

# Esolang Codegolf #6

## Transceternal Writeup

@\_hiromi\_mi  
(Version 2020.5.10.2)

# 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

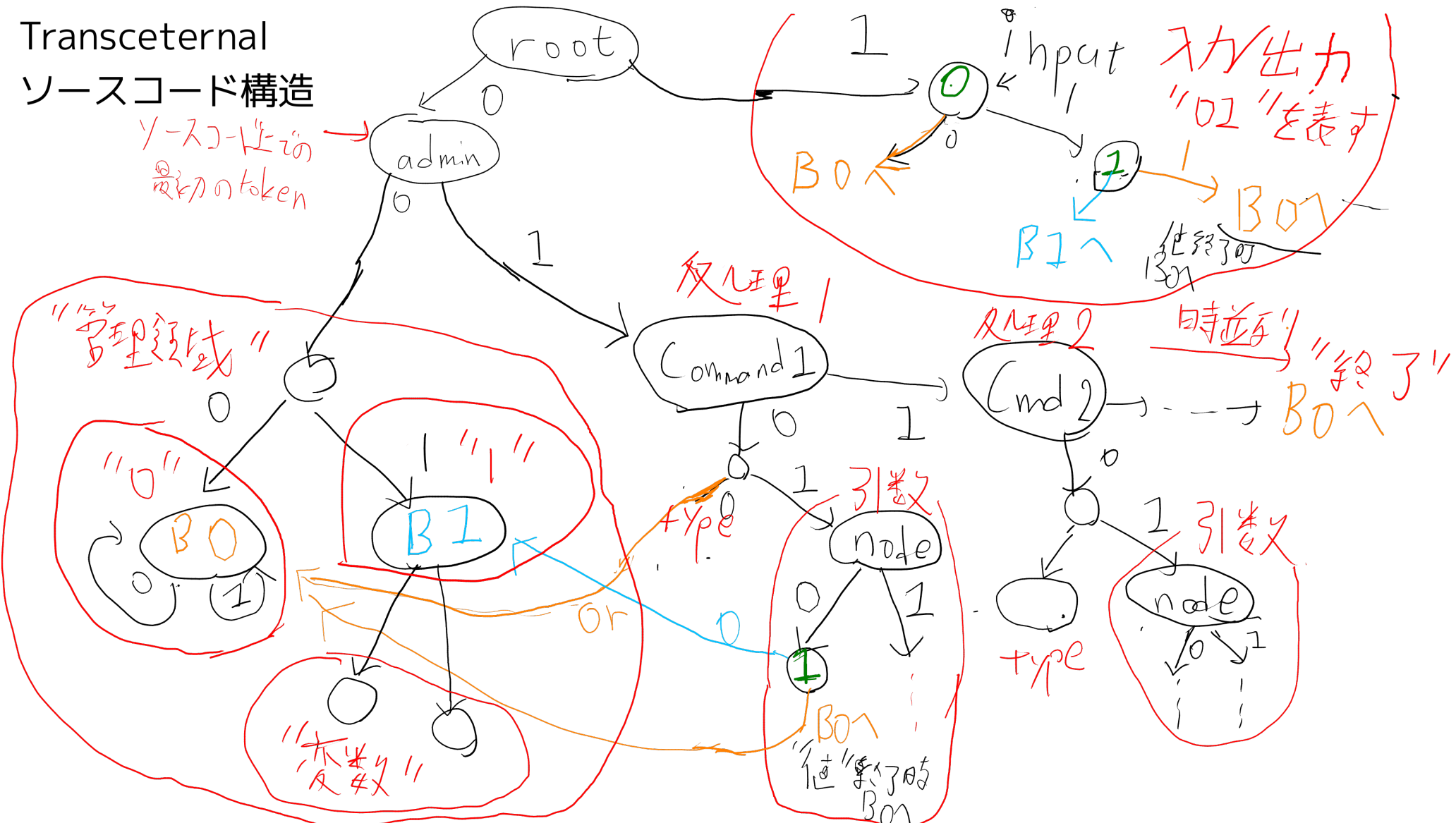
- input: 入力データをグラフ構造にしたものの最初の節点
- admin: ソースコード中での最初のノード.
  - (0-pointer, 1-pointer) = (管理領域の一番上, 次の状態)
- root: 一番上にあるノード. インタプリタが生成.
  - (0-pointer, 1-pointer) = (admin, input) となる

# 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
  - 諸々の判定基準となる
  - グラフ構造をSerialize して値を求めるときに 0 の扱い
  - 終了判定は絶対アドレス 01 が B0 と一致するか
- 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` に対応するノード

# Transceteral ソースコード構造

10. 1000 tokens



# 実行の流れ

- 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). グラフノード内を以下に述べる規則で移動
- Command が B0 になれば main loop 脱出
- `1` 以下を serialize して出力内容を生成

# 予備知識: “token”

- ソースコードをグラフの形に直す際に
- グラフのノードの“名前”
- 以下で述べられる処理は
- 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
- 各トークンごとに、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

