UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação
SCC0605 Teoria da Computação e Compiladores

Analisador Léxico para PL/0

Professor:
Thiago Alexandre Salgueiro Pardo

Autores:
Alexandre Lopes Ferreira - Nusp: 11801199
Diogo Barboza de Souza - Nusp: 12745657
Luigi - Nusp: 11800563
Pedro - Nusp: 11212289
Yudi Asano Ramos - Nusp: 12873553

17 de Maio, 2024

1 Introdução

Este relatório apresenta o processo de desenvolvimento de um analisador léxico para a linguagem PL/0. O analisador léxico é capaz de processar um arquivo de texto contendo um programa escrito em uma linguagem de programação, nes caso, PL/0, produzindo como saída outro arquivo de texto contendo pares de token-classe por linha, incluindo a identificação de erros léxicos, caso necessário.

O algoritmo foi estruturado de forma a possuir uma função principal dedicada exclusivamente a invocar o procedimento do analisador léxico repetidamente, a fim de processar todo o programa do usuário. Cada chamada resulta na obtenção de um par token-classe, o qual é impresso pela função principal, simulando, assim, a interação do analisador sintático com o léxico.

No decorrer do desenvolvimento, optou-se por implementar uma tabela hash para garantir eficiência e rapidez nas operações de busca de palavras e símbolos reservados e utilizou-se a ferramenta JFlap para a criação dos autômatos necessários.

2 Decisões de Projeto

Afim de facilitar o entendimento e possibilitar o reuso, este trabalho foi modularizado de forma que há sete arquivos de funções C que são chamados e organizados pela *main*. Estes códigos serão explicados nos próximos tópicos.

2.1 Main

Esta função recebe o arquivo de *input* que vai passar pelo analisador e chama as demais funções para perceber erros e retornar se o código está dentro dos padrões estabelecidos pela linguagem *PL/0*.

2.2 check_symbol

Este módulo tem como objetivo definir funções para identificar que tipo de símbolo foi encontrado. Em outras palavras, se uma letra for encontrada e a função *isLetter* for chamada, o código responderá *TRUE* garantindo que se trata, realmente, de uma letra. Estas funções serão muito úteis para as demais etapas, já que determinados símbolos não podem aparecer em qualquer lugar segundo a gramática. Abaixo, seguem as funções presentes neste módulo.

2.2.1 isLetter

Identifica se o símbolo é uma letra, seja ela maiúscula ou minúscula.

2.2.2 isDigit

Identifica se o símbolo é um dígito (0-9).

2.2.3 isSpace

Verifica se o caractere fornecido é um espaço em branco. Facilitando o programa a ignorá-lo quando necessário.

2.2.4 isNewLine

Verifica se o caractere fornecido é uma quebra de linha.

2.2.5 BeginComment

Verifica se o caractere fornecido é uma chave de abertura de comentário ('{').

2.2.6 CloseComment

Verifica se o caractere fornecido é uma chave de fechamento de comentário ('}').

2.2.7 isSimpleKeySymbol

Verifica se o caractere fornecido é um símbolo reservado simples, como `+`, `-`, `/`, `*`, `,`, `;`, `.`, `(`, ou `)`.

2.2.8 isDoubleDotsKeySymbol

Verifica se o caractere fornecido é dois pontos (':').

2.2.9 isEqualKeySymbol

Verifica se o caractere fornecido é um sinal de igual ('=').

2.2.10 isBiggerKeySymbol

Verifica se o caractere fornecido é um sinal de maior ('>').

2.2.11 isLowerKeySymbol

Verifica se o caractere fornecido é um sinal de menor ('<').

2.2.12 isUnderScore

Verifica se o caractere fornecido é um *underscore* ('_').

