UNIVERSIDADE SÃO JUDAS TADEU - USJT
CIÊNCIAS DA COMPUTAÇÃO

Conceitos - Analisador léxico

Alexandre H. Vieira Rodrigues RA: 820145437
Arthur Santos Matumoto RA: 820145865
Bruno Lorimier Moreira RA: 820141738
Karen Mujica Mendoza RA: 820145148
Lucas Pires Latorre RA: 82118391
Marcelo M. Ferreira RA: 821159191
Ryan Pizani RA: 819230704

Prof ª Érica Oliveira da Silva
Prof. José Carmino Gomes Junior
Prof. Calvetti

SUMÁRIO

1. Interpretação x Compilação **4**

2. Compilador **5**

2.1 Processo de compilação 7

3. Expressões Regulares **9**

3.1 Conceitos básicos das expressões regulares 10

3.2 Expressões regulares - Análise Léxica 11

4. Gramática livres de contexto **12**

4.1 Árvore de derivação 13

5. Autômatos finitos **14**

5.1 Algoritmo de Thompson 15

6. Análise léxica **16**

6.1 Conceitos básicos - Analisador Léxico 16

6.2 Associação com linguagens regulares 17

6.3 Analisador léxico e autômatos finitos 18

7. Tabela de símbolos **20**

Referências **22**

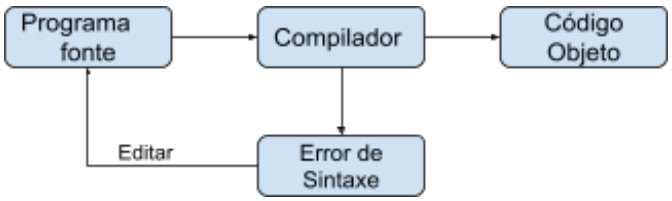
Resumo

Na ciência da computação, análise léxica é o processo que converte uma sequência de caracteres em uma sequência de tokens. O analisador léxico ou scanner, é um programa que implementa um autômato finito, reconhecendo strings como símbolos válidos de uma linguagem. No compilador o analisador léxico é o primeiro componente a entrar em contato com o código fonte. Sua função básica é ler os caracteres de entrada e agrupá-los em tokens da linguagem, os identificadores, os símbolos especiais, as palavras chaves e os números.

Palavras-chave: tokens, autômato finito, scanner, código-fonte.

1. Interpretação x Compilação

As linguagens de programação são utilizadas para escrever todos os tipos de software que estão sendo executados pelos computadores. A maior parte da programação de um programa é realizada utilizando uma linguagem de programação de alto nível. Essa linguagem de alto nível pode ser muito diferente da linguagem da máquina, então para poder executá-la usa-se o compilador.



Um compilador trata-se de um programa que recebe como entrada um código fonte em uma determinada linguagem de alto nível e, a partir daí, cria um programa semanticamente equivalente, porém em outra linguagem, código de objeto (linguagem de máquina), específica para um processador e sistema operacional.

Ao longo desse processo de tradução, o compilador vai tentar identificar e relatar erros inseridos no código fonte. Quando utilizamos linguagens de alto nível para programar , temos um grande impacto na velocidade de desenvolvimento dos programas. Devido:

- A notação usada pelas linguagens de programação pode ser facilmente compreendida pelos humanos.
- O compilador pode detectar alguns erros de programação óbvios cometidos pelo programador.
- Programas escritos em uma linguagem de alto nível tendem a ser mais curtos.
- O mesmo programa pode ser compilado em muitas linguagens de máquina diferentes, portanto, executado em muitas máquinas diferentes.

Programas escritos diretamente em linguagem de máquina, são traduzidos de forma mais rápida. Por tanto é possível programas que são parcialmente escritos em linguagem de máquina.

O compilador se diferencia de um interpretador. O interpretador irá ler o seu código linha a linha em tempo de execução e irá traduzir cada linha para uma linguagem alvo (que geralmente será bytecode ou linguagem de máquina). No mesmo momento que ele traduz, ele executa o código gerado, sem processos adicionais.

No processo de compilação, é necessário primeiro compilar para somente depois executar. A criação de um programa compilado requer várias etapas. Primeiro, o programador escreve o código-fonte, para em seguida fazer a compilação do programa, prossegue com a tradução de tudo em código de máquina para o computador entender.