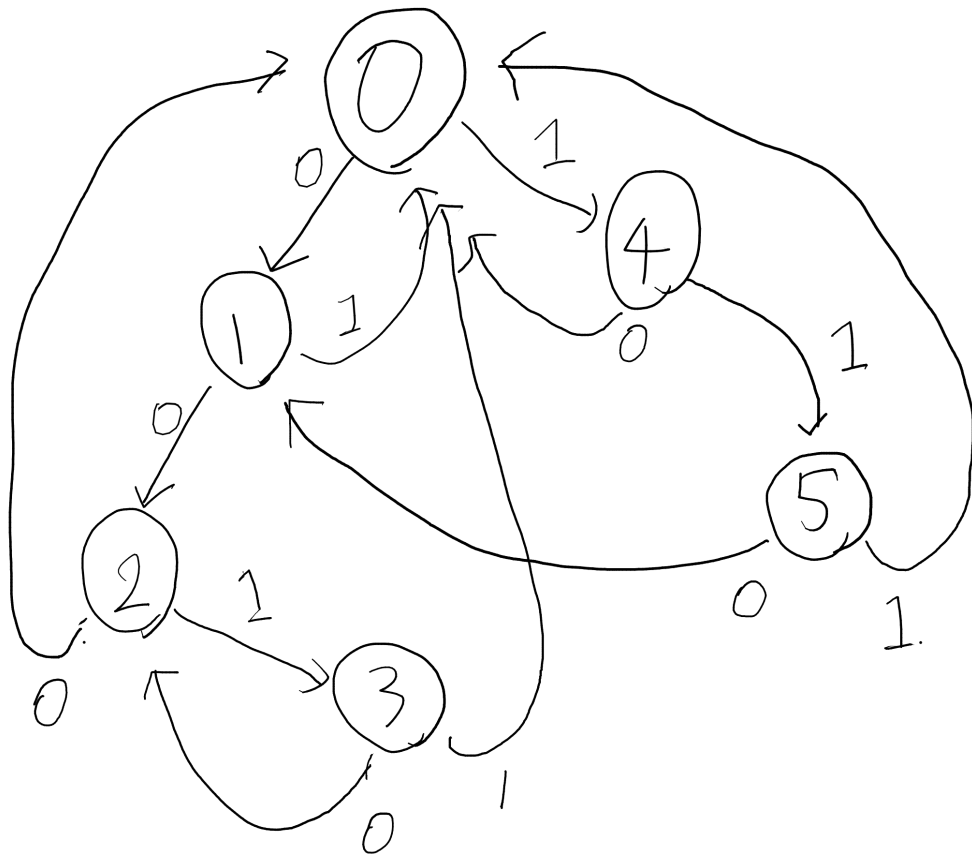


# Initial Memory Consumption 例

012032004051025

Step1 (0)

0: \_ \_



# 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

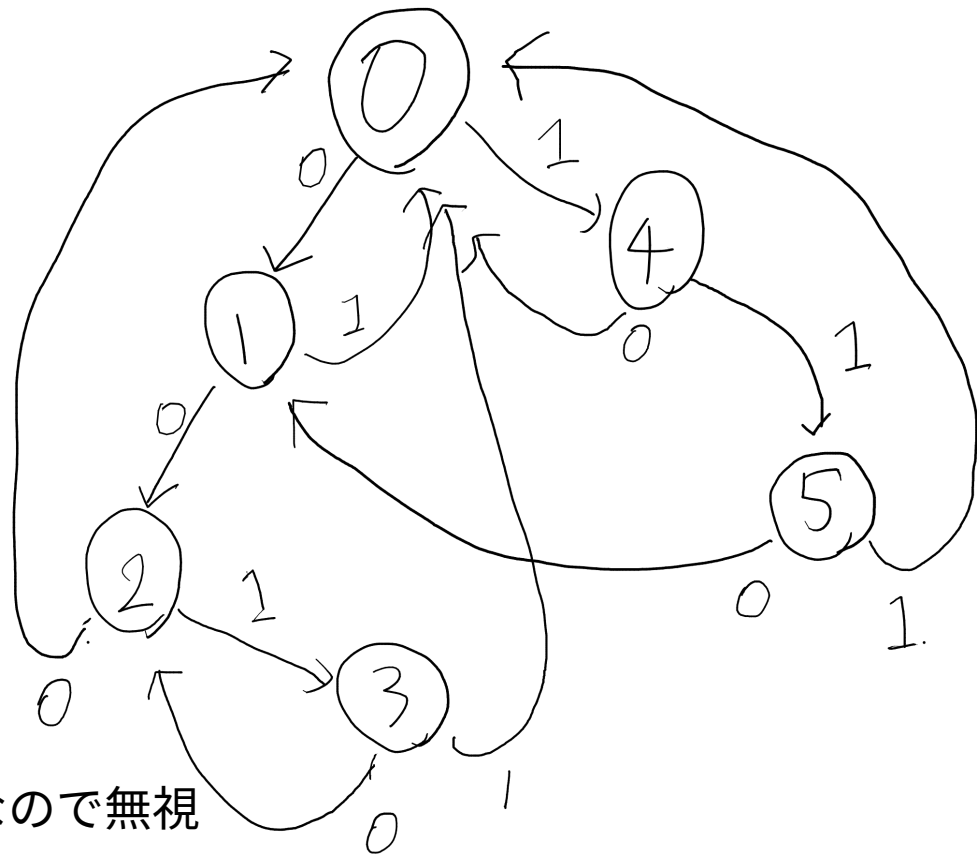
1: 2 0

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
- “parsed memory” などの言葉でも表わされる

