CAPÍTULO 1

Introdução

1 INTRODUÇÃO

“The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.”

Edsger Dijkstra

A humanidade desenvolveu a capacidade de entender o átomo graças a sua capacidade de pensar. A principal ferramenta do pensamento é a linguagem. Linguagens diferentes permitem formas diferentes de pensar, formas diferentes de resolver problemas.

Graças a criação de linguagens desenvolvidas matemática e formalmente, a humanidade conseguiu criar máquinas capazes de resolver qualquer problema imaginável em tempo hábil. Não seria nenhum exagero afirmar que todo o desenvolvimento que vimos durante a segunda metade do Século XX foi devido as linguagens de programação. Neste capítulo veremos um pouco da história da computação, os conceitos de compilação e interpretação e a matemática que suporta a criação de linguagens de programação.

Que não restem dúvidas a leitora, durante todo este livro, o termo máquina se refere aos computadores, de qualquer tamanho, gênero e grau. Se você chegou até este ponto já sabe que computadores são coleções de software e hardware que usamos para resolver problemas. A referência a este conjunto complexo de forma abstrata como máquina parece ser a forma mais simples de definir o universo com o qual nos preocupamos. Contudo, se olharmos com um pouco mais de detalhe poderemos traçar uma linha divisória entre o que é sólido e físico e o que abstrato e virtual. Neste livro, vamos tratar de máquinas abstratas feitas para entender a linguagem que os programadores usam para resolver problemas.

Programadores usam linguagens de programação para escrever o código que irá solucionar problemas os problemas que enfrentamos cotidianamente e permitir que a humanidade avance com a criação de novas tecnologias que trarão novos problemas para os programadores resolverem. Uma parte muito importante

da eficiência destas soluções reside na forma como este código será entendido e executado pela máquina. Neste livro, vamos explorar estes processos os processos de tradução que serão utilizados por duas outras máquinas virtuais, compiladores e interpretadores.

1.1 Um pouco de História

Poderíamos dizer que a história da computação começou quando o primeiro homem colocou algumas pedras no chão para saber com quantas ovelhas saiu para o pasto e com quantas voltou. A computação começou junto com a matemática e com a contabilidade.

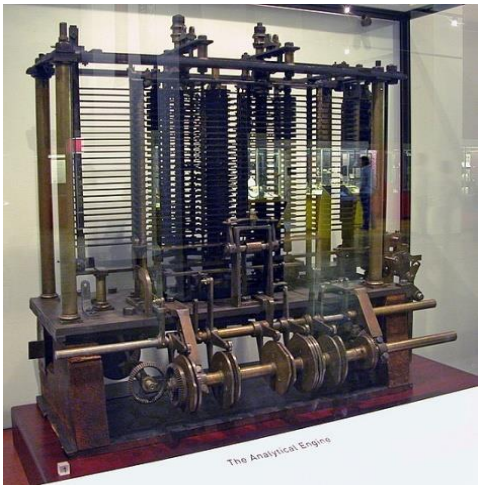


Figura 1 - Fonte: BARRAL, 2009 - A Máquina Analítica de Charles Babbage

A própria palavra computação tem origem no verbo latino computare que significa fazer cálculos. Mas, temos que marcar um ponto na história e muitos autores que se dedicam a este tema marcam o começo da computação na criação da Máquina Analítica por Charles Babbage.

Charles Babbage, um matemático do Século XIX teve seu trabalho reconhecido além da matemática 1a partir da década de 1930 quando pesquisadores dos dois lados do Oceano Atlântico estavam desenvolvendo máquinas eletrônicas para cálculos. Ainda assim, a primeira grande bibliografia de Charles Babbage só seria publicada em 1982. Do ponto de vista da computação dois dos seus conceitos são considerados como a base da computação: a máquina analítica e a máquina diferencial. No desenvolvimento da máquina analítica, mostrada da Figura 1, Babbage foi parcialmente ajudado por Ada Lovelace, dama da sociedade britânica e filha do poeta George Gordon Byron – Lord Byron).

Ada Lovelace traduziu para o Inglês um manuscrito escrito em francês por um matemático italiano (Luigi Manabrea) e, atendendo uma sugestão do próprio Babbage, fez anotações pessoais na sua tradução sobre como usar a máquina analítica. Todo o trabalho de Ada e Babbage seria apenas uma nota de rodapé na história não fossem as instruções que Ada escreveu sobre como seria possível utilizar a máquina analítica para calcular os Números de Bernoulli. Estas notas são consideradas como sendo a expressão do primeiro programa. A tradução de Ada e suas notas estão disponíveis on line e podem ser encontradas no Fourmilab MENABREA, 1842.

O conceito de linguagem de programação começou a ser construído a partir de 1946 com a criação de Plankalkül por Konrad Suze, um engenheiro alemão que criou o primeiro computador com relés e sua própria linguagem de programação, Plankalkül. Esta linguagem, ainda que embrionária era imperativa e altamente tipada o trabalho de Suze não foi publicado até a década de 1970 por causa das restrições impostas a Alemanha pela Segunda Guerra Mundial. Em 1949 John McCauly desenvolve a linguagem chamada de Short Code, a base para o desenvolvimento UNIVAC a primeira máquina que pode ser chamada de computador eletrônico. No caso da linguagem Short Code, a compilação era feita a mão, uma instrução era posta após a outra, formando uma lista de instruções na ordem em que deveriam ser executadas e, talvez tenha sido neste momento que a palavra compilador começou a ter o sentido que tem hoje.

Podemos correr o risco de afirmar que o compilador, foi criado em 1951 pela equipe liderada por Grace Hopper enquanto trabalhava na Remington Rand, e chamado de **A-0**. Em 1955 a linguagem utilizada por este compilador foi liberada para uso com o nome **Math-Matic**. Este ambiente, compilador e linguagem de programação, foram utilizados em máquina comercias: UNIVAC e UNIVAC II. Poucos anos depois, em 1957 **John Backus,** trabalhando na IBM, criou o Fortran (**FOR**mula **TRAN**slation), rapidamente substituída pelo **Fortran II** em 1958.

O Fortran II permitia o uso de sub-rotinas, na época uma inovação revolucionária, dando origem as linguagens de programação modulares. Contudo,

como Fortran II estava umbilicalmente ligado a IBM, apesar do seu sucesso, a linguagem foi evitada por organismos técnicos e científicos com o objetivo de manter todo o processo de criação de programas de computadores livres das influências e restrições comerciais que poderiam ser impostas pela IBM e garantir a portabilidade dos programas entre os diversos fabricantes de hardware na esperança de garantir a interoperabilidade entre programas diferentes, programadores diferentes e empresas diferentes.

O ano de 1957, o ano do lançamento do Fortran foi também o ano em que Noan Chomsky publicou o livro Syntactic Structures[1] sintetizando anos de seu trabalho no estudo da complexidade das gramáticas. Entre os conceitos apresentados por Chomsky estava o conceito de gramáticas livres de contexto, que se mostrou extremamente útil para a descrição da sintaxe de linguagens formais, principalmente para as linguagens de programação. Sob a influência do trabalho de Chomsky, as especificações do ALGOL foram publicadas por um comitê, auto proclamado como mundial, de desenvolvimento incluindo engenheiros dos EUA representados pela ACM (*Association of Computing Machinery*[2]) e da Europa, representados pelo GAMM (*Gesellschaft für Angewandte Mathematik und Mechanik*[3]) no Instituto Federal de Tecnologia em Zurique. Com três sintaxes diferentes: referência, publicação e implementação, o ALGOL podia ser utilizada com sistemas de numeração diferentes e em idiomas diferentes.

O ALGOL tem um lugar especial na história da computação. John Backus, criou uma linguagem para descrição de uma linguagem de programação, a Backus Normal Form. Anos depois, em 1960 Peter Naur, reviu e expandiu a linguagem criada por Backus e, atendendo uma sugestão de Donald Knuth, mudou o seu nome para Backus-Naur Form. Esta versão do ALGOL, o ALGOL-60, marca a

[1] Em tradução livre: estruturas sintáticas

[2] ACM – Association for Computing Machinery. Em tradução livre: Associação para a Máquinas Computacionais.

[3] GAMM - Gesellschaft für Angewandte Mathematik und Mechanik. Em tradução livre: Sociedade para matemática e mecânica aplicadas.

primeira vez em que uma notação formal, **Backus-Naur Form**¸ foi usada para definir uma linguagem e, graças a isso, foi também responsável por estabelecer as bases para a criação do compilador moderno NAUR, BACKUS, *et al.*, 1963. O impacto do ALGOL-60 foi gigantesco no mundo das linguagens de programação. Contudo, comercialmente, a linguagem foi um fracasso. Talvez por motivos comerciais, a maior parte do hardware da época era produzido pela IBM, talvez por causa da popularidade do FORTRAN, talvez por causa da complexidade da linguagem.

Em 1958, enquanto o Fortran II decolava entre os pesquisadores científicos, no MIT John McCarthy, fundador da pesquisa em inteligência artificial, inicia a pesquisa no desenvolvimento da linguagem LISP, a linguagem que dá origem a praticamente todas as linguagens funcionais. Nova revolução tecnológica e em 1959 surge o Lisp 1.5 e cria definitivamente, os paradigmas do que chamamos hoje de linguagem funcional.

Figura 2 – Admiral Grace Hopper

Em 1959, foi publicado o COBOL (*Common Business Oriented Language*[4]), por um comitê de pesquisadores liderados por Grace Hopper. A mesma Grace Hopper que alguns anos antes havia criado o primeiro compilador.

No dia 1º de Maio de 1964, aproximadamente as 4:00h foi executado o primeiro programa em Basic, fruto do trabalho de John G. Kemeny e Thomas E. Kurtz, no Dartmouth College.

A década de 1970 é profícua na criação de linguagens de programação: Mumps, Forth, Smaltalk e Scheme (um dialeto do Lisp) e a linguagem C. O C, criado

[4] Em tradução livre: Linguagem Comum Oriented Language

em 1972 por Dennis Ritchie nos laboratórios da Bell para ser utilizada no Sistema Operacional Unix desenvolvida sobre outra linguagem a B, que fora desenvolvida por Ken Thompson para permitir o porte do Unix para o computador PDP-11. O B era muito lento e as modificações criadas por Ritchie permitiram que uma versão do C e seu compilador fossem incluídas na versão 2 do Unix. O C evoluiu rapidamente e parte desta história pode ser vista na Tabela 1.

LINGUAGEM	ANO	DESENVOLVIDA POR
Algol	1960	International Group (Backus-Naur)
BCPL	1967	Martin Richard
B	1970	Ken Thompson
C	1972	Dennis Ritchie
K & R C	1978	Brian Kernighan & Dennis Ritchie
ANSI C	1989	Comitê ANSI
ANSI/ISO C	1990	Comitê ISO
C99	1999	Comitê de Padronização da ISO
C18	2018	Comitê de Padronização da ISO

Tabela 1 - História das Versões da linguagem C

No mesmo ano da criação do C, Robin Milner, trabalhando com seus colegas da Universidade de Edinburgh na aplicação da lógica em funções computáveis apresentou a linguagem ML (Meta-Language) com o objetivo de criar um mecanismo interativo para prova automática de teoremas. Esta linguagem influenciou a criação das linguagens C++, Haskell e hoje seus dialetos mais importantes são o SML (Standard ML) e o OCaml.

Em 1978 surge a AWK, uma linguagem para o processamento de textos nomeada em homenagem aos seus autores: Aho, Weinberger e Kernighan. Este é

o mesmo Aho que escreveu Compilers: Principles, Techniques e Tools[5] junto com Monica S. Lam, Ravi Sethi e Jeffrey D. Ullman. Este livro ainda é, quase cinquenta anos depois, o livro de referência para a ciência da compilação.

Em 1984, em plena revolução dos microcomputadores, tanto a Microsoft quanto a Digital Research lançam versões do C para sistemas operacionais de microcomputadores. O DOS, pela Microsoft e o DR-DOS, pela Digital Research. Este lançamento é o pano de fundo da guerra do DOS. A Microsoft saiu vencedora principalmente por que o Windows 3.11 não era 100% compatível com o DR-DOS e, se tiver tempo, esta é uma história com cheia de espionagem e golpes mercadológicos muito interessantes, mas que fogem do escopo deste livro. É também em 1984 que é publicada a primeira versão do C++ por Bjarne Stroustrup, a partir da tese desenvolvida em seu doutorado no ano de 1979, dando um impulso significativo as linguagens de programação orientadas a objeto.

O próximo salto na história das linguagens de programação acontece em 1991. O Java criado por James Gosling da Sun Microsystems que, além de adotar todos os paradigmas da programação orientada a objetos inova na criação de novas técnicas de compilação e interpretação.

E chegamos a era da web, Rasmus Lerdorf com o PHP (1994), Guido Van e Brendan Eich com o Javascript (1995) movimentam a Internet permitindo a expansão da web e a criação de tecnologias novas. A web acaba fortalecendo o Python de Guido Van Rossum (1989) como a linguagem mais utilizada em ciência de dados. Logo depois disso, surgem as linguagens R e Julia, voltadas para o processamento estatístico de grandes quantidades de dados e, na segunda década do Século XXI, chegamos as linguagens voltadas para a computação quântica: CIRQ, do Google; Q# da Microsoft e Q da IBM.

Esta nossa viagem pela história das linguagens de programação está longe de ser completa. Na verdade, é uma lista até modesta. Foram citadas apenas as

[5] Em tradução livre: Compiladores, princípios, técnicas e ferramentas.

linguagens de programação que tiveram um grande impacto nos processos de compilação e interpretação. Afinal, este é um livro sobre compiladores e interpretadores. Somos obrigados a falar de linguagens de programação, mas apenas daquelas que interessam.

1.2 Linguagem de Máquina

Máquinas. Usamos máquinas para resolver problemas desde que o primeiro homem utilizou um graveto para pegar uma fruta ou matar um animal. Estas máquinas evoluíram do graveto ao computador quântico, passando pela alavanca, o motor a vapor e o avião em pouco mais de 100.000 anos. Apesar de tanta evolução, apenas no Século XX conseguimos nos comunicar com as máquinas e conseguimos dizer o que queremos que elas façam. Ainda que para isso tenhamos sido forçados a desenvolver dois tipos de linguagens uma, formal e regular para que os homens possam entender e outra, também formal e regular para que a máquina possa entender o que queremos que ela faça.

Nestes tempos de anglicismos tecnológicos a máquina é o hardware. A linguagem com que nos comunicamos com as máquinas é o software. Para os nossos objetivos, tanto a palavra hardware quanto a palavra software são por demais imprecisas para uso neste livro.

O hardware diz respeito a toda a máquina, e nós interessa apenas o conjunto formado pela unidade central de processamento (CPU), as memórias e os dispositivos de entrada e saída. Considere este generalíssimo um mal necessário. É com este conjunto de dispositivos que podemos descrever o modelo de computação desenvolvido no começo do Século XX por John Von Neumann que, com poucas modificações até o começo do Século XXI é a base de todos os computadores que regem nossa vida moderna.

O conjunto formado pela CPU, memória e dispositivos de entrada e saída é responsável pela execução direta de todas as instruções que desejamos que a máquina execute. Este conjunto de dispositivos entende apenas os símbolos 0 e 1

(zero e um). Esta é uma linguagem, baseada apenas em um conjunto de símbolos, ou alfabeto, definido por: $\Sigma_b = \{0,1\}$. Com o alfabeto Σ_b podemos especificar um conjunto de cadeias (*string*s[6]) finito, por sua vez o conjunto destas *string*s define a linguagem que a única linguagem que máquina entende. A essa linguagem damos o nome de linguagem de máquina, ou código de máquina.

Se excluirmos a computação quântica e, um ou outro, computador analógico que ainda exista por aí, podemos dizer que toda a computação realizada pela humanidade e suas máquinas é devida a uma linguagem composta de *string*s de zeros e uns. Cada uma destas *string*s contém as informações necessárias para a execução, ou para o preparo da execução, de alguma instrução que desejamos ver executada.

Não é raro que, em livros de hardware, ou mesmo livros de programação, seja usada o termo palavra, ou word em inglês para se referir a estas *string*s. Felizmente este é um livro de compiladores e interpretadores e podemos ser um pouco mais formais. Como são conjuntos de símbolos chamaremos de *string*s, em inglês mesmo, para evitar constrangimentos. Destaque-se que estas *string*s e a linguagem que elas formam, são características únicas e exclusivas de um hardware específico composto de CPU, memória e dispositivos de entrada e saída. Ao conjunto de *string*s que permitem que a máquina execute nossos comandos, daremos o nome de conjunto de instruções, ou em inglês, Instruction set.

A Intel[7] chama este conjunto de instruções de x86_64 Instruction Set[8] e específica exatamente qual a combinação de zeros e uns devem ser utilizadas para que qualquer instrução seja executada. Não podemos esquecer, a essa altura, que existem instruções que precisam de um conjunto de *string*s para que sejam entendidas e executadas. Esta linguagem x86_64 só pode ser entendida por

---

[6] Em tradução livre: cadeias. Mas, neste livro, vamos ignorar isso e usar *string*s.
[7] Intel; http:\\www.intel.com.br
[8] Em tradução livre: conjunto de instruções x86_64 (cada *string* da linguagem tem 64 símbolos)

máquinas definidas de acordo com as especificações da Intel. Não será possível pegar um conjunto de *string*s desta linguagem e executar, por exemplo, em um computador da arquitetura ARM, ou RISC. Esta limitação é importante e define o mercado.

A linguagem de máquina é a mais formal e específica linguagem envolvida nos processos de computação. Isso a torna praticamente impossível de uso por seres humanos. Existe aqui, alguns detalhes históricos interessantes para sua reflexão. No começo, as máquinas eram programadas apenas com uma sequência de *string*s do alfabeto Σ_b, depois, os engenheiros perceberam que estas *string*s podiam ser representadas de forma mais simples por *string*s de um alfabeto composto por números do sistema hexadecimal representado pelo conjunto $\Sigma_h = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$. O que não passou de uma simples representação de dígitos em um sistema de numeração por dígitos em outro sistema de numeração. Evoluímos um pouco mais e criamos uma linguagem com mnemônicos, para representar estas *string*s e as funções que eles deveriam executar. E assim, surgiu a família de linguagens chamada de Assembly.

A primeira linguagem Assembly foi escrita em 1955 por Stan Poley para o computador IBM 650 e a chamou de Symbolic Optimal Assembly Program[9] ou SOAP. A partir deste ponto, os programadores não seriam mais forçados a memorizar sequências numéricas, nem em Σ_b nem em Σ_h apenas comandos cuja sintaxe lembrava a função que deveria ser executada. Por exemplo a função **mov rbp, rsp** para mover o registrador **rsp** no registrador **rbp**, em lugar do conjunto de símbolos do alfabeto Σ_b.

