UniRuy & Área 1 | Wyden

PROGRAMA DE CIÊNCIA DA COMPUTAÇÃO

TEORIA DE COMPILADORES

JOÃO MARCELO TAVARES SOUZA MATOS

Análise Léxica (Fases do Compilador)

Salvador - Bahia - Brasil

2022

JOÃO MARCELO TAVARES SOUZA MATOS

Análise Léxica (Fases do Compilador)

Trabalho Acadêmico elaborado junto ao programa de Engenharia UniRuy & Área 1 | Wyden, como requisito para obtenção de nota parcial da AV1 na disciplina Teoria de Compiladores no curso de Graduação em Ciência da Computação, que tem como objetivo consolidar os tópicos do plano de ensino da disciplina.

Orientador: Prof. MSc. Heleno Cardoso

Salvador - Bahia - Brasil
2022

da Tal, Aluno Fulano

Teoria de Compiladores: Resenha / Mapa Mental / Perguntas

– Aluno Fulano de Tal. Salvador, 2022.
18 f. : il.

Trabalho Acadêmico apresentado ao Curso de Ciência da Computação, UniRuy & Área 1 | Wyden, como requisito para obtenção de aprovação na disciplina Teoria de Compiladores.

Prof. MSc. Heleno Cardoso da S. Filho.

1. Resenha
2. Mapa Mental
3. Perguntas/Respostas (Mínimo de 03 – Máximo de 05)
4. Conclusão

I. da Silva Filho, Heleno Cardoso II. UniRuy & Área 1 | Wyden. III. Trabalho Acadêmico

CDD:XXX

TERMO DE APROVAÇÃO

JOÃO MARCELO TAVARES SOUZA MATOS

ANÁLISE LÉXICA (FASES DO COMPILADOR)

Trabalho Acadêmico aprovado como requisito para obtenção de nota parcial da AV1 na disciplina Teoria de Compiladores, UniRuy & Área 1 | Wyden, pela seguinte banca examinadora:

BANCA EXAMINADORA

Profº. MScº. Heleno Cardoso
Wyden

Salvador, 09 de Outubro de 2022

Dedico este trabalho acadêmico a todos que contribuíram direta ou indiretamente com minha formação acadêmica.

Agradecimentos

Primeiramente agradeço a Deus. Ele, sabe de todas as coisas, e através da sua infinita misericórdia, se fez presente em todos os momentos dessa trajetória, concedendo-me forças e saúde para continuar perseverante na minha caminhada.

E a todos aqueles que contribuíram direta ou indiretamente para a minha formação acadêmica.

"A educação tem raízes amargas, mas os seus frutos são doces".

Aristóteles.

Resumo

Sempre que pensamos em linguagem de programação e com ela funciona, muita das vezes esquecemos o que roda por trás desses codigos inteligentes, como o computador interpreta isso e como ele sabe que o que está escrito, está certo?

O compilador, junto com suas analises transformam o código fonte em linguagem de máquina, possibilitando interpretar, ler e executar o código escrito no programa fonte.

Em sua analise léxica, o analisador faz com que o computador transforme o código fonte, caracter por caracter em tokens, usando expressões regulares, automatos de estado finito e gramatica regular.

Palavras-chaves: Linguagem de programação, Compilador, Analise léxica.

Abstract

Whenever we think about a programming language and how it works, we often forget what runs behind these intelligent codes, how the computer interprets it and how does it know that what is written is right?

The compiler, together with its analysis, transform the source code into machine language, making it possible to interpret, read and execute the code written in the source program.

In its lexical analysis, the parser has the computer transform the source code character-by-character into tokens, using regular expressions, finite-state automates, and regular grammar.

Keywords: Programming language, Compiler, Lexical analysis.

Lista de figuras

Lista de abreviaturas e siglas

AFNs	- Automatos Finitos não deterministicos
ER	- Expressões regulares
AFD	- Automatos finitos deterministicos
AEF	- Automatos de estado finito

Sumário

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Análise Léxica

1.1 Introdução

Sendo um dos pilares do compilador, a analise Léxica é a primeira fase do processo de compilação, tem a responsabilidade de ler os caracteres do código e verificar se eles pertecem ao alfabeto da linguagem.

Todos os comentarios e espaços em branco são removidos e o processo resulta em simbolos léxicos ou tokens.

1.1.1 Perguntas e Respostas - Mínimo de 2 e Máximo de 5

1° Pergunta: Execute a análise léxica dos caracteres (int x = 5, y = 10, z= 20): identifique os tipos de tokens e lexemas.

