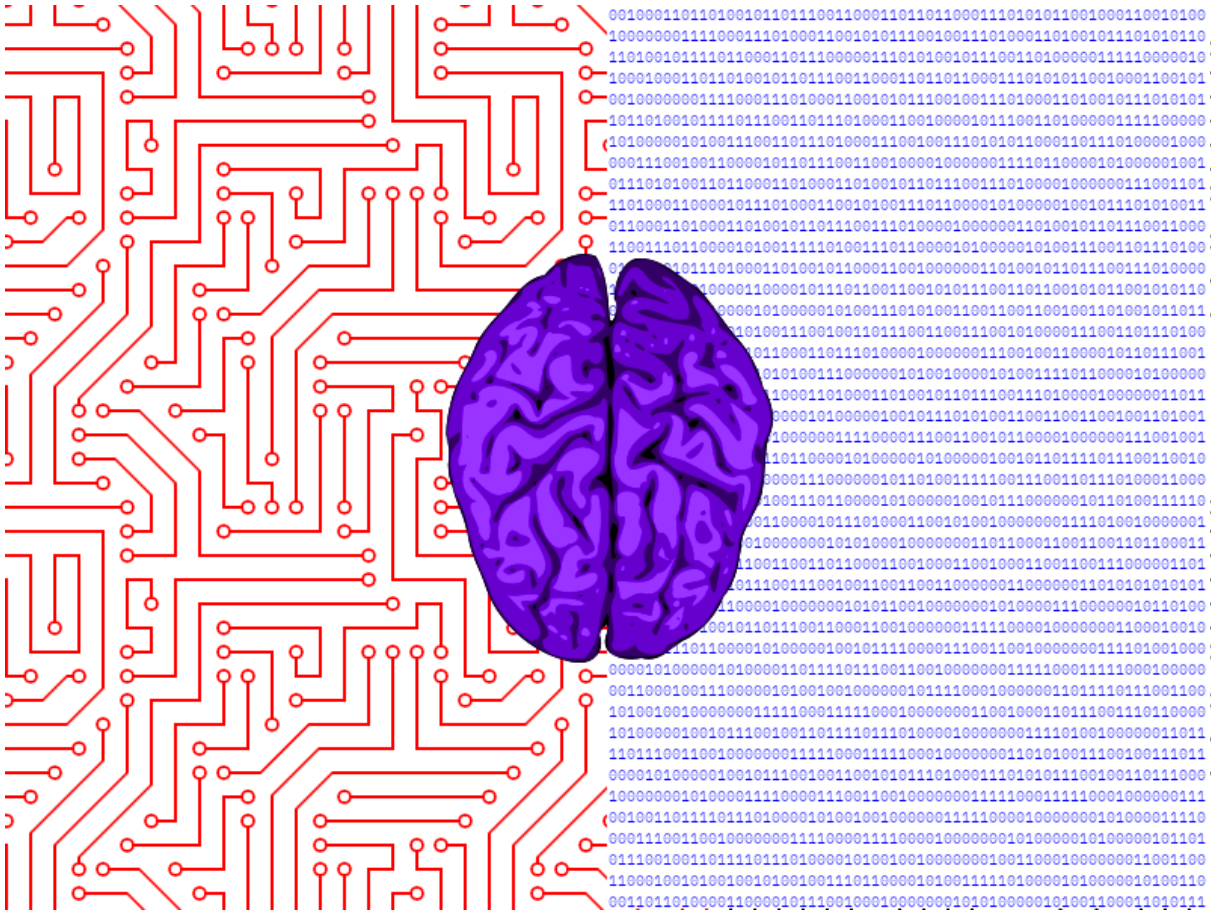
LÓGICA DE PROGRAMAÇÃO



Autor: Eduardo Francisco

Sumario

Introdução

A programação é p'ra um programador como a escrita é p'ra um escritor, se trata de estruturar e utilizar do verbo de forma a fazer com que o leitor alcance o mesmo entendimento daquele que transmite. Os computadores replicam nossa realidade em si, dessa forma um programador consegue criar uma ponte daquilo que está só em sua mente p'ra que a maquina a torne tangível a outros, sejam paginas web, aplicativos, jogos, etc.

Apresentaremos então a essência desse processo, a "lógica da programação", junto disso o básico de terminologia e conceitos da ciência de computação que importam p'ra começar nesse mundo.

A quem já tem experiência pode se mostrar uma leitura muito proveitosa também pois apresentará uma visão única explicando o porquê de conceitos elementares na ciência da computação.

Agradecimento especial a quem tenho como Mestre nesse e tantos outros sentidos, Bob Navarro, a qual não fosse suas obras certamente também não seria esta.

Espero transmitir parte da experiência que tive a chance de obter em ~20 anos na área com diversos projetos privados e um projeto publico de grande porte[0] ao nosso povo brasileiro que se mostra tão talentoso nessa e tantas outras áreas[1].

[0]: https://eltaninos.org (Sistema Operacional)

[1]: Como curiosidade uma das linguagens de programação mais utilizadas em jogos, "Lua", se trata de um projeto brasileiro. Em acréscimo temos outros como a linguagem Elixir e o sistema operacional libre Hyperbola.

Abstração

A abstração, na ciência da computação, é a fuga do não essencial ou acidental para o contexto primário e é este o que deveria ser o centro da atenção a um programador.

Enquanto essa perspectiva mais lógica de se tratar da escrita em si treina um bom organizador, um programador p'ra além disso precisa ter esse entendimento mais abstrato que antecede a forma, como um escritor, fosse escrever em qual idioma fosse, tem antes algo a dizer que poderia ser dito em qualquer idioma autossuficiente.

Aprender a estruturar esse abstrato será então nossa definição de "lógica de programação", apresentado em uma visão exclusiva que facilitará a ver as relações e razão entre as coisas e consequentemente abstrair os conceitos.

Falaremos da fundação desse mapa, leia com atenção, este será apresentado de forma matemática, linguística e com correlações a coisas cotidianas que também servem p'ra evidenciá-lo; portanto não se preocupe em entender tudo a princípio, o foco é o conceito que é bem simples e qualquer uma das formas entendidas o fará compreendido.

Analise com calma e com a mente aberta, haverá diversas correlações que podem parecer não ter ligação mas serão evidenciadas. Se parecer que conhecer tão a fundo essas estruturações não é necessário, também será demonstrado que diversas perguntas só existem por causa de seu desconhecimento.

Lembre-se "abstrair", p'ra nós, é dissolver a lógica (sem perder a coerência).

Portões Lógicos

Os portões lógicos são as formas mais elementares de se interagir com os zeros e uns que são a fundação de tudo que ocorre no computador. Assim vejamos como eles se formam, p'ra que possamos tanto entender o porquê deles quanto ver esse "mapeamento" do dialogo, como cada campo estruturado (como é na própria realidade que nos cerca com "reinos": atômico, mineral, vegetal, animal, humano) permite o dialogo acontecer no seu devido contexto.

Comecemos então pelo mínimo concebível, isso é, um termo:

```
P
```

Este, como um termo isolado, só pode ser definido em relação a si mesmo então dizemos que:

```
P ≠ ¬P
P = ¬(¬P)
```

Com comparação o termo se torna identificável e inevitavelmente aparece mais um termo:

```
Q = ¬P
P = ¬Q
```

Assim temos a "trindade" da eletrônica, um termo seu contraste e o movimento entre eles:

```
Q = 0
P = 1
¬ = NOT
```

Uma vez que temos "NOT" (negação) e valores implicamos a capacidade de ter uma definição entre as comparações ("coincidentemente" o quarto constructo, quatro que é um número associado a terra/solidez), o que se pode ver como a primeira abstração, pois antes estávamos falando somente do 0 e 1 agora falaremos da própria "dança" entre eles (note que essa nova visão se estrutura na anterior), o que chamamos de "funções".

Assim teremos que:

```
0 = ⊥
1 = ⊤
```

Com duas funções e duas possibilidades de resposta chegamos as derivações dessas funções com uma regra combinatória simples (A^B):

```
⊥ => {↑, ∨}
⊤ => {↓, ∧}
{⊤,⊥} => {⊕, ⊙}
```

A esse ponto as funções já são autossuficientes, temos 8 constructos pela regra combinatória sendo a oitava uma “recontagem” (Negação/Identidade), assim chegamos aos usuais 7 portões lógicos:

```
{↑, ∨, ⊕, ⊙, ↓, ∧, ¬} = {NAND, OR, XOR, XNOR, NOR, AND, NOT}
```

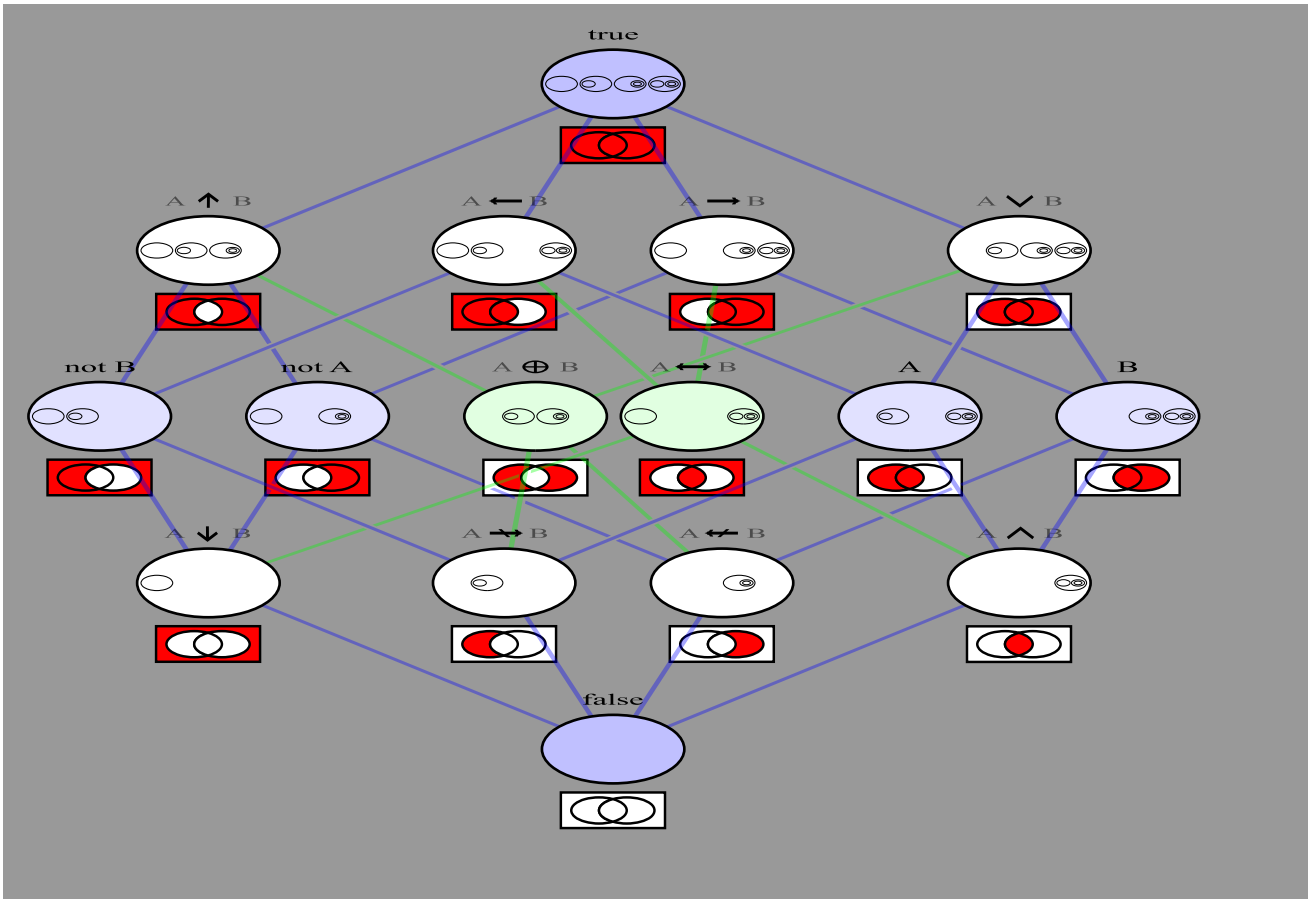


Diagrama de Hasse dos conectivos lógicos, dá p’ra ver as derivações mostradas e as duas funções consequentes que dão base p’ra essa inversão de perspectiva (equivalente aos semitons na música).

