Korešpondenčný seminár z programovania
XXXIII. ročník, 2015/16
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Úlohy 2. kola letnej časti, kategória T

Termín odoslania riešení tejto série je pondelok **13. júna 2016**.

1. Tajná základňa

kat. T; 0 b za popis, 20 b za program

Najčastejšie dve aktivity v "Krčme u starého psa" sú bitie a pitie. Dokonca aj na pomery spoločensky unavených ľudí sa bitky odohrávajú príliš často. Nie je to náhoda – krčma je totiž súčasťou tajnej vojenskej základne a bitkári sú v skutočnosti veteráni trénujúci *drunken-style*.

Základňa pozostáva z viacerých miestností a medzi každou dvojicou miestností vedie práve jedna chodba. Za účelom čo najintenzívnejšieho tréningu sú niektoré chodby vybavené lávovými gejzírmi, padajúcimi stropmi, pichliačmi, kyvadlami, ...

Občas na základňu prídu významné osobnosti, ako napríklad pán prezident či najvyšší generál. Pán prezident by sa rád dostal do centrály a generál by chcel vidieť svojho syna. Nemajú však celý deň a chceli by sa dostať na miesto určenia čo najrýchlejšie.

Bežný vojak by sa do cieľa dostal priamo cez chodbu s rotujúcimi čepeľami a chodbou s rozpadajúcou sa podlahou pokrytou klzkým ľadom. Väčšina návštevníkov ale na takéto nebezpečné cesty nemá potrebné skúsenosti. Našťastie, niektoré chodby neskrývajú žiadne nástrahy a sú bezpečné.

Vedenie základne vás požiadalo, aby ste naprogramovali navigačný systém pre návštevníkov. Systém, ktorý nájde pre daný cieľ najkratšiu bezpečnú cestu – cestu bez nástrah.

Úloha

Zadaný máte graf, ale nie štandardným spôsobom. Namiesto toho, aby ste dostali zoznam hrán, ktoré v grafe sú, dostanete zoznam hrán, ktoré v ňom **nie sú** (zoznam nebezpečných chodieb).

Všetky hrany (medzi dvoma rôznymi vrcholmi), ktoré nedostanete na vstupe, v grafe sú.

Tiež dostanete zadaný vrchol *vstupnej haly*.

Na záver dostanete zadaných niekoľko cieľových vrcholov. Pre každý nájdite **najkratšiu** cestu v grafe vedúcu zo vstupnej haly do daného vrchola. Dĺžka cesty je počet hrán na nej. Ak je najkratších ciest viac, vypíšte ľubovoľnú.

Formát vstupu

Na prvom riadku vstupu sú tri celé čísla n, m, s – počet vrcholov, počet hrán, ktoré nie sú v grafe, a číslo vrchola vstupnej haly. Platí $n \leq 10^5$, $m \leq 5 \cdot 10^5$, $0 \leq s < n$.

Na každom z nasledujúcich m riadkov sú dve čísla $0 \leq a, b < n$, hovoriace o neexistencii hrán medzi vrcholmi a a b v oboch smeroch. Každá dvojica sa na vstupe vyskytne najviac raz a $a \neq b$.

Na ďalšom riadku je číslo q, udávajúce počet cieľových vrcholov. Na každom z nasledujúcich q riadkov je jedno celé číslo $0 \leq c < n$ – číslo cieľového vrchola. Je zaručené, že žiadne dva cieľové vrcholy nie sú rovnaké.

Formát výstupu

Pre každý cieľový vrchol vypíšte na samostatný riadok cestu vedúcu zo vstupnej haly doň v nasledujúcom formáte: $d\ v_1\ v_2\ \ldots\ v_d$, kde d je dĺžka cesty (počet chodieb na ceste), s je prvý navštívený vrchol, v_1 je nasledujúci navštívený, ..., v_d je posledný navštívený vrchol (zhodný s cieľovým vrcholom). Ak cesta neexistuje, vypíšte namiesto toho -1.

