

# Esolang Codegolf #6

## Transceternal Writeup

@\_hiromi\_mi  
(Version 2020.5.12.2)

# おしながき

- 問題の復習
- Transceternal の説明
  - グラフの各ノードの意味
  - 入出力と deserialize, serialize
  - コマンドの意味
  - 解答するための部材
- 解答の説明
- 今後 (Golf の方針)

# 問題の復習

- <https://esolang.hakatashi.com/contests/6/rule> より
  - 空白1文字で区切られた2つ組の3桁の数値が、32組、改行区切りで与えられる。
  - 7文字の入力のうち、空白で表された場所に、ワクチン1～ワクチン9のいずれかを投与する。
  - ワクチン投与によって消滅させられるウイルスの数を最大化するようなワクチンの種類を、対応する1～9の数字で答えよ。

# 例

- 123 456 -> (任意)
  - どのワクチン投与しても効果なし (3つ並ばない)
- 122 456 -> 2
  - 2 を投与すると 1222456 と3つ並び
- 125 567 -> 5
- 注: 222 457
  - 3つ既に連続するものはデータセットに含まれていない

# ロジックの基本方針

- $b[2] == b[3]$  なら  $b[2]$  それ以外は  $b[5]$ 
  - $b$  は1行を表す文字列
- $l[4] = l[5]$  なら  $l[4]$  or  $l[5]$  それ以外は  $b[2]$
- $X=abc(\text{数値})$   $Y=def(\text{数値})$ 、 $X\%100\%11==0$  なら  $X\%10$  そうでなければ  $Y/100$

(以上, 赤チーム内共有文章ロジックリストより)

# おしながき

- 問題の復習
- Transceternal の説明
  - グラフの各ノードの意味
  - 入出力と deserialize, serialize
  - コマンドの意味
  - 解答するための部材
- 解答の説明
- 今後 (Golf の方針)

# Transceternal

- グラフ構造に全ての状態が埋められた esolang
- 作者による解説: <https://esolangs.org/wiki/Transceternal>
- 処理系: <https://github.com/Hakerh400/esolangs/>
  - \$ node index transceternal program.txt input.txt output.txt
  - ソースコード: src/langs/transceternal/index.js

# 例: cat (入力をそのまま返す)

- いろいろな表現ができる
  - catacat (公式サイトの例)
  - B C D D D E E E D (筆者作)
  - 123334443 (筆者作)



# グラフ構造

- (実行時) root を頂点とした有向グラフ
- 全ての節点は2つの枝をもち, それぞれ 0-pointer, 1-pointer とよばれる
- 二分木に似ているが、木ではなく各pointer は自分自身に戻る, 他ノードへ結びつくなど自由に動く. 二分「木」ではない

# グラフ構造: 特別なノード

- input: 入力データをグラフ構造にしたものの最初の節点
- admin: ソースコード中での最初のノード.
  - (0-pointer, 1-pointer) = (管理領域の一番上, 次の状態)
- root: 一番上にあるノード. インタプリタが生成.
  - (0-pointer, 1-pointer) = (admin, input) となる

# グラフ構造: 特別なノード

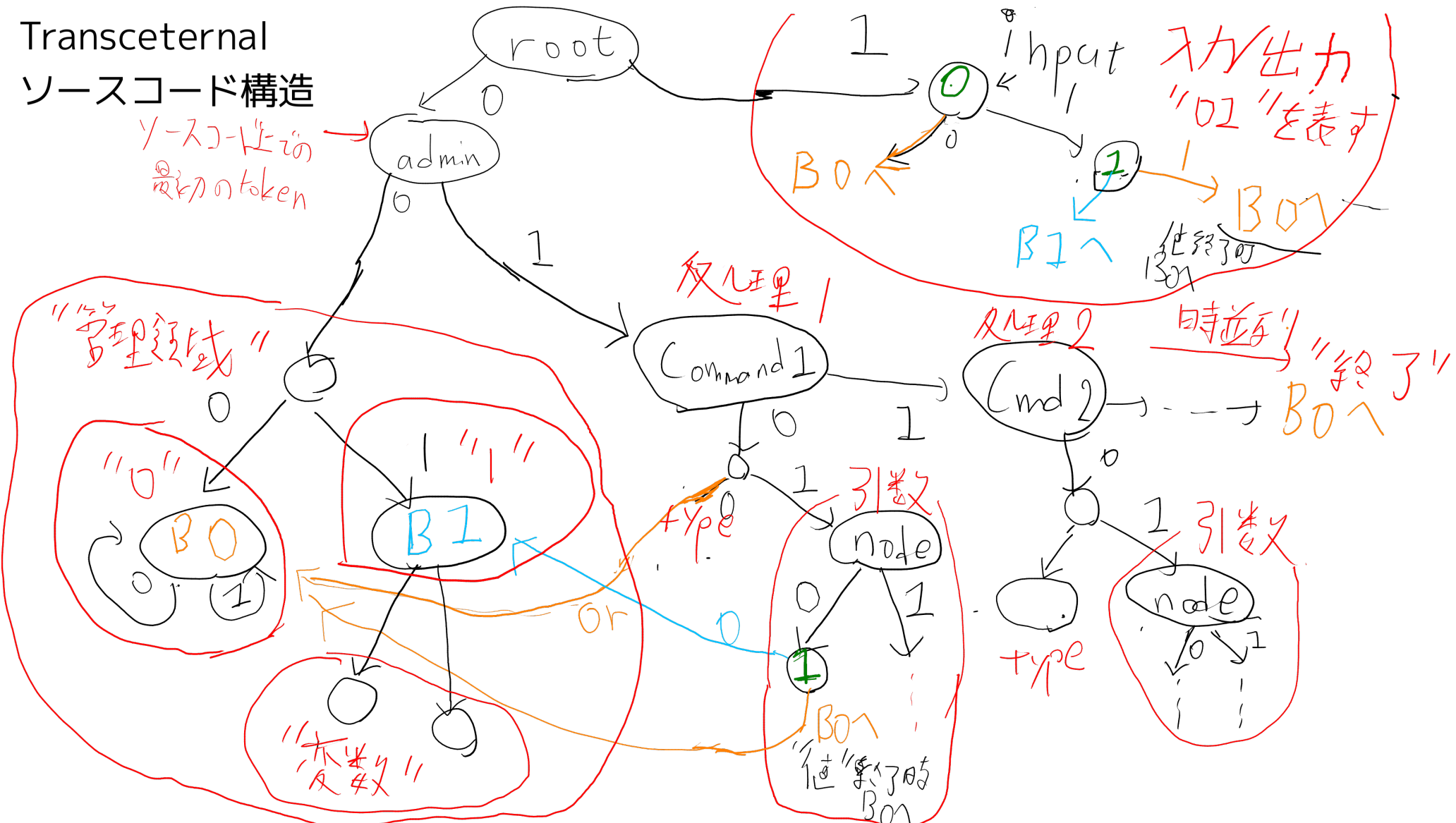
- B0: 絶対アドレス 000 にあるノード
  - 以下のようなものの判定基準となる
  - 入力をDeserialize したときに `0` に対応する節点
  - Serialize するときに 0-pointer がB0 だと `0` に対応する
  - 終了判定は絶対アドレス 01 が B0 と一致するかで判定
- B1: 絶対アドレス 001 にあるノード
  - 入力が Deserialize されたときに `1` に対応する節点

# グラフ構造

- 次のスライドに全体の構造を示します
- 大体の構造を把握してもらえたらと思います
  - その先の細かな構造は後ほど説明します

# Transceteral ソースコード構造

10. 文のtoken



# 実行の流れ

- ソースコードを token に分割する
- 分割した token をグラフ構造になおす (Initial Memory Consumption)
- 入力を deserialize してグラフ構造になおす
- 新root ノードを作成し, `0` にソースコード, `1` に入力を接続する
- 実行 (main loop). グラフノード内を以下に述べる規則で移動する
- Command が B0 になれば main loop 脱出, それまでは繰り返す
- `1` 以下を serialize して結果を生成する
- 標準出力に生成した結果を出力して終了

# “token”

- ソースコードをグラフの形に直す際に使う単位
  - グラフのノードの“名前”とでも考えるとよい
- 1文字ならスペース省略可能
  - catacat は “c” “a” “t” “a” “c” “a” “t”
- 2文字以上なら (本問は節点数が多いのでこちら)
  - hoge fuga は “hoge” “fuga”

# Initial Memory Consumption

- ソースコードをグラフ構造に直す方法
- スタックを埋めていく
  - Token名: 0-pointer の節点 1-pointerの節点
- 初期状態は 最初のToken: \_\_ ではじまる
- 各Token ごとに、スタックが埋まるまで以下の処理を繰り返す



# Initial Memory Consumption

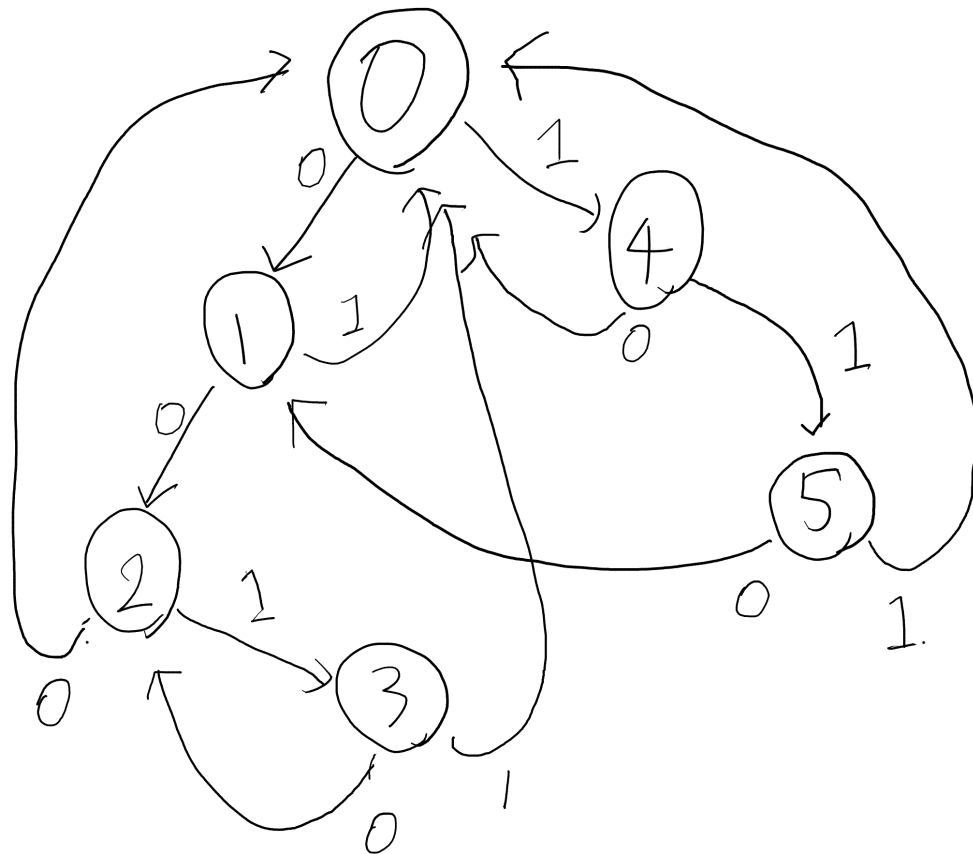
- スタック左側 (0-pointer) を優先し、深さ優先探索にて最初に見つけた場所に埋めていく
- 未知の節点は 0-pointer に加えると同時にスタックを一段下に伸ばす
- 0-pointer, 1-pointer が埋まれば“stackから削除する”
  - 実装上はそうなるが、(作者解説を含め) 以下の説明では残してある

# Initial Memory Consumption 例

012032004051025

Step1 (0)

0: \_ \_



出典: 公式サイトの例

# Initial Memory Consumption 例

012032004051025

Step1 (0)

0: \_ \_

01203...

Step2 (1)

0: 1 \_

1: \_ \_

01203...

Step3 (2)

0: 1 \_

1: 2 \_

2: \_ \_

# Initial Memory Consumption 例

- 012032004051025

Step1 (0)	Step2 (1)	Step3 (2)	Step4 (0)	Step5 (3)	Step6 (2)
0: _ _	0: 1 _	0: 1 _	0: 1 _	0: 1 _	0: 1 _
	1: _ _	1: 2 _	1: 2 _	1: 2 _	1: 2 _
		2: _ _	2: 0 _	2: 0 3	2: 0 3
				3: _ _	3: 2 _

Step7 (0)	Step8 (0)	Step9 (4)
0: 1 _	0: 1 _	0: 1 4
1: 2 _	1: 2 0	1: 2 0
2: 0 3	2: 0 3	2: 0 3
3: 2 0	3: 2 0	3: 2 0
		4: _ _

# Initial Memory Consumption 例

- 012032004051025

Step9 (4)	Step10 (0)	Step11 (5)	Step12 (1)	Step13 (0)
0: 1 4	0: 1 4	0: 1 4	0: 1 4	0: 1 4
1: 2 0	1: 2 0	1: 2 0	1: 2 0	1: 2 0
2: 0 3	2: 0 3	2: 0 3	2: 0 3	2: 0 3
3: 2 0	3: 2 0	3: 2 0	3: 2 0	3: 2 0
4: _ _	4: 0 _	4: 0 5	4: 0 5	4: 0 5
		5: _ _	5: 1 _	5: 1 0

2,5 は残るが埋まっており、かつ既存ノードなので無視

# Initial Memory Consumption 例

- 012032004051025

Step13 (0)

0: 1 4

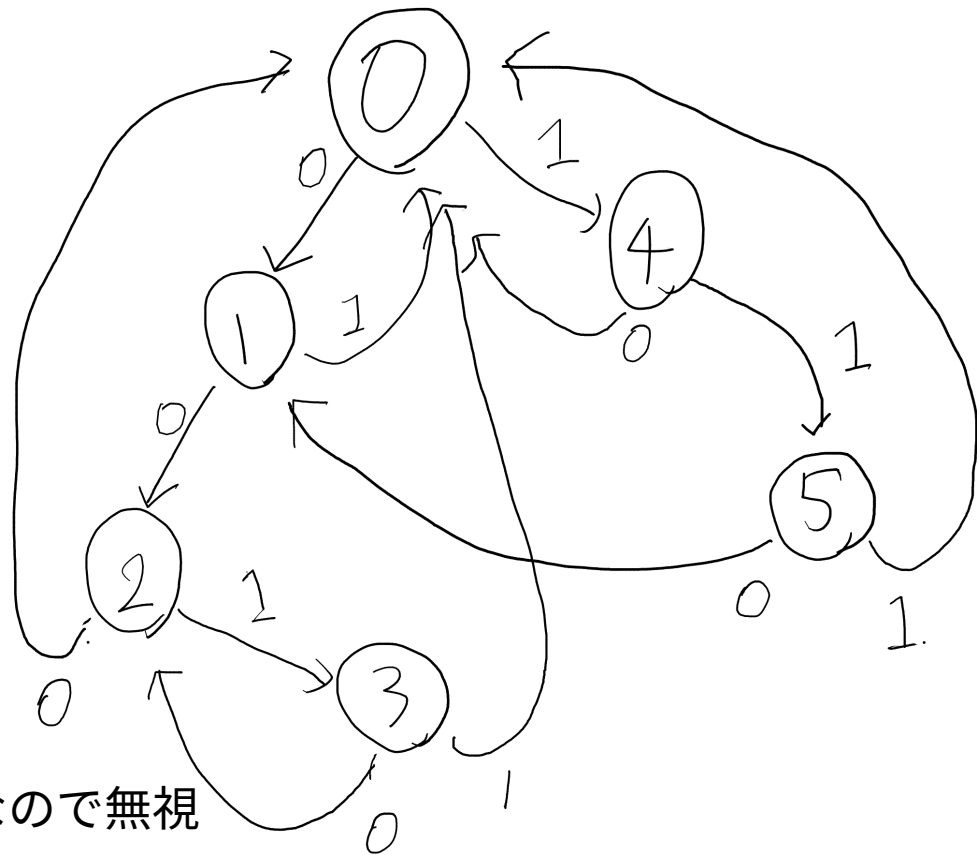
1: 2 0

2: 0 3

3: 2 0

4: 0 5

5: 1 0



2,5 は残るが埋まっており、かつ既存ノードなので無視

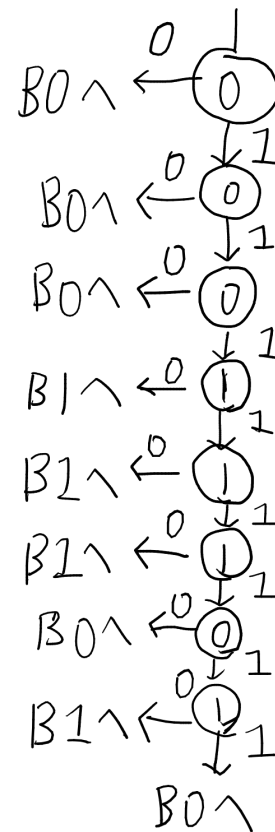
# Deserialize

- 入力データをグラフ構造に書き換えたい
- 入力を 0,1 のビット列にしたものをグラフ構造に置き換える手法
  - 一方向リンクリスト構造の変形版のようなかたち
- “parsed memory” などの言葉でも表わされる

