso da sintaxe dinâmica, existem aspectos mais complexos do que os da sintaxe convencional de uma linguagem de programação que devem manipular mudanças estruturais, definidas pelo usuário. Tais mudanças exigem que o próprio compilador possa ser modificado em tempo de compilação dos programas-fontes, em resposta às modificações requeridas pelo usuário sobre a linguagem original.

Neste trabalho, discute-se como manipular estes aspectos da sintaxe dinâmica através da descrição de tais características em função da teoria dos autômatos adaptativos. Será definido um mecanismo de extensão, implementado sobre uma seqüência simplificada de comandos para definição de novas construções sintáticas sobre uma linguagem-base. A linguagem MINLA, descrita no capítulo anterior, será utilizada como linguagem-base. A metalinguagem definida para o mecanismo de extensão será responsável pelo mapeamento das novas construções sintáticas ao seu significado, expresso através de comandos e instruções já reconhecidos pelo processador da linguagem-base ou por qualquer extensão sintática previamente definida (dependência

mútua).

4.4.2 Componente Léxica

A componente léxica de uma linguagem extensível é descrita por expressões regulares (LEWIS; PAPADIMITRIOU, 1997; RAMOS; NETO; VEGA, 2009) da linguagem-base, acrescidas das expressões regulares que compõem a linguagem que define o mecanismo de extensão. No caso da linguagem EXTLA, a diferença básica está na inclusão de novas palavras-chave, responsáveis pela definição das novas construções da linguagem.

Dessa forma, esta componente continua permitindo o mapeamento direto dos elementos que compõem um programa escrito nessa linguagem em autômatos de estados finitos que descrevem todos os possíveis átomos da linguagem, incluindo nomes de variáveis, números, palavras reservadas, operadores, entre outros. Além disso, a análise léxica também é responsável por identificar e desconsiderar os caracteres especiais que não têm influência sobre o funcionamento da linguagem, tais como espaços em branco e comentários. Portanto, o resultado a ser produzido pela componente léxica é uma seqüência de átomos, que servirá como entrada para a componente sintática, que, por sua vez, terá um tratamento diferenciado da componente sintática da linguagem-base, devido à capacidade de incorporar novas funcionalidades de forma incremental, por solicitação do usuário, mediante a definição de extensões.

4.4.3 Componente Sintática

Para estender a sintaxe de um linguagem-base deve-se fazer com que o compilador possa aceitar uma nova coleção de regras gramaticais, definidas em tempo de programação através de mecanismos declarativos incorporados à linguagem-base. Tais regras podem ser especificadas em notação de Wirth. A associação entre a nova sintaxe e a semântica correspondente requer uma seção que defina a seqüência de comandos da linguagem-base que realizam uma operação equivalente ao novo comando que está sendo especificado.

Dessa forma, partindo da gramática livre de contexto que define a nova sintaxe, pode-se obter facilmente, através de mecanismos já conhecidos (NETO; PARIENTE; LEONARDI, 1999; PISTORI, 2003), o autômato de pilha estruturado equivalente à nova estrutura sintática que está sendo definida. Este novo construto pode ser facilmente incorporado ao compilador através de ações adaptativas, em tempo de compilação, fazendo com que o reconhecedor da nova sintaxe seja incorporado ao da linguagem-

base e a nova construção sintática seja assim completamente integrada à linguagem definida.

Exemplo 5 *Definição da componente sintática para a linguagem EXTLA:*

prog = **BEGIN** *(decl* | *ext)* { *";" (decl* | *ext)* }
START { *firstnterm* } .
decl = **DECLARE identificador** *":"* (**VAR** | **LABEL**)
{ *","* **identificador** *":"* (**VAR** | **LABEL**) } .
ext = **DEFINE identificador** *":"*
NEW *oldnterm*
AS *wirth*
WHERE identificador IS *oldnterm*
{ *","* **identificador IS** *oldnterm* }
MEANING *oldwirth*
ENDDEFINE .
firstnterm = *programa* .
wirth = (*newnterm* | *term* | *"(" wirth ")"* | *"ε"*) (*"#"* **numero** | ε)
{ *"|"* (*newnterm* | *term* | *"(" wirth ")"* | *"ε"*)
(*"#"* **numero** | ε) } .
oldwirth = (*oldnterm* | *term* | *"(" oldwirth ")"* | *"ε"*) (*"#"* **numero** | ε)
{ *"|"* (*oldnterm* | *term* | *"(" oldwirth ")"* | *"ε"*)
(*"#"* **numero** | ε) } .
term = **identificador** | **numero** | *":"* | *"="* | *","* | *"<"* | *">"* |
"(" | *")"* | *"+"* | *"-"* | *";"* | *"#"* | *"ε"* | *"|"* |
LET | **GOTO** | **READ** | **PRINT** | **IF** | **THEN** |
BEGIN | **START** | **END** | **DECLARE** | **VAR** |
LABEL | **DEFINE** | **NEW** | **AS** | **WHERE** | **IS** |
MEANING | **ENDDEFINE** .
oldnterm = **nterm** | *newnterm* .
newnterm = $\emptyset$.

A linguagem EXTLA inclui seções de declarações de variáveis, rótulos e extensões sintáticas. A conexão com a linguagem-base é feita através da regra de produção associada ao não-terminal firstnterm. *O não-terminal* programa *corresponde à regra de produção inicial da linguagem MINLA, usada como linguagem-base associadao ao mecanismo de extensão definido. Os termos em negrito representam átomos gerados pelo respectivo analisador léxico e tomados como entrada pelo analisador sintático. Vale ressaltar que alterando-se a linguagem-base, deve-se adaptar o mecanismo de extensão à linguagem utilizada.*

4.4.4 Componente Semântica

Um programa em linguagem EXTLA é definido como um texto iniciado por *BEGIN*, seguido de uma seção de declarações de variáveis, rótulos e/ou extensões, separadas por ponto-e-vírgula, seguido de *START*, e depois de um programa em MINLA, que consiste de uma seqüência de comandos, cuja estrutura foi definida na Seção 3.3.2.

A seção de declaração, iniciada por *DECLARE*, é seguida por uma lista de elementos a serem declarados. Cada elemento compõe-se do identificador, seguido de dois pontos, e finalizado pelo tipo a ele associado (LABEL ou VAR). Os elementos são separados entre si por vírgulas. Além de variáveis e rótulos, a linguagem EXTLA permite a declaração de extensões sintáticas, através de um mecanismo de extensão que faz uso da estrutura da linguagem-base ou de construções sintáticas previamente definidas. A sintaxe dos comandos pertencentes ao mecanismo de extensão serão explicadas adiante.

Os comandos da linguagem-base foram definidos no Capítulo 3 e podem representar expressões envolvendo identificadores e inteiros, operações aritméticas básicas, estruturas condicionais simples e operações de entrada e saída. Cada uma das operações definidas na linguagem-base foi mapeada em uma seqüência de primitivas que atua diretamente sobre o modelo adaptativo que está sendo proposto.

Na seção seguinte, é definido o conjunto de comandos que pertecem a um mecanismo de extensão que será incorporado à linguagem-base para a descrição das novas construções sintáticas. Para cada uma de suas partes existe uma operação primitiva associada ao modelo dos autômatos adaptativos. Cada uma das primitivas será descrita nos mesmos moldes do que foi feito no capítulo anterior.

4.4.4.1 Declaração de Extensões

As declaração das novas construções sintáticas iniciam-se pela cláusula *DEFINE*, seguido do novo não-terminal a ser declarado e dois-pontos. A cláusula *NEW* se associa ao nome de uma classe sintática já definida a qual deve pertencer o novo não-terminal que está sendo definido. A cláusula *AS* se associa a uma expressão na notação de Wirth que define a nova sintaxe, seguida de *MEANING* para introduzir o significado, que também é expresso através de uma expressão na notação de Wirth, referenciando apenas terminais e não-terminais já definidos. As restrições impostas à classe de não-terminais à qual devem pertencer os identificadores utilizados na definição da nova sintaxe são declaradas através da cláusula *WHERE*, após a qual, entre vírgulas, uma lista de associações entre os identificadores e os correspondentes não-terminais já definidos

deve ser incluída. Estas associações são realizadas, mencionando-se o identificador, seguido de *IS* e de algum dos não-terminais já definidos. A declaração da extensão termina com ENDDEFINE.

Esses novos não-terminais são, inicialmente, reconhecidos como identificadores que, posteriormente, são incorporados à gramática como novos não-terminais que representam classes sintáticas correspondentes às novas formas sintáticas que estão sendo declaradas. Inicialmente o conjunto de novos não-terminais é vazio, e vai recebendo novos elementos à medida que novas sintaxes vão sendo declaradas.

As expressões na Notação de Wirth envolvem terminais e não-terminais originais da linguagem-base, bem como os novos não-terminais que já tenham sido previamente definidos pelo usuário, incluindo o novo não-terminal que estiver sendo definido no momento. Cada elemento da expressão pode, eventualmente, estar associado a um número inteiro, para fins de identificação. Isso pode ser feito através da sentença *#numero*, que permite a associação de valores na notação utilizada nas cláusulas *AS* e *MEANING*, com o objetivo de facilitar a referência aos respectivos elementos.

É importante observar que as expressões na Notação de Wirth associadas à cláusula *MEANING* devem se restringir ao uso de terminais, não-terminais originais da linguagem e os que já tenham sido previamente definidos pelo usuário, excluindo o que estiver sendo definido no momento. Obviamente, esta restrição deve ser imposta devido ao fato de que, para a definição corrente, o não-terminal que está sendo definido ainda não possui um significado completo associado.

4.5 O Ambiente de Execução e Interpretação para EXTLA

Da mesma forma que para a linguagem MINLA, definida no capítulo anterior, para o funcionamento adequado do mecanismo de extensão, e da linguagem EXTLA de forma geral, deve ser projetado um ambiente de execução compatível e capaz de realizar e processar todas as construções sintáticas e semânticas da linguagem definida. Portanto, cada um dos comandos de extensão deve ser corretamente compreendido e interpretado. Nesta seção, apresenta-se uma breve descrição dos efeitos de cada um dos comandos da linguagem. Tal descrição pode ser tomada como base para a definição de um conjunto de primitivas e, posteriormente, da biblioteca de funções semânticas e adaptativas que as implementem. Para exemplificar o uso do mecanismo de extensão, a seguir é apresentada a definição de um comando do tipo “WHILE” sobre a linguagem MINLA, que contém apenas “IF-THEN” e “GOTO”.

Exemplo 6 *Definição do comando "WHILE" usando a linguagem EXTLA:*

```
DEFINE comando_while:
NEW comando
AS
        WHILE boolexp DO comseq ENDWHILE ";"
WHERE
        boolexp IS expcomp,
        comseq IS comando
MEANING
        @loop ":"
        IF boolexp "=" "0" THEN GOTO @out ";"
                comseq
                GOTO @loop ";"
        @out ":"
ENDDEFINE
```

Em seguida, cada um dos passos para a definição do comando "*WHILE*" é detalhado para a compreensão adequada do uso da linguagem do mecanismo de extensão, inspirado no trabalho realizado em (SILVA; NETO, 2005).

