Alloy で『白と黒のとびら』

はじめに

書籍『白と黒のとびら』は、オートマトンと形式言語の基本を学べる物語である
パズル仕立てになっており、ルールに則った装置の中はどうなっているのか、主人公と一緒に考え解き明かしていく
せっかくなので、読み進めながら軽量形式手法ツール Alloy の使い方を練習してみよう

学問的な知識が足りないため、この資料ではとっつきやすさを求めて、あえて正しい用語を使わない部分が多い
わかっている人が見るとツッコミどころだらけなはず
わかっている人は正しい道をどんどん進むがよい
そして素人にわかりやすく解説してくれるとありがたい

Visual Studio Code に Alloy プラグインがあるので、それを使って試している
また、この資料自体も VS Code 上で Markdown Preview Enhanced プラグインを使って書いている
文章内に埋め込んだ Alloy ソースをいつでも実行できるため、見直し中に疑問が出たらすぐ確認できてとても便利

理解が進むにつれてより洗練された書き方ができるようになるが、その時の自分が理解できていた範囲という点を尊重し、ソースに関しては誤りでない限りなるべく修正せず掲載する方針とした

(参考書籍)

- 川添 愛: 白と黒のとびら
- Daniel Jackson: 抽象によるソフトウェア設計 —Alloyではじめる形式手法

第2章 帰郷：灯台の地下 — はじめての Alloy

実家に帰った主人公は、祖父が暮らしていた「灯台」の地下にある神殿へ入った弟を追う
祖父の手紙、弟と父の試行、遺跡の年代からわかる情報を元に、生涯1回限りという神殿の謎に挑む

祖父

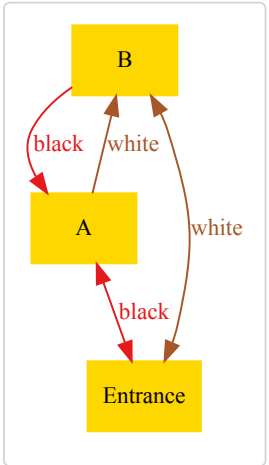
ということで、はじめての Alloy プログラミング？に挑もう

事前情報の段階でモデルを書く

- 遺跡の各部屋には白と黒の扉がある
- 扉の先は別の部屋につながっている（この条件入れていいのか？）
- 同じ部屋にある白と黒の扉は行先が異なる
- 条件を満たすと、入口に戻る
- 全ての部屋を通る最短経路を正解とする
  白白 や 黒黒 は正解ではない

```
module book/灯台の地下

abstract sig Room {
    black, white : one Room
}
one sig Entrance extends Room {}

fact 旧遺跡 {
    no r: Room | r.black = r            // 元の部屋ではない
    no r: Room | r.white = r            // 元の部屋ではない
    no r: Room | r.black = r.white      // 白と黒の行先は別の部屋
}
fact 古代語 {
    Entrance.white.white = Entrance     // 白白 は受け入れる（正解ルートではない）
    Entrance.black.black = Entrance     // 黒黒 は受け入れる（正解ルートではない）
}

one sig A extends Room {}

run {}
```

上の条件で実行すると、`No instance found. Predicate may be inconsistent.` と出る
`one sig A,B extends Room {}` と部屋数を増やすと、`Instance found. Predicate is consistent.` に変わる
この段階では、最低の部屋数と条件を満たす構造が発見できた



なお、後に排他を示す `disj` を知ったので、「白と黒の行先は別の部屋」の書き方を変えた

```
module book/灯台の地下

abstract sig Room {
    disj black, white : one Room        // 白と黒の行先は異なるので disj
}
one sig Entrance extends Room {}

fact 旧遺跡 {
    no r: Room | r.black = r            // 元の部屋ではない
    no r: Room | r.white = r            // 元の部屋ではない
}
```

『(再) 第2章 帰郷：灯台の地下』でもっと短くなる

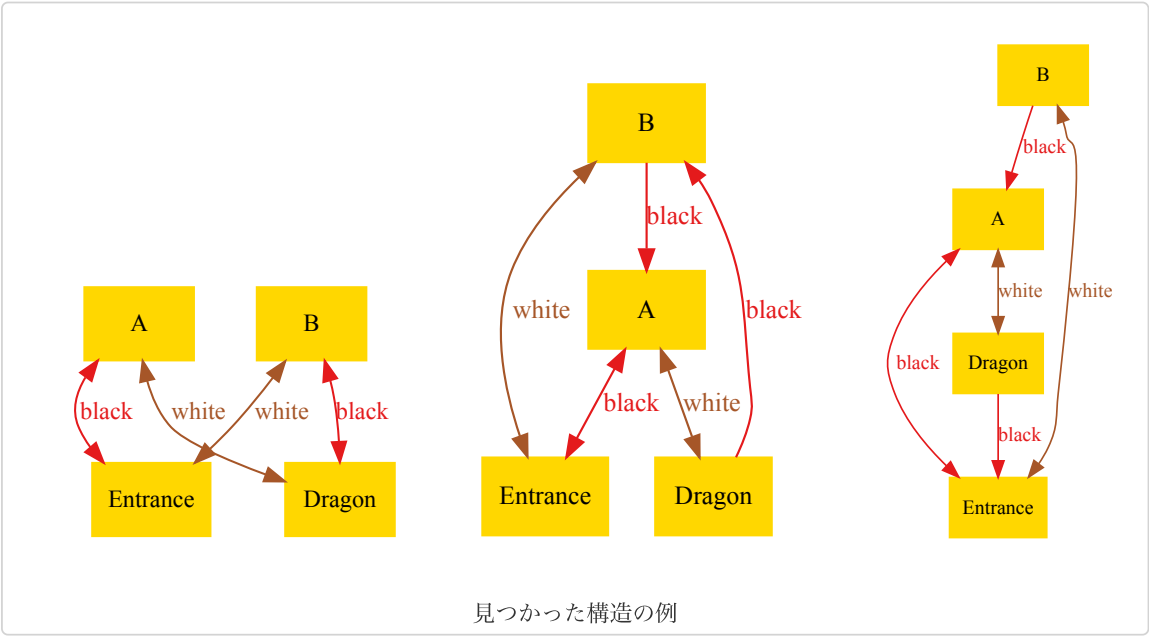
祖父の探索結果を反映する

- 入口から 黒白 で竜の部屋
  そこから 白黒 で入口に戻った
- 正解ルートではなかった

ということで、以下を追加

```
one sig A, Dragon extends Room {}
fact 祖父 {
    Entrance.black != Dragon            // 入口から 黒 は竜の部屋ではない
    Entrance.black.white = Dragon       // 入口から 黒白 で竜の部屋
    Dragon.white != Entrance            // 竜の部屋から 白 は入口ではない
    Dragon.white.black = Entrance       // 竜の部屋から 白黒 で入口
}
```

竜の部屋があったので、部屋Bを置き換えてみる
しかし実行すると、 `No instance found. Predicate may be inconsistent.` と出る
`one sig A,B, Dragon extends Room {}` にすると、 `Instance found. Predicate is consistent.` に変わる
もうひと部屋（以上）あったことが確定、条件を満たす構造はいくつかあることがわかる



