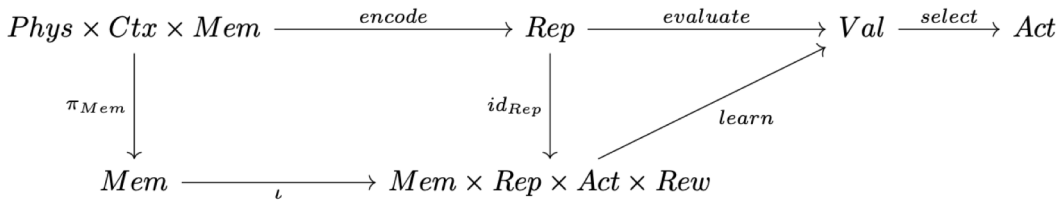
「心の計算」最少モデル



上段：3層アーキテクチャ（心 → 意識 → 行動）

1. `encode : Phys × Ctx × Mem → Rep`
   - 生理状態（Phys）、状況（Ctx）、既存の記憶（Mem）から、
     その瞬間の **心の内的表象 Rep** を作る射。
   - FEP-lite の「内部信念更新」に対応。
2. `evaluate : Rep → Val`
   - 内部表象 Rep から、
     いくつかの行動候補に対する **価値ベクトル Val** を計算する射。
   - MDP-lite の Q 値や、情動を含んだ「評価」。
3. `select : Val → Act`
   - 価値ベクトル Val から、**1つの行動／判断 Act** を選ぶ射。
   - winner-take-all や softmax に対応。
   - ここが **極小意識モデルそのもの**（意識のゲート）。

