Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação

Trabalho 1: Análise Léxica

SCC0605 - Teoria da Computação e Compiladores

Gabriel Balbão Bazon - 13676408
Giovanna de Freitas Velasco - 13676346
Guilherme Lorete Schmidt - 13676857
Jean Carlos Pereira Cassiano - 13864008
Lucas Fernandes Martins - 11800389

Prof. Thiago A. S. Pardo

São Carlos
29 de abril de 2025

1 Introdução

O trabalho que segue visa o desenvolvimento de um analisador léxico para a linguagem PL/0. Para isso, foi elaborado um autômato de estados finitos e seu respectivo código em C. No que concerne ao relatório, este pode ser dividido em três partes: a primeira apresenta e explica de maneira sucinta cada parte do autômato e a segunda expõe a implementação do analisador léxico, juntamente com as decisões de projeto que foram empregadas para a construção do código. Por fim, mostramos como executar o código e alguns exemplos de saídas.

É importante destacar que a linguagem PL/0 é projetada para ser uma versão simplificada de Pascal e, por isso, é bastante limitada. Destaca-se como exemplo que todas as variáveis e constantes são declaradas explicitamente, que os inteiros são os únicos tipos de dado e que só existem operadores aritméticos e relacionais. Essas características moldam e justificam certas decisões do trabalho.

2 Autômato de Estados Finitos: Máquina de Mealy Estendida

A análise léxica pode ser representada com um autômato de estados finitos. Para fazer a representação, optamos por fazer uma Máquina de Mealy estendida. Essa escolha permite a impressão dos dígitos e letras conforme são lidos pelo analisador léxico, o que é mais vantajoso do que ter um estado para cada letra e dígito possíveis. Além disso, também representamos as ações que o código poderia realizar naquele cenário (por exemplo, retroceder o ponteiro) nas transições entre os estados.

A representação do autômato foi feita no JFLAP e está presente em anexo ao relatório, com o nome `"automata_compiler.jff"`. Para fins didáticos, optamos por explicar cada parte do autômato separadamente no relatório. No entanto, é importante manter em mente que cada trecho explicado faz parte de um único autômato que representa o comportamento do analisador léxico. Assim, evidenciaremos a seguir cada trecho do autômato com base nas cadeias que serão retornadas.

2.1 Autômato de Números e Identificadores

O trecho do autômato que lê e classifica os identificadores e números está presente na Figura 1.

O autômato de números consiste em receber dígitos de 0 a 9 e imprimi-los continuamente no estado q_1, até que o autômato receba um símbolo que não seja um dígito. Então, a máquina imprime `", numero"` e, como foi lido um símbolo a mais, retrocede o ponteiro em uma posição.

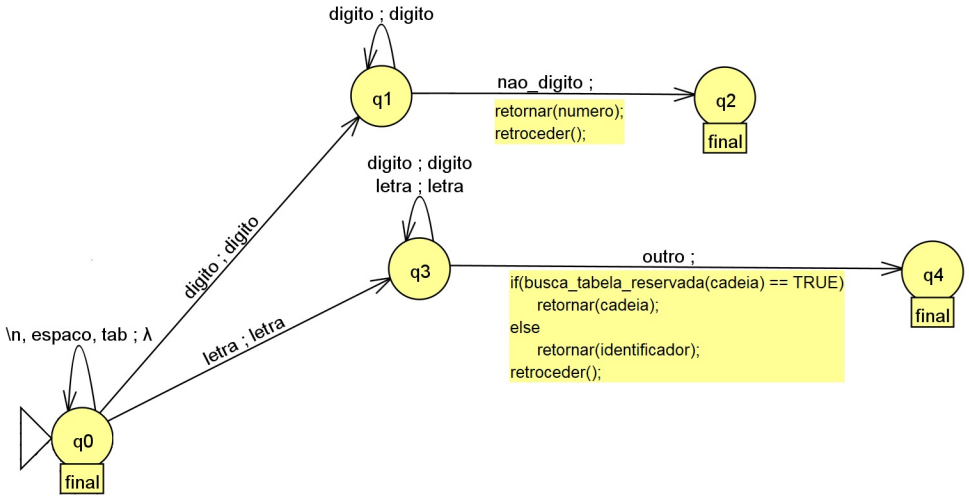

Figura 1 – Autômato de Números e Identificadores.