2.2.13 isSeparator

Verifica se o caractere fornecido é um separador. Um separador pode ser um espaço, uma nova linha, um símbolo reservado simples, dois pontos, um sinal de igual, um sinal de maior, um sinal de menor, uma chave de comentário de abertura, uma chave de comentário de fechamento ou o fim do arquivo (*EOF*). Portanto, esta função utiliza as respectivas funções definidas anteriormente neste mesmo módulo (*isSpace*, *isNewLine*, *isSimpleKeySymbol*, etc.).

2.3 files_actions

Nesta esfera do programa fizemos as duas principais funções que utilizaremos para mexer com os arquivos de entrada e de saída.

2.3.1 backtrack

Esta função utiliza `fseek` para mover o ponteiro do arquivo `file` uma posição para trás, a partir da posição atual (`SEEK_CUR`).

2.3.2 write_token

Esta função usa `fprintf` para escrever o `token` e a `token_class` no arquivo `file`, separados por uma vírgula e seguidos por uma nova linha. Esta operação é útil para registrar de maneira eficiente no arquivo de saída.

2.4 open_close

Complementarmente ao item anterior, as funções definidas neste arquivo servem para abrir e fechar de maneira correta e desejada os arquivos de leitura e de escrita que são utilizados.

2.4.1 open_file

Esta função abre os arquivos do modo desejado, escrita para a saída e leitura par a entrada. Além disso, para ambos os casos, se não for possível abrir o arquivo uma mensagem de erro é retornada.

2.4.2 close_file

Fecha os arquivos utilizados quando chamada, garantindo, assim, boas práticas.

2.5 table

Este módulo mostra a decisão de utilizar uma tabela *hash* como estrutura de dados para armazenar palavras-chave e símbolos reservados. Além disso, define funções para manipulação de estados finais e de erro em um analisador léxico.

2.5.1 hash_value

Esta função calcula o valor do índice *hash* de um *token*. Para tanto, escolhemos como operação a multiplicação o valor atual por 31 adicionado ao valor ASCII do caractere atual. O resultado final para a posição é o valor do *hash* dividido por seu tamanho `HASH_SIZE` e depois deslocado com o resto da divisão.

2.5.2 insert_token

Esta função chama a anterior para calcular o índice do *token*, aloca memória para um novo nó e o insere na tabela *hash* na posição obtida.

2.5.3 search_token

Esta função busca um *token* na tabela *hash*, retornando 1 se for encontrado e 0 caso contrário.

2.5.4 get_token_class

Esta função obtém a classe de um *token* na tabela *hash*, retornando-a caso encontrada.

2.5.5 make_KeyWords

Esta função cria a tabela *hash* e insere nela palavras-chave da linguagem.

2.5.6 make_KeySymbols

Analogamente ao item anterior, cria uma tabela *hash* com símbolos reservados.

2.6 states

Este pedaço define as operações de saída da máquina de estados, ou por chegar à um estado final ou encontrando um erro

2.6.1 final_states

Esta função realiza operações adequadas para estados finais. Dependendo do estado final alcançado, ela pode retornar um caractere, conferir se o *token* é: ou uma palavra reservada; ou um identificador; ou um número e escreve-lo junto de sua classe no arquivo de saída.

2.6.2 error_states

Dependendo do estado de erro alcançado, estas linhas escrevem uma mensagem de erro específica no arquivo de saída.

2.7 lexical_analyzer

Neste fragmento de código tratamos as etapas do analisador léxico diretamente como uma máquina de estados. Desta forma, é nessa parte que dividimos os estados e configuramos as passagens entre eles.

2.7.1 transition_rules

Esta função define as regras de transição de estados. Em outras palavras, para cada estado (current_state), a função verifica o símbolo atual e decide qual será o próximo estado.

