JOÃO JOSÉ NETO

CONTRIBUIÇÕES À METODOLOGIA DE CONSTRUÇÃO DE COMPILADORES

Tese apresentada à Escola Politécnica da Universidadede São Paulo para a obtenção do título de Professor Livre-Docente junto ao Departamento de Engenharia de Computação e Sistemas Digitais .

São Paulo, 1993

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

JOÃO JOSÉ NETO

Engenheiro Eletricista pela Escola Politécnica da USP, 1971
Mestre em Engenharia Elétrica pela Escola Politécnica da USP, 1975
Doutor em Engenharia Elétrica pela Escola Politécnica da USP, 1980

CONTRIBUIÇÕES À METODOLOGIA DE CONSTRUÇÃO DE COMPILADORES

Tese apresentada à Escola Politécnica da Universidadede São Paulo para a obtenção do título de Professor Livre-Docente junto ao Departamento de Engenharia de Computação e Sistemas Digitais .

Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
Laboratório de Sistemas Digitais

São Paulo, 1993

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

José Neto, João
Contribuições à metodologia de construção de compiladores. São Paulo, 1993.
272 p. + anexos
Tese (Livre-Docência) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Computadores digitais 2. Linguagens formais e autômatos 3. Compiladores I. Universidade de São Paulo. Escola Politécnica Departamento de Engenharia de Computação e Sistemas Digitais II. T

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

À minha esposa Mei Lee
e a nossa filha Regina

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

RESUMO

O presente trabalho documenta uma pesquisa no sentido de buscar, para a elaboração de reconhecedores sintáticos, uma solução que se revele ao mesmo tempo simples e eficiente, que incorpore recursos para o tratamento uniforme dos problemas sintáticos usualmente encontrados na confecção de compiladores, e que exiba ainda um potencial para ser utilizada de forma fácil, econômica e automática, quer na fase de projeto, quer na época da utilização do seu produto.

Para tanto, introduz-se o conceito de autômato adaptativo, uma classe de máquinas de estados finitos com memória organizada em pilha, e com recursos de aprendizagem, baseados na alteração dinâmica da configuração da máquina, em função das transições efetuadas por este autômato.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ABSTRACT

The present work documents a research whose intent was searching an efficient and simple solution for the construction of parsers, featuring uniform handling of syntax problems usually found in compiler construction.

That solution should be used simply, cheaply and automatically, both at design and execution time.

For that, the concept of adaptive automata is introduced, standing for a class of finite state devices with stack memory and learning capabilities, achieved through transition-driven self-modification.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

PREFÁCIO À VERSÃO ELETRÔNICA

É com satisfação que, dez anos após a publicação dessa tese, volto novamente a editá-la. Muita coisa ocorreu nesses dez anos, tendo sido muito significativos os resultados da evolução da pesquisa sobre tecnologias adaptativas, que foi iniciada com a publicação deste material.

Um fato que muito nos anima a prosseguirmos nesse caminho foi a crescente conscientização da comunidade de Computação, em suas diversas modalidades, acerca da importância da pesquisa científica para o desenvolvimento da tecnologia da área.

Isso cria um interesse cada vez maior dos profissionais de Ciëncia da Computação por assuntos da área de Engenharia e vice-versa, o que comprova a interdependência das especialidades, e realça o fato de que sem ciência a tecnologia se esvazia, e sem aplicações a ciência perde grande parte de seus atrativos.

Desde sua publicação, esta tese produziu e vem produzindo inúmeros frutos tecnológicos, em diversas áreas do conhecimento, resultados esses que estão relatados em diversos trabalhos publicados, e que suscitaram o constatado grande aumento da procura pelo acesso ao seu conteúdo.

Por essa razão, resolvi disponibilizá-la nos populares formatos eletrônicos .pdf e .ps, os quais, ao contrário do formato no qual a tese foi originalmente produzida, podem com facilidade ser armazenados, visualizados e transmitidos pelos meios modernos de comunicação, tornando-se dessa maneira amplamente acessível aos interessados.

Deve-se observar que o conteúdo final do material impresso, embora tenha sido mantido praticamente inalterado em sua essência, sofreu pequenas modificações superficiais, para facilitar ou mesmo para permitir a utilização de alguns recursos de editoração.

Pequenos erros presentes no texto original foram corrigidos, muitas fórmulas, redigitadas, e todas as figuras, refeitas. Disso resultaram inevitáveis diferenças na aparência, na diagramação e até na paginação do presente material em relação ao original, a despeito da identidade de conteúdo.

Esta tese, assim como teses e dissertações adicionais, bem como muitas outras publicações mais recentes, produzidas pelos membros do LTA – Laboratório de Linguagens e Tecnologias Adaptativas – estão sendo mantidas em repositório para acesso livre na internet, na página do LTA, acessível na seção de publicações, do endereço eletrônco www.pcs.usp.br/~lta .

Agradeço a todos os que se interessam por esse material, e espero que sua disponibilização mais ampla permita que seu conteúdo possa ser aproveitado como base para o aprofundamento das pesquisas nessa área.

Solicito aos que puderem colaborar com o aprimoramento deste material que quaisquer erros que venham a ser encontrados nos sejam reportados, para que possamos sempre manter acessível a todos um texto que seja o mais correto possível.

São Paulo, fevereiro de 2003
PROF. Dr. João José Neto
Coordenador do LTA

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

PREFÁCIO

As atividades relacionadas com a especificação, análise, projeto e implementação de linguagens de programação apresentam um grande fascínio técnico, dada a diversidade de problemas suscitados, os quais despertam muitos importantes interesses científicos e tecnológicos.

Deste fato originou-se a principal motivação desta pesquisa, nascida inicialmente de um esforço com o objetivo de criar meios de popularização das técnicas de processamento de linguagens, por meio da elaboração de um método simples e intuitivo de construção de reconhecedores sintáticos para linguagens de programação.

O progresso da pesquisa nesta direção trouxe à tona o fato de ser economicamente viável a geração automática de reconhecedores sintáticos eficientes, através do uso de um método com características de simplicidade tais que o tornam indicado para ser utilizado sem prejuízos mesmo em cursos introdutórios, ministrados a platéias que não exibam obrigatoriamente uma formação teórica muito profunda na área.

Algumas contribuições adicionais, resultantes desta pesquisa, são relatadas em detalhe, ressaltando-se suas aplicações mais significativas. Entre elas, destaca-se a introdução do conceito básico do transdutor adaptativo, o qual fornece substrato suficiente para que um conjunto substancial de problemas complexos ligados à análise sintática passem a ter um tratamento mais simples, intuitivo e uniforme.

Procura-se, neste texto, fornecer um suporte conceitual a todo o processo em questão, ao lado da apresentação de técnicas, métodos e aplicações considerados relevantes a este estudo.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

AGRADECIMENTOS

São inúmeros os agradecimentos devidos a quem, de uma ou outra forma, colaboraram para que este trabalho se completasse. Gostaria, porém, de tornar pública minha gratidão a alguns que de alguma forma mais especial me levaram a conclui-lo:

Os professores Giorgio Gambirasio, Geraldo Lino de Campos e Francisco José de Oliveira Dias, pelo estímulo e apoio recebidos.

O engenheiro Mário Eduardo Santos Magalhães, pela parceiria na concepção inicial de uma parcela significativa deste material.

Os profesores João Batista Camargo Jr. e Paulo Sérgio Muniz Silva, pelo incentivo, troca de idéias e sugestões.

Os professores Marcus Vinicius Midena Ramos e Italo Santiago Vega, pela disponibilidade, idéias técnicas e pelo importante auxílio prestado na fase final do trabalho.

Os engenheiros Newton Kiyotaka Miura e Washington Komatsu, pelos ensaios e programas realizados em computador.

A minha esposa, Cheng Mei Lee, pela permanente e obstinada paciência, entusiasmo, apoio e compreensão, e pela imprescindível e ativa participação técnica, troca de idéias, críticas e sugestões, decisivos para a viabilização desta tese.

A todos os meus familiares, especialmente à Regina, pela compreensão e paciência com que por tantos anos suportaram minha distância durante a elaboração deste infindável trabalho.

São Paulo, 1993
João José Neto

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

S U M Á R I O

RESUMO **5**

ABSTRACT **6**

PREFÁCIO **7**

AGRADECIMENTOS **9**

S U M Á R I O **10**

1 INTRODUÇÃO **13**

1.1 APRESENTAÇÃO 13
1.2 MOTIVAÇÕES 14
1.3 OBJETIVOS 14
1.4 HISTÓRICO 15
1.5 ESTRUTURA DA TESE 15

2 CONCEITOS **17**

2.1 TERMINOLOGIA 17
2.2 LINGUAGENS REGULARES E LIVRES DE CONTEXTO 17
2.3 GRAMÁTICAS LINEARES E LIVRES DE CONTEXTO 18
2.4 AUTÔMATOS FINITOS E DE PILHA 18
2.5 AUTÔMATOS DE PILHA ESTRUTURADOS 19
2.6 ÁRVORES DE DERIVAÇÃO E DE RECONHECIMENTO 20
2.7 NÃO-DETERMINISMOS E AMBIGÜIDADES 20
2.8 SIMPLIFICAÇÃO E OTIMIZAÇÃO DE RECONHECEDORES 20
2.9 RECUPERAÇÃO DE ERROS 21
2.10 LINGUAGENS EXTENSÍVEIS E MECANISMOS DE EXTENSÃO 22
2.11 DEPENDÊNCIAS DE CONTEXTO E SUA FORMALIZAÇÃO 23
2.12 AUTÔMATOS DE PILHA ADAPTATIVOS 25

3 CONSTRUÇÃO DE RECONHECEDORES LIVRES DE CONTEXTO **26**

3.1 CONSTRUÇÃO DE AUTÔMATOS DE PILHA SUB-ÓTIMOS 26
3.1.1 A preparação da gramática *27*
Exemplo 1 .... *30*
Exemplo 2 .... *30*
3.1.2 Designação de estados ao autômato *31*
Procedimento de designação de estados *32*
3.1.3 Construção das produções do autômato *33*
Procedimento de obtenção das produções do autômato *33*
3.1.4 Exemplo de Aplicação *34*
Preparação da gramática *35*
Designação de estados *35*
Construção das produções do autômato *35*
Considerações sobre a Implementação *36*
3.2 CONSTRUÇÃO DE EXTENSÕES PARA A RECUPERAÇÃO DE ERROS 38
3.2.1 Erros simples em autômatos finitos determinísticos *39*
Procedimento para a construção de extensões de recuperação de erros em autômatos finitos determinísticos *40*
Casos particulares a serem considerados: *40*
3.2.2 Erros simples em autômatos finitos não-determinísticos *40*
3.2.3 Erros múltiplos em autômatos finitos *41*
3.2.4 Erros simples em autômatos de pilha estruturados *42*
Procedimento para a obtenção do conjunto de sucessores *43*

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Procedimento de montagem das extensões de recuperação de erros simples em autômatos de pilha estruturados ... 44
Recuperação interna à sub-máquina: ... 44
Recuperação em estados finais de sub-máquina: ... 45
Recuperação em chamadas de sub-máquinas: ... 45
3.2.5 Erros múltiplos em autômatos de pilha estruturados ... 46
3.3 OBTENÇÃO DE ANALISADORES SINTÁTICOS ... 47
3.3.1 Núcleos de Transdução Sintática ... 47
3.3.2 Aplicação dos Núcleos à Geração de Analisadores Sintáticos ... 49
Preparação da Gramática para a Construção do Transdutor ... 50
Construção do Autômato de Suporte do Transdutor ... 52
Construção das Regras de Mapeamento ... 53
3.3.3 Exemplos Ilustrativos do Uso do Transdutor ... 54
Exemplo 1 ... 54
Exemplo 2 ... 55
Exemplo 3 ... 56

4 AUTÔMATOS ADAPTATIVOS ... 60

4.1 CONCEITUAÇÃO DO AUTÔMATO ADAPTATIVO ... 60
4.2 NOTAÇÕES PARA A ESPECIFICAÇÃO DE AUTÔMATOS ADAPTATIVOS ... 64
4.2.1 Uma Notação Algébrica para os Autômatos Adaptativos ... 64
4.2.2 Uma Notação Gráfica para os Autômatos Adaptativos ... 68
4.3 MECANISMOS DE OPERAÇÃO DO AUTÔMATO ADAPTATIVO ... 69
4.3.1 Mecanismo de execução de uma função adaptativa ... 69
4.3.2 Operação das ações adaptativas elementares ... 69
4.3.3 Operação geral do autômato adaptativo ... 70
4.4 EXEMPLOS DE APLICAÇÃO DO AUTÔMATO ADAPTATIVO ... 72
4.4.1 Simulação de uma Pilha ... 72
4.4.2. Coletor de nomes ... 73
4.4.3 Analisador léxico simples ... 75
4.4.4 Declarações de identificadores com tipos associados - caso 1 ... 80
4.4.5 Declarações de identificadores com tipos associados - caso 2 ... 82
4.4.6 Verificação de tipos ... 83
4.4.7 Analisador léxico para uma linguagem estruturada em blocos ... 85
4.4.8. Expansão de Macros ... 92
4.4.8.1 Análise léxica ... 93
4.4.8.2 Declaração de Macros Paramétricas ... 98

5 CONSIDERAÇÕES FINAIS ... 110

5.1 APLICAÇÕES ... 110
Aplicação à Compilação ... 110
Aplicação à Comunicação Digital ... 110
Aplicação à Engenharia de Software ... 110
Aplicação à Construção de Ferramentas ... 111
Aplicação à Meta-Programação ... 111
Aplicação à Inteligência Artificial ... 112
Aplicação Didática ... 112
Aplicação em Ensino Auxiliado por Computador ... 112
5.2 AVALIAÇÕES ... 112
5.3 PERSPECTIVAS ... 113
5.4 CONCLUSÕES ... 113
Validade da Pesquisa ... 114
Importância Pedagógica ... 114
Eficiência ... 114
Unificação dos Modelos de Linguagens ... 114
Recuperação de Erros ... 114
Relação com Outros Modelos Formais ... 114

ANEXO A DESCRIÇÃO DAS METALINGUAGENS ... 116

A.1 A NOTAÇÃO DE WIRTH ... 116

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

A.2 A NOTAÇÃO DE WIRTH MODIFICADA 116

ANEXO B MECANISMOS FORMAIS PARA A REPRESENTAÇÃO DE LINGUAGENS 118

B.1 GRAMÁTICAS 118
B.2 RECONHECEDORES 118

ANEXO C FORMALIZAÇÃO DOS AUTÔMATOS DE PILHA ESTRUTURADOS 121

ANEXO D UM FORMATO PARA A REPRESENTAÇÃO DE ÁRVORES 123

ANEXO E O MAPEAMENTO DE GRAMÁTICAS EM RECONHECEDORES 124

ANEXO F AVALIAÇÕES DE DESEMPENHO 126

REFERÊNCIAS BIBLIOGRÁFICAS 128

BIBLIOGRAFIA RECOMENDADA 130

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

1 INTRODUÇÃO

Neste primeiro capítulo procura-se mostrar de maneira sumária as principais motivações e metas do trabalho realizado, relatando a seqüencia em que foi desenvolvido, e finalizando com a indicação da organização do texto desta tese.

1.1 Apresentação

Este texto destina-se a documentar os resultados de alguns anos de um trabalho de pesquisa, em que o principal tema considerado foi a engenharia e os fundamentos dos métodos de construção de compiladores.

Essencialmente resolvido, o problema da obtenção de reconhecedores sintáticos para linguagens livres de contexto recebeu no passado uma grande atenção dos autores na literatura internacional, pródiga em bons textos, orientados à apresentação e à divulgação de inúmeros métodos e técnicas, e das suas características e vantagens relativas.

Da mesma forma que a maioria das soluções gerais adotadas para problemas inerentemente complexos, os métodos para a análise de linguagens livres de contexto irrestritas, apresentados na literatura (HARRISON, 1978), (SALOMAA, 1973), (NIJHOLT, 1980), (HOPCROFT; ULLMAN, 1979a) exibem sérias deficiências de ordem prática, já que se fundamentam em métodos não-determinísticos, cuja eficiência deixa a desejar, comprometendo seriamente o desempenho de compiladores reais neles baseados.

A principal causa das dificuldades encontradas reside no fato de que as soluções gerais mencionadas propõem-se a resolver um problema que exige, por parte de máquinas e modelos inevitavelmente finitos, a manipulação de parâmetros não-limitados das linguagens a serem reconhecidas.

Por esta razão, proliferaram pesquisas que procuram, ao invés de alternativas de solução para o problema original, soluções de casos particulares mais simples, mas cuja abrangência não seja exageradamente menor que a do problema inicial.

Nesta linha de raciocínio, a literatura busca impor restrições às linguagens livres de contexto, criando desta forma classes de linguagens menos gerais, que atendam porém as exigências dos casos de maior interesse, reduzindo desta forma a complexidade do problema original, e permitindo construir para tais linguagens reconhecedores eficientes e de baixo custo (AHO; ULLMAN, 1972), (BARRETT; COUCH, 1979), (TREMBLAY; SORENSON, 1985), (FISCHER; LEBLANC, 1988), (LEWIS; ROSENKRANTZ; STEARNS, 1976).

Esta tendência resultou inclusive na criação de toda uma taxonomia para as linguagens livres de contexto, de acordo com a aplicabilidade ou não às mesmas dos diversos métodos particulares de análise disponíveis (BAUER; EICKEL, 1976), (BACKHOUSE,1979), (GRIES, 1971), (GOUGH, 1988), (LEWI et al, 1979-1982).

Uma característica muito freqüente dos trabalhos relatados na literatura refere-se à preocupação dos seus autores com a abrangência de cada método, bem como com as relações de inclusão exibidas pelas correspondentes classes de linguagens.

Uma forma alternativa para se enfrentar esta situação, e que foi adotada como uma das diretrizes da presente pesquisa, corresponde a se considerar a linguagem ou a gramática que a descreve em função apenas de suas propriedades inerentes, abstraindo-se o estado da arte das técnicas de construção de reconhecedores.

Procurou-se dar ainda, neste trabalho, uma pequena contribuição, através da apresentação de uma nova formulação matemática, aplicável à modelagem de reconhecedores sintáticos, que se comporta de forma equivalente à dos tradicionais autômatos de pilha, com potencial portanto para o reconhecimento de linguagens livres de contexto gerais.

O modelo em questão permite a aplicação de um método estruturado de reconhecimento sintático, o que facilita a obtenção de reconhecedores com operação hierarquizável, possibilitando desta forma a construção, a baixo custo, de reconhecedores muito eficientes.

Adicionalmente, acompanhando o modelo, é apresentado um método para a confecção de tais reconhecedores diretamente a partir de descrições convencionais, livres de contexto, da linguagem.

Este método procura contornar as dificuldades anteriormente mencionadas, referentes às características não-finitas do processo geral de reconhecimento, através da imposição de limitantes para certos parâmetros do reconhecedor que efetua o seu tratamento.

Assim, explorando o fato de que os textos a serem analisados são sempre finitos na prática, e que, dada uma aplicação, existe sempre um limite além do qual os recursos dos reconhecedores gerais rarissimamente seriam utilizados, torna-se possível uma simplificação do problema por meio da eliminação das características não-finitas do mesmo.

Isto pode ser feito sem dificuldades através da imposição de valores convenientes para os parâmetros em questão, determinados através de condições de compromisso entre a simplicidade que as limitações impostas podem proporcionar, e a generalidade, oferecida pelo próprio modelo em que se baseia o reconhecedor construido.

Deste modo, torna-se possível elaborar reconhecedores que apresentem eficiência máxima quando aplicados ao reconhecimento de textos de entrada que guardem as limitações impostas, e que, simultaneamente,

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

sejam de total generalidade, permitindo, inclusive, embora às custas de uma degradação de desempenho nos citados casos de menor interesse, a correta análise mesmo no caso de textos que fujam às restrições adotadas.

Para completar o presente trabalho, a versatilidade do modelo apresentado é explorada, através da conceituação de autômatos adaptativos, extensões do modelo que lhe incorporam características mais poderosas, dotando-o, por exemplo, de potencial suficiente para tratar dependências de contexto e linguagens extensíveis.

1.2 Motivações

A maioria dos compiladores dirigidos por sintaxe empregam, nos mecanismos de análise das linguagens, soluções baseadas em máquinas de estados e pilhas, não considerando, ao nível do reconhecimento sintático, diversos importantes aspectos que são, no entanto, de cunho genuinamente sintático.

Entre os principais desses aspectos podem ser destacados aqueles associados à dependência de contexto, às estruturas de blocos e escopo das variáveis, ao tratamento de macros, à consistência de uso dos tipos das variáveis, e à problemática ligada à extensibilidade sintática.

Na prática, na maior parte dos compiladores o tratamento destes aspectos da análise da linguagem se apresenta, desta maneira, ausente das partes do compilador que são autenticamente responsáveis pela análise sintática, manifestando-se, muitas vezes impropriamente, em seções cujo caráter é originalmente semântico.

O presente trabalho expõe os resultados de uma investigação em que se procura obter, a partir de técnicas clássicas simples, mecanismos capazes de integrar, em um único método de análise, os elementos necessários à resolução plena e uniforme de grande parte dos problemas de interesse na análise de linguagens, referentes a aspectos puramente sintáticos.

Com esta intenção, são apresentados os conceitos de autômato e de transdutor adaptativo, modalidades de dispositivos de reconhecimento e transdução sintática derivados do modelo formal descrito em (JOSÉ NETO; MAGALHÃES, 1981a), cujos princípios foram publicados, pela primeira vez, em (JOSÉ NETO, 1988a).

Através do emprego desta classe de dispositivos, abre-se um caminho para o resgate natural do caráter sintático autêntico apresentado pelos diversos problemas em pauta.

A potencial uniformidade de tratamento destes problemas, com o auxílio do novo modelo, proporciona a possibilidade de uma separação mais natural entre as tarefas de análise sintática e as da geração de código, tornando desta forma relativamente simples a automatização da operação de construção de compiladores dirigidos por sintaxe baseados nesta técnica.

Além desta contribuição mais significativa, no presente trabalho ainda se destacam algumas outras, de utilização bastante confortável no ensino e na prática da Engenharia de Computação:

- um método aperfeiçoado de geração automática e eficiente de reconhecedores sintáticos para linguagens livres de contexto por meio do uso direto de uma meta-linguagem derivada da Notação de Wirth (JOSÉ NETO; MAGALHÃES, 1981b), (JOSÉ NETO, 1987) (JOSÉ NETO, 1988a) (JOSÉ NETO; KOMATSU, 1988a)
- um método para o desenvolvimento de analisadores sintáticos eficientes, utilizáveis como núcleos de compiladores dirigidos por sintaxe (MAGALHÃES; JOSÉ NETO, 1983).
- um método para a incorporação de mecanismos de recuperação de erros sintáticos aos reconhecedores obtidos pela utilização dos modelos propostos.

1.3 Objetivos

O presente trabalho tem como meta documentar os principais resultados teóricos e tecnológicos das pesquisas realizadas pelo autor nas áreas de Linguagens Formais e Teoria de Autômatos, aplicados à tarefa da construção de compiladores.

Apresenta inicialmente as conclusões de um trabalho em busca de um método simples, intuitivo e poderoso para a construção automática de reconhecedores sintáticos eficientes, aplicável à mecanização da construção de processadores de linguagens de programação.

A preocupação com o fator simplicidade propiciou a obtenção de resultados que permitem não apenas a construção de reconhecedores eficientes a custos baixos, como também - e isto é de extrema importância pedagógica, não apenas em aplicações acadêmicas como ainda para fins de divulgação e treinamento profissional - o uso didático dos métodos desenvolvidos, mesmo para um público-alvo não-especialista.

Evoluções desta pesquisa levaram ainda à investigação da possibilidade de generalização dos conceitos e metodologias desenvolvidos, com a intenção de torná-los aplicáveis a situações mais complexas, tendo-se obtido, nesta linha, os dois grandes resultados abaixo relatados.

O primeiro projeta-se na área das linguagens extensíveis, com a introdução do conceito de autômatos adaptativos, os quais incorporam a possibilidade de se auto-modificarem ao longo de sua operação, estendendo-se e adaptando-se às necessidades de cada particular texto de entrada que lhes seja submetido.

O segundo resultado da pesquisa se manifesta com maior intensidade na área do tratamento de linguagens que exibem dependências de contexto.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

A existência de construções lingühsticas que apresentam dependências de contexto, fato inevitável nas linguagens de programação da prática, é impropriamente tratada, na maioria dos compiladores reais, fora do âmbito da análise sintática.

Mais uma vez com o auxílio do conceito de autômato adaptativo, este aspecto das linguagens pode passar a ser tratado de acordo com esquemas de projeto relativamente simples, que dispensam o emprego de artifícios, permitindo a sua resolução ao próprio nível da análise sintática da linguagem de entrada.

Aplicações adicionais são ainda sugeridas para as metodologias e técnicas mencionadas, procurando-se desta forma estimular a pesquisa nesta área tão interessante dos Fundamentos da Engenharia de Computação.

1.4 Histórico

O trabalho aqui documentado teve seu desenvolvimento operado em diversas fases. Inicialmente, foram efetuadas extensas pesquisas na literatura, em busca de técnicas e métodos publicados, a partir dos quais se procurou extrair uma forma eficaz e de baixa complexidade para a resolução do problema da análise sintática de linguagens livres de contexto.

Como primeiro resultado desta atividade foi desenvolvido um formalismo e uma metodologia de obtenção de analisadores sintáticos para linguagens descritas através de gramáticas livres de contexto.

Resultaram desta atividade dois artigos, através dos quais foi apresentado um modelo formal estruturado para autômatos de pilha (JOSÉ NETO; MAGALHÃES, 1981a), bem como um algoritmo para a conversão de uma gramática livre de contexto em um autômato desta categoria, que reconhece a linguagem por ela descrita (JOSÉ NETO; MAGALHÃES, 1981b).

A seguir, publicou-se mais um artigo, em que é apresentada uma extensão do modelo, a qual exibe a capacidade de incorporar funções de transdução sintática. Tal artigo fornece um exemplo de aplicação, no qual é apresentada a construção automática, a partir de uma gramática livre de contexto, de um transdutor que produz, a partir do reconhecimento de sentenças da linguagem em questão, a árvore de reconhecimento desta sentença, segundo a gramática inicial (MAGALHÃES; JOSÉ NETO, 1983).

Posteriormente, foi melhorado o método desenvolvido, e uma versão mais prática dos algoritmos de conversão de gramáticas em reconhecedores foi publicado como parte de um livro introdutório sobre compilação, destinado ao uso em cursos de Engenharia de Computação e congêneres (JOSÉ NETO, 1987).

Um aperfeiçoamento dos algoritmos de geração dos reconhecedores sintáticos baseados em autômatos de pilha estruturados encontra-se relatado em (JOSÉ NETO, 1988a) como parte do texto-base utilizado em um mini-curso sobre técnicas de compilação. Este método permite a obtenção direta de reconhecedores quase-ótimos a partir de uma gramática livre de contexto, fornecida em notação adequada.

Prosseguiu-se a pesquisa, em busca do aumento da potência para os modelos formais, de uma facilidade maior na geração automática dos transdutores, e também de uma melhora nos aspectos de simplicidade, eficiência e abrangência para as máquinas produzidas.

Como resultado, obteve-se um grande aumento no poder de representação dos modelos matemáticos empregados, por meio da inclusão de recursos de aprendizado. Os transdutores adaptativos assim idealizados, relatados pela primeira vez de forma intuitiva em (JOSÉ NETO, 1988b), são capazes de se amoldarem a cada particular sentença a ser reconhecida, proporcionando uma forma extremamente natural para o tratamento de diversos dos problemas sintáticos usualmente considerados como parte da manipulação semântica da linguagem.

Assim, além da simples capacidade de reconhecimento livre de contexto, os transdutores adaptativos incorporam recursos para tratar muitos outros problemas sintáticos, tais como dependências de contexto, escopos de identificadores, estruturas de blocos, declarações, definição e expansão de macros em geral, tabelas de símbolos, verificação estática de tipos, geração canônica de código e muitos outros recursos, em geral considerados fora do âmbito sintático, em compiladores convencionais.

A presente tese representa, nesta parte da pesquisa, uma compilação de todos os resultados finais obtidos anteriormente, bem como das últimas realizações em busca da automatização econômica da obtenção de analisadores sintáticos para uma classe significativamente ampla de linguagens de programação de interesse prático.

1.5 Estrutura da tese

O primeiro capítulo desta tese faz uma apresentação da pesquisa a ser relatada, descrevendo as suas principais motivações e finalidades, seu histórico e a descrição da organização física do texto que a compõe.

No segundo capítulo estão descritos os principais conceitos subjacentes à teoria que dá suporte ao restante do trabalho. São apresentadas as notações e termos empregados, resumindo-se em seguida os principais pontos da teoria e da técnica de linguagens formais, autômatos e compilação relevantes a este trabalho, bem como os resultados teóricos da pesquisa realizada.

No terceiro capítulo é tratada a problemática da geração automática de reconhecedores sintáticos a partir de gramáticas livres de contexto, em diversos graus de complexidade e de alcance, cobrindo aspectos variados do

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

problema, desde um aceitador sintático simples até um reconhecedor sintático completo, com recuperação de erros e um transdutor sintático gerador de árvores de reconhecimento de sentenças.

No quarto capítulo são apresentados os autômatos adaptativos e discutida a sua aplicabilidade ao tratamento da extensibilidade sintática e das dependências de contexto, nos processos de reconhecimento de linguagens de programação.

No quinto capítulo, tecem-se considerações sobre as aplicações possíveis deste trabalho, avalia-se o seu conteúdo e os seus produtos e são levantadas as principais conclusões da pesquisa realizada, relatando-se algumas perspectivas para o prosseguimento dos trabalhos desta atividade.

Após os cinco capítulos, figura um grupo de anexos onde podem ser encontrados alguns detalhes, notações e outros formalismos, relevantes à complementação do texto, mas que não foram o centro das atenções da presente publicação.

Em seguida, são apresentadas as referências bibliográficas citadas ao longo do texto e utilizadas na sua elaboração.

Concluindo, sob o título de Bibliografia Recomendada, relaciona-se um conjunto suplementar de referências à literatura, que embora não tenham sido diretamente citados, podem auxiliar o leitor que deseje aprofundar-se em diversos assuntos correlatos, mas não suficientemente cobertos ao longo da tese.

Ao leitor que tenha bons conhecimentos de linguagens formais e autômatos, é desnecessária a leitura da maior parte do conteúdo do capítulo 2 e das partes teóricas do capítulo 3, onde, por razões didáticas, foram expostos os principais conceitos e terminologias da área, para tornar o texto acessível também ao iniciante.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

2 CONCEITOS

Neste capítulo são sucintamente apresentados os principais conceitos da teoria de linguagens, autômatos e compiladores, relevantes à presente pesquisa. Alguns pontos da teoria clássica são recordados resumidamente.

Note-se que não é intenção deste texto dar um tratamento profundo ao assunto, mas somente recordar os aspectos mais ligados a este trabalho, visto que se dispõe de inúmeras publicações de boa qualidade que apresentam estes assuntos de forma muito mais satisfatória do que é possível em umas poucas páginas.

Os assuntos aos quais estão ligadas as principais contribuições deste trabalho serão detalhadas e formalizadas em outros capítulos, sendo aqui apresentados apenas do ponto de vista conceitual.

2.1 Terminologia

Neste item procura-se apresentar um texto que introduz o conjunto básico dos termos mais relevantes ao estudo dos temas desenvolvidos nesta tese.

A teoria das linguagens de programação e da construção dos correspondentes compiladores está fundamentado no estudo das seqüencias de caracteres de que se compõem os textos da linguagem a compilar. Todo texto escrito em alguma linguagem de programação apresenta-se ao seu usuário como uma cadeia de átomos da linguagem, átomos esses que pertencem a um alfabeto, característico da linguagem em questão.

Um conjunto de regras (produções) que determinam as leis de formação de um texto correto da linguagem denomina-se gramática desta linguagem. Uma linguagem, formalmente, seria então o conjunto das sentenças, ou seja, das cadeias de átomos que se possa derivar, ou construir, a partir da aplicação exclusiva das produções da gramática.

Aos átomos que formam as sentenças da linguagem dá-se o nome de terminais. Aos símbolos da gramática, que não correspondem a átomos, mas que designam formas sintáticas, especificadas por meio das produções gramaticais, dá-se o nome de não-terminais.

Para descrever uma linguagem por intermédio de uma gramática, lança-se mão de uma meta-linguagem, notação apropriada para a definição formal dos aspectos da sintaxe, ou seja, da forma dos textos da linguagem, através de um conjunto de regras de substituição.

Gramáticas são, pois, dispositivos de geração de uma linguagem. Outra forma de definição rigorosa da sintaxe de linguagens é através da formalização de um dispositivo que tenha a propriedade de aceitar ou reconhecer qualquer sentença da linguagem, e nenhuma outra. Dispositivos desta natureza são conhecidos como reconhecedores, aceitadores ou autômatos.

Um autômato, que seja dotado de recursos para indicar, para uma sentença reconhecida, qual a seqüencia de derivações a serem aplicadas a uma dada gramática para a obtenção daquela sentença, denomina-se analisador da linguagem em questão, já que determina a estrutura sintática de qualquer sentença da linguagem, com base na gramática mencionada.

Um texto submetido a um autômato ou a um analisador para aceitação denomina-se texto-fonte. Um autômato, enriquecido de recursos que permitam converter um texto-fonte em outro texto, de acordo com um conjunto de regras de mapeamento sintático, denomina-se transdutor sintático. O texto obtido como saída de tal processo de transdução é chamado texto-objeto.

Processos de síntese mais elaborados que os obtidos por transdução sintática utilizam-se de regras de mapeamento baseadas na semântica da linguagem-fonte, ou seja, em informações acerca do significado que cada uma de suas construções sintáticas representa. Se tais regras produzirem como linguagem-objeto programas equivalentes ao representado no texto-fonte, a síntese em questão se denominará compilação. Caso tais regras não gerem um texto-objeto, mas provoquem a execução de ações equivalentes às representadas no programa-fonte, tem-se um processo de interpretação da linguagem-fonte.

Transdutores especiais, capazes de alterar suas regras em função do texto-fonte particular em análise, têm a possibilidade de se configurar a particulares exigências do texto-fonte, representadas pelas chamadas dependências de contexto. As dependências de contexto se manifestam na possibilidade de múltipla interpretação de certas construções sintáticas, em função do contexto em que são empregadas.

Tais dispositivos são denominados transdutores adaptativos, e são capazes de manipular alguns aspectos mais complexos da compilação, tais como a extensibilidade sintática das linguagens, isto é, o recurso que algumas linguagens apresentam de permitir que seu usuário defina e passe a utilizar novas construções sintáticas e suas respectivas interpretações.

2.2 Linguagens regulares e livres de contexto

De maior interesse para o estudo dos processadores de linguagens, a mais simples é a classe das denominadas linguagens regulares, cuja lei de formação de sentenças é muito simples, e baseada exclusivamente na concatenação de símbolos básicos ou de sentenças menores, também regulares, de acordo com regras incondicionais e isentas de aninhamentos sintáticos.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Não tão simples, porém muito convenientes para a maioria das aplicações, são as denominadas linguagens livres de contexto. Isto se deve à sua maior generalidade e semelhança com as linguagens normalmente usadas como linguagens de programação de computadores.

Ao contrário do que ocorre com as linguagens regulares, as linguagens livres de contexto, em sua forma mais geral, embora sejam também definidas exclusivamente por intermédio de regras incondicionais, não podem ser descritas apenas através de concatenações de elementos ou de construções menores, sendo, ao contrário, necessária a inclusão de regras adicionais que descrevam leis de formação para os aninhamentos sintáticos, fartamente encontrados nas linguagens de programação usuais, e cuja presença caracteriza, em relação às linguagens regulares, esta classe mais geral de linguagens.

2.3 Gramáticas lineares e livres de contexto

Gramáticas, no sentido mais abrangente, são dispositivos que descrevem linguagens através de mecanismos de geração. Para tanto, lançam mão de leis de formação, baseadas em produções, que são regras de substituição.

Assim, partindo-se de um símbolo inicial diferenciado, a raiz da gramática, e aplicando-se sucessivas substituições, indicadas pelas produções da gramática, vão sendo obtidas as correspondentes formas sentenciais da linguagem, até que não mais reste qualquer não-terminal na forma sentencial, a qual se denominará, nesta situação, uma sentença da linguagem em questão.

Produções são pares ordenados do tipo x ::= y , e se x for obrigatoriamente representado por um não-terminal isolado, com y sendo uma cadeia qualquer de terminais e/ou não-terminais, a gramática assim construida estará especificando apenas substituições incondicionais de uma ocorrência arbitrária do não-terminal x, na forma sentencial, pela nova cadeia, y.

Por serem permitidas qualquer que seja a situação de ocorrência de x, estas substituições não dependerão do contexto em que x possa estar imerso, razão pela qual tal gramática é também denominada livre de contexto, o mesmo ocorrendo com a linguagem por ela gerada.

Pode-se construir linguagens livres de contexto particulares, em que todas as produções especifiquem substituições restritas de não-terminais, nas quais os eventuais novos não terminais que venham a surgir resultem sempre posicionados em uma mesma extremidade (ou direita ou esquerda) da forma sentencial.

Para que isto seja possível, a cadeia que especifica, na produção gramatical, a substituição a ser realizada, deve constar de uma seqüencia de terminais (eventualmente vazia), opcionalmente iniciada ou terminada por um não-terminal, conforme a extremidade em que os novos não-terminais da forma sentencial devem surgir.

A essa classe de gramáticas dá-se o nome de gramáticas regulares ou lineares, à direita ou à esquerda, respectivamente.

As linguagens livres de contexto que tais gramáticas produzem são denominadas linguagens regulares.

Convém notar que gramáticas regulares não são capazes de representar construções aninhadas, em contraste com as gramáticas livres de contexto mais gerais, sendo esta a diferença mais marcante da potencialidade das duas classes de gramáticas.

Dentre as linguagens de maior interesse para o estudo das técnicas de construção de compiladores, as linguagens regulares e as linguagens livres de contexto são aquelas que maior importância apresentam, dada a sua simplicidade e simultânea generalidade para a representação da maior parte dos aspectos sintáticos das linguagens reais de programação.

Inúmeras notações têm sido propostas como meta-linguagens para serem utilizadas na formalização de linguagens regulares e livres de contexto. Entre as mais significativas, é inegável a popularidade da famosa BNF ("Backus-Naur form"), utilizada originalmente na definição do Algol 60 (NAUR, 1963) e posteriormente estendida para a forma da notação empregada por Wirth na definição da linguagem Modula-2 (WIRTH, 1988).

No presente trabalho, será utilizada uma meta-linguagem derivada da notação de Wirth, que será denominada Notação de Wirth Modificada, e que apresenta diversas características que se mostram muito convenientes para o desenvolvimento dos métodos a serem apresentados.

Entre os efeitos derivados destas características, os mais importantes são a simplificação dos métodos de construção de reconhecedores sintáticos a partir da gramática, bem como a oportunidade que proporcionam para a obtenção direta de reconhecedores compactos e eficientes.

A Notação de Wirth e a Notação de Wirth Modificada são descritas mais rigorosamente no Anexo A.

2.4 Autômatos finitos e de pilha

De forma análoga ao que ocorre com as gramáticas, é possível também a representação de linguagens através de dispositivos aceitadores de vários tipos. Assim, em correspondência com as diversas classes de gramáticas, pode-se identificar classes equivalentes de autômatos e transdutores.

Desta maneira, para a aceitação de linguagens regulares, dispõe-se de um tipo muito simples de máquinas, as denominadas máquinas de estados finitos ou autômatos finitos. Trata-se de aceitadores baseados em transições

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

de estados, transições estas promovidas com base na identificação dos elementos básicos que compõem a cadeia correspondente à sentença em análise.

Autômatos finitos são aceitadores que não dispõem de qualquer memória auxiliar que suplemente a informação contida no estado corrente do autômato.

Com tal restrição, autômatos finitos são incapazes de tratar aninhamentos, e deste modo as formas mais gerais de linguagens livres de contexto não são manipuláveis através desta classe de reconhecedores.

Com a finalidade de permitir que os aceitadores tratem aninhamentos, e portanto as linguagens livres de contexto em sua forma mais ampla, torna-se necessário incorporar aos autômatos finitos um elemento de armazenamento, que é geralmente organizado em forma de pilha, e com esta adição cria-se o modelo do autômato de pilha, que torna viável o tratamento de quaisquer linguagens livres de contexto (JOSÉ NETO; MAGALHÃES, 1981a).

Enriquecendo-se os autômatos através da associação de funções de saída às suas transições, obtêm-se os correspondentes transdutores, que representam uma classe restrita de tradutores, mas que podem ser empregados como núcleos de compiladores, estes certamente muito mais complexos.

Cabe mencionar que a tecnologia do projeto e desenvolvimento de autômatos e transdutores permite, para o caso dos autômatos finitos, que sejam obtidos os aceitadores ótimos, ou seja, aqueles que apresentam o número mínimo de estados e transições, e também máximo desempenho, para tanto bastando que a eles sejam aplicadas regras clássicas de projeto e otimização.

No Anexo B são apresentados os mecanismos formais para a representação de linguagens, utilizados nas formalizações apresentadas nesta tese.

2.5 Autômatos de pilha estruturados

Para autômatos e transdutores de pilha, não se dispõe de métodos práticos que conduzam, em todos os casos possíveis, a resultados absolutos neste sentido, existindo, entretanto, técnicas muito boas que permitem obter, de forma econômica, versões sub-ótimas de tais máquinas, as quais exibem, entretanto, excelente desempenho.

Nesta linha, o formalismo aqui apresentado de autômato de pilha, inspirado no modelo clássico proposto em (CONWAY, 1963), popularizado em (BARNES, 1972) e posteriormente formalizado em (LOMET, 1973), fornece o substrato adequado para a busca de soluções eficientes, e é chamado autômato de pilha estruturado.

O modelo se compõe essencialmente de um conjunto de autômatos finitos, denominados sub-máquinas do autômato de pilha estruturado, a cada um dos quais cabe a tarefa de efetuar o reconhecimento de uma das diferentes classes de sub-cadeias que compõem a cadeia em análise.

Transições ocorrem internamente a tais sub-máquinas, com ou sem consumo de átomos da cadeia de entrada, e com ou sem "look-ahead". Adicionalmente, é possível efetuar transições entre sub-máquinas de forma análoga ao que ocorre quando se efetua uma chamada ou um retorno de sub-rotina em um programa, sendo utilizada para isto, nestas ocasiões, uma pilha como elemento de armazenamento de informações.

Com tais componentes, caracteriza-se uma situação do autômato em um determinado instante do reconhecimento como sendo a posição conjunta do conteúdo da pilha, do estado e da sub-máquina correntes, e da parte da cadeia de entrada que ainda não foi analisada naquele instante do processamento.

Diz-se que o autômato de pilha estruturado reconhece uma sentença quando, partindo de uma situação inicial, definida como aquela em que toda a cadeia de entrada ainda está para ser analisada, a pilha, vazia e o autômato, em um estado inicial pré-determinado, a máquina atinge uma situação final, com a pilha vazia, a cadeia de entrada esgotada e o estado corrente sendo um dos chamados estados finais pré-estabelecidos, após efetuar uma seqüencia de transições, ditada por uma função de transição, característica da máquina.

Neste esquema, os aninhamentos podem ser tratados sempre através da execução de chamadas de sub-máquinas, e posteriormente dos respectivos retornos à sub-máquina chamadora. No caso de aninhamentos sucessivos de um mesmo tipo de construção sintática, torna-se necessária a operação recursiva da correspondente sub-máquina, sendo que cada instância da ativação de tal sub-máquina corresponderá, neste caso, à presença de um nível adicional de aninhamento sintático no texto de entrada.

Como principal propriedade deste tipo de reconhecedores, é importante destacar a alta eficiência que deles se pode obter, uma vez que operam essencialmente como autômatos finitos - exceto nas ocasiões de movimentação da pilha, obrigatória apenas ao início e ao final do tratamento de construções sintáticas que exibam potenciais aninhamentos - durante portanto uma parcela substancial do trabalho de reconhecimento da grande maioria das linguagens usuais de programação.

Levando-se em conta que, para os autômatos finitos que constituem as várias sub-máquinas, é sempre possível obter uma versão ótima, e portanto determinística, conclui-se ser possível, em todos os casos implementados com tal tipo de aceitador, efetuar o reconhecimento de qualquer sentença da linguagem em tempo proporcional ao comprimento da mesma, acrescido de outra parcela, proporcional ao número de aninhamentos exibido por tal sentença.

O Anexo C apresenta uma formalização dos autômatos de pilha estruturados. No Anexo E encontra-se a descrição de uma forma de mapeamento de gramáticas em autômatos de pilha estruturados.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

2.6 Árvores de derivação e de reconhecimento

Em muitas situações, a simples aceitação de uma sentença por um reconhecedor, ou então a constatação da viabilidade de obtenção da mesma através da aplicação das regras de uma gramática é insuficiente para que se possam efetuar diversas operações tipicamente encontradas em atividades ligadas à compilação.

Torna-se necessário, em tais casos, conhecer os passos percorridos pelo reconhecedor ou pela gramática em sua operação, bem como executar uma interpretação, passo a passo, das transições do reconhecedor ou das derivações efetuadas pela gramática.

Uma forma bastante freqüente de representação de seqüencias de derivação ou de reconhecimento sintático pode ser identificada nas árvores de sintaxe da sentença, desenvolvidas com base em uma gramática que descreve a linguagem.

Estas árvores descrevem com rigor a estrutura sintática da sentença que denotam, através da representação do relacionamento existente entre os diversos não-terminais surgidos nas várias formas sentenciais intermediárias que dão origem à sentença a partir da raiz da gramática, por meio da aplicação das regras de substituição que a gramática representa.

Quando construídas a partir da derivação de sentenças, as árvores de sintaxe são denominadas árvores de derivação. São muitas vezes obtidas, entretanto, como efeito secundário da operação de reconhecedores sobre as sentenças.

Neste caso, as árvores são construidas com base em uma gramática da linguagem, e em função dos passos de reconhecimento sintático executados pelos mecanismos de aceitação, sendo então denominadas árvores de reconhecimento das sentenças a que se referem.

Caso um aceitador sintático, ou autômato, seja enriquecido com mecanismos de geração de árvores de sintaxe para as sentenças que lhe são submetidas, tal dispositivo passa a operar como um analisador sintático da linguagem aceita pelo reconhecedor em questão. Analisadores sintáticos são peças fundamentais da estrutura da maioria dos compiladores usuais. As árvores de reconhecimento assim obtidas são muito úteis para os algoritmos de geração de código em muitos compiladores desta categoria.

No Anexo D é apresentado um formato para a representação de árvores, conveniente para as aplicações descritas nesta tese.

No Cap. 3.3 é apresentada a descrição de um método para a construção de geradores de árvores sintáticas, denotadas neste formato.

2.7 Não-determinismos e ambigüidades

Um aspecto importante do estudo de linguagens e de suas representações formais é o da *ambigüidade*. Inspecionando-se algumas gramáticas, é possível nelas identificar casos em que uma dada sentença da linguagem pode ser gerada através de duas ou mais diferentes árvores válidas de derivação.

Gramáticas que se comportam desta forma são ditas *ambíguas*. Analisadores que tratam linguagens definidas por meio de gramáticas ambíguas devem ser capazes de gerar todas as árvores de sintaxe que representem as diversas possíveis derivações das sentenças da linguagem em questão pela gramática.

Outra consideração pode ser feita acerca do *determinismo*: um reconhecedor é dito determinístico quando, qualquer que seja a situação em que se encontrar, for único o movimento que qualquer símbolo da cadeia em estudo possa provocar no reconhecedor. Em autômatos finitos é sempre possível, através da eventual criação de novos estados e transições, transformar em determinístico qualquer aceitador não-determinístico. Já em autômatos de pilha esta garantia não existe, havendo mesmo linguagens para as quais não é sequer possível construir um único reconhecedor determinístico (HARRISON, 1978), (LEWIS; PAPADIMITRIOU, 1981).

Autômatos não-determinísticos fornecem reconhecedores para os quais a trajetória de reconhecimento de uma sentença nem sempre é única, exigindo que sua operação seja, muitas vezes, efetuada por meio de tentativas e erros ("*backtracking*"). Isto acarreta uma substancial perda de eficiência do reconhecedor, tornando anti-econômico o seu uso em aplicações práticas.

Ao contrário do que ocorre no caso dos autômatos finitos, os reconhecedores não-determinísticos baseados em autômatos de pilha são mais poderosos que os determinísticos, o que lhes confere a capacidade de tratar uma classe mais abrangente de linguagens livres de contexto (LEWIS; PAPADIMITRIOU, 1981), (HARRISON, 1978), (WULF et al, 1981).

2.8 Simplificação e otimização de reconhecedores

Sempre que a eficiência se apresenta como requisito imperativo de um projeto que envolva, como mecanismo básico, um reconhecedor sintático, é necessário garantir que o autômato que o implementa seja construido com o suficiente cuidado para que não apresente ineficiências decorrentes da presença de redundâncias, de estados equivalentes ou inacessíveis, do uso abusivo da memória auxiliar do reconhecedor, ou ainda da presença de transições em vazio ou não-determinísticas desnecessárias.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Os métodos usuais de obtenção de reconhecedores, entretanto, raramente levam à criação direta de autômatos ótimos, tornando-se necessário, uma vez de posse de uma versão inicial do autômato desejado, manipulá-lo no sentido de isentá-lo das causas de eventuais ineficiências apresentadas.

Assim, antes de efetuar a otimização propriamente dita do autômato em questão, é conveniente prepará-lo, simplificando sua estrutura através das transformações seguintes:

- *eliminação de estados inacessíveis*, que correspondem a partes do mesmo que seguramente jamais serão utilizadas
- *eliminação de ciclos de transições em vazio*, correspondentes a conjuntos de estados entre os quais o trânsito é livre, conjuntos estes formados portanto de estados equivalentes
- *eliminação das transições em vazio* remanescentes, que tornam não-determinística a operação do autômato
- *eliminação de não-determinismos adicionais*, causados pela presença, em alguma possível situação do autômato, de múltiplas transições lícitas como reação a um dado estímulo
- *eliminação de estados equivalentes* que tenham restado, ou seja, estados que não possam ser distinguidos entre si pela aplicação de uma seqüencia finita de estímulos.

A atividade de obtenção da versão ótima para um reconhecedor pode ser realizada com sucesso para qualquer autômato finito, conforme garante a teoria clássica dos autômatos, existindo uma grande variedade de algoritmos que desempenham esta tarefa, publicados na literatura.

Para reconhecedores de linguagens livres de contexto, entretanto, não se dispõe de tal garantia teórica, sendo assim necessário lançar mão de recursos através dos quais sejam obtidas versões sub-ótimas, que exibam entretanto um desempenho satisfatório (LEWIS; PAPADIMITRIOU, 1981), (TREMBLAY; SORENSON, 1985).

Em particular, no caso de autômatos de pilha estruturados, é possível construir um reconhecedor, para uma dada linguagem livre de contexto, em que seja minimizado o número de suas sub-máquinas, e otimizados os diversos autômatos finitos que as implementam, obtendo-se, como conseqüencia, uma máquina em que todas as transições internas às sub-máquinas consumam um símbolo de entrada, e onde a pilha seja utilizada estritamente nos casos em que houver ocorrências de construções aninhadas.

Para máquinas assim elaboradas, pode-se constatar que exibem um tempo de execução do reconhecimento proporcional ao comprimento da cadeia de entrada. Isto é um resultado excelente, dada a vasta aplicabilidade do método às linguagens de programação de maior interesse, e do fato de que muitos importantes reconhecedores publicados na literatura exibem comportamentos sensivelmente inferiores.

Em (AHO; ULLMAN, 1972), (BARRETT; COUCH, 1979), (LEWIS; ROSENKRANTZ; STEARNS, 1976), (JOSÉ NETO, 1987) e muitos outros textos clássicos podem ser encontrados diversos algoritmos para a simplificação e otimização de autômatos finitos. No Anexo F encontra-se uma formalização do estudo de desempenho dos reconhecedores sintáticos livres de contexto tratados nesta tese.

2.9 Recuperação de erros

Autômatos são dispositivos preparados para a tarefa de aceitação de sentenças, ou seja, de cadeias corretas da linguagem que descrevem. Assim sendo, caso forem alimentados com cadeias que não sejam sentenças, não atingirão uma situação final de aceitação para tais cadeias.

O fluxo de transições entre as sucessivas situações do autômato é contínuo entre as situações inicial e final do reconhecedor quando este estiver tratando uma sentença da linguagem. No entanto, tal fluxo será interrompido ou truncado sempre que for tratada uma cadeia que não seja uma sentença.

Neste caso, o reconhecimento poderá fluir normalmente enquanto estiver sendo analisada uma parte da cadeia que ainda possa vir a formar sentenças, dessincronizando-se assim que isto deixar de ocorrer. A tal circunstância é dado o nome de *manifestação de um erro* contido na cadeia de entrada.

Note-se que o ponto da cadeia de entrada em que ocorre a manifestação de um erro não corresponde, em geral, ao ponto em que se dá realmente a ocorrência do erro na cadeia. Isto se deve, em geral, a uma camuflagem do erro verdadeiro, que leva o autômato a alguma seqüencia de transições que seja incapaz de conduzir o reconhecedor à aceitação da cadeia proposta.

À atividade de localização, ou de tentativa de localização da manifestação de um erro pelos mecanismos de reconhecimento dá-se o nome de detecção do erro. Quase sempre esta tarefa leva a resultados apenas aproximados, devido à inerente impossibilidade de determinação da real intenção original do usuário, ao redigir incorretamente sua suposta sentença.

Corrigir automaticamente um texto incorreto, tarefa executada por alguns compiladores, torna-se assim uma atividade empírica, cujos resultados em sua maioria não correspondem à realidade, salvo em casos muito particulares.

Mais prática se torna a idéia de, ao invés de tentar a correção do texto de entrada, procurar-se apenas a ressincronização do autômato após a manifestação de um erro. Embora não se sirva para reparar o texto de entrada, este método leva à condução forçada do autômato para alguma situação a partir da qual possa ele prosseguir no reconhecimento da cadeia restante, sem a necessidade de reiniciar a atividade já executada, permitindo assim a detecção de erros adicionais no texto.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Muitos métodos de recuperação de erros existem, sendo a maioria de caráter bastante específico, e voltados para um particular tipo de reconhecedor. De forma geral, a maioria se baseia na ressincronização do autômato, supondo que o erro detectado seja um erro simples, isto é, que a manifestação do erro em questão seja acarretada por um único erro real, e não pela superposição dos efeitos de diversos erros.

Há três classes de erros simples possíveis em um texto: os erros causados pela inserção indevida de um símbolo no texto, os erros devidos à remoção indevida de um símbolo do texto, ou então os erros causados pela substituição de um símbolo do texto correto por algum outro símbolo. Para estes três casos é possível ampliar os autômatos finitos, de forma que possam ser automaticamente reconduzidos a estados convenientes para o prosseguimento da análise. Este método é aplicável em grandes porções das sub-máquinas que compõem os autômatos de pilha estruturados.

Os novos autômatos, uma vez incorporados os mecanismos de recuperação de erros, tornam-se muito mais complexos que os originais. Para o caso de erros múltiplos, a complexidade do problema aumenta mais ainda, devido à possibilidade de uma grande variedade de combinações entre os erros que podem ocorrer.

Por esta razão, em geral não se procura resolver deterministicamente este tipo de problema, lançando-se mão, ao invés, de heurísticas de recuperação para o tratamento de tal classe de erros. Entre as heurísticas mais populares, destaca-se a chamada "panic-mode": descarta-se parte do texto de entrada ainda não analisado, à procura de algum símbolo de sincronização no texto, forçando-se então o autômato a assumir alguma situação conveniente, a partir da qual o reconhecimento possa prosseguir.

Embora adequado ao tratamento de situações mais complexas de erros, o método acima é largamente empregado na prática também para a recuperação de erros simples em numerosos compiladores comercialmente disponíveis, em razão de sua extrema simplicidade e aplicabilidade, e a despeito de não permitir uma precisão maior na localização e no diagnóstico de erros, bem como por introduzir, por omissão, eventuais erros adicionais no texto de entrada, devidos à prática da eliminação de partes deste mesmo texto, por parte dos mecanismos de recuperação.

Uma formalização completa do problema da recuperação de erros sintáticos em reconhecedores baseados em autômatos finitos e em autômatos de pilha estruturados pode ser encontrado no capítulo 3.2 desta tese.

2.10 Linguagens extensíveis e mecanismos de extensão

Outro conceito relevante ao estudo de linguagens refere-se à sua eventual característica de extensibilidade sintática. Trata-se dos recursos que determinadas linguagens oferecem ao usuário, através dos quais é possível alterar a própria estrutura sintática da linguagem, de forma tal que o usuário possa personalizá-la de acordo com suas necessidades e preferências específicas.

Uma linguagem que merece destaque pelas facilidades com que permite ao seu usuário criar extensões sintáticas é o ALGOL 68 (Van WIJNGAARDEN et al, 1975). Esta linguagem foi implementada a partir de um núcleo básico sobre o qual, através do uso recorrente de extensões, a linguagem completa é instalada. Para o usuário, ainda é possível criar, pelos mesmos mecanismos, extensões sintáticas adicionais, configurando-a às suas necessidades.

As *linguagens extensíveis* são capazes, portanto, de dar ao usuário, em maior ou menor escala, a possibilidade de definir plenamente seus próprios comandos, declarações ou estruturas de dados. Em geral, uma linguagem extensível se apresenta como uma linguagem básica completa, *dita linguagem hospedeira*, dotada de uma meta-linguagem com sintaxe aderente, através da qual as extensões podem ser definidas.

O mecanismo mais primitivo disponível para a extensão de uma dada linguagem de programação, mas que geralmente não caracteriza, no entanto, linguagens autenticamente extensíveis, manifesta-se através da possibilidade de se declarar e utilizar sub-rotinas e funções nos programas descritos por meio da linguagem. São raras as linguagens que não exibem tal recurso.

Mais adequado ao propósito da extensibilidade das linguagens, pode-se destacar o recurso oferecido pelas declarações e chamadas de macros, em suas formas paramétricas ou não. Com o emprego de macros em linguagens de alto nível, é possível dar ao programador a possibilidade de criar abreviaturas convenientes para textos sintáticos ou seqüencias, que se mostrem muito freqüentes ou repetitivos em um dado programa (COLE, 1976), (KERNIGHAN; PLAUGER, 1981), (WEGNER, 1968), (MAGINNIS, 1972).

Recursos de extensibilidade ainda mais autêntica são aqueles proporcionados pelas *macros sintáticas*, onde formas especiais de declaração tornam possível a criação de construções sintáticas não disponíveis originalmente na linguagem, e informar ao compilador acerca das formas de interpretação a elas associadas (COLE, 1976), (LEAVENWORTH, 1966), (Van WIJNGAARDEN et al, 1975). Embora muito potente, este recurso não tem sido incluido nas linguagens de programação mais modernas.

Por outro lado, a definição de *tipos abstratos de dados* tornou-se um recurso freqüente nas boas linguagens atualmente disponíveis, sendo a mais importante representante das manifestações de extensibilidade nas linguagens modernas de programação (MARCOTTY; LEDGARD, 1986), (MacLENNAN, 1983), (TENNENT,1981), (GHEZZI; JAZAYERI, 1982).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Desejando-se definir formalmente uma linguagem extensível, deve-se considerar que os mecanismos de extensão que ela apresenta podem ser vistos como resultados de duas componentes. Na primeira, são definidas, na linguagem de programação, as diversas formas dos comandos e declarações que são responsáveis por fornecer ao programador meios para definir as extensões particulares que lhe sejam convenientes. Na outra, é definido o mapeamento entre as extensões declaradas pelo programador e sua correspondência com a linguagem original.

