Compiladores

Análise Léxica (Parte 1)

Introdução

Este material contém as notas de aula sobre *Análise Léxica*, da disciplina *Compiladores*. O objetivo é conhecer esta que é a primeira análise realizada por um tradutor. Serão explicados alguns conceitos importantes, a sequência de ações realizadas pelo analisador léxico e os erros identificados nesta fase. No fim do material, aspectos de implementação de analisadores léxicos são discutidos.

Como já visto nas Notas de Aula sobre Fases da Compilação, a análise é a primeira etapa em qualquer software de tradução, seja ele de duas ou três fases. Conhecer seus componentes é fundamental para compreender como, a partir da linguagem fonte, se chega até o processo de geração de uma representação intermediária.

Análise

Como já mencionado, a primeira ação do compilador é realizar a ***análise***. Na etapa de análise, há três análises a serem realizadas:

1. léxica
2. sintática
3. semântica

O compilador inicia pela ***análise léxica***.

A sequência dos estudos requer a revisão de alguns conceitos, a saber:

- **Alfabeto**: conjunto de símbolos ou caracteres (MENEZES, s.d.).
    - Exemplos: {a, b, c}, {a, b, c, 4, 5, 6, 7, $, &, *}.
- **Símbolo** ou **caractere**: entidade abstrata básica, não definida formalmente.
    - Exemplos: letras e dígitos (MENEZES, s.d.).
- **Palavra**: sequência finita de símbolos justapostos (MENEZES, s.d.).
    - Exemplos: abcd, Olá.
    - Em uma linguagem de programação, uma palavra é um programa (MENEZES, s.d.).
- **Linguagem**: Conjunto de *strings* definidas sob um alfabeto (FERREIRA, s.d.).
- **Autômato**: modelo matemático que descreve um sistema de processamento de símbolos de entrada de acordo com um conjunto de regras. Um autômato é composto por cinco informações: um conjunto de estados, um alfabeto de entrada, regras de transição, estado inicial e um conjunto de estados finais. Resumidamente, é uma máquina abstrata que consome uma sequência de símbolos de entrada.
    - Um autômato finito é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$, onde (adaptado de CAMBUIM, s.d.):
    1. Q é um conjunto finito de estados.
    2. Σ é um conjunto finito denominado alfabeto.
    3. $\delta: Q \times \Sigma \rightarrow Q$ é a função de transição
    4. q_0 é o estado inicial ($q_0 \in Q$)
    5. F é o conjunto de estados finais ($F \subset Q$).
- **Gramática**: conjunto de regras que determinam como palavras (e frases) são formadas.
- **Gramática Livre de Contexto**: gramática onde todas as regras de produção são da forma $A \rightarrow B$, onde A deve ser um símbolo não terminal e B é uma cadeia de terminal e/ou não terminais, sendo que B pode ser inclusive a sentença vazia.
- **Gramática Regular**: é uma Gramática Livre de Contexto que deve ser linear à direita ou linear à esquerda, ou seja, suas regras de produção possíveis têm uma das seguintes formas:
    - $A \rightarrow aB$
    - $A \rightarrow Ba$
    - $A \rightarrow a$

Análise Léxica

A Análise Léxica é a primeira etapa de análise do compilador. Essa etapa é realizada por um componente chamado ***analisador léxico***, também conhecido como ***scanner***.

Uma das ações do *scanner* é verificar se todos os caracteres do programa fonte pertencem ao alfabeto da linguagem. Sendo assim, diz-se que o *scanner* é um programa que implementa um autômato finito, reconhecendo (ou não) *strings* como símbolos válidos de uma linguagem (RICARTE, 2003). Ao realizar a leitura dos símbolos, começa também o preenchimento da tabela de símbolos, que será detalhada futuramente.

Para estudar análise léxica, deve-se conhecer os conceitos de ***léxico*** e ***lexema***. Léxico é o conjunto formado por todas as palavras de uma linguagem. Lexema é uma unidade mínima de uma linguagem, aplicada a uma família de palavras relacionadas por forma e/ou significado.