- START: Estado inicial onde a maioria das verificações ocorre, incluindo letras, dígitos, símbolos e espaços.
- KEYWORD: Continua lendo letras para identificar palavras-chave ou identifica que se tornou um identificador, caso encontre um dígito ou sublinhado.
- DOUBLE_DOTS_KEYSYMBOL: Verifica se o símbolo após os dois pontos é um igual para formar `:=`.
- LOWER_KEYSYMBOL: Verifica se o símbolo após < é =, > ou outro.
- BIGGER_KEYSYMBOL: Verifica se o símbolo após > é = ou outro.
- IDENTIFIER: Continua lendo letras, dígitos ou sublinhados para formar identificadores.
- NUMBER: Continua lendo dígitos para formar números, mas identifica erros se houver letras.
- COMMENT: Continua lendo até encontrar o fechamento do comentário.

2.7.2 lexical_analyzer

Esta função que tem o mesmo nome de seu módulo tem por finalidade ler o arquivo inteiro (até o *EOF*) aplicando as regras definidas na sub-seção anterior. Portanto, a máquina é iniciada no ponto *START* e vai caminhando até terminar em um erro ou em um estado final, também definidos anteriormente neste relatório.

2.7.3 execute_lexical_analyzer

Complementando o item anterior, esta função é responsável por implementar o ciclo de repetições para que o passo a passo seja repetido até o fim do arquivo. Além disso, a tabela *hash* de palavras e símbolos reservados também é iniciada aqui.

3 Automato Implementado

De acordo com o que foi estudado em aula, sabemos que a lógica do código feito para este analisador pode ser representada por uma maquina de estados finitos. Assim, neste tópico vamos discorrer a respeito desta representação.

Para tanto, inicialmente deve ser explicado que a seta representa o estado *START* do programa e é equivalente ao estado q_0 da máquina. Ademais, os estados representados com linhas duplas(estados q_3, q_8, ... por exemplo) são os estados finais do sistema. Sabendo disso, vamos "analisar este analisador" partindo do estado inicial e estudando suas transições de acordo com Figura 1.

3.1 q_0

3.2

Começando do estado inicial, e sabendo que a leitura de um caractere do arquivo no código equivale a uma entrada. Partindo do q_0. Caso a entrada seja um espaço ou uma quebra de linha, como o analisador léxico deveria ignora-los, o procedimento permanece no mesmo estado. Para um casos diferentes outros estados são chamados.

