6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] DEL PICCHIA, W.; MARTINS, W.W. **The numerical transform.** Part 1: Basis. São Paulo, DEE-EPUSP, 1974. 43p. (Publicação Interna nº 16).

[2] ____; ____. **The numerical transform.** Part 2: Simplification of boolean functions. São Paulo. DEE-EPUSP, 1974. 61p. (Publicação Interna 17).

[3] ____; ____. **The numerical transform.** Part 3: Resolution of boolean equations Sao Paulo, DEE EPUSP, 1974. 29p. (Publicação Interna nº 18).

[4] ____ **A simple and fast algorithm for the resolutions of boolean equations.** São Paulo, DEE-EPUSP, 1974. 18p. (Publicação Interna nº 19).

____. A numerical algorithm for the resolution of boolean equations. **IEEE Trans. Comput.**, New York, 23(9): 983-6, sep.1974.

[5] ____. **Um algoritmo numérico para o manuseio de funções booleanas:** o problema da substituição. São Paulo, DEE-EPUSP, 1974. 8p. (Publicação Interna nº 20).

[6] ____. **Sistematização do projeto de máquinas sequenciais para diferentes sistemas de realimentação.** São Paulo, DEE-EPUSP, 1974. 15p. (Publicação Interna nº 21).

[7] ROTONDARO, I.G. **A aplicação da matemática booleana na obtenção dos sistemas de funções componentes de um sistema de funções de funções.** São Paulo, 1987. Tese (Doutorado) - Universidade Mackenzie.

[8] BRUNAZO F A., MAURO O.M. **A aritmética qualitativa e suas aplicações na inteligência artificial.** São Paulo, LSI-EPUSP, 1987.

[9] MARTINS, W.W.; LAPPONI, J.C. **Aritmética booleana.** São Paulo, DEE-EPUSP, 1975. 150p.

Data de entrega: maio/1988

COMPILADOR DE GRAMÁTICAS DESCRITAS EM NOTAÇÃO DE WIRTH MODIFICADA

João José Neto, Assistente-Doutor, EPUSP
Washington Komatsu, engenheiro, EPUSP

UNITERMOS: Compiladores. Autômatos. Linguagens formais. Notação de Wirth. Geração automática de reconhecedores sintáticos

Resumo:

Este artigo contém as especificações e o projeto de um gerador automático de reconhecedores sintáticos, na forma de um compilador de gramáticas (descrições de linguagem) livres de contexto. Estas gramáticas deverão estar representadas em uma metalinguagem descrita no artigo, derivada da Notação de Wirth.

Abstract:

The present paper shows the specifications and the design of an automatic generator of parsers from language descriptions using an extension of Wirth's notation for context-free grammars.

1. Especificações do Projeto:

I - Implementar um compilador que converte uma linguagem representada por um sistema de equações na notação de Wirth modificada em um reconhecedor sintático desta linguagem. O reconhecedor sintático será um conjunto de sub-máquinas, uma para cada equação, na forma de tabela de transições. O sistema de equações tem a forma:

Sistema:

$N_1 = exp_1(N_1, N_2, N_3)$ -> 1ª sub-máquina (principal)

$N_2 = exp_2(N_1, N_2, N_3)$ -> 2ª sub-máquina (auxiliar)

$N_3 = exp_3(N_1, N_2, N_3)$ -> 3ª sub-máquina (auxiliar)

(N_1: raiz da gramática)

Em princípio as linguagens compiláveis deverão ser do tipo livre de contexto (tipo 2, segundo a classificação de Chomsky) e deterministicas, isto é, geram reconhecedores deterministicos.

II - Implementar um programa que utilize o reconhecedor sintático gerado pelo compilador para reconhecer textos escritos na linguagem definida pelo sistema de equações.

2. Notação de Wirth Modificada:

A notação de Wirth modificada, ou "notação da barrinha", simplifica a descrição de iterações existente na notação de Wirth (também chamada de Forma de Backus-Naur Modificada - "Modified Backus-Naur Form") [1] e permite a geração de reconhecedores com a eliminação de um número significativo de estados e transições redundantes. Os elementos constituintes da regras são os mesmos que os da notação de Wirth, com duas exceções:

- Os colchetes [ ] (que simbolizam construções opcionais) foram eliminados por simplificação, pois a construção entre colchetes (por exemplo, [z]) equivale à mesma construção sem os colchetes e acrescida da opção ε, a cadeia vazia (do exemplo, (z|ε));
- As chaves { }, que simbolizam iterações, foram substituídas pelo conjunto abre parêntese, "barra reversa", fecha parêntese (\).

Na figura 2.1 podemos ver a comparação entre expressões na notação de Wirth, diagrama de estados e expressões na notação de Wirth modificada:

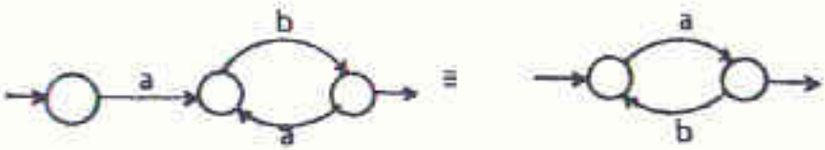
Wirth	diagrama de estados	Wirth modificada
(x)		(ϵ\x)
x(x)		(x\ϵ)
a(ba)		(a\b)

Figura 2.1: Comparação entre expressões na notação de Wirth, diagramas de estado e expressões na notação de Wirth modificada.

Uma produção na notação de Wirth modificada tem a seguinte forma:

X = E.

Onde: X - simboliza um não-terminal, cujo nome é dado por uma cadeia de caracteres qualquer;

E - expressão na notação de Wirth modificada;

= - associa ao não terminal X a expressão E;

. - delimita o fim da produção.

Uma expressão na notação de Wirth modificada (ou simplesmente expressão) pode ser definida recursivamente:

- Um terminal "y", entre aspas, forma uma expressão. y é uma cadeia de caracteres que pertence ao conjunto de átomos da linguagem.
- A cadeia vazia ε forma uma expressão.
- Um não terminal N, de nome dado por uma cadeia de caracteres qualquer, forma uma expressão.
- Uma expressão entre parênteses (E) simboliza o mesmo que a expressão isolada E. Os parênteses neste caso são usados apenas para fins de agrupamento.
- Dadas duas expressões E_1 e E_2, temos que:
  - a concatenação E_1E_2 é uma expressão;
  - a alternância $E_1|E_2$, onde E_1 e E_2 são alternativas válidas para associar ao não terminal X, é uma expressão;
  - a construção $(E_1\backslash E_2)$ é uma expressão, e representa iterações em número arbitrário das expressões E_1 e E_2, na forma E_1 ou $E_1E_2E_1$ ou $E_1E_2E_1E_2E_1$ e assim por diante (vide figura 2.1).

Observa-se que esta notação permite a especificação de gramáticas do tipo livre de contexto se for permitido apenas um único não-terminal à direita do símbolo = da produção.

3. Gramáticas na Notação de Wirth Modificada, para uso no Compilador

3.1 Definição da sintaxe

As descrições de linguagem a serem compiladas (para obtenção de reconhecedores sintáticos destas linguagens) serão fornecidas na forma de sistemas de equações, conforme já dito nas especificações do projeto. Cada equação do sistema consiste numa produção na notação de Wirth modificada.

Para fins de compilação, vamos definir uma sintaxe que represente este sistema de equações. Este sistema a ser compilado será uma seqüência de produções na notação de Wirth modificada, ligeiramente alteradas:

- Cada produção será terminada por um ponto e vírgula ; ao invés do ponto final, que será usado apenas ao fim da última produção.
- A raiz da gramática será o não-terminal definido pela primeira produção do sistema de equações.
- Os não-terminais e os terminais (átomos) das linguagens descritas terão somente um caracter (letras maiúsculas, minúsculas, algarismos ou símbolos especiais). Isso não tira a generalidade do projeto porém ajuda-o de várias formas:
  - = diminui o comprimento das produções a serem escritas;
  - = simplifica o uso e a manutenção de tabelas de símbolos;
  - = simplifica o projeto do analisador léxico.

