UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Trabalho 1 - Análise Léxica da Linguagem PL/0

Carlos Henrique Hannas de Carvalho	nº USP: 11965988
Henrique Carobolante Parro	nº USP: 11917987
Pedro Antonio Bruno Grando	nº USP: 12547166
Weverton Samuel Alves	nº USP: 11917326

Prof. Thiago A. S. Pardo

SCC0605 - Teoria da Computação e Compiladores

19 de maio de 2024

Sumário

Lista de Figuras

1 Introdução

Os compiladores transformam um código-fonte em um programa em linguagem de máquina. Há diversas etapas, como a análise léxica, sintática e semântica, do compilador para fazer essa transformação de códigos. Inicialmente, estuda-se o analisador léxico, que desempenha um papel fundamental ao transformar o código-fonte em uma sequência de *tokens* - os *tokens* são blocos de construção, para a análise sintática subsequente.

O objetivo deste projeto é desenvolver um analisador léxico para a linguagem de programação PL/0. O analisador será projetado para processar um código-fonte (em PL/0), identificar e classificar os *tokens* do programa, através de autômatos finitos, e detectar erros léxicos, de acordo com a gramática da linguagem. A gramática da linguagem PL/0 está disponível para visualização em [1].

Como citado anteriormente, para alcançar este objetivo, desenvolve-se e implementa-se, em C, autômatos finitos que reconheçam a gramática da linguagem proposta. O resultado final será um analisador léxico funcional: etapa inicial de um compilador para a linguagem PL/0.

Este relatório inclui as decisões de projeto, tais como a escolha dos autômatos e a lógica de implementação, além de fornecer instruções detalhadas para compilação e execução do código. Por fim, também será apresentado um exemplo de execução, demonstrando o analisador léxico desenvolvido.

2 Decisões de Projeto

Projetou-se 5 sub-autômatos, no ambiente JFLAP, para a linguagem proposta: autômato de comentários, autômato de identificadores, autômato de números inteiros, autômato de operadores relacionais e autômato de símbolos unitários. Além disso, desenvolveu-se, em linguagem C, um código que implementasse esses autômatos, simulando o recebimento de pedido do analisador sintático e retornando uma cadeia acompanhada de seu tipo de *token*. Os autômatos e o código serão discutidos brevemente.

3 Modelagem de Autômatos

A figura 1 mostra o autômato completo para reconhecimento léxico da linguagem PL/0.

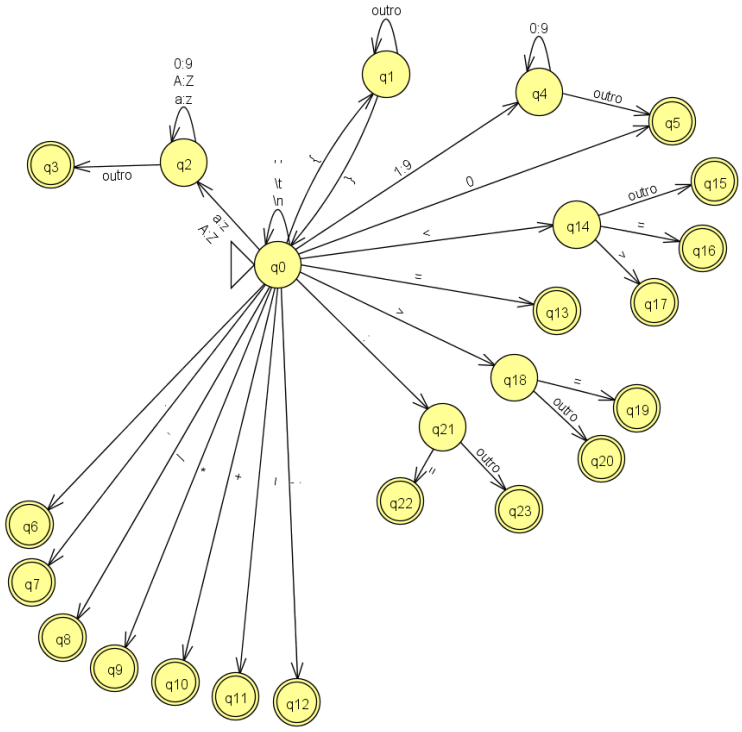

Figura 1: Autômato completo da linguagem PL/0.

O autômato principal é dividido em 5 partes: comentários, identificadores, números inteiros, operadores relacionais e símbolos unitários. Cada um dos sub-autômatos são descritos nos tópicos a seguir.

3.1 Autômato de Comentários

Os comentários na linguagem PL/0 ocorrem entre os símbolos de chaves: '{}'. Portanto, o início de um comentário é marcado pelo símbolo '{', seguido de qualquer caractere (identificado como 'outro' na figura 2), quantas vezes forem necessárias. Utiliza-se '}' para encerrar um comentário.

