JOEL CAMARGO DIAS PEREIRA

AMBIENTE INTEGRADO DE DESENVOLVIMENTO DE RECONHECEDORES SINTÁTICOS, BASEADO EM AUTÔMATOS ADAPTATIVOS

Dissertação apresentada ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Elétrica

São Paulo, 1999

JOEL CAMARGO DIAS PEREIRA

AMBIENTE INTEGRADO DE DESENVOLVIMENTO DE RECONHECEDORES SINTÁTICOS, BASEADO EM AUTÔMATOS ADAPTATIVOS

Dissertação apresentada ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Elétrica

Área de Concentração: Sistemas Digitais

Orientador: Prof. Dr. João José Neto

São Paulo, 1999

À minha querida esposa Elaine
e ao nosso Salvador Jesus Cristo

Agradecimentos

Algumas pessoas desempenharam papéis fundamentais durante o curso deste trabalho. A estas pessoas eu gostaria de agradecer publicamente pela colaboração e préstimos a mim dirigidos.

Ao meu querido e sempre presente Salvador, Mantenedor de todas as coisas, Jesus Cristo.

Ao Prof. João José Neto pela inestimável ajuda e orientação durante as fases de desenvolvimento da ferramenta RSW e elaboração deste texto.

À minha querida esposa, Elaine, pela contínua e ilimitada paciência, companheirismo, estímulo, entusiasmo e compreensão nos inúmeros momentos que nos privamos de horas de lazer junto aos nossos familiares e amigos.

À minha mãe, Elizabeth, que por diversas vezes em sua vida priorizou a minha formação acadêmica, para que eu tivesse condições de concluir esta dissertação.

Aos meus familiares, especialmente ao Carlos, Yara e mais recentemente a Mariana, pelo companheirismo constante.

Aos familiares da minha esposa, pelos vários momentos de ausência ao longo destes anos.

São Paulo, 1999

Joel Camargo Dias Pereira

Sumário

Lista de Figuras

Listagens

Lista de Tabelas

Resumo

Novas propostas teóricas sobre a construção de reconhecedores sintáticos surgem todo ano. Para muitas destas propostas, há a necessidade de implementação de tais teorias, muitas vezes para verificar sua viabilidade na prática.

As ferramentas de auxílio ao desenvolvimento de reconhecedores devem apresentar as mesmas características das ferramentas de programação disponíveis no mercado. Podemos citar, entre outras, as seguintes características: edição de programas ou códigos-fonte, compilação e tratamento de erros de compilação, execução, depuração e otimização dos reconhecedores. Todas estas tarefas devem ser realizadas no mesmo ambiente, ganhando-se assim, em tempo e praticidade.

Este trabalho introduz uma nova ferramenta de auxílio ao desenvolvimento de reconhecedores sintáticos, denominada RSW. A ferramenta RSW tem como meta proporcionar aos seus usuários um ambiente integrado onde se encontrem as características acima citadas. Os reconhecedores sintáticos reconhecidos e gerados pela ferramenta são baseados na teoria dos autômatos finitos, autômatos de pilha estruturados e nos autômatos adaptativos.

A possibilidade de implementação de reconhecedores baseados em Autômatos Adaptativos constitui uma das metas importantes alcançadas pela ferramenta RSW, comprovando sua viabilidade prática.

Abstract

Every year new theoretical proposals covering the construction of language parsers appear. Many of these, must be implemented in order to verify its viability in real world.

New tools supporting the developing of parsers should have the same features in available commercial tools. Some of these features include source code editing, compiling, error handling, execution, debugging and parser profiling. Such tasks should be executed in the same environment, in order to save time and to increase productive.

This work presents RSW, a new tool for parser generator. The purpose of RSW is to make available an integrated developing environment to its users, where the features aforementioned are available. Syntax recognizers generated by RSW are based on the theory of finite-state-, pushdown- and adaptive-automata.

The possibility of implementing recognizers based on adaptive automata is the main goal reached by RSW, showing in practice its viability.

Capítulo I - Introdução

Os autômatos adaptativos têm sido estudados nos últimos anos, na teoria da computação, como dispositivos cujo funcionamento é basicamente semelhante ao dos autômatos finitos ou de pilha, com o acréscimo da capacidade de alterar automaticamente seu modelo estrutural à medida que a cadeia de entrada vai sendo consumida.

Com isso é possível atribuir a tais autômatos, tarefas antes impossíveis de serem executadas pelos autômatos finitos e de pilha. Algumas destas tarefas costumam ser tradicionalmente tratadas em nível semântico. Porém com a capacidade do autômato de gravar informações da cadeia de entrada para futura utilização, assim como de alterar sua própria topologia, é possível transferir parte destas tarefas para a análise sintática, resultando assim um modelo mais aderente às suas raízes conceituais.

Geralmente os autômatos propostos pela teoria são desenvolvidos e/ou formalizados através de sistemas matemáticos, mediante notações adequadas para este fim. Adicionalmente, determinam-se aplicações práticas em que tal teoria poderia ser utilizada. Dentre estas aplicações, a área de construção de compiladores é uma das que apresentam um grande interesse. Portanto é também de grande interesse ferramentas de cunho prático que auxiliem o estudo e o entendimento dos modelos matemáticos propostos. Uma destas ferramentas constitui os Meta Reconhecedores Sintáticos.

Estas ferramentas auxiliam a construção de compiladores, e se utilizam de algumas técnicas derivadas das teorias matemáticas. Geralmente deve-se especificar as regras de reconhecimento do compilador através de uma gramática. O Meta Reconhecedor Sintático passa então a gerar o código-fonte do reconhecedor desejado a partir da especificação gramatical.

O código-fonte, gerado por um Meta Reconhecedor Sintático, na sua maioria, se utiliza de alguma linguagem de alto nível, tal como Pascal, C, Módula, entre outras. Essa característica permite ao usuário adicionar outras rotinas ou módulos que complementem o comportamento do reconhecedor, mas que são difíceis de serem geradas automaticamente. O reconhecedor gerado também se utiliza de alguma técnica de implementação de reconhecedores como, por exemplo, reconhecedores recursivos descendentes, top-down, bottom-up, entre outros.

Assim, os Meta Reconhecedores Sintáticos são ferramentas essenciais para o desenvolvimento confortável de novas linguagens de programação e de seus respectivos reconhecedores, pois além de aumentarem a rapidez com que se produzem os reconhecedores, também são ótimos meios de documentação, sem contar com a imensa vantagem de se obter código-fonte livre de erros de programação.

Passaremos a estudar os Meta Reconhecedores Sintáticos disponíveis, bem como suas vantagens e desvantagens, notações, técnicas e formalismos. A seguir cobriremos os aspectos teóricos dos Autômatos Adaptativos. Passaremos então a descrever o produto proposto por esta dissertação, que é um ambiente integrado de desenvolvimento de reconhecedores sintáticos, que opera fundamentado no modelo dos Autômatos Adaptativos .

Capítulo II - Conceitos

Neste capítulo cobriremos alguns aspectos da teoria que fornece as bases do trabalho desenvolvido no decorrer desta dissertação.

Inicialmente estudaremos as ferramentas de apoio ao desenvolvimento de reconhecedores, denominadas Meta Reconhecedores Sintáticos , ou Sistemas Geradores de Reconhecedores Sintático. Apresentaremos o conceito básico desta ferramenta, seu esquema de funcionamento e o processo de *bootstrap* como prática eficiente e produtiva de desenvolvimento. Em seguida detalharemos características dos Meta Reconhecedores Sintáticos em uso, tais como linguagens de especificação, formas de interface com o usuário, linguagens utilizadas para os compiladores gerados e algoritmos utilizados no processamento das especificações.

A outra seção deste capítulo é dedicada a uma revisão sobre Autômatos Adaptativos. Serão expostos brevemente os conceitos teóricos, as características da forma geral do modelo, bem como as restrições necessárias para a equivalência com modelos mais limitados como os Autômatos de Pilha Estruturado e os Autômatos Finitos.

Meta Reconhecedor Sintático

Os Meta Reconhecedores Sintáticos são ferramentas de auxílio para o desenvolvimento de novos reconhecedores, utilizados em linguagens de programação já existentes ou em linguagens em evolução. Como veremos nesta seção, esta categoria de ferramenta possui várias implementações disponíveis com inúmeras linguagens para especificação do reconhecedor desejado e vários algoritmos para o processamento destas especificações.

Estudaremos a seguir, de maneira geral, a arquitetura de funcionamento dos Meta Reconhecedores Sintáticos .

Modelo Geral de Funcionamento

A Figura 1 ilustra de forma simples, o funcionamento de um compilador. Ela foi extraída de [2], onde também encontramos a seguinte definição:

> *"Simply stated, a compiler is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program."*

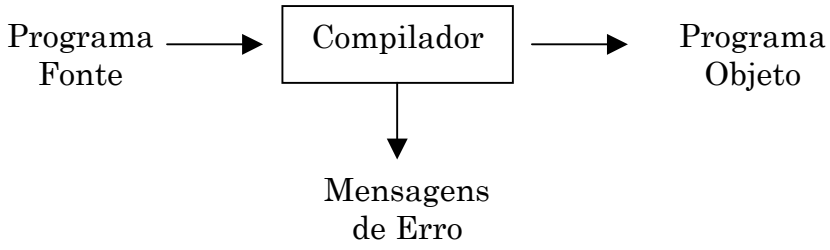

Figura 1 – Um compilador

A operação de um compilador pode ser visualizada como um conjunto de fases dispostas numa ordem específica para atingir seu objetivo ilustrado acima, isto é, traduzir uma especificação ou linguagem para uma outra forma qualquer desejada[2,12]. Comumente falando, os compiladores traduzem programas descritos através de uma linguagem de programação (como C ou Pascal), para uma seqüência de comandos executáveis pelos processadores da máquina hospedeira em que se deseja executar o referido programa.

Cada fase de operação do compilador recebe o texto de entrada numa certa representação interna que será transformada e servirá de entrada para a fase seguinte. A Figura 2 representa o esquema descrito.

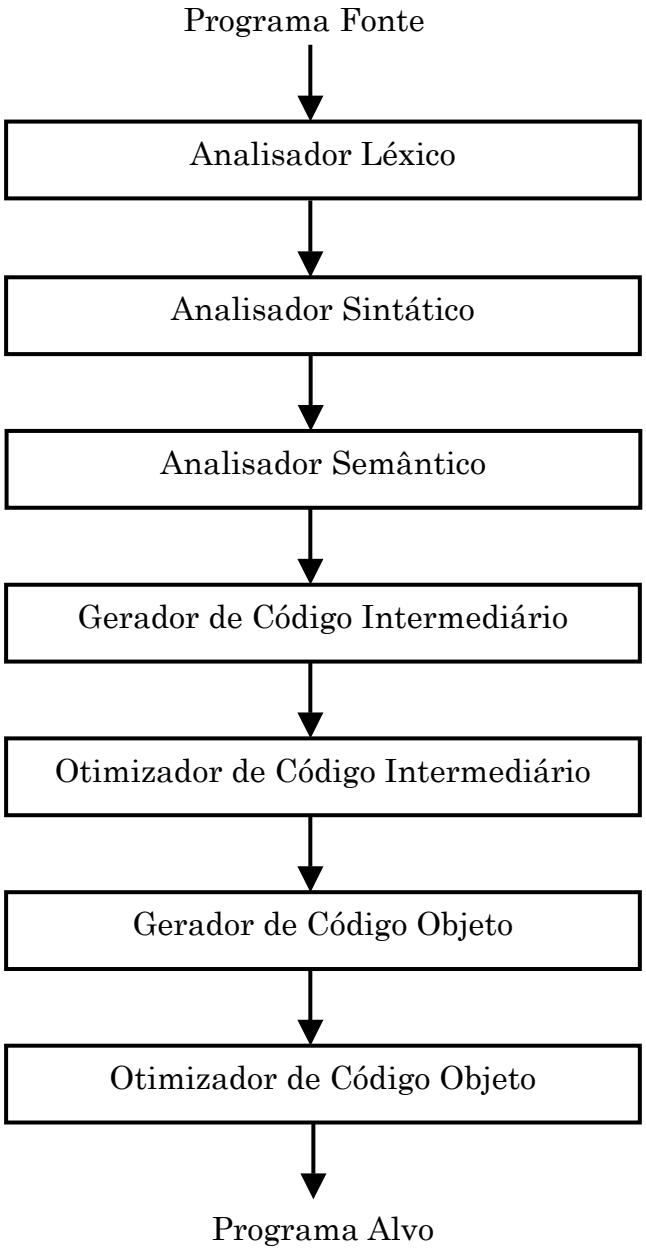

Figura 2 – Fases da compilação

Ainda considerando o esquema da Figura 2 podemos classificar as fases em dois grupos distintos. O primeiro grupo é formado pelas fases responsáveis por reconhecer a linguagem do programa objeto e assegurar o cumprimento de todas as suas regras gramaticais. Como resultado bem sucedido deste processo de análise, obtém-se o programa-fonte no formato de código intermediário. Este grupo é denominado de *Front-end*, e é constituído dos analisadores léxico, sintático, semântico e o gerador de código intermediário.

O segundo grupo tem a responsabilidade de gerar o programa executável a partir do código intermediário, considerando todos os aspectos do sistema operacional e características do equipamento alvo. Este grupo é denominado de *Back-end* e é constituído pelo otimizador de código intermediário, o gerador de código objeto e o otimizador de código objeto. Portanto o grupo *Front-end* depende basicamente da linguagem de programação, enquanto o *Back-end* depende das características do equipamento alvo.

Apresentamos esta classificação para esclarecer que os Meta Reconhecedores Sintáticos são utilizados com o objetivo de auxiliar a geração automática do grupo *Front-end*. As ferramentas que se propõe a gerar todos os blocos de um compilador são denominadas Meta-Compiladores ou *compiler-compilers, compiler-generators, ou ainda translator-writing systems*. Um exemplo deste tipo de ferramenta é o Cigale[11,22], desenvolvido na Universidade de Nice, França.

Freqüentemente, os Meta Reconhecedores Sintáticos possibilitam a geração dos analisadores léxico e sintático. É comum encontrarmos ferramentas distintas para a geração específica dos analisadores léxico e sintático. Neste caso, as ferramentas definem suas próprias linguagens de entrada, bem como uma interface de comunicação entre os analisadores. Como resultado, cria-se um par de ferramentas que produzem analisadores compatíveis, ou seja, o analisador léxico produzido por uma ferramenta pode ser invocado por um analisador sintático produzido por uma outra ferramenta. O par de ferramentas mais famoso mundialmente são os programas Lex e Yacc[23,18,17,24,33]. O programa Lex é responsável por gerar o analisador léxico, e o programa Yacc, o analisador sintático.

Além da geração dos analisadores léxico e sintático, encontramos ferramentas mais poderosas que auxiliam também a geração do analisador semântico. A gramática de atributos[7] é uma das notações encontradas para a descrição de regras semânticas. Cabe aqui citar que muitas das atividades

realizadas por tais analisadores semânticos são de origem puramente sintática[30,34,35]. Entretanto é da responsabilidade dos analisadores semânticos efetuar seu tratamento, em função da impossibilidade de serem associadas ao analisador sintático. Os Autômatos Adaptativos[27,28,] e as chamadas 'Evolving Grammars'[5] modificam este panorama, permitindo que tais tarefas antes realizadas pelo analisador semântico, sejam retiradas das rotinas implementadas manualmente, e sejam colocadas no processo de geração automática do analisador sintático. Desta forma podemos descrever quase todo o grupo *Front-end* de forma gramatical, excluindo-se apenas o gerador de código intermediário.

Um exemplo de ferramenta desenvolvida neste sentido é o Lahey Multibox Parser Generator[9], que utiliza uma extensão do modelo tradicional de analisadores léxico e sintático. Esta ferramenta foi utilizada na construção do compilador Lahey Fortran 90, disponível comercialmente. Ela pode ser usada para a geração do *Front-end* quase completo, ou ainda para gerar um analisador léxico compatível com o Yacc.

Interação com Usuário

A forma mais tradicional de interação com o usuário que os geradores de reconhecedores utilizam é a textual. Isto se deve basicamente ao fato de que o estudo destes dispositivos, e também da teoria associada a eles, datam de épocas anteriores à disponibilização de ambientes gráficos tais como Motif ou Microsoft Windows. O programa Yacc foi, e ainda é, distribuído em inúmeros sistemas operacionais Unix. Portanto ele se utiliza dos meios que o sistema operacional Unix oferecia para interagir com o usuário: terminais de texto. Assim como o Yacc, encontramos hoje um grande número de ferramentas de auxilio à geração de reconhecedores, e até de compiladores, utilizando esta forma de interface.

Embora seja, seguramente, a forma mais utilizada, a interface textual não é a mais adequada quando analisamos aspectos relacionados com a facilidade de manuseio e operação da ferramenta, com a produtividade na

implementação de novas linguagens e com a simplicidade na realização de alterações nas linguagens em fase de manutenção.

Apesar de serem claras e bem conhecidas as vantagens de uma interface gráfica, muitas destas ferramentas não sofreram ainda, ou estão sofrendo atualmente, um processo de melhoria e transporte para os novos ambientes gráficos. Apenas as ferramentas propostas em épocas mais recentes já atendem ao pré-requisito de serem altamente amigáveis. Encontramos na literatura esforços no sentido de atribuir, a certas ferramentas, facilidades de uso durante o processo de desenvolvimento. Podemos citar como um destes esforços a ferramenta Ytracc[13] que se propõe a auxiliar o usuário nas tarefas de depuração dos reconhecedores gerados pelo programa Yacc. Ytracc é um software que permite a visualização do processo de compilação, auxiliando o desenvolvedor da gramática a localizar erros da especificação através da análise da árvore sintática gerada pelo texto de entrada. O desenvolvedor tem a possibilidade de avançar ou de desfazer um reconhecimento já efetuado.

O Ytracc não é um exemplo isolado de expansão das funcionalidade do programa Yacc, pois o próprio autor do programa, S. C. Johnson, produziu um protótipo de uma ferramenta, o y++[16], para gerar programa na linguagem C++[36,10] usando gramáticas de atributos. Podemos citar ainda o esforço feito para que Yacc fosse capaz de reconhecer gramáticas não LR(k)[26], tais como a linguagem C++. O resultado deste esforço é um conjunto de alterações no programa Yacc para o reconhecimento de ambigüidades e gramáticas que necessitem um valor indefinido de *lookahead*.

Além destas ferramentas vistas como extensões de versões anteriores, encontramos iniciativas de desenvolver novos produtos de acordo com as tendências mais atuais das interfaces homem-máquina. Lisa[25] é uma destas iniciativas desenvolvida num ambiente completamente gráfico. Esta ferramenta foi desenvolvida e projetada utilizando a tecnologia de orientação a objetos, e é executada no ambiente Microsoft Windows.

Através da sua utilização, o usuário é capaz de gerar o analisador léxico, o analisador sintático e o analisador semântico de um reconhecedor, tendo-se como entrada uma especificação textual da gramática a ser reconhecida.
Portanto o desenvolvimento de novos conceitos de interface homem-máquina suscitaram esforços para atualizar antigas ferramentas, bem como a maneira de operá-las, sendo que pouco se fez em relação à forma com que se descreve uma gramática como veremos a seguir.
A próxima seção analisa alguns aspectos relacionados com a especificação de entrada, ou seja, a forma como se descreve uma gramática para os geradores de reconhecedores ou compiladores.

Especificação de Entrada

O meio mais popular de se descrever uma nova linguagem é definindo as regras gramaticais que deverão ser obedecidas quando da construção de uma frase daquela linguagem específica. O conjunto de todas as regras de formação de uma linguagem forma sua gramática, comumente expressa na forma textual. Em relação à notação usada para redigir tais expressões gramaticais, as ferramentas apresentam uma certa variedade de opções. Entretanto as notações de Wirth, BNF e BNF estendido detêm um número expressivo de adeptos. As ferramentas que decidem não adotar uma destas notações se utilizam de notações muito semelhantes, com a adição de apenas alguns detalhes proprietários.
Os geradores recebiam como entrada um arquivo texto contendo a gramática de uma certa linguagem. Depois de ter sido analisada e depurada, obtinha-se um reconhecedor, que aceitaria construções daquela linguagem. Este processo continua, de certa forma, válido ainda nas ferramentas atuais. Entretanto, hoje em dia é possível e desejável elaborarmos um formato gráfico para as especificações de entrada, tirando-se vantagem dos ambientes gráficos disponíveis. Tal formato resulta em um ótimo meio para se descrever uma gramática, extremamente legível, e de fácil absorção pelo leitor que começa a tomar contato com a especificação de uma nova linguagem.

Outra forma possível de se descrever uma linguagem é através do conjunto de máquinas, estados e transições do autômato que reconhece a referida linguagem. Esta representação possibilita ao desenvolvedor um altíssimo grau de controle e flexibilidade sobre as ações do reconhecedor, fazendo com que certas otimizações sejam possíveis. Por outro lado, este tipo de descrição tende a apresentar para o usuário uma complexidade mais alta, e o volume físico dessas representações geralmente é maior, se comparado às descrições semelhantes na forma gramatical.

Reconhecedor Gerado

Uma das formas mais utilizadas na geração dos reconhecedores é através de tabelas de transição, as quais serão submetidas posteriormente, quando do reconhecimento de uma cadeia de entrada, a um núcleo do reconhecedor, que efetuará as transições conforme os dados existentes nesta tabela e os átomos da cadeia de entrada.

Os reconhecedores obtidos a partir de uma especificação podem ser completamente descritos através de uma tabela que armazene todas as possibilidades de átomos de entrada válidos para cada estado existente, e também informações complementares tais como, estado inicial onde se devem começar as transições, estados finais nos quais será considerado bem sucedido o reconhecimento da cadeia de entrada e rotinas semânticas associadas às transições.

A tabela de transição, formada pela combinação dos estados existentes e dos átomos de entrada, normalmente é uma matriz esparsa, pois os átomos de entrada válidos para um determinado estado formam um subconjunto muito restrito do alfabeto de entrada(o conjunto de todos os possíveis átomos de entrada do reconhecedor). Portanto tais tabelas podem se utilizar de algoritmos de compressão para serem fisicamente representadas. Estes algoritmos devem ser criteriosamente escolhidos[6] tendo em vista os dois principais, e também conflitantes, requisitos de representação.

Primeiramente a tabela de transições é intensamente manipulada, em torno de 40 por cento do tempo total de reconhecimento de uma cadeia de entrada,

pois ela é utilizada para a implementação do analisador léxico e do analisador sintático. De acordo com Abmann[1], um compilador gasta em torno de 25 por cento do seu tempo na análise léxica, e em torno de 15 por cento na análise sintática[15]. Por isso, o desempenho do reconhecedor está diretamente associado à velocidade de acesso às células desta tabela. Por outro lado, temos o segundo requisito que é a necessidade de compressão, por se tratar de uma tabela esparsa e talvez muito grande tendo em vista o porte do equipamento alvo para a execução do reconhecedor. O conceito de grandeza associado ao tamanho desta tabela, pode ser questionado em função da disponibilidade que encontramos nos dias de hoje de grandes quantidades de memória a um custo acessível. Desta forma a rapidez do processo de reconhecimento se torna prioritária, mas não exclusiva.

Portanto encontramos várias alternativas para a representação das tabelas de transição, e certamente a técnica denominada comb-vector[2] é a mais utilizada pelos geradores de reconhecedores. Porém outros métodos[37] como o chamado graph colouring[8] com algumas variações[3,38,4,20,19] já foram estudados em confronto com o comb-vector[21]. Ainda assim, comb-vector continua sendo a melhor técnica de representação quanto ao seu comportamento, tendo em vista os dois critérios mencionados.

Apesar da ampla aceitação e utilização das tabelas de transição pelos geradores, existe outra opção, muitas vezes adotada para representar um reconhecedor. Esta opção consiste em escrever trechos de código cada qual representando o comportamento do autômato em um dado estado. Neste trecho de código, são realizadas as comparações necessárias para verificar se o átomo correntemente disponível na cadeia de entrada pode ser considerado válido para aquele estado. Caso o átomo seja válido, determina-se o estado destino, e a execução é então transferida para o trecho de código que codifica aquele estado-destino. Em função da necessidade de se alterar abruptamente o local da linha de execução, ou seja, o trecho de código sendo executado, a linguagem de programação utilizada na implementação deste método deve possuir um mecanismo similar ao comando goto da linguagem

C. Assim cada estado possui um rótulo associado para o qual é desviada a execução quando o referido estado se tornar o estado corrente.

Este método é freqüentemente denominado de *hard-coded*, por causa da implementação das transições como linhas de código-fonte do reconhecedor[31,14]. Uma das características mais importantes deste método de representação do reconhecedor é a velocidade. Estudos[31] mostram que o mesmo pode chegar a ser até 10 vezes mais rápido do que o mesmo reconhecedor representado por tabelas de transição. A desvantagem é que o tamanho do reconhecedor tende a aumentar rapidamente conforme cresce o tamanho da gramática. Uma alternativa seria a utilização desta técnica para representar o analisador léxico e utilizar as tabelas de transição para representar o analisador sintático, já que o analisador léxico costuma ser pouco modificado em função de alterações no tamanho da gramática, além de consumir mais tempo de processamento do que o analisador sintático[1,15].

Autômatos Adaptativos

Os Autômatos Adaptativos[27,28] são a base formal em que se apoia este trabalho. Eles podem ser vistos como uma evolução ou extensão dos autômatos de pilha estruturados, os quais, por sua vez, correspondem também a uma extensão dos autômatos finitos. Esta seção tem como objetivo rever resumidamente os conceitos associados aos Autômatos Adaptativos.

Modelo Geral

Os autômatos de pilha estruturados[29] podem ser vistos como conjuntos de sub-máquinas, que operam de forma semelhante aos autômatos finitos, com o acréscimo de uma pilha de estados. A chamada transferência de controle de uma sub-máquina por outra se dá sempre que for executada uma transição de chamada de sub-máquina. Nesta ocasião, antes de desviar para o estado inicial da sub-máquina chamada, é empilhada uma indicação do estado para onde o retorno deverá ser efetuado ao final da operação da sub-máquina chamada. Este retorno ocorre quando, estando a sub-máquina

chamada em um estado final, não houver qualquer outra transição interna possível, sendo então executada uma transição de retorno à sub-máquina chamadora. O reconhecimento de uma cadeia de entrada é obtido através da execução das transições válidas das sub-máquinas que compõem o autômato, e consideramos que uma cadeia foi corretamente reconhecida quando a mesma tiver sido totalmente consumida, desde que o estado corrente seja um dos estados finais da sub-máquina inicial do autômato, e que a pilha esteja vazia.

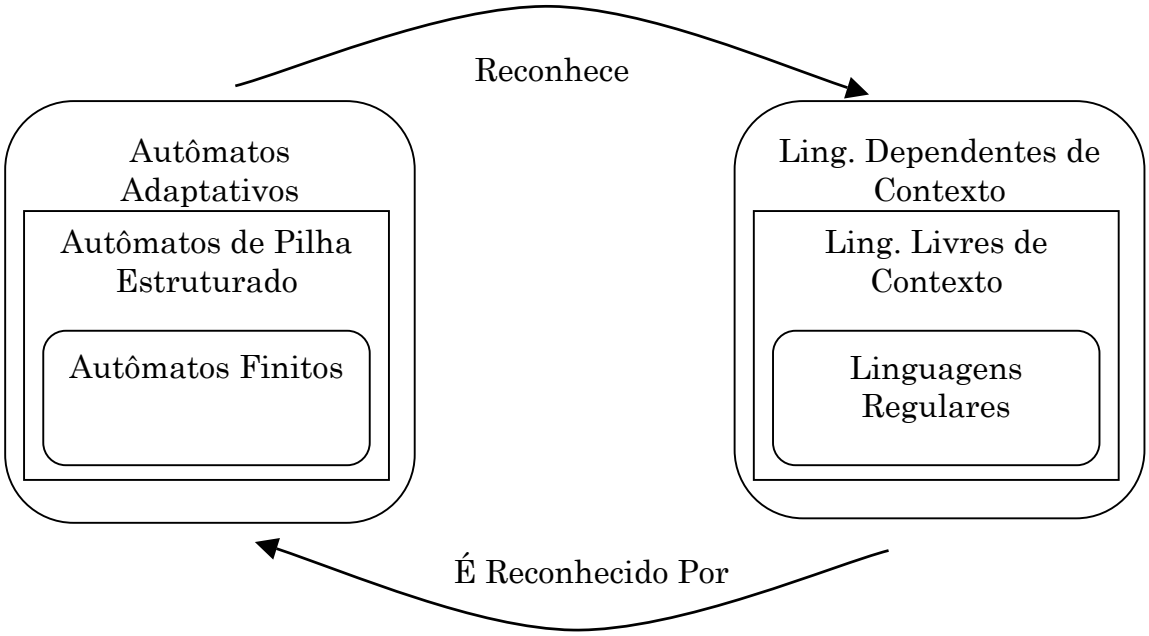

Figura 3 – Relacionamento entre Autômatos e Linguagens

Autômatos finitos, isoladamente, são capazes de reconhecer apenas linguagens regulares. Já os autômatos de pilha estruturados estendem essa capacidade ao reconhecimento de linguagens livres de contexto. Linguagens dependentes de contexto podem ser tratadas por meio dos Autômatos Adaptativos, que são, por sua vez, extensões dos autômatos de pilha estruturados. A Figura 3 ilustra a relação existente entre os autômatos, a relação existente entre as linguagens que estes reconhecem e a associação entre as linguagens e os autômatos que são capazes de reconhecê-las.

Autômatos adaptativos podem ser vistos como autômatos de pilha estruturados que têm a possibilidade de auto-modificação. No início de sua operação, os autômatos adaptativos se apresentam como um conjunto de máquinas de estados, numa configuração inicial. As dependências de contexto da linguagem reconhecida pelos autômatos adaptativos são tratadas através das transições adaptativas. Estas transições, à medida em que são executadas, armazenam na estrutura do autômato adaptativo, informações sobre a cadeia de entrada consumida na forma de novos estados e novas transições. As transições adaptativas são transições as quais se associa pelo menos uma chamada de função adaptativa. A função adaptativa é o agrupamento lógico de ações adaptativas. Estas podem ser de três tipos: ações de consulta ao conjunto de transições existentes, ações de inclusão de uma nova transição e ações de eliminação de uma transição já existente. São as ações adaptativas elementares que efetivamente alteram a configuração das máquinas de estados, sendo assim, responsáveis por armazenar informações sobre a cadeia de entrada, como parte do tratamento das dependências de contexto.

Em relação à operação dos autômatos adaptativos temos a seguinte descrição:

"A operação do autômato adaptativo pode ser interpretada como uma seqüência de evoluções sucessivas de um autômato de pilha estruturado inicial, em que se baseia, evoluções essas promovidas pela execução de ações adaptativas: partindo da configuração original, o autômato de pilha que implementa o autômato adaptativo em um dado instante opera normalmente, consumindo sucessivos símbolos da cadeia de entrada, até que seja executada uma transição adaptativa. As alterações impostas por esta evoluem o autômato de pilha estruturado corrente, formando um novo autômato, que irá continuar a tratar o restante da cadeia de entrada até que nova ação adaptativa seja executada, e assim sucessivamente, até que a cadeia de entrada seja totalmente consumida e o autômato atinja um estado final, dizendo-se neste caso que a cadeia de entrada foi aceita pelo autômato. Impasses de qualquer

natureza na operação do autômato correspondem à rejeição da cadeia de entrada."

Texto extraído de [32]

Vamos exemplificar esta operação através da resolução do seguinte problema: reconhecer uma cadeia de entrada descrita pela regra $a^n b^n c^n$, com $n > 0$. Ou seja, o autômato deverá reconhecer as cadeias "abc", "aabbcc", "aaabbbccc", e assim por diante.

Vale a pena observar que tal problema não pode ser resolvido apenas com a utilização dos autômatos finitos nem dos autômatos de pilha estruturados, em função do caráter dependente de contexto da linguagem em questão.

Existem pelo menos duas soluções para o problema proposto. A primeira será apresentada a seguir. A segunda será descrita na seção Exemplo 3 - Expressão $a^n b^n c^n$, com $n>0$, na página 87. A diferença entre as soluções apresentadas é o fato de o autômato, na sua configuração final, após o tratamento, bem ou mal sucedido, de uma cadeia de entrada, não poder ser reutilizado para o reconhecimento de uma nova cadeia de entrada. Por isso, a segunda solução é denominada reentrante, pois admite reutilização.

Consideremos o autômato da Figura 4, solução para o problema proposto para o caso específico de n=1.

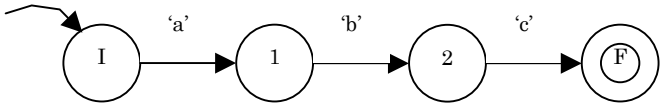

Figura 4 – Autômato inicial para $a^n b^n c^n$, com n=1

Uma das soluções gerais consiste em adicionar dois novos estados entre os estados 1 e 2, para cada átomo 'a' consumido no estado 1, através da chamada de uma função adaptativa. Esta função adaptativa cria dois novos estados, por exemplo 3 e 4, e terão uma transição consumindo 'b' do estado 1 para o estado 3, outra transição consumindo 'b' do estado 3 para o estado 4 e

mais uma transição consumindo ‘c’ do estado 4 para o estado 2. Assim, o autômato descrito pode ser representado pela Figura 5.

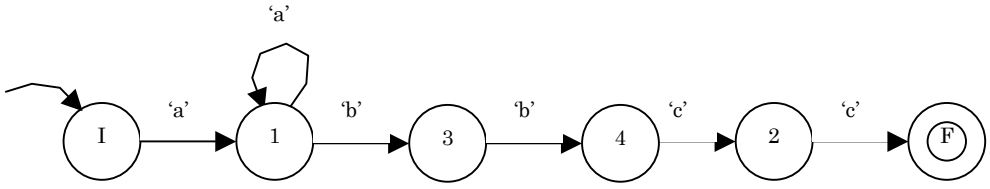

Figura 5 - Autômato depois de consumido o segundo 'a'

Para concluir o autômato ainda existe um detalhe a ser observado. Depois de realizada a adição dos novos estados 3 e 4, a posição onde deve ser incluído os próximos dois novos estados também deve ser atualizada. Esta informação fica registrada através dos parâmetros passados à função adaptativa. Na primeira vez que a função é chamada são passados os estados 1 e 2. Na segunda chamada da função adaptativa devem ser passados os estados 3 e 4, e entre estes estados é que devem ser inseridos os dois novos estados, por exemplo, os estados 5 e 6.

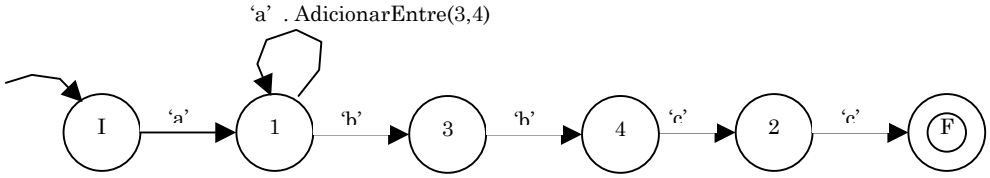

Figura 6 - Autômato completo depois de consumido o segundo 'a'

A função adaptativa AdicionarEntre(3,4) deve implementar as seguintes operações:

- Criar dois novos estados
- Remover a transição consumindo ‘b’ entre os estados representados pelo 1º e o 2º parâmetros
- Adicionar uma transição consumindo ‘b’ entre os estados representados pelo 1º parâmetro e o 1º estado criado.
- Adicionar uma transição consumindo ‘c’ entre o 1º e o 2º estado criados.

- Adicionar uma transição consumindo ‘c’ do 2º estado criado para o estado representado pelo 2º parâmetro.
- Remover a transição consumindo ‘a’, com chamada da função adaptativa AdicionarEntre(1º parâmetro, 2º parâmetro), do estado 1 para ele mesmo.
- Adicionar uma transição consumindo ‘a’, com chamada da função adaptativa AdicionarEntre(1º estado criado, 2º estado criado), do estado 1 para ele mesmo.

O Anexo III – Reconhecedor da expressão $a^n b^n c^n$ - apresenta formalmente esta solução descrita na linguagem RSW.

Finalizamos esta seção com algumas considerações sobre os autômatos adaptativos:

> “O modelo do autômato adaptativo tem uma característica muito desejável: sendo geral, e sendo também derivado do autômato de pilha estruturado por simples agregação de recursos, sem alterar o formalismo deste, torna possível que seja utilizado um único formalismo para a especificação e tratamento de linguagens das diversas categorias, sem que seja necessário usar formalismos diferentes.
>
> Do ponto de vista prático, uma única ferramenta pode ser assim utilizada para o tratamento de linguagens regulares, livres de contexto ou dependentes de contexto, sem que o seu usuário tenha a necessidade de assimilar outras notações, ou de operar diferentes ferramentas.
>
> Outra vantagem marcante é que toda a simplicidade e eficiência que caracterizam os autômatos finitos e os autômatos de pilha estruturados continuam a ser usufruídos pelos usuários dos autômatos adaptativos, pois a forma de operação destes só se afasta da anterior nas específicas ocasiões, usualmente pouco freqüentes, em que ocorrem as alterações estruturais, permanecendo rigorosamente idêntica nos demais casos.”
>
> Texto extraído de [32]

Capítulo III - Projeto

Este capítulo apresenta uma descrição mais detalhada do projeto da ferramenta RSW, bem como da linguagem reconhecida por seu compilador. Denominamos tal linguagem de Linguagem RSW. Após comentarmos a proposta do trabalho realizado, seguiremos para a seção da descrição gramatical da Linguagem RSW, na notação de Wirth. Em seguida é realizada uma análise de correspondência entre a notação algébrica dos autômatos adaptativos, apresentada em [28], e a Linguagem RSW. Esta análise tem como objetivo identificar como as construções expressas na notação algébrica podem ser definidas na Linguagem RSW.

Proposta do Trabalho

O Ambiente Integrado de Desenvolvimento de Reconhecedores Sintáticos ou Meta Reconhecedor Sintático para Windows (RSW) é uma implementação de parte da teoria apresentada em [28]. Seu objetivo é servir como uma ferramenta prática para a geração de reconhecedores que se utilizem dos autômatos adaptativos, bem como facilitar o processo de desenvolvimento através de uma Interface Gráfica como meio de interação com o usuário.

A parte não implementada da teoria diz respeito à manipulação livre da pilha do autômato adaptativo. A teoria não impede que se façam consultas e alterações no topo da pilha, sem que isto seja necessariamente a conseqüência da execução de uma transição de chamada ou retorno de uma sub-máquina. Em termos práticos, esta característica possibilitaria, por exemplo, que uma sub-máquina, ao término de sua operação, ou em qualquer outro estado possível, não retornasse ao estado que a invocou, e sim a um estado qualquer do autômato. Este tipo de utilização da pilha é de limitado uso prático. Por estas razões, optou-se por implementar a pilha de

estados através da pilha de execução do programa, ganhando-se consideravelmente nos aspectos de eficiência e desempenho do reconhecedor. Adicionalmente, pode-se implementar um esquema alternativo de pilha secundário, para o requisito de consulta ao topo da pilha, com fins de *lookahead*,

Especificação da Linguagem RSW

No Anexo IV – Gramática da Linguagem RSW, na notação de Wirth, é descrita na forma de gramática a linguagem RSW que é reconhecida pelo compilador RSW.

A linguagem RSW permite a definição num único módulo, de uma sub-máquina composta de um conjunto de produções e funções adaptativas. Um projeto RSW é composto de pelo menos um módulo contendo a descrição de uma sub-máquina. Cada sub-máquina possui um nome que a identifica univocamente perante as outras sub-máquinas do autômato. Após o nome, são definidas as propriedades da sub-máquina, tais como o conjunto de átomos de retorno e a sub-máquina de entrada.

O conjunto de átomos de retorno especifica os possíveis átomos de retorno para a sub-máquina em questão. Um átomo de retorno é aquele empilhado pela transição de retorno de sub-máquina, antes de efetivamente retornar a execução ao estado constante no topo da pilha de estados. Especificando mais de um átomo de retorno para uma sub-máquina, possibilita ao estado que a invoca, obter informação de qual das possíveis construções foi utilizada pela sub-máquina chamada para o reconhecimento da cadeia de entrada até aquele ponto.

A outra propriedade definida, a sub-máquina de entrada, especifica qual a sub-máquina que deve ser acionada toda vez que a sub-máquina em questão necessitar de um átomo de entrada. Assim, sub-máquinas que representem analisadores sintáticos, definem como sub-máquina de entrada uma sub-máquina que represente o analisador léxico. Este por sua vez, define como

sub-máquina de entrada a sub-máquina do sistema denominada 'CadeiaEntrada'. A sub-máquina 'CadeiaEntrada' encapsula as operações do arquivo físico, tais como abertura, leitura, armazenamento em memória e fechamento.