A família Assembly é constituída por um conjunto de linguagens formais, definidas para uso humano, são as linguagens de programação que estão mais próximas da linguagem de máquina e por isso são conhecidas como linguagens de baixo nível. São linguagens formais que, assim como as linguagens de máquina,

[9] Em tradução livre: programa de montagem simbólica otimizado.

são especificas de uma determinada arquitetura o que dificulta o uso de códigos desenvolvidos para uma arquitetura em outra.

A especificidade do código de máquina, específico de uma determinada arquitetura permite que o código escrito para representar o algoritmo possa fazer uso de todas as características de uma determinada arquitetura, permitindo mais desempenho em termos de velocidade e recursos.

Em outras palavras, a linguagem Assembly desenvolvida para a arquitetura x86_64 será diferente de uma linguagem Assembly desenvolvida para a arquitetura RISC-V. O código escrito para uma arquitetura não rodará na outra. E isto, acredite, é uma coisa boa.

As linguagens da Família Assembly também evoluíram ao longo do tempo, hoje, não é difícil encontrar ambientes de desenvolvimento para linguagens Assembly cujo código pode ser traduzido em código de máquina para mais de uma arquitetura, chamamos estes ambientes de forma genérica de *Cross-Assembler*.

A Figura 3 mostra o código Assembly para o cálculo do quadrado de um número inteiro em uma arquitetura ARM, a esquerda, e em uma arquitetura x86_64, a direita.

```
1   |int square(int)| PROC
2           push        {r0,r1}
3           push        {r11,lr}
4           mov         r11,sp
5           sub         sp,sp,#8
6   |$M4|
7           ldr         r2,[sp,#0x10]
8           ldr         r3,[sp,#0x10]
9           mul         r3,r2,r3
10          str         r3,[sp]
11          ldr         r0,[sp]
12  |$M3|
13          add         sp,sp,#8
14          pop         {r11}
15          ldr         pc,[sp],#0xC
16  |$M5|
17
18          ENDP  ; |int square(int)|, square
```

```
1   square(int):
2           push    rbp
3           mov     rbp, rsp
4           mov     DWORD PTR [rbp-4], edi
5           mov     eax, DWORD PTR [rbp-4]
6           imul    eax, eax
7           pop     rbp
8           ret
```

Figura 3 - Exemplos de Assembly para o cálculo do quadrado de um

O Assembly não é linguagem de máquina. Só para enfatizar, vou repetir: o Assembly não é linguagem de máquina. Sequer a linguagem baseada no alfabeto

Σ_hé uma linguagem de máquina. Isso quer dizer que tanto o Assembly, quanto qualquer outra forma de memorização que utilizamos terá que ser traduzida em *string*s do alfabeto $\Sigma_b$antes que suas instruções possam ser executadas. No caso do Assembly esta tradução é feita por um programa especial chamado de *Assembler* que, em última análise, é um compilador, simples, mas compilador.

1.3 Entendendo o *Assembler*

O *Assembler* traduz uma linguagem de baixo nível, formal e desenhada para uso humano em código de máquina. Ainda que a linguagem Assembly, objeto o *Assembler*, seja muito simples, pouco mais que uma linguagem puramente simbólica, e muito próxima do nível da linguagem de máquina, ela tem estrutura léxica, sintática e semântica como qualquer outra linguagem de programação. O *Assembler* é um programa que lê um arquivo de textos e traduz este arquivo de textos em um conjunto de instruções em formato binário que será executado por uma máquina específica.

Cada instrução em Assembly corresponde a uma, e somente uma, instrução em código de máquina. Implicando que cada linha de código escrito em Assembly, corresponde a uma instrução em código de máquina. E aí está a simplicidade e a beleza desta ideia e a complexidade que ela esconde. Compiladores e interpretadores são estruturas software, ou hardware, que fazem a tradução de uma linguagem formal e regular, criada para uso humano em uma linguagem de máquina. Se você está pensando que um *Assembler* é uma espécie de compilador, não está muito longe da verdade. Lá no passado, quando as coisas estavam começando, dizer uma sandice destas poderia levar um profissional da área para a fila do desemprego. Tal é a diferença de complexidade entre um *Assembler* e um compilador. No começo do Século XXI esta fronteira já não está tão definida assim.

Cada arquitetura de hardware, x86_64, ARM, RISC-V tem as suas próprias características de forma que um conjunto de zeros e uns pode significar coisas completamente diferente em máquinas diferentes. Isto faz com que o *Assembler*

seja um tradutor altamente especializado. Que, além de conhecer, todos os comandos que a arquitetura aceita, e seu significado, também precisa conhecer a arquitetura propriamente dita: estrutura de registradores, função destes registradores, estrutura de memória, cache ou pilhas. Uma instrução Assembly, terá a forma: $mnemonico\ operadore(s);$ onde $mnemonico$ representa uma instrução de máquina e $operadore(s)$ os operadores que são necessários para a execução desta instrução. E isso era tudo que tínhamos no começo deste processo.

À medida que a tecnologia evolui as linguagens Assembly incorporaram algumas facilidades. Um exemplo interessante são as referências. Comandos que podemos utilizar para integrar fragmentos de código em lugares diferentes do mesmo código Assembly para, além de facilitar a redação do algoritmo, diminuir os erros causados pela digitação. Isso forçou a criação de algumas famílias diferentes de *Assembler*s:

- ***One-pass Assembler***: um *Assembler* que lê o texto contendo o código fonte e transforma em código de máquina varrendo este texto, do começo ao fim, uma única vez. Este é um *Assembler* rápido e simples, em geral os *Assembly* definidos para este *Assembler* são simples, não usam referências em código *Assembly* que não sejam específicos da arquitetura destino.

- ***Two-pass Assembler***: como no nome diz, lê o arquivo contendo o código fonte duas vezes. Isso permite o uso de instruções específicas para a criação de código, como as referências a fragmentos de código específico para melhorar o processo de redação de código fonte e diminuir o erro. Trata-se de um método um pouco mais lento que mantém a ligação explícita entre *Assembly*, *Assembler* e arquitetura da máquina.

- ***Macro-Assembler***: o *Assembler* que permite o uso de macros. Uma macro, neste contexto é equivalente a uma sub-rotina que, por sua vez, representa um fragmento de código que é escrito uma vez e pode ser reutilizado muitas vezes. A diferença real entre uma macro e uma sub-rotina está na operacionalização. A sub-rotina é armazenada em memória uma única vez, enquanto as macros, serão repetidas todas as

vezes que forem necessárias. O conceito e a implementação de macros em *Assemblers* pode ser traçado até a década de 1950. Um dos *Assemblers* comerciais de maior sucesso nesta categoria é o Microsoft Macro *Assembler*[10].

- ***Cross-Assembler***: um *Assembler* capaz de rodar em uma arquitetura enquanto gera código de máquina que será utilizado em outra arquitetura. Os *Assemblies* definidos para este tipo de *Assembler*, mais tarde, seriam classificados como linguagens de mais alto nível entre os *Assembly*. Isso, para garantir não só que você possa escrever código em arquiteturas diferentes, por exemplo, usar sua máquina Linux *x86_64* para gerar código de máquina RISC-V, mas também para usar o mesmo código fonte gerado para gerar código de máquina para arquiteturas diferentes.

Seria possível estender esta classificação por mais duas ou três páginas, falando de algumas dezenas de *Assemblers* diferentes. Como este não é o foco deste livro, devemos entender o trabalho que deve ser realizado por um *Assembler* da classe Two-pass *Assembler*:

O *Assembler* irá ler o texto com o código em Assembly duas vezes, neste cenário, o código contém macros e referências nominais, em inglês label[11]. Estas referências nominais são criadas quando o programador utiliza uma label para nomear um determinado bloco de código, visando o uso em momentos diferentes do fluxo de execução. Uma label, é um nome dado a um determinado endereço no código em Assembly que, quando convertido em código de máquina, será chamado e executado mais de uma vez, em momentos diferentes.

Durante a primeira leitura first-pass o *Assembler* não tem como saber o endereço da referência criada pela label, até que encontre definição do bloco de código que foi identificado por esta label. Na primeira leitura, ou primeira passagem

[10] O Microsoft Macro Assembler pode ser instalado junto com o Visual Studio, de forma gratuita MICROSOFT, 2010.

[11] Em tradução livre: etiqueta.

ou, ainda, em inglês first-pass, o *Assembler* irá:

- **Verificar a sintaxe de cada instrução**, parar e emitir uma mensagem de erro todas as vezes que encontrar um erro na formação de uma instrução;
- **Determinar se o tamanho da instrução** e o dado que ela está manipulando para reservar espaço de memória, ou registrador, para seu uso;
- **Determinar os endereços** de *labels* ou de seções de código;
- **Criar uma tabela de símbolos** contendo as definições de cada *label* e seu endereço em memória;

Na segunda leitura, ou *second pass*, tomando todo o cuidado para não alterar os endereços das labels gerados na primeira leitura, o *Assembler* irá:

- Parar e emitir uma mensagem de erro se encontrar uma referência indefinida em alguma instrução ou operador;
- Codificar as instruções em código de máquina utilizado os endereços calculados na primeira leitura;
- Recolocar instruções em endereços diferentes, sempre que possível trazendo operações para os registradores;
- Gerar as informações necessárias para eventuais *debugs*[12];
- Finalmente gerar o código objeto.

Este processo de leitura e interpretação de código, pode ser classificado de várias formas, simples não é uma delas. Ao longo deste livro você verá como as relações estreitas que existem entre o processo de interpretação dos *Assembler*s

[12] Em tradução livre: correção de erro. Esta é uma outra referência a Grace Hopper e, se me permitir a audácia, sugiro que pesquise na internet: Grace Hopper e Bug. Esta é uma história que vale seu tempo.

e os interpretadores e compiladores que usamos no começo do Século XXI.

1.4 As linguagens de programação

As linguagens de programação são linguagens formais, desenvolvidas para uso humano que foram criadas para explicitar as funcionalidade mínimas necessárias para tornar possível que um pobre ser humano possa escrever um algoritmo, e entender aquilo que está escrevendo, e preservar a esperança que outros seres humanos também serão capazes de entender o que ele escreveu. Para tanto elas precisam atender um conjunto bem específico de capacidades. Na pior das hipóteses uma linguagem de programação de ter:

- **Inteligibilidade:** o entendimento de uma linguagem de programação é muito maior que o entendimento de uma linguagem de máquina. Este é o motivo para que estas linguagens tenham ortografia, gramática e sintaxe próximas das linguagens informais, a língua escrita e falada naturalmente.
- **Portabilidade:** a linguagem de programação é independente da máquina. Um código escrito em C poderá ser compilado para rodar na arquitetura de hardware que se desejar e, na maioria das vezes, sem qualquer tipo de alteração. Isso não é possível com linguagens de máquina.
- **Redigibilidade:** a linguagem de programação dever ser simples de forma que a redação do algoritmo seja simples e, se possível, instintiva. Quanto mais próximo das linguagens naturais uma linguagem de programação for, maior será a sua redigibilidade.
- **Ortogonalidade:** a linguagem de programação deve permitir que seja possível combinar seus artefatos e conceitos básicos sem que se produzam erros provenientes desta combinação.
- **Reusabilidade:** a capacidade de reutilizar o código fonte escrito uma vez

ou mais vezes. Talvez esta seja a característica mais desejável. Imagine se, todas as vezes que fosse escrever um programa tivesse que escrever todo o código necessário para, por exemplo: elevar um número ao quadrado.

- **Modificabilidade:** trata-se da característica da linguagem de programação que permite que o código seja modificado para, por exemplo: a inclusão de novas funcionalidades. Quanto maior for a característica de modificabilidade de uma linguagem de programação mais simples será fazer manutenção no código que você escrever.
- **Universalidade:** toda linguagem de programação deve ser capaz de resolver qualquer problema que possa ser resolvido por um algoritmo quando executado por um computador. Não devem existir limites para os problemas que uma linguagem de programação pode resolver. E, quanto mais universal esta linguagem for, melhor será seu entendimento pelos seres humanos.

Os formalismos matemáticos, que veremos em breve, permitiram a criação de um conjunto bem diversificado de linguagens de programação, como vimos anteriormente neste mesmo capítulo.

Podemos classificar as linguagens de programação em alto e baixo nível. As linguagens de programação de baixo nível são aquelas que estão próximas do código de máquina. O Assembly é o melhor exemplo de linguagem de baixo nível que podemos usar para resolver problemas no Século XXI, permanece viva e pulsante há mais de 70 anos e, certamente estará aqui por mais algumas décadas. As linguagens de alto nível estão longe da máquina e próximas do homem.

As linguagens de programação de alto nível, além de utilizar um conjunto de sintática (forma) e semântica (significado) que seja familiar ao ser humano, suportam um conjunto de artefatos de código que permitem a criação de metáforas de ações de computação e cálculo, como variáveis, constantes, funções e objetos que facilitam a codificação do algoritmo. Entretanto, é preciso ressaltar que as

semelhanças entre a linguagem formal e regular, ou linguagem de programação, e a linguagem natural são limitadas. Uma linguagem natural como o português, ou o inglês, não sofrem restrições de modificação ou evolução por suas próprias regras de sintaxe e semântica. No real, natural e orgânico, estas regras servem apenas como uma referência temporal da linguagem que impõem uma estrutura de comunicação a uma população específica durante um tempo limitado. Estas regras, ainda que pareçam rígidas no tempo de uma existência humana, são alteradas, modificadas e adaptadas, de acordo com forças originadas do relacionamento pessoal, da comunicação de massa e, principalmente, do poder econômico.

Todas as linguagens de programação possuem uma sintaxe específica. A sintaxe consiste em um conjunto de regras que especificam a forma como cada artefato da linguagem deve ser construído, sejam simples declarações, funções ou objetos. A sintaxe determina como os algoritmos serão escritos e lidos. Da mesma forma, todas as linguagens de programação possuem uma semântica. As regras semânticas permitem que o compilador, ou interpretador, possa identificar o significado de cada instrução, ou artefato. A semântica determina como o algoritmo será codificado e entendido por outros programadores. O conjunto sintaxe/semântica garante que uma instrução corretamente escrita seja traduzida corretamente para código de máquina. A sintaxe sendo o conjunto de regras que permitirá que o código fonte seja lido por um compilador/interpretador e a semântica garante que este mesmo código fonte será corretamente interpretado durante o processo de tradução em código de máquina.

A execução de um determinado algoritmo é feita seguindo-se uma sequência de comandos nos quais variáveis são alocadas, modificas ou lidas, alterando, ou não o estado da máquina de acordo com o resultado de cada comando. Em ciência da computação chamamos estes comandos de declarações, do inglês statement, a palavra instrução também pode ser utilizada com o mesmo sentido.

Os comandos, ou instruções são classificados em três grandes grupos:

- **comandos para controle** explícito da sequência de execução (por

exemplo: *goto*, *;*, *begin/end*);

- **comandos condicionais** ou para a seleção que permitem a seleção de fluxos de execução alternativos (por exemplo: *if, case*);

- **comandos de iteratividade**, comandos que permitem a repetição de um determinado conjunto de outros comandos (por exemplo: *for*, *while*, *do while*).

Comandos explícitos para o controle da sequência de execução incluem por exemplo o operador ; que determina, em muitas linguagens, o fim de um comando e o início de outro. O par begin / end[13] marcam o início e o fim de um determinado bloco de comandos. Esta também é a função dos operadores { } das linguagens C e C++.

O comando, o *goto*, foi incluído nas primeiras linguagens de programação e continua sendo incluído em linguagens que estão sendo criadas no Século XXI. Este comando aparece em duas formas diferentes uma forma simples em que o fluxo de execução é desviado para outra área de memória e a forma condicional na qual, para que o fluxo seja desviado, é necessário que uma certa condição tenha ocorrido. O comum entre estas duas formas é que uma vez que o fluxo tenha sido desviado, a única forma de que esse fluxo volte ao ponto de desvio é a execução de outro comando *goto*.

O *goto* ocupa um destaque especial no mundo das linguagens de programação que pode ser resumido em uma única frase: não use. Ainda que pesquisadores renomados como Donald Knuth tenham advogado a favor KNUTH, 1974 do uso deste artefato de código mostrando que em algumas situações este uso é a melhor situação. Neste caso, e neste livro, vou ficar do lado de Edsger Dijkstra que é contra o uso do *goto*, em qualquer situação DIJKSTRA, 1968 e da minha opinião própria: o *goto* deveria ocupar um lugar de destaque na lista de

[13] Em tradução livre: início e fim.

grandes erros da humanidade.

Existem dois motivos muito fortes para que você nunca, em hipótese nenhuma, use um *goto*. A primeira é de caráter prático, à medida que o código fica mais complexo as mudanças de fluxo provocadas pelo *goto* transformam o programa em um caos [14]. Códigos com *goto* são ilegíveis, ininteligíveis e predispostos e comportamentos inesperados, tudo que você não quer em dos seus códigos. A segunda é de caráter teórico.

Em 1966 Corrado Böhm e Joseph Jacopini BÖHM e JACOPINI, 1966 demonstraram que qualquer algoritmo pode ser codificado usando apenas três classes diferentes de comandos, os comandos explícitos de sequência, comandos de seleção e comandos de interatividade ou repetição, permitindo a recursividade mostrando que qualquer algoritmo codificado com o uso do *goto* poderia ser substituído por outro, tão ou mais eficiente que ele, sem o uso deste comando de desvio de fluxo. Neste ponto, talvez seja necessário ressaltar que uma function ou procedure não são estão na mesma classe do *goto* no que diz respeito a desvio de fluxo já que, nos dois casos, o controle do fluxo de execução voltará exatamente para o próximo passo de execução, o passo que fica imediatamente após a chamada da function ou procedure.

Até o momento, nos preocupamos apenas com a execução sequencial de instruções, comandos, esta é uma das muitas formas de se codificar um algoritmo. As linguagens de programação foram desenvolvidas para atender a um determinado paradigma, ou modelo, de programação. Aqui você precisa ter cuidado.

O que existe são os paradigmas de programação. Sendo assim, as linguagens são definidas para atender um determinado paradigma de programação. A palavra paradigma, significa modelo ou padrão. Um paradigma, na ciência da computação é um modelo que pode ser seguido como forma de atingir

[14] Temos uma expressão para esse caso: código macarrônico.

um determinado objetivo de forma mais simples. Podemos simplificar esta definição ressaltando que paradigma é uma forma de programar. E programar é uma forma de codificar um algoritmo para resolver um determinado problema. Existem diversos paradigmas de programação. Se assim for, um paradigma de programação é um modelo para codificação de algoritmos para facilitar a solução de problemas.

Um paradigma de programação determina como a máquina irá acessar os dados necessários durante o tempo de execução, modificar estes dados e apresentar o resultado sempre que um determinado programa for executado ou interpretado.