# Deserialize

- 入力を1ビットごとに並べる
  - 下位ビットが先: 00000001 ではなく 10000000
- そして上から一つ一つを節点に変換
  - 0 なら 0-pointer を B0 に
  - 1 なら 0-pointer を B1 に
  - 1-pointer は次のビットを表わす節点
- 入力の終点は B0 に繋ぐ

10111000 の Deserialize  
下位bit から並べられるので  
00011101 の順序になる

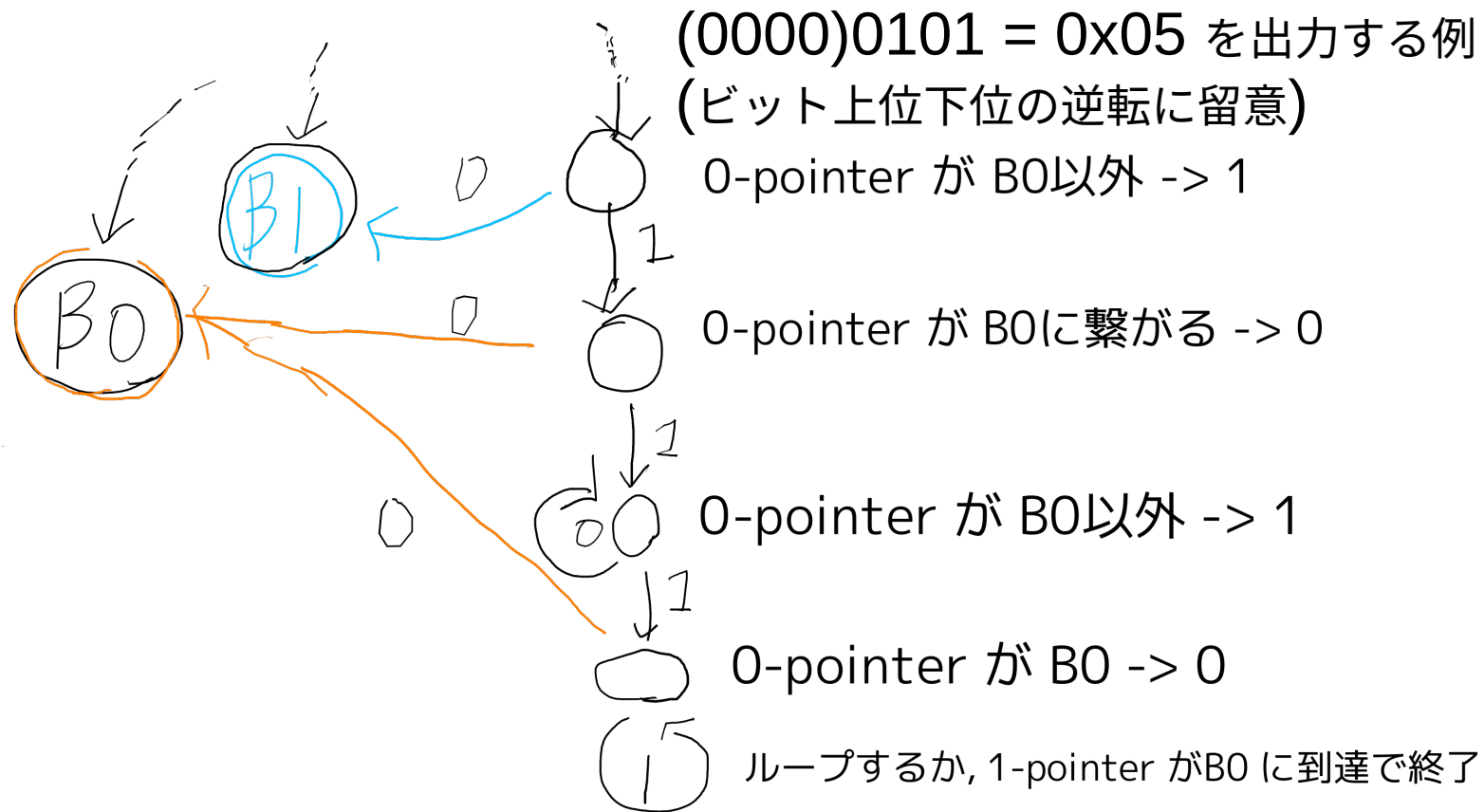




# Serialize

- 出力生成時、および後述のCommand の引数処理時に行なわれる
- グラフ構造で表されたものを 0,1 のビット列に変換
  - 起点から1-pointer をたどっていきながら各ノードについて判定. B0 にたどりつく or 同じノードを2度たどると終了.
  - 各ノードについて 0-pointer がB0 -> “0”
  - 各ノードについて 0-pointer がB1 -> “1”

# 例: 出力時のserialize



# 絶対アドレスと相対アドレスと値

- (この言葉は作者は使っていません; 筆者の命名です)
- 絶対アドレス: root ノードから 0, 1 とたどっていったときのそのノードの位置関係
  - 例: B0 ノードは `000`, B1 は `001`
- 相対アドレス: その位置からたどっていったときのそのノードの位置関係
- 値: そのノードから serialize したときにとるビット列
  - 例: 入力の値は 000101
- どれも `01010..` と表わされるので混乱しないよう注意

# Command

- Commandは3つ
  - Case 1: ノードの代入
  - Case 2: ノードの追加
  - Case 3: 条件分岐
- (それぞれの Case は本家では命名されていない)

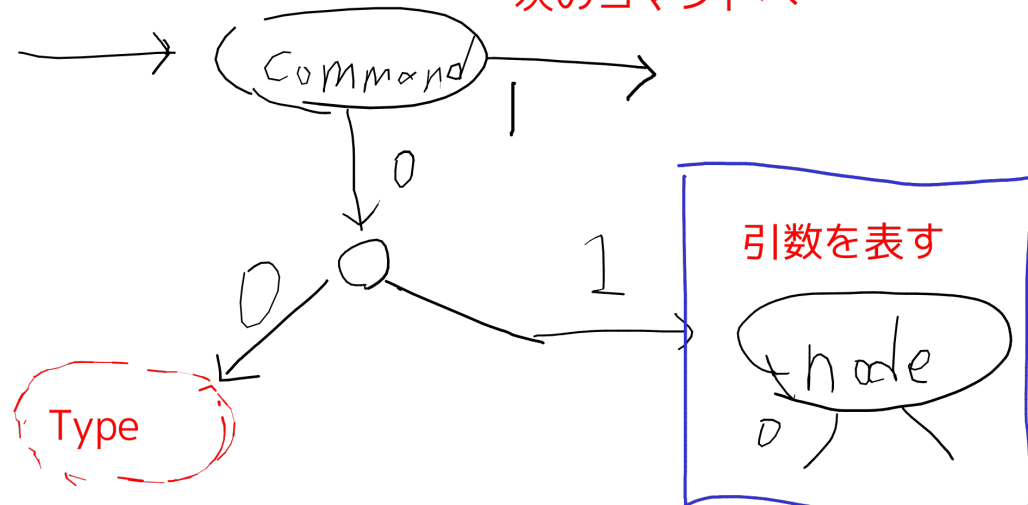
# Command の構造

- Command の一番上のノードを”相対アドレス”の基準とする
  - 0-pointer に種別や引数などのノードが連なる
  - 1-pointer に次のCommand が繋がる
- 作者は相対アドレス00 を type, 01 を node と(実装内で) 命名
- 相対アドレス00: type (Command の種類)
- 相対アドレス 01: 引数は node の中に入れる
- 次の処理は相対アドレス1に (Condition のThen を除く)

# Transceternal コマンドのグラフ構造

前のコマンドより

次のコマンドへ



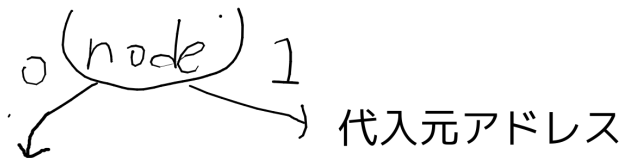
B0 : 代入

B1 : ノードの追加

それ以外 : 条件分岐

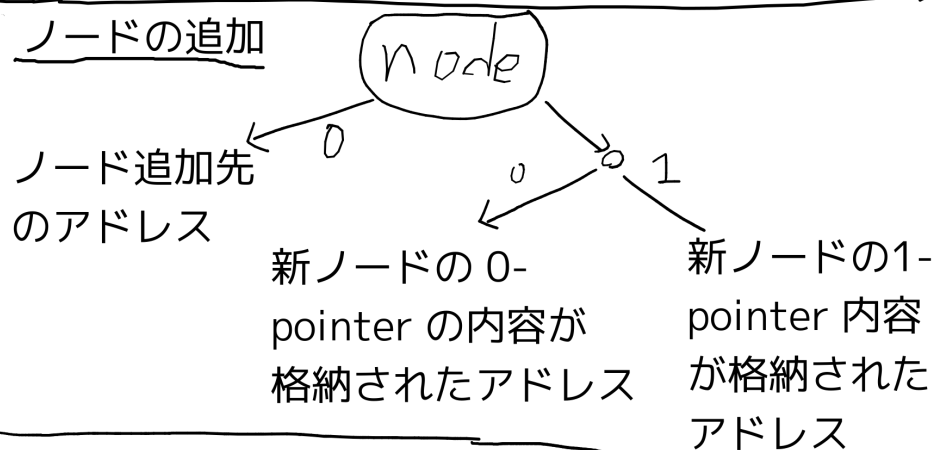
右図へ

代入

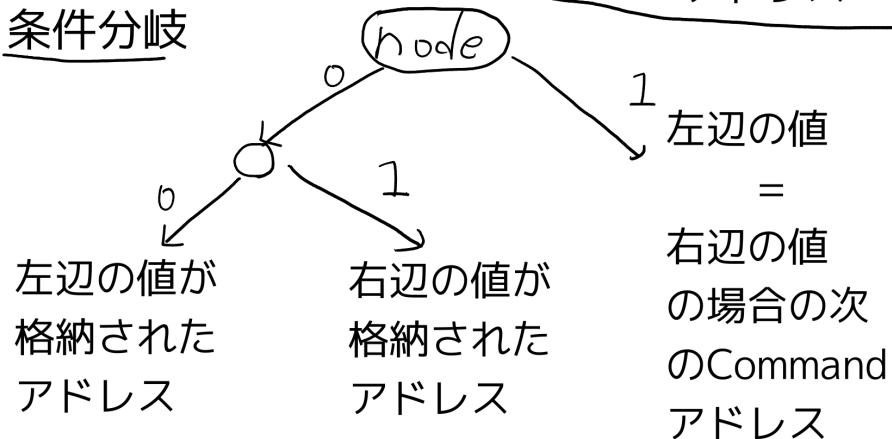


代入先のアドレス

ノードの追加



条件分岐



# Case 1: Assign (Set) ノードの代入

- type: 相対アドレス00 を B0 に
- 引数
  - 相対アドレス010 で指定された値のアドレスに
  - 相対アドレス011 で指定された値をアドレスのノードを代入

# Case 2: Allocate ノードの追加

- type: 相対アドレス00 を 001 (B1)
- 引数は3つ
  - 01010 の値をノード配置先のアドレスに
  - 010110 の値を 0-pointer に
  - 010111 の値を1-pointer に
  - それぞれを0-pointer, 1-pointerにもつノードを作成し01010 のアドレスに代入
- 今回の回答では未使用



# Case 3: Condition 条件分岐

- type: 相対アドレス00 は B0 か B1 以外
- 引数は3つ
  - 相対アドレス0100 : 比較対象の左辺
  - 相対アドレス 0101: 比較対象の右辺
  - 相対アドレス 011 : 真な場合のジャンプ先
- 相対アドレス0100 と相対アドレス0101 の値をアドレスにしたときのノードが同一のノードかどうかを比較
  - then : 相対アドレス011 を 01 に代入
  - else : 絶対アドレス011 を 01 に代入 (他のCommandと同じ)

# 注意: 次のCommandの指定先

- “root” “admin” ではない
  - 最初忘れていた
- 次の状態となるのは「Command の一番上のノード」
  - 絶対アドレス01の内容がroot にもしあると無限ループに陥る.

# 実例で流れを追う

- cat を例にして説明します
- プログラム
  - cat
- 入力
  - d (0x64 = 0b01100100)
- 出力
  - d (0x64 = 0b01100100)

# 実行の流れ :cat編

- ソースコードを token に分割
- ソースコードをグラフ構造に落とす (Initial Memory Consumption)
- 入力を deserialize してグラフ構造に
- 新root ノードを作成し, `0` にソースコード, `1` に入力接続
- 実行 (main loop). グラフノード内をグラフノードを操作しながら移動
  - cat ではここは0回で終了
- Command が B0 になれば main loop 脱出
  - cat ではいきなり B0 になる
- `1` 以下を serialize して出力内容を生成

# ソースコードをtoken に分割: cat編

- catacat は1文字ずつToken に
- “c” “a” “t” “a” “c” “a” “t”

# Initial Memory Consumption: cat編

- “c” “a” “t” “a” “c” “a” “t” をもとに Initial Memory Consumption を行ないグラフ構造に

Step1: (c)

c: \_ \_

Step2: (a)

c: a \_

a: \_ \_

Step3: (t)

c: a \_

a: t \_

t: \_ \_

Step4: (a)

c: a \_

a: t \_

t: a \_

Step5: (c)

c: a \_

a: t \_

t: a c

Step6: (a)

c: a \_

a: t a

t: a c

Step7: (t)

c: a t

a: t a

t: a c

# Initial Memory Consumption: cat編

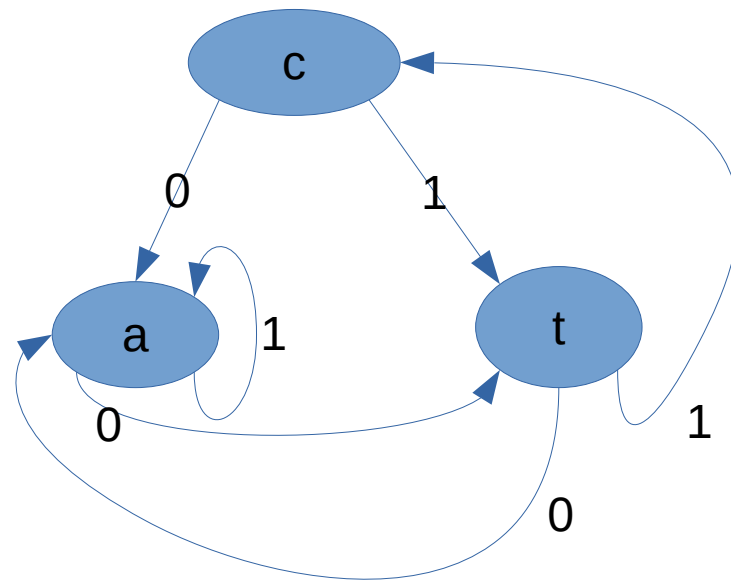
- “c” “a” “t” “a” “c” “a” “t” をもとに Initial Memory Consumption を行ないグラフ構造に

Step7: (t)

c: a t

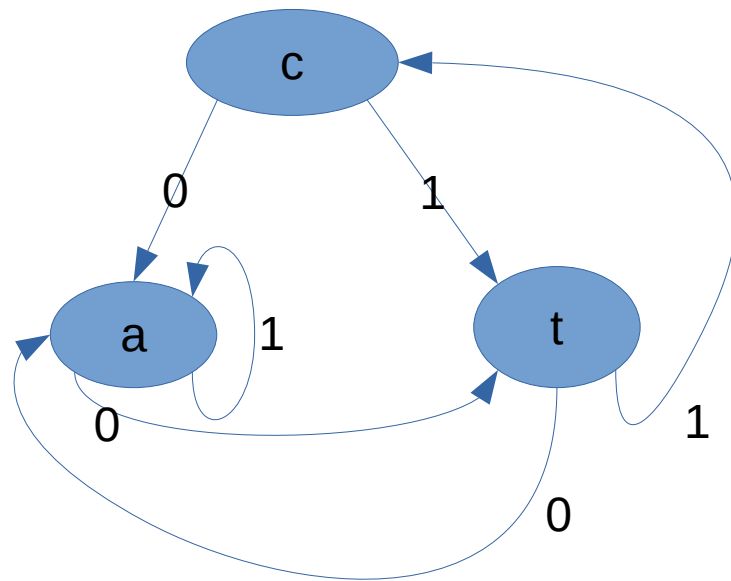
a: t a

t: a c



# 特別なノード: cat編

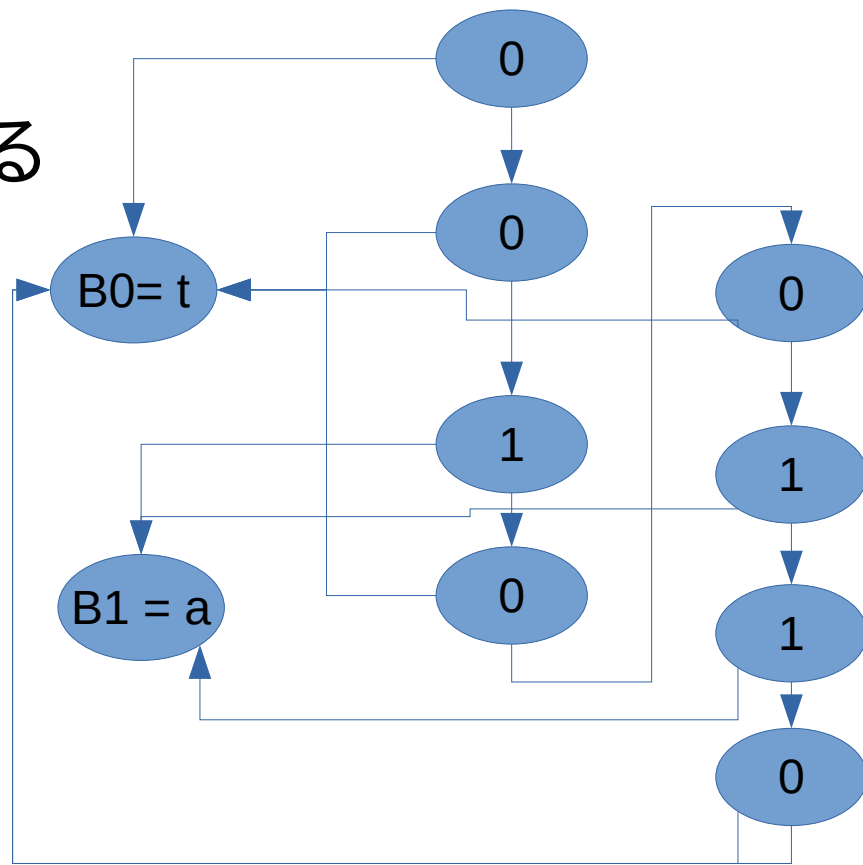
- root
  - c
- admin
  - c  $\rightarrow$  a とたどれるので a
- B0
  - c  $\rightarrow$  a  $\rightarrow$  t とたどれるので t
- B1
  - c  $\rightarrow$  a  $\rightarrow$  a とたどれるので a