- Os símbolos:

  (, abre parêntese,

) , fecha parêntese,

  | , "ou", símbolo de alternância,

  \ , a "barra reversa" da seqüência (\),

  " , aspas, que delimitam um terminal,

  = , "define-se como",

  € , épsilon, metasímbolo da cadeia vazia,

  ; , ponto e vírgula, que simboliza fim de uma produção e

  . , ponto final, que simboliza fim da descrição,

  não podem ser usados como não-terminais, por confundirem-se com os símbolos da metalinguagem. Mas podem ser usados livremente como terminais (isto é, entre aspas), o que significa que não há restrições nas linguagens a serem definidas, do ponto de vista de símbolos utilizáveis (para não-terminais usam-se geralmente letras do alfabeto, mnemônicas do nome da sub-máquina representada pelo não-terminal; por exemplo "P" de "Programa", "C" de "Comando", "E" de "Expressão" etc.);

- Comentários poderão ser inseridos no texto, delimitados pelos símbolos compostos (* e *). O símbolo (*, abre parêntese e asterisco, serve para abrir um comentário e o símbolo *), asterisco e fecha parêntese, para fechar um comentário.

- O formato da linguagem será livre. Os espaços em branco entre os átomos serão usados como separadores, para aumentar a legibilidade. O compilador só necessita de espaços separadores para fazer a distinção entre os símbolos compostos (* e *) (abertura e fechamento de comentário) e os

símbolos (* (abre parêntese seguido por espaço e por asterisco) e *) (asterisco seguido de espaço e de fecha parêntese). Isso será comentado no item 3.3.

- O fim de linha também servirá como separador.

Em relação à construção (\) e o simbolo de alternância | , devemos observar que as três expressões exemplificadas abaixo representam exatamente a mesma coisa:

('g' B | C D \ 'a' F)

(('g' B | C D) \ ('a' F))

((('g' B) | (C D)) \ ('a' F))

Estes três exemplos mostram que o uso de parênteses podem ajudar a diminuir as eventuais ambiguidades (para um leitor humano, pois para o compilador não há ambiguidade) que possam existir numa expressão, porém sobrecarregando a notação. O compromisso entre o uso de parênteses para esclarecer procedências ou não usá-los para não sobrecarregar a descrição deve ser feito apenas em função da inteligibilidade para o leitor.

A seguir, temos dois exemplos de descrições:

Primeiro exemplo:

```
(* Exemplo de definição de uma linguagem *)
(* Primeira Produção *)  A = (ε\B'a'|C'b')'c';
(* Segunda Produção  *)  B = (ε\C'c')'a';
(* Terceira Produção *)  C = (ε\'c')'b'.
```

Segundo exemplo:

```
(* Este exemplo é rebuscado...

*)

(* Primeira Produção *) $ = ( * * | & \ # # );

(* Segunda

Produção *)  * = '-' | '?' | '@' ; (*Terceira Produção*) & =
'6' |  '^' '^'  ; #  = '|'  '(' ';'  | ')' '.' (* Esta foi a
Quarta Produção*)

(*Agora acabou*) .
```

Naturalmente, para aumentar a legibilidade é preferível colocar uma produção em cada linha, começar cada produção na mesma coluna, usar letras ou símbolos mnemônicos como não-terminais, colocar comentários onde for necessário etc. O ponto final é o último símbolo que deve existir na descrição.

3.2 Descrição da metalinguagem

A seguir apresentamos a descrição da notação de Wirth modificada, na própria notação:

```
(* Notação de Wirth Modificada, para uso em compilador *)

(* Descrição:      *)  D = ( P \ ";" ) "." ;

(* Produção:       *)  P = N "=" E ;

(* Expressão:      *)  E = T | "ε" | N | "(" E ")" |
                           "(" E "\" E ")" | E "|" E | EE ;

(* Não-terminal:   *)  N = I ;

(* Terminal:       *)  T = "'" I "'" | "'" M "'" ;

(* Identificador:  *)  I = "A" | "B" | "C" | "D" | "E" |
                           "F" | "G" | "H" | "I" | "J" |
                           "K" | "L" | "M" | "N" | "O" |
                           "P" | "Q" | "R" | "S" | "T" |
                           "U" | "V" | "W" | "X" | "Y" |
                           "Z" |
                           "a" | "b" | "c" | "d" | "e" |
                           "f" | "g" | "h" | "i" | "j" |
                           "k" | "l" | "m" | "n" | "o" |
                           "p" | "q" | "r" | "s" | "t" |
                           "u" | "v" | "w" | "x" | "y" |
                           "z" |
                           "1" | "2" | "3" | "4" | "5" |
                           "6" | "7" | "8" | "9" | "0" |
                           "!" | "#" | "$" | "%" | "&" |
                           "'" | "*" | "+" | "," | "-" |
                           "/" | ":" | "<" | ">" | "?" |
                           "@" | "[" | "]" | "^" | "_" |
                           "`" | "{" | "}" | "~" ;

(* Metasímbolo     *)  M = "ε" | "'" | "(" | ")" | ";" |
                           "=" | "\" | "|" | "." .
```

Observa-se, segundo a descrição acima, que a metalinguagem especifica linguagens do tipo 2, sendo ela própria uma linguagem do tipo 2. Isso significa que o compilador a ser construído pode ser usado para compilar a sua própria descrição; isso será feito na fase de testes, para verificar a coerência dos resultados. Porém, não será

usada a descrição acima, pois a compilação dela resultaria num autômato não-determinístico. Usaremos uma descrição já devidamente manipulada, que resulte num autômato determinístico.

Nesta descrição dada não existem restrições quanto à utilização de identificadores. O controle do uso inadequado deles (por exemplo, o uso de não-terminais que não foram definidos) será atribuição das ações semânticas do compilador, as quais encarregar-se-ão da manutenção das tabelas de símbolos e tarefas correlatas, juntamente com as funções de "geração de código" (no nosso caso, geração das tabelas de transição do reconhecedor da gramática compilada). Outro ponto importante não coberto pela descrição é a identificação da raiz da gramática a ser compilada; convencionada como o primeiro não-terminal da descrição, tal convenção só fará sentido para as ações semânticas.

Nota-se também que a definição apresentada não descreve os comentários. A eliminação dos comentários é feita no analisador léxico (vide comentários no item 3.3).

3.3 Observações sobre alguns critérios adotados

Devemos comentar neste ponto a necessidade de usarmos um símbolo composto para delimitar os comentários. Isto se deve à natureza peculiar deste compilador, o qual não oferece restrições sobre os identificadores que podem ser usados, a menos das restrições sobre o tamanho deles (que devem ocupar apenas um caracter de comprimento) e sobre os não-terminais (os quais não podem ser metasímbolos).

Assim, os metasímbolos (neste caso não existem "palavras reservadas" tais como numa linguagem de

programação comum) e os identificadores "normais" (definidos/usados numa particular descrição, que são os terminais e os não-terminais da linguagem definida) só são distinguíveis uns dos outros por meio de análise sintática. Um símbolo de um só caracter para delimitar os comentários poderia ser confundido com algum identificador ou vice-versa, sendo impossível eliminar os comentários no analisador lêxico. A solução seria a incorporação dos comentários na definição da metalinguagem, através de metasímbolos de delimitação de comentários. A conseqüente incorporação da eliminação de comentários no analisador sintático (na prática, isso seria feito não atribuindo ações semânticas ao trecho do autômato reconhecedor que consome os comentários) aumentaria excessivamente o tamanho deste analisador. O esforço de manipulação desta gramática com comentários, não acrescentaria nenhum conceito novo, que valesse a pena explorar, no projeto. Por "esforço de manipulação" entenda-se manipular a gramática, eliminar não-determinismos, minimizar o autômato resultante, passar a limpo no processador de texto etc.

Adotamos as aspas " " para delimitar os terminais da linguagem a ser compilada, por ser usada também na notação de Wirth e por ser mais natural (um símbolo entre aspas tem o sentido de representar exatamente o que está "confinado" entre as aspas). Poderíamos ter usado outra convenção para os terminais, como na implementação da Microsoft para a linguagem de programação LISP. Nesta implementação usa-se apenas um apóstrofo ' antes de um literal para fazer esta distinção. No nosso caso esta convenção poderia ser interessante, já que terminais e não-terminais ocupam um caracter de comprimento, sendo imediato determinar onde "acaba" um terminal (isto é, achado um apóstrofo, saberíamos que o caracter seguinte é um terminal). De qualquer modo, esta é uma modificação que pode ser implantada muito facilmente no compilador, bastando

alterar ligeiramente a máquina de estados do analisador sintático (não haveria modificações nas ações semânticas).