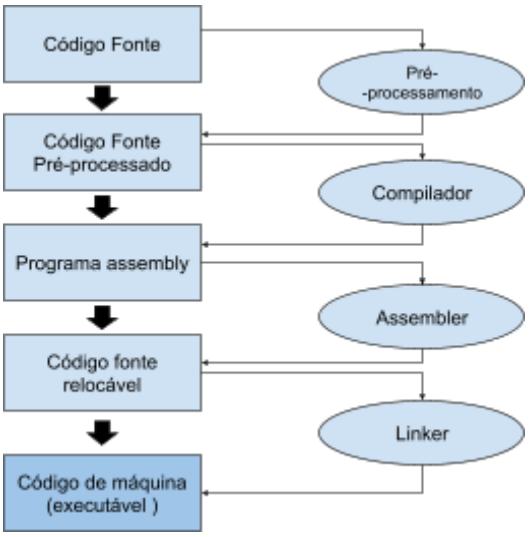
Um código compilado consome menos recursos da CPU, durante as fases de tradução e carregamento. Porém sua execução não é direta. Primeiro é preciso compilar, para somente depois executar.

Um programa interpretado tem um intervalo menor entre a codificação e a execução do programa. Além disso, costuma também ter mais facilidades na linguagem e os erros são mais fáceis de serem encontrados. Porém, tem maior custo computacional e a execução é mais lenta.

2. Compilador

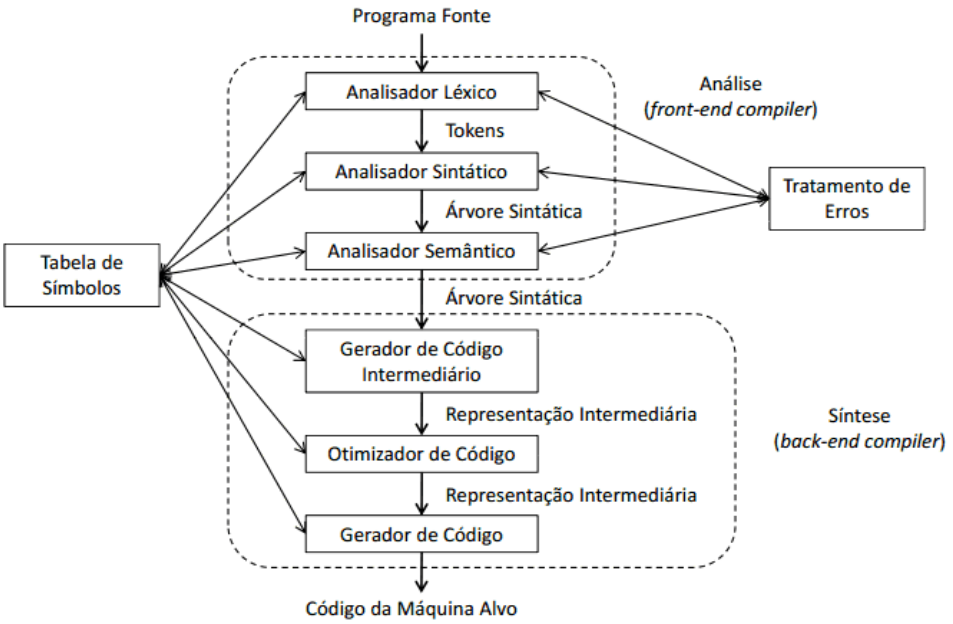
Um compilador é um programa de sistema que traduz um programa descrito em um linguagem de alto nível para um programa equivalente em código de máquina para um processador. Em geral, um compilador não produz diretamente o código de máquina, mas sim um programa em linguagem simbólica (assembly) semanticamente equivalente ao programa em linguagem de alto nível. O programa em linguagem simbólica é então traduzido para o programa em linguagem de máquina através de montadores.

- Pré-processador para preparar o código-fonte para o compilador adequado.
- O Compilador sendo apropriado para consumir a saída limpa do pré-processador, ele irá verificar e analisar o código-fonte, executar a verificação de tipos e outras rotinas semânticas, bem como produzir a linguagem Assembly como saída.
- Montador (assembler) irá consumir o código do Assembly e produzir o código-objeto (código da máquina). Nesse momento, o código-objeto ainda não é completamente executável, ainda há lacunas que devem ser preenchidas pelo linker.
- O linker é um utilitário que costuma ser complementar ao compilador. Ele pega um código binário gerado pelo compilador e junta tudo, tornando ele em um executável.