A palavra reservada 'producao' inicia o bloco de definição das produções. Cada produção é finalizada com o átomo '.'. A palavra reservada 'funcao' marca o término do bloco de definição das produções e o início do bloco de definição das funções adaptativas. A Figura 7 ilustra os blocos comentados até aqui, de um módulo RSW. Os blocos de definição das produções e funções adaptativas são descritos detalhadamente na próxima seção, Correspondência entre a notação algébrica e a Linguagem RSW.

submaquina Identificador (AtomoRetorno1, AtomoRetorno2) entrada CadeiaEntrada
producao %Bloco de definição das produções
funcao %Bloco de definição das funções adaptativas

Figura 7 - Blocos de um módulo RSW

Correspondência entre a notação algébrica e a Linguagem RSW

No intuito de facilitar a comparação entre a notação algébrica e a linguagem RSW, considere as expressões regulares abaixo.

Produções Simples =>
‘(’ [‘^’ ‘,’] <estado0> ‘,’ <entrada0> ‘)’ [< Função >] ‘->’ (<estado1> | ‘(’ [‘^’ [<estado0>] ‘,’] <estado1> [‘,’ <entrada1>] ‘)’) [< Função >]

Produções de Retorno =>
‘(’ <estado0> ‘)’ |
‘(’ ‘^’ ‘*’ ‘,’ <estado0> ‘)’ [< Função >] ‘->’ ‘(’ ‘^’ ‘,’ ‘*’ ‘,’ <entrada1> ‘)’ [<Função>]
onde
< Função > => ‘:’ <identificador da função> [‘(’ <argumento> { ‘,’ <argumento> } ‘)’]

Vamos dividir as produções em dois tipos distintos: as produções com transições simples para estado destino dentro ou não da sub-máquina corrente, e as produções que executam o retorno para a sub-máquina anterior ou chamadora. As expressões regulares acima apresentam as respectivas regras de formação para os tipos de produções definidos.

Definição das Produções

Ambas as notações determinam por convenção um meta-símbolo para a representação do conteúdo genérico da pilha, a saber: γ para a notação algébrica e ^ para a linguagem RSW. Para o conteúdo da cadeia de entrada, a notação algébrica determina o símbolo α e a linguagem RSW adotou a ausência de qualquer símbolo, pois o conteúdo representado por este símbolo é irrelevante para a decisão da aplicação ou não de uma produção, e não

produziria nenhum benefício secundário em relação à implementação da linguagem RSW. O símbolo ^ ou γ é relevante quanto à decisão da aplicação da produção, portanto, ele foi adotado na implementação da linguagem RSW.

A notação algébrica apresenta elementos para um conjunto de produções de autômato adaptativo em um dos seguintes formatos:

$$\mathbf{(e , s) : A , \rightarrow e' , B}$$

ou $\mathbf{(\gamma\, g , e , s\, \alpha) : A , \rightarrow (\gamma\, g' , e' , s'\, \alpha) , B}$

sendo que o primeiro abrevia o segundo para o caso particular em que g = g' = s' = ε , representando assim uma transição simples de um estado origem para um destino sem considerar o conteúdo da pilha, e sem a inserção de símbolos na cadeia de entrada.

Item	s'	g	g'	e'
1	= ε	≠ ε,	= ε	= g
2	= ε	= ε	≠ ε, = **e**	≠ ε
3	= ε	= ε	= ε	≠ ε
4	≠ ε	≠ ε	= ε	= g
5	≠ ε	= ε	≠ ε, = **e**	≠ ε
6	≠ ε	= ε	= ε	≠ ε
7	Qualquer	≠ ε, = g'	≠ ε, = g	≠ ε
8	Qualquer	≠ ε, ≠ g'	≠ ε, ≠ g	≠ ε, ≠ g
9	Qualquer	≠ ε	= ε	≠ g

Tabela 1 - Possibilidades analisadas de correspondência

Para o primeiro formato não existe a necessidade de nenhuma regra de conversão entre as notações, pois tal conversão se dá diretamente. Tendo em vista a sintaxe adotada na Linguagem RSW, temos o seguinte

mapeamento do primeiro formato de produção da notação algébrica para a Linguagem RSW:

- O símbolo e corresponde ao símbolo <estado0> que representa o estado de origem da produção em questão
- O símbolo s corresponde ao símbolo <entrada0> que representa o átomo a ser consumido pela aplicação da transição em questão
- O símbolo e' corresponde ao símbolo <estado1> que representa o estado de destino da produção em questão

O segundo formato das produções possibilita uma série de alternativas que iremos analisar individualmente, expressando as construções correspondentes da linguagem RSW. A Tabela 1 apresenta as possibilidades analisadas de correspondência entre a notação algébrica e a linguagem RSW.

1. $s' = \varepsilon$, $g \neq \varepsilon$, $g' = \varepsilon$ e $e' = g$. Nesta configuração temos uma transição de retorno de sub-máquina. Isso indica que a aplicação da produção está vinculada à existência de um estado no topo da pilha, sendo que este estado será o próximo a se tornar o estado corrente. Em outras palavras, esta transição corresponde ao retorno de uma sub-máquina ao estado que a invocou, ou ao estado empilhado naquela ocasião.

   Há duas situações particulares a serem analisadas para a conversão de sentenças da notação algébrica para RSW. A primeira ocorre quando se tem uma transição de retorno independente da cadeia de entrada ou do átomo disponível para consumo. A segunda situação ocorre quando se deve consumir o átomo disponível da cadeia de entrada e posteriormente efetuar o retorno ao estado apropriado.

   Para denotar, na linguagem RSW, uma transição de retorno pertencente ao primeiro caso, temos:

   $\mathbf{(\gamma\ g\ ,\ e\ ,\ \alpha\ ) \rightarrow\ (\gamma\ ,\ g\ ,\ \alpha)}$ - na notação algébrica

   correspondendo a

(<estado0>). - na linguagem RSW

***onde <estado0> equivale a* e**

Já para o segundo caso, onde existe a necessidade de se consumir um átomo antes do retorno ao estado apropriado, a solução é criar um estado intermediário para que seja efetuado consumo do átomo da cadeia de entrada e em seguida retornar ao estado apropriado. Portanto a produção

(γ g , e , sα) → (γ , g , α) - na notação algébrica

corresponde ao conjunto

(<estado0> , <entrada0>) → <estado1>.

(<estado1>). - na linguagem RSW

***onde <estado0> equivale a* e**

<estado1> equivale ao estado intermediário

***<entrada0> equivale a* s**

Note-se que na especificação do funcionamento dos autômatos adaptativos, é necessário que seja empilhado um átomo de retorno[1] pela sub-máquina que está finalizando seu reconhecimento. Para as expressões acima da Linguagem RSW, quando o átomo de retorno não for especificado explicitamente, utiliza-se, por convenção, o primeiro átomo definido logo após o nome da sub-máquina. O item 4 discute o caso em que se define mais de um átomo de retorno, devendo-se, portanto, especificar qual dos átomos de retorno utilizar quando da transição de retorno da sub-máquina.

A produção apresentada para executar o retorno de uma sub-máquina não considera o valor do topo da pilha para decidir se será ou não

[1] O átomo de reconhecimento é aquele empilhado por uma sub-máquina quando do reconhecimento bem sucedido de uma cadeia de entrada. Uma sub-máquina pode possuir mais de um átomo de reconhecimento. Cada átomo representa um caminho de transições executadas pela sub-máquina para a obtenção do reconhecimento da cadeia de entrada.

aplicada. Sendo <estado0> o estado corrente, e não havendo a possibilidade de aplicação de uma produção de transição simples ou de chamada de sub-máquina, então a produção de retorno será aplicada, independentemente do estado que invocou a sub-máquina corrente. A execução passará ao estado que estiver registrado no topo da pilha, logo após ter sido empilhado o átomo de retorno.

Mais adiante também, no item 6, analisaremos o outro caso em que é necessário o empilhamento de um átomo na cadeia de entrada antes do retorno da sub-máquina.

2. s' = ε , g = ε, g' = e, g' ≠ ε e e'≠ ε. Nesta configuração temos uma transição de chamada de sub-máquina. Isso indica que a aplicação da produção produzirá o empilhamento do estado de origem, convertendo em estado corrente o estado inicial da sub-máquina que estiver sendo chamada. Após a sub-máquina chamada executar as transições possíveis em função do conteúdo da cadeia de entrada, esta deve empilhar um átomo de reconhecimento, que será consumido pela estado de origem para a determinação do estado destino, dentro da sub-máquina corrente.

   Portanto analisando uma chamada de sub-máquina sem o consumo de átomo da cadeia de entrada antes da chamada, obtém-se a seguinte sentença na notação algébrica:

   (γ , e , α) → (γ e , e' , α)

   corresponde a

   (<estado0> , <entrada0>) → (^ <estado0>, <submaquina0> , <entrada0>).

   (<estado0> , <entrada1>) → <estado1>.

   onde ***<estado0> equivale a*** **e**

   <submaquina0> equivale a **e'**

   <entrada0> equivale ao símbolo utilizado para "lookahead"

<entrada1> equivale ao átomo de retorno empilhado por <submaquina0>

na linguagem RSW.

Note a existência do elemento <entrada0> que representa o conteúdo da cadeia de entrada na configuração original e final. Esta transição não altera a configuração da cadeia de entrada, entretanto, ela está condicionada à existência do símbolo <entrada0> na cadeia de entrada para ser aplicada. A primeira produção implementa o mecanismo de "*lookahead*", mais a chamada da sub-máquina desejada, que terá seu estado inicial como estado destino. A segunda será aplicada quando do retorno da sub-máquina chamada, e consumirá o símbolo <entrada1> por ela empilhado.

Outra possibilidade para a chamada de uma sub-máquina é a necessidade de se consumir o átomo disponível da cadeia de entrada antes de transformar o estado inicial da mesma no estado corrente. A solução para esta situação é a utilização de um estado intermediário, que será acionado quando do consumo do átomo desejado. Este estado efetuará a chamada da sub-máquina e o consumo do átomo por ela empilhado, ativando o estado destino apropriado. Portanto para o caso acima exposto a conversão da sentença:

$(\gamma$, e , sα) $\rightarrow$ (γ e , e' , α)

em notação algébrica para a linguagem RSW será

(<estado0> , <entrada0>) $\rightarrow$ <estado1> .

(<estado1> , <entrada1>) $\rightarrow$ (^ <estado1>, <submaquina0> , <entrada1>).

(<estado1> , <entrada2>) $\rightarrow$ <estado2>.

onde ***<estado0> equivale a* e**

<estado1> equivale ao estado intermediário

***<submaquina0> equivale a* e'**

<entrada0> equivale ao átomo a ser consumido

<entrada1> equivale ao átomo de "lookahead"

<entrada2> equivale ao átomo de retorno empilhado por <submaquina0>

3. s' = ε , g = ε, g' = ε e e' ≠ ε. Esta configuração já foi analisada no início da nossa discussão e representa o primeiro formato da notação algébrica. A conversão é simples e direta, pois as notações em questão possuem os mesmos símbolos e significados para expressar tal configuração.

   (e , sα) → (e' , α)

   ou

   (e , sα) → e'

   converte-se para

   (<estado0> , <entrada0>) → <estado1> .

   ***onde <estado0> equivale a* e**

   ***<estado1> equivale a* e'**

   ***<entrada0> equivale ao átomo a ser consumido* s**

4. s' ≠ ε , g ≠ ε, g' = ε e e' = g. Esta configuração representa o retorno de uma sub-máquina. Toda sub-máquina tem associada a ela possíveis átomos de retorno, para indicar o que foi reconhecido pela mesma. No item 1 analisamos o caso em que o átomo de retorno não era especificado explicitamente. A diferença desta configuração para a do item 1 é a declaração explicita do átomo de retorno, portanto, para a aplicação desta produção é necessário que seja empilhado o átomo por ela especificado e não outro qualquer.

   Desta maneira, para a sentença

   (γ g , e , α) → (γ , g , s' α)

   temos as seguintes produções

   (^ *, <estado0>) → (^ , * , <saida0>).

   ***onde <estado0> equivale a* e**

***<saida0> equivale a* s'**

E no caso em que seja necessário consumir um átomo antes do retorno, teremos

(γ g , e , sα) -> (γ , g , s' α) - na notação algébrica

correspondendo ao conjunto

(<estado0> , <entrada0>) -> <estado1>.

(^ *, <estado1>) -> (^ , * , <saida0>).

***onde* <estado0> *equivale a* e**

<estado1> equivale ao estado intermediário

***<entrada0> equivale a* s**

***<saida0> equivale a* s'**

As observações feitas no item 1 também são válidas para este caso. Temos ainda que ressaltar o significado do meta-símbolo *. Ele é utilizado para representar o valor que está presente no topo da pilha, pois como já discutimos, este valor não está disponível diretamente.

5. s' ≠ ε , g = ε, g' = e, g' ≠ ε e e'≠ ε. A configuração em questão representa a chamada de uma sub-máquina semelhante à do item 2, porém, com o acréscimo de uma ação: o empilhamento de um átomo na cadeia de entrada. Cabe ainda lembrar que na chamada de uma sub-máquina o átomo da cadeia de entrada não é consumido. Este é utilizado, quando explicitado, apenas na operação de "lookahead" que determina a aplicação da produção que representa a chamada da sub-máquina. Por isso se empilharmos um átomo antes de efetuarmos a chamada da sub-máquina, este será o primeiro átomo a ser analisado pela sub-máquina chamada. Consumindo-se o átomo empilhado, o próximo será o átomo utilizado na operação de "look-ahead".

   Portanto na notação algébrica, a sentença

   (γ , e , α) → (γ e , e' , s' α)

corresponde a

(<estado0> , <entrada0>) → (^ <estado0>, <submaquina0> , <entrada1>).

(<estado0> , <entrada2>) → <estado1>.

onde ***<estado0> equivale a* e**

***<submaquina0> equivale a* e'**

<entrada0> equivale ao átomo de "lookahead"

***<entrada1> equivale a* s'**

<entrada2> equivale ao átomo empilhado quando do retorno da sub-máquina chamada

No caso em que há a necessidade do consumo do átomo da cadeia de entrada antes da chamada à sub-máquina, adota-se a solução do estado intermediário. Define-se uma transição do estado corrente para o estado intermediário consumindo o átomo desejado, e posteriormente define-se a produção de chamada da sub-máquina.

6. s' ≠ ε , g = ε, g' = ε e e' ≠ ε. Esta configuração dá origem a produções de transição simples, que consomem o átomo disponível na cadeia de entrada, e empilham outro átomo, representado pelo símbolo s'.

   A conversão da sentença

   (e , s α) -> (e' , s' α)

   em notação algébrica para a linguagem RSW eqüivale a

   (<estado0> , <entrada0>) -> (<estado1>, <entrada1>).

   onde ***<estado0> equivale a* e**

   ***<estado1> equivale a* e'**

   <entrada0> equivale a s*, átomo a ser consumido*

   <entrada1> equivale a* s', átomo a ser empilhado na posição corrente da cadeia de entrada*

   Em outras palavras, a aplicação da produção acima troca o átomo da cadeia de entrada <entrada0> pelo átomo <entrada1>.

7. g ≠ ε , g' ≠ ε e g' = g, e' ≠ ε, para qualquer s'. A aplicação de uma produção que assuma este tipo de configuração estará condicionada à existência do símbolo g no topo da pilha, isto é, a sub-máquina corrente deverá ter sido chamada por uma transição que empilhou o estado representado por g. Após a aplicação da produção, o topo da pilha permanece inalterado, contendo o símbolo g', que por sua vez é igual a g. As variações possíveis dentro desta configuração correspondem ao consumo ou não do átomo, bem como o empilhamento eventual de um átomo na cadeia de entrada.

   Portanto este tipo de produção expressa na notação algébrica resulta na seguinte sentença:

   (γg, e , s α) -> (γg, e' , s' α)

   A construção deste tipo de produção na linguagem RSW está vinculada à implementação da pilha explícita para consulta, e poderá ser expressa através da seguinte sentença:

   (^<submaquina0>, <estado0> , <entrada0>) -> (^<submaquina0>, <estado1>, <entrada1>).

   onde ***<submaquina0> equivale a*** **g**

   <estado0> equivale a **e**

   <estado1> equivale a **e'**

   <entrada0> equivale a s*, átomo a ser consumido*

   <entrada1> equivale a **s'***, átomo a ser empilhado na posição corrente da cadeia de entrada*

8. g ≠ ε , g' ≠ ε e g' ≠ g, e' ≠ ε e e' ≠ g, para qualquer s'. Esta configuração possibilita produções que trocam o valor do topo da pilha. Para ser aplicada, o topo da pilha deve conter o valor representado por g, e após aplicada o topo da pilha conterá o valor representado por g'. É possível também variar as produções para que seja ou não consumido o átomo disponível da cadeia de entrada, ou então empilhar outro átomo.

Este tipo de produção, expressa na notação algébrica, resulta na seguinte sentença:

(γg, e , s α) -> (γg', e' , s' α)

Como já mencionado no início deste capítulo, a atual implementação da ferramenta RSW não possibilita a construção de produções que alterem o topo da pilha, sem que seja uma operação de chamada ou retorno de sub-máquina. Portanto este tipo de construção não apresenta correspondência na linguagem RSW.

9. $g \neq \varepsilon$, $g' = \varepsilon$, $e' \neq g$, para qualquer s'. Esta configuração é semelhante aos itens 1 e 4, diferindo apenas no valor do estado destino. Nos itens citados o estado de destino assumia o valor especificado para o topo da pilha, gerando assim, uma produção de retorno de sub-máquina. A produção resultante desta configuração realiza a remoção do topo da pilha, sem que haja um retorno de sub-máquina, pois o estado destino não assume o valor do topo da pilha da configuração inicial, e nem é realizada uma operação de *lookahead* no topo da pilha Esta construção, a exemplo da anterior, não apresenta correspondência na linguagem RSW.

A Tabela 2 resume as possibilidades analisadas da notação algébrica e seu significado para a Linguagem RSW.

Item	s’	g	g’	e’	Significado para RSW	Exemplo
1	= ε	≠ ε	= ε	= g	Retorno de sub-máquina	(^*, <estado1>) -> (^, *).
2	= ε	= ε	≠ ε	≠ ε	Chamada de sub-máquina	(<est1>, <atomo>) -> (^<est1>, <submaq>). (<est1>, <atRetorno>) -> <est2>.
3	= ε	= ε	= ε	≠ ε	Transição simples com consumo de átomo	(<estado1>, <atomo>) -> <estado2>.
4	≠ ε	≠ ε	= ε	= g	Retorno de sub-máquina com empilhamento de átomo na pilha	(^*, <estado1>) -> (^, *, <atRetorno>).

5	≠ ε	= ε	≠ ε, = **e**	≠ ε	Chamada de sub-máquina com empilhamento de átomo na pilha	(<est1>, <atomo>) -> (^<est1>, <submaq>, <atEmpilhar>). (<est1>, <atRetorno>) -> <est2>.
6	≠ ε	= ε	= ε	≠ ε	Transição simples com empilhamento de átomo na pilha	(<est1>, <atomo>) -> (<est2>, <atEmpilhado>).
7	Qualquer	= g’	= g	≠ ε	Transição simples com ou sem empilhamento de átomos vinculada à consulta ao topo da pilha	(^<submaq>, <est1>, <atomo>) -> (^<submaq>, <est2>, <atRetorno>).
8	Qualquer	≠ ε e ≠ g’	≠ ε e ≠ g	≠ ε, ≠ g	Transição simples com ou sem empilhamento de átomos e mudança no topo da pilha	Sem correspondência

9	Qualquer	$\neq \varepsilon$	$= \varepsilon$	$\neq g$	Transição com ou sem empilhamento de átomos e remoção do estado do topo da pilha	Sem correspondência

Tabela 2 - Resumo da correspondência entre a notação algébrica e a linguagem RSW

Definição das Ações Adaptativas

Nesta seção será analisada a correspondência entre as chamadas das funções adaptativas nas produções da notação algébrica e na Linguagem RSW.

Ambas as notações prevêem a chamada de funções adaptativas em dois momentos distintos da aplicação de uma produção:

- Imediatamente antes da alteração da configuração de origem para a de destino, denominada chamada adaptativa anterior.
- Logo após a alteração para a configuração de destino, denominada chamada adaptativa posterior.

Nos dois momentos previstos, a notação para a chamada de uma função adaptativa é a mesma, distinguindo-se apenas em função da posição relativa dentro da produção em que está localizada a chamada. Para ilustrar, as chamadas anterior e posterior estão representadas abaixo, na notação algébrica, pelos símbolos **A** e **B** respectivamente.

$$\mathbf{(e , s) : A , \rightarrow e' , B}$$

ou $\mathbf{(\gamma\, g , e , s\, \alpha) : A , \rightarrow (\gamma\, g' , e' , s'\, \alpha) , B}$

Tais chamadas das funções adaptativas (**A** e **B**) devem ser denotadas no formato seguinte:

$$\mathbf{F_i\ (\tau_{i,1}, \tau_{i,2}, \ldots , \tau_{i,n})}$$

onde

$\mathbf{F_i}$ é o nome da função adaptativa

$\boldsymbol{\tau_{i,\,n}}$ são os argumentos passados para função

Na Linguagem RSW as chamadas são definidas da mesma forma, sendo que a chamada de função adaptativa anterior deve ser denotada logo após a descrição da configuração inicial, e antes do símbolo '**->**', enquanto a chamada de uma ação adaptativa posterior deve ser sempre denotada no final da expressão, imediatamente antes do símbolo '.' .

(<estado0> , <entrada0>) : <AcaoPre0> ->

(^ <estado0>, <submaquina0>, <entrada1>) : <AcaoPos1> .

Na linguagem RSW, a correspondência dos argumentos da chamada com os parâmetros definidos na função é posicional com verificação forte de tipos.

Em RSW, podem ser utilizados como argumentos todos os estados, todos os átomos definidos e todos os nomes de sub-máquinas que estejam sendo utilizadas pelo autômato adaptativo no momento da chamada, incluindo os que forem criados dinamicamente pelo próprio autômato.

Declaração das Funções Adaptativas

As funções adaptativas são declaradas com a finalidade de passar informações ao compilador, de modo que possam ser invocadas pelas produções, na forma de ações adaptativas. Tais funções são formadas por um conjunto de ações adaptativas elementares de consulta, remoção ou inclusão de transições.

As ações elementares manipulam os estados, átomos e sub-máquinas existentes no autômato no momento de sua ativação, bem como faz acesso aos parâmetros de entrada descritos na declaração da função, aos geradores e às variáveis, definidos no corpo da função.

Uma função adaptativa é denotada através de um cabeçalho(no qual se definem os parâmetros de entrada e seus respectivos tipos) e do corpo da função, em que são inicialmente declarados os geradores, variáveis, e chamadas de funções adaptativas seguidos de uma lista de ações elementares de consulta, remoção e inserção a serem executadas quando da sua ativação na forma de uma ação adaptativa.

A seguir faremos uma discussão analítica comparativa, da declaração de uma função adaptativa, na notação algébrica e na Linguagem RSW.

Cabeçalho da Função Adaptativa

O cabeçalho da função adaptativa se inicia com a indicação do seu nome, seguido por uma lista de parâmetros formais, separados por vírgulas e denotados entre parênteses. O número de parâmetros de uma função pode

ser nulo, devendo-se neste caso omitir os parênteses. Quando definidos, todos os parâmetros serão de entrada, ou seja, a função não gera através dos parâmetros qualquer informação de retorno ao ponto de chamada da função. Os parâmetros são definidos no momento da execução da função, e não sofrem nenhuma alteração até a conclusão da mesma. O símbolo separador entre o cabeçalho e o corpo da função adaptativa é o átomo '='. Assim sendo, a forma geral do cabeçalho de uma função adaptativa, na notação algébrica, pode ser representado por:

$$\mathbf{F}_i\ (\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n}) =$$

Em conseqüência da verificação forte de tipos, a linguagem RSW impõe a declaração do tipo correspondente a cada parâmetro da função. Assim, a expressão acima, na notação algébrica, corresponde à seguinte expressão na linguagem RSW:

<Funcao> (<tipo$_0$><parâmetro$_0$>, <tipo$_1$><parâmetro$_1$>, ...) =

A imposição do tipo do parâmetro foi adotada por facilitar significativamente no reconhecimento sintático, e principalmente por assegurar uma prática de programação segura, que traz legibilidade e clareza ao texto-fonte, e maior consistência ao código resultante.

Corpo da Função Adaptativa

No corpo da função adaptativa definem-se as ações adaptativas elementares a serem impostas sobre o autômato. Esse bloco de definições é localizado logo após o cabeçalho da função, e é delimitado entre chaves. Esta sintaxe vale tanto na notação algébrica, quanto na linguagem RSW.

O primeiro bloco, contido no corpo da função adaptativa, é a declaração de nomes, mecanismo sintático para a indicação dos identificadores das variáveis e geradores a serem utilizados. Tais identificadores seguem a regra de representação usualmente encontrada nas linguagens de programação, utilizando –se, para denotar os símbolos, cadeias de letras e de dígitos, mais o caracter subscrito "_". Na notação algébrica são ainda empregadas algumas letras gregas. Entretanto, para a simplificação do editor de seus programas-fonte, a linguagem RSW limita-se às letras do

alfabeto ASCII. Outra alteração na forma de declarar um nome da notação algébrica para a linguagem RSW, está nos geradores. Na primeira os geradores são declarados como um identificador acrescido do sufixo "*". Na linguagem RSW o gerador é declarado como um identificador acrescido do prefixo "*". Além disso, para indicar o término da declaração de nomes é utilizado o caracter ":".

Portanto se uma declaração de nomes na notação algébrica apresentar o seguinte formato (para m, n => 0):

$$\mathbf{v_1, v_2, \ldots, v_m, g_1*, g_2*, \ldots, g_n*}$$

tal expressão assumirá o seguinte aspecto, na linguagem RSW (para m, n => 0):

$$\mathbf{<}tipo_1\mathbf{>}\ v1, \ldots, \mathbf{<}tipo_m\mathbf{>}\ v_m, \mathbf{<}tipo_1\mathbf{>}\ *g_1, \ldots, \mathbf{<}tipo_n\mathbf{>}\ *g_n\ :$$

Logo em seguida ao primeiro bloco, apresenta-se o bloco de declarações das ações adaptativas elementares. Este bloco é formado por uma lista, eventualmente vazia, de ações adaptativas elementares dos tipos inclusão, eliminação e/ou consulta, precedida e seguida de uma chamada de função adaptativa.

Esta lista é formada por uma cadeia de ações elementares, com sintaxes iguais na notação algébrica e na linguagem RSW, a saber:

- Ação elementar de consulta

? [produção]

- Ação elementar de inclusão

+ [produção]

- Ação elementar de eliminação

- [produção]

Apesar das representações das ações elementares serem iguais na notação algébrica e na linguagem RSW, a produção em si apresenta algumas diferenças, como já foi descrito na seção Definição das Produções, página 21 deste capítulo.

Observações

Quanto à notação empregada na linguagem RSW, e à notação algébrica originalmente apresentada em [28], pode-se concluir que:

a) na linguagem RSW são incluídas declarações explícitas de tipos, ausentes na notação algébrica.
b) há pequenas diferenças léxicas e sintáticas nos elementos principais tais como na declaração dos gerados, no uso das letras do alfabeto ASCII, nos símbolos que representam a pilha de estados e o topo da mesma.
c) A linguagem RSW restringe alguns pontos em relação à forma geral da notação algébrica, por motivos de simplicidade, sempre que tais restrições não causem dificuldades conceituais, nem impeçam a expressão dos conceitos necessários nos casos normais de utilização dos autômatos adaptativos. Por exemplo, a manipulação de forma geral do topo da pilha de estados, para operações de remoção ou adição de estados do topo da pilha que não se constituam, respectivamente, em retorno e chamada de sub-máquina.

Capítulo IV - Aspectos de Implementação

Neste capítulo são apresentados os principais aspectos de implementação do projeto RSW, incluindo algumas informações sobre o ambiente operacional tanto de *hardware*, quanto de *software*, e sobre a estrutura da ferramenta desenvolvida. Em particular, são descritas classes de objetos utilizados pelo compilador RSW, e é apresentada a técnica através da qual a ferramenta evoluiu até disponibilizar o recurso da adaptabilidade.

Plataforma de Hardware

A configuração de hardware utilizada na implementação do RSW foi de um micro computador com um processador Intel Pentium II de 400 MHz, com 128 megabytes de RAM, 6,2 gigabytes de disco rígido conectado a uma controladora IDE, 512 kilobytes de memória cache, monitor colorido, mouse e teclado. O equipamento descrito foi utilizado sem conexão à rede.

Plataforma de Software

O ambiente de desenvolvimento inicialmente foi o sistema operacional Microsoft DOS versão 6.22, Microsoft Windows versão 3.11 e Borland Delphi versão 1.0.

No início do ano de 1996, a Borland lançou o Delphi versão 2.0 para ambientes de 32 bits. Por esta razão não se podia instalar o produto num ambiente tipicamente de 16 bits como o Windows 3.11. Além de trabalhar nos sistemas operacionais Windows 95 ou NT, a ferramenta apresentava outras características positivas para que a mesma fosse adotada. Uma delas é a possibilidade de múltiplas *threads*, ou da concorrência que pode ser implementada nos reconhecedores. Portanto decidiu-se migrar o ambiente inicial para o descrito abaixo.

O ambiente de desenvolvimento final foi o sistema operacional Windows NT 4.0 com a ferramenta Delphi versão 3.0 da Inprise.

Ferramenta de Desenvolvimento

Esta seção contém uma breve descrição da ferramenta utilizada no desenvolvimento do RSW: Delphi 3.0. Suas principais características são:

- Ambiente integrado com interface gráfica para edição, compilação, execução e depuração de programas.
- Utilização de um dialeto da linguagem Object Pascal, que permite a programação orientada a objetos.
- Geração de código executável, e não interpretado.
- Disponibilização de uma ampla biblioteca de componentes reutilizáveis, tais como, controles gráficos, estruturas de dados e manipulação de arquivos.

Arquitetura do Sistema

Esta seção se dedica a detalhar a arquitetura da ferramenta RSW. O primeiro item a ser discutido é o esquema de funcionamento de um gerador de reconhecedor sintático. Ou seja, como obter um reconhecedor sintático, na forma de código executável, a partir da descrição de suas transições na linguagem RSW.

Em seguida será descrita a biblioteca de classes que formam o coração do reconhecedor sintático. Esta biblioteca é responsável por tarefas como execução das transições com ou sem consumo de átomos, empilhamento e desempilhamento de estados na pilha, execução das ações adaptativas e funções adaptativas, e chamada das rotinas semânticas.

Esquema de Funcionamento

O sistema desenvolvido é denominado Meta Reconhecedor Sintático Para Ambiente Windows(RSW). Este sistema é um compilador, que reconhece um texto, onde estão representadas produções que descrevem a sintaxe da linguagem a ser reconhecida por um outro compilador. Uma vez sendo

corretamente reconhecido o texto de entrada, é gerada, em linguagem Pascal, uma versão destas produções. Estas produções utilizam a biblioteca de objetos do RSW durante o reconhecimento de uma cadeia de entrada qualquer.

Esta descrição pode ser trabalhada ou simplesmente compilada com a ferramenta Delphi para se obter o código executável para a máquina alvo. Este código executável pode ser registrado, e assim incorporado ao ambiente de trabalho da ferramenta RSW, dando a possibilidade ao usuário, que textos expressos na linguagem do reconhecedor gerado sejam reconhecidos através da própria ferramenta RSW.

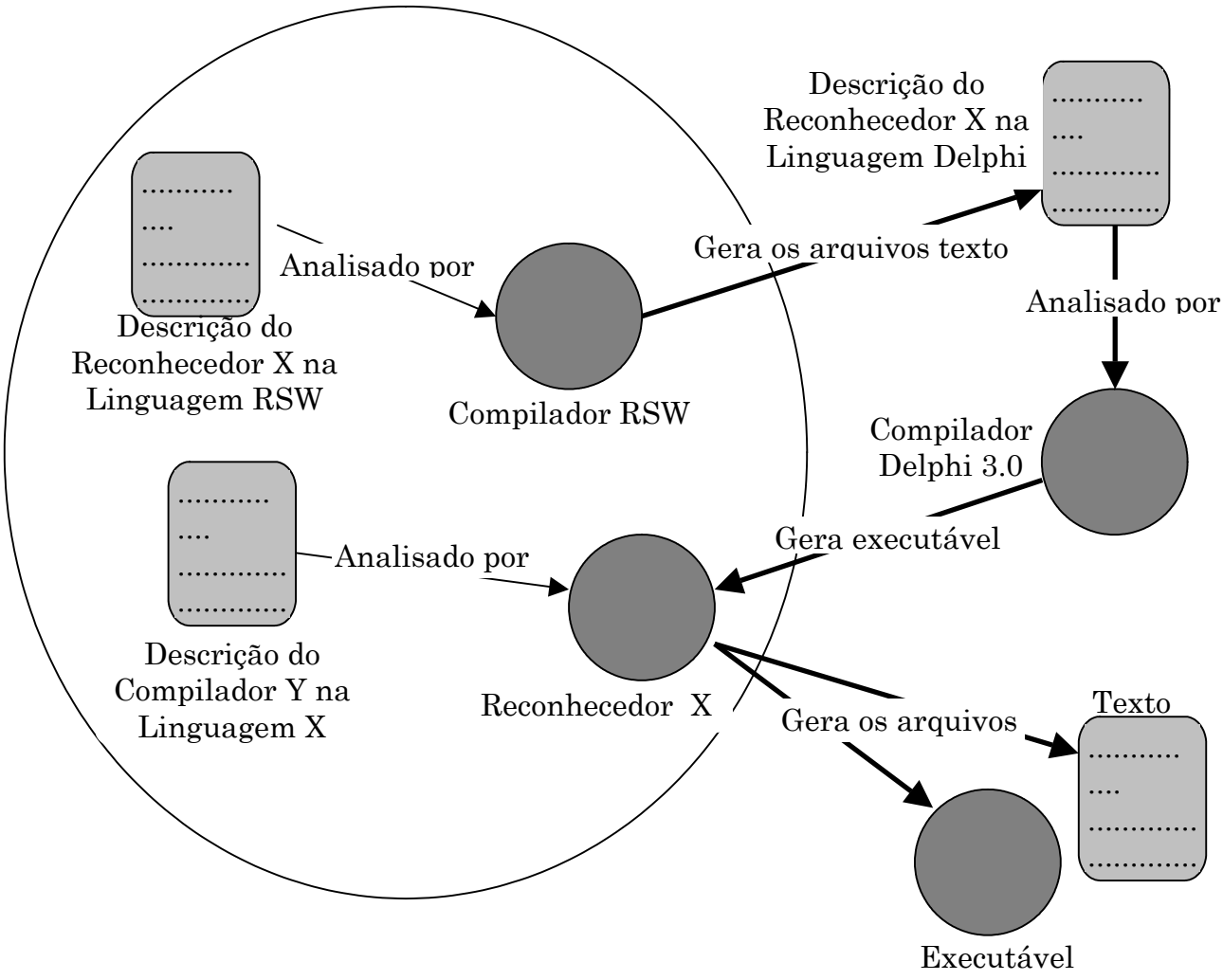

Figura 8 - Ambiente de trabalho da ferramenta RSW

Biblioteca de classes RSW

O RSW possui uma coleção de classes e objetos básicos para a implementação de um compilador. Estas classes foram utilizadas no próprio desenvolvimento do compilador que reconhece a linguagem RSW, ou seja, o compilador RSW.

Nas classes estão contidas as informações na forma de propriedades ou atributos, e o comportamento na forma de métodos básicos, necessários à implementação de um compilador. Assim, para utilizá-las, deve-se criar classes delas derivadas, contendo informações tais como, configuração inicial, definição de propriedades específicas, o nome do objeto, entre outras.

As classes que compõem um compilador, segundo este modelo, são: aCompilador, aSubMaquina, aEstado, aFuncao, aAcaoAdaptativa, aTabelaMaquinas, aTabelaEstados e aTabelaTransicao.

Podemos dividi-las em dois grupos de classes, que apresentam características comuns entre si: o primeiro grupo é constituído pelas classes aCompilador, aSubMaquina, aFuncao e aEstado, o segundo grupo pelas classes aTabelaMaquinas, aTabelaEstados e aTabelaTransicao.

O primeiro grupo, que inclui as classes aCompilador, aSubMaquina, aFuncao e aEstado, apresenta as seguintes características comuns:

- As classes instanciadas são identificadas por um nome unívoco em relação aos objetos de mesmo tipo;
- Uma das propriedade das classes é a instância de um objeto do segundo grupo;
- Apresentam métodos responsáveis pela descrição lógica utilizada na geração do código-fonte, em Pascal, do reconhecedor gerado;
- Apresentam um método que define a configuração inicial, o qual pode ser executado a qualquer momento.

A classe aAcaoAdaptativa não se classifica neste primeiro grupo. A classe deve ser instanciada diretamente, sem a necessidade de criar uma classe descendente.

O segundo grupo, que inclui aTabelaMaquinas, aTabelaEstados e aTabelaTransicao, apresenta as seguintes características comuns:

- implementam uma estrutura de dados do tipo vetor, armazenando um elemento para cada valor de índice. As classes aTabelaMaquinas e aTabelaEstados ainda podem recuperar um elemento armazenado anteriormente, através de seu nome;
- apresentam métodos de localização ou busca, recuperação, adição, e remoção dos elementos armazenados no vetor;
- são instanciados por uma classe do primeiro grupo.

aCompilador

A classe aCompilador representa, de forma geral, um compilador, pois é responsável por iniciar toda a configuração necessária para executar o reconhecimento de um texto denotado em uma linguagem qualquer.
Uma cadeia de caracteres pode ser associada a esta classe, com o objetivo de identificá-la univocamente no sistema. Além do nome, esta classe apresenta uma tabela de sub-máquinas, armazenadas em uma instância de aTabelaMaquinas. Uma destas sub-máquinas possui uma função especial e é denominada sub-máquina inicial. Seu propósito é descrito na seção de Funcionamento Geral deste objeto.

Funcionamento Geral

(a) Na Criação do Objeto

Para se criar um compilador qualquer, é necessário descrever uma classe descendente de aCompilador. Apenas um procedimento deve ser definido, o CreateName. Este procedimento é um construtor do objeto, já definido na classe básica como virtual para que possa ser sobreposto pelas classes descendentes.
Este procedimento deve instanciar um objeto de cada sub-máquina que o compilador contém, e adicioná-los à tabela de sub-máquinas representada

pela variável TabelaMaquinas, definida em aCompilador como protegida, ou seja, acessível apenas às classes derivadas de aCompilador.

Para cada instância de sub-máquina criada deve ser acionado o método que define a configuração inicial da mesma. Durante a configuração inicial da sub-máquina, muitas vezes é necessária a localização de uma outra sub-máquina existente no compilador com o intuito de se definir uma transição de chamada de sub-máquina. Por esta razão deve-se primeiro instanciar todas as sub-máquinas existentes no compilador, e em seguida acionar o método apropriado para a configuração inicial.

Uma destas sub-máquinas instanciadas deve ser também configurada como a sub-máquina inicial. A sub-máquina inicial é a primeira sub-máquina acionada quando do início do reconhecimento de uma cadeia qualquer.

(b) No Reconhecimento

Após ser executada a criação do objeto, descendente de aCompilador, teremos em memória, pronto para o reconhecimento, o compilador desejado. Para iniciar o reconhecimento de um texto, informamos o modo de armazenamento do mesmo e acionamos o método de início do reconhecimento. O modo de armazenamento pode ser ou direto de um arquivo em disco, ou através de um controle da interface gráfica do Windows que esteja editando o arquivo.

O reconhecimento completo de uma cadeia se dá através dos seguintes passos:

- O analisador léxico é acionado uma vez para disponibilizar o primeiro átomo a ser consumido.
- Uma sub-máquina, configurada como inicial, é acionada com o primeiro átomo disponibilizado pelo analisador léxico. Esta chamada à sub-máquina inicial desencadeia todo o processo de reconhecimento da cadeia de entrada.
- Após o retorno da sub-máquina inicial, o átomo associado a esta sub-máquina deve ser o átomo correntemente disponível para análise na

cadeia de entrada. O objeto instanciado a partir de aCompilador consome este átomo, e verifica se o próximo átomo disponibilizado é o de final de cadeia, indicando assim que não existe mais nada a ser consumido. Caso o último átomo disponível não seja o de final de cadeia, então será emitido um erro sintático para o átomo restante.

Exemplo

As listagens abaixo apresentam trechos de código-fonte em Pascal, com a definição e implementação da classe aRSWv200, descendente de aCompilador. Estes trechos foram extraídos do módulo RSWv200.pas gerado pela ferramenta RSW.

```
1- ...
2- aRSWv200= class ( aCompilador )
3-
4- public
5-    CONSTRUCTOR CreateName(
6-          AOwner:TComponent;
7-          ACallBack: aCallBackProcs;
8-          nm: string );
9-    end;
```

Listagem 1 - Definição da classe aRSWv200

A classe aRSWv200 representa o compilador RSW na versão 2.0. A linha 2 apresenta a definição da classe como sendo descendente da classe aCompilador.