Outro ponto que poderia ter sido desenvolvido na definição adotada seria o uso de opções de compilação para trocar os metasímbolos adotados (os metasímbolos "default") por outros à escolha do usuário. Isto poderia ser interessante no sentido de aumentar a legibilidade da gramática a compilar, no caso de uma gramática que usasse muito os metasímbolos por nós adotados (naturalmente, como terminais).

Os dois pontos comentados acima podem ser facilmente implementados por um usuário que compreenda os conceitos envolvidos neste compilador e a sistemática adotada para implementá-lo. Basta submeter ao próprio compilador uma descrição da metalinguagem, alterando-se os pontos desejados (aspas ou apóstrofos, metasímbolos adotados etc.). Ao analisador sintático resultante são então incorporadas as ações semânticas, que são as mesmas. O novo reconhecedor seria tão semelhante ao original que não haveria dificuldade nesta incorporação de ações semânticas.

4. Projeto dos Analisadores Sintático e Léxico

4.1 Esquema adotado para a implementação

Este projeto de compilador será organizado num esquema de compilação dirigida pela sintaxe ("syntax-driven"). Neste esquema o analisador sintático, implementado como um autômato, aciona o analisador léxico de modo a obter átomos que permitam a esse autômato transitar de um estado a outro. O analisador léxico obtém estes átomos a partir de uma descrição de linguagem (texto-fonte) armazenada num arquivo do sistema de computação onde estará residindo o compilador. Às transições do autômato estão associadas as ações semânticas, que encarregam-se de gerar o "código-objeto", no nosso caso um conjunto de tabelas de transição que representam um reconhecedor da linguagem compilada.

Para maior facilidade de implementação e portabilidade, o compilador será construido numa linguagem de alto nível, a princípio Pascal, num ambiente de microcomputador.

O analisador sintático será um autômato de pilha baseado na definição do item 3.2, de modo que tal autômato servirá para reconhecer descrições de linguagem na notação de Wirth modificada.

Este autômato de pilha será implementado na forma de tabelas de transição explícitas. Tais tabelas especificam o próximo estado do autômato na sub-máquina em função do estado atual e da saída fornecida pelo analisador léxico, ou, dependendo do estado, especificam chamadas e retorno de sub-máquinas; as sub-máquinas são representadas justamente por estas tabelas explícitas.

As tabelas deste autômato serão "percorridas" por uma rotina interpretadora de tabelas, também usada para acionar o analisador léxico e a ação semântica correspondente à transição efetuada. Chamadas e retorno de sub-máquinas serão controlados com o uso de uma pilha tipo LIFO ("last in, first out").

O presente esquema de tabelas explícitas foi adotado por causa de pelo menos três motivos: o primeiro é didático; tabelas de transição explícitas tornam mais claro o funcionamento da máquina.

O segundo motivo é o desejo de se facilitar a implementação; por causa da clareza de funcionamento é facilitado o trabalho de desenvolvimento e "debug" do compilador, inclusive simplificando modificações no autômato do analisador sintático, se necessário, bastando alterar-se as tabelas e deixando inalterada a rotina interpretadora.

O terceiro motivo é a economia de esforços obtida: o projeto completo do compilador prevê um programa que utilize o reconhecedor sintático obtido pelo compilador a partir da descrição de uma linguagem para reconhecer textos escritos nesta linguagem. O reconhecedor obtido estará na forma de tabelas de transição; assim é imediato utilizarmos o interpretador de tabelas do compilador para implementarmos este programa (com pequenas modificações no analisador léxico e retirando-se as chamadas de ações semânticas).

Deste modo, achamos que este esquema de implementação é o mais adequado para as finalidades do nosso projeto, apesar de existirem abordagens que permitem um melhor desempenho do compilador, principalmente em termos de velocidade. Seria o caso de um esquema que mapeasse o analisador sintático e rotinas associadas diretamente em um programa (do tipo "se o analisador léxico forneceu um átomo x, então execute rotina R, senão ...").

4.2 Projeto do Analisador Sintático

Da definição do item 3.2 podemos construir um autômato de pilha que reconheça descrições na notação de Wirth modificada.

Inicialmente iremos manipular a gramática obtida, de modo a determinar os não-terminais essenciais, eliminar não-determinismos e simplificar a descrição original, para fins de análise sintática. Tal descrição "enxugada" servirá também para dimensionarmos as necessidades do analisador léxico, isto é, dado o analisador sintático saberemos o que o analisador léxico precisará fornecer a ele, dado um texto-fonte (descrição de linguagem) para compilar. Assim, o analisador léxico será desenvolvido no item a seguir.

Nas manipulações de gramática desenvolvidas neste item, usaremos a notação de Wirth modificada conforme originalmente apresentada no item 2, para simplificar as produções.

Vamos à manipulação: o não-terminal "Descrição" é a raiz da gramática; ele pode ser expresso por meios dos terminais da gramática e do não-terminal "Expressão", eliminando-se o não-terminal "Produção". O não-terminal "Não-terminal" é substituído por "Identificador". Assim, temos a seguinte produção para "Descrição":

D = (I "=" E \ ";") "." .

O não-terminal "Expressão" necessita de um pouco mais de trabalho; dada a produção que representa "Expressão":

E = T | "ε" | N | "(" E ")" | "(" E "\" E ")" |
E "|" E | EE .

Colocando-se alguns termos em evidência:

```
E = T | "ε" | N | "(" E ( ")" | "\" E ")" ) |
    E ( "|" E | E ) .
```

Ou ainda:

```
E = T | "ε" | N | "(" E ( ")" | "\" E ")" ) |
    E ( "|" | ε ) E .
```

Observamos que a última opção (E ("|" | ε) E) significa que "Expressão" (E, onde E é qualquer opção válida) pode ter a forma E, EE, EEE ou assim por diante (ou seja (E \ ε)); ou ter a forma E, E|E, E|E|E ou assim por diante (isto é, (E \ "|")); ou então, uma combinação dos dois tipos ((E \ (ε | "|"))). Assim, "Expressão" pode ser:

```
E = T | "ε" | N | "(" E ( ")" | "\" E ")" ) |
    ( E \ ( "|" | ε ) ) .
```

Mas as quatro primeiras opções de "Expressão" são opções válidas para E, podendo portanto substituir E na última opção:

```
E = ( ( T | "ε" | N | "(" E ( ")" | "\" E ")" )) \
    ( "|" | ε ) ) .
```

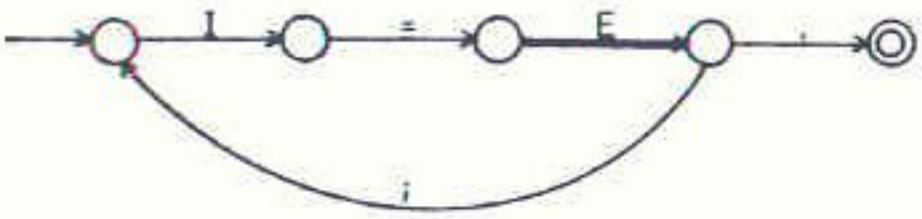
"Terminal" é "Identificador" ou "Metasímbolo", entre aspas; "Não-terminal" é "Identificador"; daí:

```
E = ( ( "'" ( I | M ) "'" | "ε" | I |
    "(" E ( ")" | "\" E ")" ) ) \ ( "|" | ε ) ) .
```

Para facilitar a visualização, vamos representar a gramática obtida na forma de um diagrama de estados. Devido à relativa simplicidade da gramática, vamos mapeá-la diretamente em um diagrama (alguns mapeamentos de gramáticas em diagramas de

estados foram vistos na figura 2.1). Tal diagrama pode ser visto na figura 4.1:

Descrição:



Expressão:

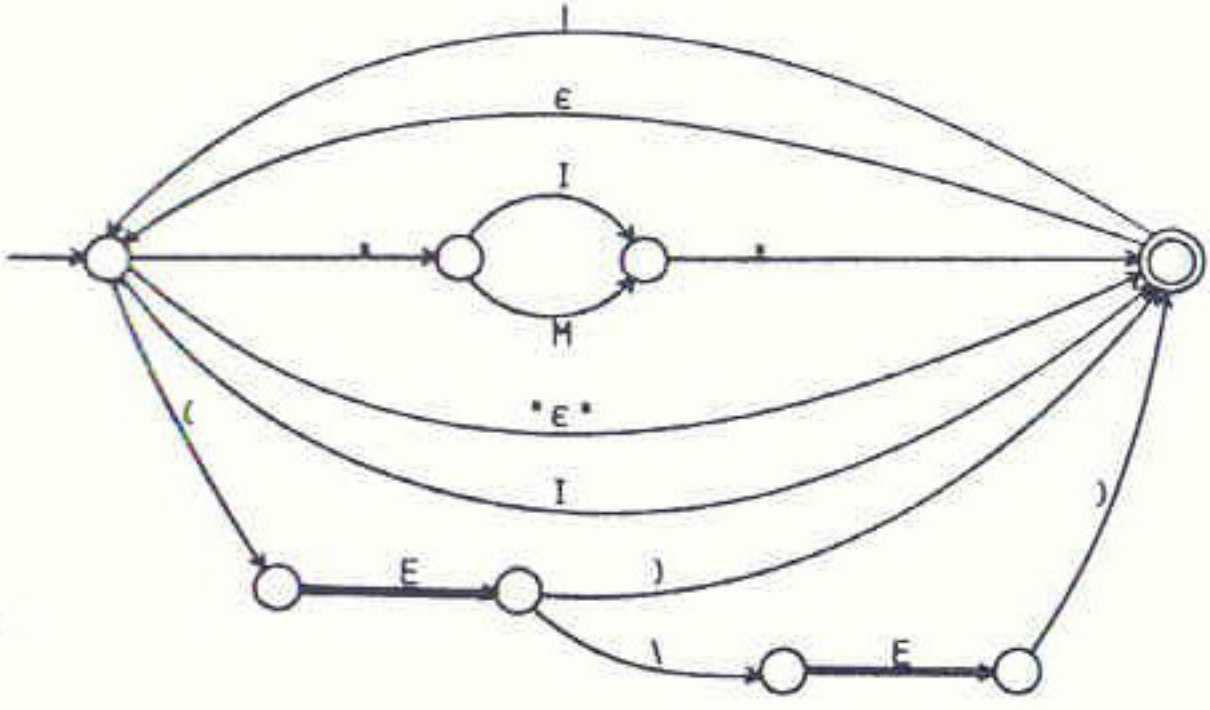


Figura 4.1: Diagramas de estado das produções "Descrição" e "Expressão". Notar a diferença entre ϵ e "ϵ". ϵ simboliza a cadeia vazia, portanto a transição associada é uma transição em vazio; "ϵ" simboliza o caracter épsilon, um metasímbolo da notação de Wirth modificada e um terminal da sub-máquina "Expressão".

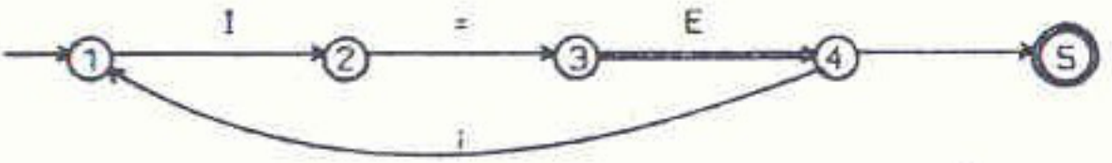
Da figura 4.1 podemos verificar, por inspeção, que o autômato "Descrição" é determinístico e mínimo, pois não possui transições em vazio, estados inacessíveis nem transições em vazio ou transições com o mesmo átomo para estados diferentes, a partir do mesmo estado.

Entretanto, "Expressão" ainda possui uma transição em vazio que o torna não-determinístico. Para eliminar este não-determinismo devemos suprimir esta transição em vazio incorporando ao estado de onde ela parte todas as transições que partem do estado alcançado por ela (a transição em vazio).

O analisador léxico deverá fornecer ao analisador sintático dois tipos de terminais: metasímbolos e identificadores. Os metasímbolos necessitam ser discriminados uns dos outros para o autômato poder transitar de um estado a outro. Já os identificadores por sua vez não necessitam ser discriminados no analisador sintático; basta o autômato saber que o terminal extraído do texto-fonte é um identificador para poder fazer a transição. A discriminação dos identificadores só terá função para as ações semânticas. Portanto, para proporcionar uma uniformidade de tratamento, o não-terminal "M", "Metasímbolo", deve ser substituido pelos respectivos terminais; o não-terminal "I", "Identificador", pode ser considerado um átomo, para fins de análise sintática (maiores comentários sobre o analisador léxico serão feitos durante o seu projeto, no item a seguir).

Deste modo, podemos ver na figura 4.2 os diagramas de estado das duas sub-máquinas que compõem o analisador sintático. Da figura podemos ver, por inspeção, que a máquina "Expressão" também pode ser considerada mínima e determinística.

Descrição:



Expressão:

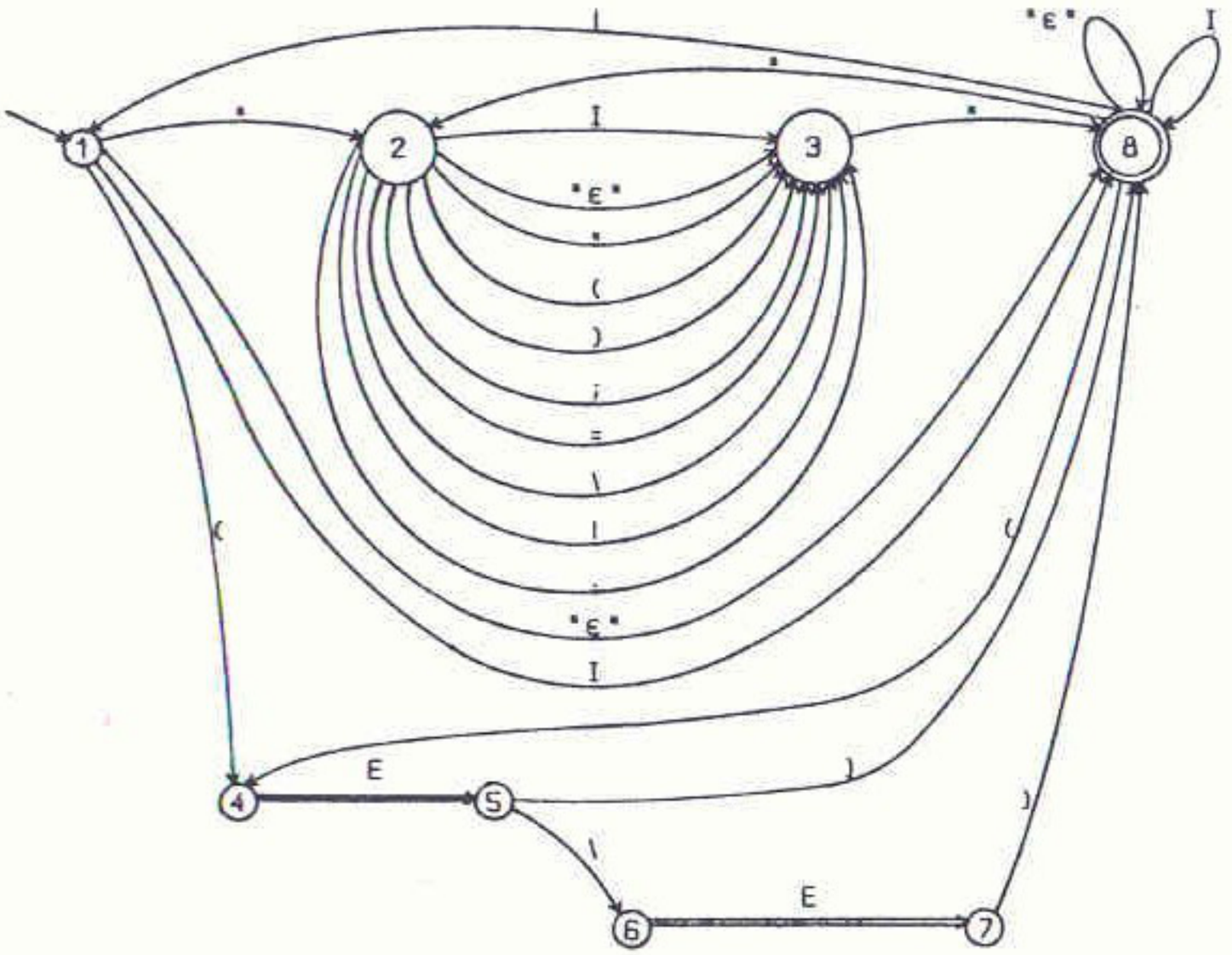


Figura 4.2: Diagrama de estados das submáquinas "Descrição" e "Expressão", que compõem o analisador sintático do compilador de gramáticas. A numeração dos estados é usada na construção das correspondentes tabelas de transição.