見つかった構造の例

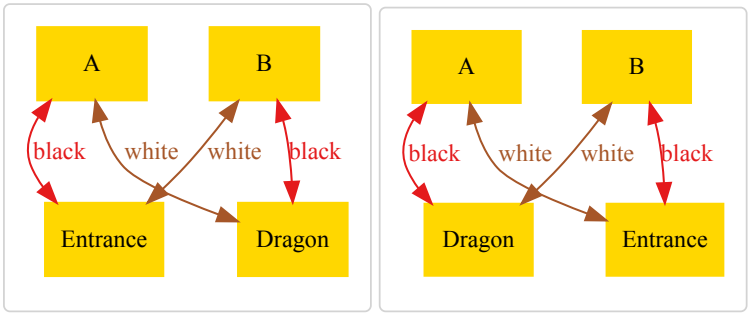
弟 レウリ

弟は長くさまよっていて詳しいことはわからない

- 入口から 白黒 で竜の部屋
- 白い扉を14回、黒い扉を8回通って入口に戻った
  竜の部屋は2回に1回くらい通った
- 竜の部屋から 黒白 で入口に戻る

とのことなので、以下を追加

```
fact 弟 {
    Entrance.white != Dragon            // 入口から 白 は竜の部屋ではない
    Entrance.white.black = Dragon       // 入口から 白黒 で竜の部屋
    Dragon.black != Entrance            // 竜の部屋から 黒 は入口ではない
    Dragon.black.white = Entrance       // 竜の部屋から 黒白 で入口
}
```



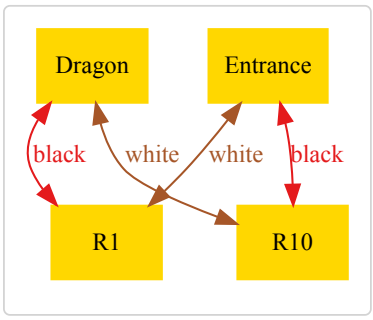
部屋 A,B の区別は不要なので、実はこの時点で(4部屋だけなら)構造が確定している
竜の部屋に着いてから、黒黒 や 白白 ばかりを繰り返していたことになる
なにやってんだ、レウリ

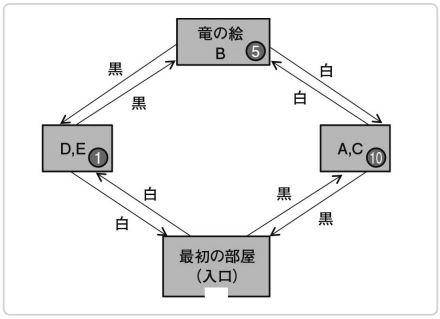
父 ヨシム

父は部屋ごとにコインでマーキングして、より正確な情報を得た
部屋 A,B をマーキングにしたがって R1,R10 に変えて、以下を追加

```
one sig R1, R10 extends Room {}
fact 父 {
    Entrance.white = R1                 // 入口から 白 の部屋に 1ガナ
    R1.black = Dragon                   // 竜の部屋に 5ガナ、落とし物
    Dragon.white = R10                  // 竜の部屋から 白 の部屋に 10ガナ
    Dragon.white.white = Dragon         // そこから 白 で竜の部屋
    // Entrance.white.black.white.white.white.black = Entrance
                                        // 通ったルートは 白黒白白白黒
}
```

構造は以下のようになった
無事、主人公の描いた設計図と一致している





正解の「全ての部屋を通る最短経路」は、主人公の言う通り 白黒白黒 と 黒白黒白 の２通りになる
これを発見させるところまでやりたいものだが、今はまだパス

> 第八古代ルル語
> 第八古代ルル語は、偶数個の○と偶数個の●を含む文字列すべてが文であり、そうでないものは文ではない。
> たとえば、『○○』『●●』『●○○●』『○●○●』『○●○○○●』のような文字列には、○も●も偶数個含まれるから、第八古代ルル語の文である。

リスト

ここまでの記述は以下の通り

白と黒のとびら01.als

```
module book/灯台の地下

abstract sig Room {
    disj black, white : one Room        // 白と黒の行先は異なるので disj
}
one sig Entrance extends Room {}

fact 旧遺跡 {
    no r: Room | r.black = r            // 元の部屋ではない
    no r: Room | r.white = r            // 元の部屋ではない
}
fact 古代語 {
    Entrance.white.white = Entrance     // 白白 は受け入れる（正解ルートではない）
    Entrance.black.black = Entrance     // 黒黒 は受け入れる（正解ルートではない）
}
// one sig A,B extends Room {}

one sig Dragon extends Room {}
fact 祖父 {
    Entrance.black != Dragon            // 入口から 黒 は竜の部屋ではない
    Entrance.black.white = Dragon       // 入口から 黒白 で竜の部屋
    Dragon.white != Entrance            // 竜の部屋から 白 は入口ではない
    Dragon.white.black = Entrance       // 竜の部屋から 白黒 で入口
}

fact 弟 {
    Entrance.white != Dragon            // 入口から 白 は竜の部屋ではない
    Entrance.white.black = Dragon       // 入口から 白黒 で竜の部屋
    Dragon.black != Entrance            // 竜の部屋から 黒 は入口ではない
    Dragon.black.white = Entrance       // 竜の部屋から 黒白 で入口
}

one sig R1, R10 extends Room {}
fact 父 {
    Entrance.white = R1                 // 入口から 白 の部屋に 1ガナ
    R1.black = Dragon                   // 竜の部屋に 5ガナ、落とし物
    Dragon.white = R10                  // 竜の部屋から 白 の部屋に 10ガナ
    Dragon.white.white = Dragon         // そこから 白 で竜の部屋
    // Entrance.white.black.white.white.white.black = Entrance
                                        // 通ったルートは 白黒白白白黒
}

run {}
```

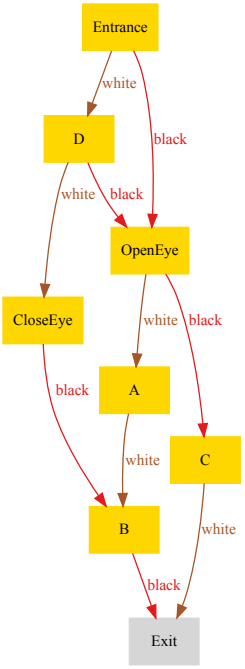
第1章 遺跡：人喰い岩

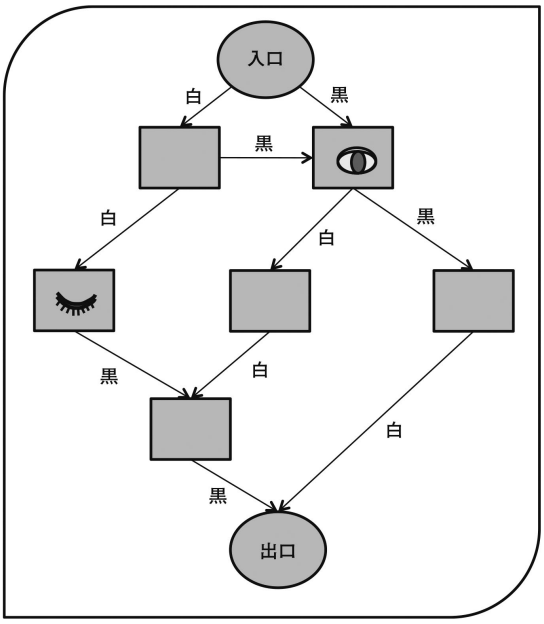
前章に戻って、『灯台の地下』より複雑な上に物語中で完全解明されていないため飛ばしていた『人喰い岩』を確認してみる
ここでは部屋の他に、扉のない地点（出口, 遠隔地）が現れる
今回も少しずつ書いて確認していくが、書き終えた全体リストは以下のようになった

