UNIVERSIDADE FEDERAL DE PELOTAS
Cetro de Desenvolvimento Tecnológico
Ciência da Computação

DOCUMENTAÇÃO TRABALHO DE LINGUAGENS FORMAIS
Analisador Léxico e Sintático de Pseudo-Pascal

Autores:
Alex Borges
Maurício Balboni
Renata Sarmento

Orientadora:
Luciana Foss

SUMÁRIO

1. INTRODUÇÃO

Segundo as especificações repassadas pela definição do trabalho (presente no AVA), o trabalho foi desenvolvido seguindo-se as regras estipuladas, que serão abordadas ao longo deste documento. Basicamente, um arquivo escrito em pseudo-pascal, deve ser desenvolvido e salvo na pasta do projeto do analisador, salvo com o nome de “pascal.pas”, e no terminal, ao executar o “./lex”, o programa executará a análise léxica, verificando se o código utiliza os simbolos permitidos pelo vocabulário, se sim, gera uma biblioteca dos tokens utilizados no arquivo pascal. Depois, executa a análise sintática, utilizando como base a biblioteca de tokens, verificando se todos os tokens foram utilizados de maneira correta.

Em caso de algum erro, retorna uma mensagem apropriada, senão, avisa que o mesmo está correto.

O documento está dividido em setores, primeiro abordamos o funcionamento dos códigos, depois apresentamos a tabela de simbolos utilizada pelos analisadores, e, por fim, os automatos utilizados para confecção do programa.

2. VOCABULÁRIO

Como qualquer linguagem, existem simbolos que são aceitos, a apartir desses simbolos, se controi as palavras que serão válidas dentro dessa linguagem. Todos os simbolos aceitos pela linguagem do pseudo-pascal são: 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm', 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '.', '_', '(', ')', ';', ':', '+', '-', '=', '<', '>', '*', '/', ' ,'.

Contudo a linguagem possui subvocabulário, cada um representando um dos três principais grupos de palavras aceitas, são elas:

Nomes = 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm', '_', 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '0', '9', '8', '7', '6', '5', '4', '3', '2', '1';

Números = '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '.';

Caracteres Especiais = '(', ')', ':', ';', '=', '<', '>', '-', '+', '*', '/', ',';

Há uma lógica interna que separa cada grupo de tokens em um dos tipos de sub-vocabulário, como será visto ao longo deste.

3. CÓDIGOS

3.1 Main.c

No arquivo “main.c”, há pequenas tarefas, entre elas, a de abrir o arquivo “pascal.pas” e aparecer a mensagem de sucesso. As demais, são delegadas à biblioteca “function.h”, que têm como base a leitura do arquivo aberto.

3.2 Function.h

Esta biblioteca contêm as funções necessárias para realizar as análises, cada uma das funções serão argumentados, vide mais informações diretamente no código.

3.2.1. Funções de Análise Léxica

As funções apresentadas nessa sessão, contribuem para a criação da tabela de simbolos, seguindo-se para tanto, uma série de processos, que pouco a pouco, transformam os dados lidos no arquivo externo em uma sequência de tokens, e por seguinte, em um array de caracteres padrão, isso é, os tokens em si.

3.2.1.1. lerFile(char[], File*)

Esta função possui como parâmetro um array e um arquivo externo. O array serve para receber as informações lidas no arquivo externo, como não é possível retornar um array pela linguagem C, optou-se por usar passagem de parâmetros por referência para realizar essa tarefa.

3.2.1.2. lexicar(char[])

Esta função possui como parâmetro o array lido anteriormente, é nesta função que está a chamada de cada sub-etapa da análise léxica.

3.2.1.2.1. isVocabulary(char[])

Esta função possui como parâmetro o array contendo os dados do arquivo externo, e ele verifica se todos os caracteres presentes no array fazem parte do vocabulário principal. Ele faz isso comparando cada caractere com todos os permitidos, se por algum motivo um caractere não for reconhecido, retorna mensagem de erro.

3.2.1.2.2. tiraTAB(char[])

Esta função possui como parâmetro o array contendo os dados do arquivo externo, e foi criada apenas por comodidade. Ela simplesmente varre o array, e substitui todos os caracteres '\t' por um espaço em branco. Evitando um processo de ignorar caracteres neutros na linguagem.