2° Pergunta: O que é um erro Léxico?

1.2 Conclusão

Conforme os tópicos citados nesse artigo, a importância da analise léxicas e dos analisadores é imprescindível para toda a estrutura de um compilador, porém não é suficiente para determinar e identificar erros de código e por isso é necessario percorrer todas as etapas do compilador para que tenha a maior acertibilidade no encontro de erros.

2 Definição de símbolos léxicos válidos de uma linguagem de programação usando gramática regulares, expressões regulares ou autômatos finitos

2.1 O que é a analise Lexica

Para entendermos mais sobre a definição de símbolos léxicos, precisamos primeiro entender o que é a analise léxica e o seu papel dentro de um compilador. Portanto:

A analise léxica é a primeira fase de um compilador, ele lê o código fonte caractere por caractere buscando a identificação desse código a partir de símbolos léxicos ou tokens, sua função basica é identificar os tokens presentes no código fonte e passar essa informação para o parser. O procedimento consiste em ler os caracteres de entrada e agrupa-los em conjuntos que tenham sentido para os demais componentes do compilador(MARANGON, 2015)

Esses símbolos lexicos, também chamados de Tokens podem ser construidos a partir de:

- Cadeia e linguagens
- Linguagens
- Expressões regulares
- Autômatos de estado finito
- Gramática linear a esquerda

2.2 Cadeia e linguagens

Alfabeto é um conjunto finito não vazio de símbolos. São exemplos de símbolos letras,dígitos e sinais de pontuação. Exemplos de alfabeto:(MALAQUIAS, 2014)

- Alfabeto binário: {0,1}
- ASCII
- Unicode

Uma cadeia em um alfabeto é uma sequência finita de símbolos desse alfabeto. Por exemplo, 1000101 é uma cadeia no alfabeto binário.

O tamanho de uma cadeia s, denotado por |s|, é o número de símbolos em s. Por exemplo, $|1000101| = 7$.

A cadeia vazia, representada por ϵ, é a cadeia de tamanho zero. Um prefixo de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do final de s. Por exemplo, ama, amar, a e ϵ são prefixos da cadeia amarelo.

Um sufixo de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do início de s. Por exemplo, elo, amarelo, o e ϵ são sufixos da cadeia amarelo. Uma subcadeia de uma cadeia s é qualquer cadeia obtida removendo-se qualquer prefixo e qualquer sufixo de s. Por exemplo, are, mar, relo e ϵ são subcadeias da cadeia amarelo.

Um prefixo, um sufixo ou uma subcadeia de uma cadeia s é dito próprio se não for a cadeia vazia e ou a própria cadeia s.

Uma subsequência de uma cadeia s é qualquer cadeia formada pela exclusão de zero ou mais símbolos (não necessariamente consecutivos) de s. Por exemplo, aaeo é uma subsequência da cadeia amarelo.

A concatenação x y (também escrita como x · y) de duas cadeias x e y é a cadeia formada pelo acréscimo de y ao final de x. Por exemplo, ver de · amarelo = ver deamarelo. A cadeia vazia é o elemento neutro da concatenação: para qualquer cadeia s, ϵs = sϵ = s. A concatenação é associativa: para quaisquer cadeias r, s e t, (r s)t = r(st) = r st. Porém a concatenação não é comutativa.(MALAQUIAS, 2014)

2.3 Linguagens

Uma linguagem é um conjunto contável de cadeias de algum alfabeto.(JR, 2015) Exemplos de linguagens:

- o conjunto vazio, ø
- o conjunto unitário contendo apenas a cadeia vazia $\{\epsilon\}$
- o conjunto dos números binários de 3 dígitos, 000,001,010,011,100,101,110,111
- o conjunto de todos os programas Java sintaticamente bem formados
- o conjunto de todas as sentenças portuguesas gramaticalmente corretas

2.4 Expressões regulares

Expressão regular é uma forma compacta de denotar todos os possíveis strings que compõe uma determinada linguagem, usando para isso a chamada notação estrela (*)

de Kleene para representar a ocorrência de zero ou mais instâncias de um determinado símbolo ou conjunto de símbolos terminais. (JR, 2015)

O que a notação * de Kleene faz é apenas indicar que uma sequência de um determinado caracter "c"é expressa pelo símbolo "c*", representando uma sequência de pelo menos zero ocorrências do caracter "c". Assim, um número inteiro pode ser representado por "dd*", em que "d"representa um dos algarismos entre 0 e 9. Da mesma forma, um identificador em pascal ou modula-2 pode ser representado por "a(a|d)*", em que "a"representa uma letra entre a e z enquanto "d"é um algarismo entre 0 e 9. (JR, 2015)