白と黒のとびら02.als

```
module book/人喰い岩

abstract sig Point {}
abstract sig Room extends Point {
    disj black, white : one Point       // 白と黒の行先は異なるので disj
}
one sig Entrance extends Room {}
one sig Exit, NG extends Point {}

fact 旧遺跡 {
    no r: Room | r.black = r            // 元の部屋ではない
    no r: Room | r.white = r            // 元の部屋ではない
}

fact 仮定 {
    no r: Room | r.black = Entrance     // 入口に戻るルートはないと仮定
    no r: Room | r.white = Entrance     // 入口に戻るルートはないと仮定
}

one sig OpenEye, A, B extends Room {}
fact 判明済ルート {
    Entrance.black.white.white.black = Exit     // 長老
    Entrance.white.black.black.white = Exit     // 長老の兄

    // 長老ルートのどこかに OpenEye
    (Entrance.black = OpenEye) || (Entrance.black.white = OpenEye)
        || (Entrance.black.white.white = OpenEye)
    // 長老兄ルートのどこかに OpenEye
    (Entrance.white = OpenEye) || (Entrance.white.black = OpenEye)
        || (Entrance.white.black.black = OpenEye)
}

fact 最初の試行 {
    Entrance.white.black = OpenEye
}

one sig C extends Room {}
fact 追加情報 {
    Entrance.white.black.white.black = NG
}

one sig CloseEye, D extends Room {}
fact 最後の試行 {
    Entrance.white != CloseEye
    Entrance.white.white = CloseEye
    CloseEye.white = NG

    // 長老,長老兄ルートで通っていないようなので
    (Entrance.black != CloseEye) && (Entrance.black.white != CloseEye)
        && (Entrance.black.white.white != CloseEye)
    (Entrance.white != CloseEye) && (Entrance.white.black != CloseEye)
        && (Entrance.white.black.black != CloseEye)

    CloseEye.black.black = Exit     // 予想
}

fact クージュの図案集 {     // 載っていない扉は全てNGとしてみる
    // OpenEye.white.black = NG    // 白黒白黒 で既出
    OpenEye.black.black = NG
    CloseEye.black.white = NG
}

run {}
```

「閉じた目から 黒黒 で出口」という予想の有効／無効によらず、構造は一意にならない
さらに『クージュの図案集』に記載されていない扉を全てNGに飛ばすと、なんとか一致する
NGを非表示にするとすっきり





クージュの図案集より

第一古代ルル語

第一古代ルル語の文は、以下の五つしかない。
一つは、●●○。次は、○●○○●。あとは、●○○●と、○●●○と、○○●●。

(再) 第2章 帰郷：灯台の地下 — 見直しと各種確認

Alloy の学習が進んで、書き方がもう少しわかってきた
より正確に集合を意識して書き直しながら、各時点での状況を確認していこう

祖父

事前状態はこうなった

- 引数を持たない module 宣言は省略可能とのこと
- 扉の先が元の部屋でないことは、差集合 `Room - this` に属させることでクリア
- `abstract` , `extends` の使いどころを見直した結果、今回はどちらも使わない
  `abstract` なシグネチャの集合は、`extends` されたもののみで構成されることに注意
- 入口からの規則はシグネチャファクト `sig XXX {...} { ここ }` に書いた
  `this` 以外からの参照には `@` をつける必要がある(つけないと `this.` が補われる)
- 部屋数条件を増やして試すとき、到達できない部屋は不要なので除外
  通常の `fact { }` 内に書くなら、`all r:Room | r in Entrance.^(black + white)` となる
  `^` で再帰的にたどるイメージだが、今回の例のように無限に続いても大丈夫(実際たどるわけではないし)
- 字面のイメージに近い結果になるので誤解しがちだが、下例での `black` や `white` が `Room` の所有物**ではない**ことや、`Entrance.black.black` は何を意味するかとか、Alloyガール あたり眺めてフワッと理解くらいはした方がいい
  見た目は似ているが全く別の発想でできていることがわかって面白い
  大事なことなので、本資料でもあとでざっくり説明する

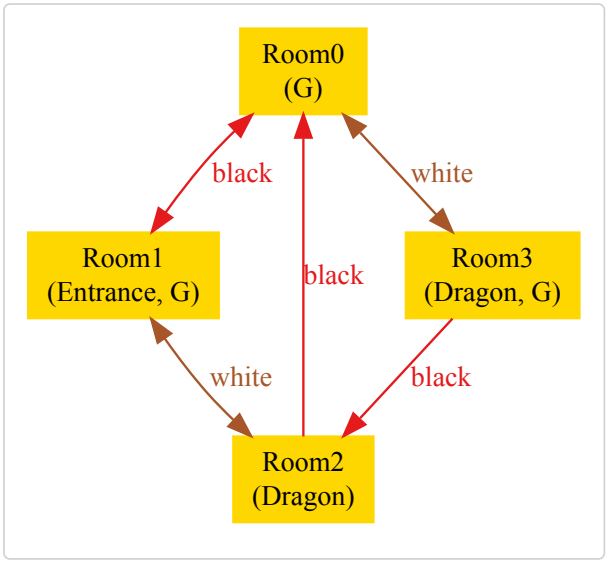
```
// 灯台の地下 ふたたび

sig Room {
    disj black, white : Room - this     // 白と黒の行先は別々の部屋で、しかも元の部屋ではない
}
one sig Entrance in Room {} {           // 入口はそんな部屋のひとつ
    white.@white = this                 // 白白 で入口に戻る
    black.@black = this                 // 黒黒 で入口に戻る
    this.^(@black + @white) = Room      // 全ての部屋は入口からたどり着ける
}
```

祖父の探索結果

- 竜の部屋をひとつ見たが、他にもないとは限らないので `Dragon` は `Room` の部分集合に
- 見やすさのため、通った部屋の集合 `G` を作る
- 竜の部屋が全部でいくつあるかは `#Dragon` で、部屋集合 `G` 内の竜の部屋数は `#(G & Dragon)` で表せる
  後者はつまり、集合 (`G` かつ `Dragon`) の要素数
  `= 1` は代入ではなく、要素数が一つだけであるという制約を記述している(そもそも代入という概念がない)

```
// 祖父
some sig Dragon in Room {}                  // 竜の部屋があった（他にもあるかも？）
some sig G in Room {}                       // 通った部屋
fact 祖父 {
    Entrance.black.white in Dragon                  // 入口から 黒白 で竜の部屋
    Entrance.black.white.white.black = Entrance // 黒白白黒 で入口に戻った

    // 通った部屋のうち、竜の部屋はひとつだけ
    G = Entrance +
        Entrance.black +
        Entrance.black.white +
        Entrance.black.white.white
    #(G & Dragon) = 1
}
run {} for 4 Room   // たかだか４部屋の範囲で
```