Em seguida, nas linhas 5 à 8, é declarado o único método necessário, que é o construtor da classe, denominado CreateName. O construtor recebe três parâmetros: AOwner, ACallBack e nm. AOwner e ACallBack são parâmetros relacionados, respectivamente, com a arquitetura da ferramenta de desenvolvimento Delphi e com o ambiente integrado de desenvolvimento RSW.

```
1- CONSTRUCTOR aRSWv200.CreateName(
2-             AOwner:TComponent;
3-             ACallBack:aCallBackProcs;
4-             nm:string );
5- var
6-    Maquina : aSubMaquina;
7- begin
8-    inherited CreateName(AOwner, ACallBack, nm);
9-
10-     MaquinaInicial :=
11-       aGram200.CreateName(Self, CallBackProcs,
   'Gram200' );
12-     AdicionarMaquina(Maquina);
13-     DefinirMaquinaInicial(Maquina);
14-     AdicionarMaquina(
15-       aFunc200.CreateName(Self, CallBackProcs,
   'Func200' ));
16-     AdicionarMaquina(
17-       aProd200.CreateName(Self, CallBackProcs,
   'Prod200' ));
18-     AdicionarMaquina(
19-       aScan200.CreateName(Self, CallBackProcs,
   'Scan200' ));
20-
21-     if LocalizarMaquinaString( 'Func200' , Maquina)
   then begin
22-        Maquina.DefinirConfiguracaoInicial;
23-     end else begin
24-        CallBackProcs.AdicionarErro(0,0,0,
25-                 erInicializacao +  'Func200' )
26-     end;
27-     if LocalizarMaquinaString( 'Gram200' , Maquina)
   then begin
28-        Maquina.DefinirConfiguracaoInicial;
29-     end else begin
30-        CallBackProcs.AdicionarErro(0,0,0,
31-                 erInicializacao +  'Gram200' )
32-     end;
33-     if LocalizarMaquinaString( 'Prod200' , Maquina)
   then begin
34-        Maquina.DefinirConfiguracaoInicial;
35-     end else begin
36-        CallBackProcs.AdicionarErro(0,0,0,
37-                 erInicializacao +  'Prod200' )
38-     end;
39-     if LocalizarMaquinaString( 'Scan200' , Maquina)
   then
40-     begin
41-        Maquina.DefinirConfiguracaoInicial;
42-     end else begin
```

```
43-         CallBackProcs.AdicionarErro(0,0,0,
44-                   erInicializacao +  'Scan200' )
45-      end;
46-  end;
```

Listagem 2 - Implementação da classe aRSWv200

O terceiro parâmetro nm contém o nome do compilador, que o identificará perante a existência de outros objetos instanciados a partir da classe aCompilador.

A implementação da classe aRSWv200 apresenta apenas um único método, o construtor CreateName. A primeira ação tomada pelo construtor é invocar o construtor CreateName da classe aCompilador, passando os mesmos parâmetros recebidos. Esta chamada encontra-se na linha 19.

Após ter sido instanciada a classe descendente de aCompilador, deve-se instanciar as sub-máquinas que compõe o compilador. O compilador RSWv200 é composto de quatro sub-máquinas: Gram200, Prod200, Func200 e Scan200. Cada instância criada deve ser adicionada à tabela de sub-máquinas. As linhas 21 à 30 apresentam as operações descritas. Observem que a primeira sub-máquina criada tem tratamento especial. Além de ser criada e adicionada na tabela de sub-máquinas, ela também é definida como a sub-máquina inicial do compilador. Desta maneira, a instância da classe aGram200, identificada por "Gram200", será a primeira a ser invocada no momento do reconhecimento de uma cadeia de entrada.

O código apresentado nas linhas 32 à 62 possibilitam que as sub-máquinas criadas anteriormente, tenham a oportunidade de definir suas configurações para o início de um reconhecimento. A descrição da configuração inicial de uma sub-máquina é apresentada na seção que trata da classe aSubMaquina.

aSubMaquina

A classe aSubMaquina implementa o comportamento básico que uma sub-máquina deve ter no sistema. Associada a ela pode haver uma cadeia de caracteres, que deve ser única, por ser utilizada quando da execução de uma operação de busca através do objeto de tipo aTabelaMaquinas.

Da mesma forma que a classe aCompilador necessita de uma sub-máquina inicial, a classe aSubMaquina necessita de uma referência à instância da classe aEstado, que representa o estado inicial da sub-máquina.
Os estados que a instância da classe aSubMaquina possui são armazenados na variável TabelaEstados, declarada como protegida e pertencente a classe aTabelaEstados.

Funcionamento Geral

(a) Na Criação do Objeto

Uma das características comuns às classes do grupo ao qual aSubMaquina pertence, é que elas devem ser utilizadas através da declaração de classes descendentes, com a sobreposição dos métodos apropriados.
A classe aSubMaquina deve servir de base para uma classe descendente, e esta, por sua vez, deve sobrepor apenas o método de definição da configuração inicial da sub-máquina.
Por ocasião da criação da sub-máquina, são determinadas duas propriedades do objeto: a primeira é o nome que a sub-máquina assumirá para identificá-la no sistema, a segunda é uma referência de qual objeto, de classe aCompilador, a possui. A primeira propriedade é fundamental quando houver uma busca através do objeto de tipo aTabelaMaquinas que se utiliza exatamente deste nome para distinguir as sub-máquinas por ele armazenadas. Já a segunda é utilizada também no processo de localização, quando um estado tem que definir uma transição com a chamada de sub-máquina, pois neste momento é necessária a referência do objeto que representa aquela sub-máquina.
Após a criação da sub-máquina por um objeto da classe aCompilador, o próximo passo realizado é a execução da configuração inicial da sub-máquina. Este método é definido na classe descendente, e é responsável pela criação dos estados existentes na sub-máquina, bem como pela configuração inicial destes estados.
A primeira tarefa do método que define a configuração inicial é assegurar que todos os estados existentes na sub-máquina, estejam instanciados e

disponíveis através da variável TabelaEstados. Ele deve inicialmente procurar o estado, e se este não existir, então instanciá-lo e adicioná-lo à tabela. Note o detalhe de primeiro procurar o estado e depois instanciá-lo se houver necessidade. Isto se deve ao fato de que este método pode ser acionado a qualquer instante da execução de um reconhecimento. Portanto estados podem ter sido criados, e estados podem ter sido removidos. Não importando o momento no qual foi acionado o método, a primeira tarefa a ser executada é a da disponibilização de todos os estados que fazem parte da configuração inicial.

A segunda tarefa é definir o estado inicial. Assim como a sub-máquina inicial servia para ser utilizada como a primeira sub-máquina acionada, o estado inicial servirá como estado corrente para a busca de uma transição quando do acionamento desta sub-máquina.

A terceira e última tarefa é varrer todos os estados instanciados que fazem parte da configuração inicial da sub-máquina e acionar o método de definição da configuração inicial destes estados. Finalizada esta tarefa a sub-máquina estará pronta para o reconhecimento de uma cadeia de entrada qualquer.

(b) No Reconhecimento

O objeto da classe aCompilador inicia o reconhecimento chamando a sub-máquina configurada como sub-máquina inicial. Esta, por sua vez, toma os seguintes passos na tentativa de reconhecer a cadeia de entrada:

- Verifica se existe um estado configurado como estado inicial. Caso haja um estado inicial, invoca um método deste estado que executa uma transição. Este método pode ou não ter consumido o átomo disponível na cadeia de entrada, e a transição é finalizada com o retorno do próximo estado.

- Se o método invocado retornou um estado válido, executa-se um laço que torna o estado retornado, o estado corrente a ser analisado. Depois aciona-se o método de transição deste novo estado para se

obter o próximo estado. Este laço se encerra quando não existir um novo estado a ser analisado.

- Não existindo um novo estado válido, é analisado o estado corrente no sentido de se verificar se o mesmo é um estado configurado como final. Caso seja um estado final, a sub-máquina simplesmente encerra sua execução, retornando a sub-máquina que a invocou. Caso contrário, é emitida uma mensagem de um erro sintático informando a posição exata no texto de entrada, do último átomo analisado que não permitiu que o autômato prosseguisse no reconhecimento da cadeia de entrada.

Exemplo

As listagens abaixo apresentam trechos de código-fonte em Pascal, com a definição e implementação da classe aGram200, descendente de aSubMaquina. Estes trechos foram extraídos do módulo Gram200.pas gerado pela ferramenta RSW.

```
1- aGram200= class ( aSubMaquina )
2-
3-    public
4-       PROCEDURE DefinirConfiguracaoInicial;
5-       override;
6-    end;
7-
```

Listagem 3 - Definição da classe aGram200

A linha 1 apresenta a definição da classe aGram200, descendente de aSubMaquina. Na linha 4 o método DefinirConfiguracaoInicial é definido. Este é o único método necessário de ser definido quando se utiliza a classe aSubMaquina como ancestral, pois todo o comportamento de uma sub-máquina está encapsulado na classe aSubMaquina. A linha 5 apresenta a

palavra reservada em Pascal que indica que este método deve ser sobreposto ao definido na classe ancestral aSubMaquina.

```
1- PROCEDURE aGram200.DefinirConfiguracaoInicial;
2-
3- var
4-    Estado : aEstado;
5- begin
6-    GerarImagemAtomo := FALSE;
7-    FNomeMaquinaEntrada :=  'Scan200' ;
8-    if NOT (Self.Owner as
  aCompilador).LocalizarMaquinaString(
9-         FNomeMaquinaEntrada,
10-            FMaquinaEntrada) then
11-     begin
12-        raise EAbort.CreateFmt(
13-            erMaquinaEntradaInexistente,
14-            [FNomeMaquinaEntrada,
15-            Self.Nome]);
16-     end;
17-
18-     if NOT LocalizarEstadoString( 'e1' , Estado) then
19-     begin
20-        Estado:= ae1.CreateName(
21-                 Self,
22-                 CallBackProcs,
23-                 'e1' );
24-        AdicionarEstado(Estado);
25-     end;
26-     DefinirEstadoInicial(Estado);
27-  ...
28-
29-  ...
30-     if NOT LocalizarEstadoString( 'e12' , Estado) then
31-     begin
32-        AdicionarEstado(ae12.CreateName(
33-                 Self,
34-                 CallBackProcs,
35-                 'e12' ));
36-     end;
37-
38-     if LocalizarEstadoString( 'e1' , Estado) then
39-     begin
40-        Estado.DefinirConfiguracaoInicial;
41-     end else begin
42-        CallBackProcs.AdicionarErro(0,0,0,
  erInicializacao
43-                   + 'e1' )
```

```
44-     end;
45-  ...
46-
47-  ...
48-     if LocalizarEstadoString( 'e12' , Estado) then
49-     begin
50-        Estado.DefinirConfiguracaoInicial;
51-     end else begin
52-        CallBackProcs.AdicionarErro(0,0,0,
  erInicializacao
53-                + 'e12')
54-     end;
55-
56-  end;
```

Listagem 4 - Implementação da classe aGram200

Após ter sido instanciado um objeto da classe aGram200, será invocado o método DefinirConfiguracaoInicial. O comportamento completo deste método foi descrito no item (a) Na Criação do Objeto, página 49. A listagem acima apresenta a implementação parcial do comportamento descrito.

Nas linhas 14 à 23 é apresentada a implementação para se definir a sub-máquina de entrada para a classe aGram200. A variável FMaquinaEntrada, da classe aSubMaquina, conterá a sub-máquina de entrada obtida através da consulta a tabela de sub-máquinas existentes no objeto compilador. Caso não seja encontrada nenhuma sub-máquina que corresponda ao nome da sub-máquina de entrada definida para aGram200, o reconhecimento será interrompido e emitida uma mensagem de erro. A interrupção do reconhecimento se deve ao fato da sub-máquina não ter condições de realizar nenhuma transição, já que não existe sub-máquina capaz de gerar átomos de entrada.

A segunda atividade implementada neste exemplo, é a instanciação de todos os estados que compõe a sub-máquina aGram200. Como medida preventiva, é consultada a tabela de estado para verificar se já existe um estado com o mesmo identificador do estado que será criado. Caso o estado não exista, é criado um novo objeto com o identificador consultado e adicionado a tabela de estados da sub-máquina. Caso o estado já exista, nada é feito, pois mais

adiante este estado terá sua configuração inicial definida novamente. Esta medida preventiva é necessária caso o método DefinirConfiguracaoInicial seja invocado mais de uma única vez durante o reconhecimento de uma cadeia de entrada. Esta chamada pode ser realizada a partir de uma rotina semântica qualquer.

Observe o tratamento especial para o estado que deve ser configurado como inicial, ou seja, aquele estado que será o primeiro a ser invocado quando a sub-máquina aGram200 for chamada. A linha 33 apresenta a chamada que configura um estado qualquer como inicial.

A atividade seguinte é obter todos os estados que compõem a sub-máquina, e chamar a rotina DefinirConfiguracaoInicial para cada um deles.

aEstado

A classe aEstado representa um estado que possui um conjunto de transições possíveis. Uma de suas propriedades é o nome, uma cadeia de caracteres que a identifica no sistema, e é utilizada nas operações de procura no objeto da classe aTabelaEstados, responsável por armazenar todos os estados criados.

Outra propriedade muito importante é a que indica se o estado é ou não final, ou seja, se durante o reconhecimento de uma cadeia qualquer a sub-máquina que contém este estado ficar impossibilitada de executar novas transições, e este for o estado corrente, então a mesma pode considerar bem sucedido o reconhecimento.

Associada à propriedade do estado final, temos a propriedade do átomo de retorno. Este átomo é adicionado à cadeia de entrada quando o estado for final e não houver mais transições válidas a serem feitas. Este átomo simboliza um reconhecimento bem sucedido por parte da sub-máquina que contém este estado. Ao retornar o controle à sub-máquina chamadora, esta analisará a cadeia de entrada, na busca de algum átomo de retorno da sub-máquina chamada.

Todas as transições válidas para uma instância qualquer de aEstado ficam armazenadas em uma instância de aTabelaTransicao. Nela armazenam-se também, informações tais como os átomos a serem empilhados como conseqüência da execução de uma transição, as rotinas semânticas e as ações adaptativas anterior e posterior, invocadas durante o reconhecimento, e uma sub-máquina a ser chamada.

Funcionamento Geral

(a) Na Criação do Objeto

A classe aEstado segue a mesma regra para sua utilização que a do seu grupo: deve-se criar uma nova classe descendente, com alguns métodos sobrepostos. Neste caso necessitamos sobrepor somente um método, aquele que define a configuração inicial das transições válidas para o estado.

Já vimos que o objeto da classe aCompilador, quando da sua criação, instancia todas as sub-máquinas existentes em sua configuração. Estas sub-máquinas, por sua vez, instanciam todos os estados que as compõem. Após assegurar que todos os estados que fazem parte de sua configuração inicial estejam instanciados, a sub-máquina aciona o método que define a configuração inicial de cada um destes estados. É exatamente este método que a classe descendente deve implementar, pois a configuração inicial de um estado é completamente dependente de cada estado em particular.

O método básico, implementado na classe aEstado, define o estado com a seguinte configuração:

- como um estado não final;
- sem nenhuma transição válida em sua tabela de transições.

Portanto o método de configuração inicial do estado implementado na classe descendente de aEstado, deve invocar o homônimo da classe aEstado antes de começar a definir o conjunto de transições válidas.

A definição de uma transição é feita através de um método do objeto aEstado. Este método deve necessariamente receber o átomo de entrada, para o qual a transição será configurada, e o estado de destino, que deve se

tornar o estado corrente após efetuada a transição. Além destes parâmetros pode-se indicar ainda uma sub-máquina a ser chamada quando da análise das transições para o átomo de entrada. Pode-se indicar também uma ação semântica a ser invocada, um átomo que deverá ser empilhado na cadeia de entrada, e as ações adaptativas a serem executadas antes ou depois de se realizar a transição.

Para definir uma transição que chama uma sub-máquina, deve haver uma maneira de ganhar acesso àquela sub-máquina, dentro do método de configuração inicial do objeto aEstado. Para isso, quando um objeto aEstado é instanciado, um dos parâmetros recebidos é o nome que o identificará no sistema, o outro é o acesso à sub-máquina a que o mesmo pertence. Através da sub-máquina que possui o estado criado, pode-se acessar qualquer uma das sub-máquinas do sistema, pois a sub-máquina contém o acesso ao compilador, e este, por sua vez, contém todas as sub-máquinas do sistema. Assim, quando uma transição necessita fazer referência a qualquer sub-máquina, esta referência é requisitada à instância da classe aCompilador, através da sub-máquina à qual pertence o estado que definirá a transição.

Definidas todas as transições válidas, pode-se ainda atribuir valores diferentes a outras propriedades do estado. A propriedade que indica se o estado é final ou não pode ser alterada. Caso esta propriedade seja alterada para indicar que o estado é um estado final, deve-se também atribuir um valor para a propriedade do átomo de retorno. Este átomo será empilhado na cadeia de entrada e disponibilizado como átomo corrente caso o estado em questão não apresente nenhuma transição válida, havendo, portanto, a necessidade de retornar à sub-máquina chamadora. Este átomo representará, para a sub-máquina chamadora, o reconhecimento bem sucedido por parte da sub-máquina em questão.

Portanto toda sub-máquina deverá conter pelo menos um estado definido como final, e um átomo a ser empilhado na cadeia de entrada, como átomo de retorno. Pode haver mais de um estado final, e cada um destes estados pode ter definido como átomo de retorno, um átomo específico diferente.

Definidas as transições e as propriedades do estado criado, o mesmo estará pronto para o início do reconhecimento de uma cadeia qualquer.

(b) No Reconhecimento

O ponto de partida do processo de reconhecimento de uma cadeia de entrada se dá em uma instância da classe aCompilador. Nele deve haver uma sub-máquina especial, denominada sub-máquina inicial. Esta sub-máquina será a primeira a ser chamada para o consumo do primeiro átomo da cadeia de entrada. Vimos também que uma das propriedades da sub-máquina é o seu estado inicial. Portanto a sub-máquina inicial deverá possuir também um estado inicial. É a partir desse estado inicial que a sub-máquina inicial é ativada para analisar o primeiro átomo de entrada disponível. Desta análise obtém-se um estado destino, com ou sem o consumo do átomo de entrada disponível, ou então a especificação da chamada a uma outra sub-máquina do compilador, e ainda a possibilidade de empilhar um átomo na cadeia de entrada e executar as ações adaptativas e uma rotina semântica.
Embora esta análise, baseada no estado corrente, apresente um dos resultados acima, existe uma ordem específica no teste de cada um destes pontos que deve ser seguida, de acordo com a teoria em [28]. A ordem é a seguinte:

- Verificar inicialmente a possibilidade de chamar uma outra sub-máquina. Se existir tal possibilidade, acionar a sub-máquina.
- Obter a transição em função do átomo de entrada disponível, definindo assim a transição corrente.
- Verificar se existe uma chamada de ação adaptativa anterior. Se existir, executar tal ação adaptativa anterior. Após a execução da ação adaptativa é refeita a consulta da transição corrente.
- Se existir um estado de destino na transição corrente:
    - Verificar se existe um átomo a ser empilhado na cadeia de entrada. Se existir, empilhar o átomo na cadeia de entrada.

    - Verificar se existe uma ação adaptativa posterior. Se existir, executar a ação adaptativa posterior.
    - Verificar se existe uma rotina semântica a ser executada. Se existir, executar a rotina semântica.
    - Consumir o átomo de entrada disponível, obtendo um novo átomo para análise.
- Se não existir um estado de destino válido, verificar se existe um estado destino para o átomo que representa uma transição em vazio. Se existir, executar os passos descritos quando existir um estado de destino válido na transição corrente.
- Se ainda não existir um estado de destino:
    - Verificar se o estado está configurado como estado final. Se estiver configurado como estado final, empilhar o átomo de retorno.
- Se não houver um estado de destino válido e o estado não estiver configurado como final, nenhuma atitude deve ser tomada pelo objeto aEstado. Entretanto esta condição deve causar a emissão de um erro sintático pela sub-máquina corrente.

Exemplo

As listagens abaixo apresentam trechos de código-fonte em Pascal, com a definição e implementação da classe ae4, descendente de aEstado. Estes trechos foram extraídos do módulo Gram200.pas gerado pela ferramenta RSW.

```
1-    ae4= class ( aEstado )
2-
3-    public
4-       PROCEDURE DefinirConfiguracaoInicial;
5-       override;
6-    end;
```

Listagem 5 - Definição da classe ae4

A linha 1 apresenta a definição da classe ae4, descendente de aEstado. Na linha 4 o método DefinirConfiguracaoInicial é definido sem parâmetros de entrada ou saída. Este é o único método que exige ser definido quando se utiliza a classe aEstado como ancestral, pois todo o comportamento de um estado está encapsulado na classe aEstado. A linha 5 apresenta o comando Pascal que indica que este método deve substituir o definido na classe ancestral.

```
1- PROCEDURE ae4.DefinirConfiguracaoInicial;
2- begin
3-    inherited DefinirConfiguracaoInicial;
4-
5-    DefinirTransicao(
6-         ORD(functionsTK),
7-         (Self.Owner as aSubMaquina).ObterEstadoNome(
   'e12' ),
8-         ORD(emptyTK),
9-         nil, nil, nil, nil);
10-
11-     DefinirTransicao(
12-           ORD(lparenTK),
13-           (Self.Owner as aSubMaquina).ObterEstadoNome(
   'e4' ),
14-           ORD(emptyTK),
15-           ((Self.Owner as aSubMaquina).Compilador as
16-                aCompilador).ObterMaquinaNome('Prod200'
   ),
17-           nil, nil, nil);
18-
19-     DefinirTransicao(
20-           ORD(SBProductionTK),
21-           (Self.Owner as aSubMaquina).ObterEstadoNome(
   'e8' ),
22-           ORD(emptyTK),
23-           nil,
24-           DefinirTransicaoAnalisada, nil, nil);
25-
26-     DefinirTransicao(
27-           ORD(eofTK),
28-           (Self.Owner as aSubMaquina).ObterEstadoNome(
   'e9' ),
```

```
29-             ORD(SBGrammarTK),
30-             nil,
31-             VerificarErrosSemanticos, nil, nil);
32-  end;
```

Listagem 6 - Implementação da classe ea4

A implementação do método DefinirConfiguracaoInicial da classe ae4 é responsável por definir todas as transições possíveis da sub-máquina Gram200 a partir do estado e4. Para adicionar uma transição na tabela de transições de um estado, utiliza-se o método DefinirTransicao definido na classe aEstado. Este método recebe vários parâmetros, que descrevem a transição a ser inserida na tabela. Os parâmetros são: átomo de entrada, próximo estado, átomo de retorno ou átomo a ser empilhado na cadeia de entrada, sub-máquina a ser invocada, rotina semântica, e ações adaptativas pré e pós.

aFuncao

A classe aFuncao implementa o comportamento de uma função adaptativa invocada pela chamada adaptativa anterior ou posterior de uma transição qualquer. A função adaptativa pode receber parâmetros do tipo estado, átomo ou sub-máquina, sendo que os correspondentes argumentos devem estar definidos no momento da chamada da função. Além dos parâmetros, pode-se definir variáveis e geradores no corpo da função adaptativa. Após tais definições encontramos um conjunto de ações elementares que irão modificar as configurações das sub-máquinas existentes no sistema.

Funcionamento Geral

(a) Na Criação do Objeto

Seguindo o mesmo procedimento adotado na utilização das classes aCompilador, aSubMaquina e aEstado, a classe aFuncao deve servir de base para a implementação de uma função adaptativa. É necessário definir uma classe derivada de aFuncao com a sobreposição de alguns métodos. Estes

irão especializar o comportamento genérico de aFuncao para um comportamento específico de uma dada função adaptativa.

Após uma sub-máquina ter sido instanciada como parte do procedimento de criação de um compilador, é iniciada a criação de todos os estados que compõem aquela sub-máquina. Depois dos estados, serão criadas todas as funções adaptativas declaradas naquela sub-máquina. Na criação de uma função adaptativa são realizadas atividades básicas implementadas nos construtores da classe aFuncao. Estas atividades são a iniciação das variáveis da classe, que armazenam o número de parâmetros da função adaptativa, o número de variáveis e geradores definidos no corpo da função adaptativa, a definição do nome da função que a identifica univocamente, e a criação da tabela de ações elementares da função adaptativa.

Depois que todos os estados e funções adaptativas estiverem criados, a sub-máquina começa a invocar o método de definição da configuração inicial de cada estado, para que este possa definir o conjunto das possíveis transições. Finalizada a configuração inicial dos estados, a sub-máquina passa então a invocar o método de definição da configuração inicial de cada uma das funções adaptativas existentes. Este é o momento em que a função irá montar sua tabela de ações elementares, determinar o número de parâmetros que devem ser passados, o número de variáveis e o número de geradores definidos no corpo da função adaptativa.

O método de definição da configuração inicial de uma função, denominado DefinirConfiguracaoInicial, deve inicialmente atribuir os valores corretos do número de parâmetros, variáveis e geradores da função às propriedades apropriadas, respectivamente NumeroParametros, NumeroVariaveis e NumeroGeradores.

É importante observar que só depois de atribuir os valores descritos acima é que deve ser invocado o método básico de definição da configuração inicial da classe aFuncao. Este método irá alocar o espaço necessário para abrigar todos os parâmetros, variáveis e geradores da função adaptativa.

Em seguida, começa a definição das ações elementares através do método DefinirAcaoElementar, implementado na classe aFuncao. Este método recebe os seguintes parâmetros:

- Tipo da ação elementar, podendo ser de inclusão, remoção ou consulta
- Todos os elementos de uma transição
- Os índices da tabela de variáveis, nos casos em que a transição utilize elementos variáveis na definição dos estados de origem e destino, ou nos átomos de entrada e retorno

(b) No Reconhecimento

A função adaptativa é ativada através de uma transição qualquer que contenha uma ação adaptativa anterior ou posterior. Na declaração da ação adaptativa encontra-se o nome da função adaptativa e os argumentos que serão passados como parâmetros.

No instante apropriado, durante a análise de transição que um objeto da classe aEstado realiza, obtém-se a referência para função adaptativa a ser ativada, já localizada anteriormente através de seu nome. Em seguida são transferidos os argumentos esperados pela função adaptativa, e finalmente transferido o controle para a execução propriamente dita da função adaptativa.

Antes de iniciar a execução das ações elementares, são iniciadas com vazio as variáveis definidas no corpo da função. Estas variáveis assumirão algum valor através de uma ação elementar de consulta realizada durante a execução da função adaptativa. Em seguida, são criados os novos estados correspondentes aos geradores, também definidos no corpo da função.

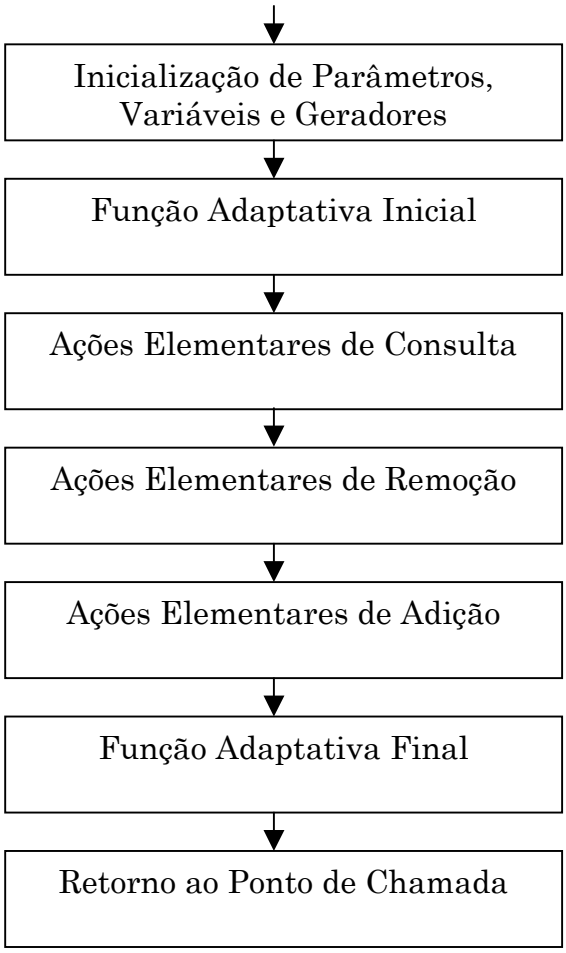

Figura 9 - Seqüência de Execução da Função Adaptativa

Realizada a iniciação dos parâmetros, variáveis e geradores, começa a execução da função adaptativa com a chamada a uma outra função adaptativa, caso a mesma tenha sido declarada. Depois de retornado o controle à função adaptativa chamadora, são executadas as ações elementares de consulta, remoção e adição, nesta ordem necessariamente. E finalmente, antes de finalizar a execução da função adaptativa, existe a possibilidade da chamada a uma nova função adaptativa, caso a mesma tenha sido declarada. A Figura 9 ilustra a seqüência descrita.

Exemplo

As listagens abaixo apresentam trechos de código-fonte em Pascal, com a definição e implementação da classe aa, descendente de aFuncao. Estes

trechos foram extraídos do módulo PalindromeImpar.pas gerado pela ferramenta RSW. Uma listagem completa do módulo PalindromeImpar.pas se encontra no Anexo II – Módulos em Pascal do Exemplo 1.

```
1- aa= class ( aFuncao )
2-
3-    public
4-       PROCEDURE DefinirConfiguracaoInicial;
5-       override;
6-    end;
```

Listagem 7 - Definição da classe aa

A linha 1 apresenta a definição da classe aa4, descendente de aFuncao. Na linha 4 o método virtual DefinirConfiguracaoInicial é definido sem parâmetros de entrada ou saída. Este é o único método necessário de ser definido quando se utiliza a classe aFuncao como ancestral, pois todo o comportamento de uma função adaptativa está encapsulado na classe aFuncao. A linha 5 apresenta a palavra reservada em Pascal que indica que este método deve ser sobreposto ao método virtual definido na classe ancestral.

```
1- PROCEDURE aa.DefinirConfiguracaoInicial;
2-
3- var
4-    AcaoPre : aAcaoAdaptativa;
5-    AcaoPos : aAcaoAdaptativa;
6- begin
7-    %Definicao do numero de parametros,
8-    %variaveis e geradores da funcao
9-    NumeroParametros := 3;
10-      NumeroVariaveis := 0;
11-      NumeroGeradores := 2;
12-
13-      %Inicializa as tabelas internas de variaveis
14-      %conforme os numeros ja definidos
15-      inherited DefinirConfiguracaoInicial;
16-
17-      %Definicao das acoes adaptativas elementares
18-
19-      %-[(n, "(") -> i : a(i, j, n)]
20-      AcaoPos := aAcaoAdaptativa.CreateFunction(
```

```
21-     (Self.Owner as aSubMaquina).ObterFuncaoNome( 'a'
  ));
22-     AcaoPos.AdicionarParametroOffset(0);
23-     AcaoPos.AdicionarParametroOffset(1);
24-     AcaoPos.AdicionarParametroOffset(2);
25-     DefinirAcaoElementar( 'e' , nil, 40, nil,
26-                 ORD(emptyTK), nil, nil, nil, AcaoPos,
  2, 0);
27-
28-     %+[(i, "(") -> k : a(k, m, i)]
29-     AcaoPos := aAcaoAdaptativa.CreateFunction(
30-            (Self.Owner as aSubMaquina).ObterFuncaoNome(
  'a' ));
31-     AcaoPos.AdicionarParametroOffset(3);
32-     AcaoPos.AdicionarParametroOffset(4);
33-     AcaoPos.AdicionarParametroOffset(0);
34-     DefinirAcaoElementar( 'i' , nil, 40, nil,
35-                 ORD(emptyTK), nil, nil, nil, AcaoPos,
  0, 3);
36-
37-     %+[(k, "a") -> m]
38-     DefinirAcaoElementar( 'i' , nil, 97, nil,
39-                 ORD(emptyTK), nil, nil, nil, nil, 3,
  4);
40-
41-     %+[ (m, ")") -> j ]
42-     DefinirAcaoElementar( 'i' , nil, 41, nil,
43-                 ORD(emptyTK), nil, nil, nil, nil, 4,
  1);
44-
45-     %+[ (n, "(") -> i ]
46-     DefinirAcaoElementar( 'i' , nil, 40, nil,
47-                 ORD(emptyTK), nil, nil, nil, nil, 2,
  0);
48-  end;
```

Listagem 8 - Implementação da classe aa

A implementação do método DefinirConfiguracaoInicial da classe aa, ou de qualquer outra classe representando uma função adaptativa, é responsável por definir o número de parâmetros, variáveis e geradores, iniciar as tabelas internas da classe, definir as funções adaptativas inicial e final, e definir o conjunto de ações adaptativas elementares que serão executadas quando a função for acionada.

A linha 21 apresenta a forma de iniciar as tabelas internas da classe aFuncao. O comando **inherited DefinirConfiguracaoInicial** ativa o

método homônimo da classe base, que no caso é aFuncao. Este método deve sempre ser ativado após a iniciação das propriedades NumeroParametros, NumeroVariaveis e NumeroGeradores.

As linhas 26 à 56 apresentam exemplos de definição de ações adaptativas elementares através do método DefinirAcaoElementar descrito anteriormente.

Definido o conjunto de ações adaptativas elementares, o objeto instanciado está pronto para ser utilizado no reconhecimento.

aAcaoAdaptativa

A classe aAcaoAdaptativa implementa o comportamento da chamada de uma função adaptativa. Cada transição pode conter até duas instâncias deste objeto, uma correspondente à chamada da ação adaptativa anterior e outra à chamada da ação adaptativa posterior. Cada objeto possui um número de argumentos que serão transferidos para a função adaptativa selecionada. Este número, bem como seu tipo, átomo, estado ou sub-máquina, devem estar de acordo com o declarado no cabeçalho da função adaptativa.

Funcionamento Geral

(a) Na Criação do Objeto

Diferente das classes aCompilador, aSubMaquina, aEstado e aFuncao, a classe aAcaoAdaptativa não necessita de uma classe descendente que sobreponha métodos da classe base para a definição da configuração inicial do objeto. Para criar uma instância de aAcaoAdaptativa, basta declarar uma variável do tipo aAcaoAdaptativa, e utilizar os métodos disponíveis para especificar as informações da ação adaptativa.

No momento da instanciação de um objeto da classe aAcaoAdaptativa, uma referência à função adaptativa que será ativada, é passada como argumento para o construtor da classe aAcaoAdaptativa, denominado CreateFunction. A forma da referência da função adaptativa é um ponteiro para um objeto da classe aFuncao.

A definição dos argumentos passados como parâmetros à função adaptativa é realizada através dos métodos para adição dos parâmetros, denominadas AdicionarParametroSubMaquina, AdicionarParametroEstado, AdicionarParametroAtomo e AdicionarParametroOffset,. Os três primeiros métodos adicionam ao conjunto de parâmetros, respectivamente, um parâmetro do tipo aEstado, inteiro e aSubMaquina. O quarto método, denominado AdicionarParametroOffset, é utilizado numa situação muito particular, em que uma chamada de função adaptativa está sendo definida dentro de uma ação adaptativa elementar e é utilizado como argumento para a chamada da função adaptativa, uma variável local qualquer definida na função adaptativa em que a ação elementar está sendo definida. Neste caso é necessário apenas registrar no objeto da classe aAcaoAdaptativa, qual a posição no vetor de variáveis locais da função adaptativa, correspondente à variável utilizada na chamada da função adaptativa. No momento da execução da ação adaptativa elementar, seja ela de inclusão, remoção ou consulta, será recuperada, do vetor de variáveis locais, através de seu índice numérico, uma referência para o objeto que assumirá o papel do argumento da chamada da função adaptativa. Assim, antes de adicionar uma transição ao conjunto de transições existentes num estado, ou remover uma transição deste mesmo conjunto, o objeto da classe aAcaoAdaptativa tem as referências, na forma de ponteiros para objetos, de todos seus argumentos.

O vetor de variáveis locais é formado pelos parâmetros, variáveis e geradores declarados na função adaptativa, nesta ordem necessariamente. Portanto o comprimento total do vetor é igual à soma do número de parâmetros com o número de variáveis e o número de geradores, sendo que o primeiro parâmetro definido assume a posição de índice zero no vetor, o segundo parâmetro definido assume a posição de índice um, e assim por diante.

Definida a função adaptativa e os argumentos que serão passados para esta quando da sua chamada, um objeto da classe aAcaoAdaptativa estará pronto para ser utilizado no reconhecimento de uma cadeia de entrada. Este objeto

pode ser utilizado na definição de uma transição que contenha uma ação adaptativa anterior e/ou posterior.

(b) No Reconhecimento

Durante o reconhecimento de uma cadeia de entrada, o estado corrente, baseado no átomo de entrada, determinará uma transição a ser executada. Esta transição pode apresentar as ações adaptativas anterior ou posterior, representadas respectivamente por um ou dois objetos da classe aAcaoAdaptativa. É de responsabilidade do estado corrente acionar o método Executar, de um objeto da classe aAcaoAdaptativa, no momento apropriado, tendo em vista se o objeto representa uma ação adaptativa anterior ou posterior.

O método Executar da classe aAcaoAdaptativa inicia a operação de chamada uma função adaptativa verificando se a referência para o objeto da classe aFuncao é válido. Sendo válido, é chamado o método Inicializar da classe aFuncao. Tendo o objeto da classe aFuncao iniciado, começa a transferência dos argumentos definidos em aAcaoAdaptativa para a função adaptativa. Finalizada estas duas etapas, o método Executar da classe aFuncao é chamado, marcando o final da operação de chamada da função adaptativa e dando prosseguimento à execução propriamente dita da função adaptativa.

Exemplo

A Listagem 9 e a Listagem 10 apresentam trechos de código-fonte em Pascal, com a utilização da classe aAcaoAdaptativa em diferentes momentos. Estes trechos foram extraídos do módulo PalindromeImpar.pas gerado pela ferramenta RSW. A listagem completa do módulo PalindromeImpar.pas encontra-se no Anexo II – Módulos em Pascal do Exemplo 1.

```
1- procedure ae1.DefinirConfiguracaoInicial;
2- var
3-    AcaoPos : aAcaoAdaptativa;
4- begin
5-    inherited DefinirConfiguracaoInicial;
```

```
6-
7-    AcaoPos := aAcaoAdaptativa.CreateFunction(
8-        (Self.Owner as aSubMaquina).ObterFuncaoNome( 'a'
  ));
9-
10-     AcaoPos.AdicionarParametroEstado(
11-         (Self.Owner as aSubMaquina).ObterEstadoNome(
  'e2' ));
12-     AcaoPos.AdicionarParametroEstado(
13-         (Self.Owner as aSubMaquina).ObterEstadoNome(
  'e3' ));
14-     AcaoPos.AdicionarParametroEstado(
15-         (Self.Owner as aSubMaquina).ObterEstadoNome(
  'e1' ));
16-
17-     DefinirTransicao(40,
18-              (Self.Owner as
  aSubMaquina).ObterEstadoNome( 'e2' ),
19-              ORD(emptyTK), nil, nil, nil, AcaoPos);
20-
21-     DefinirTransicao(97,
22-              (Self.Owner as
  aSubMaquina).ObterEstadoNome( 'e4' ),
23-              ORD(emptyTK), nil, nil, nil, nil);
24-
25-  end;
```

Listagem 9 - Uso da classe aAcaoAdaptativa dentro de aEstado

A Listagem 9 apresenta o uso da classe aAcaoAdaptativa dentro da implementação do método DefinirConfiguracaoInicial, da classe ae1, descendente da classe aEstado. A linha 7 da Listagem 9 apresenta a criação de um objeto da classe aAcaoAdaptativa, através do construtor CreateFunction. Este método recebe como parâmetro a referência de um objeto da classe aFuncao, que representa a função adaptativa denominada 'a'. Observe que a procura da função adaptativa é realizada através do método ObterFuncaoNome, da classe aSubMaquina,.

A função adaptativa 'a', como apresentada na Listagem 11 - Palíndrome ímpar da forma $(^{n}a)^{n}$, página 81, define três parâmetros do tipo estado. As linhas 10 à 15 definem os três argumentos obrigatórios para a chamada da função adaptativa 'a'. Os três estados são recuperados através do método ObterEstadoNome, da classe aSubMaquina. A procura dos estados se

baseia no nome dos mesmos, que é definido pela cadeia de caracteres passada como argumento na chamada do método ObterEstadoNome.

Definido o objeto do tipo aFuncao que representa a função adaptativa a ser chamada, e os parâmetros obrigatórios que a função adaptativa espera, a instância da classe aAcaoAdaptativa está pronta para ser utilizada na definição de transições do estado. As linhas 17 a 19 apresentam a adição de uma transição ao conjunto de transições do objeto da classe ae1. Esta transição contém uma ação adaptativa posterior, correspondendo à chamada da função adaptativa a(e2,e3,e1).