Obmedzenia na vstupy

Sada	1	2	3	4	5	6	7	8	9	10
$n \leq$	1000	1000	1000	10000	10000	100000	100000	100000	100000	100000
$m \leq$	2000	20000	500000	50000	50000	300000	300000	300000	300000	300000
$q =$	n	n	n	n	n	1	1	n	n	n

Príklad

vstup

```
5 6 0
0 2
0 3
0 4
1 3
1 4
2 4
5
0
1
2
3
4
```

výstup

```
0
1 1
2 1 2
3 1 2 3
4 1 2 3 4
```

vstup

```
7 9 6
0 2
0 3
0 4
0 5
0 6
1 3
1 4
1 5
1 6
7
0
1
2
3
4
5
6
```

výstup

```
3 2 1 0
2 2 1
1 2
1 3
1 4
1 5
0
```

2. Toľko voľného miesta!

kat. T; 0 b za popis, 20 b za program

Povedali ste si niekedy po nástupe do autobusu: "Toľko voľného miesta! Prečo tu nie sú sedadlá?". Nuž, časy, keď ste boli nútení stáť v poloprázdnom autobuse, budú už čoskoro minulosťou. Prichádza revolúcia v hromadnej doprave.

Na trhu sa totiž objavil superbus – autobus, v ktorom je každý centimeter štvorcový pokrytý čalúnenou sedačkou. Interiér superbusu obsahuje n radov a v každom z nich je m sedačiek vedľa seba. Navyše, sedačky nie sú všetky otočené dopredu. Aby si cestujúci mohli vybrať smer, ktorým sa chcú počas jazdy pozerať, v superbuse sú sedadlá pootáčané do každého zo 4 smerov.

Pri prvom testovaní superbusu sa ale zistilo, že má jednu fatálnu chybu – niektoré sedačky sú otočené čelom k sebe. A keď sú obe obsadené, niet miesta pre nohy cestujúcich.

Dizajnér superbusu preto potrebuje niektoré sedačky otočiť niekoľkokrát o deväťdesiat stupňov tak, aby žiadne dve sedačky neboli k sebe otočené čelom. Nakoľko už bolo vyrobených veľa superbusov, v každom z nich bude nutné tieto sedačky otočiť. To je veľa roboty – je preto žiadúce, aby bolo potrebné sedačky otáčať čo najmenejkrát.

Úloha

Na vstupe je mriežka, v ktorej je každé políčko – sedadlo – otočené na sever, východ, juh alebo na západ.

O mriežke hovoríme, že je v *dobrom* stave práve vtedy, keď žiadne dve susedné sedadlá nie sú natočené proti sebe.

V jednom kroku je povolené otočiť niektoré sedadlo o deväťdesiat stupňov doľava alebo doprava. Zistite, koľko najmenej krokov je potrebných na to, aby sme mriežku dostali do dobrého stavu.

Netradične vám k tejto úlohe naznačíme riešenie. Ak nápovedu nechcete vedieť, nepozerajte do poznámky pod čiarou. Ak si s úlohou neviete poradiť, možno vám poznámka pod čiarou[1] pomôže.

Formát vstupu

Na prvom riadku vstupu sú dve celé čísla $1 \leq n, m \leq 50$ – rozmery mriežky.

Nasleduje n riadkov, na každom z nich je m znakov popisujúcich natočenie danej sedačky. Znak `^` zodpovedá natočeniu na sever, `>` na východ, `v` na juh, a `<` na západ.

Formát výstupu

Na výstup vypíšte jedno celé číslo – najmenší počet krokov, na ktorý je možné dostať mriežku do dobrého stavu.

Príklad

vstup	výstup
3 3 >v< >^< ^^^	2

Otočíme sedadlo v strede prvého riadku dvarát o 90 stupňov.

vstup	výstup
1 10 >>><<<>>><	2

Napríklad môžeme otočiť tretie sedadlo zľava a tretie sedladlo od konca smerom na sever.

3. Teatrálny Programovací Jazyk

kat. T; 0 b za popis, 20 b za program

Ak máte akékoľvek otázky k tejto úlohe, napíšte Mišovi Štrbovi na faiface@ksp.sk.