4.5.1 Cláusulas *DEFINE* e *NEW*

No exemplo, a cláusula "*DEFINE*" marca o início da definição da nova extensão sintática. O identificador "**comando_while**" corresponde ao não-terminal que aparecerá à esquerda da regra de produção que descreverá a nova construção sintática.

A cláusula "*NEW*" vem acompanhada de um não-terminal que indica a qual das regras de produção existentes na gramática da linguagem-base será associado o novo não-terminal que acompanha a cláusula "*DEFINE*". De forma geral, suponha a definição:

```
DEFINE id:
NEW rule
...
```

O identificador "**id**" corresponde a um novo não-terminal que está sendo definido e "**rule**" corresponde a um não-terminal presente na linguagem. Portanto, deve exis-

tir uma regra de produção presente na instância corrente da gramática da linguagem, escrita na forma:

...
rule = $R_1|R_2|\ldots|R_n$
...

A esta regra, será associado o identificador definido na cláusula "*DEFINE*", da seguinte forma:

...
rule = $R_1|R_2|\ldots|R_n|$ **id**
...

Em nosso exemplo, as componentes R_1 até R_n correspondem às definições existentes (ou previamente definidas) para a regra de produção "*rule*". As componentes "**id**" e "**rule**" correspondem, respectivamente, ao novo não-terminal "**comando_while**", que está sendo definido, e ao não-terminal "**comando**", presente na gramática da linguagem-base, que está à esquerda da regra de produção, à qual será associado o novo não-terminal "**comando_while**".

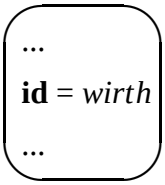
Portanto, a nova regra de produção para a gramática de nosso exemplo será:

```
...
comando  =  { identificador ":" }
            ( LET identificador ":" "=" exparit
            | GOTO identificador
            | READ ( identificador { "," identificador } )
            | PRINT ( exparit { "," exparit } )
            | IF expcomp THEN comando
            | ε
            | comando_while ) .
...
```

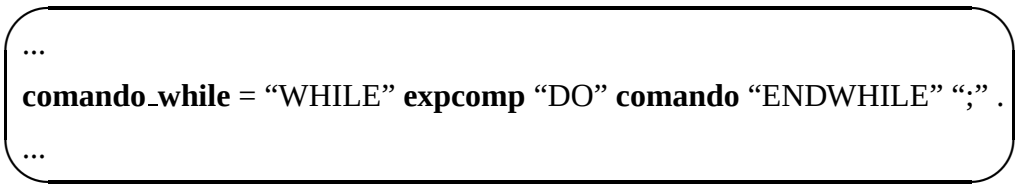
4.5.2 Cláusulas *AS* e *WHERE*

A cláusula "*AS*" descreve a estrutura sintática, em notação de Wirth, do novo não-terminal que está sendo definido, e a cláusula "*WHERE*" define as restrições impostas à classe de não-terminais à qual devem pertencer os identificadores utilizados na

definição da nova sintaxe. A combinação da descrição da estrutura sintática com as restrições impostas corresponde à regra que será gerada e associada ao identificador (não-terminal) da cláusula "*DEFINE*". Neste caso, deve ser gerada uma regra de produção na forma:



Em nosso exemplo, "**id**" corresponde ao não-terminal "**comando_while**" e *wirth* corresponde à descrição sintática da cláusula "*AS*". Portanto, além da regra de produção, cuja modificação foi definida na seção anterior, será criada a seguinte nova regra de produção:



Observe que, com a inclusão desta nova regra de produção, novos terminais devem ser inseridos de forma adequada, como palavras reservadas da linguagem, no autômato que reconhece os *tokens* léxicos da linguagem. Dessa forma, "*WHILE*", "*DO*" e "*ENDWHILE*" devem ser incluídos no analisador léxico e devidamente identificados como novas palavras reservadas da linguagem. A inclusão destes *tokens* ocorre através da inclusão de chamadas de funções adaptativas parametrizadas que verificam a existência dos *tokens* correspondentes e, caso não estejam definidos, provocam a inserção de um novo "trecho" do autômato que os reconhece. O *token* ";" já pertence à linguagem e, dessa forma, não precisa ser incluído.

4.5.3 Cláusula *MEANING* e *ENDDEFINE*

A cláusula "*MEANING*" introduz o significado do novo comando que está sendo definido. O significado também é expresso através de uma expressão na notação de Wirth, referenciando apenas não-terminais definidos na cláusula "*WHERE*" ou já presentes na gramática. Se necessário, rótulos auxiliares são declarados para a representação do significado. Isso pode ser feito através da cadeia *@numero*, que pode ser utilizado apenas em cláusulas "*MEANING*", permitindo que o usuário crie identificadores únicos que podem ser associados a trechos de código contidos na declaração das extensões. Estes identificadores são instanciados todas as vezes em que uma extensão é

invocada. Dessa forma, cada chamada da extensão, mesmo em chamadas recursivas, gera novas instâncias desses identificadores. A cláusula "*ENDDEFINE*" marca o final da definição da nova extensão sintática.

4.6 Considerações Sobre a Implementação de Extensibilidade

A implementação do mecanismo de extensão, como está sendo proposto, passa pela definição e implementação adequada de todos os aspectos da componente estática da sintaxe de uma linguagem-base, em termos de autômatos adaptativos. Em seguida, a componente dinâmica, representada pelo mecanismo de extensão, exige a definição de novas construções sintáticas que podem ser tratadas, separadamente, pela camada adaptativa, antes mesmo de seu processamento. Neste caso, é utilizada uma estratégia semelhante à expansão de macros, tornando o processo de extensão mais expressivo. Conceitualmente, existe a possibilidade de se utilizar o recurso que permite aos autômatos adaptativos alterar a cadeia de entrada quando necessário. Para isso, pode-se considerar a expansão de uma macro, detectando-se inicialmente sua chamada (que pode corresponde a uma chamada de uma ação adaptativa), e inserindo-se em seu lugar, na cadeia de entrada, um texto correspondente à expansão da macro, retornando-se então à análise do novo texto como se a parte correspondente à chamada da macro não tivesse existido.

Este capítulo descreveu aspectos do uso e da aplicação dos autômatos adaptativos como uma forma natural para o processamento de linguagens extensíveis, partindo de uma linguagem-base imperativa definida sobre o formalismo adaptativo. O capítulo seguinte discute alguns aspectos práticos relacionados com a implementação desta estratégia, envolvendo a codificação de programas adaptativos.

5 Ambiente para Codificação de Programas Adaptativos

Este capítulo apresenta um panorama geral das estratégias e mecanismos para a implementação de uma linguagem para a codificação de programas adaptativos e de um ambiente ofereça suporte para isso. Inicialmente, discute-se o estado da arte sobre o assunto, referenciando resultados apresentados em (FREITAS; NETO, 2007b) e (PELEGRINI; NETO, 2008). Em seguida, apresenta-se uma breve descrição sobre uma das estratégias de implementação de um ambiente de interpretação e execução de códigos representados na forma de autômatos adaptativos, aderente às propostas formuladas nos capítulos anteriores.

5.1 Linguagens para Codificação de Programas Adaptativos: Estado da Arte

O processamento de uma linguagem programação para codificação de programas adaptativos exige o desenvolvimento de um ambiente composto por um mecanismo de processamento de uma linguagem-base imperativa, que sirva de núcleo, e de um módulo de controle, representado pela máquina adaptativa, cuja responsabilidade será a de gerenciar a execução da parte auto-modificável dos códigos escritos nessa linguagem. Os comandos que realizam a auto-modificação de código podem ser concebidos diretamente ou gerados por extensões da linguagem-base, considerando a existência de uma biblioteca de funções capazes de alterar a própria estrutura do código em execução.

Para isso, também é necessário um ambiente de interpretação e execução que seja capaz de tratar tais funções. Conceitualmente, um programa adaptativo é formado pelo espaço de códigos $CF_1, CF_2, \ldots, CF_n$ de tal modo que, a partir do código inicial CF_1 e por meio da execução das funções de auto-modificação de código previamente definidas, o programa evolua para as sucessivas configurações $CF_2, CF_3, \ldots, CF_n$, à medida que a execução for sendo processada (FREITAS; NETO, 2006c; FREITAS; NETO, 2005; FREITAS; NETO, 2007b).

O projeto e a implementação de um ambiente com este propósito exige a utilização de um ambiente de *run-time* que possa tratar a biblioteca de operadores de auto-modificação de forma simples e adequada. As chamadas paramétricas de funções adaptativas (ou seja, as ações adaptativas) irão propiciar as características de auto-modificação do código, neste caso, escrito originalmente em alguma linguagem imperativa de programação. Essas funções são constituídas de chamadas dos operadores elementares, disponíveis na camada adaptativa, os quais, acionados através de ações adaptativas elementares de consulta, inclusão e remoção de elementos do programa, proporcionam os efeitos da adaptatividade sobre o programa auto-modificável em execução. As funções adaptativas são definidas por meio da composição dessas ações elementares e correspondem às regras que, em tempo de execução, promovem as devidas mudanças no código (adaptativo) do programa.

Obviamente, existem diversos problemas e aspectos a serem formalizados e tratados para a implementação completa de uma linguagem com essa característica. Um dos principais é a definição de um ambiente de interpretação e execução que seja totalmente baseado no funcionamento dos autômatos adaptativos. Uma das estratégias adicionais se relaciona com o mecanismo utilizado pelas funções adaptativas para que as mesmas possam endereçar os trechos de código que compõem um código objeto em execução. Esse mecanismo é de fundamental importância para que uma ação adaptativa possa especificar a eliminação ou inserção de uma transição num autômato adaptativo, possa converter essa referência em cada uma das transições ou estados que mapeiam o autômato original correspondente ao código objeto em execução.

Adaptatividade por modificação de código tem recebido um interesse recente na literatura acadêmica, principalmente na área de prevenção de engenharia reversa indesejada (ANCKAERT; MADOU; DE BOSSCHERE, 2007). Embora a engenharia reversa tenha como objetivo recriar o projeto de software por meio da análise do produto final, ela pode também ser empregada para fins maléficos. Código auto-modificável é adequado para estas aplicações uma vez que é a sua presença dificulta a ação da engenharia reversa (CIFUENTES; GOUGH, 1995). Além da área de proteção de software, adaptatividade por modificação de código também pode ser empregada em dispositivos que utilizam regras dinâmicas tais como aprendizagem computacional, inferência, jogos, reconhecimento de padrões, processamento de imagens, compactação de arquivos, entre outros.

Códigos auto-modificáveis, especialmente os empregados em linguagens de baixo nível, são usualmente difíceis de serem escritos, documentados e mantidos (MCKINLEY et al., 2004). Esta proposta, no entanto, está centrada no emprego da tecnologia adaptativa, a qual utiliza para isso funções adaptativas. Estas devem ser concebidas por meio

de regras que podem, se bem projetadas, assegurar maior usabilidade ao mecanismo proposto.