2.5 Autômatos de estado finito

Como foi dito no capítulo anterior, um autômato é uma entidade capaz de reconhecer se uma string está ou não corretamente definida dentro de uma dada gramática. Pela classificação de Chomsky vimos também que uma gramática regular pode ser reconhecida pelo que chamamos de autômato de estado finito (AEF), o qual é descrito pelos cinco objetos discriminados a seguir: M = (\SIGMA, Q,\DELTA, q0, F) Em que

- \SIGMA é o alfabeto compreendido pelo autômato.
- Q é o conjunto de estados definidos para o autômato.
- \ DELTA é o seu conjunto de regras de transição.
- q0 é o seu estado inicial e
- F é o conjunto de estados finais para o autômato.

Um autômato pode ter suas regras de transição representadas de forma gráfica ou tabular. Na forma gráfica temos também a representação visual dos estados do AEF, o que facilita o entendimento de como o autômato passa de um estado a outro. Já a forma tabular se presta bem ao armazenamento das transições num computador, o que vai ser muito interessante durante o projeto de um analisador léxico. A figura a seguir ilustra a representação gráfica de um AEF, enquanto que a tabela depois da figura representa as transições do mesmo autômato em sua forma tabular. (JR, 2015)

Um autômato reconhece ou aceita uma string para uma dada linguagem se, ao final da string, parar num estado pertencente ao conjunto de estados finais F. Caso isso não ocorra diz-se que o autômato rejeita a string. Além disso, diz-se que um autômato aceita

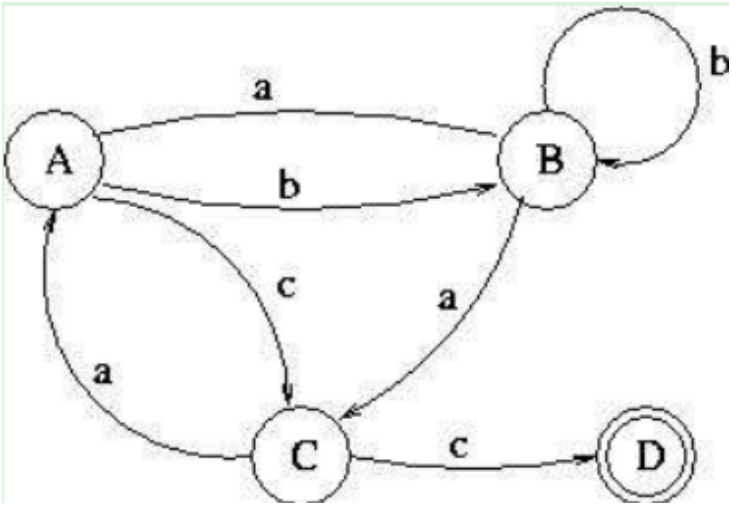

Figura 1 – Forma gráfica para representação de um AEF

uma linguagem se e somente se as strings definidas para aquela linguagem são aceitas pelo autômato, isto é, para qualquer string pertencente à linguagem o autômato irá parar num de seus estados finais. Partindo da definição da aceitação de uma linguagem podemos comparar dois autômatos sobre suas condições de equivalência, isomorfismo etc. Assim, diz-se que dois autômatos são EQUIVALENTES quando eles aceitam a mesma linguagem. Já quando, além de aceitarem a mesma linguagem, eles também tiverem os mesmos estados, diz-se que eles são ISOMORFOS. Por outro lado, quando um autômato tem o menor número de estados entre todos os seus equivalentes dizemos que ele é REDUZIDO. (JR, 2015)

Um AEF ainda pode ser classificado segundo o tipo de regras de transição que ele apresenta. Por este critério ele pode ser determinístico ou não-determinístico.

Um AEF não-determinístico é aquele em que pode ocorrer a transição vazia, que é aquela em que o AEF pode passar de um estado a outro sem que ocorra a entrada de um caractere da string.