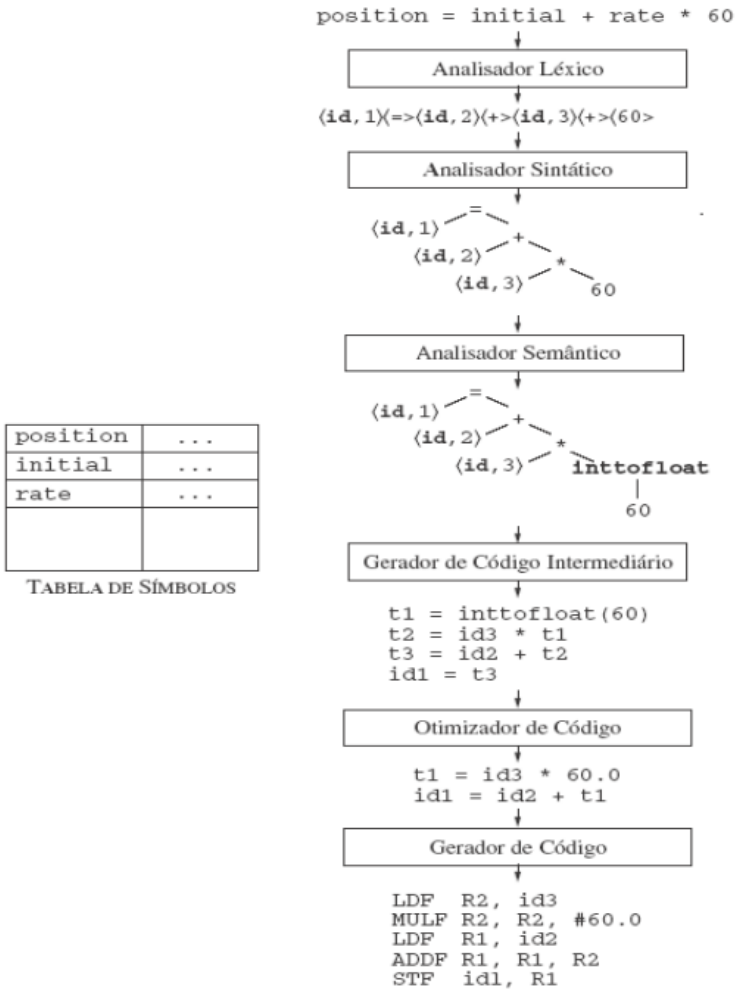
O compilador mapeia um programa de origem em um programa de destino que seja semanticamente equivalente. Esse mapeamento é dividido em duas partes:

- Análise → O programa de origem é dividido em partes constituintes, impondo uma estrutura gramatical. Essa estrutura é utilizada para criar uma representação intermediária do programa de origem. Qualquer pequeno engano no código-fonte, será suficiente para impedir a criação do arquivo executável e o compilador gerará uma mensagem de erro.
- Síntese → O programa destino é construído com base na representação intermediária e na tabela de símbolos.



2.1 Processo de compilação

O processo de compilação se dá por sequências. A tabela de símbolos é usada por todas as fases do compilador.



Principais fases de compilação:

- Análise Léxica → A primeira fase da compilação, onde o fluxo de caracteres do programa de origem é lido e também ocorre o agrupamento dos caracteres em sequências (lexemas). Para cada lexema, o analisador léxico vai produzir um **token** como saída, que será repassado para a seguinte fase.

  { nome-token, valor-atributo}

  As informações de entrada da tabela de símbolos são necessárias para o analisador

semântico e gerador de código.

- Análise sintática → A segunda fase do processo de compilação, na qual o analisador utiliza os primeiros componentes dos tokens produzidos pela análise léxica e cria uma representação intermediária semelhante a uma árvore (fluxo de tokens). Na árvore, podemos ver a ordem em que as operações devem ser realizadas.

- Análise semântica → A partir dessa árvore, o analisador semântico verifica, no programa de origem, se há consistência semântica considerando a linguagem definida, tomando como base as informações na tabela de símbolos. Depois, o analisador semântico junta todas as informações de tipo e guarda na árvore de sintaxe ou na tabela de símbolos. A verificação de tipo (type checking) é uma parte crucial dessa fase.

- Geração de código intermediário → Grande parte dos compiladores passa a gerar uma representação intermediária explícita de baixo nível. Essa representação intermediária deve levar em conta duas propriedades essenciais: (1) ela deve ser fácil de produzir e (2) deve ser fácil de traduzir para a máquina de destino.

- Otimização de código → A fase de otimização tenta aprimorar o código intermediário para que o código de destino seja melhor em termos de velocidade, ou outras coisas relacionadas ao seu tamanho (mais curto) ou consumo de energia.