Na análise léxica, separam-se os lexemas de cada expressão do código. No contexto de Compiladores, um lexema é "*uma sequência de caracteres do programa fonte que corresponde ao padrão de um token e é identificado pelo analisador léxico como uma instância desse token*" (AHO et al., 2007).

A partir das explicações acima, pode-se citar como exemplos de lexemas:

- `x`
- `y`
- `:=`
- `=`
- `;`
- `while`
- `tmp`
- `15`

De uma forma resumida, a análise léxica deve quebrar o texto do programa fonte em lexemas, verificar a categoria a qual cada lexema pertence e produzir uma sequência de símbolos

léxicos chamados de ***tokens***[1] (MARANGON, 2017).

Um ***token*** (ou ***item léxico***) é um par que consiste em um nome de *token* e um valor de atributo opcional. O nome do *token* é um símbolo abstrato que representa um tipo de unidade lexical, tal como uma palavra-chave específica ou uma sequência de caracteres de entrada que denotam um identificador. Os nomes dos *tokens* são os símbolos de entrada que o analisador processa (AHO et al., 2007).

A Figura 1 mostra informações sobre *tokens*. Para cada token, a figura exibe seu padrão, exemplos de lexemas que podem estar associados a ele e uma descrição.

Token	Padrão	Lexema	Descrição
`<const, >`	Sequência das palavras c, o, n, s, t	const	Palavra reservada
`<while, >`	Sequência das palavras w, h, i, l, e	while, While, WHILE	Palavra reservada
`<if, >`	Sequência das palavras i, f	If, IF, iF, If	Palavra reservada
`<=, >`	<, >, <=, >=, ==, !=	==, !=	
`<numero, 18>`	Dígitos numéricos	0.6, 18, 0.009	Constante numérica
`<literal, "Olá">`	Caracteres entre ""	“Olá Mundo”	Constante literal
`<identificador, 1>`	Nomes de variáveis, funções, parâmetros de funções.	nomeCliente, descricaoProduto, calcularPreco()	Nome de variável, nome de função
`<=, >`	=	=	Comando de atribuição
`<{, >`	{, }, [,]	{, }, [,]	Delimitadores de início e fim

Figura 1. Exemplos de uso dos termos durante a análise léxica. Fonte: MARANGON, 2017.

[1] É importante observar que token possui como sinônimos símbolo léxico e item léxico. Essas expressões não são sinônimos de lexema.

Na análise léxica, portanto,

- o texto é quebrado em lexemas
- os *tokens* são produzidos
- inicia a geração da tabela de símbolos
- elimina espaços em branco
- elimina comentários
- elimina tabulações (`\t`) e nova linha (`\n`)
- detecta e informa erros léxicos

Exemplos de erros léxicos:

- Uso de caracteres que não pertencem ao alfabeto da linguagem utilizada.
- Uso de caracteres não aceitos em determinadas construções da linguagem (exemplo: nome de variável).
- Não utilizar um delimitador final de comentário (encerra o arquivo fonte inesperadamente).
- Atribuição de variável com valor inferior ao limite mínimo ou superior ao limite máximo para constantes.

Os itens léxicos são divididos em categorias/classes, conforme mostra a coluna *Descrição* da Figura 1. Todo item faz parte de alguma categoria. Os itens podem ser categorizados como:

- Palavras reservadas: `while`, `do`, `goto`, `if`, `break`.
- Identificadores: `main`, `getTamanho`, `x`, `num`.
- Constantes: `1094`, `3`, `-9`.
- Operadores: `=`, `>=`, `<=`, `-`, `*`, `/`, `+`.
- Textos: "`Ricardo`", "`Hello, world`".
- Delimitadores: `{} ; , ()`.

A sequência de *tokens* gerados é passada para a árvore sintática. Na etapa de análise léxica, a relação entre os lexemas não é analisada. Isto é feito a partir da análise sintática.

