Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Gerador de Analisadores Léxicos $\mathcal{GAL}^{\mathcal{SIC}}$ para ambiente Windows

por

Mariza Andrade da Silva Bigonha
Erika Hamacek Pinto

RT 007/98

Caixa Postal, 702

30.161 - Belo Horizonte - MG

January 20, 2023

Contents

Abstract

$\mathcal{GAL}^{\mathcal{SIC}}$ is a tool built for constructing lexical analyzers from special-purpose notations based on regular expressions. This system is based in $\mathcal{LEX}$, another tool built to specify lexical analyzers. The main difference between them is the environment they run. $\mathcal{LEX}$ works in UNIX environment, is written in C language and generates the lexical analyzers in C. $\mathcal{GAL}^{\mathcal{SIC}}$ works in WINDOWS environment, was designed using Delphi and generates the lexical analyzers in C. $\mathcal{GAL}^{\mathcal{SIC}}$ was projected to work with $\mathcal{SIC}$ (Compiler Implementation System) also implemented in DCC-ICEx-UFMG.

Keywords: lexical analyzers, regular expressions, compiler generator.

1 Introdução

O Sistema de Implementação de Compiladores, $\mathcal{SIC}$, é um sistema desenvolvido no Departamento de Ciência da Computação da UFMG pelo Grupo de Linguagens de Programação para auxiliar o projetista de compiladores. Atualmente existem duas versões do sistema, uma para o ambiente MSDOS [6] e a outra para o ambiente WINDOWS NT [7].

O $\mathcal{SIC}$ gera automaticamente, a partir da especificação da sintaxe de uma linguagem, tabelas LALR(1) compactadas e rotinas para efetuar a análise sintática [1, 2, 3] . A saída é um programa em Pascal ou C dependendo da versão utilizada, constituindo o reconhecedor sintático acrescido de um mecanismo de recuperação de erro independente da linguagem.

Atualmente para usar o $\mathcal{SIC}$, o usuário deve escrever um analisador léxico manualmente. O analisador léxico lê a cadeia de caracteres da entrada, reconhece os *tokens* da linguagem e passa esta informação ao analisador sintático. O *token* passado ao analisador sintático é um par (`tipo`,`valor`) onde o `tipo` é um inteiro representando o tipo do *token* e `valor` diz respeito ao atributo do *token*, por exemplo: `"addop,mais"` representa o operador de adição, `"+"` . O analisador sintático e o analisador léxico devem usar a mesma nomenclatura para estas informações para que a comunicação estre eles se proceda.

O projeto proposto tem como objetivo a especificação e implementação um Gerador de Analisadores Léxicos, $\mathcal{GAL}^{\mathcal{SIC}}$. Este gerador será implementado usando o compilador Pascal do Delphi [8, 14] e gerará o analisador léxico na linguagem C [16, 15, 9] para ser integrado ao $\mathcal{SIC}$, versão WINDOWS NT.

Concluído este projeto teremos um sistema completo em ambiente WINDOWS NT que servirá como ferramenta adequada à escrita de compiladores, em particular, de *front-end* de compiladores.

Para atingir nosso objetivo, produzir uma ferramenta para gerar a análise léxica para o $\mathcal{SIC}$, este projeto foi dividido em quatro fases:

- revisão da literatura;
- desenvolvimento do $\mathcal{GAL}^{\mathcal{SIC}}$;
- desenvolvimento da interface e sua integração ao $\mathcal{SIC}$;
- elaboração de manuais.

Este relatório apresenta o resultado final do desenvolvimento do gerador de analisadores léxicos $\mathcal{GAL}^{\mathcal{SIC}}$.

Ele está organizado da seguinte forma: o Capítulo 2 apresenta a motivação na escolha do tema proposto. No Capítulo 3 é feita uma revisão da literatura, descrevendo inicialmente o sistema $\mathcal{SIC}$ implementado no ambiente WINDOWS NT e MSDOS. Em seguida, são descritos alguns sistemas

de geradores de analisadores léxicos existentes. O Capítulo 4 apresenta o Gerador de Analisadores Léxicos ($\mathcal{GAL}^{\mathcal{SIC}}$) proposto. Neste capítulo são discutidas as decisões de projeto, as dificuldades encontradas e sua implementação. O Capítulo 5 apresenta as contribuições e conclusões do trabalho. O Apêndice A mostra a especificação de um analisador léxico para uma linguagem *pascal-like* que constitui a entrada para o $\mathcal{GAL}^{\mathcal{SIC}}$. O Apêndice B apresenta o resultado da submissão do exemplo mostrado no Apêndice A, ou seja, o analisador léxico gerado.

2 Motivação

O que motivou a elaboração deste projeto é que não existe, hoje na literatura, um sistema que possibilite a automação da fase de análise léxica de um compilador para o ambiente WINDOWS. No ambiente Unix, existe o $\mathcal{LEX}$ [12] , ferramenta útil para construir analisadores léxicos. Os analisadores léxicos produzidos por ele foram projetados para trabalhar em conjunto com o YACC *(Yet Another Compiler Compiler)* [11]. A especificação para estes analisadores léxicos é feita usando expressões regulares ao invés de gramática livre do contexto.

3 Revisão da Literatura

3.1 Sistema de Implementação de Compiladores $\mathcal{SIC}$

$\mathcal{SIC}$ é uma ferramenta de auxílio à escrita de compiladores destinado aos projetistas de compiladores na linguagem Pascal [6] e C [7]. A partir da especificação da sintaxe, ele gera tabelas LALR(1) compactadas e um reconhecedor sintático acrescido de um mecanismo de recuperação de erro independente de linguagem que fornece mensagens de erros automaticamente.

Ainda permite o uso de gramáticas ambíguas na produção de analisadores sintáticos determinísticos.

A parte da sintaxe do $\mathcal{SIC}$ relevante ao desenvolvimento do $\mathcal{GAL}^{\mathcal{SIC}}$ é descrita a seguir. $\mathcal{SIC}$ define os símbolos básicos como sendo:

- símbolos terminais ou *token*: sequência de caracteres diferentes de aspas;
- símbolos não-terminais ou nt: identificadores;
- índices: utilizados para se ter acesso a uma ocorrência específica de um símbolo terminal ou não-terminal em uma produção;
- texto: sequências de construções da linguagem C;

- identificadores: sequências de letras e/ou dígitos iniciados sempre por uma letra. *Underscores* também podem ser usados na formação de identificadores. Podem ter qualquer tamanho, entretanto, só os quinze primeiros caracteres serão considerados, truncando-se o restante.

- Palavras reservadas:

  - YYEOF é uma constante do tipo inteira representando o código da marca de fim do arquivo;
  - YYSAIDA é um arquivo do tipo texto, declarado pelo $\mathcal{SIC}$, no qual são gravados os programa fonte, as mensagens de erros sintáticos geradas, a coleção canônica LR(0), a tabela compactada LALR(1) e a gramática;
  - YYSIMB é uma variável global do tipo inteira gerada e declarada pelo $\mathcal{SIC}$, devendo ser usada para retornar o tipo do símbolo reconhecido pelo analisador léxico. Funciona como elo de comunicação entre o analisador léxico YYSCAN escrito pelo usuário e o analisador sintático YYPARSER gerado pelo $\mathcal{SIC}$;
  - YYPOS é uma variável global do tipo inteira gerada e declarada pelo $\mathcal{SIC}$ servindo para retornar a posição do último símbolo reconhecido pelo YYSCAN. Pode ser usado como atributo de leitura de qualquer símbolo da gramática;
  - YYLINHA é uma variável global do tipo inteira gerada e declarada pelo $\mathcal{SIC}$ servindo para informar ao YYPARSER, o número corrente da linha que contém o último *token* lido por YYSCAN;
  - YYSCAN nome do analisador léxico do compilador a ser projetado, que deve ser escrito pelo usuário. Retorna valores ao analisador sintático por meio das variáveis YYSIMB, YYLINHA, YYPOS e de atributos de terminais;
  - YYPARSER nome do procedimento de análise sintática. O projetista deve invocar este procedimento explicitamente do corpo de seu programa fonte;