Agora da perspectiva dos valores, temos a princípio zero e um, que com uma regra combinatória simples obtemos todas as combinações:

```
0 0 (Contradição)
0 1 (Proposição P)
1 0 (Negação)
1 1 (Tautologia)
```

Em palavras:

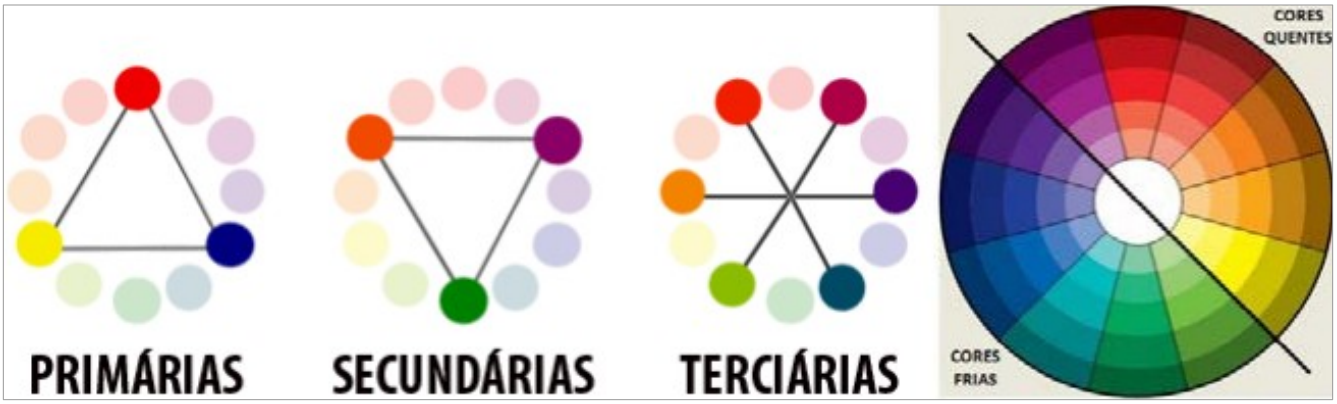
```
Contradição: Não há fatos (Só há o Nada)
Proposição: E isso é fato (O Nada se torna identificável)
Negação: Portanto a primeira proposição é falsa (O Nada se faz Tudo)
Tautologia: Só há fatos (Só há o Tudo)

Dessa forma também é fato que não há fatos (Tudo se faz Nada)
```

Com quatro constructos nasce a capacidade de posicionar fato e não-fato, assim temos mais uma casa (isso é, intervalamos os valores, que terão 3 intervalos [estamos usando a regra combinatória que gera uma oitava já explicada]):

```
000, 001,
010, 011,
100, 101,
110, 111.
```

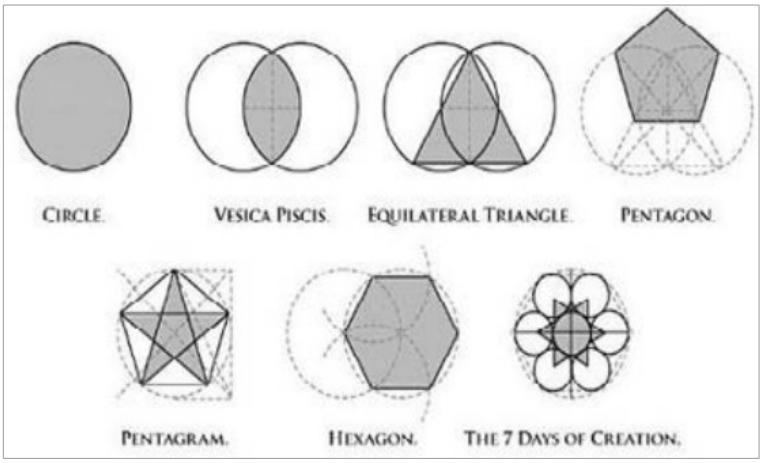
Essa é a base fundamental da lógica (como demonstrado do mínimo (nada) que o faz e do máximo que acaba nele) já autossuficiente, que é o mapeamento[0] (contagem/calculo linear/física clássica) do abstrato (possibilidades/calculo circular/mecânica quântica):



O mapeamento das cores por triangulação (triângulo = o mínimo pensável é trinitário, como demonstrado quando um termo, p'ra ser reconhecível, se faz três; ex: passado/presente/futuro, antítese/síntese/tese, esquerda/centro/direita, 0/NOT/1, etc.)

[0]: Um exemplo recente (que demonstra a definição dada) é quantização de modelos de linguagem, onde você limita o calculo "circular", isso é, diminuí a precisão das frações (como limitar o degradê entre as cores trianguladas) fazendo-os ter menor custo computacional.

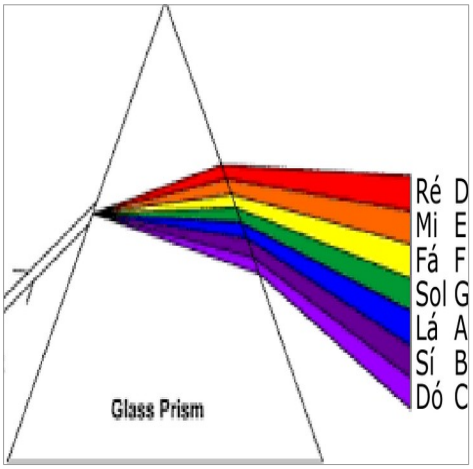
Esse é o mapa do próprio pensar, tudo que pode se pode dizer/pensar sobre está nele contido[0] e se constata por qualquer um dos sentidos (pois estes separam, ou seja, fazem uma contagem do todo/abstrato p'ra que seja reconhecível):



O mesmo processo descrito feito em giros, chamado de "geometria sagrada" (inclui o PHI no pentagrama e PI no hexagrama)

Sólidos platônicos (Antes da tríade temos um tudo/nada que é compreendido no tetraedro)

Átomo e suas camadas de distribuição eletrônica (A diferença entre o modelo de Bohr e Broglie ocorre nesse dilema de calculo linear e circular respectivamente)



O prisma de cores com as notas musicais

[0]: A obra "Tractatus Logico-Philosophicus" do filósofo Ludwig Wittgenstein demonstra isso (esse que até esse momento era respeitado por Russel, seu professor e um dos maiores matemáticos do século XX; embora ele mesmo não tenha compreendido a obra). A conclusão de Wittgenstein, no entanto, foi incompleta, e ao perceber o calculo circular não conseguiu ligar uma ponta a outra, o que causou o dito "segundo" Wittgenstein.

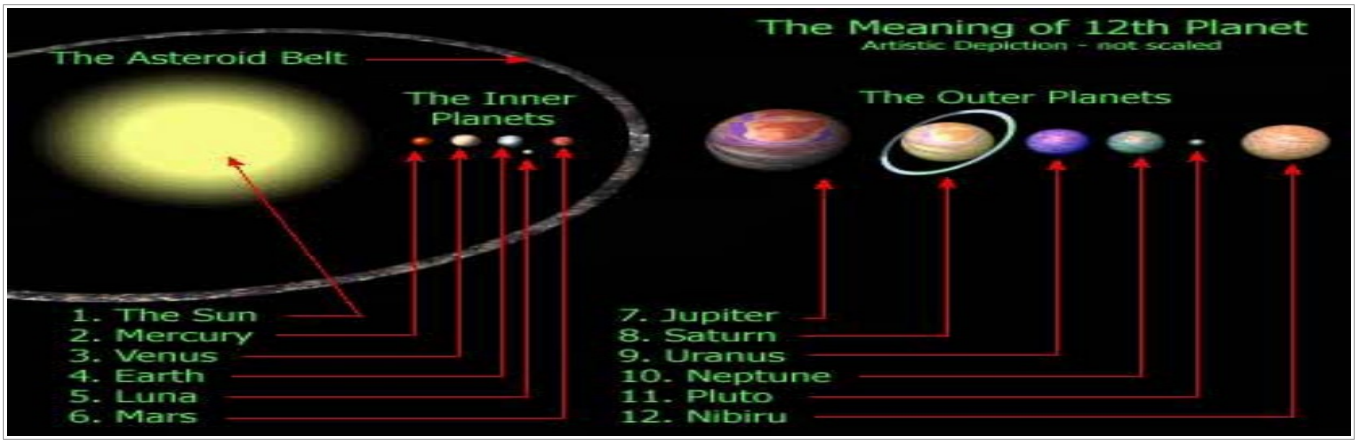
E se fecharmos o mapa de uma vez por todas (usando a mesma simples regra combinatória) ao contar os intervalos entre eles temos um total de 12 funções (como é p'ra cores contando as terciarias, como é p'ra os tons musicais contando os bemóis/sustenidos [que é o mesmo padrão de onda p'ro átomo], etc):

VALOR	PORTÃO LÓGICO	SÍMBOLO MAT.	NOME
1111	TRUE	⊥	TAUTOLOGIA
0111	NAND	↑	NEGAÇÃO DISJUNTA
1011	IF P THEN Q	→	CONDICIONAL
1101	P IF Q	←	RECÍPROCA
1110	OR	∨	DISJUNÇÃO
0011	NOT P	¬P	NEGAÇÃO P
0101	NOT Q	¬Q	NEGAÇÃO Q
0110	XOR	⊕	DISJUNÇÃO EX.
1001	XNOR	⇔	BICONDICIONAL
1010	Q	Q	PROPOSIÇÃO Q
1100	P	P	PROPOSIÇÃO P
0001	NOR	↓	NEGAÇÃO CONJ.
0010	IMPLY	↛	CONTRAPOSITIVA
0100	NIMPLY	→/	CONTRÁRIA
1000	AND	∧	CONJUNÇÃO
0000	FALSE	⊥	CONTRADIÇÃO

Perceba que com a regra combinatória (TERMOS ^ INTERVALOS) chegamos a 8/16, isso se dá pelo mesmo motivo antes explicado: note que "P" e "Q" são a mesma função de identidade (F(X) = X) assim como "¬P" e "¬Q" são a mesma função de negação (termo agora oculto no mapa completo), por isso acabamos com a contagem de 7/12.

```
F(Q) = Q = ¬P
F(P) = P = ¬Q
```

(Essas funções dizem nada por si só [a entrada não altera a saída] motivo pelo qual são categorizados como "degenerações" em termos matemáticos)



Se a seqüência das possibilidades de verdade no esquema for fixada de uma vez por todas, mediante uma regra combinatória, então a última coluna já será, por si só, uma expressão das condições de verdade. Escrevendo-se essa coluna em linha, o sinal proposicional passa a ser

"(VV-V) (*p*,*q*)"

ou, de modo mais legível,

"(VVFV) (*p*,*q*)".