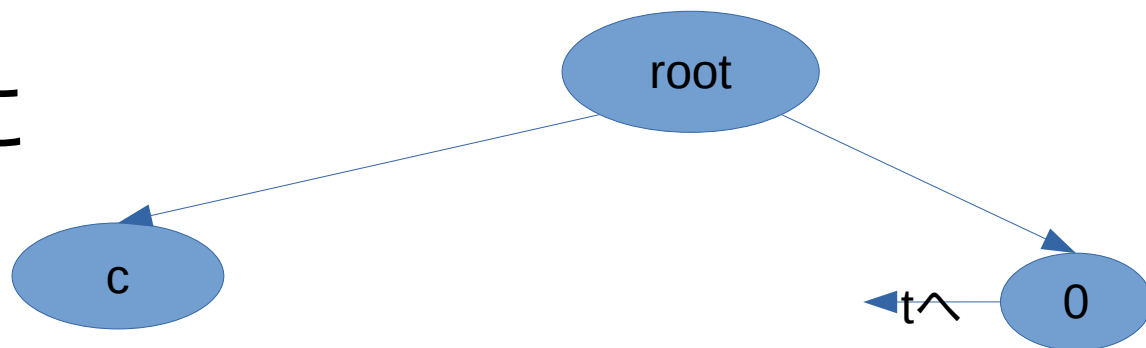
# 引数の Deserialize: cat編

- 入力: 'a' = 0b01100100 は 00100110 として読み込まれる
  - '0' は 0-pointer を B0 に
  - '1' は 0-pointer を B1 に
  - 1-pointer は次の文字に

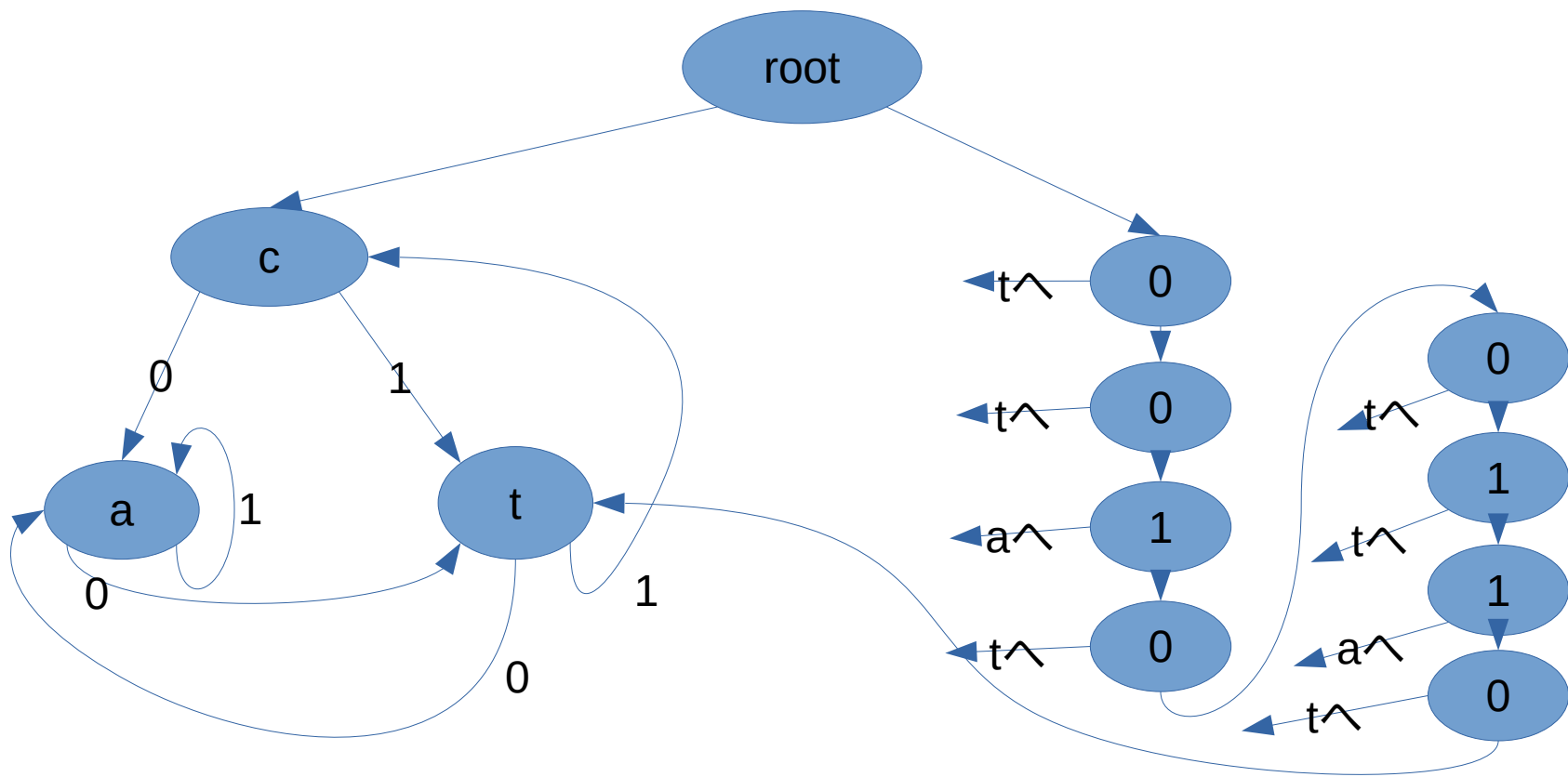


# 新しいrootノードを構築

- 新しいrootノードが構築され
- Initial Memory Consumption で作られたものを 0-pointer に
- 入力を 1-pointerに

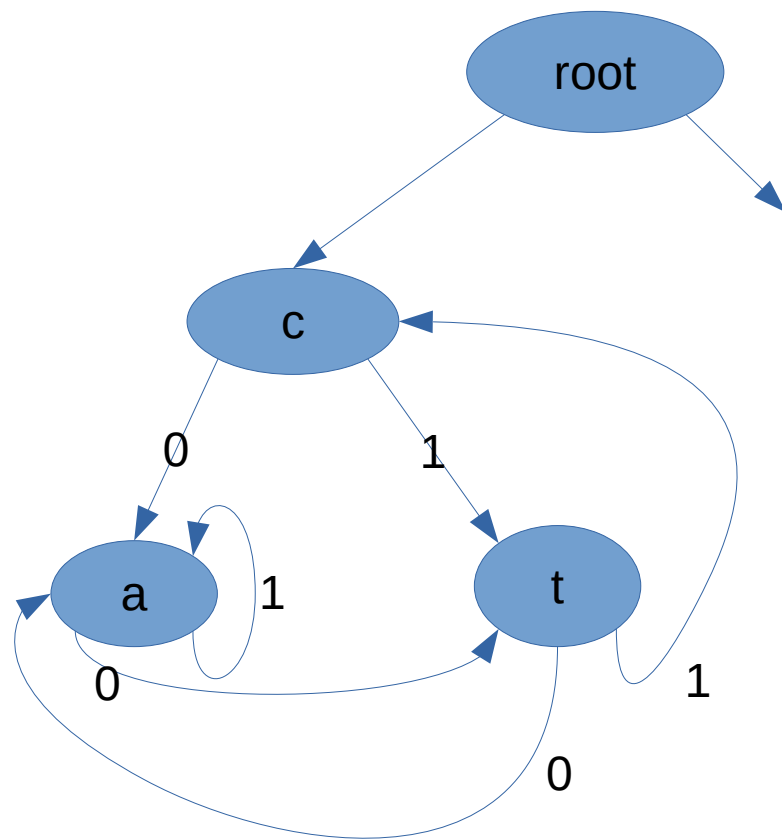


# 新しいrootノードを構築



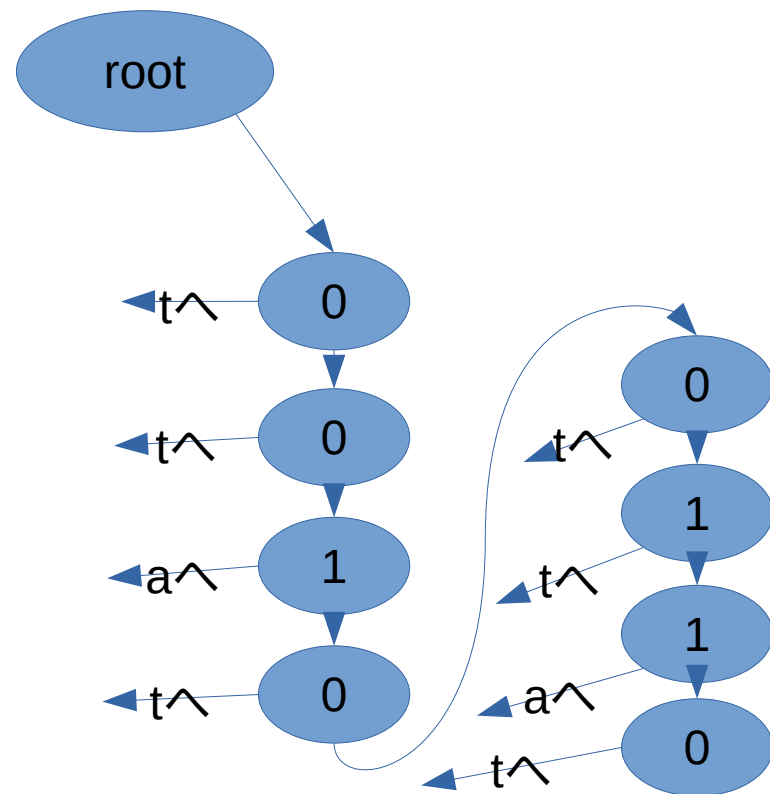
# 実行

- 終了条件の判定
  - 000 の節点 が 01の節点と一致するか
  - 000 は root -> c -> a -> t
  - 01 は root -> c -> t
  - 一致
- 一致しなければ 01 のCommand  
実行



# 出力内容のSerialize

- 絶対アドレス1 を起点にSerialize していく
- 0-pointer が  $B0 = t \rightarrow 0$
- 1-pointer が  $B0$  以外  $\rightarrow 1$
- “01000110” を上位ビットから並びかえ “0x64 = d” が出力されて終了
- 以上により “cat” が実現できた



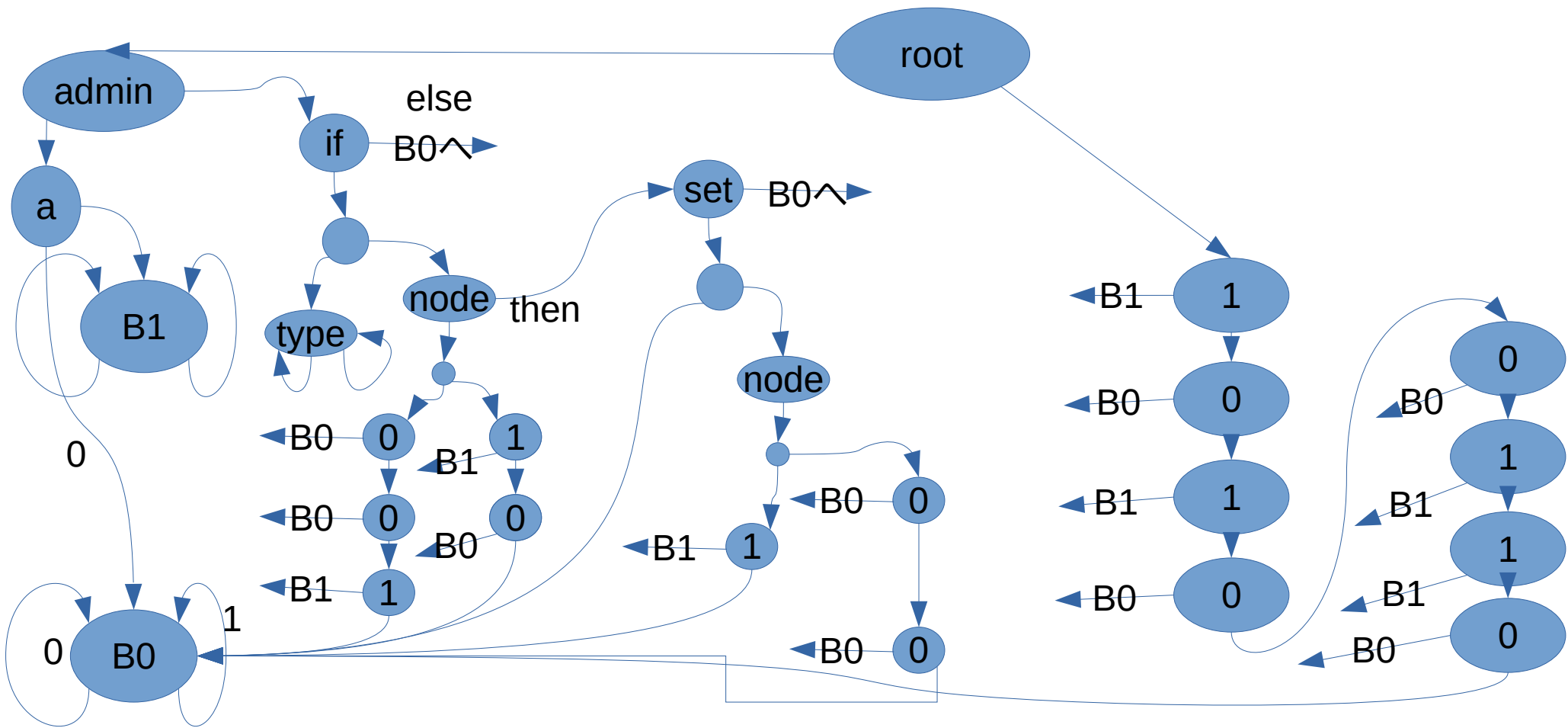
# Q&A

- B0 (000) と B1 (001) の節点はどこ？
  - B0, B1 は他と共通で独立していなくてもよい
  - a は 絶対アドレス00 と B1 の節点を兼ねている
  - t は B0 でもあり最初に実行されるCommand (終了条件) を示す
  - Transceternal の構造を示した図などでは別々にしている
  - cat の例にならうとソースコードをより小さくできるかも？