Em outras palavras, é necessário que se estabeleça primeiramente a forma através da qual o programador pode definir novas construções na linguagem hospedeira, ou seja, a sintaxe da meta-linguagem a ser empregada pelo programador para definir as suas particulares extensões da linguagem.

Em geral, as extensões definidas pelo usuário não são apenas formas sintáticas sem interpretação, mas indicam também o significado a ser adotado para cada possível instância da sintaxe em questão.

Desta maneira, torna-se necessário estabelecer a maneira através da qual a correspondência declarada pelo usuário entre a nova sintaxe e a antiga seja incorporada aos mecanismos de reconhecimento e interpretação da linguagem extensível.

Note-se que certas linguagens extensíveis não se limitam a permitir simples ampliações à sintaxe da linguagem hospedeira básica, mas dão ao usuário, inclusive, a possibilidade de alterar a interpretação de construções sintáticas existentes ou pré-definidas, caracterizando assim uma forma de sintaxe dinâmica para a linguagem.

Uma vez incorporadas as extensões à linguagem hospedeira, obtém-se uma nova linguagem, estendida, e o formalismo que define tal linguagem deverá ser composto a partir da definição da linguagem hospedeira e de informações extraidas das novas regras, indicadas pelo próprio programador, quando da sua caracterização das particulares extensões desejadas para a linguagem.

Isto indica que, durante a análise de um texto redigido em linguagem extensível, os próprios mecanismos de análise da linguagem hospedeira devem ser capazes de se adaptar à análise e à interpretação das extensões definidas pelo programador.

Isto exige dos mecanismos originais de análise da linguagem hospedeira a capacidade de se auto-modificar, em função do conteúdo do texto-fonte que se lhe é submetido, de acordo com os mecanismos de extensão fornecidos pela linguagem e com as particulares extensões definidas pelo usuário.

Torna-se conveniente utilizar, desta forma, na definição formal de linguagens extensíveis, formalismos também extensíveis, capazes de descrever a forma através da qual se dá a assimilação de informações novas aos mecanismos básicos de análise, adaptando-se estes, desta forma, à particular realidade de cada texto-fonte submetido como entrada ao reconhecedor.

Os autômatos adaptativos, adiante apresentados, atendem a esta exigência, mostrando-se por essa razão uma opção natural para a solução desta classe de problemas, sendo por isso ideais como base conceitual na elaboração de substratos para os compiladores deste tipo de linguagens.

2.11 Dependências de contexto e sua formalização

Uma característica apresententada pela esmagadora maioria das linguagens de programação de alto nível é a da *dependência de contexto*. Em contraposição às linguagens livres de contexto, tais linguagens geralmente se definem através de regras de substituição eventualmente condicionais, que restringem as situações de contexto nas quais podem ser permitidas as substituições correspondentes.

Desta maneira, enquanto as linguagens livres de contexto podem ser representadas através de um conjunto de regras de substituição aplicáveis diretamente e em qualquer ordem sobre os não-terminais, independentemente de sua localização física, as dependentes de contexto exigem, antes de uma substituição, um teste para verificar a aplicabilidade da regra de substituição em cada caso particular.

Nas linguagens usuais de programação, as dependências de contexto se manifestam das mais diversas formas, tais como:

- na delimitação de escopos para os identificadores, no sentido de garantir que cada identificador seja utilizado estritamente em regiões do programa determinadas pelas regras de escopo impostas pela linguagem
- nas possíveis interpretações múltiplas de um mesmo símbolo, que pode ser interpretado diferentemente conforme o ponto do programa em que for utilizado
- no uso de identificadores que tenham sido previamente declarados, garantindo que sua utilização seja coerente com a respectiva declaração.
- em linguagens de programação extensíveis, onde a utilização de mecanismos de extensão para introduzir extensões na linguagem ou para explorar extensões previamente definidas manifestam uma particular classe de dependências de contexto

As meta-linguagens usualmente empregadas na definição de linguagens livres de contexto, como é o caso do BNF e de meta-linguagens similares, não dispõem de recursos suficientes para representar tais características da linguagem, exigindo-se desta maneira a utilização de um formalismo com mais recursos, como o que se encontra nas *gramáticas de atributos* (TREMBLAY; SORENSON, 1985), (FISCHER; LeBLANC, 1988) ou nas *gramáticas de dois níveis* (PAGAN, 1981), (McGETTRICK, 1980), (Van WIJNGAARDEN et al, 1975).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Nas implementações usuais de linguagens de programação, as dependências de contexto são, quase sempre, consideradas em separado do formalismo livre de contexto da linguagem, de forma que, menos complexa, esta versão simplificada da linguagem original possa ser manipulada através de mecanismos de análise baseados em autômatos finitos ou de pilha.

Insuficientes para a manipulação de todos os aspectos da linguagem original, estes autômatos se encarregam apenas da sintaxe básica, restando os demais detalhes para tratamento a parte, efetuado em geral pelas rotinas encarregadas do tratamento semântico da linguagem.

Reconhecedores adaptativos podem ser empregados no tratamento integrado dos aspectos livres de contexto e das dependências de contexto da linguagem, mostrando-se portanto uma ferramenta muito conveniente para que seja dado a tal reconhecimento um caráter puramente sintático.

Desta maneira, é possível afirmar que a introdução dos reconhecedores adaptativos permite incorporar ao reconhecimento sintático usual uma nova componente, capaz de levar em consideração muitos aspectos da semântica estática e da sintaxe dinâmica das linguagens de programação, tradicionalmente tratados fora da esfera sintática.

Dependências de contexto são exibidas pela quase totalidade das linguagens de programação existentes, uma vez que refletem diversas propriedades e características encontradas com grande freqüencia nas linguagens da prática.

Como já foi comentado, tais características não podem ser descritas através de formalismos livres de contexto, já que exigem mecanismos mais poderosos, voltados ao tratamento da interdependência entre elementos pertencentes a construções sintáticas aparentemente estanques, como ocorre, por exemplo, em um programa, entre as declarações de identificadores e os comandos que utilizam os objetos a eles associados.

Entre as dependências de contexto mais freqüentemente encontradas nas linguagens de programação destacam-se:

- aquelas que surgem devido à estrutura de blocos da linguagem, relacionada com o escopo dos identificadores
- a associação de atributos aos identificadores quando da sua declaração, exigindo uma subseqüente verificação da coerência de sua utilização nos comandos que os referenciam
- a interpretação múltipla de símbolos conforme o contexto em que são empregados, como ocorre, por exemplo, no caso dos símbolos especificadores de formatos de transferência de dados, frente aos seus eventuais homônimos, referentes a objetos declarados pelo programador.

Para descrever estes aspectos das linguagens, a literatura oferece diversas classes de notações ou meta-linguagens, sendo que se destacam, entre outras, as gramáticas de dois níveis, como é o caso, por exemplo, das gramáticas W, os formalismos baseados em semântica operacional, como ocorre no caso da notação VDL, e as gramáticas de atributos (PAGAN, 1981), (McGETTRICK, 1980).

As *gramáticas W*, idealizadas por Van Wijngaarden (Van WIJNGAARDEN et al, 1975), são formalismos gramaticais de dois níveis, que permitem a especificação de um número infinito de regras de substituição para definir uma linguagem.

O primeiro nível essencialmente se encarrega de especificar a lei de formação do conjunto de regras da geração da linguagem, enquanto o segundo nível, representado pelas regras especificadas pelo primeiro, encarrega-se da geração propriamente dita das sentenças da linguagem (BAUER; EICKEL, 1976).

A notação *VDL*, desenvolvida pela IBM nos seus laboratórios em Viena (McGETTRICK, 1980), (PAGAN, 1981), corresponde a uma espécie de linguagem de programação em máquina abstrata, que permite desenvolver, através do paradigma de *semântica operacional*, especificações formais de interpretadores e compiladores de linguagens de programação, apresentando recursos para a especificação do relacionamento entre os diversos elementos da linguagem e os seus atributos declarados.

Gramáticas de atributos (TREMBLAY; SORENSON, 1985), (GOUGH, 1988), (KASTENS; HUTT; ZIMMERMANN, 1982), (PITTMAN; PETERS, 1992), (RECHENBERG; MÖSSENBÖCK, 1989) são formalismos capazes de efetuar as associações entre os objetos da linguagem e os atributos declarados para os mesmos. Englobam toda uma classe de meta-linguagens empregadas para a definição formal de linguagens dependentes de contexto.

Dos mais recentes trabalhos publicados acerca da formalização de linguagens dependentes de contexto, para aplicação prática à construção de compiladores e similares, destaca-se a presença de formalismos gramaticais ditos genericamente *gramáticas adaptáveis*.

A exemplo das gramáticas W, estas apresentam uma componente variável segundo o texto de entrada, e também buscam superar a necessidade do emprego da semântica estática na formalização da sintaxe das linguagens dependentes de contexto.

Um excelente trabalho de compilação das publicações sobre este tema foi publicado em fins de 1990 (CHRISTIANSEN, 1990), apresentando uma visão do estado da arte na época, além de informações técnicas introdutórias ao assunto.

De particular interesse ao presente trabalho são as *gramáticas evolutivas* (CABASINO; PAOLUCCI; TODESCO, 1992) e as *gramáticas modificáveis* (BURSHTEIN, 1990a e 1990b), resultantes de trabalhos paralelos

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ao apresentado nesta tese, e que guardam, embora sob outra ótica, uma relação muito próxima do formalismo aqui apresentado.

Finalmente, encontram-se trabalhos publicados sobre a geração incremental de reconhecedores sintáticos, com o objetivo de dar apoio a ambientes interativos de formalização de linguagens (HEERING; KLINT; REKERS, 1989), que relatam avanços na utilização interativa incremental de métodos clássicos de reconhecimento sintático para linguagens livres de contexto, instanciando mais uma vez a tendência rumo à utilização de formalismos dinâmicos.

2.12 Autômatos de pilha adaptativos

Os autômatos de pilha adaptativos são dispositivos reconhecedores que exibem a estrutura básica dos autômatos de pilha, mas que incorporam, em adição, mecanismos capazes de efetuar alterações dinâmicas em sua própria estrutura, em resposta a cada particular texto de entrada analisado.

Estas auto-modificações são feitas de forma tal que resultem na incorporação, ao próprio reconhecedor sintático, de informações que lhe permitam tratar aspectos dinâmicos da sintaxe, tais como os representados pelos atributos e escopos dos identificadores, pelos mecanismos de extensão, macros, tipos, etc.

No seu formalismo, os autômatos de pilha adaptativos são definidos por regras que associam um conjunto inicial de transições de estados do autômato, às correspondentes indicações acerca da forma de alteração do próprio conjunto de regras.

À medida que as regras que definem o autômato vão sendo executadas, as respectivas ações de alteração promovem a inclusão ou a remoção de estados e de regras de transição no autômato, proporcionando sua alteração estrutural, em conformidade com o texto-fonte analisado.

Isto incorpora ao autômato de pilha uma característica adaptativa, que o torna capaz de memorizar na sua própria estrutura informações inicialmente ausentes, mas que, uma vez assimiladas, dotam o autômato da capacidade de tratar aspectos da sentença que não poderiam ser considerados em reconhecedores livres de contexto.

Se a um autômato adaptativo forem acrescentados mecanismos encarregados de, a cada transição efetuada, gerar uma cadeia de elementos de um alfabeto de saída pré-determinado, obtém-se como resultado um *transdutor* para a linguagem definida pelo autômato.

Se forem acrescentados meios para que as próprias alterações sofridas pelo autômato possam incorporar também eventuais gerações de cadeias de saída, tem-se caracterizado um *transdutor adaptativo*.

Naturalmente, a aplicação mais intuitiva deste tipo de transdutores é na geração de código-objeto para a linguagem definida pelo formalismo. Se este for o caso, os transdutores adaptativos deverão, no seu caso geral, encarregar-se de todos os aspectos da tradução da linguagem que se refiram às dependências de contexto, à sintaxe dinâmica e à transdução sintática.

Em outras palavras, os transdutores adaptativos deverão exibir mecanismos capazes de representar desde os aspectos sintáticos estáticos, livres de contexto, da linguagem, passando pela resolução dos problemas acarretados pelas dependências de contexto e pelo possível caráter extensível exibido pela linguagem, até os aspectos de sua semântica estática, relativos à geração de código-objeto como resultado da interpretação sintática do texto-fonte.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

3 CONSTRUÇÃO DE RECONHECEDORES LIVRES DE CONTEXTO

O objetivo deste capítulo é apresentar uma forma de automatização para a tarefa da construção de reconhecedores sintáticos clássicos que possam ser utilizados na prática como núcleos básicos de processadores de linguagens de programação.

Para isto, uma versão livre de contexto da linguagem deve estar descrita na forma de gramática, de modo que possa ser empregada como especificação formal de um reconhecedor livre de contexto a ser elaborado. Eventuais dependências de contexto, certamente presentes na linguagem nesta etapa da construção do compilador ou interpretador, devem ser abstraídas e reservadas para tratamento posterior, ao nível semântico.

Em (JOSÉ NETO, MAGALHÃES, M.E.S., 1981a), (JOSÉ NETO, 1987), (JOSÉ NETO, 1988a) encontram-se publicações sobre o método que deu origem ao material deste capítulo, método este que se baseia no objetivo de reduzir, onde possível, a operação do reconhecedor sintático de uma linguagem, à de um autômato finito, de forma que o reconhecimento possa ser realizado de forma eficiente.

Uma melhoria de desempenho também é obtida limitando-se a utilização de uma área de armazenamento, organizada como pilha, apenas para o tratamento dos casos em que seja inviável a operação do reconhecedor estritamente nos moldes de um autômato finito, como ocorre nas situações em que no texto-fonte se manifestam construções sintáticas com potencial aninhamento.

Neste capítulo é apresentada uma variante do método mencionado, em que foram incorporados diversos aperfeiçoamentos que viabilizam a obtenção direta de autômatos ótimos em um número significativo de casos de interesse, dispensando otimizações posteriores na quase totalidade dos casos.

É apresentado ainda um mecanismo de geração de extensões aos autômatos obtidos, de forma que os autômatos assim estendidos possam recuperar-se de erros sintáticos detectados, efetuando, em particular, a recuperação absoluta de erros simples.

O capítulo se encerra com a apresentação de um mecanismo de transdução sintática baseada em autômatos de pilha estruturados, através do qual, a título de ilustração, é desenvolvido um gerador de dispositivos de análise sintática para gramáticas livres de contexto. Alguns exemplos de aplicação deste gerador são apresentados, cobrindo situações com diferentes graus de dificuldade.

3.1 Construção de autômatos de pilha sub-ótimos

Existem inúmeros métodos, reportados na literatura, que permitem a obtenção de autômatos a partir de gramáticas, de acordo com diversos paradigmas, conforme a particular política de escolha do não-terminal a ser considerado em cada passo do reconhecimento de uma sentença.

São muito populares, neste contexto, os reconhecedores determinísticos descendentes, também conhecidos como LL(k), como resultado da eficiência que são capazes de exibir, assim como os reconhecedores LR(k) ascendentes, que abrangem a maior classe de linguagens livres de contexto tratáveis através de métodos determinísticos.

O método de reconhecimento aqui apresentado procura aliar a generalidade dos métodos ascendentes à eficiência dos descendentes, exibindo as virtudes de ambos e proporcionando na prática, portanto, um significativo benefício, tanto ao projetista como ao usuário.

É possível explorar as vantagens das duas técnicas, efetuando-se todo o projeto do reconhecedor através de métodos inspirados nas técnicas LR(k) para se determinar desta forma os estados do reconhecedor, e, para aproveitar as características de eficiência de execução das técnicas LL(k), projetam-se as transições do reconhecedor impondo-se disciplinas que garantam a geração da transição mais econômica em cada particular situação.

Como resultado, são obtidos autômatos que operam, na grande maioria de seus movimentos, como simples autômatos finitos, utilizando-se de uma pilha de controle, indispensável ao reconhecimento de formas aninhadas, apenas nas ocasiões em que for estritamente necessário, ou seja, quando do efetivo aparecimento de construções que possam exibir aninhamentos sintáticos no texto analisado.

Isto garante uma grande generalidade para o método, herdada das técnicas LR(k), e um desempenho muito alto de execução dos reconhecedores resultantes, pelo fato de a sua eficiência de operação ser, na prática, essencialmente comparável à de um autômato finito.

Uma vantagem adicional deste método é que o tratamento de gramáticas que sejam ambíguas ou não-determinísticas não é excluido, permitindo a construção de reconhecedores eficientes também para as linguagens por elas descritas.

Um fator que em muito contribui para reduzir a eficiência dos reconhecedores sintáticos é a incorporação de recursos para a geração da árvore de derivação das sentenças, com base na gramática original, transformando-os de simples aceitadores em analisadores propriamente ditos da sintaxe da linguagem.

Para as situações em que não seja estritamente necessário efetuar a análise sintática, e sim apenas um reconhecimento das sentenças, torna-se conveniente, desta forma, não incorporar aos mecanismos reconhecedores

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

qualquer memória da gramática original, simplificando assim a sua estrutura e melhorando conseqüentemente o seu desempenho.

Contribui substancialmente, neste sentido, a redução do número de não-terminais, o que sugere um trabalho prévio de modificação da gramática original para outra, na qual estejam presentes apenas aqueles não-terminais que sejam indispensáveis ou, ao menos, convenientes.

Não-determinismos ocorrem como conseqüencia da existência de mais de uma opção para a substituição de não-terminais em determinados pontos do reconhecimento. Fatorações sucessivas, intercaladas com substituições de não-terminais pelas expressões que representam a sintaxe a eles associadas podem, no caso, ser empregadas com sucesso na eliminação da grande maioria dos não-determinismos desta classe.

Há casos em que certos não-determinismos se mostram não-elimináveis, uma vez que reaparecem sempre que se tentar eliminá-los por substituição.

Nestas situações pode-se, controlando o número das substituições e fatorações mencionadas, optar por um compromisso entre a velocidade do autômato (diretamente dependente da eliminação do não-determinismo) e a economia de área de armazenamento relativa ao autômato em questão.

Uma vez preparada desta forma a gramática, segue-se a identificação dos estados correspondentes a cada uma das situações possíveis de reconhecimento, decorrentes do percurso das expressões relativas aos não-terminais, sendo associada, a cada não-terminal da gramática preparada, uma sub-máquina correspondente do reconhecedor que está sendo construido.

Conforme o tipo de construção sintática simbolizada em cada ponto da expressão, são montadas então as transições necessárias à implementação do reconhecedor.

Esta montagem é feita de tal modo que, preferencialmente, a operação do autômato assim construido seja determinística, isenta de transições em vazio, consuma um terminal a cada transição interna efetuada e opere com um conjunto reduzido de estados.

É descrito, a seguir, um método que incorpora as idéias levantadas, proporcionando um algoritmo prático para a geração de reconhecedores muito eficientes e compactos, aplicável a qualquer gramática livre de contexto.

3.1.1 A preparação da gramática

Descreve-se a seguir um roteiro para a geração de reconhecedores de linguagens livres de contexto a partir de gramáticas expressas em notação de Wirth modificada. Uma apresentação desta notação pode ser encontrada no Anexo A.

Para o procedimento descrito a seguir, a linguagem deverá estar denotada por meio de uma gramática representada segundo a Notação de Wirth Modificada.

Sendo muito similares entre si as diversas metalinguagens usualmente empregadas na descrição da sintaxe de linguagens livres de contexto, se a linguagem estiver definida em qualquer outra metalinguagem, poderá ser facilmente convertida para esta notação.

Em (JOSÉ NETO, 1987) encontra-se um conjunto de regras que permitem representar em notação de Wirth tradicional gramáticas expressas em diversas outras notações habituais. Para traduzi-las para a notação de Wirth modificada, devem ser observadas as seguintes correspondências:

Notação de Wirth		Notação de Wirth Modificada
{ a }	=	(ε \ a)
[a]	=	(ε | a)
a { a }	=	(a \ ε)
a { b a }	=	(a \ b)

A seguir, a gramática deve ser manipulada de forma que sejam eliminadas todas as auto-recursões que não caracterizem aninhamentos, denotando-se assim de forma iterativa todos os não-terminais que estejam definidos através de expressões que façam uso deste tipo de recursão.

Para tanto, devem ser levantadas as dependências mútuas dos não-terminais, para a identificação de auto-recursões centrais indiretas, o que pode ser feito com mais facilidade com o auxílio de uma árvore da gramática.

Para construir uma árvore para a gramática desejada, cujos nós correspondem às expressões que definem os não-terminais presentes nas expressões associadas ao nó-pai, cria-se inicialmente um nó-raiz da árvore, em correspondência ao não-terminal que representa a raiz da gramática.

Em seguida, a cada passo do desenvolvimento da árvore, incluem-se como filhos de cada nó da árvore novos nós, associados às eventuais ocorrências dos diversos não-terminais encontrados na expressão correspondente ao não-terminal em questão.

Para que o procedimento seja finito, deve-se tomar o cuidado de, ao expandir a árvore da gramática, não criar nós-filhos para não-terminais que estejam associados a algum dos seus nós-ancestrais.

Assim, a expansão da árvore deverá estar concluida formando uma árvore com folhas correspondendo a não-terminais direta ou indiretamente auto-dependentes, ou então, a cadeias de terminais.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Coletam-se então, dentre as folhas da árvore, os não-terminais associados que tenham a si próprios como ancestrais, selecionando-se dentre estes, como auto-recursivos centrais, os não-terminais N que obedeçam à definição de auto-recursão central:

$$N \Rightarrow^* \alpha N \beta$$

com α e β simultaneamente não-vazios.

Identificados os não-terminais auto-recursivos centrais, passa-se a eliminar todos aqueles que não sejam necessários ou convenientes - são essenciais apenas a raiz da gramática e os não-terminais que definam aninhamentos independentes - preservando os demais, por serem essenciais, ou por conveniência do projeto, ou até mesmo por preferências pessoais do projetista.

A preservação de não-terminais que não sejam essenciais deve ser sempre efetuada com muito critério, de modo que não redunde em prejuízo da clareza nem da eficiência do reconhecedor produzido, resultando, desta prática, um compromisso entre a extensão do reconhecedor, o número de seus estados e sub-máquinas, e o seu desempenho.

Uma primeira providência consiste em se manipular as diversas expressões que definem os não-terminais da gramática, de forma que sejam eliminadas auto-recursões diretas ou indiretas que denotem construções sintáticas que não sejam aninhadas.

Para isto, antes de mais nada, cada não-terminal deve ser definido através de uma única expressão, em que figurem todas as possíveis opções de substituição para o não-terminal:

Para cada não-terminal **N**, definido por produções do tipo

$$N = a_i \ .$$

com $i = 1, \ldots, k$, agrupar as k opções em uma única expressão:

$$N = a_1 \mid \ldots \mid a_k .$$

Pode-se a seguir efetuar uma busca de definições de iterações sintáticas que estejam representadas na gramática, denotando-as de tal modo que esta característica se torne explícita, para facilitar a geração direta de um reconhecedor eficiente.

Para isto, as expressões obtidas devem ser fatoradas de tal forma que sejam explicitadas todas as auto-recursões diretas não-centrais que estejam eventualmente presentes nas expressões.

Inicialmente, coletam-se, dentre as k opções que definem **N**, todas aquelas em que o não-terminal **N** esteja simultaneamente presente nos extremos direito e esquerdo da opção:

$$N \ d_i \ N$$

com $i = 1, \ldots, m$, substituindo-se tal expressão por

$$N \ d \ N$$

onde **d** representa um agrupamento da forma

$$(\ d_1 \mid \ldots \mid d_m \ )$$

A seguir, devem ser coletadas as opções da forma

$$N \ c_i$$

com $i = 1, \ldots, n$, e substituindo tal grupo pela expressão

$$N \ c$$

onde **c** representa um agrupamento da forma

$$(\ c_1 \mid \ldots \mid c_n \ )$$

A seguir, devem ser coletadas opções da forma

$$b_i \ N$$

com $i = 1, \ldots, p$, e substitui-se estas p opções pela expressão

$$b \ N$$

onde **b** representa um agrupamento da forma

$$(\ b_1 \mid \ldots \mid b_p \ )$$

Finalmente, coletam-se as q opções restantes

$$a_1 \ , \ \ldots \ , \ a_q$$

substituindo-as pela expressão **a**, que representa um agrupamento da forma

$$(\ a_1 \mid \ldots \mid a_q \ )$$

obtendo-se, finalmente, para cada não-terminal **N**, uma expressão da forma:

$$N = a \mid b \ N \mid N \ c \mid N \ d \ N$$

onde os coeficientes **a, b, c, d** representam as expressões em evidência, denotadas na notação de Wirth modificada, e nas quais o não-terminal **N** pode mesmo figurar, proveniente das eventuais ocorrências de auto-recursões centrais explícitas.

A manipulação seguinte visa à eliminação de todas as auto-recursões não-centrais, e consiste na aplicação, à expressão acima, da seguinte identidade:

$$N = (\ (\ \varepsilon \setminus b \ ) \ a \ (\ \varepsilon \setminus c \ ) \setminus d \ )$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Desta maneira, todas as auto-recursões não-centrais de N na expressão original foram substituidas por formas iterativas sintaticamente equivalentes.

Conhecidos os não-terminais a serem mantidos na gramática final e eliminadas todas as suas auto-recursões à direita e à esquerda, pode-se passar à eliminação dos demais não-terminais da gramática, ou seja, dos *não-terminais acessórios*.

Para tanto, devem ser efetuadas substituições sucessivas das ocorrências de não-terminais acessórios nas expressões que definem os não-terminais essenciais, de maneira similar à que se emprega na resolução de equações algébricas simultâneas pelo método de substituição de variáveis:

Para cada ocorrência de algum não-terminal acessório **M**, definido como

```
M = a.
```

substitui-se **M** pela expressão

```
( a )
```

na expressão onde **M** ocorre. Assim,

```
N = .... x M y ....
```

torna-se

```
N = .... x ( a ) y ....
```

Após cada substituição, podem surgir novas ocorrências explícitas do não-terminal **N** na expressão resultante, exigindo neste caso novas aplicações das regras de eliminação de auto-recursões não-centrais.

Note-se que, nesta ocasião, devido às fatorações a que foi submetida, a forma da expressão nem sempre se mostrará adequada à reaplicação das operações descritas, sendo necessário, para que o trabalho de fatoração a ser realizado seja completo, desfazer previamente ao menos parte das fatorações efetuadas, de modo que a eliminação de auto-recursões possa ser adequadamente executada.

Para o trabalho de defatoração, são úteis as seguintes identidades:

```
a ( b | c )     =     a b | a c
( b | c ) a     =     b a | c a
( a \ b )       =     ( a ) ( ε \ ( b ) ( a ) )
( ε \ b )       =     ( b ) ( ε \ b ) | ε
( ( a ) )       =     ( a )
```

Uma vez obtidas expressões isentas de não-terminais não-essenciais, deve-se procurar manipulá-las no sentido de detectar eventuais fontes de não-determinismos no autômato a ser gerado, eliminando-as sempre que for possível.

Estas são representadas em três situações habituais:

– cadeia vazia como opção não-única em um dado agrupamento:

```
a ( ε | b | ... | c ) d
```

– opções diferentes, de um mesmo agrupamento, iniciadas por um mesmo prefixo não-vazio:

```
a ( b x | b y | ... | c )
```

– uma ou mais opções iniciadas por não-terminal em um agrupamento não-unitário:

```
a ( X b | c | ... | d )
```

A presença, em um mesmo agrupamento, da cadeia vazia como opção, ou de opções iniciadas com o mesmo prefixo, pode ser facilmente eliminada pela fatoração adequada da expressão, ou então por manipulação posterior do autômato resultante, através de métodos clássicos de redução, usualmente aplicados à minimização de autômatos finitos.

Já as transições com não-terminal apresentam uma complexidade maior, visto envolverem, em seu estudo, mais de uma sub-máquina ao mesmo tempo.

Para assegurar que o autômato resultante seja determinístico torna-se, pois, necessário eliminar a ocorrência de não-terminais como elementos mais à esquerda de sub-expressões alternativas.

Para isto, uma possível solução consiste em substituir as ocorrências de tais não-terminais explícitos pelas expressões que os definem, refatorando-se a expressão resultante.

Podem, nestas circunstâncias, ocorrer os seguintes fenômenos: ou desaparecem todos os não-terminais mais à esquerda, situação em que ou o problema do não-determinismo se resolve, ou então surgem novas ocorrências de não-terminais mais à esquerda na expressão resultante.

Observando-se os não-terminais que surgiram, pode-se identificar situações novas (ocorrências de outros não-terminais ou então dos mesmos não-terminais, porém em outras situações sintáticas), ou então a reincidência de situações já ocorridas em estágios anteriores da maniplulação da expressão.

No primeiro caso, pode-se continuar o processo, reaplicando-se o procedimento de substituição para os não-terminais em questão. No segundo, constata-se a impossibilidade de eliminação do não-determinismo, devendo-se neste caso adotar uma solução de compromisso: efetuar um número suficiente de substituições do não-

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

terminal, de forma que se garanta o funcionamento determinístico do autômato resultante, dentro de limites considerados satisfatórios como metas de projeto.

No procedimento apresentado em 3.3.2 está descrito detalhadamente um método similar, baseado neste mesmo princípio, voltado à preparação de gramáticas para a construção de transdutores sintáticos.

São apresentados a seguir dois exemplos, a título de ilustração dos casos de não-determinismos elimináveis ou não-elimináveis que podem surgir neste tipo de manipulação.

Exemplo 1

Neste primeiro exemplo, é mostrado um caso de não-determinismo eliminável por substituições e refatorações sucessivas.

Sejam as duas regras gramaticais seguintes:

X → Y a | b e

Y → b Y c | d

- substituindo-se em **X** a expressão que define **Y** tem-se:
  X = (b Y c | d) | b e
- eliminando-se os parênteses desnecessários:
  X = b Y c | d | b e
- fatorando-se a expressão, obtém-se:
  X = b (Y c | e) | d

Observar o aparecimento do não-terminal Y na extremidade esquerda da primeira das opções entre parênteses.

- substituindo-se a ocorrência do não-terminal Y na expressão acima pela expressão que o define tem-se:
  X = b ((b Y c | d) c | e) | d
- defatorando-se esta expressão completamente tem-se:
  X = b (b Y c c | d c | e) | d
  X = b b Y c c | b d c | b e | d
- refatorando-se a expressão colocando-se em evidência o terminal b , observa-se o total desaparecimento do não-determinismo causado pela presença de Y na extremidade esquerda da expressão inicialmente estudada:
  X = b (b Y c c | d c | e) | d
  ficando assim a expressão pronta para uso na elaboração de um autômato determinístico para a linguagem desejada.

■

Exemplo 2

Um segundo exemplo ilustra o caso de uma linguagem livre de contexto definida através de regras cuja manipulação não permite eliminar o não-determinismo nela existente.

Sejam as duas regras gramaticais seguintes:

X → Y a | b e

Y → b Y a | b b e | d

- substituindo-se a expressão que define o não-terminal Y na expressão que formaliza X tem-se:
  X = (b Y a | b b e | d) a | b e
- defatorando-se esta expressão obtém-se:
  X = b Y a a | b b e a | d a | b e
- refatorando-se através da colocação em evidência do terminal b, surge uma ocorrência do não-terminal Y na extremidade esquerda de uma das opções de que é formado o agrupamento entre parênteses:
  X = b (Y a a | b e a | e) | d a
- substituindo-se esta ocorrência do não-terminal Y pela expressão que o define obtém-se:
  X = b ((b Y a | b b e | d) a a | b e a | e) | d a
- defatorando-se totalmente a expressão acima, obtém-se:
  X = b (b Y a a a | b b e a a | d a a | b e a | e) | d a
  X = b b Y a a a | b b b e a a | b d a a | b b e a | b e | d a
- colocando-se o terminal b em evidência por fatoração resulta:
  X = b (b Y a a a | b b e a a | d a a | b e a | e) | d a
- colocando-se em evidência novamente o terminal b no interior do agrupamento entre parênteses, constata-se o reaparecimento do não-terminal **Y** na extremidade esquerda de uma das opções do agrupamento resultante:
  X = b (b (Y a a a | b e a a | e a) | d a a | e) | d a

Logo, não é possível eliminar este não-determinismo, já que novas tentativas de eliminação por substituições e fatorações fazem com que se recaia na mesma situação encontrada inicialmente.

■

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Note-se que os autômatos a serem obtidos a partir das expressões construidas desta forma tenderão a apresentar um número de estados bastante elevado, já que as substituições efetuadas implicam em reinstanciações de expressões, das quais resultarão reinstanciações dos correspondentes grupos de estados no autômato.

Por outro lado, o ganho no desempenho do reconhecedor final, bem como a ampla aplicabilidade exibida por este método, são significativos o suficiente para que se justifique a sua escolha na maioria dos casos práticos em que se exija um alto desempenho do reconhecedor.

Preparada desta forma a gramática, tem-se para cada não-terminal uma expressão, na notação de Wirth modificada, que define uma sintaxe associada, de maneira similar àquela através da qual uma expressão regular define uma linguagem regular.

Isto vem a sugerir que seja utilizado o modelo dos autômatos de pilha estruturados para a geração de reconhecedores para a linguagem assim definida, uma vez que as sub-máquinas dos autômatos de pilha estruturados têm uma topologia e um comportamento muito aderentes aos dos autômatos finitos, permitindo portanto a utilização de métodos clássicos de mapeamento da gramática para um autômato equivalente.

3.1.2 Designação de estados ao autômato

A gramática, neste ponto, está preparada para ser mapeada para a forma de um autômato de pilha estruturado. No caso particular de ser a linguagem representada apenas em função de terminais, ou seja, se a raiz da gramática não for auto-recursiva, e se, além disso, for este o único não-terminal restante, ter-se-á em mãos uma linguagem regular, e o reconhecedor correspondente deverá ser, portanto, um autômato finito.

Para a obtenção de um autômato a partir da gramática preparada anteriormente, deve ser estabelecida inicialmente uma correspondência entre os diversos pontos da expressão e os estados do autômato, através da análise das construções da metalinguagem.

Em seguida, através de um mapeamento direto de cada construção da metalinguagem para um padrão topológico correspondente de autômato, obtém-se finalmente o esquema do reconhecedor desejado.

Uma otimização subseqüente das diversas sub-máquinas assim obtidas, efetuada através de métodos clássicos de redução de autômatos finitos, pode levar a formas mais eficientes do reconhecedor, embora pela aplicação das regras de mapeamento indicadas isto seja necessário em poucos casos apenas.

A construção de um reconhecedor a partir de uma gramática segue a filosofia descrita acima, e consta de uma seqüencia de etapas que pode ser esquematizada como segue.

Admite-se que neste ponto a linguagem esteja descrita na notação de Wirth modificada, e que a gramática tenha sido manipulada e preparada conforme o procedimento anteriormente descrito.

Obtida a expressão final que representa cada não-terminal, passa-se, a exemplo do que é feito nos métodos de reconhecimento ascendentes, à marcação dos pontos das expressões, e à designação dos correspondentes estados do reconhecedor, associados a cada um dos pontos marcados da expressão.

Da observação da relação existente entre as diversas construções da expressão que define a sintaxe da linguagem e a forma assumida pela parte correspondente do reconhecedor associado, é possível estabelecer regras que impõem uma disciplina geral para a construção do autômato a partir da expressão.

Assim, desejando-se produzir um autômato com um número reduzido de estados, convém evitar a atribuição de estados diferentes a pontos da expressão que correspondam a estados equivalentes do autômato a ser obtido.

A incidência de tais situações é maior em construções cíclicas, em grupos de alternativas e em construções sintáticas opcionais. Estas circunstâncias propiciam o aparecimento de transições em vazio, e provavelmente de estados distintos mas equivalentes no reconhecedor construido.

Como as três formas sintáticas em questão são representadas, na notação de Wirth modificada, através de agrupamentos entre parênteses, é nestes que podem ser concentrados os esforços iniciais para evitar proliferação de estados equivalentes.

Um caso notável, que merece atenção especial devido à propensão que exibe à ocorrência de estados equivalentes, evitáveis a priori, é o dos aninhamentos de agrupamentos entre parênteses, em particular os que envolvem construções cíclicas.

Estudando-se caso a caso as diversas possibilidades de aninhamentos dos agrupamentos entre parênteses, constata-se que construções cíclicas aninhadas não podem compartilhar estados a não ser que a mais interna constitua uma das extremidades, ou ambas, da opção sintática, única, contida no agrupamento imediatamente mais externo.

Isto se deve ao fato de que o eventual compartilhamento (incorreto) de estados entre um ciclo interno e um ciclo externo introduz caminhos adicionais nos autômatos resultantes, permitindo que sejam aceitas construções sintáticas não pertencentes à linguagem especificada pela gramática original.

O mesmo pode acontecer se forem (indevidamente) tratados como equivalentes estados relativos a ciclos mais internos e estados referentes a expressões utilizadas como elos de realimentação de construções cíclicas externas.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Com base nestas reflexões, pode-se estabelecer uma prática segundo a qual se inicie a designação dos estados do autômato pelos pontos extremos das expressões que compõem os agrupamentos entre parênteses, e, no caso de aninhamentos, pelos níveis mais externos, e propagando-se, onde houver possibilidade, os estados já atribuidos para os pontos do nível mais interno de aninhamento que sejam reconhecidamente associados a estados equivalentes.

Na designação de estados, pode-se adotar a técnica da numeração seqüencial, atribuindo-se a cada novo estado ainda não designado um número natural que seja uma unidade maior que o último já utilizado.

Descreve-se a seguir um dos possíveis procedimentos para a designação de estados aos pontos das expressões marcadas da notação de Wirth modificada.

Constata-se facilmente que da aplicação deste procedimento resulta uma designação de estados que leva à obtenção de autômatos muito compactos e com poucas redundâncias.

Conseqüentemente, para a maioria das aplicações normais, tais autômatos podem ser usados diretamente, sem que haja necessidade de manipulações de otimização.

Procedimento de designação de estados

a. Localizar inicialmente os agrupamentos (eventualmente aninhados) entre parênteses da expressão, cíclicos ou não.

b. Isolar um agrupamento completo ainda não tratado. Iniciando sempre pelos níveis mais externos de agrupamentos entre parênteses, designar novos números aos pontos extremos das expressões que compõem cada uma de suas opções.

c. Se existirem números **r** e/ou **s** já designados aos estados correspondentes aos pontos imediatamente à esquerda ou à direita do agrupamento entre parênteses, de uma das formas:

$$\underset{r}{\uparrow}\ (\ \uparrow a_1 \uparrow\ |\ \dots\ |\ \uparrow a_m \uparrow\ )\ \underset{s}{\uparrow}$$

ou

$$\underset{r}{\uparrow}\ (\ \uparrow a_1 \uparrow\ |\ \dots\ |\ \uparrow a_m \uparrow\ \ \uparrow b_1 \uparrow\ |\ \dots\ |\ \uparrow b_n \uparrow\ )\ \underset{s}{\uparrow}$$

há os seguintes casos a considerar:

- Existindo apenas **r** , fazer **x = r** , caso contrário, associar a **x** um número correspondente a um novo estado e designar **x** aos pontos extremos esquerdos de todas as opções internas ao agrupamento.
- Existindo **s** , fazer **y = s** , caso contrário, associar a **y** um número correspondente a um novo estado. Designar **y** aos pontos extremos direitos de todas as opções internas ao agrupamento.

d. Para agrupamentos da forma particular

$$\underset{r}{\uparrow}\ (\ \uparrow \varepsilon \uparrow\ \ \uparrow b_1 \uparrow\ |\ \dots\ |\ \uparrow b_n \uparrow\ )\ \underset{s}{\uparrow}$$

se não houver conflito com eventuais estados anteriormente designados, internamente ao agrupamento, às extremidades das expressões b_i , criar um único estado **x = y** , que será associado a cada uma das extremidades de tais opções.

e. Para os demais casos, sempre que for preciso criar novos estados, gerar sempre estados **x** e **y** distintos.

f. Numerar então os estados extremos de cada uma das expressões que compõem o agrupamento, obtendo-se uma das seguintes configurações, conforme o caso:

$$\underset{r}{\uparrow}\ (\ \underset{x}{\uparrow} a_1 \underset{y}{\uparrow} |\ \dots\ |\ \underset{x}{\uparrow} a_m \underset{y}{\uparrow}\ )\ \underset{s}{\uparrow}$$

$$\underset{r}{\uparrow}\ (\ \underset{x}{\uparrow} a_1 \underset{y}{\uparrow}\ |\ \dots\ |\ \underset{x}{\uparrow} a_m \underset{y}{\uparrow}\ \backslash\ \underset{y}{\uparrow}\ b_1 \underset{x}{\uparrow}\ |\ \dots\ |\ \underset{y}{\uparrow}\ b_n \underset{x}{\uparrow}\ )\ \underset{s}{\uparrow}$$

ou

$$\underset{r}{\uparrow}\ (\ \underset{x}{\uparrow} \varepsilon \underset{y}{\uparrow}\ \backslash\ \underset{y}{\uparrow}\ b_1 \underset{x}{\uparrow}\ |\ \dots\ |\ \underset{y}{\uparrow}\ b_n \underset{x}{\uparrow}\ )\ \underset{s}{\uparrow}$$

com **x = y** apenas nesta última expressão.

g. Propagar, na expressão obtida, o estado **x** para o ponto externo imediatamente à esquerda do agrupamento entre parênteses caso a este ponto da expressão ainda não tenha sido designado nenhum estado:

$$\underset{x}{\uparrow}\ (\ \underset{x}{\uparrow}\ a_1 \underset{y}{\uparrow}\ |\ \dots\ |\ \underset{x}{\uparrow}\ a_m \underset{y}{\uparrow}\ \backslash\ \underset{y}{\uparrow}\ b_1 \underset{x}{\uparrow}\ |\ \dots\ |\ \underset{y}{\uparrow}\ b_n \underset{x}{\uparrow}\ )\ \uparrow$$

h. Propagar **y** para o ponto externo imediatamente à direita do agrupamento entre parênteses caso a este ponto da expressão ainda não tenha sido designado nenhum estado:

$$\uparrow\ (\ \underset{x}{\uparrow}\ a_1 \underset{y}{\uparrow}\ |\ \dots\ |\ \underset{x}{\uparrow}\ a_m \underset{y}{\uparrow}\ \backslash\ \underset{y}{\uparrow}\ b_1 \underset{x}{\uparrow}\ |\ \dots\ |\ \underset{y}{\uparrow}\ b_n \underset{x}{\uparrow}\ )\ \underset{y}{\uparrow}$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

i.Caso a ambos os pontos externos imediatamente à direita e à esquerda do agrupamento ainda não tenha sido designado nenhum estado, efetuar as duas propagações acima, obtendo-se:

```
↑  ( ↑  a1 ↑  | . . . | ↑  am ↑ \ ↑  b1 ↑  | . . . | ↑  bn ↑  ) ↑
x    x    y            x    y   y    x             y    x    y
```

j. Para o caso particular em que não haja mais de uma única opção, a_1 , entre as expressões a_i:

```
↑  ( ↑ a1 ↑ \ ↑ b1 ↑ | . . . | ↑ bn ↑  ) ↑
x    x    y   y    x           y    x    y
```

e a expressão a_1 contiver um agrupamento cíclico em sua extremidade esquerda:

```
↑ ( ↑ c1 ↑ | . . . | ↑ cp ↑ \ ↑ d1 ↑ | . . . | ↑ dq ↑ ) . . .
x
```

é possível propagar o estado **x** para o interior deste agrupamento cíclico:

```
↑  ( ↑ c1 ↑ | . . . | ↑  cp ↑ \ ↑ d1 ↑ | . . . | ↑ dq ↑ ) . . .
x    x                   x          x              x
```

k. Ainda para este mesmo caso particular, se a expressão a_1 contiver um agrupamento cíclico em sua extremidade direita:

```
. . .        ( ↑ c1 ↑ | . . . | ↑ cp ↑ \ ↑ d1 ↑ | . . . | ↑ dq ↑ ) ↑
                                                                  y
```

é possível propagar o estado **y** para o interior deste agrupamento cíclico:

```
. . .   ( ↑  c1 ↑ | . . . | ↑ cp ↑ \ ↑ d1 ↑ | . . . | ↑ dq ↑ ) ↑
          y                    y    y                  y        y
```

l. Se as duas condições acima forem verdadeiras, com a expressão a_1 da forma particular seguinte, em que ela se confunde com o próprio agrupamento cíclico mencionado:

```
↑ ( ↑ c1 ↑ | . . . | ↑ cp ↑ \ ↑ d1 ↑ |. . . | ↑ dq ↑ ) ↑
x                                                      y
```

ambas as propagações são permitidas, obtendo-se:

```
↑  ( ↑ c1 ↑  | . . . | ↑  cp ↑  \ ↑ d1 ↑ |. . . |↑  dq ↑  ) ↑
x    x    y              x    y   y    x             y    x   y
```

m. Propagar, de maneira similar à que foi acima apresentada, para o interior de todos os agrupamentos (não-cíclicos) ainda não considerados, os estados previamente associados às extremidades direita e esquerda dos agrupamentos.

n. Completar a designação de estados atribuindo arbitrariamente novos estados a cada um dos pontos da expressão aos quais ainda não tenha sido feita nenhuma designação.

■

3.1.3 Construção das produções do autômato

Efetuada a designação de estados às expressões marcadas que definem a linguagem, resta apenas interpretá-las para que disto resulte o conjunto das sub-máquinas que compõem o autômato de pilha desejado.

Para isto as expressões devem ser inspecionadas, com a finalidade de localizar as ocorrências de terminais, não-terminais, cadeia vazia, agrupamentos de alternativas e construções cíclicas.

Os números associados aos pontos da expressão marcada adjacentes aos elementos identificados no levantamento acima corresponderão aos estados associados às transições que envolvem tais elementos.

Os tipos de transições associados a cada caso devem ser escolhidos em função do paralelo conceitual existente entre as expressões que definem a sintaxe e os autômatos que implementam o reconhecimento correspondente.

Para obter o autômato a partir das expressões com as devidas designações de estados bastará portanto aplicar de forma sistemática as transformações seguintes:

Procedimento de obtenção das produções do autômato

a. Para toda ocorrência de um meta-símbolo ε :

```
↑ ε ↑
x   y
```

se $x \neq y$, criar uma transição em vazio de **x** para **y** :

$$\gamma\ x\ \alpha \ \rightarrow\ \gamma\ y\ \alpha$$

ignorando tal ocorrência no caso de $x = y$.

b. Para toda ocorrência de terminais **t** na expressão:

```
↑ t ↑
x   y
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

construir uma transição interna entre os estados **x** e **y**, consumindo o terminal **t**:

$$\gamma\ x\ t\ \alpha \rightarrow \gamma\ y\ \alpha$$

c. Para toda ocorrência de um não-terminal **N** na expressão:

```
↑ N ↑
x   y
```

criar uma transição em vazio partindo do estado x para o estado inicial n_0 da sub-máquina associada ao não-terminal **N**, salvando o estado **y na** pilha do autômato, para ser utilizado pela sub-máquina chamada quando da execução de sua transição de retorno:

$$\gamma\ x\ \alpha \rightarrow \gamma\ y\ n_0\ \alpha$$

d. Para cada agrupamento cíclico:

```
↑ ( ↑ a ↑ \ ↑ b ↑ ) ↑
x   y   z   z   y   t
```

se **x** e **y** forem diferentes, criar uma transição em vazio de **x** para **y** , realizando a conexão entre a parte do autômato relativa à expressão à esquerda do agrupamento e aquela que implementa o agrupamento cíclico:

$$\gamma\ x\ \alpha \rightarrow \gamma\ y\ \alpha$$

se **z** e **t** forem diferentes, criar uma transição em vazio de **z** para **t** , efetuando uma conexão similar na parte direita do agrupamento cíclico:

$$\gamma\ z\ \alpha \rightarrow \gamma\ t\ \alpha$$

e. Para cada estado n_f , associado ao final de alguma das opções que definem um não-terminal **N** , criar uma transição em vazio que execute o retorno ao estado **y** da sub-máquina chamadora, previamente empilhado pela produção de chamada da sub-máquina associada ao não-terminal **N** :

$$\gamma\ y\ n_f\ \alpha \rightarrow \gamma\ y\ \alpha$$

f. interpretar como sendo o estado inicial da sub-máquina associada ao não-terminal **N** o estado correspondente à extremidade esquerda das expressões que definem este não-terminal.

g. Os estados inicial e final da sub-máquina associada ao não-terminal que representa a raiz da gramática serão os estados inicial e final do autômato, respectivamente.

O autômato assim obtido está pronto para ser fisicamente realizado diretamente, ou, se necessário, depois de otimizado para a eliminação de estados ou transições redundantes, transições em vazio ou outras fontes de ineficiências.

Se necessário ou conveniente, incorporam-se extensões de recuperação de erros às sub-máquinas, para finalmente integrá-las, formando o reconhecedor final, como será discutido mais adiante neste capítulo.

O conjunto de produções assim obtido corresponde a um projeto conceitual completo do reconhecedor sintático desejado, restando mapeá-lo de forma tal que possa ser executado por um computador.

Para isto, há diversas alternativas, sendo possível implementá-lo, por exemplo, através de tabelas de transições, associadas a cada sub-máquina, com um programa que as interprete, ou então por meio da criação de um programa que implemente diretamente o algoritmo que o autômato representa.

Para o caso de uma única sub-máquina não-recursiva, ou seja, de um autômato finito, o uso da segunda opção leva a reconhecedores muito bons. Para o caso geral, implementações que seguem a tendência dos reconhecedores descendentes recursivos oferecem uma grande flexibilidade e dão margem à criação de compiladores muito eficientes e de fácil manutenção.

A implementação através da interpretação de tabelas é de grande simplicidade e muito prática, sendo em geral indicada para a construção de protótipos de processadores de linguagens, bem como para uso em aplicações didáticas e durante a fase de desenvolvimento do compilador ou da linguagem.

Em qualquer das opções, deve-se notar que existe sempre a possibilidade de se construir ferramentas que automatizem o procedimento, delegando à máquina a tarefa da construção física dos programas e tabelas, liberando o projetista para a execução de atividades menos mecânicas e efetuando de forma mais confiável e rápida as operações de implementação física, que são sempre muito repetitivas e susceptíveis a falhas humanas.

■

3.1.4 Exemplo de Aplicação

Para ilustrar a utilização do método acima descrito, apresenta-se a seguir um exemplo completo de geração de um pequeno reconhecedor para a clássica linguagem definida pelas regras gramaticais abaixo:

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow a$$
$$F \rightarrow [\ E\ ]$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Preparação da gramática

Inicialmente a gramática fornecida é manipulada conforme as regras estabelecidas, para que a linguagem seja inicialmente representada através da Notação de Wirth Modificada. Nova manipulação é feita para que sejam então eliminados os não-terminais supérfluos. Após o agrupamento das regras em produções únicas para cada não-terminal, e a eliminação de auto-recursões não-centrais, obtém-se:

```
E = ( T \ + )
T = ( F \ * )
F = a | [ E ]
```

Eliminando-se os não-terminais **T** e **F** por substituições sucessivas na expressão que define o não-terminal **E** chega-se **a**:

```
E = ( ( a | [ E ] \ * ) \ + )
```

Designação de estados

Passa-se a efetuar uma associação de estados aos diversos pontos da expressão resultante, mediante um procedimento de numeração seqüencial.

Inicia-se pela designação dos estados associados às extremidades das expressões que compõem o agrupamento mais externo entre parênteses:

```
E = ↑ ( ↑ ( ↑ a ↑ | ↑ [ ↑ E ↑ ] ↑ \ ↑ * ↑ ) ↑ \ ↑ + ↑ ) ↑
    1                                       2   2   1
```

Notar que foram identificados com o mesmo número **1** o ponto extremo esquerdo da expressão interna esquerda do agrupamento, e também o ponto extremo direito da expressão interna direita do mesmo agrupamento, os quais correspondem ambos ao estado inicial da parte do autômato responsável pelo reconhecimento da construção cíclica que este agrupamento representa.

Analogamente, foram associados ao número **2** os extremos direito da expressão interna esquerda, e esquerdo da expressão interna direita deste mesmo agrupamento, os quais estão associados a um mesmo estado, aquele em que cada ciclo se completa.

Uma vez designados os estados associados ao agrupamento mais externo, propagam-se os seus estados extremos para fora do agrupamento, já que não há qualquer condição que impeça tal atitude:

```
E = ↑ ( ↑ ( ↑ a ↑ | ↑ [ ↑ E ↑ ] ↑ \ ↑ * ↑ ) ↑ \ ↑ + ↑ ) ↑
    1   1                                   2   2   1   2
```

Procura-se então a propagação de estados para o agrupamento cíclico aninhado mais interno. Note-se que, sendo esta a única opção que consta na expressão esquerda do agrupamento cíclico externo, nada impede que sejam identificados os seus estados extremos com os daquele agrupamento:

```
E = ↑ ( ↑ ( ↑ a ↑ | ↑ [ ↑ E ↑ ] ↑ \ ↑ * ↑ ) ↑ \ ↑ + ↑ ) ↑
    1   1   1   2   1           2   2   1   2   2   1   2
```

Notar a propagação do estado 1 para os pontos mais à esquerda de todas as alternativas esquerdas do agrupamento mais interno entre parênteses e para o ponto mais à direita da expressão direita do mesmo, bem como a propagação do estado 2 para os pontos mais à direita das expressões alternativas esquerdas, e para o ponto mais à esquerda da expressão direita do mesmo.

Não havendo mais agrupamentos entre parênteses a considerar, completa-se a designação dos estados pela numeração dos pontos da expressão ainda não numerados. A estes são associados números diferentes e arbitrários, designando os correspondentes estados do autômato, completando finalmente esta designação de estados:

```
E = ↑ ( ↑ ( ↑ a ↑ | ↑ [ ↑ E ↑ ] ↑ \ ↑ * ↑ ) ↑ \ ↑ + ↑ ) ↑
    1   1   1   2   1   3   4   2   2   1   2   2   1   2
```

Construção das produções do autômato

Terminada a designação de estados, pode-se passar a interpretar a expressão marcada resultante, criando para cada número designado um estado homônimo, e interligando os estados por meio de transições conforme a interpretação da expressão acima.

Inicialmente, coletam-se todas as ocorrências de terminais na expressão, criando para cada qual uma produção responsável pelo consumo do terminal em questão, a partir do estado associado ao número designado à esquerda do terminal na expressão, e tendo como destino o estado correspondente ao número desgnado à sua direita:

```
γ 1 a α  →  γ 2 α
γ 1 [ α  →  γ 3 α
γ 4 ] α  →  γ 2 α
γ 2 * α  →  γ 1 α
γ 2 + α  →  γ 1 α
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Procede-se analogamente com as ocorrências de não-terminais, criando para cada uma delas uma transição de chamada de sub-máquina correspondente.

Assim, para o único não-terminal **E** encontrado cria-se uma transição de chamada da sub-máquina correspondente (sendo $e_0 = 1$ seu estado inicial), com a indicação do endereço de retorno associado:

$$\gamma \ 3 \ \alpha \ \rightarrow \ \gamma \ 4 \ e_0 \ \alpha$$

a qual empilha o estado **4** de retorno.

Criam-se então transições de retorno das sub-máquinas associadas aos diversos não-terminais, a partir dos estados finais de cada uma das sub-máquinas.

Os estados finais correspondem aos estados associados às extremidades direitas de todas as expressões que constituem alternativas para a definição dos não-terminais através da Notação de Wirth Modificada.

Neste exemplo, a única transição de retorno deverá corresponder à do estado final da sub-máquina associada ao não-terminal E , raiz da gramática, que foi o único não-terminal remanescente da manipulação da gramática:

$$\gamma \ z \ 2 \ \alpha \ \rightarrow \ \gamma \ z \ \alpha$$

a qual promove o retorno para o último estado z previamente empilhado pela transição que ativou a sub-máquina corrente.

Não havendo ocorrências de opções correspondentes à cadeia vazia, não haverá as correspondentes transições em vazio, e nestas condições restaria apenas neste ponto a geração das transições necessárias à implementação das partes do autômato responsáveis pela realização dos agrupamentos das expressões de Wirth modificadas.

No caso, entretanto, dado que os eventuais estados a serem interligados por transições em vazio já foram identificados por equivalência, na época da designação de estados, nada mais haverá a ser gerado, e portanto o autômato está pronto para ser implementado no computador.

Considerações sobre a Implementação

Uma possível e confortável forma de realização deste reconhecedor simples é o do mapeamento direto para uma linguagem de programação disponível.

Para cada não-terminal da gramática cria-se uma função booleana que deve produzir como resultado uma indicação do sucesso ou fracasso no reconhecimento sintático do não-terminal associado, em função dos resultados dos testes efetuados pela execução de suas transições.

- A lógica destas funções deve ser implementada com base no conjunto anteriormente calculado de produções do autômato, conforme o roteiro abaixo.
- A função que tiver êxito no reconhecimento efetuado deve retornar **TRUE** , e aquela que fracassar nesta tentativa, **FALSE** .
- Para cada estado da função associa-se um rótulo interno correspondente no programa **R i** ; para iniciar e finalizar a atividade da função, criam-se os rótulos **INICIO** e **FIM** , respectivamente.
- Para cada conjunto de produções que partem de um dado estado, cria-se junto ao rótulo associado um desvio condicional correspondente para o rótulo referente ao estado-destino da transição em questão.
- Transições que consomem átomos devem efetuar chamadas de uma rotina de análise léxica, que deverá responsabilizar-se pela extração e classificação do átomo, e pela movimentação do cursor sobre a cadeia de entrada.
- Transições em vazio executarão desvios incondicionais para o rótulo associado ao estado-destino da transição.
- Transições de chamada de sub-máquina deverão efetuar chamadas da função correspondente, sem parâmetros.
- Transições de retorno deverão corresponder a um desvio para o rótulo de retorno **FIM** , associado ao final da execução da função.
- Não há necessidade de preocupações com o estado de retorno, cuja manipulação fica automática, de responsabilidade dos mecanismos de implementação da linguagem hospedeira.
- Transições impossíveis deverão provocar desvios para um rótulo **ERRO** , onde poderá ser chamada uma rotina apropriada de manipulação da ocorrência de tal anormalidade.
- Para integrar o conjunto de sub-máquinas formado, é necessário um programa principal, encarregado apenas de dar partida ao processo de reconhecimento, através da criação do ambiente necessário à execução do reconhecedor.
- Assim, fica a cargo do programa principal a declaração das variáveis, o posicionamento das estruturas de dados em seus valores iniciais e a chamada da função booleana responsável pelo reconhecimento sintático correspondente ao não-terminal que representa a raiz da gramática.