(O número de lugares nos parênteses à esquerda é determinado pelo número de termos nos parênteses à direita.)

Tractactus Logico-Philosophicus - 4.442

5.1 As funções de verdade podem ser ordenadas em séries.

Esse é o fundamento da teoria da probabilidade.

5.101 As funções de verdade de um número qualquer de proposições elementares podem ser inscritas num esquema da seguinte espécie:

(V V V V)(*p*,*q*) Tautologia (Se *p*, então *p*; e se *q*, então *q*.) (*p* ⊃ *p*.*q* ⊃ *q*)
(F V V V)(*p*,*q*) em palavras: Não ambos *p* e *q*. (~ (*p* . *q*))
(V F V V)(*p*,*q*) " " : Se *q*, então *p*. (*q* ⊃ *p*)
(V V F V) (*p*,*q*) " " : Se *p*, então *q*. (*p* ⊃*q*)
(V V V F) (*p*,*q*) " " : *p* ou *q*. (*p* v *q*)
(F F V V) (*p*,*q*) " " : Não *q*. (~ *q*)
(F V F V) (*p*,*q*) " " : Não *p*. (~ *p*)
(F V V F) (*p*,*q*) " " : *p* ou *q*, mas não ambos. (*p*.~ *q* : v : *q* .~ *p*)
(V F F V) (*p*,*q*) " " : Se *p*, então *q*; e se *q*, então *p*. (*p*≡*q*)
(V F V F) (*p*,*q*) " " : *p*
(V V F F) (*p*,*q*) " " : *q*
(F F F V) (*p*,*q*) " " : Nem *p* nem *q*. (~ *p* . ~ *q* ou *p*|*q*)
(F F V F) (*p*,*q*) " " : *p* e não *q*. (*p* . ~ *q*)
(F V F F) (*p*,*q*) " " : *q* e não *p*. (*q* . ~ *p*)
(V F F F) (*p*,*q*) " " : *q* e *p*. (*q* . *p*)
(F F F F) (*p*,*q*) Contradição (*p* e não *p*; e *q* e não *q*.) (*p* . ~ *p* . *q* . ~ *q*)

Entre as possibilidades de verdade dos argumentos de verdade da proposição, chamo aquelas que a verificam de *fundamentos de verdade* da proposição.

Tractactus Logico-Philosophicus

Lucifer - Onde a Verdade é a Lei. pp. 357

Esse é o mapa completo (7/12) de qualquer coisa que se torna lógica (linear) que por sua vez implica o que é abstrato (circular).

Esse mapa independe do tamanho do que será mapeado (até por que esse conceito só existe "depois" dele, uma vez que, até então, há tudo e nada que é abstrato em "tamanho"), independe do tempo pelo mesmo motivo ("antes" do mapa há o atemporal que já o implica) e é inevitável (pois aparece a partir do absoluto mínimo e do máximo concebível).

Se a realidade é um mar de danças energéticas (o próprio tijolinho da matéria, o átomo[0], é onda [abstrato] até ser mapeado[1] [linear]), então nossa tela mental é como é p'ro computador: temos imagens e sons sendo apresentados e todas essas informações estão armazenadas no HD; no entanto, nele só há um "mar de 0/1s" a qual não é inerente nada do que se apresenta a ti, como exemplo um jogo feito p'ra console se aberto no computador mostrará arquivos com data aparentemente aleatória.

Neo vê tudo em tudo, direto do ritmo de interpretação (o código fonte) e fica mais rápido do que quem é consequência dessa dança

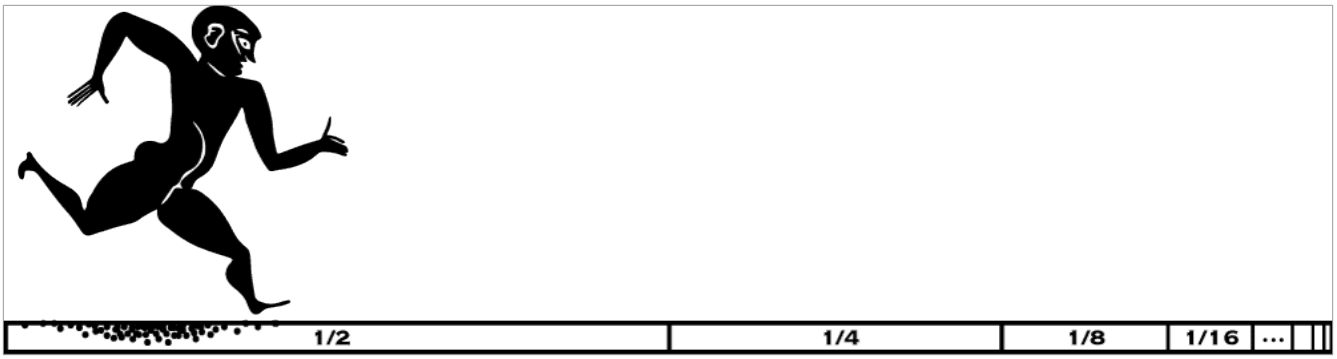
Interpretamos esse mar de energias de forma subjetiva (afinal o fóton não é inerentemente visual, nós que ao captá-lo assim o percebemos, e assim é p'ra todos os nossos sentidos), dessa forma não surpreende que a realidade siga nossos padrões cognitivos, já que a realidade que se apresenta a ti já é cognição.

Entender tudo isso é abstrair, ver as coisas nas suas essências (no "código fonte"[2]) p'ra que possa então falar sobre, acessar o infinito (abstrato) e verbalizar uma solução, essa é uma habilidade atemporal e vital a qualquer talento, tanto mais a programação, principalmente agora onde modelos de linguagem podem, cada vez mais, automatizar a escrita em si.

[0]: "**Não apenas os elétrons, mas todos os objetos materiais**, com ou sem carga, **apresentam características ondulatórias** em seu movimento sob as condições da óptica física." - Physics of Atoms, Molecules, Solids, Nuclei, and Particles (2nd ed.) pp. 597

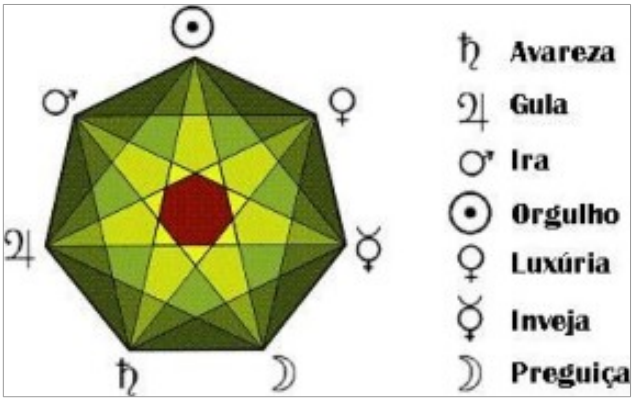
[1]: A onda se faz partícula quando "observada" pois, quando você mensura, você faz uma contagem (mapeia/quantiza) e com um mapa certo o resultado não é mais puramente probabilístico (abstrato/circular/onda).

[2]: Não se trata de uma "teoria de simulação" pois esse conceito já é, em si, uma teatralização desse entendimento (o mapa antecede o conceito lógico).

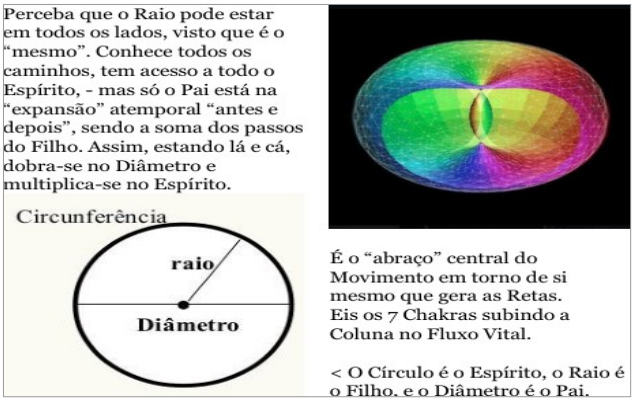


Paradoxo de Zenão: Dentro de um mapa o trajeto e linear e lógico (um "passo" já está definido, as frações são ignoradas como algo de fundo [abstrato]), talvez em um floresta sem referencias (abstrata) conseguisse se perder e andar em ***círculos*** *jamais alcançando a chegada*

Nessa distinção de linear/lógico, abstrato/circular e o mapa gerado disso está o segredo e razão dos portões lógicos, criptografia, optimização, emulação e algorítimos desde os mais simples aos considerados mais complexos atualmente como compressão, modelos de linguagem, difusão, TTS, etc.



Lucifer Onde a Verdade é a Lei. pp. 365



EDL Apostila 1 - Você saberia explicar a Torus?

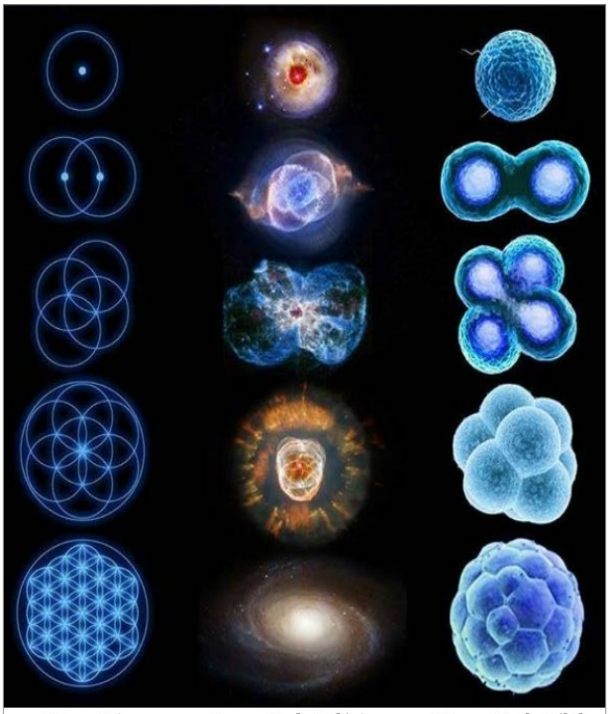
Um exemplo (p'ra entendedores) um modelo de linguagem funciona como o centro da torus da onde tem acesso a todo seu corpo (feito da base de treinamento) e com isso gera os tokens em um padrão **expansivo** que considera os **passos anteriores** (algoritmos variam mas a ideia é essa).