Obviamente, nas técnicas adaptativas, há o perigo de a auto-modificação de código gerar uma explosão de espaço de código e de tempo, portanto convém impor disciplinas sobre as operações executadas pelas ações adaptativas para que as alterações delas decorrentes possam ser previsíveis e controladas. Para isso, como sugestão de trabalhos futuros é desejável que um estudo e formalização do crescimento de código seja desenvolvido para quantificar o tempo e o espaço do código produzido pelas sucessivas aplicações das ações adaptativas. Essas limitações podem ser, em princípio, estudadas com mais profundidade entre as propriedades dos formalismos adaptativos, juntamente com os aspectos de determinismo e não-determinismo, apresentados no Capítulo 2. Com isso, poderão ser realizados estudos analíticos sobre o crescimento de código e de tempo para a definição dos métodos de projeto de funções adaptativas.

Para auxiliar o projeto das funções adaptativas, é recomendado que o ambiente adaptativo disponibilize ao projetista ferramentas de depuração do código dinamicamente alterado neste tipo de programas, ferramentas essas que deverão efetuar todo o controle das regiões alteráveis do código, indicando, de forma também dinâmica, as partes do programa sujeitas à adaptatividade, aquelas que já foram modificadas, a modificação em curso, entre outras. Para simplificar o projeto deste ambiente, como ambiente de execução será utilizado o AdapTools.

5.2 Proposta de um Ambiente de Execução

Uma das alternativas para o processamento adequado e confortável de linguagens para a codificação de programas auto-modificáveis passa pelo desenvolvimento de um ambiente completo, totalmente fundamentado nos autômatos adaptativos. O objetivo é mostrar a viabilidade de representar e operar, empregando autômatos adaptativos, os elementos empregados para a representação de estruturas de dados e mecanismos de controle nas computações usuais.

Para isso, no Capítulo 3 foi introduzida uma proposta de formas de representação de constantes e variáveis, bem como para as operações usuais de atribuições de valor, operações aritméticas e lógicas, seqüenciamentos, comparações, desvios condicionais, entre outros.

O projeto e a implementação deste ambiente exige a definição da estrutura básica da máquina de execução de programas, totalmente baseada nos autômatos adaptativos. Em princípio, podem ser implementadas as seguintes rotinas:

- Carga do programa para a memória da máquina, cujo segmento de código pode ser representado por um estado do autômato que aponta para o estado correspondente à primeira instrução do respectivo código;

- Busca da próxima instrução a ser executada no segmento de código, seguida de sua decodificação e execução.

Como já foi dito anteriormente, a escolha ou a implementação de um ambiente de execução adequado é muito vantajosa do ponto de vista experimental, uma vez que os comandos para a auto-modificação de código serão tratados confortavelmente e sem a necessidade de grandes esforços para a adequação do ambiente de execução. Portanto, apesar de não refletir completamente o modelo teórico, o AdapTools (PISTORI, 2003) se apresenta como uma ferramenta muito útil para ser utilizada como o ambiente de projeto e de execução em todas as etapas desta proposta. Além disso, o uso e a aplicação desta ferramenta pode conduzir à detecção precisa de suas limitações no tratamento do modelo de computação baseado no autômato adaptativo, bem como na implementação ou propostas de soluções para as limitações detectadas.

O Anexo C mostra a tabela correspondente ao código AdapTools para a máquina de execução baseada no autômato adaptativo. As funções do ambiente de execução estão implementadas através das ações adaptativas `initProg`, `nextip`, `restaura` e `continue`, cujas linhas de código são explicadas na seção seguinte.

5.2.1 A Arquitetura do Ambiente Definida no AdapTools

Para dar suporte ao processo de execução de programas de código adaptativo auto-modificável, pode-se pensar na proposta de um modelo para uma arquitetura totalmente baseada nos autômatos adaptativos. Tal modelo pode ser projetado de forma que a execução de cada instrução esteja associada a efeitos que podem ser simulados por ações adaptativas associadas. Para tanto, existem duas estratégias possíveis:

- Partindo de uma linguagem extensível de programação L_{ext} e de sua formalização baseada nos autômatos adaptativos, pode-se definir novas construções sintáticas capazes de incorporar, aos programas em execução, a capacidade de auto-modificação, considerando a definição completa da linguagem extensível sobre um ambiente de execução totalmente baseado nos autômatos adaptativos e a existência de funções de biblioteca que possam tratar as operações de inserção e remoção de código.

- Partindo de uma linguagem imperativa de programação (não-extensível) L_{bas} e de sua formalização baseada nos autômatos adaptativos e considerando a definição completa de um ambiente de execução totalmente baseado nos autômatos adaptativos e a existência de funções de biblioteca que possam tratar as operações de inserção e remoção de código, pode-se implementar programas auto-modificáveis de forma confortável, uma vez que o próprio ambiente já possui a capacidade de incorporar a aplicação de mudanças sobre o código dos programas em execução.

O processo de obtenção de programas auto-modificáveis parte de uma estrutura previamente desenvolvida para dar suporte a uma dada linguagem de programação e de uma biblioteca de tratamento dos aspectos de adaptatividade, que interpretam os comandos de auto-modificação, promovendo as modificações no código-fonte em tempo de execução.

Observe-se que, se considerarmos um ambiente de execução totalmente desenvolvido na forma de autômatos adaptativos, a biblioteca de tratamento dos construtos adaptativos pode fazer uso dos próprios recursos do ambiente de execução e, dessa forma, o trabalho de pesquisa pode assim se restringir ao estudo das disciplinas de projeto, bem como de métodos e técnicas de desenvolvimento de programas adaptativos.

A Figura 5.1 mostra o autômato que representa uma proposta de modelo para a máquina de execução adaptativa.

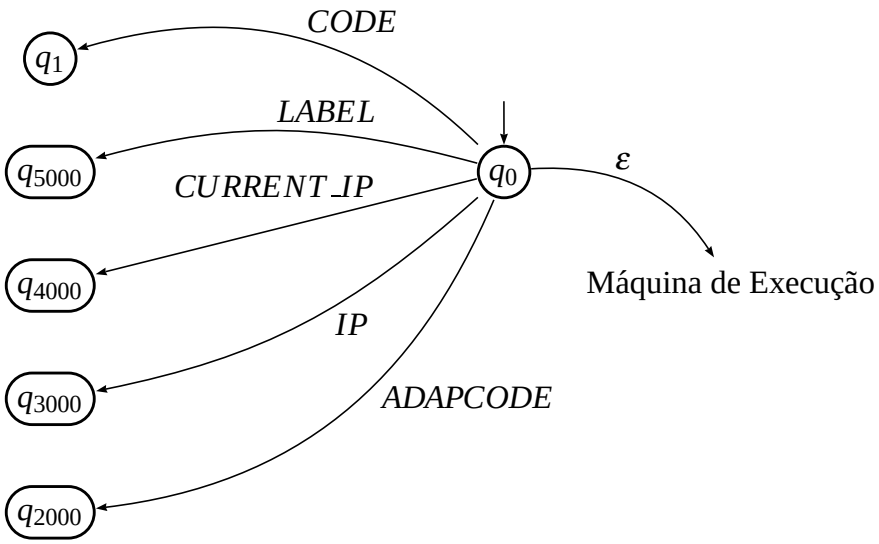


Figura 5.1: Modelo proposto como arquitetura simplificada para a execução de programas no AdapTools.

No modelo proposto, o estado q_0 é o estado inicial, ao qual estão ligados todos os estados que possuem alguma função específica dentro da máquina de execução. Esses estados são identificados pelo rótulos das transições que incidem sobre cada um deles. Tais rótulos são utilizados para facilitar a implementação das ações adaptativas elementares de consulta que, eventualmente, podem estar contidas na biblioteca de ações adaptativas implementadas. A Tabela 5.2.1 descreve a função de cada um destes estados.

Estado	Rótulo da transição	Função
q_1	CODE	Aponta o estado ao qual se associa o início do autômato que representa o código do programa a ser executado, gerado por um interpretador. Esse código é representado por um autômato cujas transições correspondem cada qual a um átomo do programa.
q_{2000}	ADAPCODE	Aponta o estado ao qual será associado o construto adaptativo que representa o comando para o qual foi gerado a representação adaptativa.
q_{3000}	IP	Aponta o estado que corresponde ao início da parte do autômato associado à próxima instrução a ser executada (interpretada).
q_{4000}	CURRENT_IP	Aponta o estado que corresponde ao início da parte do autômato associado à instrução corrente (em execução).
q_{5000}	LABEL	Aponta os estados do autômato associados aos rótulos definidos e utilizados pelo programa.

Tabela 5.1: Tabela que descreve os estados e transições que indentificam os componentes do ambiente de execução definido no AdapTools.

Na seção seguinte, são mostradas as representações gráficas do núcleo da máquina de execução e das funções adaptativas responsáveis pelas tarefas auxiliares do sistema que promovem a interpretação e execução de um programa carregado neste ambiente.

5.2.2 Núcleo da Máquina de Execução

A outra componente do modelo proposto é a máquina de execução, que é responsável pela carga e pela execução de cada uma das instruções do programa. A Figura 5.2 mostra a representação gráfica do núcleo da máquina de execução e os estados auxiliares, utilizados como referência para o conjunto de informações utilizadas durante a execução de um programa.

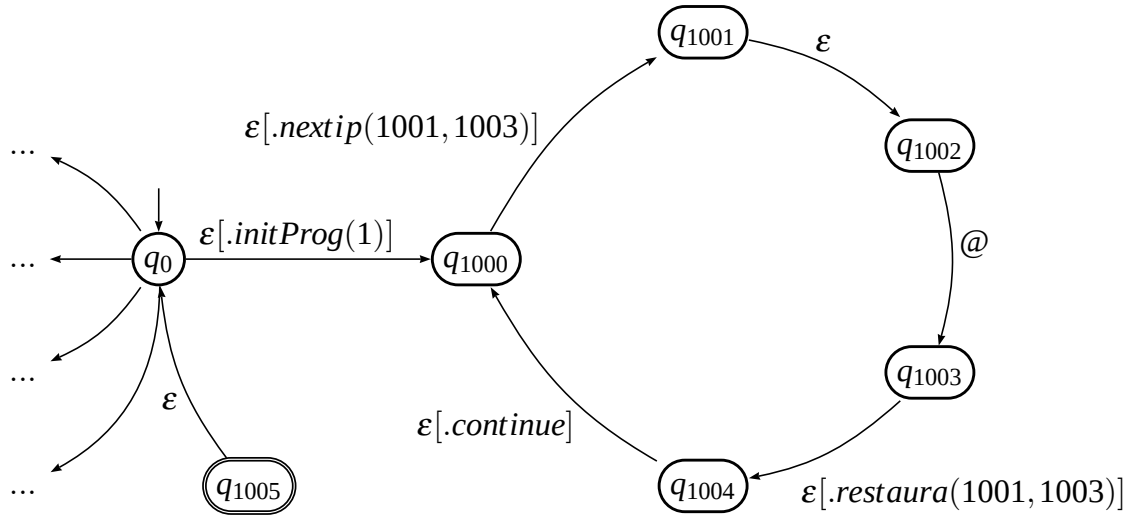


Figura 5.2: Representação gráfica do trecho do autômato adaptativo que corresponde à máquina de execução, responsável pela interpretação dos comandos da linguagem.

