Esolang Codegolf #6 Transceternal Writeup

@_hiromi_mi

(Version 2020.5.9.0)

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
- 回答の説明
- 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(数値) Y=def (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
  - 入出力と deserialize, serialize
  - グラフの各ノードの意味
  - コマンドの意味
- 回答の説明
- 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

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: (ソースコード中から管理する最初のノード)
  - (管理領域, 次の状態)
- root: 一番上にあるノード
  - (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
- 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

実行の流れ

- ソースコードを token に分割
- ソースコードをdeserialize してグラフ構造に
- 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 (後述)
- `1` 以下を serialize して出力内容を生成

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)
  - ＢＣＤＤＤＥＥＥＤ (筆者作)
  - 123334443 (筆者作)

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”

ソースコードの deserialize

- スタック構造に乗せられて各バイトごとに順次解釈
- その枝が 0 -> B0 (`000` にある)に繋ぐ
- その値が 1 -> B1 (`111` にある) に繋ぐ

入力の 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

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かえす

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: _ _

01203...

Step2 (1)
0: 1 _
1: _ _

01203...

Step3 (2)
0: 1 _
1: 2 _
2: _ _

Initial Memory Consumption 例

- 012032004051025

Step1 (0)
0: _ _

Step2 (1)
0: 1 _
1: _ _

Step3 (2)
0: 1 _
1: 2 _
2: _ _

Step4 (0)
0: 1 _
1: 2 _
2: 0 _

Step5 (3)
0: 1 _
1: 2 _
2: 0 3
3: _ _

Step6 (2)
0: 1 _
1: 2 _
2: 0 3
3: 2 _

Step7 (0)
0: 1 _
1: 2 _
2: 0 3
3: 2 0

Step8 (0)
0: 1 _
1: 2 0
2: 0 3
3: 2 0

Step9 (4)
0: 1 4
1: 2 0
2: 0 3
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 は残っているが埋まっており、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

Serialize

- グラフ構造で表わされたものを 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”

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とたどっていった値
  - 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値
- 値: そのノードを中心に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

処理の説明

- Commandは3つ
  - Case 1: SET
  - Case 2: New
  - Case 3 Condition
- (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

SET

- 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

NEW

- (本問では未使用)

Condition

- type: B0 か B1 以外
- then : node->11 を 01 に代入
- else : 011 を 01 に代入

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
- 回答の説明
- 今後 (Golf の方針)

やりたいこと

- で変数, 入出力 (の書き換え), if, while を実現したい

変数は？

- 絶対アドレス `000` or `001` 以下は 0-pointer, 1-pointer ともに自由に使える
- 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

次の状態の指定

- “root” ではない (← これを忘れていた)

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
- 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

- then_node として 元々のノードを繋ぐ
- for文、というよりカウンタ機構の実装が面倒

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,... とならべる
- “たどる” 処理の実現に工夫が必要

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
- なので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
- 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....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わされる
- 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
- ++ は (処理内容の最後に置くことで実現

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
- 回答の説明
- 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 2I 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 v

方針

- Python でジェネレータを書いた

今回の場合

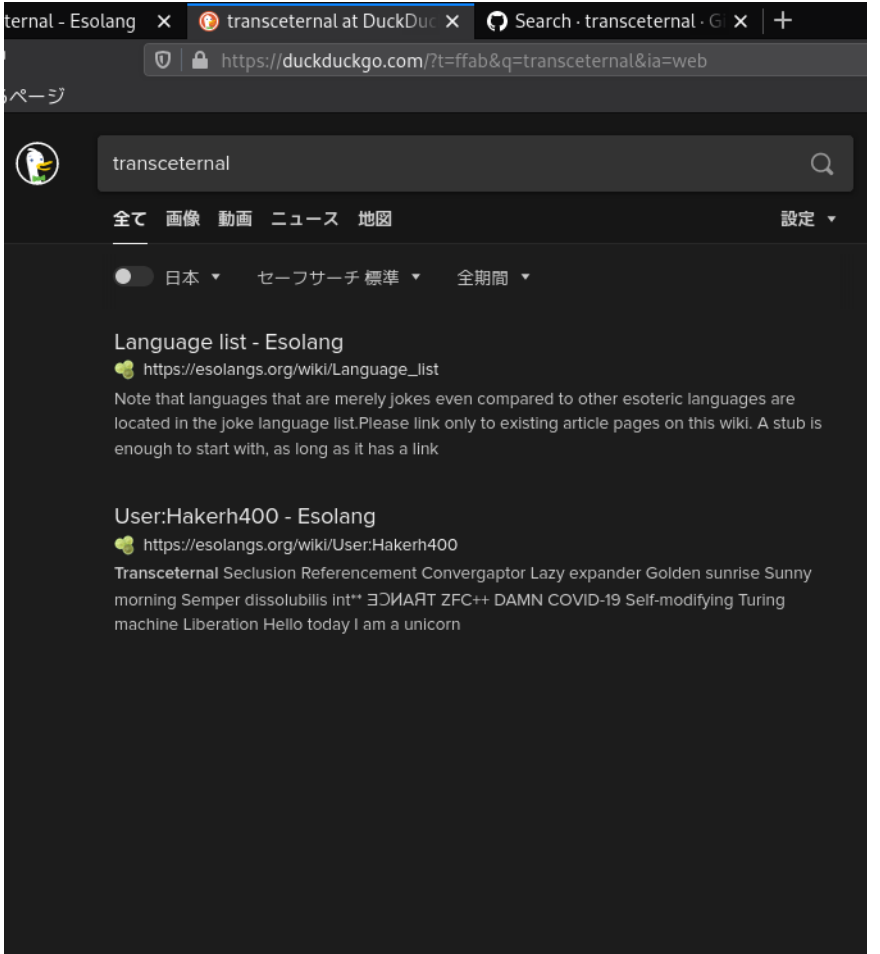
- token 列は0-9 + A-Z + a-z 62進法で生成

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
- 回答の説明
- 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
  - かけ算割り算などの構築

- 大会主催の @hakatashi さんと
- 助言をいただけた @drafear さん, @satos___jp さんに感謝