present

Your animal friends

```
using SymEngine     # install with `] add SymEngine`
```

```
rand_animal (generic function with 1 method)
```

```
rand_animal(size...) = Basic.(Symbol.('🐀' .+ rand(0:60, size...)))
```

This is a vector

```
v1 =  [🐮, 🐹, 🐡]
```

```
v1 = rand_animal(3)
```

This is a matrix

```
m1 = 3×3 Matrix{Basic}:
      🐏  🐏  🐌
      🐰  🐢  🐖
      🐬  🐉  🐄
```

```
m1 = rand_animal(3, 3)
```

This is a tensor of rank 4

```
t1 = 2×2×2×2 Array{Basic, 4}:
    [:, :, 1, 1] =
      🐞  🐒
      🐛  🐁

    [:, :, 2, 1] =
      🐈  🐩
      🐝  🐕

    [:, :, 1, 2] =
      🐊  🐐
      🐬  🐔

    [:, :, 2, 2] =
      🐩  🐱
      🐮  🐼
```

```
t1 = rand_animal(2, 2, 2, 2)
```

Matrix-matrix multiplication

```
using OMEinsum
```

This is an einsum notation, defined by `@ein_str` string literal.

```
code_mm = ij, jk -> ik
```

```
code_mm = ein"ij,jk->ik"
```

```
mm_1 = 3×3 Matrix{Basic}:
        🐟   🐝   🐛
        🐝   🐴   🐁
        🐭   🐥   🐹
```

```
mm_1 = rand_animal(3, 3)
```

```
mm_2 = 3×3 Matrix{Basic}:
        🐂   🐧   🐕
        🐖   🐴   🐟
        🐉   🐓   🐪
```

```
mm_2 = rand_animal(3, 3)
```

You can call it

```
3×3 Matrix{Basic}:
 🐂*🐟 + 🐉*🐛 + 🐝*🐖   🐛*🐓 + 🐧*🐟 + 🐴*🐝   🐝*🐟 + 🐟*🐕 + 🐪*🐛
 🐂*🐝 + 🐉*🐁 + 🐴*🐖   🐁*🐓 + 🐧*🐝 + 🐴^2   🐁*🐪 + 🐝*🐕 + 🐴*🐟
 🐂*🐭 + 🐥*🐖 + 🐹*🐉   🐭*🐧 + 🐴*🐥 + 🐹*🐓   🐥*🐟 + 🐭*🐕 + 🐹*🐪
```

```
ein"ij,jk->ik"(mm_1, mm_2)
```

It is equivalent to

```
3×3 Matrix{Basic}:
 🐂*🐟 + 🐉*🐛 + 🐝*🐖   🐛*🐓 + 🐧*🐟 + 🐴*🐝   🐝*🐟 + 🐟*🐕 + 🐪*🐛
 🐂*🐝 + 🐉*🐁 + 🐴*🐖   🐁*🐓 + 🐧*🐝 + 🐴^2   🐁*🐪 + 🐝*🐕 + 🐴*🐟
 🐂*🐭 + 🐥*🐖 + 🐹*🐉   🐭*🐧 + 🐴*🐥 + 🐹*🐓   🐥*🐟 + 🐭*🐕 + 🐹*🐪
```

```
let
    mm_out = zeros(Basic, 3, 3)
    for i=1:3
        for j=1:3
            for k=1:3
                mm_out[i,k] += mm_1[i,j] * mm_2[j, k]
            end
        end
    end
    mm_out
end
```

It can also be constructed as

```
ij, jk -> ik
```

```
EinCode([['i', 'j'], ['j', 'k']], ['i', 'k'])
```

or

```
1∘2, 2∘3 -> 1∘3
```

```
EinCode([[1, 2], [2, 3]], [1, 3])
```

```
n^3
```

```
let n = Basic(:n)
    # NOTE: `flop` counts the number of iterations!
    flop(code_mm, Dict('i'=>n, 'j'=>n, 'k'=>n))
end
```

or, for convenience

```
n^3
```

```
flop(code_mm, uniformsize(code_mm, Basic(:n)))
```