- as palavras-chave podem ser escritas em letras maiúsculas e minúsculas;

- os delimitadores simples em $\mathcal{SIC}$ são os seguintes caracteres especiais e pares de caracteres: (), { }, ; , . , : , = , | ;

- brancos e caracteres de controle de impressão, atuam como delimitadores. Comentários são iniciados com o símbolo (* e terminam como *);

- $\mathcal{SIC}$ apresenta uma seção de terminais que tem por finalidade estabelecer um mecanismo de comunicação entre os analisador sintático

gerado pelo $\mathcal{SIC}$, YYPARSER, e o analisador léxico, YYSCAN, projetado pelo usuário. O usuário deve usar esta seção para associar a cada token um identificador que será tratado pelo $\mathcal{SIC}$ como uma constante que especifica o tipo léxico do símbolo. Dentro do analisador léxico o usuário deverá usar estes identificadores para definir os valores retornados em YYSIMB. O analisador sintático somente reconhece os *tokens* listados nesta seção, referindo-se a eles por meio dos identificadores das constantes que definem os seus tipos léxico;

- YYSCAN deve ser um procedimento sem parâmetros que será chamado por YYPARSER, o analisador sintático, sempre que o próximo *token* no fluxo de entrada tiver que ser obtido. A cada chamada, YYSCAN deve retornar em YYSIMB o tipo do *token* lido; em YYLINHA o número da linha fonte que contém o *token* e em YYPOS a posição do *token* dentro da linha;

- os tipos terminais devem necessariamente ser aqueles especificados na seção de terminais, de forma a permitir a interpretação correta de valores em YYSIMB pelo analisador sintático. Caso o *token* possua um valor, este deverá ser atribuído a um de seus atributos;

3.2 Geradores de Analisadores Léxicos

A análise léxica é a primeira fase do processo de compilação e a única a interagir diretamente com a representação externa dos programas. Um analisador léxico converte uma sequência de caracteres que formam um programa em uma sequência de *tokens*. Embora a principal tarefa de um analisador léxico seja o reconhecimento dos *tokens* usados na análise sintática, outras funções podem ser executadas durante a análise léxica. Ele também é responsável pela eliminação de espaços e comentários [4].

O fato do analisador léxico ser um programa separado do analisador sintático torna as fases seguintes da compilação mais eficientes pois as mudanças na representação dos *tokens* podem ser confinadas apenas ao analisador léxico, sem impactos em outras partes do compilador. Além disso, algoritmos muito eficientes, porém pouco poderosos para tratar com a análise sintática, podem ser usados em reconhecedores léxicos, aumentando a eficiência final do compilador.

3.2.1 Gerador de Analisadores Léxicos $\mathcal{LEX}$

$\mathcal{LEX}$ [12] é um gerador de analisadores léxicos em ambiente Unix cujo objetivo é auxiliar o projetista de compiladores no desenvolvimento mais rápido de seu programa. A partir da especificação das unidades léxicas da linguagem na forma de expressões regulares e fragmentos de programas que

definem ações a serem tomadas, ele gera automaticamente um analisador léxico para a linguagem dada.

O $\mathcal{LEX}$ funciona da seguinte forma: a tabela de expressões regulares e fragmentos de programas são transformados em um programa que lê uma entrada, copiando e particionando a entrada em *strings* que marcam uma dada expressão. A reorganização das expressões é transformada em um autômato finito determinístico.

O programa gerado denomina-se yylex e é escrito na linguagem C [16, 15, 9].

O formato geral do $\mathcal{LEX}$ consiste em:

definições
%%
regras
%%
rotinas do usuário

onde:

- definições e as rotinas do usuário podem ser omitidas. A segunda ocorrência de %% é opcional, mas a primeira é obrigatória para marcar o início das regras;
- as regras representam as decisões de controle do usuário. Elas constituem uma tabela cuja coluna da esquerda contêm espressões regulares e a coluna da direita contêm ações e fragmentos de programas que são executados quando as expressões são usadas;
- uma expressão regular especifica um conjunto de *strings* para ser marcado. Ela contêm um texto de caracteres e caracteres operadores. As letras do alfabeto e os dígitos são sempre caracteres texto. Os operadores são: [ ], `^`, ‘‘, −, ?, `.` , ∗, +, |, (), `\,/,` { }, %, <> e quando usados como caracteres texto tem que vir entre aspas (“) ou com o operador \.
- Dados x,y e z representando caracteres quaisquer, as expressões regulares podem ter as seguintes formas:
  - x representa o próprio caractere x;
  - `‘‘x’’` representa um x, sempre que x for um operador;
  - `[xy]` representa o caractere x ou y;
  - `[x-z]` representa os caracteres entre x e z inclusive;
  - `[^x]` representa qualquer caractere exceto x;
  - `.` representa qualquer caractere exceto newline;

  - `^x` representa um x no começo da linha;
  - <y>x representa um x quando $\mathcal{LEX}$ é uma condição de começo y;
  - x$ representa um x no final da linha;
  - x? representa um x opcional;
  - x$*$ representa zero ou mais instâncias de x;
  - x+ representa uma ou mais instâncias de x;
  - x|y representa um x ou um y;
  - (x) representa um x;
  - `x/y` representa um x se e somente se seguido de y;
  - {xx} representa uma definição de xx da seção de definição;
  - {m,n} representa de m até n ocorrências de x.

- As definições contêm a combinação de:

  a) código incluído na forma %{ *code* %}, onde *code* é qualquer sequência válida de código em C;

  b) condições de começo, dado na forma %S *name1*, *name2*;

  c) definições, na forma de *"name space translation"*, onde *name* é uma *string* de substituição e a *translation* é qualquer expressão regular;

  d) mudanças para o tamanho de *arrays* internos na forma, %x nnn, onde nnn é um inteiro decimal representando um tamanho do *array* e x seleciona o parâmetro como segue:
    - p: positivo;
    - n: estados;
    - e: nodos de árvores;
    - a: translações;
    - k: pacotes de classes de caracteres;
    - o: tamanho de *array output*.

- As rotinas do usuário são constituídas de qualquer sequência de código em C e representam as ações do $\mathcal{LEX}$Pode ser definida pelo usuário ou utilizar a ação *default* do $\mathcal{LEX}$ que consiste em copiar o *input* para o *output*.

3.2.2 *Legible Denotational Semantics* LDS

LDS (*Legible Denotational Semantics*) [13] é um sistema baseado em Semântica Denotacional Legível que apresenta uma nova abordagem para a especificação formal.

A linguagem de especificação sintática em LDS é SSL (*Syntax Specification Language*). SSL é basicamente uma linguagem para formulação de gramáticas livres do contexto e apresenta uma notação semelhante a BNF, porém, SSL permite especificar separadamente os aspectos léxicos da linguagem, i.e., o reconhecimento das suas unidades textuais, *tokens*, e a correspondência entre a sintaxe abstrata e concreta de uma linguagem.

LDS recebe como entrada a especificação formal em e gera automaticamente um compilador para a linguagem dada. Nele a definição sintática é processada para obtenção dos analisadores léxico e sintático baseados em um algoritmo de análise LALR(1).

Uma descrição SSL está dividida em três partes: SYNTAX, DOMAINS e LEXIS. Estas partes determinam, respectivamente, os aspectos sintáticos, os domínios sintáticos e os aspectos léxicos da linguagem. As especificações sintática e léxica têm basicamente a mesma forma, um conjunto de produções. Essas produções indicam as regras de substituição de símbolos para composição de programa de linguagem.