# 実例2

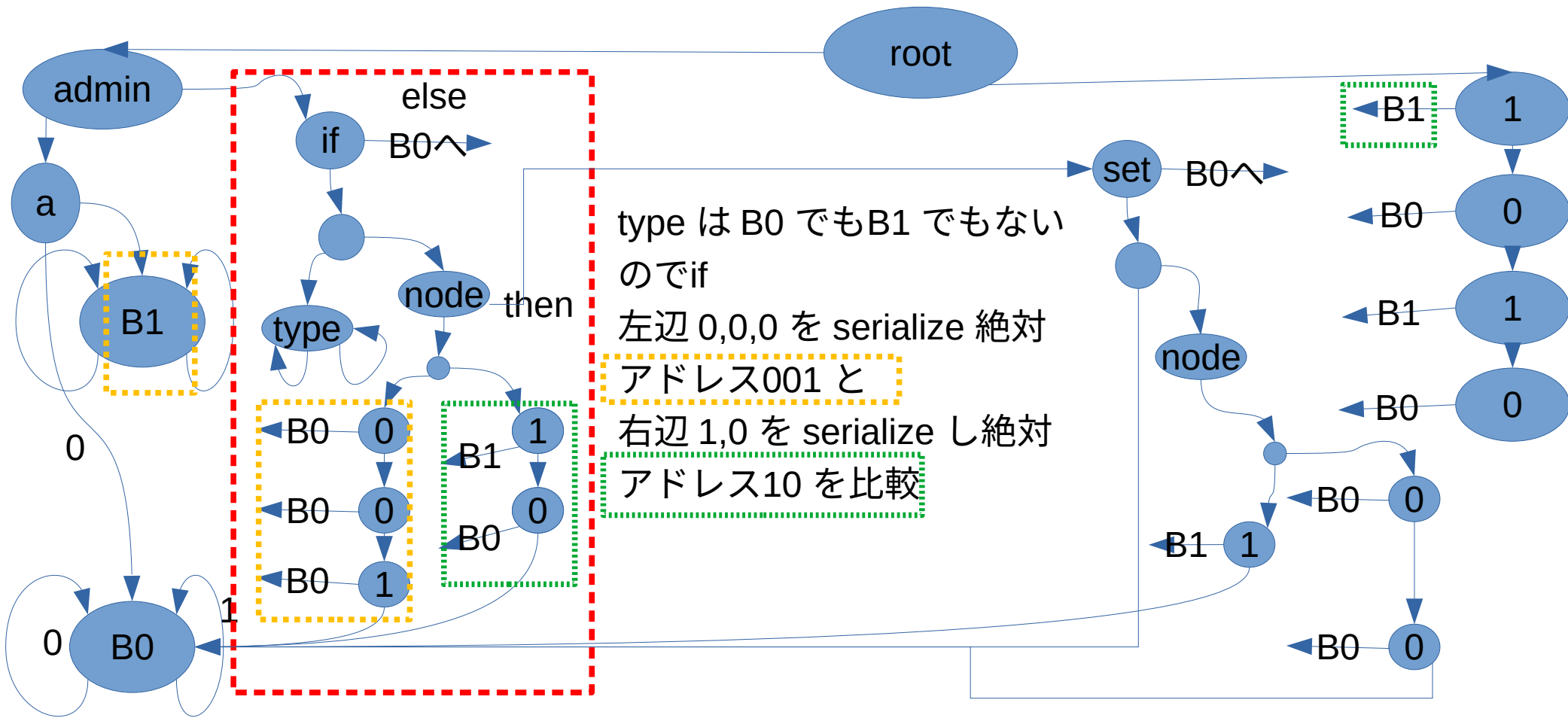
- 入力の8ビット目が1なら00000010 を, それ以外なら入力そのままの内容を出力します
- 全過程を説明すると長すぎるため構築過程を飛ばしてグラフを示します
  - Commandの使い方
  - 遷移の方法
- 入力
  - 01100101 (8ビット目は1)