Deste modo, temos a seguinte descrição na notação de Wirth modificada para a gramática manipulada:

```
(* Descrição da gramática usada para implementar *)
(* o analisador sintático *)
(* Sub-máquina Descrição *)

   D = ( "I" "=" E \ ";" ) "." ;

(* Sub-máquina Expressão *)
   E = ( "ϵ" | "I" | "*" ( "I" | "ϵ" | "*" | "(" | ")" |
                           ";" | "=" | "\" | "|" | "." ) "*" |
         "(" E ( ")" | "\" E ")" ) ) \ ( "I" | ϵ ) .
```

Note-se que ao considerarmos "Identificador" como terminal para o analisador sintático, "I" fica entre aspas como os demais terminais.

Da figura 4.2 podemos obter as tabelas de transição das duas submáquinas que compõem o analisador sintático:

Descrição:

	I	=	E	;	.
1	2				
2		3			
3			4		
4				1	5
5					

Expressão:

	I	ϵ	*	(	)	;	=	\	|	.	E
1	8	8	2	4							
2	3	3	3	3	3	3	3	3	3	3	
3			8								
4											5
5					8			6			
6											7
7					8						
8	8	8	2	4					1		

A coluna com Є no cabeçalho da tabela "Expressão" representa transições com o terminal "Є" (caracter épsilon), um metasímbolo.

A rotina interpretadora de tabelas terá uma previsão para aceitar tabelas com transições em vazio, embora elas não ocorram nas tabelas das sub-máquinas do analisador sintático. Estas transições em vazio serão usadas numa convenção de menor prioridade, isto é, a transição em vazio só será usada se não for possível transitar por algum outro caminho. Assim, se o terminal fornecido pelo analisador léxico não servir para nenhuma alternativa que parta do presente estado, só restando a transição em vazio como opção, transita-se em vazio. Observe-se que esta convenção nem sempre é válida. Podem ocorrer casos de duas transições em vazio partindo do mesmo estado, uma transição em vazio e uma transição para sub-máquina a partir do mesmo estado ou ainda o caso em que a transição em vazio é a alternativa correta mesmo existindo uma transição correspondente ao terminal extraído pelo analisador léxico. Todos estes casos implicam num não-determinismo incontornável com a convenção de menor prioridade que as demais opções para a transição em vazio.

Implementaremos um interpretador de tabelas que consiga tratar transições em vazio para podermos ter um interpretador de tabelas mais versátil, que pudesse ser usado na segunda parte do projeto. Queremos esta vesatilidade pois prevemos que o compilador irá gerar tabelas de transição com várias transições em vazio. Isso ocorrerá porque muitas vezes as transições em vazio são usadas para implementar iterações do tipo {x} na notação de Wirth, que na notação de Wirth modificada fica (Є\x) (vide figura 2.1); ou são usadas para "isolar" elos de realimentação, como na expressão:

(A \ B) (C \ D), cujo correspondente diagrama de estados está na figura 4.3:

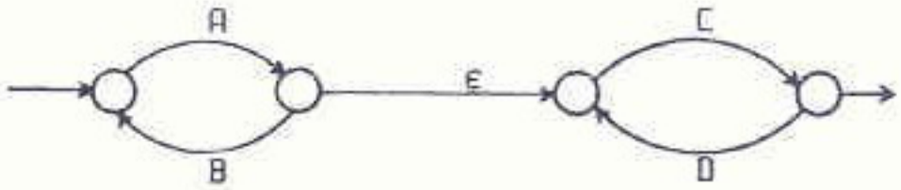


Figura 4.3: Transições em vazio usadas para "isolar" elos de realimentação.

Sem a transição em vazio separando as duas construções (\), o diagrama de estados poderia reconhecer textos do tipo "ACDBAC", não permitidos pela expressão original.

Naturalmente existe um diagrama de estados equivalente sem transições em vazio, que impede da mesma forma que a transição em vazio um "curto-circuito" entre os elos de realimentação D e B. Mas é muito mais simples para o compilador gerar as transições correspondentes a (A \ B) e "isolar" a construção (\) da construção que vier a seguir com uma transição em vazio (qualquer que seja essa construção), do que descobrir que a construção a seguir, (C \ D), é outra construção (\) e tomar providências para evitar o "curto-circuito". A dificuldade aumenta se A, B, C e D forem expressões e não átomos, existirem construções (\) encadeadas uma dentro da outra etc. Ademais, uma tabela de transições com transições em vazio pode ser otimizada (eliminando-se transições em vazio, estados equivalentes a outros e estados inacessíveis a partir do estado inicial) mecanicamente, sendo isso mais simples que implementar ações semânticas rebuscadas no compilador, pelo menos nesse ponto do projeto (isto é, seria melhor deixar os aperfeiçoamentos para uma fase posterior).

Nesse ponto podemos esboçar em palavras o algoritmo da rotina interpretadora de tabelas. Grosso modo

as estruturas de dados necessárias são as tabelas de transição, uma pilha tipo LIFO ("last in, first out") que armazena a sub-máquina e o estado para onde deve retornar o autômato após processar uma chamada de sub-máquina, além de variáveis para o estado atual, tabela (sub-máquina) atualmente em processamento etc.

O algoritmo inicia-se no estado correspondente à raiz da gramática, isto é, no primeiro identificador da descrição, conforme convencionado. Nesse ponto a rotina extrai, com o auxílio do analisador léxico, um átomo do texto-fonte e verifica o que fazer em função do estado e da sub-máquina atuais e do átomo extraído. Podem ocorrer três alternativas, testadas nesta seqüência (se não for a primeira alternativa, verificar a segunda; se não for a segunda verificar a terceira):

- O átomo extraído não existe no cabeçalho da tabela atual. Verificar, então, na seqüência abaixo, as alternativas:
  - Se o átomo extraído for um EOF ("End Of File", isto é, o fim do arquivo que contêm a descrição a compilar) e o estado atual for o estado final da submáquina raiz da gramática e a pilha estiver vazia, acabou a análise sintática, sem erros;
  - Se existir alguma chamada de sub-máquina no estado atual, armazenar o estado de retorno mais a sub-máquina atual na pilha e partir para o estado inicial da sub-máquina chamada, preservando-se o átomo extraído; repetir a rotina sem chamar o analisador léxico;
  - Se o estado atual for estado final da sub-máquina em processamento e a pilha não estiver vazia, retornar ao estado indicado no topo da pilha, preservando o átomo extraído e repetir a rotina sem a chamada do analisador léxico (isto é, verificar o que fazer em função do estado, sub-máquina e átomo extraído);

= Se existir alguma transição em vazio no estado atual, transitar para o estado indicado preservando-se o átomo extraído e repetir a rotina sem chamar o analisador léxico;

- A tabela atual fornece o próximo estado. Daí é só atualizar o estado atual e repetir a rotina (chamar o analisador léxico, etc.);

- Não há transição indicada na tabela com o átomo extraído, apesar dele existir no cabeçalho da tabela. O programa verifica então, na seqüência:

  = Se existir alguma chamada de sub-máquina no estado atual, armazenar o estado de retorno mais a sub-máquina atual na pilha e partir para o estado inicial da sub-máquina chamada, preservando-se o átomo extraído; repetir a rotina sem chamar o analisador léxico;

  = Se o estado atual é final na presente tabela e existir algum estado e sub-máquina de retorno armazenados no topo da pilha, retornar para este estado, preservando o átomo extraído; repetir a rotina a menos da chamada do analisador léxico;

  = Se existir alguma transição em vazio, transitar para o estado indicado preservando-se o átomo extraído e repetir a rotina sem chamar o analisador léxico.

As alternativas não cobertas pelo algoritmo implicam em erros de sintaxe, que dessincronizam o autômato. Note que o algoritmo funciona para autômatos determinísticos (com transições em vazio só usadas se não houver outra alternativa), assim deverá funcionar sem problemas no analisador sintático. No caso da aplicação como interpretador das tabelas geradas no compilador devemos tomar cuidados como prever a interrupção do processamento quando da ocorrência de erros, acompanhada de alguma mensagem de aviso.