O autômato de identificadores funciona de forma análoga. Porém, é importante manter em mente o formato dos identificadores na linguagem PL/0: eles devem iniciar com uma letra e em seguida podem ser formados por letras e dígitos. Isso é verificado no autômato na transição entre os estados q_0 e q_3. Note que a máquina de Mealy está imprimindo cada letra e dígito que são recebidos entre as transições. Se o autômato receber um símbolo que não seja um dígito ou uma letra, ele vai para o estado final q_4, o que indica que o identificador está completo. O analisador léxico deve, então, verificar se o identificador faz parte da tabela de símbolos reservados. Assim, se o identificador corresponder a uma palavra reservada, a máquina de Mealy retornará uma saída no formato `", palavra_reservada"`, já que a palavra inteira já foi escrita nas transições anteriores. Do contrário, ela retornará uma saída na forma `", identificador"`. Em ambos os casos, é preciso retroceder o ponteiro para ler novamente o símbolo na próxima iteração do analisador.

Durante a elaboração do autômato, foi preciso escolher uma estratégia para lidar com possíveis erros na geração de números e identificadores. A abordagem escolhida para lidar com problemas de formação de identificadores e números foi semelhante: identificar todos os caracteres até que apareça um símbolo diferente e classificá-los de acordo, deixando para o analisador sintático lidar com o erro. Para ilustrar esse escolha, suponha que recebemos a entrada `2minhavariavel`. O analisador poderia retornar saídas distintas a depender da abordagem utilizada. Identificamos duas abordagens distintas que poderiam ocorrer nesse exemplo:

Abordagem 1:

```
2, número
minhavariavel, identificador
```

Abordagem 2:

```
2minhavariavel, <ERRO_NUMERO_MAL_FORMADO>
```

Optamos por utilizar a abordagem do caso 1, já que o analisador sintático irá identificar posteriormente que há um erro nessa construção e, como cada tipo já está classificado, a informação do nome da variável já está disponível. Assim, caso o usuário tenha digitado o 2 por engano, a informação do nome do identificador não é perdida.

Devemos também verificar o caso em que recebemos um identificador mal formado. Tome por exemplo a entrada `minh@variavel`. Similarmente ao caso anterior, percebemos duas abordagens distintas:

Abordagem 1:

```
minh, identificador
@, <ERRO_CARAC_INVALIDO>
variavel, identificador
```

Abordagem 2:

```
minh@variavel, <ERRO_IDENT_MAL_FORMADO>
```

Escolhemos implementar a primeira abordagem, já que ela permite uma identificação precisa de onde está o erro, preservando a informação dos outros identificadores formados. Assim, o analisador sintático irá lidar com esse erro posteriormente.

2.2 Autômato de Operadores Relacionais

O trecho do autômato que realiza a identificação de operadores relacionais está presente na Figura 2.

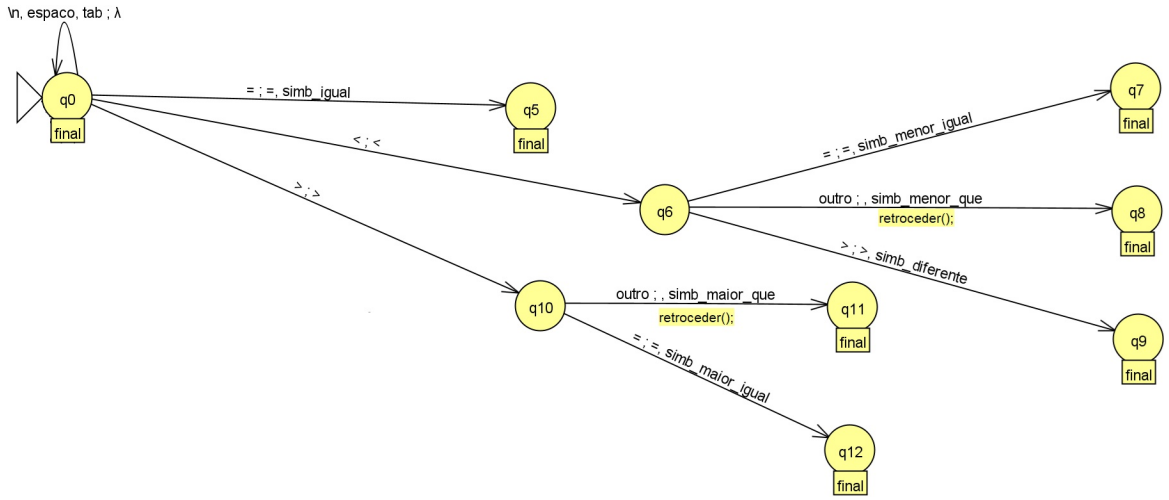

Figura 2 – Autômato de Operadores Relacionais.