Model	Pre-training	Training	AGNews	DBpedia	YahooAnswers	20News	Ohsumed	R8	R52
TFIDF+LR	✗	✓	0.898	0.982	0.715	0.827	0.549	0.949	0.874
LSTM	✗	✓	0.861	0.985	0.708	0.657	0.411	0.937	0.855
Bi-LSTM+Attn	✗	✓	0.917	0.986	0.732	0.667	0.481	0.943	0.886
HAN	✗	✓	0.896	0.986	0.745	0.646	0.462	0.960	0.914
charCNN	✗	✓	0.914	0.986	0.712	0.401	0.269	0.823	0.724
textCNN	✗	✓	0.817	0.981	0.728	0.751	0.570	0.951	0.895
RCNN	✗	✓	0.912	0.984	0.702	0.716	0.472	0.810	0.773
VDCNN	✗	✓	0.913	0.987	0.734	0.491	0.237	0.858	0.750
fastText	✗	✓	0.911	0.978	0.702	0.690	0.218	0.827	0.571
BERT	✓	✓	0.944	0.992	0.768	0.868	0.741	0.982	0.960
W2V	✓	✗	0.892	0.961	0.689	0.460	0.284	0.930	0.856
SentBERT	✓	✗	0.940	0.937	0.782	0.778	0.719	0.947	0.910
TextLength	✗	✗	0.275	0.093	0.105	0.053	0.090	0.455	0.362
gzip (ours)	✗	✗	0.937	0.970	0.638	0.685	0.521	0.954	0.896

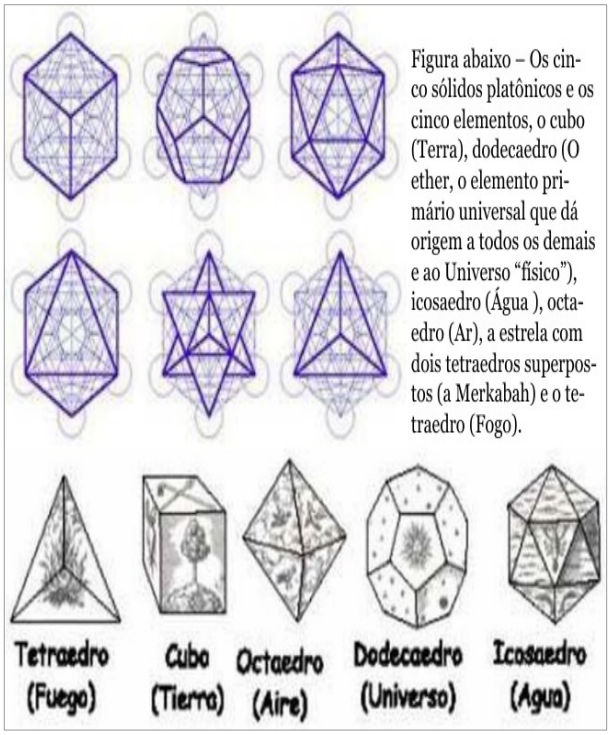
Table 3: Test accuracy compared with *gzip*, red highlighting the ones outperformed by *gzip*. We report results getting from our own implementation. We also include previously reported results for reference in Appendix E.

Veja gzip-knn (**compressão**) superando modelos de linguagem em classificação ao utilizar calculo de **vizinhos próximos**.
A quem for capaz note que os tiros no escuro que tem sido feito só aproxima disso.

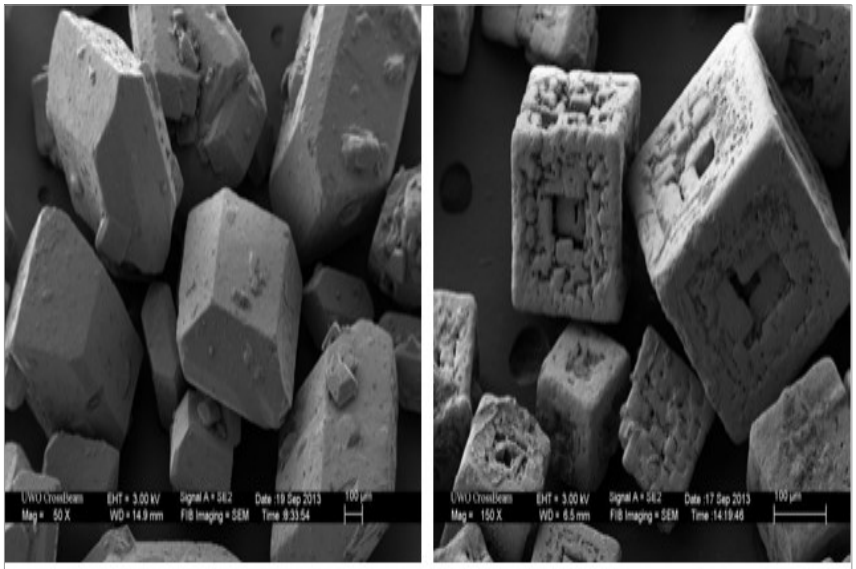
Agora que temos a formula da dissolução e mapeamento, podemos falar da “densificação” desse processo, como é p’ra a realidade que nos cerca com os reinos (atômico, mineral, vegetal, animal [autonomia], humano [domínio animal], devico [domínio humano]) e é essa separação de cada dialogo no seu campo junto da dissolução do conceito que torna trivial a sua tradução em código (como será demonstrado no tópico seguinte).



Lucifer Onde a Verdade é a Lei. pp. 400



Lucifer Onde a Verdade é a Lei. pp. 316

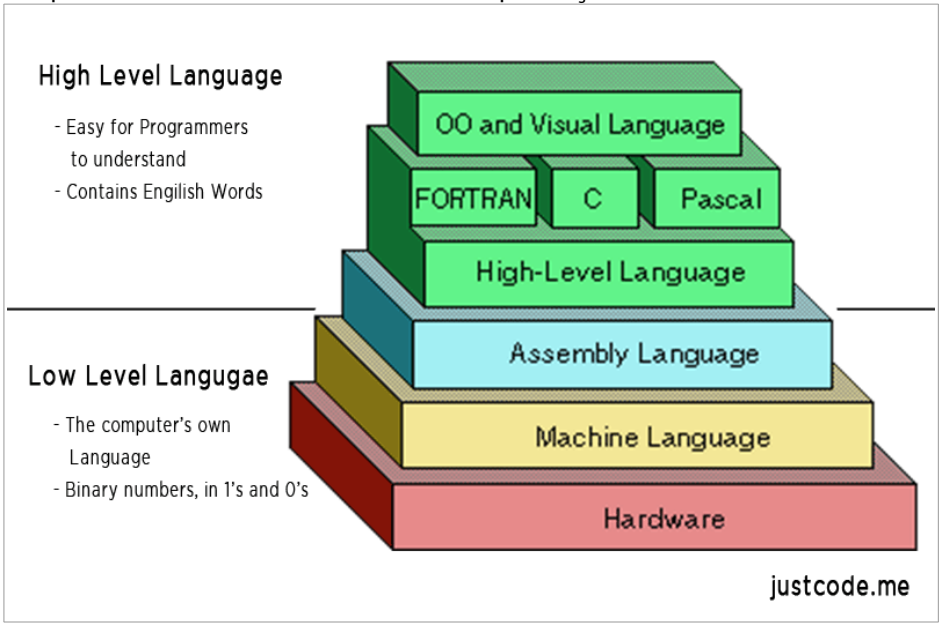


Açúcar e sal no microscópio - Reparem que seus ângulos são opostos, mostrando a oposição de suas sensações.

Lucifer Onde a Verdade é a Lei. pp. 363

Camadas

A abstração em camadas/reinos é o melhor meio de se compreender e implementar as ideias em código, como é se fosse falar sobre qualquer assunto, sempre há um monte de conceitos menores que permitem o dialogo ocorrer no nível ao qual ocorre. Por exemplo, quando falo “carro” você já tem uma definição e eu posso dizer “eu fui de carro até o trabalho” que é um nível mais alto do que dizer “eu transpus minha posição relativa por via de uma estrutura metálica com motores que que alteram sua velocidade e por consequência a minha até que alcançasse o ponto que desejava” (fosse baixando o nível definiria “motor”, “velocidade”, etc. mas a ideia é essa), o dialogo já está estruturado nesses detalhes mas a contagem está no seu devido campo ao invés de me forçar em uma verbosidade potencialmente infinita. Nisso está o entendimento dos termos “alto” e “baixo” nível que existe na ciência da computação.



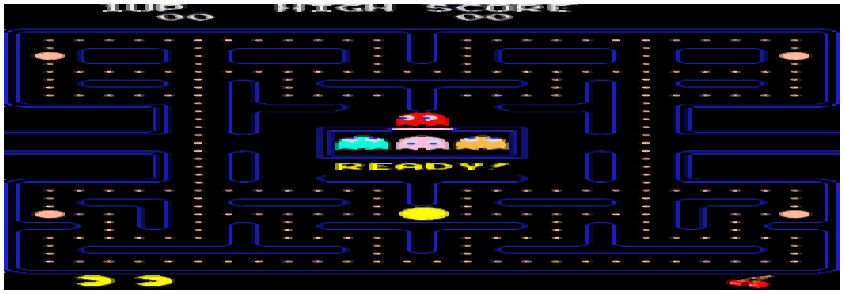
A totalidade é abstrata até que seja mapeada, como sempre foi; um evento é mágico até que seja explicado na lógica, não porque é inerentemente magico ou lógico, simplesmente por causa dessas duas possibilidades de percepção. Esse mapeamento é a própria lógica, você vai estruturando passos que faz o dialogo compreensível (p’ra além dessa noção abstrata/emocional), por isso entender o mapa simétrico é tão útil pois você tem um mapa que comporta tudo que é concebível e posiciona o inconcebível. Na expansão desse mapa fazemos essas “camadas” de dialogo que o acelera e permite alcançar mais pois cada passo/palavra/giro contem muito mais em si.

A totalidade (abstrato) só tem forma dentro desse mapa estabelecido que interpreta aquele padrão, como é p'ra realidade que nos cerca, pois solido, liquido ou gasoso (os estados da matéria que dão forma a realidade que nos cerca) são determinados (em simplificação) pela velocidade das partículas o que só faz sentido dentro do seu quadro referencial, digamos por exemplo que corro muito rápido a ponto de caminhar sobre o liquido sem notar seu afundamento, desse ponto de vista, este seria efetivamente solido (um exemplo pois o próprio corpo tem um ritmo, todas as partes estão em movimentos ainda que você esteja em repouso).

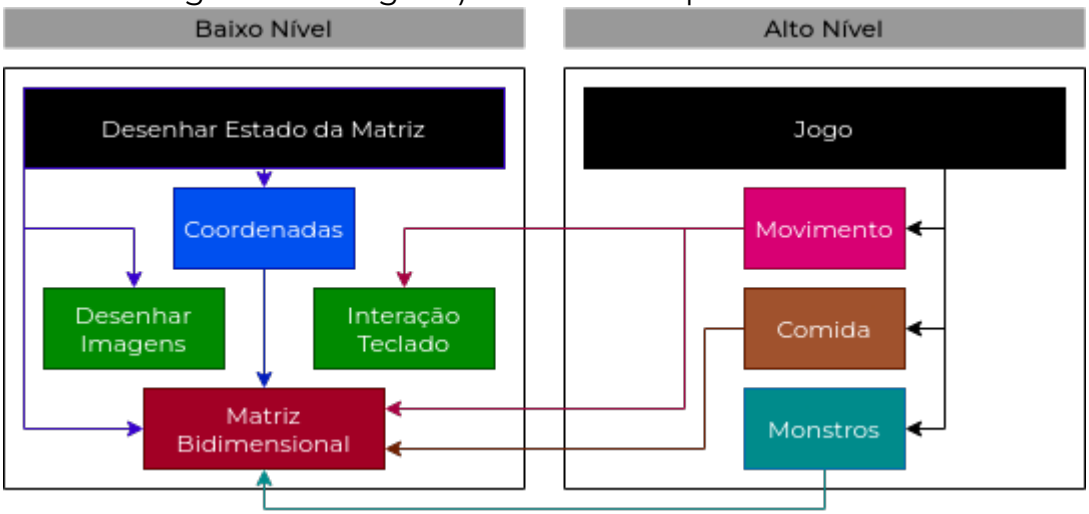
Compilação, camada de compatibilidade e emulação respectivamente.