未通過の部屋（ `G` なし）が存在したり、さらにそれが竜の部屋である可能性などが示された



弟 レウリ

- 弟は入口から 白黒 で竜の部屋に着いた
- かなりさまよったため、最後に見た竜の部屋が最初に見たものと同じかは不明と考える
  最後に見た竜の部屋から 黒白 で入口に戻った
  → 竜の部屋のいくつかからは 黒白 で入口に到達する

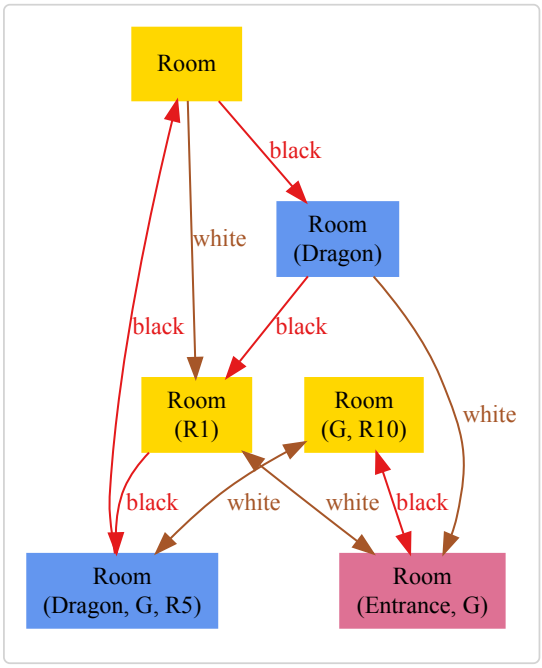
```
// 弟が通ったルートは 白黒...黒白
fact 弟 {
    // 入口から 白黒 で竜の部屋
    Entrance.white not in Dragon
    Entrance.white.black in Dragon
    // 竜の部屋から 黒白 で入口
    some d:Dragon | (d.black not in Dragon) && (d.black.white = Entrance)
}
```

最初に見た竜の部屋 と 最後に見た竜の部屋 は同じものか

最初に見た『竜の部屋』から 黒白 で入口に戻れることを確認する `assert` を書き、反例を探索させる

```
// 最初に見た竜の部屋 と 最後に見た竜の部屋 が同じなら、間を省略しても入口に戻るはず
assert simpleRouteEnable { Entrance.white.black.black.white = Entrance }
check simpleRouteEnable for 6 Room
```

５部屋以上あったなら、最初と最後の『竜の部屋』が別であった可能性が示された
ただし、５部屋の場合は竜の部屋が隣接してしまい、「２回に１回くらい竜の部屋」との証言に合わない



NOTE 上図の構造はいい感じに弟がさまよえるが、遺跡本来の目的には合わない「失敗作」になる

弟の試行は役に立ったか

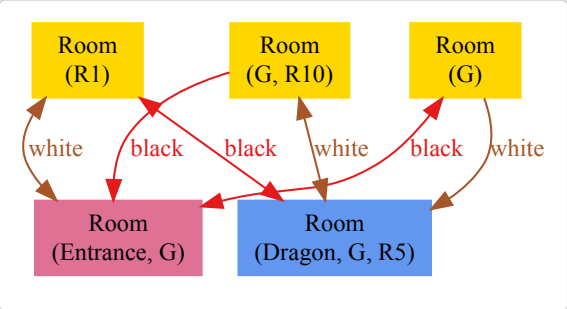
後述の主人公部分まで書いた後、`fact 弟 {...}` をコメントアウトしても、発見される構造例は変わらなかった
弟いらなかったんやー

父 ヨシム

父のマーキングについてはあまり変わらないが、竜の部屋が複数ある可能性を意識して修正

```
// 父が通ったルートは 白黒白白白黒
one sig R1, R5, R10 in Room {}          // コインでマーキングした
fact 父 {
    Entrance.white = R1                 // 入口から 白 の部屋に 1ガナ
    R1.black = R5                       // そこから 黒 の部屋に 5ガナ、(落とし物発見)
    R5 in Dragon                        // 竜の部屋だった
    R5.white = R10                      // そこから 白 の部屋に 10ガナ
    R5.white.white = R5                 // そこから 白 で竜の部屋
    R5.white.black = Entrance           // 白黒 で入口
    #((Entrance + R1 + R5 + R10) & Dragon) = 1  // 通った部屋のうち、竜の部屋はひとつだけ
}
run {} for 5 Room   // たかだか5部屋の範囲で
```

5部屋あったなら、他の構成がまだあり得る（6部屋、竜の部屋2つの構成は弟の図を参照）

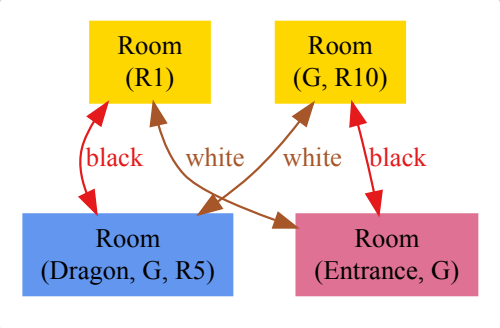


主人公 ガレット

ここで主人公の確認結果が生きてくる
以下を追加して、10部屋まで探索させてみる

```
fact 僕 {                               // 通ったルートは 黒白黒白
    Entrance.black = R10                // 父が通らなかった扉
    R10.white = R5                      // 父ルートで既知
    R5.black = R1                       // 父が通らなかった扉
    R1.white = Entrance                 // 父が通らなかった扉
}
run {} for 10 Room   // たかだか10部屋の範囲で
```

10部屋以内では1つの構成しか発見できず、4部屋構成で正しかったことが確認できた
（探索範囲を1000部屋に広げたら他の構成が見つかるかもしれないが、ないと期待しよう）
父の試さなかった扉が予想外の部屋につながらないことを確認しながら進んでおり、主人公は堅実な選択をしたと言える
とはいえ、最初の部屋がR10でなかった場合（父の5部屋例など）や、3部屋目がR1でなかった場合（弟の6部屋例など）はきっと途方に暮れたことだろう