Paulínka si otvorila zadanie tejto úlohy (ale až po zverejnení ... naozaj!) a zvolala:

"Čo si tí KSPáci zase vymysleli? Bojím, bojím, vyzerá to na nejaký programovací jazyk. To od nás chcú, aby sme si naprogramovali vlastný jazyk? Ja si pamätám, na Zenite raz bola taká úloha... Animassembler, to bolo škaredéé."

Áno, Paulínka, uhádla si to správne. V tejto úlohe si máte naprogramovať programovací jazyk. Ale nie vlastný, my sme vám už nejaký vymysleli.

"Och, to bude určite nejaký škaredý ezoterický jazyk, nejaké nenaprogramovateľné veci, bojím, bojím. No, určite to nebude normálny programovací jazyk, bude to niečo šialené."

Tak, sklamem ťa Paulínka, nie je to žiaden ezoterický jazyk. Ale báť sa môžeš stále. **Je to totiž úplne normálny programovací jazyk.** Podobný, ako skoro všetky, v ktorých programujete KSP. Tak, poďme sa na tento programovací jazyk pozrieť.

VEĽKÉ UPOZORNENIE: Pre získanie čiastočných bodov stačí naprogramovať len veľmi malú časť tohto jazyka a aj to veľmi nekompletným spôsobom. Nikdy sa nevzdávajte.

Zoznámte sa

```
func main() {
    print 1;
}
```

Takto vyzerá *Hello, World!* v Teatrálnom Programovacom Jazyku (ďalej len TPJ). V skutočnosti je to skôr *Hello, one!*, keďže program vypisuje číslo 1. Tento jazyk totiž nepozná stringy (to je pre vás asi potešujúca správa). Predtým, než sa pozrieme na syntax TPJ detailnejšie, ukážme si ešte zopár príkladov.

Program, ktorý vypíše súčin 7×9:

[1] Pozor, blíži sa prezradenie nápovedy. Odporúčame naštudovať si riešenie problému minimum-cost maximum flow. Na internete nájdete veľa dobrých aj zlých materiálov k tejto téme. Nezabudnite však, že program, ktorý odovzdáte, musí byť váš vlastný, nie skopírovaný z cudzieho zdroja.

```
func main() {
    var a;
    var b;

    a = 7;
    b = 9;

    print a*b;
}
```

Program, ktorý vypíše čísla od jedna po desať:

```
func main() {
    var i;
    i = 1;
    while i <= 10 {
        print i;
        i = i + 1;
    }
}
```

Program s funkciou na počítanie faktoriálu:

```
func factorial(n) {
    if n <= 1 {
        return 1;
    }
    return n * factorial(n-1);
}

func main() {
    print factorial(7);
}
```

Základy

TPJ je kompilovaný jazyk. To znamená, že ak použijete nedeklarovanú premennú alebo zadáte zlý počet parametrov funkcie alebo niečo podobné, tak program sa neskompiluje. Napríklad tento program by sa neskompiloval, lebo premenná `b` nebola deklarovaná:

```
func main ( ) {
    var a;
    print b;
}
```

Nedodržanie pravidiel jazyka treba vždy vyhodnotiť ako `COMPILATION ERROR`.

Každý program v jazyku TPJ musí mať funkciu `main()`. Neskôr sa pozrieme aj na to, ako písať v TPJ vlastné funkcie. Funkcia `main()` vyzerá takto:

```
func main() {

}
```

Funkcia `main()` **nesmie** brať žiadne parametre. Program v jazyku TPJ sa začne spustením funkcie `main()`. Teraz sa pozrieme na všetky vymoženosti, ktoré môžeme písať dovnútra funkcie `main()`.

Premenné

Všetky premenné v jazyku TPJ sú typu **signed 32-bit integer**, čiže normálny `int` v C. Sú to celé čísla vyjadrené pomocou 32 bitov v doplnkovom binárnom kóde. Nekontroluje sa pretečenie (overflow), takže tento program vypíše `-2147483648`:

```
func main() {print 2147483647 + 1;}
```

Premennú v jazyku TPJ deklarujeme pomocou kľúčového slova `var`, takto:

```
var meno_premennej;
```

Pred použitím je nutné premennú deklarovať. **Priamo po deklarácii má premenná hodnotu 0.**