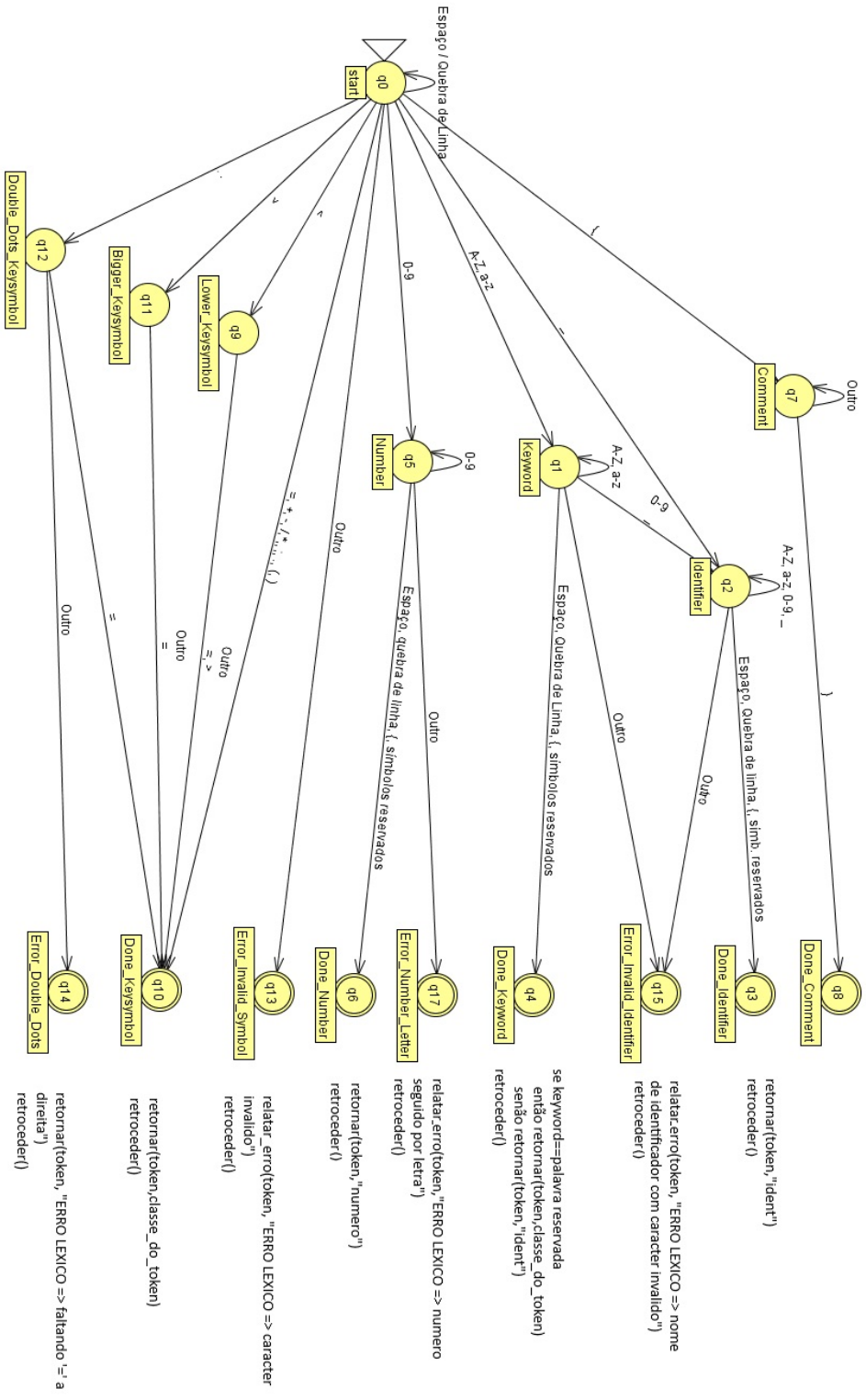

Figure 1: Representação gráfica do automato do analisador léxico que foi implementado no código

3.3 q_1

Caso o simbolo analisado no q_0 seja uma letra, maiúscula ou minúscula, há uma checagem para identificar se é uma palavra reservada. Assim, enquanto não aparecer um caractere diferente de uma letra não saímos do q_1. Já que não existem palavras-chave com *underscore*, a aparição de um garante que, se a gramática estiver sendo respeitada, se trata de um identificador que falaremos mais sobre nos próximos estados, logo, o estado muda para q_2. Outra transição possível é a para q_4 ocorre após a aparição de um dos seguintes: Espaço; Quebra de linha; `{`; Símbolo reservado (`>` por exemplo). De acordo com a gramática, qualquer outra leitura não mencionada, não deveria ocorrer neste estado e resulta na ida para o estado de erro q_15.

3.4 q_2

Caso o simbolo analisado no q_0 seja um *underscore* o único resultado previsto é que se trata de um identificador. Sendo assim, enquanto a palavra não tenha sido lida por completo, outras letras ou mais *underscore*s, o estado q_2 é mantido. Analogamente ao item anterior, o estado final q_3 é chamado para os *inputs previstos* (espaço, quebra de linha, `{`, símbolo reservado). Demais entradas resultam no estado de erro q_15.

3.5 q_3

Como dito anteriormente, este estado só é chamado caso um identificador seja descoberto. Deste modo, este é um estado final da gramática e o código retorna o *token*.

3.6 q_4

Muito semelhante a q_3 este estado, também final, vem do q_1 então podem ser uma palavra-chave. Com isso, o código procura esta palavra na tabela de palavras reservadas retornando-a caso encontrada. Caso contrario se trata de um identificador e procede análogo a q_3

3.7 q_{15}

Este estado só é chamado se surgir um simbolo impróprio à gramática num possível identificador ou palavra-chave. Assim, este é um estado final que representa um erro de identificador invalido.