A Listagem 10 apresenta o uso da classe aAcaoAdaptativa dentro da implementação do método DefinirConfiguracaoInicial, da classe aa, descendente de aFuncao. A criação do objeto da classe aAcaoAdaptativa se dá da mesma forma da Listagem 9. A diferença está na definição dos argumentos utilizados na chamada da função adaptativa 'a'.

No corpo da função adaptativa 'a' temos uma ação adaptativa elementar de eliminação de uma transição, que por sua vez contém uma ação adaptativa posterior, chamando a função adaptativa 'a'. Esta chamada apresenta três argumentos do tipo estado, sendo que estes argumentos são variáveis locais da função adaptativa. As variáveis locais da função adaptativa só existem em tempo de execução. Portanto quando a configuração inicial da função adaptativa está sendo definida, não é possível recuperar uma referência válida para a variável local que será definida quando a função adaptativa estiver sendo chamada. É necessário armazenar o índice dentro do vetor de variáveis locais, onde se encontrará a variável desejada. No caso exemplificado nas linhas 13 a 15, as variáveis que serão utilizadas como argumentos para a chamada da função adaptativa 'a', estarão, respectivamente, nas posições de índices 0, 1, e 2 do vetor de variáveis locais. A linha 5 apresenta a definição do número de parâmetros da função adaptativa, que é igual a três. Como o primeiro parâmetro definido na função adaptativa assume a posição de índice zero, concluímos que os

argumentos definidos nas linhas 13 a 15 são os três parâmetros recebido pela função adaptativa no momento da sua execução.

Finalizada a definição dos argumentos a serem utilizados na chamada da função adaptativa, o objeto da classe aAcaoAdaptativa está pronto para servir como argumento na adição de uma ação adaptativa elementar, seja ela de eliminação, adição ou consulta.

```
1- procedure aa.DefinirConfiguracaoInicial;
2- var
3-    AcaoPos : aAcaoAdaptativa;
4- begin
5-    %Definicao do numero de parametros, variaveis e
  geradores
6-    NumeroParametros := 3;
7-    NumeroVariaveis := 0;
8-    NumeroGeradores := 2;
9-    inherited DefinirConfiguracaoInicial;
10-     %Criacao do objeto aAcaoAdaptativa
11-     AcaoPos := aAcaoAdaptativa.CreateFunction(
12-              (Self.Owner as
  aSubMaquina).ObterFuncaoNome( 'a' ));
13-     %Definicao dos parametros do objeto aAcaoAdaptativa
14-     AcaoPos.AdicionarParametroOffset(0);
15-     AcaoPos.AdicionarParametroOffset(1);
16-     AcaoPos.AdicionarParametroOffset(2);
17-     %Definicao do conjunto de acoes adaptativas
  elementares
18-     DefinirAcaoElementar( 'e' , nil, 40, nil,
  ORD(emptyTK),
19-                           nil, nil, nil, AcaoPos, 2,
  0);
20-
21-     DefinirAcaoElementar( 'i' , nil, 97, nil,
  ORD(emptyTK),
22-                           nil, nil, nil, nil, 3, 4);
23-     DefinirAcaoElementar( 'i' , nil, 41, nil,
  ORD(emptyTK),
24-                           nil, nil, nil, nil, 4, 1);
25-     %Criacao do objeto aAcaoAdaptativa
26-     AcaoPos := aAcaoAdaptativa.CreateFunction(
27-              (Self.Owner as
  aSubMaquina).ObterFuncaoNome( 'a' ));
28-     %Definicao dos parametros do objeto aAcaoAdaptativa
29-     AcaoPos.AdicionarParametroOffset(3);
30-     AcaoPos.AdicionarParametroOffset(4);
31-     AcaoPos.AdicionarParametroOffset(0);
```

```
32-
33-     DefinirAcaoElementar( 'i' , nil, 40, nil,
  ORD(emptyTK),
34-                           nil, nil, nil, AcaoPos, 0,
  3);
35-
36-     DefinirAcaoElementar( 'i' , nil, 40, nil,
  ORD(emptyTK),
37-                           nil, nil, nil, nil, 2, 0);
38-
39-  end;
```

Listagem 10 - Uso da classe aAcaoAdaptativa dentro de aFuncao

As Classes aTabelaMaquinas e aTabelaEstados

Estas classes representam uma estrutura de dados do tipo vetor, em que cada um de seus elementos são objetos das classes aSubMaquina e aEstado, respectivamente.

Os métodos públicos disponíveis são responsáveis pela adição, remoção e busca destes elementos.

Cada elemento armazenado pode ser obtido através de um valor numérico que representa o índice em correspondência ao qual este elemento está localizado em relação ao início do vetor, ou através de uma cadeia de caracteres que está associada a cada elemento. Esta cadeia de caracteres é a mesma atribuída ao nome do objeto, e através dela é que se realiza a maioria das buscas.

Objetos destes tipos são criados no momento em que seus proprietários, aCompilador e aSubMaquina, também o são. Estas classes são descendentes de uma classe da biblioteca de estrutura de dados do Delphi 3.0, ferramenta de desenvolvimento escolhida para a implementação da ferramenta RSW. Uma das principais características desta classe é o comprimento variável da estrutura de dados vetor. Inicialmente, esse comprimento é zero. Conforme os objetos vão sendo adicionados, o comprimento deste vetor vai sendo incrementado. Por esta razão, o número de sub-máquinas e estados que podem ser armazenados está vinculado à

quantidade de memória disponível no momento da utilização das instâncias das classes aTabelaMaquinas e aTabelaEstados .

aTabelaTransicao

Apesar de também representar uma estrutura de dados do tipo vetor, aTabelaTransicao apresenta características totalmente diferentes. Cada elemento do vetor armazena um registro que descreve uma transição.
Os componentes do registro armazenado nesta estrutura são:

- um ponteiro para o objeto que representa o estado de destino
- um inteiro que representa o átomo de saída, podendo este ser o átomo de retorno ou o átomo quer deverá ser empilhado na cadeia de entrada
- um ponteiro para o objeto que representa a ação adaptativa anterior
- um ponteiro para o objeto que representa a ação adaptativa posterior
- um ponteiro para o objeto que representa a sub-máquina a ser chamada, nos casos de transições de chamada de sub-máquina
- um ponteiro para a rotina semântica

O átomo de entrada é definido pela posição relativa ou índice em que este registro é armazenado no vetor. O último parâmetro que falta para termos uma produção completa, a sub-máquina corrente, é obtida através da linguagem de implementação do sistema.
Outra diferença desta classe, é a forma de alocação da memória para os elementos do vetor. O comprimento do vetor é fixo e definido pelo número de diferentes átomos que podem aparecer na cadeia de entrada, mais o átomo que representa o final da cadeia de entrada, mais o átomo que representa uma transição em vazio.

Ciclo de Implementação

A ferramenta RSW foi desenvolvida através de um processo incremental de funcionalidades, ou seja, a partir de uma versão com um certo número

mínimo de funcionalidades, é executado um "bootstrap", e, em conjunto com o desenvolvimento manual de outras funções, obtém-se a versão seguinte, com um maior número de funcionalidades. Cabe salientar que a versão obtida como resultado deste processo implementa um número maior de funcionalidades que o mesmo desenvolvimento manual implementaria na versão anterior, sem o processo de "Bootstrap".

A seguir teremos uma descrição das funcionalidades apresentadas em cada versão da ferramenta RSW, bem como as restrições sobre a teoria envolvida que se fizeram necessárias ao longo do processo de desenvolvimento.

Versão 1.00

Características

Os itens abaixo foram desenvolvidos manualmente, resultando na versão 1.00 do RSW:

- Implementação das classes básicas aCompilador, aSubMaquina, aEstado, aTabelaTransicao.
- O objeto aCompiladorRSW100, descendente de aCompilador
- Duas sub-máquinas : aGramatica e aProducao, descendentes de aSubMaquina
- Geração do código dos objetos desenvolvidos.
- Objeto de geração de código para linguagem Pascal, compatível com a ferramenta Delphi
- Objeto para manipulação e gerenciamento dos arquivos de código gerados

Restrições

Estão implementadas as seguintes restrições, que, caso sejam violadas, emitirão uma mensagem de erro:

- Toda sub-máquina definida no projeto deve ter pelo menos um estado caracterizado como estado final.

- Toda sub-máquina definida no projeto deve ter somente um estado caracterizado como inicial
- A chamada de uma sub-máquina deve ser definida através de duas transições, da seguinte maneira:
- (EstadoOrigem, AtomoOrigem) -> (^EstadoOrigem, SubMaquina [,AtomoOrigem])
- (EstadoOrigem, AtomoSubMaquina) -> (EstadoDestino, AtomoDestino).

Onde

EstadoOrigem é o estado corrente quando da análise da transição.

EstadoDestino é o próximo estado que deve ser analisado após a transição, ou da execução bem sucedida da sub-máquina chamada.

AtomoOrigem é o átomo que determina a chamada de uma sub-máquina. Este átomo não é consumido pela transição da chamada da sub-máquina.

AtomoDestino é o átomo que deverá ser empilhado caso a execução da sub-máquina chamada seja bem sucedida.

AtomoSubMaquina é o átomo empilhado pela sub-máquina chamada quando da execução bem sucedida.

SubMaquina é a sub-máquina que deve ser chamada.

- Uma transição de retorno de sub-máquina não possui ações adaptativas nem rotina semântica associada a ela.
- Uma sub-máquina referenciada numa transição de chamada de sub-máquina deve estar definida no projeto.

Versão 1.01

Características

Características desenvolvidas utilizando-se a versão 1.00 do RSW:

- Extensão da sub-máquina Gramática para definição de mais de um átomo de reconhecimento da sub-máquina
- Extensão da sub-máquina Produção para possibilitar a definição do átomo a ser empilhado quando do reconhecimento da sub-máquina como um dos átomos definidos no cabeçalho da sub-máquina
- Definição da sub-máquina Função para reconhecimento da sintaxe de descrição das funções adaptativas
- Extensão da sub-máquina Gramática para a chamada da nova sub-máquina Função

Características desenvolvidas manualmente:

- Ações semânticas para definição das funções e ações adaptativas e ações adaptativas elementares
- Implementação das classes aFuncao e aAcaoAdaptativa

Versão 2.00

Características

Características desenvolvidas utilizando-se a versão anterior:

- Reconhecimento dos tipos dos parâmetros da ação adaptativa e da função adaptativa
- Reconhecimento dos tipos das variáveis e geradores declarados na função adaptativa.

Características desenvolvidas manualmente:

- Implementação das ações adaptativas elementares de eliminação e adição

Versão 3.00

Características

Características desenvolvidas utilizando-se a versão anterior:

- Reconhecimento e coleta dos identificadores pelo analisador léxico
- Associação dos tipos das variáveis e geradores pelo analisador léxico
- Gerenciamento dos identificadores válidos tendo em vista a abertura e fechamento do bloco que definem uma função adaptativa

Características desenvolvidas manualmente:

- Ajustes das rotinas semânticas para a eliminação da tabela de símbolos

Capítulo V - Experimentos Realizados

Os exemplos desenvolvidos e relatados aqui, são os mesmos propostos em [40] quando da apresentação dos autômatos adaptativos. O objetivo da implementação dos exemplos abaixo é a verificação do funcionamento da ferramenta RSW, pois os mesmos foram fundamentais na fase de depuração da ferramenta. O objetivo de incluir e comentar estes exemplos nesta dissertação é facilitar a absorção da linguagem RSW e de sua sintaxe.

O primeiro exemplo simula o comportamento de uma pilha, quando esta é utilizada para armazenar o número de vezes que foi reconhecido um determinado átomo. O problema proposto é um autômato que reconhece a linguagem definida pela expressão $(^{n}a)^{n}$, denominada também palíndrome impar. Através da alteração do autômato inicial armazenamos o número de vezes com que é consumindo o átomo '(' e portanto o número de vezes que obrigatoriamente deverá ser consumido o átomo ')', depois do consumo do átomo 'a'.

O segundo exemplo apresenta uma solução para coletores de nomes. O coletor de nomes é responsável por reconhecer identificadores e variáveis. Um identificador é formado por letras e dígitos, iniciando-se apenas por letras. Uma variável é um identificador já reconhecido anteriormente. Através da alteração do autômato inicial são armazenados os identificadores já reconhecidos, e estes passam, a partir de então, a serem reconhecidos como variáveis.

O terceiro exemplo apresenta a segunda solução para o problema proposto no capítulo Conceitos, página 15, sobre expressões do tipo $a^{n}b^{n}c^{n}$, com n>0. Foi apresentado um autômato não reentrante como primeira solução. Aqui será apresentado um autômato reentrante que resolve o mesmo problema.

O quarto exemplo apresentado é o próprio compilador RSW desenvolvido nesta dissertação, na sua versão 3.0. Este é um exemplo interessante de aplicação real, envolvendo a linguagem RSW.

O quinto e último exemplo apresentado é um tradutor da notação de Wirth para a linguagem RSW, baseado no algoritmo desenvolvido em [29].

Exemplo 1 - Simulação de uma Pilha

Este exemplo simula a utilização de uma pilha para efetuar o reconhecimento de uma palíndrome ímpar da forma $(^n\ a\ )^n$. Sua solução foi proposta inicialmente em [28], página 178. Embora ela consista num autômato adaptativo muito simples, é com freqüência que encontramos este tipo de problema em diversas aplicações, onde normalmente se utiliza uma pilha explícita.

```
1- submaquina PalindromeImpar ( eofTK )
2-
3- %Obtêm os átomos de entrada da cadeia de entrada,
4- %ou seja, os caracteres ASC.
5- entrada CadeiaEntrada
6-
7- producoes
8-
9-    (e1, "a") -> e4.
10-  (e1, "(") -> e2 : a (e2,e3,e1).
11-
12-  (e2, "a") -> e3.
13-
14-  (e3, ")") -> e4.
15-
16-  %Reconhecimento finalizado.  Retorna a execucao
17-  %empilhando o átomo eofTK
18-  (^*, e4 ) -> ( ^, *, eofTK).
19-
20-  funcoes
21-
22-  a(estado i, estado j, estado n) =
23-  {
24-     %declaracao dos novos estados
25-       estado *k, estado *m:
26-
27-     %Adiciona os novos estados na estrutura do autômato
28-     %correspondentes ao reconhecimento de expressao ( a
  )
29-       +[ (k, "a") -> m ]
30-       +[ (m, ")") -> j ]
31-       +[ (n, "(") -> i ]
32-
33-     %Atualiza a posição e os parametros
34-     %da chamada da funcao adaptativa conforme
```

```
35-      %os novos estados
36-        +[ (i, "(") -> k : a(k, m, i) ]
37-        -[ (n, "(") -> i : a(i, j, n) ]
38-   }
```

Listagem 11 - Palíndrome ímpar da forma $(^n a)^n$

Para entendermos como o autômato adaptativo implementa, em sua estrutura, o comportamento de pilha, vamos visualizar a máquina de estados em sua configuração inicial, analisar as alterações realizadas pela função adaptativa e a posterior configuração da máquina.

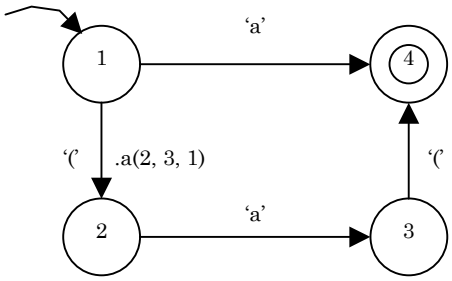

Figura 10 - Máquina inicial do simulador de pilha

A máquina inicial apresentada na Figura 10 é capaz de reconhecer a expressão 'a' ou '(' 'a' ')'. A transição do estado 1 para 2 ativa a função adaptativa a. A cada execução da função adaptativa a são incorporados ao autômato dois novos estados e transições entre eles que reconhecem a seqüência adicional '(' 'a' ')'. Assim sendo, a configuração da máquina após reconhecido o primeiro átomo '(' e acionada a função adaptativa a, é ilustrada na Figura 11.

A máquina apresentada na Figura 11 é capaz de reconhecer as seqüências reconhecidas pela máquina anterior (Figura 10), e também a expressão '(' '(' 'a' ')' ')'. O próximo átomo '(' que vier a ser consumido ativará novamente a função adaptativa a, e esta por sua vez alterará a máquina para que seja reconhecida a expressão '(' '(' '(' 'a' ')' ')' ')', e assim por diante.

Uma característica interessante a ser observada neste autômato é que, após o reconhecimento bem sucedido de uma expressão qualquer, o autômato assumirá uma configuração que, além de reentrante, será mais eficiente em desempenho do que a anterior para qualquer cadeia de entrada de

comprimento menor ou igual à maior cadeia já reconhecida anteriormente. No caso de a nova cadeia de entrada apresentar um maior número de átomos '(' que a anterior, o autômato deverá realizar as alterações necessárias, através da execução da função a, para reconhecer o número de átomos '(' que excede a primeira cadeia de entrada. Portanto em lugar de executar a função o número de vez que o átomo '(' aparece na segunda cadeia de entrada, esta só será acionada o número de vezes correspondente a diferença numérica de átomos '(' entre a cadeia corrente e a mais longa cadeia previamente reconhecida.

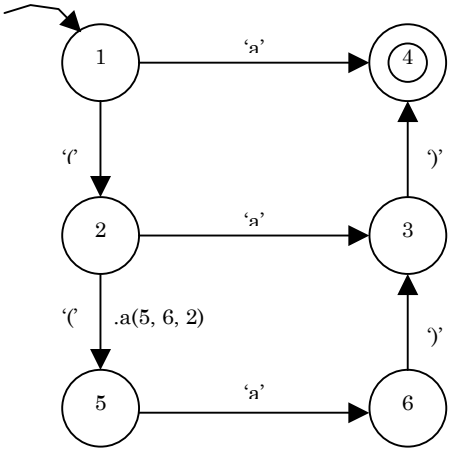

Figura 11 - Simulador após a primeira execução da função a

No caso de a nova cadeia de entrada apresentar um número menor de átomos '(', o autômato já se apresentará na configuração necessária para reconhecê-la diretamente, e ainda de forma tão eficiente quanto a de um autômato finito.

Aninhamentos em geral, característicos de linguagens livres de contexto não-regulares, podem ser tratados desta forma, através de autômatos extremamente eficientes quanto ao tempo de resposta, já que operam essencialmente como autômatos finitos exceto na ocasião das transições adaptativas.

Exemplo 2 - Coletor de Nomes

Este exemplo simula a coleta de nomes da cadeia de entrada. A proposta do coletor de nomes é que cada nome coletado seja classificado, na primeira vez que for encontrado, como identificador. A partir da segunda vez que o mesmo nome for encontrado no texto de entrada, sua classificação passa a ser de variável. A solução adotada para o coletor de nomes, apresentada na Listagem 13, e complementada pela Listagem 12, está detalhadamente disponível em [28], página 180.

A importância deste exemplo será observada no Exemplo 4 – Compilador RSW, quando apresentamos o próprio compilador RSW utilizando a técnica aqui apresentada, mais precisamente na máquina responsável pela análise léxica.

Encontramos abaixo duas listagens, na linguagem RSW. A Listagem 12 descreve uma máquina que tem o único objetivo de acionar repetidas vezes o coletor de nomes. Esta máquina consome os dois possíveis átomos retornados pelo coletor de nomes até que a cadeia de entrada seja totalmente consumida. Associada às transições que consomem os átomos retornados pelo coletor de nomes, existem ações semânticas. As ações semânticas ImprimirVariavel e ImprimirNovoIdentificador são acionadas como o objetivo de criar um arquivo texto de saída com os nomes coletados da cadeia de entrada, e sua respectiva classificação.

```
1- submaquina Exemplo2 ( eofTK )
2-
3- entrada ColetorNomes
4-
5- producoes
6-
7- (e1, varTK) -> e1 <ImprimirVariavel>.
8- (e1, idTK) -> e1 <ImprimirNovoIdentificador>.
9- (^*, e1 ) -> ( ^, *, eofTK).
```

Listagem 12 - Máquina acionadora do coletor de nomes

O coletor de nomes apresentado na Listagem 13, na linguagem RSW, reconhece como identificador válido, nomes iniciados por letras, seguidos de uma seqüência de letras e dígitos, eventualmente vazia.

Resumidamente, a idéia adotada pelo coletor de nomes é a de se auto-modificar através da função adaptativa B( ), conforme as letras e dígitos de um novo nome vão sendo consumidos, criando assim, um caminho de estados e transições que reconhecem a cadeia consumida. Atingido o término do identificador, ou seja, encontrando caracteres delimitadores, é realizada uma transição para o estado Est8, a qual empilhará o átomo correspondente ao reconhecimento de um novo identificador, idTK, e também executará a função adaptativa D( ), responsável por alterar o estado final a ser utilizado no próximo reconhecimento daquele mesmo nome. Esta função adaptativa define um novo estado final, que empilhará o átomo correspondente ao reconhecimento de uma variável.

```
1- submaquina ColetorNomes ( varTK, idTK, eofTK )
2-
3- entrada CadeiaEntrada
4-
5- producoes
6-
7- %( Est3, <Letras> ):B(Est3,<Letra>) -> Est3a.
8- (Est3, "a") : B(Est3, "a") -> Est3a.
9- (Est3, "b") : B(Est3, "b") -> Est3a.
10-  %... Transicoes de b a y ...
11-  (Est3, "z") : B(Est3, "z") -> Est3a.
12-
13-  %(Est3a, <Letras e digitos>) -> Est3a.
14-  (Est3a, "a")-> Est3a.
15-  (Est3a, "b")-> Est3a.
16-  %... Transicoes de b a y ...
17-  (Est3a, "z")-> Est3a.
18-  (Est3a, "0")-> Est3a.
19-  %... Transicoes de 1 a 8 ...
20-  (Est3a, "9")-> Est3a.
21-
22-  %Final do identificador
23-  (Est3a, 10) -> Est8 : D(Est3a) . %Line Feed
24-  (Est3a, 13) -> Est8 : D(Est3a) . %Carriage Return
25-  (Est3a, 32) -> Est8 : D(Est3a) . %Espaco
26-  (Est3a, 09) -> Est8 : D(Est3a) . %Tab
```

```
27-
28-  %Reconhecido Novo Identificador
29-  (^*, Est8 ) -> ( ^, *, idTK).
30-
31-  %Reconhecido Variavel
32-  (^*, Est9 ) -> ( ^, *, varTK).
33-
34-
35-  funcoes
36-
37-  %Funcao Adaptativa acionada antes de realizar uma
38-  %transicao a partir do estado I, consumindo o atomo A
39-  B( estado I, atomo A)=
40-  {
41-     estado *j:
42-
43-  %Adiciona uma transicao para o novo estado
44-  %consumindo o atomo corrente
45-     +[(I, A) -> j]
46-
47-  %Remove a chamada desta funcao adaptativa
48-  %na transicao do estado I consumindo o atomo A
49-     -[(I, A):B(I,A) -> Est3a]
50-
51-  %Adiciona o conjunto de transicoes para os possiveis
52-  %atomos a serem consumidos a partir do novo estado
53-     +[(j, "a"):B(j, "a")-> Est3a.
54-     +[(j, "b"):B(j, "b")-> Est3a.
55-  %... Transicoes de b a y ...
56-     +[(j, "z"):B(j, "z")-> Est3a.
57-     +[(j, "0"):B(j, "0")-> Est3a.
58-  %... Transicoes de 1 a 8 ...
59-     +[(j, "9"):B(j, "9")-> Est3a.
60-
61-  %Final do identificador
62-     +[(j, 10) -> Est8 : D(j)] %Line Feed
63-     +[(j, 13) -> Est8 : D(j)] %Carriage Return
64-     +[(j, 32) -> Est8 : D(j)] %Espaco
65-     +[(j, 09) -> Est8 : D(j)] %Tab
66-  }
67-
68-  %Funcao adaptativa acionada depois de realizada
69-  %a transicao para Est8
70-  D(estado I)=
71-  {
72-     %Remove as transicoes de termino do identificador,
73-     %a partir do estado corrente para Est8
74-     -[(I, 10) -> Est8 : D(I)] %Line Feed
75-     -[(I, 13) -> Est8 : D(I)] %Carriage Return
76-     -[(I, 32) -> Est8 : D(I)] %Espaco
```

```
77-     -[(I, 09) -> Est8 : D(I)] %Tab
78-
79-     %Adiciona as transicoes de termino do
  identificador,
80-     %do estado corrente para Est9
81-     +[(I, 10) -> Est9] %Line Feed
82-     +[(I, 13) -> Est9] %Carriage Return
83-     +[(I, 32) -> Est9] %Espaco
84-    +[(I, 09) -> Est9] %Tab
85-  }
```

Listagem 13 - Coletor de nomes

Exemplo 3 - Expressão $a^n b^n c^n$, com n>0

O problema proposto no Capítulo II, página 12, apresenta pelo menos duas soluções, ambas descritas neste trabalho. A primeira solução foi analisada no mesmo capítulo do enunciado do problema. A segunda solução será analisada a seguir.

A Listagem 14 apresenta a descrição do reconhecedor, na linguagem RSW, que reconhece expressões da forma $a^n b^n c^n$, com n>0.

```
1- submaquina Solucao2AnBnCn ( eofTK )
2-
3- entrada CadeiaEntrada
4-
5- producoes
6-
7- (e0, "a") -> e1 : RemoverTransicao( e2, e3).
8-
9- (e1, "a") -> e1 : AdicionarEstados( e1, e2, e3).
10-  (e1, "b") -> e2 : DefinirTransicao( e2, e3).
11-
12-  (e2, "c") -> e3.
13-
14-  (e3, emptyTK) -> e4.
15-
16-  (^*, e4 ) -> ( ^, *, eofTK).
17-
18-  funcoes
19-
20-  AdicionarEstados ( estado EstadoCorrente,
21-                     estado EstadoB,
22-                     estado EstadoC ) =
23-  {
24-       estado *NovoEstadoBloco1,
25-     estado *NovoEstadoBloco2,
26-     estado *NovoEstadoBloco3
27-     :
28-
29-       -[(EstadoCorrente,"a")
30-        :Adicionar(EstadoCorrente, EstadoB, EstadoC)
31-           -> EstadoCorrente]
32-       +[(EstadoCorrente, "a") -> NovoEstadoBloco1]
33-       +[(NovoEstadoBloco1, "a")
34-        :Adicionar(NovoEstadoBloco1,
35-                   NovoEstadoBloco2,
```

```
36-                    NovoEstadoBloco3)
37-            -> NovoEstadoBloco1]
38-       +[(NovoEstadoBloco1, "b") -> e2
39-        : DefinirTransicao( NovoEstadoBloco2,
  NovoEstadoBloco3)]
40-       +[(EstadoB,"b") -> NovoEstadoBloco2]
41-       +[(EstadoC,"c") -> NovoEstadoBloco3]
42-  }
43-
44-  DefinirTransicao( estado EstadoB, estado EstadoC)=
45-  {
46-
47-       +[(EstadoB, "c")     -> e3]
48-       +[(EstadoC, emptyTK) -> e4]
49-
50-       +[(e0, "a") -> e1 : RemoverTransicao( EstadoB,
  EstadoC)]
51-  }
52-
53-  RemoverTransicao( estado EstadoB, estado EstadoC)=
54-  {
55-     -[(EstadoB, "c")     -> e3]
56-     -[(EstadoC, emptyTK) -> e4]
57-
58-       -[(e0, "a") -> e1 : RemoverTransicao( EstadoB,
  EstadoC)]
59-       +[(e0, "a") -> e1]
60-  }
```

Listagem 14 - Solução em RSW para a expressão $a^n b^n c^n$, com n>0

A solução aqui apresentada é dita reentrante, pois podemos reutilizar a configuração final da máquina após o reconhecimento de uma cadeia de entrada qualquer, e para compreendermos a idéia desta solução vamos acompanhar as alterações realizadas na máquina inicial, ao longo do reconhecimento da cadeia de entrada "aaabbbccc".

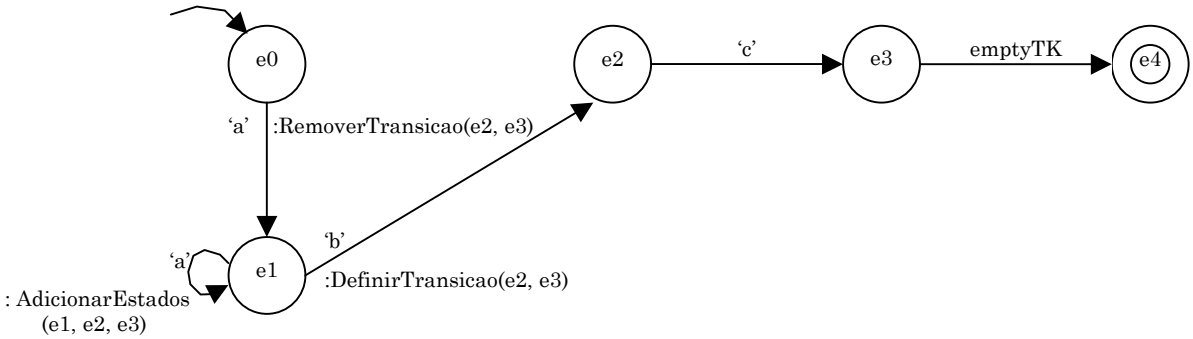

Figura 12 – Configuração inicial da solução 2 para $a^n b^n c^n$, com n>0

A Figura 12 apresenta a configuração inicial do autômato descrito na Listagem 14. Desconsiderando as ações adaptativas presentes no autômato, observamos que é reconhecida a cadeia de entrada 'abc', o caso particular para n=1. Antes de analisarmos os passos para a solução geral, vamos considerar um autômato qualquer que reconheça a linguagem $a^n b^n c^n$. Para facilitar o entendimento da solução 2, considere este autômato composto por três blocos de estados e transições, sendo que cada um destes blocos é responsável por reconhecer parte da linguagem $a^n b^n c^n$. Assim sendo, o primeiro bloco de estados e transições é responsável por reconhecer a linguagem a^n, o segundo bloco de estados e transições é responsável por reconhecer a linguagem b^{n-1}, e finalmente o terceiro bloco de estados e transições é responsável por reconhecer a linguagem c^{n-1}. Para maior simplicidade, vamos denominar estes blocos, respectivamente, de bloco1, bloco2 e bloco3. Estes blocos são unidos através de uma transição que determina o término do bloco e o início do outro, e no caso do bloco3, esta transição é em vazio para o estado final que empilha o átomo de reconhecimento. Portanto a configuração inicial da Figura 12 é composta de três blocos de estados e transições, bloco1, cbloco2 e bloco3 que reconhecem, respectivamente, a^1, b^0 e c^0. A Figura 13 ilustra a divisão dos blocos de estados e transições na configuração inicial do autômato.

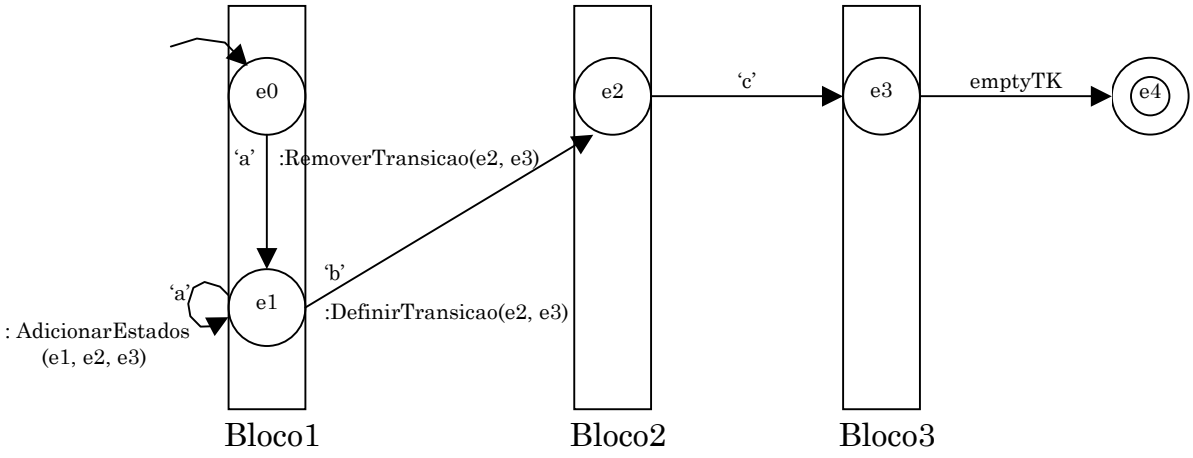

Figura 13 - Autômato dividido em blocos

Para reconhecer a linguagem $a^n b^n c^n$, para n>1, serão realizadas as seguintes alterações na configuração inicial:

- Depois de consumido o primeiro átomo ‘a’, para cada novo átomo ‘a’ consumido, será atualizado os três blocos de estados e transições para que estes reconheçam, respectivamente, as cadeias de entrada a ^{n}, b^{n-1} e c^{n-1}, sendo n o número de átomos ‘a’ já consumidos. Esta atualização além de expandir os blocos de estados e transições adicionando um novo estado, deve também adicionar uma transição consumindo o átomo ‘b’ a partir do novo estado do bloco1 para o estado inicial do bloco2. Esta ação é tomada pela função adaptativa AdicionarTransicao.
- Finalizado o consumo de todos os átomos ‘a’, a primeira transição que consume o átomo ‘b’, deverá adicionar uma transição consumindo o átomo ‘c’ no estado apropriado do bloco2 para o estado inicial do bloco3, e também uma transição em vazio no estado apropriado do bloco3 para o estado final. Portanto a primeira transição que consume o átomo ‘b’ é responsável por determinar as transições que finalizam os blocos bloco2 e bloco3. Esta ação é tomada pela função adaptativa DefinirTransicao.
- Para tornar o autômato reentrante, é necessário armazenar a posição das transições que finalizam os blocos bloco2 e bloco3, para serem eliminadas no início de um novo reconhecimento. Esta ação é tomada pela função adaptativa RemoverTransicao.

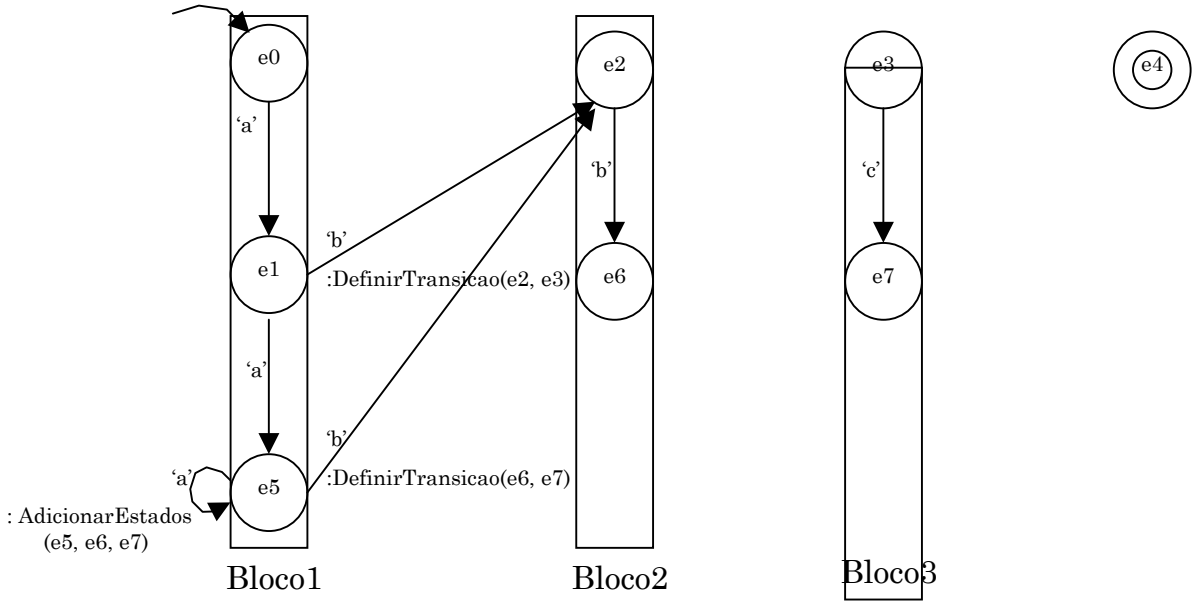

Figura 14 – Configuração após a primeira execução da função adaptativa AdicionarEstados

A Figura 14 apresenta a configuração do autômato após o consumo do segundo átomo ‘a’ e da execução da função adaptativa AdicionarTransição. Observe que as transições do bloco2 para o bloco3, e do bloco3 para o estado final foram eliminadas pela função adaptativa RemoverTransicao. Estas transições serão definidas pela função adaptativa DdefinirTransicao, quando o primeiro átomo ‘b’ for consumido.

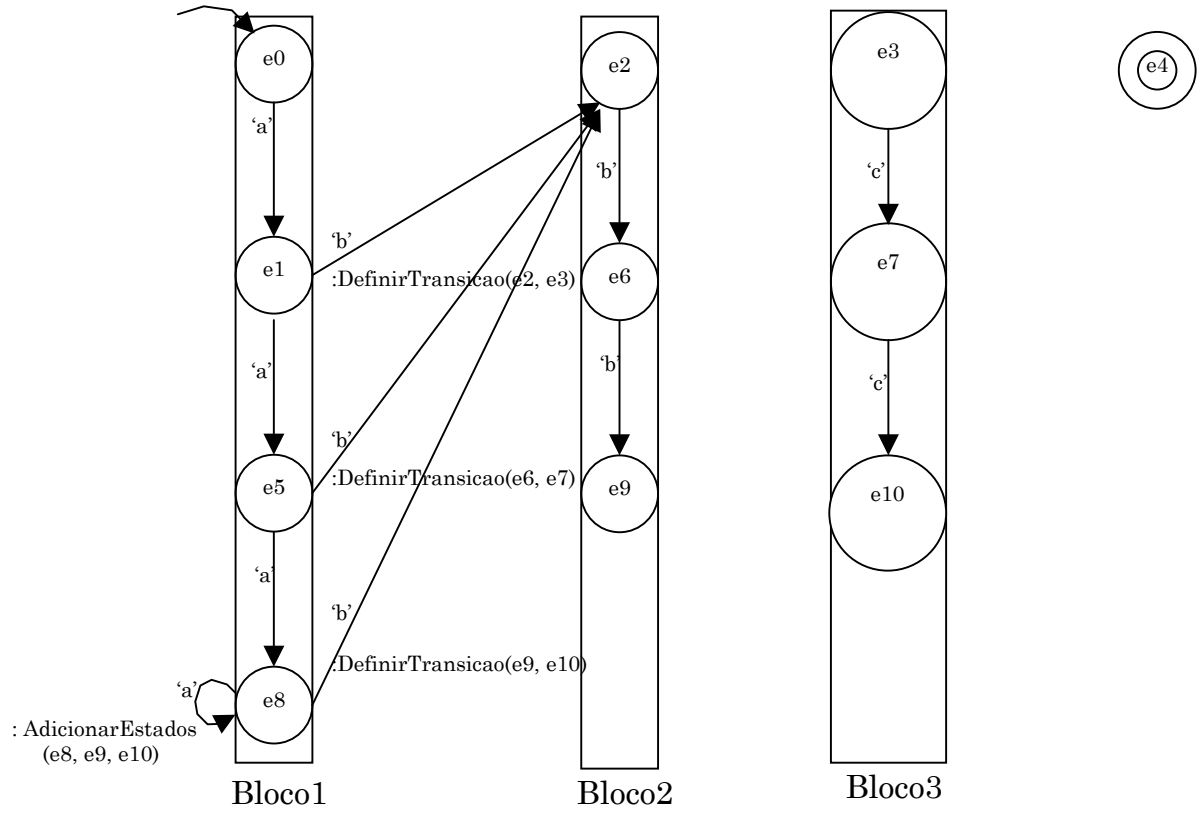

Figura 15 - Autômato depois de consumida a entrada "aaa"

Após consumir a primeira parte da cadeia de entrada, “aaa”, a configuração do autômato assume a representação da Figura 15. Neste momento, o átomo disponível na cadeia de entrada é ‘b’, e o estado corrente é e8. Para realizar a transição de e8 para e2, consumindo o átomo ‘b’, a função adaptativa DefinirTransicao é ativada, tendo os estados e9 e 10 como argumento.

A função adaptativa DefinirTransicao adiciona a transição consumindo o átomo ‘c’ do bloco2 para o bloco3, a transição em vazio do bloco3 para o estado final, e, para tornar o autômato reentrante, atualizará a transição do

estado e0 para e1. A Figura 16 apresenta o autômato após o consumo do primeiro átomo 'b' e a execução da função adaptativa DefinirTransicao. Esta configuração também é a configuração final do autômato, pois não existe nenhuma outra função adaptativa associada às transições dos blocos bloco2 e bloco3.