リスト

書き直した全体はこうなった
短くなった部分もあるが、厳密化などのため全体としては長くなっている

白と黒のとびら03.als

```
// 灯台の地下 ふたたび

sig Room {
    disj black, white : Room - this     // 白と黒の行先は別々の部屋で、しかも元の部屋ではない
}
one sig Entrance in Room {} {           // 入口はそんな部屋のひとつ
    white.@white = this                 // 白白 で入口に戻る
    black.@black = this                 // 黒黒 で入口に戻る
    this.^(@black + @white) = Room      // 全ての部屋は入口からたどり着ける
}

// 祖父
some sig Dragon in Room {}              // 竜の部屋があった（他にもあるかも？）
some sig G in Room {}                   // 通った部屋
fact 祖父 {
    Entrance.black.white in Dragon              // 入口から 黒白 で竜の部屋
    Entrance.black.white.white.black = Entrance // 黒白白黒 で入口に戻った

    // 通った部屋のうち、竜の部屋はひとつだけ
    G = Entrance +
        Entrance.black +
        Entrance.black.white +
        Entrance.black.white.white
    #(G & Dragon) = 1
}
run {} for 4 Room

// 弟が通ったルートは 白黒...黒白
fact 弟 {
    Entrance.white not in Dragon
    Entrance.white.black in Dragon          // 入口から 白黒 で竜の部屋
    // 竜の部屋から 黒白 で入口
    some d:Dragon | (d.black.white = Entrance) && (d.black not in Dragon)
}
// 最初に見た竜の部屋 と 最後に見た竜の部屋 が同じなら、間を省略しても入口に戻るはず
assert onlySimpleRoute { Entrance.white.black.black.white = Entrance }
check onlySimpleRoute for 6 Room

// 父が通ったルートは 白黒白白白黒
one sig R1, R5, R10 in Room {}              // コインでマーキングした
fact 父 {
    Entrance.white = R1                     // 入口から 白 の部屋に 1ガナ
    R1.black = R5                           // そこから 黒 の部屋に 5ガナ、（落とし物発見）
    R5 in Dragon                            // 竜の部屋だった
    R5.white = R10                          // そこから 白 の部屋に 10ガナ
    R5.white.white = R5                     // そこから 白 で竜の部屋
    R5.white.black = Entrance               // 白黒 で入口
    #((Entrance + R1 + R5 + R10) & Dragon) = 1  // 通った部屋のうち、竜の部屋はひとつだけ
}
run {} for 6 Room   // たかだか６部屋の範囲で

fact 僕 {                                   // 通ったルートは 黒白黒白
    Entrance.black = R10                    // 父が通らなかった扉
    R10.white = R5                          // 父ルートで既知
    R5.black = R1                           // 父が通らなかった扉
    R1.white = Entrance                     // 父が通らなかった扉
}
run {} for 10 Room   // たかだか10部屋の範囲で
```

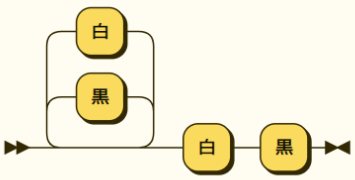
第3章 復元：妖精の遺跡

第３章では、同じ遺跡（構造）を示す２つの詩（表現）が出る

- サルク
  もし望むならば、夜と昼は望むまま。しかし最後に見るものは、一度の昼と、一度の夜。
- タビハ
  もし望むならば、夜を望むだけ繰り返せ。望まなければ夜は来ぬ。望まなくとも、一度の昼は訪れる。
  昼を望むだけ繰り返し、一度の夜と共に終われ。ひとたびこれを生き抜けば、望むだけ繰り返すべし。

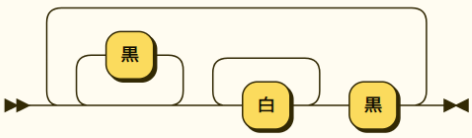
要するにこうだ　※ Railroad Diagram Generator 便利

サルク:




```
サルク      ::= ( '黒' | '白' )* '白' '黒'
```

タビハ:




```
タビハ      ::= ( '黒'* '白'+ '黒' )+
```

これを、３つの部屋で実現していたとのこと
せっかく書いたが、上図は気にせず進める

> 第四十七古代ルル語
> 第四十七古代ルル語の文は「最後に必ず○●で終わる文字の列」

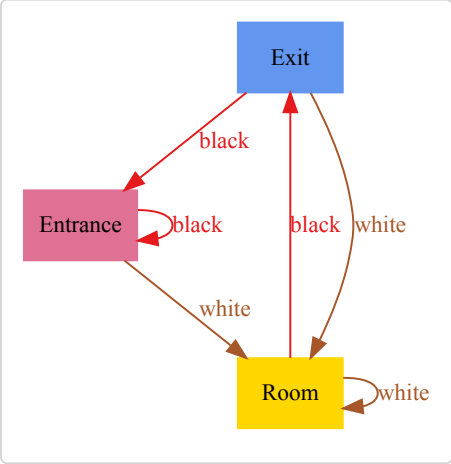
サルク

サルク側の詩はこうだろう

白と黒のとびら04-1.als

```
module book/妖精の遺跡_サルク

sig Room { white, black : Room }
one sig Entrance, Exit extends Room {}
fact サルク {
    // もし望むならば、夜と昼は望むまま。しかし最後に見るものは、一度の昼と、一度の夜
    Exit not in (Entrance.black + Entrance.white)
    all r:Room | r.white.black = Exit                    // どの部屋からでも 白黒 で出口
    no r:Room | (r.white=Exit) || (r.black.black=Exit)   // 白 や 黒黒 では出口に到達しない
}
run {} for 3 Room
```

３部屋までの場合、条件を満たす構造はひとつだけだった



サルク文法と形が異なるが、これは文法表現に「どこからでも 白 と 黒 が選択可能」「３部屋だけ」という遺跡の特徴が入っていないためだろう

タビハ

タビハ側はこう

白と黒のとびら04-2.als

```
module book/妖精の遺跡_タビハ

sig Room { white, black : Room }
one sig Entrance, Exit extends Room {}

fact タビハ {
    // もし望むならば、夜を望むだけ繰り返せ。望まなければ夜は来ぬ。望まなくとも、一度の昼は訪れる。
    // 昼を望むだけ繰り返し、一度の夜と共に終われ。ひとたびこれを生き抜けば、望むだけ繰り返すべし
    Exit not in (Entrance.black + Entrance.white)
    no r:Room | (r.white=Exit) || (r.black.black=Exit)   // 白 や 黒黒 では出口に到達しない

    all r: (Entrance.*black) | r.white.black = Exit      // 入口から 黒* 白黒 で出口
    all r: (Entrance.^white) | r.black = Exit            // 入口から 白+ 黒 で出口
    all r: (Exit.*black) | r.white.black = Exit          // 出口からでも同じ
    all r: (Exit.^white) | r.black = Exit
}

run {} for 3 Room
```

無事、サルクと同じ結果になった

第4章 金と銀と銅：風変わりな家

「せいラウのみず　120トラヌ。　おつりは　でません」と書いてある風変わりな家
家の中身はどうなっているか？

- 請求額が120トラヌ以上になれば中庭に出る
  ならないうちは家の中
- 金の扉だけ2回通ったら200トラヌ請求された
  銀の扉だけ3回通ったら150トラヌ請求された
  金の扉は1回100トラヌ、銀の扉は1回50トラヌ、の推測で正しいと確約された
- 銅の扉だけ12回通ったら120トラヌ請求された