下段：学習（記憶更新）はどこで起きるか？

4. `π_{Mem} : Phys × Ctx × Mem → Mem`
   - 入力から純粋に「今の記憶 Mem だけを取り出す」射（単なる射影）。
5. `ι : Mem → Mem × Rep × Act × Rew`
   - 「記憶 Mem を、今の Rep・Act・Rew と一緒に学習装置に渡す」射。
   - 実装的には「タプルに突っ込むだけ」の埋め込み（hook）。
6. `learn : Mem × Rep × Act × Rew → Mem`
   - 学習（記憶更新）そのものを表す射。
   - 「この状態 Rep でこの行動 Act をして、この結果 Rew だった」
     という経験を使って Mem を少し変える。
   - ここは **心（表象層）の内部で行われる学習処理**。

```
(* =========================*)
(*Basic settings*)
(* =========================*)
ClearAll[nPhys, nCtx, nMem, nRep, nAct, actions, wEval, encode,
  evaluate, select, piMem, embedMRAR, learn, rewardFunction, stepOnce];

(*次元の設定：必要なら後で変えてOK*)
nPhys = 2; (*Phys の次元*)
nCtx = 2; (*Ctx の次元*)
nMem = 2; (*Mem の次元*)
nRep = nPhys + nCtx + nMem;
nAct = 3; (*行動の種類*)

(*行動ラベル*)
actions = {"Avoid", "Approach", "Wait"};

(*評価用の重み（例としてランダム）*)
SeedRandom[1];
wEval = RandomReal[{-1, 1}, {nRep, nAct}];
(* =========================*)
(*1. encode:Phys×Ctx×Mem→Rep*)
(* =========================*)
encode[phys_List, ctx_List, mem_List] /; Length[phys] == nPhys &&
    Length[ctx] == nCtx && Length[mem] == nMem := Join[phys, ctx, mem];

(* =========================*)
(*2. evaluate:Rep→Val*)
(*Val は「各行動のスカラー価値」のリスト*)
(* =========================*)

evaluate[rep_List] /; Length[rep] == nRep :=
  rep.wEval; (*長さ nAct のベクトル*)

(* =========================*)
(*3. select:Val→Act*)
(*Val を受け取って「1つの行動」を選ぶ*)
(* =========================*)

select[val_List] /; Length[val] == nAct :=
  Module[{idx},
   idx = First@Ordering[val, -1]; (*ArgMax*)
   <|"Index" → idx, "Label" → actions[[idx]]|>];

(* =========================*)
(*4. π_Mem:Phys×Ctx×Mem→Mem*)
(*生理・文脈からの「素の」記憶更新（前処理）*)
```

```
(* =========================*)

piMem[phys_List, ctx_List, mem_List] /;
    Length[phys] == nPhys && Length[ctx] == nCtx && Length[mem] == nMem :=
  Module[{drift}, (*例：Ctx の平均に応じたドリフトを少し加えるだけの素朴モデル*)
   drift = 0.1 Mean[ctx];
   mem + drift
  ];

(* =========================*)
(*5. ℒ:Mem→Mem×Rep×Act×Rew*)
(*MRAR=Mem×Rep×Act×Rew に埋め込む役割*)
(*ここでは act,rew は引数で受け取る形にする*)
(* =========================*)
embedMRAR[mem_List, rep_List, act_, rew_] :=
  {mem, rep, act, rew};

(* =========================*)
(*6. learn:Mem×Rep×Act×Rew→Mem*)
(*実際の学習規則（ここでは超単純な TD もどき）*)
(* =========================*)
learn[mem_List, rep_List, actAssoc_Association, rew_?NumericQ] /;
    Length[mem] == nMem && Length[rep] == nRep :=
  Module[{alpha = 0.3, idx, tdError},
   idx = actAssoc["Index"];
   (*rep と行動インデックスから「期待価値」を作る例（かなり素朴）*)
   tdError = rew - (rep.wEval)〚idx〛;
   (*mem の一部を誤差方向に更新するだけの極小モデル*)
   mem + alpha tdError Table[1., {nMem}]
  ];

(* =========================*)
(*報酬関数 Rew:Phys×Ctx×Act→R*)
(*例：Approach が報酬を得やすいが、Phys/Ctx で変動*)
(* =========================*)
rewardFunction[phys_List, ctx_List, actAssoc_Association] :=
  Module[{idx = actAssoc["Index"], base, arousal, difficulty},
   arousal = phys〚1〛;
   difficulty = ctx〚1〛;
   base = Which[
     idx == 1, -0.2 + 0.3 difficulty, (*Avoid*)
     idx == 2, 0.5 + 0.5 arousal - 0.3 difficulty, (*Approach*)
     idx == 3, 0.1 - 0.2 arousal, (*Wait*)
     True, 0.];
   base
  ];
```

```
(* ========================*)
(*1ステップの「心→意識→行動＋学習」ループ*)
(* ========================*)stepOnce[state_Association] :=
  Module[{phys, ctx, mem, rep, val, act, memPrior, rew, memNew, mrar},

   (*現在の状態を取り出し*)
   phys = state["Phys"];
   ctx = state["Ctx"];
   mem = state["Mem"];

   (*上段：心→意識→行動*)
   rep = encode[phys, ctx, mem];
   val = evaluate[rep];
   act = select[val];

   (*下段：記憶の前更新 π_Mem*)
   memPrior = piMem[phys, ctx, mem];

   (*報酬の計算*)
   rew = rewardFunction[phys, ctx, act];

   (*MRAR への埋め込み ι*)
   mrar = embedMRAR[memPrior, rep, act, rew];

   (*learn による最終的な記憶更新*)
   memNew = learn @@ mrar;

   <|
    "Phys" → phys,
    "Ctx" → ctx,
    "Mem" → memNew,
    "Rep" → rep,
    "Val" → val,
    "Act" → act,
    "Rew" → rew|>
  ];
```

In[ ]:=
```
(* ========================*)
(*初期状態の例とテスト*)
(* ========================*)
initialState = <|
   "Phys" → {0.5, 0.2}, (*例：適度な覚醒、やや低い疲労 など*)
   "Ctx" → {0.3, 0.7}, (*例：課題難度0.3,社会的圧力0.7 など*)
   "Mem" → {0., 0.}    (*例：ニュートラルな記憶状態*)|>;
```

In[•]:=
```
(*1ステップ回してみる*)
step1 = stepOnce[initialState]
```
Out[•]=
```
⟨| Phys → {0.5, 0.2}, Ctx → {0.3, 0.7}, Mem → {-0.132557, -0.132557},
 Rep → {0.5, 0.2, 0.3, 0.7, 0., 0.}, Val → {0.498522, -1.05679, 0.401545},
 Act → ⟨| Index → 1, Label → Avoid |⟩, Rew → -0.11 |⟩
```