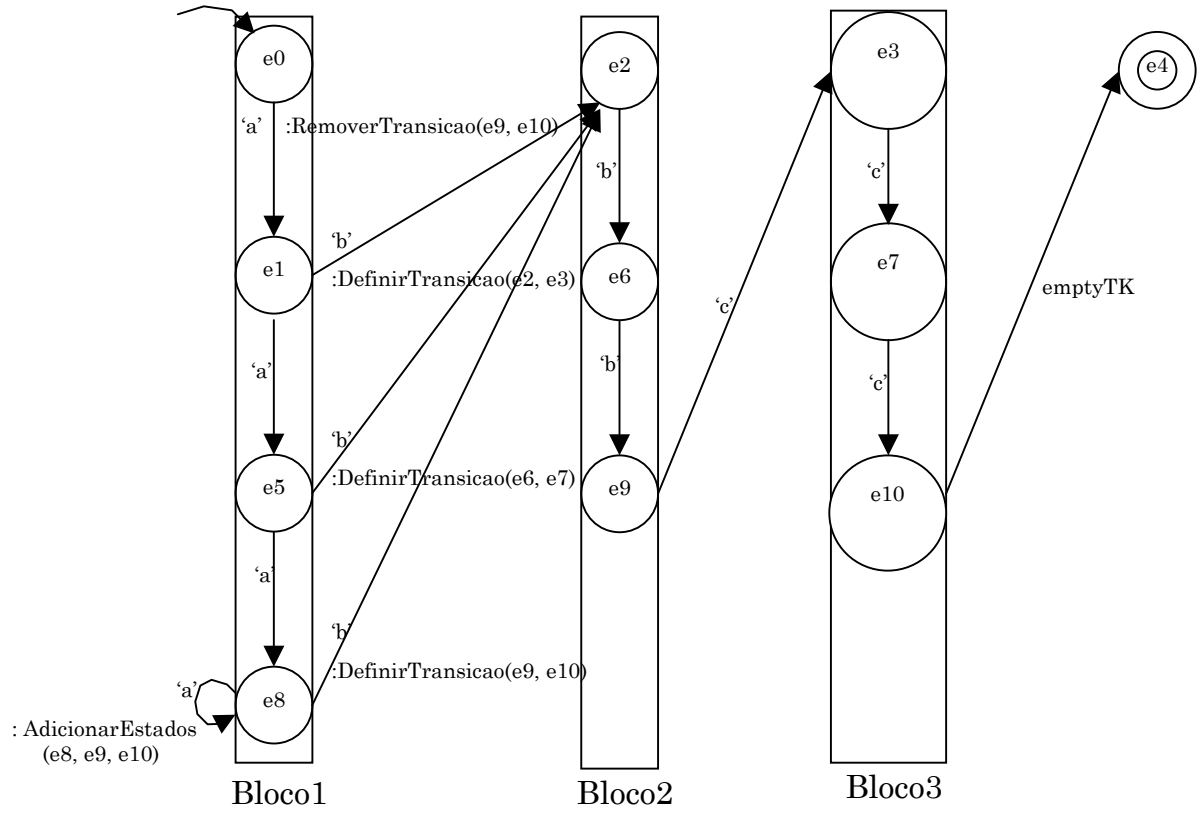

Figura 16 - Autômato final para a expressão "aaabbbccc"

Exemplo 4 – Compilador RSW

O compilador RSW foi desenvolvido para reconhecer a linguagem RSW e realizar a geração do código fonte das descrições encontradas no texto de entrada, na linguagem Pascal. Já vimos, na página 73, o ciclo de implementação do qual resultou a última versão do compilador RSW, até a época da elaboração desta dissertação. A versão 3.00 do compilador RSW apresenta, em sua estrutura, ações adaptativas capazes de coletar novos identificadores, reconhecer identificadores já declarados, associar um átomo de retorno específico ao identificador que represente o seu tipo corretamente, gerenciar a abertura e o fechamento do bloco de definição de identificadores.

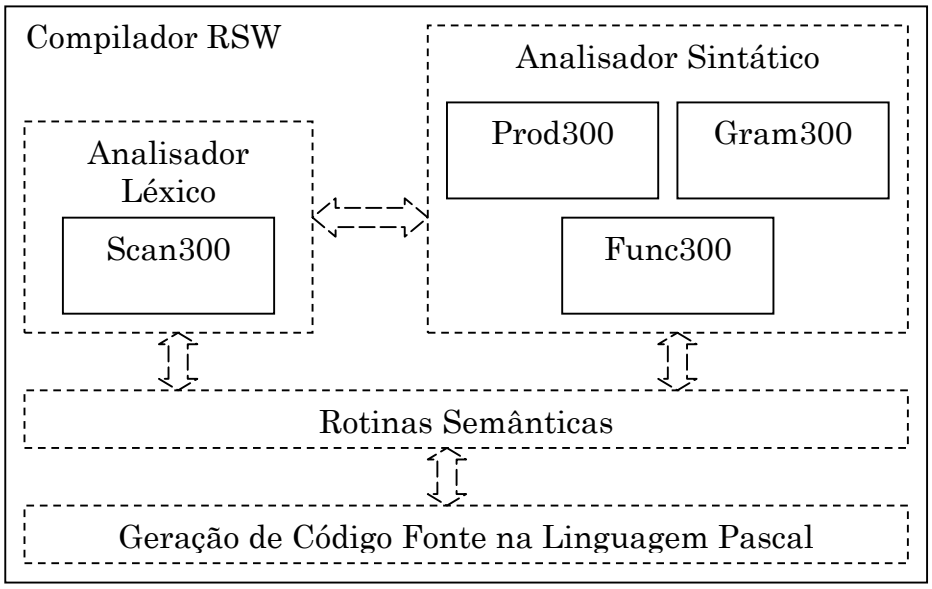

Figura 17 - Arquitetura do compilador RSW

O compilador RSW é estruturado em quatro sub-máquinas. A primeira sub-máquina é denominada Gram300. Ela é a sub-máquina inicial do compilador e é responsável pelo reconhecimento do cabeçalho da sub-máquina, até o início da seção de declaração das produções. As produções propriamente ditas, são reconhecidas pela segunda sub-máquina, denominada Prod300. Para cada produção encontrada, a sub-máquina

Gram300 ativa a sub-máquina Prod300 para realizar o reconhecimento da mesma. A palavra reservada 'funcoes' marca o término da seção de declaração das produções e o início da seção de declaração das funções adaptativas. Para cada função adaptativa encontrada, a sub-máquina Gram300 ativa a terceira sub-máquina, denominada Func300. A sub-máquina Func300 é responsável pelo reconhecimento de uma função adaptativa, isto é, a declaração dos parâmetros, variáveis locais e ações adaptativas elementares. A quarta sub-máquina, denominada Scan300, é ativada sempre que as demais sub-máquinas necessitarem de um átomo de entrada. Apenas a sub-máquina Scan300 manipula os caracteres no formato ASCII, diretamente do texto de entrada. As demais sub-máquinas sempre consomem átomos empilhados pela sub-máquina Scan300.

De acordo com a Figura 17, a sub-máquina Scan300 corresponde ao analisador léxico do compilador RSW, e é formada por um autômato adaptativo. Vamos analisar as funções adaptativas desta sub-máquina, que implementam as seguintes funcionalidades:

- reconhecimento de identificadores já encontrados no texto de entrada
- coleta de novos identificadores
- associação de átomos de reconhecimento que representam o tipo do identificador encontrado
- tratamento do escopo de validade dos identificadores para as funções adaptativas

De forma geral, podemos observar pelo menos três modos de operação possíveis por parte deste analisador léxico adaptativo. O primeiro modo de operação se caracteriza por permitir que novos identificadores sejam coletados. Neste caso, todo identificador deve ser coletado, e caso o mesmo tenha sido encontrado anteriormente, conclui-se que o identificador em questão está sendo redefinido no mesmo escopo. O segundo modo de operação se caracteriza pela verificação dos identificadores anteriormente encontrados. Neste caso, quando se encontra um novo identificador, conclui-se que o mesmo não foi declarado no escopo corrente. O terceiro modo de

operação se caracteriza por permitir a coleta e o reconhecimento simultâneos dos identificadores simultaneamente. Neste caso, jamais é detectado o caso de identificadores não definidos, pois se o mesmo ainda não foi definido, deve-se então coletá-lo. Outro caso jamais detectado é a redefinição de um identificador, pois se o mesmo já foi definido, deve-se então empilhar seu átomo de retorno associado.

A sub-máquina Scan300 apresenta estes três modos de operação. O primeiro modo de operação entra em funcionamento na declaração dos parâmetros e variáveis locais de uma função adaptativa. Os identificadores que representam os parâmetros e as variáveis locais são coletados e armazenados na própria estrutura da sub-máquina Scan300. O segundo modo de operação entra em funcionamento na declaração das ações adaptativas elementares. Neste momento, todos os identificadores encontrados no texto de entrada devem ter sido previamente declarados. O terceiro modo de operação entra em funcionamento durante a análise das produções, quando identificadores representando estados, átomos e sub-máquina são coletados e reconhecidos simultaneamente. A razão de se adotar o terceiro modo de operação para as produções é decisão de projeto da linguagem RSW de não forçar o usuário a declarar o tipo de cada identificador antes de sua utilização. Por isso, para se criar um estado identificado pela cadeia de caracteres 'e0', basta apenas utilizá-lo como estado de origem ou de destino em alguma produção. A Figura 18 apresenta um pequeno programa na linguagem RSW, com as indicações dos três modos de operação da sub-máquina Scan300.

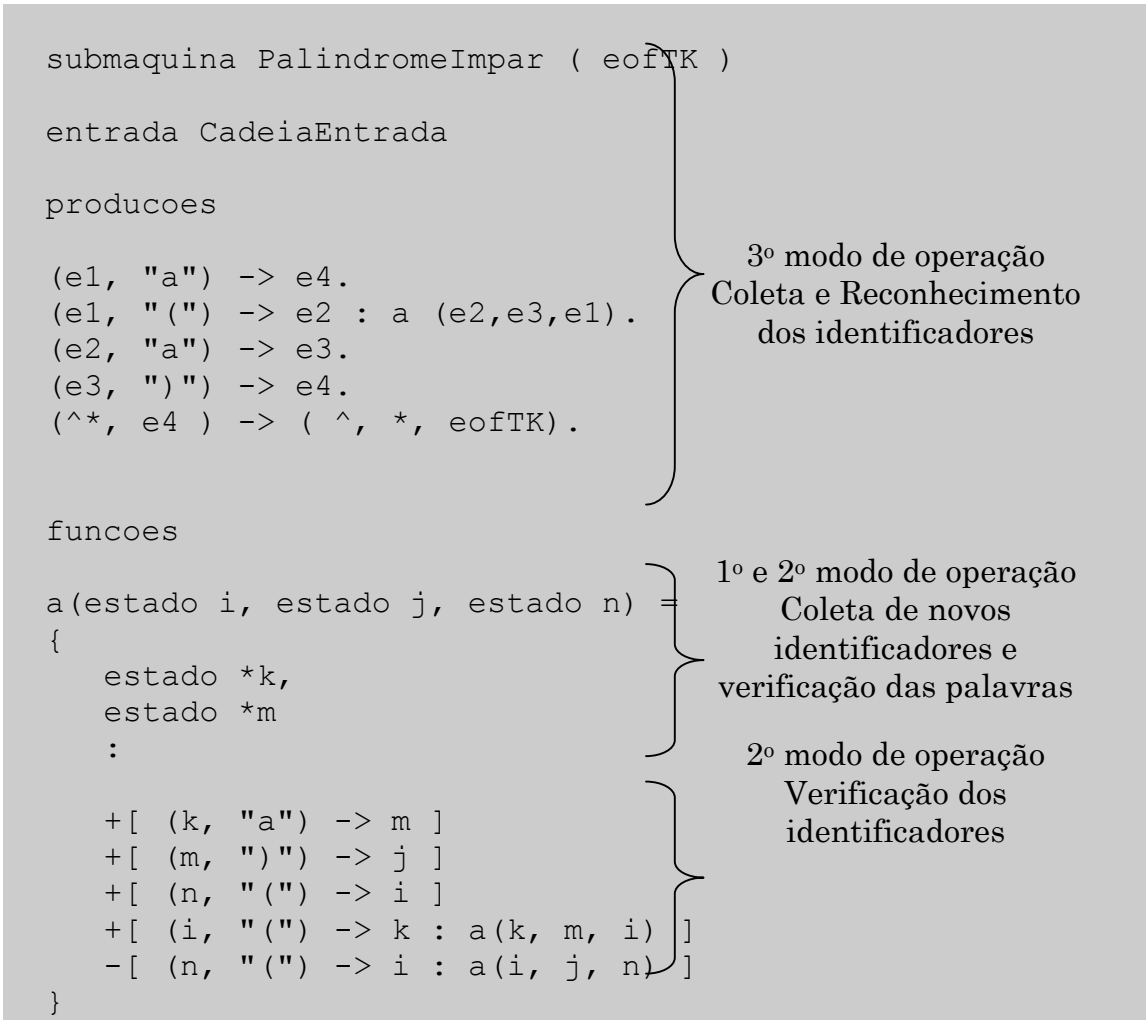

Figura 18 - Regiões de utilização dos modos de operação da sub-máquina Scan300 em um texto RSW

A seguir, apresentaremos apenas alguns trechos dos programas-fonte que descrevem os componentes do compilador RSW, escritos na própria linguagem RSW. No Anexo I – Compilador RSW versão 3.00 na Linguagem RSW – encontra-se a listagem de todos os componentes do compilador RSW. Passaremos, agora, para a análise da configuração inicial da sub-máquina Scan300, seus modos de operação, bem como suas funções adaptativas.

```
1- submaquina Scan300 % Analisador Lexico
```

```
2-
3- (
4-   lparenTK,       %'('
5-   rparenTK,       %')'
6-   commaTK,        %','
7-   dotTK,          %'.'
8-   synTK,          %'^'
9-   timesTK,        %'*'
10-    ddotTK,         %':'
11-    lessTk,         %'<'
12-    greaterTK,      %'>'
13-    arrowTK,        %'->'
14-    automatoTK,     %'submaquina'
15-    inputTK,        %'entrada'
16-    stateTK,        %'estado'
17-    tokenTK,        %'atomo'
18-    productionsTK, %'producoes'
19-    stringTK,       %'"'{qualquer}'"'
20-    idTK,           %(letter+'_') {letter+'_'+digits}
21-    idRedefinedTK, %Redefinicao de identificador no
  mesmo bloco
22-    idAutomatoTK,  %identificador do tipo sub-maquina
23-    idStateTK,     %identificador do tipo estado
24-    idTokenTK,     %identificador do tipo atomo
25-    idFunctionTK,  %identificador do tipo funcao
26-    numTK,          %digits {digits}
27-    eofTK ,         %fim da cadeia de entrada
28-    anyTK ,         %'qualquer'
29-    minusTK,        %'-'
30-    plusTK,         %'+'
31-    functionsTK,   %'funcoes'
32-    lbraceTK,       %'{'
33-    rbraceTK,       %'}'
34-    lbracketTK,     %'['
35-    rbracketTK,     %']'
36-    equalTK,        %'='
37-    interTK         %'?'
38-  )
39-
40-  entrada CadeiaEntrada %Cadeia de Caraceteres primaria
```

Listagem 15 - Sub-máquina Scan300 - Átomos de Retorno

A Listagem 15 apresenta as declarações de todos os possíveis átomos de retorno, empilháveis pela sub-máquina Scan300. Para cada símbolo, palavra reservada ou identificador reconhecido, é empilhado um átomo de retorno distinto. Observem os átomo definidos nas linhas 22 a 25. Tais

átomos de retorno serão utilizados para especificar o tipo do identificador reconhecido, isto é, se o identificador encontrado representa uma sub-máquina, um estado, um átomo, ou ainda uma função adaptativa.

```
1- producoes
2- % Secao de declaracao das producoes
3-
4- % Estado inicial que permite alteracoes estruturais
   antes
5- % da analise da cadeia de entrada
6-   ( e0, emptyTK) -> e1.
7-
8- %Declaracao dos simbolos unitarios e compostos
9-
10-     ( e1, 255) -> (e2, eofTK).
11-     ( e1, "(") -> ( e2, lparenTK).
12-     ( e1, ")") -> ( e2, rparenTK).
13-     ( e1, ",") -> ( e2, commaTK).
14-     ( e1, ".") -> ( e2, dotTK).
15-     ( e1, "^") -> ( e2, synTK).
16-     ( e1, "*") -> ( e2, timesTK).
17-     ( e1, ":") -> ( e2, ddotTK).
18-     ( e1, "<") -> ( e2, lessTK).
19-     ( e1, ">") -> ( e2, greaterTK).
20-     ( e1, "{") -> ( e2, lbraceTK).
21-     ( e1, "}") -> ( e2, rbraceTK).
22-     ( e1, "[") -> ( e2, lbracketTK).
23-     ( e1, "]") -> ( e2, rbracketTK).
24-     ( e1, "=") -> ( e2, equalTK).
25-     ( e1, "+") -> ( e2, plusTK).
26-     ( e1, "?") -> ( e2, interTK).
27-     ( e1, "-") ->  e3 .
28-     ( e3, ">") -> ( e2, arrowTK).
29-     ( e3, emptyTK) -> ( e2, minusTK).
30-
31-     ( ^*, e2 ) -> ( ^, * , emptyTK).
```

Listagem 16 - Sub-máquina Scan300 - Símbolos simples

A Listagem 16 apresenta as produções responsáveis pelo reconhecimento dos símbolos mais simples, formados por um ou dois caracteres. A linha 10 apresenta a produção que identifica o término da cadeia de entrada. O átomo consumido nesta produção é o caracter inteiro 255, pois a sub-

máquina CadeiaEntrada utiliza tal caracter para sinalizar o término do texto de entrada.

```
1- % Declaracao das Palavras Reservadas
2-
3- %  ( e1, "submaquina") -> e4.
4-
5- %  ( e1, "entrada")    -> e5.
6-
7- %  ( e1, "estado")     -> e18.
8-
9- %  ( e1, "producoes" ) -> e6.
10-
11-  %  ( e1, "funcoes" )   -> e17.
12-
13-  %  ( e1, "qualquer" )  -> e15.
14-
15-  %  ( e1, "atomo" )    ->  e19.
16-
17-  % Declaracao de strings
18-
19-    ( e1, '"') -> e7. %Inicio da String
20-    ( e7, qualquer - '"' ) -> e7.
21-    ( e7, '"') -> e8. %Termino da String
22-    ( ^*, e8 ) -> ( ^, *, stringTK ).
23-
24-    ( e1, "'") -> e11. %Inicio da String
25-    ( e11, qualquer - "'") ->e11.
26-    ( e11, "'") -> e12. %termino da String
27-    ( ^*, e12) -> ( ^, *, stringTK ).
```

Listagem 17 - Sub-máquina Scan300 - Palavras reservadas e *strings*

A Listagem 17 mostra as produções que reconhecem as cadeias de caracteres delimitadas por aspas duplas ou simples, freqüentemente denominadas *strings*. As produções que reconhecem as palavras reservadas foram abreviadas, em função do espaço que ocuparia tal listagem.

```
1- % Declaracao dos numeros inteiros
2-
```

```
3-   ( e1, "0" ) -> e14.
4-   ( e1, "1" ) -> e14.
5-   ( e1, "2" ) -> e14.
6-   ( e1, "3" ) -> e14.
7-   ( e1, "4" ) -> e14.
8-   ( e1, "5" ) -> e14.
9-   ( e1, "6" ) -> e14.
10-    ( e1, "7" ) -> e14.
11-    ( e1, "8" ) -> e14.
12-    ( e1, "9" ) -> e14.
13-
14-    ( e14, "0" ) -> e14.
15-    ( e14, "1" ) -> e14.
16-    ( e14, "2" ) -> e14.
17-    ( e14, "3" ) -> e14.
18-    ( e14, "4" ) -> e14.
19-    ( e14, "5" ) -> e14.
20-    ( e14, "6" ) -> e14.
21-    ( e14, "7" ) -> e14.
22-    ( e14, "8" ) -> e14.
23-    ( e14, "9" ) -> e14.
24-
25-  ( ^*, e14 ) -> ( ^, *, numTK). %Reconhece um numero
  inteiro
26-  % Declaracao dos comentarios
27-
28-  ( e1, "%" ) -> e10.
29-  ( e10, qualquer - 13) -> e10.
30-  ( e10, 13) -> e1.
31-
32-  % Caracteres ignorados
33-
34-  ( e1, 10) -> e1. %Line Feed
35-  ( e1, 13) -> e1. %Carriage Return
36-  ( e1, 32) -> e1. %Espaco
37-  ( e1, 09) -> e1. %Tab
```

Listagem 18 - Sub-máquina Scan300 - Números inteiros, comentários e delimitadores

A Listagem 18 apresenta as produções responsáveis pelo reconhecimento de números inteiros, por desprezar os comentários e os caracteres de preenchimento, tais como, espaços em branco, caracter de tabulação, avanço e retorno de linha.

É importante notar que o estado e1 é o estado inicial para o reconhecimento de todos os átomos até aqui analisados. A partir do estado e1 são reconhecidos os símbolos simples, tais como, '=', '{', entre outros, as palavras reservadas, as cadeias de caracteres, os números inteiros, e também o descarte dos comentários e caracteres de preenchimento ou delimitadores. Como veremos mais à frente, os identificadores do texto de entrada também são reconhecidos a partir do estado e1.

```
1- % Indica se a maquina esta no modo de verificacao
2-   ( Verificacao, emptyTK) -> Verificacao.
3-
4- % Indica se a maquina esta no modo de coleta simples
5-   ( ColetaSimples, emptyTK) -> ColetaSimples.
6-
7- % Indica o estado que empilhou o atomo de retorno
8- % antes do desvio para a maquina chamadora.
9-   ( EstadoRetorno, emptyTK) -> erro.
```

Listagem 19 - Sub-máquina Scan300 - Estados especiais

Os autômatos adaptativos disponibilizam alguns mecanismos para o armazenamento de informações em sua estrutura. A sub-máquina Scan300 utiliza os estados e transições da Listagem 19 para armazenar informações sobre o modo de operação do analisador léxico e o estado que empilhou o último átomo idTK, ou seja, um identificador sem tipo definido.

Tendo em vista o modo de operação do analisador léxico, o mecanismo utilizado consiste em verificar a existência, ou não, de uma transição em vazio de um estado para ele mesmo. A Tabela 3 apresenta o relacionamento entre os estados e suas transições, e os respectivos modos de operação do analisador léxico.

Estado	1º modo de	2º modo de operação	3º modo de operação

	operação Coleta	Verificação	Coleta e Verificação
Verificacao	Sem transição	Transição em vazio	Transição em vazio
ColetaSimples	Transição em vazio	Sem transição	Transição em vazio

Tabela 3 - Equivalência entre estados e modos de operação

O estado que empilhou o último átomo, representado um identificador sem tipo associado, é obtido através de uma ação elementar de consulta onde o estado de origem é o estado EstadoRetorno, o átomo de entrada é vazio (emptyTK) e o estado de destino é o estado pesquisado. Em outras palavras, quando o estado EstadoRetorno apresentar uma transição em vazio (emptyTK) para algum estado, este será o estado que empilhou o último átomo idTK.

```
1-   ( e1, "_" ) : CriarTransicao(e1,"_") -> erro.
2- % ( e1, "a" ) -> erro. Tratado na palavra reservada
  atomo
3-   ( e1, "b" ) : CriarTransicao(e1,"b") -> erro.
4-   ( e1, "c" ) : CriarTransicao(e1,"c") -> erro.
5-   ( e1, "d" ) : CriarTransicao(e1,"d") -> erro.
6- % ( e1, "e" ) -> erro. Tratado na palavra reservada
  entrada/estado
7- % ( e1, "f" ) -> erro. Tratado na palavra reservada
  funcoes
8-   ( e1, "g" ) : CriarTransicao(e1,"g") -> erro.
9-
10-  ...
11-
12-    ( e1, "o" ) : CriarTransicao(e1,"o") -> erro.
13-  % ( e1, "p" ) -> erro. Tratado na palavra reservada
  producoes
14-  % ( e1, "q" ) -> erro. Tratado na palavra reservada
  qualquer
15-    ( e1, "r" ) : CriarTransicao(e1,"r") -> erro.
16-  % ( e1, "s" ) -> erro. Tratado na palavra reservada
  submaquina
17-    ( e1, "t" ) : CriarTransicao(e1,"t") -> erro.
18-    ( e1, "u" ) : CriarTransicao(e1,"u") -> erro.
19-    ( e1, "w" ) : CriarTransicao(e1,"w") -> erro.
20-    ( e1, "v" ) : CriarTransicao(e1,"v") -> erro.
21-    ( e1, "x" ) : CriarTransicao(e1,"x") -> erro.
```

```
22-    ( e1, "y" ) : CriarTransicao(e1,"y") -> erro.
23-    ( e1, "z" ) : CriarTransicao(e1,"z") -> erro.
24-    ( e1, "A" ) : CriarTransicao(e1,"A") -> erro.
25-
26-  ...
27-
28-    ( e1, "X" ) : CriarTransicao(e1,"X") -> erro.
29-    ( e1, "Z" ) : CriarTransicao(e1,"Z") -> erro.
```

Listagem 20 - Sub-máquina Scan300 - Identificadores

A Listagem 20 apresenta, resumidamente, as produções que dão início ao reconhecimento dos identificadores, a partir do estado e1. Os identificadores são formados por uma seqüência de letras, dígitos e o caracter '_', iniciada necessariamente por uma letra ou pelo caracter '_'. Os átomos de entrada das produções nas linhas 2, 6, 7, 13, 14, e 16, correspondem às primeiras letras das palavras reservadas da linguagem RSW. Por esta razão, estas produções foram comentadas, já que o reconhecimento das palavras reservadas apresentam produções com estes mesmos átomos de entrada, a partir do estado e1.

A função adaptativa CriarTransicao, apresentada na Listagem 21, é responsável por expandir a estrutura da sub-máquina Scan300 conforme os caracteres dos identificadores vão sendo consumidos. Observe-se que esta expansão só será realizada caso o modo de operação admita a coleta dos identificadores. No caso de o modo de operação não admitir tal coleta, a transição que ativou a função adaptativa define o estado erro como estado-destino. Esta transição é realizada, e conseqüentemente um erro é detectado.

```
1- CriarTransicao ( estado EstadoCorrente, atomo
   AtomoCorrente) =
2- {
3-   estado *NovoEstado,
4-   atomo  Coletando
5-   :
6-
7-   ?[(ColetaSimples, Coletando) -> ColetaSimples]
8-
9-   -[( EstadoCorrente, AtomoCorrente) :
```

```
10-
  CriarTransicao(EstadoCorrente,AtomoCorrente) ->
11-                                        (erro, Coletando)
  ]
12-   +[( EstadoCorrente, AtomoCorrente) -> (NovoEstado,
  Coletando) ]
13-
14-   +[( NovoEstado, "_" ) :
  CriarTransicao(NovoEstado,"_") ->
15-                                                 (erro,
  Coletando) ]
16-    +[( NovoEstado, "a" ) :
  CriarTransicao(NovoEstado,"a") ->
17-                                                 (erro,
  Coletando) ]
18-  ...
19-
20-    +[( NovoEstado, "z" ) :
  CriarTransicao(NovoEstado,"z") ->
21-                                                 (erro,
  Coletando) ]
22-    +[( NovoEstado, "0" ) :
  CriarTransicao(NovoEstado,"1") ->
23-                                                 (erro,
  Coletando) ]
24-  ...
25-
26-    +[( NovoEstado, "9" ) :
  CriarTransicao(NovoEstado,"0") ->
27-                                                 (erro,
  Coletando) ]
28-    +[( NovoEstado, "A" ) :
  CriarTransicao(NovoEstado,"A") ->
29-                                                 (erro,
  Coletando) ]
30-  ...
31-
32-   +[( NovoEstado, "Z" ) :
  CriarTransicao(NovoEstado,"Z") ->
33-                                                 (erro,
  Coletando) ]
34-   +[( NovoEstado, emptyTK) :
  DefinirIdDesconhecido(NovoEstado) ->
35-                                                 (erro,
  Coletando) ]
36-  }
```

Listagem 21 - Sub-máquina Scan300 - CriarTransicao( )

Após o consumo do último caracter do identificador, a transição em vazio do estado corrente ativará a função adaptativa DefinirIdDesconhecido, apresentada na Listagem 22.

```
1- DefinirIdDesconhecido(estado EstadoCorrente) =
2- {
3-   estado *NovoEstado,
4-   atomo  Coletando
5-   :
6-
7-   ?[(ColetaSimples, Coletando) -> ColetaSimples]
8-   -[(EstadoCorrente, emptyTK) :
9-        DefinirIdDesconhecido(EstadoCorrente) ->
10-                                  (EstadoCorrente,
  Coletando)]
11-    +[(EstadoCorrente, emptyTK) -> (NovoEstado,
  Coletando)]
12-    +[(^*, NovoEstado, Coletando) -> (^, *, idTK) :
13-                           DefinirEstadoRetorno
  (NovoEstado)]
14-  }
```

Listagem 22 - Sub-máquina Scan300 - DefinirIdDesconhecido( )

Caso o modo de operação admita a coleta de identificadores, será adicionada uma transição em vazio a partir do estado corrente para um novo estado que empilhará o átomo idTK, indicando o reconhecimento de um identificador sem tipo associado. Caso o modo de operação seja apenas de verificação, a transição executada resultará numa condição de erro. Antes de empilhar o átomo idTK, o novo estado adicionado pela função DefinirIdDesconhecido, ativa a função DefinirEstadoRetorno, apresentada na Listagem 23, responsável por atualizar a transição em vazio do estado EstadoRetorno.

```
1- DefinirEstadoRetorno( estado NovoEstado) =
2- {
3-   estado EstadoAnterior:
4-
5-   -[(EstadoRetorno, emptyTK)  -> EstadoAnterior]
6-   +[(EstadoRetorno, emptyTK)  -> NovoEstado]
7- }
```

Listagem 23 - Sub-máquina Scan300 - DefinirEstadoRetorno( )

Somente as funções adaptativas CriarTransicao, DefinirIdDesconhecido e DefinirEstadoRetorno, são acionadas durante o primeiro bloco de declaração, ou seja, até o início do reconhecimento das funções adaptativas. O primeiro bloco de declarações utiliza o terceiro modo de operação do analisador léxico, conforme ilustrado na Figura 18, página 96, permitindo a coleta e verificação dos identificadores simultaneamente.

```
1- AbrirBloco =
2- {
3-   estado *NovoEstado1,
4-   estado *NovoEstado2
5-   :
6-
7-   -[(e0, emptyTK) -> e1]
8-   +[(e0, emptyTK) -> NovoEstado1]
9-   +[(NovoEstado1, emptyTK) -> NovoEstado2]
10-    +[(NovoEstado2, emptyTK) -> e1)]
11-
12-    +[(NovoEstado1, "_") :
13-       CriarTransicaoBloco(NovoEstado1, NovoEstado2,
   "_") ->
14-
   NovoEstado2]
15-    +[(NovoEstado1, "a") :
16-       CriarTransicaoBloco(Novoestado1, NovoEstado2,
   "a") ->
17-
   NovoEstado2]
18-  ...
19-
20-    +[(NovoEstado1, "z") :
21-       CriarTransicaoBloco(Novoestado1, NovoEstado2,
   "z") ->
22-
   NovoEstado2]
23-    +[(NovoEstado1, "A") :
24-       CriarTransicaoBloco(Novoestado1, NovoEstado2,
   "A") ->
25-
   NovoEstado2]
26-  ...
27-
28-    +[(NovoEstado1, "Z") :
29-       CriarTransicaoBloco(Novoestado1, NovoEstado2,
   "Z") ->
```

```
30-
  NovoEstado2]
31-  }
```

Listagem 24 - Sub-máquina Scan300 - AbrirBloco( )

Os identificadores que representam os parâmetros e as variáveis locais das funções adaptativas devem ser coletados para posterior verificação, durante a declaração das ações adaptativas elementares. Estes identificadores não são válidos fora do escopo da função adaptativa em que foram declarados. A declaração dos parâmetros da função adaptativa marca o início deste escopo, e o fechamento do bloco de declaração das ações adaptativas elementares pelo símbolo '}', marca o fim deste escopo. A função adaptativa AbrirBloco, apresentada na Listagem 24, é ativada ao início do escopo descrito. Ela adiciona, entre os estados e0 e e1, dois novos estados, responsáveis por coletar os identificadores válidos apenas para o referido escopo. A partir destes dois novos estados, serão armazenados os novos identificadores, através da expansão da estrutura entre eles. O funcionamento desta estrutura entre os estados e0 e e1 é descrito mais à frente. A função adaptativa FecharBloco, apresentada na Listagem 25, é ativada no final do escopo descrito. Ela elimina as transições que permitem o acesso à estrutura contendo os identificadores válidos para o escopo descrito. Em termos de implementação, é de responsabilidade do ambiente RSW detectar e remover da memória os estados que não estão mais em uso.

```
1- FecharBloco =
2- {
3-   estado NovoEstado1,
4-   estado NovoEstado2
5-   :
6-
7-   -[(e0, emptyTK) -> NovoEstado1]
8-   -[(NovoEstado1, emptyTK) -> NovoEstado2]
9-   -[(NovoEstado2, emptyTK) -> e1]
10-    +[(e0, emptyTK) -> e1]
11-  }
```

Listagem 25 - Sub-máquina Scan300 - FecharBloco( )

A função adaptativa CriarTransicaoBloco, apresentada na Listagem 26, realiza a expansão na estrutura de estados e transições localizados entre os estados e0 e e1, com a finalidade de armazenar os identificadores encontrados. Durante a operação de coleta, cada caracter consumido adiciona dois novos estados. O primeiro novo estado objetiva o reconhecimento do identificador para uma segunda análise. O segundo novo estado objetiva a recomposição dos átomos consumidos durante uma tentativa fracassada de reconhecimento de um identificador armazenado entre os estados e0 e e1. A recomposição da cadeia de entrada é realizada para serem verificados os identificadores armazenados a partir do estado e1. Portanto ao invés de determinar uma situação de erro quando o reconhecimento de um identificador fracassar, os caracteres consumidos pelas transições entre os estados e0 e e1 são empilhados novamente na cadeia de entrada para serem posteriormente analisados. Apesar de a linguagem RSW apresentar apenas dois escopos, esta mesma técnica pode ser adotada no processamento de linguagens que permitam um número arbitrário de blocos aninhados.

```
1- CriarTransicaoBloco ( estado EstadoCorrente, estado
   ProximoEstado,
2-                       atomo AtomoCorrente) =
3- {
4-   estado *NovoEstado1,  estado *NovoEstado2,
5-   atomo  Coletando   :
6-
7-   ?[( ColetaSimples, Coletando) -> ColetaSimples]
8-   -[( EstadoCorrente, AtomoCorrente) :
9-        CriarTransicaoBloco(EstadoCorrente,
10-      ProximoEstado, AtomoCorrente) -> (ProximoEstado,
   Coletando) ]
11-     +[( EstadoCorrente, AtomoCorrente) -> (NovoEstado1,
   Coletando]
12-     +[( NovoEstado1, Coletando) :
   DefinirIdDesconhecidoBloco
13-                        (NovoEstado1,NovoEstado2) ->
   NovoEstado2]
14-     +[( NovoEstado2, Coletando) -> (ProximoEstado,
   AtomoCorrente)]
15-
```

```
16-    +[( NovoEstado1, "_" ) :
17-       CriarTransicaoBloco(NovoEstado1, NovoEstado2,
  "_") ->
18-                                          (NovoEstado2,
  Coletando) ]
19-    +[( NovoEstado1, "a" ) :
20-        CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"a") ->
21-                                           NovoEstado2,
  Coletando) ]
22-  ...
23-
24-    +[( NovoEstado1, "z" ) :
25-          CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"z") ->
26-                                          (NovoEstado2,
  Coletando) ]
27-    +[( NovoEstado1, "0" ) :
28-          CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"1") ->
29-                                          (NovoEstado2,
  Coletando) ]
30-  ...
31-
32-    +[( NovoEstado1, "9" ) :
33-         CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"9") ->
34-                                          (NovoEstado2,
  Coletando) ]
35-  +[( NovoEstado1, "A" ) :
36-         CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"A") ->
37-                                          (NovoEstado2,
  Coletando) ]
38-  ...
39-
40-    +[( NovoEstado1, "Z" ) :
41-         CriarTransicaoBloco(NovoEstado1,
  NovoEstado2,"Z") ->
42-                                          (NovoEstado2,
  Coletando) ]
43-    +[( NovoEstado1, emptyTK) :
44-      DefinirIdDesconhecidoBloco(NovoEstado1,
  NovoEstado2) ->
45-                                          (NovoEstado2,
  Coletando) ]
46-  }
```

Listagem 26 - Sub-máquina Scan300 - CriarTransicaoBloco( )

```
1- DefinirIdDesconhecidoBloco(estado EstadoCorrente,
2-                             estado ProximoEstado) =
3- {
4-   estado *NovoEstado,
5-   atomo  Coletando
6-   :
7-
8-   ?[(ColetaSimples, Coletando) -> ColetaSimples]
9-   -[(EstadoCorrente, emptyTK) :
10-
   DefinirIdDesconhecidoBloco(EstadoCorrente,ProximoEstado)
   ->
11-                                        (ProximoEstado,
   Coletando) ]
12-     +[(EstadoCorrente, emptyTK) :
13-        DefinirIdRedefinido(EstadoCorrente, NovoEstado) -
   >
14-                                           (NovoEstado,
   Coletando)]
15-     +[(^*, NovoEstado, Coletando) -> (^, *, idTK) :
16-                                  DefinirEstadoRetorno
   (NovoEstado)]
17-
18-  }
```

Listagem 27 - Sub-máquina Scan300 - DefinirIdDesconhecidoBloco()

As funções adaptativas CriarTransicaoBloco e DefinirIdDesconhecidoBloco, apresentadas, respectivamente, na Listagem 26 e Listagem 27, executam atividades semelhantes às funções adaptativas CriarTransicao e DefinirIdDesconhecido.

Como funcionalidades adicionais, temos a recomposição da cadeia de entrada na ocasião de um reconhecimento fracassado, e também a detecção da tentativa de coletar o mesmo identificador mais de uma vez. Esta situação ocorre quando, por exemplo, o usuário declarar dois estados com o mesmo nome, ou um estado e um átomo com o mesmo nome. Por esta razão, antes de empilhar um átomo de retorno pela segunda vez, a função adaptativa DefinirIdRedefinido, apresentada na Listagem 28, é ativada para a devida verificação. Se o modo de operação corrente permitir a coleta de identificadores, a função DefinirIdRedefinido altera o átomo a ser empilhado para idRedefinedTK, representando a detecção do erro. Caso contrário, ela

elimina a transição que a ativou para que não seja realizada novamente tal verificação.

```
1- DefinirIdRedefinido(estado EstadoCorrente, estado
   NovoEstado) =
2- {
3-   estado *NovoEstado1,
4-   atomo  Coletando,
5-   atomo  Verificando
6-   :
7-
8-   ?[(ColetaSimples, Coletando) -> ColetaSimples]
9-   ?[(Verificacao,   Verificando) -> Verificacao]
10-
11-    -[(EstadoCorrente, emptyTK) :
   DefinirIdRedefinido(EstadoCorrente) -> NovoEstado]
12-
13-    +[(EstadoCorrente, Verificando) -> NovoEstado]
14-
15-    +[(EstadoCorrente, Coletando) -> NovoEstado1]
16-    +[(^*, NovoEstado1, Coletando) -> (^, *,
   idRedefinedTK]
17-  }
```

Listagem 28 - Sub-máquina Scan300 - DefinirIdRedefinido( )

As funções adaptativas DefinirModoColeta e DefinirModoVerificacao, apresentadas na Listagem 29, atualizam as transições em vazio dos estados Verificacao e ColetaSimples, conforme o modo de operação desejado para o analisador léxico. Como já vimos, a Tabela 3, página 102, ilustra a correspondência entre a existência das transições em vazio e os modos de operação do analisador léxico.

```
1- DefinirModoColeta =
2- {
3-   -[( Verificacao,   emptyTK) -> Verificacao]
4-   +[( ColetaSimples, emptyTK) -> ColetaSimples]
5- }
6-
7- DefinirModoVerificacao =
8- {
9-   +[( Verificacao,   emptyTK) -> Verificacao]
10-    -[( ColetaSimples, emptyTK) -> ColetaSimples]
11-  }
```

Listagem 29 - Sub-máquina Scan300 - DefinirModoColeta( ) e DefinirModoVerificacao( )

```
1- DefinirIdAtomo =
2- {
3-    estado UltimoEstado :
4-
5-    -[(EstadoRetorno,emptyTK) -> UltimoEstado]
6-    -[(^*, UltimoEstado) -> (^, *, idTK)]
7-    +[(^*, UltimoEstado) -> (^, *, idTokenTK)]
8- }
9-
10-  DefinirIdFuncao =
11-  {
12-     estado UltimoEstado :
13-
14-     -[(EstadoRetorno,emptyTK) -> UltimoEstado]
15-     -[(^*, UltimoEstado) -> (^, *, idTK)]
16-     +[(^*, UltimoEstado) -> (^, *, idFunctionTK)]
17-  }
18-
19-  DefinirIdEstado =
20-  {
21-     estado UltimoEstado :
22-
23-     -[(EstadoRetorno,emptyTK) -> UltimoEstado]
24-     -[(^*, UltimoEstado) -> (^, *, idTK)]
25-     +[(^*, UltimoEstado) -> (^, *, idStateTK)]
26-  }
27-
28-  DefinirIdSubMaquina =
29-  {
30-     estado UltimoEstado :
31-
32-     -[(EstadoRetorno,emptyTK) -> UltimoEstado]
33-     -[(^*, UltimoEstado) -> (^, *, idTK)]
34-     +[(^*, UltimoEstado) -> (^, *, idAutomatoTK)]
35-  }
```

Listagem 30 - Sub-máquina Scan300 - DefinirIdSubMaquina DefinirIdAtomo, DefinirIdFuncao, e DefinirIdEstado

Finalizando a análise da sub-máquina Scan300, temos as funções adaptativas apresentadas na Listagem 30. Estas são responsáveis por

atualizar o átomo de retorno do último identificador encontrado. O novo átomo de retorno especifica o tipo associado ao identificador em questão, podendo ele ser do tipo sub-máquina, estado, átomo ou função.