Da forma como está sendo proposta, a máquina de execução permite a definição de novos estados e transições que podem identificar outros componentes do ambiente, tais como a tabela de símbolos, as constantes e outros, que venham a ser definidos posteriormente.

5.2.3 A Função Adaptativa *initProg*

A função adaptativa *initProg* é responsável pela inicialização da máquina de execução. Ela recebe como parâmetro o estado correspondente ao início do programa gerado pelo interpretador que converte o programa escrito em MINLA em um autômato adaptativo o qual traduz a sequência de átomos encontrados no programa. A mesma função adaptativa pode ser modificada de modo que fique responsável pela carga do programa intermediário, gerado para ser interpretado e executado nesta máquina de execução. A Figura 5.3 mostra a representação gráfica da função *initProg*.

O novo estado gerado $*n$ corresponde ao estado inicial do construto adaptativo que representa o código gerado após o processamento (transdução) de todo o programa.

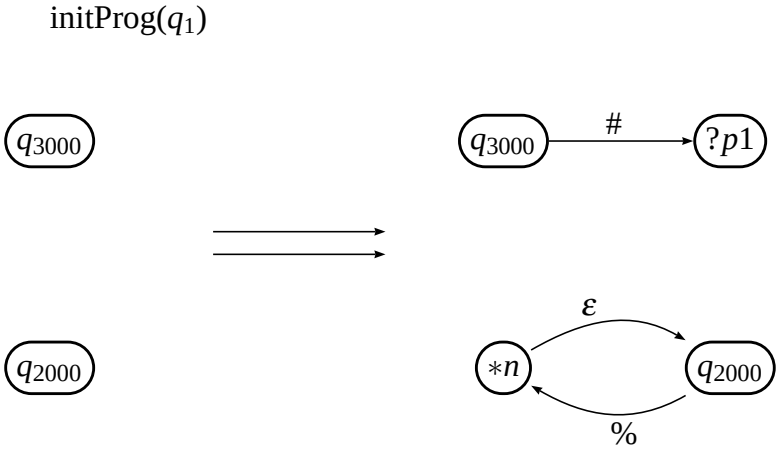


Figura 5.3: Representação gráfica da função adaptativa *initProg*.

Os símbolos # e % marcam, respectivamente, o estado corrente do programa a ser transduzido e o estado corrente do programa-objeto que está sendo gerado.

5.2.4 A Função Adaptativa *nextip*

A função adaptativa *nextip* é responsável pela carga do trecho de código a ser transduzido para o ciclo de execução. Ela recebe como parâmetro os dois estados que marcam o trecho onde será introduzida a instrução corrente. Para isso, existe uma transição marcada com o símbolo @, que serve como referência para o local onde será carregada a respectiva instrução. A Figura 5.4 mostra a representação gráfica da função adaptativa *nextip*.

A transição $[?c, ?s, ?x]$ corresponde à instrução corrente a ser executada (interpretada). Para isso, ela deve ser carregada dentro do trecho da máquina que é responsável pela realização do ciclo de execução da respectiva instrução. Este trecho é marcado pela transição $[?a, @, q_{1003}]$, compreendido entre os estados q_{1001} e q_{1003}. Após a execução da função *initProg*, os estados q_{1001} e q_{1003} passam a compreender a instrução corrente. Além disso, o estado q_{3000} passa a apontar para o "endereço" (estado) da próxima instrução.

Logo após a execução da instrução, a máquina de execução deve ser restaurada, ficando pronta para a busca e execução da próxima instrução do programa. Existem duas funções adaptativas que são responsáveis por esta tarefa: a função *restaura* e a função *continue*. Ambas são descritas na seção seguinte.

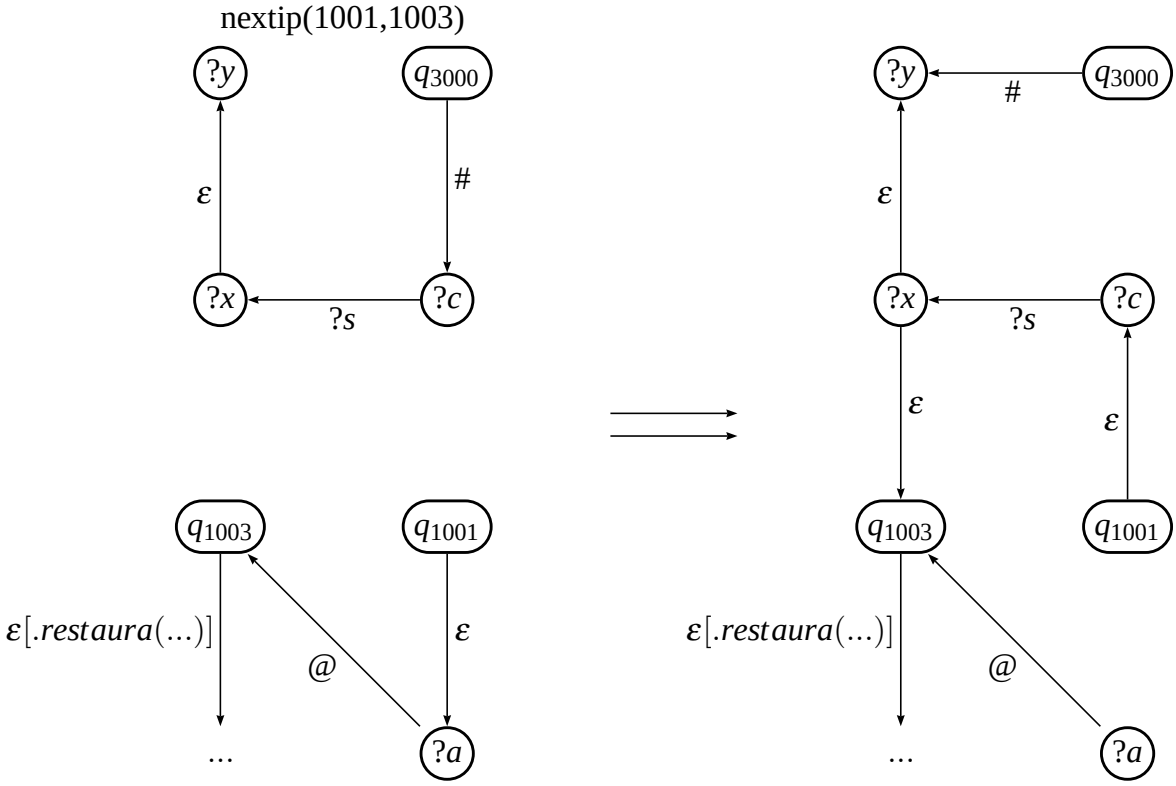


Figura 5.4: Representação gráfica da função adaptativa “nextip”.

5.2.5 As Funções Adaptativas *restaura* e *continue*

As funções adaptativas *restaura* e *continue* são responsáveis pela restauração do ambiente e pela respectiva adaptação da configuração da máquina para a execução da próxima instrução. A função adaptativa *continue* também descarta as transições correspondentes a instrução que acabou de ser executada.

A função adaptativa *restaura* recebe como parâmetros os estados que identificam a transição que representa a instrução que acabou de ser executada. Em seguida, ela retira a respectiva instrução do ciclo de execução, restaurando a transição identificada pelo símbolo @. Uma vez restaurada, o ciclo de execução prossegue com a execução da função adaptativa *continue*.

A função adaptativa *continue* descarta a transição que representa a instrução que foi executada, desconectando os seus estados de origem e destino do autômato e organiza o autômato para a busca da próxima instrução. Em seguida, a função adaptativa *nextip*, definida na Seção 5.4, é executada novamente e o ciclo de execução prossegue até que todas a transição que representa a instrução END seja executada.

6 Conclusões

Este capítulo descreve as contribuições teóricas e metodológicas do trabalho de pesquisa desenvolvido nesta tese, com ênfase no método para o projeto e implementação de linguagens de programação, utilizando o autômato adaptativo como modelo formal para a sua especificação e de seu ambiente de interpretação e execução. São apresentadas algumas considerações sobre a codificação de programas adaptativos, bem como as principais contribuições deste trabalho de pesquisa e seu uso e aplicação como subsídio para prosseguimento da pesquisas nesta área.

6.1 Contribuições

Uma das metas desta tese foi o de apresentar e discutir os avanços sobre os aspectos relacionados com o projeto de linguagens para codificação de programas de código adaptativo e, considerando as características dos autômatos adaptativos, executar um modelo capaz de tratar de forma adequada e confortável os aspectos de extensibilidade sintática e de auto-modificação de código, em tempo de execução.

Para isso, foi definida uma linguagem imperativa simplificada de programação e apresentado um conjunto de primitivas para o mapeamento de suas instruções em efeitos sobre a representação na forma de construtos adaptativos. Com isso, foi possível o desenvolvimento de uma proposta de criação de um ambiente de execução totalmente baseado e que possa ser facilmente mapeado em tal modelo. Foi proposta uma linguagem para a definição de extensões, partindo da linguagem base previamente definida e passando pela definição de um mecanismo de extensão associado. Para esta linguagem extensível, também foi conceitualmente descrito um modelo de execução e interpretação totalmente baseado nos autômatos adaptativos, tratando de forma bastante confortável os aspectos de incorporação das extensões definidas pelo usuário. Todo o processo descrito neste trabalho serve de base para um método bem definido para a obtenção de uma linguagem de programação para a produção de programas adaptativos.

As etapas abaixo resumem, de forma simplificada, a visão geral do método pro-

posto neste trabalho:

1. Definição de uma linguagem-base imperativa L_{bas}.
   (a) Construção de um compilador para L_{bas}.
   (b) Definição do ambiente de execução para programas em L_{bas}.
2. Definição de uma linguagem extensível L_{ext}.
   (a) Definição da gramática do mecanismo de extensão M.
   (b) Combinação de L_{bas} e M para obtenção da linguagem extensível L_{ext}.
   (c) Construção de um compilador para L_{ext}.
   (d) Adequação do ambiente de execução de L_{bas} para torná-lo aderente à L_{ext}.
3. Definição de novas construções sintáticas que simulam as operações elementares de um formalismo adaptativo sobre um programa em execução.
   (a) Criação de uma biblioteca para tratamento dos comandos adaptativos de inserção e remoção de linhas de código.

A ferramenta AdapTools foi utilizada para a realização de pequenos ensaios e simulações do ambiente operacional proposto. Entretanto, uma implementação completa exige o desenvolvimento de uma biblioteca mais ampla de rotinas semânticas (em Java) ou de funções adaptativas que possam ser tratadas diretamente pela ferramenta. Esta biblioteca pode partir da implementação das primitivas propostas neste trabalho e da descrição conceitual do comportamente de cada uma das instruções das linguagens definidas.