Além disso, não AEF não-determinístico podem ocorrer indeterminações na ação do AEF, isto é, determinados estados podem ter mais do que uma transição possível para um dado caracter

O AEF determinístico é aquele em que transições vazias não ocorrem e em que não existam indeterminismos no momento de transitar de um estado a outro após a leitura de um caracter, ou seja, é todo aquele que não for não-determinístico. (JR, 2015)

2.5.1 Transformação de uma gramática regular em um AEF

Como um AEF é um dispositivo capaz de reconhecer as strings de uma linguagem e uma gramática é um meio formal de definir uma linguagem, devemos ter mecanismos para a partir de uma gramática especificar o AEF que reconheça tal linguagem. Isto é feito de forma bastante simples, seguindo-se as regras na figura abaixo:

GRAMÁTICA	AEF
alfabeto \SIGMA	alfabeto \SIGMA
Para todo símbolo não-terminal em N	cria-se um estado q pertencente a Q
	cria-se um estado final qf
S	q0
Para toda produção A -> xB	cria-se uma transição \delta(A,x) = B
Para toda produção A -> x	cria-se uma transição \delta(A,x) = qf

Figura 2 – Regras da transormação de uma gramática regular em um Automato Finito

2.5.2 Gramática linear a esquerda

Quando examinamos o processo de transformação de uma gramática num AEF construímos todas as regras baseando-nos numa gramática linear a direita, isto é, com produções do tipo A -> xB e A -> x. Entretanto, nem sempre temos uma gramática deste tipo em mãos. Uma forma de se obter tal gramática é usar expressões regulares como uma forma intermediária no processo de transformação de uma gramática linear a esquerda em outra a direita. Como já temos as regras de transformação entre ER's e gramáticas lineares a direita, resta-nos apenas determinar as regras de transformação entre gramática linear a esquerda e expressões regulares. (JR, 2015)

Produções	Expressão Regular
A -> Bx e B -> y	A -> yx
A -> Ax e A -> y	A -> yx*
A -> x e A -> y	A -> x | y

Figura 3 – Regra de produção da gramática linear

3 Construção de tabela de símbolos

As tabelas de símbolos são estruturas de dados usadas pelos compiladores para conter informações sobre as construções do programa fonte. As informações são coletadas de modo incremental pelas fases de análise e usadas pelas fases de síntese para gerar o código objeto. (BALDO, 2008)

Exemplo:

Na entrada: {int x; char y; bool y; x; y; x; y;z}

Na saída: { x: int; y: bool; } x:int; y: char;}

Suas principais operações são:

- **Inserir:** Usada para armazenar informações,tais como,tipo e escopo,na tabela de símbolos
- **Verificar:** Usada para recuperar informações um nome, guare esse nome é utilizado como um código.
- **Remover:** Usada para remover informações quando não mais utilizadas

(ERINALDO, 2008)

3.1 Uso da tabela de Símbolos

O papel de uma tabela de símbolos é passar informações de declarações para usos.

Uma ação semântica entra com informações sobre o identificador x na tabela de símbolos, quando a declaração de x é analisada.

Uma ação semântica associada a uma produção como factor $\rightarrow$ id recupera a informação sobre o identificador da tabela de símbolos (ERINALDO, 2008)

4 Implementação de um analisador léxico

Um analisador léxico pode ser implementado ou através do AEF que o representa ou através de sua gramática regular. Embora o resultado de ambas as técnicas deva ser o mesmo, estas formas têm diferenças fundamentais quanto à implementação. Se partirmos do AEF para a implementação do scanner, o trabalho de manipulação é maior do que se partirmos de sua gramática regular, que é um processo que pode ser feito de forma automática. A seguir descrevemos estas duas abordagens, apresentando também os problemas envolvidos na especificação do analisador léxico, tais como a implementação da tabela de símbolos, tamanho do alfabeto de entrada e tratamento da passagem de valores (tokens) ao analisador sintático, entre outros. (JR, 2015)

4.1 Implementação via Automato de Estado Finito

A implementação do scanner através de um AEF é bastante simples. Basicamente o que é feito é usar um par de comandos do tipo CASE (switch em C) para representar os estado e os caracteres sendo lidos pelo autômato.(JR, 2015)

Assim, por exemplo, um primeiro CASE faz sua decisão a partir do estado atual do autômato (que está guardado numa variável atualizada a cada transição), enquanto o segundo CASE tomará a sua decisão a partir do caractere que acaba de ser lido. A ordem em que estes dois cases são realizados pode ser alterada segundo a conveniência do processo de busca, isto é, caso seja mais conveniente primeiro selecionar pelo caractere de entrada e depois pelo estado, podemos fazê-lo assim sem maiores complicações. A decisão sobre qual ordem a ser adotada é deixada então a cargo do implementador. (JR, 2015)

Como pode ser visto, uma vez definido o AEF temos que a sua implementação é trivial se o mesmo puder ser expresso por um conjunto reduzido de estados e de caracteres de entrada. Entretanto isso não é o que ocorre num compilador, fazendo com que a implementação do AEF se torne trabalhosa devido ao grande número de estados e do tamanho do alfabeto existentes numa gramática regular real. A solução para este problema é o uso de geradores automáticos de scanners, tais como o lex ou o flex presentes no ambiente UNIX. (JR, 2015)