Estas funções são ativadas a partir das sub-máquinas Gram300, Func300 e Prod300. A Listagem 31 exemplifica o uso destas funções adaptativas. Observe o momento apropriado para a ativação das funções adaptativas de definição do modo de operação da sub-máquina Scan300 nas linhas 4, 7 a 9 e 21 a 23.

```
1- ( e1, idTK) -> e2 : DefinirIdFuncao
   <DefinirFuncaoCorrente> .
2- ( e1, idFunctionTK) -> e2 : AbrirBloco
   <DefinirFuncaoCorrente> .
3-
4- ( e2, lparenTK) -> e3 : DefinirModoVerificacao.
5- ( e2, equalTK)  -> e8.
6-
7- ( e3, AutomatoTK) -> e3_1 : DefinirModoColeta.
8- ( e3, StateTK)    -> e3_2 : DefinirModoColeta.
9- ( e3, TokenTK)    -> e3_3 : DefinirModoColeta.
10-
11-  ( e3_1, idTK) : DefinirModoVerificacao -> e4 :
12-           DefinirIdSubMaquina
   <AdicionarParametroSubMaquinaFuncao>.
13-  ( e3_2, idTK) : DefinirModoVerificacao -> e4 :
14-               DefinirIdEstado
   <AdicionarParametroEstadoFuncao>.
15-  ( e3_3, idTK) : DefinirModoVerificacao -> e4 :
16-                DefinirIdAtomo
   <AdicionarParametroAtomoFuncao>.
17-
18-  ( e4, commaTK) -> e5.
19-  ( e4, rparenTK) -> e7.
20-
21-  ( e5, AutomatoTK) -> e5_1 : DefinirModoColeta.
22-  ( e5, StateTK)    -> e5_2 : DefinirModoColeta.
23-  ( e5, TokenTK)    -> e5_3 : DefinirModoColeta.
24-
25-  ( e5_1, idTK) : DefinirModoVerificacao -> e6 :
26-          DefinirIdSubMaquina
   <AdicionarParametroSubMaquinaFuncao>.
27-  ( e5_2, idTK) : DefinirModoVerificacao -> e6 :
```

```
28-              DefinirIdEstado
  <AdicionarParametroEstadoFuncao>.
29-  ( e5_3, idTK) : DefinirModoVerificacao -> e6 :
30-               DefinirIdAtomo
  <AdicionarParametroAtomoFuncao>.
31-
32-  ( e6, commaTK) -> e5.
33-  ( e6, rparenTK) -> e7.
```

Listagem 31 - Produções da sub-máquina Func300

Exemplo 5 – Tradutor da notação Wirth para linguagem RSW

O algoritmo utilizado no tradutor da notação Wirth para a linguagem RSW foi inicialmente apresentado em [29]. O tradutor é formado por duas sub-máquinas apresentadas a seguir. A Listagem 32 descreve a sub-máquina Gerador responsável por reconhecer uma expressão no seguinte formato:

<identificador> ::= <sentença na notação Wirth> .

```
1- submaquina Gerador ( GeradorTK )
2-
3-
4- entrada Scan
5-
6- producoes
7-
8- (e1, idTK) -> e2.
9-
10-  %Empilha dois novos estado. Define como estado
   corrente
11-  %o estado no topo da pilha
12-  (e2, equalTK) -> e3 <InicializarPilha>.
13-
14-  (e3, emptyTK) -> (^e3, Wirth, emptyTK).
15-  (e3, WirthTK ) -> e4.
16-
17-  %Criar transicao em vazio do estado corrente para
18-  %topo-1.  Desempilhar o ultimo par de estado da pilha
19-  (e4, dotTK) -> e5 <FinalizarAnalise>.
20-
21-  (e5, idTK) -> e2.
22-
23-  (^*, e5) -> (^, *, GeradorTK).
```

Listagem 32 - Máquina principal do tradutor Wirth-RSW

A segunda sub-máquina, denominada Wirth, reconhece a parte da expressão que diz respeito às construções da notação Wirth. A Listagem 33 apresenta a descrição completa da sub-máquina Wirth. Os comentários antes de cada produção explicam em detalhes as atividades que as rotinas semânticas devem realizar para se obter as produções de saída.

```
1- submaquina Wirth ( WirthTK )
2-
3- entrada Scan
4-
5- producoes
6-
7- %Criar transicao do estado corrente para novo estado
8- %consumindo o simbolo expresso por stringTK.  Definir o
9- %novo estado como estado corrente.
10-  (e1, stringTK)   -> e8  <ConsumirTerminal>.
11-
12-  %Criar transicao do estado corrente para o estado
   topo-1 da
13-  %pilha, consumindo o simbolo expresso por idTK.
   Definir o
14-  %novo estado como estado corrente.
15-  (e1, idTK)       -> e8  <ConsumirNaoTerminal>.
16-
17-  %Empilhar o estado corrente e novo estado
18-  (e1, lparenTK)   -> e2  <ConsumirAberturaParenteses>.
19-
20-  %Empilhar o estado corrente e novo estado, e
21-  %Criar transicao em vazio do estado corrente para o
   novo estado
22-  (e1, lbracketTK) -> e3  <ConsumirAberturaColchetes>.
23-
24-  %Empilhar novo estado 2 vezes e
25-  %criar transição em vazio do estado corrente
26-  %para o novo estavo empilhado
27-  (e1, lbraceTK)   -> e4  <ConsumirAberturaChaves>.
28-
29-  (e2, emptyTK) -> (^e2, Wirth, emptyTK).
30-  (e2, WirthTK ) -> e5.
31-
32-  (e3, emptyTK) -> (^e3, Wirth, emptyTK).
33-  (e3, WirthTK ) -> e6.
34-
35-  (e4, emptyTK) -> (^e4, Wirth, emptyTK).
36-  (e4, WirthTK ) -> e7.
37-
38-  %Criar transicao em vazio do estado corrente para o
39-  %estado em topo-1.  Desempilhar um par de estados.
40-  (e5, rparenTK)   -> e8  <FechamentoNivel>.
41-  (e6, rbracketTK) -> e8  <FechamentoNivel>.
42-  (e7, rbraceTK)   -> e8  <FechamentoNivel>.
43-
44-  %Criar transicao em vazio do estado corrente para
   topo-1 da
```

```
45-  %pilha.  Definir o estado corrente como estado do topo
  da pilha
46-  (e8, synTK)      -> e1  <ConsumirAtomoOU>.
47-
48-  (e8, idTK)       -> e8  <ConsumirNaoTerminal>.
49-  (e8, stringTK)   -> e8  <ConsumirTerminal>.
50-  (e8, lparenTK)   -> e2  <ConsumirAberturaParenteses>.
51-  (e8, lbracketTK) -> e3  <ConsumirAberturaColchetes>.
52-  (e8, lbraceTK)   -> e4  <ConsumirAberturaChaves>.
53-  (^*, e8)         -> (^, *, WirthTK).
```

Listagem 33 - Máquina auxiliar do tradutor Wirth-RSW

Podemos observar que o algoritmo implementado nas rotinas semânticas poderia ser facilmente implementada através de ações e funções adaptativas.

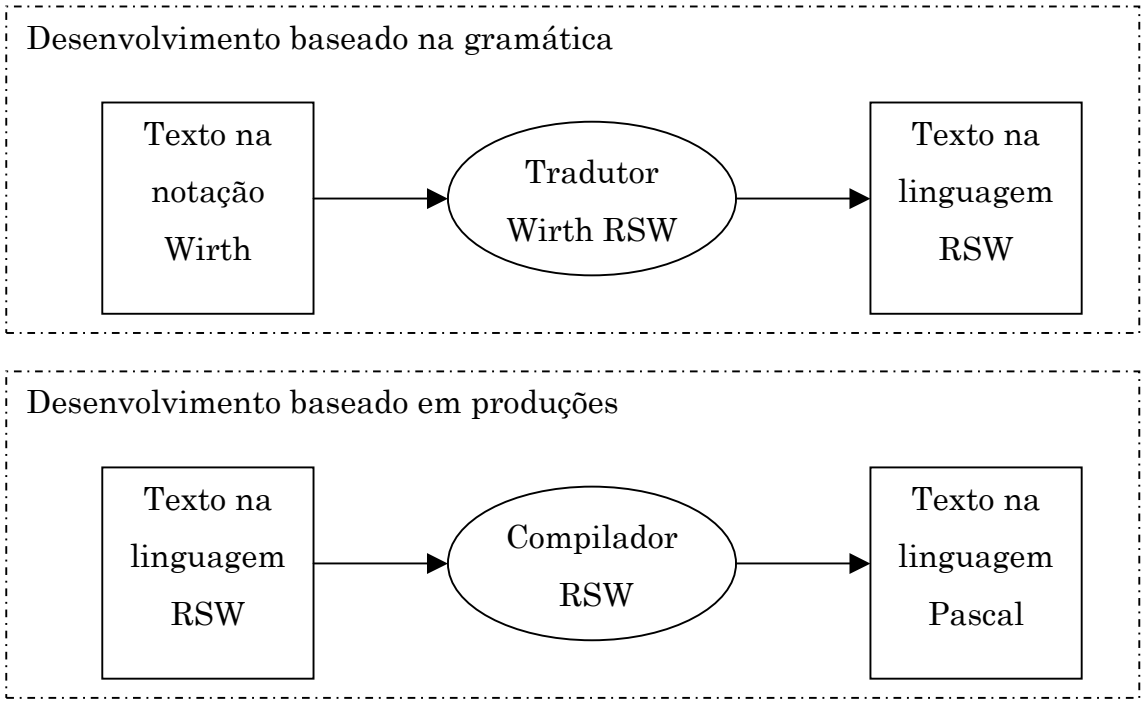

Figura 19 - Ciclo de implementação com o tradutor Wirth-RSW

A utilidade deste exemplo vai além da simples utilização da solução de um problema objetivando o entendimento da linguagem RSW. Dando condições ao usuário de transformar sentenças na notação Wirth para a linguagem RSW, possibilitamos que as linguagens sejam descritas através das regras da sua gramática. Este formato, além de mais prático, exige menos trabalho

de digitação por parte do usuário, conseqüentemente, menos erros serão introduzidos durante a fase de desenvolvimento de um reconhecedor sintático. Portanto com algumas modificações e incrementos nas sub-máquinas apresentadas é possível integrar o reconhecimento da notação Wirth no ambiente RSW. A Figura 19 ilustra um possível ciclo de implementação de um reconhecedor sintático, uma vez tendo disponível a funcionalidade descrita acima.

O ciclo de implementação apresentado possibilita que o usuário tire proveito das vantagens de trabalhar com base tanto nas regras da gramática, quanto no detalhe de cada produção. O esquema sugerido é análogo ao desenvolvimento efetuado misturando linguagens de programação de alto nível como, por exemplo, o Visual Basic, com linguagens de programação de baixo nível como, por exemplo, uma lingaguem de montagem, ou mesmo, código de máquina.

Capítulo VI - Conclusão

Metas Alcançadas

Uma das metas alcançadas mais significativas deste trabalho foi a implementação de um compilador que utiliza as técnicas e os mecanismos dos autômatos adaptativos para reconhecer na íntegra, através de mecanismos puramente sintáticos, a linguagem proposta. O compilador RSW implementa várias soluções para problemas tradicionalmente considerados como de responsabilidade das rotinas semânticos, diretamente nos analisadores léxico e sintático. Formado por quatro sub-máquinas, a interação entre os analisadores léxico e sintático possibilitam a coleta de novos identificadores da cadeia de entrada, à medida que esta vai sendo consumida, possibilitam também a verificação dos identificadores encontrados anteriormente, o tratamento da validade de identificadores declarados em vários níveis de escopo e a detecção da redefinição de um identificador dentro do mesmo escopo.

Outra meta significativa alcançada, é a possibilidade que o compilador RSW, associado ao ambiente RSW, disponibiliza aos seus usuários, de desenvolverem, de maneira totalmente uniforme, reconhecedores sintáticos baseados em autômatos finitos, autômatos de pilha estruturados ou autômatos adaptativos. O processo de desenvolvimento pode ser realizado com o auxilio de uma ferramenta baseada em ambiente gráfico, que possibilita a execução das principais atividades de desenvolvimento, tais como, a edição dos códigos-fonte, a compilação, e detecção e recuperação de erros de compilação, a execução e a depuração do reconhecedor sintático durante a análise de uma cadeia de entrada qualquer.

Contribuições

As contribuições desta dissertação são descritas na lista abaixo.

- Notação unificada para descrição dos analisadores léxico e sintático.
- Criação de uma ferramenta para o tratamento de linguagens dependentes de contexto utilizando uma notação uniforme. Observamos a existência de inúmeras ferramentas para o tratamento de linguagens livres de contexto, entretanto, poucas são as ferramentas disponíveis para o tratamento de dependências de contexto.
- Tratamento puramente sintático para todos os aspectos sintáticos da linguagem, incluindo a semantica estática (dependência de contexto) e mecanismos de extensão (sintaxe dinâmica).
- Ambiente amigável para uso didático nas áreas de compiladores, linguagens de programação, autômatos, linguagens formais, geração automática de programas, entre outros.

Possíveis Evoluções

Inúmeras são as possíveis evoluções que foram identificadas ao longo deste trabalho. Temos evoluções relacionadas com a arquitetura adotada na construção da ferramenta RSW, novas funcionalidades no ambiente de desenvolvimento integrado, extensões da linguagem RSW no sentido de possibilitar construções mais simples e legíveis, extensões no modelo dos autômatos adaptativos, entre outras. Abaixo temos a descrição de algumas evoluções identificadas:

- Definição de uma gramática adaptativa como ponto de partida para especificação dos Autômatos Adaptativos .
- Extensão do modelo dos Autômatos Adaptativos acrescentando funcionalidades de transdução, possibilitando assim a definição de ações de saída e geração sintática de saídas
- Extensão do modelo dos Autômatos Adaptativos acrescentando a manipulação de atributos, para facilitar a integração entre os analisadores léxico e sintático com o analisador semântico.
- Uso de uma interface adequada para uso de notação gráfica para a entrada das especificações das transições dos Autômatos Adaptativos, com possível animação gráfica.
- Por razões de desempenho, a manipulação da pilha de estado foi implementada através da pilha do código executável do compilador RSW. Este método não possibilita, de forma aceitável, a manipulação da pilha do programa, para operações de consulta, inserção ou remoção. Desta maneira, a implementação da pilha explícita, onde forem possível tais operações, será uma evolução na ferramenta hoje disponível.

- Extensão da linguagem RSW para o tratamento de conjuntos de átomos, podendo-se então definir um conjunto de átomos e utilizá-lo no lugar dos átomos de entrada.
- Implementação de uma nova representação para os átomos e da tabela de transições associados aos estados, para que seja possível a geração dinâmica de átomos.

Anexo I – Compilador RSW versão 3.00 na Linguagem RSW

```
1-    submaquina  Gram300 ( SBGrammarTK )

entrada Scan300  % Analisador Lexico do Projeto

producoes

  ( e1, automatoTK) ->  e2.

  ( e2, idTK) -> e3 : DefinirIdSubMaquina  <DefinirMaquinaCorrente>.
  ( e2, idAutomatoTK) -> e3 <DefinirMaquinaCorrente>.

  ( e3, lparenTK) -> e5.

  ( e5, idTK) -> e6 : DefinirIdAtomo <DefinirAtomoSubMaquina>.
  ( e5, idTokenTK) -> e6 <DefinirAtomoSubMaquina>.

  ( e6, rparenTK) -> e7.
  ( e6, commaTK) -> e5.

  ( e7, inputTK) -> e10.

  ( e4, lparenTK) -> (^e4, Prod300, lparenTK).  % Chamada da Sub-Maquina Producao
  ( e4, SBProductionTK) -> e8  <DefinirTransicaoAnalisada>.
  ( e4, functionsTK) -> e12.
  ( e4, eofTK) -> ( e9, SBGrammarTK) <VerificarErrosSemanticos>.

  ( e8, dotTK) -> e4.

  ( ^*, e9 ) -> ( ^, *, emptyTK ).

  ( e10, idTK) -> e11 : DefinirIdSubMaquina <DefinirSubMaquinaEntrada>.
  ( e10, idAutomatoTK) -> e11 <DefinirSubMaquinaEntrada>.

  ( e11, productionsTK) -> e4 <InicioCompilacaoProducao>.

  % Chamada da Sub-Maquina Funcao
  ( e12, idTK) -> (^e12, Func300, idTK) <InicioCompilacaoFuncao>.
  ( e12, SBFunctionTK) -> e12 : DefinirModoColetaVerificacao <TerminoCompilacaoFuncao>.
  ( e12, eofTK) -> (e9, SBGrammarTK) <VerificarErrosSemanticos>.

funcoes

%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo submaquina
DefinirIdSubMaquina =
{
  Scan300.DefinirIdSubMaquina
}


%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo estado
```

```
DefinirIdEstado =
{
  Scan300.DefinirIdEstado
}


%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo atomo
DefinirIdAtomo =
{
  Scan300.DefinirIdAtomo
}

DefinirModoColetaVerificacao =
{
  Scan300.DefinirModoColetaVerificacao
}
```

Listagem 34 - A sub-máquina Gram300 - Arquivo Gram300.rsm

```
1-    submaquina  Prod300 ( SBProductionTK )

entrada Scan300  % Analisador Lexico do Projeto

producoes

  ( e1, lparenTK) -> e2  <InicializarProducao>.

  ( e2, synTK) -> e3.
  ( e2, idTK) -> e6 : DefinirIdEstado <LocalizarCriarEstadoOrigem>.
  ( e2, idStateTK) -> e6 <LocalizarCriarEstadoOrigem>.

  ( e3, timesTK) -> e4.
  ( e3, commaTK) -> e5.

  ( e4, commaTK) -> e36.

  ( e5, idTK) -> e6 : DefinirIdEstado <LocalizarCriarEstadoOrigem>.
  ( e5, idStateTK) -> e6 <LocalizarCriarEstadoOrigem>.

  ( e6, commaTK) -> e7.
  ( e6, rparenTK) -> e35  <DefinirEstadoOrigemFinal>.

  ( e7,stringTK ) -> e8 <DefinirAtomoOrigemString>.
  ( e7,idTK ) -> e8 : DefinirIdAtomo <DefinirAtomoOrigem>.
  ( e7,idTokenTK ) -> e8 <DefinirAtomoOrigem>.
  ( e7,numTK ) -> e8  <DefinirAtomoOrigemNumero>.
  ( e7,anyTK ) -> e7_1.

  ( e7_1,minusTK ) -> e7_2.

  ( e7_2,stringTK ) -> e8 <DefinirAtomoOrigemStringMenos>.
  ( e7_2,numTK ) -> e8  <DefinirAtomoOrigemNumeroMenos>.

  ( e8, rparenTK) -> e9.

  ( e9, ddotTK) -> e10.
```

```
( e9, arrowTK) -> e17.

( e10, idTK) -> e11 : DefinirIdFuncao <DefinirAcaoAdaptativaPre>.
( e10, idFunctionTK) -> e11 <DefinirAcaoAdaptativaPre>.

( e11, lparenTK) -> e12.
( e11, arrowTK) -> e17.

( e12, idTK) -> e13 <AdicionarArgumentoAcao>.
( e12, idAutomatoTK) -> e13 <AdicionarArgumentoAcaoMaquina>.
( e12, idStateTK) -> e13 <AdicionarArgumentoAcaoEstado>.
( e12, idTokenTK) -> e13 <AdicionarArgumentoAcaoAtomo>.

( e13, commaTK) -> e14.
( e13, rparenTK) -> e16.

( e14, idTK) -> e15   <AdicionarArgumentoAcao>.
( e14, idAutomatoTK) -> e15 <AdicionarArgumentoAcaoMaquina>.
( e14, idStateTK) -> e15 <AdicionarArgumentoAcaoEstado>.
( e14, idTokenTK) -> e15 <AdicionarArgumentoAcaoAtomo>.

( e15, commaTK) -> e14.
( e15, rparenTK) -> e16.

( e16, arrowTK) -> e17.

( e17, lparenTK) -> e18.
( e17, idTK) -> e25 : DefinirIdEstado <LocalizarCriarEstadoDestino>.
( e17, idStateTK) -> e25 <LocalizarCriarEstadoDestino>.

( e18, synTK) -> e19.
( e18, idTK) -> e22 : DefinirIdEstado <LocalizarCriarEstadoDestino>.
( e18, idStateTK) -> e22 <LocalizarCriarEstadoDestino>.

( e19, idTK) -> e20 : DefinirIdEstado <LocalizarCriarEstadoDestino>.
( e19, idStateTK) -> e20 <LocalizarCriarEstadoDestino>.
( e19, commaTK) -> e21.

( e20, commaTK) -> e21.

( e21, idTK) -> e22 : DefinirIdSubMaquina <LocalizarCriarMaquinaDestino>.
( e21, idAutomatoTK) -> e22 <LocalizarCriarMaquinaDestino>.

( e22, rparenTK) -> e25.
( e22, commaTK) -> e23.

( e23, idTK) -> e24 : DefinirIdAtomo <DefinirAtomoDestino>.
( e23, idTokenTK) -> e24 <DefinirAtomoDestino>.
( e23, stringTK) -> e24 <DefinirAtomoDestinoString>.

( e24, rparenTK) -> e25.

( ^*, e25 ) -> ( ^, *, SBProductionTK) .
( e25, ddotTK) -> e26.
( e25, lessTK) -> e33.

( e26, idTK) -> e27 : DefinirIdFuncao <DefinirAcaoAdaptativaPos>.
( e26, idFunctionTK) -> e27 <DefinirAcaoAdaptativaPos>.
```

```
  ( ^*, e27 ) -> ( ^, *, SBProductionTK) .
  ( e27, lparenTK) -> e28.
  ( e27, lessTK) -> e33.

  ( e28, idTK) -> e29  <AdicionarArgumentoAcao>.
  ( e28, idAutomatoTK) -> e29 <AdicionarArgumentoAcaoMaquina>.
  ( e28, idStateTK) -> e29 <AdicionarArgumentoAcaoEstado>.
  ( e28, idTokenTK) -> e29 <AdicionarArgumentoAcaoAtomo>.

  ( e29, commaTK) -> e30.
  ( e29, rparenTK) -> e32.

  ( e30, idTK) -> e31  <AdicionarArgumentoAcao>.
  ( e30, idAutomatoTK) -> e31 <AdicionarArgumentoAcaoMaquina>.
  ( e30, idStateTK) -> e31 <AdicionarArgumentoAcaoEstado>.
  ( e30, idTokenTK) -> e31 <AdicionarArgumentoAcaoAtomo>.

  ( e31, commaTK) -> e30.
  ( e31, rparenTK) -> e32.

  ( ^*, e32 ) -> ( ^, *, SBProductionTK) .
  ( e32, lessTK) -> e33.

  ( e33, idTK) -> e34  <DefinirRotinaSemantica>.

  ( e34, greaterTK) -> e35.

  ( ^*, e35 ) -> ( ^, *, SBProductionTK) .

  ( e36, idTK) -> e37 : DefinirIdEstado <LocalizarCriarEstadoOrigem>.
  ( e36, idStateTK) -> e37 <LocalizarCriarEstadoOrigem>.

  ( e37, rparenTK) -> e38.

  ( e38, arrowTK) -> e39.

  ( e39, lparenTK) -> e40.

  ( e40, synTK) -> e41.

  ( e41, commaTK) -> e42.

  ( e42, timesTK) -> e43.

  ( e43, commaTK) -> e44.

  ( e44, idTK) -> e45 : DefinirIdAtomo <DefinirAtomoDestino>.
  ( e44, idTokenTK) -> e45  <DefinirAtomoDestino>.
  ( e44, stringTK) -> e45 <DefinirAtomoDestinoString>.

  ( e45, rparenTK) -> e35  <DefinirEstadoAtomoFinal>.


funcoes

%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo submaquina
DefinirIdSubMaquina =
{
```

```
  estado UltimoEstadoIdentificador :

  -[(UltimoEstadoIdentificador,emptyTK) -> Scan300.IdentificadorDesconhecido]

  +[(^*, UltimoEstadoIdentificador) -> (^, *, idAutomatoTK)]
}


%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo estado
DefinirIdEstado =
{
  estado UltimoEstadoIdentificador :

  -[(UltimoEstadoIdentificador,emptyTK) -> Scan300.IdentificadorDesconhecido]

  +[(^*, UltimoEstadoIdentificador) -> (^, *, idStateTK)]
}


%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo atomo
DefinirIdAtomo =
{
  Scan300.DefinirIdAtomo
}

%Funcao adaptativa que define o ultimo identificador
%reconhecido como do tipo atomo
DefinirIdFuncao =
{
  Scan300.DefinirIdFuncao
}
```

Listagem 35 - A sub-máquina Prod300 - Arquivo Prod300.rsm

```
1-   submaquina  Func300 ( SBFunctionTK )
2-
3-   entrada Scan300
4-
5-   producoes
6-
7-      ( e1, idTK) -> e2 : DefinirIdFuncao <DefinirFuncaoCorrente> .
8-      ( e1, idFunctionTK) -> e2 : AbrirBloco <DefinirFuncaoCorrente> .
9-
10-    ( e2, lparenTK) -> e3 : DefinirModoVerificacao.
11-    ( e2, equalTK)  -> e8.
12-
13-    ( e3, AutomatoTK) -> e3_1 : DefinirModoColeta.
14-    ( e3, StateTK)    -> e3_2 : DefinirModoColeta.
15-    ( e3, TokenTK)    -> e3_3 : DefinirModoColeta.
16-
17-    ( e3_1, idTK) : DefinirModoVerificacao -> e4 : DefinirIdSubMaquina
      <AdicionarParametroSubMaquinaFuncao>.
18-    ( e3_2, idTK) : DefinirModoVerificacao -> e4 : DefinirIdEstado
      <AdicionarParametroEstadoFuncao>.
```

```
19-   ( e3_3, idTK) : DefinirModoVerificacao -> e4 : DefinirIdAtomo
      <AdicionarParametroAtomoFuncao>.
20-
21-   ( e4, commaTK) -> e5.
22-   ( e4, rparenTK) -> e7.
23-
24-   ( e5, AutomatoTK) -> e5_1 : DefinirModoColeta.
25-   ( e5, StateTK)    -> e5_2 : DefinirModoColeta.
26-   ( e5, TokenTK)    -> e5_3 : DefinirModoColeta.
27-
28-   ( e5_1, idTK) : DefinirModoVerificacao -> e6 : DefinirIdSubMaquina
      <AdicionarParametroSubMaquinaFuncao>.
29-   ( e5_2, idTK) : DefinirModoVerificacao -> e6 : DefinirIdEstado
      <AdicionarParametroEstadoFuncao>.
30-   ( e5_3, idTK) : DefinirModoVerificacao -> e6 : DefinirIdAtomo
      <AdicionarParametroAtomoFuncao>.
31-
32-   ( e6, commaTK) -> e5.
33-   ( e6, rparenTK) -> e7.
34-
35-   ( e7, equalTK ) -> e8 .
36-
37-   ( e8, lbraceTK) -> e9 : DefinirModoVerificacao.
38-
39-   ( e9, plusTK)     -> e16 <DefinirAcaoElementarInclusao>.
40-   ( e9, minusTK)    -> e16 <DefinirAcaoElementarExclusao>.
41-   ( e9, interTK)    -> e16 <DefinirAcaoElementarConsulta>.
42-   ( e9, rbraceTK)   -> e20.
43-   ( e9, AutomatoTK) -> e12_1.
44-   ( e9, StateTK)    -> e12_2.
45-   ( e9, TokenTK)    -> e12_3.
46-
47-   ( e10, commaTK) -> e12.
48-   ( e10, ddotTK) -> e13.
49-
50-   ( e11_1, idTK) -> e10 : DefinirIdSubMaquina <AdicionarGeradorSubMaquinaFuncao>.
51-   ( e11_2, idTK) -> e10 : DefinirIdEstado     <AdicionarGeradorEstadoFuncao>.
52-   ( e11_3, idTK) -> e10 : DefinirIdAtomo      <AdicionarGeradorAtomoFuncao>.
53-
54-   ( e12, AutomatoTK) -> e12_1.
55-   ( e12, StateTK)    -> e12_2.
56-   ( e12, TokenTK)    -> e12_3.
57-
58-   ( e12_1, idTK) -> e10 : DefinirIdSubMaquina <AdicionarVariavelSubMaquinaFuncao>.
59-   ( e12_1, timesTK) -> e11_1.
60-   ( e12_2, idTK) -> e10 : DefinirIdEstado <AdicionarVariavelEstadoFuncao>.
61-   ( e12_2, timesTK) -> e11_2.
62-   ( e12_3, idTK) -> e10 : DefinirIdAtomo <AdicionarVariavelAtomoFuncao>.
63-   ( e12_3, timesTK) -> e11_3.
64-
65-   ( e13, plusTK) -> e16  <DefinirAcaoElementarInclusao>.
66-   ( e13, minusTK) -> e16  <DefinirAcaoElementarExclusao>.
67-   ( e13, interTK) -> e16  <DefinirAcaoElementarConsulta>.
68-   ( e13, rbraceTK) -> e20 .
69-
70-   ( e16, lbracketTK) -> e17 .
71-
72-   ( e17, lparenTK) -> ( ^e17, Prod300, lparenTK).
73-   ( e17, SBproductionTK) -> e18 <DefinirTransicaoAnalisada>.
```

```
74-
75-    ( e18, rbracketTK) -> e19 <AdicionarAcaoElementar>.
76-
77-    ( e19, plusTK) -> e16 <DefinirAcaoElementarInclusao>.
78-    ( e19, minusTK) -> e16 <DefinirAcaoElementarExclusao>.
79-    ( e19, interTK) -> e16  <DefinirAcaoElementarConsulta>.
80-    ( e19, rbraceTK) -> e20 : FecharBloco.
81-
82-    ( ^*, e20 ) -> ( ^, *, SBFunctionTK).
83-
84- funcoes
85-
86- %Funcao adaptativa que define o ultimo identificador
87- %reconhecido como do tipo submaquina
88- DefinirIdSubMaquina =
89- {
90-   Scan300.DefinirIdSubMaquina
91- }
92-
93-
94- %Funcao adaptativa que define o ultimo identificador
95- %reconhecido como do tipo estado
96- DefinirIdEstado =
97- {
98-   Scan300.DefinirIdEstado
99- }
100-
101-
102-     %Funcao adaptativa que define o ultimo identificador
103-     %reconhecido como do tipo atomo
104-     DefinirIdAtomo =
105-     {
106-       Scan300.DefinirIdAtomo
107-     }
108-
109-     %Funcao adaptativa que define o ultimo identificador
110-     %reconhecido como do tipo funcao
111-     DefinirIdFuncao =
112-     {
113-       Scan300.DefinirIdFuncao
114-     }
115-
116-     AbrirBloco =
117-     {
118-       Scan300.AbrirBloco
119-     }
120-
121-     FecharBloco =
122-     {
123-       Scan300.FecharBloco
124-     }
125-
126-     DefinirModoColeta =
127-     {
128-       Scan300.DefinirModoColeta
129-     }
130-
131-     DefinirModoVerificacao =
132-     {
```

```
133-         Scan300.DefinirModoVerificacao
134-      }
```