SYNTAX e LEXIS constituem dois conjuntos de produções sintáticas. O conjunto LEXIS representa a parte de análise léxica, é iniciada pela palavra reservada LEXIS e descreve a micro-sintaxe e regras da formação de *token* da linguagem. Essa divisão leva a geração de dois analisadores distintos: um analisador trata dos aspectos léxicos e o outro analisador trata dos aspectos sintáticos da linguagem. O LEXIS identifica os *tokens* da linguagem a partir de uma sequência de caracteres transformando o programa fonte em uma lista de *tokens*.

As produções de LEXIS não têm valores *default*, pois os valores produzidos durante a análise léxica, que servem de comunicação entre os analisadores léxico e sintático, podem variar bastante. As expressões associadas às alternativas de UNIT, símbolo não-terminal da gramática, devem ser tuplas, mas nenhuma conversão foi estabelecida quanto aos componentes dessas tuplas. Os componentes das tuplas determinam os valores passados à análise sintática pelo analisador léxico gerado da definição de LEXIS.

Duas restrições são impostas pelo compilador SSL à forma das gramáticas especificadas em LEXIS. A gramática de LEXIS deve ser regular e essa restrição é necessária devido ao uso de propriedades de autômatos pelo algoritmo de geração de analisadores léxicos. Outra restrição está associada ao uso dos símbolos não-terminais na gramática. Essa restrição tem o propósito de determinar o critério de eliminação dos não-terminais. Para atender esta restrição é sugerido em [10], que o conjunto dos não-terminais da gramática seja particionado em três grupos:

1. grupo dos elementos de texto, *tokens*:
   as produções desses não-terminais fazem uso apenas de símbolos terminais, não-terminais do grupo 3 ou recursão direta;

2. grupo dos não-terminais que geram *tokens*:
   no caso do SSL, apenas UNIT pertence a esse grupo;

3. grupo dos não-terminais que geram sequências de símbolos terminais.

3.2.3 Gerador *Pipeline* de Analisadores Léxicos Multilex

Multilex [5] é um gerador projetado para facilitar a criação de analisadores léxicos, particularmente analisadores léxicos para analisadores sintáticos LALR(1) de uma dada linguagem. As características de inovação de Multilex incluem uma arquitetura *pipeline*, combinação de padrões léxicos para um grande número de objetos em vez de caracteres, reconfigurabilidade das linguagens que incluem sublinguagens e um mecanismo de dicionário de escopos léxicos.

Multilex provê padrões de expressões regulares para reconhecimento de padrão de nível de *token* e um *pipeline* para organizar os diferentes tipos de padrões a medida que são discriminados. Finalmente, Multilex provê facilidades particularmente convenientes para relembrar os tipos de identificadores na linguagem e para mudar os estilos léxicos e entradas léxicas.

Na descrição do Multilex, os elementos chaves são:

- executados sobre um número de objetos e não de caracteres.
- Os objetos tem valores e atributos que podem ser caracteres ou tipos mais complexos. Exemplos de atributos típicos são:
  - valor: descreve o valor base do objeto, sendo o mesmo usado para o padrão nas regras do tradutor;
  - tipo: descreve o tipo do valor;
  - objetos: descreve uma sequência de objetos combinados em padrões;
  - origem: descreve o nome do arquivo de origem ou função, ou a *string* corrente de entrada;
  - linha: descreve o número da linha deste objeto na origem;
  - coluna: descreve o número da coluna deste objeto na origem;
- combinação de padrões de expressões regulares sobre os objetos;
- o analisador léxico gerado é um tradutor *pipeline*. Cada etapa do tradutor lê uma série de objetos de entrada e produz uma série de objetos de saída para a próxima etapa do tradutor. Em uma certa etapa do tradutor, ele consome uma sequência, podendo ser vazia, de objetos de entrada, modifica seus valores e atributos, e produz uma sequência, podendo ser vazia, de objetos de saída. O fluxo entre as etapas do tradutor são colocadas em um *buffer*. Cada etapa do tradutor é uma série de pares de `ações` e `padrões`. Se o `padrão` de uma regra combina com a entrada do tradutor então a `ação` é executada;

- um dado tradutor no *pipeline* léxico pode temporariamente ou permanentemente reconstruir o *pipeline* para receber uma entrada alternativa ou executar um léxico alternativo. Este mecanismo é a idéia de expansão macro, incluindo arquivos, e gerando sub-linguagens;
- o analisador léxico inclui um mecanismo de dicionário para organização de escopo para análise dos dados;
- Uma sequência de um ou mais tradutores é definido como uma configuração. Um tradutor pode mudar uma configuração global e recolocá-la em outra configuração.

As limitações de Multilex são:

- não consegue resolver problemas como manter informações extra sintáticas como os comentários, linhas de origem e arquivos e macros originadas do código. Multilex não tem a resposta mágica para a integração destas informações na árvore sintática e a tentativa de incluí-las produz erros;
- é mais devagar com analisadores léxicos de estados simples, rodando vários estados;
- função *pipeline* é assíncrona, em particular, força decisões que são executadas mais cedo, ocasionando uma nova análise do *token*.

4 Desenvolvimento do Trabalho Proposto

4.1 Especificação do Gerador de Analisadores Léxicos $\mathcal{GAL}^{\mathcal{SIC}}$

Nesta seção é descrita a especificação de $\mathcal{GAL}^{\mathcal{SIC}}$, sistema projetado para especificar analisadores léxicos para linguagens de programação. $\mathcal{GAL}^{\mathcal{SIC}}$ segue o mesmo padrão utilizado no $\mathcal{LEX}$ para a especificação da entrada para o analisador léxico. A principal diferença entre esses dois sistemas é o ambiente para o qual cada um foi projetado. Além disso, o $\mathcal{LEX}$ opera em Unix e o $\mathcal{GAL}^{\mathcal{SIC}}$ opera em Windows. A linguagem de desenvolvimento do $\mathcal{LEX}$ é C; a linguagem de desenvolvimento de $\mathcal{GAL}^{\mathcal{SIC}}$ é Delphi. Associado ao sistema $\mathcal{GAL}^{\mathcal{SIC}}$ para descrever a sua especificação de entrada é utilizada uma linguagem $\mathcal{GAL}^{\mathcal{SIC}}$.

Uma especificação $\mathcal{GAL}^{\mathcal{SIC}}$ é constituída de três partes separadas por "%%" :

1. declarações.
2. regras de traduções.
3. procedimentos auxiliares.

As declarações são divididas em duas partes. A primeira parte apresenta definições do usuário em linguagem C entre os delimitadores "%{" e "%}" . Estes delimitadores são obrigatórios. A segunda apresenta definições de expressões regulares que o usuário associa a um nome. Estas definições são escritas em duas colunas, onde a primeira coluna representa o nome dado pelo usuário e a segunda coluna representa a expressão definida por ele.
Exemplo de declaração:

```
%{
#define $program 1
#define $integer 2
#define $char 3
%{
letra [a-zA-Z]
digito [0-9]
```

As regras de tradução são delimitadas pelo símbolo "%%" e representam as decisões de controle do usuário. Constituem uma tabela de duas colunas, onde a primeira coluna representa as expressões regulares que especificam um conjunto de *strings* que contêm caracteres e operadores para serem marcados e a segunda coluna representa ações e fragmentos de programas em linguagem C que são executados quando as expressões são reconhecidas.
Exemplo de regras de tradução:

```
%%
program   {printf("reservada:%d%s\n" ,$program,yytext);}
[0-9]+    {printf("integer" );}
"+"       {printf("operadormais" );}
%%
```

As expressões regulares representadas na primeira coluna são constituidas de:

- as letras do alfabeto e os dígitos são sempre caracteres texto;
- os operadores de $\mathcal{GAL}^{\mathcal{SIC}}$ são: [ ], ’’, −, ?, ∗, +, |, ( ), \,+ {, }, % e quando usados como caracteres texto em que vir entre (“) ou com o operador \;
- `x` representa o próprio caractere `x` quando é uma letra do alfabeto `e/ou` dígito;
  Exemplo:

program {printf("reservada:%d%s\n" , $program,yytext);}

- `x` ou \x representa o próprio caractere `x` quando este é um operador; Exemplo:

  `"+"` {printf("operadormais");}

- `[xy]` representa o caractere `x` ou `y`. O par de operadores [ ] define exclusão e há dois tipos de operadores especiais: `"\"` e `"−"` . O operador `"−"` é usado entre dois pares de caracteres que podem ser letras ou dígitos e significa que a exclusão é feita do primeiro caractere até o segundo caractere inclusive. O operador `"\"` é definido para caracteres especias como: \t significa TAB, \b significa espaço em branco, \n significa NEWLINE, \f significa LINEFEED e \v significa TAB-VERTICAL; Exemplo:

`[a-zA-Z]`	{printf("caracterletra");}
`[\b\t\n\f\v]`	{ /∗faz nada ∗/ }

- (x)? representa um x opcional. O operador `"?"` significa que o caractere é opcional na expressão regular. O parênteses é obrigatório mesmo que seja apenas um caractere. Este operador também pode vir após o par de operadores [ ];
  Exemplo:

`[a-zA-Z]?`	{printf("letra opcional");}
`"+"` (a)?abc	{printf("expressão reservada"); }

- (x)∗ representa zero ou mais instâncias de x. O parênteses é obrigatório mesmo que seja apenas um caractere e também pode vir após o par de operadores [ ];
  Exemplo:

`[a-zA-Z]`∗	{printf("zero ou mais letras"); }
abc(m)∗	{printf("palavra reservada"); }

- (x)+ representa uma ou mais instâncias de x. O parênteses é obrigatório mesmo que seja apenas um caractere e também pode vir após o par de operadores [ ];

- x|y representa um x ou um y. O operador | significa exclusão;

- {xx} representa uma definição xx na seção de definição. É usado quando se quer referir a uma expressão regular de nome xx definida na seção de definição;

Exemplo:

```
letra [a-zA-Z]
...
{letra} {printf("qualquer letra" ); }
```

Os procedimentos auxiliares são códigos em linguagem C que o usuário define.
Exemplo de procedimentos auxiliares:

```
%%
yywrap()
{
int i;
printf("Tamanho do numero de palavras \n" );
for(i=0;i<100;i++)
if(lengs[i]>0)
printf("%5d%10d\n" ,i,lengs[i]);
return(1);
}
```

Tanto as declarações quanto os procedimentos auxiliares podem ser omitidos. As regras de tradução e seus delimitadores são obrigatórios. Se as declarações não são omitidas então seus delimitadores são obrigatórios.

Para criar um analisador léxico usando o sistema $\mathcal{GAL}^{\mathcal{SIC}}$ o projetista de compiladores deve escrever um programa fonte, GAL.1, na linguagem $\mathcal{GAL}^{\mathcal{SIC}}$ como apresentado no início dessa seção. Em seguida, GAL.1 é compilado pelo $\mathcal{GAL}^{\mathcal{SIC}}$ e produz como resultado um programa na linguagem C. Este programa é denominado gal.yy.c e representa a versão tabular de um diagrama de transições abtido a partir das expressões regulares de GAL.1, juntamente com um procedimento padrão que utiliza a tabela a fim de reconhecer os lexemas.

As ações semânticas associadas às expressões regulares em GAL.1 são copiadas diretamente em gal.yy.c.

Depois dessa etapa, gal.yy.c é compilado por um compilador C e produz como resultado um programa objeto YYSCAN.out que é o analisador léxico, cujo objetivo é ler os caracteres de entrada e os transformar em uma sequência de *tokens*.

A Figura 1 adaptada de [3] ilustra a arquitetura do sistema.

A seguir é apresentado na Figura 2 a especificação de um analisador léxico em $\mathcal{GAL}^{\mathcal{SIC}}$ para reconhecer *tokens*.

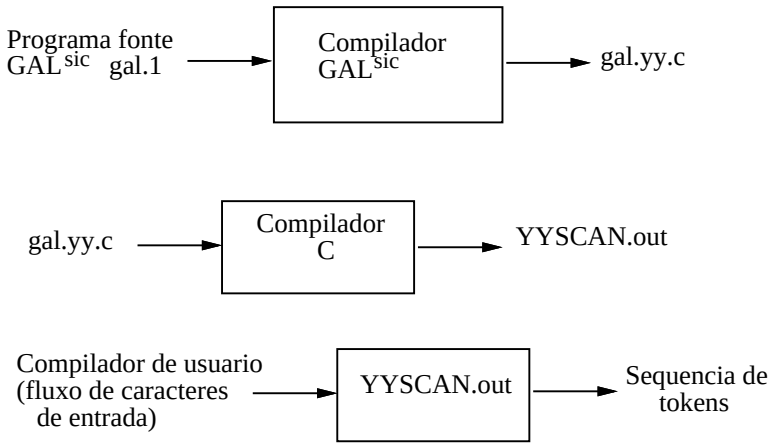


Figure 1: Arquitetura de $\mathcal{GAL}^{\mathcal{SIC}}$

4.2 Metodologia de Implementação

4.2.1 Introdução

Em relação a metodologia de desenvolvimento foram utilizados os algoritmos do livro *Compilers, Principles, Techniques and Tools* [3] com algumas modificações para atender ao nosso propósito e adequar ao sistema $\mathcal{SIC}$. Também foram utilizadas as técnicas empregadas no Gerador de Analisadores Léxicos $\mathcal{LEX}$ [**?**].

Dado o arquivo de entrada do gerador de analisadores léxicos contendo expressões regulares, as mesmas são transformadas em um diagrama de transições generalizado chamado autômato finito. Este autômato é uma estrutura de dados capaz de reconhecer precisamente os conjuntos regulares de uma linguagem. Ele pode ser: não-determinístico ou determinístico. No primeiro, mais de uma transição deixa um estado para o mesmo símbolo de entrada. No segundo somente é permitida uma transição para fora de um estado para o mesmo símbolo de entrada.

Normalmente em sistemas geradores de analisadores léxicos, as expressões regulares são transformadas em autômatos finitos não-determinísticos pois estes autômatos finitos são mais simples de serem construídos e podem ser menores do que o seu correspondente autômato determinístico (AFD).

Autômatos finitos não-determinísticos (AFN) são modelos matemáticos que consistem em:

1. um conjunto de estados S;

2. um conjunto de símbolos de entrada;

Expressao Regular	Token	Valor de Atributo
if	**if**	-
then	**then**	-
program	**program**	-
<	**relop**	LT
<=	**relop**	LE
<>	**relop**	NE

Figure 2: Analisador léxico $\mathcal{GAL}^{\mathcal{SIC}}$

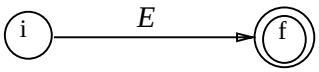
3. uma função de transição que mapeia pares `estado` e `símbolo` em conjuntos de estados;

4. um estado So que é caracterizado como estado de partida ou inicial;

5. um conjunto de estados F caracterizados como estados de aceitação ou finais.

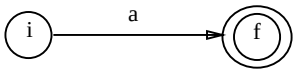
Para se construir o autômato finito não-determinístico usando uma expressão regular `"r"` foi utilizado o algoritmo denominado *Construção de Thompson* [3]. Este algoritmo utiliza duas variaáveis `i` e `j`, onde `i` corresponde ao estado inicial e `f` corresponde ao estado de aceitação. Neste algoritmo são identificados três situações:

1. Para o símbolo vazio $\mathcal{E}$, constroi-se o AFN:

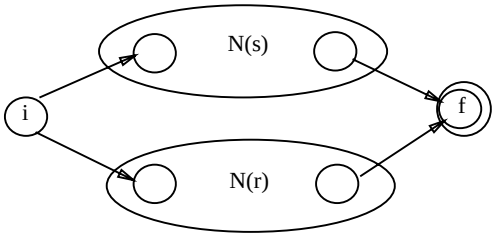


2. Para um símbolo qualquer `"a"` , constroi-se o AFN:

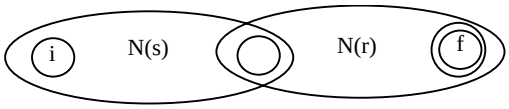
3. Suponha que N(s) e N(t) sejam os AFNs para as expressões regulares s e t.