Durante a implementação, para facilitar a depuração do programa, pode-se imprimir os átomos extraídos, junto com mensagens significativas como o estado e sub-

máquinas atuais, qual ação está sendo executada pelo programa etc.

4.3 Projeto do Analisador Léxico

Nos itens 4.1 e 4.2 foram esboçados alguns requisitos para o analisador léxico do nosso compilador. O analisador léxico é uma rotina ativada pelo analisador sintático cuja função será extrair os átomos necessários para acionar o analisador sintático, a partir de um texto que contenha a gramática a ser compilada. Este texto ficará armazenado num arquivo do sistema de computação onde o compilador for implementado. O uso de um arquivo facilita o uso do compilador, pois a descrição de linguagem pode ser gerada e modificada por um editor de texto do sistema, bastando submeter o arquivo editado ao compilador (análogamente, pretendemos também que o resultado final da compilação, as tabelas de transição do compilador, acabem num arquivo).

O analisador léxico também tem a função de filtro, eliminando os comentários, brancos e mudanças de linha ("carriage return"/"line feed" - CR/LF), de modo a apenas enviar terminais e a indicação de fim do arquivo da descrição em compilação (EOF - "End Of File") ao analisador sintático. Os terminais podem ser divididos entre identificadores e metasímbolos. Conforme já dito no item 4.2, somente os metasímbolos necessitam ser discriminados para uso do analisador sintático; os identificadores só necessitam ser identificados como uma classe. Porém, as ações semânticas necessitarão dessa discriminação dos identificadores, assim como dos metasímbolos. Portanto o analisador léxico irá fornecer como saída os metasímbolos já discriminados, enquanto os identificadores terão como saída

uma classificação "identificador", acompanhados de uma informação complementar que é justamente o caracter do identificador. Uma lista dos identificadores e metasímbolos possíveis está na definição da notação da metalinguagem, item 3.2.

A implementação do analisador léxico é simples. Será implementado como um autômato finito, que transita de um estado a outro em função dos caracteres extraídos da descrição da linguagem. Estes caracteres serão extraídos através de uma rotina de leitura de arquivo da linguagem de programação em que o analisador for implementado. O autômato finito será mapeado diretamente para um programa nesta linguagem de programação.

A figura 4.4 mostra o diagrama de estados do autômato finito que implementa o analisador léxico.

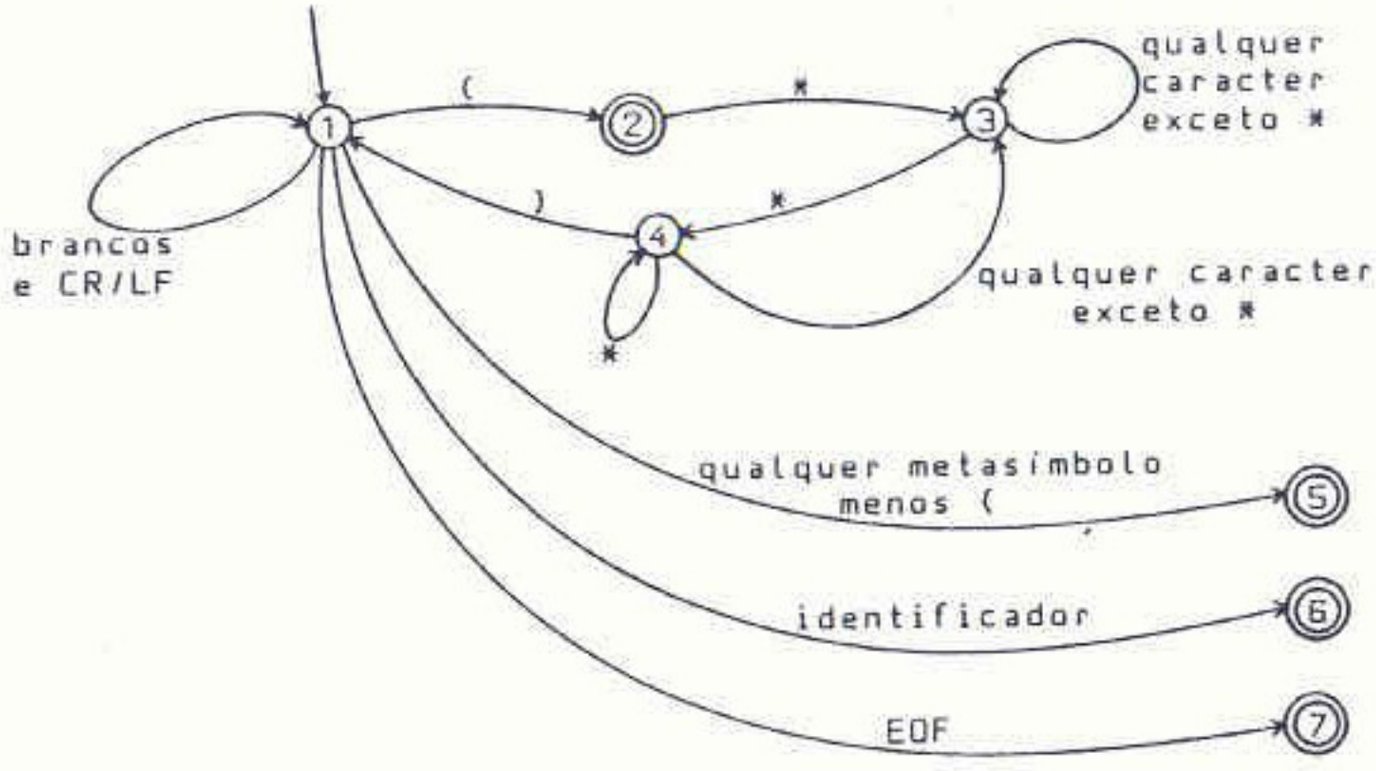


Figura 4.4: Autômato finito do analisador léxico.

Sempre que o analisador léxico é chamado ele inicia do estado 1, conforme figura 4.4, lendo os caracteres da descrição até atingir algum estado final, quando fornece algum terminal (ou EOF) ao analisador sintático. Os estados 2, 3 e 4 fazem a eliminação de comentários. Note-se que o estado final 2, que tem como saída do analisador léxico o metasimbolo (, só é confirmado com a leitura do caracter seguinte da descrição. Durante a implementação devemos tomar cuidado neste estado final pois não deve ser ignorado que este caracter extra foi lido quando chamar-se novamente o analisador léxico, sob pena de perdermos este caracter. Os estados 5, 6 e 7, que fornecem respectivamente os outros metasimbolos além do (, os identificadores e o caracter de fim de arquivo (EOF), não necessitam deste cuidado.

5. Projeto das Ações Semânticas

As ações semânticas devem ficar associadas às transições efetuadas no analisador sintático, de modo que gerem um reconhecedor sintático da descrição fornecida ao compilador. Tal reconhecedor deve estar na forma de tabelas de transição estruturadas de forma igual às tabelas do analisador sintático do compilador; assim podemos usar as tabelas para reconhecer textos usando o mesmo interpretador de tabelas do analisador sintático. A estrutura de dados destas tabelas do analisador sintático não foi definida ainda, devendo esta definição ser feita no momento da implementação do analisador sintático em uma linguagem de programação. Portanto para as ações semânticas também é prematuro especificar de modo exato e detalhado as rotinas semânticas, que ficam dependentes da implementação em linguagem de programação.

Assim, neste capítulo iremos fazer um esboço das estruturas de dados necessárias e das rotinas semânticas associadas ao analisador sintático. As rotinas semânticas servirão na implementação para preencher as tabelas de transição do reconhecedor compilado. Estas tabelas serão preenchidas com cinco tipos de informação:

- Informação de qual sub-máquina está sendo implementada em determinada tabela; novas sub-máquinas (isto é, a deteção de uma nova produção no sistema de equações compilado) levam à necessidade de se iniciar novas tabelas de transição correspondentes a estas máquinas;
- Numa dada tabela, qual é o estado inicial da sub-máquina (único por sub-máquina);

- Transição interna à sub-máquina; um terminal específico leva o autômato a transitar de um estado a outro dentro da mesma tabela;

- Transição de chamada de sub-máquina, na qual um determinado estado da sub-máquina em processamento (mais um certo conjunto de condições) obriga o autômato a transitar para outra sub-máquina, armazenando o estado de retorno para a sub-máquina de origem numa pilha (note-se que o estado de retorno geralmente não é o estado de onde o autômato transitou para a sub-máquina chamadora; portanto este estado de retorno também deve estar disponível na tabela de transições);

- Estado final de sub-máquina; pode existir mais de um por sub-máquina e indica (junto com um certo conjunto de condições) transição de retorno de sub-máquina ou término do reconhecimento (no caso da sub-máquina correspondente à raiz da gramática).