This is summation

```
sum_1 =  [🐣, 🐷, 🐞]
```

```
sum_1 = rand_animal(3)
```

```
0-dimensional Array{Basic, 0}:
🐞 + 🐣 + 🐷
```

```
ein"i->"(sum_1)
```

```
sum_2 = 2×2×3 Array{Basic, 3}:
        [:, :, 1] =
         🐢  🐺
         🐻  🐓

        [:, :, 2] =
         🐔  🐓
         🐓  🐳

        [:, :, 3] =
         🐪  🐺
         🐣  🐄
```

```
sum_2 = rand_animal(2, 2, 3)
```

```
 [🐓 + 🐢 + 🐺 + 🐻, 2*🐓 + 🐔 + 🐳, 🐄 + 🐣 + 🐪 + 🐺]
```

```
ein"ijk->k"(sum_2)
```

```
n^3
```

```
flop(ein"ijk->k", uniformsize(ein"ijk->k", Basic(:n)))
```

Repeating a vector

```
rp_1 =  [🐉, 🐌, 🐆]
```

```julia
rp_1 = rand_animal(3)
```

```
3×4 Matrix{Basic}:
 🐉  🐉  🐉  🐉
 🐌  🐌  🐌  🐌
 🐆  🐆  🐆  🐆
```

```julia
ein"i->ij"(rp_1; size_info=Dict('j'=>4))
```

```
3×4 Matrix{Basic}:
 🐉  🐉  🐉  🐉
 🐌  🐌  🐌  🐌
 🐆  🐆  🐆  🐆
```

```julia
let
    rp_out = zeros(Basic, 3, 4)
    for i=1:3
        for j=1:4
            rp_out[i, j] += rp_1[i]
        end
    end
    rp_out
end
```

Star contraction

```
star_1 = 2×2 Matrix{Basic}:
          🐯  🐬
          🐮  🐆
```

```julia
star_1 = rand_animal(2, 2)
```

```
star_2 = 2×2 Matrix{Basic}:
          🐋  🐦
          🐣  🐻
```

```julia
star_2 = rand_animal(2, 2)
```

```
star_3 = 2×2 Matrix{Basic}:
          🐤  🐓
          🐺  🐣
```

```julia
star_3 = rand_animal(2, 2)
```

```
2×2×2 Array{Basic, 3}:
[:, :, 1] =
 🐋*🐯*🐤 + 🐺*🐮*🐣  🐯*🐤*🐦 + 🐺*🐻*🐮
 🐋*🐬*🐤 + 🐺*🐆*🐣  🐬*🐤*🐦 + 🐺*🐻*🐆

[:, :, 2] =
 🐮*🐣^2 + 🐋*🐯*🐔  🐯*🐦*🐔 + 🐻*🐮*🐣
 🐆*🐣^2 + 🐋*🐬*🐔  🐬*🐦*🐔 + 🐻*🐆*🐣
```

```julia
ein"ai, aj, ak->ijk"(star_1, star_2, star_3)
```


```
2×2×2 Array{Basic, 3}:
[:, :, 1] =
 🐋*🐯*🐤 + 🐺*🐮*🐣  🐯*🐤*🐦 + 🐺*🐻*🐮
 🐋*🐬*🐤 + 🐺*🐆*🐣  🐬*🐤*🐦 + 🐺*🐻*🐆

[:, :, 2] =
 🐮*🐣^2 + 🐋*🐯*🐔  🐯*🐦*🐔 + 🐻*🐮*🐣
 🐆*🐣^2 + 🐋*🐬*🐔  🐬*🐦*🐔 + 🐻*🐆*🐣
```


```julia
let
    star_out = zeros(Basic, 2, 2, 2)
    for i=1:2
        for j=1:2
            for k=1:2
                for a=1:2
                    star_out[i, j, k] += star_1[a,i] * star_2[a,j] * star_3[a,k]
                end
            end
        end
    end
    star_out
end
```

Automatic differentiation

```julia
using Zygote
```

```julia
a, b = randn(2, 2), randn(2);
```

```
2×2 Matrix{Float64}:
 0.639865  -0.597905
 0.639865  -0.597905
```

```julia
Zygote.gradient(x->ein"i->"(ein"ij,j->i"(x, b))[], a)[1]
```

```
let
    A, B, C = randn(2,2,2,2), randn(2,2,2,2,2), randn(2,2,2,2,2)
    size_dict = uniformsize(ein"ijkl,lmkcd,asedf->dfas", 2)
    O = ein"ijkl,lmkcd,asedf->dfas"(A, B, C; size_info=size_dict)

    Ō = randn(2,2,2,2)
    # exchange input/output labels and tensors
    Ā = ein"dfas,lmkcd,asedf->ijkl"(Ō, B, C; size_info=size_dict)
    B̄ = ein"ijkl,dfas,asedf->lmkcd"(A, Ō, C; size_info=size_dict)
    C̄ = ein"ijkl,lmkcd,dfas->asedf"(A, B, Ō; size_info=size_dict)
    Ā, B̄, C̄
end;
```