Segundo Ricarte (2003), para desempenhar a análise léxica, o compilador deve ter conhecimento de quais são os *tokens* válidos da linguagem, assim como suas palavras chaves e regras para formação de identificadores. Por exemplo, a declaração abaixo deve resultar em

erro nesta etapa da análise, pois `1var` não é um identificador válido em C:

```
int 1var; // Em C, variáveis podem conter apenas letras (maiúsculas ou minúsculas), números (exceto se for o primeiro caractere) e underline ("_").
```

Como seriam as regras de produção de uma gramática livre de contexto que gera os nomes aceitos para variáveis em C?

S → AB

A → letra | _

B → letra B | dígito B | _ B | B | Ø

letra → a | b | c | ... | Y | Z

dígito → 0 | 1 | ... | 9

E como seriam as regras de produção para uma gramática regular?

id → [letra | _] [letra | _ | dígito]*

letra → a | b | c | ... | Y | Z

dígito → 0 | 1 | ... | 9

A Figura 2 exibe um autômato que representa os estados possíveis para aceitar nomes de identificadores na linguagem Java. Considerando o autômato da Figura, responda:

a. Que autômato é este?
b. O símbolo cifrão (`$`) é um nome de identificador válido?

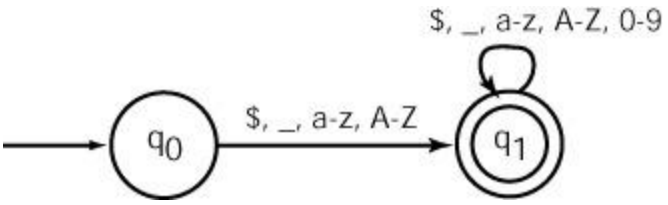

Figura 2. Autômato representando os estados possíveis para aceitar identificadores em Java.

Observando o trecho de código do quadro a seguir, quais são os lexemas presentes e suas respectivas categorias? E os *tokens*?

```
int y = 1;
```

```
int x = 0;
x = x + y * 10;
```

Os lexemas e suas respectivas categorias são:

Lexema	Categoria
int	Palavra reservada
y	Identificador
=	Operador
1	Constante
;	Delimitador
int	Palavra reservada
x	Identificador
=	Operador
0	Constante
;	Delimitador
x	Identificador
=	Operador
x	Identificador
+	Operador
y	Identificador
*	Operador
10	Constante
;	Delimitador

Os *tokens* são:

<int, >

<id, 1>
<=, >
<constante, 1>
<;, >
<int, >
<id, 2>
<=, >
<constante, 0>
<;, >
<int, >
<id, 2>
<=, >
<id, 2>
<+, >
<id, 1>
<*, >
<constante, 10>
<;, >

É possível notar que os *tokens* foram criados no formato `<v1, v2>`, pois pela definição vista eles são formados por um par que contém um nome e, opcionalmente, um valor. No caso de identificadores, o token possui o formato `<id, num>`, onde `id` representa a categoria de identificadores e `num` é um valor numérico iniciado em `1`. Isso foi feito porque ao gerar os *tokens* inicia-se a construção da tabela de símbolos. Os identificadores estarão nesta tabela, cada um em uma entrada (linha). Desta forma, uma tabela de símbolos resumida[2] poderia ser criada da seguinte maneira:

Linha	**Nome**
1	`y`
2	`x`

[2] Uma tabela de símbolos real contém mais informações, a serem vistas futuramente.

Observação: São encontradas outras categorias na literatura, bem como outras formas de representação de *tokens*. Por exemplo, em vez de *constante* é possível encontrar o termo *número*. Há também livros que apresentam diretamente o identificador em vez da sua posição na tabela de símbolos. Há também autores que removem a vírgula quando o segundo item fica em branco, substituindo, por exemplo, `<int, >` por `<int>`.

Após esta breve introdução sobre o funcionamento do analisador léxico de um compilador, considere a instrução abaixo e responda:

```
x = (y + tmp) * 30;
```

1) Quais são os lexemas da expressão e suas respectivas categorias?