Note que, na gramática PL/0, existem os operadores relacionais `'='`, `'<'`, `'>'`, `'>='`, `'<='`, `'<>'`.Como o autômato verifica a entrada caractere por caractere, é preciso verificar o

que vem após o primeiro símbolo para classificá-lo de forma correta. Assim, após identificar, por exemplo, um símbolo `'<'`, há três possibilidades: receber em seguida um símbolo `'='`, o que denotaria que esse conjunto forma um simb_menor_igual; receber em seguida um símbolo `'>'`, o que denotaria que esse conjunto forma um símbolo `'<>'`; ou receber em seguida um símbolo diferente desses, o que denotaria que esse é um símbolo do tipo `"simb_menor_que"`. Perceba que, quando o autômato recebe um outro símbolo, é necessário retroceder o ponteiro para identificá-lo e classificá-lo posteriormente.

2.3 Autômato de Pontuação e Operações Aritméticas.

O trecho do autômato que realiza a identificação de símbolos de pontuação e operações aritméticas está presente na Figura 3.

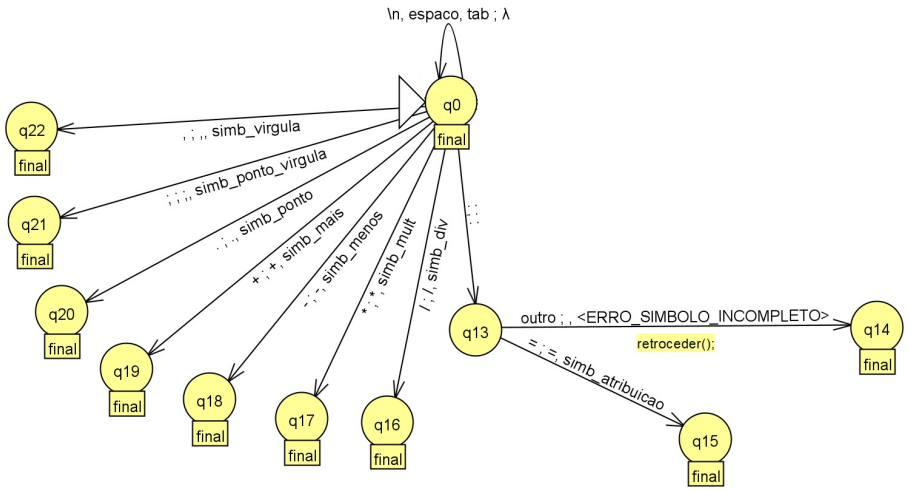

Figura 3 – Autômato de Pontuação e Operações Aritméticas.

O autômato que identifica operações aritmética recebe uma das operações aceitas pela gramática, ou seja, `'+'`, `'-'`, `'*'`, `'/'`, e imprime-a na transição juntamente com sua classificação.

Já o autômato de pontuação funciona da mesma maneira, considerando os símbolos `'.'`, `','`, `';'`. Há um aspecto importante a ser considerado: na gramática PL/0, existe o símbolo de atribuição composto por `':='`. Porém, não há o símbolo de `':'` sozinho. Assim, se o autômato receber o símbolo `':'`, é preciso verificar dois casos: no primeiro, o autômato recebe um sinal de `'='` e classifica o conjunto dos símbolos como `"simb_atribuicao"`. No segundo caso, ele recebe qualquer símbolo diferente de `'='`. Isso significa que o usuário tentou utilizar o símbolo `':'` sozinho no código, o que não existe nessa linguagem. Portanto, nesse cenário, a máquina de Mealy imprime uma mensagem de erro de símbolo incompleto e retrocede o ponteiro para analisar o símbolo que foi lido nessa transição.

2.4 Autômato de Comentários e Caracteres Inválidos

O trecho do autômato que realiza a identificação de comentários e caracteres inválidos está presente na Figura 4.