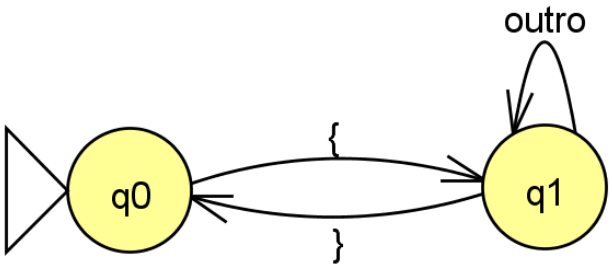

Figura 2: Autômato de comentário da linguagem PL/0.

Não considera-se um estado final no autômato de comentário, porque os estados finais são reservados para identificação de *token*. Nesse caso, há apenas consumo de caracteres.

3.2 Autômato de Identificadores

Os identificadores são formados pela seguinte regras: eles são iniciados por letras (maiúsculas e minúsculas), seguido de qualquer combinação de letras do alfabeto ou dígitos numéricos. Além disso, a linguagem PL/0 é *case sensitive*, ou seja, as letras `a` e `A` são considerados identificadores distintos. A figura 3 mostra o autômato de identificadores:

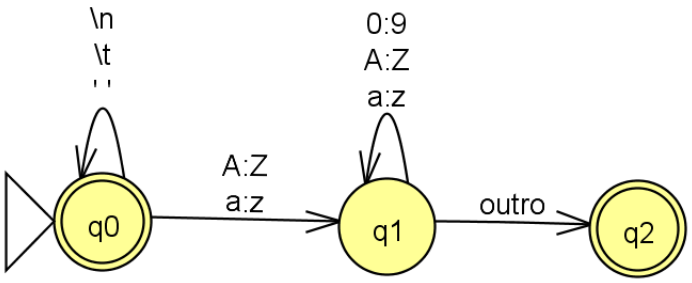

Figura 3: Autômato de identificadores da linguagem PL/0.

Alguns exemplos de identificadores válidos são: `x`, `count`, `Variable1`. Um exemplo de identificador inválido é: `1variable`.

3.3 Autômato de Números Inteiros

Na linguagem PL/0, os números são inteiros e formados por um ou mais dígitos, sendo que não há zeros à esquerda. Na figura 4, a transição de $q0$ para $q1$ lê um número no intervalo de 1 até 9. No estado $q1$ faz-se a leitura caso o número seja maior ou igual à 10. Caso o número seja zero, acontece a transição do estado inicial $q0$ para final $q2$. Os números negativos são considerados na análise sintática do compilador.

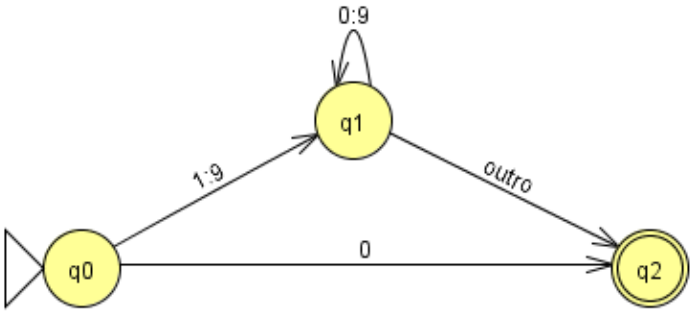

Figura 4: Autômato de número inteiro da linguagem PL/0.

As cadeias aceitas pelo autômato são os números inteiros.

3.4 Autômato de Operadores Relacionais

A figura 5 representa o autômato de operadores relacionais. Eles são utilizados para comparar valores e expressões, na linguagem PL/0, e são fundamentais para construção de condições, estruturas de controle e fluxo de programas.

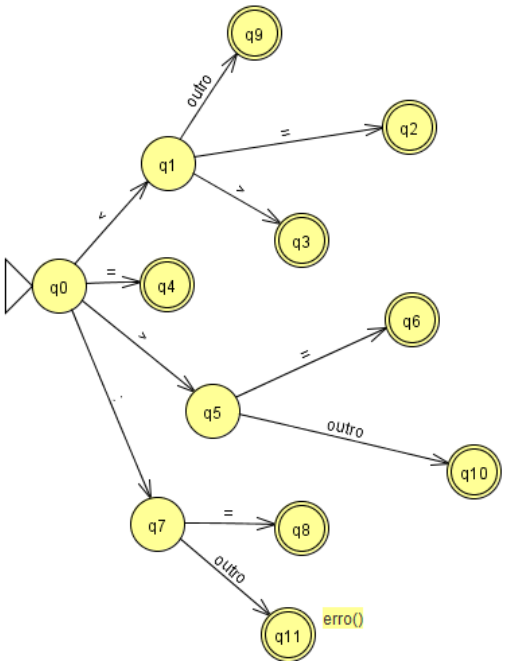

Figura 5: Autômato de operadores relacionais da linguagem PL/0.