## 实例2

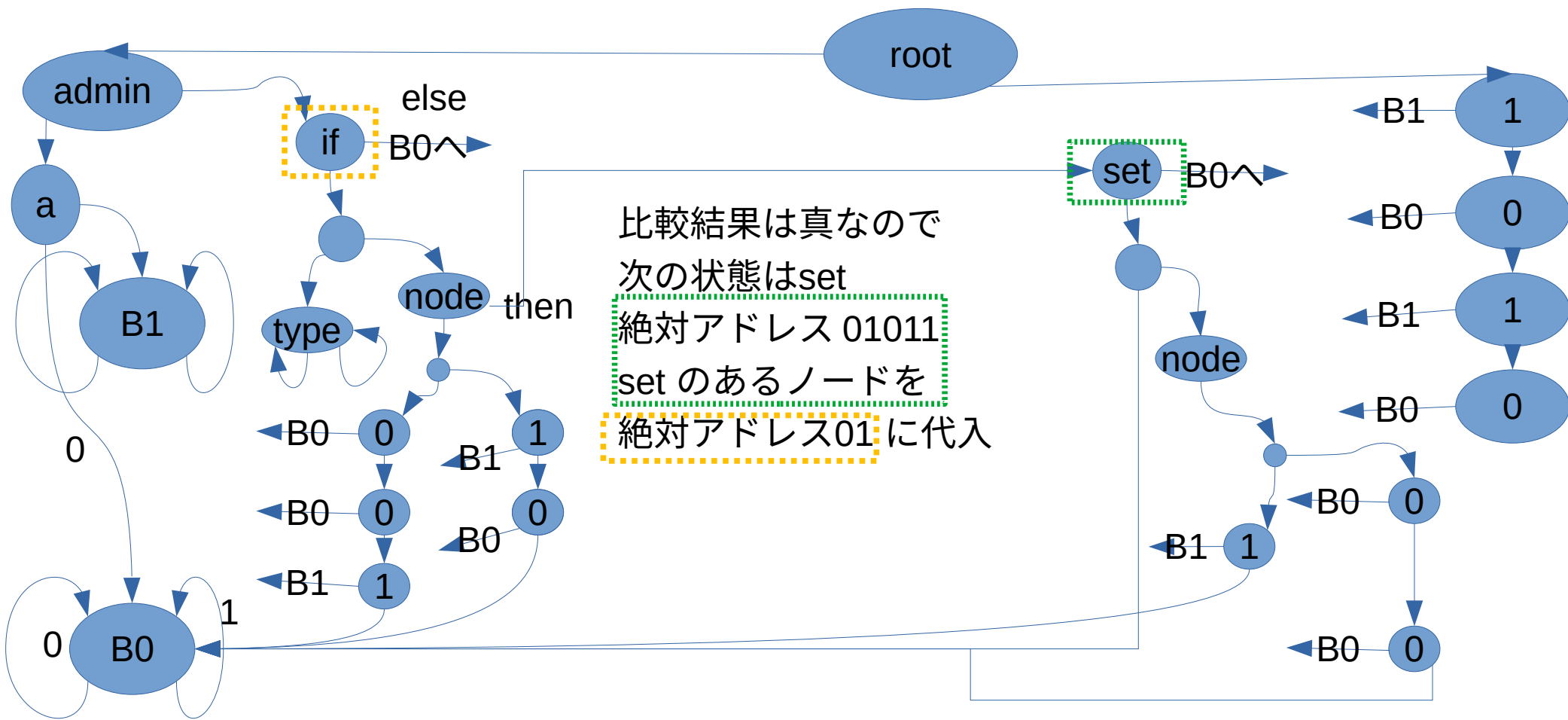




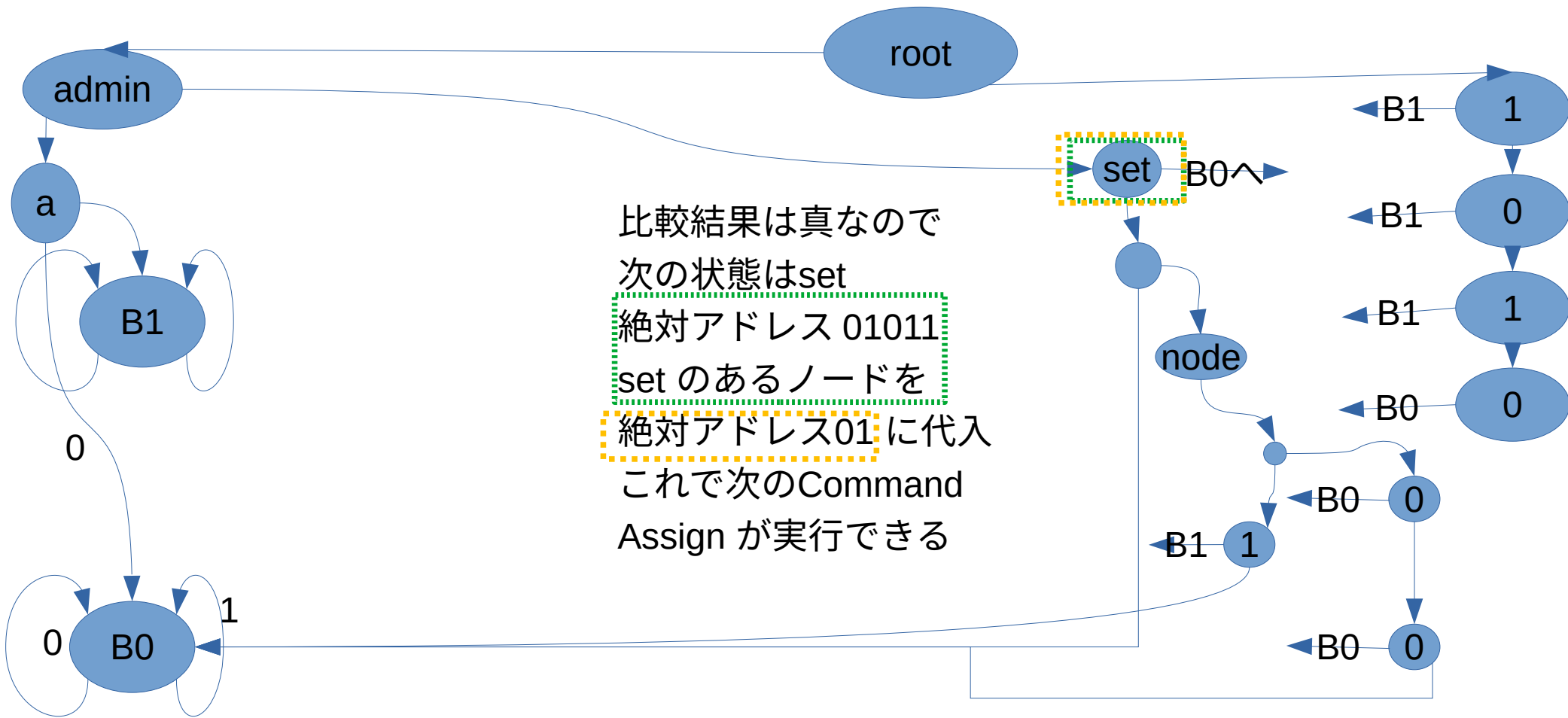
# 実例2の動きを見る: if



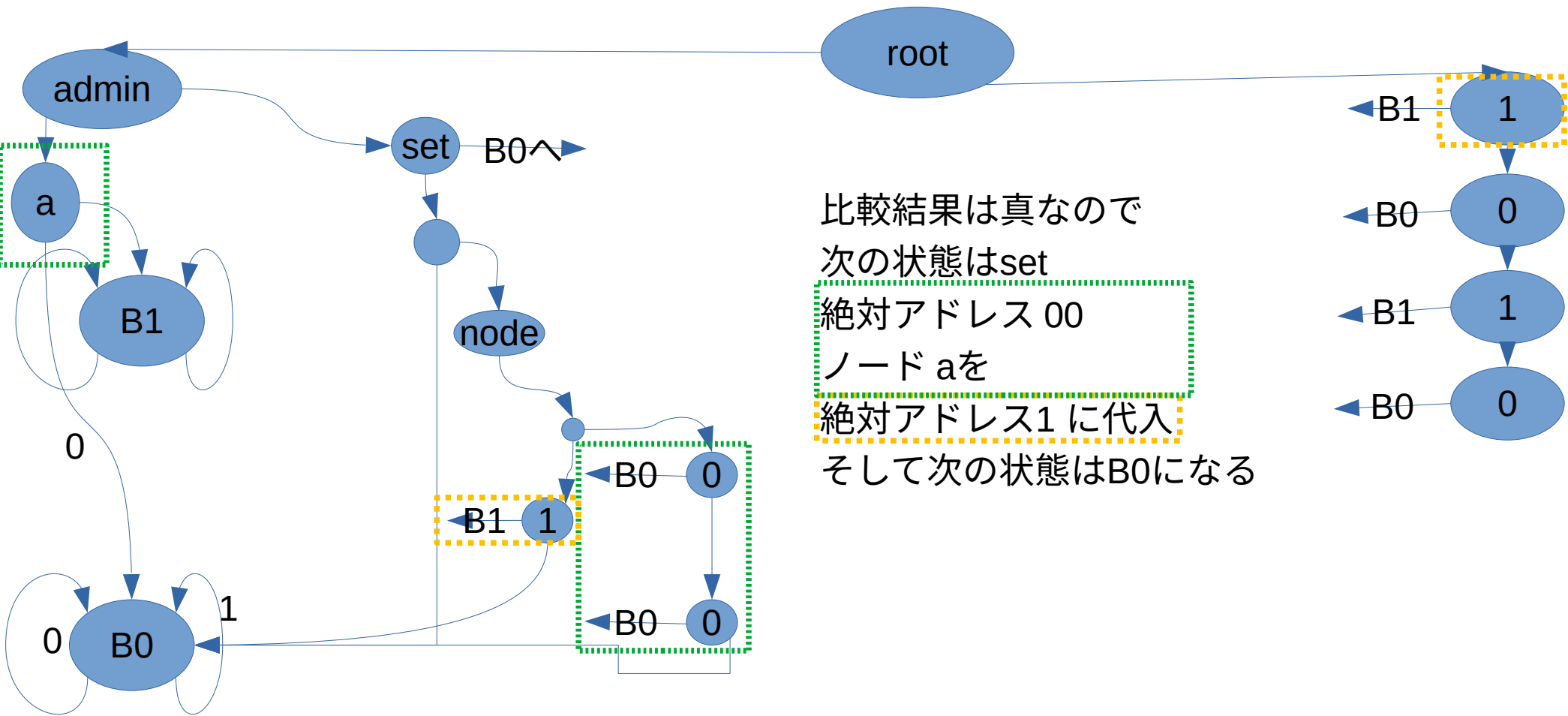
# 実例2の動きを見る: if後の状態遷移



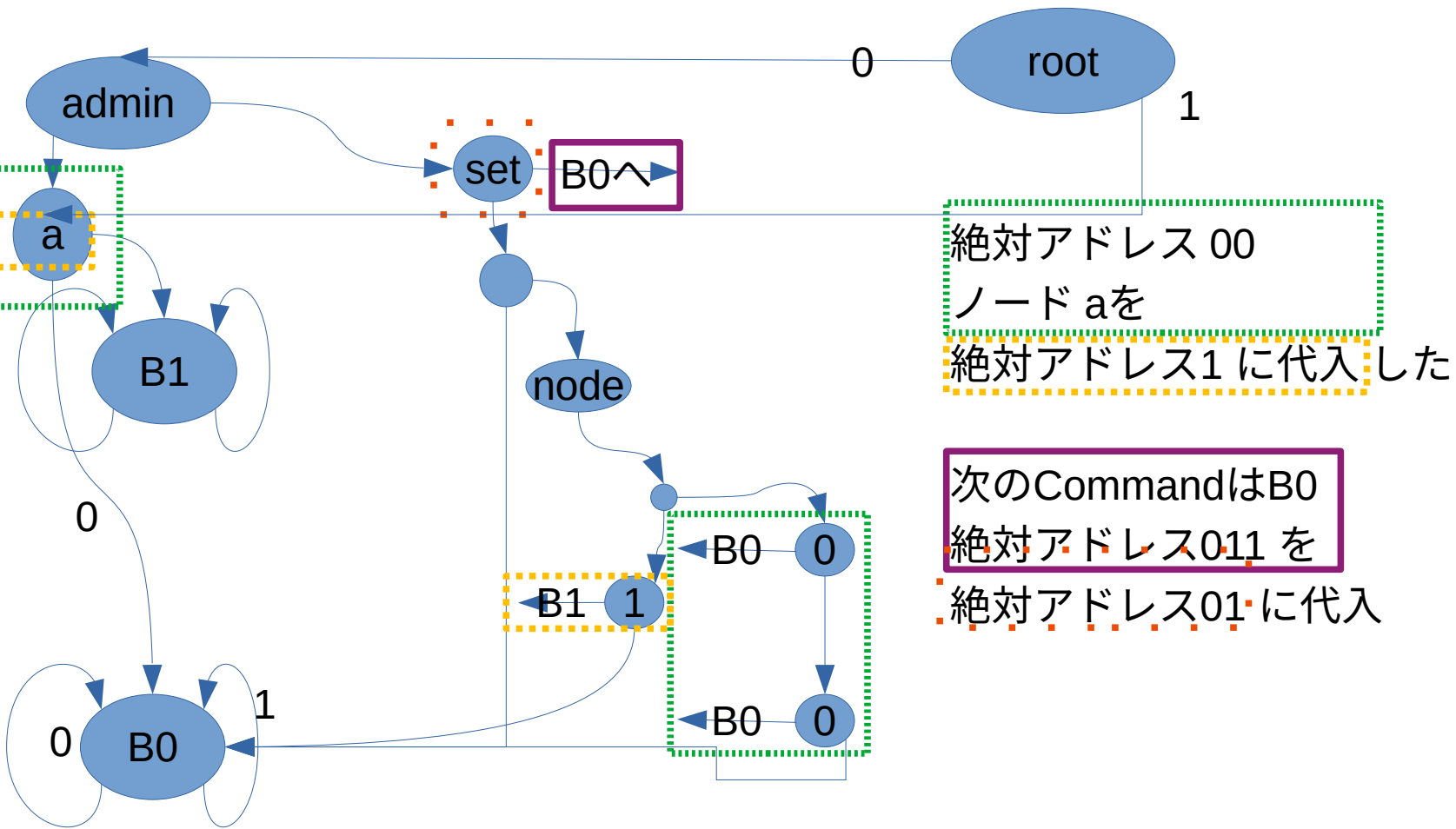
# 実例2の動きを見る: if後の状態遷移



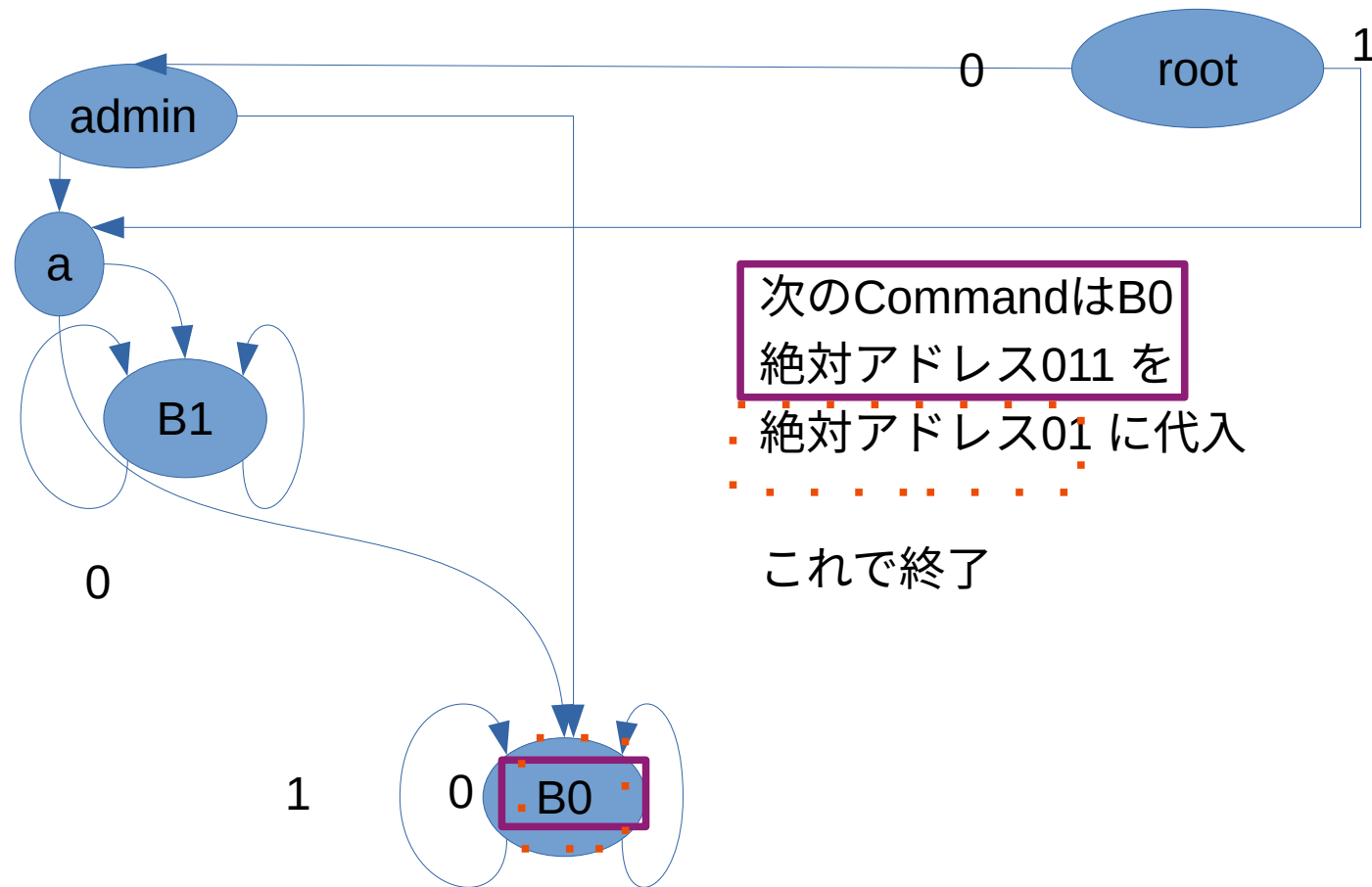
# 実例2の動きを見る: Assign



# 実例2の動きを見る: Assign



# 実例2の動きを見る: Assign



次のCommandはB0

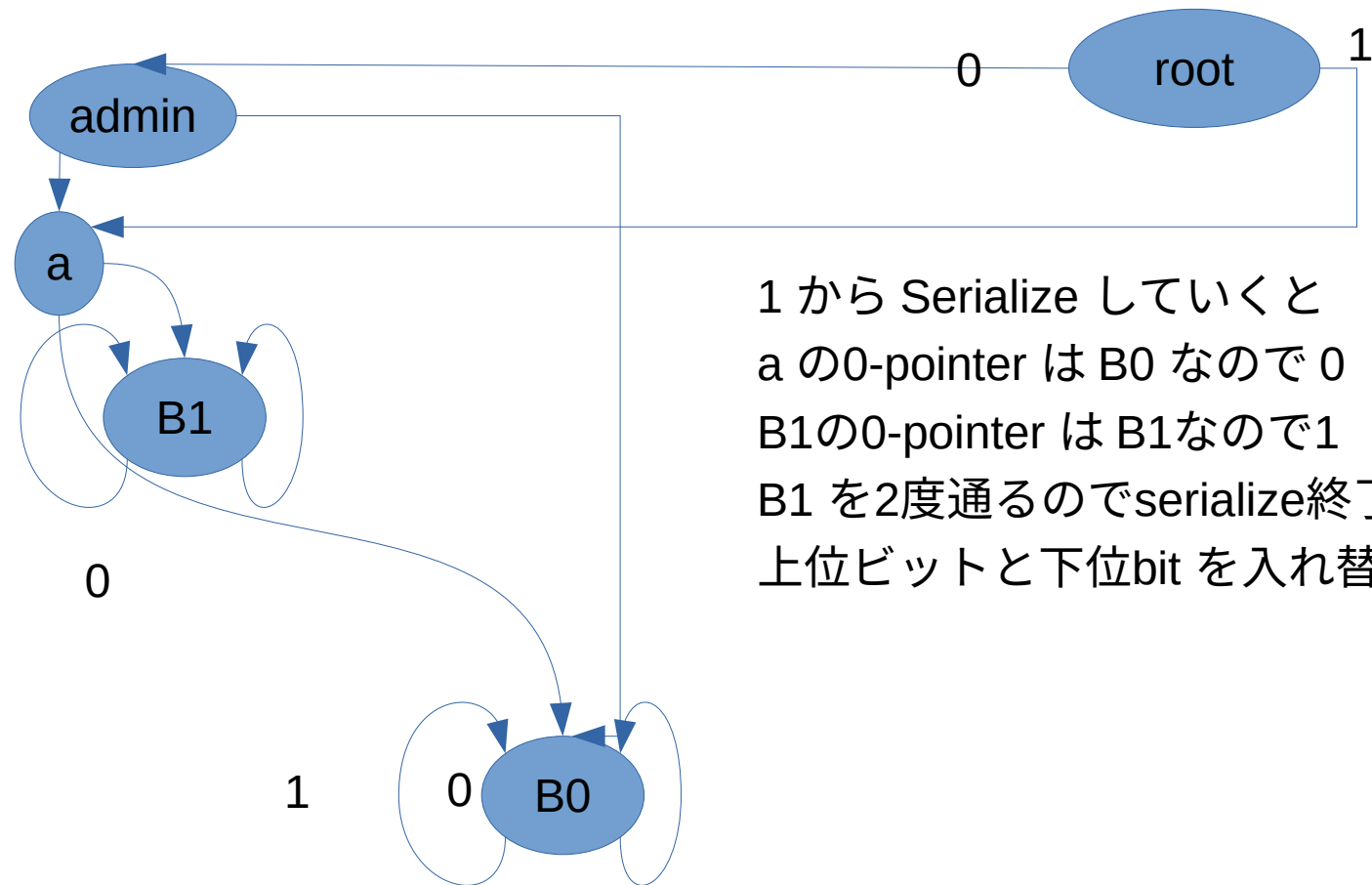
絶対アドレス011 を

絶対アドレス01 に代入

.....

これで終了

# 実例2の動きを見る: Serialize



1 から Serialize していくと  
a の0-pointer は B0 なので 0  
B1の0-pointer は B1なので1  
B1 を2度通るのでserialize終了  
上位ビットと下位bit を入れ替え出力

# Q&A

- 代入したノードの参照関係は？
  - 複製はなされない.
  - 複製が必要なときはノード追加必要あり
- プログラム実行中の入出力は可能か？
  - 不可能.



# Q&A

- 入出力を参照するのが不可能では？
  - できる. Assign Command の「絶対アドレス00100」が指すノード、ではなく「絶対アドレス00100」以下が指すノードをDeserialize した結果をアドレスとして取る
- 各Command の引数としては任意の絶対アドレスへのポインタを構成できる
- 引数に 文字 “x” などの定数を使いたい
  - 素直にはできない. 後ほどの定数の使い方の項を参照

# やりたいこと

- 言語仕様は把握できた
- ものの、実際に問題を解くためにはまだまだ足りない
  - 定数/変数
  - 配列 (一方向リンクリスト)
  - 入出力 (の書き換え)
  - If-then-else
  - While
- 以上を実現したい

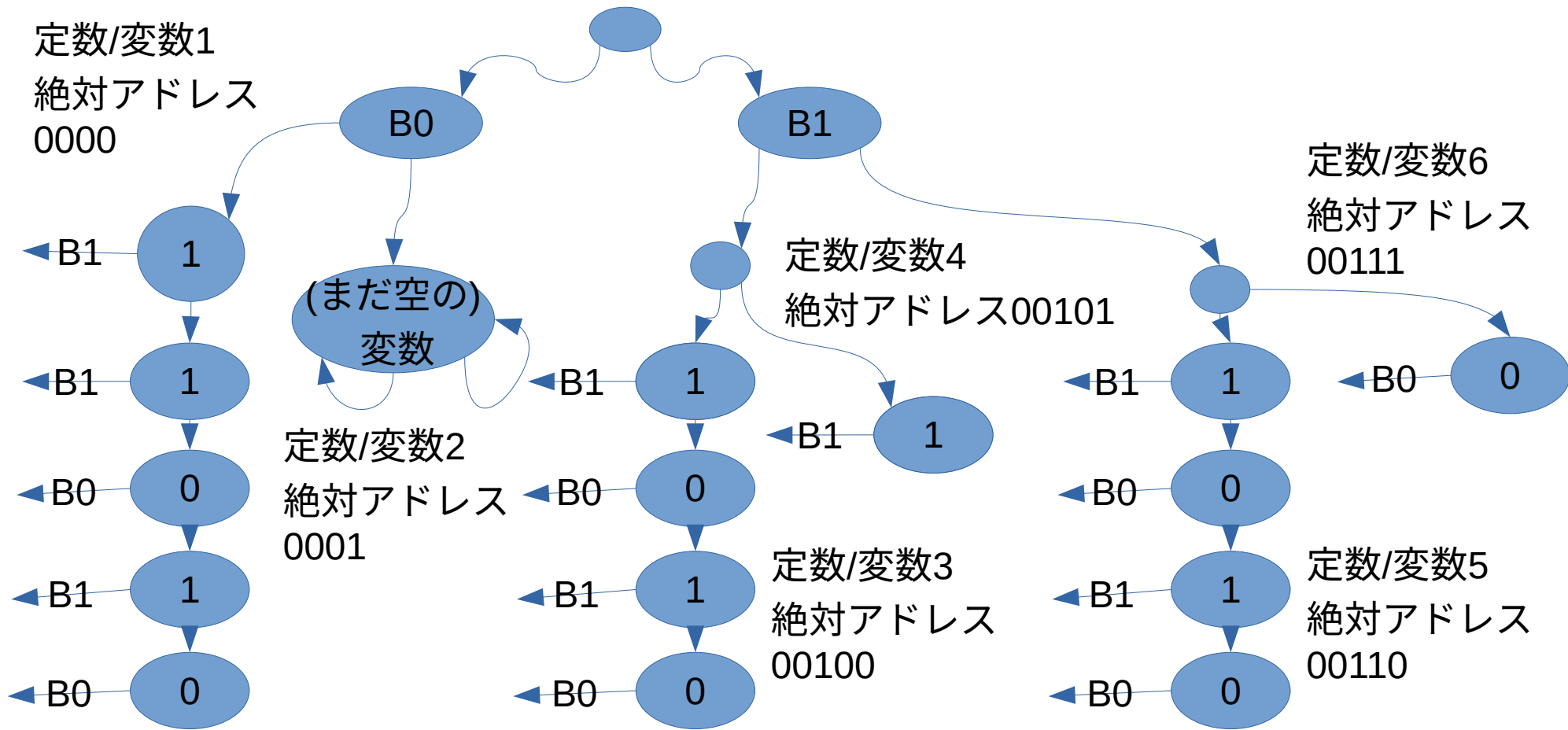
# 定数と変数を実現する

- 定数が必要な理由
  - if 文のアドレスとして定数cを使いたい
  - 特定のデータを格納したい
  - 0や1以外の数値を表したい
  - などなど...
- Command の引数はserialize したときのアドレスとして解釈される.
- 特定のアドレスに「serializeしたときに定数c になるノード」を置く
  - どこに置くのか？ -> 次のスライド

# 定数と変数を実現する

- 大抵の領域は自由には使えない
  - 絶対アドレス `1` 以下は入出力のためのデータ領域
  - 絶対アドレス `01` 以下はそれぞれのCommandを表す
- 一方絶対アドレス `000` or `001` 以下は 0-pointer, 1-pointer とともに自由に使える
- 各変数を `001` 以下の特定の領域に格納
- 複数の変数の実現
  - 二分木のどこかの要素に割り付ける
    - 二分木なので  $O(\log(\#変数))$  の深さで指定できる
    - なお、長くなると変数の絶対アドレス指定にノードを浪費するのでなるべく減らしたい

# 定数と変数を実現する：変数6個の例

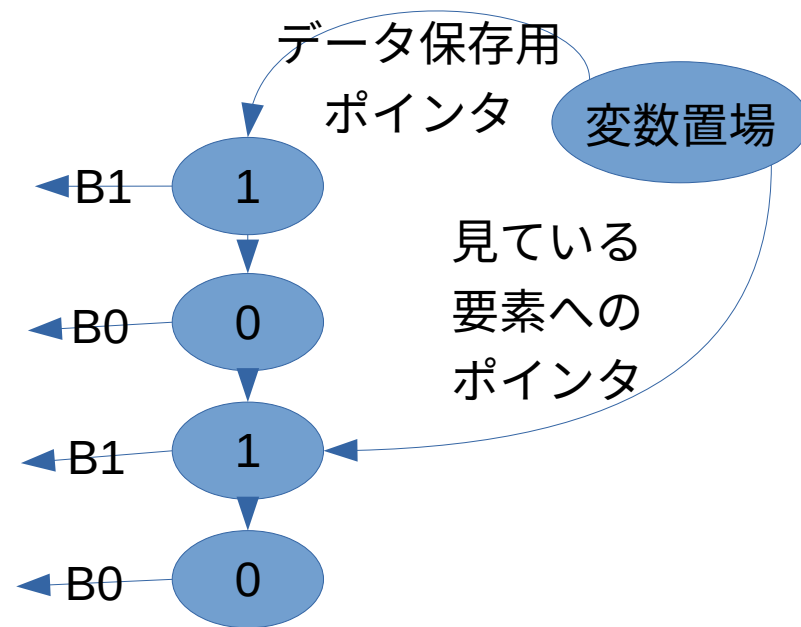


# 単方向リンクリスト

- リスト構造そのものの実現は
  - 0, 1, 0,... とならべ、1-pointer をたどっていけば実現できる
- “たどる” 処理の実現に工夫が必要
  - Command のノードで指定できる値はあくまで固定されたアドレス
  - それに対して具体的な処理を行うのに“特定のアドレスを指したときの場所がどんどん変わっていく”変数を導入する必要あり

# 単方向リンクリスト

- 複数の要素があるので任意の位置を指定したい
  - データ保存用ノードと、それを指すポインタノードを作成
  - ポインタノードを介してやりとり
- ポインタそのものをつけかえると意味がない
  - 接続が壊れないようポインタノードの「次」を指定する



# 入出力の扱い

- 入力をそのままにしておけば勝手に出力になる
  - なので `cat` はシンプルに書ける
- 出力の特定部分の取り出しは特定bit を複製すればよい
  - “続くbit全体の取り出しは絶対アドレス 1111...1
  - そのbit単体の取り出しは絶対アドレス 111...10 として0 をどこかに代入, それが0 か1 かで判定
- 入力を書き換えるには 1 に向かってset すればよい



# (カウンタ変数のための)引き算

- (今回は定数の引き算のみ扱う)
- 簡単な扱いかた: 1進法 (‘1111....1’) をひたすら並べる、その桁数が値となる.
- 定数の引き算
  - 定数のアドレスを 111 とすると
  - ‘111’ + ‘差し引きたい数分の1’ を ‘111’ に代入すればよい
- 2進法表記での扱いは将来的な課題

# if文

- 基本的に Command を使えばよいが...
- 例: 入力の最初が0 か1 かを調べる
  - `1` と `000` (B0) を調べるのでは不適切
  - ノードそのものの一致を調べるため、入力が存在しない (入力の終端はB0 になるから) かを調べることになる
- 末尾に `0` をつけることでそのビット単体が 0か 1かを比較できる
  - `10` と `000` を引数にとればよい