Speed up your code with GPU

- step 1: import CUDA library.
- step 2: upload your array to GPU with `CuArray` function.

```
using CUDA
```

CUDA error: initialization error (code 3, ERROR_NOT_INITIALIZED)

```
1. throw_api_error(::CUDA.cudaError_enum) @ error.jl:91
2. macro expansion @ error.jl:101 [inlined]
3. cuDeviceGet @ call.jl:26 [inlined]
4. CuDevice @ devices.jl:16 [inlined]
5. TaskLocalState @ state.jl:50 [inlined]
6. task_local_state!() @ state.jl:73
7. active_state @ state.jl:106 [inlined]
8. #_alloc#174 @ pool.jl:183 [inlined]
9. #alloc#173 @ pool.jl:173 [inlined]
10. alloc @ pool.jl:169 [inlined]
11. CUDA.CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}(::UndefInitializer, ::Tuple{Int64,
    Int64}) @ array.jl:44
12. CuArray @ array.jl:290 [inlined]
13. CuArray @ array.jl:295 [inlined]
14. CuArray @ array.jl:304 [inlined]
15. top-level scope @ Local: 2 [inlined]
```

```
let
    cuarr1, cuarr2 = CuArray(randn(2, 2)), CuArray(randn(2))
    result = ein"ij,j->i"(cuarr1, cuarr2)
    typeof(result)
end
```

Note: if you do not have a Nvidia GPU, the above code will give you an error, please do not panic.

Summary

- Einsum can be defined as: iterating over unique indices, accumulate product of corresponding input tensor elements to the output tensor.
- Einsum's representation power
  - ein"ij,jk->ik" is matrix multiplication
  - ein"i->" and ein"ijk->k" is summation
  - ein"i->ij" is repeating axis
  - ein"ai,aj,ak->ijk" is a star contraction
- The time complexity of an einsum notation is $O(n^{(\#\ of\ unique\ labels)})$
- Features in OMEinsum
  - Automatic differentiation
  - GPU
  - Generic programming

Contraction order matters

Multiplying a sequence of matrices

```
code_seq_1 = ij, jk, kl, lm -> im
```

```julia
code_seq_1 = ein"ij,jk,kl,lm->im"
```

```
seq_1 = 2×2 Matrix{Basic}:
         🐑   🐷
         🐂   🐏
```

```julia
seq_1 = rand_animal(2,2)
```

```
seq_2 = 2×2 Matrix{Basic}:
         🐕   🐚
         🐊   🐟
```

```julia
seq_2 = rand_animal(2,2)
```

```
seq_3 = 2×2 Matrix{Basic}:
         🐇   🐱
         🐀   🐇
```

```julia
seq_3 = rand_animal(2,2)
```

```
seq_4 = 2×2 Matrix{Basic}:
         🐓  🐏
         🐀  🐊
```

```
seq_4 = rand_animal(2,2)
```

```
2×2 Matrix{Basic}:
 🐀*🐚*🐑*🐓 + 🐀*🐰*🐚*🐑 + 🐰*🐑*🐓*🐕 + 🐱*🐀*🐑*🐕 + 🐱*🐷*🐊*🐀 + 🐷*🐀*🐟*🐓 + 🐷*
 🐂*🐀*🐚*🐓 + 🐂*🐀*🐰*🐚 + 🐂*🐰*🐓*🐕 + 🐊*🐏*🐰*🐓 + 🐏*🐀*🐟*🐓 + 🐏*🐀*🐰*🐟 + 🐱*
```

```
ein"ij,jk,kl,lm->im"(seq_1, seq_2, seq_3, seq_4)
```

```
n^5
```

```
flop(code_seq_1, uniformsize(code_seq_1, Basic(:n)))
```

```
code_seq_2 = ik, mk -> im
             ├─ kl, lm -> mk
             │  ├─ lm
             │  └─ kl
             └─ ij, jk -> ik
                ├─ jk
                └─ ij
```

```
code_seq_2 = ein"(ij,jk),(kl,lm)->im"
```