3.2.1.2.3. putSpace(char[])

Esta função possui como parâmetro o array contendo os dados do arquivo externo. Esta função serve para separar os tokens por um caractere de espaço, facilitando o processo de análise dos tokens. É aqui que se encontra a lógica pela detecção dos sub-vocabulários, varrendo, diversas vezes, o array atrás de mudanças de sub-vocabulário, e a cada mudança, uma função auxiliar (desloca) inclui esse caractere de espaço para separá-los.

a) desloca(char[], int)

Esta função possui como parâmetro o array contendo os dados do arquivo externo e um inteiro que representa a posição onde se deseja incluir o caractere espaço. A função simplesmente realiza a trasferencia dos caracteres na posição a-1 para a. Até que a seja igual a posição informada, aí, é incluído o caractere espaço, e também o caractere nulo no final do array, pois esse não é transferido no processo de deslocamento.

3.2.1.2.4. getToken(char[])

Esta função possui como parâmetro o array contendo os dados do arquivo externo já com os espaçamentos devidamente colocados, identifica cada tipo de token, e associa esse token a um caractere padrão, que será armazenado num array, biblioteca de tokens do programa, que na análise sintática será base. Esta função, sozinha, identifica os tokens numéricos, e utiliza duas funções de apoio.

A lista dos tokens e seus caracteres padrão podem ser observados na Tabela 1.

a) whoSpecial(char[], int)

Esta função possui como parâmetro o array contendo os dados do arquivo externo e um inteiro que representa apartir de onde ele deve iniciar a análise de tipografia do token. Pois podem haver vários Tokens do mesmo tipo juntos. Esta função auxilia na busca por tokens com caracteres especiais. O detalhe neste tipo de token, é que alguns devem possuir dois caracteres, outros um. Há uma série de condicionais separando os tipos e os classificando conforme a Tabela 1. E retorna, após não encontrar mais tokens do tipo especial, a quantidade de caracteres do tipo especial encontrados, a fim de dar procedência na varredura do array pela classificação de tokens.

b) isKeyword(char[], int)

Esta função possui como parâmetro o array contendo os dados do arquivo externo e um inteiro que representa apartir de onde ele deve iniciar a análise de tipografia do token. É preciso ficar atento que existem diversos tipos de tokens strings que são reservados à linguagem (palavras-chaves), esse é o mais importante, pois existem limitados conjuntos permitidos, e por isso, a função possui uma forma de comparar o token obtido com todas as Palavras-Chaves válidas, e se não encontrar nenhuma, portanto, é um token do sub-tipo Identificador. Este, são variáveis e nomes que encontramos pelo programa.

As palavras chaves aceitas são: *and*, *begin*, *boolean*, *div*, *else*, *end*, *false*, *if*, *or*, *then*, *true*, *read*, *real*, *program*, *var* e *write*.

3.2.2 Funções de Análise Sintática

As funções apresentadas nessa sessão, possuem como base a tabela de simbolos construida anteriormente, e funciona simplesmente comparando cada simbolo com o seguinte, e verificando se são compatíveis.

3.2.2.1 sintaxar(char[])

Esta função possui como parâmetro o array contendo os dados do arquivo externo, sem exata necessidade, foi usado para realizar testes de verificação. Pode ser retirado posteriormente. A base para essa função é o array biblioteca, que foi gerado pela função getToken(). Ele analisa cada

valor nesse array, e compara com os valores que podem existir depois dele, segundo mostra a Tabela 1. Se algum valor não for válido, ele chama uma função de apoio para indicar o erro.

3.2.2.1.1 EhPareado()

Esta função não possui parâmetro, ele apenas verifica se os tokens begin e end, abre parênteses e fecha parênteses estão pareados, isso é, a quantidade deles são pares. Para tanto, a função executa uma soma para cada inicio (begin ou abre parênteses) e uma subtração para cada fim (end e fecha parênteses) em uma variável controle, ao final, se essa variável for igual a zero, então está correto o pareamento.

Também é feito nessa função, a verificação se os tokens if, then e else estão em quantidades corretas. Para isso, uma variável controle é somada por dois a cada if, e subtraida de um a cada then ou else. No final, se a variável controle for menor que zero, então um erro é encontrado.

3.2.2.1.2 erroSintaxa(int)

Esta função possui como parâmetro um inteiro, que representa a posição do token onde foi encontrado um erro. Neste, há um elemento que informa a linha do código que contém esse token, e essa informação serve exatamente para mostrar ao programador a linha onde o erro foi encontrado. Já que caracterizar o tipo de erro é deveras complicado.