3.8 q_5

De volta a q_0, se, ao invés de uma letra ou *underscore*, a leitura resulte em um número, o estado q_5 é chamado. Não sairemos deste até que o número seja lido por completo ($input$ = (0,1,..,9)). Aqui, caso o número acabe da forma prevista (espaço, quebra de linha, `{`, símbolo reservado) vamos para o estado q_6. Caso contrario se trata de um erro e vamos para o estado q_17

3.9 q_6

Este estado final aparece quando a leitura de um número é feita dentro dos padrões estabelecidos, portanto o código retorna o *token* do mesmo.

3.10 q_{17}

Este estado final de erro aparece caso um número inválido seja percebido. Retorno de erro.

3.11 q_7

O estado q_7 diz respeito aos comentários na linguagem PL/0. Estes devem aparecer entre chaves. Sendo assim, este estado é encontrado na leitura de um `{` em q_0 e permanecemos em q_7 enquanto `}` não apareça. Em outras palavras, só vamos para o estado q_8 se o fechamento de chaves seja encontrado.

3.12 q_8

Como visto anteriormente, encontrar este estado significa a leitura bem sucedida de um comentário. Estado final.

3.13 q_9

Caso seja lido um sinal de menor em q_0 como entrada, faz-se necessária uma verificação para descobrir se estamos lidando com um `<` ou com um simbolo de 2 caracteres como `<=` ou `<>`. Assim, apesar de qualquer leitura resultar no estado q_10, é preciso identificar o segundo caractere para entender a operação.

3.14 q_{11}

Analogamente ao item anterior, se em q_0 for obtido um `>`, o estado q_11 garante a leitura do segundo caractere para saber se é um `>=` antes de ir para q_10.

3.15 q_{12}

Seguindo a mesma linha dos dois itens anteriores, se a entrada em q_0 for :, verificamos se a próxima leitura resulta em um `=`. Se sim, vamos para q_10. Porém, diferentemente de seus estados similares, o símbolo de pontos duplos só pode aparecer antes de um sinal de igual. Deste modo, qualquer outra entrada leva ao estado de erro q_14.

3.16 q_{10}

Este estado é alcançado pelas condições dos últimos 3 estados explicados ou pela leitura de `=, +, |, *, (, ), .` em q_0. Uma vez neste estado final a operação desejada já foi encontrada com sucesso e seu *token* é retornado.

3.17 q_{14}

Este estado é alcançado se : for lido sem um sinal de `=` o seguindo. Como isso não é aceito na gramática retornamos um erro especifico.

4 passo a passo para compilar

Para facilitar a compilação e execução, o algoritmo possui um arquivo makefile, contendo as seguintes instruções:

```
all:
	gcc -Wall -o lexical_analyzer src/*.c

run:
	./lexical_analyzer

clean:
	rm lexical_analyzer
```

Assim, para compilar e rodar o código-fonte do analisador léxico desenvolvido, siga as instruções abaixo: Certifique-se de que você está utilizando um sistema operacional compatível com o GCC (GNU Compiler Collection) e makefile (caso não seja possível utilizar o makefile, deve-se utilizar os comandos presentes no arquivo makefile diretamente).

Abra o terminal e navegue até o diretório onde os arquivos do código-fonte estão localizados. Execute o seguinte comando para compilar o código-fonte:

```
Make all
```

Esse comando compila todos os arquivos ‘.c‘ presentes no diretório e criará um executável chamado ‘lexical_analyzer‘.

Após a conclusão da compilação sem erros, você pode executar o analisador léxico com o seguinte comando:

```
Make run
```

Isso iniciará a execução do analisador léxico, esperando o usuário indicar o nome do arquivo de entrada, após a especificação, este é processado e produz-se o arquivo de saída.