Názov premennej môže byť ľubovoľný reťazec zložený z veľkých a malých písmen anglickej abecedy, číslic a podtržítka (_). Nesmie však začínať číslicou. Nasledujúci program ukazuje deklarovanie premenných (vypíše tri nuly):

```
func main() {
    var a;
    var b;
    print a;
    print b;

    var uplne_zmysluplna_premenna_123;
    print uplne_zmysluplna_premenna_123;
}
```

Premenné môžu byť deklarované hocikde vo funkcii, avšak použité môžu byť až po deklarácii. Navyše, TPJ je **úplne pokrokový jazyk** a **nemá globálne premenné**. Každá premenná musí byť deklarovaná vo funkcii.

Nie je možné dvakrát deklarovať tú istú premennú. Tento program sa neskompiluje:

```
func main() {
    var a;
    print a;

    var a;
}
```

Do premennej je možné priradiť hodnotu pomocou príkazu `=`. **Pozor, `=` nie je operátor, nie je možné ho použiť vo výrazoch.**

```
func main() {
    var hello;
    hello = 42;
    hello = 47;
}
```

Výrazy

Číslo v jazyku TPJ je ľubovoľný reťazec číslic v desiatkovej sústave. Číslo teda môže začínať veľa nulami, ktoré nemajú žiaden význam. Ak sa číslo nezmestí do 32 bitov, zoberie sa jeho zvyšok po delení 2^{32} a ten sa interpretuje v doplnkovom binárnom kóde. Znamienko `-` nie je súčasťou čísla. Namiesto toho je to unárny operátor. `-123` je teda číslo `123`, na ktoré je použitý operátor `-`.

Operátory

Jazyk TPJ podporuje nasledujúce operátory:

- Matematické unárne:
  - `+` – nespraví nič, zachová hodnotu operandu
  - `-` – vráti hodnotu operandu vynásobenú -1
- Matematické binárne:
  - `+` – sčítanie
  - `-` – odčítanie
  - `*` – násobenie
  - `/` – celočíselné delenie

  - `%` – zvyšok po delení, výsledok má znamienko ľavého operandu – ako v jazyku C (a nie ako v Pythone)

- Porovnávacie:
  - `==` – vráti 1, ak sa operandy rovnajú, inak 0
  - `!=` – vráti 1, ak sa operandy nerovnajú, inak 0
  - `<` – vráti 1, ak je ľavý operand menší ako pravý, inak 0
  - `>` – vráti 1, ak je ľavý operand väčší ako pravý, inak 0
  - `<=` – vráti 1, ak je ľavý operand menší alebo rovnaký ako pravý, inak 0
  - `>=` – vráti 1, ak je ľavý operand väčší alebo rovnaký ako pravý, inak 0
- Logické unárne:
  - `!` – pre nulu vráti 1, pre hocičo nenulové vráti 0
- Logické binárne:
  - `&&` – logické *a*, vráti 1, ak sú oba operandy nenulové, inak 0
  - || – logické *alebo*, vráti 1, ak je aspoň jeden z operandov nenulový, inak 0

Pri vyhodnocovaní operátorov `&&` a || sa **vždy vyhodnotia oba operandy**.

Ak dôjde k deleniu (`/`) alebo modulovaniu (`%`) nulou, tak výsledok je vždy nula. Nasledujúci program vypíše `5`:

```
func main() {
    print 5 + 5/(5 - 5);
}
```

Vyhodnocovanie výrazov

Operátory v jazyku TPJ nemajú žiadne prednosti, všetky operátory sú rovnocenné a výrazy sa vyhodnocujú **sprava doľava**. Napríklad tento výraz sa vyhodnotí na -280 a nie na 37:

```
8*4-3*2+11
```

Výrazy môžete aj zátvorkovať. **Zátvorky** sú jediný spôsob, ako explicitne určiť poradie vyhodnocovanie výrazu. Takto uzátvorkovaný výraz sa už vyhodnotí na 37:

```
((8*4) - (3*2)) + 11
```

V takomto výraze sa najprv vyhodnotí funkcia `c()`, potom `b()` a nakoniec `a()`:

```
a() + b() + c()
```