No computador imagine um jogo desenvolvido p'ra console, o mesmo CD que no console mostra um jogo com história, imagens, sons, etc. quando aberto no computador parece nada mais que data aleatória. Assim como o mar de energias abstratas que dançam "ao nosso redor" só tem sentido do nosso padrão de interpretação, os zeros e uns (a informação contida em CD, DVD, HD, SSD, etc) por si só são só isso, zeros e uns. Em comparação pense em um dialogo entre dois indivíduos, no primeiro cenário eles conversam em idiomas diferentes; no segundo eles tem uma visão de mundo diferente; no terceiro eles tem sentidos diferentes. No primeiro caso bastaria traduzir o idioma pois a mensagem em si pode ser compreendida; no segundo a mensagem não é compreendida mas pode ser encaixada na visão já existente, digamos a cura mágica se torna placebo, os barulhos misteriosos são estalos na casa devido a dilatação causada pelo calor, etc; no terceiro caso a própria base interpretativa e outra, então copiamos o interpretador por completo em cima do nosso padrão, por exemplo podemos ver infravermelho com um aparelho que encaixa esse espectro que não veríamos naturalmente dentro do nosso espectro de luz visível (esse campo interage somente com o aparelho e nós somente com ele). Nesse mesmo raciocínio se entende coisas como compilação ("idioma diferente"), layers de compatibilidade ("visão de mundo"), emulação ("diferentes sentidos), etc.

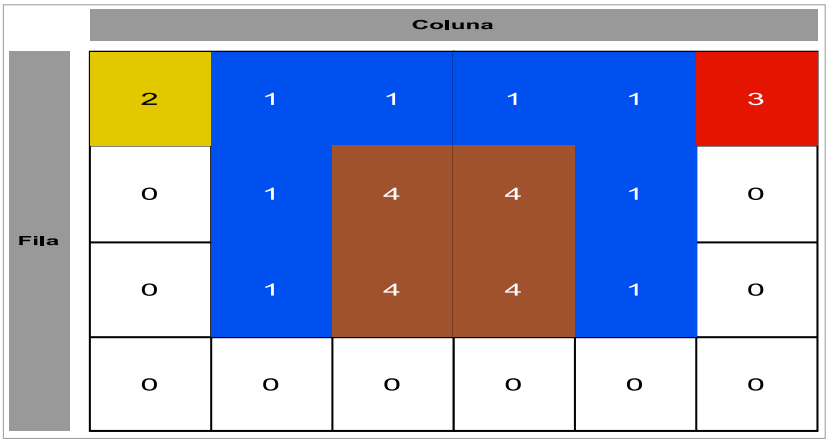
Entendendo tudo isso compreenderá por que é fácil planejar conceitualmente mas na hora de implementar muitos travam (a ideia até ser mapeada está abstrata).



Imagine um jogo como pacman que é bem simples, p'ra muitos fica difícil de conceber como isso pode ser feito a partir de código somente (geralmente isso vale p'ra interfaces gráficas em geral). Com um mapa fica de entender:



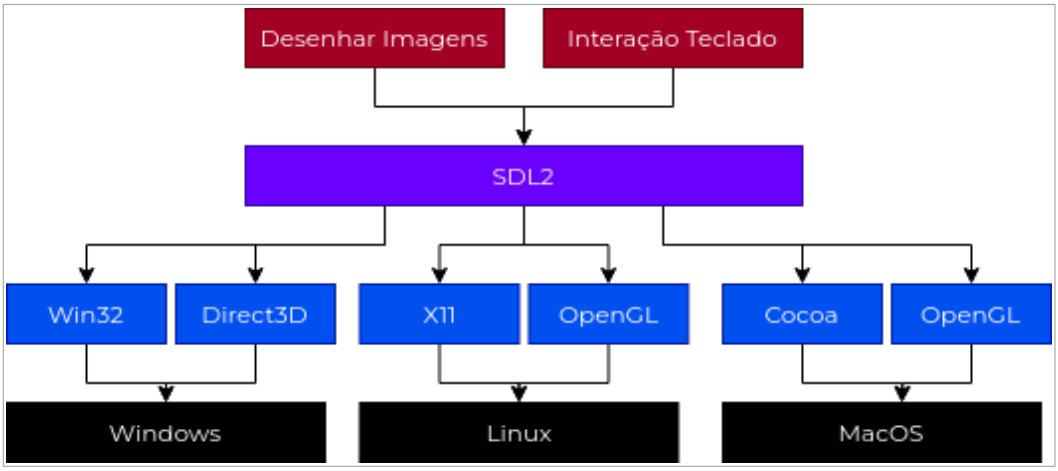
Em baixo nível temos uma matriz bidimensional (uma tabela de linhas e colunas) representando o mapa de forma que cada valor representa um estado (parede, personagem, fantasma, comida), então temos funções p'ra desenhar imagens e captar o pressionar de teclas (ex SDL2) e um sistema de coordenadas que adapta o tamanho da matriz para tela p'ra que finalmente possamos ter uma função que desenha o estado da matriz ou seja a representa visualmente.




```
0. Caminho
1. Parede
2. Personagem
3. Fantasma
4. Comida
```

(Lógica similar pode ser utilizada p'ra interfaces gráfica)

Em cima do conceito de matriz podemos subir o nível, utilizar o teclado e estado da matriz p’ra falar diretamente de movimento, utilizar de numeradores nomeados (“enum”), estruturas ou funções para falar de monstros e comidas, assim fica fácil de avançar pois a cada a campo o dialogo ocorre no seu devido contexto e a logica é isolada em suas respectivas camadas.



O mapa p’ra cada peça do dialogo o faz compreensível quando se expande a ponto de ser autossuficiente (sempre que não flui a implementação há de se perceber que falta um passo entre o campo ao qual o dialogo será escrito e o qual será executado), nisso importa também a camada/reino ao qual ocorre, apesar da distinção pobre (baixo/alto nível) que há na ciência de computação (coloquialmente se usa a mesma distinção[baixo/alto] em todo lugar p’ra compensar), é sabido que há esses níveis de hardware, sistema operacional e software. Na imagem acima temos o “SDL2” (que serve como layer de compatibilidade do ponto de vista do software, isso é, quem conversa com o software e possui diversas “visões de mundo” p’ra conversar com cada sistema operacional) fazendo essa ponte entre o sistema operacional e software (por existir em ambos os níveis), assim estruturando nosso dialogo nele fazemos o mesmo ocorrer completamente a nível de software e nisso está a explicação e razão da portabilidade.

Linguagem de Programação

A linguagem é o vinculo entre o abstrato que é tudo que se pode expressar e o mapa que permite um encontro dentro dessa totalidade, poderia se dizer que programar é como um radio, um processo de sintonia naquilo que se deseja ver/ouvir, pois já era um algo, no sentido de que já estava entre as possibilidades do que poderia se expressar, assim algo encontrado, não criado de fato.

```
“O que foi voltará a ser, o que aconteceu, ocorrerá de novo, o que foi feito se fará outra vez; nada há que seja novo debaixo do sol.”
- Eclesiastes 1:9
```

Então o próximo tópico será a esse respeito, definir um linguagem p’ra que se possa ver os principais conceitos e terminologia que importa a um programador entender e também p’ra que tenha essa sensibilidade de conseguir conceber o que se faz e o que se deseja fazer, uma vez que, sendo uma linguagem inexistente, restará a ti ser o interpretador.

Paradigmas

Os paradigmas são os padrões de mapeamento, como as diferentes formas de dizer uma mesma coisa, uma pessoa lógica diria de um modo, uma pessoa mais mistica embora no fim seja a mesma coisa sendo dita.

Imperativo

O paradigma imperativo é definido na mudança constante de estado e controle de fluxo explicito declarado como uma sequencia de ações (geralmente e erroneamente dado como sinônimo de “algoritmo”).

```
Função fatorial(N) {
     RESULTADO = 1;
     I = 1;
     Enquanto I for menor que N {
          RESULTADO = RESULTADO * I;
     }
     retornar RESULTADO;
}
```

Funcional

O paradigma funcional é definido na imutabilidade e com controle de fluxo explicito com uso de recursão no lugar de iteração (como no exemplo anterior).

```
Função fatorial(N) {
     Se (N == 0) {
          retornar 1;
     } senão {
          retornar (N * fatorial(N - 1));
     }
}
```

Declarativo

O paradigma declarativo é definido em dizer o “que” será feito ao invés de dizer “como”, portanto tem controle de fluxo implícito.

```
Fatorial de 0 é F = F é 1
Fatorial de N é F =
     Onde N é maior que 1
     N1 é (N - 1)
     F1 é o F de Fatorial(N1)
     F é N * F1
```

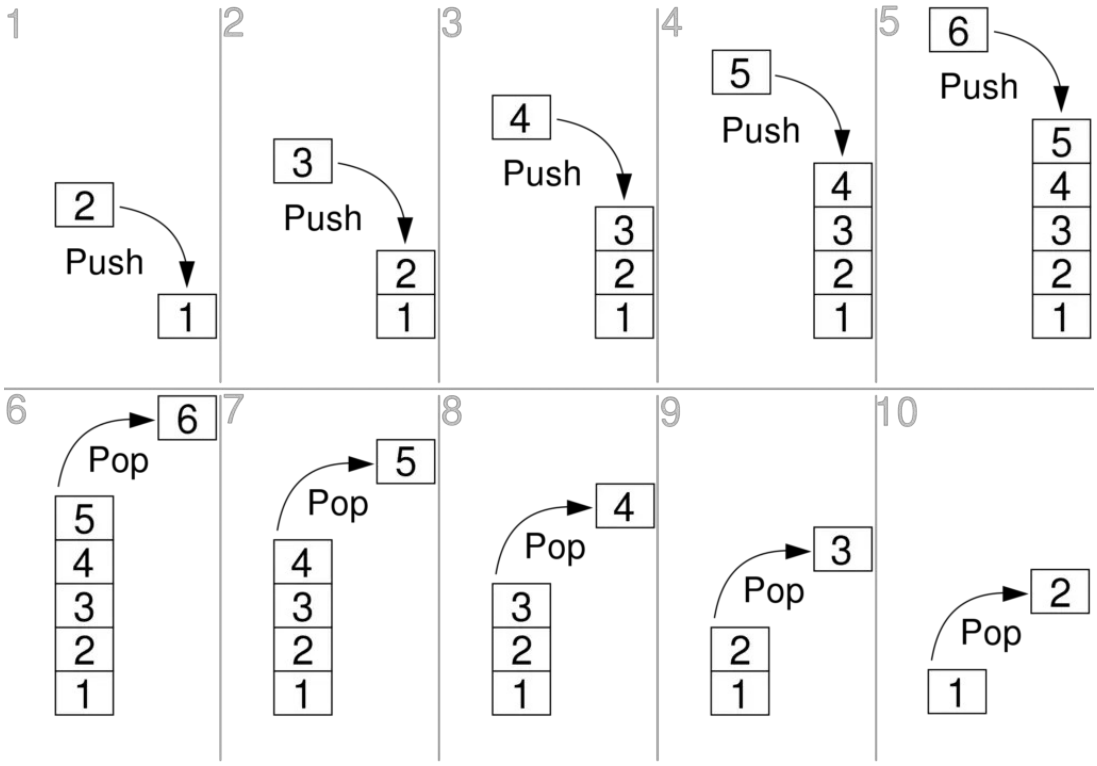
Estrutura de Dados

Qualquer código que há se trata dessa transformação dos dados e nesse processo trabalhamos fazendo conjuntos deles como listas, grafos, etc.