Os "certos conjuntos de condições", associados às informações de chamada de sub-máquina e estado final, são aqueles descritos no algoritmo interpretador de tabelas do analisador sintático (item 4.2).

As principais estruturas de dados necessárias para a implementação das ações semânticas são uma variável inteira i que guarda o estado atual da tabela que está sendo gerada, uma variável inteira s que guarda o estado seguinte ao estado que será processado em seguida ao estado atual, variáveis inteiras j, k e l para rascunho e cinco pilhas tipo LIFO: **AbreParêntese**, que empilha o estado i do momento em que foi consumido um (no analisador sintático; **EstadoAntesDaBarra**, que empilha o estado i do momento em que foi consumido um \ no analisador sintático; **FimDeAlternativa**, que armazena elementos compostos em dois campos: um com o estado obtido do topo de AbreParêntese ou

EstadoAntesDaBarra (que chamaremos de campo Início) e o outro com o estado ¡ existente no instante em que o analisador sintático consome um ¡ (que chamaremos de campo Fim) ; **Definidos**, que armazena os identificadores (isto é, a informação complementar fornecida pelo analisador léxico junto com o terminal "identificador" - vide item 4.3) definidos pelas produções da gramática em compilação; **Chamados**, que armazena os identificadores usados como chamadas de sub-máquinas nas produções da gramática em compilação.

As pilhas Definidos e Chamados servem para verificar a coerência entre as sub-máquinas definidas pelas produções da gramática a ser compilada com as chamadas de sub-máquina existentes nestas produções, pois não pode haver chamada de sub-máquina não definida, sob pena do reconhecedor gerado não funcionar. O caso contrário, definições que não são chamadas, podem ocorrer embora o reconhecedor fique com uma sub-máquina inacessível, portanto redundante. Assim, quando o analisador sintático terminar de verificar o texto-fonte a compilar (com a atuação conjunta das ações semânticas), uma ação semântica irá verificar se tudo que foi chamado também foi declarado.

Vamos esboçar em palavras as rotinas necessárias para a implementação das ações semânticas associadas a cada transição do analisador sintático, tendo como base os diagramas de estado da figura 4.3 (os diagramas que serão usados para implementar o analisador sintático). Na implementação estas rotinas serão chamadas de acordo com a transição feita pelo analisador sintático. Nas transições com o terminal I devemos observar que o dado efetivamente usado pelas ações semânticas (para preencher cabeçalhos de tabelas e as pilhas de controle de sub-máquinas Definidos e Chamados) é a informação complementar fornecida pelo analisador léxico. Esta informação complementar (que não passa de um caracter classificado pelo analisador léxico

como "Identificador") será referenciada daqui por diante como o "complemento de I".

Inicialmente, todas as estruturas de dados estão vazias (pilhas) ou "zeradas" (variáveis). O campo Inicio do topo de FimDeAltenativa deve ter um valor que não seja nem símbolo de pilha vazia nem algum estado válido; por exemplo, -1. As transições não citadas não possuem ações semânticas associadas.

Sub-máquina "Descrição":

1-▶2 (com I): Iniciar o preenchimento da tabela de transições da sub-máquina definida pelo complemento de I. Empilhar este caracter na pilha Definidos. Atribuir o número 1, convencionado como o estado inicial, à variável i (i := 1);

4-▶1 (com ;): O estado contido na variável i é final; incluir tal informação na tabela em montagem, encerrando o seu preenchimento;

4-▶5 (com .): O estado contido em i é final; incluir tal informação na tabela em montagem, encerrando o seu preenchimento. Verificar se todas as sub-máquinas contidas na pilha Chamados (isto é, os caracteres que as representam) também estão na pilha Definidos. Para cada uma que não não estiver, acusar o erro com uma mensagem "Erro: não-terminal X não foi definido", onde X é a sub-máquina existente em Chamados mas inexistente em Definidos.

Sub-máquina "Expressão":

2-▶3 (com qualquer terminal): é um consumo de átomo, portanto devemos expressar uma transição interna à sub-máquina na tabela: o autômato transita do estado atual i para o estado seguinte, com o terminal

extraído (no caso do terminal I, com a informação complementar). Para determinar este estado seguinte, devemos testar a variável s, que contém o estado seguinte no caso dele (o estado seguinte) não ser igual a i+1. Testar s; caso s tenha valor igual a zero, o estado seguinte a i é i+1, portanto atribuir i+1 à variável j (j := i+1); senão, o estado seguinte a i está em s, portanto atribuir s à variável j (j := s) e zerar s em seguida (s := 0). Assim, incluir na tabela a informação de que existe uma transição do estado i para o estado j com o terminal extraído. Atualizar o valor da variável i com o valor da variável j (i := j);

1-▶8, 8-▶8 (com "€"): Transição interna: i transita em vazio para o estado seguinte. Se s for igual a zero, atribuir i+1 a j (j := j+1); senão, atribuir s a j (j := s) e zerar s (s := 0). Informar à tabela que existe uma transição em vazio do estado i ao estado j. Atualizar i com o valor de j (i := j);

1-▶8, 8-▶8 (com I): Chamada de sub-máquina. O autômato transita para a sub-máquina definida pelo complemento de I. Preencher o estado de retorno j na posição de tabela dada pelo estado i (linha da tabela) e pelo nome da sub-máquina para onde transita o autômato (coluna da tabela). O estado de retorno j é igual a i+1 (j := i+1) caso s seja igual a zero; senão, j é dado por s (j := s), zerando s após a atribuição. Armazenar o complemento de I na pilha Chamados. Atualizar i com j (i := j);

1-▶4, 8-▶4 (com (): Abre parêntese. Pode ser para agrupar um conjunto de opções, uma única opção ou uma construção (\). Empilhar a variável i em AbreParêntese; empilhar o número -1 em EstadoAntesDaBarra (supondo-se que -1 não seja símbolo de pilha vazia);

5-▸6 (com \): Surgiu a barra da construção (\). Retirar o número -1 que estava empilhado no topo da pilha EstadoAntesDaBarra, substituindo-o por i. Podem existir uma série de opções antes da \ (alguma coisa na forma (A | B | C \ D)). Assim, verificar se o conteúdo do topo da pilha AbreParêntese é igual ao campo Início do elemento no topo da pilha FimDeAlternativa, sem alterar o conteúdo das pilhas. Se forem iguais, significa que apareceu pelo menos um símbolo | entre o (e o \ ; então, desempilhar o topo de FimDeAlternativa, colocando o campo Fim na variável j. Inserir na tabela a informação de uma transição em vazio do estado j ao estado i;

5-▸8 (com)): Fecha parêntese. Sabemos que não foi um parêntese da construção (\), devido à posição desta transição no diagrama de estados, podendo ser um conjunto de opções entre parênteses. Assim, descartar o conteúdo do topo de EstadoAntesDaBarra (desempilhar e jogar fora), desempilhar o conteúdo do topo de AbreParêntese em j. Verificar então se j é igual ao campo Início do elemento no topo da pilha FimDeAlternativa (sem alterar seu conteúdo). Se for, desempilhar o topo da pilha FimDeAlternativa, colocando o conteúdo do campo Fim na variável k. Finalmente, preenche-se a tabela com a informação de que existe uma transição em vazio do estado k ao estado i:

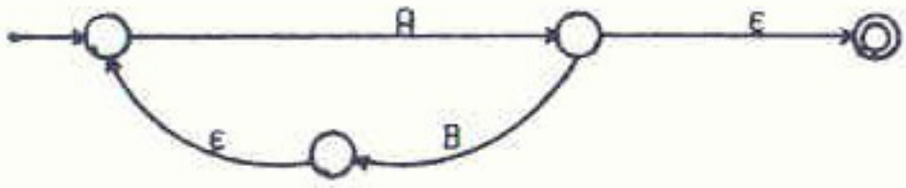
7-▸8 (com)): É o fecha parêntese de (\). Desempilhar o topo de EstadoAntesDaBarra em j e o topo de AbreParêntese em k. Podem existir uma série de opções entre o \ e o). Assim, testar se j é igual ao campo Início do topo de FimDeAlternativa (sem alterar seu conteúdo). Se for, desempilhar o topo de FimDeAlternativa, colocando o campo Fim na variável l e preencher a tabela com uma transição em vazio de l para i; preencher a tabela

com duas transições em vazio: uma de j a i+1, a outra de i para k. Independentemente do teste, atualizar i (i := i+1);