- Geração de código → Como entrada temos uma representação intermediária do programa de origem. Em seguida, começa o mapeamento para a linguagem de destino. Caso o idioma de destino seja o código de máquina, são selecionados os registros ou locais de memória para cada uma das variáveis que foram utilizadas no programa. Lembrando que é essencial que a atribuição de registradores para conter variáveis seja criteriosa.

- Gerenciamento da tabela de símbolos → Essa tabela é uma estrutura de dados que possui um registro para cada nome de variável, com campos para os atributos do nome. Essa estrutura deve ser projetada para permitir que o compilador encontre o registro para cada nome de forma rápida. O agrupamento dessas fases consiste na organização lógica de um compilador.

3. Expressões Regulares

Expressões regulares ou regex são uma forma simples e flexível de identificar cadeias de caracteres em palavras, ou seja, são formas concisas de descrever um conjunto de strings que satisfazem um determinado padrão, uma vez que diferentes tipos de usuários e desenvolvedores precisam compreendê-los. As expressões regulares podem facilitar a validação de dados, aumentam a produtividade e reduzem o tempo de busca em função dos padrões estabelecidos.

As regex são utilizadas por compiladores, pois são usadas para dar origem a algoritmos de autômatos finito determinísticos e autômatos finitos não determinísticos que são utilizados por analisadores léxicos

3.1 Conceitos básicos das expressões regulares

Uma expressão regular (r) sobre um conjunto de símbolos que formam um alfabeto (Σ) pode ser representada por meio de uma linguagem regular L(r) e definida com base nesses conceitos:

- Ø representa uma linguagem vazia;
- Ɛ representa uma linguagem que contém somente a palavra vazia {Ɛ};
- Se a e b são expressões regulares e representam as linguagens A e B respectivamente, então:
    - (a + b) é uma expressão regular para a operação de união, linguagem A ∪ B;
    - (ab) é uma expressão regular de concatenação, linguagem AB = {u ∈ A e v ∈ B};

- (a*) é uma expressão regular de concatenação sucessiva, linguagem A*.

A maioria dos caracteres são tratados como literais — eles casam somente com eles próprios (por exemplo, a casa "*a*"). As exceções são chamadas metacaracteres ou metas sequências, definidos abaixo:

Metacaracteres	Descrição
.	Corresponde a qualquer caractere único, exceto a um caractere de nova linha.
\	Marca o próximo caractere como um caractere especial ou como uma literal.
|	Representa alternância.
(...)	É usado para criar um agrupamento de expressões.
$	Corresponde ao final da entrada.
^	Corresponde ao início da entrada.
[ ]	Um conjunto de caracteres(lista). Corresponde a qualquer um dos caracteres circundados.
[^]	Um conjunto de caracteres negativo. Corresponde a qualquer caractere que não esteja delimitado.
[a-z]	Um intervalo de caracteres. Corresponde a qualquer caractere do intervalo especificado.
[^b-z]	Um intervalo negativo de caracteres. Corresponde a qualquer caractere que não esteja no intervalo especificado.
{n}	n é um número inteiro não negativo. Corresponde a exatamente **n** vezes.
{n,m}	As variáveis m e n são números inteiros não negativos. Corresponde ao caractere anterior no mínimo n e no máximo m vezes.

Quantificadores: são tipos de metacaracteres que definem um número permitido de repetições.

- ➔ **?** indica que há zero ou uma ocorrência do elemento precedente.
- ➔ ***** indica que há zero ou mais ocorrências do elemento precedente.
- ➔ + indica que há uma ou mais ocorrências do elemento precedente.
- ➔ **{n}** indica exatamente n ocorrências
- ➔ **{n,m}** indica no mínimo n ocorrências e no máximo m.

3.2 Expressões regulares - Análise Léxica

A análise léxica, primeira fase do compilador, é responsável pela leitura de caractere por caractere de um arquivo texto de uma linguagem de programação e os traduz em tokens, utilizando expressões regulares, que podem ser representados por autômatos finitos não determinísticos.

O AFND lê uma cadeia de símbolos de entrada; para cada símbolo, existe uma transição para um novo estado até que sejam lidos todos os símbolos de entrada. Caso seja lido um mesmo símbolo em um determinado estado, existem outras possibilidades de estado de destino, chegando a um estado de aceitação ou não.

4. Gramática livres de contexto

As gramáticas livres de contexto auxiliam na construção de gramáticas atuais das linguagens de programação, como Python, Java, C++, entre outras, e de compiladores. O compilador precisa validar se uma determinada cadeia de caracteres escrita em um arquivo pertence ou não a uma linguagem de programação. Podemos apresentar uma gramática como uma quádrupla:

G = (V, T, P, S)

V : conjunto de símbolos variáveis ou não-terminais

T : representa o alfabeto da linguagem G, contendo o conjunto de símbolos terminais.