Digamos que a estruturação é o posicionamento desses dados dentro do mapa enquanto os algorítimos fazem o trabalho de interação.

O intuito aqui é terminologia e conceitos sem adentrar muito em detalhes como algorítimos e tipos de estruturas então veremos somente uma que servirá p'ra definir a linguagem, stack.

O stack é um estrutura de dados LIFO (Last In First Out[0]), usada p'ra guardar valores de forma linear e tomá-los de forma previsível, temos então duas operações "pop" e "push", que retiram e adicionam valores respectivamente.



Em nossa linguagem será utilizada de forma implícita, toda palavra posta entra no stack que poderá ser consumida por operações.

[0]: Último a entrar, primeiro a sair

Operações

Em nossa linguagem precisaremos de operações matemáticas simples:

Operação	Símbolo
Soma	+
Subtração	-
Divisão	/
Modulo	%

Essas operações consome dois valores e retornam um valor.

Operação	Símbolo
Igual	=
Maior	>
Menor	<
Maior ou Igual	>=
Menor ou Igual	<=
Negação	!

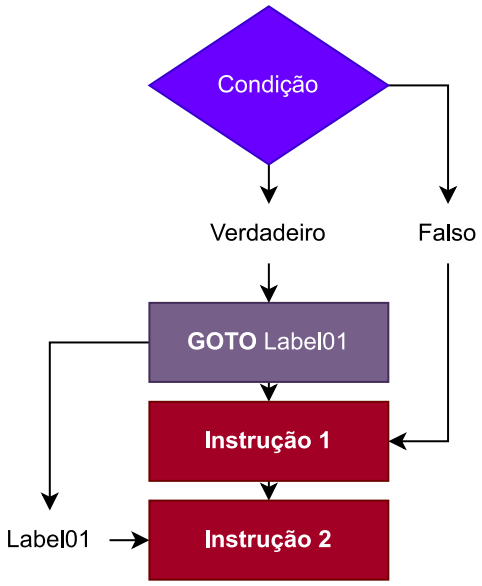
Essas operações consomem dois valores e retornam um valor “booleano”, isso é verdadeiro(1) ou falso(0).

```
# Qualquer texto após esse caractere “#” é um comentário
10 + 10 # (20) Notação convencional
10 10 + # (20) Notação polonesa reversa (stack)
# A notação polonesa será como as funções são utilizadas na nossa
# linguagem.
25 10 * 50 + # 10 * 25 + 50 (500)
50 25 10 * + # 50 + (10 * 25) (500)
10 5 >   # 10 é maior que 5? (1)
10 5 > ! # 10 é maior que 5? (0)
```

Controle de Fluxo

Operações de controle de fluxo servem p’ra ordenar a execução de instruções. Existe controle implícito (como em linguagens declarativas) e explícito (como em linguagens imperativas).

Em nossa linguagem faremos uso de um controle explícito, por meio de somente duas operações, sendo estas “if” e “goto” que servem de condicional e pulo respectivamente.

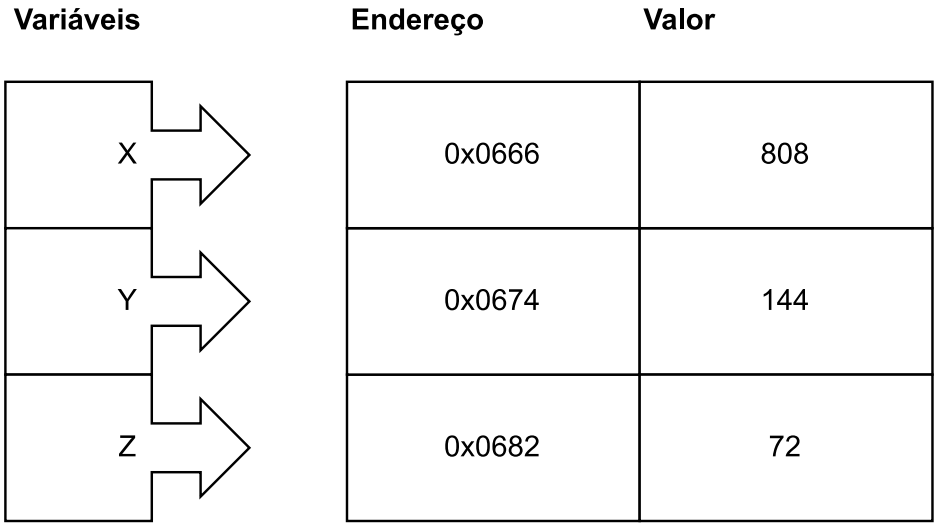


O “if” será uma palavra reservada que deve ser sucedida de dois blocos, a condição e o corpo de instruções, esses devem estar isolados pelos caracteres “{}”.

O “goto” será uma palavra reservada que deve ser sucedida de um “label” (o ponto de pulo), que já deve estar declarado da seguinte forma “nome:”.

Variáveis

As variáveis são referências nomeadas, um ponteiro que leva a um endereço de memória que possui a informação atrelada a ela. Por meio desta referência se torna possível reter e alterar valores (nesse sentido de “referência” se entende coisas como “vazamento de memoria”, que é não conseguir alcançar uma região de memoria que foi pedida p’ra uso devido a ter perdido todas as referências, dessa forma não há como devolvê-la ao sistema).
Qualquer alteração na referência é refletida na região a qual a mesma aponta (Nisso se entende o paradigma “funcional”, cada objeto é uma copia de forma que interagir com um não afeta os outros, portanto não há “efeitos colaterais”).



Em nossa linguagem elas serão declaradas da seguinte forma: “valor → nome”; A operação irá consumir do stack o valor atribuído à variável.

Sintaxe

Sintaxe é o conjunto de regras que determinam as possibilidades de associação das palavras.

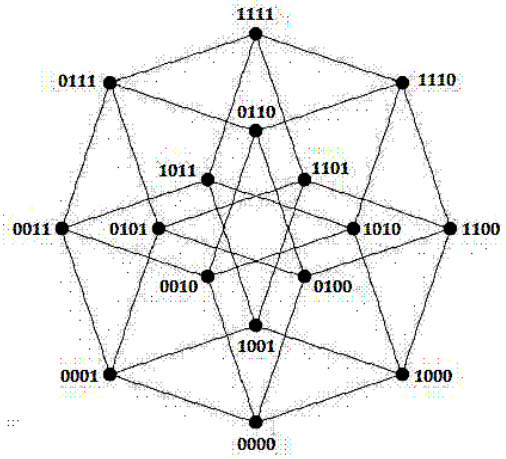
- Mim ser um brasileiro nativo # *Erro de sintaxe*
- Eu sou um brasileiro nativo

No exemplo acima você pode perceber que embora o significado seja o mesmo em ambas as frases, uma delas está incorreta, pois não se pode usar “mim” enquanto se refere ao sujeito da oração.

Com isso em mente vamos ver a sintaxe da nossa linguagem até o momento:

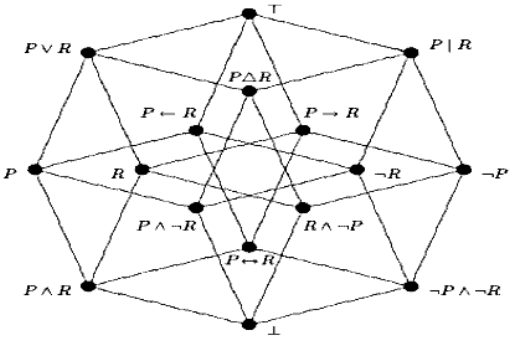
```
# Definições
<numeros> ::= 0-9
<strings> ::= “[A-Za-z_]+”
<instruções> ::= qualquer operação | <numeros> | <strings>
<booleano> ::= valor “booleano”
<comentario> ::= qualquer texto após o caractere “#”
<nome> ::= qualquer palavra (Aa a Zz)
# Regras
  ::= <instruções>
    | if { <booleano> } { <instruções> }
    | goto <nome>
    | <nome>:
```

No modelo das regras “|” significa “OU”. Essa notação é bem similar a chamada “[E]BNF” que serve p’ra descrever a sintaxe de uma linguagem e até p’ra escrever compiladores (como com [b]yacc + [f]lex).



The 16 subsets of a 4-set or the 16 points in the affine 4-space over the two-element field

From 'A Refined Geometry of Logic,' by David Miller -- Slides for a talk given at the Department of Mathematics, University of Warwick, November 2005:



The Lindenbaum–Tarski algebra of a classical propositional language generated by the two variables $\{P, R\}$.

Semântica

Semântica trata do significado e a relação entre significantes.

- Eu fui forçado a quarta-feira
- Ontem dormi verde horas antes de subir a América

Esses são exemplos de erros de semântica, dado que não podemos tirar sentido algum dessas frases.

Vejamos então a parte da semântica em nossa linguagem:

```
if { <condição> } { <expressões> }
# O segundo bloco só é executado se a condição for verdadeira

<nome>:
# O “nome” se torna um ponto de pulo para operações “goto”

goto <label>
# O fluxo de execução é movido ao ponto “nome”

→ <nome>
# O “nome” se torna uma variável e o valor
# atribuido é consumido do stack
```

Como pode ver a semântica, sendo o significado, pode descrever o comportamento de uma linguagem de programação.

Estilo de Código

O estilo de código é um conjunto de regras, além das sintáticas e semânticas, que é utilizado para organizar o código em si (não afeta significado ou .

Esse é um ponto bem subjetivo; o mais importante é ser consistente com um estilo após escolhido.

O estilo de nossa linguagem será o seguinte:

```
# uma atribuição quebra a linha
10 10 + 15 * ! ! → x

# um goto quebra a linha
goto label01

# instruções de bloco estruturadas da seguinte
# forma:
if {x} {
     ”Olá” # indentação dentro de blocos
label01: # labels (pontos de pulo) não são indentados
}
```

Esteja ciente de que um estilo geralmente se aplica somente a linguagem utilizada (embora haja fatores comuns como indentação).

Como Prosseguir

Logo a seguir veremos alguns problemas que podem ser solucionados com a linguagem descrita, a ideia é por em pratica todo o processo descrito com ela, uma vez que não há compiladores/interpretadores você tem de conceber todo o processo na mente.

O foco não é a solução per se, bastaria procurar e p'ra quase qualquer problema encontraria facilmente alguma solução. A ideia é ter esse mapa mental do que está sendo e feito e de como representar isso.

Problemas

O primeiro problema é um comum em entrevistas chamado "FizzBuzz" (supostamente um numero significante de iniciantes tem dificuldade com este), o problema consiste em imprimir os valores de 1 a 100, mas para múltiplos de 3 ou 5 deve-se imprimir as frases "Fizz" e "Buzz" respectivamente; e para múltiplos de 3 e 5 deve-se imprimir a frase "FizzBuzz".