(a). Para a expressão regular N(s)|N(t), é construido o seguinte AFN composto:



(c) Para a expressão regular N(s)N(t), é construido o seguinte AFN composto:



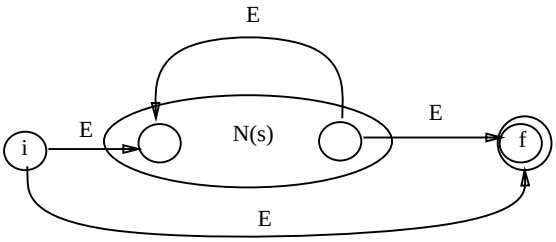
(b) Para a expressão regular N(s)$*$, é construido o seguinte AFN composto:

Um dos problemas encontrados no uso dos autômatos finitos não-determinísticos está relacionado com o fato de que a função de transição para certas entradas podem ser multiavaliadas, causando ambiguidade e consequentemente tornando difícil a simulação em programas. Uma forma de solucionar este problema é transformá-lo em um autômato finito determinístico, dado que este possui no máximo uma função de transição a partir de cada estado para qualquer símbolo de entrada.

Autômatos finitos determinísticos (AFD) são portanto um caso especial dos autômatos finitos não-determinísticos, no qual:

1. nenhum estado possui uma transição vazia;
2. para cada estado "S" e símbolo de entrada "a" existe no máximo uma aresta rotulada "a" deixando "S" .

Para construir o autômato finito determinístico a partir de um autômato finito não-determinístico foi utilizado o algoritmo denominado *Construção de Subconjuntos* [3]. Este algoritmo é constituido das seguintes operações:



- fechamento-vazio(S):

  conjunto de estados do AFN atingíveis a partir de um estado "S" somente por meio de transições vazias;

- fechamento-vazio(T):

  conjunto de estados do AFN atingíveis a partir de um estado "S" , pertencentes ao conjunto "T" , somente em transições vazias;

- movimento(T,a):

  conjunto de estados do AFN para o qual existe uma transição no símbolo de entrada "a" , a partir de algum estado "S" do AFN pertencente ao conjunto "T" .

Cada entrada no AFN é um conjunto de estados e cada entrada no AFD é exatamente um único estado. A idéia geral do algoritmo é que cada estado do AFD corresponde a um conjunto de estados do AFN, controlando todos os possíveis estados que o AFN poderia estar após ler cada símbolo de entrada. Isto significa que após ler a entrada, o AFD estará em um estado que representa o subconjunto "T" dos estados do AFN que são atingíveis a partir do estado inicial do AFN.

4.2.2 Estruturas de Dados

Para representar o autômato finito não-determinístico foi utilizada uma tabela de transição, consistindo em uma linha para cada estado e uma coluna para cada símbolo de entrada incluindo o símbolo vazio. Para cada item desta tabela têm-se o número de estados juntamente com os estados armazenados. A Figura 3 mostra o formato da `Tabela de Transição`.

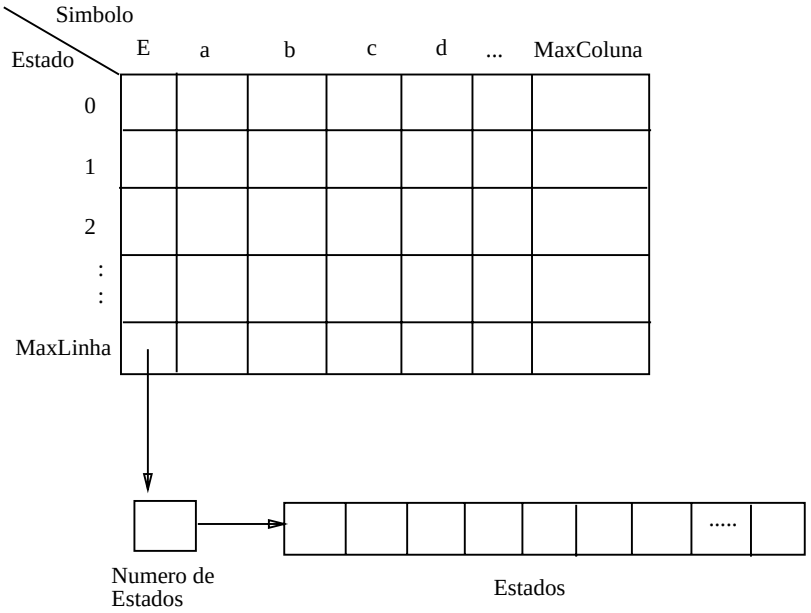


Figure 3: Estrutura de dados *Tabela de Transição*

As Figuras 4 e 5 mostram respectivamente um exemplo de um autômato finito não-determinístico e como ficaria sua representacção nesta `tabela de transição`.

Para transformar um AFN em um AFD, é necessário utilizar duas estruturas de dados auxiliares: *EstadosD* e *Pilha*. Cada item do *EstadosD* representa um conjunto de estados do AFN. Esta estrutura contém o número de estados no conjunto e uma marcação. A *Pilha* é utilizada para percorrer todos os estados do AFN. A Figura 6 apresenta o formato da estrutura *EstadosD*.

A Figura 7 ilustra o formato de *Pilha*:

Para representar um autômato finito determinístico foi utilizado uma tabela de transição chamada *Dtran*, onde cada linha representa um estado e cada coluna representa um símbolo de entrada. A Figura 8 ilustra esta estrutura.

Dado o exemplo do AFN apresentado na Figura 4 o seu correspondente AFD é ilustrado na Figura 9

e sua representação utilizando a estrutura de dados *Dtran*, é mostrada na Figura 10.

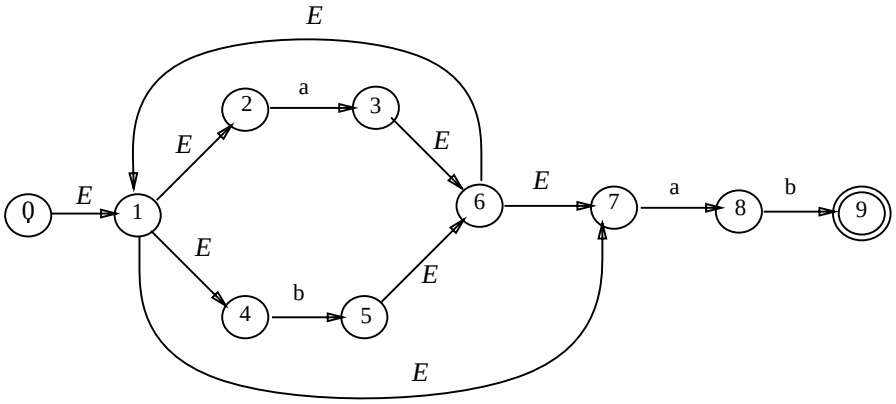


Figure 4: Exemplo de Autômato Finito Não-Determinístico

4.2.3 Implementação

Para a estrutura de dados *Tabela de Transição* definida na Figura 3 foi utilizado uma matriz com as seguintes dimensões: $0 \cdots$ MaxLinhaTab por $0 \cdots$ MaxColunaTab, onde MaxLinhaTab e MaxColunaTab são constantes com valores 20 e 30 respectivamente. Esta estrutura armazena registros que representam os estados possíveis e possui os seguintes campos:

- `Estado`: variável inteira;
- `NumeroDeEstados`: variável inteira;

Para a estrutura de dados *Dtran* definida na Figura 8 foi utilizada uma matriz com as seguintes dimensões: $0 \cdots$ `MaxLinhaDtran` por $0 \cdots$ `MaxColunaDtran`, onde `MaxLinhaDtran` e `MaxColunaDtran` são constantes com valores 100 e 260 respectivamente. Esta estrutura armazena inteiros que representam os estados.