O paradigma de programação mais importante é o próprio código de máquina. Trata-se do paradigma de mais baixo nível possível. Programas desenvolvidos neste paradigma atenderão uma arquitetura de máquina específica, como o x86_64 da Intel, ou a arquitetura ARM ou ainda a arquitetura RISC-V, para citar apenas três das centenas de arquiteturas em uso, e estarão limitados a ordem de execução de comandos escritos em binário.

A criação de paradigmas de programação segue a par e passo a criação de linguagens de programação. Não é raro que se crie uma linguagem para validar a criação de um paradigma e vice-versa. Não temos como abordar a todos então neste livro serão abordados apenas os paradigmas de programação imperativa, funcional, orientada a objetos e programação lógica.

1.4.1 Programação imperativa

Neste paradigma assumimos que a máquina é capaz de manter o registro de cada estado durante o processo de computação. Em geral, a persistência do estado da máquina durante o processo de computação é feita por meio de um conjunto de registradores, internos da unidade central de processamento ou por meio de variáveis e estruturas de dados mais complexas como array e records.

A unidade básica de abstração do paradigma de programação imperativa é

a procedure[15] um conjunto de instruções que devem ser executadas em sequência para atingir um objetivo computacional, específico. E este é o motivo por alguns autores chamarem o paradigma imperativo de paradigma procedural. Em algumas linguagens de programação a procedure é chamada de sub-rotina. Em outras linguagens existe a procedure e a function nos dois casos o código executado não faz parte do fluxo do programa. E a única diferença entre procedure e function está no fato que a function, depois de executada devolve ao fluxo normal do programa um valor qualquer que foi indicado na definição da function enquanto a procedure, depois de executada, não devolve nenhum valor ao ciclo principal do programa.

O paradigma de programação imperativa teve origem no trabalho de John Von Neumann e deu origem a linguagens de programação como *Fortran, Algol, Cobol, Basic, Pascal e Ada*. Estas linguagens quebraram o simbolismo estrito característico do Assembly e permitiram o uso, durante a codificação de um algoritmo, de um conjunto de instruções e comandos criados a partir de um subconjunto de uma linguagem natural, na maioria das vezes, o inglês. Talvez este tenha sido um dos segredos do seu sucesso. O outro foi, sem dúvida, sua inspiração no funcionamento do hardware.

Figura 4 - John Von Neumann
WIKIMEDIA COMMONS, 2019

As linguagens que adotam este modelo de computação foram inspiradas no uso dos recursos do hardware. Onde, por exemplo a memória física é utilizada para armazenar o estado da máquina que pode ser transportado para o ambiente da linguagem de programação por artefatos de código diversos, como por exemplo as variáveis e os valores que são armazenados por elas.

[15] Em tradução livre: procedimento

Códigos que atendem o paradigma de programação imperativa obedecem a uma sequência de comandos executados em uma ordem temporal, mantendo o registro do estado da máquina após a execução de cada comando. De forma que a mudança no estado da máquina que ocorre entre o começo e fim do algoritmo é feita de forma incremental e forçando que a ordem de execução dos comandos seja um fator crítico para o sucesso do processo. O que pode ser sintetizado na frase: primeiro faça isso e depois aquilo.

A extinção da abominação chamada *goto* do paradigma de programação imperativa permitiu a criação do paradigma de programação estruturada. Que se destaca da programação imperativa apenas por este detalhe. Então, no que diz respeito a este livro, programação imperativa, procedural e estruturada serão abordadas como um único paradigma de programação. Vamos deixar para um livro apenas sobre linguagens de programação a obrigação de destacar as pequenas diferenças entre eles.

1.1.1 Programação funcional

1.1.2 Programação lógica

CAPÍTULO 2

COMPILAÇÃO E INTERPRETAÇÃO

2 COMPILAÇÃO E INTERPRETAÇÃO

Chegamos a um ponto interessante da nossa jornada. Já vimos um pouco da história de linguagens desenvolvidas para o uso humano e percebemos a necessidade de uma linguagem que fosse entendida por máquinas. Nos resta entender como podemos traduzir uma na outra. Ou seja, como podemos escrever um conjunto de instruções em uma linguagem que os humanos entendam e ter como resultado, um conjunto de instruções em uma linguagem que as máquinas entendam. Como podemos traduzir uma linguagem na outra?

Vamos dividir este processo de tradução em três técnicas distintas, mas não excludentes, a compilação, a interpretação e os processos híbridos. Contudo, antes de colocar a mão na massa e começar a destrinchar estas técnicas, precisamos fazer algumas considerações importantes, a seguir, principalmente porque se estamos em um livro sobre linguagem será bom se todos estivermos usando a mesma.

- **Algoritmo:** uma sequência finita de instruções que precisamos seguir para cumprir determinada tarefa ou resolver um determinado problema. Os algoritmos datam do Século IX e são atribuídos a mente criativa de Abu Abdullah Muhammad ibn Musa Al-Khwarizmi, um matemático Persa, também conhecido como o fundador da álgebra. A palavra algoritmo parece ter origem em uma publicação dos trabalhos de Al-Khwarizmi na Europa do Século XII, nesta publicação seu nome foi latinizado para Algorithmi. Algoritmos ideias escritas em uma linguagem de programação.
- **Linguagem de programação:** vamos chamar de linguagem de programação uma linguagem formal e regular desenvolvida para uso humano, constituída por um conjunto finito e pré-determinado de *string*s formadas de símbolos retirados de um alfabeto finito Σ qualquer. Este é o momento em que você deve pensar nas linguagens de programação que você conhece: C, Python, Assembly ou qualquer

outra da sua preferência. Todas estas são linguagens formais e todas são linguagens de programação e todas são utilizadas para escrever um algoritmo na forma de um código.

- **Código:** vamos chamar de código ao texto que você irá escrever na sua linguagem de programação preferida. O texto deverá obedecer a gramática, semântica e sintaxe da linguagem de programação que será utilizada pelo programador. Também podemos chamar, eventualmente, de código, o conjunto de instruções que será executado pela máquina, mas neste caso sempre usaremos a expressão código de máquina. O algoritmo, escrito na linguagem de programação é o código fonte do processo de compilação ou interpretação.
- **Programa:** esta é uma definição controversa que enche a cabeça dos acadêmicos há algumas décadas. Neste livro, a palavra programa será utilizada apenas para representar o código de máquina que está em memória, sendo executado, ou que está armazenado em algum dispositivo para ser executado quando for conveniente. O programa é o resultado do processo de tradução de uma linguagem desenvolvida para humanos em uma linguagem desenvolvida para máquinas. O programa é o que resulta quando o compilador, ou o interpretador, traduz o seu código fonte, escrito em uma linguagem de programação em código de máquina.
- **Tempos:** vamos dividir em tempos as fases do processo de criação de um programa. Sendo assim, temos o tempo de redação, este é o momento em que o programador está organizando suas ideias e escrevendo o código que representa seu algoritmo em uma determinada linguagem de programação. Em seguida, temos o tempo de compilação. Este é o momento em que o compilador está traduzindo seu código em código de máquina. Por fim, temos o tempo de execução, este é o momento em o seu programa está na memória e está sendo executado.

- **Níveis:** frequentemente nos referimos as linguagens de programação como de alto ou baixo nível. Esta referência está diretamente ligada a distância que existe entre a linguagem e a máquina. Quanto mais perto do código de máquina uma linguagem de programação for, menor será o seu nível, Assembly é uma das linguagens de mais baixo nível. Por outro lado, quanto mais próximo das linguagens naturais uma linguagem de programação estiver, maior será o seu nível. O Python, o C e o Haskell são exemplos de linguagens de alto nível.

2.1 Compilação

A palavra compilação significa o ato de selecionar informações, ou materiais relevantes e adicioná-los a uma coleção. Neste livro vamos chamar de compilação ao processo de transformar um texto, escrito em uma linguagem formal e regular desenvolvida para Humanos em uma linguagem formal e regular desenvolvida para máquinas de forma persistente. Uma tradução.

A última parte da nossa definição “de forma persistente”, é muito importante para o entendimento do processo de compilação. Este “de forma persistente” significa que o resultado do processo de compilação, o código de máquina, de alguma forma permanece além da existência do compilador, ou do programa, em algum dispositivo de armazenamento.

Jogando um pouco mais de luz no parágrafo anterior podemos dizer que um compilador é um dispositivo, de hardware ou software, que traduz um texto fonte, ou texto original, em outro texto, chamado de texto objetivo. O texto fonte, ou código fonte, será escrito em uma linguagem de programação, uma linguagem forma. O texto objetivo, ou código de máquina, será escrito em uma linguagem de máquina. O resultado do processo de compilação, o texto objetivo, será, de alguma forma, armazenado para transporte e para o uso posterior.

A última frase do parágrafo anterior é muito importante: o resultado do processo de compilação, o código de máquina, será, de alguma forma, armazenado

para transporte e uso posterior. Esta condição determina que o resultado do trabalho do compilador é perene. Ele deve durar por um tempo indefinido depois que o trabalho do compilador se encerra assim, podemos armazenar o código de máquina, transportar o código entre máquinas e executar este código em máquinas diversas, não importando nenhuma referência de lugar e tempo. A única restrição a execução do código de máquina gerado é que este código só poderá ser executado em máquinas que atendam a arquitetura que escolhida para a geração do código de máquina.

Uma consequência direta destas funcionalidades e do fato de que uma linguagem de programação é uma linguagem formal e regular é que os compiladores são capazes de emitir mensagens de erro, durante o processo de compilação e, mais recentemente, durante o processo de redação do algoritmo. A Figura 4 mostra, de forma genérica, o processo de compilação como vimos até o momento.

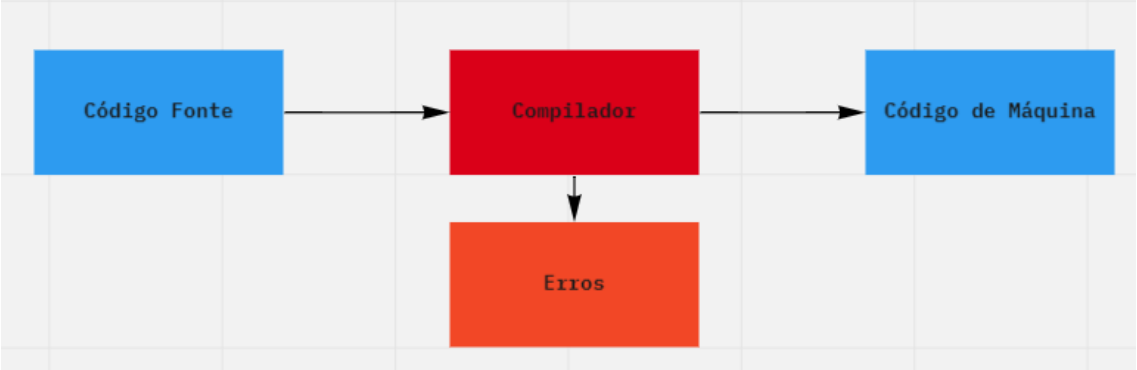


Figura 5 - Conceito de compilação

O processo de compilação tem o seu próprio tempo separado do tempo de redação e do tempo de execução. O compilador pode usar este tempo para conseguir o melhor nível de otimização possível em busca do máximo de eficiência no uso de recursos e velocidade durante a execução do código, o tempo de execução. Os recursos utilizados para a execução de um determinado código de máquina serão sempre, raros e caros. Desta forma, é imprescindível que cada instrução seja analisada e otimizada em busca do menor custo e da maior velocidade.

Vamos tentar entender todo o processo de compilação como sendo uma série de blocos especializados. Cada um destes blocos capaz de refinar, otimizar e

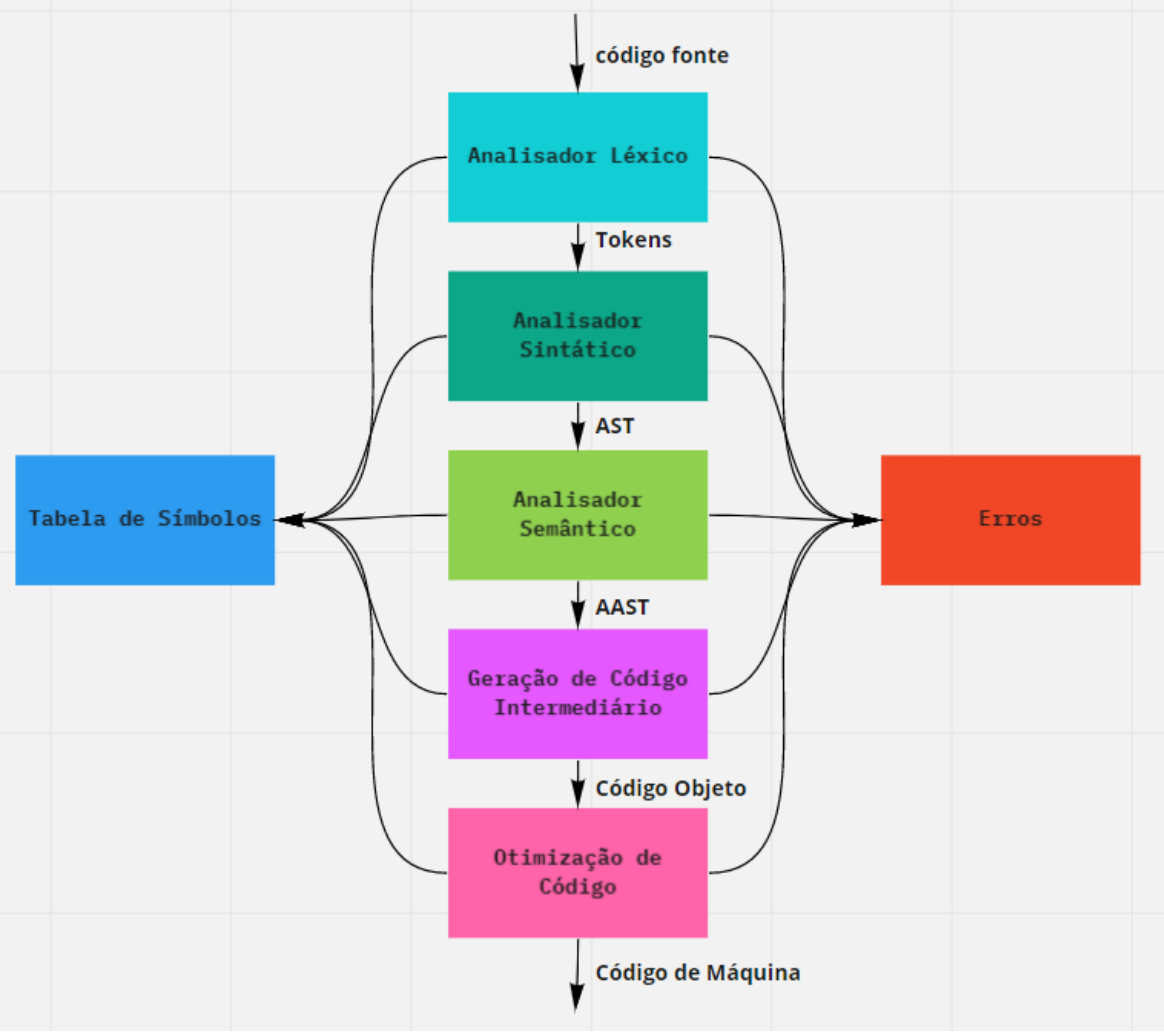


Figura 6 - - Módulos componentes de um compilador

modificar as informações originadas no código fonte, em ordem, de forma que o bloco seguinte seja capaz de agregar valor ao processo alterando e expandindo a forma da informação envolvida no processo de tradução. Nesta estrutura cada bloco conhece apenas o protocolo, ou padrão que define o formato dos dados que irá receber e o formato dos dados que deve passar para o bloco seguinte. Se fosse perfeita, tal estrutura de blocos iria permitir o máximo de especialização em cada um dos blocos. Neste livro, vamos utilizar como referência a divisão em blocos que pode ser vista na Figura 6.

Cada um destes módulos tem funções específicas para tratar a informação

de entrada e passar esta informação para o próximo módulo de forma a permitir que o próximo módulo trate essa informação da forma mais eficiente possível.

O Analisador Léxico é responsável por analisar a formação das palavras, operadores e símbolos da linguagem de programação, classificar estes símbolos e passar a lista para o Analisador Sintático. Por sua vez, cabe ao Analisador sintático verificar a forma como cada instrução foi escrita, verificando a adequação da instrução as regras de sintaxe da linguagem, uma vez que a instrução tenha sido validada, criar uma árvore sintática abstrata (AST)[16] .

O Analisador Semântico irá analisar a AST de forma criar a Arvore Sintática Abstrata Aumentada (AAST). Esta árvore contém informações referentes ao sentido de cada uma das instruções do código. Para criar esta árvore coube ao Analisador Semântico a tarefa de entender o sentido da instrução e marcar esta instrução com etiquetas que facilitem a criação do código intermediário.

O módulo responsável pela Geração do Código intermediário irá transformar a AAST em um código simples, próximo da linguagem Assembly para a entrega ao módulo responsável pela otimização do código. O uso deste código intermediário surge como uma forma de tornar mais simples e eficiente fazer otimizações neste código antes de transformá-lo em linguagem de máquina[fa1].

Ao longo do livro vamos ver cada um destes módulos com detalhe. Neste momento você irá perceber que esta divisão é apenas didática e que pode ser estendida ou reduzida de acordo com o objetivo de cada autor.

2.2 Interpretação

O processo de interpretação consiste em traduzir um texto escrito em uma linguagem formal e regular, desenvolvida para uso humano, em uma linguagem formal criada para o uso por uma máquina específica. Ou, caso não tenha sido

[16] AST é uma abreviação da expressão inglesa Abstract Sintact Tree, ou, em tradução livre: árvore sintática abstrata.

óbvio, algo muito parecido com as tarefas executadas pelo compilador. Existem, no entanto, alguns detalhes importantes. No passado, o processo de interpretação era feito linha a linha e o resultado deste processo era executado ou, quando fosse o caso, passado para a próxima linha. Este processo é característico de linguagens como o Basic. Hoje, com as linguagens muito mais complexas, contando com artefatos como funções, objetos e templates, a interpretação linha a linha é impossível, tendo sido indispensável a criação de um processo de interpretação de código inteiro ou de blocos de código auto contidos.

Enquanto o compilador opera em seu próprio tempo, o interpretador é um programa, ou dispositivo, criado para operar em tempo de execução. Podemos ressaltar isso: interpretadores são sistemas criados para a fazer a tradução entre uma linguagem de programação e o código de máquina em tempo de execução. Isso implica que o interpretador deve ser muito rápido. Esta necessidade exige que o processo de tradução, executado pelo interpretador ocorre de forma simultânea ao processo de execução do código fonte. Para que isso seja possível, o interpretador começa a traduzir e executar blocos de código fonte assim que o programa começa a ser executado. No processo de interpretação, não existe tempo disponível para ciclos repetitivos de otimização. Esta desvantagem aparente pode ser superada com algumas técnicas de otimização em tempo de execução.