No campo teórico, realizou-se um estudo sobre os aspectos de determinismo e não-determinismo em autômatos de estados finitos adaptativos (CASTRO JR.; NETO; PISTORI, 2007). Tais resultados poderão ser utilizados como base para a definição de métodos de escrita da camada adaptativa que envolve o projeto de linguagens para a codificação de programas aderentes ao modelo dos autômatos adaptativos. A intenção é a de impor disciplina ao projeto das funções adaptativas que virão associadas aos comandos da linguagem de programação projetadas e implementadas sob a ótica adaptativa, evitando a falta de controle sobre o espaço de memória utilizado, o aumento exagerado da complexidade de tempo de processamento, bem como outros efeitos provenientes das diferentes interpretações e formas de tratamento das componentes presentes em um autômato adaptativo (PISTORI, 2003).

A utilização do AdapTools, mesmo que apenas para simples simulações, vem proporcionando avanços e aperfeiçoamentos na ferramenta, promovendo a otimização da máquina virtual do AdapTools e a implementação de novas funcionalidades, tais como o tratamento de não-determinismos e das rotinas semânticas (JESUS et al., 2007).

A linguagem *MINLA* foi apresentada como linguagem-base imperativa para a componente experimental desta tese. A obtenção do reconhecedor sintático desta linguagem foi realizada com a aplicação de metacompilador Wirth2APE proposto em (NETO; PARIENTE; LEONARDI, 1999) e implementado no AdapTools por (PISTORI, 2003). A Figura 6.1 representa o esquema de obtenção do reconhecedor sintático da MINLA a partir de sua definição em notação de Wirth e usando como ferramenta para geração do reconhecedor apropriado o Wirth2APE implementado no AdapTools.

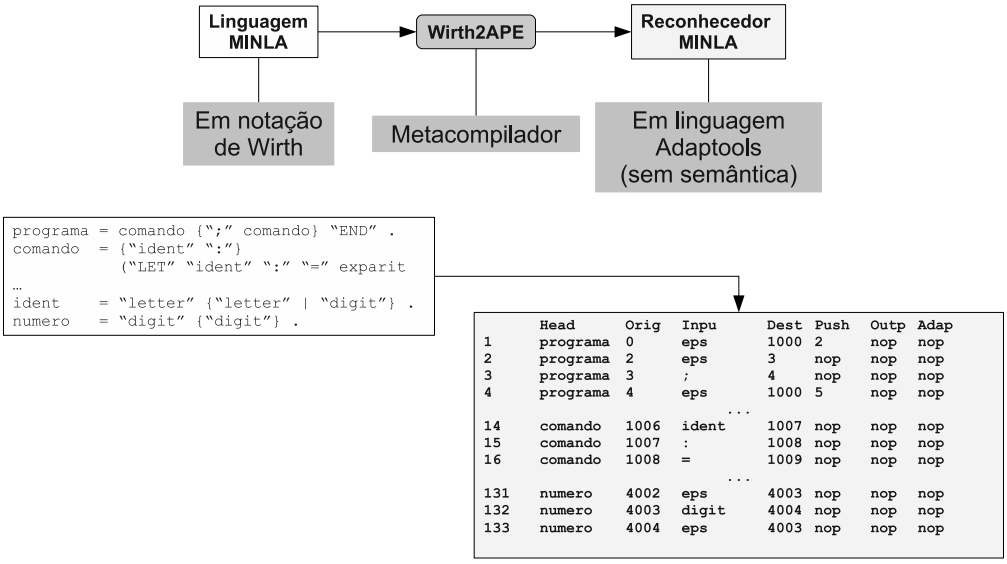


Figura 6.1: Obtenção do reconhecedor sintático da MINLA a partir de sua gramática, usando a implementação AdapTools do Wirth2APE.

Ao reconhecedor sintático obtido pelo processo esquematizado na Figura 6.1 é associada uma biblioteca de ações semânticas. No AdapTools, estas ações semânticas são inseridas na coluna apropriada da tabela do AdapTools que representa o reconhecedor obtida na etapa anterior. A inserção das ações semânticas, implementadas em JAVA, transformam o reconhecedor obtido em um transdutor para a linguagem MINLA ou, simplesmente, transdutor MINLA. O processo de inserção das ações semânticas e obtenção do transdutor MINLA é representado na Figura 6.2.

O transdutor MINLA, obtido pelo processo descrito nas Figuras 6.1 e 6.2, é utilizado, por sua vez, para a geração de uma máquina intermediária, denominada *autômato de transdução*. A obtenção do autômato de transdução é realizada a partir de um programa em MINLA e está representada na Figura 6.3.

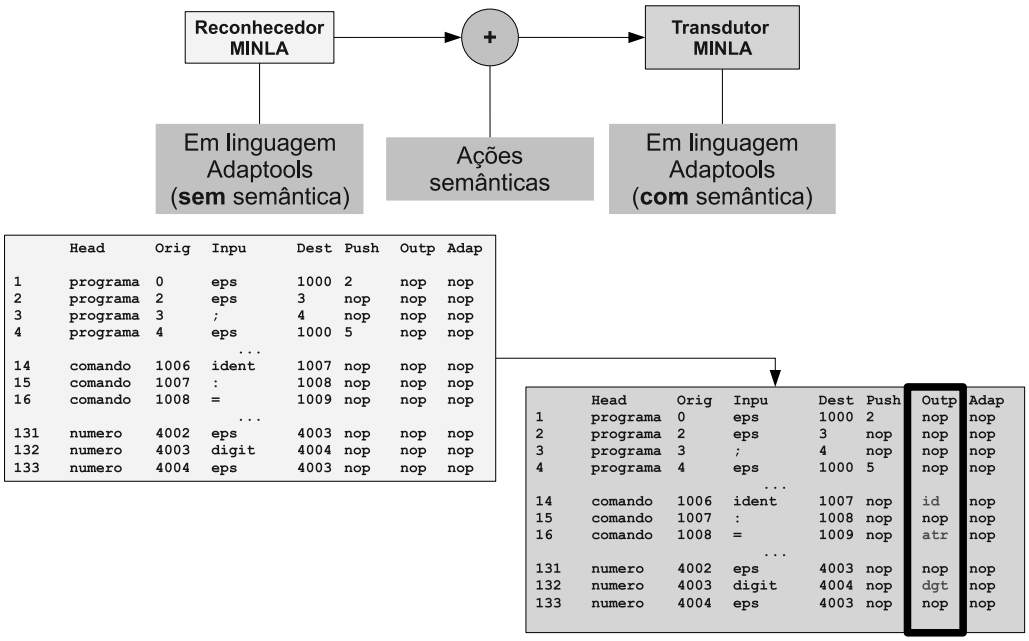


Figura 6.2: Inclusão das ações semânticas no reconhecedor obtido na etapa descrita pela Figura 6.1 para obtenção de um transdutor para a linguagem MINLA.

A finalidade do autômato de transdução é, partindo de um programa escrito em MINLA, produzir um autômato de execução para o respectivo programa em MINLA. O autômato de execução é, na verdade, uma seqüência de submáquinas que correspondem a seqüência de comandos do programa em MINLA representados na forma de um autômato adaptativo. A forma geral de um autômato de execução é apresentada na Figura 6.4. Portanto, tanto o autômato de transdução quando o de execução são obtidos a partir de um programa em MINLA. Dessa forma, para cada programa na linguagem MINLA é obtido um autômato de transdução diferente. O mesmo vale para o autômato de execução.

6.2 Trabalhos Futuros

Durante o desenvolvimento desta tese, constatou-se que o desenvolvimento de uma linguagem para produção de código adaptativo introduz uma nova forma de pensar o programa, a exemplo do que ocorre com a programação paralela e com a programação orientada a objeto. Para ser completamente formalizada, esta nova forma de pensar deve ser estudada de forma mais ampla. Tal estudo, envolve, entre outros aspectos a proposta de métodos e técnicas para disciplinar a escrita de programas adaptativos, uma vez que o comportamento do programa em execução está diretamente relacionado à execução de ações adaptativas, cuja operação define o comportamento dinâmico do mesmo.

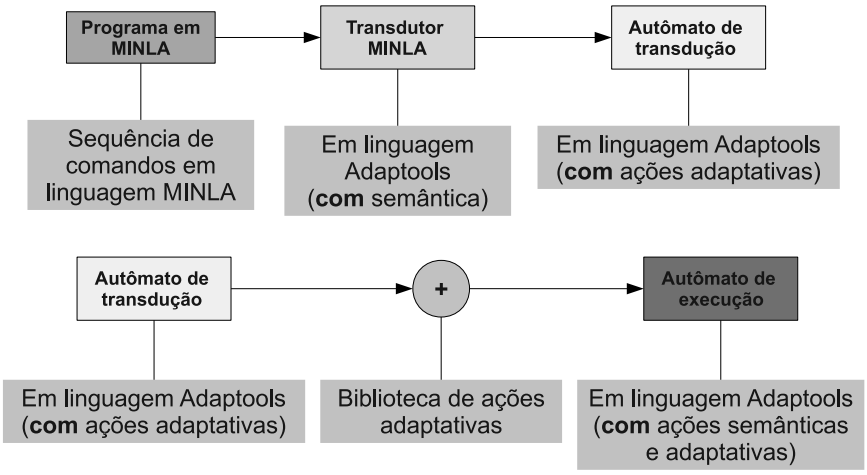


Figura 6.3: Obtenção do autômato de transdução e de execução para um programa escrito em MINLA.

Implementação da biblioteca de funções adaptativas para o tratamento dos aspectos de extensibilidade dentro do próprio ambiente do AdapTools. Esta biblioteca pode ser associada ao ambiente de execução proposto neste trabalho para viabilizar a simulação destes aspectos.

As considerações realizadas neste trabalho sobre o método para o projeto e a implementação de linguagens de programação que possam gerar código adaptativo sugere, ainda, como continuidade deste trabalho, a implementação completa do ambiente operacional baseado nos autômatos adaptativos. Este ambiente proporcionaria a simulação completa do ambiente proposto e, consequentemente, a melhoria e otimização do conjunto de primitivas da linguagem.

Além disso, pode-se pensar no desenvolvimento de um trabalho de pesquisa que possa realizar uma análise detalhada da complexidade de tempo e espaço do ambiente proposto, servindo de base para o desenvolvimento de ferramentas aplicações adaptativas para problemas complexos, começando pelos problemas apresentados em (NETO, 2000).