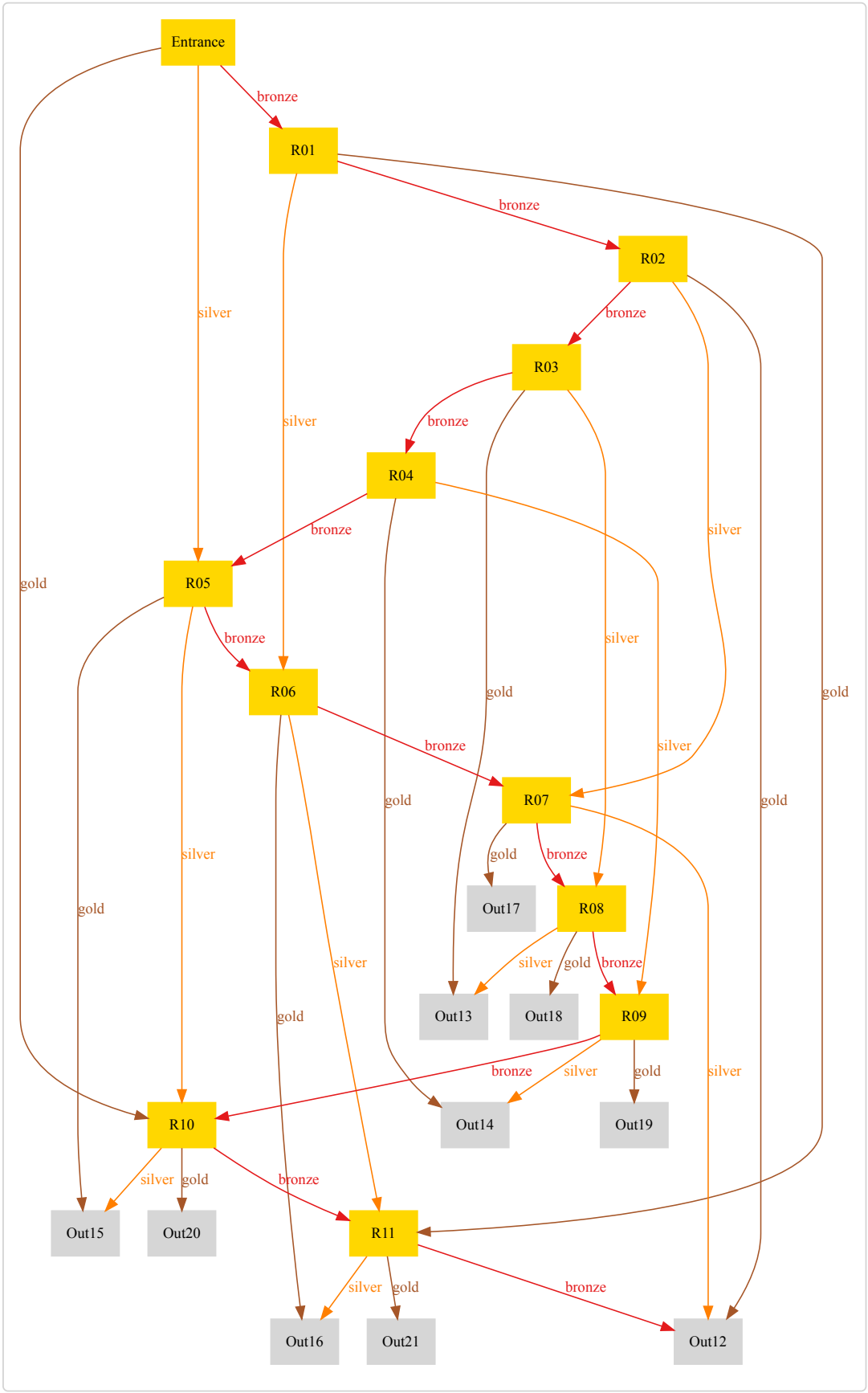
描かせてみよう

書くことはできたが、図は大きい
手書きならもっと整然とした図にできるのだが…

白と黒のとびら05.als

```
module book/風変わりな家

abstract sig Point { price: Int }
abstract sig Room extends Point { gold, silver, bronze: Point }

one sig Entrance extends Room {} { price = 0 }
one sig R01 extends Room {} { price = 1 }
one sig R02 extends Room {} { price = 2 }
one sig R03 extends Room {} { price = 3 }
one sig R04 extends Room {} { price = 4 }
one sig R05 extends Room {} { price = 5 }
one sig R06 extends Room {} { price = 6 }
one sig R07 extends Room {} { price = 7 }
one sig R08 extends Room {} { price = 8 }
one sig R09 extends Room {} { price = 9 }
one sig R10 extends Room {} { price = 10 }
one sig R11 extends Room {} { price = 11 }
one sig Out12 extends Point {} { price = 12 }
one sig Out13 extends Point {} { price = 13 }
one sig Out14 extends Point {} { price = 14 }
one sig Out15 extends Point {} { price = 15 }
one sig Out16 extends Point {} { price = 16 }
one sig Out17 extends Point {} { price = 17 }
one sig Out18 extends Point {} { price = 18 }
one sig Out19 extends Point {} { price = 19 }
one sig Out20 extends Point {} { price = 20 }
one sig Out21 extends Point {} { price = 21 }

fact {
    all r:Room | all p:Point | p.price = r.price.add[10] => r.gold = p
    all r:Room | all p:Point | p.price = r.price.add[5] => r.silver = p
    all r:Room | all p:Point | p.price = r.price.add[1] => r.bronze = p
}

run {} for 6 int, 22 Point
```

Intに使われるビット数は未指定時4bit(符号あり)で、22まで表すには足りないので6bit(符号あり)を指定する



中庭に面した扉は16個あり、どのエリアに出てきたかで10通りの請求金額がわかるのだろう
扉だらけで、出口を見張るのもなかなか大変だ

銅の扉が 10 トラヌでない可能性はあったか？

銅の扉を開けないうちに回答することはできなかったか？

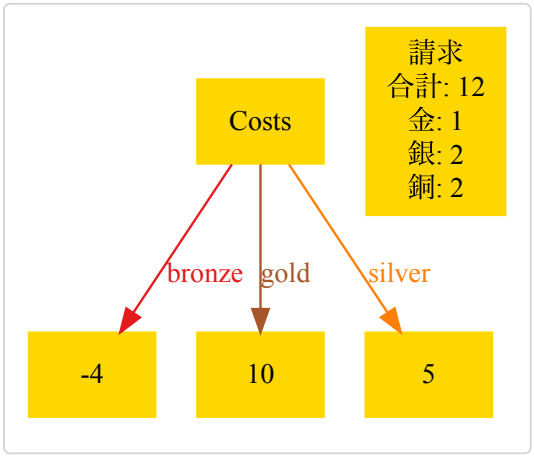
わかっている情報を素直に書いていく

- どの部屋の金の扉も1回通るごとに同じ額を加算、2回で200トラヌ
  銀も同様で、3回で150トラヌ
  銅も同様だが、未確認
- 合計120トラヌ以上で請求
- 120トラヌ以上ならいくら出しても出られるのが自慢
  120, 130, 140, … 210 がすべて可能だと言っているのだろう
  書きにくそうなのでとりあえず無視
  家の中で 10, 20, … 110 をすべて取れればいいはず