Uma diferença importante entre um compilador e um interpretador é que compilador jamais executa o código de máquina, enquanto o interpretador faz isso todas as vezes que é executado. Observe que, graças a isso, o resultado do processo de compilação é persistente enquanto o resultado do processo de interpretação e efêmero.

Vamos, apenas para entendimento, considerar que não há nenhuma diferença entre o tempo de interpretação e o tempo de execução. Se considerarmos assim, o código de máquina resultante está na memória no mesmo instante de tempo em que o processo de interpretação está sendo executado. Esta hipótese permite a criação de rotinas de otimização para o código que está em execução. De fato, a linguagem Java e o V8 (interpretador de Javascript do Navegador Chrome) usam estes tipos de técnicas e este uso é parte do sucesso tanto do

Chrome quanto do Java. Estes são os processos híbridos[fa2].

2.3 Processos Híbridos

Acho que já escrevi isso, mas, não custa nada repetir: recursos computacionais serão sempre raros e caros. E esta é a alavanca que impulsiona todo o desenvolvimento tecnológico além da criação de soluções para problemas novos que a própria tecnologia criou. Quando conseguimos resolver estes problemas de forma mais rápida, usando menos recursos, geramos mais riqueza. Nesta busca de eficiência podemos combinar conceitos de compilação e interpretação em ambientes híbridos com compiladores e interpretadores trabalhando lado a lado de forma que o resultado do todo seja melhor que o resultado da soma das partes.

O usuário digita o código fonte e manda executar. Imediatamente o processo de interpretação coloca o código de máquina na memória e inicia o tempo de execução. Em tempo de execução, com o programa rodando, podemos observar a eficiência de cada artefato de código em memória, identificar artefatos que possam ser otimizados, disparar o compilador em paralelo para otimizar este artefato ao máximo e, assim que os clico extra de otimização estejam completos, substituir este artefato em memória. E esta tecnologia que fez o sucesso do Java.

No caso do Java o compilador produz um código intermediário, o código objeto que no caso do Java é chamado de Bytecode. este código é persistente e altamente otimizado. Depois de gerado o Bytecode pode ser interpretado em qualquer máquina virtual. Ou, em outras palavras, você escreve seu código fonte uma única vez, gera o Bytecode e, portando este código intermediário você pode executar seu programa em qualquer arquitetura de hardware desde que exista uma máquina virtual Java rodando nesta arquitetura.

O processo híbrido original do Java em duas fases distintas com um interregno de portabilidade, não é a única opção de hibridismo que podemos encontrar. O Javascript, uma linguagem criada para ser executada em navegadores web é, geralmente, interpretada. O código fonte é carregado junto com uma

determinada página e cabe ao navegador interpretar este código e executar as instruções desejadas. A fluidez da navegação web depende da velocidade e eficiência do interpretador Javascript dos navegadores. O Google Chrome V8 engine é um dos melhores exemplos de sucesso nesta área.

O V8 engine é um motor de interpretação Javascript, desenvolvido em C++, em código aberto, que está em desenvolvimento constante pela equipe de desenvolvedores do Google na Alemanha. O V8 usa dois processos de tradução. O primeiro, um compilador muito rápido, cujo processo de otimização é o mínimo, para que o código seja executado imediatamente. O segundo, um compilador de otimização que gera código de máquina altamente eficiente e rápido. Este segundo compilador atua em segundo plano, depois que o código já está rodando, e observa o processo de uso do código de máquina, em memória e durante a execução do programa, observando a existência, ou não de artefatos de código que precisam de níveis mais profundos de otimização. Leia este parágrafo novamente.

Acabei de falar que o interpretador do V8 engine é composto de dois compiladores. Essa linha entre compilação e interpretação é tênue e autores diferentes traçam esta linha em pontos diferentes. Neste livro vamos tentar explicar as características de cada um, os algoritmos e técnicas envolvidas e a forma como você poderá fazer uso destes algoritmos para, além de projetar seu próprio compilador, ou interpretador, usar este conhecimento para criar soluções mais rápidas, eficientes e baratas.

3 LINGUAGENS FORMAIS E REGULARES

> *"The purely formal language of geometry describes adequately the reality of space. We might say, in this sense, that geometry is successful magic. I should like to state a converse: is not all magic, to the extent that it is successful, geometry?" Rene Thom*

Podemos começar o estudo das linguagens formais lembrando que apenas a evolução da nossa espécie levou a criação da linguagem natural. Um processo aleatório, resultado da evolução e do relacionamento social. São naturais todas as

linguagens que falamos e que são resultado desta evolução.

O relacionamento social, certamente, foi o fator decisivo para a criação de um conjunto de regras de uso e formação para cada uma das linguagens naturais que existe de forma a permitir que estas linguagens naturais pudessem ser aprendidas, praticadas e preservadas por todos os indivíduos de uma aglomeração humana. O entendimento do que o outro fala e a forma como ele fala fundamentam a convivência e evitam a violência e afastam a guerra. Este resultado da evolução, as linguagens naturais, por sua vez, permitiram a criação das linguagens artificiais. As linguagens artificiais são as linguagens criadas pelo homem com um objetivo específico, para entender uma necessidade ou resolver um problema. São exemplos de linguagens artificiais, o Haskell, a Lógica Proposicional de Aristóteles e as Partituras Musicais de Guittone d' Arezzo. As duas últimas são exemplos de linguagens artificiais ancestrais e servem para nos recordar que a criação de linguagens artificiais não é exclusividade nem da modernidade nem da computação.

A partir da metade do Século XX, como vimos no Capítulo 1, as linguagens artificiais explodiram em quantidade e diversidade. Notadamente graças as linguagens de programação. Ao nosso estudo interessam apenas as linguagens artificiais formais e regulares e usaremos os termos linguagem formal e linguagem regular como se sinônimos fossem até que destacar a diferença entre elas seja inevitável. Todas as linguagens regulares são formais, mas nem todas as linguagens formais são regulares. Da mesma forma, todas as linguagens formais são artificiais, mas nem todas as linguagens artificiais são formais. O Esperanto é o exemplo clássico de linguagem artificial não formal e, portanto, está fora do escopo deste livro.

Ao longo deste livro usaremos os termos formal ou regular, para nos referir à alguma coisa que seja passível de ser computável e, neste caso, "passível de ser computável" implica forçosamente na necessidade do uso de um algoritmo. E, muitas vezes vamos usar os termos linguagem formal e linguagem regular de forma como se fossem sinônimos. Contudo, existem linguagens formais que não são regulares. As linguagens formais e regulares são o objeto de estudo deste livro.

Para que uma linguagem seja formal é necessário que a forma das suas sentenças (sintaxe), e o seu significado (semântica), seja precisa, rígida e algoritmicamente determinado. Veremos que uma linguagem será considerada formal se suas instruções puderem ser identificadas por um autômato finito ou por uma expressão regular, atendendo ao Teorema de Kleene.

Uma linguagem, seja ela natural ou artificial, será um conjunto de cadeias de símbolos, *string*s, formadas com base em um alfabeto finito e pré-definido. A sintaxe irá determinar as formas destas *string*s sem se preocupar muito com a forma dos seus menores componentes, os Lexemas.

Lexema é uma palavra originada do grego que representa a unidade mínima de distinguível de um sistema semântico, seus sinônimos mais comuns são morfema e palavra. Para nós, neste livro, o lexema é a menor unidade significativa de uma linguagem. Pode ser, por exemplo a palavra for ou o operador +. Cabe ao Analisador Léxico identificar os lexemas específicos da linguagem que será compilada. Mas isso é andar um pouco rápido demais. Vamos voltar as linguagens formais.

O estudo das linguagens formais começa com os conceitos de alfabeto, *string*, operações com *string*s e a matemática que suporta estes conceitos e operações.

3.1 A Base Matemática

Compiladores e interpretadores trabalham com linguagens formais e linguagens formais são estruturas matemáticas desenvolvidas na linguística computacional. E podemos começar com a definição de alfabeto.

1.1.3 Alfabetos, *string*s e linguagens

Um alfabeto é um conjunto finito de símbolos, muitas vezes chamados de símbolos terminais não repetitivos. Para todos os efeitos neste livro, um símbolo é uma unidade abstrata que não tem significado em si próprio trata-se da unidade mais primitiva de dado que pode ser representada em um alfabeto. Em alguns

alfabetos os símbolos detêm significado graças ao seu uso específico. Como exemplo, podemos citar as letras maiúsculas ou minúsculas do português. Cada letra em si, não tem significado, mas nós as reconhecemos.

$$\Sigma_{ptbr} = \{a..z, A..z\}$$

Os alfabetos não admitem repetições de símbolos e, atendem a definição matemática de conjunto. Não existe nenhuma restrição quanto ao tipo de símbolo que compõe um alfabeto, eles podem ser compostos por dígitos, números reais, letras, caracteres, emoji ou qualquer outro tipo de do. Representamos o alfabeto pelo caractere sigma maiúsculo: Σ. Dessa forma podemos ter:

$$\Sigma = \{a_1, a_2, a_3, \dots, a_k\}$$

Que representa a existência de um alfabeto com k elementos onde k é a cardinalidade do alfabeto e representa seu número de membros. Por exemplo, a cardinalidade de um alfabeto Σ_{prbr} formado apenas pelas letras do alfabeto do português seria dada por:

$$\left|\Sigma_{ptbr}\right| = |\{a..z, A..z\}| = 52$$

Com um pouco mais de formalidade, podemos dizer que, se um alfabeto contém um elemento a dizemos que $a \in \Sigma$ e este elemento e a será chamado de membro de Σ. Da mesma forma se Σ não contém c então dizemos que $c \notin \Sigma$.

O alfabeto que não contém símbolos é um conjunto vazio e é representado por $\emptyset$ observe que a cardinalidade de um conjunto vazio é zero. Observe também que a ordem dos membros em um alfabeto não é relevante.

Uma *string* em Σ é um subconjunto de Σ, ordenado e com possíveis repetições. A quantidade de símbolos em uma *string*, determina sua cardinalidade que podemos chamar de comprimento.

A ordem dos elementos em uma *string* é importante de tal forma que duas *string*s com os mesmos elementos em ordem diferente, são *string*s diferentes. Uma *string* que não contém símbolos do seu alfabeto ainda é uma *string*, vamos chamá-la de vazia e representar pelo símbolo ε, que será lido com epsilon. Note que a *string* vazia ε é comum a todos os alfabetos e que sua cardinalidade é zero

A definição do símbolo ε para representar uma *string* vazia implica no

banimento deste símbolo de todos os alfabetos que usaremos neste livro para evitar qualquer tipo ambiguidade quanto ao seu significado seu significado.

Exemplo 1:

Considere o alfabeto $\Sigma = \{a, b\}$. Defina o conjunto de todas as *string*s possíveis deste alfabeto cujo comprimento seja inferior a três.

$$S = \{a, b, aa, ab, ba, bb\}$$

Formalmente representamos as *string*s usando a mesma norma que usamos para a representação de conjuntos. Uma *string* s de um alfabeto $\Sigma = \{1, a\}$, deveria ser representada por:

$$s = \{a, 1, a\}$$

vamos adotar esta representação. Exceto quando formos tratar especificamente de operações com *string*s podemos representar $s = \{a1a\}$, ou simplesmente "$a1a$".

Podemos aproveitar a oportunidade e definir outra entidade matemática interessante a tupla. Tuplas são sequências finitas de elementos. o número de elementos da sequência dá nome a tupla. Logo, uma tupla com três elementos é uma $3 - tupla$, uma tupla com cinco elementos é uma $5 - tupla$. A tupla com dois elementos é chamada de par. Se a ordem dos elementos é importante, na definição da tupla, então as chamamos de tuplas ordenadas. No caso da $2 - tupla$ termos o par ordenado. Dessa forma, podemos melhorar a definição de *string*: uma *string* de um alfabeto Σ é de comprimento n maior que zero é uma $n - tupla$ ordenada do alfabeto Σ.

Exemplo 2:

Considere os alfabetos $\Sigma_1 = \{a, b, c\}$ e $\Sigma_2 = \{3\}$. Neste caso, podemos ter as seguintes *string*s:

a) $s_1 = \{abc\}$ a qual é uma *string* de Σ_1;
b) $s_2 = \varepsilon$ e $s_3 = \{3333\}$ as quais são *string*s de Σ_2;

O conceito de cardinalidade, discutido em relação a alfabetos e *string*s permite a criação de uma notação um pouco diferente. Tomemos a como exemplo a *string* $aabb$, neste caso a cardinalidade será dada pelo número de símbolos da *string* então:

$$|aabb| = 4$$

Podemos afirmar, sem sombra de dúvida que $|\varepsilon| = 0$. Ou seja, a cardinalidade, ou comprimento, da *string* vazia é zero. Com estas duas definições solidificadas podemos criar uma notação que especifique a quantidade de elementos de um determinado tipo na *string* emprestando um pouco do conceito de cardinalidade. Neste caso teremos:

$$|aabb| = 4$$
$$|aabb|_a = 2$$
$$|aabb|_b = 2$$

Resta definir a igualdade entre *string*s tomemos agora duas *string*s quaisquer:

$$s_1 = \{a_1, a_2, a_3, \dots, a_i\} \wedge s_2 = \{b_1, b_2, b_3, .., b_i\}$$

Podemos dizer que $s_1 = s_2$ se e somente se $a_i = b_i$ para todo e qualquer valor de i. Ou se preferir, um pouco menos de formalidade, podemos determinar a igualdade entre duas *strings* primeiro comparando seus comprimentos e dois, comparando, da esquerda para direita, todos os seus elementos. As *string*s serão iguais se o comprimento for igual e todos os seus elementos forem iguais um a um na mesma ordem.

A concatenação de *string*s é uma operação binária interessante. Definimos a concatenação como uma operação binária que recebe como operandos duas *string*s e retorna uma nova *string* formada pelas duas anteriores. Podemos representar a concatenação de duas *string*s $s_1 = \{ab\}$ e $s_2 = \{ba\}$ como:

$$s_1 s_2 = \{ab\}\{ba\} = \{abba\}$$

Formalmente diríamos que a concatenação de s_1 e s_2 é definida por:

$$s_1 s_2 = \{xy | (x \in s_1) \wedge (y \in s_2)\}$$

Que vem a ser a definição exata de produto cartesiano se apenas substituirmos xy pelo par ordenado (x, y) e de onde podemos inferir algumas

propriedades interessantes:

a) $s\varepsilon = s$;
b) $\varepsilon s = s$;
c) $\varepsilon\varepsilon = \varepsilon$;
d) $|\varepsilon s| = |s|$;
e) a concatenação não é comutativa logo: $s_1 s_2 \neq s_2 s_1$;
f) a concatenação é associativa logo: $s_1(s_2 s_3) = (s_1 s_2)s_3$;
g) $|s_1 s_2| = |s_1| + |s_2|$,

Outra operação interessante com *string*s é a reversão, ou como preferem alguns autores reflexo. A reversão é uma operação unária que, quando aplicada a uma *string* resulta em outra *string* de mesmo comprimento e com todos os elementos na ordem inversa. Considere a *string* $s = \{abcd\}$ sua reversão será dada por:

$$s^R = \{abcd\}^R = \{dcba\}$$

Novamente existem algumas propriedades que devemos observar:

a) $\varepsilon^R = \varepsilon$;
b) $|s^R| = |s|$;
c) Se $s = \{a\}$ então $s^R = \{a\}$;
d) $s^R x = x s^R$
e) $(s^R)^R = s$;
f) $(s_1 s_2)^R = s_2^R s_1^R$

Por último temos a repetição. Usamos a repetição para denotar *string*s que sejam a concatenação da *string* com ela mesma. A repetição será indicada por um número m inteiro positivo $\mathbb{Z}^+$. Por definição vamos considerar que $m = 0$ indica a *string* vazia. Ou seja:

$$\begin{cases} s^m = s^{m-1}s, m > 0 \\ s^0 = \varepsilon \end{cases}$$

Exemplo 3

Considere a *string* $s_1 = \{ab\}$, teremos:

1. $s_1^0 = \{\varepsilon\}$;
2. $s_1^1 = s_1 = \{ab\}$;
3. $s_1^2 = s_1 s_1 = \{abab\}$;
4. $\varepsilon^0 = \varepsilon^1 = \varepsilon^2 = \varepsilon$

Agora que temos as definições de alfabeto e *string* podemos começar a definir a linguagem L como sendo um conjunto de *string*s de um alfabeto especificado. Esta definição não contém um limite implícito para a cardinalidade deste conjunto. Isto implica que podemos ter uma linguagem L qualquer que tenha um número infinito de membros, um número finito de membros ou nenhum membro. No caso do estudo de compiladores e interpretadores, consideraremos apenas linguagens formais com cardinalidade finita. Ou seja,

$$0 \leq |L| \leq \infty$$

Formalmente podemos usar a notação de conjuntos para definir uma linguagem ser você considerar uma linguagem L derivada do alfabeto $\Sigma = \{1,2,3,4,5,6,7,8,9,10\}$ podemos definir L da seguinte forma:

$$L = \{x | x \in \Sigma^* \wedge x \in \mathbb{Z} \, | 30 < x < 90\}$$

Em bom português podemos dizer que é uma linguagem formada por *string* que representam os números inteiros maiores que 30 e menores que 90.

Exemplo 4:

Considere o alfabeto $\Sigma = \{a, b\}$ e defina a linguagem L_1 e L_2 sabendo que L_1 é formada por todas as *string*s começadas com o símbolo a com no máximo três membros e que L_2 é formada por todas as *string*s que contém a mesma quantidade de símbolos a e b.

$$L_1 = \{a, aa, ab, aaa, aab, aba, abb\}$$

$$L_2 = \{ab, ba, aabb, abab, baba, bbaa, aaabbb \ldots\}$$

Neste ponto podemos observar que uma linguagem pode ser entendida como sendo um conjunto de conjuntos em dois níveis distintos. No primeiro, um conjunto não ordenado de membros, as *string*s no segundo nível, um conjunto de símbolos terminais, o alfabeto.

Dada uma linguagem L todas as *string*s pertencentes a essa linguagem serão chamadas de lexemas ou palavras. Tomando a linguagem L_2 podemos dizer que $aabb$ é lexema, ou palavra, de L_2 ou, formalmente dizemos que $aabb \in L_2$.