De forma objetiva, sugere-se, como trabalhos futuros, um estudo aprofundado dos seguintes tópicos:

- Descrição completa de um ambiente de execução totalmente baseado nos autômatos adaptativos.
- Implementação da biblioteca para ensaio e tratamento do mecanismo de extensão como suporte para o tratamento de uma linguagem extensível de programação, usando o ambiente de execução proposto;

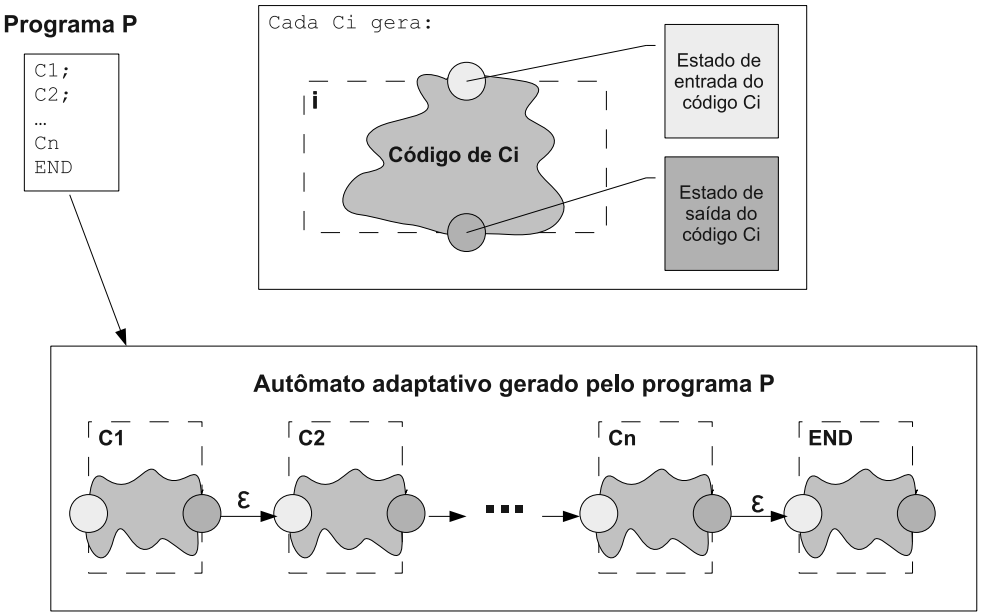


Figura 6.4: Forma geral de um autômato de execução para um programa escrito em MINLA.

- Implementação completa de uma linguagem adaptativa de programação, usando o ambiente de execução proposto;
- Refinamento da biblioteca de primitivas, definidas no Capítulo 3;
- Descrição completa de uma máquina virtual para execução de código adaptativo, baseada no AdapTools e nas propostas de implementação descritas nesta pesquisa;
- Refinamento e amplicação da biblioteca de ações semânticas e funções adaptativas definidas neste trabalho de pesquisa que possam dar um suporte completo a todas as etapas de tratamento de linguagens;
- Proposta de métodos e técnicas para desenvolvimento de dispositivos adaptativos bem comportados;
- Elevação do nível de abstração da interface com ambientes de simulação de autômatos adaptativos, disponibilizando uma versão de uso mais simplificado;
- Validação de propostas teóricas de soluções para problemas complexos baseadas em autômatos adaptativos, usando o ambiente proposto;
- Difusão da adaptatividade como ótica para o desenvolvimento de programas (linguagem) e para a interpretação dos mesmos (ambiente de execução);

- Desenvolvimento de um ambiente de depuração de código adaptativo que pode ser obtido, em princípio, através da engenharia reversa para conversão de “templates” de autômatos em trechos de código.

Anexo A – Definição do Conjunto de Primitivas da Linguagem

Aqui está definido o conjunto de primitivas para o ambiente de execução. O termo **construto** é utilizado para identificar qualquer elemento da linguagem representado, exclusivamente, na forma de um autômato adaptativo (nome, inteiro ou outros). Tais elementos aparecem, na notação, precedidos pelo sinal ?. Os demais parâmetros correspondem a elementos cuja representação não é definida na forma de um construto.

Primitivas	Interpretação
eval(?o, v)	Cria o construto adaptativo que representa o valor *v* e o associa ao construto *o*.
def(?o, n)	Cria/busca o construto adaptativo ?*o* que representa a variável (nome) *n*.
defl(?o, n)	Cria/busca o construto adaptativo ?*o*, que representa o rótulo *n*, e o associa ao trecho de código correspondente.
val(?o, n)	Busca o construto adaptativo que representa o valor associado ao identificador *n* e o associa ao construto *o*.
succ(?o)	Cria o construto adaptativo que representa o sucessor de ?*o*.
pred(?o)	Cria o construto adaptativo que representa o antecessor de ?*o*.
repl(?o,?o_1)	Cria o construto adaptativo que corresponde à cópia do construto ?o_1 e o associa ao construto ?*o*.
conn(?o_1, ?o_2)	Cria um construto adaptativo que representa a conexão entre os estados de saída e de entrada dos construtos ?o_1 e ?o_2, respectivamente, e o associa a o_1.

add(?o, ?o_1, ?o_2)	Cria um construto adaptativo que representa o resultado da soma dos construtos ?o_1 e ?o_2 e o associa ao construto ?o_1.
neg(?o)	Cria um construto adaptativo que representa o construto ?*o* com sinal invertido.
cmpig(?o, ?o_1, ?o_2)	Cria o construto lógico resultante da operação de comparação $>$ entre os construtos ?o_1 e ?o_2 e associa o resultado ao construto *o*.
cmpil(?o, ?o_1, ?o_2)	Cria o construto lógico resultante da operação de comparação de $<$ entre os construtos ?o_1 e ?o_2 e associa o resultado ao construto *o*.
cmpie(?o, ?o_1, ?o_2)	Cria o construto lógico resultante da operação de comparação de $=$ entre os construtos ?o_1 e ?o_2 e associa o resultado ao construto *o*.
jcond(?o_1, ?o_2, ?o_3)	Avalia o valor lógico do construto ?o_1. Se for *true*, cria uma transição do estado corrente em execução para o estado de entrada do trecho de código associado ao construto ?o_2, caso contrário, a transição é criada para o construto ?o_3.
jump(?o)	Cria uma transição do estado corrente em execução para o estado de entrada do trecho de código associado ao construto ?*o*.
read(?o)	Cria a representação adaptativa do dado lido e o associa ao construto *o*.
write(?o)	Converte o construto ?*o* para o valor correspondente a ser escrito pelo dispositivo de saída padrão.

Tabela A.1: Conjunto de primitivas que geram os construtos adaptativos correspondentes aos comandos da linguagem MINLA.

Anexo B – Meta-Compilador Wirth para AdapTools

Logo abaixo estão as tabelas que representam as máquinas que implementam, no Adaptools, o metacompilador Wirth para Adaptools. Ambas as máquinas foram implementadas na própria ferramenta e possuem ações semânticas associadas às suas regras.

A tabela seguinte contém o analisador léxico do metacompilador Wirth para Adaptools. As ações semânticas associadas a algumas das regras da tabela pertencem à biblioteca de ações semânticas padrão do Adaptools. As ações semânticas `_initBuffer` e `_appendToBuffer` são exemplos de ações responsáveis pelo processamento da sequência de caracteres da entrada. As ações semânticas `_terminal` e `_nonTerminal` são exemplos de ações responsáveis pela identificação de átomos da linguagem.

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	Main	0	spc	0	fin	nop	nop
2	Main	0	”	1	nop	_initBuffer	nop
3	Main	0	=	2	nop	_metaSymbol	nop
...	...	...	...	...	...	...	...
12	Main	0	.	2	nop	_metaSymbol	nop
13	Main	0	eps	100	3	_initBuffer	nop
14	Main	1	”	2	nop	_terminal	nop
15	Main	1	eps	100	1	nop	nop
...	...	...	...	...	...	...	...
19	Main	3	eps	2	nop	_nonTerminal	nop
20	Main	2	eps	0	fin	nop	nop
21	Special	1	spc	1	nop	_appendToBuffer	nop
22	Special	1	(	1	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
52	Special	1	.	1	nop	_appendToBuffer	nop
53	Escape	1		10	nop	nop	nop

54	Escape	10		1	nop	_appendToBuffer	nop
55	Escape	10	”	1	nop	_appendToBuffer	nop
56	Digi	200	0	201	nop	_appendToBuffer	nop
57	Digi	200	1	201	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
65	Digi	200	9	201	nop	_appendToBuffer	nop
66	Digi	201	eps	pop	fin	nop	nop
67	Lett	100	a	101	nop	_appendToBuffer	nop
...	...	...	...	...	...	...	...
118	Lett	100	Z	101	nop	_appendToBuffer	nop
119	Lett	101	eps	pop	fin	nop	nop

Tabela B.1: Componente léxica do metacompilador (PISTORI, 2003) utilizado para gerar o código da máquina reconhecedora da linguagem MINLA.

A tabela seguinte contém o analisador sintático do metacompilador Wirth para Adaptools. Os átomos identificados e rotulados pelo analisador léxico são enviados como entrada para o analisador sintático. As biblioteca de ações semânticas utilizadas pelas regras desta tabela são responsáveis por gerarem o respectivo código Adaptools para cada uma das regras de produção em notação de Wirth. As ações semânticas que realizam esta tarefa correspondem ao conjunto `rs0` a `rs1`.

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	Gram	0	n	1	nop	rs0	nop
2	Gram	1	=	2	nop	rs2	nop
3	Gram	2	eps	5	3	nop	nop
4	Gram	3	.	4	nop	rs8	nop
5	Gram	4	n	1	nop	rs7	nop
6	Gram	4	eps	pop	nop	fin	nop
7	Expr	5	&	6	nop	rs1	nop
8	Expr	5	t	6	nop	rs1	nop
9	Expr	5	n	6	nop	rs1	nop
10	Expr	5	(	7	nop	rs2	nop
11	Expr	5	[	9	nop	rs3	nop
12	Expr	5	{	11	nop	rs4	nop

13	Expr	6	&	6	nop	rs1	nop
14	Expr	6	t	6	nop	rs1	nop
15	Expr	6	n	6	nop	rs1	nop
16	Expr	6	—	5	nop	rs6	nop
17	Expr	6	(	7	nop	rs2	nop
18	Expr	6	[	9	nop	rs3	nop
19	Expr	6	{	11	nop	rs4	nop
20	Expr	6	eps	pop	nop	nop	nop
21	Expr	7	eps	5	8	nop	nop
22	Expr	8	)	6	nop	rs5	nop
23	Expr	9	eps	5	10	nop	nop
24	Expr	10	]	6	nop	rs5	nop
25	Expr	11	eps	5	12	nop	nop
26	Expr	12	}	6	nop	rs5	nop

Tabela B.2: Componente sintática do metacompilador (PISTORI, 2003) utilizado para gerar o código da máquina reconhecedora da linguagem MINLA.

Vale ressaltar que a adição de novas bibliotecas de ações semânticas pode ser feita através da implementação de novas classes *Semantics.java*. Uma melhoria importante na última versão do Adaptools trata exatamente da adição de novas rotinas semânticas. O sistema dispões de uma classe, *adaptools.vm.Semantics*, que através da simples implementação de novas classes *Semantics.java* gera novos módulos para o tratamento de rotinas semânticas. Após a implementação e compilação do novo módulo, e necessária apenas a cópia da nova classe semânticas para uma pasta específica do ambiente, localizado junto ao arquivo executável do AdapTools. Com isso, as bibliotecas de rotinas semânticas disponíveis na ferramenta, ficam visíveis através da barra de menus do ambiente, simplificando a seleção da biblioteca de ações a ser utilizada pela máquina correspondente.