# Deserialize

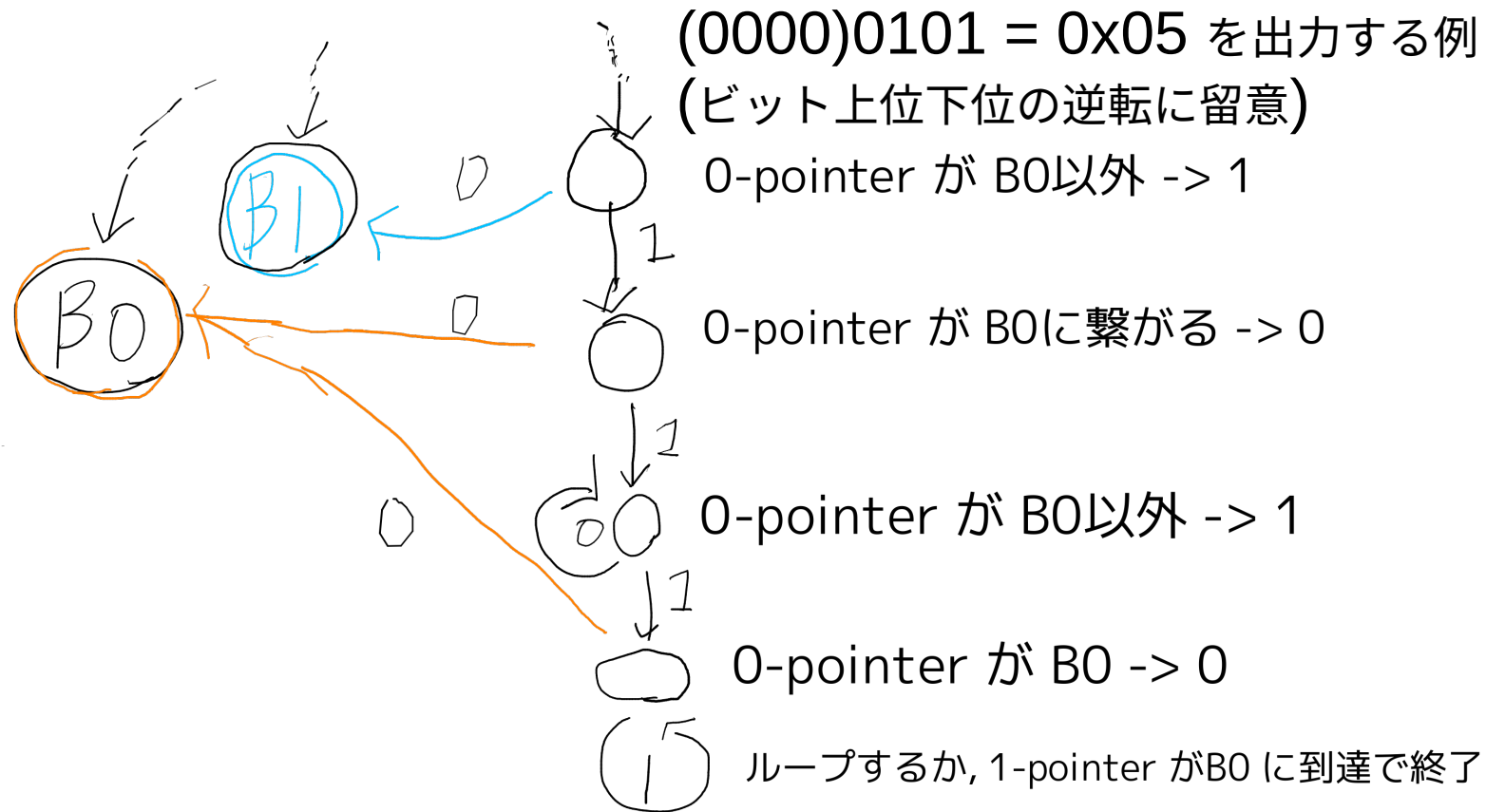
- 入力を1ビットごとに並べる
  - 下位ビットが先: 00000001 ではなく 10000000
- そして上から一つ一つを節点に変換
  - 0 なら 0-pointer を B0 に
  - 1 なら 0-pointer を B1 に
  - 1-pointer は次のビットを表わす節点
- 入力の終点は B0 に繋ぐ

# Serialize

- 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; 個人的に使っています)
- 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)