4. TABELA DE SIMBOLOS

TABELA 1 – Tabela de Simbolos. Na tabela abaixo apresentamos os tipos de tokens identificados pelo programa e seus simbolos, e também, a lista de caracateres utilizados na Tabela de Simbolos e suas regras de procedência.

Nome:	No código aparece:	Caracter Padrão (token)	Podem aparecer depois:
Abre parenteses	(	a	a i l g k r
Atribuição	:=	b	a i l g k r
Início de Bloco	begin	c	f i j o u
Dois pontos	:	d	q
Condicional Senão	else	e	c
Fim de Bloco	end	f	e f i j o u \0
Falso booleano	false	g	h n
Fecha parenteses	)	h	h i l p m n
Identificador	$(a\|..\|Z\|\_)^{+}(a\|..\|Z\|\_\|0\|..\|9)^{*}$ - Palavras-Chaves	i	b d h m n p s t
Condicional Se	if	j	a i l
Inversor booleano	not	k	a i g k r
Número	$(0\|..\|9)^{+}(.\|\varepsilon)(0\|..\|9)^{*}$	l	h m n
Operador	+ - / div * and or < <= > >=	m	a i l
Ponto e Vírgula	;	n	c e f i j s o u
Comando de Leitura	read	o	a
Condicional Então	then	p	c
Tipo de variável	real boolean	q	n
Verdadeiro booleano	true	r	h n
Declarador de Variável	var	s	i
Virgula	,	t	i
Comando de escrita	write	u	a
Início de programa	program	v	i
Operador Especial	<> =	w	a i l g k r
Fim de programa		\0	

5. AUTOMATOS

Para auxiliar na construção lógica do programa, automatos foram gerados, segundo as expecificações dadas em aula. O correto seria ter utilizado apenas dois automatos, um para cada analisador, contudo, o desenrolar deste acabou fazendo a tarefa com a utilização de três automatos, dois léxicos e um sintático.

5.1. Análise Léxica:

O funcionamento do analisador léxico, é dividido em dois autômatos, um que quebra o código recebido em tokens e outro que reconhece esses tokens (gerando a tabela de simbolos). É possível ver na Figura 1.1 o automato de quebra, que, ao retornar ao simbolo inicial, ele chama uma função (putSpace), que incrementa na fita, o simbolo do espaço, isso auxilia o automato da Figura 1.2 no reconhecimento dos simbolos, já que os tokens são apenas identificados pelos seus primeiros caracteres (exceto o das palavras chaves), e os estados que vão para o q6 informam o tipo de token reconhecido, em q6, ele procura o simbolo de espaço para recomeçar o processo de reconhecimento.

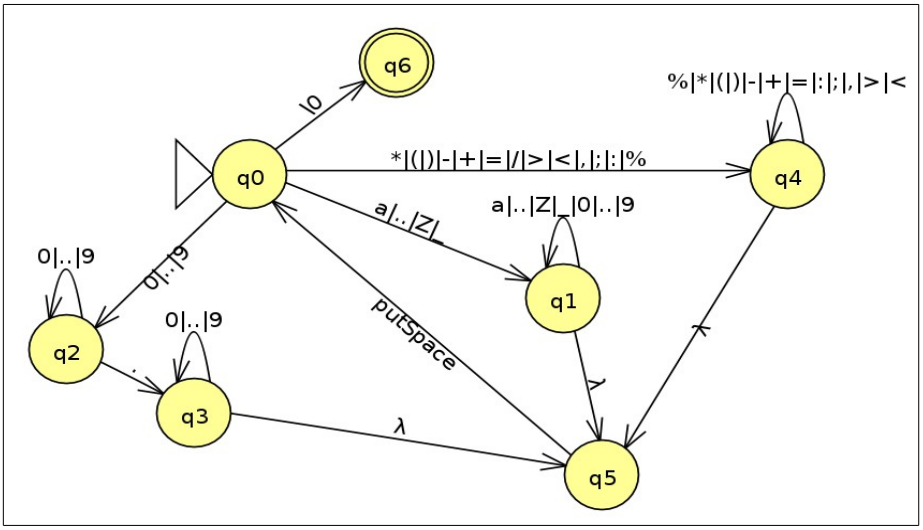
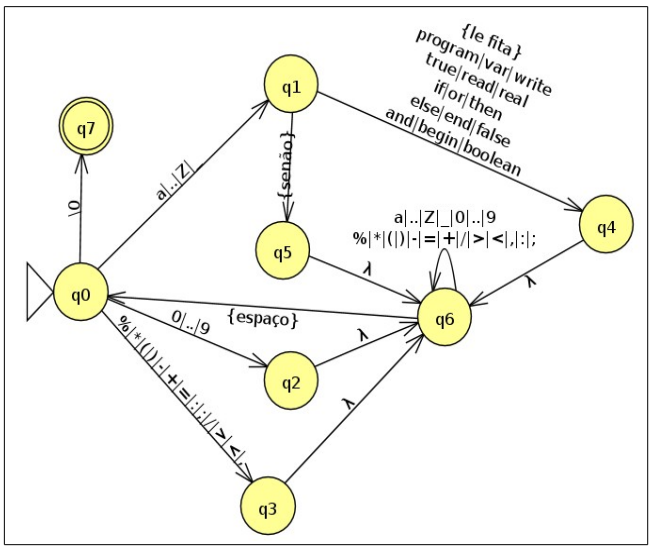


