Interpretadores e compiladores, introdução

Frank Coelho de Alcantara

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 proposital, matemática e formalmente, a humanidade conseguiu criar máquinas capazes de resolver qualquer problemas de forma rápica e precisa. Não seria nenhum exagero afirmar que todo o desenvolvimento que vimos durante a segunda metade do Século XX foi devido as linguagens de programação.

Introdução

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.

Vamos tratar de máquinas abstratas feitas para entender a linguagem que os programadores usam para criar algoritmos e resolver problemas.

Usamos a palavra programadores de forma displicente, quase como use fora uma gíria, para representar, cientistas da computação, entenheiros de computação, analistas de sistemas e outros profissionais relacionados a ciência da computação tanto na sua forma pura quanto em aplicações.

Programadores usam linguagens de programação para escrever o código que, quando executado, irá solucionar problemas e permitir que a humanidade avance com a criação 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. Este é um trabalho delegado as tecnologias de compilação e interpretação, ainda que a fronteira entre estas tecnologias seja difusa.

Vamos explorar os processos de tradução que são utilizados por compiladores e interpretadores. E, principalmente, entender esta tecnologia.

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.

Muitos estudiosos acreditam que a computação começou junto com a matemática e a contabilidade. A própria palavra computação tem origem no verbo latino *computare* que significa fazer cálculos. Outros 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 a 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*, que pode ser vista na Figura 1, Babbage foi parcialmente ajudado por Ada Lovelace, dama da sociedade britânica e filha do poeta **George Gordon Byron**, Lord Byron).

Figura 1: A máquina Analítica de Charles Babbage.

Fonte: (BARRAL, 2009)

Ada Lovelace, Lady Byron, 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 analitica* nos cálculos sugeridos no manuscrito. Todo o trabalho de Lovelace e Babbage seria apenas uma nota de rodapé na história não fossem as instruções que **Ada Lovelace** 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).

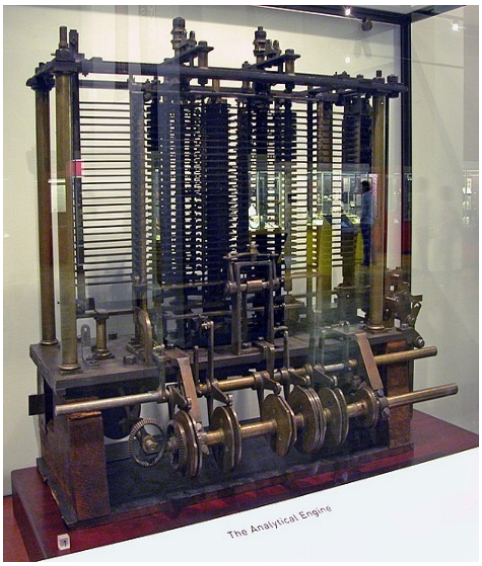

Figure 1: (BARRAL, 2009) - A Máquina Analítica de Charles Babbage

Vamos marcar o começo da computação em algum trabalho científico do começo do Século XX. Talvez, os Trabalhos de Turing sejam o ponto ideal mas, falaremos sobre isso em outro tópico.

Os primeiros anos

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 com a sua própria linguagem de programação, ***Plankalkül***.

Esta linguagem ***Plankalkül***[1], ainda que embrionária já poderia ser classificada como imperativa e altamente tipada. o trabalho de **Suze** não foi publicado até a década de 1970, devido ao preconceito inerente a ciência por causa das restrições impostas a Alemanha pela Segunda Guerra Mundial. Em 1949 **John W. Mauchly** apresenta uma linguagem chamada de *Short Code*, a base para o desenvolvimento UNIVAC a primeira máquina que pode ser chamada de computador eletrônico e que então representava (BERGIN, 1996) o maior conjunto de equipamentos eletrônicos interligados . No caso da linguagem ***Short Code***, a compilação era feita a mão, uma instrução era posta em execução 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. Compilar, é na verdade, a tarefa de colocar em listas.

[1] Em tradução livre: cálculos planos.

Grace Hopper

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 pelo sistema de colocação em listas automático, um sistema capaz de compilar uma lista, foi liberada pela equipe de para uso com o nome ***Math-Matic***. Este novo ambiente tecnológico, compilador e linguagem de programação, foi utilizado em máquina comercias: UNIVAC e UNIVAC II.

Fortran, Algol e Cobol

Poucos anos depois, em 1957 **John Backus**, trabalhando na IBM, criou o ***Fortran*** (**FOR**mula **TRAN**slation).

A vida do ***Fortran*** foi curta, a linguagem foi modificada e rapidamente substituída pelo ***Fortran II*** em 1958. Sendo esta versão a que deve ser considerada quando estamos falando do avanço da ciência da computação. A amável leitora há de perdorar este escriba quando ele tomar a liberdade de, a partir deste momento, usar apenas a palavra ***Fortran*** para se referir a esta linguagem. Afinal, o ***Fortran*** continua, até o ano de 2020 gozando de boa saúde e sendo representado pelo ***Fortran 2018*** que, como o nome indica, foi lançado em 2018 e inclui, entre outras maravilhas conceitos de programação paralela e concorrente.