# (do-)while / for

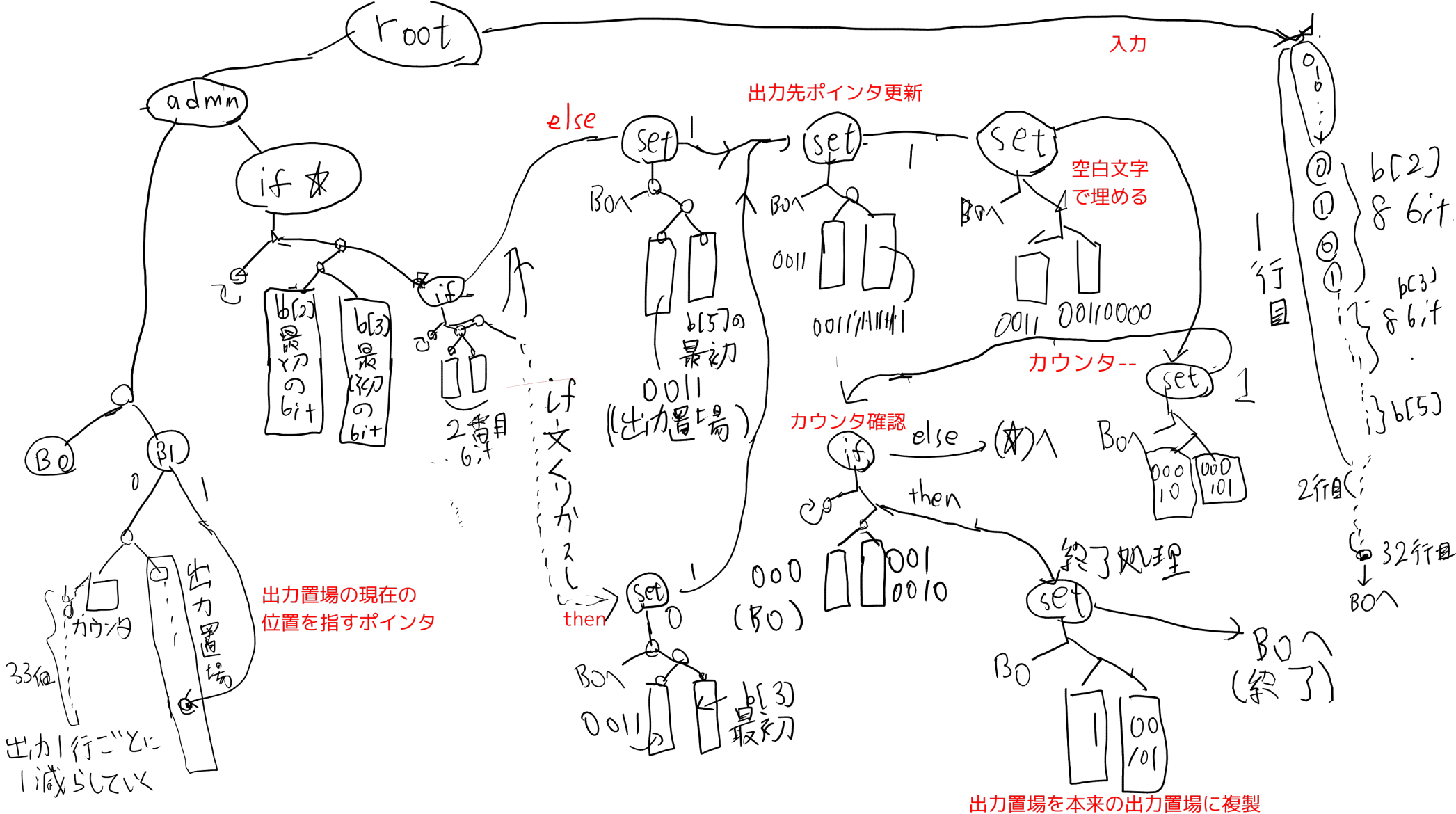
- if 文とカウンタ変数を用意
- 終了条件が満たされなければ else 以下に飛ぶ  
という仕様にする

# おしながき

- 問題の復習
- Transceternal の説明
  - グラフの各ノードの意味
  - 入出力と deserialize, serialize
  - コマンドの意味
  - 解答するための部材
- 解答の説明
- 今後 (Golf の方針)

# ソースコード (1560バイト)

```
0 1 2 2 2 3 4 5 3 6 3 7 3 8 3 9 3 A 3 B 3 C 3 D 3 E 3 F 3 G 3 H 3 I 3 J 3 K 3 L 3 M 3 N 3 O 3 P 3 Q
3 R 3 S 3 T 3 U 3 V 3 W 3 X 3 Y 3 Z 3 a 3 b 3 c 2 c d 3 2 d 2d 2e 2f 2f 2f 2g 2h 1p 3 1q 3 1r 3 1s 3
1t 3 1u 3 1v 3 1w 3 1x 3 1y 2 1y 1h 3 1i 3 1j 3 1k 3 1l 3 1m 3 1n 3 1o 3 1p 2Y 2Z 2a 2a 2a 2b 2c
1o 1g 3 1h 2T 2U 2V 2V 2V 2W 2X 1n 1f 3 1g 2O 2P 2Q 2Q 2Q 2R 2S 1m 1e 3 1f 2J 2K 2L 2L 2L
2M 2N 1l 1d 3 1e 2E 2F 2G 2G 2G 2H 2l 1k 1c 3 1d 29 2A 2B 2B 2B 2C 2D 1j 1b 3 1c 24 25 26 26
26 27 28 1c 1k 1z 20 21 21 21 22 23 1a 3 1b 1i e u 2 t f 2 g 2 h 3 i 3 j 3 j k 3 l 3 m 3 n 3 o 3 p 3 q 3
r 3 s 3 s 2i 3o 2 3n 2j 3 2j 2k 3 2l 3 2m 3 2n 3 2o 3 2p 3 2q 3 2r 3 2s 3 2t 3 2u 3 2v 3 2w 3 2x 3 2y
3 2z 3 30 3 31 3 32 3 33 3 34 3 35 3 36 3 37 3 38 3 39 3 3A 3 3B 3 3C 3 3D 3 3E 3 3F 3 3G 3 3H
3 3I 3 3J 3 3K 3 3L 3 3M 3 3N 3 3O 3 3P 3 3Q 3 3R 3 3S 3 3T 3 3U 3 3V 3 3W 3 3X 3 3Y 3 3Z 3
3a 3 3b 3 3c 3 3d 3 3e 3 3f 3 3g 3 3h 3 3i 3 3j 3 3k 3 3l 3 3m 3 3m 3p 47 2 46 3q 2 3r 2 3s 3 3t 3
3t 3u 2 3v 2 3w 3 3x 3 3y 3 3z 3 40 3 41 3 42 3 43 3 44 3 45 3 45 48 4N 2 4M 49 2 4A 2 4B 3 4C 3
4D 3 4D 4E 2 4F 2 4G 2 4H 3 4I 2 4J 2 4K 2 4L 2 4L 4O 4d 2 4c 4P 2 4Q 2 4R 3 4S 2 4T 2 4U 3
4U 4V 2 4W 2 4X 3 4Y 2 4Z 2 4a 3 4b 3 4b 4y 4z 50 50 50 51 52 4o 2 4p 2 4q 2 4q 4r 2 4s 2 4t 3
4u 2 4v 2 4w 3 4x 2 4x 4e 4n 2 4m 4f 3 4f 4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l 2 2d v 1Z 2 1Y w 2 x 2 y 3
z 3 10 3 10 11 3 12 3 13 3 14 3 15 3 16 3 17 3 18 3 19 3 1A 3 1B 3 1C 3 1D 3 1E 3 1F 3 1G 3 1H
3 1I 3 1J 3 1K 3 1L 3 1M 3 1N 3 1O 3 1P 3 1Q 3 1R 3 1S 3 1T 3 1U 3 1V 3 1W 3 1X 3 1X 2i v v v v
v v v v
```



# 方針

- Python でジェネレータを記述
  - グラフ構造をまとめる関数を作成
  - スタックの構成をシュミレーションしながらソースコード文字列を出力 (idea by @drafear)
- ジェネレータのソースコード
  - <https://hiromi-mi.github.io/trans.py>
  - ライセンスは CC0 とします

# 今回の場合

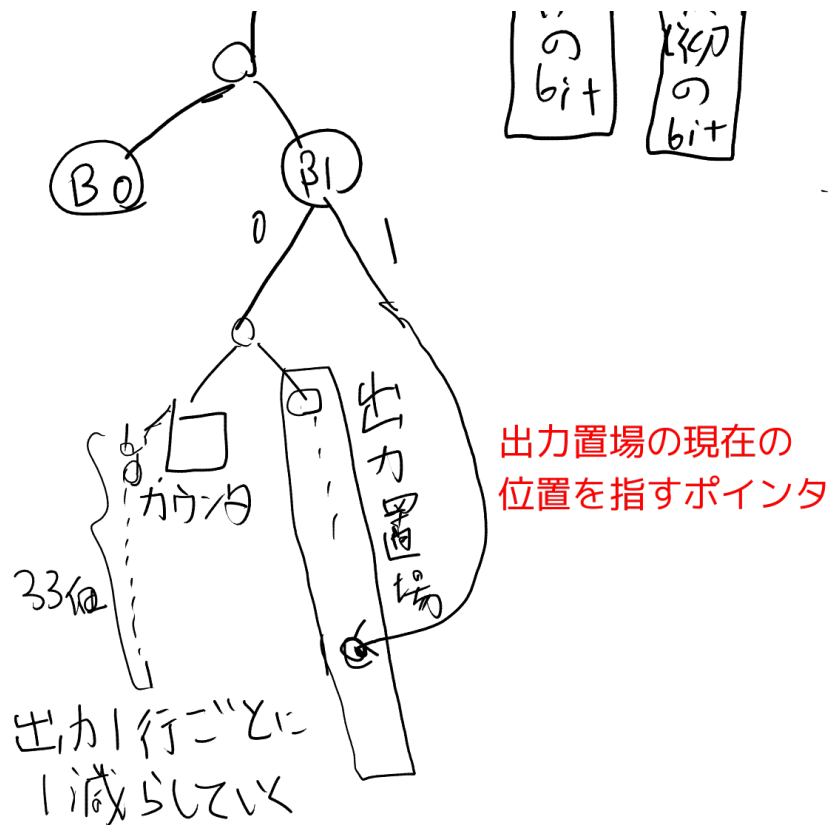
- token 列は0-9 + A-Z + a-z 62進法で生成
  - なるべく1バイトtoken で記述される節点を増やすとソースコードのバイト数が削減できる



# ソースコードの概要

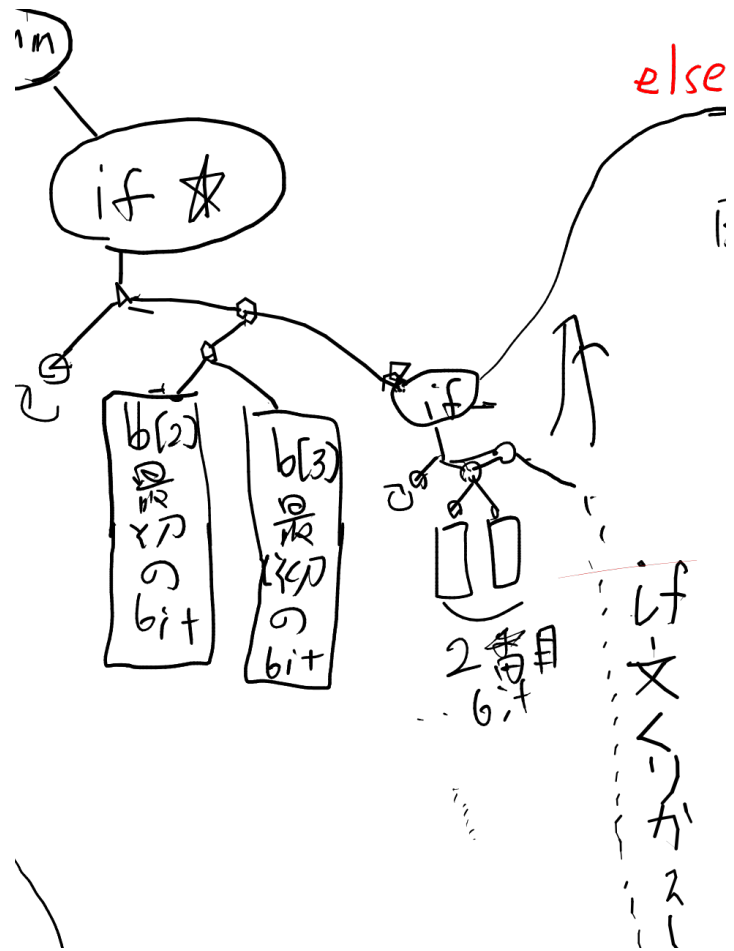
- 定数置場

- 001 以下に変数を置く
  - 000 も使えるものの使用していない
- 1. 単方向リンクリストの一番端を表すポインタ (00101)
- 2. 単方向リンクリストの現在の位置を表すポインタ (00111)
- 3. ループ回数の残り回数を表す変数 (00100)
  - 最初は32 を表すノードにて初期化



# ソースコードの概要

- if 文
  - 文字列1行分に対し  $b[2] == b[3]$  かどうかの判定
  - それぞれのバイトに対して繰り返す
- 各バイトに対して似たような引数をとると無駄が多い
  - 節点数削減のために“1...1”の文字列を定数として表すノードを先に構築
  - Python ジェネレータ内 `traverse()` 関数を用い適切な場所を指定して使い回し



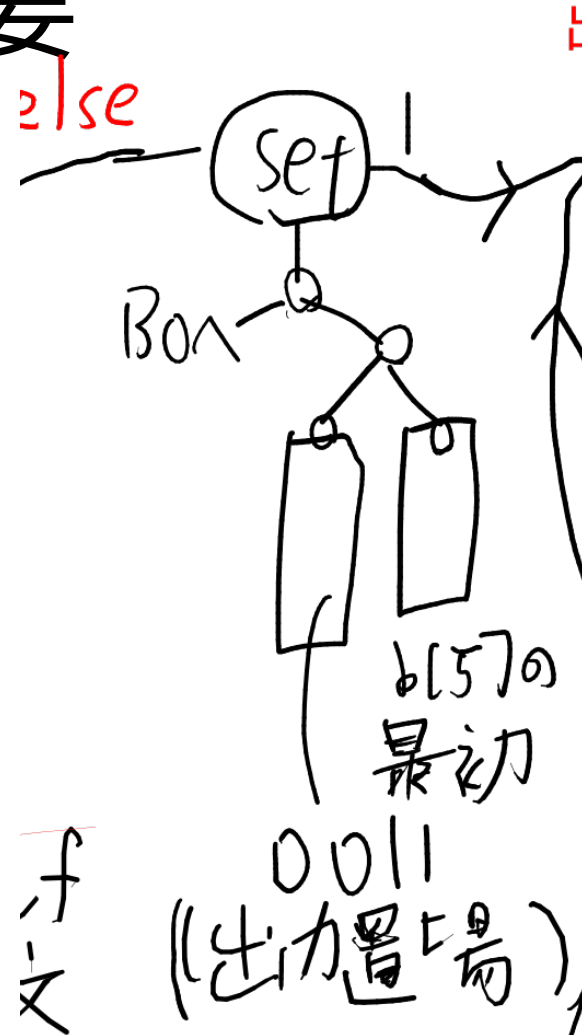
# ソースコードの概要

- 全て同一と分かったとき
- b[3] を書込む.
  - `1` は別の用途にて使われるので配列を指すポインタとして `00111` を使う
  - `00111` に `b[3]` の内容へのポインタを書込む



# ソースコードの概要

- 全て同一とは限らない場合
- b[5] を書込む.
  - `1` は別の用途にて使われる配列を指すポインタとして `00111` を使う
  - `00111` に `b[5]` の内容へのポインタを書込む

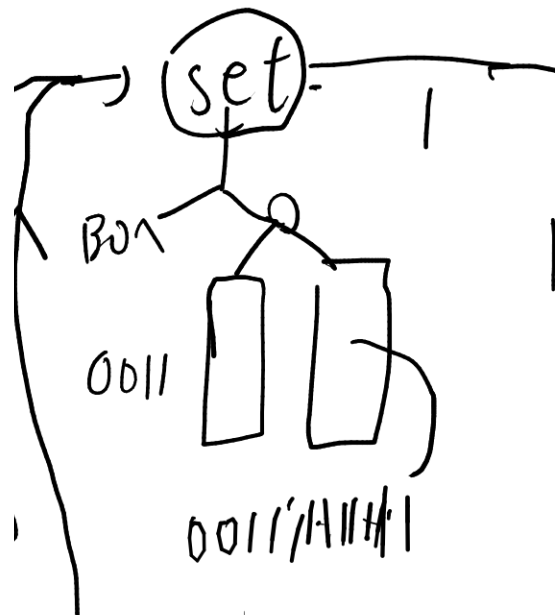


# ソースコードの概要

- 単方向リンクリストのポインタのつけかえ

出力先ポインタ更新

- ループ後に次の処理を行うために、次回ループ時に書込むべき節点の位置をset で変更
- 1文字分ずらすので8bit 分の“1”を加える
- “0011”を“00111”に書き換えてしてしまうとそのポインタを指すアドレスそのものが変わる
  - “00111”に“00111111111111”を指定