Figura 1.1 - Automato Finito Deterministico da Análise Léxica: Quebra de Tokens
Fonte: do Autor



Fgura 1.2 - Automato Finito Deterministico da Análise Léxica: Geração de Tokens
Fonte: do Autor

5.2. Análise Sintática:

Como já explicado anteriormente, o analisador sintático apenas trabalha com a tabela de simbolos, ele lê a fita gerada, e a cada leitura ele vai para um novo estado, obrigatoriamente, até chegar ao estado q23, que é o final. Caso ocorrer de algum estado não levar a outro, ou não chegar ao q23, o automato aborta e retorna uma mensagem de erro. Infelizmente, o automato é grande, o que gera um pequeno espaguetti de linhas.

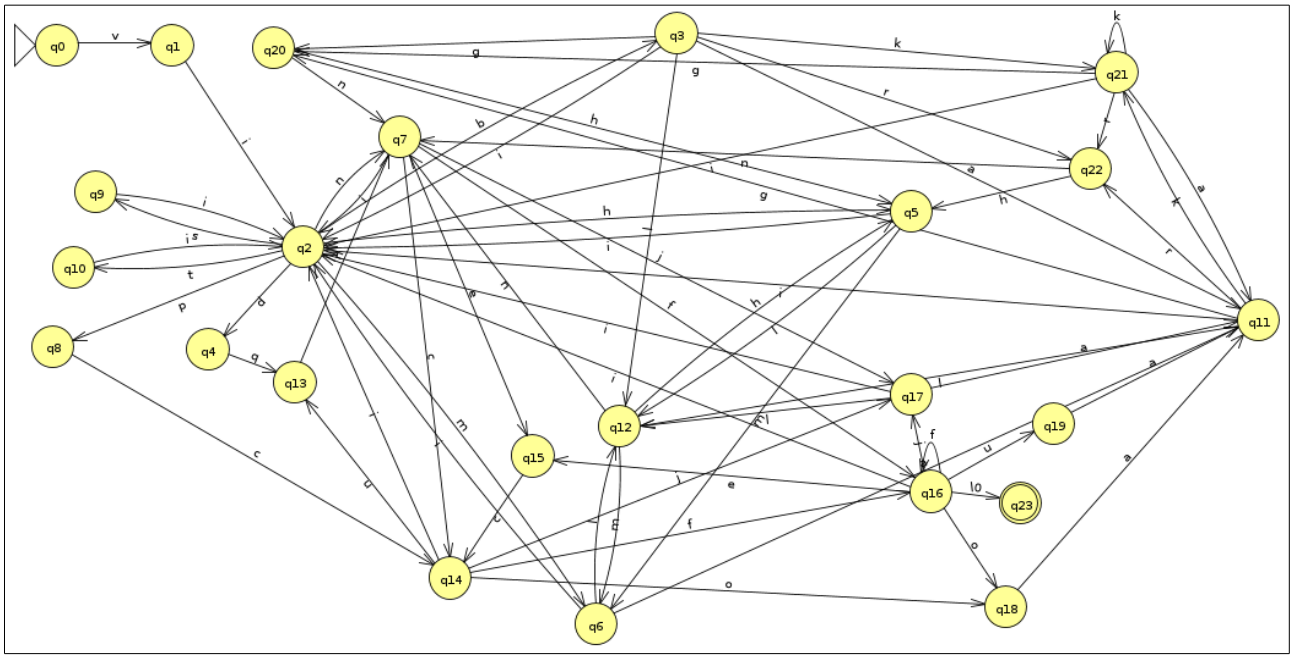


Figura 2 - Automato Finito Deterministico da Análise Sintática
Fonte: do Autor