(opcional): Se desejar limpar os arquivos gerados durante a compilação, você pode executar o seguinte comando:

```
Make clean
```

Isso removerá o executável do analisador léxico do diretório.
Certifique-se de fornecer um programa escrito em PL/0 para que o analisador léxico possa processá-lo corretamente e produzir a saída desejada.

5 Exemplos de execução

5.1 Exemplo 1:

Entrada sem erros:

```
VAR a,b,c;

BEGIN
      a:=2;
      b:=3;
      c:=2 + b
END.
```

Saída:

```
VAR,VAR
a,ident
,,simbolo_virgula
b,ident
,,simbolo_virgula
c,ident
;,simbolo_ponto_virgula
BEGIN,BEGIN
a,ident
:=,simbolo_atribuicao
2,numero
;,simbolo_ponto_virgula
b,ident
:=,simbolo_atribuicao
3,numero
;,simbolo_ponto_virgula
```

```
c,ident
:=,simbolo_atribuicao
=,simbolo_comparacao
2,numero
+,simbolo_mais
b,ident
END,END
.,simbolo_ponto
```

A entrada do analisador léxico é um programa escrito na linguagem PL/0. Sendo um programa simples que declara variáveis e realiza algumas operações básicas de atribuição e aritmética. Agora, na saída cada linha consiste em um par "token,classe", onde "token" é uma string representando o "token" extraído do programa de entrada, e classe é a categoria à qual esse token pertence.

Palavras Reservadas e Identificadores: Tokens como VAR, BEGIN, e END são identificados como palavras reservadas e mapeados para suas respectivas classes (VAR, BEGIN, END). Os identificadores a, b, e c são classificados como ident.

Símbolos de Pontuação e Operadores: Os símbolos de pontuação como "," e ";" são categorizados como "simbolo_virgula" e "simbolo_ponto_virgula", respectivamente. O operador de atribuição ":=" é classificado como "simbolo_atribuicao", e o operador aritmético "+" como simbolo_mais.

Números: Os números 2 e 3 são identificados e classificados como numero.

Erros Léxicos: Não há indicação de erros léxicos na saída fornecida, sugerindo que todos os tokens foram reconhecidos corretamente.

5.2 Exemplo 2:

Entrada com erros:

```
VAR a_,b,____c;

BEGIN
    a: @ + 1;
    b}:=3aa;
    c:=@ + b
END.
```

Saída:

```
VAR,VAR
a_,ident
,,simbolo_virgula
b,ident
,,simbolo_virgula
____c,ident
;,simbolo_ponto_virgula
BEGIN,BEGIN
a,ident
:,ERRO LEXICO => faltando '=' a direita
@,ERRO LEXICO => caracter invalido
+,simbolo_mais
1,numero
```

```
;,simbolo_ponto_virgula
b},ERRO LEXICO => nome de identificador com caracter invalido
:=,simbolo_atribuicao
3a,ERRO LEXICO => numero seguido por letra
a,ident
;,simbolo_ponto_virgula
c,ident
:=,simbolo_atribuicao
@,ERRO LEXICO => caracter invalido
+,simbolo_mais
b,ident
END,END
.,simbolo_ponto
```

Desta vez o código de entrada inclui variáveis com diferentes formatos, operadores, símbolos de pontuação e caracteres que não são válidos na linguagem PL/0. O código foi propositalmente modificado para testar a capacidade do analisador léxico de identificar e relatar erros.

Erros Léxicos: A saída identifica e relata vários erros léxicos no código de entrada:

- O token ":" é marcado como um erro léxico com a mensagem faltando '=' a direita, indicando que o analisador esperava o operador de atribuição ":=".
- O caractere "@" é marcado como carácter invalido, já que não faz parte da gramática de PL/0.
- O token "b}" é considerado um erro devido ao caractere } ser inválido em um identificador.
- O token "3a" é considerado um erro indicando que um número não deve ser seguido imediatamente por uma letra.