Lexema	**Categoria**
x	Identificador
=	Operador
(	Delimitador
y	Identificador
+	Operador
tmp	Identificador
)	Delimitador
*	Operador
30	Constante
;	Delimitador

2) Quais são os tokens da expressão?

x	<identificador, 1>
=	<=, >
(	<(, >
y	<identificador, 2>
+	<+, >

tmp <identificador, 3>

) <), >

* <*, >

30 <constante, 30>

; <;, >

3) Como fica a tabela de símbolos resumida para esta expressão?

Linha	**Nome**
1	`x`
2	`y`
3	`tmp`

Motivação para a Análise Léxica

As principais motivações para a existência de uma etapa distinta somente para análise léxica são a **organização** e a **simplicidade**. A separação entre análise léxica e sintática simplifica a implementação das duas fases. Além disso, aspectos específicos da arquitetura, tais como alfabeto utilizado e gerência de arquivos, ficam separados na análise léxica.

Implementação da Analisadores Léxicos

As técnicas de implementação de analisadores léxicos são as mesmas usadas na implementação de qualquer programa que precise reconhecer padrões de texto (sequências de caracteres), tais como processadores de texto. Portanto, a implementação de um analisador léxico é basicamente a implementação de um reconhecedor de linguagens regulares.

Essa implementação pode ser feita de duas formas: manual ou usando ferramentas (DU BOIS, 2011).

Independentemente da técnica de implementação adotada, o analisador léxico (*scanner*) funciona como uma subrotina do analisador sintático (*parser*). Os *tokens* não são passados

todos de uma vez; em vez disso, o *scanner* possui uma rotina do tipo `obter_proximo_token()` que vai sendo chamada pelo analisador sintático à medida em que este vai construindo a árvore sintática.

A dinâmica do fluxo que envolve esta etapa da análise está presente graficamente na Figura 3.

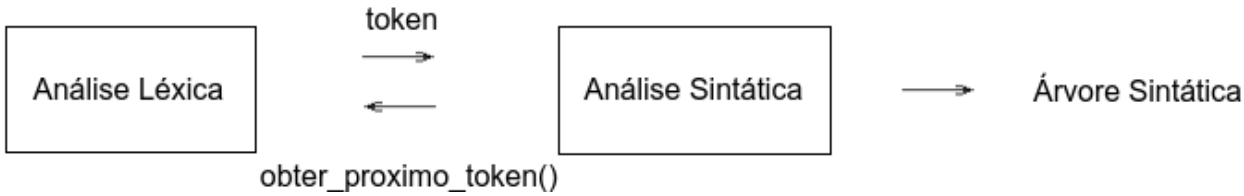

Figura 3. Fluxo de passagem de *tokens* do *scanner* para o *parser* até a geração da árvore sintática.

Existem diversas ferramentas geradoras de analisadores léxicos. As principais são:

- `lex`
- `flex` (`sudo apt-get install flex`)
- `perl`
- `javacc`
- `awk`
- `shell linux`

Ricarte (2003) diz que uma das ferramentas mais tradicionais é o programa `lex`, originalmente desenvolvido para Unix. O objetivo da ferramenta `lex` é gerar uma rotina para o *scanner* em C a partir de um arquivo de especificação, que deve conter a especificação das expressões regulares e trechos de código C do usuário que serão executados quando sentenças daquelas expressões forem reconhecidas.

CURIOSIDADE

A ferramenta `lex` foi utilizada na construção da calculadora `bc`, do `gcc` e do próprio programa `lex`.

Exemplo: `echo "((1+10*30)-4/2)" | bc`

Manual

Implementação pura em uma linguagem de programação. O pseudocódigo do quadro a seguir mostra, em linhas gerais, como deve ser essa implementação:

```
caractere = getChar();
enquanto (caractere == ' ') {
      caractere = getchar();
}
se (ehLetra(caractere)) {
      enquanto (ehLetra(caractere) || ehDigito(caractere)) {
            identificador = identificador ++ caractere;
            caractere = getchar();
      }
}
```

Observação: aqui não constam caracteres especiais que eventualmente linguagens de programação podem conter, tais como "_" (underline), "$" (cifrão) e outros.