4.2 Implementação via gramática

O scanner pode ser implementado usando-se um gerador automático, tal como o lex. Para tanto, usamos gramáticas reescritas na forma de expressões regulares, que devidamente arranjadas segundo a sintaxe do lex podem gerar de forma automática o código do scanner. (JR, 2015) Esta abordagem tem enormes vantagens com relação à implementação direta a partir do autômato pois, como a gramática (e as expressões regulares que a representam) tem que ser definida de qualquer maneira, uma vez que é a partir dela que sabemos quais são os strings permitidos para a dada linguagem, torna-se desinteressante transformarmos a gramática num autômato que teria que ser implementado manualmente, se sabemos que existem geradores automáticos de scanners a partir das produções da própria gramática. Entretanto, em ambos os casos existem problemas que precisam ser resolvidos para que a implementação seja funcional, tanto quanto à sua correção como quanto à sua eficiência. A seção seguinte faz um tratamento destes problemas. (JR, 2015)

5 Resultado do processo de compilação

A figura abaixo ilustra a fase final de um compilador e seu resultado final do processo de compilação.

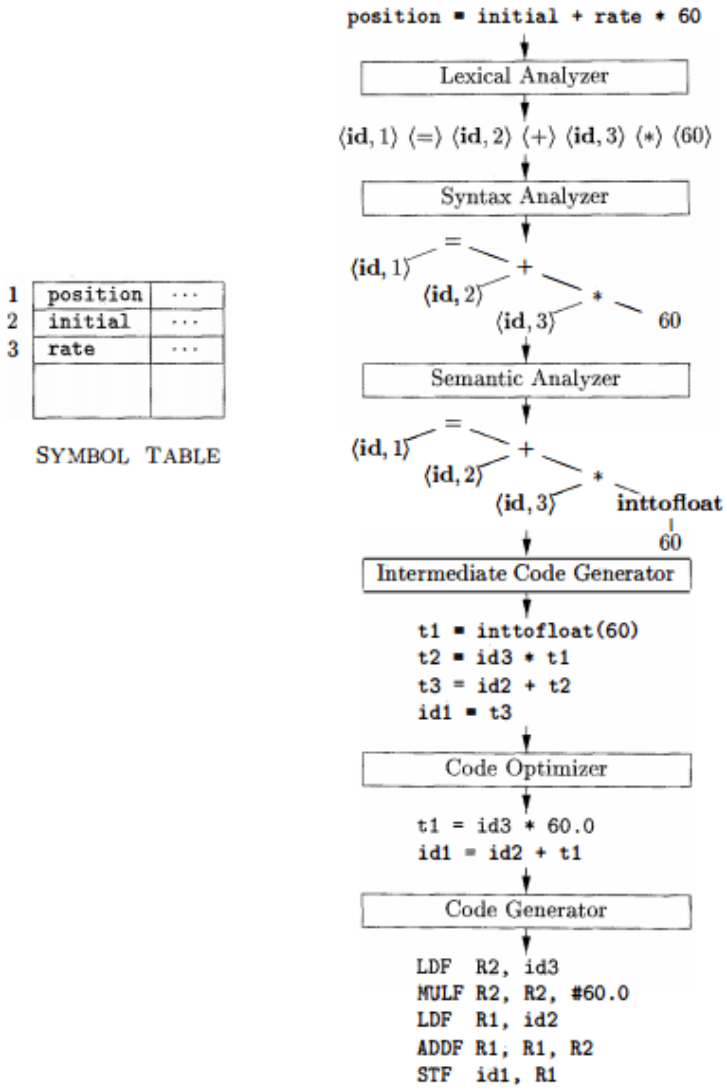

Figura 4 – Resultado do processo de compilação

Referências[1]

BALDO, F. *Tabela de Símbolos, Análise Semântica A Tabela de Símbolos*. 2008. Citado na página 19.

ERINALDO. *Tabelas de Símbolos*. 2008. Citado na página 19.

JR, A. M. *Capítulo 2 - Analisador Léxico*. 2015. Citado 6 vezes nas páginas 14, 15, 16, 17, 20 e 21.

MALAQUIAS, J. R. *Análise Léxica*. 2014. Citado 2 vezes nas páginas 13 e 14.

MARANGON, J. D. *Compiladores Para Humanos*. 2015. Citado na página 13.

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.