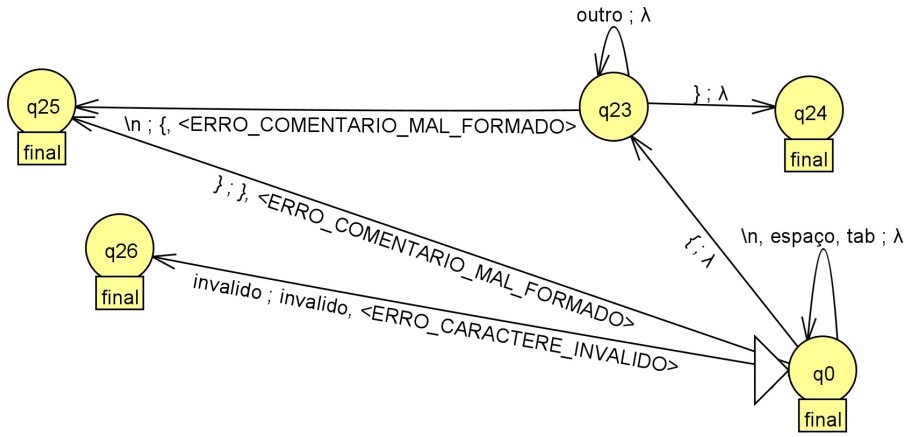

Figura 4 – Autômato de Comentários e Caracteres Inválidos.

Os comentários são uma ferramenta para auxiliar na legibilidade do código, assim, não devem ser considerados na análise léxica do código. Na gramática PL/0, os comentários são feitos linha a linha entre chaves. Assim, quando o autômato identifica um `'{'`, ele deve permanecer lendo tudo o que estiver posteriormente até que encontre um `'\n'` . Porém, o usuário pode ter esquecido de fechar o comentário na mesma linha, o que geraria um problema para o funcionamento do código. Assim, se no estado q_{23} o autômato receber uma quebra de linha, indicada por `'\n'`, é preciso indicar um erro de comentário mal formado. Similarmente, se o autômato detectar um `'}'` no estado q_0, isso indica que houve uma tentativa de fechar um comentário sem haver um comentário já aberto, o que também indica um erro.

Já o autômato de caracteres inválidos consiste apenas no estado q_{26}. Caso o autômato receba um caractere não aceito pela gramática, como '@' ou '?', a transição para o estado q_{26} indica um erro.

3 Implementação do Analisador Léxico

3.1 Parser

Um dos desafios primordiais do projeto foi a transformação dos autômatos em JFLAP em um formato de dados que possa ser utilizado pelo interpretador léxico. Objetivou-se evitar qualquer tipo de *hardcode* da máquina de estados no código em C. Para tanto, constatou-se que os autômatos JFLAP são salvos (extensão .jff) em formato XML. Assim, optamos por escrever

um *parse*r capaz de traduzir o formato XML em um CSV (comma separated values), em que cada linha representa uma linha da matriz de transição da máquina da estados.

Uma vez que o *parser* não é parte integrante do analisador léxico, escolhemos implementá-lo em linguagem Python, a fim de utilizar as bibliotecas da linguagem que facilitam o *parsing* de arquivos XML.

3.2 Utilizar a Tabela de Transições

Como mencionado anteriormente, o *parser* é responsável por traduzir o arquivo .jff em um .csv. Em seguida, criamos um arquivo que gera a tabela de transição a partir da leitura do arquivo csv, chamado `transition_matrix_creator.c`. A estrutura de dados utilizada é uma matriz na qual cada linha representa um estado e cada coluna representa um possível caractere lido, totalizando 256 colunas (supõe-se que os caracteres de entrada são ASCII).

Cada posição da matriz é preenchida por um objeto (struct) que conta com três campos: próximo estado, output (pois trata-se de uma máquina de mealy) e uma flag indicando se o estado é final ou não. Inicialmente, a matriz é inteiramente preenchida com a indicação de que o caractere lido é inválido (não existente no alfabeto da linguagem). Durante a leitura do .csv, os caracteres válidos serão progressivamente preenchidos, deixando apenas caracteres que não existem na linguagem ('?' e '@', por exemplo), apontando para um estado de erro léxico.