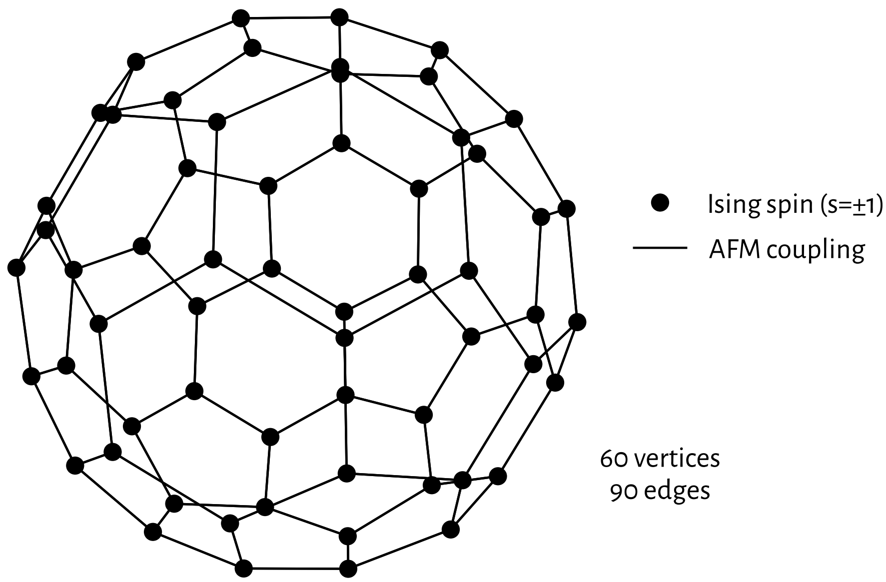
```
3*n^3
```

```
flop(code_seq_2, uniformsize(code_seq_2, Basic(:n)))
```

The Song Shan Lake Spring School (SSSS) Challenge

Song Shan Lake Spring School Github

In 2019, Lei Wang, Pan Zhang, Roger and me released a challenge in the Song Shan Lake Spring School, the one gives the largest number of solutions to the challenge quiz can take a macbook home (**@LinuxDaFaHao**). Students submitted many **solutions to the problem**. The second part of the quiz is



θ =

φ =

In the Buckyball structure shown in the figure, we attach an ising spin $s_i = \pm 1$ on each vertex. The neighboring spins interact with an anti-ferromagnetic coupling of unit strength. Count the degeneracy of configurations that minimizes the energy

$$E(\{s_1, s_2, \ldots, s_n\}) = \sum_{i,j \in edges} s_i s_j$$

```
# returns atom locations
function fullerene()
    φ = (1+√5)/2
    res = NTuple{3,Float64}[]
    for (x, y, z) in ((0.0, 1.0, 3φ), (1.0, 2 + φ, 2φ), (φ, 2.0, 2φ + 1.0))
        for (α, β, γ) in ((x,y,z), (y,z,x), (z,x,y))
            for loc in ((α,β,γ), (α,β,-γ), (α,-β,γ), (α,-β,-γ), (-α,β,γ), (-α,β,-γ),
(-α,-β,γ), (-α,-β,-γ))
                if loc ∉ res
                    push!(res, loc)
                end
            end
        end
    end
    return res
end;
```

```
c60_xy = fullerene();
```

```
c60_edges = [[i,j] for (i,(i2,j2,k2)) in enumerate(c60_xy), (j,(i1,j1,k1)) in
enumerate(c60_xy) if i<j && (i2-i1)^2+(j2-j1)^2+(k2-k1)^2 < 5.0];
```

```
c60_code =
1∘3, 2∘4, 5∘7, 6∘8, 9∘10, 11∘12, 13∘17, 14∘18, 15∘19, 16∘20, 21∘22, 23∘24, 25∘26, 27∘28, 2
```

```
c60_code = EinCode(c60_edges, Int[])
```

```
90
```

```
length(getixsv(c60_code))  # number of input tensors
```

```
[]
```

```
getiyv(c60_code)  # labels for the output tensor
```

```
60
```

```
length(uniquelabels(c60_code))  # number of unique labels
```

```
n^60
```

```
flop(c60_code, uniformsize(c60_code, Basic(:n)))
```