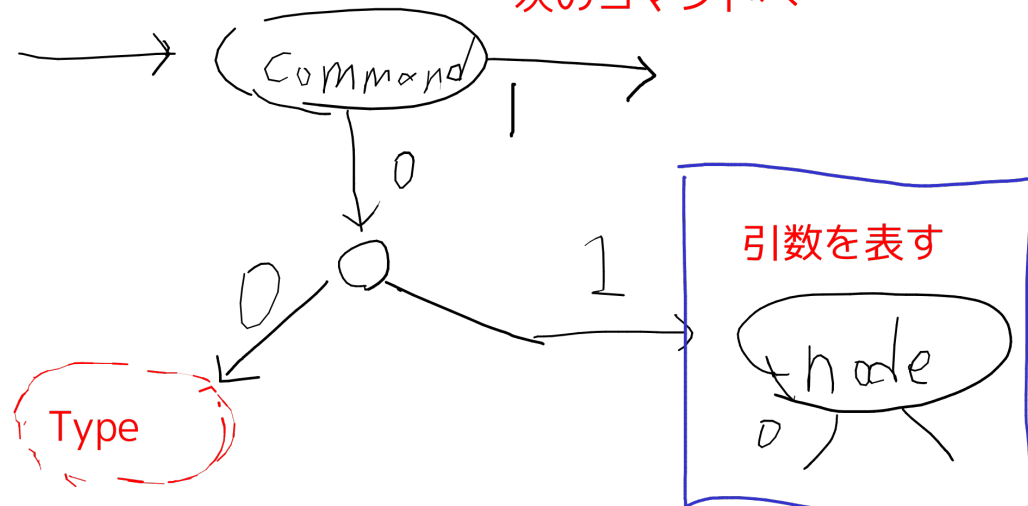
# Command の構造

- 作者は相対アドレス00 を type, 01 を node と(実装内で) 命名
- type を 相対アドレス00 に
- 引数は node (相対アドレス 01) 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)
- 引数で指定された処理の後絶対アドレス01 に絶対アドレス 011 を代入して次のIteration に

# 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



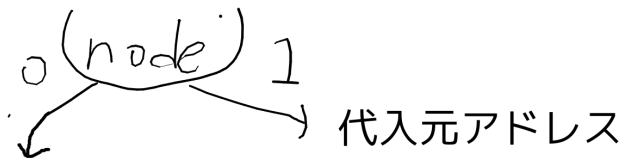
B0 : 代入

B1 : ノードの追加

それ以外 : 条件分岐

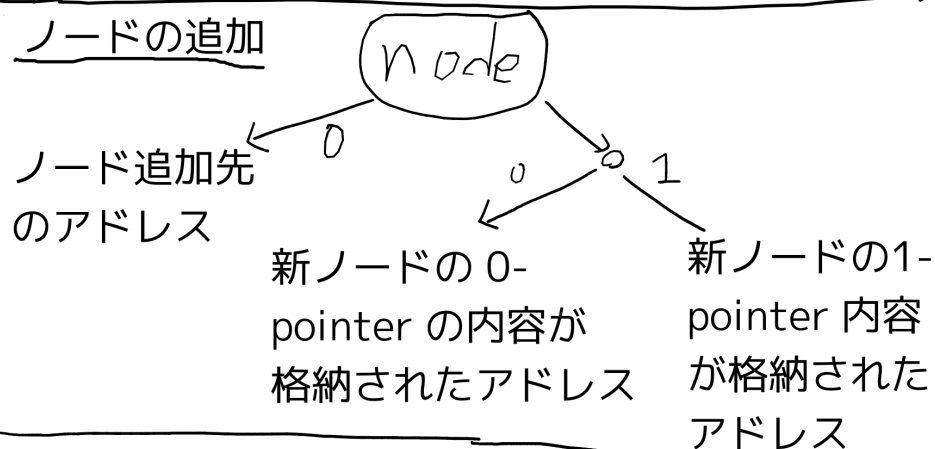
右図へ

代入

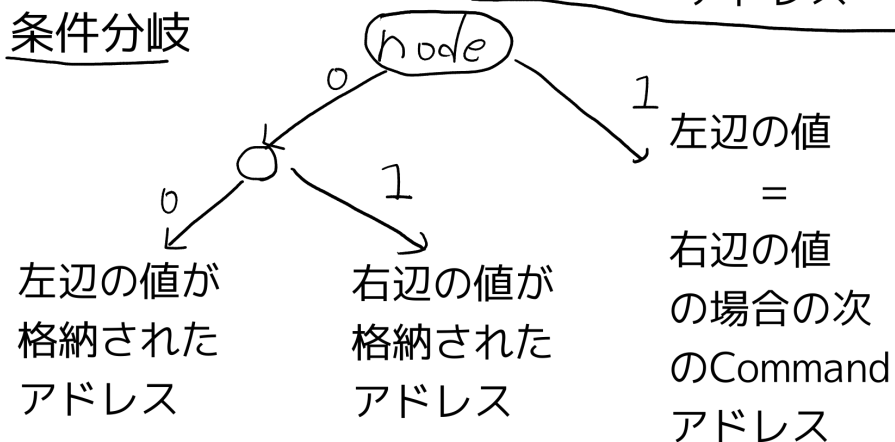


代入先のアドレス

ノードの追加



条件分岐



# Case 1: Assign

- 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: 001 (B1)
- 引数は3つ
  - 01010 の値をノード配置先のアドレス
  - 010110 の値を 0-pointer
  - 010111 の値を1-pointer
  - それぞれを0-pointer, 1-pointerい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: B0 か B1 以外
- 引数は 010100 と 010101 と 01011
- 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 : node->11 を 01 に代入
  - else : 011 を 01 に代入