Anexo C – Máquina de Execução de Programas Escritos em Adaptools

Aqui está a tabela que contém a biblioteca de funções adaptativas que implementam uma máquina de execução totalmente baseada no Adaptools. Qualquer código intermediário gerado no formato de um autômato, tendo suas transições especificadas como uma sequência de instruções a ser executada pode ser “carregado” para o segmento de código, representado na linha 12 da tabela para ser executado. Vale ressaltar que o código correspondente pode conter ações semânticas ou adaptativas. Estas ações correspondem são responsáveis por gerarem os efeitos das primitivas da linguagem, projetadas para produzirem o construto correspondente a cada uma das construções sintáticas da linguagem.

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	S	0	eps	1000	nop	nop	initProg(1).
2	S	1000	eps	1001	nop	nop	.nextip(1001,1003)
3	S	1001	eps	1002	nop	nop	nop
4	S	1002	@	1003	nop	nop	nop
5	S	1003	eps	1004	nop	nop	.restaura(1001,1003)
6	S	1004	eps	1000	nop	nop	.continue
7	S	1004	eps	1005	fin	nop	nop
8	S	0	ADAPCODE	2000	nop	nop	nop
9	S	0	IP	3000	nop	nop	nop
10	S	0	CURRENT_IP	4000	nop	nop	nop
11	S	0	LABEL	5000	nop	nop	nop
12	S	0	CODE	1	nop	nop	nop
13	S	1	...	...	...	...	...
INSERIR AQUI O CÓDIGO A SER EXECUTADO							
...	...	...	...	...	...	...	...
21	S	9	END	10	nop	nop	end.termina
22	S	10	eps	0	nop	nop	nop

...	...	...	...	...	...	...	...
INSERIR AQUI A BIBLIOTECA DE AÇÕES ADAPTATIVAS QUE IMPLEMENTAM AS PRIMITIVAS DA LINGUAGEM							
...	...	...	...	...	...	...	...
34	+initProg	3000	#	?p1	nop	nop	nop
35	+initProg	2000	%	*n	nop	nop	nop
36	+initProg	*n	eps	2000	nop	nop	nop
37	?nextip	?a	@	?b	?z1	?z2	?z3
38	?nextip	3000	#	?c	nop	nop	nop
39	?nextip	?c	?s	?x	?z7	?z8	?z9
40	?nextip	?x	eps	?y	nop	nop	nop
41	-nextip	3000	#	?c	nop	nop	nop
42	-nextip	?p1	eps	?a	nop	nop	nop
43	-nextip	?x	eps	?y	nop	nop	nop
44	+nextip	3000	#	?y	nop	nop	nop
45	+nextip	?p1	eps	?c	nop	nop	nop
46	+nextip	?x	eps	?p2	nop	nop	nop
47	?restaura	?p1	eps	?a	nop	nop	nop
48	?restaura	?b	eps	?p2	nop	nop	nop
49	?restaura	3000	#	?c	nop	nop	nop
50	?restaura	?w	@	?t	nop	nop	nop
51	-restaura	?p1	eps	?a	nop	nop	nop
52	-restaura	?b	eps	?p2	nop	nop	nop
53	+restaura	?b	eps	?c	nop	nop	nop
54	+restaura	?p1	eps	?w	nop	nop	nop
55	?continue	0	CODE	?x	nop	nop	nop
56	?continue	3000	#	?y	nop	nop	nop
57	?continue	?a	eps	?y	nop	nop	nop
58	?continue	?x	?w	?a	?z1	?z2	?z3
59	-continue	?x	?w	?a	?z1	?z2	?z3
60	-continue	?a	eps	?y	nop	nop	nop
61	-continue	0	CODE	?x	nop	nop	nop
62	+continue	0	CODE	?y	nop	nop	nop

Tabela C.1: Máquina de execução projetada para a execução de código no formato Adaptools.

Anexo D – Reconhecedor Sintático da Linguagem MINLA

	Head	Orig	Inpu	Dest	Push	Outp	Adap
1	programa	0	eps	1000	2	nop	nop
2	programa	2	eps	3	nop	nop	nop
3	programa	3	;	4	nop	nop	nop
4	programa	4	eps	1000	5	nop	nop
5	programa	5	eps	3	nop	nop	nop
6	programa	3	END	6	nop	nop	nop
7	programa	6	eps	1	nop	nop	nop
8	programa	1	eps	pop	fin	nop	nop
9	comando	1000	eps	1002	nop	nop	nop
10	comando	1002	identificador	1003	nop	nop	nop
11	comando	1003	:	1004	nop	nop	nop
12	comando	1004	eps	1002	nop	nop	nop
13	comando	1002	LET	1006	nop	nop	nop
14	comando	1006	identificador	1007	nop	nop	nop
15	comando	1007	:	1008	nop	nop	nop
16	comando	1008	=	1009	nop	nop	nop
17	comando	1009	eps	2000	1010	nop	nop
18	comando	1010	eps	1005	nop	nop	nop
19	comando	1002	GOTO	1011	nop	nop	nop
20	comando	1011	identificador	1012	nop	nop	nop
21	comando	1012	eps	1005	nop	nop	nop
22	comando	1002	READ	1013	nop	nop	nop
23	comando	1013	identificador	1015	nop	nop	nop
24	comando	1015	eps	1016	nop	nop	nop
25	comando	1016	,	1017	nop	nop	nop
26	comando	1017	identificador	1018	nop	nop	nop
27	comando	1018	eps	1016	nop	nop	nop
28	comando	1016	eps	1014	nop	nop	nop

29	comando	1014	eps	1005	nop	nop	nop
30	comando	1002	PRINT	1019	nop	nop	nop
31	comando	1019	eps	2000	1021	nop	nop
32	comando	1021	eps	1022	nop	nop	nop
33	comando	1022	,	1023	nop	nop	nop
34	comando	1023	eps	2000	1024	nop	nop
35	comando	1024	eps	1022	nop	nop	nop
36	comando	1022	eps	1020	nop	nop	nop
37	comando	1020	eps	1005	nop	nop	nop
38	comando	1002	IF	1025	nop	nop	nop
39	comando	1025	eps	3000	1026	nop	nop
40	comando	1026	THEN	1027	nop	nop	nop
41	comando	1027	eps	1000	1028	nop	nop
42	comando	1028	eps	1005	nop	nop	nop
43	comando	1002	eps	1029	nop	nop	nop
44	comando	1029	eps	1005	nop	nop	nop
45	comando	1005	eps	1001	nop	nop	nop
46	comando	1001	eps	pop	fin	nop	nop
47	expcomp	3000	eps	2000	3002	nop	nop
48	expcomp	3002	<	3004	nop	nop	nop
49	expcomp	3004	eps	3003	nop	nop	nop
50	expcomp	3002	=	3005	nop	nop	nop
51	expcomp	3005	eps	3003	nop	nop	nop
52	expcomp	3002	>	3006	nop	nop	nop
53	expcomp	3006	eps	3003	nop	nop	nop
54	expcomp	3003	eps	2000	3007	nop	nop
55	expcomp	3007	eps	3001	nop	nop	nop
56	expcomp	3001	eps	pop	fin	nop	nop
57	exparit	2000	identificador	2003	nop	nop	nop
58	exparit	2003	eps	2002	nop	nop	nop
59	exparit	2000	numero	2004	nop	nop	nop
60	exparit	2004	eps	2002	nop	nop	nop
61	exparit	2002	eps	2005	nop	nop	nop
62	exparit	2005	+	2007	nop	nop	nop
63	exparit	2007	eps	2006	nop	nop	nop
64	exparit	2005	−	2008	nop	nop	nop
65	exparit	2008	eps	2006	nop	nop	nop
66	exparit	2006	identificador	2010	nop	nop	nop

67	exparit	2010	eps	2009	nop	nop	nop
68	exparit	2006	numero	2011	nop	nop	nop
69	exparit	2011	eps	2009	nop	nop	nop
70	exparit	2009	eps	2005	nop	nop	nop
71	exparit	2005	eps	2001	nop	nop	nop
72	exparit	2001	eps	pop	fin	nop	nop
122	identificador	2000	letter	2002	nop	nop	nop
123	identificador	2002	eps	2003	nop	nop	nop
124	identificador	2003	letter	2004	nop	nop	nop
125	identificador	2004	eps	2003	nop	nop	nop
126	identificador	2003	digit	2005	nop	nop	nop
127	identificador	2005	eps	2003	nop	nop	nop
128	identificador	2003	eps	2001	nop	nop	nop
129	identificador	2001	eps	pop	fin	nop	nop
130	numero	4000	digit	4002	nop	nop	nop
131	numero	4002	eps	4003	nop	nop	nop
132	numero	4003	digit	4004	nop	nop	nop
133	numero	4004	eps	4003	nop	nop	nop

Tabela D.1: Código Adaptools do reconhecedor para a linguagem MINLA obtido através do metacompilador Wirth2APE (PISTORI, 2003).

Referências Bibliográficas

AHO, A.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Rio de Janeiro/RJ: LTC, 1995.

ALLAUZEN, C.; MOHRI, M. The determinizability of weighted automata and transducers. In: *Workshop of Weighted Automata: Theory and Applications (WATA)*. Dresden, Germany: [s.n.], 2002.

ALLAUZEN, C.; MOHRI, M. An efficient pre-determinization algorithm. In: *CIAA - 8th International Conference on Implementation and Application of Automata*. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 2759), p. 83–95.

ANCKAERT, B.; MADOU, M.; DE BOSSCHERE, K. A model for self-modifying code. In: CAMENISCH, J. (Ed.). *Proceedings of the 8th Information Hiding Conference*. Berlin Heidelberg: Springer-Verlag, 2007. v. 4437, n. 4437, p. 232–248.

BRABRAND, C.; SCHWARTZBACH, M. I. Growing languages with metamorphic syntax macros. In: *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM Press, 2002. p. 31–40. ISBN 1-58113-455-X.

BROOKER, R. A.; MORRIS, D. A general translation program for phrase structure languages. *J. ACM*, ACM Press, New York, NY, USA, v. 9, n. 1, p. 1–10, 1962. ISSN 0004-5411.

CASTRO JR., A. A.; NETO, J. J.; PISTORI, H. Determinismo em autômatos de estados finitos adaptativos. *Revista IEEE América Latina*, v. 5, p. 515–521, 2007.

CHANDLER, G. D. Metapi - a language for extensions. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 8–9, 1971. ISSN 0362-1340.

CHEATHAM JR., T. E. Motivation for extensible languages. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 45–49, 1969. ISSN 0362-1340.

CHEATHAM JR., T. E. Extensible language - where are we going. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 146–147, 1971. ISSN 0362-1340.

CHRISTENSEN, C.; SHAW, C. J. (Ed.). *Proceedings of the International Symposium on Extensible Languages*. New York, NY, USA: ACM Press, 1969.

CHRISTIANSEN, H. A survey of adaptable grammars. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 25, n. 11, p. 35–44, 1990. ISSN 0362-1340.

CIFUENTES, C.; GOUGH, K. J. Decompilation of binary programs. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 25, n. 7, p. 811–829, 1995. ISSN 0038-0644.