O ***Fortran*** 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 um ano muito produtivo.

Em 1957 **Noan Chomsky** publicou o livro ***Syntactic Structures***[2] 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 Com-*

[2] Em tradução livre: estruturas sintáticas.

puting Machinery[3]) e da Europa, representados pelo GAMM (*Gesellschaft für Angewandte Mathematik und Mechanik*[4]) 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.

BNF e Algol

John Backus, criou uma linguagem de programação apenas para provar que um processo de descrição, por meio de uma linguagem específica, poderia ser usado para criar uma linguagem de programação, esta meta linguagem recebeu o nome de ***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***.

O ***ALGOL-60***, marca a primeira vez em que uma linguagem baseada em uma notação formal, ***Backus-Naur-Form***¸ foi usada para definir uma linguagem de programação.

A linguagem ***ALGOL-60*** foi responsável por estabelecer as bases para a criação do compilador moderno (NAUR, BACKUS, *et al.*, 1963) e está diretamente ligada a todas a linguagues de programação que usamos neste início de Século XXI. 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 voltados para aplicações científicas e matemáticas, no MIT - **Massachusetts Institute of Technology**, **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. Em 1959 surge o *Lisp 1.5* e cria definitivamente, os paradigmas do que chamamos hoje de linguagem funcional.

Figura 2: Grace Hopper - Uma das inventoras do compilador moderno.

Fonte: (DAVIS, 1984) .

Em 1959, é publicado o ***COBOL***, por um comitê de pesquisadores liderados por

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

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

Figure 2: Grace Hopper (DAVIS, 1984) – Uma das inventoras do compilador moderno.

Grace Hopper. A mesma Grace Hopper que alguns anos antes havia trabalhado na criação do primeiro compilador.

Os anos 1970, o Basic e o C

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 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, sob demanda e de forma urgente, por **Ritchie** permitiram que uma versão do ***C*** e seu compilador fossem incluídas na versão 2 do Unix. Mas, precisamos ser justos, o ***C*** é o resultado de uma longa linha evolucionária que pode ser traçada até o **Algol**, como pode ser visto na Tabela 1.

Tabela 1 - História das Versões da linguagem C

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

Fonte: o autor (2020) .

Como **Ritchie** e **Thompson** trabalharam juntos na transformação do B em C, não é raro atribuir a parternidade do C a estes dois pesquisadores. 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. Está vivo e sendo atualizado.

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 da guerra do **DOS** 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.

O C++ e o Java

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.

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

Stroustrup criou uma das linguagens de programação com maior sucesso na história. O **C++** deve muitos dos seus conceitos tanto ao **C** quanto a **SIMULA**, projeto de **Ole-Johan Dahl** e **Kristen Nygaard** no Norwegian Computing Centre (NCC) em Oslo entre 1962 e 1967, a primeira linguagem de programação a aplicar conceitos de programação orientada à objetos e criada para ser uma lingagem capaz de descrever e simular objetos e eventos da vida real. A **Simula** também tem sua origem relacionada ao **ALGOL**. O próximo salto tecnológico deve ser atribuído ao **JAVA**.

Criado por **James Gosling**, da Sun Microsystems, além de adotar todos os paradigmas da programação orientada a objetos inova na criação de novas técnicas de compilação e interpretação. Na busca do Eldorado representado pela possibilidade de escrever um único código e rodar em qualquer máquina. Lembre-se, este foi um dos motivos que lá nos anos 1960, o mercado não adotou o **Fortran** como linguagem de programação universal.

E surge a WeB e a computação Quântica

E chegamos a era da web, **Rasmus Lerdorf** com o **PHP** (1994), **Guido Van Rossum** com o **Python** (1989) e **Brendan Eich** com o Javascript (1995) movimentam a Internet permitindo a expansão da web e a criação de tecnologias remotas e interativas.

A toda a interatividade e multiplicação de conhecimento devida a web, cria um ambiente propício para o desenvolvimento de técnicas e tecnologias relacionadas a inteligência artificial e ao aprendizado de máquina e fortalece o **Python** de **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.

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 linguagens de programação que tiveram 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.

Linguagem de Máquina

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.

Apenas no Século XX conseguimos nos comunicar com as máquinas e conseguimos dizer o que queremos que elas façam.

Ainda que para conseguir comunicar nossas ideias às máquinas 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 é o *software*. Para os nossos objetivos, tanto a palavra hardware quanto a palavra software são por demais imprecisas.

O hardware diz respeito a toda a máquina, e a 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.

Hardware

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, que chamaremos de binário, podemos especificar um conjunto de cadeias (*strings*[6]) finito, por sua vez o conjunto destas *strings* 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, podemos dizer que toda a computação realizada pela humanidade e suas máquinas é devida a uma linguagem composta de *strings* de zeros e uns. Cada uma destas *strings* 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 *strings*. Felizmente este é um livro de compiladores e interpretadores e podemos ser um pouco mais formais. Destaque-se que estas *strings* 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.