Find a good contraction order

```
using OMEinsumContractionOrders
```

```
c60_optcode =
 SlicedEinsum(Slicing([]), 21∘22, 21∘22 ->
                           ├─ 21∘45, 21∘45∘22 -> 21∘22
                           │  ├─ 29∘45∘9∘22, 29∘9∘21 -> 21∘45∘22
                           │  │  ├─ 29∘53, 9∘53∘21 -> 29∘9∘21
                           │  │  │  ├─ 9∘53, 21∘53 -> 9∘53∘21
                           │  │  │  │  ├─ 21∘53
                           │  │  │  │  └─ 9∘53
                           │  │  │  └─ 29∘53
                           │  │  └─ 19∘3∘20∘6∘45∘22∘38∘40∘29, 3∘19∘6∘29∘9∘20∘40∘38∘22 -> 2
                           │  │     ├─ 23∘6∘20∘40∘9∘38∘22, 3∘19∘23∘6∘29∘9 -> 3∘19∘6∘29∘9∘2
                           │  │     │  ├─ 15∘19∘23∘6, 15∘3∘29∘9∘23 -> 3∘19∘23∘6∘29∘9
                           │  │     │  │  ⋮
                           │  │     │  │
                           │  │     │  └─ 9∘56∘38∘40∘22, 56∘23∘6∘20∘40 -> 23∘6∘20∘40∘9∘38∘
                           │  │     │     ⋮
                           │  │     │
                           │  │     └─ 11∘33∘19∘3∘20∘60∘6, 33∘11∘45∘22∘38∘60∘20∘40∘29∘3 ->
                           │  │        ├─ 49∘45∘29∘3∘33, 49∘33∘11∘45∘22∘38∘60∘20∘40 -> 33∘
                           │  │        │  ⋮
                           │  │        │
                           │  │        └─ 11∘27∘33∘3∘19, 20∘27∘60∘19∘6 -> 11∘33∘19∘3∘20∘60
                           │  │           ⋮
                           │  │
                           │  └─ 21∘45
                           └─ 21∘22
```

```
# optimize use the `TreeSA` optimizer
c60_optcode = optimize_code(c60_code, uniformsize(c60_code, 2), TreeSA())
```

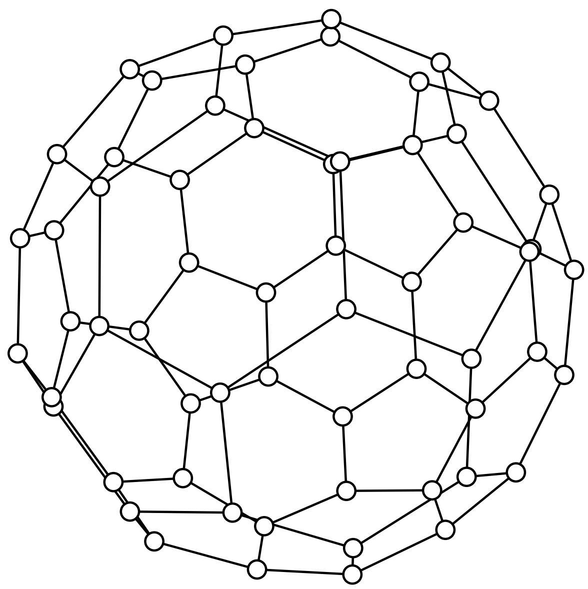
c60_elimination_order =

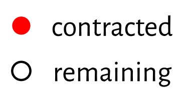
[59, 43, 35, 28, 52, 51, 8, 27, 41, 1, 13, 17, 37, 25, 57, 44, 36, 2, 42, 4, more ,20, 6,

```
c60_elimination_order = OMEinsum.label_elimination_order(c60_optcode)
```

contraction step = 0

The resulting contraction order produces time complexity = n^2 + 30*n^3 + 27*n^4 + 12*n^5 + 7*n^6 + 3*n^7 + 3*n^8 + 4*n^10 + n^11 + n^12





The partition function

Z = 1.1667408970547074e34

```
Z = c60_optcode([(J = 1.0; β = 1.0; expJ = exp(β*J); [1/expJ expJ; expJ 1/expJ]) for i=1:90]...)[]
```

1.3073684577607942

```
log(Z) / 60
```