Para o exemplo da expressão, aplicando-se estas regras obtém-se um programa semelhante ao seguinte (supõe-se existente a rotina **lexico** , que deposita em entrada o átomo extraído):

```
Boolean Procedure E();
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
        { INICIO: go to 1;
        1: if entrada= "a" then  { lexico(); go to 2 };
           if entrada= "[" then  { lexico(); go to 3 };
           go to erro;
        2: if entrada in { "*" , "+" } then  { lexico(); go to 1 };
           go to erro;
        3: if E() then go to 4;
           go to erro;
        4: if entrada= "]" then  { lexico(); go to 2 };
           go to erro;
        erro: E:= FALSE; return;
        FIM: E := TRUE; return;    }
Programa Principal:
        { lexico();
          if E() then print "Sentença Correta"
                 else print "Erro de Sintaxe";
          stop;     }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

3.2 Construção de extensões para a recuperação de erros

Reconhecedores de uma linguagem são dispositivos destinados à aceitação de cadeias de átomos que formem sentenças da linguagem. Na prática, entretanto, as cadeias fornecidas como entrada aos reconhecedores apresentam-se, muitas vezes, mal redigidas, não representando, portanto, sentenças corretas.

Nestas situações, frente a uma seqüencia não-esperada de átomos na cadeia de entrada, o reconhecedor deverá simplesmente rejeitá-la, uma vez que não tem condições de completar sua análise, por encontrar-se diante de uma cadeia que, não sendo uma sentença válida, não atende às especificações da linguagem aceita pelo autômato.

Para que seja possível ao autômato investigar a presença de novas incorreções no texto de entrada, convém que, uma vez detectado um erro, seja o autômato conduzido a uma situação em que os átomos que formam a cadeia incorreta possam, ainda assim, ser consumidos, e o autômato, reconduzido a uma situação que lhe permita prosseguir o reconhecimento do texto-fonte.

O estudo da recuperação de erros é extremamente complexo no caso geral, mesmo quando aplicado a reconhecedores de baixa complexidade, como é o caso dos autômatos finitos (BACKHOUSE, 1979), (TREMBLAY; SORENSON, 1985), (BARRETT; COUCH, 1979), (AHO; ULLMAN, 1979), (WAITE; GOOS, 1984), (LLORCA; PASCUAL, 1986), (HUNTER, 1981).

Por outro lado, a incidência de erros, nos casos reais, mostra um comportamento que permite, para as situações mais freqüentemente encontradas, simplificar muito a tarefa da recuperação de erros.

É porque se pode constatar, da observação de muitos casos, que é significativamente maior a incidência de erros simples que a de erros múltiplos, e que, no caso de se desejar a recuperação de erros múltiplos, é em geral satisfatório um tratamento simples e radical, em oposição aos complexos métodos de recuperação, minuciosos e dispendiosos, exigidos para a recuperação absoluta, mesmo de erros simples.

Tratando-se os erros múltiplos como casos de exceção, torna- se possível concentrar as atenções nos casos de recuperação de erros simples, em que a distância entre ocorrências sucessivas de erros é tal que nunca haja superposição entre as regiões de influência de dois quaisquer erros adjacentes ocorridos no texto de entrada.

Caracteriza-se a manifestação da presença de um erro em um texto-fonte como sendo a ocasião em que o reconhecedor atinge uma situação onde é impossível efetuar qualquer transição compatível com os símbolos que se encontram presentes no texto de entrada.

Em geral, a manifestação da presença de um erro não ocorre obrigatoriamente na posição exata do texto de entrada em que o erro realmente ocorreu: muitas vezes, algumas incorreções no texto-fonte são tais que permitem que o autômato continue a evoluir por uma seqüencia de transições que, embora incorreta no caso em questão, poderia ser válida para outras configurações legítimas do texto de entrada.

Isto faz com que, no caso geral, os resultados de uma eventual recuperação corretiva mecânica de erros deixem muito a desejar em relação aos que podem ser obtidos por meio de uma autêntica correção manual, efetuada pelo próprio autor do texto-fonte.

Desta maneira, a recuperação de erros não deve ser vista como um mecanismo de correção e eliminação de erros do texto-fonte, mas, simplesmente, como uma forma de ressincronização do reconhecedor, após atingido algum ponto de manifestação de erro, com a finalidade de permitir que, a despeito da presença do erro, o reconhecedor possa, ainda que rudimentar ou precariamente, prosseguir e completar a análise do texto de entrada.

Estas considerações levam à busca de um método em que, partindo-se de um autômato determinístico qualquer, seja possível incorporar com facilidade ao autômato extensões que lhe permitam conviver com a presença de erros no texto-fonte, recuperando-se de seus efeitos, de forma que o autômato possa sempre efetuar as transições previstas, ao longo de todos os trechos do texto-fonte que estejam situados fora das áreas de influência dos erros presentes.

Analisando-se os tipos possíveis de erros simples, constatam- se três possibilidades apenas: omissões de átomo, inclusões indevidas de átomo e substituições indevidas de um átomo por outro.

A recuperação de um erro de omissão de um átomo consiste, naturalmente, em se simular sua reinserção na posição correta do texto-fonte. No autômato, isto corresponde à criação, a partir do estado em que a omissão se manifesta, das mesmas transições que emergem do estado para onde o autômato transitaria caso não tivesse ocorrido a omissão do átomo.

No caso de ser encontrado, no texto-fonte, um átomo não-esperado, isto poderá ser interpretado como sendo devido a um dos dois seguintes fatores: a inserção, no texto-fonte, de um átomo para o qual não haja no autômato uma transição prevista partindo do estado corrente, ou então, a substituição indevida de um átomo correto por outro para o qual não haja transição prevista.

Em qualquer caso, torna-se necessário eliminar o átomo encontrado, e, em função do átomo seguinte, simular ou não a inserção de um novo átomo correto no texto-fonte.

Do ponto de vista do autômato, a eliminação do átomo incorreto é facilmente implementada através de uma transição, com origem no estado onde o erro se manifestou, e que consuma qualquer átomo que não seja

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

nenhum daqueles que normalmente o autômato consumiria naquele estado, e evoluindo para um novo estado, especialmente criado para esta finalidade.

Para completar a recuperação de um erro de inserção de átomo, é ainda necessário criar transições que, partindo do novo estado a que o átomo eliminado conduziu o autômato, reconduzam o autômato a estados-destino respectivamente idênticos aos das transições normais que emergem do estado onde o erro se manifestou.

Para a recuperação de um erro de substituição de átomo, devem ser ainda acrescentadas ao autômato transições, também partindo do estado a que o autômato foi conduzido após a eliminação do átomo estranho, e que exibam efeitos idênticos aos de todas as transições que emergem dos estados para onde o autômato original poderia ter sido conduzido como resultado de transições normais.

Um cuidado adicional, que apresenta efeitos benéficos do ponto de vista da preservação da estrutura do autômato original, consiste em se criar uma transição em vazio, partindo do estado em que o erro se manifestou, e com destino a um estado adicional correspondente, isolando, desta forma, o autômato original das extensões responsáveis pela recuperação dos erros.

Em operação, esta transição em vazio só deverá ser acionada caso nenhuma transição normal possa ser efetuada, ou seja, apenas quando, no estado em questão, ocorrer a manifestação de um erro.
Isto corresponde a atribuir uma prioridade maior à execução das transições normais, ignorando-se as demais, a não ser no caso da manifestação de erros.

O procedimento acima, que promove a geração da extensão necessária para que um autômato se recupere da presença de eventuais erros simples no texto-fonte, pode ser completado pelo acréscimo de transições adicionais, responsáveis pela recuperação de erros múltiplos.

Uma possível forma de *recuperação* para *erros múltiplos* é implementada aniquilando-se, da cadeia de entrada, uma vizinhança do primeiro ponto de manifestação de erro, vizinhança esta que seja suficientemente extensa para permitir que o autômato seja conduzido a um estado conveniente, onde seja altamente provável sua ressincronização com o texto de entrada restante.

Esta técnica, apesar de agressiva e radical, produz resultados satisfatórios na maioria dos casos, através de uma prática muito simples, como a descrita a seguir.

Elege-se, por inspeção da gramática, um conjunto de átomos, ditos *átomos sincronizadores*, cuja presença no texto-fonte determine inequivocamente o final, o início ou algum ponto notável do reconhecimento da construção sintática, e que se mostre estatisticamente muito freqüente nos textos-fonte.

De cada um dos estados criados para a extensão de recuperação, correspondentes aos pontos onde átomos espúrios são descartados, deve partir um conjunto de transições capaz de consumir qualquer átomo que não seja pertencente ao conjunto dos átomos sincronizadores, retornando ao mesmo estado, para prosseguir no consumo dos átomos do texto enquanto não for encontrado nenhum destes átomos.

Do mesmo estado deve adicionalmente partir outro conjunto de transições, encarregadas de consumir os átomos sincronizadores encontrados, e que tenham respectivamente como estados-destino os estados do autômato original para onde o autômato é conduzido quando do aparecimento normal de tais átomos.

Esta prática reconduz o autômato a uma situação na qual tudo se passa, para efeito do reconhecimento do restante do texto-fonte, como se o trecho do texto-fonte, consumido pela extensão de recuperação, tivesse sido substituido por outro, que levasse o autômato ao novo estado corrente.

Cabe aqui observar que, dependendo da escolha das transições adicionadas, os estados atingidos por ação das transições pertencentes às extensões de recuperação de erros múltiplos podem nunca ser alcançados por vias normais, a partir de transições provocadas por textos-fonte corretos que tenham como prefixo a parte do texto-fonte consumida antes da manifestação do erro.

Por ter implementação tão mais simples quanto menor for o grau de exigência imposto ao aproveitamento do texto que segue o ponto de manifestação do erro múltiplo, esta técnica tem ampla aplicabilidade, constatando-se freqüentemente na prática seu emprego em inúmeras implementações comercialmente disponíveis de linguagens de programação.

3.2.1 Erros simples em autômatos finitos determinísticos

Com base no conjunto de considerações acima, é possível estabelecer um conjunto de regras tais que, dado um autômato finito determinístico, isento portanto de transições em vazio, permita criar estados e transições adicionais capazes de implementar a recuperação total dos erros simples, e ainda tratar erros múltiplos através da técnica descrita anteriormente.

Descreve-se a seguir um procedimento para a construção de extensões de recuperação de erros em autômatos finitos determinísticos.

É necessário para isto considerar cada estado i do autômato como sendo um potencial ponto de manifestação da presença de erros, simples ou múltiplos.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Procedimento para a construção de extensões de recuperação de erros em autômatos finitos determinísticos

- Seja **j** um estado pertencente ao conjunto de primeiros sucessores do estado **i**, ou seja, um estado **j** tal que exista, para algum terminal **a**, uma transição do tipo

$$(i , a) \rightarrow j$$

- Seja **k** um estado pertencente ao conjunto de segundos sucessores do estado **i**, ou, em outras palavras, um estado que pertença ao conjunto-união dos conjuntos de primeiros sucessores dos estados **j** anteriormente obtidos, existindo portanto transições da forma:

$$(j , b) \rightarrow k$$

- Para cada tripla **(i , j , k)** nestas condições, criar, em correspondência ao estado **i**, dois novos estados, i_1 e i_2 , da extensão de recuperação do autômato.
- Criar, para cada estado **i** do autômato, transições em vazio para isolar a extensão de recuperação de erros, preservando assim a topologia básica do autômato original:

$$(i_1 , \varepsilon) \rightarrow i$$

- Criar, para cada estado **i** , um conjunto de transições encarregadas da eliminação de átomos **σ** espúrios:

$$(i_1 , \sigma) \rightarrow i_2$$

  para cada um dos átomos **σ** do alfabeto, que não coincida com nenhum dos átomos consumidos pelas transições normais que emergem do estado **i**.
- Criar, para cada estado i_2 , um conjunto de transições que se encarreguem de, em uma situação pós-eliminação do átomo espúrio, reconduzir o autômato ao estado correto, consumindo átomos a que deveriam ter sido encontrados no estado **i**, recuperando desta maneira o erro de inserção:

$$(i_2 , a) \rightarrow j$$

- Criar, analogamente, transições para recuperar erros de substituição indevida de átomos:

$$(i_2 , b) \rightarrow k$$

- Criar, por raciocínio similar, para cada estado i_2 , um conjunto de transições encarregadas da recuperação de erros de eliminação de átomos **a** :

$$(i_1 , b) \rightarrow k$$

- Repetir este procedimento para todos os estados **i** do autômato.

■

Ressalve-se que foi descrito acima o tratamento de estados **i** que não são estados finais, e que apresentam primeiro(s) e segundo(s) sucessor(es), **j** e **k.**

Para completar os procedimentos de extensão para a recuperação de erros simples em autômatos finitos determinísticos, cumpre ainda considerar os casos em que cada um destes estados possa ser um estado final, e aqueles em que não existam segundos ou mesmo primeiros sucessores para algum estado **i** :

Casos particulares a serem considerados:

- É irrelevante ser ou não final o estado **k** .
- Se nem **i** nem **j** forem estados finais, os estados i_1 e i_2 não deverão ser feitos estados finais.
- O estado i_1 deverá ser feito estado final caso qualquer um dos estados **i** ou **j** seja um estado final.
- O estado i_2 deverá ser feito estado final somente se o estado **i** o for.
- Caso não ocorra, para algum estado **j** existente, nenhum sucessor **k**, deve-se omitir, obviamente, a construção das transições que teriam destino em **k**.
- Caso não exista um único sucessor **j** sequer para o estado **i**, omite-se também a construção das transições que teriam como destino o estado **j**.

■

Isto completa os procedimentos de extensão para a recuperação de erros simples em autômatos finitos determinísticos.

3.2.2 Erros simples em autômatos finitos não-determinísticos

Para o caso de máquinas não-determinísticas, estes procedimentos devem ser modificados, para levar em conta a presença de transições em vazio, bem como de transições responsáveis pelo consumo de um mesmo átomo, originadas em um mesmo estado, e com estados-destino diversos.

As alterações necessárias ao procedimento de geração das extensões para a recuperação de erros confinam-se essencialmente a modificações no processo de obtenção do conjunto dos primeiros sucessores **j** de

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

um dado estado **i**, permanecendo inalterado o algoritmo de construção da extensão de recuperação de erros do autômato. Obviamente a obtenção do conjunto dos segundos sucessores, neste caso geral, deverá ser baseada na aplicação repetida deste algoritmo.

Dado um estado **i**, pode-se reinterpretar seu primeiro sucessor como sendo um estado para onde o autômato pode evoluir, partindo deste estado e percorrendo eventualmente uma cadeia de transições em vazio, até consumir finalmente um primeiro átomo **a**.

Torna-se assim necessário levantar, para cada estado **i**, as seqüencias de transições do seguinte tipo:

$$(i, \varepsilon) \rightarrow k_1$$
$$(k_1, \varepsilon) \rightarrow k_2$$
$$\ldots$$
$$(k_{n-1}, \varepsilon) \rightarrow k_n$$
$$(k_n, a) \rightarrow j$$

incluindo os estados **j** no conjunto dos estados primeiros sucessores de **i** para cada uma das seqüencias deste tipo que forem encontradas.

Observando-se o comportamento das transições cujo não- determinismo tenha como causa a evolução do autômato de um estado **i** para primeiros sucessores j_p distintos, através do consumo final de um mesmo átomo **a** $(n \geq 0, p=1, \ldots, m)$:

$$(i, \varepsilon) \rightarrow k_1$$
$$(k_1, \varepsilon) \rightarrow k_2$$
$$\ldots$$
$$(k_{n-1}, \varepsilon) \rightarrow k_n$$
$$(k_n, a) \rightarrow j_p$$

pode-se notar que a presença desta classe de não-determinismos não é relevante do ponto de vista dos algoritmos de obtenção das extensões do autômato para efeito de recuperação de erros.

Note-se que os grupos de transições acima podem exibir $n=0$ nos casos em que não haja transições em vazio a partir do estado **i**. Neste caso, o consumo do átomo **a** é efetuado diretamente a partir do estado **i**, com destino aos estados $j_1, \ldots, j_m$, reduzindo a uma única transição toda a seqüencia.

Em qualquer caso, o efeito destes não-determinismos sobre as extensões é notada apenas pelo fato de que, do estado i_2 da extensão, irão emanar **m** transições múltiplas, com destinos finais $j_1, j_2, \ldots, j_m$, que consomem o mesmo átomo, **a**.

Conclui-se que o método não sofre alterações, a menos do algoritmo de obtenção dos estados sucessores, permitindo construir quaisquer extensões de recuperação de erros simples para autômatos finitos, que contenham ou não transições não-determinísticas.

3.2.3 Erros múltiplos em autômatos finitos

Uma vez montada a extensão responsável pela recuperação de erros simples, de acordo com o procedimento acima, pode-se inserir extensões adicionais para a recuperação de *erros múltiplos*.

Para isto, escolher, com base na gramática e em informações sobre a freqüencia de aparecimento das diversas construções sintáticas da linguagem em textos reais, um conjunto S de *átomos de sincronização*.

O conjunto **S** deve ser convenientemente escolhido no conjunto de símbolos de entrada do autômato, de forma que seus elementos representem símbolos cuja presença no texto-fonte indique de alguma forma a existência de uma alta probabilidade de estar sendo iniciada ou terminada alguma construção sintática bem definida, ou que, ao menos, um ponto notável do texto-fonte tenha sido atingido.

Na prática, é usual a inclusão, neste conjunto, de símbolos ou palavras-chave da linguagem que iniciam ou terminam comandos, expressões ou agrupamentos, bem como operadores, sinais de pontuação e separadores. Dentre estes, os mais convenientes são aqueles que não participam de duas ou mais formas sintáticas que possam ocorrer simultaneamente em qualquer ponto do texto-fonte.

Para cada um dos estados i_2, gerados na extensão de recuperação de erros simples pelo procedimento anteriormente descrito, criar transições com destino ao próprio estado i_2, responsáveis pelo consumo de átomos σ que não pertençam ao conjunto **S**, e que excluam todos os átomos **a** e **b** que possam ser consumidos pelas transições que partem de i_2, na extensão de recuperação de erros simples já construida:

$$(i_2, \sigma) \rightarrow i_2$$

Para cada átomo **s** do conjunto S criar uma transição para o estado i_s do autômato original, correspondente ao estado-destino da transição que consome **s** neste autômato, ou seja, para cada transição do tipo

$$(x, s) \rightarrow i_s$$

do autômato original, criar um conjunto de transições

$$(i_2, s) \rightarrow i_s$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

para cada estado i_2 criado para a recuperação de erros simples.

Naturalmente, são mais convenientes para esta finalidade os átomos s tais que, ao menos no contexto em que deve ser desenvolvida a recuperação de erros em pauta, o correspondente estado i_s seja único.

Cabe ainda outra observação, relativa à utilização da técnica de recuperação de erros múltiplos descrita anteriormente: embora no texto acima a técnica tenha sido aplicada como complemento de uma recuperação absoluta de erros simples, isto não é obrigatório, podendo-se, em aplicações onde não for essencial uma recuperação precisa de erros simples, utilizá-la de forma isolada, diretamente sobre o autômato original.

Isto completa a criação da extensão do autômato original, encarregada da recuperação de erros. Deve-se notar que é usual que o autômato assim estendido apresente muito mais estados e transições que o autômato original, e que apresente um comportamento não-determinístico, exigindo, para que possa ser adequadamente utilizado na prática, a aplicação de procedimentos de simplificação.

Pelo fato de serem independentes as partes da extensão do autômato referentes a cada estado, torna-se possível considerar cada uma destas partes como um autômato separado, responsável pela recuperação exclusiva dos erros manifestados em cada particular estado do autômato original.

Convém ressaltar que, como as extensões do autômato só são ativadas quando da ocorrência de erros, seu comportamento é de fato independente das atividades do autômato original, não sendo portanto desejável que este seja afetado pelos algoritmos de redução eventualmente aplicados nesta fase.

Esta independência cria ainda uma opção para o projetista, permitindo que, de acordo com as conveniências, sejam utilizados apenas as extensões desejadas, correspondentes apenas a uma parte dos estados do autômato, podendo para os demais ser omitida a recuperação de erros, ou ser dado um tratamento diferenciado, a critério do projetista.

Uma opção prática consiste no pré-cálculo das extensões de recuperação para cada estado, sem implementação física no autômato, ocorrendo sua ativação apenas quando da real manifestação de um erro que exija ações de recuperação.

3.2.4 Erros simples em autômatos de pilha estruturados

É vasta a literatura que versa sobre o tema da recuperação de erros nos autômatos de pilha gerados pelos métodos tradicionais, baseados nas técnicas de reconhecimento LL(k) e LR(k) (AHO; ULLMAN, 1979), (TREMBLAY; SORENSON, 1985), (GOUGH, 1988), (LLORCA; PASCUAL, 1986).

Por serem profundamente baseados no uso de autômatos finitos, pode-se dizer que, para os autômatos de pilha estruturados, as técnicas aplicáveis aos autômatos finitos podem ser diretamente empregadas na construção parcial das extensões dos autômatos de recuperação de erros, ou, mais especificamente, na recuperação dos erros que se manifestam durante a execução exclusiva de transições internas às sub-máquinas.

Para completar o desenvolvimento do método, é necessário incorporar aos algoritmos já estabelecidos para os autômatos finitos alguns mecanismos adicionais, que se responsabilizem pelos erros manifestados na ocasião da execução de transições entre uma sub-máquina e outra, ou seja, de transições de chamada ou de retorno de sub-máquina, transições essas que se dão em vazio.

É necessário ainda que seja devidamente considerada a interpretação correta do papel dos estados finais das sub-máquinas, conceitualmente diverso daquele exibido pelos estados finais do autômato finito, uma vez que nestes, tais estados são destinados a identificar o final do reconhecimento das sentenças da linguagem, enquanto nos autômatos de pilha estruturados sua função é de identificar especificamente o final do reconhecimento da construção sintática definida pela sub-máquina em questão.

Por esta razão, enquanto nos autômatos finitos a manifestação de um erro em um estado final deve levar ao acionamento de uma transição de recuperação de erros, nos autômatos de pilha estruturados tal manifestação em uma sub-máquina deve ser, geralmente, interpretada simplesmente como uma condição válida para promover o retorno à sua sub-máquina chamadora.

Assim, estudando-se o papel dos estados finais das sub-máquinas, pode-se chegar à conclusão que, caso ocorra uma suposta manifestação de erro em algum estado final de sub-máquina que não seja também estado final do autômato, torna-se necessário, para a caracterização do tratamento a ser usado na recuperação, investigar o comportamento do autômato nos seus estados sucessores, ou seja, em todos os estados que podem ser atingidos, após o consumo de um átomo, uma vez executada a transição de retorno a partir do estado corrente.

Desta forma, além da recuperação dos erros correspondentes às transições internas, em tudo similar à que é feita em autômatos finitos, é ainda preciso incorporar mecanismos que considerem as inter-relações entre as diversas sub-máquinas que compõem o autômato.

A alteração mais significativa no método, em relação ao apresentado para os autômatos finitos, relaciona-se novamente com a obtenção do conjunto dos primeiros sucessores de um dado estado, e, conseqüentemente, do conjunto dos possíveis átomos que podem ser consumidos para atingi-los a partir do estado em questão.

A seguir, é descrito, de forma recursiva, um método para a obtenção do conjunto dos primeiros sucessores $q_{x,y}$ de um estado $Q_{m,i}$ (i-ésimo estado da m-ésima sub-máquina do autômato).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Ao final, são apresentadas as definições de alguns conjuntos úteis para este estudo: **PRIMEIRO**, **ATOMOS**, **SEGUNDO** e **SEGUINTE**, os quais serão especificados como funções dependentes dos estados em que se deseja efetuar a recuperação de erros.

Procedimento para a obtenção do conjunto de sucessores

- Sucessores referentes a transições internas: incluir no conjunto de primeiros sucessores de $q_{m,i}$ todos os estados diretamente atingidos através do consumo de átomos **c** ao final de uma seqüencia do tipo seguinte, envolvendo apenas transições internas à sub-máquina **m**. Neste caso, **x=m** e **y=j**, com $n \geq 0$:

$$\gamma\ q_{m,i}\ \alpha \rightarrow \gamma\ q_{m,x}\ \alpha$$
$$\gamma\ q_{m,x1}\ \alpha \rightarrow \gamma\ q_{m,x2}\ \alpha$$
$$\gamma\ q_{m,xn-1}\ \alpha \rightarrow \gamma\ q_{m,xn}\ \alpha$$
$$\gamma\ q_{m,xn}\ c\alpha \rightarrow \gamma\ q_{m,j}\ \alpha$$

  Devem neste caso ser incluidos no conjunto de primeiros sucessores do estado $q_{m,i}$ todos os estados $q_{m,j}$ atingíveis a partir de $q_{m,i}$ ao fim de uma cadeia de transições deste tipo.
  Para $n = 0$ tem-se $x_n = i$. Estes são os estados para os quais se aplica o método já estudado de recuperação de erros em autômatos finitos.
- sucessores referentes às transições de retorno de sub-máquina: caso $q_{m,i}$ seja um estado final da sub-máquina **m**, incorporar, ao conjunto dos seus sucessores referentes às transições internas, todos os estados que sejam primeiros sucessores do estado $q_{r,s}$ para onde o retorno deve ser efetuado. Assim, para cada produção do tipo

$$\gamma(r,\ s)\ q_{m,i}\ \alpha \rightarrow \gamma\ q_{r,s}\ \alpha$$

  devem ser incluidos no conjunto dos primeiros sucessores de $q_{m,i}$ todos os primeiros sucessores do estado $q_{r,s}$.
- sucessores referentes às transições de chamada de sub-máquina:
  Para cada produção do tipo

$$\gamma\ q_{m,i}\ \alpha \rightarrow \gamma(m,\ j)\ q_{r,0}\ \alpha$$

  em que $q_{m,i}$ seja caracterizado como um estado a partir do qual alguma sub-máquina **r** seja chamada, deverão ser incluidos no conjunto dos primeiros sucessores de $q_{m,i}$ todos os estados que sejam primeiros sucessores do estado inicial $q_{r,0}$ da sub-máquina **r** chamada.

■

São definidos a seguir alguns conjuntos auxiliares, úteis para o desenvolvimento subseqüente do método de extensão do autômato de pilha estruturado para a recuperação de erros simples.

Seja $q_{m,i}$ um estado onde se deseja efetuar o estudo para a criação da extensão de recuperação de erros.

A cada um dos estados $q_{x,y}$, primeiros sucessores de $q_{m,i}$, pode-se associar um conjunto **ATOMOS**($q_{m,i}$, $q_{x,y}$), formado por todos os possíveis átomos que o autômato pode consumir em primeiro lugar, partindo de $q_{m,i}$, para atingir o estado $q_{x,y}$.

Unindo-se todos estes conjuntos obtém-se, para o estado $Q_{m,i}$, o conjunto **PRIMEIRO**($q_{m,i}$), formado por todos os possíveis átomos que podem ser consumidos em primeiro lugar pelo autômato, a partir de $q_{m,i}$.

Colecionam-se em cada um dos conjuntos **ATOMOS**($q_{m,i}$, $q_{x,y}$) os átomos **c** que possam ser consumidos pelas transições internas da sub-máquina **x** que finalizam as cadeias de transições que permitem levar o autômato do estado $q_{m,i}$ ao estado $q_{x,y}$:

$$\gamma\ q_{x,z}\ c\alpha \rightarrow \gamma\ q_{x,y}\ \alpha$$

Em autômatos determinísticos os conjuntos assim construidos são disjuntos, de forma que, partindo-se de um estado $q_{m,i}$, cada possível átomo a ser consumido só pode levar o autômato a um único estado que seja seu primeiro sucessor, e que lhe é característico. Esta propriedade pode ser explorada para melhorar o desempenho das implementações do modelo.

Pode-se definir ainda o conjunto **SEGUNDO**($q_{m,i}$) como sendo a união de todos os conjuntos **PRIMEIRO**($q_{x,y}$), para todo $q_{x,y}$ que seja primeiro sucessor do estado $q_{m,i}$ .Este conjunto indica todos os átomos que podem ser consumidos em segundo lugar a partir do estado $q_{m,i}$.

Outra definição útil é a do conjunto **SEGUINTE**($q_{m,i}$, **c**), que é a coleção de todos os elementos de **SEGUNDO**($q_{m,i}$) que podem ser consumidos pelo autômato após o consumo do átomo c, partindo-se do estado $q_{m,i}$.

Sendo $q_{x,y}$ um estado sucessor de $q_{m,i}$ tal que o átomo **c** pertença ao conjunto **ATOMOS**($q_{m,i}$, $q_{x,y}$), é possível obter o conjunto **SEGUINTE**($q_{m,i}$, **c**) construindo-se simplesmente o conjunto **PRIMEIRO**($q_{x,y}$).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

De posse destas definições e da filosofia de recuperação de erros a ser adotada, passa-se a desenvolver os métodos de extensão com os quais se pretende recuperar erros em autômatos de pilha estruturados, para os quais serão delineados procedimentos similares aos apresentados anteriormente para autômatos finitos.

■

Procedimento de montagem das extensões de recuperação de erros simples em autômatos de pilha estruturados

Pode-se então levantar as regras de construção das extensões ao autômato de pilha estruturados que se responsabilizem pela sua recuperação no caso da ocorrência de erros simples.

A recuperação de erros simples em autômatos de pilha estruturados deve considerar os seguintes casos:

- recuperação de erros interna a uma sub-máquina
- recuperação de erros ocorridos em estados finais de sub-máquina
- recuperação de erros ocorridos quando da chamada de sub-máquina

Seja $q_{m,i}$ um estado qualquer do autômato, a ser considerado como ponto potencial de manifestação da presença de erros na cadeia de entrada.

Seja $q_{m,j}$ um estado-destino de alguma transição interna da sub-máquina m, com origem em $q_{m,i}$ ou um estado de retorno de alguma transição de chamada de sub-máquina com origem em $q_{m,i}$:

$$\gamma\ q_{m,i}\ \alpha \rightarrow \gamma\ q_{m,j}\ \alpha$$

ou $$\gamma\ q_{m,i}\ c\alpha \rightarrow \gamma\ q_{m,j}\ \alpha$$

ou $$\gamma\ q_{m,i}\ \alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\ \alpha$$

Seja $q_{m,k}$ estado-destino de alguma transição com origem em $q_{m,j}$, ou um estado de retorno de alguma transição de chamada de sub-máquina, com origem em $q_{m,j}$:

$\gamma\ q_{m,k}\ \alpha \rightarrow \gamma\ q_{m,k}\ \alpha$

ou $\gamma\ q_{m,j}\ c\alpha \rightarrow \gamma\ q_{m,k}\ \alpha$

ou $\gamma\ q_{m,j}\ \alpha \rightarrow \gamma(m,\ k)\ q_{n,0}\ \alpha$

São a seguir analisados os três casos de recuperação de erros acima mencionados:

Recuperação interna à sub-máquina:

Inicialmente, considerando que o comportamento das sub-máquinas é idêntica à dos autômatos finitos enquanto estiverem realizando apenas transições internas, já que não há movimentação da pilha, basta transpor para os autômatos de pilha estruturados as regras desenvolvidas anteriormente para automatos finitos:

- Para cada estado $q_{m,i}$, criar, para a extensão de recuperação de erros, os estados $q_{m,i1}$ e $q_{m,i2}$.
- Criar uma transição em vazio para isolar o autômato principal da sua extensão de recuperação de erros:

  $$\gamma\ q_{m,i}\ \alpha \rightarrow \gamma\ q_{m,i1}\ \alpha$$

- Criar uma transição para a eliminação de átomos espúrios:

  $$\gamma\ q_{m,i1}\ \sigma a \rightarrow \gamma\ q_{m,i2}\ \alpha$$

  onde σ denota qualquer átomo que não seja pertencente ao conjunto **ATOMOS**$(q_{m,i}\ ,\ q_{m,j}\ )$.
- Criar para cada átomo c do conjunto **ATOMOS**$(q_{m,i}\ ,\ q_{m,j}\ )$, uma transição para a recuperação de erros de inserção de átomos espúrios:

  $$\gamma\ q_{m,i2}\ c\alpha \rightarrow \gamma\ q_{m,j}\ \alpha$$

- Criar para cada átomo **b** do conjunto **ATOMOS**$(q_{m,j},\ q_{m,k})$, uma transição responsável pela recuperação dos erros de substituição indevida de átomos:

  $$\gamma\ q_{m,i2}\ b\alpha \rightarrow \gamma\ q_{m,k}\ \alpha$$

- Este procedimento deve ser repetido para cada um dos estados $q_{m,i}$ do autômato.

Resta, para completar a construção da extensão responsável pela recuperação de erros simples no âmbito estrito de uma dada sub-máquina **m**, considerar o efeito causado pelo fato de algum dos estados $q_{m,i}$, $q_{m,j}$ ou $q_{m,k}$ ser estado final do autômato ou da sub-máquina **m**:

- É irrelevante se o estado $q_{m,k}$ é ou não estado final
- Se nenhum dos estados $q_{m,i}$ ou $q_{m,j}$ for estado final, então nem $q_{m,i1}$ nem $q_{m,i2}$ deverão ser feitos estados finais.
- O estado $q_{m,i1}$ deverá ser feito estado final caso $q_{m,i}$ ou $q_{m,j}$ ou ambos sejam estados finais.
- O estado $q_{m,i2}$ deve ser feito estado final caso o estado $q_{m,i}$ seja um estado final.
- As produções que fazem dos estados $q_{m,i1}$ e $q_{m,i2}$ estados finais são, respectivamente, das seguintes formas:

  $$\gamma(x,\ y)\ q_{m,i1}\ \alpha \rightarrow \gamma\ q_{x,y}\ \alpha$$

  e $$\gamma(x,\ y)\ q_{m,i2}\ \alpha \rightarrow \gamma\ q_{x,y}\ \alpha$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- Não ocorrendo, para algum $q_{m,k}$ existente, nenhum sucessor $q_{m,k}$, omite-se, naturalmente, a construção das transições da extensão de recuperação que teriam destino em $q_{m,k}$.
- Não havendo sucessores $q_{m,j}$ para o estado $q_{m,i}$, devem ser também omitidas da extensão todas as produções que representem transições de recuperação que teriam como destinos aqueles estados.
- Para completar a extensão de recuperação do autômato, é preciso ainda incorporar às extensões já estudadas os efeitos da interação entre sub-máquinas, os quais se manifestam apenas se algum dos estados $q_{m,i}$ ou $q_{m,j}$ for estado final da sub-máquina (estado de retorno) sem ser estado final do autômato, ou então se ao menos um dos estados $q_{m,i}$ ou $q_{m,j}$ for um estado de onde parte alguma chamada de sub-máquina.

■

Recuperação em estados finais de sub-máquina:

Este método de recuperação de erros simples em estados finais de sub-máquina adota a política de forçar o retorno à sub-máquina chamadora apenas se o átomo corrente for garantidamente um daqueles que poderão ser consumidos uma vez efetivado o retorno.

- Se $q_{m,i}$ for estado final, tem-se no autômato uma produção do tipo

$$\gamma(r,\ s)\ q_{m,i}\ \alpha \rightarrow \gamma\ q_{r,s}\ \alpha$$

Neste caso, é preciso garantir que o retorno à sub-máquina chamadora só seja efetuado diretamente se o próximo átomo a ser consumido estiver entre os elementos do conjunto **PRIMEIRO**$(q_{r,s})$, permitindo que os demais casos sejam analisados sob o prisma da recuperação de erros.

- Para garantir que o retorno se processe apenas nos casos corretos, substitui-se a produção incondicional por produções do tipo:

$$\gamma(r,\ s)\ q_{m,i}\ c\alpha \rightarrow \gamma\ q_{r,s}\ c\alpha$$

para cada átomo **c** do conjunto **PRIMEIRO**$(q_{r,s})$, passível de vir a ser consumido a partir do estado $q_{r,s}$, como decorrência do retorno a partir do estado $q_{m,i}$.

- Incluir, correspondentemente, produções do tipo

$$\gamma(r,\ s)\ q_{m,i2}\ \ c\alpha \rightarrow \gamma\ q_{r,s}\ \ c\alpha$$

para recuperar erros de inserção de átomos no estado $q_{m,i}$.

Note-se que os átomos c não são consumidos pelas produções acima, sendo apenas consultados ("look-ahead") para permitir a decisão acerca da transição a ser executada em cada caso.

- Para recuperar erros de omissão e de substituição de átomos em $q_{m,i}$ incluem-se respectivamente transições das formas:

$$\gamma(r,\ s)\ q_{m,i1}\ \ d\alpha \rightarrow \gamma\ q_{r,s}\ \ d\alpha$$

e $$\gamma(r,\ s)\ q_{m,i2}\ \ d\alpha \rightarrow \gamma\ q_{r,s}\ \ d\alpha$$

para cada átomo **d** pertencente ao conjunto **SEGUNDO**$(q_{r,s})$.

Estas produções garantem que o retorno se dê apenas se houver a garantia de que o átomo corrente será de fato consumido após o retorno à sub-máquina chamadora.

Observe-se que o átomo em questão não é consumido neste ponto, o que permite que a sub-máquina chamadora efetue as operações de recuperação de forma mais apropriada que seria possível neste ponto.

Isto tem grande utilidade na elaboração prática de compiladores baseados nesta classe de reconhecedores.

- Caso $q_{m,j}$ seja um estado final, devem ser ainda criadas produções das formas:

$$\gamma(r,\ s)\ q_{m,i1}\ \ c\alpha \rightarrow \gamma\ q_{r,s}\ c\alpha$$

e $$\gamma(r,\ s)\ q_{m,i}\ \ c\alpha \rightarrow \gamma\ q_{r,s}\ c\alpha$$

para cada átomo c pertencente ao conjunto **PRIMEIRO**$(q_{r,s})$, destinadas à recuperação de erros de omissão, inserção e de substituição, que se manifestem no estado $q_{m,i}$.

■

Recuperação em chamadas de sub-máquinas:

Resta, para completar o método de recuperação de erros simples, considerar a recuperação dos erros que se manifestam em estados em que são efetuadas chamadas de sub-máquina. Há dois casos a tratar: as chamadas de sub-máquina originadas no estado $q_{m,i}$ e aquelas originadas em $q_{m,j}$.

- Cada chamada de alguma sub-máquina n, originada no estado $q_{m,i}$ é dada por uma produção da forma:

$$\gamma\ q_{m,i}\ \ \alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\ \ \alpha$$

- Para tais casos, substitui-se a produção de chamada incondicional acima por produções da forma:

$$\gamma\ q_{m,i}\ \ c\alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\ \ c\alpha$$

para cada átomo **c** pertencente ao conjunto **PRIMEIRO**$(q_{n,0})$, para garantir que a chamada só se realize caso o átomo corrente puder ser efetivamente consumido após executada a chamada.

- Para recuperar erros de inserção, incluem-se, correspondentemente, produções da forma:

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$\gamma\ q_{m,i2}\quad c\alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\quad c\alpha$

- Erros de inserção e de substituição são recuperados através da inclusão de produções do tipo

$\gamma\ q_{m,i1}\quad d\alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\quad d\alpha$

e $\gamma\ q_{m,i2}\quad d\alpha \rightarrow \gamma(m,\ j)\ q_{n,0}\quad d\alpha$

para cada átomo d pertencente ao conjunto **SEGUNDO**$(q_{n,0})$.

- Caso o estado $q_{m,j}$ seja a origem da chamada, tem-se uma produção da forma:

$\gamma\ q_{m,j}\quad \alpha \rightarrow \gamma(m,\ k)\ q_{n,0}\quad \alpha$

- Nesta situação, desde que haja no mínimo uma produção com consumo de átomo, originada em $q_{m,i}$ e com destino em $q_{m,j}$:

$\gamma\ q_{m,i}\quad c\alpha \rightarrow \gamma\ q_{m,j}\quad \alpha$

Incluem-se, para a recuperação de erros de inserção e de substituição, produções do tipo:

$\gamma\ q_{m,i1}\ c\alpha \rightarrow \gamma(m,\ k)\ q_{n,0}\ c\alpha$

e $\gamma\ q_{m,i2}\ c\alpha \rightarrow \gamma(m,\ k)\ q_{n,0}\ c\alpha$

para cada átomo **c** pertencente ao conjunto **PRIMEIRO**$(q_{n,0})$.

- Não sendo observada a condição de haver no mínimo uma produção responsável pelo consumo de algum átomo na transição do estado $q_{m,i}$ para o estado $q_{m,j}$, não deve ser incluída qualquer produção de recuperação relativa à chamada de sub-máquina originada em $q_{m,j}$.

■

Por meio deste procedimento, todos os erros simples podem ser tratados pelo mecanismo de recuperação apresentado, restando a serem considerados apenas os casos de erros simultâneos que ocorram em pontos muito próximos, na cadeia de entrada.

3.2.5 Erros múltiplos em autômatos de pilha estruturados

De forma similar ao que foi apresentado no caso dos autômatos finitos, a recuperação absoluta de erros múltiplos mostra-se uma tarefa de elevada dificuldade, em virtude do caráter do problema e da maior complexidade do autômato de pilha em relação ao autômato finito.

Assim, em adição aos já intrincados mecanismos de recuperação necessários no caso dos autômatos finitos, tem-se agora um problema adicional a considerar, decorrente da presença de uma pilha e das consultas que a ela devem fazer as produções do autômato.

Apresenta-se aqui um método para a recuperação de erros múltiplos inspirado nos mesmos princípios utilizados para os autômatos finitos, levando, portanto, a uma recuperação aproximada e não absoluta, como é o caso na recuperação de erros simples.

Para efeitos práticos dois casos apenas serão considerados formalmente: o caso em que o erro se manifesta na submáquina a que pertence o estado-destino das transições de recuperação do erro, e um caso mais complexo, em que tal estado pertence à submáquina que ativou a submáquina em que o erro se manifestou. Para outros casos, serão apenas recomendados alguns critérios empíricos.

No primeiro caso, a recuperação de erro pode ser efetuada localmente, de modo similar à empregada nos autômatos finitos, uma vez que envolve apenas transições internas à submáquina.

No segundo, surge a necessidade de se forçar transições entre as submáquinas envolvidas, exigindo-se para tanto a análise do conteúdo da pilha. Opta-se, no caso, pela escolha das transições de recuperação que tenham como destino a submáquina ancestral a que pertença o estado referenciado no topo da pilha.

Recuperações de erros envolvendo desvios entre a submáquina corrente e alguma outra submáquina externa deverão ser acompanhadas de desempilhamentos suficientes para que uma referência a tal submáquina seja atingida na pilha, desempilhando-se, no caso, inclusive o estado de retorno correpondente.

A extensão do autômato de pilha estruturado, responsável pela recuperação de erros múltiplos, dá-se acrescentando-se a cada submáquina **m** um estado adicional $q_{m,M}$, onde uma transição

$\gamma\ q_{m,M}\ \sigma a \rightarrow \gamma\ q_{m,M}\ \alpha$

responsável pelo consumo de todos os átomos $\sigma \notin S$ não-pertencentes ao conjunto de átomos de sincronização escolhidos para a gramática.

O acesso ao estado $q_{m,M}$ se dá a partir dos estados $q_{m,i2}$,anteriormente criados, através do consumo de átomos $\sigma \notin S$:

$\gamma\ q_{m,i2}\ \sigma a \rightarrow \gamma\ q_{m,M}\ \alpha$

A partir de $q_{m,M}$, emergem, com destino a estados da própria submáquina, transições que consomem átomos $s \in S$, tais que s sejam consumidos pela mesma submáquina em transições do tipo

$\gamma\ qm,u\ s\alpha \rightarrow \gamma\ q_{m,v}\ \alpha$

Para estas, as transições de recuperação assumem a forma:

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$\gamma\ q_{m,M}\ s\alpha \rightarrow \gamma\ q_{m,v}\ \alpha$

Para cada $s \in S$, com s não consumido na mesma submáquina, mas sim por transições de uma submáquina m' , com $m' \neq m$, do tipo seguinte:

$\gamma\ q_{m',u}\ s\alpha \rightarrow \gamma\ q_{m',v}\ \alpha$

devem ser criadas transições da forma:

$\gamma(m' , x)\ q_{m,M}\ s\alpha \rightarrow \gamma\ q_{m',v}\ \alpha$

as quais desempilham (m' , x) , eliminando o retorno normal para o estado $q_{m',x}$ e forçando em seu lugar um desvio para $q_{m',v}$.

Note-se que a eficácia deste método baseia-se fundamentalmente na escolha do conjunto S , cujos elementos devem, de preferência, ter propriedades disjuntas, uma vez que, no caso oposto, poderiam surgir indecisões adicionais.

A escolha adequada de S permite a criação de extensões de que efetuam a recuperação desejada entre a submáquina e suas chamadoras.

Também aqui valem os critérios para a escolha dos elementos de S com base em símbolos indicadores de pontos notáveis da sintaxe, porém um critério empírico adicional pode ser empregado, o qual leva em conta não apenas os pontos que sejam estratégicos para a ressincronização interna da submáquina em que o erro se manifestou mas também os da sua chamadora e os de eventuais submáquinas que ela esteja prestes a chamar.

As extensões implementadas pelas transições acima introduzidas promovem um desempilhamento condicionado ao átomo corrente e ao conteúdo do topo da pilha apenas, resultando a recuperação em questão em uma operação entre submáquinas restrito a um único nível de chamadas de submáquina.

Note-se que, apesar de ser possível estender o raciocínio para cobrir os casos restantes, o método se apresentaria pouco eficaz, visto que a certeza da recuperação é substancialmente diminuida à medida que cresce o número de chamadas de submáquinas a serem consideradas na recuperação.

Assim, torna-se recomendável, neste caso, a aplicação de um tratamento radical, forçando que sejam ignorados do texto-fonte átomos suficientes para que o autômato passa ressincronizar-se com o início de alguma construção sintática significativa presente no texto de entrada.

A pilha deverá ser alterada, caso seja necessário, para que seja possível o devido encadeamento de chamadas e retornos das submáquinas envolvidas. Trata-se de um trabalho não-trivial, de cunho artesanal, a ser resolvido caso a caso de acordo com as peculiaridade de cada linguagem ou gramática.

3.3 Obtenção de Analisadores Sintáticos

Embora em uma vasta maioria dos casos práticos seja possível a construção de bons núcleos para interpretadores ou compiladores das linguagens de programação de interesse, há casos em que se exige sua atuação como reconhecedores sintáticos autênticos, e não apenas como meros aceitadores.

Nestas situações torna-se conveniente a construção de uma árvore de sintaxe para as sentenças que são submetidas à análise, árvore esta que deve indicar as derivações necessárias à geração da sentença a partir da aplicação das produções da gramática original que define a linguagem, e não das versões gramaticais que tenham sido manipuladas para a construção do reconhecedor.

Este é o caso que ocorre quando se deseja utilizar a árvore de sintaxe como linguagem intermediária em compiladores multi-passos, ou em otimizadores independentes de máquina que operem por transformações sucessivas do texto-fonte ou de suas formas equivalentes.

Assim, ao se construir o analisador, é preciso que seja preservada a informação acerca da origem de cada uma de suas partes, em relação à gramática inicial a partir da qual se estiver aplicando o método.

Para tanto, deve-se enriquecer o reconhecedor, de tal forma que fiquem associadas às diversas transições do mesmo as informações acerca do correspondente comportamento da gramática original da linguagem.

Com este objetivo, criou-se uma extensão para o autômato, cujo papel é o de reter informações acerca da relação guardada entre as produções da gramática original e as diversas transições do autômato.

Isto é necessário uma vez que os diversos estados e transições construidos no autômato nem sempre são diretamente provenientes da gramática original, mas têm sua origem nas expressões gramaticais resultantes da manipulação daquela pelos algoritmos de geração do reconhecedor.

A ampliação em questão, associada à transformação do autômato em transdutor pela inclusão de um alfabeto de saída e de regras de geração de texto-objeto associadas a cada transição, introduz uma forma natural de associação de ações semânticas ao analisador sintático obtido, facilitando ao projetista a tarefa de incorporar ao transdutor da linguagem um tratamento semântico complementar.

3.3.1 Núcleos de Transdução Sintática

O processo mencionado pode ser formalizado através de um modelo teórico aplicável à representação de núcleos de transdução sintática para processadores de linguagens dirigidos por sintaxe.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Por uma questão de clareza preserva-se em evidência o *autômato de suporte* em seu formalismo original, acrescentando-se-lhe, para a obtenção do transdutor que deverá compor o núcleo em questão, um *alfabeto de saída*, um *alfabeto de pilha de tradução* e um conjunto de *regras de mapeamento*.

Um núcleo desta natureza assume pois a forma de uma quádrupla

$N = (R, \Lambda, \Pi, M)$

em que:

- **R** = reconhecedor básico (autômato de suporte de **N**)
- Λ = alfabeto de saída do núcleo transdutor
- Π = alfabeto da pilha do transdutor
- **M** = conjunto de regras de mapeamento $\mu_{i,j}$

onde, para cada produção $p_{i,j}$ do reconhecedor R , $\mu_{i,j}$ é uma regra que tem a função de provocar a execução de operações que se encontram associadas à aplicação de $p_{i,j}$ e são responsáveis pela geração de uma parcela do texto de saída, formado de uma cadeia de símbolos pertencentes ao alfabeto Λ de saída do transdutor.

$$M : \Pi^* \times \Lambda^* \rightarrow \Pi^* \times \Lambda^*$$

ou seja, **M** é uma aplicação que mapeia o produto cartesiano de Π^* por Λ^* neste mesmo produto cartesiano.

Assim, um dado transdutor particular exibirá um conjunto **M** formado por regras de mapeamento $\mu_{i,j}$ da forma

$$\mu_{i,j} : (\pi_{i,j}, \lambda_{i,j}) \rightarrow (\pi'_{i,j}, \lambda'_{i,j})$$

associadas às respectivas produções $p_{i,j}$ do autômato de suporte, sendo, portanto, $\pi_{i,j}$ e $\pi'_{i,j}$ cadeias de Π^* que representam, respectivamente, a situação da pilha do transdutor antes e depois da aplicação de $p_{i,j}$.

Analogamente, $\lambda_{i,j}$ e $\lambda'_{i,j}$ são cadeias de Λ^* , que representam a situação da cadeia de saída do transdutor antes e depois da aplicação de $p_{i,j}$, respectivamente.

A operação do núcleo assim definido pode ser interpretada como segue. Dada uma cadeia de entrada $\omega\perp$, com ω formada de símbolos do alfabeto de entrada do autômato de suporte, e com um símbolo $\perp$, não pertencente a tal alfabeto, indicativo de final do texto de entrada, pode-se definir:

- um *passo* de operação do transdutor, como sendo a aplicação conjunta de alguma produção $p_{i,j}$ do autômato de suporte e da correspondente regra de mapeamento $\mu_{i,j}$.
- *instante de transdução*, ou simplesmente instante, k , como sendo a ocasião imediatamente seguinte à execução do k-ésimo passo de operação do transdutor.
- instante inicial, correspondente àquele em que ainda nenhum passo de operação tenha ocorrido, sendo sempre adotado para isto, por convenção, o instante 0 (zero).
- uma numeração dos passos do transdutor, para uma cadeia de entrada $\omega\perp$, dada pela seqüencia dos números naturais $0, 1, 2, 3, \ldots$, correspondentes à seqüencia dos instantes do processo de transdução percorrida pelo transdutor para a cadeia $\omega\perp$.
- a *situação* t_k do transdutor ao k-ésimo passo de sua operação para a cadeia $\omega\perp$ como sendo a quíntupla

$$t_k = (\gamma_k, q_k, \omega_k\perp, \pi_k, \lambda_k)$$

onde:

- γ_k representa o conteúdo total da pilha do autômato de suporte no instante de transdução k (Observe-se que γ_0 representa o indicador Z_0 de pilha vazia)
- q_k é o estado exibido pelo autômato de suporte no instante k (Em particular, q_0 é o estado inicial do autômato de suporte)
- $\omega_k \perp$ é a parte da cadeia de entrada que ainda não foi tratada pelo transdutor no instante k (Notar que $\omega_0 = \omega$ é a própria cadeia de entrada inicialmente apresentada ao transdutor)
- π_k é o conteúdo da pilha de transdução no instante k (esta pilha está inicialmente vazia no instante $k = 0$, ou seja, $\pi_0 = \varepsilon$ na ocasião)
- λ_k é a parcela da cadeia de saída gerada até o instante k pelo processo de transdução (naturalmente a cadeia de saída é vazia no instante 0 , ou seja, $\lambda_0 = \varepsilon$ ao final do processamento)

- a *situação inicial* do transdutor, como sendo

$$t_0 = (Z_0, q_0, \omega\perp, \varepsilon, \varepsilon)$$

- uma *transição*, que corresponde ao movimento do transdutor entre a sua situação no instante k , e a situação seguinte, no instante $k+1$, resultante da aplicação do ($k+1$)-ésimo passo de transdução à cadeia $\omega\perp$. Denota-se tal transição como

$$t_k \Rightarrow t_{k+1}$$

- diz-se que o núcleo **N** *traduz* a cadeia de entrada ω para a cadeia de saída λ se e somente se houver alguma seqüencia

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$$t_0 \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow t_{n-1} \Rightarrow t_n$$

ou, abreviadamente,

$$t_0 \Rightarrow^* t_n$$

onde os t_k denotam situações do transdutor $(0 \leq k \leq n)$, e

$$t_n = (Z_0 , q_f , \perp , \varepsilon , \lambda)$$

onde q_f é algum dos estados finais do autômato de suporte.

- dá-se a t_n acima a denominação de situação final do transdutor **N** para a cadeia ω .
- diz-se que, na situação final, a cadeia de saída λ é a tradução da cadeia de entrada ω , realizada pelo núcleo **N** .

Adotando-se para o autômato de suporte o modelo dos autômatos de pilha estruturados, sabe-se que qualquer linguagem livre de contexto poderá ser por ele reconhecida, e, já que o transdutor é uma simples ampliação do modelo do reconhecedor, sem a imposição de qualquer alteração estrutural, conclui-se que esta característica é diretamente transferida ao transdutor.

3.3.2 Aplicação dos Núcleos à Geração de Analisadores Sintáticos

O formalismo acima permite as mais diversas utilizações do modelo apresentado, uma vez que nenhuma restrição foi imposta ao autômato nem às regras de mapeamento a serem empregadas.

Isto permite, entre inúmeras outras possíveis opções de aplicação, criar, a partir do modelo básico do núcleo de transdução sintática, um formalismo para a representação de analisadores sintáticos.

Com isto, é possível incorporar ao modelo adotado de reconhecedor sintático, baseado em autômatos de pilha estruturados, recursos capazes de reter as informações complementares acerca da gramática original que forem necessárias para convertê-lo em um analisador sintático.

A título de exemplo de aplicação, descreve-se a seguir uma possível forma através da qual se pode construir um dispositivo de geração da árvore sintática de uma cadeia de entrada a partir da seqüencia de transições executadas pelo autômato de suporte que a reconhece, com o auxílio de regras de mapeamento adequadas.

Para tanto, será aplicado, para a geração do reconhecedor básico do transdutor, um método similar ao descrito anteriormente, porém adaptado para tornar mais simples a obtenção dos efeitos específicos para este caso.

Pode-se observar que, aplicando-se este método modificado a uma gramática livre de contexto, criam-se autômatos de pilha estruturados cujas sub-máquinas, embora não se apresentem minimizadas nem isentas de transições em vazio, exibem uma topologia muito adequada às atividades dos transdutores desejados.

Como, para preservar pontos adequados à inclusão de informações sobre a gramática, não são tomadas providências no sentido de se eliminar potenciais estados equivalentes, resulta da aplicação deste método o surgimento de estados e de transições em vazio adicionais, em contraste com o que ocorre no processo de geração descrito no capítulo 3.2.

Entretanto, estes elementos aparentemente supérfluos são aqui mantidos por desempenharem no transdutor um papel de grande importância para a retenção de informações acerca da estrutura sintática formalizada pela gramática original.

A gramática livre de contexto a ser usada como base para a elaboração do analisador sintático deve ser preparada de forma análoga à que foi discutida anteriormente, com pequenas alterações de procedimento, e com o acréscimo de alguns mecanismos complementares necessários à sua operação.

Para isto devem ser associadas, às diversas produções que envolvam estados correspondentes aos pontos rotulados da expressão que define cada não-terminal, as regras de mapeamento encarregadas da geração da cadeia de saída e da manutenção da pilha do transdutor que implementará o analisador sintático desejado.

Durante a análise, a cada execução das produções às quais estejam associadas tais regras, as ações descritas pelas regras de mapeamento associadas devem ser efetuadas, gerando texto de saída e memorizando informações na pilha do transdutor, de tal forma que, ao final da análise, resulte a tradução esperada do texto de entrada como cadeia de saída.

No caso particular da geração de analisadores sintáticos, a ser discutida a seguir, a meta do transdutor será a de converter nas correspondentes árvores de derivação as sentenças da linguagem definida através da gramática livre de contexto a partir da qual o transdutor foi construido.

Por questão de facilidade de geração será adotado, para as árvores a serem geradas pelo analisador sintático a ser desenvolvido, o formato textual de lista definido no anexo D.

Para a obtenção do transdutor a partir da gramática pode ser utilizado o procedimento descrito a seguir.

Partindo-se da gramática livre de contexto fornecida, inicia-se agrupando, fatorando e marcando as expressões que definem os não-terminais, rotulando-os previamente e manipulando consistentemente os rótulos durante as suas transformações até a obtenção de uma expressão que possa ser transformada diretamente no transdutor sintático desejado.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

A meta é de, a cada ponto marcado da expressão final, associar-se, se necessário, um rótulo indicativo do início ou do término, naquele ponto, de alguma seqüencia de símbolos que constitua inequivocamente alguma construção sintática definida pelo não-terminal associado ao rótulo em questão.

Para iniciar o processo de rotulação, tomam-se as produções fornecidas na gramática original, e rotulam-se suas extremidades, indicando o início e o final da seqüencia sintática que aquela particular produção representa.

Os rótulos em questão devem encerrar as informações necessárias sobre a gramática (não-terminal associado, identificação da produção, tipo de produção etc.).

Com o auxílio de tais rótulos e das identificações dos estados designados aos diversos pontos marcados da expressão, constroem-se as produções que definem o autômato de suporte do núcleo transdutor.

Associam-se então, adequadamente, às transições que referenciam estados rotulados, as correspondentes regras de mapeamento, responsáveis pela memorização de informações e geração da cadeia de saída.

Isto posto, passa-se a descrever passo a passo um possível método para se efetuar a conversão de uma gramática livre de contexto em um transdutor sintático equivalente, baseado no modelo teórico descrito e no modelo dos autômatos de pilha adaptativos.

Este procedimento, que não restringe as gramáticas inicialmente fornecidas, pode ser utilizado inclusive com gramáticas ambíguas, obtendo-se neste caso um analisador determinístico que gera uma das possíveis árvores de derivação para as sentenças da linguagem definida pela gramática fornecida.

Preparação da Gramática para a Construção do Transdutor

1. Numerar arbitrariamente (por exemplo, seqüencialmente) as produções da gramática para identificá-las univocamente.
2. Para cada não-terminal **A** da gramática, agrupar separadamente todas as produções **p[i]** que definem cada uma das suas variantes sintáticas:
   a) auto-recursões à esquerda, do tipo
      $\mathbf{A} \rightarrow \mathbf{A}\ \alpha$
      onde α pode eventualmente ser da forma $\beta\ \mathbf{A}$
   b) auto-recursões à direita mas não simultaneamente à esquerda:
      $\mathbf{A} \rightarrow \alpha\ \mathbf{A}$
      onde α não é da forma $\mathbf{A}\ \beta$
   c) demais produções:
      $\mathbf{A} \rightarrow \alpha$
      onde α não é da forma $\mathbf{A}\ \beta$, nem da forma $\beta\ \mathbf{A}$, podendo entretanto ser da forma $\beta\ \mathbf{A}\ \beta'$, com β e β' não-vazios.
3. Sendo o lado direito de cada produção original uma cadeia de elementos μ_j do conjunto $\{\ \varepsilon\ \} \cup \mathbf{V}$, onde $\mathbf{V}$ é o vocabulário da gramática, sendo $n > 0$ e μ'_j vazio se μ_j for um não-terminal, e igual a μ_j em caso contrário, rotular os pontos extremos e intermediários de todas as produções de acordo com as seguintes convenções:

a) auto-recursões à esquerda:

$$\mathbf{A} \rightarrow \mathbf{A}\ \mu_1 \quad \mu_2 \quad \ldots \quad \mu_n$$

ficam:

$$[\qquad \mu'_1 \quad \mu'_2 \qquad \mu'_n\ \mathbf{Ai}\ ]$$
$$\mathbf{p[i]} : \mathbf{A} \rightarrow \uparrow \mathbf{A} \uparrow \mu_1 \uparrow \mu_2 \uparrow \ldots \uparrow \mu_n \uparrow$$

b) auto-recursões à direita mas não simultaneamente à esquerda:

$$\mathbf{A} \rightarrow \mu_1 \quad \mu_2 \quad \ldots \quad \mu_n \quad \mathbf{A}$$

ficam:

$$[\qquad \mu'_1 \quad \mu'_2 \qquad \mu'_n \qquad \underline{\mathbf{A}_i}\ ]$$
$$\mathbf{p[i]} : \mathbf{A} \rightarrow \uparrow \mu_1 \uparrow \mu_2 \uparrow \ldots \uparrow \mu_n \uparrow \mathbf{A} \uparrow$$

c) demais produções:

$$\mathbf{A} \rightarrow \mu_1 \quad \mu_2 \quad \ldots \quad \mu_n$$

ficam:

$$[\qquad \mu'_1 \quad \mu'_2 \qquad \mu'_n \qquad \underline{\underline{\mathbf{A}_i}}\ ]$$
$$\mathbf{p[i]} : \mathbf{A} \rightarrow \uparrow \mu_1 \uparrow \mu_2 \uparrow \ldots \uparrow \mu_n \uparrow \mathbf{A} \uparrow$$

Em particular, para o caso de $n = 1$, com $\mu_1 = \varepsilon$, tem-se:

$$[\quad \varepsilon\underline{\underline{\mathbf{A}_i}}\ ]$$
$$\mathbf{p[i]} : \mathbf{A} \rightarrow \uparrow \varepsilon \uparrow$$

5. Para cada grupo de produções do mesmo tipo que definem um dado não-terminal, agrupar entre parênteses (se $n > 1$) as expressões α definidas acima, rearranjando os seus rótulos. Abreviando-se

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$$\mu''_j = \overset{\mu'_1}{\uparrow} \mu_1 \overset{\mu'_2}{\uparrow} \mu_2 \uparrow \ldots \overset{\mu'_n}{\uparrow} \mu_n \uparrow$$

pode-se escrever informalmente a forma das expressões obtidas:

a) auto-recursões à esquerda:

$$\mathbf{A} \rightarrow \overset{[}{\uparrow} \mathbf{A} \; (\; \mu''_i \overset{A_i}{\uparrow} \mid \mu''_j \overset{A_j}{\uparrow} \mid \ldots \mid \mu''_k \overset{A_k}{\uparrow} \;) \overset{]}{\uparrow}$$

b) auto-recursões à direita:

$$\mathbf{A} \rightarrow \overset{[}{\uparrow} \; (\; \mu''_i \overset{\underline{A_i}}{\uparrow} \mid \mu''_j \overset{\underline{A_j}}{\uparrow} \mid \ldots \mid \mu''_k \overset{\underline{A_k}}{\uparrow} \;) \; \mathbf{A} \overset{]}{\uparrow}$$

c) demais produções:

$$\mathbf{A} \rightarrow \overset{[}{\uparrow} \; (\; \mu''_i \overset{\underline{\underline{A_i}}}{\uparrow} \mid \mu''_j \overset{\underline{\underline{A_j}}}{\uparrow} \mid \ldots \mid \mu''_k \overset{\underline{\underline{A_k}}}{\uparrow} \;) \overset{]}{\uparrow}$$

Cada não-terminal será, assim, definido por uma combinação adequada de uma, duas ou três expressões dos formatos acima.

6. Usando a fórmula geral já conhecida de eliminação de recursões à direita e à esquerda, pode-se fatorar em uma única expressão todas as opções válidas para o não-terminal $\mathbf{A}$:

$$\mathbf{A} \rightarrow \overset{[}{\uparrow} \; (\; \uparrow \varepsilon \uparrow \backslash \uparrow \beta''_1 \overset{\underline{A_1}}{\uparrow} \mid \uparrow \beta''_2 \overset{\underline{A_2}}{\uparrow} \mid \ldots \;)$$
$$\uparrow \; (\; \uparrow \alpha''_1 \overset{\underline{\underline{A_1}}}{\uparrow} \mid \uparrow \alpha''_2 \overset{\underline{\underline{A_2}}}{\uparrow} \mid \ldots \;)$$
$$\uparrow \; (\; \uparrow \varepsilon \uparrow \backslash \uparrow \gamma''_1 \overset{A_1}{\uparrow} \mid \uparrow \gamma''_2 \overset{A_2}{\uparrow} \mid \ldots \;) \overset{]}{\uparrow}$$

7. Eliminar não-determinismos causados pela ocorrência de prefixos iguais em duas ou mais opções contidas em algum dos agrupamentos de expressões resultante. Há os seguintes casos específicos a considerar:

a) Presença de prefixos comuns explícitos - resolve-se esta classe de conflitos colocando-se em evidência, em primeiro lugar, os prefixos mais longos possível que sejam comuns ao maior número possível de opções, e repetindo-se a operação enquanto em qualquer dos agrupamentos resultantes restar algum prefixo que possa ser posto em evidência. Dadas duas opções rotuladas, com prefixo comum $\alpha_1 \alpha_2 \ldots \alpha_n$:

$$\overset{R_0}{\uparrow} \alpha_1 \overset{R_1}{\uparrow} \alpha_2 \overset{R_2}{\uparrow} \ldots \overset{R_{n-1}}{\uparrow} \alpha_n \overset{R_n}{\uparrow} \beta_1 \overset{T_1}{\uparrow} \beta_2 \overset{T_2}{\uparrow} \ldots \overset{T_{p-1}}{\uparrow} \beta_p \overset{T_p}{\uparrow}$$

e

$$\overset{S_0}{\uparrow} \alpha_1 \overset{S_1}{\uparrow} \alpha_2 \overset{S_2}{\uparrow} \ldots \overset{S_{n-1}}{\uparrow} \alpha_n \overset{S_n}{\uparrow} \gamma_1 \overset{U_1}{\uparrow} \gamma_2 \overset{U_2}{\uparrow} \ldots \overset{U_{q-1}}{\uparrow} \gamma_q \overset{U_q}{\uparrow}$$

Nesta expressão, tem-se

$$R_0 \; R_1 \ldots R_m = S_0 \; S_1 \ldots S_m \;, \quad 0 < m \leq n$$

obtendo-se, então, a expressão fatorada seguinte:

$$\overset{R_0}{\uparrow} \alpha_1 \overset{R_1}{\uparrow} \alpha_2 \overset{R_2}{\uparrow} \ldots \overset{R_{m-1}}{\uparrow} \alpha_m \overset{R_m}{\uparrow} \alpha_{m+1} \uparrow \ldots \uparrow \alpha_n \uparrow$$
$$(\; \uparrow \beta_1 \overset{R_{m+1} \ldots R_n \; T_1}{\uparrow} \beta_2 \overset{T_2}{\uparrow} \ldots \overset{T_{p-1}}{\uparrow} \beta_p \overset{T_p}{\uparrow} \mid$$
$$\uparrow \gamma_1 \overset{R_{m+1} \ldots R_n \; U_1}{\uparrow} \gamma_2 \overset{U_2}{\uparrow} \ldots \overset{U_{q-1}}{\uparrow} \gamma_q \overset{U_q}{\uparrow} \;) \uparrow$$

Esta expressão deve ser reanalisada quanto a não-determinismos remanescentes no agrupamento criado.

b) Presença de um não-terminal na extremidade esquerda de uma das opções - neste caso, substitui-se o não-terminal em questão pela expressão rotulada que define o não-terminal, posta entre parênteses, e analisando-se a expressão obtida.

Sendo a opção em questão da forma

$$\overset{R}{\uparrow} \mathbf{N} \overset{R_0}{\uparrow} \mu_1 \overset{R_1}{\uparrow} \mu_2 \overset{R_2}{\uparrow} \ldots \overset{R_{n-1}}{\uparrow} \mu_n \overset{R_n}{\uparrow}$$

e sendo $\mathbf{N}$ dado por uma expressão rotulada δ , que seja um caso particular qualquer da forma geral apresentada no item 6, obtém-se a seguinte expressão:

$$\overset{R}{\uparrow} \; (\; \delta \;) \overset{R_0}{\uparrow} \mu_1 \overset{R_1}{\uparrow} \mu_2 \overset{R_2}{\uparrow} \ldots \overset{R_{n-1}}{\uparrow} \mu_n \overset{R_n}{\uparrow}$$

a qual deverá ser novamente analisada quanto à presença de manifestações não-determinísticas.

c) Presença explícita de (cadeia vazia) como uma das opções válidas do agrupamento - Seja uma expressão do tipo

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$$\begin{array}{cccccccc} R_0 & R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 \\ \uparrow\ \alpha & \uparrow\ (& \uparrow\ \varepsilon & \uparrow\ \backslash & \uparrow\ \mu & \uparrow\ ) & \uparrow\ \beta & \uparrow \end{array}$$

com α , β , μ arbitrários, eventualmente vazios, sendo α e β fatores.
Para levantar as cadeias geradas por este tipo de expressão, pode-se transformá-la em uma expressão equivalente

$$\begin{array}{cccccccc} R_0 & & R_1R_2R_3R_6 & R_7 & R_0 & R_1R_4 & R_5R_6 & R_7 \\ \uparrow & \alpha & \uparrow\ \beta & \uparrow\ | & \uparrow\ \alpha & \uparrow\ \mu & \uparrow\ \beta & \uparrow \end{array}$$

cujos dois termos poderão ser manipulados juntamente com outras opções do mesmo nível para efeito de eliminação de situações não-determinísticas.

d) Presença de uma construção cíclica na extremidade esquerda de uma das opções - Este tipo de construção dá origem implicitamente à cadeia vazia, encobrindo eventuais prefixos comuns às cadeias geradas pelo fator do qual este ciclo é prefixo. Para explicitar estas situações, pode-se substituir o fator cíclico

$$\begin{array}{cccccc} R_0 & R_1 & R_2 & R_3 & R_4 & R_5 \\ \uparrow\ (& \uparrow\ \varepsilon & \uparrow\ \backslash & \uparrow\ \alpha & \uparrow\ ) & \uparrow \end{array}$$

pela expressão rotulada equivalente

$$\begin{array}{cccccccccc} R_0R_1 & R_2R_5 & R_0R_1R_2R_3 & R_4 & & R_3 & R_4 & R_5 \\ \uparrow\ \varepsilon & \uparrow\ | & \uparrow\ \alpha & \uparrow\ (& \uparrow\ \varepsilon & \uparrow\ \backslash & \uparrow\ \alpha & \uparrow\ ) & \uparrow \end{array}$$

recaindo-se no caso c) anterior, com ε explícito.

8. Continuar o trabalho de eliminação de não-determinismos descrito no item 7, até que não mais restem situações não-determinísticas, ou até que, na tentativa de eliminação de uma delas, se recaia em alguma situação tratada previamente. A reincidência de uma configuração já considerada identifica a presença de um não-determinismo não-eliminável. Neste caso, repetir o processo o número de vezes julgado satisfatório para que se garanta o funcionamento determinístico do autômato em todos os casos em que isto seja essencial. Para os demais casos o autômato operará não-deterministicamente.

■

Uma vez obtida e devidamente manipulada uma expressão rotulada única para cada não-terminal necessário, passa-se a gerar, de acordo com o procedimento seguinte, as produções do autômato correspondente.

Tomar o cuidado de, ao aplicar as regras abaixo para a geração das produções, criar em paralelo as regras de mapeamento do transdutor, conforme o roteiro apresentado mais adiante.

Construção do Autômato de Suporte do Transdutor

1. Designar o mesmo estado a todos os pontos marcados à esquerda de cada opção de um dado agrupamento de expressões alternativas. Designar assim estados diferentes para cada um destes agrupamentos das expressões. Completar a designação de estados do autômato associando a cada ponto marcado restante um estado diferente. Passa-se então a construir as produções do autômato aplicando as seguintes regras (abstrai-se aqui a rotulação das expressões):
2. Para toda ocorrência de um meta-símbolo ε :

$$\begin{array}{ccc} \uparrow & \varepsilon & \uparrow \\ \mathbf{x} & & \mathbf{y} \end{array}$$

criar uma transição em vazio de **x** para **y** :

$$\gamma\ \mathbf{x}\ \alpha \ \rightarrow\ \gamma\ \mathbf{y}\ \alpha$$

3. Para toda ocorrência de terminais **t** na expressão:

$$\begin{array}{ccc} \uparrow & \mathbf{t} & \uparrow \\ \mathbf{x} & & \mathbf{y} \end{array}$$

construir uma transição interna entre os estados **x** e **y**, consumindo o terminal **t**:

$$\gamma\ \mathbf{x}\ \mathbf{t}\ \alpha \ \rightarrow\ \gamma\ \mathbf{y}\ \alpha$$

4. Para toda ocorrência de um não-terminal **N** na expressão:

$$\begin{array}{ccc} \uparrow & \mathbf{N} & \uparrow \\ \mathbf{x} & & \mathbf{y} \end{array}$$

criar uma transição em vazio partindo do estado **x** para o estado inicial $\mathbf{n_0}$ da sub-máquina associada ao não-terminal **N**, salvando o estado **y** na pilha do autômato, para ser utilizado pela sub-máquina chamada quando da execução de sua transição de retorno:

$$\gamma\ \mathbf{x}\ \alpha \ \rightarrow\ \gamma\ \mathbf{y}\ \mathbf{n_0}\ \alpha$$

5. Para cada agrupamento cíclico:

$$\begin{array}{cccccc} \uparrow\ (& \uparrow\ \varepsilon & \uparrow & \uparrow\ \mu & \uparrow\ ) & \uparrow \\ \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{t} & \mathbf{u} & \mathbf{v} \end{array}$$

criar as produções seguintes, que implementam o controle da repetição denotada por este agrupamento:

$$\gamma\ \mathbf{x}\ \alpha \ \rightarrow\ \gamma\ \mathbf{y}\ \alpha$$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$$\gamma\ z\ \alpha \rightarrow \gamma\ v\ \alpha$$
$$\gamma\ z\ \alpha \rightarrow \gamma\ t\ \alpha$$
$$\gamma\ t\ \alpha \rightarrow \gamma\ v\ \alpha$$
$$\gamma\ u\ \alpha \rightarrow \gamma\ t\ \alpha$$

6. Para cada estado n_f , associado ao final de alguma das opções que definem um não-terminal **N** , criar uma transição em vazio que execute o retorno ao estado **y** da sub-máquina chamadora, previamente empilhado pela produção de chamada da sub-máquina associada ao não-terminal **N** :

$$\gamma\ y\ n_f\ \alpha \rightarrow \gamma\ y\ \alpha$$

7. Interpretar como sendo o estado inicial da sub-máquina associada ao não-terminal **N** o estado correspondente à extremidade esquerda das expressões que definem este não-terminal.
8. Os estados inicial e final da sub-máquina associada ao não-terminal que representa a raiz da gramática serão os estados inicial e final do autômato, respectivamente.

■

Construção das Regras de Mapeamento

Para a construção das regras de mapeamento, aplica-se a tabela abaixo a cada uma das produções do autômato, usando sempre o rótulo associado ao estado-destino da produção.

Cada rótulo corresponde a uma cadeia de comprimento maior ou igual a zero de rótulos elementares, a serem considerados seqüencialmente da esquerda para a direita.

Notar que rótulos elementares relativos a auto-recursões à esquerda de um não-terminal **A** são denotados por $\underline{A}$; auto-recursões à direita, por $\underline{\underline{A}}$ e os associados às demais produções, por **A** .

Movimentos na pilha de transdução são indicados pelos operadores prefixos ↓ (indicando empilhamento) e (indicando desempilhamento), atuando sobre uma cadeia denotada à sua direita.

A presença do meta-símbolo π simboliza o conteúdo da parte da pilha de transdução compreendida entre o seu topo e a primeira ocorrência do delimitador **]** , exclusive; terminais e não-terminais são respectivamente simbolizados por σ e **A** , e a cadeia vazia, por ε ; índices **i** em um não-terminal indicam uma referência à i-ésima produção que o define na gramática original.

Rótulo	Saída	Pilha
ε	()	
σ	(σ)	
A_i	$A_i)$	
$\underline{\underline{A_i}}$	$A_i)\pi$	$\uparrow\pi$
$\underline{A_i}$	(	$\downarrow)A_i$
[	[(	↓]
]	π]	$\uparrow\pi$]

A ocorrência dupla de π , tanto na cadeia de saída como na de pilha, em uma linha da tabela acima, indica que naquele caso uma cópia do conteúdo π da pilha deve ser emitido como saída.

Aplicando-se a tabela acima a uma produção que apresente

$$R = r_1\ r_2\ \ldots\ r_n$$

como rótulo associado ao seu estado-destino, para cada r_i encontram-se, na linha correspondente da tabela, as indicações da alteração da saída (s_i) e da pilha de transdução (p_i) .

Ao rótulo **R** completo fica assim associada uma alteração **S** da cadeia de saída, dada pela concatenação ordenada dos s_i , e uma alteração da pilha de transdução, P_i , obtida também pela concatenação dos correspondentes p_i .

Obtém-se assim a regra de mapeamento desejada:

$$(\ \gamma\ ,\ \lambda\ ) \rightarrow (\ \gamma P\ ,\ \lambda S\ )$$

Interpretam-se λ e γ como sendo a situação da cadeia de saída e da pilha do transdutor, respectivamente, imediatamente antes da aplicação da regra de mapeamento.

Após sua aplicação, a situação se torna λS e γP , respectivamente, indicando que às cadeias λ e γ foram concatenados à direita as cadeias **S** e **P** , respectivamente.

Construídas as produções e as correspondentes regras de mapeamento, pode-se dizer que o projeto do transdutor está terminado, ficando ele pronto para ser aplicado a alguma particular gramática, para a obtenção do seu correspondente analisador sintático, o que está ilustrado a seguir.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

3.3.3 Exemplos Ilustrativos do Uso do Transdutor

Ilustra-se a utilização do método desenvolvido anteriormente através do exercício do gerador de analisadores sintáticos que acaba de ser construído, aplicando-o a três pequenas gramáticas, portadoras de diversas peculiaridades, críticas para o método.

Para não delongar desnecessariamente este texto, os dois primeiros exemplos não serão desenvolvidos completamente em todos os detalhes, enfatizando-se apenas as suas partes mais relevantes.

Exemplo 1

Apresenta-se a seguir a obtenção de um analisador sintático para uma linguagem definida pela gramática livre de contexto de raiz **X** , e formalizada pelas seguintes produções:

1. X → a	3. X → Y Z	5. Y → ε
2. X → Y b	4. Y → a	6. Z → b c

Esta gramática define, de uma forma um tanto exótica, o conjunto finito de sentenças seguinte: **{ a, b, ab, bc, abc }** , e foi escolhida para exercitar a manipulação de rótulos em caso de não-determinismos não-explícitos.

Rotulando individualmente as produções, e agrupando-as em seguida em expressões únicas para cada não-terminal tem-se:

```
         [         aX1               bX2        X3        ]
X → ↑ ( ↑ a ↑  | ↑ Y ↑ ( ↑ b ↑  | ↑ Z  ↑  ) ↑ ) ↑
                     [        aY4      εY5   ]
           Y →  ↑ ( ↑ a ↑  | ↑ ε ↑   ) ↑
                          [   b    cZ6 ]
                 Z →  ↑ b ↑  c ↑
```

Efetuando a substituição das expressões que definem **Y** e **Z** na definição de **X** obtém-se

```
     [         aX1        [         aY4      εY5   ]
X →  ↑ ( ↑ a ↑  | ↑ (  ↑ ( ↑ a ↑  | ↑ ε ↑  ) ↑ ) ↑
                              bX2        [   b    cZ6]      X3        ]
                      ( ↑ b ↑  | ↑ (  ↑ b ↑  c ↑       ) ↑  ) ↑ ) ↑
```

Eliminando parênteses nas substituições realizadas:

```
     [         aX1        [     aY4]  [    εY5]
X →  ↑ ( ↑ a ↑  | ↑ (  ↑  a ↑   | ↑ ε ↑    ) ↑
                              bX2      [   b    cZ6]X3         ]
                      ( ↑ b ↑   |   ↑ b ↑  c ↑      ) ↑ ) ↑
```

Notar o surgimento de um prefixo **b** comum às duas opções do último agrupamento. Colocando-o em evidência tem-se:

```
    [         aX1        [     aY4]  [    εY5]
X → ↑ ( ↑ a ↑  | ↑ (  ↑  a ↑   | ↑ ε ↑    ) ↑
                             bX2           [bcZ6]X3          ]
              ( ↑ b ↑ ( ↑ ε ↑  |  ↑  c   ↑          ) ↑ ) ↑
```

Defatorando-se mais uma vez a expressão, por meio da aplicação da propriedade distributiva, chega-se a

```
    [         aX1
X → ↑ ( ↑ a ↑  |
    [   aY4]          bX2          [bcZ6]X3
    ↑  a ↑    b ↑ ( ↑ ε ↑  | ↑   c   ↑         ) ↑ |
       [εY5]        bX2           [bcZ6]X3          ]
    ↑ b ↑     ( ↑ ε ↑  |  ↑  c   ↑         ) ↑  ) ↑
```

Colocando em evidência o prefixo comum, **a** , entre as duas primeiras opções do agrupamento, observar a migração dos rótulos para o interior do agrupamento resultante como conseqüencia desta operação, já que eles diferem nas diversas opções envolvidas:

```
     [                   aX1        [aY4]        bX2        [bcZ6]X3
X → ↑ ( ↑ a ↑ ( ↑ ε ↑  | ↑ b ↑    ( ↑ ε ↑  | ↑ c ↑          ) ↑ ) ↑ |
       [εY5]        bX2           [bcZ6]X3          ]
    ↑ b ↑     ( ↑ ε ↑  |  ↑  c   ↑         )  ↑  ) ↑
```

Nesta expressão não restam conflitos entre prefixos, permitindo que ela seja utilizada para a montagem do autômato determinístico de suporte do transdutor, bem como das regras de mapeamento correspondentes (omitidas neste texto).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Aplicando diretamente a expressão acima às cinco sentenças da linguagem, e interpretando a seqüencia de rótulos correspondente aos estados percorridos conforme foi convencionado anteriormente, obtêm-se as traduções seguintes:

Sentença	Seq. de rótulos	Tradução
a	$[a\underline{\underline{X}}_1]$	$[((a)X_1)]$
b	$[[\varepsilon\underline{\underline{Y}}_5]b\underline{\underline{X}}_2]$	$[([(()Y_5)](b)X_2)]$
ab	$[[a\underline{\underline{Y}}_4]b\underline{\underline{X}}_2]$	$[([((a)Y_4)](b)X_2)]$
bc	$[[\varepsilon\underline{\underline{Y}}_5][bc\underline{\underline{Z}}_6]\underline{\underline{X}}_3]$	$[([(()Y_5)][(b)(c)Z_6)]X_3)]$
abc	$[[a\underline{\underline{Y}}_4][bc\underline{\underline{Z}}_6]\underline{\underline{X}}_3]$	$[([((a)Y_4)][(b)(c)Z_6)]X_3)]$

Interpretando estas cadeias de saída como representações de árvores, de acordo com a convenção inicialmente estabelecida, constata-se sem dificuldade tratarem-se de fato das árvores de derivação das sentenças correspondentes.

Exemplo 2

Este exemplo ilustra a aplicação do método a gramáticas que exibem não-determinismos advindos de construções cíclicas, prefixos comuns e cadeias vazias explícitas.

Seja a gramática de raiz **X** , definida pelas seguintes produções:

1. $X \rightarrow Y\ b$	3. $Y \rightarrow a\ Z$	4. $Z \rightarrow \varepsilon$
2. $X \rightarrow a\ b\ c$		5. $Z \rightarrow Z\ a$

Rotulando, agrupando e eliminando auto-recursões resulta:

```
          [            bX1      a   b   cX2   ]
    X →  ↑  ( ↑ Y ↑ b ↑   | ↑ a ↑ b ↑ c ↑   ) ↑
                        [   a      Y3]
                  Y →  ↑ a ↑  Z  ↑
          [      εZ4                   aZ5   ]
    Z → ↑ ( ↑ ε ↑  ) ↑ ( ↑ ε ↑ \ ↑ a ↑  ) ↑
```

Substituindo **Z** em **Y**:

```
       [   a   [       εZ4                  aZ5  ]    Y3]
  Y →  ↑ a ↑  ( ↑ ( ↑ ε ↑  ) ↑ ( ↑ ε ↑ \ ↑ a ↑  ) ↑  ) ↑
```

Desenvolvendo uma vez o ciclo mais à direita (notar o surgimento do rótulo **aZ5** fora do ciclo) vem:

```
    [   a   [      εZ4            aZ5           aZ5   ]       Y3]
Y → ↑  a ↑  ( ↑ ( ↑ ε ↑  )↑(↑ ε ↑|↑ a ↑  (↑ ε ↑ \ ↑ a ↑  ) ↑ ) ↑) ↑
```

Eliminando parênteses (observar a migração dos rótulos):

```
    [   a        [εZ4]   [εZ4   aZ5            aZ5   ]     Y3]
Y → ↑ a ↑ ( ↑  ε   ↑    |  ↑    a ↑  ( ↑ ε ↑ \ ↑  a ↑  ) ↑   ) ↑
```

Defatorando mais uma vez:

```
    [   a[εZ4 ]Y3 ]  [   a[εZ4   aZ5              aZ5  ]Y3 ]
Y → ↑ a ↑          | ↑ a ↑     a ↑  ( ↑ ε ↑ \ ↑ a ↑  ) ↑
```

Colocando **a** em evidência (observar a migração da parte comum do seu rótulo para fora do agrupamento resultante):

```
    [    a[εZ4       ]Y3]      aZ5              aZ5   ]Y3]
Y → ↑ a  ↑     ( ↑ ε ↑  | ↑ a ↑  ( ↑ ε ↑ \ ↑ a ↑   ) ↑   ) ↑
```

Substituindo na definição de **X** :

```
    [   [   a[εZ4        ]Y3]      aZ5             aZ5  ]Y3]      bX1
X → ↑ ( ↑ a ↑     ( ↑ ε ↑  | ↑ a ↑  ( ↑ ε ↑ \ ↑ a ↑  ) ↑   ) ↑ b ↑    |
          a   b   cX2   ]
     ↑ a ↑ b ↑ c ↑   ) ↑
```

Defatorando (notar a fusão de rótulos adjacentes não-vazios):

```
       [   [   a[εZ4]Y3]   bX1
   X → ↑ ( ↑ a ↑          b ↑     |
          [   a[εZ4    aZ5              aZ5   ]Y3]      bX1
          ↑ a ↑     a ↑  ( ↑ ε ↑ \ ↑ a ↑  ) ↑        b ↑     |
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
                         a    b    cX2     ]
                     ↑  a ↑ b ↑  c ↑     ) ↑
```

Fatorando o prefixo comum a surgido (aqui também ocorre a fusão de rótulos na ocasião da concatenação de expressões) vem:

```
          [                  [a[εZ4]Y3]bX1
   X →  ↑  ( ↑ a ↑ ( ↑ b ↑                      |
                             [a[εZ4 aZ5              aZ5   ]Y3]    bX1
                        ↑  a ↑         ( ↑ ε ↑ \ ↑ a ↑   ) ↑    b ↑      |
                             ab    cX2    ]
                        ↑  b ↑   c ↑    ) ↑
```

Fatorando **b** , resulta finalmente, pronta para ser usada:

```
          [                  [a[εZ4aZ5               aZ5   ]Y3]    bX1
   X →  ↑  ( ↑ a ↑ ( ↑ a ↑       ( ↑ ε ↑ \ ↑ a ↑   ) ↑    b ↑      |
                                 [a[εZ4 ]Y3 ]bX1         abcX2              ]
                 ↑ b ↑ ( ↑ ε  ↑                 |  ↑ c ↑      ) ↑  ) ↑  ) ↑
```

Exemplo 3

Neste exemplo é apresentado um projeto completo, com acompanhamento passo a passo para ilustrar a parte dinâmica da operação do transdutor.

A gramática escolhida, de raiz **S** , foi definida através do conjunto abaixo de produções, de forma ambígua e apresentando simultaneamente produções recursivas de todos os tipos estudados:

1. S → E	2. E → E * E	5. F → < E >
	3. E → E + F	6. F → a F
	4. E → ε	7. F → b

Marcando e rotulando individualmente as produções tem-se

```
     [   S1]             [      *   E2]             [   <        >F5]
  S → ↑ E ↑          E →↑ E ↑  * ↑ E ↑          F →↑ < ↑ E ↑ > ↑
                         [      +   E3]             [   a   F6]
                     E →↑ E ↑ + ↑ F ↑           F → ↑ a ↑  F ↑
                         [    εE4]                  [   bF7]
                     E →↑ ε  ↑                  F →↑ b ↑
```

Como a gramática já se encontra com as produções agrupadas e numeradas, pode-se passar a fatorar os não-terminais auto-recursivos **E** e **F** .

Eliminando a auto-recursão em **E** à esquerda das produções **2** e **3**, e fatorando as expressões rotuladas resulta:

```
            [          εE4                      *    E2        +    E3    ]
     E →  ↑ ( ↑ ε ↑    ) ↑ ( ↑ ε ↑ \ ↑ * ↑ E ↑   | ↑ + ↑ F ↑   ) ↑
```

Eliminando agora a auto-recursão em **F** à direita da produção, e fatorando adequadamente as expressões tem-se:

```
          [                 aF6          <         >F5          bF7    ]
   F →  ↑ ( ↑ ε ↑ \ ↑ a ↑    ) ↑  ( ↑ < ↑ E ↑ > ↑      | ↑ b ↑   ) ↑
```

Substituindo-se esta expressão na anterior vem:

```
          [         εE4                       *    E2
   E →  ↑  ( ↑  ε ↑   ) ↑ ( ↑ ε ↑ \ ↑  * ↑ E ↑   | ↑
        +   [                 aF6          <       >F5          bF7     ]      E3     ]
    ↑ + ↑  ( ↑ ( ↑ ε ↑ \ ↑  a ↑    ) ↑ ( ↑ < ↑ E ↑ > ↑     | ↑ b ↑    ) ↑ )  ↑    ) ↑
```

A gramática manipulada resultante consta portanto da expressão acima, definindo o não-terminal **E** (auto-recursivo central, essencial) e da expressão

```
      [      S1]
  S → ↑   E   ↑
```

que define o não-terminal **S** (raiz da gramática).

O não-terminal **S** representa a raiz da gramática, e a ele corresponderá portanto a submáquina principal do autômato.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

A construção das produções desta submáquina é trivial: designando estados 0' e 1' , respectivamente, aos pontos extremos esquerdo e direito da expressão acima tem-se

```
      [     S1]
S →   ↑  E  ↑
      0'    1'
```

Admitindo que o estado inicial da submáquina que implementa **E** seja o estado 0 , as produções da submáquina serão:

$$\gamma \ 0' \ \alpha \rightarrow \gamma \ 1' \ 0 \ \alpha$$

$$\gamma \ 1' \ \alpha \rightarrow \gamma \ 1' \ \alpha$$

a segunda produção ocorre no estado final do autômato, e sua execução deve encerrar o processamento.

Para implementar a submáquina correspondente ao não-terminal **E** , inicia-se designando o estado **0** ao início da expressão, por compatibilidade com a hipótese acima. Atribuindo estados iguais aos pontos extremos esquerdos das opções dos agrupamentos tem-se:

```
     [E     εE4                  *   E2
E →  ↑ ( ↑  ε ↑  ) ↑ ( ↑ ε ↑ \ ↑ * ↑ E ↑  | ↑
     0   1           2         3
     +   [             aF6          <      >F5       bF7    ]    E3    ]
 ↑ + ↑ ( ↑ ( ↑ ε ↑ \ ↑ a ↑  ) ↑ ( ↑ < ↑ E ↑ > ↑   | ↑ b ↑  ) ↑ )  ↑  ) ↑
 3         4     5              6                 6
```

Numerando os estados restantes a expressão fica:

```
     [E     εE4                  *   E2
E →  ↑ ( ↑  ε ↑  ) ↑ ( ↑ ε ↑ \ ↑ * ↑ E ↑  | ↑
     0   1  7    8   2   9   3   10  11
     +   [             aF6          <      >F5       bF7    ]    E3    ]
↑ + ↑ ( ↑ ( ↑ ε ↑ \ ↑ a ↑  ) ↑ ( ↑ < ↑ E ↑ > ↑   | ↑ b ↑  ) ↑ )  ↑  ) ↑
3  12  13  4  14  5   15   16  6  17  18  19   6   20   21   22   23
```

Aplicando as técnicas estudadas, geram-se as produções e as regras de mapeamento correspondentes, listadas na tabela abaixo.

Nesta tabela, ao invés de se denotar algebricamente as produções e regras de mapeamento, apresentam-se apenas os seus elementos: os estados origem (**e**) e destino (**s**) da produção; o tipo de produção σ : se ε , em vazio; se não-terminal, chamada de submáquina; se átomo, consumo do átomo; adicionalmente, o rótulo associado, o movimento da pilha e a cadeia de saída gerada.

e	σ	s	rótulos	pilha	saída
0	ε	1	[	↓]	[(
8	ε	2			
9	ε	3			
11	ε	3			
22	ε	3			
13	ε	4			
14	ε	5			
15	ε	5			
16	ε	6			
1	ε	7	$\varepsilon \underline{E_4}$	↑π	$()E_4)\pi$
7	ε	8			
2	ε	9			
3	*	10	*		(*)
10	E	11	E_2		$E_2)$
3	+	12	+		(+)
12	ε	13	[	↓]	[(
4	ε	14			
5	a	15	$a\underline{F_6}$	$\downarrow)F_6$	(a)(
5	ε	16			
6	<	17	<		(<)
17	E	18			
18	>	19	$>\underline{F_5}$	↑π	$(>)F_5)\pi$

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

6	b	20	b$\underline{\underline{F_7}}$	↑π	(b)F_7)π
19	ε	21	]	↑π]	π]
20	ε	21	]	↑π]	π]
21	ε	22	E_3		E_3)
3	ε	23	]	↑π]	π]

Tabela das transições e das regras de mapeamento associadas. A coluna de rótulos indica os rótulos da expressão original associados aos estados-destino da produção correspondente.

Completado o projeto do transdutor, pode-se exercitá-lo com uma cadeia de entrada tal como:

*** * + b + a a b + < > * + < * + a b >**

Aplicando o conjunto de produções e regras de mapeamento acima, pode-se acompanhar a evolução do transdutor através da trajetória de execução esquematizada na lista abaixo.

Para abreviar esta lista, não é indicado todo o caminho percorrido, mas apenas os efeitos provocadas pela análise de cada símbolo de entrada: partindo do estado inicial, o transdutor vai executando uma seqüencia de transições até consumir o símbolo de entrada indicado.

Neste percurso, vão sendo visitados estados rotulados, e as cadeias formadas pela concatenação seqüencial dos rótulos coletados entre os consumos de símbolos adjacentes são indicados na tabela ao lado do símbolo consumido na última transição, juntamente com a cadeia de saída correspondente à seqüencia de rótulos coletada.

Convém lembrar que os rótulos elementares são interpretados na seqüencia em que forem sendo encontrados, acarretando na ocasião a execução das regras de mapeamento associadas.

Os conteúdos da pilha de transdução e da pilha de estados de retorno são tembém indicados, com a finalidade de facilitar ao leitor o acompanhamento da operação do transdutor.

A saída gerada pelo transdutor a partir da cadeia de entrada apresentada acima, devidamente diagramada para realçar os aninhamentos das diversas partes da árvore de derivação que representa, é a seguinte:

[(() E_4)
(*)
[(() E_4)
(*)
[(() E_4)
(+)
[((b) F_7)] E_3)
(+)
[((a) ((a) ((b) F_7) F_6) F_6)] E_3)
(+)
[((<) [(() E_4)] (>) F_5)] E_3)
(*)
[(() E_4)
(+)
[((<)
[(() E_4)
(*)
[(() E_4)
(+)
[((a) ((b) F_7) F_6)] E_3)
] E_2)
]
(>) F_5)
] E_3)
] E_2)
] E_2)
] E_2)
]

A obtenção desta cadeia de saída a partir do texto de entrada está representada na tabela abaixo.

Símbolo	Rótulos	Saídas	Pilha	Pilha de Retorno
*	[ε$\underline{\underline{E_4}}$*	[(()E_4)(*)	]	Z_0

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

				Z_0,11
*	[ε$\underline{E}_4$*	[(()E_4)(*)	]]	
				Z_0,11,11
+	[ε$\underline{E}_4$+	[(()E_4)(+)	]]]	
b	[b$\underline{F}_7$	[((b)F_7)	]]]]	
+	]E_3+	]E_3)(+)	]]]	
a	[a$\underline{F}_6$	[((a)(	]]]])F_6	
a	a$\underline{F}_6$	(a)(	]]]])F_6)F_6	
b	b$\underline{F}_7$	(b)F_7)F_6)F_6)	]]]]	
+	]E_3+	]E_3)(+)	]]]	
<	[<	[((<)	]]]]	
				Z_0,11,11,18
ε	[ε$\underline{E}_4$]	[(()E_4)]	]]]]	
				Z_0,11,11
>	>$\underline{F}_5$	(>)F_5)	]]	
*	]E_3*	]E_3)(*)	]]]	
				Z_0,11,11,11
+	[ε$\underline{E}_4$+	[(()E_4)(+)	]]]]	
<	[<	[((>)	]]]]]	
				Z_0,11,11,11,18
*	[ε$\underline{E}_4$*	[(()E_4)(*)	]]]]]]	
				Z_0,11,11,11,18,11
+	[ε$\underline{E}_4$+	[(()E_4)(+)	]]]]]]]	
a	[a$\underline{F}_6$	[((a)(	]]]]]]]])F_6	
b	b$\underline{F}_7$	(b)F_7)F_6)	]]]]]]]]	
ε	]E_3]	]E_3)]	]]]]]]	Z_0,11,11,11,18
ε	E_2]	E_2)]	]]]]]	Z_0,11,11,11
>	>$\underline{F}_5$	(>)F_5)	]]]]]	
ε	]E_3]	]E_3)]	]]]	
ε	E_2]	E_2)]	]]	Z_0,11,11
ε	E_2]	E_2)]	]	Z_0,11
ε	E_2]	E_2)]		Z_0

Acompanhamento do tratamento da cadeia do exemplo pelo transdutor

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

4 AUTÔMATOS ADAPTATIVOS

Nesta seção são discutidos os princípios dos autômatos adaptativos e da sua operação, com a finalidade de apresentar uma forma como estes aceitadores podem ser empregados no reconhecimento de linguagens dependentes de contexto.

Um dos principais resultados deste estudo refere-se ao fato de que o emprego do conceito do autômato adaptativo como reconhecedor para tal classe de linguagens permite a obtenção de aceitadores muito eficientes, baseados em máquinas de estados finitos, pilhas de controle para o tratamento de formas sintáticas livres de contexto e mecanismos de ampliação para o tratamento de dependências de contexto.

Embora incorpore estes três mecanismos de reconhecimento e não imponha restrições ao seu uso, o modelo proposto para o reconhecedor não força a utilização compulsória de nenhum deles, a não ser que a particular sentença a ser reconhecida exiba, de fato, complexidade tal que justifique sua utilização.

Desta maneira, através do emprego de aceitadores baseados neste modelo, linguagens regulares poderão ser reconhecidas exclusivamente através de mecanismos similares a máquinas de estados finitos, enquanto linguagens livres de contexto utilizarão, no caso geral, conjuntos de sub-máquinas que se comportam cada qual de maneira similar a um autômato finito, podendo ser projetadas de tal forma que lancem mão de uma pilha de controle exclusivamente nas ocasiões em que forem detectados aninhamentos sintáticos na particular sentença que estiver sendo reconhecida.

Linguagens dependentes de contexto, no caso geral, utilizarão os três mecanismos de reconhecimento, sendo que, para as sentenças mais simples, que não apresentem sequer aninhamentos sintáticos, o reconhecimento será processado exclusivamente através da operação como autômatos finitos, enquanto sentenças com aninhamentos sintáticos exigirão o uso da pilha de controle, e aquelas que contiverem construções autenticamente dependentes de contexto, como ocorre no caso das declarações de variáveis e sua posterior utilização, promoverão a ativação de transições adaptativas de ampliação do reconhecedor quando necessário.

Este comportamento do autômato adaptativo pode ser atribuido à forma como se dá sua operação: o autômato deve exibir um conjunto de estados e de transições que pode variar conforme a cadeia de entrada a ser reconhecida.

À medida que o reconhecimento progride, a configuração do autômato evolui, a partir de uma configuração inicial, através de sucessivas transições de estados, chamadas e retornos de sub-máquinas, e ampliações do autômato resultantes da execução de transições adaptativas, de acordo com as necessidades da particular cadeia de entrada em análise.

Assim, o autômato poderá utilizar, dos mecanismos oferecidos pelo modelo adaptativo, apenas os recursos de reconhecimento que forem estritamente necessários à aceitação de cada cadeia de entrada, permitindo desta forma que a operação do autômato seja feita sempre da forma mais econômica.

4.1 Conceituação do autômato adaptativo

A seguir, estão apresentados os principais conceitos que serão utilizados na formalização dos autômatos adaptativos. Apresenta-se inicialmente o conceito de espaço de máquinas de estados para designar o conjunto de todos os possíveis autômatos de pilha, cujos elementos correspondem às diversas configurações físicas que podem ser assumidas pelos autômatos adaptativos.

Como se sabe, uma máquina de estados é constituida de estados, memória e transições, destacando-se, como estados diferenciados, um estado inicial e um conjunto de estados finais da máquina, os quais exercem um papel importante no controle da operação do autômato.

Um autômato adaptativo consta de uma máquina de estados, que se apresenta, antes do reconhecimento de uma sentença, na forma de uma máquina de estados inicial fixa, à qual são impostas sucessivas alterações durante a sua operação, alterações estas ditadas pelos movimentos do autômato, resultantes da aplicação de suas regras de movimentação, de acordo com o texto-fonte.

Desta maneira, estados e transições podem ser eliminados ou incorporados ao autômato em decorrência de cada um dos passos de aprendizado executados pelo mesmo durante a análise da entrada.

A cada execução de uma transição adaptativa, ou seja, a cada ocasião na qual a máquina de estados que implementa o autômato adaptativo sofre alguma mudança em sua configuração, tudo se passa como se surgisse uma nova máquina de estados, caracterizando, para o autômato, a execução de um passo adicional em uma *trajetória de reconhecimento* do texto de entrada, em um *espaço de máquinas de estados*.

Desta forma, caracterizam-se, como máquinas de estados diferenciadas, a máquina de estados inicial, máquinas de estados intermediárias e a máquina de estados final, para cada cadeia de entrada que se estiver considerando.

O reconhecimento de um texto de entrada por um autômato adaptativo dá-se, desta maneira, através dos seguintes passos:

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- início do reconhecimento da cadeia de entrada, com o autômato posicionado no estado inicial da sua máquina de estados inicial
- execução de uma seqüencia de transições que se mostre possível enquanto a cadeia de entrada não se esgotar:
  - transição própria da máquina de estados corrente, com ou sem consumo de átomo da cadeia de entrada: transição interna a uma submáquina, transição de chamada de uma submáquina ou transição de retorno a uma submáquina chamadora
  - transição adaptativa, provocando mudança da máquina de estados corrente e evolução para um estado conveniente da nova máquina de estados
- término do reconhecimento:
  - normal, em algum estado final de uma máquina de estados final, com a cadeia de entrada esgotada, ou
  - por erro de sintaxe, se a cadeia de entrada se esgotar sem que tenham sido atingidas as condições de término normal, ou então, se não for possível executar nenhuma transição a partir do estado corrente, com a cadeia de entrada ainda não esgotada

■

A seqüencia das transições adaptativas que são executadas pelo autômato durante o reconhecimento de uma cadeia de entrada estabelece uma trajetória no espaço das máquinas de estados, trajetória esta que faz parte integrante do processo de reconhecimento da cadeia.

Cada uma das máquinas de estados que compõem a trajetória de reconhecimento descrita pelo autômato adaptativo para a aceitação de uma dada sentença é responsável pelo reconhecimento de uma sub-cadeia deste texto-fonte, e a concatenação seqüencial de todas estas sub-cadeias compõe a cadeia de entrada original.

A evolução entre uma máquina de estados e a sua sucessora na trajetória de reconhecimento ocorre ao ser detectado, na cadeia de entrada, algum tipo de construção potencialmente pertencente à linguagem, mas que a máquina de estados corrente não reconhece.

Tipicamente a evolução entre uma máquina de estados e a sua sucessora ocorre ao serem encontradas, na sentença em análise, declarações de novos identificadores, delimitadores de escopos e outras situações características da presença de dependências de contexto no texto-fonte.

Descreve-se a seguir o autômato adaptativo, procurando enfatizar os aspectos mais intuitivos do seu funcionamento para caracterizar mais rigorosamente a sua constituição.

Pode-se identificar em um autômato adaptativo os seguintes elementos componentes:

- ω , representando toda a cadeia de entrada a ser reconhecida
- E^0 , representando a máquina de estados inicial do autômato adaptativo, que o implementa ao início de sua operação.
- E^n, representando a máquina de estados que implementa o autômato adaptativo ao final do reconhecimento da cadeia de entrada ω ($n \geq 0$).
- E^i, representando a máquina de estados que implementa o autômato adaptativo imediatamente após a i-ésima execução de suas transições adaptativas ($0 \leq i \leq n$).

Isto posto, o reconhecimento de ω pelo autômato adaptativo poderá ser caracterizado macroscopicamente pela seqüencia de reconhecimentos parciais de sucessivas sub-cadeias α^i da cadeia ω pelas correspondentes máquinas de estados E^i ($0 \leq i \leq n$), ou seja, para reconhecer ω o autômato terá percorrido, no espaço de máquinas de estados, uma trajetória de reconhecimento correspondente à seqüencia

$$(E^0,\alpha^0),\ (E^1,\alpha^1),\ \ldots\ ,\ (E^n,\alpha^n)\ ,$$

com $\omega = \alpha^0\ \alpha^1 \ldots\ \alpha^n$, onde (E^i,α^i) representa o consumo da sub-cadeia α^i pela máquina de estados E^i .

Denota-se esta trajetória de reconhecimento de ω pelo autômato adaptativo como

$$(E^0\ ,\alpha^0\ ) \rightarrow (E^1\ ,\alpha^1) \rightarrow \ldots \rightarrow (E^n\ ,\alpha^n\ )\ .$$

Representando-se por $\rightarrow^*$ o fechamento recursivo e transitivo da operação $\rightarrow$ de movimentação do autômato adaptativo através da sua trajetória de reconhecimento da cadeia de entrada, pode-se descrever tal trajetória, alternativamente, de forma abreviada, como $(E^0\ ,\alpha^0\ ) \rightarrow^* (E^n\ ,\alpha^n\ )$.

Caracteriza-se a seguir a operação do autômato adaptativo em um nível mais refinado, procurando-se visualizar inicialmente o papel de cada uma das máquinas de estados E^i em sua tarefa de processamento da correspondente sub-cadeia α^i da cadeia de entrada ω .

Inicialmente, identificam-se os elementos deste processo:

- $\omega^0 = \omega$, representando a cadeia de entrada completa, submetida inicialmente à máquina de estados E^0 .
- ω^i , representando a parte da cadeia de entrada submetida à máquina de estados E^i , imediatamente após a ação de uma transição adaptativa executada por E^{i-1} ($0 < i \leq n$), ou então a cadeia de entrada completa, imediatamente após a ativação inicial do autômato adaptativo ($i = 0$).

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- α^i , representando o prefixo de ω^i efetivamente consumido por E^i , desde sua ativação até o final do reconhecimento da cadeia de entrada ($i = n$), ou até a execução de alguma transição adaptativa, que promoverá a ativação de E^{i+1} ($0 \le i < n$).

Desta maneira, é possível decompor o processo de reconhecimento de ω como sendo formado por uma seqüencia de etapas, em cada qual:

- uma sub-cadeia $\omega^i = \alpha^i \underline{\alpha}^i$ é submetida a uma máquina de estados E^i
- E^i consome um prefixo α^i da sub-cadeia ω^i
- Neste ponto, tem-se duas situações:
  - ou E^i executa uma transição adaptativa que promove a ativação da máquina de estados E^{i+1} , a qual recebe como entrada a parte restante da cadeia inicial ω^i , ou seja, a sub-cadeia $\omega^{i+1} = \alpha^i$ $(0 \le i < n)$,
  - ou então, E^i encerra o processamento do autômato adaptativo após esgotar a cadeia de entrada ($i = n$).

Observe-se que, se ao início da sua operação, cada máquina de estados Ei recebe uma cadeia ω^i , da qual consumirá o prefixo α^i , restando $\underline{\alpha}^i$ a ser submetido à máquina de estados E^{i+1} , então $\underline{\alpha}^i = \alpha^{i+1} \ldots \alpha^n$ $(0 \le i < n)$, com $\underline{\alpha}^n = \varepsilon$.

Caracterizado o funcionamento macroscópico do autômato adaptativo, é necessário definir, em seguida, a maneira através da qual se processará uma transição adaptativa, ou seja, a forma de determinação de uma máquina de estados que possa ser a sucessora daquela que, em um dado momento, implementa o autômato adaptativo.

Embora nada impeça que sejam definidas transições adaptativas capazes de impor alterações irrestritas a uma dada máquina de estados, é conceitualmente mais conveniente apresentá-las na forma de seqüencias de alterações elementares, correspondentes às operações de inserção ou de eliminação de elementos, sobre o conjunto de transições na máquina de estados:

- inserção de uma nova transição, pela ação de uma operação de *ampliação elementar*
- eliminação de uma transição existente, pela ação de uma operação de *redução elementar*

Embora limitados, estes recursos permitem a representação de muitas operações mais complexas, como, por exemplo, a alteração do próprio conjunto de submáquinas do autômato, ou mesmo a alteração das suas operações de auto-modificação implementadas pelas transições adaptativas.

Isto pode ser feito através da ação conjunta de operações elementares de alteração, representada pelo agrupamento adequado de ações de ampliação e de redução elementares, formando as transições adaptativas em sua forma mais geral.

Assim, as transições adaptativas podem ser vistas como uma forma integrada de representação de um grupo de operações mais primitivas de auto-modificação do autômato adaptativo, a serem executadas durante a sua operação, sempre que for atingida uma determinada situação pré-estabelecida, a ela associada.

Sendo P^i o conjunto de produções de E diz-se que uma máquina de estados E' , com conjunto de produções P^{i+1}, é uma *ampliação elementar* da máquina E (denota-se $E' \approx E$) sempre que E' incorporar alguma nova transição p' , ausente em E , preservando os estados e transições de E e incluindo os eventuais novos estados, símbolos de entrada ou símbolos de pilha que estejam referenciados em p' mas ausentes na definição da máquina de estados corrente E :

$$E' \approx E \Leftrightarrow P^{i+1} = P^i \cup \{p'\} \quad \text{, com} \quad p' \notin P^i$$

Note-se que pode ocorrer a execução de operações de ampliação elementar que especifiquem a inclusão de uma transição p' que já exista no conjunto de transições de E . Neste caso, a ação de tal operação de ampliação é vazia.

Analogamente, diz-se que uma máquina de estados E' é uma *redução elementar* da máquina E (denota-se $E' \leftrightarrow E$) sempre que E' preservar todas as transições de E , exceto uma transição p' , eliminando eventualmente estados, símbolos de entrada ou símbolos de pilha referenciados em p' mas ausentes no conjunto de transições restante, que define a máquina de estados E' :

$$E' \leftrightarrow E \Leftrightarrow P^{i+1} = P^i - \{p'\} \quad \text{, com} \quad p' \in P^i$$

Note-se que aqui também pode ocorrer a execução de operações de redução elementar que especifiquem a eliminação de uma transição p' que não exista no conjunto de transições de E. Neste caso, naturalmente, a ação de tal operação de redução não terá qualquer efeito sobre o autômato.

Uma vez definidas as operações de ampliação e de redução elementares, pode-se introduzir o conceito mais geral de transição adaptativa, que corresponde à execução simultânea de mais de uma operação elementar de ampliação e/ou redução como resposta à ocorrência de uma situação associada pré-estabelecida, durante a operação do autômato adaptativo.

Uma transição adaptativa pode ser caracterizada por uma quádrupla $p_i = (t_i , \mathcal{A}_i , t'_i , \mathcal{B}_i)$, em que os seguintes elementos podem ser identificados:

- t_i , a configuração específica a que o autômato deve estar submetido para que a transição adaptativa p possa ser aplicada

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- $\mathcal{A}_i$, a ação adaptativa que deve ser aplicada ao autômato adaptativo imediatamente antes da alteração da sua situação para t' .
- t'_i , a configuração-padrão à qual o autômato deve ser conduzido pela aplicação da transição p_i , desde que a execução de $\mathcal{A}_i$ não tenha inviabilizado tal operação pela eliminação da transição p_i .
- $\mathcal{B}_i$, a ação adaptativa que deve ser aplicada ao autômato adaptativo logo após a alteração da sua situação para t'_i.

Note-se que tanto $\mathcal{A}_i$ como $\mathcal{B}_i$ podem, em muitos casos, ser omitidas, como ocorre no caso de transições que não exijam ações de alteração do autômato, quando ambas são omitidas. Em outros casos, pode ser necessária a utilização de apenas uma ou outra, e em algumas situações mais raras, a de ambas.

É necessário, para completar a descrição do funcionamento de um autômato adaptativo, definir a composição das funções e das ações adaptativas.

Para que possam ser chamadas, como ações adaptativas, as funções adaptativas devem ser previamente declaradas.

Uma declaração de função adaptativa pode ser descrita como uma ênupla do tipo

$$(\mathcal{F} , \mathcal{P} , \mathcal{V} , \mathcal{G} , \mathcal{C} , \mathcal{E} , \mathcal{I} , \mathcal{A} , \mathcal{B})$$

onde

- $\mathcal{F}$ representa o nome da função adaptativa declarada.
- $\mathcal{P}$ representa uma lista ordenada dos parâmetros formais $(\rho_1 , \rho_2 , \ldots)$ da função adaptativa $\mathcal{F}$, preenchidos automaticamente, antes do início da execução da função adaptativa, com os valores assumidos pelos argumentos da chamada da função, preservando tais valores durante toda a execução da função.
- $\mathcal{V}$ representa o conjunto de nomes de variáveis, símbolos associados a elementos cujos valores são desconhecidos no instante da chamada da função, e que são preenchidos, uma única vez durante cada execução da função adaptativa, como resultado de ações adaptativas de consulta.
- $\mathcal{G}$ representa um conjunto de nomes de geradores, variáveis especiais que são automaticamente preenchidas a cada chamada da função adaptativa, com valores novos, não utilizados pelo autômato até esta ocasião, permanecendo com este valor durante toda a execução da função.
- $\mathcal{C}$ designa uma seqüencia de padrões a serem consultados no conjunto de produções da máquina de estados corrente, para preenchimento das variáveis indefinidas referenciadas.
- $\mathcal{E}$ designa uma seqüencia dos padrões das produções a serem consultadas para preenchimento das variáveis indefinidas referenciadas, para depois serem eliminadas do conjunto de produções da máquina de estados corrente.
- $\mathcal{I}$ designa uma seqüencia de padrões de produções a serem inseridas no conjunto das produções da máquina de estados corrente.
- $\mathcal{A}$ representa uma seqüencia de chamadas de ações adaptativas, a ser efetuada antes da execução da função adaptativa que está sendo declarada.
- $\mathcal{B}$ representa uma seqüencia de chamadas de ações adaptativas, a ser efetuada depois da execução da função adaptativa que está sendo declarada.

Convém frisar que os padrões referidos acima, embora assumam a forma de produções adaptativas, não são executados, e sim consultados, eliminados ou inseridos no conjunto de produções que compõem o autômato adaptativo corrente quando da execução das ações adaptativas.

Uma ação adaptativa corresponde à chamada de uma função adaptativa, e é dada por um par ordenado $(\mathcal{F} , \pi)$ onde:

- $\mathcal{F}$ representa o nome da função adaptativa correspondente à ação adaptativa em questão
- π simboliza uma seqüencia, eventualmente vazia, de argumentos $\tau_1 , \tau_2 , \ldots$ onde os τ_i correspondem posicionalmente aos parâmetros formais $\rho_1 , \rho_2 , \ldots$ da função adaptativa $\mathcal{F}$, e designam valores a serem utilizados pela função adaptativa em substituição aos correspondentes parâmetros formais para a composição de transições adaptativas a serem inseridas, consultadas ou eliminadas do autômato. Os argumentos τ_i podem assumir quaisquer valores compatíveis com o papel que representam nas produções adaptativas em que são referenciados.

Para formalizar a operação do autômato adaptativo, define-se inicialmente para um autômato adaptativo M, ao qual está sendo submetida uma cadeia de entrada ω , o conceito de uma *configuração* $t_k = (E_k , \gamma_k , q_k , \omega_k)$ em um instante k como sendo o conjunto das informações que definem todos os seus parâmetros naquele instante (o índice superior i identifica, na notação empregada abaixo, a situação da máquina M após i transições adaptativas):

- E_k , alguma *máquina de estados* que implementa M naquele instante, após a execução de i transições adaptativas de M , ou seja, $E_k = E^i$.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- γ_k , o conteúdo completo da *pilha* do autômato no instante k , representando a seqüencia dos retornos de sub-máquina pendentes naquele instante, e que deverão pois ser realizados antes do encerramento normal do reconhecimento de ω_i.
- q_k , o *estado* corrente da máquina de estados E^i no instante k , ou seja, algum dos estados $q^i_{r,s}$ do conjunto Q^i dos estados de E^i (observe-se que $q = q^i_{r,s}$ representa o s-ésimo estado da r-ésima submáquina do autômato de pilha que implementa E^i naquele instante k).
- ω , a parte da *cadeia de entrada* ω_k que ainda não foi consumida pelo autômato adaptativo M no instante k , ou, mais especificamente, o sufixo da sub-cadeia ω^i ainda não processado pela máquina de estados E^i naquele instante.

Uma *configuração inicial* $t_0 = (E_0 , \gamma_0 , q_0 , \omega_0)$ pode ser caracterizada como sendo a situação em que o autômato deve ser posicionado imediatamente antes de iniciar a sua operação. Na configuração inicial deve-se ter:

- E_0 , a máquina de estados inicial do autômato adaptativo, ou seja, $E_0 = E^0$.
- γ_0 , o conteúdo inicial da pilha, que deve ser sempre vazio, ou seja, $\gamma_0 = Z_0$, o marcador de pilha vazia apenas.
- q_0 , o estado inicial do autômato adaptativo, que deve sempre corresponder ao estado inicial de sua máquina de estados inicial E , portanto deve-se ter $q_0 \in Q^0_0$, logo $q_0 = q^0_{0,0}$, o estado inicial da submáquina inicial da máquina de estados inicial do autômato adaptativo.
- ω_0 , a cadeia de entrada inicial do autômato adaptativo, ou seja, $\omega_0 = \omega$.

Analogamente, pode-se caracterizar como configuração final qualquer situação da forma

$$t_f = (E_f , \gamma_f , q_f , \omega_f)$$

que possa ser atingida pelo autômato ao encerrar com sucesso o reconhecimento da cadeia de entrada ω . Em uma *configuração final*, o autômato adaptativo deve exibir:

- E_f , a máquina de estados final, ou seja, aquela em que a cadeia de entrada é finalmente esgotada, ou seja, $E_f = E^n$.
- γ_f , o conteúdo final da pilha, que deve atender ao critério de aceitação imposto pela definição das máquinas de estado E^i . é intuitivo adotar-se como critério para o final correto de reconhecimento, que a pilha esteja vazia, ou seja, $\gamma_f = Z_0$, em que a pilha contém apenas o marcador de pilha vazia.
- q_f , um dos estados finais do autômato adaptativo, ou seja, um dos elementos do conjunto F^n de estados finais da máquina de estados final E^n do autômato adaptativo: $q_f \in F^n$.
- ω_f , a parte da cadeia de entrada ainda não consumida, deve estar vazia ao final do reconhecimento: $\omega_f = \varepsilon$.

4.2 Notações para a Especificação de Autômatos Adaptativos

São apresentadas a seguir duas formas - algébrica e gráfica - através das quais o autômato adaptativo é denotado neste trabalho.

4.2.1 Uma Notação Algébrica para os Autômatos Adaptativos

Admita-se que, em um determinado momento do seu funcionamento, o autômato adaptativo seja submetido a uma situação em que o estado corrente seja E , que o próximo elemento da cadeia de entrada a ser consumido pelo autômato seja σ , e que no topo da pilha corrente figure o elemento π .

Por convenção, os meta-símbolos γ e α representam, respectivamente, de forma implícita, os conteúdos da pilha e da cadeia de entrada que ainda não tenham sido analisados na ocasião da aplicação da produção.

O conteúdo real das partes da pilha e da cadeia de entrada, representados respectivamente por γ e α , é irrelevante para efeito de aplicação da produção.

Adotadas estas convenções, pode-se denotar a situação corrente do autômato adaptativo através da tripla $(\gamma\ p , E , \sigma\ a)$.

Seja P o conjunto corrente de produções do autômato, cujos elementos se apresentam em algum dos dois seguintes formatos:

$$(e , s) : \mathcal{A} , \rightarrow e' , \mathcal{B}$$

ou $$(\gamma\ g , e , s\ \alpha) : \mathcal{A} , \rightarrow (\gamma\ g' , e' , s'\ \alpha) , \mathcal{B}$$

Observe-se que, nestas condições:

- O primeiro formato da produção abrevia o segundo no caso particular em que $g = g' = s' = \varepsilon$, ou seja, na situação em que tais elementos são omitidos, de modo que a produção se apresente na forma seguinte:

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

$$(\gamma\ ,\ e\ ,\ s\ \alpha)\ :\ \mathcal{A}\ ,\ \rightarrow\ (\gamma\ ,\ e'\ ,\ \alpha)\ ,\ \mathcal{B}$$

representando uma produção que pode ser aplicada sem levar em consideração o conteúdo da pilha, e que não efetua qualquer inclusão de símbolo na cadeia de entrada.

- **e** representa o estado-origem da transição descrita pela produção em questão.
- **e'** representa o estado-destino da transição que a produção descreve.
- **g** (opcional) indica o conteúdo exigido do topo da pilha antes da execução da transição. Este elemento é desempilhado pela aplicação da produção. No caso de omissão de **g** , a aplicação da produção não será dependente do conteúdo do topo da pilha.
- **g'** (opcional) representa o elemento a ser empilhado como resultado da aplicação da produção. Caso seja omitido, nenhum empilhamento será efetuado como decorrência da execução da produção.
- Caso $g = g' \neq \varepsilon$, tais elementos aparecem explicitamente na produção, a qual neste caso representa uma transição que não efetua alterações no conteúdo da pilha, mas cuja aplicação está condicionada a que o topo da pilha contenha o valor $g = g'$.
- **s** (opcional) quando representado por um elemento do alfabeto de entrada, designa o símbolo a ser consumido pela aplicação da produção. Este elemento pode ser um dos símbolos básicos de que se compõe o texto de entrada, ou então qualquer dos símbolos nele empilhados pela ação da execução das produções do autômato adaptativo. Representa-se o mapeamento de um símbolo ASCII **"x"** para a forma de código (não-terminal) através da notação ∇**"x"**.

  Não havendo ambigüidade, é possível abreviar ∇**"x"** simplesmente como **x** .

Há portanto para **s** os seguintes casos a estudar:

- ▶ No caso em que **s** representa a cadeia vazia (denotada por ε nas produções do primeiro tipo e omitida nas do segundo tipo), trata-se de uma transição em vazio, e neste caso a aplicação da produção não é condicionada ao conteúdo da cadeia de entrada.
- ▶ No caso de **s** corresponder a um símbolo ASCII denotado entre aspas, isto indica que tal símbolo deve ser utilizado diretamente pela transição, ou seja,

  $$(e\ ,\ "\sigma")\ :\ \mathcal{A}\ ,\ \rightarrow\ e'\ ,\ \mathcal{B}$$

  representa apenas uma transição convencional consumindo "σ" **.**

  Observar que, a bem da clareza, convém confinar o uso deste tipo de produções apenas à parte do autômato que executa a função de análise léxica, onde é extensivamente utilizado para a extração de átomos a partir do texto-fonte.
- ▶ No caso em que, em uma produção denotada como

  $$(\ e\ ,\ s\ )\ :\ \mathcal{A}\ ,\ \rightarrow\ e'\ ,\ \mathcal{B}$$

  o elemento **s** corresponder a qualquer símbolo que não seja um dos símbolos ASCII, deve-se subentender que este símbolo esteja representando um não-terminal, ao qual deverá estar associada uma sub-máquina que, a partir de algum dos seus estados finais, empilhe na cadeia de entrada o símbolo s correspondente.

  Supondo que **I** seja o estado inicial de tal sub-máquina, a produção acima deverá ser interpretada como sendo a abreviatura do seguinte par de produções:

  $$(\gamma\ ,\ e\ ,\ \alpha)\ :\ \mathcal{A}\ ,\ \rightarrow\ (\gamma\ e\ ,\ I\ ,\ \alpha)$$

  $$(\gamma\ ,\ e\ ,\ s\ \alpha)\ :\ \rightarrow\ (\gamma\ ,\ e'\ ,\ \alpha)\ ,\ \mathcal{B}$$

  desde que se convencione que todos os retornos de sub-máquinas empilhem na cadeia de entrada uma informação, associada ao estado final por onde se deu o retorno, correspondente ao símbolo não-terminal s esperado. Notar que $\mathcal{A}$ e $\mathcal{B}$ são opcionais, e que a sub-máquina mencionada efetua normalmente a função de um analisador léxico.

  Por questões de clareza, convém que nas partes de autômato encarregadas da análise sintática não figurem produções que representem transições diretas com caracteres ASCII.
- ▶ No caso de s representar um símbolo ASCII σ , e este não estiver denotado entre aspas ou então se estiver denotado na forma ∇"σ" :

  $$(\ e\ ,\ \sigma\ )\ :\ \mathcal{A}\ ,\ \rightarrow\ e'\ ,\ \mathcal{B}$$

  ou

  $$(\ e\ ,\ \nabla\ "\sigma"\ )\ :\ \mathcal{A}\ ,\ \rightarrow\ e'\ ,\ \mathcal{B}$$

  deve-se entender que tal produção abrevia o seguinte par:

  $$(\gamma\ ,\ e\ ,\ "\sigma"\ \alpha)\ :\ \mathcal{A}\ ,\ \rightarrow\ (\gamma\ e\ ,\ I\ ,\ "\sigma"\ \alpha)$$

  e

  $$(\gamma\ ,\ e\ ,\ s\ \alpha)\ :\ \rightarrow\ (\gamma\ ,\ e'\ ,\ \alpha)\ ,\ \mathcal{B}$$

  com **s** representando, portanto, para efeito de análise sintática, um meta-símbolo que simboliza o código ASCII correspondente.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

O uso de transições envolvendo símbolos formados de caracteres ASCII isolados na parte sintática do autômato pode ser indicada utilizando-se este tipo de produções em lugar de transições diretas com caracteres ASCII.

- **s'** (opcional) corresponde a um símbolo, a ser incluido na cadeia de entrada como resultado da execução da transição descrita pela produção.
- Caso **s = s'** $\neq \varepsilon$, tais elementos aparecem explicitamente na produção, que então representa uma transição que não efetua alterações sobre a cadeia de entrada, mas que exige, para ser aplicada, que o próximo símbolo de entrada a ser consumido seja **s = s'** , implementando assim uma operação de look-ahead.
- Os elementos **g , g', s** e **s'** explicitam apenas a parte da pilha ou da cadeia de entrada que são relevantes à aplicação da produção.
- **(**γ **g , e , s** α**)** é a situação do autômato adaptativo à qual a produção pode ser aplicada.
- **(**γ **g' , e' , s'** α**)** é a situação obtida como resultado da aplicação da produção à situação **(**γ **g , e , s** α**) .**
- $\mathcal{A}$ (opcional) representa uma ação adaptativa a ser executada antes de a transição de estado ser efetuada.
- $\mathcal{B}$ (opcional) simboliza uma ação adaptativa a ser executada após a realização da transição.
- Em casos de múltiplos caminhos de transição a partir de um dado estado do autômato adaptativo, a escolha da transição a ser executada é feita de acordo com a seguinte convenção de precedências:
  - ▶ prioridade máxima para transições diretas, consumindo códigos elementares (ASCII) não-interpretados.
  - ▶ segunda prioridade para transições que consomem símbolos não-terminais, empilhados pelas transições de retorno de sub-máquinas explícita ou implicitamente chamadas.
  - ▶ terceira prioridade para transições de chamada de sub-máquina, representadas por mudanças de estado com consumo de símbolos que não são códigos elementares.
  - ▶ quarta prioridade para transições em vazio.
  - ▶ última prioridade para transições de retorno em estados finais de sub-máquina.

São descritas a seguir a forma e a interpretação dos diversos elementos que descrevem o autômato adaptativo.

- **Ações adaptativas**

As ações adaptativas $\mathcal{A}$ **e** $\mathcal{B}$, quando presentes, são formadas de listas de chamadas de funções adaptativas, a serem executadas seqüencialmente na ocasião da aplicação da produção que as referencia.

Em sua forma mais geral, uma ação adaptativa é denotada como uma seqüência, entre chaves, de chamadas simples φ de funções **i** adaptativas, separadas por vírgulas:

$$\{ \varphi_1 , \varphi_2 , \ldots , \varphi_n \}$$

Em determinados casos particulares podem ser adotadas as seguintes abreviações à notação acima:

- Ações sem efeito (correspondentes a uma seqüencia vazia **{}**) podem ser omitidas.
- Ações formadas de uma seqüencia unitária (tal como **{** φ_1 **}**) podem ser abreviadas omitindo-se as chaves e denotando-se-as como simples chamadas da única função a ser executada (ou seja, simplesmente φ_1)
- **As Ações Adaptativas Iniciais**

A chamada (opcional) de uma função adaptativa inicial (aquela que precede as ações) indica que a correspondente ação adaptativa deverá ser executada antes que sejam efetuadas quaisquer das alterações ou consultas ao autômato que porventura estejam indicadas na lista especificada de ações adaptativas elementares.

Portanto, os eventuais argumentos destas ações deverão ser obrigatoriamente constantes ou então depender apenas dos parâmetros formais da função que está sendo declarada.

- **As Ações Adaptativas Finais**

A chamada (também opcional) de uma função adaptativa final (aquela que sucede as ações) indica uma ação adaptativa a ser executada após efetuadas todas as consultas e alterações ao autômato indicadas na lista de ações adaptativas elementares especificada.

Neste caso, todas as variáveis e geradores também poderão ser utilizados para compor os argumentos destas ações.

- As Listas de Ações Adaptativas Elementares

A lista de ações adaptativas elementares (eventualmente vazia) indica as consultas e alterações explícitas a serem realizadas sobre a configuração corrente do autômato adaptativo.

Esta lista consta de uma cadeia, sem separadores, de ações adaptativas elementares de três tipos possíveis:

- ações adaptativas elementares de consulta a uma produção, na forma seguinte:

? [produção]

- ações adaptativas elementares de eliminação de uma produção, denotadas como

- [produção]

- ações adaptativas elementares de inclusão de uma produção, indicadas através da notação

+ [produção]

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Listas de ações adaptativas similares, que se distinguem apenas pela variação de valor de um dos seus componentes básicos (estado, símbolo de entrada, símbolo de pilha ou argumento de função) podem ser abreviadas especificando-se em seu lugar um conjunto, em que é dada a forma geral da produção e uma variável local, à qual se associa um conjunto de valores que tal variável deve assumir:

$$\{ \textit{lista de ações} \ \forall \ \textit{variável local} \in \textit{conjunto} \ \}$$

A *lista de ações* representa um conjunto qualquer de ações adaptativas elementares, cuja forma foi definida acima, denotadas sem separadores, e que devem ser instanciadas para cada valor dos elementos do *conjunto* indicado, valores estes a serem substituidos consistentemente em todas as ocorrências da *variável local* nas ações elementares indicadas.

A variável local tem seu escopo limitado exclusivamente ao âmbito da abreviatura em que é referenciada, não devendo pois ser declarada no cabeçalho da função adaptativa, mas podendo portanto ser reutilizada em outras abreviações similares, que nela não estejam aninhadas.

Note-se que este é apenas um artifício para reduzir o trabalho de escrita de tais listas, e opera como macro, não existindo fisicamente em época de operação do autômato.

- **Chamadas de Funções Adaptativas**

No caso geral, os elementos φ_i , que representam chamadas de funções adaptativas de nome $\mathcal{F}_i$, com n_i parâmetros ($n_i \geq 0$), explicitam os seus n_i argumentos $\tau_{i,j}$, $(1 \leq j \leq n_i)$, assumindo no caso o formato seguinte:

$$\mathcal{F}_i \ (\ \tau_{i,1} \ , \ \tau_{i,2} \ , \ \ldots \ , \ \tau_{i,ni} \ )$$

A correspondência entre os argumentos e os respectivos parâmetros formais, indicados na declaração da função (definida adiante) é posicional, e a transferência dos valores dos argumentos para os parâmetros correspondentes é feita automaticamente antes da execução de qualquer outra atividade indicada na função.

Podem ser usados como argumentos de uma função adaptativa: as constantes (números inteiros, o meta-símbolo ε , que denota a cadeia vazia, os símbolos que denotam os estados, os símbolos dos alfabetos de entrada ou de pilha), e os identificadores das variáveis, dos geradores e dos parâmetros formais.

Não podem ser usados, como argumentos de função, nomes de funções adaptativas nem os meta-símbolos utilizados nas notações empregadas para descrever o autômato.

- **Declaração de Funções Adaptativas**

Para bem definir o autômato adaptativo, é necessário que sejam declaradas as funções adaptativas, para que elas possam ser referenciadas (chamadas) nas ações adaptativas elementares que definem a função.

Na declaração da função é indicado o conjunto das alterações a serem efetuadas no autômato como decorrência da execução de ações adaptativas que acionam tais funções.

A declaração de uma função adaptativa, de nome $\mathcal{F}$ e com n parâmetros τ_1 , τ_2 , ... , τ_n , é denotada através de um cabeçalho (em que são indicados o nome e os parâmetros formais da função), e do corpo da declaração da função, que segue o cabeçalho (onde são relacionadas as ações a serem executadas quando da ativação da função adaptativa).

- **O Cabeçalho da Declaração da Função Adaptativa**

No cabeçalho, é apresentado o nome $\mathcal{F}$ da função, seguido de uma lista (eventualmente vazia), entre parênteses, dos seus parâmetros formais separados por vírgulas, finalizado-se o cabeçalho por meio de um sinal **=** :

$$\mathcal{F} \ (\ \tau_1 \ , \ \tau_2 \ , \ \ldots \ , \ \tau_n \ ) \ =$$

Note-se que, diferentemente do que ocorre na maioria das linguagens de programação, somente parâmetros de entrada são permitidos para as funções adaptativas, os quais são preenchidos, a cada chamada, e antes da execução da função, com os valores dos argumentos a eles associados, indicados na chamada da ação adaptativa responsável pela ativação da função, permanecendo inalterados durante toda a execução da função.

- **Corpo da Declaração da Função Adaptativa**

Em seqüencia ao cabeçalho, a declaração da função deve indicar as ações adaptativas elementares a serem impostas pela função ao autômato.

O corpo da declaração da função, que descreve as alterações a serem impostas ao autômato adaptativo pela ação adaptativa representada pela função, é denotado entre chaves, devendo apresentar-se com a seguinte estrutura:

```
{       declaração de nomes (opcional)
        :
        declaração de ações (opcional)       }
```

- **A Declaração de Nomes**

Na *declaração de nomes* devem ser indicados todos os identificadores de variáveis e de geradores utilizados na função.

Em ambos os casos os identificadores são utilizados para denotar estados, ou então símbolos de entrada ou de pilha, ou mesmo argumentos de funções adaptativas que sejam referenciadas no corpo da função.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Nela devem ser relacionados, e separados entre si por vírgulas, os identificadores vi de todas as variáveis utilizadas, e os identificadores gj de todos os geradores (conceituados adiante) empregados, nesta ordem.

Note-se que o conjunto de identificadores de variáveis, o de geradores, ou mesmo ambos, podem ser eventualmente vazios.

Tanto na declaração como na utilização das variáveis, a representação empregada é a de identificadores convencionais, denotados por letras gregas, ou então por cadeias de letras e dígitos, iniciadas por uma letra.

- **As Variáveis**

As variáveis operam de forma diferente da usualmente adotada em linguagens algorítmicas: são preenchidas por ações adaptativas elementares de consulta ou de eliminação, e, uma vez preenchidas, mantêm permanentemente o valor assim recebido enquanto a função estiver sendo executada, não podendo portanto ser alteradas durante a execução da função.

- **Os Geradores**

Geradores são variáveis diferenciadas, que são preenchidas automaticamente, sem solicitação explícita, com valores únicos, ainda não utilizados na definição corrente do autômato, a cada início da execução da ação adaptativa.

A exemplo das variáveis, os geradores também não podem ser alterados pela execução das ações adaptativas elementares que compõem a função.

Denotam-se, na declaração, os geradores através de identificadores normais, acrescidos do sufixo **`*`** , mas quando empregados em ações adaptativas elementares omite-se tal sufixo, adotando-se portanto para os geradores a mesma notação empregada para denotar as variáveis, ou seja, a de identificadores simples.

Observe-se que, enquanto durar a execução da função, todas as referências a um mesmo gerador assumem um mesmo único valor, aquele com que o gerador for automaticamente preenchido antes do início da execução da ação adaptativa.

A forma de uma declaração de nomes assume pois o seguinte aspecto:

$$v_1 , v_2 , \ldots , v_m , g_1* , g_2* , \ldots , g_n*$$

com $m,n \geq 0$.

- **A Declaração de Ações**

A declaração de ações apresenta-se como uma lista de ações adaptativas elementares, eventualmente vazia, opcionalmente precedida e/ou seguida de uma chamada de função adaptativa.

4.2.2 Uma Notação Gráfica para os Autômatos Adaptativos

Uma forma mais confortável para a denotação dos autômatos adaptativos é a que explora recursos gráficos, os quais facilitam a inspeção visual do seu funcionamento.

As representações gráficas do autômato adaptativo aqui apresentadas procuram acompanhar aquelas que são usualmente adotadas para a representação de autômatos finitos e de pilha, porém incorporando os elementos ausentes naqueles modelos, como é por exemplo o caso das ações adaptativas.

Nas representações gráficas do autômato adaptativo, os estados são denotados como retângulos no interior dos quais consta o nome do estado, e de onde saem ou para onde chegam setas, que indicam transições.

Conexões entre estados, denotadas em linhas tracejadas, indicam transições não existentes inicialmente, mas que serão eventualmente incluidas por obra das ações adaptativas a serem executadas durante a operação do autômato.

Próximo às setas constam indicadores do tipo de transição a realizar: meta-símbolos **(ε , λ)** , símbolos da cadeia de entrada ou não-terminais a serem consumidos pela transição correspondente, eventualmente acompanhados das ações adaptativas a serem executadas na ocasião.

O meta-símbolo **ε** denota transições em vazio.

O meta-símbolo λ denota transições sem consumo de átomos, que devem ocorrer quando não for possível efetuar nenhuma transição explícita a partir do estado corrente.

Não-terminais reconhecidos na análise sintática são denotados pelos seus nomes, escritos em itálico, de forma similar à utilizada nas produções.

Os não-terminais resultantes da ação do analisador léxico podem assumir duas notações:

- nomes em itálico, representando cadeias de caracteres com conotação própria, e
- a notação ∇**"x"** para indicar um não-terminal associado ao símbolo ASCII x.

Não havendo ambigüidade, é possível abreviar ∇ **"x"** simplesmente como **x** .

Assim, por exemplo, o código ASCII **"&"** (e-comercial) pode ser representado das seguintes maneiras: **"&"** , quando se tratar do próprio símbolo ASCII; ∇ **"&"** ou e-com quando se tratar do código (não-terminal) associado a este símbolo, para uso por parte do analisador sintático.

Os estados iniciais das sub-máquinas são identificados pela presença de alguma seta que os aponta mas que não provém de outro estado do diagrama.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Estados finais do autômato são representados por retângulos em linha sólida.

Os estados finais de sub-máquinas são identificados como sendo aqueles de onde parte uma seta, acompanhada da indicação de um empilhamento de não-terminal na cadeia de entrada, mas que não tenha outro estado como destino explícito.

Os empilhamentos na cadeia de entrada são denotados indicando-se o elemento a empilhar, precedido de um símbolo "↓".

Setas partindo de um estado, sem destino explícito e sem indicação de empilhamento, indicam que este estado está representado em outro diagrama, denotando assim apenas conexões gráficas entre o diagrama corrente e algum outro, representado em outra parte do documento.

As ações adaptativas são denotadas como nas produções, na forma de chamadas de funções adaptativas escritas por extenso, e precedidas ou sucedidas por um sinal "." (ponto), cuja posição indica se a ação deve ser executada antes ou depois de consumada a mudança de estado: se o ponto preceder a ação adaptativa, isto indica que a ação deve ser executada após a mudança de estado, e se o ponto suceder a ação adaptativa esta deverá ser executada antes que tal mudança de estado seja realizada.

4.3 Mecanismos de Operação do Autômato Adaptativo

Nesta seção são descritos de forma conceitual alguns procedimentos que implementam os mecanismos teóricos inicialmente apresentados para os autômatos adaptativos.

Estes procedimentos realizam a operação de autômatos adaptativos formalizados através da notação acima descrita.

4.3.1 Mecanismo de execução de uma função adaptativa

Durante a operação do autômato adaptativo, ao ser aplicada uma produção pode ocorrer a chamada de funções adaptativas, chamadas estas responsáveis pela realização do comportamento adaptativo do autômato, e cujo funcionamento se passa a descrever através da seguinte seqüencia:

- preenchem-se inicialmente, com os valores indicados nos argumentos da chamada da função, os respectivos parâmetros formais, guardando a correspondência posicional.
- atribuem-se valores novos únicos a cada gerador, instanciando-os para a particular chamada corrente da função adaptativa.
- marcam-se como indefinidas todas as variáveis declaradas.
- executa-se, se existir, a chamada da função adaptativa que precede as ações adaptativas elementares que compõem a função, desde que nenhum de seus argumentos esteja indefinido.
- para as ações adaptativas elementares que não referenciem variáveis indefinidas, executam-se os procedimentos abaixo, na ordem seguinte (desprezando todas as ações elementares ou chamadas de função adaptativa que dependam de qualquer argumento que esteja com valor indefinido):
  - ▶ executam-se, na seqüencia que proporcionar o preenchimento do maior número possível de variáveis indefinidas, as operações de consulta correspondentes às produções que estejam indicadas em todas as operações elementares de consulta e de eliminação indicadas na função.
  - ▶ executam-se em seguida, em qualquer ordem, todas as ações elementares de eliminação de produções especificadas.
  - ▶ executam-se a seguir todas as ações elementares de inclusão de produções indicadas, em qualquer ordem.
- executa-se, se existir, a chamada da função adaptativa que sucede as ações adaptativas elementares, desde que seus argumentos estejam todos definidos.
- retorna-se ao ambiente que ativou a ação adaptativa cuja execução acaba de ser concluida.

4.3.2 Operação das ações adaptativas elementares

Para completar a descrição do funcionamento das ações adaptativas, é preciso que seja estabelecida a forma como devem ser executadas as ações adaptativas elementares.

- As ações adaptativas elementares de consulta a uma produção, denotadas como

  $$? [\ (e , s) : \mathcal{A} , \rightarrow e' , \mathcal{B} \ ]$$

  ou $? [\ (\gamma g , e , s \alpha) : \mathcal{A} , \rightarrow (\gamma g' , e' , s' \alpha) , \mathcal{B} \ ]$

  devem ser interpretadas como sendo solicitações para que produções da forma das indicadas entre colchetes sejam pesquisadas no conjunto P de produções correntes do autômato.

  A pesquisa se realiza com base na busca de produções cujas componentes tenham os valores das correspondentes componentes das produções indicadas entre colchetes, cujas variáveis definidas, parâmetros e geradores tenham sido devidamente preenchidos previamente. Obtém-se desta forma um conjunto de produções da forma indicada, cujas valores das componentes deverão preencher as restantes variáveis indefinidas que figuram na ação de consulta solicitada. Obviamente, este tipo de ação adaptativa elementar só

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

tem sentido se houver ao menos uma variável indefinida representando alguma componente da produção indicada entre colchetes.

- ações adaptativas elementares de eliminação de uma produção, denotadas como $- [(e , s) : \mathcal{A} , \rightarrow e' , \mathcal{B}]$ ou $- [(\gamma g , e , s \alpha) : \mathcal{A} , \rightarrow (\gamma g' , e' , s' \alpha) , \mathcal{B}]$ são interpretadas em duas etapas: na primeira, uma operação de consulta é efetuada, nos moldes indicados acima para ações elementares de consulta, e na segunda, uma operação de remoção propriamente dita é executada, eliminando do conjunto **P** de produções corrente a produção indicada. Estas ações elementares de remoção somente alteram o conjunto corrente de produções **P** no caso de todas as componentes da produção indicada entre colchetes estarem definidas após a consulta efetuada, caso contrário a correspondente operação de eliminação será desprezada, sem prejuízo da operação de consulta, já realizada, cujos efeitos são preservados. Assim, é inócua a aplicação de uma ação adaptativa elementar de remoção de uma produção não existente em **P** .
- ações adaptativas elementares de inclusão de uma produção, indicadas através da notação

$$+ [(e , s) : \mathcal{A} , \rightarrow e' , \mathcal{B}]$$

ou $+ [(\gamma g , e , s \alpha) : \mathcal{A} , \rightarrow (\gamma g' , e' , s' \alpha) , \mathcal{B}]$
têm efeito somente se todas as suas componentes estiverem preenchidas com valores defindos, sendo ignoradas em caso contrário. Sua atuação corresponde à alteração do conjunto **P** , que receberá uma produção adicional, da forma indicada entre colchetes, instanciada para os valores correntes das eventuais variáveis, geradores e parâmetros formais indicados. Caso a produção a ser incluida já se encontrar em **P** , a aplicação deste tipo de ação adaptativa elementar será inócua.

4.3.3 Operação geral do autômato adaptativo

Estabelecido o funcionamento de cada ação adaptativa elementar, bem como o das chamadas das funções adaptativas, passa-se a definir a operação geral do autômato.

Inicialmente, suponha-se o autômato adaptativo posicionado em uma situação inicial (Z_0 , E_0 , α_0), onde Z_0 designa a pilha vazia, E_0 o estado inicial da máquina de estados inicial do autômato, e α_0 a cadeia de entrada completa a ser processada.

Um cursor deverá então apontar o início da cadeia de entrada, permitindo que seja iniciada a extração de seus símbolos.

Nestas condições, o procedimento seguinte efetua a operação do autômato adaptativo, processando a cadeia de entrada e aceitando-a se for uma sentença da linguagem descrita pelo autômato adaptativo ou rejeitando-a em caso contrário. Um cursor deverá, nesta ocasião, apontar o início da cadeia de entrada.

1. Estando o autômato na situação corrente $(\gamma p , E , \sigma a)$, pesquisar em **P** as produções que representam transições com consumo do átomo σ que sejam aplicáveis a esta situação, representadas por aquelas cuja primeira componente seja de uma das formas $(\gamma p , E , \sigma a)$, $(\gamma , E , \sigma a)$ **ou** (E , σ). Há a considerar duas situações:
   - Na primeira, tem-se $\sigma \in$ **ASCII**, devendo-se ativar um analisador léxico antes de prosseguir
   - Na segunda, $\sigma \notin$ **ASCII**, não havendo necessidade de ativar o analisador léxico por estar sendo esperado um meta-símbolo em lugar de um elemento do alfabeto de entrada.

   Isto feito, podem acontecer as seguintes alternativas:
   - Encontrando uma única produção nestas condições, prosseguir executando-a, no passo 2.
   - Se mais de uma produção deste tipo for encontrada, constata-se uma situação não-determinística, devendo-se neste caso passar ao passo 4 para tratá-la.
   - Não sendo encontrada nenhuma produção deste tipo, pesquisar em P as transições em vazio aplicáveis, ou seja, aquelas em que a primeira componente seja de uma das formas seguintes:

     $$(\gamma p , E , \alpha) , (\gamma , E , \alpha) \text{ ou } (E , \varepsilon) .$$

     Neste caso, podem ocorrer as situações a seguir:
     - Encontrando-se uma única produção nestas condições, prosseguir no passo 2 para executá-la.
     - Se existir mais de uma, constatou-se uma situação de não-determinismo. Passar ao passo 4 para tratar o caso.
     - Não havendo sequer produções que especifiquem transições em vazio, detectou-se uma situação de erro, e a cadeia de entrada será rejeitada. Passar então ao passo 5.
2. Uma vez determinada uma produção a ser executada, das formas:

   $$p = (E , \sigma) : \mathcal{A} , \rightarrow E' , \mathcal{B}$$

   ou $p = (\gamma p , E , \sigma a) : \mathcal{A} , \rightarrow (\gamma p' , E' , \sigma' \alpha) , \mathcal{B}$
   passa-se aplicá-la à situação corrente do autômato:

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- Existindo $\mathcal{A}$, executá-la.
- Se da eventual execução de $\mathcal{A}$ resultar a eliminação da própria produção p , voltar ao passo 1.
- Para produções que especifiquem alguma manipulação da pilha,
  - promover o desempilhamento de π , se π estiver explícito.
  - promover o empilhamento de π' , se π' estiver explícito.
- Para produções que especifiquem alterações da cadeia de entrada,
  - promover o consumo de σ , se σ estiver explícito.
  - promover a inserção (empilhamento) de σ' na posição corrente da cadeia de entrada, se σ' estiver explícito na produção.
- Transitar para o estado ε', fazendo de ε' o estado corrente, em lugar de **E** .
- Existindo $\mathcal{B}$, executá-lo.
- Se da aplicação da produção resultar o consumo de um símbolo de entrada, remover da cadeia de entrada tal elemento, posicionando o cursor para apontar o elemento seguinte, que deverá ser consumido por alguma transição a ser posteriormente executada. Esta é uma atividade elementar de extração de elementos da cadeia de entrada, a ser efetuada por um analisador léxico da linguagem, o qual pode ser implementado como submáquina no autômato adaptativo.
- Terminada a execução da produção, passar ao passo 3 para decidir se deve ser efetuada mais alguma transição ou encerrar a operação do autômato.

3. Se foi atingida uma situação final, no qual a cadeia de entrada estiver esgotada, a pilha estiver vazia e o estado atingido for um estado final do autômato adaptativo, encerrar com sucesso a operação do autômato, passando ao passo 6, e aceitando a cadeia de entrada.
   Caso não seja atingida uma situação final, voltar ao passo 1 para efetuar mais uma transição.
4. Detectada uma situação não-determinística, em que mais de uma transição é aplicável à situação corrente, deve-se examinar exaustivamente as diversas possibilidades de transição, adotando aquela(s) que levar(em) à aceitação da cadeia de entrada, e rejeitando as demais. Para isto,
   - memorizar o conjunto de duas ou mais transições aplicáveis
   - para prosseguir a operação do autômato, remover sucessivamente as produções deste conjunto, alimentando com elas chamadas recursivas deste mesmo algoritmo, até que uma das seguintes situações ocorra:
     - haja uma aceitação da cadeia de entrada (o que encerra com sucesso a operação do autômato, desviando para o passo 6)
     - seja esgotado o conjunto das produções aplicáveis (o que também encerra a operação do autômato, porém rejeitando a cadeia e desviando para o passo 5)
5. Foi detectada uma condição de rejeição da transição.
   - Se não houver outras produções não-determinísticas a serem testadas, esgotaram-se as alternativas correntes, havendo dois casos a considerar:
     - Havendo outros conjuntos não-determinísticos ainda não esgotados, retornar da recursão, removendo a próxima produção a testar, e prosseguindo o teste no passo 1.
     - Caso contrário, esgotaram-se todas as possibilidades não-determinísticas previstas. Trata-se de uma condição de erro de sintaxe, e a cadeia de entrada deverá ser rejeitada, desviando para o passo 7.
   - Caso contrário, extrair do conjunto de produções não-determinísticas a próxima produção a testar, restaurar as condições para seu teste e desviar para o passo 1.
6. Houve aceitação da cadeia.
   - Tratando-se de um caso determinístico, desviar para o estado 7, para reportar a aceitação determinística da cadeia.
   - Caso haja transições não-determinísticas ainda não testadas, memorizar o presente reconhecimento para reportá-lo posteriormente, extrair do conjunto de produções não-determinísticas a próxima produção a testar, restaurar as condições para que ela possa ser testada e desviar para o passo 1.
   - Caso todas as transições não-determinísticas tenham sido testadas, memorizar o presente reconhecimento para reportá-lo posteriormente, e desviar para o passo 7.
7. Final de operação do autômato adaptativo.
   - Caso tenha ocorrido erro de sintaxe, ou seja, um insucesso no reconhecimento determinístico ou não-determinístico, reportar o erro e finalizar o procedimento.
   - Caso tenha havido um reconhecimento determinístico bem sucedido, reportá-lo e finalizar o procedimento.
   - Caso tenha havido um reconhecimento não-determinístico bem sucedido, reportar todos os casos de sucesso na aceitação da cadeia e finalizar o procedimento.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

4.4 Exemplos de aplicação do autômato adaptativo

Nesta seção são apresentados, a título de ilustração da aplicabilidade dos autômatos adaptativos, diversos exemplos práticos que se mostram úteis para a resolução exclusivamente sintática de diversos dos problemas típicos mais freqüentemente enfrentados no reconhecimento de linguagens dependentes de contexto.

A combinação adequada das soluções aqui apresentadas permite que, com pequenas complementações de projeto, sejam resolvidos numerosos problemas de dependências de contexto que se apresentam ao se efetuar o reconhecimento das linguagens de programação usuais.

Em geral, a elaboração, para estas linguagens, de reconhecedores sintáticos que incluam não apenas suas versões simplificadas livres de contexto, mas também o tratamento de seus aspectos dependentes de contexto, faz uso freqüente de rotinas semânticas externas, que desta maneira acabam por implementar o tratamento da chamada "semântica estática", que na realidade é uma parte autenticamente sintática destas linguagens.

O uso dos autômatos adaptativos traz, no caso, uma forma alternativa de formalização através da qual os aspectos dependentes de contexto não são excluidos do tratamento sintático da linguagem, mas fazem parte integrante do mesmo.

Nos exemplos que compõem esta seção, os autômatos adaptativos são representados de duas formas: através das produções que os definem, e através das representações gráficas correspondentes, denotadas de acordo com as convenções anteriormente descritas.

Nos exemplos desta seção serão amplamente utilizados os seguintes conjuntos de símbolos:

```
DIGITOS = {
"0" , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" }

LETRAS = {
"a" , "b" , "c" , "d" , "e" , "f" , "g" , "h" , "i" , "j" , "k" , "l" , "m" ,
"n" , "o" , "p" , "q" , "r" , "s" , "t" , "u" , "v" , "w" , "x" , "y" , "z" ,
"A" , "B" , "C" , "D" , "E" , "F" , "G" , "H" , "I" , "J" , "K" , "L" , "M" ,
"N" , "O" , "P" , "Q" , "R" , "S" , "T" , "U" , "V" , "W" , "X" , "Y" , "Z" }

ASCII = LETRAS ∪ DIGITOS ∪ {
"`" , "-" , "=" , "" , "~" , "!" , "@" , "#" , "$" , "%" , "" , "&" , "*" ,
"(" , ")" , "_" , "+" , "|" , "{" , "}" , "[" , "]" , ":" , """ , ";" , "'" ,
"<" , ">" , "?" , "," , "." , "/" , " " }

Σ= { ∇ θ | θ ∈ ASCII ,  ∇ "&" = e-com } ∪ {
id , num , int , real , macro , dec , dmac , atrib , bloco , v.int , v.real }
```

4.4.1 Simulação de uma Pilha

Neste exemplo é apresentado um autômato adaptativo muito simples, capaz de simular em sua estrutura uma pilha, ilustrando a implementação, através de autômatos adaptativos, de um mecanismo muito útil em diversas aplicações como alternativa para o uso de pilhas explícitas.

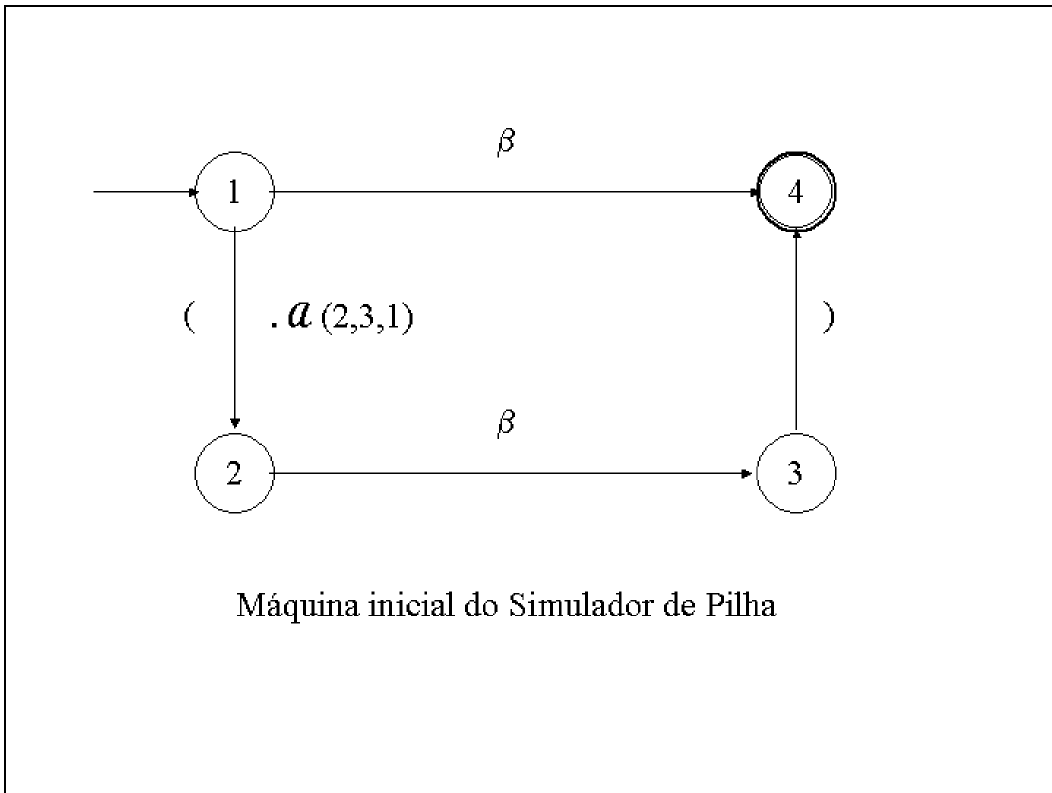
O autômato adaptativo aqui descrito efetua o reconhecimento de uma palíndrome ímpar da forma $(^n\ \beta\ )^n$, de acordo com o seguinte princípio de funcionamento:

- A máquina de estados inicial M0 do autômato adaptativo reconhece a cadeia $(^0\ \beta\ )^0$, ou seja, a cadeia β , incorporando ainda transições para reconhecer $(^1\ \beta\ )^1$, ou seja, a cadeia $(\ \beta\ )$.
- Em cada instante do seu funcionamento, o autômato adaptativo reconhecerá cadeias $(^n\ \beta\ )^n$ com $0 \leq n \leq k+1$, onde **k** é o maior dos valores de **n** assumidos até o momento pelas cadeias já reconhecidas pelo autômato.
- Ao encontrar uma cadeia para a qual $n = z \geq k+1$, uma ação adaptativa ocorre, com a finalidade de ampliar o autômato de modo que passe a reconhecer cadeias com $0 \leq n \leq k+2$, e assim sucessivamente até que o autômato venha a reconhecer cadeias com $0 \leq n \leq z+1$.
- A ação adaptativa que altera a estrutura do autômato deve, a cada ampliação, ser ajustada de forma que cadeias com n < k passem a ser reconhecidas como se o reconhecedor não fosse adaptativo. Isto é feito removendo-a da transição à qual está acoplada, e reinserindo-a, depois de devidamente ajustada à nova situação, como parte da correspondente transição adicionada.

```
/* Simulador de Pilha */

                          Produções iniciais:
```

```
( 1 , "β" ) : → 4
( 2 , "β" ) : → 3
( 3 , "(" ) : → 4
( 1 , "(" ) : → 2 , 𝒜 ( 2 , 3 , 1 )
```



Máquina inicial do Simulador de Pilha

```
                         Função Adaptativa:
𝒜 ( i , j , n ) = { k* , m* :
                    + [ ( k , "β" ) : → m ]
                    + [ ( m , ")" ) : → j ]
                    + [ ( i , "(" ) : → k , 𝒜 ( k , m , i ) ]
                    - [ ( n , "(" ) : → i , 𝒜 ( i , j , n ) ]
                    + [ ( n , "(" ) : → i ] }
```

4.4.2. Coletor de nomes

Neste exemplo é apresentado outro caso simples cujo princípio tem uma vasta aplicabilidade. Trata-se de um coletor de nomes (identificadores), que, partindo de uma situação inicial anterior à coleta de qualquer nome, vai reconhecendo identificadores no texto de entrada, classificando-os como tendo sido previamente encontrados ou não, e alterando a estrutura do autômato que o implementa, de modo que passe a reconhecer como já encontrados previamente cada um dos identificadores recém-coletados.

Com base na técnica ilustrada neste exemplo é possível efetuar, sem o auxílio de tabelas, o tratamento dos nomes das variáveis de uma linguagem com escopo único e sem diversidade de atributos.

Assim, o princípio implementado neste exemplo pode servir como inspiração no projeto de reconhecedores que substituam as tabelas de símbolos por um mecanismo puramente sintático de coleta de identificadores.

O funcionamento deste esquema é o seguinte:

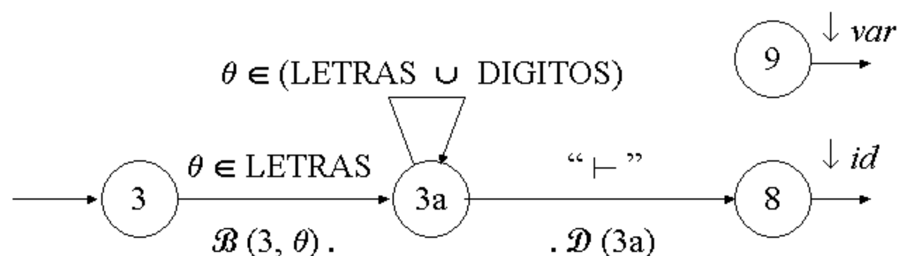
- Inicialmente, o autômato adaptativo reconhece, através de sua máquina de estados inicial, qualquer identificador válido, classificando-o como identificador desconhecido, e efetuando uma auto-modificação estrutural, de tal modo que, ao ser encontrado novamente o mesmo identificador, este não mais seja classificado como desconhecido.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

- Para tanto, é criado um conjunto de transições tal que ao novo identificador seja associada uma trajetória pelos estados do autômato que o levem a um estado final, que aceite o identificador e o caracterize como identificador conhecido.
- Deve-se, naturalmente, eliminar do autômato a antiga trajetória, que aceitava o identificador em questão e cujo percurso proporcionou a execução das ações de ampliação descritas.
- Devem também ser remanejadas as chamadas das funções adaptativas responsáveis por tais operações, de modo que novas ampliações possam ocorrer como conseqüencia do reconhecimento de identificadores com prefixos comuns aos dos identificadores já encontrados anteriormente.
- Em maiores detalhes, a cada novo identificador encontrado é feita no autômato a inserção de um conjunto de transições capazes de reconhecer o novo símbolo (sem deixar de considerar a presença dos demais identificadores previamente incorporados), e é também efetuada a eliminação das transições que implementam o caminho que dá acesso ao reconhecimento de tal identificador no autômato original.
- A distinção entre um nome desconhecido ou não é efetuada quando do término do reconhecimento do identificador, que no caso se dá por estados finais distintos, permitindo que seja retornada para a cadeia de entrada, através de uma operação de empilhamento na cadeia de entrada, uma indicação da classe do identificador encontrado.

```
/* Coletor de Nomes */

                         Produções iniciais:
        { ( 3 , θ ) : ℬ ( 3 , θ ), → 3a    ∀ θ ∈ LETRAS }
        { ( 3a , θ ) : → 3a  ∀ θ ∈ ( LETRAS ∪ DIGITOS ) }
        ( 3a , "⊥" ) : → 8 , 𝒟 ( 3a )
        { (γ z , 8 , α) : → (γ , z , 3id α)  ∀ z ∈ γ }
        { (γ z , 9 , α) : → (γ , z , 3var α)  ∀ z ∈ γ }
```



Máquina inicial do Coletor de Nomes

```
                         Funções adaptativas:
ℬ ( i , σ ) :  { j* :
        + [ ( i , σ ) : → j ]
        { + [ ( j , θ ) : ℬ ( j , θ ), → 3a ]  ∀ θ ∈ ( LETRAS ∪ DIGITOS ) }
        + [ ( j , "⊥" ) : → 8 ; 𝒟 ( j ) ]
        - [ ( i , σ ) : ℬ ( i , σ ), → 3a ]
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
                  }
𝒟  ( i ) :  { :
              - [ ( i , "⊥" ) : → 8 ;  𝒟 ( i ) ]
              + [ ( i , "⊥" ) : → 9 ]
             }
```

4.4.3 Analisador léxico simples

Este exemplo tem como objetivo ilustrar o emprego de autômatos adaptativos em tarefas de extração e classificação de seqüencias especificadas da cadeia de entrada, como é o caso da transdução implementada em atividades de análise léxica.

O exemplo aqui apresentado corresponde a um analisador léxico simplificado, que extrai do texto-fonte e classifica cadeias correspondentes a números inteiros sem sinal, identificadores e algumas palavras reservadas.

Por meio de uma simples adaptação é possível modificar o autômato proposto para incorporar a extração de outros tipos de seqüencias e palavras reservadas adicionais.

O autômato adaptativo do exemplo está preparado para reconhecer e extrair as seguintes seqüencias de símbolos básicos:

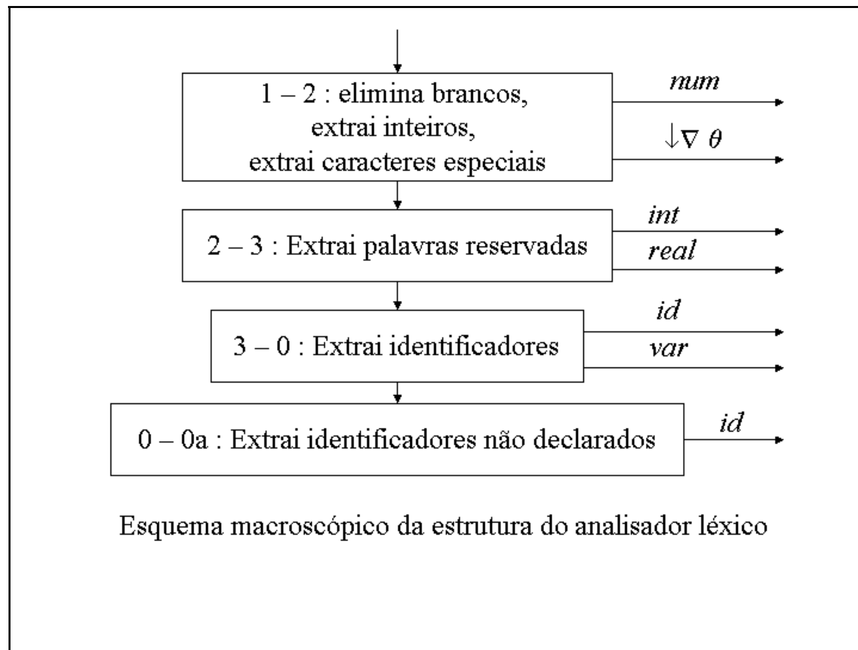
- Identificadores - cadeias compostas de letras e dígitos, iniciadas por letra. O autômato do exemplo reconhece três categorias de identificadores:
  - Palavras reservadas: duas palavras reservadas são identificadas pelo autômato do exemplo: **INT e REAL**. Os caracteres básicos que os compõem são extraídos do texto-fonte, e, uma vez constatado que a seqüencia é exatamente uma das duas palavras reservadas, no texto-fonte é inserido, em seu lugar, o meta-símbolo ***int*** ou ***real***, respectivamente.
  - Identificadores novos: Ao início da operação do autômato adaptativo do exemplo, qualquer seqüencia que forme um identificador, mas que não seja uma das palavras reservadas, é classificada como um identificador novo, sendo substituída pelo meta-símbolo ***id*** na cadeia de entrada. À medida que novos identificadores são extraídos, o autômato adaptativo vai ganhando novos estados e as transições responsáveis por sua extração e reconhecimento, de forma que, nas ocorrências subseqüentes do mesmo identificador, este não mais seja classificado como identificador novo.
  - Identificador já conhecido - Em cada instante de sua operação, o autômato adaptativo é capaz de extrair e de reconhecer identificadores já extraídos anteriormente, classificando-os como identificador de variável, através da inserção do meta-símbolo ***var*** na cadeia de entrada. No exemplo, os identificadores não são classificados de acordo com sua função, mas uma extensão do mecanismo pode propiciar este efeito, conforme apresentado em outros exemplos.
  - Número inteiro sem sinal - Cadeias formadas somente por dígitos. O autômato adaptativo limita-se, neste caso, a extrair a seqüencia de dígitos da cadeia de entrada, e substitui-la por um meta-símbolo ***num*** no texto-fonte.

No autômato deste exemplo, as cadeias extraídas são delimitadas à direita por qualquer símbolo elementar que não esteja previsto como parte da cadeia que está sendo extraída. Como conseqüencia, da cadeia de entrada são sempre extraídas seqüencias de máximo comprimento, apesar de todas as sub-cadeias com mesmos prefixos serem igualmente classificáveis como identificadores.

Neste analisador léxico, símbolos espaçadores (brancos) são ignorados, e símbolos que não iniciam qualquer das cadeias acima descritas, ou seja, símbolos que não sejam brancos, nem letras nem dígitos, são mantidos intactos na cadeia de entrada, não sendo substituidos por meta-símbolos.

Antes de passar ao detalhamento do analisador léxico propriamente dito, apresenta-se, na figura abaixo, um esquema da sua arquitetura macroscópica, com a finalidade de facilitar a compreensão do funcionamento do autômato adaptativo que o implementa. Os números designam os estados que interligam as várias partes deste autômato no detalhamento subseqüente.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor



Esquema macroscópico da estrutura do analisador léxico

- O bloco **1-2** designa a parte do analisador léxico responsável por eliminar brancos, e extrair inteiros e caracteres especiais. Neste bloco distinguem-se os seguintes estados: os estados finais **1b** (que empilha na cadeia de entrada o meta-símbolo ***int***) e **1** (que empilha os meta-símbolos correspondentes aos caracteres especiais ASCII θ - Denota-se este empilhamento por ↓ ∇ θ).
- O bloco **2-3** é responsável pela extração de palavras reservadas, tendo os seguintes estados finais: **2k** (empilha o meta-símbolo ***int***) e **2p** (empilha o meta-símbolo ***real***)
- O bloco **3-0** extrai identificadores previamente encontrados: estados finais **8** (empilha ***id***) e **9** (empilha ***var***)
- O bloco **0-0a** extrai, finalmente, os demais identificadores desconhecidos, no estado final **0a** (empilha ***id***)

/* Analisador léxico simples: eliminação de brancos, extração de números inteiros e de caracteres especiais */

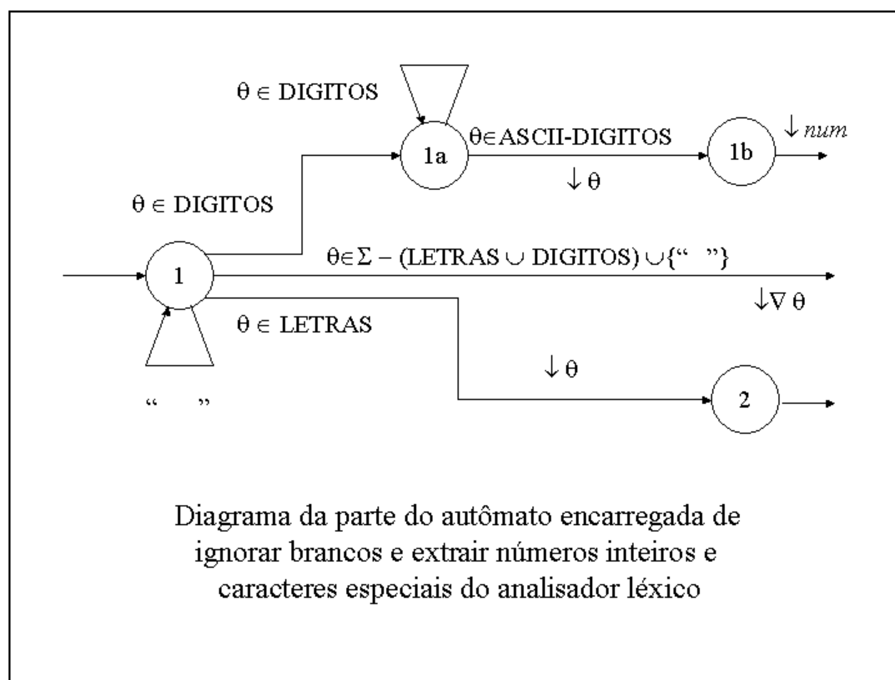


Diagrama da parte do autômato encarregada de ignorar brancos e extrair números inteiros e caracteres especiais do analisador léxico

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
                         Produções iniciais:
( 1 , " " ) : → 1  /* elimina brancos */

/* Extração de inteiros */
{ ( 1 , θ ) : → 1a  ∀ θ ∈ DIGITOS } /* primeiro dígito */
{ ( 1a , θ ) : → 1a  ∀ θ ∈ DIGITOS } /* dígitos seguintes */
{ (γ , 1a , θ a) : → (γ , 1b , θ a)  /* finaliza com não-dígito */
                         ∀ θ ∈ ASCII - DIGITOS  }
{ (γ z , 1b , α) : → (γ , z , num α)   ∀ z ∈ γ  } /* retorna num */

/* extração de caracteres especiais:*/
/* θ = caracter ASCII; ∇ θ = seu código para efeito de transições */
{ (γ z , 1 , θ a) : → (γ , z , ∇ θ a)
        ∀ θ ∈ σ - (LETRAS ∪ DIGITOS ∪ { " " })  }

/* átomo começa com letra: vai verificar se é palavra reservada */
{ (γ , 1 , θ a) : → (γ , 2 , θ a)  ∀ θ ∈ LETRAS }
```

```
/* Extração de palavras reservadas */
```

Notação: No diagrama seguinte, o meta-símbolo λ denota qualquer caracter ASCII que não seja consumido por transições que partem do estado-origem da transição rotulada com λ. Estas não consomem o caracter ASCII em questão, retornando-o simplesmente à cadeia de entrada para ser reutilizado.