Antes de tudo vamos dissecar o problema, ele consiste da interação de 1 a 100, uma operação aritmética básica (modulo que retorna o resto de uma divisão) e impressão. Ao último consideraremos que "imprimir" é deixar o estado de stack com o que seriam as saídas de impressão (com operações de push p'ra cada string) e utilizando de uma quebra de linha ("\n") p'ra indicar o fim de uma string (mensagem).

```
# FizzBuzz
0 → i
loop:
i 1 + → i
if {i 15 % 0 =} {
     ”FizzBuzz”
     goto next
}
if {i 3 % 0 =} {
     "Fizz"
     goto next
}
if {i 5 % 0 =} {
     "Buzz"
     goto next
}
i
next:
“\n”
if {i 100 <} {
     goto loop
}
```

A solução possui três condicionais para fazer um “push” de cada “string” (mensagem) de acordo com as condições dadas ao problema, se não for nenhuma o próprio numero é dado ao stack (a variável “i”), em sequência um pulo é feito até a última condicional que verifica se o processo já foi feito 100 vezes, não sendo o caso ele é repetido até que tenha sido cumprido essa condição.

Nota: *Surpreendentemente se entendeu e/ou conseguiu resolver este problema está afrente de um número imenso de programadores, até PHD que segundo muitos entrevistadores não conseguem resolver este (e problemas similares) de forma independente. Quem dirá entender matematicamente o porquê da fundação da eletrônica e por consequência a própria programação como apresentado aqui.*

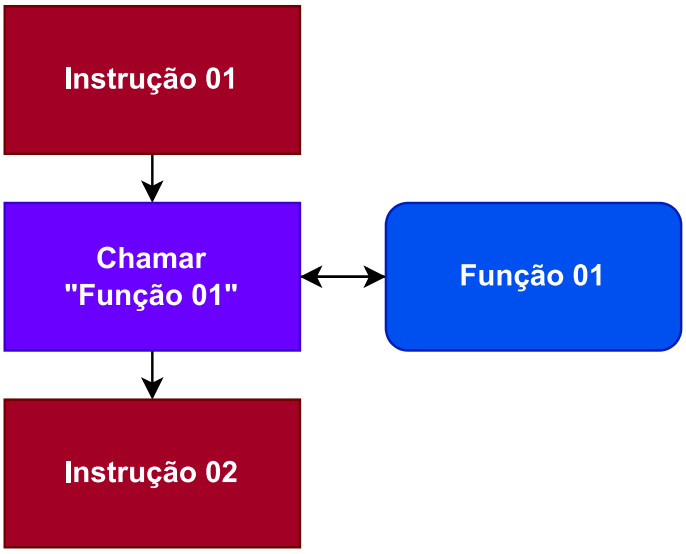
O próximo problema é igualmente simples, desenhar uma pirâmide com tamanho determinado por uma variável, usando os caracteres “*” e “#” como espaço e parte da piramide respectivamente.

```
5 → tamanho
0 → espaço
0 → tmp
loop0:
"*"
if {tmp} {
     tmp 1 - → tmp
     goto loop0
}
tamanho → tmp
loop1:
"#"
if {tamanho} {
     tamanho 1 - → tamanho
     goto loop1
}
"\n"
tmp 2 - → tamanho
espaço 1 + → espaço
espaço → tmp
if {tamanho 0 >=} {
     goto loop0;
}
```

A solução possui duas condicionais que são executadas em um loop até preencher o stack com uma parte da pirâmide e seu espaçamento, em relação a parte anterior; em seguida uma quebra de linha é “impressa” e o loop é executado novamente, com o tamanho da próxima parte reduzido em dois, e isso se repeti até o tamanho ser inferior ou igual a zero.

Funções

As funções são rotinas, como se fosse a estruturação das instruções, temos um bloco que altera o fluxo de execução p'ra si, executando suas instruções e retornando ao fluxo comum.



Na nossa linguagem será adicionado da seguinte forma:

```
# Definições
<argumentos> ::= lista de <nomes> separados por virgula
# Sintaxe + Semântica
func <nome> <argumentos> { <instruções> }
# após declarada, a função pode ser chamada como uma operação,
# que atribui variáveis de acordo com os <argumentos>

return <valor>
# retorna ao fluxo comum de execução e faz push do "valor" no stack
```

Em notação polonesa dá p'ra vê-lo em forma de estrutura como um bloco nomeado, assim como são as operações.

Funções Variádicas

As funções variádicas são como funções comuns, com diferença que podem receber um número indeterminado de argumentos. Não acrescentaremos esse conceito em nossa linguagem, no entanto é bom conhecer.

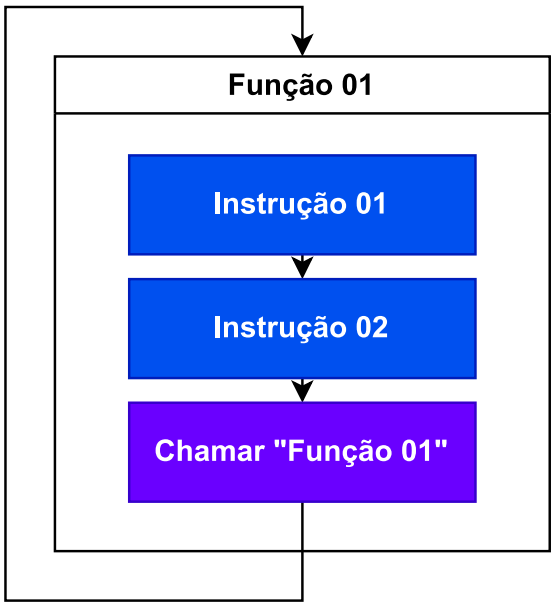
```
Função(Argumentos) # Notação mais comum
Função Argumentos  # Notação polonesa
Argumentos Função  # Notação polonesa reversa

# Função Comum
FC(x, y, z)
FC(10, 20, 30)     # correto
FC(10)             # incorreto
FC(10, 20, 30, 40) # incorreto

# Função variádica
FV(…)              # definição
FV(50)             # correto
FV(10, 20, 30)     # correto
FV(10, 20, 30, 40) # correto
```

Recursão

A recursão é um processo em que um de seus passos envolve a chamada de si mesmo. Em funções são funções que chamam o próprio nome, estruturas que tem como de seus objetos a si mesmo, etc.



2D Rec
+ X: número inteiro
+ Y: número inteiro
+ XY: 2D REC

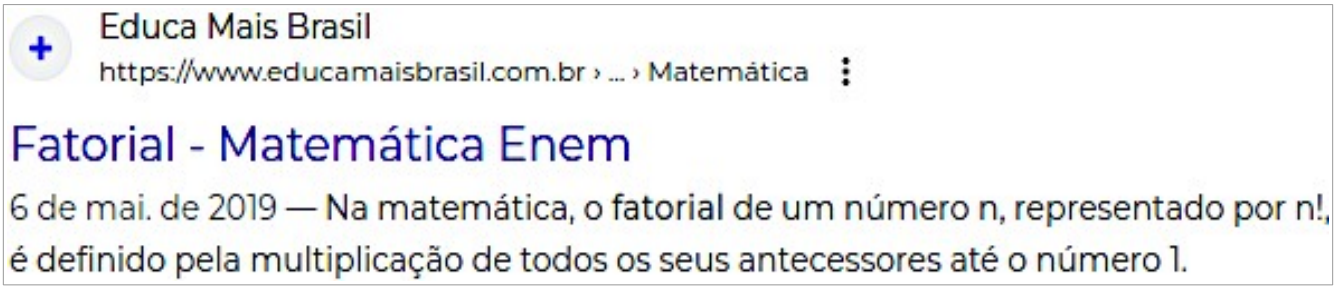
A recursão é um excelente para quebrar um problema, em um monte de partes menores (na lógica do “dividir para conquistar”), e assim possibilitar uma solução mais compreensível para certos problemas que seriam de difícil compreensão de forma iterativa por exemplo (como um “merge sort”).

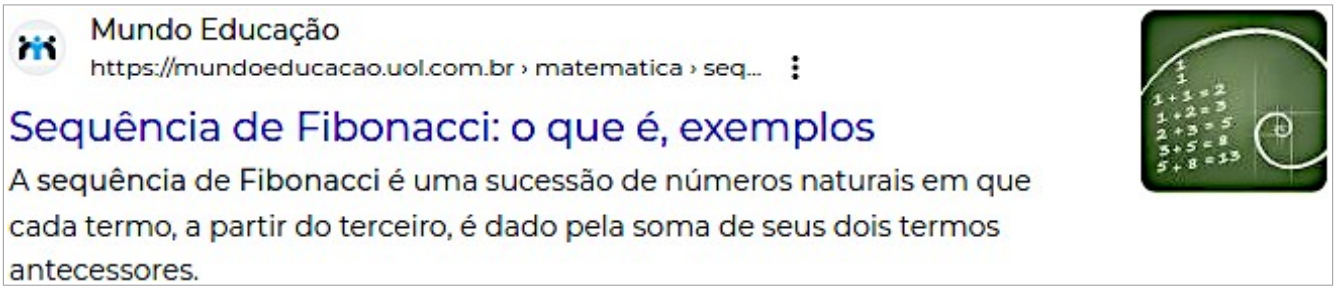
E alguns paradigmas tornam o uso desse modelo essencial, como é o caso do paradigma funcional.

Problemas Recursivos

Vejamos abaixo alguns problemas e suas soluções, essas que deverão se dar de forma recursiva.

1. Crie uma função que retorne o Fatorial de um dado número.
2. Crie uma função que retorne a sequência Fibonacci de um dado número





```
func fatorial x {
     if {x 1 <=} {
          return 1
     }
     # em notação comum
     # x = x * fatorial(x - 1)
     x x 1 - fatorial * -> x
     return x
}
```

Uma característica a notar dessa solução é que a recursão é o último evento da função, o que torna a recursão uma “recursão de calda”, assim permitindo compiladores optimizarem a mesma.

```
func fib x {
     if {x 1 <=} {
          return x
     }
     # em notação comum
     # x = fib(x-1) + fib(x-2)
     x 1 - fib x 2 - fib + -> x
     return x
}
```

Arranjos

Os arranjos, as chamadas “arrays”, são uma estrutura de dados que guarda uma coleção de elementos de forma sequencial na memória, dessa forma permitindo o acesso do mesmo por indexes.

Arranjo Unidimensional com 5 Elementos

Index:	0	1	2	3	4
Valor:	813	21	415	643	248

Essa estrutura tem um bom uso de cache do processador (devido a memória contígua), acesso randômico rápido (por indexes) e tamanho fixo (toda extensão de tamanho requer criar uma nova array e copiar a antiga).