DUBY, J. J. Extensible languages: A potential user's point of view. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 137–140, 1971. ISSN 0362-1340.

FREITAS, A. V.; NETO, J. J. Adaptive device with underlying mechanism defined by a programming language. In: *Proceedings of the 4th WSEAS International Conference on Information Security, Communications and Computers*. Tenerife, Espanha: [s.n.], 2005. p. 423–428.

FREITAS, A. V.; NETO, J. J. Adaptive languages and a new programming style. In: *Proceedings of the 6th WSEAS International Conference on APPLIED COMPUTER SCIENCE (ACS'06)*. Tenerife, Espanha: [s.n.], 2006. p. 220–225.

FREITAS, A. V.; NETO, J. J. Conception of adaptive programming languages. In: *Proceedings of the XVII IASTED International Conference on Modelling and Simulation*. Montreal, Canadá: [s.n.], 2006. p. 453.

FREITAS, A. V.; NETO, J. J. Um ambiente para o processamento de linguagens adaptativas de programação. In: *XII Congreso Argentino de Ciencias de la Computación - CACIC*. Potrero de Los Funes, Argentina: [s.n.], 2006.

FREITAS, A. V.; NETO, J. J. Adaptivity in programming languages. In: *WSEAS Transactions on Information Science and Applications*. [S.l.: s.n.], 2007. v. 4, p. 779–796.

FREITAS, A. V.; NETO, J. J. Linguagens de programação aderentes ao paradigma adaptativo. *Revista IEEE América Latina*, v. 5, p. 522–526, 2007.

FREITAS, A. V. de. *Considerações sobre o Desenvolvimento de Linguagens Adaptativas de Programação*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo - POLI/USP, 2008.

GIFFIN, J.; CHRISTODORESCU, M.; KRUGER, L. Strengthening software self-checksumming via self-modifying code. In: APPLIED COMPUTER ASSOCIATES. *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*. Tucson, AZ, USA: IEEE, 2005. p. 18–27.

HROMKOVIC, J. et al. Measures of nondeterminism in finite automata. In: *Proc. ICALP 2000, Lecture Notes in Computer Science*. [S.l.: s.n.], 2000. v. 1853, p. 199–210.

HUBBARD JR., R. D. Language extensibility and program design. In: *DAC '76: Proceedings of the 13th conference on Design automation*. New York, NY, USA: ACM Press, 1976. p. 371–376.

JESUS, L. et al. Adaptools 2.0: Aspectos de implementação e utilização. *Revista IEEE América Latina*, v. 5, p. 527–532, 2007.

KOLBLY, D. M. *Extensible Language implementation*. Tese (Doutorado) — University of Texas, 2002.

LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elements of the Theory of Computation*. 2nd.. ed. [S.l.]: Prentice Hall, 1997.

LUZ, J. C. *Tecnologia Adaptativa Aplicada à Otimização de Código em Compiladores*. Dissertação (Mestrado) — Escola Politécnica da Universidade de São Paulo - POLI/USP, 2004.

MAILLOUX, B. J.; PECK, J. E. L. Algol 68 as an extensible language. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 9–13, 1969. ISSN 0362-1340.

MARCOTTY, M.; LEDGARD, H. *Programming Language Landscape: Syntax, Semantics and Implementation*. 2nd.. ed. [S.l.]: Macmillan Publishing Company, 1986.

MCILROY, M. D. Macro instruction extensions of compiler languages. *Commun. ACM*, ACM Press, New York, NY, USA, v. 3, n. 4, p. 214–220, 1960. ISSN 0001-0782.

MCILROY, M. D. Alternatives to extensible languages. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 50–52, 1969. ISSN 0362-1340.

MCKINLEY, P. K. et al. Composing adaptive software. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 7, p. 56–64, 2004. ISSN 0018-9162.

METZNER, J. R. A graded bibliography on macro systems and extensible languages. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 14, n. 1, p. 57–64, 1979. ISSN 0362-1340.

NETO, J. J. *Introdução à Compilação*. [S.l.]: LTC, 1987.

NETO, J. J. Contribuição à metodologia de construção de compiladores. *Post-Doctoral Tese de Livre Docência, Escola Politécnica, Universidade de São Paulo*, São Paulo, Brasil, 1993.

NETO, J. J. Adaptive automata for context-sensitive languages. *SIGPLAN NOTICES*, v. 29, n. 9, p. 115–124, September 1994.

NETO, J. J. Solving complex problems efficiently with adaptative automata. In: *Conference on the Implementation and Application of Automata - CIAA 2000*. Ontario, Canada: [s.n.], 2000.

NETO, J. J. Adaptative rule-driven devices - general formulation and a case study. In: *CIAA'2001 Sixth International Conference on Implementation and Application of Automata*. Pretoria, South Africa: [s.n.], 2001. p. 234–250.

NETO, J. J.; DE MORAES, M. Using adaptive formalisms to describe context-dependencies in natural language. *Lecture Notes in Artificial Intelligence. N.J. Mamede, J. Baptista, I. Trancoso, M. das Graças, V. Nunes (Eds.): Computational Processing of the Portuguese Language 6th International Workshop, PROPOR 2003*, Faro, Portugal, v. 2721, p. 94–97, June 2003.

NETO, J. J.; IWAI, M. K. Adaptive automata for syntax learning. In: *Anais da XXIV Conferencia Latinoamericana de Informática - CLEI 98*. Quito, Equador: [s.n.], 1998. p. 135–149.

NETO, J. J.; PARIENTE, C. B.; LEONARDI, F. Compiler construction - a pedagogical approach. In: *Proceedings of the V International Congress on Informatic Engineering - ICIE 99*. Buenos Aires, Argentina: [s.n.], 1999.

NETO, J. J.; ROCHA, R. L. de A. *Memórias do WTA2008: Segundo Workshop de Tecnologia Adaptativa*. 1. ed. São Paulo/SP: EPUSP - Escola Politécnica da Universidade de São Paulo, 2008.

NIELSON, H. R.; NIELSON, F. *Semantics with Applications: A Formal Introduction*. [S.l.]: Wiley Professional Computing, 1999.

NOTKIN, D.; GRISWOLD, W. G. Extension and software development. In: *ICSE '88: Proceedings of the 10th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988. p. 274–283. ISBN 0-89791-258-6.

PAAKKI, J. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, v. 27, n. 2, p. 196–255, 1995.

PAGAN, F. G. *Formal Specification of Programming Languages: A Panoramic Primer*. [S.l.]: Prentice Hall, 1981.

PELEGRINI, E. J.; NETO, J. J. Um modelo de execução para código dinamicamente variável baseado em autômatos adaptativos. In: *Workshop de Tecnologia Adaptativa - WTA2008*. São Paulo, SP: [s.n.], 2008.

PERLIS, A. J. Introduction to extensible languages. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 3–5, 1969. ISSN 0362-1340.

PFLÚCKER, O.; NETO, J. J. Gama-nw: Un enfoque reutilizable para un metalenguaje y un metacompilador. In: *Memóris del CLEI 2007 - Congreso Latinoamericano de Informática*. San José: [s.n.], 2007.

PISTORI, H. *Tecnologia Adaptativa em Engenharia de Computação: Estado da Arte e Aplicações*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo - POLI/USP, 2003.

PISTORI, H.; MARTINS, P. S.; CASTRO JR., A. A. de. Adaptive finite state automata and genetic algorithms: Merging individual application and populations evolution. In: *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms - ICANNGA*. Coimbra, Portugal: [s.n.], 2005.

PISTORI, H.; NETO, J. J. *AdapTools: Aspectos de Implementação e Utilização*. São Paulo/SP, 2003.

PISTORI, H.; NETO, J. J. Decision tree induction using adaptive FSA. *CLEI Electronic Journal*, v. 6, n. 1, 2003. ISSN 0717-5000.

PISTORI, H.; NETO, J. J. A free software for the development of adaptive automata. In: *Proceedings of the IV Workshop on Free Software - WSL (IV International Forum on Free Software)*. Porto Alegre, Brasil: [s.n.], 2003.

RAMOS, M. V. M.; NETO, J. J.; VEGA Ítalo S. *Linguagens Formais: Teoria, Modelagem e Implementação*. [S.l.]: Bookman, 2009.

SCHUMAN, S. A. (Ed.). *Proceedings of the International Symposium on Extensible Languages*. New York, NY, USA: ACM Press, 1971.

SEBESTA, R. W. *Concepts of Programming Languages*. 6th.. ed. [S.l.]: Addison Wesley, 2006.

SILVA, P. S. M.; NETO, J. J. An adaptive framework for the design of software specification languages. In: *International Conference on Adaptive and Natural Computing Algorithms - ICANNGA 2005*. Coimbra, Portugal: [s.n.], 2005.

SOUSA, M. A. A.; HIRAKAWA, A. H. Robotic mapping and navigation in unknown environments using adaptive automata. In: *Proceedings of International Conference on Adaptive and Natural Computing Algorithms - ICANNGA 2005*. Coimbra, Portugal: [s.n.], 2005.

STANDISH, T. A. Some compiler-compiler techniques for use in extensible languages. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 55–62, 1969. ISSN 0362-1340.

STANDISH, T. A. Some features of ppl, a polymorphic programming language. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 20–26, 1969. ISSN 0362-1340.

STANDISH, T. A. Ppl - an extensible language that failed. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 144–145, 1971. ISSN 0362-1340.

STANDISH, T. A. Extensibility in programming language design. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 10, n. 7, p. 18–21, 1975. ISSN 0362-1340.

STEARNS, R. E. Deterministic versus nondeterministic time and lower bound problems. *Journal of the ACM*, v. 50, n. 1, p. 91–95, 2003.

STEELE, G. L. Growing a language. *Higher-Order and Symbolic Computation*, v. 12, n. 3, p. 221–236, 1999. Disponível em: <citeseer.ist.psu.edu/steele99growing.html>.

TSCHUDIN, C. F.; YAMAMOTO, L. Harnessing self-modifying code for resilient software. In: HINCHEY, M. G. et al. (Ed.). *WRAC*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3825), p. 197–204.

TURBA, T. N. A facility for the downward extension of a high-level language. In: *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA: ACM Press, 1982. p. 127–133. ISBN 0-89791-074-5.

VAN GILS, A. J. M. The philosophy of an extensible language. *ALGOL Bull.*, Computer History Museum, Mountain View, CA, United States, n. 34, p. 82–88, 1972. ISSN 0084-6198.

VAN GILS, T. Syntactic definition mechanisms. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 67–74, 1971. ISSN 0362-1340.

VAN WYK, E. Domain specific meta languages. In: *ACM Symposium on Applied Computing*. Como, Italy: Association of Computing Machinery, 2000. v. 2, p. 799–803.

WEGNER, P. Panel on the concept of extensibility. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 4, n. 8, p. 53–54, 1969. ISSN 0362-1340.

WODON, P. L. A syntax macro processor. *SIGPLAN Not.*, ACM Press, New York, NY, USA, v. 6, n. 12, p. 48–50, 1971. ISSN 0362-1340.