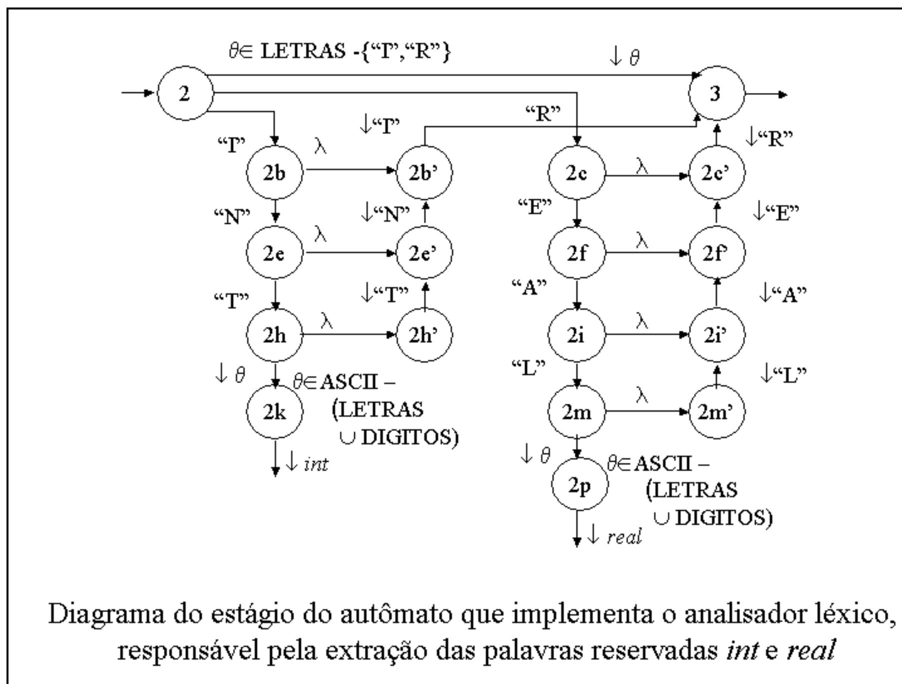


Diagrama do estágio do autômato que implementa o analisador léxico, responsável pela extração das palavras reservadas *int* e *real*

```
/* Rejeição: palavra não começa por  I ou R */
{ (γ , 2 , θ a) : → (γ , 3 , θ a)  ∀ θ ∈ LETRAS - { I , R } }
/* Extração da palavra reservada INT */
( 2 , "I" ) : → 2b
( 2b , "N" ) : → 2e
( 2e , "T" ) : → 2h
{ (γ , 2h , θ a) : → (γ , 2k , θ a)
                    ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS ) }
{ (γ z , 2k , α) : →  (γ , z , int α)  ∀ z ∈ γ }

/* Extração da palavra reservada REAL */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
( 2  , "R" ) : → 2c
( 2c , "E" ) : → 2f
( 2f , "A" ) : → 2i
( 2i , "L" ) : → 2m
{ (γ , 2m , θ a) : → (γ , 2p , θ a)
                ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS) }
{ (γ z , 2p , α) : →  (γ , z , real α)  ∀ z ∈ γ }

/* Recuperação: rejeição de seqüencia parcial de INT */
{ (γ , 2b , θ a) : → (γ , 2b' , θ a)
                ∀ θ ∈ ASCII - { "I" }  }
(γ , 2b' , α) : → (γ , 3 , "I" α)
{ (γ , 2e , θ a) : → (γ , 2e' , θ a)
                ∀ θ ∈ ASCII - { "N" }  }
(γ , 2e' , α) : → (γ , 2b' , "N" α)
{ (γ , 2h , θ a) : → (γ , 2h' , θ a)
                ∀ θ ∈ ASCII - { "T" }  }
(γ , 2h' , α) : → (γ , 2e' , "T" α)

/* Recuperação: rejeição de seqüencia parcial de REAL */
{ (γ , 2c , θ a) : → (γ , 2c' , θ a)
                ∀ θ ∈ ASCII - { "R" }  }
(γ , 2c' , α) : → (γ , 3 , "R" α)
{ (γ , 2f , θ a) : → (γ , 2f' , θ a)
                ∀ θ ∈ ASCII - { "E" }  }
(γ , 2f' , α) : → (γ , 2c' , "E" α)
{ (γ , 2i , θ a) : → (γ , 2i' , θ a)
                ∀ θ ∈ ASCII - { "A" }  }
(γ , 2i' , α) : → (γ , 2f' , "A" α)
{ (γ , 2m , θ a) : → (γ , 2m' , θ a)
                ∀ θ ∈ ASCII - { "L" }  }
(γ , 2m' , α) : → (γ , 2i' , "L" α)

/* Não sendo palavra reservada, a seqüencia encontrada foi
restaurada na cadeia de entrada para ser reanalisada como
identificador. */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

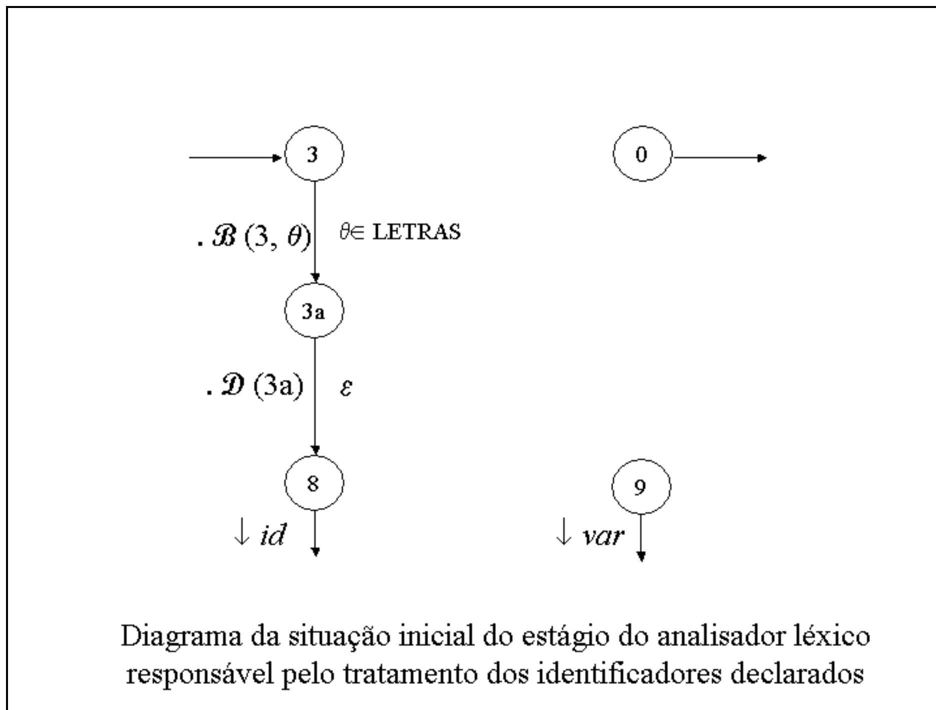


Diagrama da situação inicial do estágio do analisador léxico responsável pelo tratamento dos identificadores declarados

```
{ ( 3 , θ ) : ℬ ( 3 , θ ) , → 3a   ∀ θ ∈ LETRAS }
(γ , 3a , θ a) : → (γ , 8 , θ a) , 𝒟 ( 3a )
```

/* Retorno com empilhamento de id (identificador não declarado anteriormente). O acesso ao estado 8 é feito por transições a serem criadas pelas ações adaptativas do autômato. */

```
{ (γ k , 8 , α) : → (γ , k , id α)  ∀ k ∈ γ }
```

/* Retorno com empilhamento de var (identificador declarado anteriormente). O acesso ao estado 9 é feito por transições a serem criadas pelas ações adaptativas do autômato. */

```
{ (γ k , 9 , α) : → (γ , k , var α)  ∀ k ∈ γ }
```

/* Tratamento de identificadores não declarados */

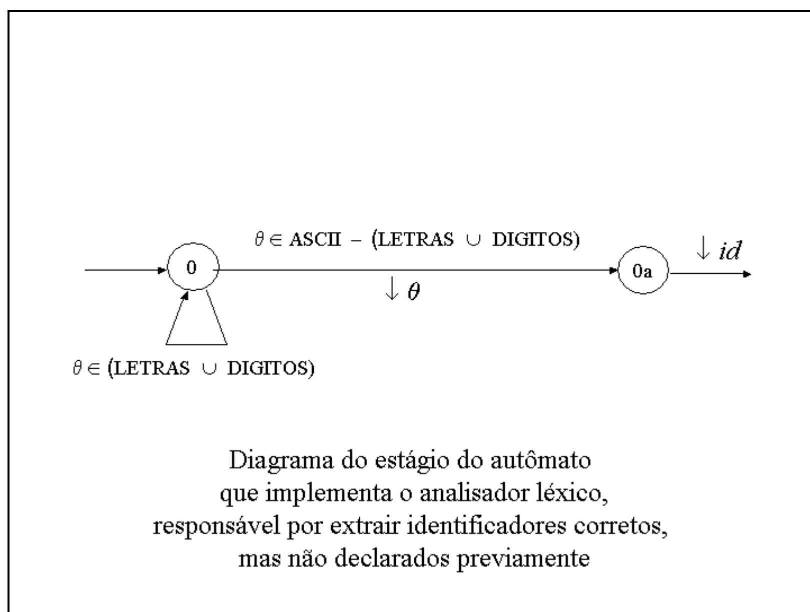


Diagrama do estágio do autômato que implementa o analisador léxico, responsável por extrair identificadores corretos, mas não declarados previamente

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
/* Caso o identificador não seja reconhecido, a parte seguinte  do
autômato promove a  sua  extração  incondicional,  classificando-o
como  id  - identificador desconhecido. */
{ ( 0 , θ ) : → 0  ∀ θ ∈ LETRAS ∪ DIGITOS }
{ (γ , 0 , θ a) : → (γ , 0a , θ a)
                 ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS) }
{ (γ z , 0a , α) : → (γ , z , id α)  ∀ z ∈ γ }
```

```
                         Funções Adaptativas:

ℬ ( i , x ) :  { j* :
        + [ ( i , x ) : → j ]
        { + [ ( j , θ ) : ℬ ( j , θ ) , → 3a ]
                              ∀ θ ∈ ( LETRAS ∪ DIGITOS ) }
        { + [ (γ , j , θ a) : → (γ , 8 , θ a) , 𝒟 ( j ) ]
                                ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS )   }
        - [ ( i , x ) : ℬ ( i , x ) , → 3a ]
                {
𝒟  ( i ) :  { :
              { - [ (γ , i , θ a) : → (γ , 8 , θ a) ,  𝒟 ( i ) ]
                                ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS )   }
              { + [ (γ , i , θa) : → (γ , 9 , θ a) ]
                                ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS )   }
             }
```

4.4.4 Declarações de identificadores com tipos associados - caso 1

Neste exemplo, é mostrado um mecanismo adaptativo aplicável à solução de problemas relacionados à associação de atributos já conhecidos a identificadores que estão sendo declarados.

Este estudo ilustra a aplicabilidade dos autômatos adaptativos à resolução sintática de diversos dos problemas, em geral resolvidos semanticamente, ligados à declaração de identificadores de variáveis, matrizes e estruturas de dados, que formam uma classe de atividades da maior importância prática na elaboração de processadores de linguagens de programação, o da coleta de símbolos, de sua memorização e classificação, segundo um atributo qualificativo conhecido na ocasião da coleta.

A solução usual deste tipo de problemas baseia-se no uso de procedimentos semânticos de coleta dos símbolos encontrados, e da correspondente classificação por meio da memorização prévia dos atributos, para sua posterior inclusão em tabelas de símbolos e de atributos. O acionamento de tais atividades é freqüentemente da responsabilidade dos procedimentos de análise léxica.

Para executar uma atividade similar, o autômato adaptativo efetua, em sua própria estrutura, uma memorização do atributo a ser associado aos identificadores, informação esta que utiliza posteriormente para caracterizar tais identificadores, assim que forem extraídos do texto-fonte por um mecanismo semelhante ao utilizado no exemplo anterior.

No caso deste exemplo, o atributo refere-se ao tipo associado às variáveis declaradas, que pode ser INTEGER, REAL, e que são indicados através da presença da palavra-chave correspondente antes da lista de variáveis a que se refere.

Assim, encontrada a palavra-chave, é ativada, com um argumento adequado, uma função adaptativa encarregada de criar uma transição que aponta, a partir de um estado convencionado do autômato, o estado final ao qual está associado o atributo.

Uma vez coletado o identificador, conforme o princípio ilustrado anteriormente, cria-se uma transição em vazio partindo do estado atingido após o consumo do identificador, apontando o estado final anteriormente memorizado, ao qual está associado o atributo desejado.

```
/* Declarações de Identificadores - Caso 1 */

                           Produções iniciais:
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
( d1 , int ) : → d2 , 𝒞 ( 20 )
( d1 , real ) : → d2 , 𝒞 ( 17 )
( d2 , id ) : → d3
( d3 , , ) : → d2
{ (γ , d3 , θ a) : → (γ , d4 , θ a) , 𝒞 ( 8 )
                          ∀ θ ∈ ASCII - { "," } }
{ (γ k , d4 , α) : → (γ , k , dec α)  ∀ k ∈ γ }
```

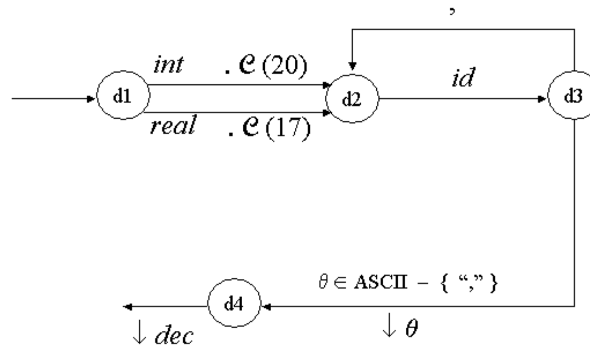


Diagrama da máquina de estados inicial da parte do autômato adaptativo responsável pelo tratamento das declarações de variáveis – primeiro caso

O analisador léxico adequado para uso neste caso é muito semelhante ao empregado no exemplo 4.4.3, dele diferindo apenas nos seguintes pontos:

- dois estados adicionais, 17 e 20, responsáveis por permitir a distinção entre os tipos inteiro e real para as variáveis, através de transições que empilham v.int ou v.real, em contraste com o tipo único **(var)**, existente no exemplo anterior.
- uma função adaptativa adicional 𝒞 , encarregada de memorizar no estado especial 50 a informação sobre o tipo a ser associado aos identificadores encontrados
- uma modificação na função adaptativa 𝒟 , de forma que a informação coletada pela função 𝒞 seja utilizada para identificar o tipo das variáveis declaradas.

```
/* Retorno com empilhamento de v.real (identificador de variável
real). O acesso ao estado 17 é feito por transições a serem
criadas pelas ações adaptativas do autômato. */
{ (γ k , 17 , α) : → (γ , k , v.real α)  ∀ k ∈ γ }

/* Retorno com empilhamento de v.int (identificador de variável
inteira). O acesso ao estado 20 é feito por transições a serem
criadas pelas ações adaptativas do autômato. */
{ (γ k , 20 , α) : → (γ , k , v.int α)  ∀ k ∈ γ }


                    Funções Adaptativas:
ℬ ( i , σ ) : idêntica à homônima do exemplo 4.4.3
𝒞 ( t ) :  { x :
/* memoriza, apontado pelo estado especial 50, o tipo default para
todas as variáveis definidas nesta declaração */
           - [ ( 50 , ε ) : → x ]
           + [ ( 50 , ε ) : → t ]
        }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
𝒟  ( i ) :  { t :
/* A menos da consulta ao estado  50,  e  uso  do  tipo-default  t
encontrado no lugar do indicador 9 (var), é igual  à  homônima  do
exemplo 4.4.3 */
             ? [ ( 50 , ε ) : → t ]
             { - [ (γ , i , θ a) : → (γ , 8 , θ a) ;  𝒟 ( i ) ]
                              ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS)   }
             { + [ (γ , i , θ a) : → (γ , t , θ a) ]
                              ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS)   }
            }
```

4.4.5 Declarações de identificadores com tipos associados - caso 2

Neste exemplo mostra-se uma possível forma de utilização do autômato adaptativo para efetuar um tratamento puramente sintático da associação de atributos a identificadores, no caso de tais atributos não serem conhecidos na ocasião da primeira aparição dos identificadores declarados.

A solução clássica destes problemas é efetuada, como no caso anterior, com o auxílio de procedimentos semânticos, que são acionados pelas rotinas de análise léxica, e que se encarregam de tabelar os identificadores encontrados, mantendo indefinidos os respectivos atributos até que estes venham a ser conhecidos, ocasião em que todos os identificadores envolvidos são a eles associados.

O exemplo ilustra a realização de uma solução adaptativa que é aplicável a diversas situações, muito freqüentes no tratamento de linguagens de programação, e usualmente efetuada com a ajuda de procedimentos semânticos.

É o caso, por exemplo, das referências à frente em comandos de desvio, da utilização de objetos antes de sua declaração, e da associação de atributos a objetos após a declaração de seus nomes, ou mesmo da utilização de operadores, procedimentos, módulos e funções a serem posteriormente declarados.

O princípio do funcionamento do autômato adaptativo deste exemplo baseia-se nos mesmos argumentos utilizados no exemplo anterior, com a diferença de que, não dispondo de informações sobre os atributos a associar ao identificador, torna-se necessário marcar os respectivos atributos como pendentes, o que é feito por meio de um estado especialmente designado para tal finalidade, para o qual o estado que finaliza o reconhecimento do identificador se mantém apontando por meio de uma transição em vazio, até que o atributo venha a ser conhecido. Nesta ocasião, removem-se tais transições em vazio, substituindo-as por outras, que apontem estados associados ao atributo desejado.
Notar que os estados 17 e 20 deste exemplo têm a mesma conotação dos homônimos do exemplo 4.4.4.

```
/* Declarações de Identificadores - Caso 2 */

                        Produções iniciais:
           ( e1 , decl ) : → e2 , 𝒞 ( 29 )
           ( e2 , id ) : → e3
           ( e3 , , ) : → e2
           ( e3 , int ) : →  e4 , ℰ ( 20 )
           ( e3 , real ) : →  e4 , ℰ ( 17 )
           { (γ , e4 , θ a) : → (γ , e5 , θ a)  ∀ θ ∈ ASCII }
           { (γ k , e5 , α) : → (γ , k , dec α)  ∀ k ∈ γ }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

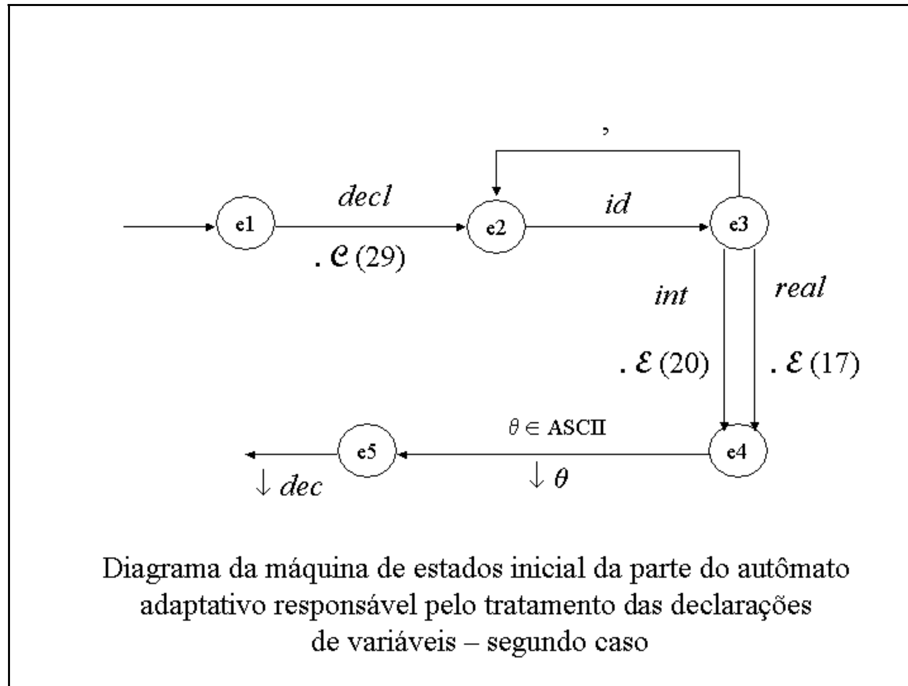


Diagrama da máquina de estados inicial da parte do autômato adaptativo responsável pelo tratamento das declarações de variáveis – segundo caso

```
                         Funções Adaptativas:
ℬ ( i , x ) : idêntica à homônima do exemplo 4.4.3.
𝒞 ( t ) :  idem
𝒟 ( i ) :  idem
ℰ ( t ) :  { i , σ :
/* Acerta o tipo para cada um dos  identificadores  marcados  como
estando com tipo indefinido (indicado pelo estado especial 29) */
             - [ ( i , σ ) : → 29 ]
             + [ ( i , σ ) : → t ]
             ∀  i , σ  }
```

4.4.6 Verificação de tipos

Neste exemplo é mostrada uma aplicação trivial do autômato adaptativo à resolução de um problema constante na elaboração de processadores de linguagens de programação.

Trata-se da verificação de tipos (type-checking), atividade que sintetiza tipicamente a maioria dos testes de coerência entre a declaração e a utilização de objetos em uma linguagem de programação.

O exemplo aqui apresentado apenas oferece uma das múltiplas possíveis soluções para situações muito freqüentes, tais como a garantia da coerência entre o tipo declarado para o objeto representado por um identificador e aquele exigido pelos diversos contextos em que tal identificador é utilizado.

A verificação é feita, no caso, mediante a execução de ações adaptativas que alteram a estrutura da máquina de estados que implementa o autômato adaptativo, de tal forma que sejam aceitas apenas as construções sintáticas em que forem obedecidos os vínculos corretos entre os tipos declarado e esperado para o identificador.

O exemplo ilustra o caso em que são disponíveis dois tipos de variáveis, e correspondentemente, dois tipos de comandos de atribuição, envolvendo respectivamente inteiros e reais.

Os comandos de atribuição em questão compõem-se de duas partes: uma variável-destino e uma expressão aritmética, extremamente simplificada por razões didáticas, envolvendo somente números e variáveis, e operadores aditivos apenas (simbolizando qualquer operador aritmético), sem outros recursos.

Encontrado o identificador que representa a variável-destino da atribuição, o tipo a ele associado é utilizado como referência por uma ação adaptativa correspondente, a qual altera a estrutura da máquina de estados que reconhece a expressão cujo resultado deve preeencher a variável-destino, de forma que apenas expressões compatíveis sejam aceitas.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Para isto, ao ser executada a função adaptativa 𝓕, com parâmetro v.int ou v.real, é inserida no autômato uma transição que consome identificadores, classificados como tais pelo analisador léxico, de forma que as expressões passem a ser compostas apenas por elementos do mesmo tipo da variável-destino da atribuição.

Ao final do reconhecimento do comando de atribuição, esta transição acrescentada é removida por outra ação adaptativa, restaurando as condições iniciais da máquina de estados.

O analisador léxico empregado é o mesmo apresentado anteriormente para uso com declarações de identificadores tipados.

O autômato completo consta de uma submáquina para análise léxica, outra para declarações, outra para comandos de atribuição e outra para a estrutura completa do programa, formada apenas por uma seqüencia de declarações seguida de uma seqüencia de atribuições.

```
/* Comando de Atribuição */

                         Produções iniciais:
( a1 , v.int ) : → a2 , 𝓕 ( v.int )
( a1 , v.real ) : → a2 , 𝓕 ( v.real )
( a2 , = ) : → a3
( a3 , num ) : → a4
( a4 , + ) : → a3
{ (γ , a4 , θ a) : → (γ , a5 , θ a)   ∀ θ ∈ σ }
{ (γ k , a5 , α) : → (γ , k , atrib α)   ∀ k ∈ γ }
```

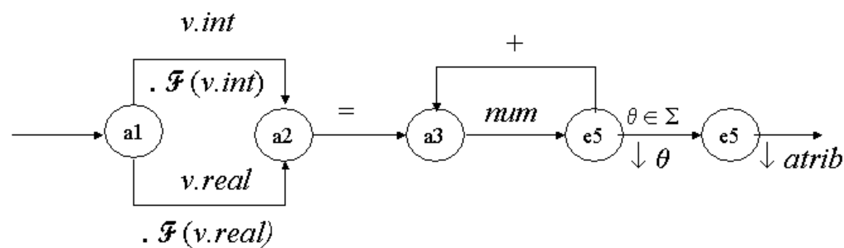


Diagrama da máquina de estados inicial da parte do autômato adaptativo responsável pelo tratamento de comandos de atribuição

```
                        Funções Adaptativas:
𝓕 ( x ) : { :
/* cria uma transição a3→a4 consumindo v.int ou v.real, conforme o
tipo da variável encontrada na transição a1→a2 */
            + [ ( a3 , x ) : → a4 ]
            { - [ (γ , a4 , θ a) : → (γ , a5 , θ a)   ∀ θ ∈ σ }
            { + [ (γ , a4 , θ a) : 𝓖 ( x ) , → (γ , a5 , θ a)
                                    ∀ θ ∈ σ }
          }

𝓖 ( x ) : { :
/* apaga a transição a3→a4 criada pela função adaptativa 𝓕 */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
          - [ ( a3 , x ) : → a4 ]
          { - [ (γ , a4 , θ a) : 𝒢 ( x ) , → (γ , a5 , θ a)
                                   ∀ θ ∈ σ }
          { + [ (γ , a4 , θ a) : → (γ , a5 , θ a)  ∀ θ ∈ σ }
        }
```

```
/* programa */
```

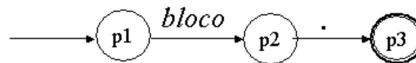


Diagrama da máquina de estados inicial de tratamento de um programa completo

```
                    Produções iniciais:
( p1 , bloco ) : → p2
( p2 ,   .   ) : → p3
```

O funcionamento desta máquina de estados é trivial: chama inicialmente a sub-máquina bloco , e após retornar desta sub-máquina, consome um sinal `"."` (ponto) e encerra seu processamento no estado final **p3** do autômato.

Não há funções adaptativas para esta parte do autômato.

4.4.7 Analisador léxico para uma linguagem estruturada em blocos

Este exemplo, mais complexo, ilustra o tratamento de identificadores por um autômato destinado a realizar atividades de análise léxica para uma linguagem estruturada em blocos.

Neste estudo, o analisador léxico incorpora, além das funções desempenhadas pelos casos mais simples vistos nos exemplos anteriores, algumas características adicionais, que o aproximam mais dos analisadores léxicos usados em compiladores reais.

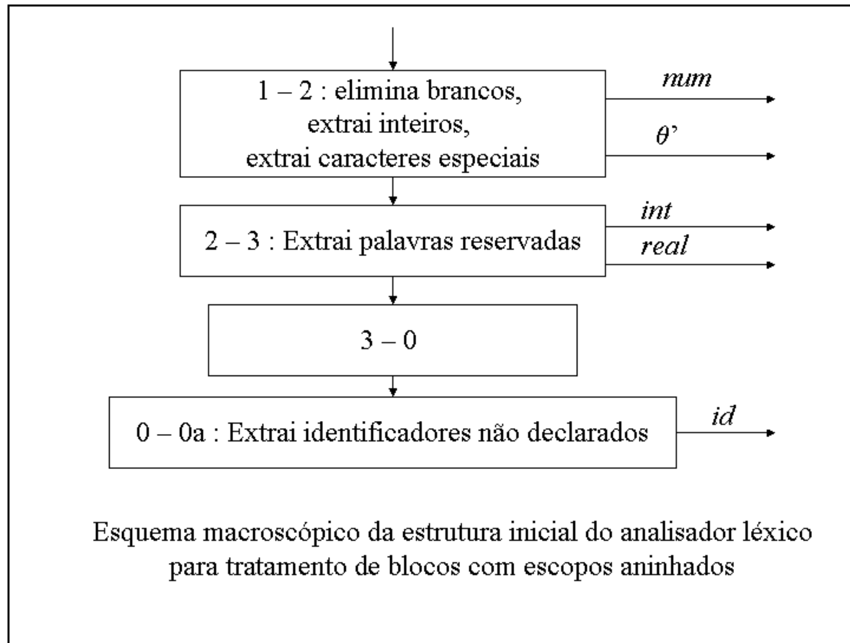
A primeira destas extensões consiste na adoção de dois modos de operação para o analisador: o primeiro modo, correspondente à fase em que identificadores são coletados e memorizados a partir de suas declarações, e o segundo, quando os identificadores são pesquisados, verificando-se a sua situação de definição ou não.

A coerência entre a declaração e o uso dos identificadores em comandos é verificada analogamente ao que foi feito em exemplos anteriores, iniciando-se a busca entre as palavras reservadas.

Para garantir o perfeito funcionamento do analisador léxico, é necessário implementar um esquema em que, durante a fase de declaração das variáveis, todo identificador ainda desconhecido deve ser acrescentado ao conjunto dos identificadores declarados no bloco corrente, enquanto durante a sua fase de utilização, ao encontrar um identificador não pertencente ao bloco corrente, devolve-o à cadeia de entrada, e efetua a busca do identificador nos sucessivos blocos, partindo-se do bloco corrente em direção a eventuais blocos ancestrais, enquanto existirem.

Isto é feito com a ajuda de uma chave, que é acionada quando da abertura de um bloco, e na ocasião em que todas as declarações já tiverem sido processadas dentro do bloco.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor



Esquema macroscópico da estrutura inicial do analisador léxico para tratamento de blocos com escopos aninhados

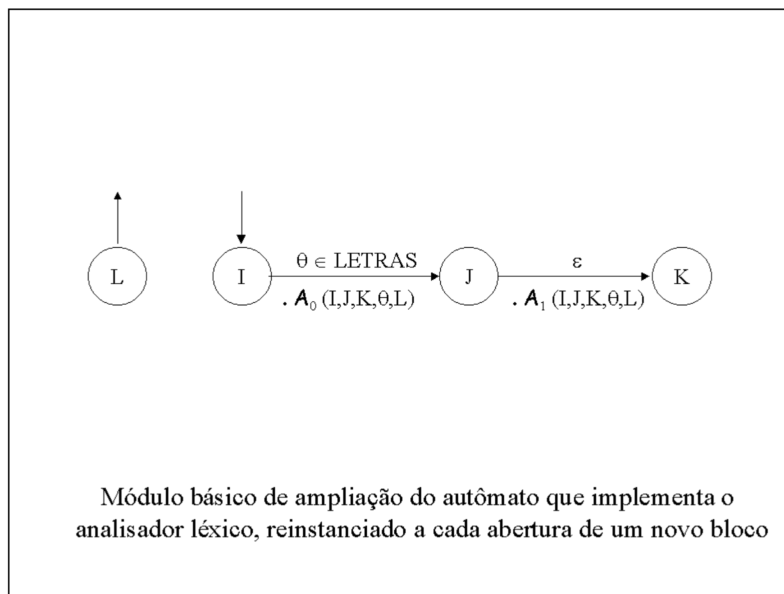
Inicialmente a estrutura deste analisador léxico é muito similar à daquela mostrada no exemplo 4.4.3, dela diferindo apenas quanto à presença de uma transição em vazio em lugar do antigo bloco 3-0, responsável pelo tratamento de identificadores, e que aqui é omitido, indicando que na ocasião não há blocos abertos.

Cada vez que um novo bloco que compõe o programa vai sendo aberto, insere-se entre o estado 3 e seu sucessor corrente (inicialmente o estado 0) um módulo responsável pelo tratamento de todos os identificadores a serem encontrados no bloco em estudo.

O módulo em questão é reinstanciado para cada novo bloco, com a estrutura mostrada na figura seguinte, apresentando um estado I de entrada, à qual o estado 3 se ligará através de uma transição em vazio, e um estado J de saída, que será ligado, também em vazio, ao antigo sucessor do estado 3.

Entre os estados I e J , e depois entre os estados J e K , o autômato se encarrega de, a partir da letra (σ) encontrada na cadeia de entrada, ampliar-se, alterando sua topologia de forma que possa passar a aceitar tal letra como inicial do identificador encontrado.

As ampliações efetuadas no autômato também operam de forma similar em relação aos demais caracteres que compõem o identificador. Estas operações são efetuadas com o auxílio das funções adaptativas $\mathcal{A}0$ e $\mathcal{A}1$, detalhadas mais adiante.



Módulo básico de ampliação do autômato que implementa o analisador léxico, reinstanciado a cada abertura de um novo bloco

Na ocasião em que um bloco é encerrado, torna-se necessário reverter a situação, desacoplando o módulo anteriormente inserido, e refazendo as conexões que existiam antes da inserção de tal módulo.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Para que seja possível utilizar adequadamente as informações contidas na topologia do autômato assim construido, o analisador léxico deve encarregar-se de duas tarefas importantes: a coleta de novos símbolos a partir das declarações, e a localização do escopo a que pertence um símbolo encontrado no texto de entrada.

Com este objetivo, as ações adaptativas da parte do autômato encarregada do tratamento da estrutura dos blocos configura o analisador léxico para operar em um dos dois modos seguintes.

O primeiro modo de operação do analisador léxico permite a declaração de identificadores no bloco corrente, rejeitando duplicações internas ao bloco e permitindo redeclarações de identificadores declarados em outros blocos.

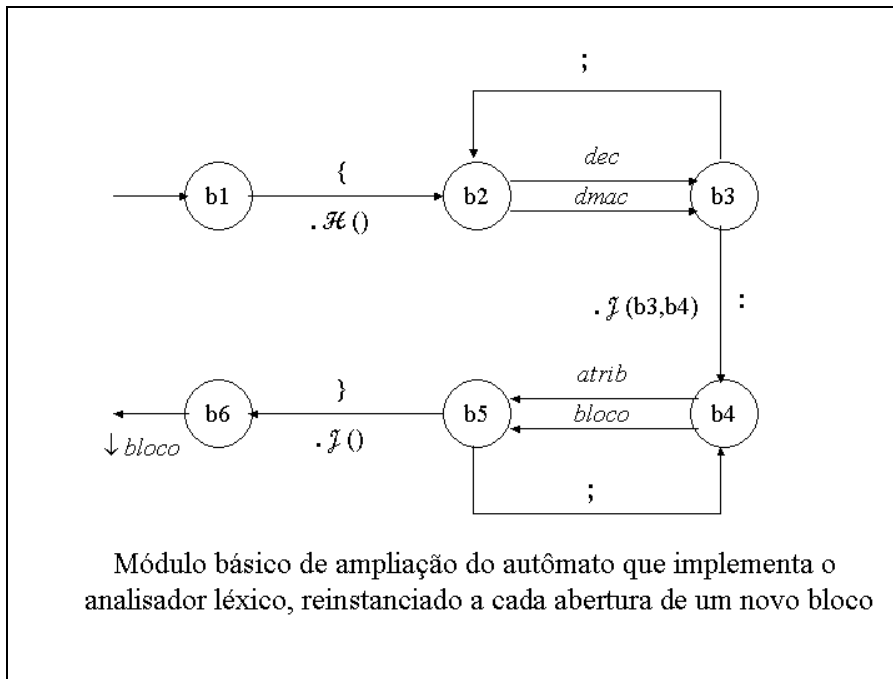
No segundo modo de operação, não são aceitas novas declarações, mas os identificadores encontrados no texto de entrada são pesquisados sucessivamente nos diversos módulos associados aos blocos que estiverem abertos na ocasião, a partir do bloco mais recente.

Detalha-se a seguir um autômato adaptativo que implementa um analisador léxico capaz de tratar, da forma acima descrita, estruturas de blocos com escopos aninhados para as variáveis.

Observe-se a presença do não-terminal dmac em uma das transições do autômato. Trata-se de indicação da ocorrência de uma declaração de macro, conforme detalhado no exemplo 4.4.8.

```
/* Estrutura de Blocos com Escopos Aninhados */
                              Produções iniciais:
( b1 , { ) : → b2 , 𝓗 ( )  /* abertura de bloco: inicia escopo */
/* encontrando a palavra reservada MACRO, chama decl.de macro */
(γ , b2 , macro α) : → (γ b2 , 20 , macro α)
(γ , b2 , α) : → (γ b2 , d1 , α) /* se não, chama declarações */
( b2 , dmac ) : → b3  /* achou uma declaração correta de macro */
( b2 , dec ) : → b3  /* achou declaração correta de variável */
( b3 , ; ) : → b2  /* separador: volta a buscar nova decl. */
/* acha ":" - fim das declarações, início do uso das variáveis */
( b3 , : ) : → b4 , 𝓘 ( b3 , b4 )
/* variáveis inteiras ou reais iniciam um comando de atribuição */
(γ , b4 , v.int α) : → (γ b4 , a1 , v.int α) /*chama atribuição */
(γ , b4 , v.real α) : → (γ b4 , a1 , v.real α) /* chama atrib. */
( b4 , atrib ) : → b5 /* encontrou atribuição correta */
/* chaves denotam início de bloco aninhado */
(γ , b4 , { α) : → (γ b4 , b1 , { α)  /* chama bloco */
( b4 , bloco ) : → b5  /* encontrou bloco correto */
( b5 , } ) : → b6 , ℬ (  )  /* chaves indicam final de escopo */
{ (γ z , b6 , α) : → (γ , z , bloco α)  ∀ z ∈ γ  }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor



Módulo básico de ampliação do autômato que implementa o analisador léxico, reinstanciado a cada abertura de um novo bloco

```
                         Funções Adaptativas:

ℋ ( ) =
      { a , b , c , d , e , r , s , t , u , v , w , x , y , z ,
         i* , j* , k* , m* , n* :
/* Tratamento da abertura de um novo bloco e do  novo  escopo  por
ele definido para as variáveis declaradas */

/* inicializações: */
- [ ( 50 , ε ) : → v ]  /* apaga antigo tipo default */
+ [ ( 50 , ε ) : → 9 ]  /* novo default = 9(variável declarada) */
- [ ( 97 , ε ) : → w ]  /* apaga antigo indicador de uso/declar.*/
+ [ ( 97 , ε ) : → 98 ] /* novo indicador = 98 (declarações) */

/*  inicialização  da  topologia  do  autômato  de  extração  dos
parâmetros formais de macros,  configurando-o  para  permitir  que
sejam declarados novos parâmetros */
-  [ ( 1f , ε ) : → 1g , 𝒜39 ( a , b , c , d , e ) ]
+  [ ( 1f , ε ) : → 1g , 𝒜1 ( 1e , 1f , 1g , ε , 1h ) ]

/* Acerto dos ponteiros da cadeia de blocos para incluir  o  bloco
corrente: insere na cadeia de segmentos do autômato,  responsáveis
pela memorização dos identificadores declarados  em  cada  escopo,
mais um segmento, encarregado de memorizar  os  identificadores  a
serem declarados no bloco que está sendo iniciado */

/* x = estado esquerdo do bloco ancestral */
/* y = estado direito do bloco ancestral */
/* i = estado esquerdo do novo bloco */
/* n = estado direito do novo bloco */
/* 3 = aponta sempre o estado esquerdo do bloco mais recente */
/* 96 = aponta sempre o estado direito do bloco mais recente */
/* e é apontado por uma lista de estados direitos dos ancestrais*/

- [ ( 3 , ε ) : → x ]  /* desfaz conex. esq. do bloco ancestral */
+ [ ( 3 , ε ) : → i ]  /* liga ao estado esq. do novo bloco */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
+ [ ( n , ε ) : → x ]  /* conecta estado dir. ao esq. ancestral */
- [ ( 96 , ε ) : → y ]  /* y deixa de ser est. dir. mais recente */
+ [ ( 96 , ε ) : → n ]  /* n assume seu lugar */


/* manutenção da pilha de ponteiros  para  o  estado  direito  dos
blocos abertos: os ponteiros são memorizados  como  argumentos  da
função 𝒵 , e a pilha é implementada como uma cadeia de transições,
terminada no estado 96. */
/* u deixa de ser o  ponteiro  para  o  estado  direito  do  bloco
ancestral */
- [ ( t , ε ) : → 96 , 𝒵 ( u ) ]
/* e é memorizado na pilha, e */
+ [ ( t , ε ) : → m , 𝒵 ( u ) ]
/* y toma o seu lugar, como estado direito do novo ancestral: */
+ [ ( m , ε ) : → 96 , 𝒵 ( y ) ]

/* inicializa as  condições  para  a  ativação  do  modo  uso  dos
identificadores. Os estados j e k são gerados e  memorizados  como
argumentos da rotina 𝒯  ,  que  é  executada  na  ocasião  em  que
terminarem as declarações e se iniciarem os  comandos  executáveis
do bloco corrente.*/
- [ ( b3 ,  :  ) : → b4 , 𝒯 ( r , s ) ]  /* subst. os arg. de 𝒯 */
+ [ ( b3 ,  :  ) : → b4 , 𝒯 ( j , k ) ]  /* na transição b3→b4 */

/* Criação do autômato responsável pela memorização  e  tratamento
dos identificadores a serem declarados no  bloco  que  está  sendo
iniciado */

{ + [ ( i , θ ) : → j , 𝒜0 ( i , j , k , θ , n ) ]
           ∀ θ ∈ LETRAS  }
 + [ ( j , ε ) : → k , 𝒜1 ( i , j , k , ε , n ) ]

} /* fim da rotina ℋ */

𝒯 ( j , k ) =
{ a , b , c , d , e , i , n , x , y , σ :
/* Término das declarações: O analisador léxico é posicionado para
operar em modo uso de identificadores já declarados */

/* inicializações: */
- [ ( 50 , ε ) : → x ]  /* apaga antigo tipo default */
+ [ ( 50 , ε ) : → 8 ]  /* novo default = 8 (identif. desconhec.)*/
- [ ( 97 , ε ) : → y ]  /* apaga antigo indicador de uso/declar.*/
+ [ ( 97 , ε ) : → 99 ] /* novo indicador = 99 (uso dos identif.)*/

/* Configura o autômato  de  extração  de  parâmetros  formais  de
macros  para  apenas  usar  os  identificadores  já  existentes  e
rejeitar os eventuais novos identificadores encontrados */
-  [ ( 1f , ε ) : → 1g , 𝒜1 ( a , b , c , d , e ) ]
+  [ ( 1f , ε ) : → 1g , 𝒜39 ( 1e , 1f , 1g , ε , 1h ) ]

/* Mudança do modo de operação do analisador léxico: de declaração
para uso das variáveis já declaradas */
-  [ ( j , ε ) : → k , 𝒜1 ( i , j , k , σ , n ) ]
+  [ ( j , ε ) : → k , 𝒜39 ( i , j , k , σ , n ) ]
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
} /* fim da rotina ℐ */

ℐ ( ) =
{ m , n , t , u , x , y , z :
/* Tratamento de final de bloco, encerrando o escopo definido pelo
bloco que acaba de ser finalizado */

- [ ( 3 , ε ) : → x ]  /* desfaz conex. esq. do bloco corrente */
- [ ( 96 , ε ) : → y ]  /* desliga pont.est.dir.bloco corrente */
- [ ( y , ε ) : → z ]  /* desliga bloco corrente do ancestral */
+ [ ( 3 , ε ) : → z ]  /* conecta estado esq. do bloco ancestral*/
- [ ( m , ε ) : → 96 , 𝒵 ( n ) ]  /*recup.est.dir. ancestral  n */
+ [ ( 96 , ε ) : → n ]  /* restaura pont. est. dir. ancestral */

/* restaura como bloco ancestral o ancestral do ancestral do bloco
corrente */
- [ ( t , ε ) : → m , 𝒵 ( u ) ] /* elimina ancestral corrente */
+ [ ( t , ε ) : → 96 , 𝒵 ( u ) ] /* aflorando seu ancestral */

}  /* fim da rotina ℐ */

𝒜0 ( 7j , 7ja , 7jb , σ , 7j1 ) =
  { m , n , p , q , r , s , t , u , v , w , x , y :
/* Esta função destina-se a inserir os parâmetros corretos,
idênticos aos seus próprios argumentos, nas chamadas da função 𝒜1
ou 𝒜39 que esteja programada na transição em vazio existente entre
os estados 7ja e 7jb.  Para isto, descobre qual das ações está
presente mediante o teste da variável x ou y,  mutuamente
exclusivas, que, quando definidas, indicam a presença da ação
correspondente, que é então alterada, mas, quando indefinidas,
inibem tal alteração. */

/* insere argumentos em 𝒜39 : época do uso das variáveis */
   - [ ( x , ε ) : → 7jb , 𝒜39 ( s , t , u , v , w ) ]
   + [ ( x , ε ) : → 7jb , 𝒜39 ( 7j , 7ja , 7jb , σ , 7j1 ) ]

/* insere argumentos em 𝒜1 : época da declaração das variáveis */
   - [ ( 7ja , ε ) : → y , 𝒜1 ( m , n , p , q , r ) ]
   + [ ( 7ja , ε ) : → y , 𝒜1 ( 7j , 7ja , 7jb , σ , 7j1 ) ]
   }

𝒜1 ( 7j , 7ja , 7jb , σ , 7j1 ) =
  { m , n , p , q , r , k* , k1* , s* , t* :
/* Esta função é característica da fase de declaração de
variáveis, e só w executada nesta época do processamento.
Destina-se a incorporar novos identificadores ao conjunto de
identificadores já encontrados, e executa esta tarefa criando
extensões adequadas no autômato, de modo que a partir de sua
execução o novo identificador seja corretamente extraído e
reconhecido como sendo o mesmo que já apareceu anteriormente. A
extensão construida é feita de tal forma que identificadores com
mesmo prefixo, mas que não tenham sido declarados ainda, sejam
caracterizados como identificadores desconhecidos. */

/* retira a transição corrente do feixe existente entre 7j e 7ja */
   - [ ( 7j , σ ) : → 7ja , 𝒜0 ( m , n , p , q , r ) ]
/* cria uma transição explícita consumindo o símbolo corrente */
   + [ ( 7j , σ ) :→ k ]
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
/* cria conexões para  letras  e  dígitos  que  seguem  o  símbolo
corrente, reconduzindo-os para 7ja */
   { + [ ( k , θ ) : → 7ja , 𝒜0 ( k , 7ja , 7jb , θ , k1 ) ]
            ∀ θ ∈ LETRAS ∪ DIGITOS }
/* achando um símbolo que não faz parte do  identificador,  não  o
consome, e desvia para finalizar a caracterização do identificador
encontrado como sendo um identificador declarado, no estado s. */
   { + [ (γ , k , θ a) : → (γ , s , θ a) ]
            ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS ) }
/* cria ação que, somente se e quando for  executada,  elimina  do
autômato esta transição e a seguinte, e em  seu  lugar  monta  uma
transição em vazio que conduz à parte do autômato que  restaura  a
cadeia  de  entrada  com  o  identificador  encontrado,  para  ser
pesquisado em outros escopos - significa que, em época de  uso  do
identificador, a função 𝒜43 ainda  não  havia  sido  chamada  para
declarar o identificador */
   + [ ( s , ε ) : 𝒜42 ( s , t , k1 ) , → t ]
/* cria uma transição para o estado 8 ( devolve ID - identificador
recém-declarado )  que  executa  a  função  𝒜43,  responsável  por
eliminar esta transição e  a  anterior,  substituindo-as  por  uma
transição em vazio para o estado 9 ( devolve VAR  -  identificador
declarado como variável) */
   + [ ( t , ε ) : → 8 , 𝒜43 ( s , t , k1 , k ) ]
/* constrói mais um passo no autômato que restaura o identificador
em caso de referência em escopo  no  qual  não  esteja  declarado,
criando uma transição de inserção do símbolo corrente na cadeia de
entrada. */
   + [ (γ , k1 , α) : → (γ , 7j1 , σ a) ]
/* cria uma transição efêmera que, uma vez consumido e devidamente
tratado o símbolo (letra ou dígito) corrente, conduz o autômato ao
estado onde deverá extrair o símbolo seguinte e se auto-destrói */
   + [ ( 7jb , ε ) : → k , 𝒜40 ( 7jb , k ) ]
   }

𝒜39 ( 7j , 7ja , 7jb , σ , 7j1 ) =
/* Em tempo de uso, um identificador não foi encontrado  entre  os
identificadores  declarados.  Esta  função  cria   uma   transição
auto-destrutiva, que promove a  restauração  do  identificador  na
cadeia de entrada para busca em outros escopos se for o caso. */
  { :
   + [ (γ , 7jb , α) : → (γ . 7j1 , σ a) , 𝒜41 ( 7jb , 7j1 ) ]
   }

𝒜40 ( x , y ) =
/* ao ser executada, esta função remove a transição que a acionou,
desde que seja uma transição em vazio no formato abaixo */
  { :
   - [ ( x , ε ) : → y , 𝒜40 ( x , y ) ]
   }

𝒜41 ( x , y ) =
/* Esta função elimina a transição que a acionou, desde que  tenha
o formato abaixo. */
  { σ :
   - [ (γ , x , α) : → (γ , y , σ a) , 𝒜41 ( x , y ) ]
   }

𝒜42 ( i , j , m ) =
/* Em tempo de declaração de variáveis, nada executa. Em  tempo  de
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
uso, elimina as transições compreendidas respectivamente entre os
pares de estados (i , j)  e  (j  ,  m),  substituindo-as por uma
transição em vazio para o estado  adequado  da  parte  do  autômato
responsável pela restauração do  identificador  não  encontrado  na
cadeia de entrada */
  { y :
/* descobre se existe uma transição de 97 para 99, indicando que  é
wpoca  de  uso  de  variáveis  previamente  declaradas.  Isto  fica
registrado na variável y, que fica indefinida durante a  época  das
declarações de variáveis. Todas as demais ações são condicionadas a
que esta variável esteja definida */
   ? [ ( 97 , y ) : → 99 ]
/* remove a transição que dispara a ação A42 */
   - [ ( i , y ) : A42 ( i , j , m ) , → j ]
/* remove a transição que dispara a ação A43 */
   - [ ( j , y ) : → 8 , A43 ( i , j , m ) ]
/* insere uma transição em vazio  para  acionar  a  restauração  do
identificador na cadeia de entrada, para análise em outro escopo */
   + [ ( i , y ) : → m ]
   }

A43 ( i , j , m , n ) =
/* Nunca é ativado em tempo de uso. Em tempo de declaração,  remove
as  transições  compreendidas  entre  (i  ,   j)   e   (j   ,   m),
substituindo-as por  uma  transição  em  vazio  para  o  estado  9,
responsável por retornar à cadeia de entrada  o  meta-símbolo  var,
que designa nome de variável declarada. */
  { x , y :
/* descobre se existe uma transição de 97 para 98, indicando que  é
wpoca de declaração de variáveis. Isto fica registrado na  variável
x, que fica definida apenas durante  a  época  das  declarações  de
variáveis. Todas as demais  ações  são  condicionadas  a  que  esta
variável esteja definida */
   ? [ ( 97 , x ) : → 98 ]
/* obtém o tipo-default  y  para as variáveis declaradas */
      ? [ ( 50 , ε ) : → y ]
/* remove a transição que ativa A43 após achar o identificador */
      - [ ( j , x ) : → 8 , A43 ( i , j , m , n ) ]
/* remove a transição que ativa A42 após achar o identificador */
      - [ ( i , x ) : A42 ( i , j , m ) , → j ]
/* insere transição para classificar o identificador como sendo  do
tipo  y  a partir da próxima  referência.  Quando  esta  regra  for
executada, a cadeia de entrada deverá receber  o  meta-símbolo  id,
indicando  que  se  trata  de  identificador  recém-declarado.  A30
registra apenas que este é o último identificador extraido. */
      + [ ( i , x ) : A30 ( n ) , → y ]
/* marca o identificador como sendo o último que foi extraido */
      A30 ( n )
      }
```

4.4.8. Expansão de Macros

Esta seção completa os exemplos desenvolvidos anteriormente, incorporando-lhes recursos para a resolução de um problema de sintaxe dinâmica que se manifesta em linguagens que exibem o recurso da utilização de macros definidas pelo usuário.

A pequena linguagem utilizada neste exemplo contém elementos representativos dos recursos encontrados na maioria das linguagens de programação clássicas, oferecendo ao seu usuário diversas construções dependentes de contexto usualmente encontradas, tais como a declaração de variáveis tipadas, a verificação de tipos, e a

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

presença de construções sintáticas que caracterizam a estruturação dos programas em blocos, definindo o escopo das variáveis declaradas.

Em adição, ilustra o emprego de autômatos adaptativos no tratamento sintático de recursos simples de extensão, representados pela utilização de macros definidas pelo usuário.

Este exemplo cobre muitos aspectos da resolução sintática dos principais problemas encontrados no tratamento de macros, já que a linguagem apresentada permite ao seu usuário definir e utilizar macros simples, macros paramétricas (aqui é também exemplificada a verificação da coerência entre os parâmetros declarados e os argumentos utilizados na chamada da macro), chamadas de macros em argumentos de outras macros, e chamadas aninhadas de macros, permitindo inclusive que macros eventualmente interdependentes exibam parâmetros formais homônimos.

Para evitar um aumento desnecessário de complexidade, a linguagem deste exemplo não permite a definição de macros recursivas, o que exigiria a incorporação de comandos de controle da recursão.

Tais comandos seriam interpretados durante a compilação, superpondo desta maneira à linguagem existente uma meta-linguagem, que, embora possa ser simples e facilmente incorporável, não traria grandes contribuições conceituais a este estudo.

Ainda para evitar redundância, a não ser que apresentem nesta versão alguma alteração, não serão repetidas aqui as partes do autômato que já tenham sido anteriormente estudadas.

Antes de passar à apresentação do autômato adaptativo que descreve o tratamento desta linguagem, convém tecer alguns comentários acerca de sua arquitetura e operação geral.

O texto-fonte de que se compõem os programas escritos na linguagem a ser tratada é inicialmente visto como uma cadeia de caracteres ASCII, que são consumidos por um analisador léxico.

O analisador léxico encarrega-se, como já foi estudado, de converter adequadamente e substituir na cadeia de entrada as seqüencias de tais caracteres ASCII do texto-fonte por códigos (não-terminais) que as classificam em categorias, destinados ao uso por outras partes do autômato adaptativo.

No caso de o analisador léxico encontrar uma seqüencia que represente a chamada de alguma macro previamente definida, isto é subentendido como indicação para que a macro seja expandida, passando-se então ao modo de operação correspondente.

Para isso, é efetuada inicialmente uma verificação de sintaxe, para garantir que o número de argumentos da chamada seja igual ao número de parâmetros formais declarados para a macro.

Simultaneamente é estabelecida uma correspondência posicional entre os argumentos e os parâmetros, efetuando-se extensões ao autômato adaptativo de tal forma que ao ser encontrada uma referência a algum parâmetro formal, as extensões correspondentes sejam utilizadas para executar ações adaptativas que promovam a sua substituição, na cadeia de entrada, pelo texto que representa o argumento a ele associado na presente chamada da macro.

O novo texto de entrada é então reprocessado pelo analisador léxico, o qual pode eventualmente encontrar novas chamadas de macros e referências a parâmetros formais, repetindo então o ciclo de expansão.

Se for encontrada uma cadeia que não corresponda a uma destas duas situações, esta cadeia será tratada normalmente, sendo classificada e substituida no texto de entrada por um código que represente um não-terminal apropriado.

Estes códigos (não-terminais) gerados pelo analisador léxico são consumidos pelas partes do autômato adaptativo encarregadas da análise sintática.

Estas, por sua vez, ao identificarem seqüencias de códigos que satisfazem à sintaxe de alguma das diversas construções lingühsticas disponíveis, inserem na cadeia de entrada o código (não-terminal) correspondente, o qual também é consumido pelo autômato adaptativo de forma similar à que foi descrita.

Conhecido o funcionamento global do autômato, passa-se a detalhar cada uma de suas partes.

4.4.8.1 Análise léxica

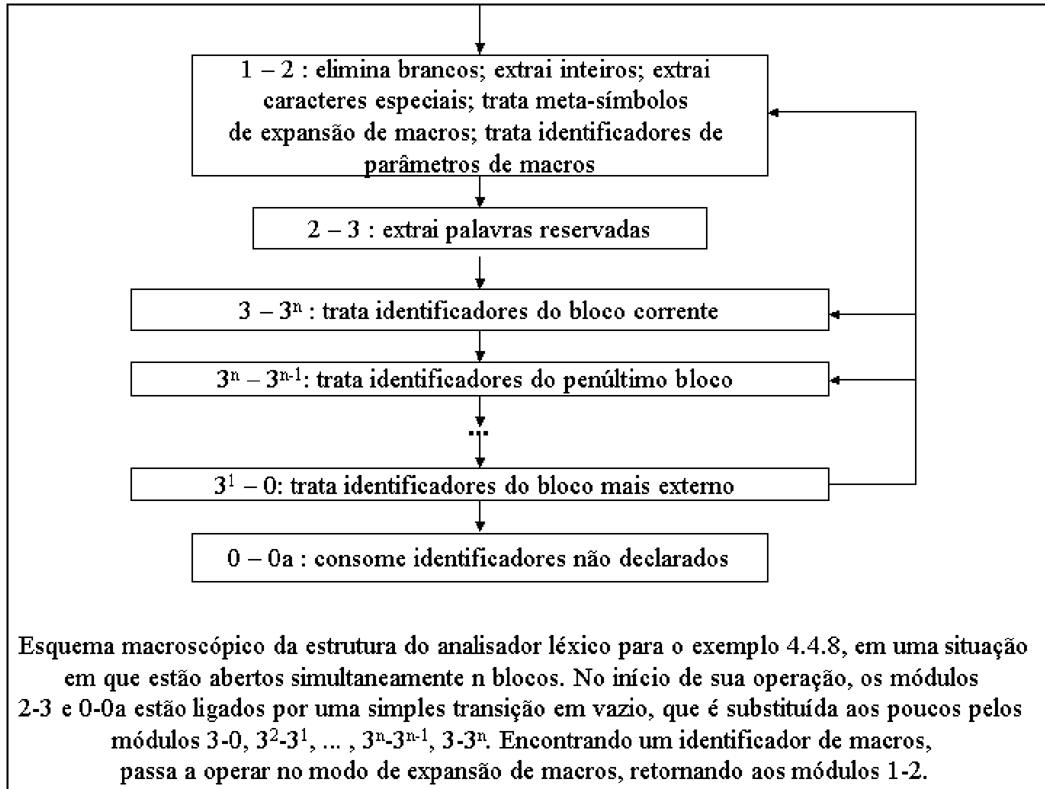
Para realizar o plano anteriormente descrito, o analisador léxico foi implementado como uma parte do autômato adaptativo, que é ativada sempre que o próximo símbolo da cadeia de entrada for um símbolo ASCII, sendo a ocasião de o analisador sintático efetuar uma transição. Opera em três modos:

- o modo normal, em que extrai uma cadeia de caracteres do texto-fonte, substituindo-a por um código (não-terminal), a ser consumido por alguma transição do analisador sintático;
- o modo símbolo, que opera de modo similar ao modo normal, extraindo entretanto um só símbolo ao invés de uma cadeia;
- o modo expansão de macro, em que extrai do texto-fonte uma chamada de macro, substituindo-a incrementalmente no texto-fonte pelo texto obtido de sua expansão, de tal modo que, sempre que ficar disponível algum átomo no início do texto-fonte, este seja imediatamente consumido pelo analisador sintático.

O esquema a seguir mostra de forma resumida os principais componentes do analisador léxico, separados em módulos funcionais. Os módulos encarregados do tratamento de identificadores operam buscando-os seqüencialmente nos conjuntos de identificadores declarados em cada um dos blocos aninhados que estejam

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

abertos na ocasião. Tratando-se de identificadores de macros já declaradas, promove sua expansão retornando ao módulo 1-2 e configurando-se para operar no modo de expansão de macro.