> "*Os geradores de analisadores léxicos automatizam o processo de criação do autômato finito e o processo de reconhecimento de sentenças regulares a partir da especificação de expressões regulares. Essa ferramentas são comumente chamadas de lex. Atualmente há diversas implementações de lex para diferentes linguagens de programação*" (MARANGON, 2022).

Implementação por ferramentas

Recebem os padrões de reconhecimento dos *tokens* (geralmente expressões regulares) e devolvem como resposta um programa que reconhece as expressões.

Exemplo: Expressões regulares → Flex → Analisador Léxico.

O Flex será visto em mais detalhes futuramente.

Leituras Sugeridas

- Material de Ricarte (2003), contendo regras para a construção de expressões regulares: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node52.html.
- Material de Marangon (2017) sobre Análise Léxica: https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html.
- Material de Peter Jandl Jr. (Arquivo Flex.pdf), disponível no repositório de materiais complementares.
- Vídeo do professor Tiago Fernandes Tavares sobre (f)lex.

Conclusão

A Análise Léxica é a primeira etapa de análise realizada pelo compilador. Nessa etapa, o analisador léxico (*scanner*) separa os lexemas, produz os *tokens* e informa eventuais erros léxicos. É também a partir desta análise que começa a construção da tabela de símbolos. Os tokens criados são utilizados pelo analisador sintático.

É possível criar *scanners* manualmente ou por meio de ferramentas, tais como o `flex`, que o faz por meio de expressões regulares. O conhecimento sobre expressões regulares será importante para o uso desta ferramenta, explanada no próximo material.

Exercícios

1) Existe(m) erro(s) léxico(s) no código abaixo? Se sim, qual?

```
#include <stdio.h>
// Função principal
int main(int argc, char const *argv[]) {
    short dvar = 4444;
    short a = 1;
```

```
    short $teste$ = 5555;
    short #teste# = 666;

    printf("%hd\n", dvar + a);
    printf("%hd\n", $teste$);
    printf("%hd\n", #teste#);

    return 0;
    /* Fim do código
}
```

2) Qual é o erro léxico do código abaixo e como ele pode ser corrigido?

```
#include <stdio.h>

int main() {
    short $teste$ = 666666 + 1;
    printf("%hd\n", $teste$);
    return 0;
}
```

3) Qual é a diferença entre *token* e lexema?

Referências

AHO, Alfred V. et al. Compilers: principles, techniques, & tools. Pearson Education India, 2007.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 18 ago. 2011.

MARANGON, Johni Douglas. Análise Léxica. 2017. Disponível em: https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html. Acesso em: 14 mai. 2024.

RICARTE, Ivan Luiz Marques. 2003. Compiladores. Disponível em: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node37.html. Acesso em: 14 mai.

2024.

RICARTE, Ivan Luiz Marques. 2003. Analisadores léxicos. https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node50.html. Acesso em: 14 mai. 2024.

Compiladores. http://professor.pucgoias.edu.br/SiteDocente/admin/arquivosUpload/17389/material/Aulao.pdf. Acesso em: 14 mai. 2024.

MENEZES, Paulo Blauth. Linguagens Formais e Autômatos. Disponível em: http://www.ic.uff.br/~ueverton/files/LF/aula02.pdf. Acesso em 14 mai. 2024.

CAMBUIM, Lucas. Aula 3: Autômatos e Linguagens (cap. 1) Linguagens Regulares. Disponível em: https://www.cin.ufpe.br/~lfsc/cursos/teoriadainformacao/unidade%201/cap%201_2%20-%20automatos%20e%20linguagem%20descricao%20formal.pdf. Acesso em 14 mai. 2024.

FERREIRA, Anderson Almeida. Linguagens Formais e Autômatos (BBC242). Disponível em: http://www.decom.ufop.br/anderson/BCC242/aula0.pdf. Acesso em: 14 mai. 2024.