最小単位になる銅の扉が20トラヌや30トラヌでは自慢にならないはず
また、1トラヌ貨や5トラヌ貨を考慮するなら、正解の10トラヌでも自慢にならない

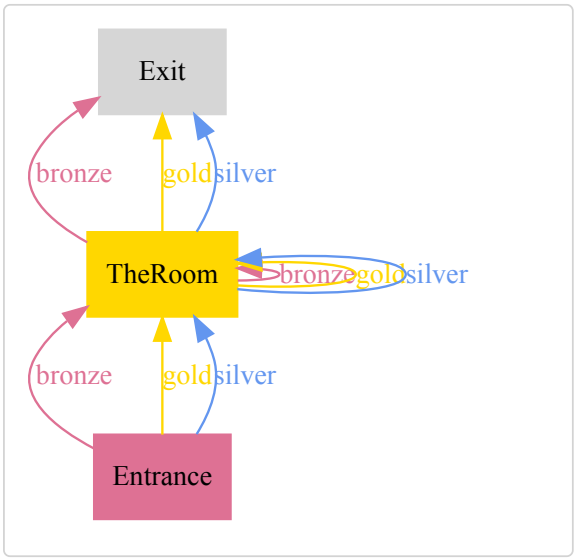
```
one sig Costs {
    gold, silver, bronze: Int
} {
    gold = 20.div[2]
    silver = 15.div[3]
    // bronzeは謎としておく
}

one sig 請求 {
    合計, 金, 銀, 銅: Int
} {
    金 >= 0 // マイナス２回通った場合とか言われても困る
    銀 >= 0
    銅 >= 0
    合計 = mul[Costs.gold, 金]
     .add[ mul[Costs.silver, 銀] ]
     .add[ mul[Costs.bronze, 銅] ]
}
run {請求.合計 = 12} for 6 int
```

ああ、マイナス扉もありうるのか



マイナス単価の扉を作れるか？

ふと、銅の扉 1 回 -4(×10)トラヌで描いたらカオスな図になるかな？ と思って試してみるとそうでもない図を見て、「ああ、マイナス部屋が足りないからか」とわかった
深く考える前に試すことで問題にはやく気づけるのが Alloy を使うメリットなので、よかったと思おう
マイナス部屋はいくつあったらいいか考えると、マイナス側には脱出条件がないので無限に必要になる
ここの建築事情では、扉を通った回数を数える仕組みや合計金額によって行先が変わる仕組みがなさそうなので、マイナス単価はなしだろう
そもそもそんな仕組みがあったら 1 部屋で十分だ



遺跡には遠隔地へ飛ばす扉とか、石が増えたり減ったする扉もあったが、ここの主には無理なのだろう

ちなみに、上の図を描くためにこんなモデルを書いた

```
one sig Entrance { gold,silver,bronze: TheRoom }
one sig TheRoom { gold,silver,bronze: set (TheRoom+Exit) } {
    gold = TheRoom + Exit
    silver = TheRoom + Exit
    bronze = TheRoom + Exit
}
one sig Exit {}
```

簡単な遷移図を描きたいときにも Alloy は便利だ

もうちょっと賢い感じに書けないか

銅の扉が1(×10)トラヌと直接書いている点もそうだが、ありうる全地点を列挙しているのは、答えを知っていて描かせただけ感が強いのが気になる
実は全ての地点を列挙せず、例えば以下のようにしても解ける

```
sig Point {
    value: Int,
    gold,silver,bronze: lone Point
} {
    value < 12  => {
        gold.@value = value.add[10]
        silver.@value = value.add[5]
        bronze.@value = value.add[1]
     } else {
        gold = none
        silver = none
        bronze = none
     }
}

one sig Entrance in Point {} { value=0 }
run {} for 6 int, 24 Point
```

しかし、可変数が増えると探索にかかる時間が一気に延びてしまう
以下の地点列挙を加えると180msで解けたものが、列挙なしでは(少し違う版での結果だが)2時間弱かかった

```
one sig Entrance extends Point {} { value=0 }
one sig p01 extends Point {} { value=1 }
one sig p02 extends Point {} { value=2 }
one sig p03 extends Point {} { value=3 }
one sig p04 extends Point {} { value=4 }
one sig p05 extends Point {} { value=5 }
one sig p06 extends Point {} { value=6 }
one sig p07 extends Point {} { value=7 }
one sig p08 extends Point {} { value=8 }
one sig p09 extends Point {} { value=9 }
one sig p10 extends Point {} { value=10 }
one sig p11 extends Point {} { value=11 }
one sig out12 extends Point {} { value=12 }
one sig out13 extends Point {} { value=13 }
one sig out14 extends Point {} { value=14 }
one sig out15 extends Point {} { value=15 }
one sig out16 extends Point {} { value=16 }
one sig out17 extends Point {} { value=17 }
one sig out18 extends Point {} { value=18 }
one sig out19 extends Point {} { value=19 }
one sig out20 extends Point {} { value=20 }
one sig out21 extends Point {} { value=21 }
one sig out22 extends Point {} { value=22 }
one sig out23 extends Point {} { value=23 }
```

実用的であることの方が重要なので、時間がかかるようなら遠慮なく列挙すべき
なお、`extends` ではなく `in` とした場合には、2.7秒かかった

もう少しがんばろう

Alloy の理解を進めてなんとかしたい

`sig Point { price: Int }` と書いた場合、`price` とは何か？

```
sig Point { price: Int }
run {} for exactly 5 Point
```

を実行して Table ビューを開くと次のように表示された

this/Point	price
Point⁰	7
Point¹	6
Point²	6
Point³	6
Point⁴	5

ここに見られるように、`price` とは `Point` から `Int` への関係群である
`Point` のプライベートメンバなどではなく、グローバルな関係表 `Point->Int` である

Evaluator を開いていくつか確認してみよう

```
#price
5
price
```

Point⁰	7
Point¹	6
Point²	6
Point³	6
Point⁴	5

```
Point
```

Point⁰	Point¹	Point²	Point³	Point⁴

$^{-1}$

```
Point.price
```

5
6
7

```
Point$2.price
```

6

```
price.6
```

Point¹
Point²
Point³

`price` の個数なんて概念がある時点でもう世界観が違う
データベースのインデックスと思えば、SQL に似ているという意見にうなづける
`Point.price` は集合 `Point` と `price` の結合であり、結果として `price` に含まれる `Int` の集合になる
もっと限定して `Point$2.price` なら、`Entrance.price` と同様に `Point$2` の `price` が得られる
そして結合 `.` を使うと、実は逆引きも同じコストで行える
`price` と `{6}` の結合 `price.6` で、`price=6` となる `Point` の一覧が得られるのだ
当然 `all p:Point | p.price = 6` などとして探すより軽い

つまり入口にある銀の扉 `Entrance.silver` の行き先候補が存在するかは `行先Point.price=5` なので `some price.5` で得られるし、候補が1つなら `Entrance.silver = price.5` が成り立つ
`price: disj Int` で同値候補を1つに限定して、だいぶ賢い感じにまとまった

```
abstract sig Point { price: disj Int } {}   // 全ての Point で price は一意
sig Room extends Point { gold, silver, bronze: Point } {
                                            // 各扉: one Point なので、存在する制約になる
    (0 =< price) && (price < 12)            // 簡単な制約は書いておいた方が速い
    gold = @price.(price.add[10])           // 全price のうち、自price+10 となるPoint群（1件）
    silver = @price.(price.add[5])
    bronze = @price.(price.add[1])
}
one sig Entrance extends Room {} { price = 0 }
sig Out extends Point {} { price>=12 }

run {} for 6 int, 22 Point
```