Esquema macroscópico da estrutura do analisador léxico para o exemplo 4.4.8, em uma situação em que estão abertos simultaneamente n blocos. No início de sua operação, os módulos 2-3 e 0-0a estão ligados por uma simples transição em vazio, que é substituída aos poucos pelos módulos 3-0, 3^2-3^1, ... , 3^n-3^{n-1}, 3-3^n. Encontrando um identificador de macros, passa a operar no modo de expansão de macros, retornando aos módulos 1-2.

```
/* Analisador léxico, Módulo 1-2: Eliminação de brancos,  Extração
de  Números  Inteiros,  Caracteres  Especiais  e  Meta-símbolos  e
Tratamento de Identificadores de Parâmetros Formais de Macros */
```

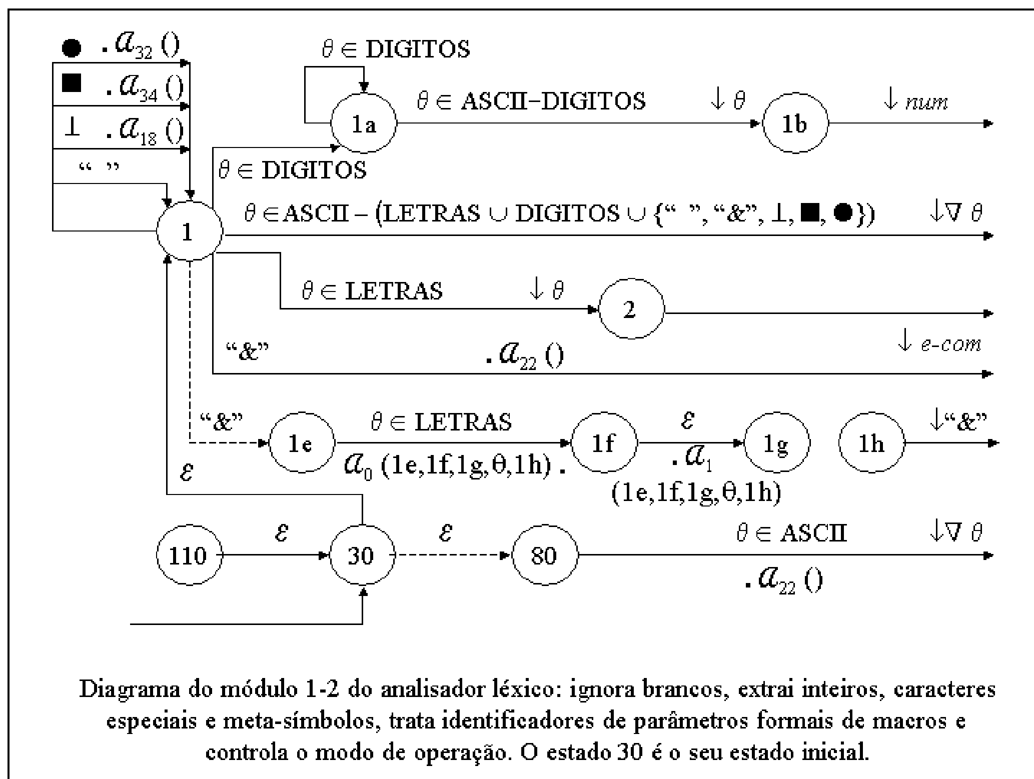


Diagrama do módulo 1-2 do analisador léxico: ignora brancos, extrai inteiros, caracteres especiais e meta-símbolos, trata identificadores de parâmetros formais de macros e controla o modo de operação. O estado 30 é o seu estado inicial.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
                    Produções iniciais:

/* Chaves de  configuração  do  modo  de  operação  do  analisador
léxico:  normal  /  caracteres  /  extração  de  identificador  de
parâmetro formal de macros: */
/* ( 30 , ε ) : → 1 = modo normal de operação: extrai átomos */
/* ( 30 , ε ) : → 80 = modo de extração de caracteres isolados */
/* { (γ z , 1 , "&" α) : → (γ , z , e-com α) , A22 () ∀ z ∈ γ }
                                   = extração de  & como átomo */
/*  ( 1 , "&" ) : → 1e   =  &  como início de parâmetro de macro */

/* inicia modo de operação do analisador léxico em normal: */
( 30 , ε ) : → 1  /* default: modo normal */

/* inicia com o tratamento de  &  como átomo isolado: */
/* default:  &  como átomo */
{ (γ z , 1 , "&" α) : → (γ , z , e-com α) , A22 () ∀ z ∈ γ }
/* Transições iniciais do analisador léxico: */

/* eliminação de brancos: */
( 1 , " " ) : → 1

/* Tratamento de marcadores para a expansão de macros: */
( 1 , 6. ) : → 1 , A32 () /*6. = fim da expansão da macro corrente*/
( 1 , ⊥ ) : → 1 , A18 () /*⊥ = fim da substit. de um parâmetro*/
( 1 , &P ) : → 1 , A34 () /*&P = fim da exp. de macro em andamento*/

/* Extração de inteiros */
{ ( 1 , θ ) : → 1a  ∀ θ ∈ DIGITOS } /* primeiro dígito */
{ ( 1a , θ ) : → 1a  ∀ θ ∈ DIGITOS } /* dígitos seguintes */
{ (γ , 1a , θ a) : → (γ , 1b , θ a)  /* finaliza com não-dígito */
                     ∀ θ ∈ ASCII - DIGITOS  }
{ (γ z , 1b , α) : → (γ , z , num α)   ∀ z ∈ γ  } /* retorna num */

/* extração de caracteres especiais e de não-terminais */
{ (γ z , 1 , θ a) : → (γ , z , ∇ θ a)
        ∀ θ ∉ (LETRAS ∪ DIGITOS ∪ { " " , "&", ⊥ , &P , 6. })  }

/*autômato inicial de extração dos nomes de parâmetros de macro:*/

{ ( 1e , θ ) : A0 ( 1e , 1f , 1g , θ , 1h ) → 1f
           ∀ θ ∈ LETRAS } /* primeira letra do identificador */
( 1f , ε ) : → 1g , A1 ( 1e , 1f , 1g , ε , 1h )

/*restaura por último o "&" que inicia nome do parâmetro rejeit.*/
{ (γ z , 1h , α) : → (γ , z , "&" α)  ∀ z ∈ γ }

/* átomo começa com letra: vai verificar se é palavra reservada */
{ (γ , 1 , θ a) : → (γ , 2 , θ a)  ∀ θ ∈ LETRAS }

/* Extração de caracteres isolados */

{ (γ z , 80 , θ a) : → (γ , z , ∇ θ a) , A22 ()  ∀ θ ∈ ASCII  }

/* O estado 110 é um estado auxiliar para onde convergem todos  os
trechos do autômato  resultantes  da  expansão  de  parâmetros  de
macro.  Contém  uma  transição  em  vazio  para   o   estado   30,
```

```
realimentando o analisador léxico: */
( 110 , ε ) : → 30

/* Extração de Palavras Reservadas - A menos da inclusão da
palavra MACRO, a extração de palavras reservadas por este
analisador léxico opera de maneira idêntica à apresentada no
analisador do exemplo 4.4.3, dispensando portanto novas
considerações. */
```

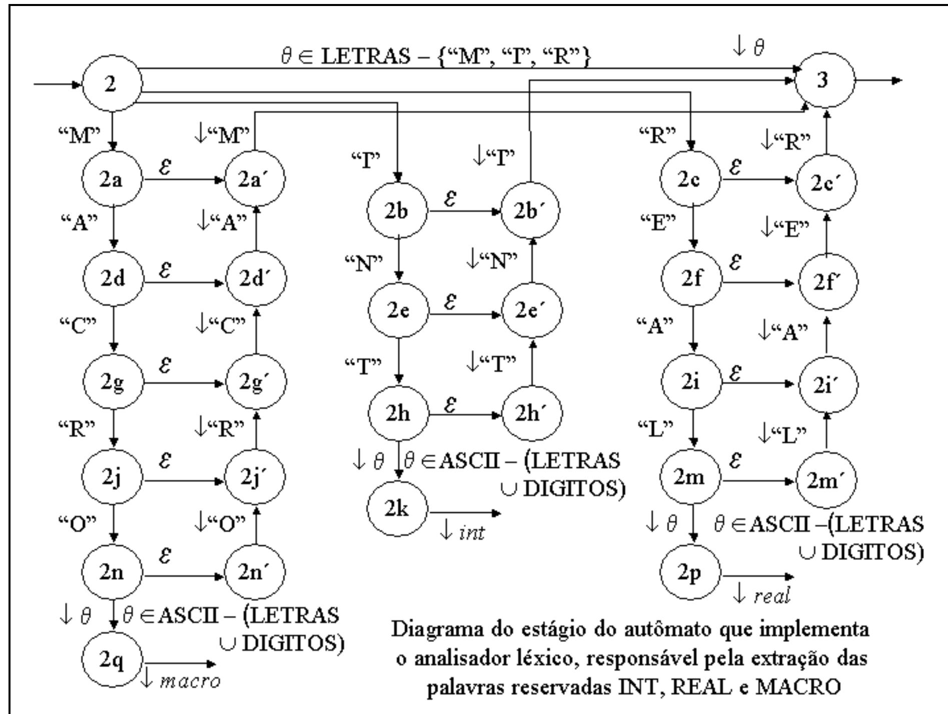


Diagrama do estágio do autômato que implementa o analisador léxico, responsável pela extração das palavras reservadas INT, REAL e MACRO

```
/* Rejeição: palavra não começa por M , I ou R */
{ (γ , 2 , θ a) : → (γ , 3 , θ a)   ∀ θ ∈ LETRAS - { M , I , R } }
/* Extração da palavra reservada MACRO */
( 2 , "M" ) : → 2a
( 2a , "A" ) : → 2d
( 2d , "C" ) : → 2g
( 2g , "R" ) : → 2j
( 2j , "O" ) : → 2n
{ (γ , 2n , θ a) : → (γ , 2q , θ a)
                ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS ) }
{ (γ z , 2q , α) : →  (γ , z , macro α)  ∀ z ∈ γ }

/* Extração da palavra reservada INT */
( 2 , "I" ) : → 2b
( 2b , "N" ) : → 2e
( 2e , "T" ) : → 2h
{ (γ , 2h , θ a) : → (γ , 2k , θ a)
                     ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS ) }
{ (γ z , 2k , α) : →  (γ , z , int α)  ∀ z ∈ γ }

/* Extração da palavra reservada REAL */
( 2  , "R" ) : → 2c
( 2c , "E" ) : → 2f
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
( 2f , "A" ) : → 2i
( 2i , "L" ) : → 2m
{ (γ , 2m , θ a) : → (γ , 2p , θ a)
                ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS ) }
{ (γ z , 2p , α) : →  (γ , z , real α)  ∀ z ∈ γ }

/* Recuperação: rejeição de seqüencia parcial de MACRO */
{ (γ , 2a , θ a) : → (γ , 2a' , θ a)
                ∀ θ ∈ ASCII - { "M" }  }
(γ , 2a' , α) : → (γ , 3 , "M" α)
{ (γ , 2d , θ a) : → (γ , 2d' , θ a)
                ∀ θ ∈ ASCII - { "A" }  }
(γ , 2d' , α) : → (γ , 2a' , "A" α)
{ (γ , 2g , θ a) : → (γ , 2g' , θ a)
                ∀ θ ∈ ASCII - { "C" }  }
(γ , 2g' , α) : → (γ , 2d' , "C" α)
{ (γ , 2j , θ a) : → (γ , 2j' , θ a)
                ∀ θ ∈ ASCII - { "R" }  }
(γ , 2j' , α) : → (γ , 2g' , "R" α)
{ (γ , 2n , θ a) : → (γ , 2n' , θ a)
                ∀ θ ∈ ASCII - { "O" }  }
(γ , 2n' , α) : → (γ , 2j' , "O" α)

/* Recuperação: rejeição de seqüencia parcial de INT */
{ (γ , 2b , θ a) : → (γ , 2b' , θ a)
                ∀ θ ∈ ASCII - { "I" }  }
(γ , 2b' , α) : → (γ , 3 , "I" α)
{ (γ , 2e , θ a) : → (γ , 2e' , θ a)
                ∀ θ ∈ ASCII - { "N" }  }
(γ , 2e' , α) : → (γ , 2b' , "N" α)
{ (γ , 2h , θ a) : → (γ , 2h' , θ a)
                ∀ θ ∈ ASCII - { "T" }  }
(γ , 2h' , α) : → (γ , 2e' , "T" α)

/* Recuperação: rejeição de seqüencia parcial de REAL */
{ (γ , 2c , θ a) : → (γ , 2c' , θ a)
                ∀ θ ∈ ASCII - { "R" }  }
(γ , 2c' , α) : → (γ , 3 , "R" α)
{ (γ , 2f , θ a) : → (γ , 2f' , θ a)
                ∀ θ ∈ ASCII - { "E" }  }
(γ , 2f' , α) : → (γ , 2c' , "E" α)
{ (γ , 2i , θ a) : → (γ , 2i' , θ a)
                ∀ θ ∈ ASCII - { "A" }  }
(γ , 2i' , α) : → (γ , 2f' , "A" α)
{ (γ , 2m , θ a) : → (γ , 2m' , θ a)
                ∀ θ ∈ ASCII - { "L" }  }
(γ , 2m' , α) : → (γ , 2i' , "L" α)

/* Não sendo palavra reservada, a seqüencia encontrada foi
restaurada na cadeia de entrada para ser reanalisada como
identificador. Em regime, o estado 3 transita em vazio para o
estado associado aos identificadores do escopo mais recente. No
início, não havendo nenhum bloco aberto, transita para o estado 0,
onde o identificador será consumido como identificador
desconhecido: */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
( 3 , ε ) : → 0

/* Tratamento de Identificadores - O tratamento de identificadores
- declarações e consumo - é efetuado como parte das ações
adaptativas executadas para a análise e reconhecimento da
estrutura de blocos da linguagem, conforme o exemplo 4.4.7 */

/* Tratamento de identificadores não declarados */
```

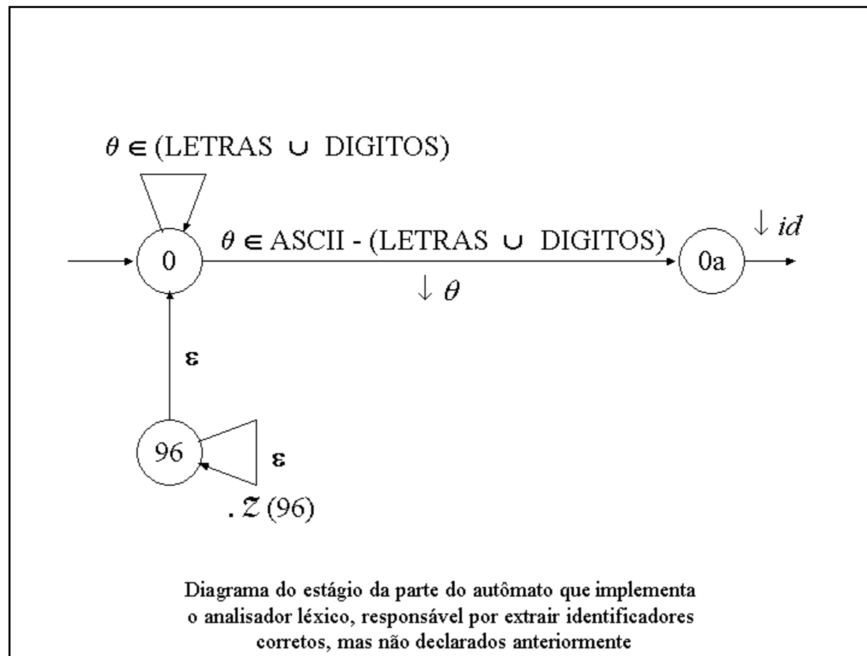


Diagrama do estágio da parte do autômato que implementa o analisador léxico, responsável por extrair identificadores corretos, mas não declarados anteriormente

```
/* Caso o identificador não seja reconhecido em nenhum dos escopos
em que esteja sendo esperado, a parte seguinte do autômato promove
a sua extração incondicional, classificando-o como id -
identificador desconhecido. */
{ (0 , θ) : → 0  ∀ θ ∈ LETRAS ∪ DIGITOS }
{ (γ , 0 , θ a) : → (γ , 0a , θ a)
                  ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS) }
{ (γ z , 0a , α) : → (γ , z , id α)  ∀ z ∈ γ }

/* 96 aponta o estado final da parte do autômato responsável pelo
tratamento dos identificadores do escopo corrente. Inicialmente
aponta o estado 0: */
( 96 , ε ) : → 0

/* 96 encabeça uma lista indicativa dos estados finais das partes
do autômato responsáveis pelos identificadores dos escopos
ancestrais. Inicialmente, aponta a si próprio, indicando que a
lista está vazia, e o indicador do estado é memorizado como
argumento da ação adaptativa "dummy" Z */
( 96 , ε ) : → 96 , Z ( 96 )
```

4.4.8.2 Declaração de Macros Paramétricas

Descreve-se a seguir o tratamento das declarações de macros, através das quais é dada ao usuário da linguagem uma construção sintática para que possa definir suas próprias macros, eventualmente paramétricas.

As ações adaptativas associadas às declarações de macros encaregam-se de coletar o nome e os parâmetros formais da macro, efetuando alterações no autômato para que seja efetuada uma verificação de consistência entre os parâmetros e os argumentos das chamadas da macro.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Além destas verificações, as ampliações do autômato promovem também a associação entre os parâmetros formais e os argumentos, de forma que as ocorrências dos identificadores dos parâmetros formais sejam substituidos pelos correspondentes argumentos quando da expansão da macro.

O diagrama seguinte mostra a estrutura da parte do autômato adaptativo incumbida do tratamento de declarações de macros.

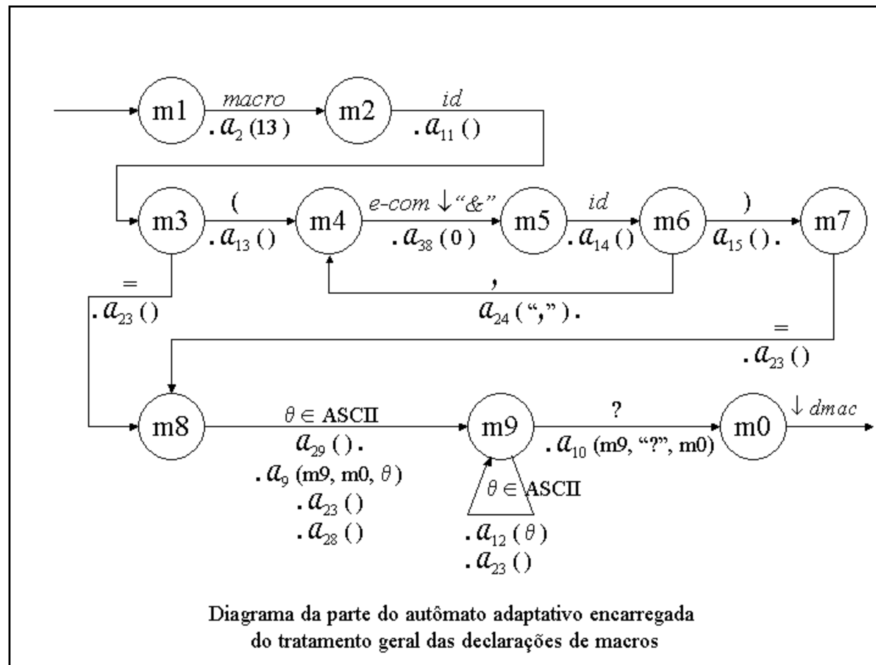


Diagrama da parte do autômato adaptativo encarregada do tratamento geral das declarações de macros

```
(m1 , macro) : → m2 , A2(13) /* configura tipo-default para void */
(m2 , id) : → m3 , A11()
(m3 ,  (  ) : → m4 , A13()  /* acrescenta ( após o identificador da
                               macro, configura tipo-default  para
                               param e acerta visibilidade da  árvore
                               de parâmetros se necessário */
(γ , m4, e-com α) : → (γ , m5, "&" α) , A38 ( 0 )
( m5 , id ) : → m6 , A14 ()
( m6 ,  ,  ) : A24 ( "," ) , → m4
( m6 ,  )  ) : A15 ( ")" ) , → m7
( m7 ,  =  ) : → m8 , A23 ()
( m3 ,  =  ) : → m8 , A23 ()
{ ( m8 , θ ) : A29 () , → m9,
                { A9 ( m9, m0, θ ), A23 (), A28 () } ∀ θ ∈ ASCII}
( m9 ,  ? ) : → m0 , A10 ( m9 , "?" , m0 )
{ ( m9 , θ ) : → m9, { A12 ( θ ) , A23 () } ∀ θ ∈ ASCII }
{ (γ x , m0 , α) : → (γ , x , dmac α)  ∀ z ∈ γ }

                         Funções Adaptativas:

A0 ( 7j , 7ja , 7jb , σ , 7j1 ) =
  { m , n , p , q , r , s , t , u , v , w , x , y :
/* Esta ação destina-se a inserir os parâmetros corretos, idênticos
aos seus próprios argumentos, nas chamadas da ação A1  ou  A39  que
esteja programada na transição em vazio existente entre os  estados
7ja e 7jb. Para isto, descobre qual das ações está presente  mediante
o teste da variável x ou  y,  mutuamente  exclusivas,  que,  quando
definidas,  indicam  a  presença  da  ação  correspondente,  que  é
então alterada, mas, quando indefinidas, inibem tal alteração. */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
/* insere argumentos em  𝒜39 : época do uso das variáveis */
   - [ ( x , ε ) : → 7jb , 𝒜39 ( s , t , u , v , w ) ]
   + [ ( x , ε ) : → 7jb , 𝒜39 ( 7j , 7ja , 7jb , σ , 7j1 ) ]

/* insere argumentos em  𝒜1 : época da declaração das variáveis */
   - [ ( 7ja , ε ) : → y , 𝒜1 ( m , n , p , q , r ) ]
   + [ ( 7ja , ε ) : → y , 𝒜1 ( 7j , 7ja , 7jb , σ , 7j1 ) ]
   }

𝒜1 ( 7j , 7ja , 7jb , σ , 7j1 ) =
  { m , n , p , q , r , k* , k1* , s* , t* :
/* Esta ação é característica da fase de declaração de variáveis, e
só w executada nesta época do processamento.  Destina-se a
incorporar novos identificadores ao conjunto de identificadores já
encontrados, e executa esta tarefa criando extensões adequadas no
autômato, de modo que a partir de sua execução o novo identificador
seja corretamente extraído e reconhecido como sendo o mesmo que já
apareceu anteriormente. A extensão construida é feita de tal forma
que identificadores com mesmo prefixo, mas que não tenham sido
declarados ainda, sejam caracterizados como identificadores
desconhecidos. */

/* retira a transição corrente do feixe existente entre 7j e 7ja */
   - [ ( 7j , σ ) : → 7ja , 𝒜0 ( m , n , p , q , r ) ]
/* cria uma transição explícita consumindo o símbolo corrente */
   + [ ( 7j , σ ) :→ k ]
/* cria conexões para letras e dígitos que seguem o símbolo
corrente, reconduzindo-os para 7ja */
   { + [ ( k , θ ) : → 7ja , 𝒜0 ( k , 7ja , 7jb , θ , k1 ) ]
            ∀ θ ∈ LETRAS ∪ DIGITOS }
/* encontrando um símbolo que não faz parte do identificador, não o
consome, e desvia para finalizar a caracterização do identificador
encontrado como sendo um identificador declarado, no estado s. */
   { + [ (γ , k , θ a) : → (γ , s , θ a) ]
            ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS ) }

/* cria ação que, somente se e quando for executada, elimina do
autômato esta transição e a seguinte, e em seu lugar monta uma
transição em vazio que conduz à parte do autômato que restaura a
cadeia de entrada com o identificador encontrado, para ser
pesquisado em outros escopos - significa que, em época de uso do
identificador, a ação 𝒜43 ainda não havia sido executada para
declarar o identificador */
   + [ ( s , ε ) : 𝒜42 ( s , t , k1 ) , → t ]
/* cria uma transição para o estado 8 ( devolve id -  identificador
recém-declarado ) que executa a ação 𝒜43, responsável por eliminar
esta transição e a anterior, substituindo-as por uma transição em
vazio para o estado 9 ( devolve var - identificador declarado como
variável) */
   + [ ( t , ε ) : → 8 , 𝒜43 ( s , t , k1 , k ) ]
/* constrói mais um passo no autômato que restaura o identificador
em caso de referência em escopo no qual não esteja declarado,
criando uma transição de inserção do símbolo corrente na cadeia de
entrada. */
   + [ (γ , k1 , α) : → (γ , 7j1 , σ a) ]
/* cria uma transição efêmera que, uma vez consumido e devidamente
tratado o símbolo (letra ou dígito) corrente, conduz o autômato ao
estado onde deverá extrair o símbolo seguinte, e se auto-destrói */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
   + [ ( 7jb , ε ) : → k , A40 ( 7jb , k ) ]
   }

A2(t) = { j :
/* estabelece como  t  o novo tipo-default para variáveis */
            - [ (50 , ε ) : → j ]
            + [ (50 , ε ) : → t ]
          }

A3 ( w , t , j ) = { σ , x :
/*rotina de controle da  montagem  do  autômato  responsável  pelo
  reconhecimento do nome do  parâmetro  formal  corrente,  para  a
  detecção da referência a homônimo no argumento associado */
            ? [ ( x , σ ) : → w ]
            A4 ( σ , x , w , t , j )
           }

A4 ( σ , x , w , t , j ) = { m* :
/*cria o  autômato  que  reconhece  o  nome  do  parâmetro  formal
  corrente,  referenciado  no  respectivo   argumento   associado,
  copiando-o  do  analisador  léxico  e  incorporando   transições
  auxiliares para tratar nomes com mesmo prefixo  ou  parcialmente
  iguais ou que contenham como prefixo o nome do parâmetro*/
         - [ ( j , e-com ) : → t , A12( "&" ) , A23() ]
         + [ ( m , σ ) : → t , A12( σ ) , A23() ]
         { + [ ( m , θ ) : → j ]  ∀ θ ∈ LETRAS ∪ DIGITOS - { σ }  }
         + [ ( j , e-com ) : → m , A12( "&" ) , A23() ]
         A3 ( x , m , j )
       }


A5 ( x ) = { y , r , s , u , v* :
/*insere na lista construida entre os estados  15  e  14  um  novo
  estado,  conectando-o  ao  estado  15,  de  modo  que  se  possa
  solicitar   à   rotina   A6   a   reprodução   do   argumento
  correspondente ao parâmetro formal homônimo */
             ? [ ( y , ε ) : → x ] /*descobre y=lista de ancestrais*/
             ? [ ( y , ε ) : → r ] /*descobre r=pont. prim. ancest.*/
             ? [ ( r , ε ) : A30( x ) → s ] /* idem s=prim. ancest.*/
             - [ ( 15 , ε ) : → u ] /* desliga lista em construção */
             + [ ( 15 , ε ) : → v ] /* insere novo estado v */
             + [ ( v , ε ) : → u ] /* reconecta p/receber cópia ...*/
             A6 ( s , v , u ) /* do argumento homônimo ancestral*/
                         }
A6 ( s , v , u ) = { x , σ :
/*copia lista iniciada em s a partir de v, ligando em  u  o  final
  da lista copiada  */
         ? [ (γ , s , α) : → (γ , x , σ a) ] /*verifica σ a empilhar*/
         A7 ( σ , v , u , x ) /* se existir, cria réplica */
                        }
A7 ( σ , v , u , x ) = { m* :
/*trabalhando em conjunto com  A6  esta  rotina  efetua  réplicas,
  uma por vez, das transições de  empilhamento  que  constituem  o
  argumento copiado */
        - [ ( v , ε ) : → u ] /* retira transição em vazio */
        + [ (γ , v , α) : → (γ , m , σ a) ] /* insere empilh. de σ */
        + [ ( m , ε ) : → u ] /* reconecta à lista */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
        𝒜6 ( x , m , u ) /* volta p/ copiar próximo empilhamento*/
                       }
𝒜8 ( φ ) = { x , y , z :
/*retira, da lista que está sendo construida entre os estados 15  e
  14, o último identificador de parâmetro inserido (que é da forma
                                              *
  & <letra> (<letra> | <dígito>)  )  */
        - [ ( 15 , ε ) : → x ] /* desliga extremidade mais recente */
        { - [ (γ , x , α) : → (γ , y , θ a) ] /* remove qualquer */
          ∀ θ ∈ ASCII - { "&" }  }  /* simbolo que não seja  &  */
        - [ (γ , x , α) : → (γ , z , "&" α) ] /* remove & (último) */
        + [ ( 15 , ε ) : → z ] /* reconecta extremidade */
        + [ ( 15 , ε ) : → y ] /*idem, no caso de ter eliminado & */
        𝒜8 ( y ) /* caso contrário, volta a eliminar outro*/
       }

𝒜9(i , j , σ) = { x :
/* elimina a transição com  ?  entre  i  e  j , substituindo-a  por
  outra com  σ  , eliminando também a transição com  σ  do  estado
  i  para o próprio estado  i */
        - [ ( i , x ) : → j, 𝒜10( i , x , j ) ] /* x é o ? corrente */
        + [ ( i , σ ) : → j, 𝒜10( i , σ , j ) ] /*substitui x por σ */
        - [ ( i , σ ) : → i , {𝒜12( σ ) ,
                                𝒜23() } ] /*impede loop com σ em i */
                    }

𝒜10( i , σ , j ) =  { k , x , y , z , r, s , t , u , v , m* :
/*restaura a transição com σ do estado i para o mesmo estado i , e
  liga o autômato que promove o empilhamento do texto que compõe o
  corpo da macro ao estado final do respectivo identificador */
            + [ ( i , σ ) : → i , { 𝒜12( σ ) , 𝒜23() } ]
            - [ ( k , ε ) : → 13 ]  /* k = fim da lista */
            - [ ( x , "&" ) : → 1 ]  /* restaura  a */
            + [ ( 1 , "&" ) : → x ]  /* extração de  &  */
            - [ ( r , "&" ) : → 1 , 𝒜22() ]  /* idem */
            + [ ( 1 , "&" ) : → r , 𝒜22() ]  /* idem */
            { - [ (γ z , 80 , "&" α) : → (γ , z , e-com α) ,
                           𝒜22 () ∀ z ∈ γ }  /* exclui & de ASCII */
            - [ ( 15 , ε ) : → t ]  /* início do empilh. do texto: */
            + [ ( k , ε ) : → t ]  /* liga-o à lista */
            - [ (γ , u , α) : → (γ , 14 , v α) ] /* final do texto: */
            + [ (γ , u , α) : → (γ , 30 , v α) ] /*realimenta léxico*/
            - [ (i , s) : → j , 𝒜10( i , s , j ) ] /* restaura i→j */
            + [ (i , ?) : → j , 𝒜10( i , ? , j ) ] /* consumindo ? */
                   }
𝒜11() = { w , x , y , z :
/*inicializa a construção do trecho do autômato  responsável  pelo
   empilhamento do  texto  correspondente  ao  corpo  ou  a  algum
   argumento de macro */
            - [ ( 15 , w ) : → x ]  /* desconecta os estados */
            - [ ( 18 , y ) : → z ]  /* auxiliares 15 e 18 */
            + [ ( 15 , ε ) : → 14 ]  /* conecta-os na ordem: */
            + [ ( 18 , ε ) : → 15 ]  /*  18 → 15 → 14   */
          }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
𝒜12( σ ) = { x , j* :
/*acrescenta o empilhamento de  σ  na cadeia de entrada à parte do
  autômato responsável pela expansão da macro */
             - [ ( 15 , ε ) : → x ] /* desconecta lista existente*/
             + [ ( 15 , ε ) : → j ] /* insere lance empilhando σ */
             + [ (γ , j , α) : → (γ , x , σ a) ] /* e reconecta */
            }

𝒜13() = { i , x , y , z , j* :
/*constrói transição com  (  após  identificador  da  macro  e  se
  prepara para receber a  lista  de  parâmetros,  enquanto  coloca
  transições adaptativas para preparar, em tempo  de  expansão,  a
  recepção dos argumentos da macro */
           - [ ( 50 , ε ) : → x ]  /*tira para evitar ambigüidade*/
           - [ ( y , ε ) : 𝒜30( i ) , → 13 ]  /* i = id. corrente */
           { - [ (γ , z , θ a) : → (γ , y , θ a) ]
                ∀ θ ∈ ASCII - ( LETRAS ∪ DIGITOS )  }
           + [ ( i , ( ) : → j ,   /* inclui trans. com  (  para */
                          {𝒜23(),  /* extrair símbolo isolado e  */
                           𝒜30(i) } ]/*marcar últ. identificador */
           + [ ( j , ε ) : → 13 ]  /*  j é o novo final da lista */
           + [ ( 50 , ε ) : → 16 ] /*  novo default = parâmetro  */
           𝒜23()  /*configura léxico para extrair símbolo isolado */
         }

𝒜14() = { w , x , n , s , m* , j* , k* , r* , t* :
/*constrói para cada parâmetro um trecho do autômato,  responsável
  pela extração do argumento correspondente e  sua  integração  ao
  analisador léxico para futura expansão */
           - [ ( 1 , "&" ) : → 70 ] /* configura léxico para */
           + [ ( 1 , "&" ) : → 17 , 𝒜22() ] /* extrair e-comercial */
           ? [ ( 90 , ε ) : → w ] /*  w = último identif. extraído */
           - [ ( n , ε ) : → 60 , 𝒜19( s ) ] /*desconecta últ.parâm*/
           + [ ( m , ε ) : → 60 , 𝒜19( w ) ] /* w é novo últ.parâm.*/
           + [ ( n , ε ) : → m , 𝒜19( s ) ] /* reconecta penúltimo */
           - [ ( i , ε ) : → 13 ] /*  i = final da lista parcial */
           { + [ ( i , θ ) : → j , /*obter delimitador do argumento*/
                   {𝒜27( j , θ , k , w ) , /* garantir par casado*/
                    𝒜23() , /*configura para extrair um símbolo */
                    𝒜25() } ] /* inicia extração de argumento */
                ∀ θ ∈ ASCII }
           { + [ ( j , θ ) : → j , /*extração do argumento da macro*/
                  { 𝒜12( θ ) , /* constrói transição com θ */
                    𝒜23() }] /*configura para extrair um caracter*/
                ∀ θ ∈ ASCII - { "&" }  } /* - exceto e-comercial - */
           + [ ( j , "&" ) : → t , 𝒜12 ( "&" ) , 𝒜23 () ]
           { + [ ( t , θ ) : → j ]
                ∀ θ ∈ (LETRAS ∪ DIGITOS)  } /* achou homônimo */
           /* inclui transições de escape se não letra nem dígito: */
           { + [ (γ , t , θ a) : → (γ , j , θ a), { 𝒜31(w), 𝒜35(w) } ]
               ∀ θ ∈ ASCII - (LETRAS ∪ DIGITOS) }
           𝒜36( w , t , j , k , x , r )  /* para complementar 𝒜14  */
         }

𝒜15( σ ) = { i , j* :
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
/*acrescenta a última transição, responsável pelo empilhamento  de
  σ na cadeia de entrada, ao autômato construído */
               - [ ( i , ε ) : → 13 ] /* desmarca final da lista */
               + [ ( i , σ ) : → j ] /* acrescenta transição */
               + [ ( j , ε ) : → 13 ] /* remarca  final da lista */
             }

𝒜16( j , k ) = { φ :
/*em tempo de expansão, conta o número de macros em  andamento,  e
  para a primeira apenas, executa 𝒜21, responsável por  configurar
  o analisador léxico para extrair nomes de parâmetros formais */
             ? [ (100 , φ) : → 100 ] /* verifica se é a primeira */
             {  𝒜21( φ , j , k ),
                𝒜20()  }
           }
𝒜17(i , ρ) = { x :
/*acrescenta uma transição adaptativa para definir o identificador
  (indefinido) com tipo default se for executada */
                 ? [ ( 50 , ε ) : → x ] /* obtém tipo default */
                 - [ ( i , ε ) : 𝒜17( i , ρ ) , → ρ ] /* substitui */
                 + [ ( i , ε ) : 𝒜30( i ) , → x ]
                                     /* i→ρ por i→x definindo id */
               }

𝒜18( ) = { x , z , r , s , t , v , w :
/*Ao final da expansão de uma macro esta rotina  se  encarrega  de
  desempilhar os argumentos desta macro, fazendo emergir  eventual
  homônimo argumento de outra macro ainda em expansão */
   - [ ( x , ε ) : → 60 , 𝒜19 ( w ) ] /*w = final do arg.*/
   - [ ( z , ε ) : → x , 𝒜19 ( t ) ] /* desempilha pont. */
   + [ ( z , ε ) : → 60 , 𝒜19 ( t ) ] /*para argumento w */
   - [ ( w , ε ) : → v , 𝒜37( w ) ] /*desl. antigo arg.corrente v*/
   - [ ( r , ε ) : → w , 𝒜37(w) ] /*desl.ant. 1o.ancestr.homôn. r*/
   - [ ( s , ε ) : → r , 𝒜37(w) ] /*desl.ant. 2o.ancestr.homôn. s*/
   + [ ( w , ε ) : → r , 𝒜37( w ) ] /* liga novo arg. corrente r*/
   + [ ( s , ε ) : → w , 𝒜37(w) ] /* liga novo 1o. ancestral s*/
           }

𝒜19( y ) = {
/*rotina "dummy" destinada apenas a guardar a informação  y  de um
  estado associado ao final do identificador de parâmetro formal */
           }

𝒜20() = { x , j*
/* incrementa contador de macros em fase de expansão (100) */
           - [ (100, ε) : → x ] /* desliga primeiro elo */
           + [ (100, ε) : → j ] /* insere estado adicional */
           + [ (j , ε) : → x ] /* conecta o novo estado */
         }

𝒜21( φ , j , k ) = {  /* φ  é parâm. dummy para exec.condicional */
/*configura o analisador léxico para interpretar  &  como  símbolo
  inicial do nome dos argumentos da macro e  extrair  corretamente
  tais nomes */
            - [ ( 1 , "&" ) : → 17 , 𝒜22() ] /*impede extração de ×/
            + [ ( 1 , "&" ) : → 70 ] /* liga na árvore de parâm.*/
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
/*as duas alterações seguintes substituem uma transição em   vazio
  por outra igual, mas que marca com  &P  o ponto exato em que  não
  mais deverá haver qualquer macro em fase de expansão */
          - [ ( j , ε ) : → k , 𝒜33( j , k ) ]
          + [ (γ , j , α) : → (γ , k , &P α) , 𝒜33( j , k ) ]
        }

𝒜22() = { :
/* configura analisador léxico para extrair átomos (modo normal) */
          - [ ( 30 , ε ) : → 80 ] /*desliga extração de caract. */
          + [ ( 30 , ε ) : → 1 ]  /* no escopo corrente */
        }
𝒜23() = { :
/*prepara o analisador léxico para  extrair  caracteres  isolados;
  após a extração, o próprio analisador léxico retorna ao normal*/
         - [ ( 30 , ε ) : → 1 ] /* desconecta extrator de átomo */
         + [ ( 30 , ε ) : → 80 ] /* conecta extrator de caract.*/
       }

𝒜24( σ ) = { i , j* :
/*acrescenta mais uma transição, responsável pelo empilhamento  de
  σ na cadeia de entrada, ao autômato construído */
             - [ ( i , ε ) : → 13 ] /* i = fim da lista parcial */
             + [ ( i , σ ) : → j , 𝒜23( ) ] /* acrescenta trans.*/
             + [ ( j , ε ) : → 13 ] /* j = novo fim da lista */
            }

𝒜25() = { x , y :
/*inibe extração de  &  e inicializa  a  lista  que  representa  o
  empilhamento do corpo ou do argumento da macro na entrada */
         𝒜11() /* inicia a lista */
         - [ ( 1 , "&" ) : → x ]  /* inibe a extração do símbolo */
         + [ ( x , "&" ) : → 1 ]  /* e-comercial como átomo */
         - [ ( 1 , "&" ) : → y , 𝒜22() ]  /* idem */
         + [ ( y , "&" ) : → 1 , 𝒜22() ]  /* idem */
          { + [ (γ z , 80 , "&" α) : → (γ , z , e-com α) ,
                          𝒜22 () ∀ z ∈ γ }  /* inclui & em ASCII */
       }

𝒜26( j , σ , k , w ) = { x , y , z , r , s , t , v  :
/*usada em tempo de expansão para a extração dos argumentos e  sua
  conexão aos parâmetros formais correspondentes */
   + [ (j , σ) : → j, {𝒜12( σ ), 𝒜23() }] /*restaura j → j */
   - [ (j , σ) : → k, 𝒜26(j , σ , k , w ) ] /*restaura j→k */
   + [ (j , .) : → k, 𝒜26(j , . , k , w ) ] /*consumindo . */
   - [ ( x , "&" ) : → 1 ]  /* restaura a extração */
   + [ ( 1 , "&" ) : → x ]  /* de  &  ( e-comercial ) */
   - [ ( s , "&" ) : → 1 , 𝒜22() ]  /* idem */
   + [ ( 1 , "&" ) : → s , 𝒜22() ] /* idem */
   { - [ (γ z , 80 , "&" α) : → (γ , z , e-com α) ,
                          𝒜22 () ∀ z ∈ γ }  /* exclui & de ASCII */
   - [ ( 15 , ε ) : → y ]  /* isola argumento corrente */
   - [ ( w , ε ) : → v , 𝒜37( w ) ] /*desl.antigo arg.corrente v*/
   - [ ( r , ε ) : → w , 𝒜37( w ) ] /*desl.antigo 1o.ancestr. r*/
   + [ ( w , ε ) : → y , 𝒜37( w ) ] /* conecta novo argumento y */
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
   + [ ( v , ε ) : → w , A37(w) ] /*conecta novo 1o.ancestral v */
   + [ ( r , ε ) : → v , A37(w) ] /*conecta novo 2o.ancestral v */
   - [ (γ , z , α) : → (γ , 14 , t α) ] /*realimenta para o */
   + [ (γ , z , α) : → (γ , 110 , t α) ] /*estado inicial 30 */
         }

A27( j , σ , k , w ) = { x :
/* equivalente a  A9 , para tempo de expansão de macro */
      - [ ( j , x ) : → k , A26( j , x , k , w ) ] /* substitui x  */
      + [ ( j , σ ) : → k , A26( j , σ , k , w ) ] /* por σ em j→k */
      - [ ( j , σ ) : → j, {A12( σ ), A23() }] /*evita cons. σ em j*/
         }

A28() = { x , σ , j :
/*configura o autômato para interpretar  σ   como  delimitador  do
  texto que compõe o corpo ou algum argumento da macro */
           - [ ( i , ε ) : → 13 ]  /* i = fim da lista parcial */
           - [ (γ , 18 , α) : → (γ , x , σ a) ] /* anexa indicador */
           + [ (γ , i , α) : → (γ , x , σ a) ] /*de n.parâm.após ) */
           - [ ( j , ε ) : → 15 ]  /* marca j como novo final da */
           + [ ( j , ε ) : → 13 ]  /* lista parcial já construida */
           A25()  /* inibe extração de  &  e inic. lista de parâm.*/
         }
A29() = { i , k , j* , m* , n* :
/*inclui imediatamente antes do corpo da macro a chamada  de  A16,
  responsável pela  contagem  dinâmica  do  número  de  macros  em
  expansão, para permitir que seja  restaurada  a  extração  de  &
  quando do término da expansão da última  macro   em   andamento;
  inclui também transição para identificar o início da expansão da
  macro corrente */
            - [ (50 , ε) : → k ] /* para evitar ambigüidade */
            - [ (i , ε) : → 13 ] /* desmarca o final  i  da lista */

/*a próxima transição incluida destina-se a iniciar a expansão  da
  macro corrente, e a ação A16 deverá marcar o ponto exato em  que
  não mais houver em andamento nenhuma macro em expansão */
            + [ ( i , ε ) : → j , A16( j , m ) ]

/*a próxima transição incluida, executada entre os estados   j   e
  m , inicialmente em vazio, e alterada para empilhar a  marca  de
  primeira macro pela ação A16 quando for o caso, se auto-restaura
  como transição em vazio pela ação adaptativa A33 */
            + [ ( j , ε ) : → m , A33( j , m ) ]

/*a próxima marca o ponto exato onde a expansão da macro  corrente
  deverá terminar */
            + [ (γ , m , α) : → (γ , n , 6. α) ]

            + [ (n , ε) : → 13 ] /* marca  j  como final da lista */
            + [ (50 , ε) : → k ] /* restaura tipo default */
          }

A30( i ) = { x :
/* memoriza, como estado sucessor do estado 90, o estado  final  do
identificador extraído por último pelo  analisador   léxico  ou  do
parâmetro corrente da macro em expansão */
               - [ ( 90 , ε ) : → x ]
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
              + [ ( 90 , ε ) : → i ]
            }

𝒜31( w ) = {
/*primeira parte do par 𝒜31 , 𝒜35 , destinado a eliminar da  lista
  que se está construindo entre os estados 14  e  15,  o  nome  do
  parâmetro em expansão, substituindo-o  pelo  argumento  homônimo
  mais recente */
              - [ ( 110 , ε ) : → 30 ]
              - [ ( 90 , ε ) : → w ]
              𝒜8 ( 1 )
            }

𝒜32() = { x , y :
/* fase de expansão: decrementa contador de macros em andamento */
           - [ (100, ε) : → x ] /* desliga primeiro elo */
           - [ (x, ε) : → y ] /* elimina o segundo elo */
           + [ (100 , ε) : → y ] /* reconecta a lista */
         }



𝒜33( j , k ) = { σ :
/*esta ação adaptativa atua sobre a transição entre os  estados  j
  e k, modificando-a para eliminar um possível empilhamento de  um
  indicador  de  primeira  macro,   previamente   incluida   nesta
  transição pela ação adaptativa 𝒜16  */
                    - [(γ , j , α) : → (γ , k , σ a) , 𝒜33(j , k)]
                    + [ ( j , ε ) : → k , 𝒜33(j , k) ]
                  }
𝒜34() = {
/*correspondente a 𝒜14 ,  para uso em tempo de expansão da  macro,
  e  destina-se  a  impedir   a   extração   dos   identificadores
  de parâmetros formais, iniciados por  &  */
           - [ ( 1 , "&" ) : → 70 ] /* configura léxico para */
           + [ ( 1 , "&" ) : → 17 , 𝒜22() ] /* extrair e-comercial */
           }
𝒜35( w ) = {
/* segunda parte do par  𝒜31 , 𝒜35 descrito em  𝒜31 */
              𝒜5 ( w )
              + [ ( 110 , ε ) : → 30 ]
              + [ ( 90 , ε ) : → w ]
            }
𝒜36( w , t , j , k , x , r ) = {
/*rotina auxiliar, destinada apenas a complementar a  operação  da
  rotina 𝒜14, garantindo que as duas partes sejam seqüenciais */
           𝒜3 ( w , t , j ) /* inclui teste de homônimo */
           + [ ( j , . ) : → k , /* final da extração do argumento */
                     𝒜26( j , . , k , w ) ] /* restaura j→k com . */
           + [ ( k , ε ) : → 13 ] /* novo final da lista = k  */
           - [ ( x , ε ) : → 15 ] /* marca a presença */
           + [ (γ , x , α) : → (γ , r , ⊥  α) ]  /*  de mais um */
           + [ ( r , ε ) : → 15 ] /* parâmetro na macro */
         }

𝒜37 ( w ) =
/* Esta ação equivale a 𝒜30, mas tem  outro  nome  para  efeito  de
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
identificação da transição que a porta. */
  { x :
    - [ ( 90 , ε ) : → x ]
    + [ ( 90 , ε ) : → w ]
    }
𝒜38 () = {  :
/* configura analisador léxico  para  interpretar  &  como  símbolo
   inicial do nome dos argumentos da macro e  extrair  corretamente
   tais nomes */
           - [ (1 , "&") :→ 17 , 𝒜22() ] /* impede extração de & */
           + [ (1 , "&") :→ 70 ] /* liga na árvore de parâmetros */
           }

𝒜39 ( 7j , 7ja , 7jb , σ , 7j1 ) =
/* Em tempo de uso, um identificador não foi  encontrado  entre  os
identificadores  declarados.  Esta  ação  cria  uma  transição
auto-destrutiva, que promove  a  restauração  do  identificador  na
cadeia de entrada para busca em outros escopos se for o caso. */
  { :
    + [ (γ , 7jb , α) : → (γ . 7j1 , σ a) , 𝒜41 ( 7jb , 7j1 ) ]
    }

𝒜40 ( x , y ) =
/* ao ser executada, esta ação remove a transição  que  a  acionou,
desde que seja uma transição em vazio no formato abaixo */
  { :
    - [ ( x , ε ) : → y , 𝒜40 ( x , y ) ]
    }

𝒜41 ( x , y ) =
/* Esta ação elimina a transição que a acionou, desde que  tenha  o
formato abaixo. */
  { σ :
    - [ (γ , x , α) : → (γ , y , σ a) , 𝒜41 ( x , y ) ]
    }

𝒜42 ( i , j , m ) =
/* Em tempo de declaração de variáveis, nada executa. Em  tempo  de
uso, elimina as transições compreendidas respectivamente  entre  os
pares de estados (i , j)   e  (j   ,  m),  substituindo-as  por  uma
transição em vazio para o estado  adequado  da  parte  do  autômato
responsável pela restauração do  identificador  não  encontrado  na
cadeia de entrada */
  { y :
/* descobre se existe uma transição de 97 para 99, indicando que  é
wpoca  de  uso  de  variáveis  previamente  declaradas.  Isto  fica
registrado na variável y, que fica indefinida durante a  época  das
declarações de variáveis. Todas as demais ações são condicionadas a
que esta variável esteja definida */
    ? [ ( 97 , y ) : → 99 ]
/* remove a transição que dispara a ação 𝒜42 */
    - [ ( i , y ) : 𝒜42 ( i , j , m ) , → j ]
/* remove a transição que dispara a ação 𝒜43 */
    - [ ( j , y ) : → 8 , 𝒜43 ( i , j , m ) ]
/* insere uma transição em vazio  para  acionar  a  restauração  do
identificador na cadeia de entrada, para análise em outro escopo */
    + [ ( i , y ) : → m ]
    }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

```
𝒜43 ( i , j , m , n ) =
/* Nunca é ativado em tempo de uso. Em tempo de declaração,  remove
as  transições  compreendidas  entre  (i   ,   j)   e   (j   ,   m),
substituindo-as por  uma  transição  em  vazio  para  o  estado  9,
responsável por retornar à cadeia de entrada  o  meta-símbolo  var,
que designa nome de variável declarada. */
  { x , y :
/* descobre se existe uma transição de 97 para 98, indicando que  é
wpoca de declaração de variáveis. Isto fica registrado na  variável
x, que fica definida apenas durante  a  época  das  declarações  de
variáveis. Todas as demais  ações  são  condicionadas  a  que  esta
variável esteja definida */
   ? [ ( 97 , x ) : → 98 ]
/* obtém o tipo-default  y  para as variáveis declaradas */
      ? [ ( 50 , ε ) : → y ]
/* remove a transição que ativa 𝒜43 após achar o identificador */
      - [ ( j , x ) : → 8 , 𝒜43 ( i , j , m , n ) ]
/* remove a transição que ativa 𝒜42 após achar o identificador */
      - [ ( i , x ) : 𝒜42 ( i , j , m ) , → j ]
/* insere transição para classificar o identificador como sendo  do
tipo  y  a partir da próxima  referência.  Quando  esta  regra  for
executada, a cadeia de entrada deverá receber  o  meta-símbolo  id,
indicando  que  se  trata  de  identificador  recém-declarado.  𝒜30
registra apenas que este é o último identificador extraído. */
      + [ ( i , x ) : 𝒜30 ( n ) , → y ]
/* marca o identificador como sendo o último que foi extraído */
      𝒜30 ( n )
  }
```

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

5 CONSIDERAÇÕES FINAIS

Com o objetivo de levantar os principais benefícios deste trabalho para a área, são discutidos criticamente neste capítulo os seus principais aspectos técnicos, explorando-se particularmente a sua aplicabilidade e as principais perspectivas de prosseguimento do trabalho.

5.1 Aplicações

Discutem-se a seguir algumas das principais aplicações dos resultados mais importantes desta tese. Naturalmente, esta discussão gira preponderantemente em torno dos aspectos da especificação e realização de linguagens de programação, mas são apresentadas diversas outras áreas onde sua aplicabilidade é marcante, como é o caso, por exemplo, da Engenharia de Software.

Aplicação à Compilação

Um dos pontos importantes para a avaliação deste trabalho é a gama de problemas aos quais os métodos e técnicas nele apresentados podem ser aplicados.

Naturalmente, a mais óbvia aplicação dos resultados desta pesquisa é para a elaboração de implementações de linguagens de programação, quer na forma de compiladores quer na de interpretadores, ou ainda na forma de softwares dirigidos pela sintaxe de sua linguagem de entrada.

Nesta classe de aplicações, os métodos aqui desenvolvidos podem ser utilizados diretamente para a construção de núcleos dirigidos por sintaxe para processadores de linguagens, na forma de reconhecedores sintáticos das linguagens de entrada de cada um dos diversos possíveis passos que devem compor o compilador ou interpretador desejado.

Identifica-se com facilidade nesta área a grande vantagem da utilização de autômatos de pilha estruturados, os quais aliam à generalidade dos métodos ascendentes, empregados como base do seu método de geração, a eficiência dos reconhecedores baseados em autômatos finitos, empregados como modelos para a execução do reconhecimento sintático por estes autômatos.

Isto permite a obtenção de um método geral e simultaneamente eficiente, que consegue operar, para linguagens determinísticas, em tempo linear, proporcional ao comprimento da cadeia de entrada, permitindo ainda a criação de versões determinísticas, ainda que à custa do aumento do número de estados do reconhecedor, para subconjuntos de interesse, controladamente restritos, de linguagens não-determinísticas.

Adicionalmente, os autômatos adaptativos fornecem ao implementador de linguagens de programação uma dimensão sintática adicional, com a qual, desvinculado das restrições, impostas nos métodos tradicionais pelas simplificações livres de contexto da linguagem tratada, passa a dispor de um modelo sintático completo para a representação integrada da sintaxe de linguagens mais reais através de um formalismo único.

Este modelo, pela sua total compatibilidade e aderência aos autômatos finitos e de pilha estruturados, dos quais pode ser visto como extensão, torna possível uma implementação muito econômica de linguagens de programação.

Isto se deve à possibilidade, que os autômatos adaptativos proporcionam, de que sejam utilizados, sob controle do projetista, apenas os recursos de reconhecimento sintático que se mostrem mais econômicos em cada situação, de forma que não seja forçada a incorporação, à implementação de uma particular linguagem, de recursos outros além daqueles que forem estritamente exigidos pela complexidade inerente da linguagem.

Aplicação à Comunicação Digital

Um outro emprego dos mecanismos e técnicas aqui explorados pode ser identificado na área das comunicações digitais, mais especificamente no campo dos protocolos de comunicação.

A maioria dos protocolos usuais podem ser vistos e tratados sintaticamente como linguagens de complexidade comparável à de alguns tipos simples de linguagens de programação.

Assim, a definição formal, bem como a implementação automática dos mecanismos sintáticos que regem estes programas, podem ser executados através da aplicação imediata das técnicas estudadas neste trabalho.

Outra aplicação do conteúdo do presente trabalho para esta área consiste na utilização do modelo aqui descrito de recuperação absoluta de erros sintáticos simples, o que pode propiciar uma forma eficiente de incorporação de mecanismos de proteção contra alguns tipos de falhas de comunicação.

Aplicação à Engenharia de Software

Cada vez mais importante devido à atualidade e significância da área da Engenharia de Software, as interfaces homem-máquina têm suscitado diversos esforços no sentido da automatização de sua realização a partir de especificações formais, principalmente em ambientes de desenvolvimento auxiliado pelo computador.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Tratando-se também de formas de linguagem, as interfaces homem-máquina mostram-se, em muitos aspectos, compatíveis com o tipo de tratamento aqui estudado, cujos métodos podem assim facilitar ou mecanizar a sua realização.

Ainda na área da Engenharia de Software, é evidente que a característica evolutiva e a estruturação hierárquica exibida pelas metodologias usuais de projeto e documentação podem explorar os métodos estruturados e adaptativos desenvolvidos nesta tese, mais ainda caracterizando, desta forma, o presente trabalho como potencial fundamento conceitual para diversas aplicações à Engenharia de Software e Metodologia de Programação.

Aplicação à Construção de Ferramentas

Embora possam ser aplicadas manualmente, as técnicas aqui apresentadas mostram-se também muito propícias à automatização, resultando ferramentas adequadas à mecanização de muitas tarefas de construção de outros sistemas complexos.

Isto torna muito natural o emprego de ferramentas baseadas nestas técnicas para a elaboração automática ou semi-automática de programas apoiados em reconhecedores, automaticamente gerados a partir de especificações formais da sua linguagem-fonte, o que proporciona uma significativa melhora na confiabilidade e na presteza do processo de elaboração de tais sistemas.

Podem ser enumerados diversos tipos de ferramentas cuja implementação pode ser beneficiada pelo emprego do modelo aqui apresentado, destacando-se aqueles ligados a sistemas de apoio a atividades de projeto e implementação de software, à geração automática ou semi-automática de compiladores, de sistemas de bancos de dados, de simuladores, de protocolos, de interfaces homem-máquina e de muitos outros.

Além da automatização da construção de núcleos baseados em versões livres de contexto das linguagens a processar, é possível libertar das restrições livres de contexto o processo de construção de reconhecedores, incorporando-se, à clássica descrição livre de contexto da sintaxe da linguagem, informações adicionais acerca de aspectos dependentes de contexto da mesma.

Esta prática dá margem, desta forma, à criação de reconhecedores mais completos para a linguagem, e não apenas para sua componente livre de contexto.

A execução das ações complementares que compõem o processador de linguagem a ser construído pode ser efetuada de várias maneiras, a mais popular das quais emprega o paradigma operacional.

Na prática, isto costuma ser implementado através da utilização direta de linguagens convencionais de programação para a especificação de algoritmos complementares, a serem executados como ações semânticas associadas às diversas transições do reconhecedor (SCHREINER; FRIEDMAN, 1985), (PYSTER, 1980), (RECHENBERG; MÖSSENBÖCK, 1989).

Uma outra alternativa, empregada em algumas ferramentas deste gênero, adota, como meta-linguagem, uma gramática de transdução, ou uma gramática de atributos, ao invés de uma simples gramática livre de contexto.

O uso de tais meta-linguagens propicia à ferramenta todas as informações necessárias à síntese não apenas do reconhecedor, como também dos mecanismos responsáveis pelo tratamento das dependências de contexto e pela geração do código-objeto.

Uma forma natural alternativa para a implementação de ferramentas resulta do emprego dos autômatos adaptativos como formalismo subjacente.

Pela natureza dinâmica destes modelos formais, os objetos que eles descrevem se apresentam como elementos com capacidade de se adaptarem às particulares situações a que são submetidos, incorporando à sua própria estrutura novos componentes conforme a particular entrada que manipulam.

Isto proporciona um substrato propício à realização de núcleos sintáticos de ferramentas de software, já que a natureza desta classe de softwares apresenta características muito aderentes à do modelo adaptativo proposto.

Aplicação à Meta-Programação

Outra área que pode ser enriquecida pela utilização do modelo dos autômatos adaptativos é a da meta-programação, ou geração automática de programas a partir de especificações formais, visto exibirem os autômatos adaptativos características que se aproximam das de muitas aplicações desta natureza.

Nesta aplicação, um autômato adaptativo pode ser criado de tal modo que reconheça inicialmente uma linguagem básica, que corresponda exatamente à meta-linguagem através da qual o software deverá ser definido.

Uma especificação formal do software desejado, denotado nesta meta-linguagem, deverá alimentar o sistema, e o seu processamento produzirá como resultado alterações adequadas nos autômatos reconhecedores da meta-linguagem inicial, de forma tal que seja incorporado, como ampliação do processador da meta-linguagem, um reconhecedor da linguagem de entrada especificada para o software em estudo.

Efetuadas todas as extensões necessárias, e incluído no reconhecedor produzido o tratamento semântico suplementar que se fizer necessário, o software especificado passará a dispor de uma implementação da sua linguagem de entrada em sua versão final, inclusive sendo possível, se conveniente, abrir mão da meta-linguagem inicial nesta ocasião.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Aplicação à Inteligência Artificial

Uma outra área que pode ser beneficiada pela utilização dos resultados deste trabalho é a da Inteligência Artificial, na qual o modelo dos autômatos adaptativos pode ser utilizado, a exemplo dos tradicionais ATN (augmented transition networks), com naturalidade para a expressão e para a resolução de diversos problemas da área, como é o caso do processamento de linguagens naturais.

Esta prática traz como vantagem a inclusão automática de mecanismos de reconhecimento dependente de contexto, além da possibilidade de exploração das características de aprendizado, usualmente implementadas por meios convencionais.

Aplicação Didática

A adequação dos autômatos adaptativos como ferramenta para a elaboração de instrumentos de cunho didático é de grande importância, conferindo a este formalismo um extraordinário potencial pedagógico em inúmeras aplicações, dadas as características que exibe.

Assim, quanto aos modelos teóricos, pode-se afirmar que, embora variantes de modelos clássicos, os autômatos de pilha estruturados e os correspondentes transdutores sintáticos, por exibirem preferencialmente a personalidade de autômatos finitos, apresentam uma potencial simplicidade, modularidade e estruturação que permitem capturar de forma intuitiva em sua própria estrutura as características sintáticas essenciais da linguagem.

Como conseqüencia, a exigência de formação prévia do aluno em assuntos teóricos adicionais, ligados a modelos de máquinas de estados, é apenas moderada, sensivelmente menor que a exigida pelos métodos tradicionais, e integralmente aderente às necessidades de outras matérias do seu curso, como é o caso de circuitos lógicos seqüenciais, protocolos, sistemas operacionais e outros, evitando assim sobrecargas curriculares desnecessárias.

Desta maneira, os modelos estruturados dos autômatos de pilha se mostram ideais para o tratamento de assuntos ligados às linguagens e compiladores em cursos de computação de caráter não predominantemente científico, criando uma forma suave de inserção, nos seus currículos, a um baixo custo, de muitos conceitos e métodos de cunho formal e teórico de grande valor para a complementação da formação destes profissionais.

Uma experiência de vários anos na utilização dos autômatos de pilha estruturados no ensino de construção de compiladores e de linguagens formais para alunos de engenharia comprova que tal modelo é muito acessível ao não-teórico, e se caracteriza por ser extremamente prático e intuitivo, garantindo uma rápida e sólida assimilação dos conceitos e das aplicações, com o enfoque apropriado à engenharia.

Aplicação em Ensino Auxiliado por Computador

A característica dinâmica do autômato adaptativo encerra uma propriedade que lhe confere um importante potencial como elemento controlador de atividades ligadas ao ensino auxiliado por computador.

Isto ocorre principalmente graças ao mecanismo adaptativo que rege o funcionamento do autômato: sendo ele próprio a realização de uma forma de aprendizado, permite uma fácil captura e o tratamento de muitas nuances do aprendizado assistido.

Desta forma, o uso dos autômatos adaptativos como estruturas subjacentes de sistemas de ensino auxiliado por computador propicia a implementação de sistemas extremamente flexíveis, que possam se adaptar, com naturalidade e a um custo reduzido, às evoluções do aprendiz, do assunto a estudar e às decisões do instrutor.

5.2 Avaliações

Da observação dos resultados a que se chegou neste trabalho, pode-se afirmar com segurança que os modelos aqui desenvolvidos se comportam de maneira muito satisfatória, quanto a diversos aspectos técnicos pertinentes - da teoria, dos métodos empregados e dos produtos resultantes da sua aplicação.

Do ponto de vista teórico, constata-se com facilidade que os três modelos aqui apresentados podem ser vistos como uma seqüencia de formalismos de complexidade progressiva, constituindo cada formalismo mais abrangente uma extensão do imediatamente anterior.

Isto lhes confere uma total compatibilidade, permitindo que sejam utilizados de modo uniforme em todas as suas aplicações, ou seja, como peças de um só modelo hierárquico mais abrangente.

Como decorrência, torna-se possível criar métodos que deles se utilizem para, a partir de uma especificação de linguagem, gerar um correspondente reconhecedor sintático, que possa ser o menos complexo possível para a linguagem em questão.

Assim, torna-se viável que, para linguagens regulares, se chegue a uma implementação baseada em autômatos finitos, enquanto para linguagens livres de contexto, não sendo possível a sua realização através de autômatos finitos, esta se faça através de autômatos de pilha estruturados.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Com os autômatos de pilha estruturados, foi visto poder-se evitar ao máximo o uso da pilha, reduzindo-se esta utilização apenas às situações em que se manifestam aninhamentos sintáticos, e operando como autômatos finitos nas situações restantes, em geral francamente majoritárias.