# ソースコードの概要

- 出力対象文字の後に空白文字を挿入
  - 上では b[3], b[5] 以降のデータを複製している
  - 6文字目以降のデータもコピー
  - 出力には含まれないので空白文字で置き換え
  - 00111 に 00010000 を埋めている



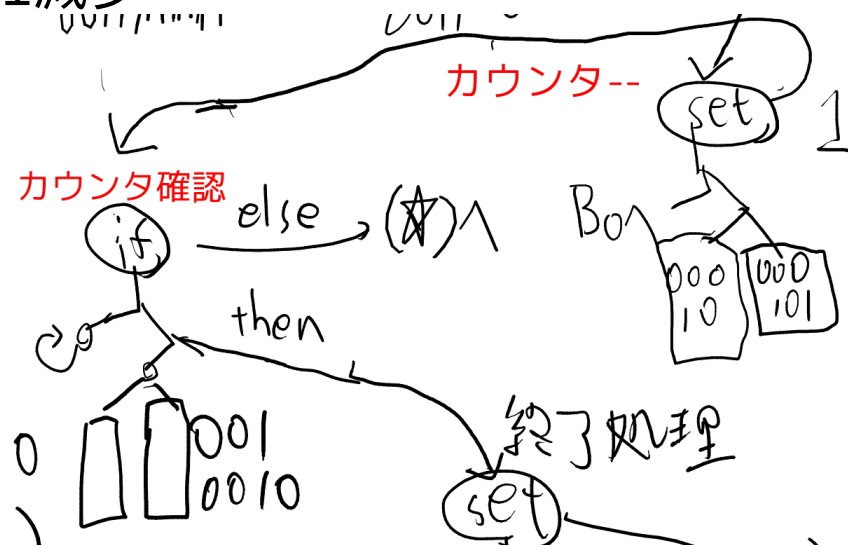
# ソースコードの概要

- 行数カウンタの処理 (for文のi--;)

- 32回分処理を行うのでループごとに1減少
- 0010011 を 001001 にAssign

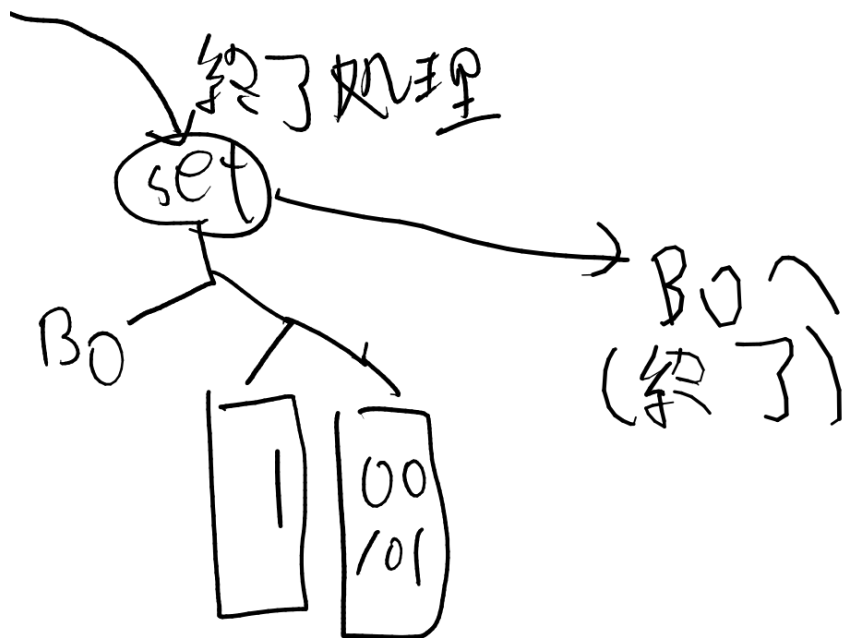
- 終了判定

- 0010010 がB0 をとれば終了.
  - 終了条件を満たす場合
    - 終了処理へ.
  - 終了条件を満たさない場合
    - 最初の (b[2] == b[3] かを判定する) if文へ



# ソースコードの概要

- 終了処理 ( `1` の意味を変更したため )
  - 出力時には `00101` (単方向リンクリストの最初を指すポインタ) のノードを1に Assign
  - 先頭行の結果, 第2行の結果,..., という順番に結果は格納されている
- この処理後には次の状態をB0 とする.
  - これにより Main Loop から抜ける.
- インタプリタ側で 1 以下の serialize 処理が行なわれて終了.



出力置場を本来の出力置場に複製



# Q&A

- なぜ `1` ではなく `00111` と別の変数を使っているのか？
  - ループするたびに `1` の指す対象が変更されると以前のループにおいて (行にて) 使っていた Assign, If コマンド群の引数が使えなくなる
  - その分それぞれのノードで `11111111` をつけたすことも考えたが手間となるため断念
  - `1` を現在処理している対象(とそれ以降)の行として処理

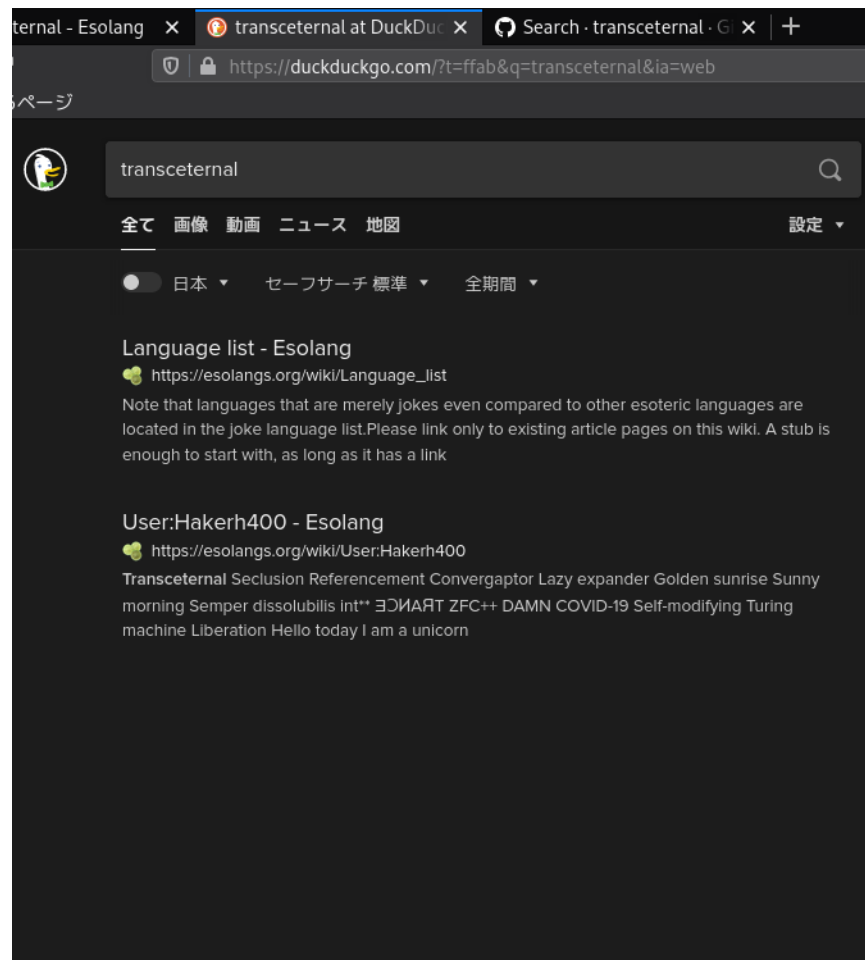
# 苦労した点

- デバッグが困難
  - 最後まで終了しないと一切文字が表示されないので 途中過程を調べられない
  - 無限ループに入る地点でどの段階まで処理が進んだか分からない
    - ちなみに今回は loop で戻る際に admin ではなく root を指定していたのが原因

# 苦労した点

- 資料不足

- 検索しても Esolang Wiki と実装しか見あたらない
- 作者以外による解説が見あたらない
  - おそらくこの資料が最初



# ソースコードの再確認

- 以上を踏まえるとソースコードを前から順に読めばある程度構造が読み解けるかもしれない
  - 最初のほうはジェネレーターのソースコードも適宜参照しました...
- 有用そうな知識:
  - 最初の文字はrootノード
  - 「2」がB0に対応
  - 「3」がB1に対応
  - 定数の終端はB0か自身に戻っていく
- 直接読みとけるとソースコードの分量を減らす方法を考えるのに役立つかも

# ソースコード (1560バイト)

```
0 1 2 2 2 3 4 5 3 6 3 7 3 8 3 9 3 A 3 B 3 C 3 D 3 E 3 F 3 G 3 H 3 I 3 J 3 K 3 L 3 M 3 N 3 O 3 P 3 Q
3 R 3 S 3 T 3 U 3 V 3 W 3 X 3 Y 3 Z 3 a 3 b 3 c 2 c d 3 2 d 2d 2e 2f 2f 2f 2g 2h 1p 3 1q 3 1r 3 1s 3
1t 3 1u 3 1v 3 1w 3 1x 3 1y 2 1y 1h 3 1i 3 1j 3 1k 3 1l 3 1m 3 1n 3 1o 3 1p 2Y 2Z 2a 2a 2a 2b 2c
1o 1g 3 1h 2T 2U 2V 2V 2V 2W 2X 1n 1f 3 1g 2O 2P 2Q 2Q 2Q 2R 2S 1m 1e 3 1f 2J 2K 2L 2L 2L
2M 2N 1l 1d 3 1e 2E 2F 2G 2G 2G 2H 2l 1k 1c 3 1d 29 2A 2B 2B 2B 2C 2D 1j 1b 3 1c 24 25 26 26
26 27 28 1c 1k 1z 20 21 21 21 22 23 1a 3 1b 1i e u 2 t f 2 g 2 h 3 i 3 j 3 j k 3 l 3 m 3 n 3 o 3 p 3 q 3
r 3 s 3 s 2i 3o 2 3n 2j 3 2j 2k 3 2l 3 2m 3 2n 3 2o 3 2p 3 2q 3 2r 3 2s 3 2t 3 2u 3 2v 3 2w 3 2x 3 2y
3 2z 3 30 3 31 3 32 3 33 3 34 3 35 3 36 3 37 3 38 3 39 3 3A 3 3B 3 3C 3 3D 3 3E 3 3F 3 3G 3 3H
3 3I 3 3J 3 3K 3 3L 3 3M 3 3N 3 3O 3 3P 3 3Q 3 3R 3 3S 3 3T 3 3U 3 3V 3 3W 3 3X 3 3Y 3 3Z 3
3a 3 3b 3 3c 3 3d 3 3e 3 3f 3 3g 3 3h 3 3i 3 3j 3 3k 3 3l 3 3m 3 3m 3p 47 2 46 3q 2 3r 2 3s 3 3t 3
3t 3u 2 3v 2 3w 3 3x 3 3y 3 3z 3 40 3 41 3 42 3 43 3 44 3 45 3 45 48 4N 2 4M 49 2 4A 2 4B 3 4C 3
4D 3 4D 4E 2 4F 2 4G 2 4H 3 4I 2 4J 2 4K 2 4L 2 4L 4O 4d 2 4c 4P 2 4Q 2 4R 3 4S 2 4T 2 4U 3
4U 4V 2 4W 2 4X 3 4Y 2 4Z 2 4a 3 4b 3 4b 4y 4z 50 50 50 51 52 4o 2 4p 2 4q 2 4q 4r 2 4s 2 4t 3
4u 2 4v 2 4w 3 4x 2 4x 4e 4n 2 4m 4f 3 4f 4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l 2 2d v 1Z 2 1Y w 2 x 2 y 3
z 3 10 3 10 11 3 12 3 13 3 14 3 15 3 16 3 17 3 18 3 19 3 1A 3 1B 3 1C 3 1D 3 1E 3 1F 3 1G 3 1H
3 1I 3 1J 3 1K 3 1L 3 1M 3 1N 3 1O 3 1P 3 1Q 3 1R 3 1S 3 1T 3 1U 3 1V 3 1W 3 1X 3 1X 2i v v v v
v v v v
```

# ソースコードを読み解く

0 1 2 2 2 3 4

- 0,1,2 と admin や B0 を生成している.
- 0 は 1 に繋がり, 1 の0-pointer は2.
- 2 の0-pointer を見ると (Stack の処理方法を復習すると、未知ノードAを発見して、そのとき0-pointer が既知ノードならその次の文字はAの 1-pointer)
- なので 1 2 2 3 4 は 1 の0-ptr が2, 2 の0-ptr が2, 1-ptr が2, 1 の1-ptr が3 と読める. 3は新出ノード.
- 3 の0-ptr は新出ノードの4. なのでここから先は新出ノードの4に処理が移る.
- 5 3 6 3 7 3 8 3 9 3 A 3 B 3 C 3 D 3 E 3 F 3 G 3 H 3 I 3 J 3 K 3 L 3 M 3 N 3 O 3 P 3 Q 3 R 3 S 3 T 3 U 3 V 3 W 3 X 3 Y 3 Z 3 a 3 b 3 c 2 c d 3 2 d 2d 2e 2f 2f 2f 2g 2h 2i 2j 2k 2l 2m 2n 2o 2p 2q 2r 2s 2t 2u 2v 2w 2x 2y 2z 2aa 2ab 2ac 2ad 2ae 2af 2ag 2ah 2ai 2aj 2ak 2al 2am 2an 2ao 2ap 2aq 2ar 2as 2at 2au 2av 2aw 2ax 2ay 2az 2ba 2bb 2bc 2bd 2be 2bf 2bg 2bh 2bi 2bj 2bk 2bl 2bm 2bn 2bo 2bp 2bq 2br 2bs 2bt 2bu 2bv 2bw 2bx 2by 2bz 2ca 2cb 2cc 2cd 2ce 2cf 2cg 2ch 2ci 2cj 2ck 2cl 2cm 2cn 2co 2cp 2cq 2cr 2cs 2ct 2cu 2cv 2cw 2cx 2cy 2cz 2da 2db 2dc 2dd 2de 2df 2dg 2dh 2di 2dj 2dk 2dl 2dm 2dn 2do 2dp 2dq 2dr 2ds 2dt 2du 2dv 2dw 2dx 2dy 2dz 2ea 2eb 2ec 2ed 2ee 2ef 2eg 2eh 2ei 2ej 2ek 2el 2em 2en 2eo 2ep 2eq 2er 2es 2et 2eu 2ev 2ew 2ex 2ey 2ez 2fa 2fb 2fc 2fd 2fe 2ff 2fg 2fh 2fi 2fj 2fk 2fl 2fm 2fn 2fo 2fp 2fq 2fr 2fs 2ft 2fu 2fv 2fw 2fx 2fy 2fz 2ga 2gb 2gc 2gd 2ge 2gf 2gg 2gh 2gi 2gj 2gk 2gl 2gm 2gn 2go 2gp 2gq 2gr 2gs 2gt 2gu 2gv 2gw 2gx 2gy 2gz 2ha 2hb 2hc 2hd 2he 2hf 2hg 2hh 2hi 2hj 2hk 2hl 2hm 2hn 2ho 2hp 2hq 2hr 2hs 2ht 2hu 2hv 2hw 2hx 2hy 2hz 2ia 2ib 2ic 2id 2ie 2if 2ig 2ih 2ii 2ij 2ik 2il 2im 2in 2io 2ip 2iq 2ir 2is 2it 2iu 2iv 2iw 2ix 2iy 2iz 2ja 2jb 2jc 2jd 2je 2jf 2jg 2jh 2ji 2jj 2jk 2jl 2jm 2jn 2jo 2jp 2jq 2jr 2js 2jt 2ju 2jv 2jw 2jx 2jy 2jz 2ka 2kb 2kc 2kd 2ke 2kf 2kg 2kh 2ki 2kj 2kk 2kl 2km 2kn 2ko 2kp 2kq 2kr 2ks 2kt 2ku 2kv 2kw 2kx 2ky 2kz 2la 2lb 2lc 2ld 2le 2lf 2lg 2lh 2li 2lj 2lk 2ll 2lm 2ln 2lo 2lp 2lq 2lr 2ls 2lt 2lu 2lv 2lw 2lx 2ly 2lz 2ma 2mb 2mc 2md 2me 2mf 2mg 2mh 2mi 2mj 2mk 2ml 2mm 2mn 2mo 2mp 2mq 2mr 2ms 2mt 2mu 2mv 2mw 2mx 2my 2mz 2na 2nb 2nc 2nd 2ne 2nf 2ng 2nh 2ni 2nj 2nk 2nl 2nm 2nn 2no 2np 2nq 2nr 2ns 2nt 2nu 2nv 2nw 2nx 2ny 2nz 2oa 2ob 2oc 2od 2oe 2of 2og 2oh 2oi 2oj 2ok 2ol 2om 2on 2oo 2op 2oq 2or 2os 2ot 2ou 2ov 2ow 2ox 2oy 2oz 2pa 2pb 2pc 2pd 2pe 2pf 2pg 2ph 2pi 2pj 2pk 2pl 2pm 2pn 2po 2pp 2pq 2pr 2ps 2pt 2pu 2pv 2pw 2px 2py 2pz 2qa 2qb 2qc 2qd 2qe 2qf 2qg 2qh 2qi 2qj 2qk 2ql 2qm 2qn 2qo 2qp 2qq 2qr 2qs 2qt 2qu 2qv 2qw 2qx 2qy 2qz 2ra 2rb 2rc 2rd 2re 2rf 2rg 2rh 2ri 2rj 2rk 2rl 2rm 2rn 2ro 2rp 2rq 2rr 2rs 2rt 2ru 2rv 2rw 2rx 2ry 2rz 2sa 2sb 2sc 2sd 2se 2sf 2sg 2sh 2si 2sj 2sk 2sl 2sm 2sn 2so 2sp 2sq 2sr 2ss 2st 2su 2sv 2sw 2sx 2sy 2sz 2ta 2tb 2tc 2td 2te 2tf 2tg 2th 2ti 2tj 2tk 2tl 2tm 2tn 2to 2tp 2tq 2tr 2ts 2tt 2tu 2tv 2tw 2tx 2ty 2tz 2ua 2ub 2uc 2ud 2ue 2uf 2ug 2uh 2ui 2uj 2uk 2ul 2um 2un 2uo 2up 2uq 2ur 2us 2ut 2uu 2uv 2uw 2ux 2uy 2uz 2va 2vb 2vc 2vd 2ve 2vf 2vg 2vh 2vi 2vj 2vk 2vl 2vm 2vn 2vo 2vp 2vq 2vr 2vs 2vt 2vu 2vv 2vw 2vx 2vy 2vz 2wa 2wb 2wc 2wd 2we 2wf 2wg 2wh 2wi 2wj 2wk 2wl 2wm 2wn 2wo 2wp 2wq 2wr 2ws 2wt 2wu 2wv 2ww 2wx 2wy 2wz 2xa 2xb 2xc 2xd 2xe 2xf 2xg 2xh 2xi 2xj 2xk 2xl 2xm 2xn 2xo 2xp 2xq 2xr 2xs 2xt 2xu 2xv 2xw 2xx 2xy 2xz 2ya 2yb 2yc 2yd 2ye 2yf 2yg 2yh 2yi 2yj 2yk 2yl 2ym 2yn 2yo 2yp 2yq 2yr 2ys 2yt 2yu 2yv 2yw 2yx 2yy 2yz 2za 2zb 2zc 2zd 2ze 2zf 2zg 2zh 2zi 2zj 2zk 2zl 2zm 2zn 2zo 2zp 2zq 2zr 2zs 2zt 2zu 2zv 2zw 2zx 2zy 2zz