Bloky

Všetko, čo sa nachádza medzi kučeravými zátvorkami { a } je **blok**. Do programu v jazyku TPJ môžete bloky vkladať do seba podľa ľubovôle. Napríklad toto je úplne platný program:

```
func main() {
    {
        {
            {}{{}}{{{}}}
        }
        {
            {{{}}}{{}}{}
        }
    }
    {
        { { } }
    }
    {
        {
        }
```

```
        {
        }
        {
        }
    }
}
```

V každom bloku môžete deklarovať premenné. **Po ukončení bloku však platnosť všetkých premenných, ktoré v ňom boli deklarované zanikne.** Napríklad tento program sa kvôli tomu neskompiluje, pretože premenná `a` zanikla po ukončení bloku, v ktorom bola deklarovaná.

```
func main() {
    {
        var a;
        a = 10;
        print a;
    }

    print a;
}
```

Nasledujúci program sa taktiež neskompiluje, ale z iného dôvodu a síce preto, že je zakázané jednu premennú znova deklarovať, ak jej platnosť ešte nezanikla:

```
func main() {
    var a;

    {
        var a;
        a = 10;
        print a;
    }

    print a;
}
```

Blok samozrejme nežije len sám pre seba. Obsahuje príkazy, ďalšie bloky a podobne. Do bloku je možné umiestňovať nasledujúce veci (po veciach, ktoré majú v zozname nižšie na konci napísanú bodkočiarku, tá **bodkočiarka musí byť**):

- iný blok – `{ }`
- deklarácia premennej – `var meno_premennej;`
- priradenie hodnoty do premennej – `premenna = 2 * x + 13 - factorial(10);`
- vetvenie `if` (vysvetlené nižšie) – `if x == 3 { ... }`
- cyklus `while` (vysvetlený nižšie) – `while x < 10 { ... }`
- príkaz `return` (vysvetlený nižšie) – `return vysledok;`
- príkaz `print` (vysvetlený nižšie) – `print cislo;`
- výraz ukončený bodkočiarkou – `3 + 3 + funkcia(10);`
- prázdny príkaz – `;`

Vetvenie `if`

Jazyk TPJ podporuje klasické `if`, ktorý poznáte z iných jazykov. Vyzerá takto (<`blokN`> a <`podmienkaN`> sú placeholdery pre skutočné výrazy a bloky príkazov):

```
if <podmienka1> {
    <blok1>
} else if <podmienka2> {
    <blok2>
} else if <podmienka3> {
```

```
    <blok3>
} else {
    <blok4>
}
```

Vetvy `else if` a `else` sú samozrejme nepovinné. V jazyku TPJ **nemusia byť okolo podmienky v príkaze `if` zátvorky**, avšak blok príkazov, ktoré sa majú vykonať ak podmienka platí, **musí byť uzavretý v kučeravých zátvorkách**. Nasledujúce vetvenia `if` nie sú skompilovateľné:

```
if x == 3 {
    print x;
    else{
        print 10;
    }
}

if ( x == 3 )print x;
```

Prvý príklad nie je skompilovateľný preto, že vetva `else` nemá nad sebou `if` a druhý preto, že okolo príkazu `print x` nie sú kučeravé zátvorky.

Cyklus `while`

Cyklus `while` je jediný cyklus podporovaný jazykom TPJ. Vyzerá veľmi jednoducho:

```
while <podmienka> {
    <blok>
}
```

To je všetko, príkazy medzi kučeravými zátvorkami sa budú vykonávať pokým bude platiť podmienka. Jazyk TPJ **nepodporuje príkazy `break` ani `continue` známe z iných jazykov. Tešte sa.**

Podobne ako vo vetvení `if`, ani v cykle `while` nie je nutné dávať podmienku do zátvoriek, ale je nutné dávať blok príkazov, ktoré sa majú vykonávať do kučeravých zátvoriek.