A cardinalidade de uma linguagem L é o número de *string*s, ou se preferir, número de instruções contidas nesta linguagem. A linguagem L_2, que definimos anteriormente, tem cardinalidade infinita enquanto L_1 tem cardinalidade igual a sete, ou seja:

$$|L_1| = |\{a, aa, ab, aaa, aab, aba, abb\}| = 7$$

Observe que uma linguagem finita é um conjunto finito de *string*s construídas a partir de um alfabeto também finito. Um caso especial de linguagem é a linguagem vazia $L_v = \{\}$ que será representada pelo símbolo $\emptyset$ cuja cardinalidade é zero $|L_v| = 0$. Outro caso especial é a linguagem denominada, fecho de Kleene e representada por Σ^*. Esta linguagem é, por definição, o conjunto de todas as *string*s de comprimento finito que podem ser criadas utilizando os símbolos do alfabeto Σ acrescido da *string* vazia. Logo, considerando o alfabeto $\Sigma = \{a, b\}$ teremos:

$$\Sigma^* = \{\varepsilon, a, b, ab, ba, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}$$

Exemplo 5:

Considere as linguagens Σ^* em relação aos alfabetos propostos:

1. Se $\Sigma = \{a, b\}$ então $\Sigma^* = \{\varepsilon, a, b, ab, ba, aa, bb, aaa, aab, \dots\}$;
2. Se $\Sigma = \{a\}$ então $\Sigma^* = \{\varepsilon, a, aa, aaa, aaaa, \dots\}$;
3. Se $\Sigma = \emptyset$ então $\Sigma^* = \{\varepsilon\}$.

Neste caso, além de apresentar a linguagem Σ^* na sua ordem lexicográfica, podemos inferir que qualquer *string* derivada do alfabeto Σ é obrigatoriamente um membro da linguagem Σ^* e qualquer linguagem L do alfabeto Σ é obrigatoriamente

um subconjunto da linguagem Σ^*. Ou se preferir $L \subset \Sigma^*$. Além disso, o conjunto formado por todos os subconjuntos possíveis de Σ^* é o conjuto de todas as linguagens possíveis derivadas do alfabeto Σ.

Existe outra linguagem importantes derivada de um alfabeto Σ qualquer e criadas por definição. A linguagem Σ^+que representa todas as *string*s do alfabeto Σ com comprimento maior ou igual a 1. Neste caso, a linguagem Σ^+ não incluí a *string* vazia ε. Em resumo temos:

a) a linguagem L de um alfabeto Σ;
b) a linguagem Σ^n para todas as *string*s de comprimento n para qualquer n pertencente aos números naturais $\mathbb{N}$;
c) A linguagem Σ^*, para todas as *string*s de Σ;
d) A linguagem Σ^+ para todas as *string*s de Σ com comprimento maior ou igual a 1;
e) A linguagem $\emptyset$ a linguagem sem nenhuma *string* do alfabeto Σ.

Exemplo 6:

Considere o alfabeto $\Sigma_b = \{0,1\}$

1. $\Sigma_b, L = \Sigma_b^2 \Rightarrow L = \{00,01,10,11\}$
2. $\Sigma_b, L = \Sigma_b^8 \Rightarrow L = \{00000000, 00000001, \ldots\}$ (bytes).

1.1.4 Operações com linguagens

Dado um alfabeto finito qualquer Σ podemos afirmar que:

a) $L = \emptyset$ é uma linguagem formal;
b) Para qualquer *string* $s \in \Sigma^*, L = \{s\}$ é uma linguagem formal;
c) Para qualquer *string* $s \in \Sigma^*, L = \{s\}$ é uma linguagem formal;
d) Se L_1 e L_2 são linguagens formais então $L_1 \cup L_2$, é uma linguagem formal;
e) Se L_1 e L_2 são linguagens formais então $L_1 L_2$, é uma linguagem formal;
f) Se L_1 é uma linguagem formal então L_1^* é uma linguagem formal.

As afirmações que indicam que $\emptyset, \{x\} e\, \{\varepsilon\}$ são linguagens formais são as afirmações básicas sobre as quais construiremos, passo-a-passo, a definição de linguagem formal. As outras três, derivadas das operações União, Concatenação e fechamento têm importância ímpar para a definição das linguagens formais. Antes de prosseguirmos, precisamos definir estas operações. Em linhas gerais, tudo que teremos que fazer é observar as operações com *string*s e aplicar a todas as *string*s da linguagem.

3.1.1.1 União

Trata-se de uma operação binária. Considere as linguagens L_1 e L_2 referentes ao alfabeto Σ. Definimos a união destas linguagens como:

$$L_1 \cup L_2 = \{x \,|(x \in L_1) \vee (y \in L_2)\}$$

Exemplo 7:

Considere o alfabeto $\Sigma = \{a, b, c\}$ e as linguagens $L_1 = \{aa, bb\}$ e $L_2 = \{cc, cb\}$ neste caso a concatenação entre estas duas linguagens será dada por:

$$L_1 \cup L_2 = \{aa, bb\}\{cc, cb\} = \{aa, bb, cc, cb\}$$

Algumas propriedades da união são importantes para o estudo das linguagens formais e regulares:

a) $L_1 \cup (L_2 \cup L_3) = (L_1 \cup L_2) \cup L_3$ (associativa);

b) $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$ (associativa para concatenação).

3.1.1.2 Concatenação

Trata-se de uma operação binária. Considere as linguagens L_1 e L_2 referentes ao alfabeto Σ. Definimos a concatenação destas linguagens como:

$$L_1L_2 = \{xy \,|(x \in L_1) \wedge (y \in L_2)\}$$

Exemplo 8:

Considere o alfabeto $\Sigma = \{a, b, c\}$ e as linguagens $L_1 = \{aa, bb\}$ e $L_2 = \{cc, cb\}$ neste caso a concatenação entre estas duas linguagens será dada por:

$$L_1L_2 = \{aa, bb\}\{cc, cb\} = \{aacc, aacb, bbcc, bbcb\}$$

A concatenação de uma linguagem L consigo mesma, repetidamente dá origem a uma representação muito próxima da representação da operação de exponenciação. Sendo assim, considerando uma linguagem $L = \{a, b\}$ teremos:

a) $L^0 = \emptyset$
b) $L^1 = L = \{a, b\}$;
c) $L^2 = LL = \{a, b\}\{a, b\} = \{aa, ab, ba, bb\}$;
d) $L^3 = LLL = \{a, b\}\{a, b\}\{a, b\} = \{aa, ab, ba, bb\}\{a, b\} = \{aaa, aba, baa, bba, aab, abb, bab, bbb\}$;
e) $L^k = LLL \ldots L$ (k vezes).

3.1.1.3 Fechamento de Kleene

O fechamento de Kleene, também chamado de fechamento recursivo transitivo é uma operação unária. Se L_1 for uma linguagem L_1^* indica que podemos pegar zero ou mais *string*s de L_1^* e concatená-los. Definimos seu fechamento por:

$$L_1^* = \bigcup_{i=0}^{\infty} L_1^i = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$$

Em palavras podemos dizer que o fechamento de Kleene de uma linguagem dada é a união de todas as linguagens possíveis, formadas pela concatenação dela com ela mesma, operação que definimos como repetição.

$$L^n = \begin{cases} \{\varepsilon\} \\ L^{n-1}L \end{cases}$$

Por causa desta relação com a concatenação este fechamento é dito como sendo um fechamento por concatenação e união, ou se preferir um pouco mais de formalismo, podemos recorrer a álgebra abstrata. O L^*, é chamado de monoide livre. E deste conceito podemos tirar o subgrupo livre, representado por L^+ definido por:

$$L_1^+ = \bigcup_{i=1}^{\infty} L_1^i = L^1 \cup L^2 \cup L^3 \ldots$$

Ou seja, todas as linguagens possíveis de serem criadas a partir de um alfabeto Σ excluindo a linguagem vazia. Ou ainda:

$$L^+ = L^* - \{\varepsilon\}$$

A linguagem Σ^+ é o resultado da operação unária chamada de fechamento transitivo.

Exemplo 9:

Considere o alfabeto $\Sigma = \{a, b, c\}$ e as linguagens $L = \{aa, bb\}$ neste caso o fechamento de L_1 será dado por:

$$L_1^* = \{aa, bb\}^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$$

Logo:

$$L_1^* = \{aa, bb\}^* = \{\varepsilon\}, \{aa, bb, \}, \{aaaa, aabb, bbaa, bbbb\}, \{aaaaa, \ldots\}, \ldots\}$$

Finalmente temos que:

$$L_1^* = \{\varepsilon, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaa, \ldots\}$$

O fechamento de Kleene de uma linguagem finita L qualquer resulta em uma linguagem infinita L^*, com apenas duas exceções: a linguagem $\emptyset$ e linguagem que contém apenas a *string* vazia $\{\varepsilon\}$, nos dois casos, o resultado da operação de fechamento será a própria linguagem.

Podemos voltar um pouco e ver as linguagens definidas anteriormente com relação a um alfabeto e observar que dado um alfabeto Σ qualquer e uma linguagem L derivada deste alfabeto o fechamento da linguagem L é igual a linguagem Σ^*, e contém todas as *string*s possíveis do alfabeto Σ. Isso mostra que:

$$L \subseteq \Sigma^*$$

Equação que podemos extrapolar e ler como: toda e qualquer linguagem L definida a partir de um alfabeto Σ é um subconjunto de Σ^*. Podemos destacar algumas propriedades interessantes do fechamento por concatenação:

f) $L \subseteq \Sigma^*$;

g) $\forall\, L_1 \in L^* \wedge L_2 \in L^* \mid L_1 L_2 \in L^*$;

h) $(L^*)^* = L^*$;

i) $(L^*)^R = (L^R)^*$;

j) $\emptyset^* = \varepsilon$;

k) $\{\varepsilon\}^* = \{\varepsilon\}$.

Por fim, já que Σ^* representa a maior linguagem possível referente a um determinando alfabeto Σ podemos definir o complemento de uma linguagem como sendo:

$$\bar{L} = \Sigma^* - L$$

Exemplo 10:

Vamos considerar uma linguagem de programação qualquer que defina os nomes dos seus identificadores, variáveis por exempo, como tendo que ser escritos com letras maiúsculas e números. Podemos avaliar este problema considerando dois alfabetos:

$$\Sigma_{letras} = \{A, B, .. Z\}$$

$$\Sigma_{números} = \{0,1,2,3,4,5,6,7,8,9\}$$

De tal forma que o nosso alfabeto será:

$$\Sigma = \Sigma_{letras} \cup \Sigma_{números}$$

Com estes dois conjuntos podemos construir uma linguagem para as variáveis utilizando a união e o fechamento de Kleene.

$$L_{var} \subseteq \Sigma^*$$

Que podemos ler como L_{var} é um subconjuto de todas as *string*s que podem ser criadas com a união dos dois alfabetos definidos. O que inclui as *string*s vazias ε. Não parece ser uma boa ideia ter uma linguagem de programação com a possibilidade de termos identificadores, variáveis, que sejam *string*s vazias. Sendo assim:

$$L_{var} \subseteq \Sigma^+$$

O que garante que podemos ter identificadores, formados por *string*s do alfabeto Σ de qualquer comprimento, usando qualquer combinação de letras e numeros inteiros.

Neste ponto podemos fazer uma afirmaçao poderosa. Uma linguagem derivada de um alfabeto será considerada regular se puder ser definida por meio

da *string* vazia e um conjunto finito de operações de união, concatenação e fechamento. Ao conjunto destas operações, capazes de determinar que uma determinada linguagem seja regular daremos o nome de expressões regulares.

3.2 Expressões regulares

Uma expressão regular é um forma lógica e algébrica para representar uma linguagem formal, regular e complexa em termos de linguagens simples combinadas por meio dos operadores de união, concatenção e fechamento. Antes de nos aprofundarmos neste processo vamos melhorar um pouco a forma como lemos estas expressões:

a) União: pode ser lida como e, e representada por $+$ ou $\vee$;
b) Concatenação: pode ser lida como “seguida por” ou “concatenado à”;
c) Fechamento: pode ser lido por “tantas ... quanto desejado”;

Assim:

$$x \vee dc^*$$

Pode ser lido como: x e d seguido de quantos c desejarmos.

Exemplo 11:

Considere o alfabeto $k = \{0,1\}$ a linguagem $L = \{0^j 1^k | m \geq 0, n \geq 0\}$, neste caso, podemos ler L como sendo a linguagem formada por *string*s de qualquer quantidade de zeros concatendos a qualquer quantidade de uns, ou seja:

$$L = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$$

Com um pouco de formalismo podemos definir uma linguagem regular R derivada de um alfabeto Σ se pudermos expressar esta linguagem a partir da aplicação de um conjunto finito de operações união, concatenação e fechamento sobre um conjunto de linguagens mais simples, que estejam contidas em R começando com a linguagem vazia $\emptyset$.

Expressões regulares são conjuntos diversos das operações de união, concatenação e fechamento em uma notação. Sendo assim, podemos afirmar que

que qualquer linguagem R derivada do alfabeto Σ será considerada regular se ela puder ser expressa apenas com as seguintes regras:

a) $\emptyset$ é uma expressão regular representa uma linguagem vazia $\{\}$;
b) ε é uma expressão regular e representa uma linguagem com uma *string*, a *string* vazia $\{\varepsilon\}$;
c) se $s \in \Sigma^*, s$ é uma expressão regular e representa uma linguagem com apenas uma *string* $\{s\}$;
d) Se R_1 e R_2 são expressões regulares então $R_1 \cup R_2$, é uma expressão regular que representa uma linguagem com todos os elementos de $R_1 e\ R_2$;
e) Se R_1 e R_2 são expressões regulares então $R_1 R_2$, é uma expressão regular que representa uma linguagem resultante da concatenação de $R_1 R_2$;
f) Se R é uma expressão regular então R^* é uma expressão regular.

Onde cada uma destas expressões regulares representa um conjunto contendo linguagens formais possíveis derivadas do mesmo alfabeto. O que pode ser facilmente percebido comparando-se estes itens com as afirmações feitas no início da seção 2.21 indicando que expressões regulares são linguagens formais e regulares e vice-versa.

As expressões regulares são as ferramentas com as quais podemos criar linguagens regulares. Podemos traçar um paralelo com a aritmética. Na aritmética podemos utilizar as operações básicas para criar expressões, no mundo da linguagem computacional usamos as expressões regulares para criar linguagens. Ainda um pouco mais, podemos utilizar as expressões regulares para validar uma determinada linguagem ou *string*.

CAPÍTULO 2

LINGUAGENS FORMAIS, REGULARES E ANÁLISE LÉXICA

4 O ANALISADOR LÉXICO

Segundo Mogensen 2010, **a palavra "léxico" tem o sentido original de "referente as palavras"**, do ponto de vista da ciência da computação é necessário entender estas palavras como sendo os símbolos usados em um determinado processo computacional. **A análise léxica consiste na identificação das palavras e símbolos contidos no código-fonte**. A análise léxica é, geralmente, realizada por meio de expressões regulares, os léxicos, uma notação algébrica utilizada para a discrição de conjuntos de *strings*. Os analisadores léxicos gerados estarão em uma classe de programas extremamente simples chamados de autômatos finitos BERGMANN, 2010.

As palavras e símbolos, usados para a criação de um programa serão, posteriormente, passados na forma de *tokens*, para os outros módulos do processo de compilação. Uma vez que as palavras, e símbolos, tenham sido corretamente identificados, cada *token* será composto de, no mínimo, duas informações: uma classe, ou tipo, e um valor. Este último, o valor, sendo opcional e utilizado apenas quando for necessário completar a informação contida no tipo.

Antes de trabalhar com estes conceitos, torna-se imprescindível estudar um pouco sobre conjuntos, *strings*, *tokens*, expressões regulares e autômatos finitos. Linguagens e strings

A sua linguagem de programação é, até o momento em que este texto está sendo escrito, obrigatoriamente, uma linguagem formal[17]. **Definimos linguagem formal como sendo uma linguagem que possa ser rígida e precisamente especificada** BERGMANN, 2010 e que, de alguma forma, seja conveniente para o uso em computação, diferenciado da linguagem natural em rigidez e precisão.

[17] Em alguns textos científicos não existem diferenças entre o uso dos termos linguagem formal e linguagem regular. Neste texto vamos nos ater ao termo linguagem formal.

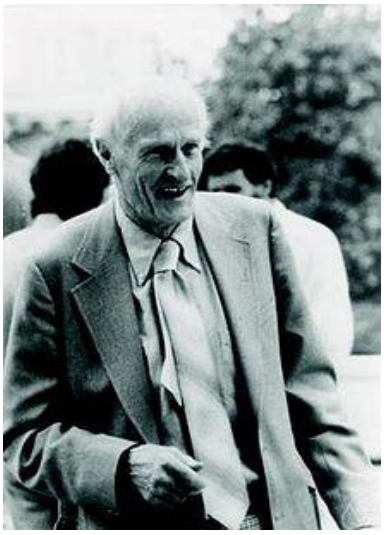


Figura 7 - Stephen Kleene (WIKIPEDIA CONTRIBUTORS, 2017)

O ***Teorema de Kleene*** afirma que, para que uma linguagem seja considerada formal, ela deve ser definida por: expressões regulares e máquinas de estado finitas. Um dos mais importantes corolários deste teorema indica que qualquer máquina de estados finita pode ser transformada em uma expressão regular e vice-versa.

A especificação de uma linguagem formal requer o uso de alguns conceitos importantes, importados da álgebra. O conceito de conjunto é o primeiro deles e da origem a todo o processo de criação e uso das linguagens de programação.

Define-se um conjunto como sendo uma coleção de objetos únicos chamados de elementos. Os conjuntos serão representados por uma letra maiúscula em negrito e itálico e seus elementos por letras minúsculas, números ou símbolos, entre chaves como pode ser visto na Equação 1.

$$\boldsymbol{A} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{e}, \mathrm{g}, \mathrm{h}\}$$

Um conjunto pode conter um número infinito de elementos, elemento nenhum ou qualquer quantidade entre estes dois extremos. Denomina-se o conjunto que não têm elementos de conjunto vazio, representado pelo símbolo ∅. Os elementos de um conjunto qualquer podem ser conjuntos de outros elementos. Sendo assim, o conjunto $\boldsymbol{B} = \{\emptyset\}$ não é um conjunto vazio. É um conjunto com um único elemento, este elemento é um conjunto vazio. Observe que ao listarmos os elementos de um conjunto cada elemento aparece uma única vez e que é permitido listar os elementos do conjunto na ordem que desejarmos sem que isso altere o conjunto ou a informação que ele contém. Em inglês, chamamos o conjunto de *set* e é esta nomenclatura que normalmente encontramos em textos técnicos da área da computação.