Listagem 36 - A sub-máquina Func300 - Arquivo Func300.rsm

```
1-   submaquina Scan300 % Analisador Lexico do Projeto
2-
3-   (
4-     lparenTK,      %'('
5-     rparenTK,      %')'
6-     commaTK,       %','
7-     dotTK,         %'.'
8-     synTK,         %'^'
9-     timesTK,       %'*'
10-    ddotTK,        %':'
11-    lessTk,        %'<'
12-    greaterTK,     %'>'
13-    arrowTK,       %'->'
14-    automatoTK,    %'submaquina'
15-    inputTK,       %'entrada'
16-    stateTK,       %'estado'
17-    tokenTK,       %'atomo'
18-    productionsTK, %'producoes'
19-    stringTK,      %""{qualquer}""
20-    idTK,          %(letter+'_') {letter+'_'+digits}[.(letter+'_') {letter+'_'+digits}]
21-    idRedefinedTK, %Redefinicao de identificador no mesmo bloco
22-    idAutomatoTK,  %identificador do tipo sub-maquina
23-    idStateTK,     %identificador do tipo estado
24-    idTokenTK,     %identificador do tipo atomo
25-    idFunctionTK,  %identificador do tipo funcao
26-    numTK,         %digits {digits}
27-    eofTK ,        %fim da cadeia de entrada (Caracter 255)
28-    anyTK ,        %qualquer
29-    minusTK,       %'-'
30-    plusTK,        %'+'
31-    functionsTK,   %'funcoes'
32-    lbraceTK,      %'{'
33-    rbraceTK,      %'}'
34-    lbracketTK,    %'['
35-    rbracketTK,    %']'
36-    equalTK,       %'='
37-    interTK        %'?'
38-  )
39-
40-  entrada CadeiaEntrada %Cadeia de Caraceteres primaria
41-
42-  producoes
43-  % Secao de declaracao das producoes
44-
45-  % Estado inicial que permite alteracoes estruturais antes
46-  % da analise da cadeia de entrada
47-    ( e0, emptyTK) -> e1.
48-
49-  %Declaracao dos simbolos unitarios e compostos
50-
51-    ( e1, 255) -> (e2, eofTK).
52-
```

```
53-    ( e1, "(") -> ( e2, lparenTK).
54-
55-    ( e1, ")") -> ( e2, rparenTK).
56-
57-    ( e1, ",") -> ( e2, commaTK).
58-
59-    ( e1, ".") -> ( e2, dotTK).
60-
61-    ( e1, "^") -> ( e2, synTK).
62-
63-    ( e1, "*") -> ( e2, timesTK).
64-
65-    ( e1, ":") -> ( e2, ddotTK).
66-
67-    ( e1, "<") -> ( e2, lessTK).
68-
69-    ( e1, ">") -> ( e2, greaterTK).
70-
71-    ( e1, "{") -> ( e2, lbraceTK).
72-
73-    ( e1, "}") -> ( e2, rbraceTK).
74-
75-    ( e1, "[") -> ( e2, lbracketTK).
76-
77-    ( e1, "]") -> ( e2, rbracketTK).
78-
79-    ( e1, "=") -> ( e2, equalTK).
80-
81-    ( e1, "+") -> ( e2, plusTK).
82-
83-    ( e1, "?") -> ( e2, interTK).
84-
85-    ( e1, "-") ->  e3 .
86-    ( e3, ">") -> ( e2, arrowTK).
87-    ( e3, emptyTK) -> ( e2, minusTK).
88-
89-    ( ^*, e2 ) -> ( ^, * , emptyTK).
90-
91-  % Declaracao das Palavras Reservadas
92-
93-  %  ( e1, "submaquina") -> e4.
94-     ( ^*, e4 ) -> ( ^, * , automatoTK ).
95-  %  ( e1, "entrada") -> e5.
96-  %  ( e1, "estado")  -> e18.
97-  ( ^*, e5 ) -> ( ^, *, inputTK).
98-  ( ^*, e18 ) -> ( ^, *, stateTK).
99-  %  ( e1, "producoes" ) -> e6.
100-         ( ^*, e6 ) -> ( ^, *, productionsTK ).
101-       %  ( e1, "funcoes" ) -> e17.
102-       ( ^*, e17 ) -> ( ^, *, functionsTK ).
103-       % ( e1, "qualquer" ) ->e15.
104-       ( ^*, e15) -> ( ^,*, anyTK).
105-       %  ( e1, "atomo" ) -> e19.
106-         ( e1, "a" ) -> e191.
107-         ( e191, "t" ) -> e192.
108-         ( e191, "_" ) : CriarTransicao(e191,"_") -> erro.
109-         ( e191, "a" ) : CriarTransicao(e191,"a") -> erro.
110-         ( e191, "b" ) : CriarTransicao(e191,"b") -> erro.
111-         ( e191, "c" ) : CriarTransicao(e191,"c") -> erro.
```

```
112-      ( e191, "d" ) : CriarTransicao(e191,"d") -> erro.
113-      ( e191, "e" ) : CriarTransicao(e191,"e") -> erro.
114-      ( e191, "f" ) : CriarTransicao(e191,"f") -> erro.
115-      ( e191, "g" ) : CriarTransicao(e191,"g") -> erro.
116-      ( e191, "h" ) : CriarTransicao(e191,"h") -> erro.
117-      ( e191, "i" ) : CriarTransicao(e191,"i") -> erro.
118-      ( e191, "j" ) : CriarTransicao(e191,"j") -> erro.
119-      ( e191, "k" ) : CriarTransicao(e191,"k") -> erro.
120-      ( e191, "l" ) : CriarTransicao(e191,"l") -> erro.
121-      ( e191, "m" ) : CriarTransicao(e191,"m") -> erro.
122-      ( e191, "n" ) : CriarTransicao(e191,"n") -> erro.
123-      ( e191, "o" ) : CriarTransicao(e191,"o") -> erro.
124-      ( e191, "p" ) : CriarTransicao(e191,"p") -> erro.
125-      ( e191, "q" ) : CriarTransicao(e191,"q") -> erro.
126-      ( e191, "r" ) : CriarTransicao(e191,"r") -> erro.
127-      ( e191, "s" ) : CriarTransicao(e191,"s") -> erro.
128-      ( e191, "u" ) : CriarTransicao(e191,"u") -> erro.
129-      ( e191, "w" ) : CriarTransicao(e191,"w") -> erro.
130-      ( e191, "v" ) : CriarTransicao(e191,"v") -> erro.
131-      ( e191, "x" ) : CriarTransicao(e191,"x") -> erro.
132-      ( e191, "y" ) : CriarTransicao(e191,"y") -> erro.
133-      ( e191, "z" ) : CriarTransicao(e191,"z") -> erro.
134-      ( e191, "1" ) : CriarTransicao(e191,"1") -> erro.
135-      ( e191, "2" ) : CriarTransicao(e191,"2") -> erro.
136-      ( e191, "3" ) : CriarTransicao(e191,"3") -> erro.
137-      ( e191, "4" ) : CriarTransicao(e191,"4") -> erro.
138-      ( e191, "5" ) : CriarTransicao(e191,"5") -> erro.
139-      ( e191, "6" ) : CriarTransicao(e191,"6") -> erro.
140-      ( e191, "7" ) : CriarTransicao(e191,"7") -> erro.
141-      ( e191, "8" ) : CriarTransicao(e191,"8") -> erro.
142-      ( e191, "9" ) : CriarTransicao(e191,"9") -> erro.
143-      ( e191, "0" ) : CriarTransicao(e191,"0") -> erro.
144-      ( e191, "A" ) : CriarTransicao(e191,"A") -> erro.
145-      ( e191, "B" ) : CriarTransicao(e191,"B") -> erro.
146-      ( e191, "C" ) : CriarTransicao(e191,"C") -> erro.
147-      ( e191, "D" ) : CriarTransicao(e191,"D") -> erro.
148-      ( e191, "E" ) : CriarTransicao(e191,"E") -> erro.
149-      ( e191, "F" ) : CriarTransicao(e191,"F") -> erro.
150-      ( e191, "G" ) : CriarTransicao(e191,"G") -> erro.
151-      ( e191, "H" ) : CriarTransicao(e191,"H") -> erro.
152-      ( e191, "I" ) : CriarTransicao(e191,"I") -> erro.
153-      ( e191, "J" ) : CriarTransicao(e191,"J") -> erro.
154-      ( e191, "K" ) : CriarTransicao(e191,"K") -> erro.
155-      ( e191, "L" ) : CriarTransicao(e191,"L") -> erro.
156-      ( e191, "M" ) : CriarTransicao(e191,"M") -> erro.
157-      ( e191, "N" ) : CriarTransicao(e191,"N") -> erro.
158-      ( e191, "O" ) : CriarTransicao(e191,"O") -> erro.
159-      ( e191, "P" ) : CriarTransicao(e191,"P") -> erro.
160-      ( e191, "Q" ) : CriarTransicao(e191,"Q") -> erro.
161-      ( e191, "R" ) : CriarTransicao(e191,"R") -> erro.
162-      ( e191, "S" ) : CriarTransicao(e191,"S") -> erro.
163-      ( e191, "T" ) : CriarTransicao(e191,"T") -> erro.
164-      ( e191, "W" ) : CriarTransicao(e191,"W") -> erro.
165-      ( e191, "U" ) : CriarTransicao(e191,"U") -> erro.
166-      ( e191, "V" ) : CriarTransicao(e191,"V") -> erro.
167-      ( e191, "Y" ) : CriarTransicao(e191,"Y") -> erro.
168-      ( e191, "X" ) : CriarTransicao(e191,"X") -> erro.
169-      ( e191, "Z" ) : CriarTransicao(e191,"Z") -> erro.
170-      ( e191, emptyTK) : DefinirIdDesconhecido(e191) -> erro.
```

```
171-
172-        ( e192, "o" ) -> e193.
173-        ( e192, "_" ) : CriarTransicao(e192,"_") -> erro.
174-        ( e192, "a" ) : CriarTransicao(e192,"a") -> erro.
175-        ( e192, "b" ) : CriarTransicao(e192,"b") -> erro.
176-        ( e192, "c" ) : CriarTransicao(e192,"c") -> erro.
177-        ( e192, "d" ) : CriarTransicao(e192,"d") -> erro.
178-        ( e192, "e" ) : CriarTransicao(e192,"e") -> erro.
179-        ( e192, "f" ) : CriarTransicao(e192,"f") -> erro.
180-        ( e192, "g" ) : CriarTransicao(e192,"g") -> erro.
181-        ( e192, "h" ) : CriarTransicao(e192,"h") -> erro.
182-        ( e192, "i" ) : CriarTransicao(e192,"i") -> erro.
183-        ( e192, "j" ) : CriarTransicao(e192,"j") -> erro.
184-        ( e192, "k" ) : CriarTransicao(e192,"k") -> erro.
185-        ( e192, "l" ) : CriarTransicao(e192,"l") -> erro.
186-        ( e192, "m" ) : CriarTransicao(e192,"m") -> erro.
187-        ( e192, "n" ) : CriarTransicao(e192,"n") -> erro.
188-        ( e192, "p" ) : CriarTransicao(e192,"p") -> erro.
189-        ( e192, "q" ) : CriarTransicao(e192,"q") -> erro.
190-        ( e192, "r" ) : CriarTransicao(e192,"r") -> erro.
191-        ( e192, "s" ) : CriarTransicao(e192,"s") -> erro.
192-        ( e192, "t" ) : CriarTransicao(e192,"t") -> erro.
193-        ( e192, "u" ) : CriarTransicao(e192,"u") -> erro.
194-        ( e192, "w" ) : CriarTransicao(e192,"w") -> erro.
195-        ( e192, "v" ) : CriarTransicao(e192,"v") -> erro.
196-        ( e192, "x" ) : CriarTransicao(e192,"x") -> erro.
197-        ( e192, "y" ) : CriarTransicao(e192,"y") -> erro.
198-        ( e192, "z" ) : CriarTransicao(e192,"z") -> erro.
199-        ( e192, "1" ) : CriarTransicao(e192,"1") -> erro.
200-        ( e192, "2" ) : CriarTransicao(e192,"2") -> erro.
201-        ( e192, "3" ) : CriarTransicao(e192,"3") -> erro.
202-        ( e192, "4" ) : CriarTransicao(e192,"4") -> erro.
203-        ( e192, "5" ) : CriarTransicao(e192,"5") -> erro.
204-        ( e192, "6" ) : CriarTransicao(e192,"6") -> erro.
205-        ( e192, "7" ) : CriarTransicao(e192,"7") -> erro.
206-        ( e192, "8" ) : CriarTransicao(e192,"8") -> erro.
207-        ( e192, "9" ) : CriarTransicao(e192,"9") -> erro.
208-        ( e192, "0" ) : CriarTransicao(e192,"0") -> erro.
209-        ( e192, "A" ) : CriarTransicao(e192,"A") -> erro.
210-        ( e192, "B" ) : CriarTransicao(e192,"B") -> erro.
211-        ( e192, "C" ) : CriarTransicao(e192,"C") -> erro.
212-        ( e192, "D" ) : CriarTransicao(e192,"D") -> erro.
213-        ( e192, "E" ) : CriarTransicao(e192,"E") -> erro.
214-        ( e192, "F" ) : CriarTransicao(e192,"F") -> erro.
215-        ( e192, "G" ) : CriarTransicao(e192,"G") -> erro.
216-        ( e192, "H" ) : CriarTransicao(e192,"H") -> erro.
217-        ( e192, "I" ) : CriarTransicao(e192,"I") -> erro.
218-        ( e192, "J" ) : CriarTransicao(e192,"J") -> erro.
219-        ( e192, "K" ) : CriarTransicao(e192,"K") -> erro.
220-        ( e192, "L" ) : CriarTransicao(e192,"L") -> erro.
221-        ( e192, "M" ) : CriarTransicao(e192,"M") -> erro.
222-        ( e192, "N" ) : CriarTransicao(e192,"N") -> erro.
223-        ( e192, "O" ) : CriarTransicao(e192,"O") -> erro.
224-        ( e192, "P" ) : CriarTransicao(e192,"P") -> erro.
225-        ( e192, "Q" ) : CriarTransicao(e192,"Q") -> erro.
226-        ( e192, "R" ) : CriarTransicao(e192,"R") -> erro.
227-        ( e192, "S" ) : CriarTransicao(e192,"S") -> erro.
228-        ( e192, "T" ) : CriarTransicao(e192,"T") -> erro.
229-        ( e192, "W" ) : CriarTransicao(e192,"W") -> erro.
```

```
230-        ( e192, "U" ) : CriarTransicao(e192,"U") -> erro.
231-        ( e192, "V" ) : CriarTransicao(e192,"V") -> erro.
232-        ( e192, "Y" ) : CriarTransicao(e192,"Y") -> erro.
233-        ( e192, "X" ) : CriarTransicao(e192,"X") -> erro.
234-        ( e192, "Z" ) : CriarTransicao(e192,"Z") -> erro.
235-        ( e192, emptyTK) : DefinirIdDesconhecido(e192) -> erro.
236-
237-        ( e193, "m" ) -> e194.
238-        ( e193, "_" ) : CriarTransicao(e193,"_") -> erro.
239-        ( e193, "a" ) : CriarTransicao(e193,"a") -> erro.
240-        ( e193, "b" ) : CriarTransicao(e193,"b") -> erro.
241-        ( e193, "c" ) : CriarTransicao(e193,"c") -> erro.
242-        ( e193, "d" ) : CriarTransicao(e193,"d") -> erro.
243-        ( e193, "e" ) : CriarTransicao(e193,"e") -> erro.
244-        ( e193, "f" ) : CriarTransicao(e193,"f") -> erro.
245-        ( e193, "g" ) : CriarTransicao(e193,"g") -> erro.
246-        ( e193, "h" ) : CriarTransicao(e193,"h") -> erro.
247-        ( e193, "i" ) : CriarTransicao(e193,"i") -> erro.
248-        ( e193, "j" ) : CriarTransicao(e193,"j") -> erro.
249-        ( e193, "k" ) : CriarTransicao(e193,"k") -> erro.
250-        ( e193, "l" ) : CriarTransicao(e193,"l") -> erro.
251-        ( e193, "n" ) : CriarTransicao(e193,"n") -> erro.
252-        ( e193, "o" ) : CriarTransicao(e193,"o") -> erro.
253-        ( e193, "p" ) : CriarTransicao(e193,"p") -> erro.
254-        ( e193, "q" ) : CriarTransicao(e193,"q") -> erro.
255-        ( e193, "r" ) : CriarTransicao(e193,"r") -> erro.
256-        ( e193, "s" ) : CriarTransicao(e193,"s") -> erro.
257-        ( e193, "t" ) : CriarTransicao(e193,"t") -> erro.
258-        ( e193, "u" ) : CriarTransicao(e193,"u") -> erro.
259-        ( e193, "w" ) : CriarTransicao(e193,"w") -> erro.
260-        ( e193, "v" ) : CriarTransicao(e193,"v") -> erro.
261-        ( e193, "x" ) : CriarTransicao(e193,"x") -> erro.
262-        ( e193, "y" ) : CriarTransicao(e193,"y") -> erro.
263-        ( e193, "z" ) : CriarTransicao(e193,"z") -> erro.
264-        ( e193, "1" ) : CriarTransicao(e193,"1") -> erro.
265-        ( e193, "2" ) : CriarTransicao(e193,"2") -> erro.
266-        ( e193, "3" ) : CriarTransicao(e193,"3") -> erro.
267-        ( e193, "4" ) : CriarTransicao(e193,"4") -> erro.
268-        ( e193, "5" ) : CriarTransicao(e193,"5") -> erro.
269-        ( e193, "6" ) : CriarTransicao(e193,"6") -> erro.
270-        ( e193, "7" ) : CriarTransicao(e193,"7") -> erro.
271-        ( e193, "8" ) : CriarTransicao(e193,"8") -> erro.
272-        ( e193, "9" ) : CriarTransicao(e193,"9") -> erro.
273-        ( e193, "0" ) : CriarTransicao(e193,"0") -> erro.
274-        ( e193, "A" ) : CriarTransicao(e193,"A") -> erro.
275-        ( e193, "B" ) : CriarTransicao(e193,"B") -> erro.
276-        ( e193, "C" ) : CriarTransicao(e193,"C") -> erro.
277-        ( e193, "D" ) : CriarTransicao(e193,"D") -> erro.
278-        ( e193, "E" ) : CriarTransicao(e193,"E") -> erro.
279-        ( e193, "F" ) : CriarTransicao(e193,"F") -> erro.
280-        ( e193, "G" ) : CriarTransicao(e193,"G") -> erro.
281-        ( e193, "H" ) : CriarTransicao(e193,"H") -> erro.
282-        ( e193, "I" ) : CriarTransicao(e193,"I") -> erro.
283-        ( e193, "J" ) : CriarTransicao(e193,"J") -> erro.
284-        ( e193, "K" ) : CriarTransicao(e193,"K") -> erro.
285-        ( e193, "L" ) : CriarTransicao(e193,"L") -> erro.
286-        ( e193, "M" ) : CriarTransicao(e193,"M") -> erro.
287-        ( e193, "N" ) : CriarTransicao(e193,"N") -> erro.
288-        ( e193, "O" ) : CriarTransicao(e193,"O") -> erro.
```

```
289-      ( e193, "P" ) : CriarTransicao(e193,"P") -> erro.
290-      ( e193, "Q" ) : CriarTransicao(e193,"Q") -> erro.
291-      ( e193, "R" ) : CriarTransicao(e193,"R") -> erro.
292-      ( e193, "S" ) : CriarTransicao(e193,"S") -> erro.
293-      ( e193, "T" ) : CriarTransicao(e193,"T") -> erro.
294-      ( e193, "W" ) : CriarTransicao(e193,"W") -> erro.
295-      ( e193, "U" ) : CriarTransicao(e193,"U") -> erro.
296-      ( e193, "V" ) : CriarTransicao(e193,"V") -> erro.
297-      ( e193, "Y" ) : CriarTransicao(e193,"Y") -> erro.
298-      ( e193, "X" ) : CriarTransicao(e193,"X") -> erro.
299-      ( e193, "Z" ) : CriarTransicao(e193,"Z") -> erro.
300-      ( e193, emptyTK) : DefinirIdDesconhecido(e193) -> erro.
301-
302-      ( e194, "o" ) -> e19.
303-      ( e194, "_" ) : CriarTransicao(e194,"_") -> erro.
304-      ( e194, "a" ) : CriarTransicao(e194,"a") -> erro.
305-      ( e194, "b" ) : CriarTransicao(e194,"b") -> erro.
306-      ( e194, "c" ) : CriarTransicao(e194,"c") -> erro.
307-      ( e194, "d" ) : CriarTransicao(e194,"d") -> erro.
308-      ( e194, "e" ) : CriarTransicao(e194,"e") -> erro.
309-      ( e194, "f" ) : CriarTransicao(e194,"f") -> erro.
310-      ( e194, "g" ) : CriarTransicao(e194,"g") -> erro.
311-      ( e194, "h" ) : CriarTransicao(e194,"h") -> erro.
312-      ( e194, "i" ) : CriarTransicao(e194,"i") -> erro.
313-      ( e194, "j" ) : CriarTransicao(e194,"j") -> erro.
314-      ( e194, "k" ) : CriarTransicao(e194,"k") -> erro.
315-      ( e194, "l" ) : CriarTransicao(e194,"l") -> erro.
316-      ( e194, "m" ) : CriarTransicao(e194,"m") -> erro.
317-      ( e194, "n" ) : CriarTransicao(e194,"n") -> erro.
318-      ( e194, "p" ) : CriarTransicao(e194,"p") -> erro.
319-      ( e194, "q" ) : CriarTransicao(e194,"q") -> erro.
320-      ( e194, "r" ) : CriarTransicao(e194,"r") -> erro.
321-      ( e194, "s" ) : CriarTransicao(e194,"s") -> erro.
322-      ( e194, "t" ) : CriarTransicao(e194,"t") -> erro.
323-      ( e194, "u" ) : CriarTransicao(e194,"u") -> erro.
324-      ( e194, "w" ) : CriarTransicao(e194,"w") -> erro.
325-      ( e194, "v" ) : CriarTransicao(e194,"v") -> erro.
326-      ( e194, "x" ) : CriarTransicao(e194,"x") -> erro.
327-      ( e194, "y" ) : CriarTransicao(e194,"y") -> erro.
328-      ( e194, "z" ) : CriarTransicao(e194,"z") -> erro.
329-      ( e194, "1" ) : CriarTransicao(e194,"1") -> erro.
330-      ( e194, "2" ) : CriarTransicao(e194,"2") -> erro.
331-      ( e194, "3" ) : CriarTransicao(e194,"3") -> erro.
332-      ( e194, "4" ) : CriarTransicao(e194,"4") -> erro.
333-      ( e194, "5" ) : CriarTransicao(e194,"5") -> erro.
334-      ( e194, "6" ) : CriarTransicao(e194,"6") -> erro.
335-      ( e194, "7" ) : CriarTransicao(e194,"7") -> erro.
336-      ( e194, "8" ) : CriarTransicao(e194,"8") -> erro.
337-      ( e194, "9" ) : CriarTransicao(e194,"9") -> erro.
338-      ( e194, "0" ) : CriarTransicao(e194,"0") -> erro.
339-      ( e194, "A" ) : CriarTransicao(e194,"A") -> erro.
340-      ( e194, "B" ) : CriarTransicao(e194,"B") -> erro.
341-      ( e194, "C" ) : CriarTransicao(e194,"C") -> erro.
342-      ( e194, "D" ) : CriarTransicao(e194,"D") -> erro.
343-      ( e194, "E" ) : CriarTransicao(e194,"E") -> erro.
344-      ( e194, "F" ) : CriarTransicao(e194,"F") -> erro.
345-      ( e194, "G" ) : CriarTransicao(e194,"G") -> erro.
346-      ( e194, "H" ) : CriarTransicao(e194,"H") -> erro.
347-      ( e194, "I" ) : CriarTransicao(e194,"I") -> erro.
```

```
348-      ( e194, "J" ) : CriarTransicao(e194,"J") -> erro.
349-      ( e194, "K" ) : CriarTransicao(e194,"K") -> erro.
350-      ( e194, "L" ) : CriarTransicao(e194,"L") -> erro.
351-      ( e194, "M" ) : CriarTransicao(e194,"M") -> erro.
352-      ( e194, "N" ) : CriarTransicao(e194,"N") -> erro.
353-      ( e194, "O" ) : CriarTransicao(e194,"O") -> erro.
354-      ( e194, "P" ) : CriarTransicao(e194,"P") -> erro.
355-      ( e194, "Q" ) : CriarTransicao(e194,"Q") -> erro.
356-      ( e194, "R" ) : CriarTransicao(e194,"R") -> erro.
357-      ( e194, "S" ) : CriarTransicao(e194,"S") -> erro.
358-      ( e194, "T" ) : CriarTransicao(e194,"T") -> erro.
359-      ( e194, "W" ) : CriarTransicao(e194,"W") -> erro.
360-      ( e194, "U" ) : CriarTransicao(e194,"U") -> erro.
361-      ( e194, "V" ) : CriarTransicao(e194,"V") -> erro.
362-      ( e194, "Y" ) : CriarTransicao(e194,"Y") -> erro.
363-      ( e194, "X" ) : CriarTransicao(e194,"X") -> erro.
364-      ( e194, "Z" ) : CriarTransicao(e194,"Z") -> erro.
365-      ( e194, emptyTK) : DefinirIdDesconhecido(e194) -> erro.
366-
367-      ( ^*, e19 ) -> ( ^, *, tokenTK ).
368-      ( e19, "_" ) : CriarTransicao(e19," _") -> erro.
369-      ( e19, "a" ) : CriarTransicao(e19,"a") -> erro.
370-      ( e19, "b" ) : CriarTransicao(e19,"b") -> erro.
371-      ( e19, "c" ) : CriarTransicao(e19,"c") -> erro.
372-      ( e19, "d" ) : CriarTransicao(e19,"d") -> erro.
373-      ( e19, "e" ) : CriarTransicao(e19,"e") -> erro.
374-      ( e19, "f" ) : CriarTransicao(e19,"f") -> erro.
375-      ( e19, "g" ) : CriarTransicao(e19,"g") -> erro.
376-      ( e19, "h" ) : CriarTransicao(e19,"h") -> erro.
377-      ( e19, "i" ) : CriarTransicao(e19,"i") -> erro.
378-      ( e19, "j" ) : CriarTransicao(e19,"j") -> erro.
379-      ( e19, "k" ) : CriarTransicao(e19,"k") -> erro.
380-      ( e19, "l" ) : CriarTransicao(e19,"l") -> erro.
381-      ( e19, "m" ) : CriarTransicao(e19,"m") -> erro.
382-      ( e19, "n" ) : CriarTransicao(e19,"n") -> erro.
383-      ( e19, "o" ) : CriarTransicao(e19,"o") -> erro.
384-      ( e19, "p" ) : CriarTransicao(e19,"p") -> erro.
385-      ( e19, "q" ) : CriarTransicao(e19,"q") -> erro.
386-      ( e19, "r" ) : CriarTransicao(e19,"r") -> erro.
387-      ( e19, "s" ) : CriarTransicao(e19,"s") -> erro.
388-      ( e19, "t" ) : CriarTransicao(e19,"t") -> erro.
389-      ( e19, "u" ) : CriarTransicao(e19,"u") -> erro.
390-      ( e19, "w" ) : CriarTransicao(e19,"w") -> erro.
391-      ( e19, "v" ) : CriarTransicao(e19,"v") -> erro.
392-      ( e19, "x" ) : CriarTransicao(e19,"x") -> erro.
393-      ( e19, "y" ) : CriarTransicao(e19,"y") -> erro.
394-      ( e19, "z" ) : CriarTransicao(e19,"z") -> erro.
395-      ( e19, "1" ) : CriarTransicao(e19,"1") -> erro.
396-      ( e19, "2" ) : CriarTransicao(e19,"2") -> erro.
397-      ( e19, "3" ) : CriarTransicao(e19,"3") -> erro.
398-      ( e19, "4" ) : CriarTransicao(e19,"4") -> erro.
399-      ( e19, "5" ) : CriarTransicao(e19,"5") -> erro.
400-      ( e19, "6" ) : CriarTransicao(e19,"6") -> erro.
401-      ( e19, "7" ) : CriarTransicao(e19,"7") -> erro.
402-      ( e19, "8" ) : CriarTransicao(e19,"8") -> erro.
403-      ( e19, "9" ) : CriarTransicao(e19,"9") -> erro.
404-      ( e19, "0" ) : CriarTransicao(e19,"0") -> erro.
405-      ( e19, "A" ) : CriarTransicao(e19,"A") -> erro.
406-      ( e19, "B" ) : CriarTransicao(e19,"B") -> erro.
```

```
407-      ( e19, "C" ) : CriarTransicao(e19,"C") -> erro.
408-      ( e19, "D" ) : CriarTransicao(e19,"D") -> erro.
409-      ( e19, "E" ) : CriarTransicao(e19,"E") -> erro.
410-      ( e19, "F" ) : CriarTransicao(e19,"F") -> erro.
411-      ( e19, "G" ) : CriarTransicao(e19,"G") -> erro.
412-      ( e19, "H" ) : CriarTransicao(e19,"H") -> erro.
413-      ( e19, "I" ) : CriarTransicao(e19,"I") -> erro.
414-      ( e19, "J" ) : CriarTransicao(e19,"J") -> erro.
415-      ( e19, "K" ) : CriarTransicao(e19,"K") -> erro.
416-      ( e19, "L" ) : CriarTransicao(e19,"L") -> erro.
417-      ( e19, "M" ) : CriarTransicao(e19,"M") -> erro.
418-      ( e19, "N" ) : CriarTransicao(e19,"N") -> erro.
419-      ( e19, "O" ) : CriarTransicao(e19,"O") -> erro.
420-      ( e19, "P" ) : CriarTransicao(e19,"P") -> erro.
421-      ( e19, "Q" ) : CriarTransicao(e19,"Q") -> erro.
422-      ( e19, "R" ) : CriarTransicao(e19,"R") -> erro.
423-      ( e19, "S" ) : CriarTransicao(e19,"S") -> erro.
424-      ( e19, "T" ) : CriarTransicao(e19,"T") -> erro.
425-      ( e19, "W" ) : CriarTransicao(e19,"W") -> erro.
426-      ( e19, "U" ) : CriarTransicao(e19,"U") -> erro.
427-      ( e19, "V" ) : CriarTransicao(e19,"V") -> erro.
428-      ( e19, "Y" ) : CriarTransicao(e19,"Y") -> erro.
429-      ( e19, "X" ) : CriarTransicao(e19,"X") -> erro.
430-      ( e19, "Z" ) : CriarTransicao(e19,"Z") -> erro.
431-
432-     % Declaracao de Expressoes Regulares
433-
434-      ( e1, "") -> e7. %Inicio da String
435-      ( e7, qualquer - "" ) -> e7.
436-      ( e7, "") -> e8. %Termino da String
437-      ( ^*, e8 ) -> ( ^, *, stringTK ).
438-
439-      ( e1, """) -> e11. %Inicio da String
440-      ( e11, qualquer - """) ->e11.
441-      ( e11, """) -> e12. %termino da String
442-      ( ^*, e12) -> ( ^, *, stringTK ).
443-
444-     %  ( e1, "letras+_" ) -> e1.
445-
446-     %Primeira Letra ou Underline
447-
448-      ( e1, "_" ) : CriarTransicao(e1,"_") -> erro.
449-     % ( e1, "a" ) -> erro. Tratado na palavra reservada atomo
450-      ( e1, "b" ) : CriarTransicao(e1,"b") -> erro.
451-      ( e1, "c" ) : CriarTransicao(e1,"c") -> erro.
452-      ( e1, "d" ) : CriarTransicao(e1,"d") -> erro.
453-     % ( e1, "e" ) -> e9. Tratado na palavra reservada entrada/estado
454-     % ( e1, "f" ) -> e9. Tratado na palavra reservada funcoes
455-      ( e1, "g" ) : CriarTransicao(e1,"g") -> erro.
456-      ( e1, "h" ) : CriarTransicao(e1,"h") -> erro.
457-      ( e1, "i" ) : CriarTransicao(e1,"i") -> erro.
458-      ( e1, "j" ) : CriarTransicao(e1,"j") -> erro.
459-      ( e1, "k" ) : CriarTransicao(e1,"k") -> erro.
460-      ( e1, "l" ) : CriarTransicao(e1,"l") -> erro.
461-      ( e1, "m" ) : CriarTransicao(e1,"m") -> erro.
462-      ( e1, "n" ) : CriarTransicao(e1,"n") -> erro.
463-      ( e1, "o" ) : CriarTransicao(e1,"o") -> erro.
464-     % ( e1, "p" ) -> e9. Tratado na palavra reservada producoes
465-     % ( e1, "q" ) -> e9. Tratado na palavra reservada qualquer
```

```
466-      ( e1, "r" ) : CriarTransicao(e1,"r") -> erro.
467-     % ( e1, "s" ) -> e9. Tratado na palavra reservada submaquina
468-      ( e1, "t" ) : CriarTransicao(e1,"t") -> erro.
469-      ( e1, "u" ) : CriarTransicao(e1,"u") -> erro.
470-      ( e1, "w" ) : CriarTransicao(e1,"w") -> erro.
471-      ( e1, "v" ) : CriarTransicao(e1,"v") -> erro.
472-      ( e1, "x" ) : CriarTransicao(e1,"x") -> erro.
473-      ( e1, "y" ) : CriarTransicao(e1,"y") -> erro.
474-      ( e1, "z" ) : CriarTransicao(e1,"z") -> erro.
475-      ( e1, "A" ) : CriarTransicao(e1,"A") -> erro.
476-      ( e1, "B" ) : CriarTransicao(e1,"B") -> erro.
477-      ( e1, "C" ) : CriarTransicao(e1,"C") -> erro.
478-      ( e1, "D" ) : CriarTransicao(e1,"D") -> erro.
479-      ( e1, "E" ) : CriarTransicao(e1,"E") -> erro.
480-      ( e1, "F" ) : CriarTransicao(e1,"F") -> erro.
481-      ( e1, "G" ) : CriarTransicao(e1,"G") -> erro.
482-      ( e1, "H" ) : CriarTransicao(e1,"H") -> erro.
483-      ( e1, "I" ) : CriarTransicao(e1,"I") -> erro.
484-      ( e1, "J" ) : CriarTransicao(e1,"J") -> erro.
485-      ( e1, "K" ) : CriarTransicao(e1,"K") -> erro.
486-      ( e1, "L" ) : CriarTransicao(e1,"L") -> erro.
487-      ( e1, "M" ) : CriarTransicao(e1,"M") -> erro.
488-      ( e1, "N" ) : CriarTransicao(e1,"N") -> erro.
489-      ( e1, "O" ) : CriarTransicao(e1,"O") -> erro.
490-      ( e1, "P" ) : CriarTransicao(e1,"P") -> erro.
491-      ( e1, "Q" ) : CriarTransicao(e1,"Q") -> erro.
492-      ( e1, "R" ) : CriarTransicao(e1,"R") -> erro.
493-      ( e1, "S" ) : CriarTransicao(e1,"S") -> erro.
494-      ( e1, "T" ) : CriarTransicao(e1,"T") -> erro.
495-      ( e1, "W" ) : CriarTransicao(e1,"W") -> erro.
496-      ( e1, "U" ) : CriarTransicao(e1,"U") -> erro.
497-      ( e1, "V" ) : CriarTransicao(e1,"V") -> erro.
498-      ( e1, "Y" ) : CriarTransicao(e1,"Y") -> erro.
499-      ( e1, "X" ) : CriarTransicao(e1,"X") -> erro.
500-      ( e1, "Z" ) : CriarTransicao(e1,"Z") -> erro.
501-
502-
503-     % Declaracao dos numeros inteiros
504-
505-      ( e1, "0" ) -> e14.
506-      ( e1, "1" ) -> e14.
507-      ( e1, "2" ) -> e14.
508-      ( e1, "3" ) -> e14.
509-      ( e1, "4" ) -> e14.
510-      ( e1, "5" ) -> e14.
511-      ( e1, "6" ) -> e14.
512-      ( e1, "7" ) -> e14.
513-      ( e1, "8" ) -> e14.
514-      ( e1, "9" ) -> e14.
515-
516-      ( e14, "0" ) -> e14.
517-      ( e14, "1" ) -> e14.
518-      ( e14, "2" ) -> e14.
519-      ( e14, "3" ) -> e14.
520-      ( e14, "4" ) -> e14.
521-      ( e14, "5" ) -> e14.
522-      ( e14, "6" ) -> e14.
523-      ( e14, "7" ) -> e14.
524-      ( e14, "8" ) -> e14.
```

```
525-       ( e14, "9" ) -> e14.
526-
527-       ( ^*, e14 ) -> ( ^, *, numTK). %Reconhece um numero inteiro
528-
529-      % Declaracao dos comentarios
530-
531-       ( e1, "%" ) -> e10.
532-       ( e10, qualquer - 13) -> e10.
533-       ( e10, 13) -> e1.
534-
535-      % Caracteres ignorados
536-
537-       ( e1, 10) -> e1. %Line Feed
538-       ( e1, 13) -> e1. %Carriage Return
539-       ( e1, 32) -> e1. %Espaco
540-       ( e1, 09) -> e1. %Tab
541-
542-      % Indica se a maquina esta no modo de verificacao
543-       ( Verificacao, emptyTK) -> Verificacao.
544-
545-      % Indica se a maquina esta no modo de coleta simples
546-       ( ColetaSimples, emptyTK) -> ColetaSimples.
547-
548-      % Indica o estado que empilhou o atomo de retorno
549-      % antes do desvio para a maquina chamadora.
550-       ( EstadoRetorno, emptyTK) -> erro.
551-
552-      funcoes
553-
554-      CriarTransicao ( estado EstadoCorrente, atomo AtomoCorrente) =
555-      {
556-       estado *NovoEstado,
557-       atomo  Coletando
558-       :
559-
560-       ?[(ColetaSimples, Coletando) -> ColetaSimples]
561-       -[( EstadoCorrente, AtomoCorrente) : CriarTransicao(EstadoCorrente,AtomoCorrente) -> (erro,
    Coletando) ]
562-       +[( EstadoCorrente, AtomoCorrente) -> (NovoEstado, Coletando) ]
563-       +[( NovoEstado, "_" ) : CriarTransicao(NovoEstado,"_") -> (erro, Coletando) ].
564-       +[( NovoEstado, "a" ) : CriarTransicao(NovoEstado,"a") -> (erro, Coletando) ].
565-       +[( NovoEstado, "b" ) : CriarTransicao(NovoEstado,"b") -> (erro, Coletando) ].
566-       +[( NovoEstado, "c" ) : CriarTransicao(NovoEstado,"c") -> (erro, Coletando) ].
567-       +[( NovoEstado, "d" ) : CriarTransicao(NovoEstado,"d") -> (erro, Coletando) ].
568-       +[( NovoEstado, "e" ) : CriarTransicao(NovoEstado,"e") -> (erro, Coletando) ].
569-       +[( NovoEstado, "f" ) : CriarTransicao(NovoEstado,"f") -> (erro, Coletando) ].
570-       +[( NovoEstado, "g" ) : CriarTransicao(NovoEstado,"g") -> (erro, Coletando) ].
571-       +[( NovoEstado, "h" ) : CriarTransicao(NovoEstado,"h") -> (erro, Coletando) ].
572-       +[( NovoEstado, "i" ) : CriarTransicao(NovoEstado,"i") -> (erro, Coletando) ].
573-       +[( NovoEstado, "j" ) : CriarTransicao(NovoEstado,"j") -> (erro, Coletando) ].
574-       +[( NovoEstado, "k" ) : CriarTransicao(NovoEstado,"k") -> (erro, Coletando) ].
575-       +[( NovoEstado, "l" ) : CriarTransicao(NovoEstado,"l") -> (erro, Coletando) ].
576-       +[( NovoEstado, "m" ) : CriarTransicao(NovoEstado,"m") -> (erro, Coletando) ].
577-       +[( NovoEstado, "n" ) : CriarTransicao(NovoEstado,"n") -> (erro, Coletando) ].
578-       +[( NovoEstado, "o" ) : CriarTransicao(NovoEstado,"o") -> (erro, Coletando) ].
579-       +[( NovoEstado, "p" ) : CriarTransicao(NovoEstado,"p") -> (erro, Coletando) ].
580-       +[( NovoEstado, "q" ) : CriarTransicao(NovoEstado,"q") -> (erro, Coletando) ].
581-       +[( NovoEstado, "r" ) : CriarTransicao(NovoEstado,"r") -> (erro, Coletando) ].
582-       +[( NovoEstado, "s" ) : CriarTransicao(NovoEstado,"s") -> (erro, Coletando) ].
```

```
583-      +[( NovoEstado, "t" ) : CriarTransicao(NovoEstado,"t") -> (erro, Coletando) ].
584-      +[( NovoEstado, "u" ) : CriarTransicao(NovoEstado,"u") -> (erro, Coletando) ].
585-      +[( NovoEstado, "w" ) : CriarTransicao(NovoEstado,"w") -> (erro, Coletando) ].
586-      +[( NovoEstado, "v" ) : CriarTransicao(NovoEstado,"v") -> (erro, Coletando) ].
587-      +[( NovoEstado, "x" ) : CriarTransicao(NovoEstado,"x") -> (erro, Coletando) ].
588-      +[( NovoEstado, "y" ) : CriarTransicao(NovoEstado,"y") -> (erro, Coletando) ].
589-      +[( NovoEstado, "z" ) : CriarTransicao(NovoEstado,"z") -> (erro, Coletando) ].
590-      +[( NovoEstado, "1" ) : CriarTransicao(NovoEstado,"1") -> (erro, Coletando) ].
591-      +[( NovoEstado, "2" ) : CriarTransicao(NovoEstado,"2") -> (erro, Coletando) ].
592-      +[( NovoEstado, "3" ) : CriarTransicao(NovoEstado,"3") -> (erro, Coletando) ].
593-      +[( NovoEstado, "4" ) : CriarTransicao(NovoEstado,"4") -> (erro, Coletando) ].
594-      +[( NovoEstado, "5" ) : CriarTransicao(NovoEstado,"5") -> (erro, Coletando) ].
595-      +[( NovoEstado, "6" ) : CriarTransicao(NovoEstado,"6") -> (erro, Coletando) ].
596-      +[( NovoEstado, "7" ) : CriarTransicao(NovoEstado,"7") -> (erro, Coletando) ].
597-      +[( NovoEstado, "8" ) : CriarTransicao(NovoEstado,"8") -> (erro, Coletando) ].
598-      +[( NovoEstado, "9" ) : CriarTransicao(NovoEstado,"9") -> (erro, Coletando) ].
599-      +[( NovoEstado, "0" ) : CriarTransicao(NovoEstado,"0") -> (erro, Coletando) ].
600-      +[( NovoEstado, "A" ) : CriarTransicao(NovoEstado,"A") -> (erro, Coletando) ].
601-      +[( NovoEstado, "B" ) : CriarTransicao(NovoEstado,"B") -> (erro, Coletando) ].
602-      +[( NovoEstado, "C" ) : CriarTransicao(NovoEstado,"C") -> (erro, Coletando) ].
603-      +[( NovoEstado, "D" ) : CriarTransicao(NovoEstado,"D") -> (erro, Coletando) ].
604-      +[( NovoEstado, "E" ) : CriarTransicao(NovoEstado,"E") -> (erro, Coletando) ].
605-      +[( NovoEstado, "F" ) : CriarTransicao(NovoEstado,"F") -> (erro, Coletando) ].
606-      +[( NovoEstado, "G" ) : CriarTransicao(NovoEstado,"G") -> (erro, Coletando) ].
607-      +[( NovoEstado, "H" ) : CriarTransicao(NovoEstado,"H") -> (erro, Coletando) ].
608-      +[( NovoEstado, "I" ) : CriarTransicao(NovoEstado,"I") -> (erro, Coletando) ].
609-      +[( NovoEstado, "J" ) : CriarTransicao(NovoEstado,"J") -> (erro, Coletando) ].
610-      +[( NovoEstado, "K" ) : CriarTransicao(NovoEstado,"K") -> (erro, Coletando) ].
611-      +[( NovoEstado, "L" ) : CriarTransicao(NovoEstado,"L") -> (erro, Coletando) ].
612-      +[( NovoEstado, "M" ) : CriarTransicao(NovoEstado,"M") -> (erro, Coletando) ].
613-      +[( NovoEstado, "N" ) : CriarTransicao(NovoEstado,"N") -> (erro, Coletando) ].
614-      +[( NovoEstado, "O" ) : CriarTransicao(NovoEstado,"O") -> (erro, Coletando) ].
615-      +[( NovoEstado, "P" ) : CriarTransicao(NovoEstado,"P") -> (erro, Coletando) ].
616-      +[( NovoEstado, "Q" ) : CriarTransicao(NovoEstado,"Q") -> (erro, Coletando) ].
617-      +[( NovoEstado, "R" ) : CriarTransicao(NovoEstado,"R") -> (erro, Coletando) ].
618-      +[( NovoEstado, "S" ) : CriarTransicao(NovoEstado,"S") -> (erro, Coletando) ].
619-      +[( NovoEstado, "T" ) : CriarTransicao(NovoEstado,"T") -> (erro, Coletando) ].
620-      +[( NovoEstado, "W" ) : CriarTransicao(NovoEstado,"W") -> (erro, Coletando) ].
621-      +[( NovoEstado, "U" ) : CriarTransicao(NovoEstado,"U") -> (erro, Coletando) ].
622-      +[( NovoEstado, "V" ) : CriarTransicao(NovoEstado,"V") -> (erro, Coletando) ].
623-      +[( NovoEstado, "Y" ) : CriarTransicao(NovoEstado,"Y") -> (erro, Coletando) ].
624-      +[( NovoEstado, "X" ) : CriarTransicao(NovoEstado,"X") -> (erro, Coletando) ].
625-      +[( NovoEstado, "Z" ) : CriarTransicao(NovoEstado,"Z") -> (erro, Coletando) ].
626-      +[( NovoEstado, emptyTK) : DefinirIdDesconhecido(NovoEstado) -> (erro, Coletando) ].
627-     }
628-
629-     DefinirIdDesconhecido(estado EstadoCorrente) =
630-     {
631-      estado *NovoEstado,
632-      atomo  Coletando
633-      :
634-
635-      ?[(ColetaSimples, Coletando) -> ColetaSimples]
636-      -[(EstadoCorrente, emptyTK) : DefinirIdDesconhecido(EstadoCorrente) -> (EstadoCorrente,
    Coletando) ]
637-      +[(EstadoCorrente, emptyTK) -> (NovoEstado, Coletando)]
638-      +[(^*, NovoEstado, Coletando) -> (^, *, idTK) : DefinirEstadoRetorno(NovoEstado)]
639-     }
640-
```

```
641-      AbrirBloco =
642-      {
643-        estado *NovoEstado1,
644-        estado *NovoEstado2
645-        :
646-
647-        -[(e0, emptyTK) -> e1]
648-        +[(e0, emptyTK) -> NovoEstado1]
649-        +[(NovoEstado1, emptyTK) -> NovoEstado2]
650-        +[(NovoEstado2, emptyTK) -> e1)]
651-
652-        +[(NovoEstado1, "_") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "_") ->
      NovoEstado2]
653-        +[(NovoEstado1, "a") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "a") ->
      NovoEstado2]
654-        +[(NovoEstado1, "b") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "b") ->
      NovoEstado2]
655-        +[(NovoEstado1, "c") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "c") ->
      NovoEstado2]
656-        +[(NovoEstado1, "d") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "d") ->
      NovoEstado2]
657-        +[(NovoEstado1, "e") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "e") ->
      NovoEstado2]
658-        +[(NovoEstado1, "f") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "f") ->
      NovoEstado2]
659-        +[(NovoEstado1, "g") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "g") ->
      NovoEstado2]
660-        +[(NovoEstado1, "h") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "h") ->
      NovoEstado2]
661-        +[(NovoEstado1, "i") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "i") -> NovoEstado2]
662-        +[(NovoEstado1, "j") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "j") -> NovoEstado2]
663-        +[(NovoEstado1, "k") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "k") ->
      NovoEstado2]
664-        +[(NovoEstado1, "l") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "l") -> NovoEstado2]
665-        +[(NovoEstado1, "m") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "m") ->
      NovoEstado2]
666-        +[(NovoEstado1, "n") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "n") ->
      NovoEstado2]
667-        +[(NovoEstado1, "o") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "o") ->
      NovoEstado2]
668-        +[(NovoEstado1, "p") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "p") ->
      NovoEstado2]
669-        +[(NovoEstado1, "q") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "q") ->
      NovoEstado2]
670-        +[(NovoEstado1, "r") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "r") ->
      NovoEstado2]
671-        +[(NovoEstado1, "s") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "s") ->
      NovoEstado2]
672-        +[(NovoEstado1, "t") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "t") -> NovoEstado2]
673-        +[(NovoEstado1, "u") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "u") ->
      NovoEstado2]
674-        +[(NovoEstado1, "w") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "w") ->
      NovoEstado2]
675-        +[(NovoEstado1, "v") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "v") ->
      NovoEstado2]
676-        +[(NovoEstado1, "x") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "x") ->
      NovoEstado2]
677-        +[(NovoEstado1, "y") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "y") ->
      NovoEstado2]
```

```
678-        +[(NovoEstado1, "z") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "z") ->
    NovoEstado2]
679-        +[(NovoEstado1, "A") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "A") ->
    NovoEstado2]
680-        +[(NovoEstado1, "B") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "B") ->
    NovoEstado2]
681-        +[(NovoEstado1, "C") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "C") ->
    NovoEstado2]
682-        +[(NovoEstado1, "D") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "D") ->
    NovoEstado2]
683-        +[(NovoEstado1, "E") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "E") ->
    NovoEstado2]
684-        +[(NovoEstado1, "F") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "F") ->
    NovoEstado2]
685-        +[(NovoEstado1, "G") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "G") ->
    NovoEstado2]
686-        +[(NovoEstado1, "H") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "H") ->
    NovoEstado2]
687-        +[(NovoEstado1, "I") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "I") ->
    NovoEstado2]
688-        +[(NovoEstado1, "J") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "J") ->
    NovoEstado2]
689-        +[(NovoEstado1, "K") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "K") ->
    NovoEstado2]
690-        +[(NovoEstado1, "L") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "L") ->
    NovoEstado2]
691-        +[(NovoEstado1, "M") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "M") ->
    NovoEstado2]
692-        +[(NovoEstado1, "N") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "N") ->
    NovoEstado2]
693-        +[(NovoEstado1, "O") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "O") ->
    NovoEstado2]
694-        +[(NovoEstado1, "P") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "P") ->
    NovoEstado2]
695-        +[(NovoEstado1, "Q") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "Q") ->
    NovoEstado2]
696-        +[(NovoEstado1, "R") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "R") ->
    NovoEstado2]
697-        +[(NovoEstado1, "S") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "S") ->
    NovoEstado2]
698-        +[(NovoEstado1, "T") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "T") ->
    NovoEstado2]
699-        +[(NovoEstado1, "U") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "U") ->
    NovoEstado2]
700-        +[(NovoEstado1, "W") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "W") ->
    NovoEstado2]
701-        +[(NovoEstado1, "V") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "V") ->
    NovoEstado2]
702-        +[(NovoEstado1, "X") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "X") ->
    NovoEstado2]
703-        +[(NovoEstado1, "Y") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "Y") ->
    NovoEstado2]
704-        +[(NovoEstado1, "Z") : CriarTransicaoBloco(Novoestado1, NovoEstado2, "Z") ->
    NovoEstado2]
705-    }
706-
707-    FecharBloco =
708-    {
709-      estado NovoEstado1,
```

```
710-        estado NovoEstado2
711-        :
712-
713-        -[(e0, emptyTK) -> NovoEstado1]
714-        -[(NovoEstado1, emptyTK) -> NovoEstado2]
715-        -[(NovoEstado2, emptyTK) -> e1]
716-        +[(e0, emptyTK) -> e1]
717-       }
718-
719-       CriarTransicaoBloco ( estado EstadoCorrente, estado ProximoEstado, atomo AtomoCorrente) =
720-       {
721-        estado *NovoEstado1,
722-        estado *NovoEstado2,
723-        atomo  Coletando
724-        :
725-
726-        ?[( ColetaSimples, Coletando) -> ColetaSimples]
727-        -[( EstadoCorrente, AtomoCorrente) : CriarTransicaoBloco(EstadoCorrente, ProximoEstado,
    AtomoCorrente) -> (ProximoEstado, Coletando) ]
728-        +[( EstadoCorrente, AtomoCorrente) -> (NovoEstado1, Coletando]
729-        +[( NovoEstado1, Coletando) : DefinirIdDesconhecidoBloco(NovoEstado1,NovoEstado2)->
    NovoEstado2]
730-        +[( NovoEstado2, Coletando) -> (ProximoEstado, AtomoCorrente)]
731-
732-        +[( NovoEstado1, "_" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2, "_") ->
    (NovoEstado2, Coletando) ].
733-        +[( NovoEstado1, "a" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"a") ->
    (NovoEstado2, Coletando) ].
734-        +[( NovoEstado1, "b" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"b") ->
    (NovoEstado2, Coletando) ].
735-        +[( NovoEstado1, "c" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"c") ->
    (NovoEstado2, Coletando) ].
736-        +[( NovoEstado1, "d" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"d") ->
    (NovoEstado2, Coletando) ].
737-        +[( NovoEstado1, "e" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"e") ->
    (NovoEstado2, Coletando) ].
738-        +[( NovoEstado1, "f" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"f") ->
    (NovoEstado2, Coletando) ].
739-        +[( NovoEstado1, "g" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"g") ->
    (NovoEstado2, Coletando) ].
740-        +[( NovoEstado1, "h" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"h") ->
    (NovoEstado2, Coletando) ].
741-        +[( NovoEstado1, "i" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"i") ->
    (NovoEstado2, Coletando) ].
742-        +[( NovoEstado1, "j" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"j") ->
    (NovoEstado2, Coletando) ].
743-        +[( NovoEstado1, "k" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"k") ->
    (NovoEstado2, Coletando) ].
744-        +[( NovoEstado1, "l" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"l") ->
    (NovoEstado2, Coletando) ].
745-        +[( NovoEstado1, "m" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"m") ->
    (NovoEstado2, Coletando) ].
746-        +[( NovoEstado1, "n" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"n") ->
    (NovoEstado2, Coletando) ].
747-        +[( NovoEstado1, "o" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"o") ->
    (NovoEstado2, Coletando) ].
748-        +[( NovoEstado1, "p" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"p") ->
    (NovoEstado2, Coletando) ].
```

```
749-        +[( NovoEstado1, "q" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"q") ->
    (NovoEstado2, Coletando) ].
750-        +[( NovoEstado1, "r" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"r") ->
    (NovoEstado2, Coletando) ].
751-        +[( NovoEstado1, "s" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"s") ->
    (NovoEstado2, Coletando) ].
752-        +[( NovoEstado1, "t" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"t") ->
    (NovoEstado2, Coletando) ].
753-        +[( NovoEstado1, "u" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"u") ->
    (NovoEstado2, Coletando) ].
754-        +[( NovoEstado1, "w" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"w") ->
    (NovoEstado2, Coletando) ].
755-        +[( NovoEstado1, "v" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"v") ->
    (NovoEstado2, Coletando) ].
756-        +[( NovoEstado1, "x" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"x") ->
    (NovoEstado2, Coletando) ].
757-        +[( NovoEstado1, "y" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"y") ->
    (NovoEstado2, Coletando) ].
758-        +[( NovoEstado1, "z" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"z") ->
    (NovoEstado2, Coletando) ].
759-        +[( NovoEstado1, "1" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"1") ->
    (NovoEstado2, Coletando) ].
760-        +[( NovoEstado1, "2" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"2") ->
    (NovoEstado2, Coletando) ].
761-        +[( NovoEstado1, "3" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"3") ->
    (NovoEstado2, Coletando) ].
762-        +[( NovoEstado1, "4" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"4") ->
    (NovoEstado2, Coletando) ].
763-        +[( NovoEstado1, "5" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"5") ->
    (NovoEstado2, Coletando) ].
764-        +[( NovoEstado1, "6" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"6") ->
    (NovoEstado2, Coletando) ].
765-        +[( NovoEstado1, "7" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"7") ->
    (NovoEstado2, Coletando) ].
766-        +[( NovoEstado1, "8" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"8") ->
    (NovoEstado2, Coletando) ].
767-        +[( NovoEstado1, "9" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"9") ->
    (NovoEstado2, Coletando) ].
768-        +[( NovoEstado1, "0" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"0") ->
    (NovoEstado2, Coletando) ].
769-        +[( NovoEstado1, "A" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"A") ->
    (NovoEstado2, Coletando) ].
770-        +[( NovoEstado1, "B" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"B") ->
    (NovoEstado2, Coletando) ].
771-        +[( NovoEstado1, "C" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"C") ->
    (NovoEstado2, Coletando) ].
772-        +[( NovoEstado1, "D" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"D") ->
    (NovoEstado2, Coletando) ].
773-        +[( NovoEstado1, "E" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"E") ->
    (NovoEstado2, Coletando) ].
774-        +[( NovoEstado1, "F" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"F") ->
    (NovoEstado2, Coletando) ].
775-        +[( NovoEstado1, "G" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"G") ->
    (NovoEstado2, Coletando) ].
776-        +[( NovoEstado1, "H" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"H") ->
    (NovoEstado2, Coletando) ].
777-        +[( NovoEstado1, "I" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"I") ->
    (NovoEstado2, Coletando) ].
```

```
778-        +[( NovoEstado1, "J" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"J") ->
    (NovoEstado2, Coletando) ].
779-        +[( NovoEstado1, "K" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"K") ->
    (NovoEstado2, Coletando) ].
780-        +[( NovoEstado1, "L" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"L") ->
    (NovoEstado2, Coletando) ].
781-        +[( NovoEstado1, "M" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"M") ->
    (NovoEstado2, Coletando) ].
782-        +[( NovoEstado1, "N" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"N") ->
    (NovoEstado2, Coletando) ].
783-        +[( NovoEstado1, "O" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"O") ->
    (NovoEstado2, Coletando) ].
784-        +[( NovoEstado1, "P" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"P") ->
    (NovoEstado2, Coletando) ].
785-        +[( NovoEstado1, "Q" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"Q") ->
    (NovoEstado2, Coletando) ].
786-        +[( NovoEstado1, "R" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"R") ->
    (NovoEstado2, Coletando) ].
787-        +[( NovoEstado1, "S" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"S") ->
    (NovoEstado2, Coletando) ].
788-        +[( NovoEstado1, "T" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"T") ->
    (NovoEstado2, Coletando) ].
789-        +[( NovoEstado1, "W" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"W") ->
    (NovoEstado2, Coletando) ].
790-        +[( NovoEstado1, "U" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"U") ->
    (NovoEstado2, Coletando) ].
791-        +[( NovoEstado1, "V" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"V") ->
    (NovoEstado2, Coletando) ].
792-        +[( NovoEstado1, "Y" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"Y") ->
    (NovoEstado2, Coletando) ].
793-        +[( NovoEstado1, "X" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"X") ->
    (NovoEstado2, Coletando) ].
794-        +[( NovoEstado1, "Z" ) : CriarTransicaoBloco(NovoEstado1, NovoEstado2,"Z") ->
    (NovoEstado2, Coletando) ].
795-        +[( NovoEstado1, emptyTK) : DefinirIdDesconhecidoBloco(NovoEstado1, NovoEstado2) ->
    (NovoEstado2, Coletando) ].
796-      }
797-
798-      DefinirEstadoRetorno( estado NovoEstado) =
799-      {
800-        estado EstadoAnterior:
801-
802-        -[(EstadoRetorno, emptyTK)  -> EstadoAnterior]
803-        +[(EstadoRetorno, emptyTK)  -> NovoEstado]
804-      }
805-
806-      DefinirIdDesconhecidoBloco(estado EstadoCorrente, estado ProximoEstado) =
807-      {
808-        estado *NovoEstado,
809-        atomo  Coletando
810-        :
811-
812-        ?[(ColetaSimples, Coletando) -> ColetaSimples]
813-        -[(EstadoCorrente, emptyTK) : DefinirIdDesconhecidoBloco(EstadoCorrente,ProximoEstado) -
    > (ProximoEstado, Coletando) ]
814-        +[(EstadoCorrente, emptyTK) : DefinirIdRedefinido(EstadoCorrente, NovoEstado) ->
    (NovoEstado, Coletando)]
815-        +[(^*, NovoEstado, Coletando) -> (^, *, idTK) : DefinirEstadoRetorno(NovoEstado) ]
816-      }
```

```
817-
818-    DefinirIdRedefinido(estado EstadoCorrente, estado NovoEstado) =
819-    {
820-     estado *NovoEstado1,
821-     atomo  Coletando,
822-     atomo  Verificando
823-     :
824-
825-     ?[(ColetaSimples, Coletando) -> ColetaSimples]
826-     ?[(Verificacao,   Verificando) -> Verificacao]
827-
828-     -[(EstadoCorrente, emptyTK) : DefinirIdRedefinido(EstadoCorrente) -> NovoEstado]
829-
830-     +[(EstadoCorrente, Verificando) -> NovoEstado]
831-
832-     +[(EstadoCorrente, Coletando) -> NovoEstado1]
833-     +[(^*, NovoEstado1, Coletando) -> (^, *, idRedefinedTK]
834-    }
835-
836-    DefinirModoColeta =
837-    {
838-     -[( Verificacao,   emptyTK) -> Verificacao]
839-     +[( ColetaSimples, emptyTK) -> ColetaSimples]
840-    }
841-
842-
843-    DefinirModoVerificacao =
844-    {
845-     +[( Verificacao,   emptyTK) -> Verificacao]
846-     -[( ColetaSimples, emptyTK) -> ColetaSimples]
847-    }
848-
849-    DefinirModoColetaVerificacao =
850-    {
851-     +[( Verificacao,   emptyTK) -> Verificacao]
852-     +[( ColetaSimples, emptyTK) -> ColetaSimples]
853-    }
854-
855-    %Funcao adaptativa que define o ultimo identificador
856-    %reconhecido como do tipo atomo
857-    DefinirIdAtomo =
858-    {
859-      estado UltimoEstado :
860-
861-      -[(EstadoRetorno,emptyTK) -> UltimoEstado]
862-
863-      -[(^*, UltimoEstado) -> (^, *, idTK)]
864-      +[(^*, UltimoEstado) -> (^, *, idTokenTK)]
865-    }
866-
867-    %Funcao adaptativa que define o ultimo identificador
868-    %reconhecido como do tipo funcao
869-    DefinirIdFuncao =
870-    {
871-      estado UltimoEstado :
872-
873-      -[(EstadoRetorno,emptyTK) -> UltimoEstado]
874-
875-      -[(^*, UltimoEstado) -> (^, *, idTK)]
```

```
876-        +[(^*, UltimoEstado) -> (^, *, idFunctionTK)]
877-      }
878-
879-      %Funcao adaptativa que define o ultimo identificador
880-      %reconhecido como do tipo estado
881-      DefinirIdEstado =
882-      {
883-        estado UltimoEstado :
884-
885-        -[(EstadoRetorno,emptyTK) -> UltimoEstado]
886-
887-        -[(^*, UltimoEstado) -> (^, *, idTK)]
888-        +[(^*, UltimoEstado) -> (^, *, idStateTK)]
889-      }
890-
891-      %Funcao adaptativa que define o ultimo identificador
892-      %reconhecido como do tipo sub-maquina
893-      DefinirIdSubMaquina =
894-      {
895-        estado UltimoEstado :
896-
897-        -[(EstadoRetorno,emptyTK) -> UltimoEstado]
898-
899-        -[(^*, UltimoEstado) -> (^, *, idTK)]
900-        +[(^*, UltimoEstado) -> (^, *, idAutomatoTK)]
901-      }
```