A Intel[7] chama este conjunto d e instruções de *x86_64 Instruction Set*[8] e

[6] Em tradução livre: cadeias. Mas, neste livro, vamos ignorar isso e usar *strings*.

[7] Intel; http://www.intel.com.br

[8] Em tradução livre: conjunto de instruções *x86_64* (cada *string* da linguagem tem 64

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 *strings* para que sejam entendidas e executadas. Esta linguagem *x86_64* só pode ser entendida por máquinas definidas de acordo com as especificações da Intel. Não será possível pegar um conjunto de *strings* desta linguagem e executar, por exemplo, em um computador da arquitetura ARM, ou RISC. Esta limitação é importante e define o mercado.

O formalismo da máquina

A linguagem de máquina é a mais formal e específica linguagem envolvida nos processos de computação. Isso a torna praticamente impossível de usada por seres humanos.

Existe aqui, alguns detalhes históricos interessantes para a reflexão própria e pessoal da amável leitora. No começo, as máquinas eram programadas apenas com uma sequência de *strings* do alfabeto Σ_b, depois, os engenheiros perceberam que estas *strings* podiam ser representadas de forma mais simples por *strings* de um alfabeto composto por números do sistema hexadecimal $\Sigma_h = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. Esta foi uma troca simples, trocamos uma representação de dígitos em um sistema de numeração por dígitos em outro sistema de numeração. Quando evoluímos um pouco mais criamos uma linguagem com mnemônicos, para representar estes *strings* e as funções que eles deveriam executar. E assim, surgiu a família de linguagens chamada de *Assembly*.

O **Assembly** é apenas uma coleção de mnemônicos organizada sobre uma sintaxe e semânticas simples que representam de forma mais adequada ao ser humano, as instruções em binário que queremos que a máquina execute.

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. Aqui, cabe um esclarecimento, os primeiros programas eram realmente montados. A máquina tinha que ter seu *hardware* trocado todas as vezes que algum programa diferente fosse executado. Neste processo, fios, relés e, mais tarde, válvulas, eram rearranjados. Literalmente a máquina era remontada. Assim, parece natural que a primeira linguagem de programação fosse uma linguagem de montagem.

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 de Σ_b.

símbolos)

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

A família *Assembly* é 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, são especificas de uma determinada arquitetura o que dificulta o uso de códigos desenvolvidos para uma arquitetura em outra. Esta especificidade, no entanto, permite que o código escrito faça uso de todas as características de uma determinada arquitetura, permitindo mais desempenho em termos de velocidade e recursos. Ou seja, A linguagem *Assembly* desenvolvida para a arquitetura *x86_64* será diferente de uma linguagem *Assembly* desenvolvida para a arquitetura RISC-V. Estas linguagens 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. 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
```

Linguagem de maquina

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_h, hexadecimal, pode ser chamada de linguagem de máquina.

Isso quer dizer que tanto o *Assembly*, quanto qualquer outra forma de memorização que utilizamos terá que ser traduzida em *strings* 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.

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.

Assembler compila?

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.

Assembler compila?

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.

Um tradutor 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 *Assemblers*:

Assembler um passo

- ***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.

Assembler dois passos

- ***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

- ***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

- ***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*:

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

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

Assembler - processo de tradução

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

[11] Em tradução livre: etiqueta.

[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.

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 sejam 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 deve ser 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 a amável leitora pode ver, 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. Pelo menos isso é verdade em todas as linguagens que não obedecem os paradigmas fucional e lógico. Em ciência da computação chamamos estes comandos de declarações, do inglês *statement*, a palavra instrução, em português, também pode ser utilizada com o mesmo sentido. Os comandos, ou instruções são classificados em três grandes grupos:

- **comandos para controle** de fluxo explícito da sequência de execução (por exemplo: *goto*, *;, begin/end*);
- **comandos condicionais** ou para 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 comando *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 foi terminantemente 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 dos grandes erros da humanidade. Mais ou menos no nível da inquisição, do holocausto e do holomodor.

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. Neste trabalho, Corrado Böhm e Joseph Jacopini (1966) mostraram 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 estão na mesma classe do *goto* no que diz respeito a desvio de fluxo já que, no

[13] Em tradução livre: início e fim.
[14] Temos uma expressão para esse caso: código macarrônico.

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*. E isso, este pequeno detalhe no retorno, faz com que o seu código fique **estruturado**. As funções, ou procedures, deram início a uma sequência de criação de novos construtores de código. A leitora há de entender, neste livro, a palavra construtores, como referência a estruturas de código. Estes novos construtores deram origem a formas diferentes de programar, que chamamos de paradigmas de programação mas, isso é assunto para outro capítulo.

Material de apoio

Você pode baixar a versão em pdf desta aula clicando aqui

Obras Citadas

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. 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. 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. NAUR, P. et al. Revised report on the algorithmic language Algol 60. Communications of the ACM, v. 6, n. 1, p. 1-17, 1963. WIKIMEDIA COMMONS. File:JohnvonNeumann-LosAlamos.gif — Wikimedia Commons