P : conjunto de regras/ produções. P é um conjunto finito de pares em que cada par é reconhecido como uma regra/ produções.

S : denominado símbolo inicial da gramática.

É uma gramática formal onde todas as regras de produções são da forma:

$$A \rightarrow \alpha$$

“A ” é um símbolo não terminal, e α é uma cadeia de terminal e/ou não terminais (α pode ser vazia). Não importa quais símbolos existem na GLC, um único símbolo não terminal existente no lado esquerdo de uma regra pode sempre ser substituído pelo lado direito. E isso é o que distingue a GLC da gramática sensível ao contexto (GSC). A expressão “livre de contexto” significa que a variável A deriva α independentemente (livre) de uma análise dos símbolos antecessores ou sucessores de A (contexto).

A derivação consiste em verificar se um conjunto de sentenças pertence a uma linguagem de programação. A derivação substitui um símbolo não terminal por símbolos terminais, e o resultado dessas substituições dará a forma que a linguagem de programação deve ter.

Com base na gramática:

E → E + E

E → E * E

E → E - E

E → (E)

E → X

A expressão (x − x) * x, pode ser gerada pela seguinte regra de derivação:

$$E \Rightarrow E * E$$
$$\Rightarrow (E) * E$$
$$\Rightarrow (E - E) * E$$
$$\Rightarrow (X - E) * E$$
$$\Rightarrow (X - X) * E$$
$$\Rightarrow (X - X) * X$$

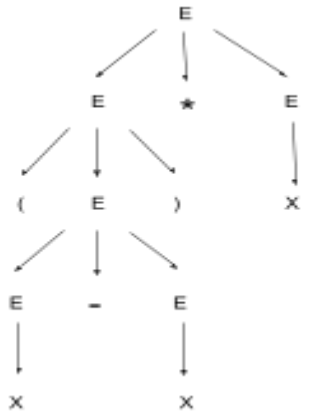
4.1 Árvore de derivação

A derivação também impõe, em certo sentido, uma estrutura hierárquica na cadeia que vai ser derivada. As derivações de um GLC podem ser representadas por meio de árvores de derivação.

Uma árvore de derivação consiste em:

- O símbolo inicial da gramática é a raiz da árvore.
- Os vértices interiores devem conter apenas símbolos não terminais (variáveis).
- Símbolos terminais ou símbolos vazios são chamados de vértices folha V→ ε.

A árvore de derivação da expressão $(x - x) * x$ usada no último exemplo é:



Essa árvore de derivação apresentada gerou uma derivação mais à direita, o que significa que as produções dos símbolos não terminais foram realizadas mais à direita. Caso as produções dos símbolos não terminais fossem realizadas mais à esquerda, a árvore de derivação geraria uma derivação mais à esquerda.

Uma gramática pode produzir a mesma cadeia de caracteres de diferentes formas, gerando uma ambiguidade. Em determinadas aplicações, a gramática não pode gerar ambiguidade, mas é um grande desafio garantir que a ambiguidade não aconteça. Uma gramática é considerada ambígua se existe uma mesma sentença com duas ou mais derivações à direita ou à esquerda.

5. Autômatos finitos

Um autômato finito tem um conjunto de estados. Há um estado inicial, que determina o ponto de partida para a realização do reconhecimento da sentença. À medida que caracteres da string de entrada são lidos, o controle da máquina passa de um estado a outro, segundo um conjunto de regras de transição especificadas para o autômato. Se após o último caráter o autômato encontrar-se em um dos estados denominados como finais, a string foi reconhecida, ou seja, pertence à linguagem. Caso contrário, a string não pertence à linguagem aceita pelo autômato. Formalmente, um autômato é descrito por uma quíntupla $M = (Q, \Sigma, \delta, q0, F)$:

- Q é o conjunto finito de estados, ;

- Σ é o alfabeto de entrada finito;
- δ, é o conjunto de transições;
- q0 é o estado inicial, onde q0 ϵ Q;
- F é o conjunto de estados finais, onde F ⊆ Q;

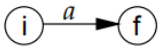
Autômatos finitos podem ser ou não determinísticos. Um autômato finito não determinístico pode ter transições que passam de um estado a outro sem a ocorrência de nenhum símbolo na entrada. Outra característica é a possibilidade de, a partir de um mesmo estado, ter mais de um estado destino possível para um mesmo símbolo de entrada.