列挙なしで、2時間が98秒に縮まった

とは言え、地点列挙した方が速いことは変わらない
列挙するだけ(price 特定なし)でも倍速になった(45秒)のでよしとしよう

白と黒のとびら06.als

```
module book/風変わりな家

abstract sig Point { price: disj Int } {}
sig Room extends Point { gold, silver, bronze: Point } {
    (0 =< price) && (price < 12)
    gold = @price.(price.add[10])
    silver = @price.(price.add[5])
    bronze = @price.(price.add[1])
}

one sig Entrance extends Room {} { price = 0 }
// sig Out extends Point {} { price>=12 }
one sig r01,r02,r03,r04,r05,r06,r07,r08,r09,r10,r11 extends Room {}
one sig o12,o13,o14,o15,o16,o17,o18,o19,o20,o21 extends Point {} { price>=12 }

run {} for 6 int //, 22 Point
```

NOTE 所要時間は2012年MacBookProでの結果で、今どきのデスクトップならずっと速い(この列挙あり版なら2秒)

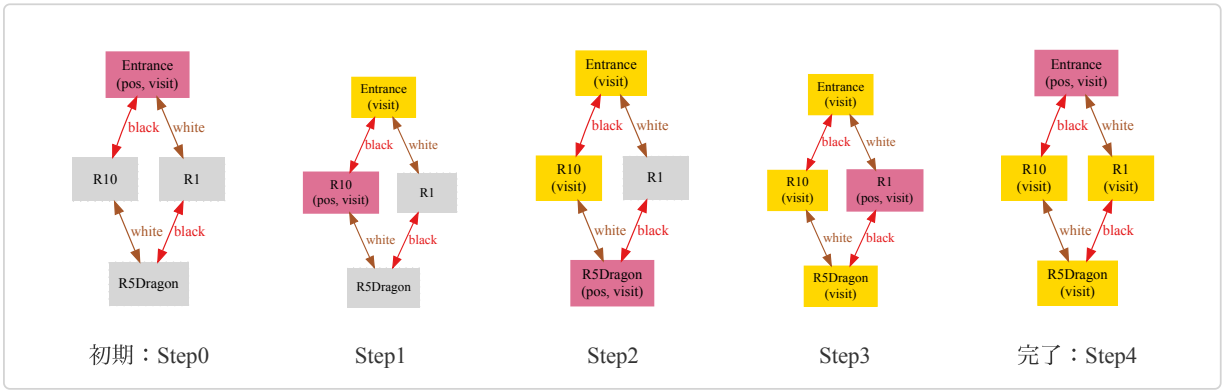
(再々) 第2章 帰郷：灯台の地下 — 正解を探せ

さて、後回しにしていた「全部屋を通る最短経路を探す」問題を解決しよう
`util/ordering [対象]` を使うと、ステップ毎に決まったあるいは自動生成される変化を記述できる
どのシグネチャについて順序を管理するかを `[対象]` 部分で指定するが、一般的には `[Time]` とするようだ
(複数使いたい場合は `open util/ordering [対象]` を複数行書く)
対象の初期状態 `first` と完了状態 `last` を指定し、その間の各ステップにおける対象の状態を記述する

今回はこうしよう

- 最初の段階では以下の状態である
  探索者は入口にいる
  見た部屋は入口だけ
- ステップごとに以下の状態が変わる
  いずれかの扉を通って次の部屋に移動(どちらの扉を通るかは任せる)
  移動先の部屋を見た部屋に追加
- 以下の状態になったら完了
  全部屋を見た かつ 入口にいる

実際に作ってみた
指定ステップ数ちょうどで完了する例の提示と、各ステップでの状態変化を確認できた（黒白黒白 の例）



`run {} for 5 Step` とすると、初期状態を含めて5ステップで完了する２ケース「黒白黒白」「白黒白黒」が見つかった
`6 Step` では見つからず、`7 Step` なら「白黒白白白黒」等が見つかる
最短であるかは確認できないので、与えるステップ数を変えて探すことになる
「灯台の地下」は４部屋なので、`5 Step` が見つかったからにはそれが最短である

リスト

他の構造でもうまく動くか試すため、遺跡を生成させて確認することもできるようにした

白と黒のとびら07.als

```
module book/灯台の地下
open util/ordering[Step]

abstract sig Room { disj black,white: Room - this }

// 灯台の地下
one sig Entrance extends Room {} { (black=R10) && (white=R1) }
one sig R10 extends Room {} { (white=R5Dragon) && (black=Entrance) }
one sig R5Dragon extends Room {} { (black=R1) && (white=R10) }
one sig R1 extends Room {} { (white=Entrance) && (black=R5Dragon) }
灯台の地下: run {} for 5 Step

// 遺跡生成するとき
// one sig Entrance extends Room {} { this.^(@black+@white) = Room }
// some sig R extends Room {}
// 遺跡生成: run {} for 10 but exactly 5 Room, 6 Step


enum Door {Black, White}
sig Step {
    room: Room,             // 今どこの部屋にいるか
    visit: set Room,        // 訪れた部屋の一覧
    nextDoor: lone Door     // 次にどちらの扉を通るか（完了時以外は指定せず）
} {
    (visit=Room) && (room=Entrance)  <=> nextDoor = none    // 完了時だけは次の扉なし
}
fact {
    // 初期状態
    first.room = Entrance
    first.visit = Entrance

    // 完了時状態
    last.nextDoor = none

    // 次のステップはどういう状態か
    all t,t':Step| t'=t.next => {
        ((t.nextDoor=Black) && (t'.room = t.room.black)) ||
        ((t.nextDoor=White) && (t'.room = t.room.white))
        t'.visit = (t.visit + t'.room)
    }
}
```

終わりに

『白と黒のとびら』は16章まで続くが、この後はパズル的に考えやすい形ではなくなっていく
最後まで面白い本だったが、Alloy の練習材料としてはここまでとしたい

まだ Alloy についてほんのさわりしか使えないが、今回の練習を通じて以下を実感できた

- 概念レベルの仕組みを記述・即確認しやすく、それゆえ上流設計の精度を高めるのに役立ちそう
- 普段使う手続き型と大きく異なるため、考え方を変える必要がある
  別の視点で考え直すことになり、これも精度を高める役に立つだろう
- 参考書籍や本家サイトは学術的な感が強く、理解するのが難しい
  とはいえこの分野では破格に平易である
- 理解度・仕様の両面から、なかなか思ったことをそのまま書けず、記述量に比べてかなり長時間かかる
- 変数を増やすなどで組合せ量を増やしてしまうと、計算時間がはね上がる
  考えるためのツールとして「気になったらすぐ確認」を行うため、書き方を工夫した上で10秒以内で済む範囲で使いたい
- 日本語版の参考書籍が出版されるなど、日本で盛り上がったのは約10年も前
  すでに生き残っていないサイトも多く、ネット情報による理解度アップも滞りがちになる
  現時点で残っている日本語おすすめサイトは以下の通り
  - 教えて！アロイ人！　：　コラムが特に同感
  - Alloyまとめ@西尾泰和のはてなダイアリー　：　Alloyガールは読んどいた方がいい
  - Alloy Analyzer で形式仕様記述
  - SlideShare で `Alloy` を検索
  - GoogleGroup: Alloy-jp
  - Qiita や GitHub で `Alloy` を検索