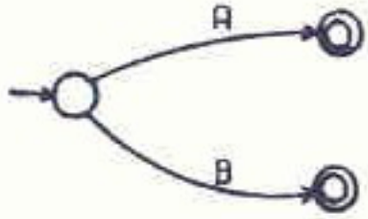
8-▶1 (com |): Término de uma opção. Existem duas possibilidades (supondo-se que não existam erros sintáticos): as opções não estão entre parênteses (neste caso cada início de opção é um estado inicial da sub-máquina e cada fim de opção é um estado final) ou estão entre parênteses. Para testar em qual caso estamos, verificar o topo da pilha AbreParêntese, sem alterar seu conteúdo. Se o topo estiver vazio (pilha vazia) então é o primeiro caso: incluir na tabela as informações de que i é estado final, armazenar o estado seguinte ao próximo estado i, i+1, na variável s (s := i+1) e ajustar o próximo estado atual para que a opção seguinte também começe do estado inicial, fazendo i igual a 1 (i := 1). Se contiver algum estado armazenado no topo, é o segundo caso: Verificar se o topo de EstadoAntesDaBarra é igual a -1. Se não for, estarmos com as opções entre o \ e o); neste caso armazenar, sem desempilhar, o topo de EstadoAntesDaBarra em j. Se o topo de EstadoAntesDaBarra for igual a -1, estaremos com as opções iniciando com (; armazenar, sem desempilhar, o topo de AbreParêntese em j. Verificar (também sem desempilhar) se o topo da pilha FimDeAlternativa tem o campo Início igual a j; se tiver, desempilhar FimDeAlternativa, colocando o campo Fim em k e inserindo na tabela uma informação de transição em vazio, do estado k ao estado i. Ainda dentro do segundo caso, mais três ações: empilhar j e i em FimDeAlternativa (j no campo Início e i no campo Fim); armazenar o estado seguinte ao próximo i, guardando i+1 em s (s := i+1); e ajustar o início da próxima opção, fazendo i igual ao estado de onde as opções se iniciam, que é o estado contido em j (i := j).

Na figura 5.1 vemos como ficam algumas expressões da notação (\), compiladas para os reconhecedores equivalentes segundo as ações semânticas descritas acima.

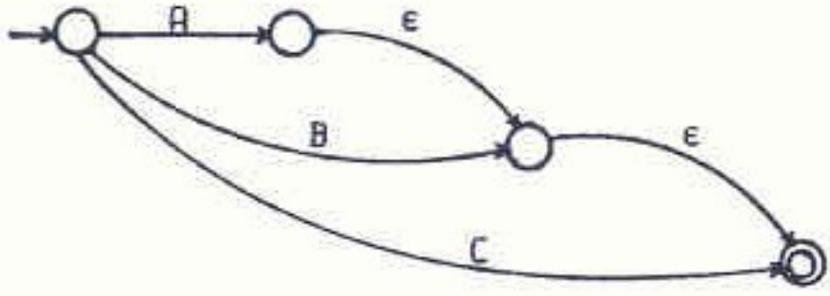
(A \ B)



A | B



(A | B | C)



(A | B \ C | D)

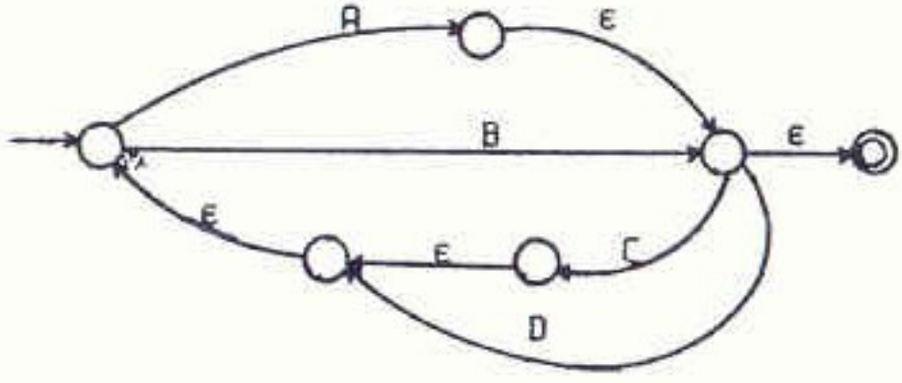


Figura 5.1: Expressões "compiladas" para diagramas de estado segundo as ações semânticas projetadas.

Podemos observar na figura 5.1 que os reconhecedores gerados não são os mínimos; existem muitas transições em vazio, inseridas para simplificar as ações semânticas. O que poderia ser feito é posteriormente projetar um programa otimizador, para pós-processar as tabelas geradas, eliminando transições em vazio, estados redundantes (equivalentes a outros já existentes), estados inacessíveis etc. Isso não será implementado no presente projeto, ficando como uma sugestão para o futuro. Outra opção óbvia seria alterar as ações semânticas, de modo a gerar reconhecedores mais otimizados. Seria necessário um estudo para avaliar qual opção seria mais simples de projetar e implementar, embora do ponto de vista de implantação a primeira opção pareca melhor, pois o otimizador é feito depois que o compilador estiver funcionando corretamente, sem "desfazer" o serviço já feito.

6. Conclusões e um Esboço dos Futuros Desenvolvimentos

O presente artigo descreve o projeto do compilador de gramáticas: especificações, definições de critérios adotados no projeto e na implementação, determinação de algoritmos e estruturas de dados.

O trabalho que será feito a seguir será a implementação do projeto, conforme as especificações do item 1. Teremos então o núcleo de um laboratório didático para o desenvolvimento e estudo de analisadores sintáticos, onde analisadores sintáticos poderão ser gerados pelo compilador a partir da descrição da linguagem a reconhecer e testados aplicando-se textos (pertencentes e não pertencentes à linguagem definida) ao reconhecedor obtido através do uso do interpretador de tabelas.

Outro projeto a ser definido e implementado após este núcleo seria o projeto de um otimizador de reconhecedores, que otimizasse os reconhecedores obtidos no compilador implementado. O uso de um otimizador ao invés de se rever as ações semânticas do compilador de gramáticas parece-nos mais interessante, pois modulariza o projeto, implementando a etapa seguinte apenas após a etapa atual estar em funcionamento.

Outros projetos anexáveis ao projeto original poderiam ser estudados: conversores de notação, que convertessem à notação modificada de Wirth outras notações usadas para descrição de linguagens como BNF (Backus-Naur Form), notação de Wirth original, notação usada para a definição do COBOL etc., para aplicar a gramática convertida ao nosso compilador; esquemas de deteção e recuperação de erros no analisador sintático do compilador e talvez até nos reconhecedores gerados. Todos estes projetos serviriam para enriquecer o núcleo original do nosso laboratório didático.

7. REFERENCIAS BIBLIOGRÁFICAS

As referências bibliográficas utilizadas neste artigo foram:

(1) JOSÉ NETO, J. **Introdução à compilação**. Rio de Janeiro, LTC, 1987. 222p. (Engenharia de Computação)

(2) JENSEN, K.; WIRTH, N. **Pascal**: user manual and report. 3.ed. New York, Springer, 1985. 266p.

(3) SEDGEWICK, R. **Algorithms**. Reading, Addison-Wesley, 1983. 551p.

A referência básica deste trabalho foi [1]; contém vários conceitos básicos aqui apresentados, como classificação de gramáticas, metalinguagens, reconhecedores baseados em autômatos de pilha estruturados (na forma de sub-máquinas), organização física e especificação das funções internas de compiladores.

A referência [2] descreve a notação de Wirth, citada na publicação como "Modified Backus-Naur Form" (Forma de Backus-Naur Modificada).

A referência [3] comenta de forma breve o conceito de geradores automáticos de reconhecedores sintáticos ("parser generators").

Data de entrega: dezembro/88