Arrays também podem possuir múltiplas dimensões, que será visto nas linguagens como múltiplos indexes: X[3][2]; cada índex representa uma dimensão.

```
# Definição
<index> ::= index numeral separado por virgulas para cada dimensão

# Sintaxe + Semântica
[<valores>] → <nome>
# consome os “valores” e cria uma array “nome”

<nome>[<index>]
# acessa o “index” da array “nome”

<valor> → <nome>[<index>]
# consome do stack o “valor” e o atribui
# a variável “nome” em “index”
```

Problemas Array

Crie uma função que some todos os elementos de uma array:

```
func soma tab, tamanho {
     0 → total
     0 → idx
loop:
     if {1 idx + tamanho =} {
          return total
     }
     tab[idx] total + → total
     idx 1 + → idx
     goto loop
}
# exemplo de uso
[78, 10, 60] → tabela
tabela 3 soma
```

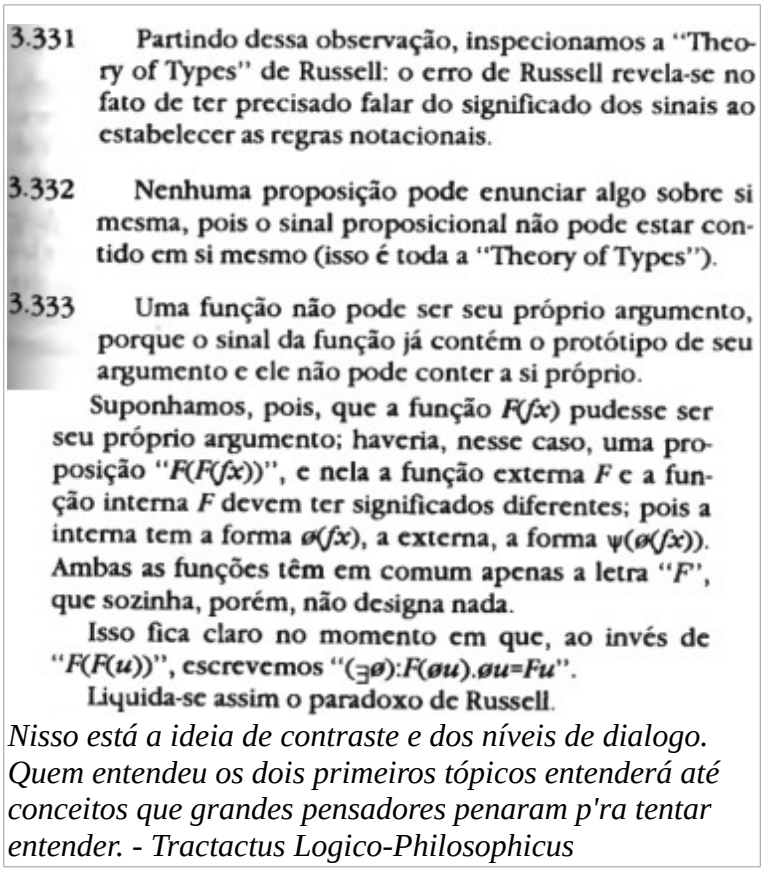
Perceba que a forma como essa função foi feita a torna equivalente a uma função variádica, vantagem dessa notação explicitar diversas equivalências e é bom notá-las pois denota que entende as essências de cada forma.

Declaremos e acessemos matrizes, isso é, arranjos multidimensionais (arranjos de arranjos):

```
[[1, 2], [3, 4]] → x # 2D
x[0, 0] # retorna 1
x[1, 0] # retorna 3
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]] → y # 3D
y[0, 0, 0] # retorna 1
y[0, 1, 1] # retorna 4
y[1, 0, 1] # retorna 6
```

Sistema de Tipos

Um sistema de tipos é um sistema lógico que declara um conjunto de regras que atribuem uma propriedade chamada “tipo” para várias das construções abstratas que fazem parte do programa, como expressões, funções, variáveis ou módulos. Os tipos determinam as regras de interação entre as estruturas, como exemplo uma função de soma aritmética não deveria receber strings (isso é a nível de abstração uma vez que as strings são só sequência de números p’ra representar letras).



3.331 Partindo dessa observação, inspecionamos a "Theory of Types" de Russell: o erro de Russell revela-se no fato de ter precisado falar do significado dos sinais ao estabelecer as regras notacionais.

3.332 Nenhuma proposição pode enunciar algo sobre si mesma, pois o sinal proposicional não pode estar contido em si mesmo (isso é toda a "Theory of Types").

3.333 Uma função não pode ser seu próprio argumento, porque o sinal da função já contém o protótipo de seu argumento e ele não pode conter a si próprio.

Suponhamos, pois, que a função $F(fx)$ pudesse ser seu próprio argumento; haveria, nesse caso, uma proposição "$F(F(fx))$", e nela a função externa F e a função interna F devem ter significados diferentes; pois a interna tem a forma $\varphi(fx)$, a externa, a forma $\psi(\varphi(fx))$. Ambas as funções têm em comum apenas a letra "F", que sozinha, porém, não designa nada.

Isso fica claro no momento em que, ao invés de "$F(F(u))$", escrevemos "$(\exists\varphi):F(\varphi u).\varphi u=Fu$".

Liquida-se assim o paradoxo de Russell.

Nisso está a ideia de contraste e dos níveis de dialogo. Quem entendeu os dois primeiros tópicos entenderá até conceitos que grandes pensadores penaram p'ra tentar entender. - Tractactus Logico-Philosophicus

A checagem de tipos pode se dar de diversas formas: estática contra dinâmica, na primeira forma a tipagem de todos os construtos podem ser verificadas em tempo de compilação (como se fosse uma verificação sintáxica), já na segunda a checagem ocorre em tempo de execução (certos tipos de checagem mais complexos tornam obrigatório esse método) ; manifesto contra inferido, na primeira forma o tipo é dado a um construto de forma explícita, já na segunda o tipo é implícito, declarado a partir do valor dado; nominal contra estrutural, na primeira forma construtos só são equivalentes/compatíveis quando declarada a equivalência de forma explícita ou por partilhar o exato mesmo tipo(nome), já na segunda os construtos podem ser compatíveis se forem estruturados de forma equivalente.

Algorítimos

Um algoritmo é a descrição da solução de um dado problema.

Os algorítimo possuem diversas classificações: classificação de fluxo, serial (um parte por vez até o fim), paralelo (múltiplas partes em execução simultânea) ou distribuído (alternância de execução das partes); classificação de retorno, exato ou heurístico (aproximado); classificação de processo, determinístico ou não-determinístico; classificação de paradigma, força bruta (tentar tudo até encontrar a melhor solução), dividir para conquistar (quebrar o problema), reducionismo (tornar o problema em um problema similar já conhecido), etc; classificação de optimização, linear, dinâmica ou gananciosa.

Também podemos categorizar a complexidade teórica de um algoritmo, tanto para espaço quanto tempo. Para isso usamos a notação “grande O” (com isso tem se noção da escalonabilidade do dado algorítimo):

NOME	CONSTANTE
Constante	O(1)
Logarítmico	O(log n)
Linear	O(n)
Polinomial	O(n^c), c > 1
Exponencial	O(c^n), c > 1
Fatorial	O(n!)

Onde “n” é relacionado ao input e “c” é uma constante (importante notar que a notação não se resumi àquela tabela).

Conclusão

No decorrer da descrição dessa linguagem inexistente tocamos nos pontos mais comuns e basilares da ciência da computação, com proposito de fazer pensar p'ra além da repetição mental e que esses conceitos e terminologias sirvam como "portas" p'ra expressar o conceito central apresentado nos primeiros tópicos, que propõe a fundação e razão p'ra todos estes e, como demonstrado, que seu entendimento resolve desde os problemas mais simples até os mais modernos e complexos. A lógica é universal ainda que varie de acordo com o objetos do mapa de cada um, no entanto, o mapa é sempre o mesmo pois é elementar que assim seja, dessa forma está aqui a essência do que muitos afirmam não existir a "lógica de programação" pois esta nada mais é do que a verbalização que estrutura a realidade se replicando em cima do mineral, nisso também está a razão e explicação da própria razão (linear) que por consequência posiciona a emoção (circular).

O conceito apresentado foi tirado do livro "Lúcifer – Onde a Verdade é a Lei"[0], do autor Bob Navarro, e apresentado de forma matemática p'ra que sua forma seja mais tangível no campo ao qual é proposto aplicá-lo e de forma que seu uso se faça autoevidente; portanto fica implicado que sua leitura irá ajudar a compreender mais profundamente o que foi posto de forma compacta. Lembre-se notar equivalências e ver como tudo se conecta é o que separa quem é bom em expressar o que entende de quem simplesmente consegue replicar o que ouviu, em outras palavras o entendimento da memorização.

No contexto de ir a pratica a linguagem apresentada não é um desperdício, pois qualquer linguagem de stack (como FORTH) será extremamente similar em muitos aspectos, com a vantagem de expressar as palavras de forma muito direta evidenciando equivalências e o processo como verbo.

A quem tiver interesse acompanhe a tentativa do próprio autor de projetar e desenvolver um linguagem: https://eltaninos.org/pt-BR/download/tools/vesper

[0]: https://www.lojalucifer.com/produto/livro-lucifer-onde-a-verdade-e-a-lei-versao-fisica/

Dicas

Sobre Idiomaticidade

As linguagens geralmente possuem estilos de código oficiais, nesses casos faça uso deles e se mantenha consistente com os mesmos, além disso leia código considerado idiomático nessa linguagem para ter ideia de como o seu código deveria se parecer para ter essa qualidade.

Sobre Complexidade

A complexidade desnecessária é a raiz de todos os problemas em programação, então prefira sempre algorítimos/estruturas mais simples e óbvios, e não compare progresso com o número de linhas, que idealmente deve ser reduzido ao máximo possível, sem sacrificar a legibilidade.

Sobre Simplicidade

O problema de muitos é confundir onde importa que essa qualidade exista e confundi-la com minimalismo. É preferível uma interface simples do que uma implementação simples e é preferível uma interface coerente do que uma debilitada pelo minimalismo excessivo ("p'ra que dois braços se já tenho um?"). https://en.wikipedia.org/wiki/Worse_is_better

Sobre Modularidade

A modularidade é uma qualidade valiosa uma vez que aumenta a interoperabilidade de um sistema, no entanto é importante fazer isso de forma racional e perceber quando não é aplicável. Como um exemplo ruim temos o SystemD no Linux (um "init-system") que clama ser modular enquanto suas partes são incapazes de interagir com outras interfaces e dependem uma das outras (em outras palavras não há interoperabilidade).

Sobre Optimização

A optimização é um processo geralmente mal executado, resultando em programas inchados; por via de regra você primeiro escreve o código se preocupando com sua legibilidade e elegância (ou seja, simples e bem estruturado), e após terminado, se e somente se o programa estiver lento (dentro de seu contexto) você o optimiza após mensurar sua execução, assim identificando os "bottlenecks" (pontos de estrangulamento, isso é, os pontos que de fato limitam o desempenho do programa), que geralmente se dão na forma de um uso pobre do cache.

Sobre Linguagens

As linguagens de programação geralmente são de proposito geral, não caia no meme de as comparar com ferramentas (como martelos), onde cada uma tem seu propósito (do contrario seriam “DSL”s); é preferível que conheça uma só linguagem e masterize-a, do que conhecer muitas e não programar bem em nenhuma.

Sobre cópia

Importa que sempre que copiar uma solução de terceiros copie-a manualmente e tente entender a solução; caso não consiga ao menos faça o processo manualmente, eventualmente as coisas começam a fazer sentido naturalmente.

Sobre Estrutura de Dados

A grande maioria dos programas essas estruturas de dados: arranjos, listas encadeadas, tabelas de hash e árvores binárias; são suficiente, e além disso fáceis de implementar e entender.

Sobre Organização

A fim de preservar a manutenibilidade de um projeto é estritamente necessário que o mesmo esteja organizado, para isso deve-se estruturar os arquivos em uma hierarquia de diretórios que faça sentido para o mesmo (por exemplo separar arquivos estáticos do código), além disso é bom considerar usar um versionador (como GIT ou Mercurial) e dessa forma ter um histórico de mudanças, por consequência ganhando maior controle sobre as mesmas.

Sobre Matemática

O conhecimento matemático não é estritamente necessário, embora útil para entender muitos assuntos no campo teórico, então isso não deveria impedi-lo de programar. A quem vem da área de matemática, lembre-se o código é escrito para seres humanos compreenderem, aprenda a utilizar sua linguagem de forma idiomática, o campo prático é geralmente consideravelmente diferente do teórico (há uma arte em expressar o que deseja de forma elegante).