O .csv é lido linha por linha, preenchendo a matriz já inicializada na RAM. Um cuidado adicional foi tomado para modelar algumas transições especiais da máquina de estado. Por exemplo, estados cuja entrada no .jff aparecem como 'outro' devem ser cuidadosamente modelados durante o carregamento da matriz de transições. Assim, tratam-se os diferentes casos especiais, substituindo todos as colunas que correspondem a cada uma das macros com a informação de transição. É interessante notar que esse processo funciona de maneira similar a expressões regulares, apenas utilizando diferentes macros para simbolizar diferentes cadeias. Por exemplo, a macro "outro_}"denota todos os caracteres exceto o '}'.

3.3 Executando a máquina de estados

Uma vez que a matriz de transições é carregada, a implementação da máquina de estados é razoavelmente direta. De fato, pode-se argumentar que a matriz de transições atua de maneira similar a uma matriz de adjacências, e cada estado futuro da máquina é dependente apenas do estado atual e do caractere lido no momento. Assim, basta acessar a posição da matriz de transições correspondente à linha referente ao estado atual, a coluna referente ao caractere lido para obter o próximo estado e o output. Ademais, a flag de estado final é utilizada para

identificar o final de uma cadeia e o retorno ao analisador sintático. Em alguns casos, como, por exemplo, na identificação de identificadores e dígitos, é necessário retroceder na fita de leitura.

4 Exemplo de Execução

Para compilar e rodar o analisador léxico, é preciso abrir um terminal na pasta onde estão localizados os arquivos do projeto, junto com o Makefile. Assim, basta rodar o seguinte comando:

```
make run FILE_PATH=caminho_para_meu_arquivo
```

Na qual *caminho_para_meu_arquivo* indica o caminho para o arquivo que deve ser analisado. Como exemplo, dentro do diretório /data deixam-se alguns exemplos possíveis.

Por exemplo,

```
make run FILE_PATH=./data/ex1
```

No qual ex1 é:

ex1:
VAR n fat;
{meu programa bem legal}
BEGIN n := 4;
END.

Gerando a saída:

ex1 saída:
VAR, VAR
n, identificador
fat, identificador
;, simb_ponto_virgula
BEGIN, BEGIN
n, identificador
:=, simb_atribuicao
4, numero
;, simb_ponto_virgula
END, END
., simb_ponto

Para compilar o exemplo 2, basta compilar o seguinte comando no mesmo local:

```
make run FILE_PATH=./data/ex2
```

ex2:
VAR END ger@ 2n 1@; @ 12345 BEGIN;
n := 4;
: {qq} :==-{ END.

Portanto, gerando a saída:

ex2: saída:
VAR, VAR
END, END
ger, identificador
@, <ERRO_CARACTERE_INVALIDO>
2, numero
n, identificador
1, numero
@, <ERRO_CARACTERE_INVALIDO>
;, simb_ponto_virgula
@, <ERRO_CARACTERE_INVALIDO>
12345, numero
BEGIN, BEGIN
;, simb_ponto_virgula
n, identificador
:=, simb_atribuicao
4, numero
;, simb_ponto_virgula
:, <ERRO_SIMBOLO_INCOMPLETO>
:=, simb_atribuicao
=, simb_igual
-, simb_menos
{, <ERRO_COMENTARIO_MAL_FORMADO>
END, END
., simb_ponto

5 Conclusão

A implementação de um analisador léxico proporciona o aprendizado prático de uma das etapas fundamentais da compilação. Em específico, a equipe pôde aprofundar-se na implementação técnica de inúmeros conceitos abordados durante as aulas da disciplina, de máquinas de estados a tabelas de transição.

Além disso, numerosas decisões de *design* foram realizadas. Por exemplo, optou-se por uma solução flexível, ao invés de *hard-coded*. Assim, este projeto respeita o princípio *Open/Closed* da Engenharia de Software: o código é fechado para mudanças, contudo, novas funcionalidades podem ser facilmente adicionadas por meio de extensões ao código. Em particular, isso pode ser realizado modificando diretamente o arquivo JFLAP com a máquina de estados. Com novos estados introduzidos, basta rodar o *parser* para gerar a versão em .csv da máquina de estados, e os novos comportamentos estarão automaticamente ativos, sem nenhuma alteração ao código fonte do projeto.

Ainda que opção por um *design Open/Closed* tenha gerado desafios adicionais de implementação (por exemplo, desenvolvimento de um *parser* dedicado), nossa equipe acredita que esses esforços suplementares valem a pena. Em conclusão, pode-se afirmar que o desenvolvimento do presente analisador léxico promoveu aprendizados valiosos, os quais serão imprescindíveis no próximo projeto prático.