Uma das dificuldades encontradas na implementação das matrizes que é não ser possível manter as duas estruturas *Tabela de Transição* e *Dtran* com todas as linhas e colunas possíveis, isto é, a *Tabela de Transição* também contendo as dimensões : 100 para a linha e 260 para a coluna, isto porque acarretaria *overflow* de espaço. Como esta estrutura é uma estrutura intermediária, a solução encontrada foi diminuir as suas dimensões e criar um vetor para armazenar o símbolo o qual aquela coluna na *Tabela de Transição* se refira. Este vetor é chamado de *TabelaColuna* e as suas dimensões são: $1 \cdots$ `MaxColunaTab`. Ele armazena inteiros que representam os valores em Código ASCII do símbolo de entrada.

Para a estrutura de dados *EstadosD* definida na Figura 6 foi utilizado um vetor que vai de $0 \cdots$ `Maximo`, onde `Maximo` é uma constante de valor 10.

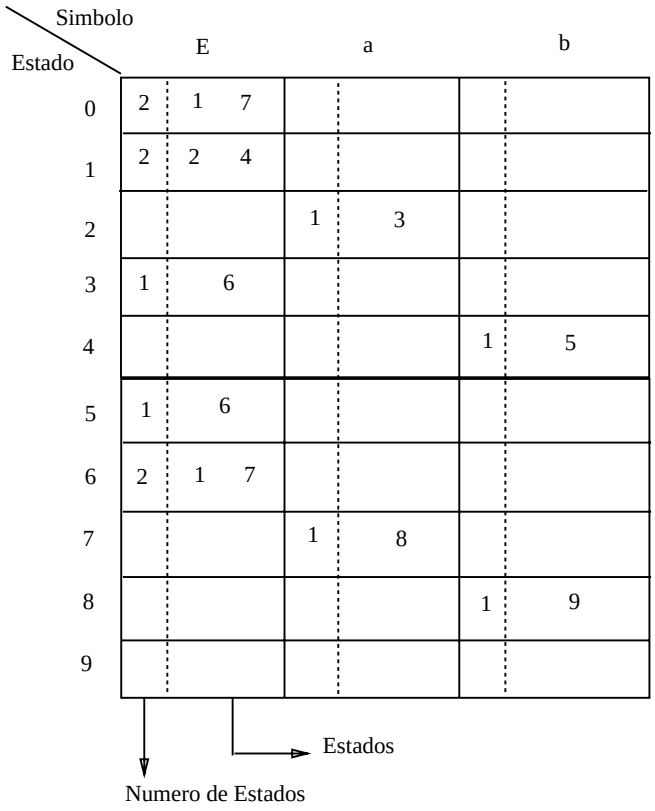


Figure 5: Resultado do Exemplo apresentado na Figura 4

Esta estrutura armazena conjuntos declarados como registros cujos campos são:

- `Marcado`: variável lógica.
- `Conj`: vetor de $1 \cdots$ Maximo que armazena inteiros.
- `NumeroElementos`: variável inteira.

Para a estrutura de dados *Pilha* definida na Figura 7 foi utilizado um vetor que vai de $1 \cdots$ `Maximo`. Esta estrutura armazena inteiros que representam os estados. Ela foi necessária para varrer todo o autômato não-determinístico e construir os *EstadosD*.

Outra dificuldade encontrada durante a implementação está relacionada com a definição de todas as dimensões das estruturas utilizadas. Isto só foi possível realizando vários testes com vários arquivos de entrada.

O algoritmo do fechamento definido em 4.2.1 foi o mais complexo e o que deu mais trabalho pois este continha muitos detalhes de execução e várias estruturas de dados envolvidas.

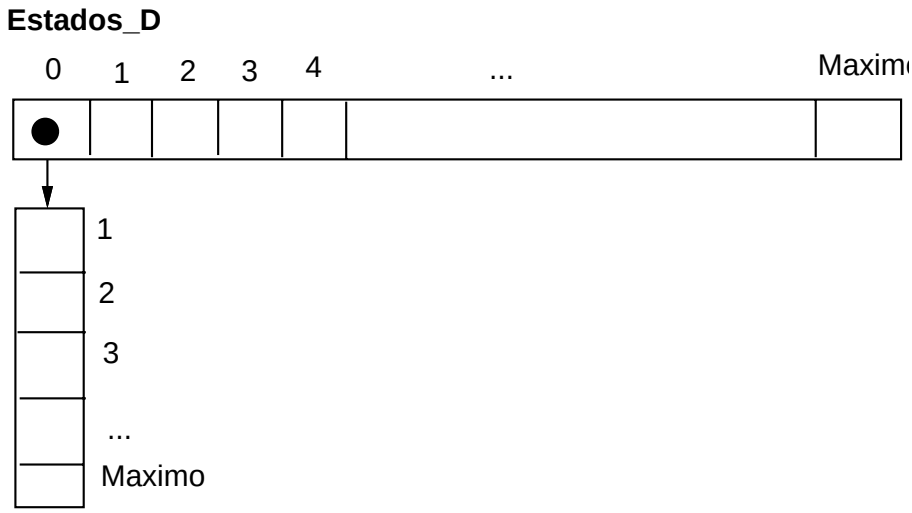


Figure 6: Estrutura de dados *EstadosD*

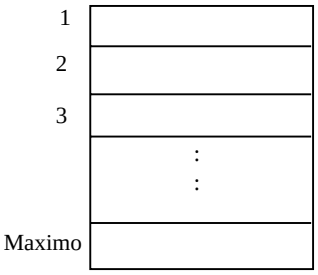


Figure 7: Estrutura de dados *Pilha*

4.3 Comparação entre $\mathcal{LEX}$ e $\mathcal{GAL}^{\mathcal{SIC}}$

Uma das principais diferenças entre $\mathcal{LEX}$ e $\mathcal{GAL}^{\mathcal{SIC}}$ é que $\mathcal{LEX}$ opera em ambiente Unix e sua linguagem de desenvolvimento é C enquanto que $\mathcal{GAL}^{\mathcal{SIC}}$ opera em ambiente Windows e sua linguagem de desenvolvimento é Delphi.

Uma das principais semelhanças é que ambos geram analisadores léxicos na linguagem C e apresentam semelhanças quanto ao arquivo de entrada como especificado nas seções 3.2.1 e 4.1.

4.4 Interface com o Sistema de Implementação de Compiladores: $\mathcal{SIC}$

A interação entre o Analisador léxico YYSCAN gerado pelo $\mathcal{GAL}^{\mathcal{SIC}}$ e o Analisador sintático YYPARSER gerado pelo $\mathcal{SIC}$ deve-se principalmente

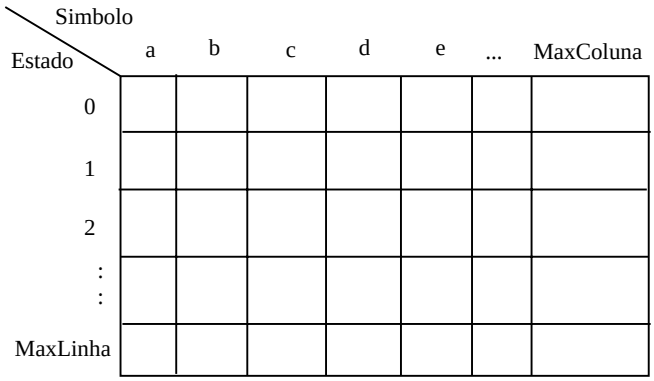


Figure 8: Estrutura de dados *Dtran*

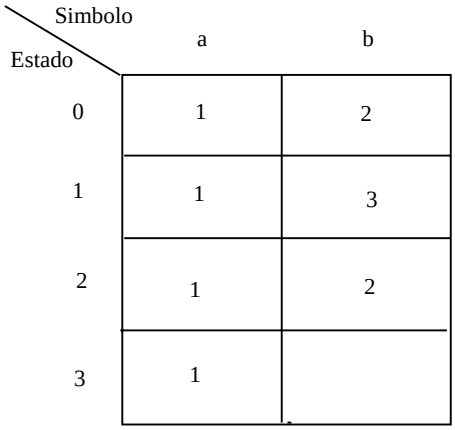


Figure 9: Representação de *Dtran* para o exemplo apresentado na Figura 4

por três variáveis globais em $\mathcal{SIC}$ que são:

- YYSIMB: retorna o tipo do símbolo reconhecido pelo analisador léxico;
- YYPOS: retorna a posição do último símbolo reconhecido pelo analisador léxico;
- YYLINHA: número corrente da linha que contém o último *token* lido pelo analisador léxico;

Dentro do esboço do $\mathcal{SIC}$ temos uma seção chamada Seção de Terminais onde é estabelecido um mecanismo de comunicação entre o analisador sintático YYPARSER gerado pelo $\mathcal{SIC}$ e o analisador léxico YYSCAN gerado pelo $\mathcal{GAL}^{\mathcal{SIC}}$Nesta seção, o usuário deve associar a cada *token* um identificador que será tratado pelo $\mathcal{SIC}$ como uma constante que especifica o

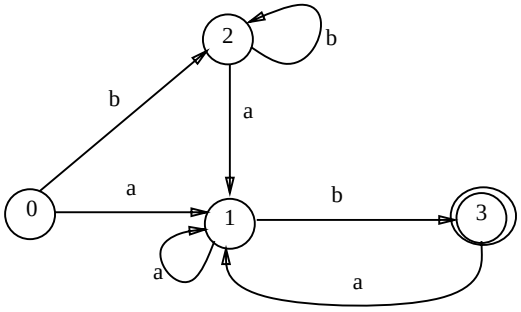


Figure 10: Autômato Finito resultante da Estrutura de Dados *Dtran*

tipo léxico do símbolo e dentro do analisador léxico o usuário deverá usar estes identificadores para definir os valores retornados em YYSIMB.

O analisador léxico YYSCAN gerado deve ser um procedimento sem parâmetros que será chamado pelo analisador sintático YYPARSER e a cada chamada deve retornar em YYSIMB o tipo do *token* lido, em YYLINHA o número da linha fonte que contém o *token* e em YYPOS a posição do *token* dentro da linha. Caso o *token* possua um valor, este deverá ser atribuído a um de seus atributos e isto deve ser tratado pelo YYSCAN.

5 Conclusão

Concluído este projeto temos um sistema completo em ambiente Windows NT que serve como ferramenta adequada à escrita de compiladores, em particular, de *front end* de compiladores.

Este sistema completo gera um *front end* de compiladores em linguagem C que o usuário especifica de acordo com as regras dadas na seção 4.1 e no manual do usuário dado em [7].

As contribuições dadas pela finalização do projeto são:

- Especificação de um gerador de analisadores léxicos.
- Implementação de um gerador de analisadores léxicos no ambiente Windows NT.
- Ferramenta de auxílio ao implementador de linguagens.
- Integração do GAl^{sic} com o gerador de analisadores sintáticos.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing Translation and Compiling*. Prentice Hall, ICN, 1972.

[2] A. V. Aho and J. D. Ullman. *Deterministic Parsing of Ambiguous Grammars*. Comm. Assoc. Comp. Mach., 1977.

[3] A. V. Aho, J. D. Ullman, and R. S. Sethi. *Compilers, Principles, Techniques and Tools*. Addison Wesley Publishing Company Co., 1986.

[4] A.V.Aho, R.Sethi, and J.D.Ullman. *Compilers:Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[5] Timothy Bickmore and Robert E. Filman. *Multilex, A Pipelined Lexical Analyzer. Software Technology Analyzer*, 1997.

[6] Mariza A. S. Bigonha and Roberto S Bigonha. *SIC – Sistema de Implementação de Compiladores – Versão 4.0*. Relatório Técnico 007/89, Departamento de Ciência da Computação, UFMG, 1989.

[7] Mariza A. S. Bigonha et al. *SIC: Sistema de Implementação de Compiladores - Manual do Usuário, Versão C 1.0* . Relatório Técnico 020/96, Departamento de Ciência da Computação, UFMG, 1996.

[8] Charles Calvert. *Delphi 2 Unleashed*. Sams, 1996.

[9] H.M. Deitel and P.J. Deitel. *How to Program*. Prentice Hall, 1994.

[10] J.Eickel F.L.DeRemmer, F.L.Bauer. *Compiler Construction: An Advanced Course, Lecture Notes in Computer Science*. Spring-Vergal, Berlin, 1974.

[11] Stephen C. Johnson. *YACC - Yet Another Compiler Compiler*. Technical Report 07974, Bell Laboratories, Murray Hill, New Jersey.

[12] Stephen C. Johnson. *Lex - A Lexical Analyser Generator*. Technical Report No39, Bell Laboratories, Murray hill, New Jersey, 1975.

[13] José da Silva Leite Júnior. *Linguagem de Definição e Geração de Analisadores Sintáticos em Semântica Denotacional Legível*. Master's thesis, Universidade Federal de Minas Gerais, Março 1993.

[14] Vicent Kellen and Bill Todd. *Delphi 2 : A Developer's Guidel*. M&T Books, 1996.

[15] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[16] Robert Sedgewick. *Algorithms in C*. Addison - Wesley, 1990.

A Especificação de um Analisador Léxico usado como Entrada para $\mathcal{GAL}^{SIC}$

```
%{
/* palavras reservadas */
/* e opcional */
#define $program 1
#define $integer 2
#define $real 3
#define $boolean 4
#define $char 5
#define $array 6
#define $of 7
#define $begin 8
#define $end 9
#define $if 10
#define $then 11
#define $else 12
#define $repeat 13
#define $while 14
#define $until 15
#define $read 16
#define $goto 17
#define $procedure 18
#define $nao 19

/* Outros tokens */
#define $ident 21
#define $addop 22
#define $relop 23
#define $mulop 24
#define $constalfanum 25
#define $constint 26
#define $constreal 27
#define $constbool 28
#define $atribuicao 29
#define $doispontos 30
#define $pontovirgula 31
#define $virgula 32
#define $fechapar 33
#define $abrepar 34

/* tipos de constbool*/
```

```
#define $false 1
#define $true 2

/* tipos de addop */
#define $mais 1
#define $menos 2
#define $ou 3

/* tipos de reolop */
#define $igual 1
#define $menorque 2
#define $menorigual 3
#define $maiorque 4
#define $maiorigual 5
#define $diferente 6

/* tipos de mulop */
#define $vezes 1
#define $dividido 2
#define $mod 3
#define $div 4
#define $e 5
%}

brancos [\b\t\v\n\f]
digito [0-9]
letra [a-zA-Z]
expoente [Ee][+-]?{digito}+
intsemsinal {digito}{digito}*

%%
"*"  {printf("ope: %d %s\n" ,$vezes,YYSIMB);}
":"  {printf("ope: %d %s\n" ,$dividido,YYSIMB);}
"+"  {printf("ope: %d %s\n" ,$mais,YYSIMB);}
"-"  {printf("ope: %d %s\n" ,$menos,YYSIMB);}
"<"  {printf("ope: %d %s\n" ,$menorque,YYSIMB); }
">"  {printf("ope: %d %s\n" ,$maiorque,YYSIMB); }
"<="  {printf("ope: %d %s\n" ,$menorigual,YYSIMB); }
">="  {printf("ope: %d %s\n" ,$maiorigual,YYSIMB); }
"="  {printf("ope: %d %s\n" ,$igual,YYSIMB); }
"<>"  {printf("ope: %d %s\n" ,$diferente,YYSIMB); }
program {printf ("reservada: %d %s\n" ,$program,YYSIMB);}
procedure {printf ("reservada: %d %s\n" ,$procedure,YYSIMB);}
integer {printf ("reservada: %d %s\n" ,$integer,YYSIMB);}
```

```
real {printf ("reservada: %d %s\n" ,$real,YYSIMB);}
char {printf ("reservada: %d %s\n" ,$char,YYSIMB);}
boolean {printf ("reservada: %d %s\n" ,$boolean,YYSIMB);}
array {printf ("reservada: %d %s\n" ,$array,YYSIMB);}
begin {printf ("reservada: %d %s\n" ,$begin,YYSIMB);}
end {printf ("reservada: %d %s\n" ,$end,YYSIMB);}
if {printf ("reservada: %d %s\n" ,$if,YYSIMB);}
not|Not|NOT {printf("operador de negacao: %d \n" ,$nao,YYSIMB);}
{brancos}+ { /* faz nada */ }
{letra}({letra}|{digito})* {printf("identificador: %d %s\n" ,$ident,YYSIMB);}
{intsemsinal} {printf("constante inteira: %d %s\n" ,$constint,YYSIMB);}
%%
```