# 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

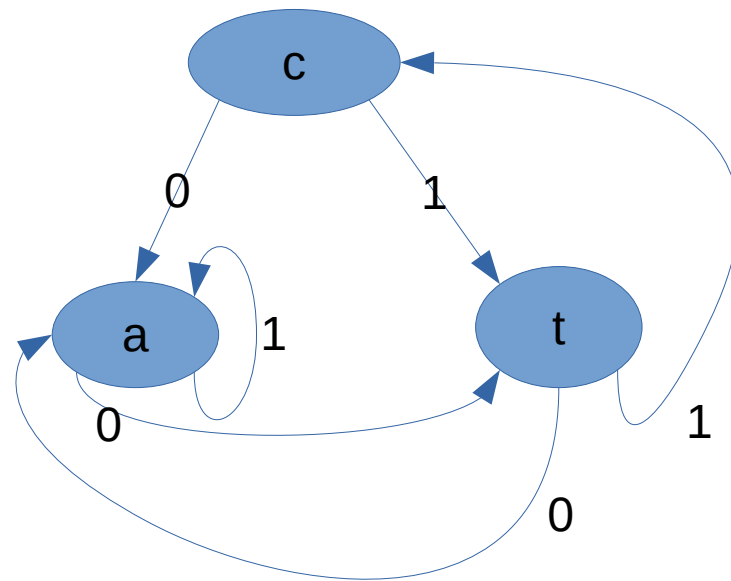
- “c” “a” “t” “a” “c” “a” “t” をもとに Initial Memory Consumption を行ないグラフ構造に

Step7: (t)