Listagem 37 - A sub-máquina Scan300 - Arquivo Scan300.rsm

Anexo II – Módulos em Pascal do Exemplo 1

```
1-  Unit PalindromeImpar;
2-
3-
4-  Interface
5-  uses RSWObjects;
6-
7-
8-
9-  TYPE
10-
11-   aPalindromeImpar= class ( aSubMaquina )
12-
13-   public
14-     PROCEDURE DefinirConfiguracaoInicial;
15-     override;
16-   end;
17-
18-
19- Implementation
20- uses
    RSWErros,Support,RSWSemantica,RSWCompilador,RSWAcaoAdaptativa,RSWFuncao,SysUtils
21- ;
22-
23-
24-
25- TYPE
26-   {Declaracao Do Estado e1}
27-
28-   ae1= class ( aEstado )
29-
30-   public
31-     PROCEDURE DefinirConfiguracaoInicial;
32-     override;
33-   end;
34-
35-   {Declaracao Do Estado e2}
36-
37-   ae2= class ( aEstado )
38-
39-   public
40-     PROCEDURE DefinirConfiguracaoInicial;
41-     override;
42-   end;
43-
44-   {Declaracao Do Estado e3}
45-
46-   ae3= class ( aEstado )
47-
48-   public
49-     PROCEDURE DefinirConfiguracaoInicial;
50-     override;
51-   end;
52-
53-   {Declaracao Do Estado e4}
```

```
54-
55-    ae4= class ( aEstado )
56-
57-    public
58-       PROCEDURE DefinirConfiguracaoInicial;
59-       override;
60-    end;
61-
62-    {Declaracao da Funcao a}
63-
64-    aa= class ( aFuncao )
65-
66-    public
67-       PROCEDURE DefinirConfiguracaoInicial;
68-       override;
69-    end;
70-
71-    {Implementacao Do Estado e1}
72-    PROCEDURE ae1.DefinirConfiguracaoInicial;
73-
74- VAR
75-    AcaoPre : aAcaoAdaptativa;
76-    AcaoPos : aAcaoAdaptativa;
77- begin
78-    inherited DefinirConfiguracaoInicial;
79-    AcaoPos := aAcaoAdaptativa.CreateFunction(
80-    (Self.Owner as aSubMaquina).ObterFuncaoNome( 'a' ));
81-    AcaoPos.AdicionarParametroEstado((Self.Owner as aSubMaquina).ObterEstadoNome( 'e2' ));
82-    AcaoPos.AdicionarParametroEstado((Self.Owner as aSubMaquina).ObterEstadoNome( 'e3' ));
83-    AcaoPos.AdicionarParametroEstado((Self.Owner as aSubMaquina).ObterEstadoNome( 'e1' ));
84-    DefinirTransicao(40, (Self.Owner as aSubMaquina).ObterEstadoNome( 'e2' ), ORD(emptyTK), nil,
85-    nil, nil, AcaoPos);
86-    DefinirTransicao(97, (Self.Owner as aSubMaquina).ObterEstadoNome( 'e4' ), ORD(emptyTK), nil,
87-    nil, nil, nil);
88- end;
89-
90- {Implementacao Do Estado e2}
91- PROCEDURE ae2.DefinirConfiguracaoInicial;
92- begin
93-    inherited DefinirConfiguracaoInicial;
94-    DefinirTransicao(97, (Self.Owner as aSubMaquina).ObterEstadoNome( 'e3' ), ORD(emptyTK), nil,
95-    nil, nil, nil);
96- end;
97-
98- {Implementacao Do Estado e3}
99- PROCEDURE ae3.DefinirConfiguracaoInicial;
100-       begin
101-          inherited DefinirConfiguracaoInicial;
102-          DefinirTransicao(41, (Self.Owner as aSubMaquina).ObterEstadoNome( 'e4' ),
      ORD(emptyTK), nil,
103-          nil, nil, nil);
104-       end;
105-
106-       {Implementacao Do Estado e4}
107-       PROCEDURE ae4.DefinirConfiguracaoInicial;
108-       begin
109-          inherited DefinirConfiguracaoInicial;
110-          DefinirTransicao(0, nil, ORD(eofTK), nil, nil, nil, nil);
111-       end;
```

```
112-
113-    {Implementacao da Funcao a}
114-    PROCEDURE aa.DefinirConfiguracaoInicial;
115-
116-    VAR
117-      AcaoPre : aAcaoAdaptativa;
118-      AcaoPos : aAcaoAdaptativa;
119-    begin
120-      NumeroParametros := 3;
121-      NumeroVariaveis := 0;
122-      NumeroGeradores := 2;
123-      inherited DefinirConfiguracaoInicial;
124-      AcaoPos := aAcaoAdaptativa.CreateFunction(
125-      (Self.Owner as aSubMaquina).ObterFuncaoNome( 'a' ));
126-      AcaoPos.AdicionarParametroOffset(0);
127-      AcaoPos.AdicionarParametroOffset(1);
128-      AcaoPos.AdicionarParametroOffset(2);
129-      DefinirAcaoElementar( 'e' , nil, 40, nil, ORD(emptyTK), nil, nil, nil, AcaoPos, 2, 0);
130-      DefinirAcaoElementar( 'i' , nil, 97, nil, ORD(emptyTK), nil, nil, nil, nil, 3, 4);
131-      DefinirAcaoElementar( 'i' , nil, 41, nil, ORD(emptyTK), nil, nil, nil, nil, 4, 1);
132-      AcaoPos := aAcaoAdaptativa.CreateFunction(
133-      (Self.Owner as aSubMaquina).ObterFuncaoNome( 'a' ));
134-      AcaoPos.AdicionarParametroOffset(3);
135-      AcaoPos.AdicionarParametroOffset(4);
136-      AcaoPos.AdicionarParametroOffset(0);
137-      DefinirAcaoElementar( 'i' , nil, 40, nil, ORD(emptyTK), nil, nil, nil, AcaoPos, 0, 3);
138-      DefinirAcaoElementar( 'i' , nil, 40, nil, ORD(emptyTK), nil, nil, nil, nil, 2, 0);
139-    end;
140-
141-    {Implementacao Da SubMaquina PalindromeImpar}
142-    PROCEDURE aPalindromeImpar.DefinirConfiguracaoInicial;
143-
144-    VAR
145-      EstadoInicial : aEstado;
146-      FuncaoAdaptativa : aFuncao;
147-    begin
148-      GerarImagemAtomo := TRUE;
149-      FNomeMaquinaEntrada :=  'CadeiaEntrada' ;
150-      FMaquinaEntrada := CadeiaEntrada;
151-      if NOT LocalizarEstadoString( 'e1' , EstadoInicial) then begin
152-        EstadoInicial := ae1.CreateName(Self, CallBackProcs,  'e1' );
153-        AdicionarEstado(EstadoInicial);
154-      end;
155-
156-      DefinirEstadoInicial(EstadoInicial);
157-      if NOT LocalizarEstadoString( 'e2' , EstadoInicial) then begin
158-        AdicionarEstado(ae2.CreateName(Self, CallBackProcs,  'e2' ));
159-      end;
160-
161-      if NOT LocalizarEstadoString( 'e3' , EstadoInicial) then begin
162-        AdicionarEstado(ae3.CreateName(Self, CallBackProcs,  'e3' ));
163-      end;
164-
165-      if NOT LocalizarEstadoString( 'e4' , EstadoInicial) then begin
166-        AdicionarEstado(ae4.CreateName(Self, CallBackProcs,  'e4' ));
167-      end;
168-
169-      if NOT LocalizarFuncaoString( 'a' , FuncaoAdaptativa) then begin
170-        AdicionarFuncao(aa.CreateName(Self,  'a' ));
```

```
171-        end;
172-
173-        if LocalizarEstadoString( 'e1' , EstadoInicial) then begin
174-          EstadoInicial.DefinirConfiguracaoInicial;
175-        end else begin
176-          CallBackProcs.AdicionarErro(0,0,0, erInicializacao +  'e1' )end;
177-
178-        if LocalizarEstadoString( 'e2' , EstadoInicial) then begin
179-          EstadoInicial.DefinirConfiguracaoInicial;
180-        end else begin
181-          CallBackProcs.AdicionarErro(0,0,0, erInicializacao +  'e2' )end;
182-
183-        if LocalizarEstadoString( 'e3' , EstadoInicial) then begin
184-          EstadoInicial.DefinirConfiguracaoInicial;
185-        end else begin
186-          CallBackProcs.AdicionarErro(0,0,0, erInicializacao +  'e3' )end;
187-
188-        if LocalizarEstadoString( 'e4' , EstadoInicial) then begin
189-          EstadoInicial.DefinirConfiguracaoInicial;
190-        end else begin
191-          CallBackProcs.AdicionarErro(0,0,0, erInicializacao +  'e4' )end;
192-
193-        if LocalizarFuncaoString( 'a' , FuncaoAdaptativa) then begin
194-          FuncaoAdaptativa.DefinirConfiguracaoInicial;
195-        end else begin
196-          CallBackProcs.AdicionarErro(0,0,0, erInicializacao +  'a' )end;
197-
198-      end;
199-
200-
201-      end.
```

Listagem 38 - Arquivo PalindromeImpar.pas

Anexo III – Reconhecedor da expressão $a^n b^n c^n$

O programa abaixo, na linguagem RSW, representa a primeira solução para o problema proposto no capítulo Conceitos, seção Autômatos Adaptativos.

```
1-   submaquina Solucao1AnBnCn ( eofTK )
2-
3-   entrada CadeiaEntrada
4-
5-   producoes
6-
7-   ( I, ‘a’) -> e1.
8-   (e1, ‘a’) -> e1 : AdicionarEntre(1,2).
9-   (e1, ‘b’) -> e2.
10-  (e2, ‘c’) -> F.
11-  (^*, F ) -> (^, *, eofTK).
12-
13-
14-  funcoes
15-
16-  AdicionarEntre( estado1, estado2)
17-  {
18-    *NovoEstado1, *NovoEstado2:
19-
20-    %Remove a transicao original consumindo ‘b’
21-    -[ (estado1, ‘b’) -> estado2 ]
22-
23-    %Adiciona os novos estados
24-    +[ (estado1, ‘b’) -> NovoEstado1 ]
25-    +[ (NovoEstado1, ‘b’) -> NovoEstado2 ]
26-    +[ (NovoEstado2, ‘c’) -> estado2 ]
27-
28-    %Ajusta os parâmetros da funcao adaptativa
29-    -[ (e1, ‘a’) -> e1 : AdicionarEntre(estado1, estado2) ]
30-    +[ (e1, ‘a’) -> e1 : AdicionarEntre(NovoEstado1, NovoEstado2) ]
31-
32-  }
```

Listagem 39 - Arquivo Solucao1.rsm

Anexo IV – Gramática da Linguagem RSW, na notação de Wirth

<autômato> ::= <gramática>.

<gramática> ::= 'submaquina' <identificador> <lista de átomos> <entrada> <produções> <funções adaptativas>.

<lista de átomos> ::= ' (' <entrada> { ',' <entrada> } ') '.

<entrada> ::= 'entrada' <identificador>.

<produções> ::= 'producoes' <lista de produções>.

<lista de produções> ::={ <produção> '.'}.

<produção> ::= <transição simples>
| <transição retorno>
.

<transição simples> ::= <simples esquerdo> '->' <simples direito>.

<transição retorno> ::= ' (' <estado> ') '
| <retorno esquerdo> '->' <retorno direito>.

<retorno esquerdo> ::= '(' <pilha padrão> ',' <estado> ')' [':' <chamada adaptativa>].

<retorno direito> ::= '(' <identificador de pilha> ',' <meta-estado> ',' <átomo> ')' [' : ' <chamada adaptativa>].

<pilha padrão> ::= <identificador de pilha> <meta-estado>.

<identificador de pilha> ::= '^'.

<meta-estado> ::= '*'.

<simples esquerdo> ::= <configuração original> [':' <chamada adaptativa>].

<simples direito> ::= <configuração destino> [':' <chamada adaptativa>].

<configuração original> ::= '(' [<identificador de pilha> ','] <estado> ',' <entrada> ')'.

<chamada adaptativa> ::=<identificador da função> ['(' <argumento> { ',' <argumento> } ')'].

<argumento> ::= <identificador>.

< configuração destino> ::= '(' [<pilha> ','] <estado> [',' entrada] ')'

| <estado>

.

<pilha> ::= <identificador de pilha> <identificador de estado>.

<sub-máquina> ::= <identificador de sub-máquinas>.

<estado> ::= <identificador de estado>.

<entrada> ::= <identificador de entrada>

| '<vazio>'.

<identificador de sub-máquinas> ::= <identificador>.

<identificador de estado> ::= <identificador>.

<identificador de entrada> ::= <identificador>.

<funções adaptativas> ::= 'Funções' <lista de funções>.

<lista de funções> ::= { <função> }.

<função> ::= <cabeçalho da função> <corpo da função>.

<cabeçalho da função> ::= <identificador da função> ['(' <tipo> <parâmetro> { ',' <tipo> <parâmetro> } ')'] '='.

<identificador de função> ::= <identificador>.

<tipo> ::= 'estado'
| 'atomo'
| 'sub-maquina'
.

<parâmetro> ::= <identificador>.

<corpo da função> ::= '{' <declaração de nomes> <declaração de ações> '}'.

<declaração de nomes> ::= <declaração de variáveis> <declaração de geradores> ' : '.

<declaração de variáveis> ::= [<tipo> <variável> { ',' <tipo> <variável> }].

<declaração de geradores> ::= [<tipo> ['*'] <variável> { ',' <tipo> ['*'] <variável> }].

<declaração de ações> ::= <função inicial> <ações elementares> <função final>.

<função inicial> ::= <chamada função adaptativa>.

<chamada função adaptativa> ::= [<identificador> '.'] <identificador> ['(' <argumento> { ',' <argumento> } ')'].

<ações elementares> ::= { ('+' | '-' | '?') '[' <produção> ']' }.

<função final> ::= <chamada adaptativa>.

<variável> ::= <identificador>.

<identificador> ::= <letra> { <letra> | <digito> | '_' }.

<letra> ::= (‘a’ | ‘b’ | ‘c’ | ‘d’| ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘w’ | ‘v’ | ‘y’ | ‘x’ | ‘z’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘W’ | ‘V’ | ‘X’ | ‘Y’ | ‘Z’).

<digito> ::= (‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ | ‘0’).

Referências Bibliográficas

1. ABMANN, W. 'A short review of high speed compilation'. Lecture Notes in Computer Science, No. 371, Springer Verlag, October 1988, pp. 1-10.
2. AHO, A.V.; SETHI, R.; ULLMAN, J. D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
3. ANDERSON, T.; EVE, J.; HORNING, J.J. 'Efficient LR(1) parsers'. Acta Inf., 2, 12-39 (1973).
4. BELL, J. R. 'A compression method for compiler precedence tables'. in J. L. Rosenfeld (ed.), Information Processing 74, North Holland, Amsterdam, 1974.
5. CABASINO, S.; PAOLUCCI, P. S.; TODESCO, G. M. 'Dynamic Parsers and Evolving Grammars'. ACM SIGPLAN Notices, Vol. 27, No. 11, 39-48(November 1992).
6. DENCKER, P.; DÜRRE, K.; HEUFT, J. 'Optimization of parser tables for portable compilers'. ACM Trans. Programming Language And Systems, 6, 546-572 (1984).
7. DERANSART, P.; JOURDAN, M.; LORHO, B. 'Attribute Grammars'. Lecture Notes in Computer Science., **323**, Springer, 1988.
8. DÜRRE, K. 'Coloring the vertices of an arbitrary graph'. Lecture Notes in economics and Math. Syst. 78, 1973.
9. DYADKIN, L.J. 'Multibox Parsers'. ACM SIGPLAN Notices, Volume 29, No 7, 54-60(July 1994).
10. ELLIS, M. E.; STROUSTRUP, B. 'The Annotated C++ Reference Manual'. Addison-Wesley, Reading, Massachusetts, 1990.
11. FÉDÈLE, C.; LECARME, O. 'Towards a Toolkit for Building Language Implementations'. Software - Practice and Experience. Vol. 22(11), 911-936(November 1992).

12.FISCHER, C.N.; LEBLANC JR., R. J. Crafting A Compiler with C. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

13.FURUTA, R.; STOTTS, P. D.; Ogata, J. 'Ytracc: a Parse Browse for Yacc Grammars'. Software - Practice and Experience, vol.21 (2), 119-132 (1991).

14.FUTAMURA, J. Partial Computation of Programs. Springer Verlag, Heidelberg, 1983.

15.GROSCH, J. 'Lalr - a Generator for Efficient Parsers'. Software - Practice and Experience, vol. 20(11), 1115-1135 (November 1990).

16.JOHNSON, S.C. 'Yacc meets C++'. In UNIX around the World, Proc. Spring 1988 EUUG Conference, 1988, pp. 53-57.

17.JOHNSON, S.C. 'YACC: yet another compiler-compiler'. Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, July 1975.

18.JOHNSON, S.C. 'YACC: yet another compiler-compiler'. In UNIX Programmer's Manual, **2** , ATT&T Bell Laboratories, Murray Hill, NJ, 1975.

19.JOLIAT, M. L. 'On the reduced matrix representation of LR(k) parser tables'. PH. D. Thesis, University of Toronto, Toronto, 1973.

20.JOLIAT, M. L. 'Practical minimization of LR(k) parser tables'. In J. L. Rosenfeld (ed.), Information Processing 74, North Holland, Amsterdam, 1974.

21.KLEIN, E.; MARTIN, M. 'The Parser Generating System PGS'. Software - Practice and Experience, Vol. 19(11), 1015-1028 (November 1989).

22.LECARME, O.; BOCHMANN, G.V. 'A (truly) usable and portable compiler writing system'. In J.L. Rosenfeld(ed.), Information Processing '74, North-Holland, Amsterdam, 1974 pp.218-221.

23.LESK, M.E.; SCHIMIDT, E. 'LEX - a lexical analyzer generator'. In UNIX Programmer's Manual, **2** , AT&T Bell Laboratories, Murray Hill, NJ, 1975.

24.MASON, T.; BROWN, D.; LEVINE, J. 'Lex & Yacc'. O'Reilly and Associates, Inc., Sebastopol, California, 1990.

25.MERNIK, M.; KORBAR, N.; ZUMER, V. 'LISA: A Tool for Automatic Language Implementation'. ACM SIGPLAN Notices, Volume 30, No.4, 71-79(April 1995).

26.MERRILL, G.H. 'Parsing non-LR(k) Grammars with Yacc'. Software - Practice and Experience, Vol. 23(8), 829-850 (August 1993).

27.NETO, J.J. Adaptive Automata for Context-Dependent Languages, ACM SIGPLAN Notices, Vol. 29, n. 9, September 1994, p.115-124.

28.NETO, J.J. Contribuições à Metodologia de Construção de Compiladores. Tese de Livre Docência - Escola Politécnica da USP, 1993.

29.NETO, J.J. Introdução à Compilação. Editora LTC, Rio de Janeiro, 1987.

30.PAGAN, F.G. Formal Specification of Programming Languages: A Panoramic Primer. Prentice-Hall, 1981.

31.PENNELLO, T. J. 'Very fast LR parsing'. SIGPLAN Notices, 21, 145-151(1986).

32.PEREIRA, J.C.D.; NETO, J.J. Um ambiente de desenvolvimento de reconhecedores sintáticos baseado em autômatos adaptativos. Simpósio Brasileiro de Linguagens de Programação, 1997.

33.SCHREINER, A. T.; FRIEDMAN, H. G. 'Introduction to Compiler Construction with UNIX'. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

34.SHUTT, J.N. Recursive Adaptable Grammars. Ph. D. Thesis, Worcester Polytechnic Institute, 1993.

35.SLONNEGER, K.; KURTZ, B.L. Formal syntax and semantics of programming languages - a laboratory based approach. Addison Wesley, 1995.

36.STROUSTRUP, B. 'The C++ Programming Language'. Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

37.TARJAN, R. E.; YAO, A. C. 'Storing a sparse table'. Comm. ACM, 22, 606-611 (1979).

38.WAITE, W.M.; GOOS, G. Compiler Construction. Springer Verlag, New York, NY, 1984.

Bibliografia Adicional

1. BOWEN, J. P.; BREUER, P. T.; LANO, K. C. 'Compendium of formal techniques for software maintenance'. Software Engineering Journal, 8(5), 253-262 (1993).
2. BREUER, P. T.; BOWEN, J. P. 'A Prettier Compiler-Compiler: generating Higher-order parsers in C'. Software and Experience, Vol. 25(11), 1263-1297 (November 1995).
3. BREUER, P. T.; BOWEN, J. P. 'The PRECC Compiler-Compiler'. In E. Davies and A. Findlay (eds.), Proc. UKUUG/SUKUG Joint New Year 1993 Conference, UKUUG/SUKUG Secretariat, Owles Hall, Buntingford, Hertz SG9 9PL, UK, pp. 167-182, January 1993.
4. BURSHTEYN, B. 'Generation and Recognition of Formal Languages by Modifiable Grammars'. ACM SIGPLAN Notices, Vol. 25, No. 12, 45-53 (December 1990).
5. BURSHTEYN, B. 'On the modification of the formal grammar at parse time'. SIGPLAN Notices, Vol.25, No.5, 117-123 (1990).
6. CHRISTIANSEN, H. 'A survey of adaptable grammars'. SIGPLAN Notices, Vol.25, No 11, 35-44 (November 1990).
7. DOBLER, H.; PIRKLBAUER, K. 'Coco-2 A New Compiler Compiler'. SIGPLAN Notices, Vol. 25, No 5.
8. EARLEY, J. 'An efficient context-free parsing algorithm'. Comms. ACM, 13(2), 94-102 (1970). (Reprinted in Comms. ACM 26(1), 57-61 (1983).).
9. FISCHER, B.; HAMMER, C.; STRUCKMANN, W. 'ALADIN: A Scanner Generator for Incremental Programming Environments'. Software - Practice and Experience, vol. 22(11), 1011-1025 (November 1992).
10. GANZINGER, H.; GIEGERICH, R.; MONCKE U.; WILHELM, R. 'A truly generative semantics-directed compiler generator'. Proc. SIGPLAN Symp. on Compiler Construction, SIGPLAN Notices, 17, (6), 172-184 (1982).

11.GROSCH, J. 'Generators for High-Speed *Front-end*s'. Lecture Notes in Computer Science, No. 371, 81-92 (October 1988).

12.HARFORD, A.G.; HEURING, V. P.; MAIN, M.G. 'A New Parsing Method for Non-LR(1) Grammars'. Software - Practice and experience, Vol. 22(5), 419-437 (May 1992).

13.HEERING, J.; KLINT, P.; REKERS, J. 'Incremental Generation of Parsers'. SIGPLAN Notices, Vol. 24, No. 7, 179-191 (1989).

14.HOPCROFT, J.E.; ULLMAN, J. D. 'Introduction to automata theory, language, and Computation'. Addison-Wesley (1979).

15.JOHNSON, S. C. 'Yacc meets C++'. In Unix around the World, Proc. Spring 1988 EUUG Conference, 1988, pp.53-57.

16.JOHNSON, W.L.; PORTER, J.H.; ACKLEY, S.I.; ROSS, D. T. 'Automatic Generation of efficient Lexical Processors Using Finite State Techniques'. Communications of the ACM, Vol. 11, No. 12, 805-813 (December 1968).

17.JUSTICE, T. P.; PANDEY, R.K.; BUDD, T. A. 'A Multiparadigm Approach to Compiler Construction'. ACM SIGPLAN Notices, Vol. 29, No 9, 29-37 (1994).

18.KASTENS, U.; HUTT, B.; ZIMMERMANN, E. GAG: a practical compiler generation. Lecture Notes in Computer Science., 141, Springer, 1982.

19.KNUTH, D. E. 'Semantics of context-free languages'. Math syst. Th., 2, (2), 127-145 (1968). Correction ibid., 5, (1), 95-96 (1971).

20.KOSKIMIES, K.; NURMI, O.; PAAKI, J. 'The Design of a Language Processor Generator'. Software - Practice and Experience, Vol. 18(2), 107-135 (February 1988).

21.KRISTENSEN, B.B.; MADSEN, O.L. 'Methods for computing LALR(K) lookahead'. ACM TOPLAS. 3(1), 60-82 (1981).

22.MOSSENBOCK, H. 'Alex - A Simple and Efficient Scanner Generator'. SIGPLAN Notices, Vol. 21, No. 12, 139-147 (December 1986).

23.PARK, J.C.H. 'y+: A Yacc Preprocessor for Certain Semantic Actions'. SIGPLAN Notices, Vol. 23, No. 6, 97-106 (June 1988).

24.REPS, T.; TEITELBAUM, T. The Synthesizer Generator Reference Manual, 3rd Ed., Springer, 1989.

25.ROBERTS, G.H. 'Recursive Ascent: A LR Analog to recursive Descent'. SIGPLAN Notices, Vol. 23, No. 8, 23-29 (August 1988).

26.SASSA, M.; Ishizuka, H.; Nakata, I. 'Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar'. Software - Practice and Experience, vol. 25 (3), 229-250 (1995).

27.SPECTOR, D. 'Efficient Full LR(1) Parser Generation'. SIGPLAN Notices, Vol. 23, No 12, August 1988.

28.YEH, D.; KASTENS, U. 'Automatic Construction of Incremental LR(1) - Parsers'. SIGPLAN Notices, Vol. 23, No.6, 33-42 (1988).

Capitulo	Esboço	v 1.0	OK	v2.0	OK	final
Introdução	01/10/98	01/10/98	2/12/98	4/1/99		
Teoria	01/10/98	01/10/98	2/12/98	4/1/99		
Proposta	01/10/98	01/10/98	2/12/98	4/1/99		
Projeto	01/10/98	12/10/98	2/12/98	4/1/99		
Implementação	01/10/98			4/1/99		
Experimentos	01/10/98			20/1/99		
Conclusão	01/10/98			22/1/99		
Glossário			2/12/98	22/1/99		
Bibliografia	01/10/98	01/10/98		22/1/99		
Bib. Adicional	01/10/98	01/10/98		22/1/99		
Anexos				22/1/99		