B Analisador Léxico gerado pelo $\mathcal{GAL}^{SIC}$

```
# include "stdio.h"
int ll,cc;
char ch;
#define ENDFILE EOF
#define NEWLINE 13
#define HTAB 9
#define LINEFEED 10
#define CTLZ 26
#define BACKSPACE 8
#define VT 11
#define FF 12
#define TLE 72
#define MaxlinhaDtran 100
FILE *FONTE;
char linha[TLE+1];
    /* palavras reservadas */
    /* e opcional */
    #define $program 1
    #define $integer 2
    #define $real 3
    #define $boolean 4
    #define $char 5
    #define $array 6
    #define $of 7
    #define $begin 8
    #define $end 9
    #define $if 10
    #define $then 11
    #define $else 12
    #define $repeat 13
    #define $while 14
    #define $until 15
    #define $read 16
    #define $goto 17
    #define $procedure 18
    #define $nao 19

    /* Outros tokens */
    #define $ident 21
    #define $addop 22
    #define $relop 23
    #define $mulop 24
```

```
    #define $constalfanum 25
    #define $constint 26
    #define $constreal 27
    #define $constbool 28
    #define $atribuicao 29
    #define $doispontos 30
    #define $pontovirgula 31
    #define $virgula 32
    #define $fechapar 33
    #define $abrepar 34

    /* tipos de constbool*/
    #define $false 1
    #define $true 2

    /* tipos de addop */
    #define $mais 1
    #define $menos 2
    #define $ou 3

    /* tipos de reolop */
    #define $igual 1
    #define $menorque 2
    #define $menorigual 3
    #define $maiorque 4
    #define $maiorigual 5
    #define $diferente 6

    /* tipos de mulop */
    #define $vezes 1
    #define $dividido 2
    #define $mod 3
    #define $div 4
    #define $e 5
    YYSCAN(){
int numstr;
while((numstr = YYGetChar()) >= -1){
switch(numstr){
case 1: {printf("ope: %d %s\n" ,$vezes,YYSIMB);}
break;
case 3: {printf("ope: %d %s\n" ,$dividido,YYSIMB);}
break;
case 5: {printf("ope: %d %s\n" ,$mais,YYSIMB);}
break;
```

```
case 7: {printf("ope: %d %s\n" ,$menos,YYSIMB);}
break;
case 9: {printf("ope: %d %s\n" ,$menorque,YYSIMB); }
break;
case 11: {printf("ope: %d %s\n" ,$maiorque,YYSIMB); }
break;
case 14: {printf("ope: %d %s\n" ,$menorigual,YYSIMB); }
break;
case 17: {printf("ope: %d %s\n" ,$maiorigual,YYSIMB); }
break;
case 19: {printf("ope: %d %s\n" ,$igual,YYSIMB); }
break;
case 22: {printf("ope: %d %s\n" ,$diferente,YYSIMB); }
break;
case 30: {printf ("reservada: %d %s\n" ,$program,YYSIMB);}
break;
case 40: {printf ("reservada: %d %s\n" ,$procedure,YYSIMB);}
break;
case 48: {printf ("reservada: %d %s\n" ,$integer,YYSIMB);}
break;
case 53: {printf ("reservada: %d %s\n" ,$real,YYSIMB);}
break;
case 58: {printf ("reservada: %d %s\n" ,$char,YYSIMB);}
break;
case 66: {printf ("reservada: %d %s\n" ,$boolean,YYSIMB);}
break;
case 72: {printf ("reservada: %d %s\n" ,$array,YYSIMB);}
break;
case 78: {printf ("reservada: %d %s\n" ,$begin,YYSIMB);}
break;
case 82: {printf ("reservada: %d %s\n" ,$end,YYSIMB);}
break;
case 85: {printf ("reservada: %d %s\n" ,$if,YYSIMB);}
break;
case 89: {printf("operador de negacao: %d \n" ,$nao,YYSIMB);}
break;
case 93: {printf("operador de negacao: %d \n" ,$nao,YYSIMB);}
break;
case 97: {printf("operador de negacao: %d \n" ,$nao,YYSIMB);}
break;
case 103: { / * faz nada* / }
break;
case 155: {printf("identificador: %d %s\n" ,$ident,YYSIMB);}
break;
```

```
case 168: {printf("constante inteira: %d %s\n" ,$constint,YYSIMB);}
break;
}
}
}

int YYESTINICIAL[] = {
0,
2,
4,
6,
8,
10,
12,
15,
18,
20,
23,
31,
41,
49,
54,
59,
67,
73,
79,
83,
86,
90,
94,
98,
104,
110
};

int YYDTRAN[][] = {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
```

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,9,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,11,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,19,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,21,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,22,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,25,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,26,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,28,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,29,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,33,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,34,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,36,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,37,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,38,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,39,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,40,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,43,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,44,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,46,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,48,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,51,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,53,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,55,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,57,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,58,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,62,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,63,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,65,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,70,0,0 ,0,0,0,0,0,0,0,0,

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,73,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,75,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,77,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,81,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,85,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,00,0,0,0,0,0,0,0,0,0,0,0,87,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,91,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,93,0,0,0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,96,0,0,0,0,0,0,0,0,0 ,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,100,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,103,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,105,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,106,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,108,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,109,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,
```

```
    int YYGetChar(){
int i,estado;

ch = ' ';
ll = 0; cc = 0;
while ((ch == '')||(ch == HTAB)||(ch == NEWLINE)||
(ch == CTLZ)||(ch == BACKSPACE)||(ch == VT)||
(ch == FF)||(ch == LINEFEED)) {
if(cc==ll){
if(feof(FONTE)){
ch = ENDFILE;
return -1;
}
ll = 1;
cc = 0;
fscanf(FONTE,"%c" ,&linha[ll];
while (linha[ll] != '\n'){
ll++;
fscanf(FONTE,"%c" ,&linha[ll];
}
ll++;
linha[ll] = NEWLINE;
fscanf(FONTE,"\n" );
YYLINHA++;
}
cc++;
ch = linha[cc];
YYPOS = cc;
```

```
}
}
if (ch == ENDFILE) YYSIMB = YYEOF;
for(i=0;i<=MaxLinhaDtran;i++){
estado = YYDTRAN[i][ch];
if (estado != 0) break;
}
i = estado;
while (ch != BACKSPACE){
if(cc==ll){
if(feof(FONTE))
ch = ENDFILE;
ll = 1;
cc = 0;
fscanf(FONTE,"%c" ,&linha[ll];
while (linha[ll] != '\n'){
ll++;
fscanf(FONTE,"%c" ,&linha[ll];
}
ll++;
linha[ll] = NEWLINE;
fscanf(FONTE,"\n" );
YYLINHA++;
}
cc++;
ch = linha[cc];
YYPOS = cc;
}
}
estado = YYDTRAN[i][ch];
}
return estado;
}
```