In[•]:=
```
step1["Act"]
```
Out[•]=
```
⟨| Index → 1, Label → Avoid |⟩
```

In[•]:=
```
step1["Rew"]
```
Out[•]=
```
-0.11
```

In[•]:=
```
step1["Mem"]
```
Out[•]=
```
{-0.132557, -0.132557}
```

```
step1["Val"](* actions={"Avoid","Approach","Wait"}*)
```
Out[•]=
```
{0.498522, -1.05679, 0.401545}
```

葛藤度の可視化

In[•]:=
```
(*Val={v1,v2,v3,...}*)
ClearAll[conflictMargin, conflictEntropy];

(*1位と2位の差*)
conflictMargin[val_List] /; Length[val] ≥ 2 :=
  Module[{sorted},
   sorted = Reverse@Sort[val];
   sorted[[1]] - sorted[[2]]
  ];

conflictMargin[val_List] /; Length[val] < 2 := 0.;

(*softmax エントロピー*)
conflictEntropy[val_List, beta_ : 3.0] :=
  Module[{z, p},
   z = Exp[beta val];
   p = z / Total[z];
   -Total[p Log[p]]
  ];
```

In[•]:=
```
ClearAll[traceConflict];

traceConflict[state0_Association, T_Integer?Positive, beta_ : 3.0] :=
  Module[{state = state0, log},

   log = Reap[
     Do[
      state = stepOnce[state];
      With[{val = state["Val"], act = state["Act"], rew = state["Rew"]},
       Sow[
        <|
         "t" → t,
         "Val" → val,
         "Margin" → conflictMargin[val],
         "Entropy" → conflictEntropy[val, beta],
         "ActLabel" → act["Label"],
         "Rew" → rew
         |>
       ]
      ],
      {t, 1, T}
     ]
    ];

   (*Reap の戻り値から Sow されたリストだけ取り出す*)
   If[Length[log[[2]]] > 0, log[[2, 1]], {}]
  ];
```

In[•]:=
```
data = traceConflict[initialState, 50]; (*50ステップ回す*)

Length[data]  (* →50*)
First[data]  (*1ステップ目のログ内容を確認*)
```

Out[•]=
```
50
```

Out[•]=
```
<| t → 1, Val → {0.498522, -1.05679, 0.401545},
 Margin → 0.0969772, Entropy → 0.712373, ActLabel → Avoid, Rew → -0.11 |>
```

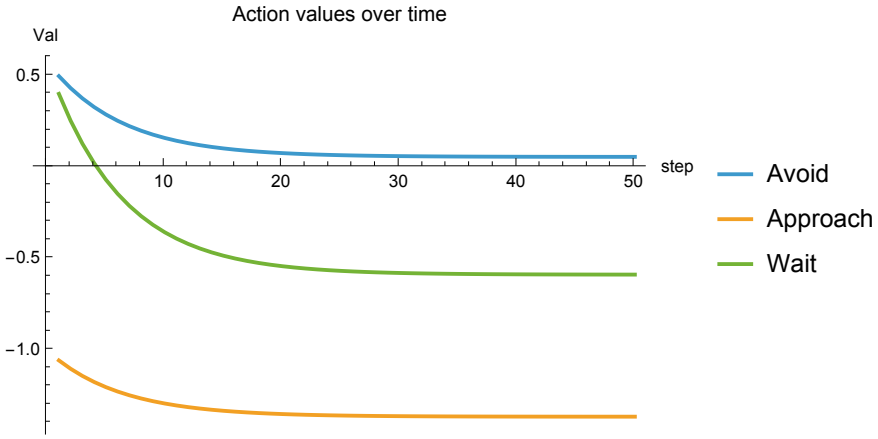
各行動の Val をプロット

In[ ]:=
```
vals = data[[All, "Val"]];   (*各ステップの {v1,v2,v3}*)

valSeries = Transpose[vals];  (*{全ステップのv1列,v2列,v3列}*)

ListLinePlot[
 valSeries,
 PlotLegends → actions,
 AxesLabel → {"step", "Val"},
 PlotLabel → "Action values over time"
]
```

Out[ ]=



葛藤度（margin or entropy）のプロット

```
margins = data[[All, "Margin"]];
entropies = data[[All, "Entropy"]];

ListLinePlot[
 {margins, entropies},
 PlotLegends → {"Margin (small = conflict)", "Entropy (large = conflict)"},
 AxesLabel → {"step", "conflict"},
 PlotLabel → "Conflict over time"
]
```

Out[ ]=