Desta forma, o desempenho dos autômatos de pilha estruturados será essencialmente similar ao dos autômatos finitos que constituem suas submáquinas, que é significativamente melhor que o dos reconhecedores sintáticos convencionais para linguagens livres de contexto.

Para linguagens mais complexas, para os quais os formalismos livres de contexto são insuficientes, os autômatos adaptativos proporcionam mecanismos suplementares através dos quais é possível efetuar-se alterações dinâmicas sobre o autômato.

Desta forma, na ocasião da ocorrência de construções sintáticas dependentes de contexto, ações adaptativas dotam o autômato de novos recursos livres de contexto capazes de permitir que as particulares dependências de contexto encontradas passem a ser reconhecidas de forma convencional pelo autômato, e permitindo ainda dispensar, em muitos casos, novas alterações do autômato quando da reincidência de tais construções na cadeia de entrada.

Assim, mesmo utilizando o autômato adaptativo, que é das três a forma mais complexa de reconhecedor desenvolvida neste trabalho, pode-se observar que é totalmente viável a obtenção de autômatos de complexidade apenas suficiente para atender às necessidades da linguagem cujo reconhecedor se deseja implementar.

Esta característica confere ao modelo uma qualidade pouco encontrada em formalismos desta natureza, os quais costumam ser projetados especificamente para o tratamento de uma ou de outra categoria de linguagens, não se aplicando propriamente às demais.

5.3 Perspectivas

Os resultados da pesquisa publicada neste trabalho mostram-se suficientes em si para justificar um prosseguimento dos trabalhos e uma concentração maior de esforços para que se possa obter alguns produtos adicionais de grande importância.

Assim sendo, urge ampliar a linguagem de descrição formal dos autômatos adaptativos para que se torne capaz de permitir não apenas a definição dos aspectos sintáticos de uma linguagem mas também a sua semântica.

Para que isto seja viável faz-se necessário que se lhe sejam incorporados mecanismos para a representação formal de ações semânticas autênticas, com as quais possam ser criados meios para a descrição de atividades de compilação propriamente ditas, tais como a geração de código, a interpretação de instruções, a otimização de código e muitos outros.

Para tanto, será indispensável a incorporação de uma linguagem, preferencialmente declarativa, para a especificaação das ações semânticas, a qual, na forma como o trabalho está apresentado, não faz parte do modelo formal proposto.

Outra meta para o prosseguimento deste trabalho consiste na elaboração de uma ferramenta completa de auxílio ao usuário de autômatos adaptativos, através da qual lhe seja possível desenvolver, com o auxílio do computador, programas de diversas naturezas, baseados neste formalismo, evitando as dificuldades naturais de utilização que o autômato adaptativo apresenta devido à sua inerente complexidade aparente.

Em relação às suas aplicações pedagógicas, pode-se apontar como outra importante meta para o prosseguimento deste trabalho a elaboração de uma ferramenta de auxílio à construção de programas com capacidade de aprendizado, com o apoio da qual será facilitada a criação de programas de cunho educacional, além de diversos outros, de aplicação em Inteligência Artificial.

Em particular, uma aplicação importante deste tipo de modelos formais é certamente o tratamento de linguagens não-convencionais, e mais especificamente, o das linguagens naturais, para as quais é interessante ver o autômato adaptativo como um modelo bastante adequado de representação.

Com o auxílio do autômato adaptativo para o tratamento de linguagens naturais surgem diversas importantes aplicações potenciais, como é o caso da tradução automática, do entendimento de texto, da captura de informações e organização do conhecimento subjacente, da correção ortográfica e sintática, da edição inteligente de textos e muita outras.

Pode-se concluir que existe uma infinidade de caminhos que podem ser trilhados a partir deste ponto da pesquisa, e que deve ser estimulado o surgimento de candidatos à elaboração de trabalhos que venham a dar continuidade a esta tese, contribuindo para o desenvolvimento deste fascinante ramo da computação.

5.4 Conclusões

Cabe neste ponto efetuar um pequeno balanço do papel que o produto desta tese representa no contexto da pesquisa atual em linguagens formais e autômatos.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Validade da Pesquisa

Primeiramente, levando em conta que o reconhecimento sintático de linguagens livres de contexto é considerado um problema resolvido, surge uma dúvida natural sobre a validade de se revisitar o problema. Do ponto de vista puramente teórico, um problema resolvido não costuma trazer contribuições ao ser reanalisado a não ser que abra novos caminhos conceituais ou metodológicos.

No caso deste trabalho, pode-se afirmar com segurança que isto realmente ocorreu, e isto pode ser facilmente constatado reexaminando-se os ganhos auferidos com a sua elaboração.

Importância Pedagógica

Inicialmente, pode-se citar a importância pedagógica e prática representada pelo resgate do antigo modelo de Conway, no qual se inspirou o autômato de pilha estruturado, e que permitiu desmistificar a atividade da construção de compiladores, tornando-a intuitiva mesmo para o leigo em teoria da computação.

Eficiência

Em seguida, ainda considerando os autômatos de pilha estruturados, cumpre mencionar novamente que, por imposição de projeto, este modelo foi feito compatível com os autômatos finitos, paradigmas de eficiência entre os reconhecedores sintáticos, cujas características os autômatos de pilha estruturados herdaram, já que geralmente operam a maior parte do tempo como autômatos finitos.

Considerando agora os autômatos adaptativos, nota-se mais uma vez que o caráter adaptativo do mesmo pode ser ativado exclusivamente nas ocasiões em que ocorrem manifestações de dependências de contexto no texto analisado, reduzindo o autômato, fora destas situações, a um autômato de pilha estruturado ou a um autômato finito, com todas as suas vantagens.

Unificação dos Modelos de Linguagens

Desta maneira, o método aqui apresentado logrou sistematizar em um modelo abrangente toda a hierarquia dos modelos mais simples, incorporando de forma progressiva a gradação representada pelas linguagens regulares, livres de contexto e dependentes de contexto.

Considerando-se que este trio constitui uma classe extremamente abrangente de linguagens, que cobrem na prática a quase totalidade das necessidades das aplicações computacionais mais freqüentes, pode-se facilmente confirmar outra vez a importância que um tratamento uniforme para elas, como é propiciado pelo modelo apresentado pode representar.

Recuperação de Erros

A proposta de uma extensão de recuperação absoluta de erros simples apresentada neste trabalho, embora não traga em si grandes novidades, constitui um exemplo de como pode ser possível empregar recursos bastante simples para a resolução de um problema não tão trivial do dia-a-dia do construtor de compiladores.

Autômatos finitos ou de autômatos de pilha estruturados que incorporam mecanismos de recuperação de erros podem ser construídos como casos particulares de autômatos adaptativos cujas ações adaptativas se restrinjam à criação e ativação das extensões de recuperação de erros, a serem executadas estritamente na ocasião da manifestação de erros sintáticos.

Isto permite que sejam exploradas as virtudes de tais modelos mais simples e que ao mesmo tempo possam ser incorporados ao reconhecedor mecanismos adaptativos de recuperação de erros, sem que para isto seja necessário abandonar o modelo simples de implementação inicialmente adotado.

Como os mecanismos de recuperação absoluta de erros são em geral dispendiosos, a adoção deste esquema adaptativo propicia que sejam incluídos no autômato exclusivamente aqueles mecanismos que se fizerem estritamente necessários por obra da manifestação de um particular tipo de erro no texto de entrada.

Desta forma, textos corretos nunca ativarão os mecanismos adaptativos nem as recuperações de erro, o que torna o autômato mais econômico nesta situação.

Isto naturalmente constitui uma grande vantagem frente aos modelos tradicionais, em que quase sempre os mecanismos de recuperação de erros são mais restritos e devem ser incorporados a priori ao reconhecedor utilizado.

Relação com Outros Modelos Formais

Os modelos formais aqui apresentados guardam uma estreita relação com os tradicionalmente adotados para a representação de linguagens regulares, livres de contexto e dependentes de contexto, uma vez que desempenham papéis similares.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Assim, pode-se verificar que os autômatos de pilha estruturados são equivalentes aos autômatos de pilha tradicionais, tendo a capacidade de reconhecer qualquer linguagem livre de contexto.

Quanto ao relacionamento dos autômatos adaptativos com as máquinas de Turing, é possível intuir que estas possam facilmente simulá-lo, já que no computador, e portanto na máquina de Turing, é possível criar sem grandes dificuldades um programa que se comporte como um autômato adaptativo.

Simular uma máquina de Turing em um autômato adaptativo requer:

- a simulação do estado da máquina de Turing por um estado correspondente do autômato adaptativo;
- a simulação de uma fita de trabalho a partir da posição do cursor para a direita pela cadeia de entrada modificável que o autômato adaptativo incorpora; e
- a simulação da parte da fita à esquerda do cursor por um conjunto de estados e transições de empilhamento adequadas.

Os movimentos da máquina de Turing podem ser simulados da seguinte forma:

- movimento para a direita como uma extensão correspondente do conjunto de estados e empilhamentos;
- movimento para a esquerda como a execução da transição de empilhamento associada, com eliminação dos estados e transições correspondentes;
- movimento de escrita como uma transição que efetua uma alteração da posição correntemente apontada na cadeia de entrada, consumindo o seu conteúdo corrente e empilhando o novo conteúdo.

Isto aparentemente é suficiente para justificar informalmente a equivalência dos dois modelos.

Os autômatos adaptativos podem ser também comparados às gramáticas W de dois níveis em seu funcionamento, assemelhando-se a estes pela característica, que ambos exibem, de possuirem meta-regras, com as quais vão se adaptando a cada caso particular de texto de entrada a tratar, o que permite ver o autômato adaptativo como sendo uma espécie de modelo similar às gramáticas W, representado na forma de autômato.

Publicações relativamente recentes apontam diversas pesquisas paralelas similares à relatada no presente trabalho, exibindo porém um enfoque estritamente gramatical.

Estas pesquisas e a intuição levam a crer que o conteúdo desta tese esteja posicionado em uma frente de vanguarda em nível internacional, apresentando para este complexo problema mais uma pequena contribuição rumo ao seu equacionamento e à obtenção de novas metodologias para a sua resolução.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO A DESCRIÇÃO DAS METALINGUAGENS

Neste anexo descreve-se sucintamente a Notação de Wirth e a Notação de Wirth Modificada, utilizadas em diversos pontos da tese para a formalização da sintaxe de linguagens livres de contexto.

A Notação de Wirth tradicional é usada pelos algoritmos referenciados e esboçados no Anexo E para a construção de reconhecedores apoiados em autômatos de pilha estruturados.

A Notação de Wirth Modificada, por suas propriedades convenientes, foi utilizada como base para um método, descrito no item 3.1, que permite a geração direta de autômatos de pilha estruturados quase-ótimos, dispensando, na maioria dos casos práticos, a aplicação de procedimentos de otimização, ou, ao menos, reduzindo a dificuldade de minimização da máquina gerada.

A.1 A Notação de Wirth

De uso muito prático para a especificação de linguagens livres de contexto, a Notação de Wirth, variante da conhecida forma normal de Backus (BNF), muitas vezes indevidamente denominada BNF estendido, pode ser vista como sendo composta dos seguintes elementos constituintes:

- *terminais*, cadeias de caracteres quaisquer, denotadas entre aspas ou entre apóstrofes: **"TERMINAL1" , 'TERMINAL2' , 'terminal3' , "$#$#"**
- *não-terminais*, identificadores iniciados por uma letra, e compostos de letras, dígitos e hífens: **NAO-TERMINAL-1 , Procedimento , Expressao-aritmetica**
- indicador da *cadeia vazia*, representado pelo meta-símbolo **"ε": ε**
- *fatores*, elementos básicos que compõem uma regra gramatical nesta notação: o indicador da cadeia vazia, terminais, não-terminais, e uma ou mais expressões entre parênteses, colchetes ou chaves:

  ε , 'xy' , nt1 , (x | "x" '"' nt2) , [a | b] , {(a | "a")x | b ['"' "'" '"'] | a {a} ("b" | nt1)}
- *termos*, formados pela concatenação de um ou mais fatores:

  x y "z" , (x | y) { "z" [x | y] } , x , ε
- *expressões*, formadas por um ou mais termos alternativos (disjuntos), separados entre si pelo meta-símbolo **"|"**:

  x y "z" , (x | y \ ε) x | (ε \ "z") , "z" | ε
- *regra*, que associa um não-terminal à expressão que o define. Uma regra é denotada por um não-terminal, seguido de um meta-símbolo **"="**, seguido de uma expressão, e finalizado por um meta-símbolo ponto **"."**:

  Programa = { comando } "." .

  x = x "y" | "z" .
- *definição-sintática*, formada por uma cadeia não-vazia de regras que definem a sintaxe da linguagem a ser descrita. A auto-formalização da definição-sintática Notação de Wirth, fornecida abaixo, exemplifica este conceito.

Mais formalmente, mas omitindo-se a definição dos conceitos de terminal e de não-terminal, pode-se formalizar a sintaxe da Notação de Wirth usando como meta-linguagem a própria notação que está sendo definida:

```
Notação-de-Wirth = regra { regra  } .
regra = não-terminal "=" expressão "." .
expressão = termo { "|" termo } .
termo = fator { fator } .
fator = não-terminal
      | terminal
      | "ε"
      | "(" expressão ")"
      | "[" expressão "]"
      | "{" expressão "}".
```

A Notação de Wirth tradicional apresenta, em relação à clássica notação BNF, a vantagem de permitir a expressão não-recursiva de formas sintáticas que se repetem, opcionalmente, um número arbitrário de vezes, além de oferecer uma forma para a expressão de agrupamentos e outra para designar construções sintáticas opcionais.

A.2 A Notação de Wirth Modificada

Por questão de uniformidade, e para facilitr comparações, a Notação de Wirth Modificada é abaixo descrita de forma similar à que foi utilizada acima para definir a notação tradicional.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

Esta meta-linguagem apresenta os seguintes elementos:

- *terminais*, *não-terminais* e indicador da *cadeia vazia*, idênticos aos da notação tradicional
- *fatores*, elementos básicos que compõem uma regra gramatical nesta notação: o indicador da cadeia vazia, terminais, não-terminais, e expressões entre parênteses, com ou sem indicação de repetição. A presença da indicação de repetição é denotada pelo meta-símbolo **"\"**, seguido de uma expressão indicativa da forma sintática a ser incluída entre quaisquer duas instâncias da sintaxe representada pela expressão à esquerda do meta-símbolo **"\"** :

  ε, "Terminal", PROGRAM, Ab, (x | 'y'| ε), (x | y \ z | ε), (ε\ x |"y")
- *termos*, formados pela concatenação de um ou mais fatores:

  x y "z" , (x | y \ "z")(x | y) , x , ε
- *expressões*, formadas por um ou mais termos alternativos (disjuntos), separados entre si pelo meta-símbolo **"|"** :

  x y "z" , (x | y \ ε) x | (ε \ "z") , "z" | ε
- *regra*, que associa um não-terminal à expressão que o define. Uma regra é denotada por um não-terminal, seguido de um meta-símbolo **"="**, seguido de uma expressão, e finalizado por um meta-símbolo ponto **"."**:

  x = x "y" | "z" .

  Programa = (comando ";") "." .
- *definição-sintática*, formada por uma cadeia não-vazia de regras que definem a sintaxe da linguagem a ser descrita. A auto-formalização abaixo, da Notação de Wirth Modificada, exemplifica este conceito.

Mais formalmente, mas omitindo-se a definição dos conceitos de terminal e de não-terminal, pode-se formalizar a sintaxe da Notação de Wirth Modificada, usando como meta-linguagem a própria notação que está sendo definida:

```
Notação-de-Wirth-Modificada = ( regra \ ε ) .
regra = não-terminal "=" expressão "." .
expressão = ( termo \ "|" ) .
termo = ( fator \ ε ) .
fator = não-terminal
      | terminal
      | "ε"
      | "(" expressão ( "\" expressão | ε ) ")".
```

Essencialmente idêntica às tradicionais formas de representação de linguagens livres de contexto, tais como o BNF ou a notação de Wirth tradicional, esta meta-linguagem também é apropriada para a formalização de tal classe de linguagens, apresentando porém a vantagem de permitir maior concisão na representação de construções repetitivas.

A Notação de Wirth Modificada introduz na notação tradicional uma alteração que evidencia, em construções iterativas, as formas sintáticas principais, que se repetem ao menos uma vez, e as formas sintáticas secundárias, que funcionam como conectivos entre duas instâncias da forma sintática principal.

Estas características permitem a criação de descrições sintáticas mais claras e compactas, além de proporcionarem adicionalmente uma forma unificada de expressão para agrupamentos, repetições e incidências opcionais de formas sintáticas.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO B MECANISMOS FORMAIS PARA A REPRESENTAÇÃO DE LINGUAGENS

Neste anexo descrevem-se conceitualmente, de forma sucinta, os formalismos gramaticais e de reconhecimento, utilizados para a representação de linguagens nesta tese.

B.1 Gramáticas

Formalmente, pode-se definir uma *gramática* como sendo uma quádrupla (V, Σ, P, S), onde $V = \Sigma \cup N$, sendo Σ o conjunto dos *terminais* da linguagem e N o conjunto dos *não-terminais* da gramática, ambos finitos não-vazios; $S \in N$ é a raiz da gramática; P representa o conjunto das *produções gramaticais*, da forma $\alpha \to \beta$ onde α é uma cadeia de elementos de V, contendo ao menos um elemento de N, e β é uma cadeia qualquer, eventualmente vazia, de elementos de V.

Denomina-se *forma sentencial* qualquer cadeia obtida pela aplicação recorrente das seguintes regras de substituição:

- S (a raiz da gramática) é por definição uma forma sentencial
- Seja $\alpha X \beta$ uma forma sentencial, com α e β cadeias quaisquer de terminais e/ou não-terminais da gramática, e admita-se que $X \to \gamma$ seja uma produção da gramática. Nestas condições, a aplicação desta produção à forma sentencial, substituindo a ocorrência de X por γ, produz uma nova forma sentencial, $\alpha\gamma b$.

Denota-se a substituição acima definida, dita *derivação direta*, por $\alpha X \beta \Rightarrow \alpha\gamma b$.

Uma seqüencia qualquer de derivações diretas $\alpha \Rightarrow \beta \Rightarrow ... \Rightarrow \xi$ é chamada *derivação não-trivial*, e pode ser abreviada $\alpha \Rightarrow^* \xi$.

Se, pela aplicação de uma derivação não-trivial à raiz S da gramática for possível obter uma cadeia ω, formada apenas de terminais, diz-se que ω é uma sentença: $S \Rightarrow^* \omega$.

Ao conjunto de todas as sentenças ω geradas por uma gramática G dá-se o nome de *linguagem* $L(G)$, definida por esta gramática:

$$L(G) = \{\omega \in \Sigma^* \mid S \Rightarrow^* \omega\}$$

Para o caso de linguagens regulares, é suficiente a utilização de formalismos gramaticais muito simples, como é o caso das gramáticas lineares à *direita* ou à *esquerda*.

Gramáticas lineares à direita correspondem a casos particulares em que todas as produções assumem uma das duas formas seguintes: $X \to \gamma Y$ ou $X \to \gamma$.

Analogamente, as gramáticas lineares à esquerda apresentam apenas produções das formas:

$$X \to Y \gamma$$

ou então

$$X \to \gamma$$

Para linguagens livres de contexto, o formalismo oferecido pelas gramáticas lineares é insuficiente, exigindo um poder maior de representação para que possam ser expressas construções aninhadas, ausentes nas linguagens regulares.

Um formalismo gramatical adequado à representação de linguagens livres de contexto é o denominado *gramática livre de contexto*. Uma gramática livre de contexto caracteriza-se por exibir apenas produções da forma $X \to \gamma$, onde X é um não-terminal e γ representa qualquer cadeia de terminais e/ou não-terminais da gramática.

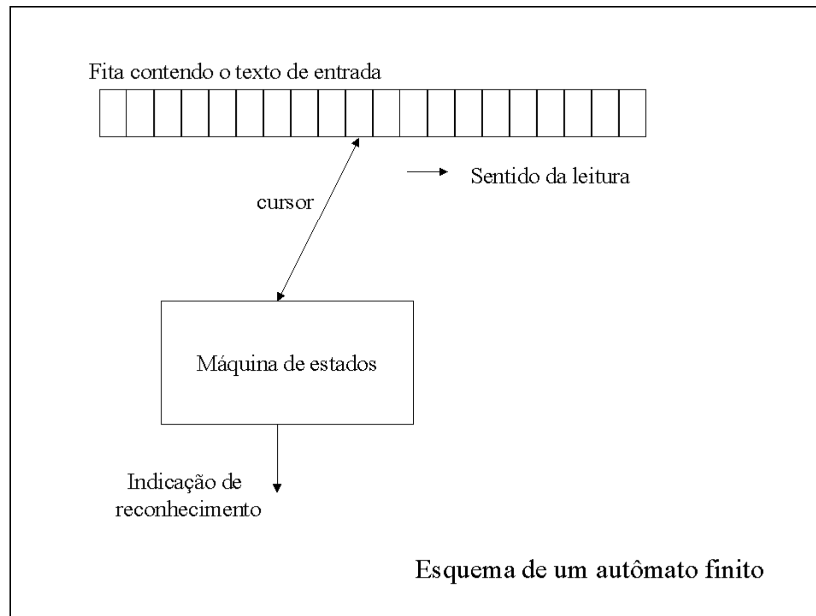
Gramáticas que apresentam não-terminais X para os quais existam derivações não-triviais do tipo $X \Rightarrow^* \alpha X \beta$, onde α e β representam quaisquer cadeias não-vazias de terminais, são chamadas, na literatura, "*self-embedded grammars*", ou seja, gramáticas que apresentam não-terminais X ditos *auto-recursivos centrais*.

B.2 Reconhecedores

Autômatos finitos constituem dispositivos reconhecedores capazes de aceitar as linguagens regulares. Demonstra-se haver equivalência entre as gramáticas lineares à direita ou esquerda e os autômatos finitos quanto à classe de linguagens que representam, que é exatamente a classe das linguagens regulares.

A figura seguinte representa o esquema conceitual de um autômato finito: note-se a ausência de memória auxiliar, fazendo com que toda a informação histórica do reconhecimento efetuado esteja contida no estado da máquina apenas.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor



Esquema de um autômato finito

Adicionalmente o autômato finito se caracteriza por restringir o modelo geral de reconhecedor nos seguintes aspectos:

- O cursor percorre o texto de entrada sempre em um único sentido
- O comprimento da fita de entrada é sempre finito
- O cursor é empregado apenas em operações de leitura
- O conteúdo da fita de entrada é inalterável

No caso particular do autômato finito, a *configuração* do reconhecedor é determinada apenas pelo estado corrente da máquina de estados finitos e pela posição do cursor na fita de entrada.

A partir da *configuração inicial*, caracterizada pela presença do cursor na posição inicial da fita de entrada, e pelo posicionamento da máquina de estados em um particular estado único, denominado *estado inicial*, o autômato finito efetua uma seqüencia de *mudanças de estado*, acompanhada de uma seqüencia de movimentações do cursor, até que seja atingida uma configuração em que não mais sejam possíveis novas transições.

Caso seja atingida uma *configuração final*, caracterizada pela presença do cursor além do último símbolo do texto na fita de entrada, e pelo posicionamento da máquina de estados em algum estado pertencente a um conjunto de *estados de aceitação*, muitas vezes ditos *estados finais*, diz-se que o autômato logrou reconhecer a cadeia de entrada.

As formas algébricas de representação dos autômatos finitos são geralmente expressas como um conjunto de triplas ordenadas, cada qual representando uma possível transição a partir de alguma configuração do autômato.

Formalmente, cada tripla $(p, \sigma) \rightarrow q$ será constituída pelos seguintes elementos:

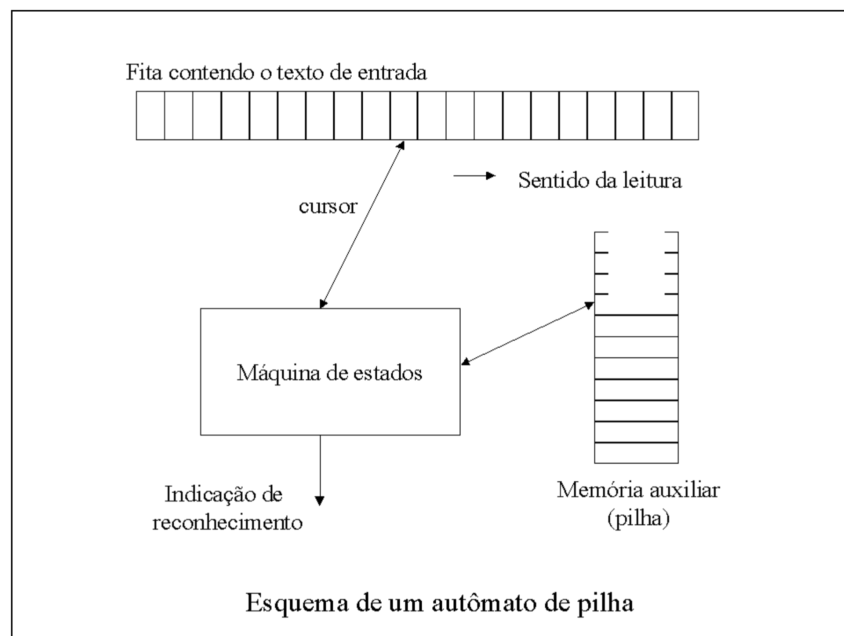
- **p:** identificação do estado-origem da transição
- **σ:** vazio ou identificação do símbolo da cadeia de entrada
- **q:** identificação do estado-destino da transição

As informações complementares, incluindo o *conjunto de estados finais e o estado inicial* do autômato, são fornecidas a parte no formalismo.

Os autômatos finitos mostram-se insuficientes para o reconhecimento de linguagens livres de contexto, as quais apresentam construções aninhadas, que exigem do reconhecedor uma capacidade de memorização de parte da história do processo de reconhecimento.

A figura seguinte apresenta o esquema conceitual de um reconhecedor dito *autômato de pilha*, adequado para o reconhecimento de linguagens livres de contexto: note-se a presença da memória auxiliar, organizada como uma pilha, fazendo com que a informação histórica do reconhecimento de construções aninhadas parcialmente efetuado seja armazenado na memória auxiliar, em complemento à informação do estado da máquina.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor



Esquema de um autômato de pilha

A partir de uma *configuração inicial*, caracterizada pela presença do cursor na posição inicial da fita de entrada, pelo posicionamento da máquina de estados em um particular estado único, denominado *estado inicial*, e pelo *conteúdo inicial da pilha* previamente convencionado (em geral vazia), o autômato de pilha efetua uma seqüencia de *mudanças de estado*, acompanhada de uma seqüencia de *movimentações do cursor e de alterações do conteúdo da pilha*, até que seja atingida uma configuração em que não mais sejam possíveis novas transições.

Caso seja atingida uma *configuração final*, caracterizada pela presença do cursor além do último símbolo do texto na fita de entrada, pela presença na pilha de um conteúdo pré-estabelecido (geralmente vazia) e não obrigatoriamente igual ao seu conteúdo inicial, e pelo posicionamento da máquina de estados em algum dos estados pertencentes a um conjunto de *estados de aceitação*, diz-se que o autômato completou com êxito o ***reconhecimento* da** cadeia de entrada.

As formas algébricas de representação dos autômatos de pilha são geralmente expressas como um conjunto de sêxtuplas ordenadas, cada qual representando uma possível transição a partir de alguma configuração do autômato.

Formalmente, cada sêxtupla

$$\gamma \; p \; \sigma \rightarrow \gamma' \; q \; \sigma'$$

será constituída pelos seguintes elementos:

- γ : especificação do conteúdo corrente da pilha
- p : identificação do estado-origem da transição
- σ : especificação da parte não-analisada da entrada
- γ' : especificação do novo conteúdo da pilha
- q : identificação do estado-destino da transição
- σ' : especificação da nova parte não-analisada da entrada

Informações adicionais são necessárias para a indicação dos *conteúdos iniciais da cadeia de entrada e da pilha, dos estados finais e do estado inicial* do autômato, e devem ser fornecidas a parte no formalismo. Os alfabetos envolvidos (de entrada e de pilha) estão implícitos no conjunto de sêxtuplas que define o autômato de pilha.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO C FORMALIZAÇÃO DOS AUTÔMATOS DE PILHA ESTRUTURADOS

Um autômato de pilha estruturado pode ser representado através de uma óctupla $(Q,A,\Sigma,\Gamma,P,Z_0,q_0,F)$, composta pelos seguintes elementos:

- Q: um conjunto finito não-vazio de *estados* $q_{i,j}$
- A: um conjunto finito não-vazio de *submáquinas* a_i
- Σ: um conjunto finito não-vazio de *símbolos de entrada* σ_i
- Γ: um conjunto finito não-vazio de *símbolos de pilha* $g_i=(m,n)$, desde que $q_{m,n}\in Q$
- P: um conjunto finito não-vazio de *produções* $p_{i,j}$
- $Z_0 \in \Gamma$: *marcador de pilha vazia*
- $q_0 \in Q$: *estado inicial* do autômato, $q_0 \equiv q_{0,0}$
- $F \subseteq Q$: conjunto não-vazio de ***estados finais*** do autômato

Cada uma das submáquinas $a_i \in A$ é representada por uma quíntupla $a_i = (Q_i , \Sigma_i , P_i , q_{i,0} , F_i)$, onde:

- $Q_i \subseteq Q$ é o conjunto de *estados* $q_{i,j}$ de a_i
- $\Sigma_i \subseteq \Sigma$ é o conjunto de *símbolos de entrada* de a_i
- $P_i \subseteq P$ é o conjunto de *produções* $p_{i,j}$ de a_i
- $q_{i,0} \in Q_i$ é o *estado inicial* de a_i
- $F_i \subseteq Q_i$ é o conjunto de *estados finais* de a_i

Cada uma das produções $p_{i,m} \in P_i$ assume a forma de uma sêxtupla $(\lambda,s,\rho,\lambda',s',\rho')$, mais didaticamente representada por um par de triplas separadas por uma seta, indicando a evolução do autômato, de uma situação representada pela tripla da esquerda, para aquela indicada pela tripla da direita da produção

$$p_{i,m} : \lambda \; s \; \rho \rightarrow \lambda' \; s' \; \rho'$$

onde:

- $\lambda \in \Gamma^*$:indica a situação da pilha antes da aplicação de $p_{i,m}$
- $s \in Q$:indica o estado do autômato antes da aplicação de $p_{i,m}$
- $\rho \in \Sigma^*$: representa a entrada antes da aplicação de $p_{i,m}$
- $\lambda' \in \Gamma^*$:indica a situação da pilha depois da aplicação de $p_{i,m}$
- $s' \in Q$: indica o estado do autômato depois da aplicação de $p_{i,m}$
- $\rho' \in \Sigma^*$: representa a entrada depois da aplicação de $p_{i,m}$

O conjunto Q de estados do autômato é a união de todos os conjuntos Q_i de estados das suas submáquinas A_i. Um estado s qualquer do autômato será sempre, portanto, algum estado $q_{i,m}$ de sua submáquina A_i.

Na denotação das produções convenciona-se que, se não for explicitado nenhum símbolo específico de entrada em ρ **e** ρ' , ou de pilha, em λ e λ', subentende-se que a execução da transição indicada pela produção independe da entrada ou do conteúdo da pilha, respectivamente.

Convenciona-se ainda que, se $\lambda = \lambda'$, não haverá alteração da pilha pela aplicação da produção. Neste caso, se λ assumir a forma γg , com g cadeia não-vazia de elementos de Γ , então a transição será condicional, sendo aplicável apenas se g coincidir com o conteúdo do topo da pilha na ocasião. Se g for omitido, a transição se processará independentemente do conteúdo do topo da pilha.

Convenções análogas são feitas para ρ **e** ρ': Caso se tenha $\rho \equiv \rho' = \alpha$, omitindo a explicitação de símbolos de entrada, isto caracterizará uma transição em vazio. Se, ao contrário, ρ for da forma $\sigma\alpha$ com σ cadeia de símbolos de Σ, e $\rho' = \alpha$, então a transição consumirá a cadeia σ sempre que for executada. Caso $\rho' = \rho = \sigma \; \alpha$, tem-se então uma transição condicional, que será executada apenas se for encontrada σ na cadeia de entrada, porém não será consumida (transição em vazio com "look-ahead").

Nos autômatos de pilha estruturados distinguem-se os seguintes tipos de transições:

- *transições internas*, com consumo de símbolos de entrada, ou em vazio, para s **e** s' , ambos pertencentes a uma mesma submáquina, ou seja, das formas $s = q_{i,j}$ e $s' = q_{i,k}$
- *transições de chamada de submáquina*, efetuadas em vazio, quando λ não explicita o conteúdo da pilha, e λ' é da forma $\gamma\phi$, onde $\phi = (m,n)$, em que $q_{m,n} \in Q_m$ para $s = q_{m,r}$, indica o estado de retorno desta transição. O estado-destino s' deverá ser o estado inicial $q_{v,0}$ de alguma submáquina $A_v \in A$. Sua forma típica é: $p_{m,t} : \gamma \; q_{m,r} \; \alpha \rightarrow \gamma(m,n) \; q_{v,0} \; \alpha$
- *transições de retorno de submáquina*, efetuadas também em vazio, quando há símbolo de pilha explícito em λ , e λ' não explicita o conteúdo da pilha. Sendo λ da forma $\gamma \; (m,n)$, deve-se ter,

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

em correspondência, $s' = q_{m,n}$, indicando o retorno de uma submáquina a_m , anteriormente chamada. Sua forma típica é a seguinte: $p_{m,t} : \gamma(m,n)\ q_{v,u}\ \alpha \rightarrow \gamma\ q_{m,n}\ \alpha$

A *operação* de um autômato de pilha estruturado, para o *reconhecimento* de uma cadeia de entrada α_0 , pode ser descrita como segue:

Seja $\Psi \notin \Sigma$ um *marcador de final da cadeia de entrada*, e seja a *situação* t do autômato representada por uma tripla do seguinte tipo:

$$t = (\lambda\ ,\ q\ ,\ \rho)$$, onde

- $\lambda \in \Gamma^*$ representa o conteúdo da pilha
- $q \in Q$ representa o estado do autômato
- $\rho \in \Sigma^*$ representa a *parte da cadeia de entrada a analisar*.

Partindo-se de uma *situação inicial* $t_0 = (Z_0\ ,\ q_{0,0}\ ,\ \rho\Psi)$, aplicam-se sucessivas transições determinadas pelas produções do autômato, até que seja atingida uma situação final representada por $t_f = (Z_0\ ,\ q_{m,n}\ ,\ \Psi)$, onde $q_{m,n} \in F$, diz-se que o autômato reconheceu a cadeia de entrada ρ . Denotando-se como $t_i \Rightarrow t_{i+1}$ uma evolução da situação t_i para a situação seguinte t_{i+1} , a cadeia de transições que efetua o reconhecimento descrito pode ser denotada abreviadamente por $t_0 \Rightarrow^* t_f$.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO D UM FORMATO PARA A REPRESENTAÇÃO DE ÁRVORES

Descreve-se neste anexo um formato de representação adequado para a geração de árvores de reconhecimento por parte de transdutores de linguagens livres de contexto baseados em autômatos de pilha estruturados, configurados para operar como analisadores sintáticos da linguagem que reconhecem.

Uma árvore de derivação de uma sentença gerada por uma gramática **(V, σ, P, S)** pode ser denotada da seguinte maneira:

- **(σ)** denota uma folha **σ** da árvore **(σ ∈ Σ ∪ { ε })**
- **(** γ_1 γ_2 **...** γ_n X_i **)** descreve uma árvore com **n** ramos, decorrente da aplicação da **i-**ésima das regras de substituição do não-terminal **X** na gramática. Seus componentes são:
  - **X** : nome do nó-raiz da sub-árvore, correspondente a um não-terminal **X ∈ V-Σ**
  - γ_j :representa o j-ésimo ramo (folha ou sub-árvore) da sub-árvore em questão **(1 ≤ j ≤ n)**
  - X_i : indica o não-terminal e o índice da regra aplicada.
- **[...** X_i **) ...** X_j **) ...** X_k **)** abrevia uma árvore do seguinte tipo: **(((...** X_i **) ...** X_j **) ...** X_k **)** , simplificando às vezes a geração da árvore, sem prejuízo da clareza da notação.
- Colchetes **[...]** operam apenas como delimitadores para o seu conteúdo, podendo ser ignorados para efeito de interpretação.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO E O MAPEAMENTO DE GRAMÁTICAS EM RECONHECEDORES

Neste anexo resume-se o procedimento de obtenção de um autômato de pilha estruturado para uma linguagem definida por uma gramática denotada em estilo BNF ou similar. Versões mais detalhadas deste procedimento podem ser encontrados em (JOSÉ NETO, 1987) e (JOSÉ NETO, 1981b). A maioria dos métodos de manipulação parciais são clássicos, indicando-se fontes da literatura onde podem ser encontrados.

- Testar se a gramática é reduzida, e manipulá-la se necessário para que assim se torne (AHO; ULLMAN, 1972).
- Reescrever as regras gramaticais, denotando-as na Notação de Wirth (JOSÉ NETO, 1987), que será adotada para a manipulação subseqüente
- Agrupar definições alternativas de cada não-terminal n_i, formando regras compostas do tipo:

$$p \;:\; n_i \rightarrow d_{i,1} \;\; | \; ... \; | \; d_{i,p}$$

- Fatorar as regras compostas:
  - colocando em evidência prefixos comuns
  - transformando em iterações as auto-recursões não-centrais
- Se a raiz da gramática n_0 for auto-recursiva central, criar uma nova raiz n' e acrescentar uma regra

$$n_0' \rightarrow n_0$$

- Criar um conjunto de todos os não-terminais que sejam auto-recursivos centrais
- Particionar este conjunto em classes tais que em classes diferentes os não-terminais não dependam ciclicamente uns dos outros, obtendo-se um conjunto C de classes independentes
- Criar um conjunto N' dos não-terminais a cada qual deverá corresponder uma das submáquinas do autômato final, onde constem:
  - a raiz da gramática,
  - os não-terminais aos quais, por opção de projeto, o usuário desejar associar uma submáquina específica
  - os não-terminais que formam classes unitárias no conjunto C de cada classe não-unitária de C, cujos elementos formem ciclos simples, o não-terminal tal que, na árvore gramatical, esteja situado à menor distância da raiz da gramática
  - de cada um dos ciclos das demais classes de C cujos elementos formem ciclos múltiplos, o não-terminal que, na árvore gramatical, estiver situado mais próximo de algum não-terminal que dele dependa, pertencente a um ciclo mais externo
- Por sucessivas substituições de não-terminais, refatorações e eliminações de auto-recursões não-centrais resultantes, obter, a partir das regras que definem os não-terminais da gramática, um novo conjunto de regras totalmente isentas de referências a não-terminais que não constem do conjunto N', permanecendo assim, da gramática inicial, apenas um subconjunto de não-terminais.
- Para cada não-terminal n que seja elemento de N_i', para o qual existe uma regra $p_i \;:\; n_i \rightarrow \delta_i$, em que δ_i é uma expressão, em notação de Wirth, obtida pela manipulação acima:
  - Eliminar não-determinismos que se manifestem nas expressões (JOSÉ NETO, 1981b), sempre que, através de fatorações da expressão e de substituições da ocorrência de não-terminais ao início de opções sintáticas pelas expressões que os definem, entre parênteses se necessário, e refatorando a expressão resultante, for obtida uma expressão isenta de não-terminais ao início de alguma opção sintática e também de opções múltiplas com o mesmo prefixo. Para eventuais não-determinismos que não possam ser eliminados desta forma, repetir, se desejado, este processo quantas vezes for necessário para tornar o autômato resultante determinístico para todos os casos de maior interesse (o não-determinismo não é eliminado, mas fica confinado aos casos de menor incidência ou de menor interesse)
- Para cada $p_i \;:\; n_i \rightarrow \delta_i$ assim obtido, criar uma submáquina a_i do autômato desejado, cujas transições devem ser extraídas das expressões δ_i (JOSÉ NETO; MAGALHÃES, 1981b), (JOSÉ NETO, 1987):
  - associam-se estados aos diversos pontos da expressão δ_i
  - criam-se transições de consumo de símbolos de entrada associadas a cada uma das ocorrências de elementos de Σ na expressão
  - criam-se transições de chamada de submáquina associadas a cada uma das ocorrências de um não-terminal n_i na expressão
  - criam-se transições de retorno de submáquina associadas ao final de cada uma das opções sintáticas que compõem as alternativas de δ_i
  - completa-se o conjunto de transições internas da submáquina acrescentando-se as transições em vazio que forem necessárias para implementar:
    - os agrupamentos simples de opções sintáticas (agrupamentos entre parênteses)
    - os agrupamentos de opções sintáticas de presença opcional(agrupamentos entre colchetes)

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

    - os agrupamentos iterativos de opções sintáticas (agrupamentos entre chaves)
  - identificam-se o estado inicial e os estados finais da submáquina, associados respectivamente às extremidades da expressão δ_i
- Por inspeção do conjunto de transições assim criadas, levantam-se os demais componentes do autômato
- Se desejado, eliminar transições em vazio e eventuais não-determinismos remanescentes, utilizando para isso algoritmos aplicáveis a autômatos finitos
- Se desejado, efetuar minimizações nas submáquinas obtidas, tratando-as como autômatos finitos e também empregando algoritmos clássicos disponíveis em diversas fontes (JOSÉ NETO, 1987), (BARRETT; COUCH, 1979), (AHO; ULLMAN, 1972)

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

ANEXO F AVALIAÇÕES DE DESEMPENHO

Neste anexo faz-se um estudo informal do desempenho dos modelos de reconhecedor adotados neste trabalho, com o objetivo de mostrar que compiladores desenvolvidos através da metodologia apresentada podem exibir um desempenho extraordinário em relação àqueles obtidos pelos métodos tradicionalmente empregados em sistemas similares.

Para efeito de comparação, é inicialmente estudado o desempenho do autômato finito. Como se sabe, autômatos finitos determinísticos apresentam-se sempre isentos de transições em vazio e nunca exibem estados de onde parta mais de uma transição consumindo o mesmo átomo.

Assim sendo, dada uma cadeia de entrada $\omega = \sigma_1 \ \ldots \ \sigma_n$, de comprimento n, o seu reconhecimento será efetuado através de exatamente n transições com consumo de átomo.

Sendo α o custo da execução de uma transição deste tipo, e supondo β o custo da operação de ativação e de encerramento da operação do autômato, pode-se concluir que o reconhecimento da cadeia ω em questão será efetuado a um custo total expresso por $\alpha \ . \ n + \beta$, proporcional portanto ao comprimento da cadeia de entrada.

Como qualquer autômato finito pode ser sempre reduzido a uma forma equivalente determinística (HARRISON, 1978), (HOPCROFT; ULLMAN, 1979a), (SALOMAA, 1973), sempre será possível obter este desempenho ótimo para as linguagens passíveis de reconhecimento por meio de autômatos finitos.

Para o caso dos autômatos de pilha, demonstra-se (JOSÉ NETO, 1981a) ser possível elaborar, a partir de qualquer gramática livre de contexto, um autômato de pilha estruturado cujas sub-máquinas, correspondentes aos não-terminais essenciais da gramática, operam essencialmente como se fossem autômatos finitos.

Na realidade, isto ocorre enquanto o autômato estiver executando apenas transições internas de suas submáquinas. Quando da detecção da ocorrência de construções sintáticas sujeitas a aninhamentos, entretanto, são executadas transições em vazio adicionais, responsáveis pelos movimentos de chamada e de retorno entre a submáquina corrente e aquela responsável pelo reconhecimento da construção sintática aninhada em questão.

Torna-se necessário, portanto, para avaliar o desempenho dos reconhecedores de linguagens livres de contexto gerais, estudar a influência da execução de tais transições adicionais sobre o custo total de reconhecimento da cadeia de entrada.

Seja uma cadeia de entrada ω, de comprimento n. Para o reconhecimento desta cadeia por um autômato de pilha estruturado, cujas submáquinas sejam apenas as essenciais, e isentas de não-determinismos e de transições em vazio:

- ativa-se inicialmente a submáquina inicial do autômato
- Enquanto for possível executar transições internas, cada uma delas deverá consumir um átomo da cadeia de entrada
- Na ocasião em que for preciso reconhecer alguma construção sintática correspondente a uma das submáquinas do autômato, uma transição de chamada de submáquina é efetuada em vazio
- Encerrado o reconhecimento por parte de uma submáquina chamada, executa-se em vazio uma transição de retorno para a submáquina chamadora
- Repete-se este procedimento até o consumo total da cadeia de entrada, tendo-se obtido sucesso no reconhecimento apenas se for atingido algum estado final do autômato

Com base nestas considerações, pode-se levantar os custos correpondentes a cada um dos passos executados nesta atividade de reconhecimento:

- À ativação da submáquina inicial associa-se um custo τ_0
- A cada consumo de átomo, associa-se um custo τ_1
- A cada chamada de submáquina, associa-se um custo τ_2
- A cada retorno de submáquina, associa-se um custo τ_3
- Ao encerramento do reconhecimento, associa-se um custo τ_4

Considerando-se que o autômato deve ao menos ser ativado e que o reconhecimento deve ser encerrado ao final da cadeia de entrada, o custo mínimo de um reconhecimento será $\tau_0 + \tau_4$.

Para a cadeia ω, de comprimento n, haverá exatamente n execuções de transições com consumo de átomos, sendo uma para cada um dos átomos de ω, logo tem-se para estas transições um custo exato total igual a $n.\tau_1$.

No caso particular de linguagens regulares implementadas com este tipo de reconhecedor com o auxílio de uma submáquina apenas, não haverá transições adicionais, e o custo total (também exato) do reconhecimento da cadeia de entrada neste caso será dado por $\tau_0 + \tau_4 + n.\tau_1$, proporcional, portanto, ao comprimento da cadeia de entrada, resultado compatível com o acima obtido para os autômatos finitos.

Havendo ativações de submáquinas, deve-se a este custo acrescentar um custo adicional, devido às transições entre submáquinas. Não se conhecendo a priori a cadeia de entrada, não é possível calcular para esta

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

parcela do custo de reconhecimento um valor exato, podendo-se, ao invés, obter limitantes superiores, com base em avaliações pessimistas.

Para tanto, é preciso estimar o maior número total possível de vezes que submáquinas podem ser chamadas durante o reconhecimento da cadeia de entrada ω.

Considerando-se a definição de aninhamento sintático, para que tal aninhamento exista é necessário que ocorra uma construção sintática, definida por um não-terminal auto-recursivo central, envolvida à esquerda e à direita por cadeias não-vazias de terminais.

Para maximizar o número de aninhamentos deve-se, portanto, reduzir ao máximo o comprimento destas cadeias envolventes. O mínimo comprimento possível para cada uma delas é de um símbolo, logo a cada aninhamento encontrado consomem-se obrigatoriamente dois símbolos envolventes da cadeia de entrada.

Para uma cadeia ω de comprimento **n** , o maior número possível de aninhamentos presentes será, portanto, **int (n / 2)**, não importando a ordem em que tais aninhamentos se encontrem na cadeia de entrada. Este será, portanto, o número máximo de chamadas (e portanto de retornos) de submáquinas a serem executadas durante o reconhecimento da cadeia, provocadas pelo aparecimento de cadeias envolventes.

No pior caso, considerado do ponto de vista do não-terminal essencial considerado, uma das possíveis formas sintáticas que implementam a base da auto-recursão central é a cadeia vazia.

Neste caso é necessário considerar que cada um dos níveis mais profundos de aninhamento com símbolos envolventes pode exigir uma ativação suplementar da submáquina associada ao não-terminal em questão.

Assim, é preciso avaliar a pior situação da cadeia de entrada em relação à presença de construções que possam dar origem a ativações de submáquinas em um mesmo nível de aninhamento sintático. O pior caso é, nesta situação, aquele em que construções aninhadas elementares (formadas apenas pelas cadeias envolventes, aninhando a cadeia vazia) formam uma seqüencia em um mesmo nível sintático.

Para uma cadeia ω , de comprimento **n** , o maior comprimento possível de uma seqüencia deste tipo ocorre quando se sucedem os pares formados apenas por duas cadeias unitárias envolventes, e portanto, em uma cadeia de comprimento n , é possível comportar, no máximo, **int (n / 2)** pares deste tipo. Cada uma destas ocorrências pode suscitar uma ativação (chamada e retorno) de submáquina.

Desta forma, considerando que a cada chamada de submáquina deve corresponder o respectivo retorno, e levando em conta a pior configuração da cadeia de entrada e a estrutura mais desfavorável para o autômato, pode-se proceder ao cálculo da parcela do custo de reconhecimento de responsabilidade das transições entre submáquinas.

O limitante superior do custo devido a transições não-internas é a soma das duas parcelas, relativas ao número máximo possível de construções aninhadas na sentença e ao maior número possível de aninhamentos sintáticos da cadeia vazia na sentença ω a ser reconhecida. Para cada um, são executadas uma chamada e um retorno de submáquina, a um custo portanto de τ_2 e τ_3 , respectivamente, contribuindo com um custo total das transições externas de

$$2 \cdot \text{int}(n / 2) \cdot (\tau_2 + \tau_3) .$$

Como

$$(n - 2) / 2 \leq \text{int}(n / 2) < n / 2$$

então **n / 2** pode ser utilizada como função limitante superior para **int (n / 2),** e assim, o custo total das transições externas terá, como limitante superior, $n \cdot (\tau_2 + \tau_3)$.

O reconhecimento de uma cadeia ω de comprimento n por um autômato de pilha estruturado tem, portanto, no pior caso, um custo dado pela soma do custo de reconhecimento de uma linguagem regular ao custo de utilização de transições entre as submáquinas do autômato:

$$\tau_0 + \tau_4 + n \cdot \tau_1 + n \cdot (\tau_2 + \tau_3)$$

ou seja,

$$(\tau_0 + \tau_4) + n \cdot (\tau_1 + \tau_2 + \tau_3)$$

mais uma vez proporcional ao comprimento n da cadeia de entrada.

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A.V.; ULLMAN, J.D. The Theory of Parsing, Translation and Compiling, vol. 1 e 2. Prentice Hall, 1972

BACKHOUSE, R.C. Syntax of Programming Languages - Theory and Practice. Prentice-Hall, 1979

BARNES, B.H. A Programmer's View of Automata. ACM Computing Surveys, 4,2, 1972

BARRETT, W.A.; COUCH, J.D. Compiler Construction - Theory and Practice. SRA, 1979. (Revised Printing)

BAUER, F.L.; EICKEL, J., ed. Compiler Construction - An Advanced Course. Springer-Verlag, 1976

BURSHTEYN, B. On the Modification of the Formal Grammar at Parse Time. ACM SIGPLAN Notices, 25, 5, pp. 117-123, 1990a

BURSHTEYN, B. Generation and Recognition of Formal Languages by Modifiable Grammars. ACM SIGPLAN Notices, 25, 10, 1990b

CABASINO, S.; PAOLUCCI, P.S.; TODESCO, G. M. Dynamic Parsers and Evolving Grammars. ACM SIGPLAN Notices 27, 11, pp. 39-48, 1992

CHRISTIANSEN, H. A Survey of Adaptable Grammars. ACM SIGPLAN Notices, 25, 11, pp. 35-44, 1990

COLE, A. J. Macro Processors. Cambridge University Press, 1976

CONWAY, M.E. Design of a Separable Transition Diagram Compiler Communications of the ACM, 6, 7, pp. 396-408, 1963

FISCHER, C.N.; LeBLANC, R.J., Jr. Crafting a Compiler Benjamin/Cummings, 1988

GHEZZI, C.; JAZAYERI, M. Programming Language Concepts. John Wiley & Sons, 1982

GOUGH, K.J. Syntax Analysis and Software Tools. Addison-Wesley, 1988

GRIES, D. Compiler Construction for Digital Computers. John Wiley & Sons, 1971

HARRISON, M.A. Introduction to Formal Language Theory Addison-Wesley, 1978

HEERING, J.; KLINT, P.; REKERS, J. Incremental Generation of Parsers. ACM SIGPLAN Notices, 24, 7, pp. 179-1|, 1989

HOPCROFT, J.E.; ULLMAN, J.D. Formal Languages and their Relations to Automata. Addison-Wesley, 1979a

HUNTER, R. The Design and Construction of Compilers. John Wiley & Sons, 1981

JOSÉ NETO, J. Introdução à Compilação. Livros Técnicos e Científicos Editora S.A., Rio de Janeiro, 1987

JOSÉ NETO, J. Uma Introdução à Compilação. Anais do congresso DECUS, Rio de Janeiro, 1988a

JOSÉ NETO, J. Uma Solução Adaptativa para Reconhecedores Sintáticos. Anais da Escola Politécnica, Departamento de Engenharia de Eletricidade, 1988b

JOSÉ NETO, J.; KOMATSU, W. Compilador de Gramáticas Descritas em Notação de Wirth Modificada. Anais EPUSP, Engenharia de Eletricidade, Série B, vol. 1, pp. 477-518, 1988

JOSÉ NETO, J.; MAGALHÃES, M.E.S. Reconhecedores Sintáticos - Uma alternativa didática para uso em cursos de engenharia. Anais do XIV CNPD. S. Paulo, 1981a

JOSÉ NETO, J.;MAGALHÃES, M.E.S. Um gerador automático de reconhecedores sintáticos para o SPD. Anais do VIII SEMISH, Florianópolis, 1981b

KASTENS, U.; HUTT, B.; ZIMMERMANN, E. GAG: A Practical Compiler Generator. Lecture Notes in Computer Science n. 141 Springer-Verlag, 1982

KERNIGAN, B.W.; PLAUGER, P.J. Software Tools in Pascal Addison-Wesley, 1981

LEAVENWORTH, B. M. Syntax Macros and Extended Translation Communications of the ACM, vol. 9, n. 11, pp. 790-793, November, 1966

LEWI, J. et al. A Programming Methodology in Compiler Construction. Part 1: Concepts; Part 2: Implementation. North Holland, 1979, 1982

LEWIS, H.R.; PAPADIMITRIOU, C.H. Elements of the Theory of Computation. Prentice-Hall, 1981

LEWIS, P.M. II; ROSENKRANTZ, D.J.; STEARNS, R.E. Compiler Design Theory. Addison-Wesley, 1976

LLORCA, S.; PASCUAL, G. Compiladores - Teoría y Construcción Paraninfo, 1986

LOMET, D.B. A Formalization of Transition Diagram Systems Journal of the ACM, 20, 2, pp. 235-257, 1973

MacLENNAN, B.J. Principles of Programming Languages: Design, Evaluation, and Implementation. Holt, Rinehart and Winston, 1983

MAGALHÃES, M.E.S.; JOSÉ NETO, J. Um gerador automático de núcleos eficientes para compiladores dirigidos por sintaxe. Anais do X SEMISH, Campinas, 1983

MAGINNIS, J.B. Elements of Compiler Construction. Meredith Corporation, 1972

MARCOTTY, M.; LEDGARD, H. Programming Language Landscape McMillan, 1986

McGETTRICK, A.D. The Definition of Programming Languages Cambridge University Press, 1980

NAUR, P. Revised Report on the Algorithmic Language ALGOL 60 Communications of the ACM, 6, 1, pp. 1-17, 1963

NIJHOLT, A. Context-Free Grammars: Covers, Normal Forms, and Parsing. Lecture Notes in Computer Science n. 93 Springer-Verlag, 1980

PAGAN, F.G. Formal Definition of Programming Languages. Prentice Hall, 1981

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

PITTMAN, T.; PETERS, J. The Art of Compiler Design - Theory and Practice. Prentice-Hall, 1992
PYSTER, A.B. Compiler Design and Construction. Van Nostrand Reinhold, 1980
RECHENBERG, P.; MÖSSENBÖCK, H. A Compiler Generator for Microcomputers. Prentice-Hall, 1989
SALOMAA, A. Formal Languages. Academic Press, 1973
SCHREINER, A.T.; FRIEDMAN, H.G., Jr. Introduction to Compiler Construction with UNIX. Prentice-Hall, 1985
TENNENT, R.D. Principles of Programming Languages. Prentice-Hall, 1981
TREMBLAY, J.P.; SORENSON, P.G. The Theory and Practice of Compiler Writing. McGraw-Hill, 1985
VAN WIJNGAARDEN, A. et al. Revised Report on the Algorithmic Language ALGOL 68. Acta Informatica, 5, pp.1-236, 1975
WAITE, W.M.; GOOS, G. Compiler Construction. Springer-Verlag, 1984
WEGNER, P. Programming Languages, Information Structures and Machine Organization. McGraw-Hill, 1968
WIRTH, N. Programming in Modula-2. Springer-Verlag, 1988
WULF, W.A. et al. Fundamental Structures of Computer Science Addison-Wesley, 1981

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor

BIBLIOGRAFIA RECOMENDADA

ABRAMSON, H. Theory and Applications of a Bottom-up Syntax-directed Translator. Academic Press, 1973
AHO, A.V.; SEHTI, R.; ULLMAN, J.D. Compilers - Principles, Techniques and Tools. Addison-Wesley, 1986
AHO, A.V.; ULLMAN, J.D. Principles of Compiler Design Addison-Wesley, 1979
BARRON, D.W., ed. Pascal - The Language and its Implementation John Wiley & Sons, 1981
BECKMAN, F.S. Mathematical Foundations of Programming Addison-Wesley, 1980
BIGGS, N.L. Discrete Mathematics. Clarendon Press - Oxford, 1989
@BRINCH-HANSEN, P. Brinch-Hansen on Pascal Compilers Prentice-Hall, 1985
BROWN, P.J. Writing Interactive Compilers and Interpreters. John Wiley & Sons, 1979
CALINGAERT, P. Assemblers, Compilers and Program Translation Computer Science Press, 1979
CUNIN, P.Y.; GRIFFITHS, M.; VOIRON, J. Comprendre la Compilation Springer-Verlag, 1980
DONOVAN, J.J. Systems Programming. McGraw-Hill, 1972
FELDMAN, J.; GRIES, D. Translator Writing Systems. Communications of the ACM, 11, 2, pp. 77-113, 1968
GORDON, M.J.C. Programming Language Theory and its Implementation. Prentice-Hall, 1988
GUESSARIAN, I. Algebraic Semantics. Lecture Notes in Computer Science n. 99. Springer-Verlag, 1981
HALSTEAD, M.H. A Laboratory Manual for Compiler and Operating System Implementation. Elsevier. North Holland, 1974
HARTMANN, A.C. A Concurrent Pascal Compiler for Minicomputers Lecture Notes in Computer Science n. 50. Springer-Verlag, 1977
HENDRIX, J.E. A Small C Compiler. M & T Books, 1990
HOLUB, A.I. Compiler Design in C. Prentice-HAll, 1990
HOPCROFT, J.E.; ULLMAN, J.D. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979b
JONES, N.D. Semantics-Directed Compiler Generation. Lecture Notes in Computer Science n. 94. Springer-Verlag, 1980
MAK, R. Writing Compilers and Interpreters - An Applied Approach John Wiley & Sons, 19
MANNA, Z. Mathematical Theory of Computation. McGraw-Hill, 1974
REES, M.; ROBSON, D. Practical Compiling with Pascal-S Addison-Wesley, 1988
RÉVÉSZ, G.E. Introduction to Formal Languages. McGraw-Hill, 1983
SALOMAA, A. Jewels of Formal Language Theory. Computer Science Press, 1981
TERRY, P.D. Programming Language Translation - A Practical Approach. Addison-Wesley, 1986
TREMBLAY, J.P.; SORENSON, P.G. An Implementation Guide to Compiler Writing. McGraw-Hill, 1982
ULLMAN, J.D. Fundamental Concepts of Programming Systems - Addison-Wesley, 1976

© Copyright 1993 João José Neto – Direitos autorais reservados – Proibida a reprodução total ou parcial sem a permissão expressa do autor