Príkaz `print`

Jazyk TPJ nevie načítavať žiaden vstup. Každý program teda pri každom spustení vypľuje ten istý výstup. Na výstup je možné vypisovať pomocou príkazu `print`. Tento príkaz vypíše **jedno číslo ukončené znakom nového riadku**. Vyzerá takto:

```
print <výraz>;
```

Funkcie

Doteraz sme prebrali všetko, čo je možné písať dovnútra funkcie. Jazyk TPJ však podporuje aj vytváranie vlastných funkcii. Každá funkcia má svoje meno a parametre. Definujeme ju takto:

```
func <názov_funkcie>(<parameter1>, <parameter2>, <parameter3>, ...) {
    ...
}
```

Pre názov funkcie aj názvy parametrov platia rovnaké obmedzenia ako pre názvy premenných. Funkcia môže brať ľubovoľný počet parametrov. Po spustení funkcie sa parametre správajú rovnako ako lokálne premenné, ktorých počiatočné hodnoty sú zadané pri volaní.

Nie je možné definovať dve funkcie s rovnakým meno.

Všetky funkcie v jazyku TPJ vracajú 32-bitový signed integer a všetky premenné sú tohto typu.

Každý príkaz sa musí nachádzať vo funkcii. Navyše je **zakázané definovať funkciu vo funkcii.** Tieto dva kódy by sa neskompilovali:

Tu je príkaz mimo funkcie:

```
func main() {
    print 1;
}

print 2;
```

A tu je funkcia vo funkcii:

```
func main() {
    func vnorena() {
        return 4;
    }

    print vnorena();
}
```

Príkaz `return`

Logicky, keďže funkcia má vracať nejakú hodnotu, potrebujeme na to prostriedky. Tie nám poskytuje príkaz `return`. Vyzerá takto:

```
return <výraz>;
```

Po vykonaní príkazu `return` funkcia skončí a riadenie sa odovzdá volacej funkcii, pričom funkcia sa vyhodnotí na vrátenú hodnotu.

Napríklad táto funkcia berie ako parametre dve čísla a vracia menšie z nich:

```
func min(a, b) {
    if a < b {
        return a;
    }
    return b;
}
```

Alebo táto funkcia vypočíta faktoriál čísla `n`:

```
func factorial(n) {
    var result;
    result = 1;

    var i;
    i = 2;
    while i <= n {
        result = result*i;
        i = i+1;
    }

    return result;
}
```

V prípade, že funkcia skončí bez toho, aby nastal príkaz `return`, tak funkcia automaticky vráti hodnotu `0`.

Volanie funkcii

Funkcie sa volajú rovnako ako vo väčšine programovacích jazykov. Volanie funkcie je *výraz*, ktorý sa vyhodnotí na návratovú hodnotu funkcie. Napríklad nasledujúci výraz sa vyhodnotí na `1` (používame funkcie definované vyššie):

```
factorial(4) == min(24,47)
```

Volanie funkcie má syntax `<názov_funkcie>(<výraz1>, <výraz2>, ..)`. Volaná funkcia **musí existovať** a musí byť volaná so **správnym počtom parametrov**. Avšak **nemusí byť definovaná pred použitím**. Takýto kód je úplne v poriadku:

```
func main() {
    print add(2, 3);
}
```

```
func add(a, b) {
    return a+b;
}
```

Rekurzia

Jazyk TPJ podporuje rekurziu. Funkcia teda môže volať sama seba a funguje to, ako keby volala inú funkciu (teda po skončení rekurzívneho volania ostanú lokálne premenné funkcie nedotknuté).

Uveďme si dva príklady.

Prvý je funkcia na vypisovanie Fibonacciho čísel. Nasledujúci program vypíše prvých 20 čísel tejto postupnosti:

```
func fibonacci (a, b, n) {
    if n > 0 {
        print a;
        fibonacci(b, a + b, n-1);
    }
}

func main() {
    fibonacci(0,1,20);
}
```

Nasledujúci program vypíše postupne čísla `10 10 9 9 8 8 ... 1 1` (samozrejme, na osobitných riadkoch). Nepoužíva priamo rekurziu, ale robí akýsi "ping-pong" medzi dvomi funkciami:

```
func a(n) {
    if n > 0 {
        print n;
        b(n);
    }
}

func b(n) {
    print n;
    a(n - 1);
}

func main() {
    a(10);
}
```

Úloha

Hurá, konečne sme si popísali celý náš krásny jazyk TPJ. Tak čo je vlastne úloha? Váš program dostane na vstupe nejaký kód v jazyku TPJ. Ak je tento kód chybný (nespĺňa niektorú z podmienok uvedených v popise jazyka), tak váš program má vypísať `COMPILATION ERROR`.