c: a t

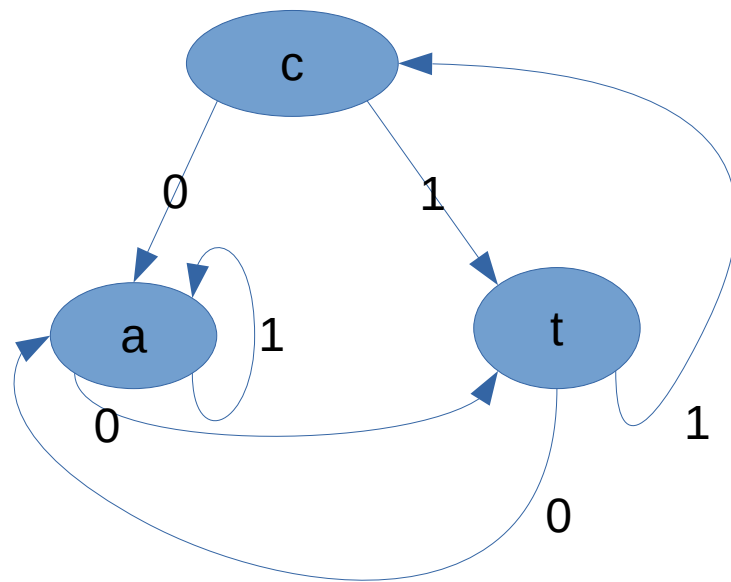
a: t a

t: a c



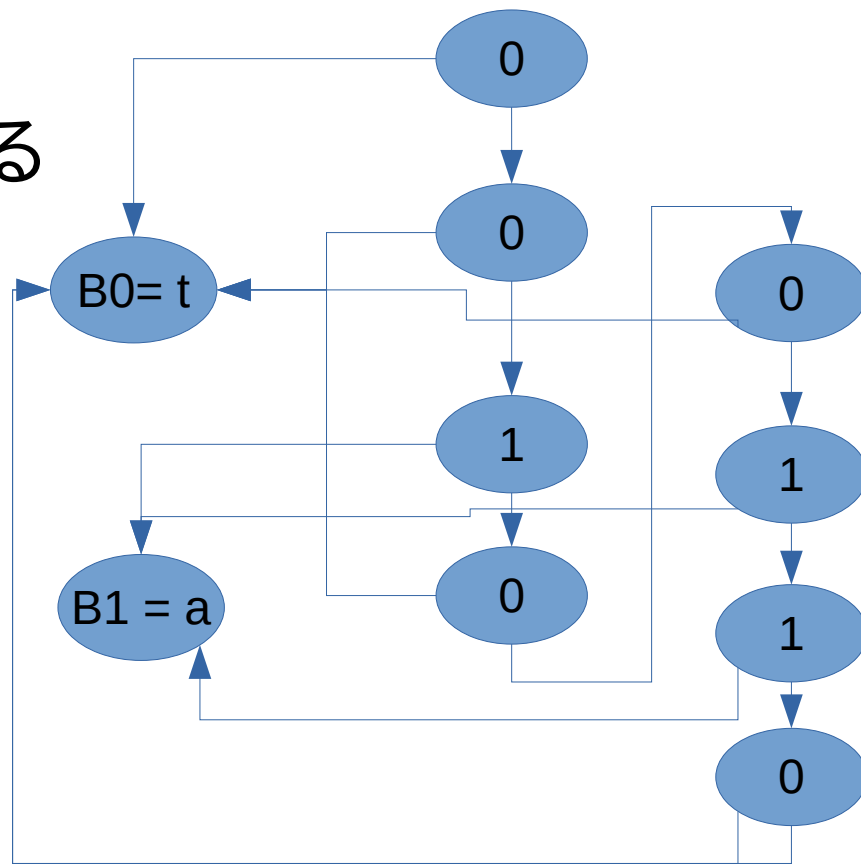
# 特別なノード: cat編

- 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



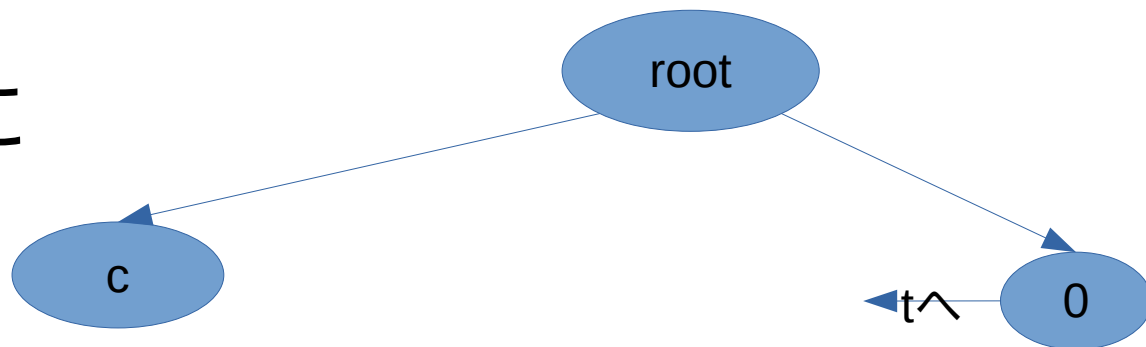
# 引数の Deserialize: cat編

- 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



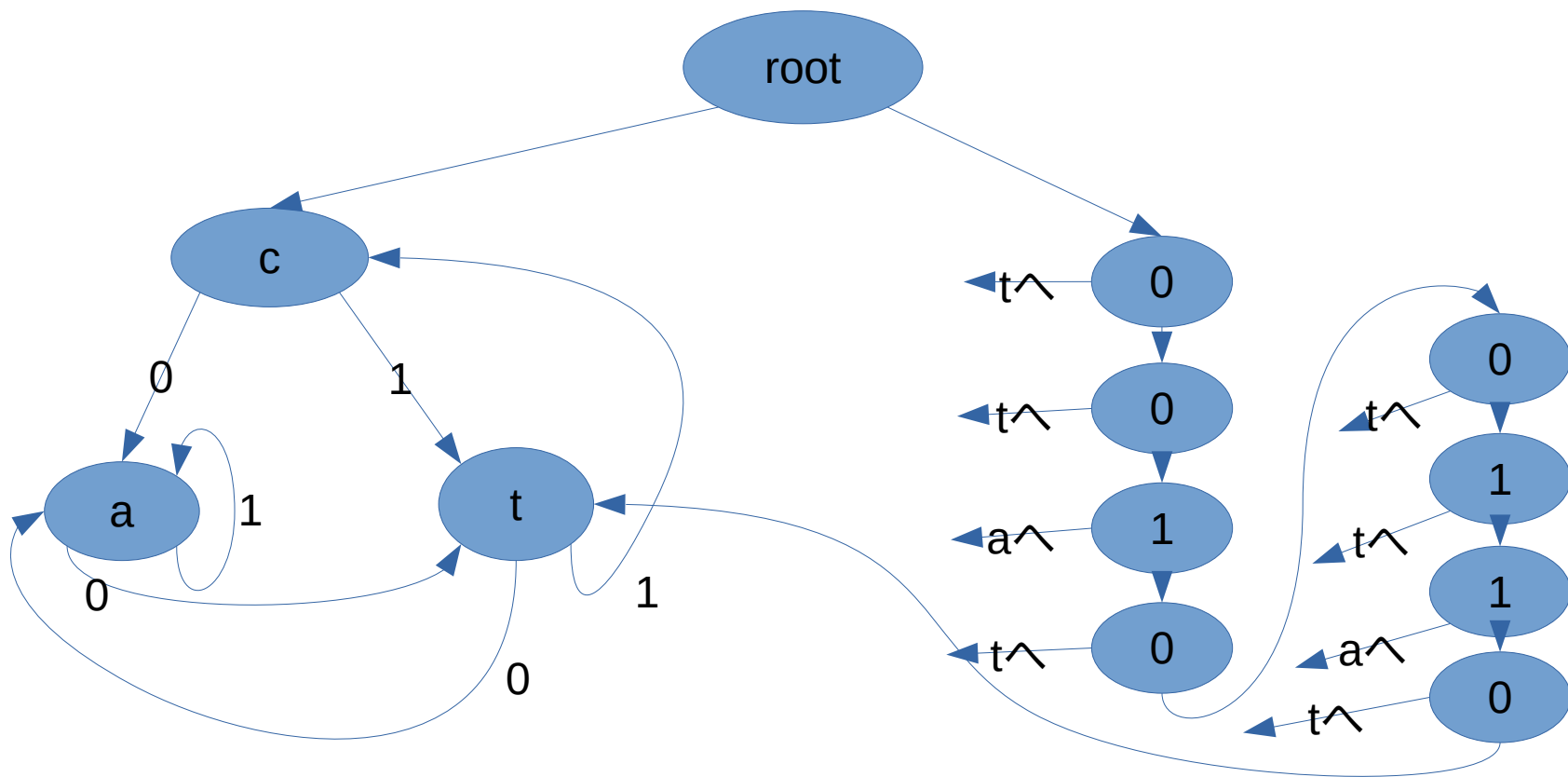
# 新しいrootノードを構築

- 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



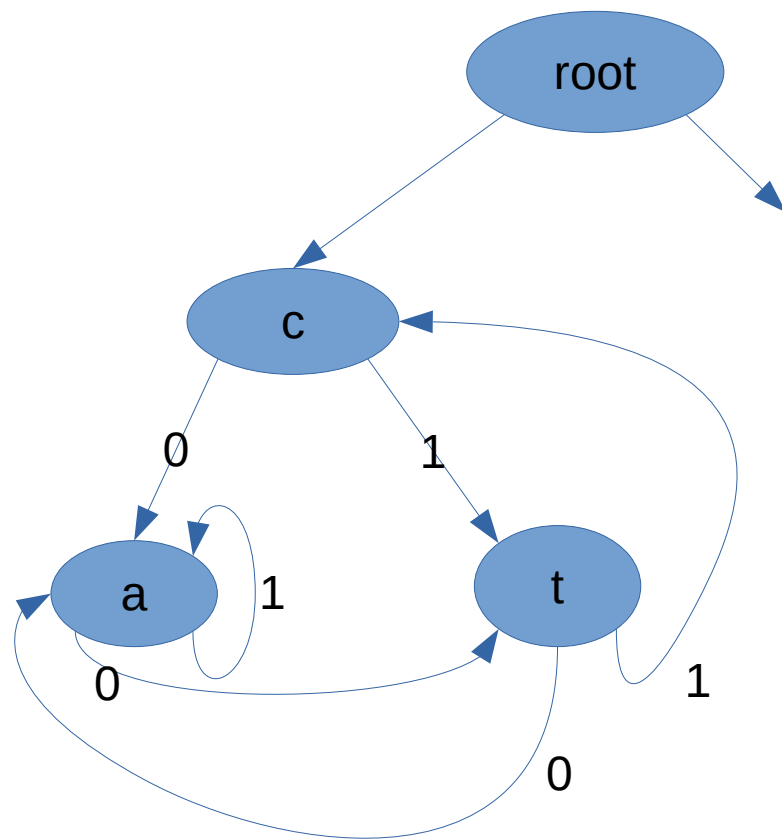


# 新しいrootノードを構築



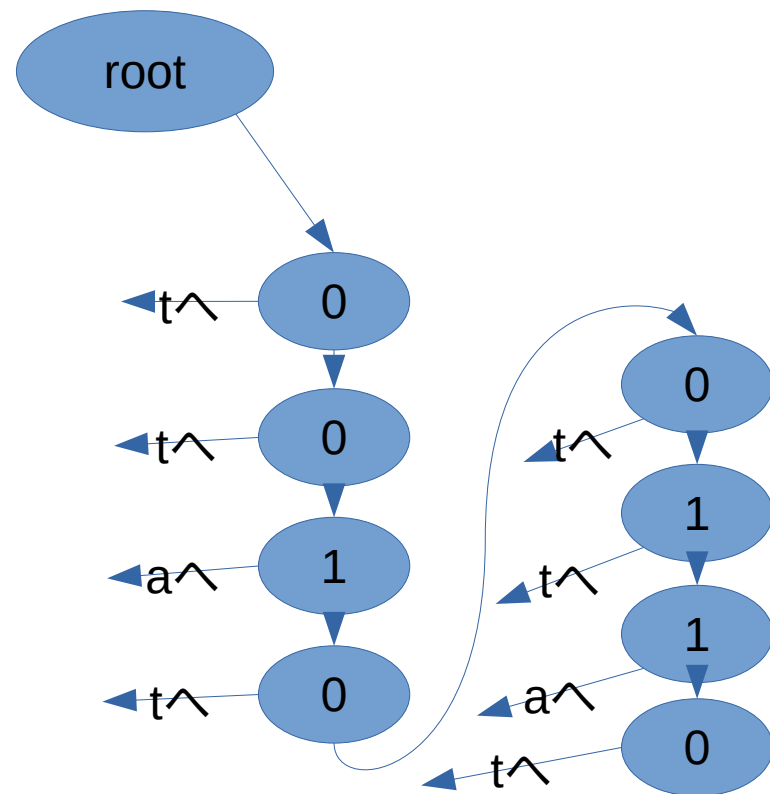
# 実行

- 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  $\rightarrow$  c  $\rightarrow$  a  $\rightarrow$  t
  - 01 は root  $\rightarrow$  c  $\rightarrow$  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

- 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足りない要素がある
  - 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

# 定数を実現する

- if 文のアドレスとして定数cを使いたい
- 引数はserialize したときのアドレスとして解釈される.
- なので特定のアドレスに「serializeしたときに定数c になるノード」を置いておく

# 変数を実現する

- 大抵の絶対アドレス内の領域は事実上使えない