Existem, do ponto de vista da computação, dois conjuntos de símbolos muito

importantes: o *alfabeto*, doravante representado por $\boldsymbol{\Sigma}$**,** e a *string*. **O alfabeto é um conjunto finito de todos os símbolos possíveis em um determinado domínio do conhecimento**. Em um alfabeto cada símbolo possível aparece apenas uma vez e a ordem em que estes símbolos aparecem é importante para a definição do alfabeto. O código ASCII usado com frequência em computação e o Unicode, contendo mais de 100.000 caracteres de diversas linguagens naturais são exemplos de alfabetos AHO, LAM, *et al.*, 2007. Usando um pouco de formalidade podemos exemplificar da seguinte forma:

a) $\boldsymbol{\Sigma_1} = \{0, 1, 2, 3, 4, 5, 6, 7,8, 9, 0\}$, alfabeto utilizado para escrever por todos os números positivos inteiros;

b) $\boldsymbol{\Sigma_2} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e} \ldots \mathrm{x}, \mathrm{y}. \mathrm{z}\}$, alfabeto utilizado para escrever palavras na língua portuguesa, se desconsiderarmos os acentos e sinais especiais.

Cuidado!!! Do ponto de vista da computação o alfabeto da língua portuguesa, por exemplo, incluí as letras maiúsculas, minúsculas e todos os símbolos que usamos. Tais como pontuação, números e símbolos matemáticos.

Por outro lado, o conjunto dos números inteiros (I), não pode ser considerado um alfabeto por ser infinito. O outro conjunto de símbolos, importante para nosso estudo, é chamado de *string*[18]. Diferindo dos conjuntos algébricos, em uma *string* a ordem em que os elementos são listados é a sua característica mais importante. A *string "compilador"* é diferente da *string "compiladro"* enquanto o conjunto $\boldsymbol{C} = \{\mathrm{c}, \mathrm{o}, \mathrm{m}, \mathrm{p}, \mathrm{i}, \mathrm{l}, \mathrm{a}, \mathrm{d}, \mathrm{r}\}$ não é diferente do conjunto $\boldsymbol{D} = \{\mathrm{a}, \mathrm{i}, \mathrm{o}, \mathrm{m}, \mathrm{p}, \mathrm{l}, \mathrm{c}, \mathrm{d}, \mathrm{r}\}$.

[18] Em português podemos traduzir o termo *string* como cadeia. Contudo, o uso da palavra inglesa é mais difundido em textos de computação e desta forma, será adotado.

É possível relacionar estes dois conjuntos especiais, o alfabeto e a *string,* e assim perceber que a *string*, s, é um conjunto de símbolos de um determinado alfabeto, $\boldsymbol{\Sigma}$, previamente especificado. Ressalte-se que uma *string* vazia ainda é uma *string* e é representada pelo símbolo $\boldsymbol{\varepsilon}$ BERGMANN, 2010. Na linguagem de programação C, por exemplo, o símbolo *'\0'* não faz parte do alfabeto utilizado para definir a linguagem. Ou seja, este símbolo pode ser utilizado como marcador do fim de uma string já que não existe a possibilidade de que este símbolo seja confundido com qualquer outro símbolo utilizado na *string* HOLUB, 1990.

Finalmente podemos definir uma *string* como uma sequência de símbolos $\boldsymbol{s}$ de comprimento $\boldsymbol{n}$ de um alfabeto $\boldsymbol{\Sigma}$ representada pela função $\boldsymbol{s}: [\boldsymbol{n}] \rightarrow \boldsymbol{\Sigma}$ com domínio $[\boldsymbol{n}]$ e contradomínio $\boldsymbol{\Sigma}$. Desta forma podemos considerar a *string* $s = ccda$ do alfabeto $\boldsymbol{\Sigma} = \{a, b, c, d\}$, com $n = 4$ e a função $s: [4] \rightarrow \boldsymbol{\Sigma}$ definida por: $s_{(1)} = c; s_{(2)} = c; s_{(3)} = d$ *e* $s_{(4)} = a$. Seguindo esta definição, uma *string* vazia v, será definida pela função $v: [0] \rightarrow \boldsymbol{\Sigma}$**.** Podemos utilizar esta definição, para exemplificar uma *string* s, do alfabeto $\boldsymbol{\Sigma_t} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$. Neste caso, podemos, por exemplo, definir as *strings*: $abc, aabc, cabba,$ ... todas em relação ao alfabeto $\boldsymbol{\Sigma_t}$ de comprimentos $n = 3,\ n = 4 \wedge n = 5$, respectivamente.

Torna-se indispensável destacar que existe uma grande diferença entre uma *string* vazia, $\boldsymbol{\varepsilon}$**,** e uma *string* nula. Tomando, novamente, a linguagem C, como exemplo, poderíamos ter uma *string* contendo apenas o símbolo "'\0'", uma string com um único caractere. Neste caso, teríamos uma *string* vazia. Como este símbolo não existe no alfabeto da linguagem C, essa *string* não contém caractere algum. Já a *string* nula consiste em um ponteiro do tipo NULL. Um ponteiro que aponta para coisa nenhuma, um endereço de memória não alocado, não existe dentro do universo do programa. Ou, se preferir, pode-se dizer que a *string vazia* é um *array* contendo apenas o símbolo '\0' já a *string* nula não possui nenhum *array* associado a ele.

Agora é possível **definir uma linguagem formal, $\boldsymbol{L}$, como sendo um**

conjunto de *strings* de um alfabeto qualquer. Contudo, definir uma linguagem a partir de suas *strings* não é uma tarefa trivial. Tome, como exemplo, a língua portuguesa. Quantas *strings* diferentes existem em português? Qual a regra de formação de *strings* usada em português? Ao definir uma linguagem de programação, e o respectivo analisador léxico, estas perguntas devem ter respostas simples, diretas e exatas. Uma forma de responder estas perguntas, ou criar uma linguagem que não caia na armadilha da complexidade, está no uso de máquina de estados finitos, ou autômatos finitos, e expressões regulares. Atendendo o **Teorema de Kleene** citado anteriormente.

4.1 Máquina de Estados Finitos

O estudo das máquinas de estados é chamado de **Teoria dos Autômatos**, por que autômato é outra palavra para máquina MOGENSEN, 2010, E consiste no estudo das regras de formação e comportamento de tais máquinas do ponto de vista da matemática e da lógica. **Definimos uma máquina de estados finitos como sendo um conjunto finito de estados submetidos a um número finito de transições entre estes estados** MOGENSEN, 2010. Cada estado sendo um momento específico no tempo onde as condições internas da máquina estão estáveis, enquanto a transição é uma função que, recebendo dois argumentos, um estado e um símbolo de entrada, resulta em outro estado. Neste livro, serão utilizadas, fundamentalmente, as **Máquinas de Mealy**[19], máquinas cuja função de transição é relacionada apenas com o estado presente da máquina e com o símbolo da entrada.

No estado inicial a máquina recebe uma *string* de símbolos do alfabeto

[19] George H. Mealy, matemático norte americano (1927-2010) **Fonte bibliográfica inválida especificada.**

escolhido. Todas as vezes que um símbolo é lido a máquina prossegue para um novo estado, indicado pela função de transição. Este novo estado será determinado pela função de transição de acordo com o estado anterior e o símbolo lido. Atendendo as características da **Máquina de Mealy**. Ao terminar a leitura de todos os símbolos de entrada a máquina ficará em um estado que indica o reconhecimento de uma sequência de símbolos (*accepting state*) ou não (*non-accepting state*). Se a máquina terminar o processo em *accepting state*, dizemos que a *string* foi aceita, ou reconhecida, caso contrário a *string* foi rejeitada. O conjunto de todas as *strings* que serão aceitas pela máquina formam uma linguagem e é dessa forma que a máquina de estados finitos consegue definir uma linguagem.

Existem duas formas interessantes de se representar uma máquina de estados: a forma gráfica, ou diagrama de estados, e a forma tabular. Na forma gráfica cada estado é representado por um círculo e as funções de transição são representadas por setas etiquetadas por um símbolo que representa a entrada desta função. O estado inicial é representado por uma seta solta, sem nenhum estado em sua origem. Em alguns textos técnicos, principalmente em inglês, o estado inicial é marcado com uma seta etiquetada com a palavra *start*. É preciso observar que ao longo do funcionamento da máquina, os estados que estiverem em modo de entrada, *accepting state*, são representados por dois círculos concêntricos. A Figura 2 apresenta um exemplo da representação gráfica de uma máquina de estados finitos.

A primeira vista a leitora vai achar que este artefato matemático é muito complicado mas, de forma simples, podemos sintetizar uma máquina de estados em apenas cinco conceitos:

1. **estados:** pontos estáveis no processo de computação da máquina, representados por círculos;
2. **estado inicial:** estado onde a máquina inicia o processamento, indicado por um círculo com uma seta apontada para ele;
3. **estados intermediários:** todos os estados intermediários têm, no mínimo, duas setas. Uma apontada para o círculo que o representa e outra seta deixando este círculo;
4. **estado final:** se a *string* for corretamente processada, a máquina deverá assumir este estado, em geral indicando que está pronta para processar outra string (*accepting state*). O estado final é representado por dois círculos concêntricos.
5. **transições:** a transição entre um estado e outro acontece quando o símbolo de entrada é o símbolo esperado por um determinado estado. Em resposta a uma transição a máquina pode ficar no mesmo estado, ou passar para o estado seguinte. As transições são representadas por uma seta com origem em um estado e destino no estado seguinte. Se a

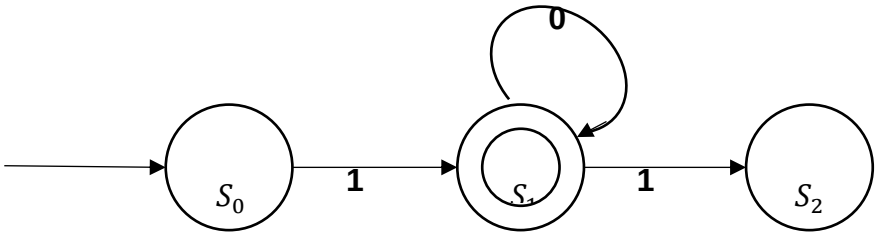


Figura 8 - Exemplo de máquina de estados finita representada por um diagrama de estados

máquina permanecer no mesmo estado, a transição será representada por uma seta que deixa o estado e aponta para ele mesmo.

Podemos usar como exemplo uma máquina de estados muito comum em circuitos, e ambientes, de comunicação de dados uma máquina de detecção de paridade. Este tipo de máquina é capaz de detectar a paridade de um conjunto de bits, fornecidos como entrada de forma sequêncial.

A definição de uma máquina de detecção de paridade começa pela especificação do alfabeto, neste caso, como estamos falando de bits vamos definir o alfabeto $\boldsymbol{\Sigma_{bits}} = \{0,1\}$; e termina na especificação do diagrama de estados que pode ser visto na Figura 9.

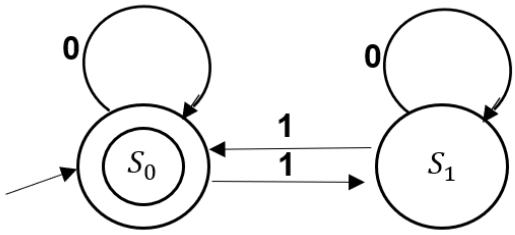


Figura 9 - Máquina de estados de um detector de paridade (números pares de uns)

Observe que todos os estados e transições desta máquina estão definidos e que ela só pode receber informações pelo estado S_0 e que só estará em *accepting state* se o número de uns na *string* de entrada for par, incluindo a *string* nula HOLUB, 1990.

Um zero aplicado ao estado S_0 manterá a máquina em *accepting state* indicando zero uns. Ainda que zero não seja par, este conceito é usado em detectores de paridade. O primeiro um da sequência de entrada leva a máquina para o estado S_1. A partir deste momento, qualquer quantidade de zeros manterá a máquina em S_1 e fora do *accepting state*. Ainda em S_1, qualquer um levará a máquina para o estado S_1 e para *accepting state*. Ou seja, esta máquina reconhece qualquer número par de uns.

A máquina apresentada na Figura 9 está perfeitamente determinada. Para

cada estado existe exatamente uma seta indicando a transição para cada símbolo de entrada possível, definindo o estado resultante do processamento deste símbolo.

Como nosso alfabeto possui apenas dois símbolos, e cada um destes símbolos pode atingir um estado, cada estado tem exatamente duas setas de saída. As máquinas que atendem esta formalidade, onde todas as transições são definidas e conhecidas são chamadas de determinísticas. Se existir uma única transição entre quaisquer dois estados que não possa ser relacionada a um dos símbolos do alfabeto, esta transição será denominada de ϵ e teremos uma máquina de estados finitos não determinística.

Outro exemplo do funcionamento das máquinas de estado finitas e determinísticas pode ser obtido do estudo da máquina da Figura 3, capaz de reconhecer todos os números binários divisíveis por dois e não divisíveis por quatro. Estamos falando dos números binários que terminam 10.

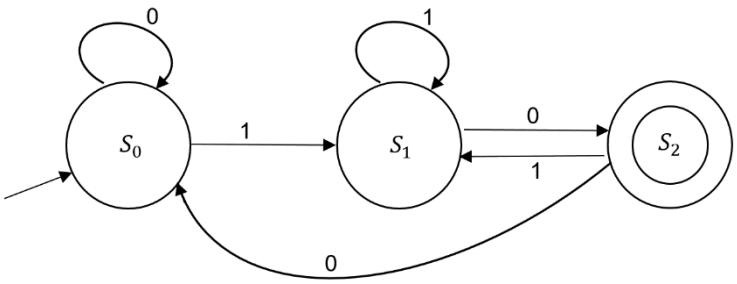


Figura 10 - Exemplo de Máquina de Estados capaz de selecionar número binários divisíveis por dois e não por quatro ZANDER, 2012.

Uma análise detalhada dos estados e transições da máquina representada pela Figura 4 permite perceber que: uma *string* que comece com qualquer números de zeros irá manter a máquina no estado S_0; o primeiro um da *string* de entrada colocará a máquina no estado S_1 neste ponto, todos os próximos uns manterão a máquina no estado S_1; O primeiro zero colocará a máquina no estado S_2, em *accepting state*; Neste estado um zero levará ao estado S_0 caso contrário a máquina irá para o estado S_1. Ou seja, a única forma de atingir o *accepting state* é receber um em S_1 e zero em S_2 ZANDER, 2012.

Usando um pouco de formalidade podemos definir uma máquina de estados finita não determinística como sendo um conjunto de estados $\boldsymbol{S}$ onde um destes estados $s_0 \in \boldsymbol{S}$, é chamado de estado inicial e existe um subconjunto s tal que: $s \subseteq \boldsymbol{Q}$ de estados que não são iniciais. Adicionalmente temos um conjunto de transições $\boldsymbol{\delta}$ onde cada transição δ_0 conecta um par de estados, s_1 e s_2, e é nomeada com um dos símbolos do alfabeto $\boldsymbol{\Sigma}$ ou por ϵ. Dessa forma, podemos definir uma transição, δ_0, com origem no estado s_1, devida ao símbolo c, como: $s_0^c\delta_0$.

Existem duas diferenças formais muito importantes entre uma máquina de estados finitos não determinística e a determinística, a inexistência de transições $\boldsymbol{\epsilon}$ é uma delas. A outra é a inexistência de transições múltiplas de saída para um mesmo símbolo. Ou seja, **em uma máquina de estados finitos determinística cada estado terá uma, e somente uma, transição para cada símbolo do alfabeto escolhido.**

Outra representação interessante das máquinas de estados finitos é a tabela. Neste caso, cada linha é nomeada como um dos estados possíveis, cada coluna representa um dos símbolos do alfabeto e a interseção entre essas linhas e colunas representa o estado resultante da transição. Assim, nossa máquina de paridade pode ser representada por:

	0	**1**
$\boldsymbol{S_0}$	S_0	S_1
$\boldsymbol{S_1}$	S_1	S_0

Tabela 2 - Tabela representando uma máquina de paridade

4.2 Expressões Regulares

Além das máquinas de estados finitos, podemos utilizar as expressões regulares, indicadas pelas letras r e t para representar as *strings* de uma determinada linguagem L derivada de um alfabeto finito $\boldsymbol{\Sigma}$. As expressões regulares constituem uma notação algébrica específica para a representar o processamento de padrões de símbolos AHO, LAM, *et al.*, 2007. Muitos autores, este inclusive,

consideram as expressões regulares como uma linguagem de programação específica. Entretanto, o importante neste momento é perceber que o fundamental para a criação de expressões regulares está no conjunto de símbolos de uma dada linguagem. Ou, se preferir, o alfabeto é o coração das expressões regulares. Dado um alfabeto $\boldsymbol{\Sigma}$ qualquer, iremos descrever as *strings,* $r, s\ ou\ t,$ de uma linguagem L, por meio de um conjunto de expressões lógicas e matemáticas que, deste momento em diante chamaremos de expressões regulares. Contudo, é necessário destacar que, para que nossa álgebra atendas as necessidades propostas é necessário considerar que os símbolos $\boldsymbol{\varepsilon, \emptyset, |, (,)\ e\ *}$ não fazem parte do alfabeto $\boldsymbol{\Sigma}$ escolhido.

Vejamos um exemplo, simples de expressões regulares considerando o alfabeto $\boldsymbol{\Sigma}$ composto de todas as letras latinas em minúsculo, podemos ter a expressão regular representada por a. Trata-se da representação de todos as *strings* que atendam ao conjunto $\{a\}$. Considerando que esta expressão regular seja relacionada a um conjunto de *strings* da linguagem L de alfabeto $\boldsymbol{\Sigma} = \{a, b, c, \dots, x, y, z\}$ a expressão regular representa o conjunto de todas as *strings* que possuem apenas um caractere, o caractere a . Tomando o Teorema de Kleene, é possível representar esta expressão regular na forma de uma máquina de estados finitos a qual pode ser vista na Figura 11.

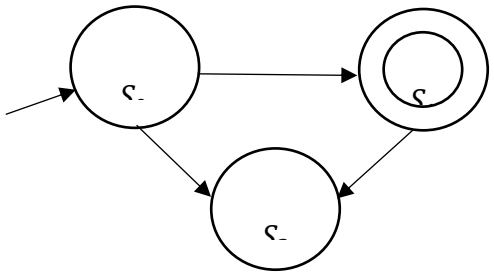


Figura 11 - Máquina de estados finitos representando a expressão regular a.

Vamos considerar agora a expressão regular a^*. Esta expressão regular representa o conjunto de todas as *strings* que possuem apenas zero ou mais caracteres a, observe a *string vazia*, faz parte deste conjunto. A representação desta expressão regular na forma de máquinas de estados finitos pode ser visto na Figura 12:

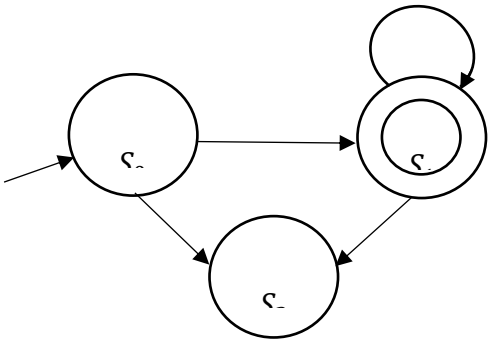


Figura 12 - Máquina de estados finitos equivalente a expressão regular a^*

Para entendermos o potencial das expressões regulares precisamos definir três operações importantes: a união, a concatenação e uma operação especial chamada de *Kleene star* ou *closure*. Estas operações definem todas operações necessárias para a criação de expressões regulares.