V opačnom prípade má spustiť program, ktorý dostal na vstupe a vypísať jeho výstup. Je zaručené, že žiaden zo vstupných programov na testovači sa nezacyklí a ani nebude bežať príliš dlho.

Všimnite si krásu jazyka TPJ. Ak sa už raz skompiluje, nie je možné aby sa počas behu zrúbal.

Formát vstupu

Na vstupe máte zdrojový kód nejakého programu v jazyku TPJ. Kód je ukončený znakom konca súboru.

Formát výstupu

V prípade, že program na vstupe nie je korektný, vypíšte reťazec `COMPILATION ERROR`. V opačnom prípade vypíšte n riadkov, kde n je počet vykonaných príkazov `print` vo vstupnom programe. Na každom z týchto riadkov nech sa nachádza číslo, ktoré daný príkaz `print` vypísal.

Sady

Aj keď to vyzerá ťažko, získať čiastočné body naozaj nie je ťažké. Takýto bude obsah testovacích sád:

1. Obyčajné príkazy `print` s jednoduchými výrazmi bez zátvoriek. Všetko je pekne naformátované, odsadené, jeden príkaz na jednom riadku. Všetky operátory a čísla sú oddeľované medzerami.

vstup

```
func main() {
    print 2;
    print 3;
    print 5 + 2 % 7;
}
```

výstup

```
2
3
7
```

2. Teraz už vo výrazoch pribudli aj zátvorky. Operátory a čísla už nemusia byť oddelené medzerami.

vstup

```
func main() {
    print (2 + 2) - (3 + 3);
    print (12 % 3) + 5 / 3 * ((9 + 9) - 2);
}
```

výstup

```
-2
0
```

3. Deklarácie premenných. Priraďovanie hodnôt a výrazov do premenných. Žiadne chyby kompilácie.

vstup

```
func main() {
    var a;
    a = 3;

    var b;
    b = 7;

    print (a + b)%47;
}
```

výstup

```
10
```

4. Použitie blokov. Treba zachytiť chybu pri kompilácii kvôli použitiu nedeklarovanej premennej alebo použitiu premennej mimo svojej scope.

vstup

```
func main() {
    var a;

    {
        var a;
        var b;
        b = 5;
        print b;
    }

    print a;
    print b;
}
```

výstup

```
COMPILATION ERROR
```

5. Pribudli štruktúry `if` a `while`. Treba zachytiť nekorektné výrazy v podmienkach.

vstup

```
func main() {
    var i;

    while i <= 10 {
        if (i % 2) == 0 {
            print 1000;
        }
        print i;
        i = i+1;
    }
}
```

výstup

```
1000
0
1
1000
2
3
1000
4
5
1000
6
7
1000
8
9
1000
10
```

6. Funkcie bez `return`. Zatiaľ volané len z `main`. Treba kontrolovať počet parametrov a správny názov funkcie pri volaní.

vstup

```
func add(x, y) {
    print x + y;
}

func main() {
    add(7, 7);
}
```

výstup

```
14
```

7. Pribúda príkaz `return`. Treba vedieť vracať hodnotu, pracovať s návratovou hodnotou a tiež korektne ukončiť funkciu pri použití `return`.

vstup

```
func add(x, y) {
    return x + y;
}

func main() {
    print add(7, 7) % 47;
}
```

výstup

```
14
```

8. Teraz sa už funkcie môžu volať navzájom a dokonca aj rekurzívne.

vstup

```
func fibonacci(a, b, n) {
    if n > 0 {
        print a;
        fibonacci(b, a + b, n - 1);
    }
}

func main() {
    fibonacci(0, 1, 10);
}
```

výstup

```
0
1
1
2
3
5
8
13
21
34
```

9. Táto sada testuje schopnosti vášho parsera vysporiadať sa s nedostatkom bielych znakov.
10. Všelijaké syntaktické a iné chyby v rôznych programoch.