  - 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
- 二分木なので  $O(\log(\#変数))$  の深さで指定できる
  - ここが長くなると変数を絶対アドレス指定が長くなる



# 変数を実現する

- 変数の値は二分木なので先に個数を決めておく必要がある
  - 本問では3個

# if文

- とあるノード = とあるノード
- それ以降の「ノード全体」を比較する
  - 入力では不適切
  - 入力そのものは B0 にならない
- なので `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

# (do-)while / for

- for文というよりカウンタ機構の実装が手間
  - カウンタ用変数を用いる.
  - 加算よりも減算のほうが Assign し直すだけで簡単に実装できるので32以下で実装
  - for文の継続条件の判定を行い継続されるなら元々のノードを繋ぐ
    - 継続されないなら出ていったノードを繋ぐ

# 一方向リンクリスト

- 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
  - “定数” に対して具体的な処理を行うのに変数を導入する必要あり

# 配列構造をつくる

- 複数bit 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
- 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

# 引き算

- (今回は定数の引き算のみ扱う)
  - 一番簡単な扱いかた: 1進法 ( $1111\dots 1$ ) をひたすら並べる、その桁数が値となる.
- そうすると定数の引き算は
  - 110 に 11001 を並べる代入にて表される
- 2進法表記での扱い方は将来的な課題

# forループ

- 変数をおきながら while 文を使い実現
- ++i; は処理内容の最後に置くことで実現



# 次の状態の指定

- “root” “admin” ではない (← これを忘れていた)
- 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.