1.1.5 União

Uma linguagem é um conjunto a união é definida exatamente da mesma forma que na teoria dos conjuntos.

$$\boldsymbol{L_1} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}\} \ \wedge \ \boldsymbol{L_2} = \{\mathrm{ea}, \mathrm{af}\}$$

$$\boldsymbol{L_1} \cup \boldsymbol{L_2} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}, \mathrm{ea}, \mathrm{af}\}$$

Observe que em alguns textos técnicos, principalmente em inglês o símbolo algébrico da união $(\cup)$ é substituído pelo símbolo da soma $(+)$. Notadamente por que essa substituição facilita o entendimento das expressões regulares, como veremos nos próximos tópicos. Sendo assim, teremos:

$$\boldsymbol{L_1} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}\} \ \wedge \ \boldsymbol{L_2} = \{\mathrm{ea}, \mathrm{af}\}$$

$$\boldsymbol{L_1} + \boldsymbol{L_2} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}, \mathrm{ea}, \mathrm{af}\}$$

É muito importante observar que a união de um conjunto com um conjunto vazio é o próprio conjunto. Desta forma:

$$\boldsymbol{L_1} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}\} \ \wedge \ \boldsymbol{L_2} = \emptyset$$

$$\boldsymbol{L_1} \cup \boldsymbol{L_2} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{eg}, \mathrm{hf}\}$$

Podemos representar um conjunto vazio $\boldsymbol{A}$ de duas formas: $\boldsymbol{A} = \emptyset$ ou $\boldsymbol{A} = \{\}$. A operação de união é tão poderosa que em alguns textos de linguística computacional dizemos que a linguagem L_1 e autocontida na operação de união.

1.1.6 Concatenação

Do ponto de vista intuitivo, o resultado da concatenação de duas *strings* é uma *string* formada pelas duas *strings* originais em sequência. Assim, a concatenação de “um belo dia” e “ começa cedo” será “um belo dia começa cedo”. Observe que, para que a operação de concatenação seja perfeita é preciso que as *strings* tenham os valores desejados. Se tirarmos o espaço que existe no começo de “ começa cedo” o resultado seria: “um belo diacomeça cedo”. A concatenação de uma *string*, com uma *string* é a própria *string*.

Considerando um pouco de formalidade matemática, devemos ressaltar que a concatenação é uma operação a ser realizada entre duas linguagens L_1 *e* L_2. Neste caso definimos a concatenação L entre L_1 e L_2 como sendo:

$$L = L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

No caso mais comum de concatenação, visto diariamente na construção de programas consideramos as linguagens L_1 e L_2 como sendo conjuntos de uma única *string* sendo assim, podemos extrapolar a definição de concatenação para este caso específico. Dadas duas *strings* representadas pelas funções $s_1 : [m] \rightarrow \boldsymbol{\Sigma}$

e $s_2: [n] \rightarrow \boldsymbol{\Sigma}$ é possível definir a concatenação como sendo a função $s_1 \cdot s_2: [m+n] \rightarrow \boldsymbol{\Sigma}$ de comprimento $l = m + n$, para todos os i, se e somente se:

$$s_{1(i)} \cdot s_{2(i)} = \begin{cases} x_{(i)}, se\ i \leq m \\ y_{(i-m)}, se\ i > m \end{cases}$$

Desta forma, se tivermos $s_1 =$ "cbba" e $s_2 =$ "ac" teremos $s_1 \cdot s_2 =$ "$cbbaac$" ou, em formato de tabelas e correndo todos os valores de i em cada *string* RANGEL, 2013:

$\boldsymbol{i}$	$\boldsymbol{s_{1(i)}}$
1	c
2	b
3	b
4	a

$\boldsymbol{i}$	$\boldsymbol{b_{2(i)}}$
1	a
2	c

$\boldsymbol{i}$	$\boldsymbol{(s_1 \cdot s_2)_{(i)}}$
1	c
2	b
3	b
4	a
5	a
6	c

Tabela 3 - Concatenação de strings método formal de concatenação

Podemos ainda usar uma notação algébrica um pouco mais refinada para indicar a concatenação de duas *strings*. Se assim for, podemos dizer que dadas as *strings* s_1 *e* s_2 a concatenação será um conjunto representado por $\boldsymbol{\Sigma}^*$ que indica um conjunto finito de *strings* relativas ao alfabeto $\boldsymbol{\Sigma}$**,** de tal forma que:

$$s_1 \cdot s_2 = \boldsymbol{\Sigma}^*$$

Assim podemos exemplificar considerando que $s_1 = ab$, $s_2 = ra$, $s_3 = cad$ termos: $s_2 \cdot s_1 = raab$, $s_1 \cdot s_1 = abab$ e $s_3 \cdot s_2 = cadra$. Importante lembrar que a concatenação pode ser realizada entre várias *strings* PITTS, 2013, mas sempre dois a dois dessa forma:

$$s_1 \cdot s_2 \cdot s_3 \cdot s_1 \cdot s_2 = (((s_1 \cdot s_2) \cdot s_3) \cdot s_1) \cdot s_2 = abracadabra$$

Dessa forma, podemos considerar que o alfabeto $\boldsymbol{\Sigma}$ é na verdade um subconjunto de $\boldsymbol{\Sigma}^*$ e que $\boldsymbol{\Sigma}^*$ nunca será um conjunto vazio já que, independente do alfabeto, $\boldsymbol{\Sigma}^*$ conterá, no mínimo a *string vazia*, $\boldsymbol{\varepsilon}$**.** Note que para qualquer $s_1, s_2, s_3 \in \Sigma^*$ teremos:

$$s_1 \cdot \boldsymbol{\varepsilon} = s_1 = \boldsymbol{\varepsilon} \cdot s_1$$

$$(s_1 \cdot s_2) \cdot s_3 = s_1 s_2 s_3 = s_1 \cdot (s_2 \cdot s_3)$$

Além disso, é preciso destacar que o comprimento, *length*, da *string* concatenada é a soma dos comprimentos das *strings*, ou seja:

$$length(s_1 \cdot s_2) = length(s_1) + length(s_2)$$

Exemplos (PITTS, 2013):

(a) Se $\boldsymbol{\Sigma} = \{a\}$ então $\Sigma^* = \{\boldsymbol{\varepsilon}, a, aa, aaa, \ldots\}$;
(b) Se $\boldsymbol{\Sigma} = \{a, b\}$ então $\Sigma^* = \{\boldsymbol{\varepsilon}, a, aa, b, ab, ba, bb, aaa, aab, \ldots\}$;
(c) Se $\boldsymbol{\Sigma} = \emptyset$ então $\Sigma^* = \{\boldsymbol{\varepsilon}\}$.

1.1.7 Kleene star

A operação *Kleene star*, frequentemente chamada de *closure*, é uma operação unária representada por um $*$ e similar ao processo de exponenciação de tal forma que *Kleene de* L, L^* é a linguagem L repetida zero ou mais vezes. Por exemplo, se L contém apenas um símbolo, L^* será este símbolo repetido zero ou mais vezes. Sendo assim, se tomarmos a linguagem L podemos definir:

(a) $L^0 = \boldsymbol{\varepsilon}$;
(b) $L^1 = L$;
(c) $L^2 = L \cdot L$;
(d) $L^n = L \cdot L^{n-1}$;
(e) $L^* = L^0 + L^1 + L^2 + L^3 + \cdots$.

Observe que aplicada sobre um conjunto vazio o resultado da *closure* será uma *string vazia.*

$$L = \emptyset \therefore \emptyset^* = \boldsymbol{\varepsilon}$$

Desta forma podemos definir a operação *Kleene star* de forma algébrica por:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

O que permite a avaliação de um exemplo: se $L_1 = \{a\}$ e $L_2 = \{b\}$ então:

$$L_1^* \cdot L_2 = \{b, ab, aab, aaab \ldots\}$$

Onde é possível perceber a propagação *Kleene* do símbolo a de zero ao infinito sendo concatenado, repetidamente ao símbolo b que não está sujeito a operação *Kleene star*.

Para tornar a notação mais simples, usaremos algumas simplificações então se o símbolo x pertence ao alfabeto escolhido então $x = \{x\}$ ou seja, o caractere x irá representar o conjunto de *strings* contendo apenas a *string* "x" o que irá tornar mais inteligíveis as expressões regulares que vamos estudar. Podemos exemplificar a *Kleene star*, considerando a linguagem $\mathrm{L} = \{0,1\}$. Dessa forma teremos:

$$L^0 = \{\boldsymbol{\varepsilon}\}$$
$$L^1 = \{0,1\}$$
$$L^2 = L \cdot L^1 = \{00,01,10,11\}$$
$$L^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$
$$l^* = \{\boldsymbol{\varepsilon}, 000, 001, 010, 011, 100, 101, 110, 111, 1000 \ldots \ \}$$

Uma expressão regular é uma expressão algébrica que envolve uma ou mais linguagens. Vimos três as três operações que usaremos para definir as expressões regulares uma unária, a *Kleene star*, e duas binárias a concatenação e a união. A

procedência será definida pelo uso de parênteses. Se não forem utilizados parênteses para definir a precedência entre as operações, *Kleene star* irá tomar a precedência sobre a concatenação e essa terá precedência sobre a união. Sendo assim:

$$a|ab^* = \left(a\middle|(a \cdot b^*)\right)$$

A Tabela 3 apresenta algumas expressões regulares básicas para fixação dos conceitos.

Regex	Álgebra	Descrição
$\boldsymbol{x}$	$\{“x”\}$	Um conjunto contendo apenas a *string* x
$\boldsymbol{\varepsilon}$	$\{“”\}$	Um conjunto contendo nenhuma string, a string vazia
$\boldsymbol{s}\|\boldsymbol{r}$	$L_{(s)} \cup L_{(r)}$	A união de strings de ambas as linguagens
$\boldsymbol{s} + \boldsymbol{r}$	$L_s \cup L_{(r)}$	A união de strings de ambas as linguagens
$\boldsymbol{s} \cdot \boldsymbol{r} = \boldsymbol{sr}$	$\{xy \mid x \in s, y \in r\}$	Strings construídas com a concatenação das strings em s_1 com as strings em s_2
$\boldsymbol{s}^*$	$\boldsymbol{\varepsilon} \cup \{xy \mid x \in L_s, y \in L_{s^*}\}$	Cada string na linguagem é a concatenação de um número qualquer de strings da linguagem de s

Tabela 4 - Expressões regulares básicas

No último caso, a operação s^* (*s star*) é definida de forma recursiva e consiste da *string vazia*, $\boldsymbol{\varepsilon}$ mais aquilo que pode ser obtido concatenando as linguagens L_s e L_{s^*}. Isto é equivalente a dizer que L_{s^*} consiste de *strings* que podem ser obtidas pela concatenação de zero ou mais *strings* de L_s. Para ficar claro, suponha que $L_s = \{a, b\}$. Então $L_{s^*} = \{\boldsymbol{\varepsilon}, a, b, aa, bb, ab, ba, \ldots\}$, ou seja, qualquer *string* composta unicamente de a e b incluindo a *string vazia*. Ainda analisando a Tabela 3 é possível concluir que existe um símbolo qualquer $x \mid x \in \boldsymbol{\Sigma}$ então, x é uma expressão regular. Da mesma forma a $\boldsymbol{\varepsilon}$ também é uma expressão regular. Especificamente podemos dizer que, dada uma expressão regular r, dizemos que**:**

Condições de concordância (reconhecimento)
r **reconhece** $a \in \Sigma$ **se e somente se** $r = a$
r **reconhece** $\varepsilon \in \Sigma$ **se e somente se** $r = \varepsilon$
Nenhuma *string* será reconhecida por $\emptyset$
r **reconhece** $a\|b$ **se e somente se** $r = a$ **ou** $r = b$

Tabela 4 - Condições de reconhecimento entre *strings* e expressões regulares

Podemos utilizar o conhecimento já adquirido para escrever uma expressão regular capaz de reconhecer qualquer número positivo do conjunto dos números inteiros desde possamos nos lembrar que os números deste conjunto são representados por um ou mais dos seguintes algarismos: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Sendo assim, podemos escrever a seguinte expressão regular:

Eq.1 $$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

A Eq. 1 mostra a concatenação entre um conjunto e seu *Kleene* já que s^* indica zero ocorrências ou mais de cada um dos algarismo, se consideramos que o conjunto vazio está implícito na operação *Kleene*, esta expressão regular representa, entre outros, os seguintes números: $0, 1, 10, 1000, 1234000, etc$. podendo ser utilizada para representar qualquer um deles MOGENSEN, 2010. Podemos excluir outros símbolos do alfabeto para escrever uma notação mais amigável. Podemos excluir os símbolos $[,], -$ e usar estes símbolos para definir um intervalo. Se fizermos isso, a Eq. 1 poderia ser representada por:

$$[0-9][0-9]^*$$

Ou poderíamos representar um conjunto de símbolos finitos qualquer, por exemplo: $[xy12]$ representa a expressão $(x|y|1|2)$, ou ainda podemos representar todas as combinações de letras do alfabeto português, menos os acentos e caracteres especiais pela concatenação de dois intervalos: [a-zA_Z]. Para simplificar ainda mais a notação podemos também excluir os símbolos $+, ?$ e determinar que s^+ representa um ou mais e que $s? = s|\varepsilon$ que representa a união de

uma *string* com a *string vazia*[20]. Se fizermos isso, a representação de um número inteiro positivo qualquer pode ser simplificada para:

$$[0-9]^+$$

A representação de uma ou mais ocorrência s^+, chamada de *positive Kleene*, pode ser representada de forma algébrica por:

$$L^* = \bigcup_{i=1}^{\infty} L^i$$

Neste ponto estamos prontos para definir expressões regulares algébrica e formalmente por seus quatro axiomas:

1. $\emptyset$ é uma expressão regular que indica um conjunto vazio;
2. $\boldsymbol{\varepsilon}$ é uma expressão regular que indica um conjunto contendo uma *string vazia;*
3. para cada símbolo $a \in \boldsymbol{\Sigma}$ existe uma expressão regular a que indica o conjunto $\{a\}$.
4. se r e s são expressões regulares referentes as linguagens R e S então, (sr), $(s|r)$ *e* s^* são expressões regulares que indicam as linguagens $S \cdot R$, SUR e S^* respectivamente.

E definir algumas propriedades derivadas destes axiomas e do que vimos sobre as operações união, concatenação e *Kleene star*:

	Propriedades
1	$\emptyset s = s\emptyset = s$

[20] Sendo assim, a expressão $b?\,a^*$ representa qualquer *string* com zero ou mais a podendo, ou não começar com o símbolo b.

2	$\boldsymbol{\varepsilon} s = s\boldsymbol{\varepsilon} = s$
3	$s\|\emptyset = s + \emptyset = \emptyset + s = s$
4	$s\|s = s + s = s$
5	$r + s = s + r$
6	$r(s + t) = rs + rt$
7	$(r + s)t = rt + st$
8	$r(st) = (rs)t$
9	$\emptyset^* = \boldsymbol{\varepsilon}$
10	$\boldsymbol{\varepsilon}^* = \boldsymbol{\varepsilon}$
11	$(\boldsymbol{\varepsilon} + r)^+ = r^*$
12	$(\boldsymbol{\varepsilon} + r)^* = r^*$
13	$r^*(\boldsymbol{\varepsilon} + r) = (\boldsymbol{\varepsilon} + r)r^* = r^*$
14	$r^*s + s = r^*s$
15	$r(sr)^* = (rs)^*r$
16	$(r + s)^* = (r^*s)^*r^* = (s^*r)^*s^*$

Tabela 5 - Propriedades das operações União, Concatenação e Kleene

Agora é possível expandir estes conceitos para a representação de algumas das classes de símbolos mais utilizadas em linguagens formais. Notadamente no universo das linguagens de programação. E tentar explicar as expressões regulares que podem ser utilizadas para seu reconhecimento MOGENSEN, 2010. Sendo

assim:

(a) **palavras-chaves, ou *keywords:*** devem ser reconhecidas por expressões regulares exatamente iguais as palavras chaves. Em C, por exemplo, a palavra-chave *if* será reconhecida pela expressão regular *if*;

(b) **nomes de variáveis:** neste caso, depende das regras de formação de cada linguagem. Em C, por exemplo, podemos escrever qualquer nome de variável desde que ele comece com uma letra, contenha números e o caractere "_" e não utilize nenhum caractere excluído do alfabeto. Estes nomes de variáveis podem ser reconhecidos por: $[a-zA-z\_][a-zA-Z0-9\_]^*$.

(c) **inteiros:** para a representação de inteiros positivos podemos usar $[0-9]^+$ porém, na linguagem C, por exemplo, precisamos representar inteiros positivos e negativos então a expressão mais correta seria: $[=|+-]?\,[0-9]^+$, indicando qualquer número inteiro podendo ter, ou não os sinais + e – na frente[21].

(d) **números de ponto flutuante:** números de ponto flutuante, tem uma quantidade indefinida de algarismos antes e depois da vírgula (ponto nas linguagens de programação), podem ter os símbolos de $+ e -$ e ainda podem estar em notação especial representados por uma mantissa e um expoente. Talvez, uma forma de representar estes números seja a expressão regular: $[+-]?\,(([0-9]^+(.[0-9]^+)?\,[0-9]^+)([eE][+-]?\,[0-9]^+)?)$ MOGENSEN, 2010.

(e) ***Strings* constantes:** neste caso, também precisamos entender as regras de formação da linguagem de programação. Na linguagem C

[21] De fato, par que isso fosse algebricamente correto, precisaríamos de um símbolo para indicar que entre [,] os símbolos + e – fazem parte do alfabeto utilizado.

podemos representar as *strings* por: “([a-zA-z0-9]\[a-zA-Z])*” MOGENSEN, 2010.

Sem dúvida, o uso das expressões regulares foge aos conceitos da álgebra e, hoje, no domínio das linguagens de programação, possui um lugar de destaque, sendo consideradas, por muitos, uma linguagem em si. Por exemplo, o padrão Posix, utilizado em máquina Unix, Linux e Windows, especifica algumas outras regras para a formação de expressões regulares. Você pode, por exemplo, especificar o número de repetições de um determinado símbolo usando {}. Ou seja, se deseja reconhecer *strings* formadas por 10 repetições dos símbolos ba você pode escrever a expressão regular $ba\{10\}$, pode ainda usar o caractere \ para excluir um determinado símbolo do alfabeto então, por exemplo, o reconhecimento de um número de CPF pode ser feito por: $[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\}\-[0-9]\{2\}$. Ou, se preferir podemos escrever $[A-Z]\{3\}-[0-9]\{4\}$ para reconhecer placas de veículos. Para maiores informações sobre o padrão Posix para expressões regulares veja o Apêndice A.