Os operadores relacionais aceitos pela linguagem PL/0 são: `<, >, <=, >=, =, <>, :=`. Eles representam condições de “menor”, “maior”,“menor ou igual”, “maior ou igual”, “igual”, “diferente”e “atribuição”, respectivamente.

3.5 Autômato de Símbolos Unitários

A figura 6 mostra, por meio de suas transições, todos os símbolos unitários aceitos pela linguagem PL/0. são eles: `, ; + - * /` . Esses símbolos são utilizados em operações, expressões, separações e instruções no código.

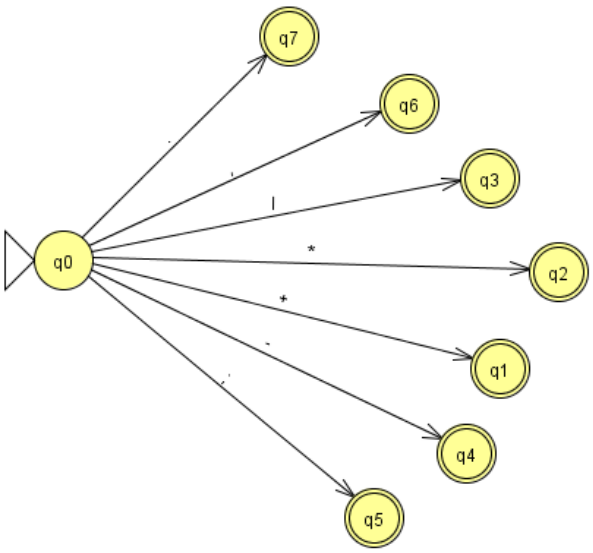

Figura 6: Autômato de símbolos unitários da linguagem PL/0.

A vírgula (`,`) e o ponto e vírgula (`;`) são utilizados como separadores; o sinal de mais (`+`), o sinal de menos (`-`), o

asterisco (*) e a barra (/) são usados para operações aritméticas básicas de adição, subtração, multiplicação e divisão, respectivamente.

4 Código

O código está estruturado em 4 arquivos, e é uma representação do autômato em linguagem C. Sobre a estrutura desses arquivos:

- **main.c**: simula as chamadas do analisador sintático, além de realizar abertura de arquivos, declaração de estruturas de dados gerais do processo e I/O;
- **hash.c**: contém a implementação do hash utilizado para estruturar a tabela de palavras reservadas, bem como outras funções auxiliares;
- **tokenization.c**: representa a análise léxica, onde serão feitas as leituras de cada caractere, as transições de estados e classificação dos tipos de tokens;
- **automata.c**: contém funções relativas à estrutura do autômato, como a criação da tabela de transições e uma função que realiza as transições do autômato.

Sobre o projeto como um todo, escolheu-se implementar o autômato através da representação explicita de sua tabela de transições, uma vez que considerou-se aceitável um gasto maior de memória (abundante em computadores modernos) para se ter maior eficiência nas transições. Essa tabela, por sua vez, contém o próximo estado de cada possível entrada de cada estado do autômato, bem como uma *flag* que indica se essa transição é imprimível (e.g., transições marcadas como 'outro' são exemplos de transições que não devem imprimir).

Além disso, manteve-se as ações dos autômatos contidas em um *switch case*, pois, apesar de que a atribuição de ponteiros de funções para os estados fosse possível, considerou-se que a complexidade não seria justificável, dado que um *switch* apresenta um desempenho satisfatório. Outro ponto interessante é o tratamento de caracteres não imprimíveis, uma decisão feita para evitar que o *buffer* onde se armazena a cadeia receba caracteres indevidos. Por fim, implementou-se alguns tratamentos de erros básicos, como a verificação de fim de arquivo inesperado, caractere invalido e limite de tamanho do token, que foi colocado com 62 caracteres.

5 Compilação de Código-Fonte

O código foi escrito e testado em uma máquina com Fedora 37 e o compilador gcc 13.2 - porém também foi testada em máquinas no Windows, na IDE VSCode.

Para compilar o código, basta utilizar o comando"make", uma vez que existe um Makefile no diretório (em caso de dúvida, é possível também consultar o Makefile para olhar as diretivas de compilação). O resultado será um executável ***tokenizer***.

Esse deverá ser executado em linha de comando com a seguinte sintaxe para sistemas Linux:

./tokenizer ArquivoDeEntrada

Caso o arquivo seja executado incorretamente, o programa avisará o usuário a sintaxe de sua execução.