0 1 2 2 2 3 4

5 3 6 3 7 3 8 3 9 3 A 3 B 3 C 3 D 3 E 3 F 3 G 3 H 3 I 3 J 3 K 3 L 3 M 3 N 3 O 3 P  
3 Q 3 R 3 S 3 T 3 U 3 V 3 W 3 X 3 Y 3 Z 3 a 3 b 3 c 2 c

- ノード4 (変数置場)から探索する. すると0-pointer が5 で新出ノード.
- 次の「3」 (B1)は5 の0-pointer. 「6」 は5 の1-pointer. 6 は新出ノード.
- 次の「3」 (B1)は6 の0-pointer. 「7」 は6 の1-pinter. 7 は新出ノード.
- これをくりかえし、「111111.....1」という列が33個続いていることが分かる.  
(インタプリタ内では count\_graph としている.) カウント回数を表す.
- 最後はc 2 c となっており、新出ノードc は 0-pointer が2 (B0), 1-pointer が自分自身とわかる.
- d 3 2 d 2d 2e 2f 2f 2f 2g 2h 1p 3 1q 3 1r 3 1s 3 1t 3 1u 3 1v 3 1w 3 1x 3 1y 2 1y  
1h 3 1i 3 1j 3 1k 3 1l 3 1m 3 1n 3 1o 3 1p 2Y 2Z 2a 2a 2a 2b 2c 1o 1g 3 1h 2T  
2U 2V 2V 2V 2W 2X 1n 1f 3 1g 2O 2P 2Q 2Q 2Q 2R 2S 1m 1e 3 1f 2J 2K 2L 2L  
2L 2M 2N 1l 1d 3 1e 2E 2F 2G 2G 2G 2H 2l 1k 1c 3 1d 29 2A 2B 2B 2B 2C 2D 1j

5 3 6 3 7 3 8 3 9 3 A 3 B 3 C 3 D 3 E 3 F 3 G 3 H 3 I 3 J 3 K 3 L 3 M 3 N 3 O 3  
P 3 Q 3 R 3 S 3 T 3 U 3 V 3 W 3 X 3 Y 3 Z 3 a 3 b 3 c 2 c

- d 3 2 d
- このd は
  - ノード4 (変数置場)から探索する. すると0-pointer が5 で新出ノード.
- ここで取り残したノード4の1-pointer なので変数置場の1-pointer がd と分かる.
- 新出ノードなので何に繋がっているか見るとそれぞれ B1, B0 となる.
- その後のd は
  - 3 の0-ptr は新出ノードの4. なのでここから先は新出ノードの4に処理が移る.
- ここで取り残したので 3 (B1) の1-pointer は d. (これは単方向リンクリストの本体およびポインタです)
- 2d 2e 2f 2f 2f 2g 2h 1p 3 1q 3 1r 3 1s 3 1t 3 1u 3 1v 3 1w 3 1x 3 1y 2 1y 1h 3 1i



- d 3 2 d
- 2d 本体処理が現れる
  - 2E
    - 2f 2f 2f : 巡回ノードなので Condition と分かる
    - 2g : その引数は？という視点で見ると
      - 2h (左辺と右辺の親)
        - 1p 3 1q 3 1r 3 1s 3 1t 3 1u 3 1v 3 1w 3 1x 3 1y 2 1y
          - 9文字の1 + 0 = 2文字目の最初
        - 1h 3 1i 3 1j 3 1k 3 1l 3 1m 3 1n 3 1o 3 1p
          - 8文字つくって上の 1p に連結 = 8文字の1 + 9文字の1 = 3文字目の最初
        - こうやってメモリ節約していると分かる
      - 2Y (then)
      - then 以下次のif 文が続く) (インデント戻します)
        - 2Z 2a 2a 2a 2a 2b 2c 1a 1a 3 1b 2T 2U 2V 2V 2V 2W 2X 1n 1f 3 1a

1n 3 1g 3 1p

- 2Y
  - 2Z
    - 2a 2a 2a : Condition
    - 2b
      - 2c
        - 1o
          - 8文字つくって上の 1p に連結の1つ手前に繋いでいる -> 2文字目の2bit
        - 1g 3 1h
          - 1g は新出, 1つ3を継ぎ足して 1h に繋ぐ
          - 1h は8文字つくって上の1p に連結 (先のif文の右辺) に更に1 を1つ増やしている -> 3文字目の2bit目
  - then: 次の文字 (2T)- (インデント戻します: if文3つ目)
  - 2T 2U 2V 2V 2V 2W 2X 1n 1f 3 1g
- 2Q 2R 2Q 2Q 2Q 2R 2S 1m 1a 3 1f 3 1K 2L 2L 2L 2L 2M 2N 1l 1d 3 1a 2E 2E 2C 2C

- 2T 2U 2V 2V 2V 2W 2X 1n 1f 3 1g
  - then は次の2O
- (インデント戻します: if文4つ目)
- 2O 2P 2Q 2Q 2Q 2R 2S 1m 1e 3 1f
  - then は次の2J
- 2J 2K 2L 2L 2L 2M 2N 1l 1d 3 1e
- if文6
- 2E 2F 2G 2G 2G 2H 2I 1k 1c 3 1d
- if文7
- 29 2A 2B 2B 2B 2C 2D 1j 1b 3 1c

- 29 2A 2B 2B 2C 2D 1j 1b 3 1c
- if文8
  - 書いてて気づきましたが一つ同一内容(2バイト目)のDebug用if文を消すのを忘れていたようです....
- 24 25 26 26 27 28 1c 1k
- 1z (if文9)
  - 20 21 21 21
  - 22 23 1a 3 1b 1l
- e おまたせしました、ようやく then (2文字目と3文字目が一致するときの処理です)に入ります
  - u
    - 2(B0へ向かうのでAssign と分かる)
  - t f 2 g 2 h 3 i 3 j 3 k l 3 m 3 n 3 o 3 p 3 q 3 r 3 s 3 s 2i 3o 2 3n 2j 3 2j 2k 3 2l  
3 2m 3 2n 3 2o 3 2p 3 2q 3 2r 3 2s 3 2t 3 2u 3 2v 3 2w 3 2x 3 2y 3 2z 3 30 3  
21 2 22 2 22 2 24 2 25 2 26 2 27 2 28 2 29 2 31 2 32 2 33 2 34 2 35 2 36 2 37 2

- 1l (おまたせしました、ようやく then (2文字目と3文字目が一致するときの処理です) に入ります)
- e
  - u
    - 2 (B0 へ向かうので Assign と分かる)
  - t (assign の左辺と右辺は何か?)
    - f 2 g 2 h 3 i 3 j 3 j
      - 00111
      - 終端は自己ループで締めている
    - k 3 l 3 m 3 n 3 o 3 p 3 q 3 r 3 s 3 s
      - 11111111
- 2i 3o 2 3n 2j 3 2j 2k 3 2l 3 2m 3 2n 3 2o 3 2p 3 2q 3 2r 3 2s 3 2t 3

- k 3 l 3 m 3 n 3 o 3 p 3 q 3 r 3 s 3 s  
- 11111111

- 次は2文字目処理したあとの`1`を64ビット分ずらす処理です

- Stackは深さ優先探索なのでずらすほうが優先されます

- 2i 3o 2 (Assign)

- 3n

- 2j 3 2j

- 絶対アドレス`1`を表す

- 2k 3 2l 3 2m 3 2n 3 2o 3 2p 3 2q 3 2r 3 2s 3 2t 3 2u 3 2v 3 2w 3 2x 3 2y 3 2z 3 30  
3 31 3 32 3 33 3 34 3 35 3 36 3 37 3 38 3 39 3 3A 3 3B 3 3C 3 3D 3 3E 3 3F 3 3G  
3 3H 3 3I 3 3J 3 3K 3 3L 3 3M 3 3N 3 3O 3 3P 3 3Q 3 3R 3 3S 3 3T 3 3U 3 3V 3  
3W 3 3X 3 3Y 3 3Z 3 3a 3 3b 3 3c 3 3d 3 3e 3 3f 3 3g 3 3h 3 3i 3 3j 3 3k 3 3l 3 3m  
3 3m

- 絶対アドレス`11.....1` (1が65個並ぶ)を表す

- 3p 47 2 46 3q 2 3r 2 3s 3 3t 3 3t 3u 2 3v 2 3w 3 3x 3 3y 3 3z 3 40 3 41 3 42 3 43 3 44 3  
45 3 45 48 4N 2 4M 49 2 4A 2 4B 3 4C 3 4D 3 4D 4E 2 4F 2 4G 2 4H 3 4I 2 4J 2 4K 2 4L  
2 4L 4O 4d 2 4c 4P 2 4Q 2 4R 3 4S 2 4T 2 4U 3 4U 4V 2 4W 2 4X 3 4Y 2 4Z 2 4a 3 4b 3  
4b 4y 4z 50 50 50 51 52 4o 2 4p 2 4q 2 4q 4r 2 4s 2 4t 3 4u 2 4v 2 4w 3 4x 2 4x 4e 4n 2

3 3d 3 3e 3 3f 3 3g 3 3h 3 3i 3 3j 3 3k 3 3l 3 3m 3 3m

- 絶対アドレス `11.....1` (1が65個並ぶ) を表す
- 続いては出力ポインタをずらす処理です
- 3p
  - 47
    - 2 (Assign)
  - 46
    - 3q
      - 2 3r 2 3s 3 3t 3 3t
        - 3 3t 3 3t などの文字は終端だと明快に分かるかと思います (0011)
      - 3u 2 3v 2 3w 3 3x 3 3y 3 3z 3 40 3 41 3 42 3 43 3 44 3 45 3 45
        - 0011 + 1が8個
  - 48 4N 2 4M 49 2 4A 2 4B 3 4C 3 4D 3 4D 4E 2 4F 2 4G 2 4H 3 4I 2 4J 2

3u 2 3v 2 3w 3 3x 3 3y 3 3z 3 40 3 41 3 42 3 43 3 44 3 45 3 45

- 続いては出力末尾を空白文字で埋める処理です
  - 出力先へのポインタがずれたので 00111 を指定すればOK
- 48
  - 4N
    - 2 (Assign)
  - 4M
    - 49 2 4A 2 4B 3 4C 3 4D 3 4D
      - 00111
    - 4E 2 4F 2 4G 2 4H 3 4I 2 4J 2 4K 2 4L 2 4L
      - 00010000 (B0 の所は自己ループしていたから)
- 4O4d2 4c 4P 2 4O 2 4R 3 4S 2 4T 2 4U 3 4U 4V 2 4W 2 4X 3



4E 2 4F 2 4G 2 4H 3 4I 2 4J 2 4K 2 4L 2 4L

- 続いてはカウンタを1減らす処理です
- 4O 4d 2 (Assign)
  - 4c
    - 4P 2 4Q 2 4R 3 4S 2 4T 2 4U 3 4U (001001)
    - 4V 2 4W 2 4X 3 4Y 2 4Z 2 4a 3 4b 3 4b (0010011)
  - 共通化できないかなと思いつつも今のところ良い方法が思いつかずじまい
- 4y 4z 50 50 50 51 52 4o 2 4p 2 4q 2 4q 4r 2 4s 2 4t 3 4u 2 4v 2 4w 3 4x 2 4x 4e 4n 2 4m 4f 3 4f 4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l 2 2d v 1Z 2 1Y w 2 x 2 y 3 z 3 10 3 10 1

4V 2 4W 2 4X 3 4Y 2 4Z 2 4a 3 4b 3 4b (0010011)

- 終了判定です
- 4y
  - 4Z
    - 50 50 50 (自己ループ)
  - 51
    - 52 4o 2 4p 2 4q 2 4q
      - 000 かどうか.
    - 4r 2 4s 2 4t 3 4u 2 4v 2 4w 3 4x 2 4x 4e 4n 2 4m 4f 3 4f
      - 001001 が 0 かどうか (残り1でないかどうか)
  - 4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l 2 2d v 1Z 2 1Y w 2 x 2 y 3 z 3  
10 3 10 11 3 12 3 13 3 14 3 15 3 16 3 17 3 18 3 19 3 1A 3

4r 2 4s 2 4t 3 4u 2 4v 2 4w 3 4x 2 4x

- 4e (終了処理に入ります. 1 に出力を貼り付けます)

- 4n 2 (Assign)

- 4m

- 4f 3 4f (`1`)

- 4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l

- 00100 を残す

- 2 終了したときの次の処理は `2 = B0` == 終了!

- あとは残した 5文字目の場合の処理です

- 2d v 1Z 2 1Y w 2 x 2 y 3 z 3 10 3 10 11 3 12 3 13 3 14 3 15 3  
16 3 17 3 18 3 19 3 1A 3 1B 3 1C 3 1D 3 1E 3 1F 3 1G 3 1H 3  
1I 3 1J 3 1K 3 1L 3 1M 3 1N 3 1O 3 1P 3 1Q 3 1R 3 1S 3 1T 3

4g 2 4h 2 4i 3 4j 2 4k 3 4l 3 4l

- 2d (8つ目のif文で、2文字目のときに駄目だったときに入る場所)
- v 1Z 2 (Assign)
  - 1Y w 2 x 2 y 3 z 3 10 3 10
    - 00111
  - 11 3 12 3 13 3 14 3 15 3 16 3 17 3 18 3 19 3 1A 3 1B 3 1C 3 1D 3 1E 3 1F 3 1G 3 1H 3 1I 3 1J 3 1K 3 1L 3 1M 3 1N 3 1O 3 1P 3 1Q 3 1R 3 1S 3 1T 3 1U 3 1V 3 1W 3 1X 3 1X
    - ‘1’ が33個並んでいます
  - 2l 1` をずらす処理へ
- V V V V V V V V

1N 3 1O 3 1P 3 1Q 3 1R 3 1S 3 1T 3 1U 3 1V 3 1W 3 1X 3 1X

- V V V V V V V V
  - 残りのif文でelse だったときの戻る箇所 (つまり5文字目に入る場所)
- 以上、ソースコード読み解けました、おつかれさまでした!

# おしながき

- 問題の復習
- Transceternal の説明
  - 入出力と deserialize, serialize
  - グラフの各ノードの意味
  - コマンドの意味
  - 解答するための部材
- 解答の説明
- 今後 (Golf の方針)

# 今後

- Esolang Code Golf の観点から
  - 容量の大半を浪費しているのは定数値指定部分
    - 複数の節点で同じ節点を指定する
    - Allocate コマンドをくりかえし用いると減らせると期待
- 言語仕様を使いこなす観点から
  - かけ算割り算などの算術演算方法の構築
  - ランダムアクセス可能なデータ構造の構築
    - 循環バッファを用いれば可能な模様

# 謝辞

- 大会主催の @hakatashi さんと
- 助言をいただいた @drafear さん, @satos\_\_\_\_jp さんに感謝します