Uma outra forma de representar um autômato, mais apropriada para fins de processamento automático, é através de tabelas de transição. Uma tabela de transição é uma matriz na qual as colunas representam os estados do autômato e as linhas os símbolos do alfabeto. Cada entrada na matriz indica qual o estado final de um transição a partir do estado indicado na coluna através do símbolo indicado na linha.

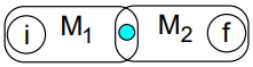
5.1 Algoritmo de Thompson

O Algoritmo de Thompson define uma sequência de passos para, a partir de uma expressão regular, obter um autômato finito não determinístico que reconhece sentenças da correspondente linguagem regular. O primeiro passo do procedimento é decompor a expressão regular que define as sentenças que deverão ser reconhecidas em termos de suas relações elementares.

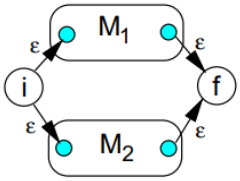
Uma vez que a expressão regular tenha sido estruturada em termos das relações elementares, é preciso construir um autômato para reconhecer cada uma das partes da expressão. Para cada relação elementar, a estrutura de um autômato correspondente é determinada. Para reconhecer um símbolo do alfabeto da linguagem , o autômato correspondente é composto simplesmente por um estado inicial que atinge um estado final através de uma transição pela ocorrência do símbolo a.



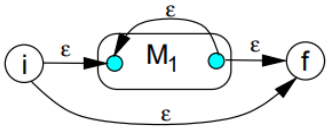
(a) Reconhecimento de um símbolo



(b) Concatenação



(c) Alternativa



(d) Repetição

6. Análise léxica

A análise léxica é o caminho para transformar um agrupamento de caracteres em um arranjo de tokens (strings com um significado designado). Um programa que realiza a análise léxica pode ser conhecido como lexer, tokenizer ou scanner.

A análise léxica é o estágio inicial no planejamento do compilador, que examina todo o código-fonte do desenvolvedor.. Um lexema é um agrupamento de caracteres distintos dentro de um conjunto de tokens. O analisador léxico é utilizado para realizar a varredura do programa em seu código-fonte, caractere por caractere, a fim de distingui-los na tabela de símbolos. Pode ocorrer erros - erro léxico.

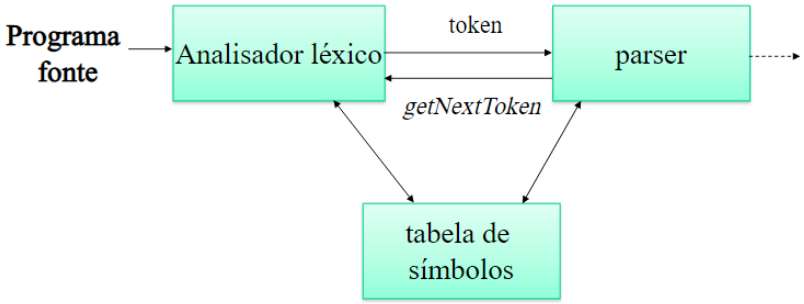
6.1 Conceitos básicos - Analisador Léxico

A análise léxica é a etapa inicial de um processo de compilação de código-fonte, seguindo os processos de:

- Extração e classificação de símbolos do código-fonte;
- Eliminação de caracteres em branco, espaço vazio e comentários;
- Recuperação de Erros;

O analisador léxico é um elemento do compilador cujas entradas são sequências de caracteres (texto do programa), que produzem como saída os chamados símbolos léxicos. É um intermediador entre o código (os textos de entrada) e o analisador sintático. O analisador léxico funciona de duas maneiras:

- **Primeiro estado da análise**: A primeira etapa lê a entrada de caracteres, um de cada vez, mudando o estado em que os caracteres se encontram.
- **Segundo estado da análise**: Nesta etapa são repassados os caracteres do léxico para produzir um valor. O tipo do léxico combinado com seu valor é o que adequadamente constitui um símbolo, que pode ser dado a um parser.



Para cada um dos lexemas, o analisador léxico irá produzir uma saída no padrão:

<nome _ token, valor _ atributo>

Os tokens constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc., e podem ser representados, internamente, através do próprio símbolo ou por um par ordenado, no qual o primeiro elemento indica a classe do símbolo, e o segundo, um índice para uma área onde o próprio símbolo foi armazenado (por exemplo, um identificador e sua entrada numa tabela de identificadores).

Além da identificação de tokens, o Analisador Léxico, em geral, inicia a construção da Tabela de Símbolos e envia mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem em questão. A saída do Analisador Léxico é uma cadeia de tokens que é passada para a próxima fase, a Análise Sintática. Em geral, o Analisador Léxico é implementado como uma sub-rotina que funciona sob o comando do Analisador Sintático.