6 Exemplo de Execução

Utilizo-se o código PL/0 da figura 7 para demonstrar o funcionamento do compilador:

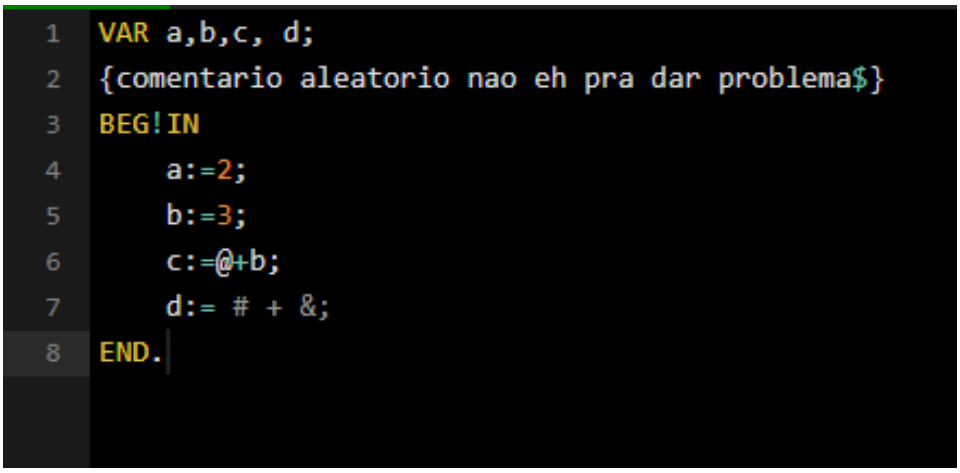

Figura 7: Código em PL/0.

Ao utilizar o código de entrada acima, o compilador produziu o seguinte resultado, encontrado na figura 8:

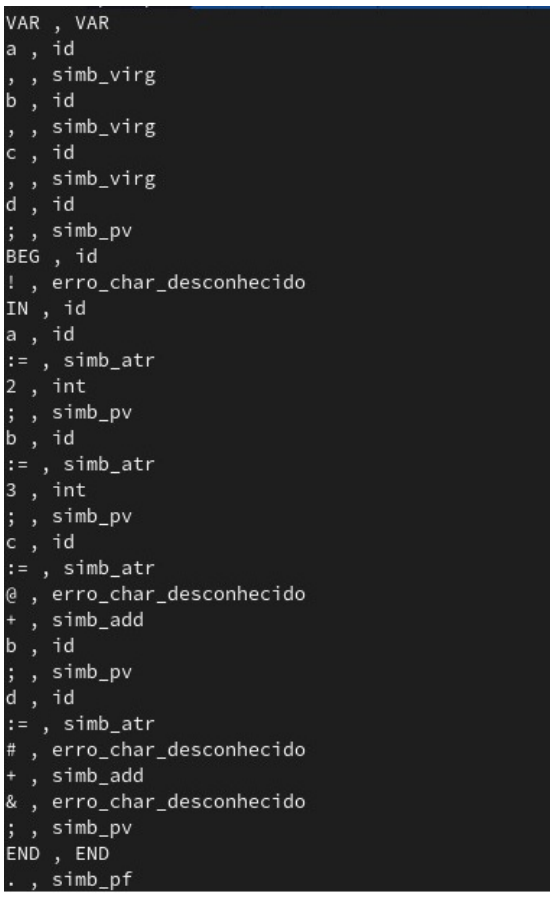

Figura 8: Análise léxica do código em PL/0.

Percebe-se que o compilador encontra os principais erros no código, como por exemplo, o `!` no lugar do `I` em `BEGIN`; o `@` na soma matemática da linha 6 do código; o `#` e `&` na soma matemática da linha 7. Além disso, o compilador ignorou corretamente o comentário na linha 2. Nota-se também que o compilador identificou os símbolos corretamente, além dos identificadores.

7 Conclusão

O compilador é essencial para o desenvolvimento de um *software*, porque ele traduz um código-fonte de alto nível, como C, para um programa em linguagem de máquina.

Estudou-se, nesse projeto, a primeira etapa de um compilador: análise léxica. Como pôde-se observar, ela faz a *tokenização*, através de autômatos, de uma determinada linguagem - nesse caso, PL/0. Fica claro que uma boa separação e identificação de *tokens* auxilia um determinado usuário a compilar e corrigir eventuais erros de compilação de códigos.

Referências

[1] Linguagem PL/0. Disponível em: https://edisciplinas.usp.br/pluginfile.php/8350382/mod_resource/content/0/Gram%C3%A1tica%20de%20PL0.pdf