4.3 Lexemas e tokens

Já vimos como podemos utilizar expressões regulares para localizar *strings* de uma determinada linguagem. Em geral, utilizamos estas técnicas para localizar estas *strings* em um texto já que a maior parte dos compiladores recebe um texto contendo o código-fonte relativo a uma determinada linguagem de programação. Podemos considerar o código, texto completo, do programa, como sendo uma única *string* composto de *strings* que denominaremos lexemas, e, a função do analisador léxico será distinguir e identificar estes lexemas e os classificar em *tokens*.

Considerando a linguagem C/C++, o analisador léxico deverá ser capaz de classificar lexemas diferentes em diversas classes de *tokens*, descritas por sua

própria máquina de estados finito, ou expressão regular. Entre as classes diferentes da linguagem C/C++ podemos destacar:

(a) **palavras chaves, *keyword*:** são palavras que possuem um significado específico na linguagem, geralmente definem um comando ou processo específico. Neste caso, estas *keywords* fazem parte da linguagem, mas não podem ser utilizadas pelo programador como identificadores (variáveis ou constantes);

(b) **identificadores, *identifiers:*** palavras que podem ser utilizadas para identificação de variáveis e constantes. Em geral, as linguagens de programação apresentam apenas uma regra de formação destas palavras. Em C, por exemplo, a regra determina que os nomes de variáveis, ou constantes, devem começar com uma letra (maiúscula ou minúscula) e podem conter algarismo e o símbolo "_";

(c) **operadores, *operators*:** símbolos utilizados para a realização de operações lógicas, aritmética ou entre caracteres, podem ser constituídos de um ou mais caracteres. Então, por exemplo, o $!=$, símbolo que representa diferente na linguagem C, é formado por dois caracteres ASCII $\{!, =\}$;

(d) **valores numéricos:** valores representado números inteiros, ou de ponto flutuante, positivos ou negativos, nas diversas notações possíveis e, em muitas linguagens, em várias bases. Em C, por exemplo, é possível incluir números em decimal, hexadecimal e octal, mas não em binário;

(e) **caracteres especiais:** geralmente usados para limitar, comandos, blocos ou caracteres. Em C, entre outros podemos citar: $\{, \}, \; „ \; ., (,), etc;$

(f) **caracteres constantes:** caracteres especiais, não gráficos, utilizados para funções especiais dentro de *strings* e declarações. Que podem ser vistos na Tabela.

Sequência de Escape	Descrição
\a	Alerta audível – campainha
\b	Retrocesso
\f	Alimentador de formulário
\n	Nova Linha
\r	Retorno de carro
\t	Tabulação horizontal
\v	Tabulação vertical
****	Contra barra
“	Aspas duplas
‘	Aspas simples
?	Interrogação
	null

(g) **comentários:** geralmente são identificados em na fase de análise léxica mas são ignorados pelo analisador léxico. Ou seja, precisam ser identificados, mas não são transformados em *tokens* e não são enviados para a próxima fase;

(h) **espaços, tabulações e novas linhas:** em geral estes caracteres são ignorados e não são transformados em *tokens* nem passados a próxima fase do processo de compilação.

Esta lista não representa todas as possiblidades de classe utilizadas em linguagens de programação. Não é sequer a lista completa da linguagem C/C++ e está longe de ser a única solução possível. Existem linguagens como menos

classes que, em geral, apresentam maiores problemas para o compilador. Por exemplo a linguagem PL/1 HOLUB, 1990 não limita o uso de palavras chaves e, desta forma, permite o uso de qualquer *string* em qualquer parte do código permitindo a criação de declarações como:

$$if\ then\ then\ then = else\ else\ else = then;$$

Que agregam um nível de complexidade desnecessário ao processo de compilação. Tomemos agora como exemplo o fragmento de código a seguir, escrito na linguagem de programação C/C++:

```
while ((ch = getchar()) != '\n' && ch != EOF);
```

Podemos, manualmente, tentar classificar cada um destes lexemas segundo as sete classes apresentadas anteriormente. O resultado desta tentativa pode ser visto na Tabela 5.

Lexema	Classe
While	Palavra Reservada
(	Caractere Especial
(	Caractere Especial
ch	Identificador
=	Operador
getchar	Identificador
(	Caractere Especial
)	Caractere Especial
)	Caractere Especial
!=	Operador
'	Caractere Especial
\ n	Caractere Constante
'	Caractere Especial
&&	Operador
ch	Identificador

!=	Operador
EOF	Identificador
)	Caractere Especial
;	Caractere Especial

Tabela 6 - Análise léxica manual, exemplo da linguagem C

O analisador léxico não analisa todo o código de uma vez, de fato, ele separa os lexemas e tenta classificar cada um deles separadamente. Se optarmos por usar expressões regulares para esta classificação o trabalho fica simplificado. Suponha que tenhamos $r_1, r_2, r_3, \ldots r_n$ como sendo um conjunto de expressões regulares capazes de identificar cada um dos lexemas de uma determinada linguagem L neste caso, lembrando das operações entre expressões regulares a expressão regular $R = r_1 + r_2 + r_3 + \cdots r_n$ será capaz de identificar todos os lexemas desta linguagem e permitir a geração dos *tokens*. O que é também é possível com máquinas de estado finito determinísticas. Tomemos o exemplo dado por Mogensen 2010 para a criação de uma máquina de estados finitos determinística capaz de identificar a palavra-chave *if*, números inteiros e números de ponto flutuante que pode ser vista na Figura 7.

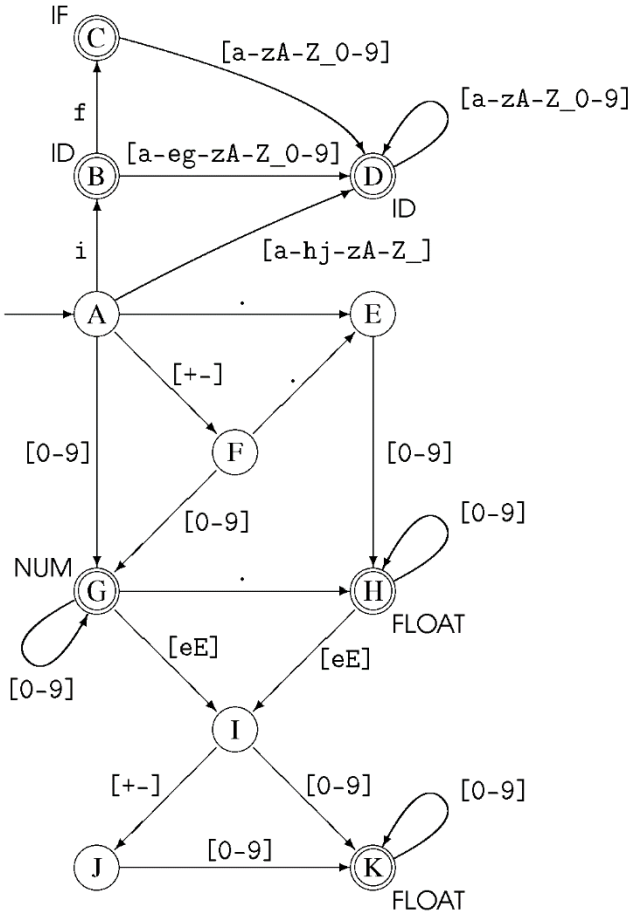


Figura 13 - Máquina de estados finitos para separação de lexemas MOGENSEN, 2010

Para cada lexema identificado deve ser gerada uma estrutura de dados para armazenar,

essencialmente, porém não exclusivamente, duas informações: um *token-name* e, opcionalmente, um *atribute-value*. De tal forma que teremos um par (*token-name*, *attribute-value*) que, em geral composto por dois valores inteiros, um que identifica a classe do *token, token-name* e outro que aponta para a tabela de símbolos onde está o valor referente a este *token* na linguagem de programação C/C++ poderia ser representado por:

```
typedef struct{
      int nome;
      int valor;
} Token;
```

Como, na maioria dos compiladores todo o processo depende da análise de um texto, o famoso código fonte, não é raro que esta estrutura tenha informações referentes a linha e coluna onde aquele determinado *token* foi localizado. Esta informação é útil, entre outras coisas, para a identificação de erros. Se assim for, poderíamos ter uma estrutura apenas para armazenar a posição dos *tokens* lidos e processados e alterar a estrutura de armazenamento do próprio *token* para acrescentar esta informação, como pode ser visto no fragmento de código a seguir:

```
typedef struct {
      int linha;
      int coluna;
} Position;
```

```
typedef struct {
      int nome;
      int valor;
      Position posicao;
} Token;
```

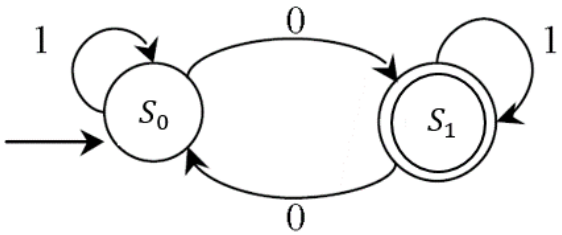
Durante o processo de análise léxica será construída uma tabela de símbolos contendo uma entrada única para cada identificador encontrado no código. Em geral é uma estrutura de dados que, armazena as informações

relevantes para o compilador. Contudo, veremos a tabela de símbolos com mais carinho em outro capítulo deste livro.

5 EXERCÍCIOS LINGUAGENS FORMAIS, MÁQUINAS DE ESTADO FINITO E EXPRESSÕES REGULARES.

1. Desenvolva uma máquina de estados capaz de identificar uma *string* contendo um número ímpar de zeros dado que $\boldsymbol{\Sigma} = \{0, 1\}$.

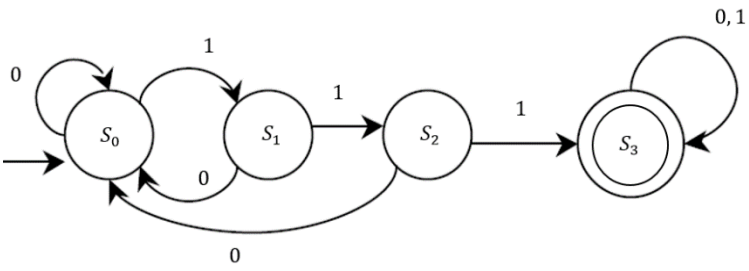
Solução:



	0	**1**
$\boldsymbol{S_0}$	S_1	S_0
$*\boldsymbol{S_1}$	S_0	S_1

2. Desenvolva uma máquina de estados capaz de identificar uma *string* contendo três uns consecutivos dado que $\boldsymbol{\Sigma} = \{0, 1\}$.

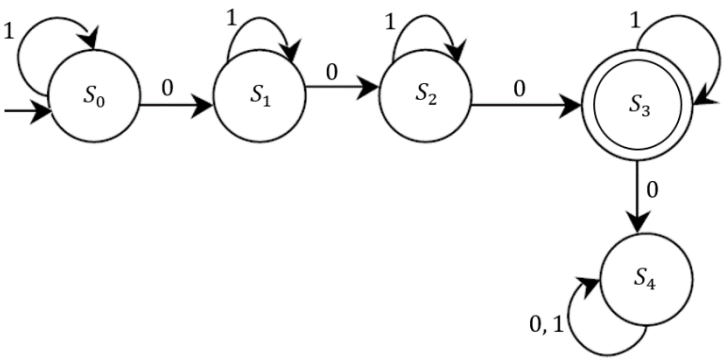
Solução:



	0	**1**
$\boldsymbol{S_0}$	S_0	S_1
$\boldsymbol{S_1}$	S_0	S_2
$\boldsymbol{S_2}$	S_0	S_3
$*\boldsymbol{S_3}$	S_4	S_3

3. Desenvolva uma máquina de estados capaz de identificar uma *string* contendo três zeros consecutivos dado que $\boldsymbol{\Sigma} = \{0, 1\}$.

Solução:



	0	1
$\boldsymbol{S_0}$	S_1	S_a
$\boldsymbol{S_1}$	S_2	S_1
$\boldsymbol{S_2}$	S_3	S_2
$*\boldsymbol{S_3}$	S_4	S_3
$\boldsymbol{S_4}$	S_4	S_4

4. Considerando a expressão regular $(a(b+c)^*)^*d$ que define a linguagem L, escreva pelo menos seis *strings* que façam parte desta linguagem.

Solução: d, ad, abd, acd, aad, abbcbd, ...

5. Considerando a expressão regular $(a+b)^*(c+d)$ que define a linguagem L, escreva pelo menos seis *strings* que façam parte desta linguagem.

Solução: c, d, ac, abd, babc, bad, ...

6. Considerando a expressão regular $(a^*b^*)^*$que define a linguagem L, escreva pelo menos seis *strings* que façam parte desta linguagem.

Solução: ε, a, b, ab, ba, aa, ...

6 APENDICE A – O PADRÃO POSIX DE EXPRESSÕES REGULARES

O Posix foi definido em 1988 pelo IEEE- *Institute of Electrical and Electronic Engineering*, para definir um conjunto de normas para como um esforço para manter um mínimo de compatibilidade entre os programas de computadores. Destacam-se entre estas normas aquelas relacionadas a redação de expressões regulares que estão sumarizadas na Tabela 5.

	Descrição
.	Reconhece qualquer caractere único. Por exemplo: $a.c$ reconhece “abc”, etc.. Contudo, $[a.c]$ reconhece apenas “a”, “.”, ou “c”.
[ ]	Reconhece o caractere que está entre os colchetes. Por exemplo: $[abc]$ reconhece “a”, “b”, ou “c”; $[a-z]$ reconhece um range de letras minúsculas entre “a” e “z”.
[^]	Reconhece todos os caracteres que não estão contidos entre colchetes. Por exemplo: $[^abc]$ reconhece qualquer caractere exceto “a”, “b”, ou “c”; $[^a-z]$ reconhece qualquer caractere que não esteja no range de “a” até “z”.
^	Reconhece a posição inicial dentro de uma *string*.
$	Reconhece a posição final de uma *string*, ou a posição imediatamente anterior ao caractere de nova linha.
( )	Os parênteses marcam subexpressões, também chamadas blocos.
$\boldsymbol{\backslash n}$	Reconhece a enésima subexpressão marcada, desde que n seja um algarismo entre 1 e 9.
*	Reconhece o elemento que precede este sinal zero ou mais vezes, trata-se da operação *Kleene star*. Por exemplo: $[xyz]^*$ reconhece ““, “x”, “y”, “z”, “zx”, “zyx”, “xyzzy”, etc.; $(ab)^*$ reconhece ““, “ab”, “abab”, “ababab”, etc..

$\{m,n\}$	Reconhece o elemento que o precede no mínimo m e no máximo n vezes. Por exemplo: $a\{3,5\}$ reconhece somente “aaa”, “aaaa”, e “aaaaa”.
?	Reconhece o elemento que o precede zero ou uma vez. Por exemplo: $ab?c$ reconhece somente “ac” or “abc”.
+	Reconhece o elemento que o precede uma ou mais vezes. Por exemplo: $ab+c$ reconhece “abc”, “abbc”, “abbbc”, etc., mas não reconhece “ac”.
|	Reconhece um elemento ou outro. Representa a operação de união e, geralmente é lido como ou. Por exemplo: $abc\|def$ reconhece “abc” ou “def”.

Tabela 7 - Comandos básico da norma Posix para expressões regulares

7 REFERÊNCIAS

AHO, A. V. et al. **Compiladores:** princípios, técnicas e ferramentas. 2º. ed. Boston, MA, USA: Pearson Education Inc. , 2007.

BARRAL, B. File:AnalyticalMachine Babbage London.jpg. **Wikipedia**, 2009. Disponivel em: <https://commons.wikimedia.org/wiki/File:AnalyticalMachine_Babbage_London.jpg#filelinks>. Acesso em: 20 Fev. 2020.

BERGIN, T. J. **50 Years of Army Computing from Eniac to MSRC**. Aberdeen, MD. USA: Army Research Laboratory, 1996.

BERGMANN, S. D. **Compiler Design:** Theory, Tools, and Examples. [S.l.]: Rowan University, 2010.

BÖHM, C.; JACOPINI, G. Flow diagrams, turing machines and languages with only two formation rules. **Communications of the ACM**, v. 9, n. 5, p. 366-371, 1966.

DAVIS, J. S. File:Commodore Grace M. Hopper, USN (covered).jpg. **Wikipedia**, 1984. Disponivel em: <https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg>. Acesso em: 20 Fev. 2020.

DIJKSTRA, E. W. A Case against the GO TO Statement. **Commun. ACM** , v. 3, n. 1, p. 147-148, Mar. 1968.

HOLUB, A. I. **Compiler design in C**. Englewood Cliffs, NJ, USA: Prentice Hall Software Series, 1990.

KNUTH, D. E. Structured programming with go to statements. **ACM Computing Surveys (CSUR)**, v. 6, n. 4, p. 261-301, 1974.

MENABREA, L. F. The Analitical Engine Invented by Charles Babbage. **Fourmilab**, 1842. Disponivel em: <http://www.fourmilab.ch/babbage/sketch.html#NoteG>. Acesso em: 20 Fev. 2020.

MICROSOFT. Microsoft Macro Assembler - Documentation. **MSDN - Microsoft Developers Network**, 2010. Disponivel em: <https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference?view=vs-2019>. Acesso em: 01 Mar. 2020.

MOGENSEN, T. Æ. **Basics of Compiler Design**. Copenhagen: Department of Computer Science University of Copenhagen, 2010.

NAUR, P. et al. Revised report on the algorithmic language Algol 60. **Communications of the ACM**, v. 6, n. 1, p. 1-17, 1963.

PITTS, A. M. **Regular Languages and Finite Automata for Part IA of the Computer Science Tripos**. Cambridge University Computer Laboratory. Cambridge, p. 58. 2013.

RANGEL, J. L. INF1626 Informática, PUC-Rio. **PUC RIO**, 2013. Disponivel em: <www.inf.puc-rio.br/~inf1626>. Acesso em: 02 Mar. 2017.

WIKIMEDIA COMMONS. File:JohnvonNeumann-LosAlamos.gif --- Wikimedia Commons

ZANDER, C. Languages (Finite State Machines). **CSS 448/548 - Introduction to Compilers Fall 2012**, 2012. Disponivel em: <http://courses.washington.edu/css448/zander/Notes/langFSM.pdf>. Acesso em: 03 Mar. 2017.