6.2 Associação com linguagens regulares

As expressões regulares são um conjunto de descrições que podem ser usadas para representar certas linguagens (linguagens regulares). Geralmente, essas expressões fornecem uma descrição da linguagem de forma compacta e compreensível pelo ser humano.

Exemplo: Considerando a seguinte expressão numérica:

((20 + 7) * 5)

Para realizar a tokenização dessa expressão usasse conceitos da GLC:

Expr → Num | PE Expr Op Expr PD

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

PE → ‘(’

PD → ‘)’

Op → ‘ + ’ | ‘ * ’

Com base nessas regras o analisador realizará o scanner para montagem das expressões (tokens e atributos). A partir dessa composição teremos essa sequência similar :

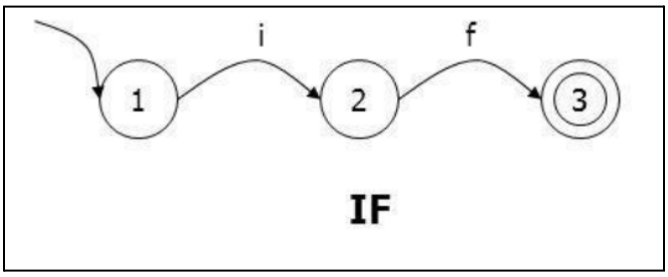
PE	PE	NUM	OP	NUM	PD	OP	NUM	PD
(	(	20	+	7	)	*	5	)

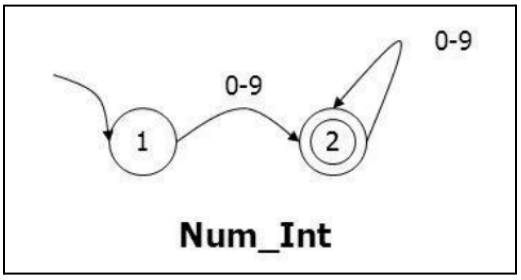
6.3 Analisador léxico e autômatos finitos

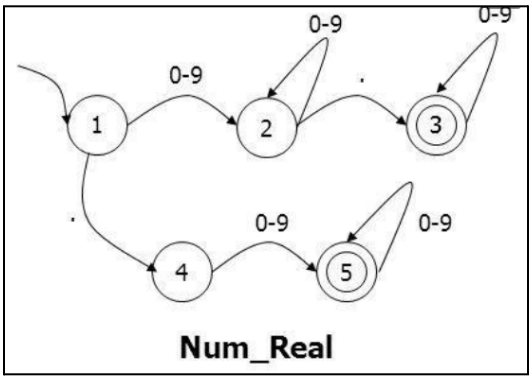
Os autômatos finitos são elementos baseados em estados, transições e símbolos de entrada. De forma geral, seu funcionamento baseia-se na leitura de caracteres (na chamada fita de entrada) e na associação com o estado atual e a transição estabelecida.

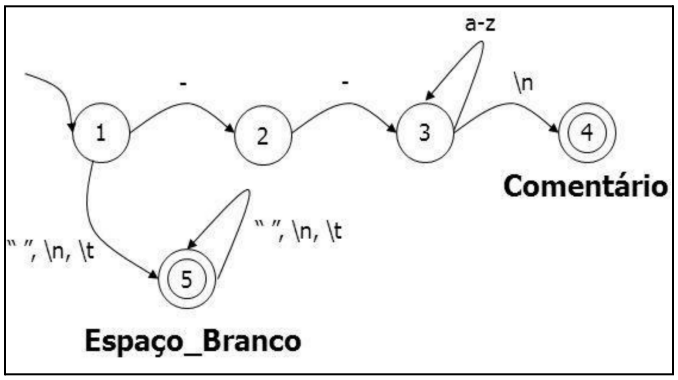
As expressões regulares são convenientes para especificar os tokens, mas se faz necessário uma representação para que seja entendida por um computador. Esse é um papel fundamental do autômato.