# 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 で記述されるノードを増やすとソースコードのバイト数が削減できる

# ジェネレータの概要

- 定数置場
  - 001 以下に変数を置く
    - 000 も使えるとは思うものの当時の無理解に伴い使用していない
  - 1. 単方向リンクリストの一番端を表すポインタ (00100)
  - 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

# ジェネレータの概要

- 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
- ノード数削減のために“1...1”の文字列を定数として表す  
ノードを専攻して構築して
- `traverse()` 関数にて必要な箇所にpoint して使う

# ジェネレータの概要

- 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として ``00111`` を使う
  - ``00111`` に ``b[3]`` の内容へのポインタを使う



# ジェネレータの概要

- 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ので配列として ``00111`` を使う
  - ``00111`` に ``b[5]`` の内容へのポインタを使う

# ジェネレータの概要

- `1` の指す対象を変更
  - 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

# ジェネレータの概要

- 単方向リンクリストのポインタのつけかえ
  - 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ノードの位置をset で変更
  - 0011 としてしまうとそのポインタそのものが書きかわり“後方データ”になってしまうので“001111”を指定

# ジェネレータの概要

- 行数の処理
  - for文のi--;
  - 32回の引数を1減少させる
  - 0010011 を 001001 にAssign すればよい
- 終了判定
  - 0010010 がB0 をとれば終了. 次の状態は終了処理へ.

# ジェネレータの概要

- 終了処理 ( `1` の意味を変更したため)
  - 出力時には `00110` (単方向リンクリストの一番後方を指すポインタ) のノードを 1 にAssign
  - `00111` に幾度も拡張したおかげで、先頭行の結果, 第2行の結果,..., として結果は格納されている
  - 終了直前にAssign 処理を行うことで出力できる
  - 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.

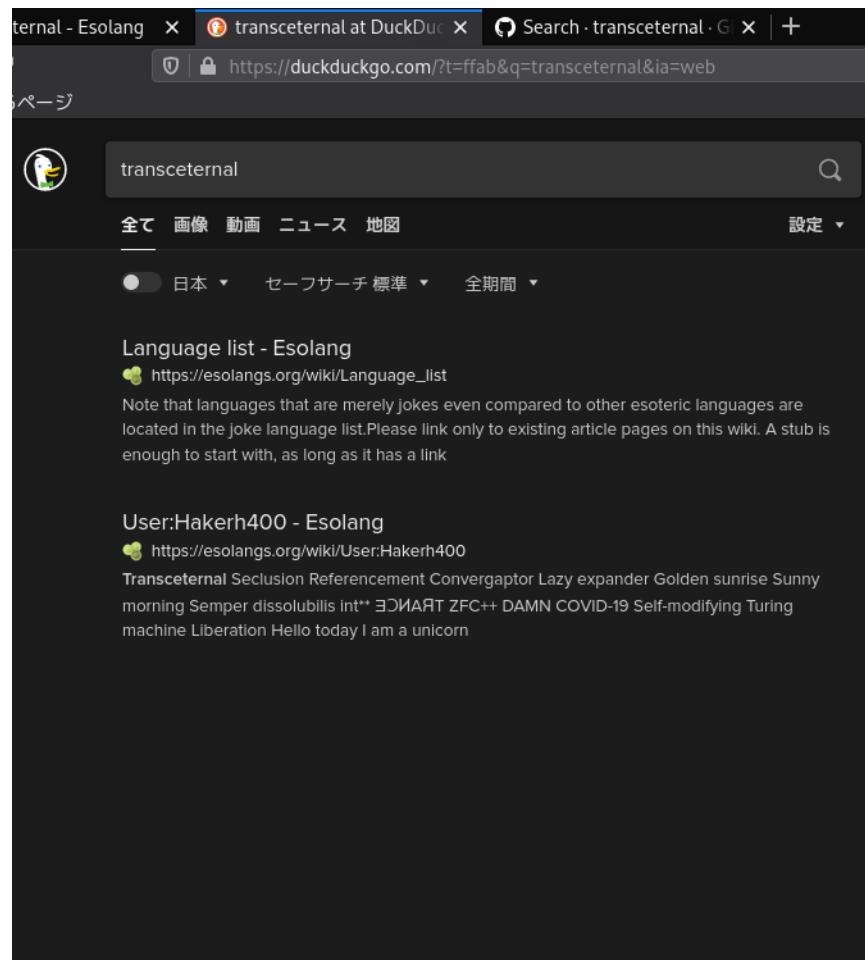
# 苦労した点

- 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



# 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
    - set 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
  - ランダムアクセスの構築

# 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