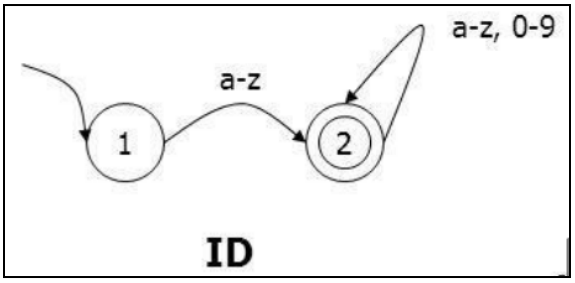
Um autômato finito determinístico (AFD) de análise léxica tem os estados finais rotulados com tipos de token. Por exemplo:

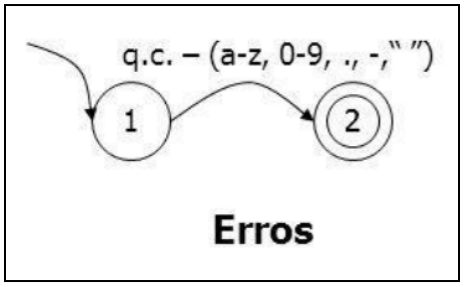












Nesses exemplos, o autômato analisa os caracteres e verifica se, após o processamento, estará no estado final. Dependendo da sequência do estado final, o autômato terá uma saída indicando, para aquela sequência de caracteres, o tipo associado ao token, gerando sua tabela e armazenando os valores e atributos.

7. Tabela de símbolos

É uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os nomes (identificadores de variáveis, de parâmetros, de funções, de procedimentos, etc) definidos no programa fonte. Essa tabela associa atributos (tipo, escopo, limites no caso de vetores e número de parâmetros no caso de funções) aos nomes que foram definidos pelo programador. A construção dessa tabela, em geral, se dá durante a análise léxica, quando os identificadores são reconhecidos.

É importante lembrar que as expressões regulares possuem um papel fundamental na construção e na análise léxica, pois apontam se determinada cadeia faz ou não parte de uma linguagem, e o analisador precisa obter todos os símbolos da expressão para que sejam realizados o processo de tokenização e a posterior análise em fases subsequentes

Quando um identificador é encontrado pela primeira vez, o analisador léxico armazena-o na tabela sem condições ainda de associar atributos a esse identificador. Geralmente essa tarefa é executada durante as fases de análise sintática e semântica.

Existem vários modos de se organizar e acessar as tabelas de símbolos sendo, os mais comuns:

- **Listas lineares** - é o mecanismo mais simples, mas seu desempenho é pobre quando o número de consultas é elevado.
- **Tabelas hash** - têm melhor desempenho, mas exigem mais memória e esforço de programação.
- **Árvores binárias -** gasta uma quantidade de tempo proporcional à altura da árvore.

As operações básicas para as estruturas de dados que implementam a tabela de símbolos são inserção, exclusão e busca.

Vejamos um exemplo da aplicação da tabela de símbolos em processo de análise léxica.

Parte de uma tabela de símbolos relacionados à linguagem Java:

&&	LOGIC _ AND
[|][|]	LOGIC _ OR
[+]	'+'
[+][+]	INC
/	‘ / ’
[.]	‘ . ’
>	‘ > ’
<	‘ < ’
while	WHILE
if	IF
for	FOR
else	ELSE
[a-zA-Z]	ID
[a-zA-Z _][a-zA-Z0-9 _]+	ID
[0-9]+	NUM
[0-9]+[.][0-9]+	NUM
[0-9]+[.]	NUM
[.][0-9]+	NUM
["]["]	STRING
["][^"\n]+["]	STRING

É importante considerar uma diferença: a tabela de símbolos, em sua essência, é diferente da tokenização de um código-fonte. A primeira é o conjunto de regras que relaciona tipos de expressões regulares e símbolos-padrão da linguagem. Esses símbolos e expressões regulares são transformados em máquinas de estados na qual a linguagem avalia e permite gerar uma saída correspondentes.

Referências

Barbosa, Cynthia da, S. et al. Compiladores. Disponível em: Minha Biblioteca, Grupo A, 2021.

https://integrada.minhabiblioteca.com.br/books/9786556902906 (Livro Ulife)

https://slideplayer.com.br/amp/325824/

https://www.dcce.ibilce.unesp.br/~aleardo/cursos/compila/cap02.pdf

Qual é a diferença entre Compilação e Interpretação? - Luby Software do seu jeito.

https://www.dca.fee.unicamp.br/~eleri/ea876/04/cap3.pdf

Expressões regulares, gramática regular e linguagens regulares – Acervo Lima

ibm.com/docs/pt-br/rational-clearquest/8.0.1?topic=tags-meta-characters-in-regular-expressions

https://www.dca.fee.unicamp.br/~eleri/ea876/04/cap5.pdf

https://pt.wikipedia.org/wiki/Gram%C3%A1tica_livre_de_contexto#Notas

Construção de compiladores/Análise léxica - Wikilivros.