Korespondenční Seminář z Programování

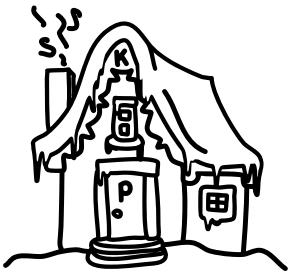
37. ročník **KSP** **Březen 2025**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám čtvrté číslo hlavní kategorie 37. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.



Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce této série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **13.4. 2025 ve 32:00 (tedy další ráno v 8:00)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/h/odevzdavatko/*

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha; Experimentální (neobvyklá) úloha

Odměna série: **Odznáček do profilu na webu** si vysloužíti nejlepší závodníci v Asteraceru.

Čtvrtá série třicátého sedmého ročníku KSP

37-4-1 Otočení vlaku — 9 bodů

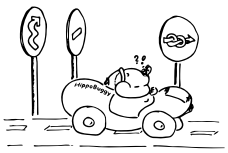
V dispečinku depa Hrochova nastala nepříjemná situace. Expresní vlaková souprava vlivem mimořádné události, kdy musela jet odklonem přes úplně jinou trasu, přijela do depa pozpátku! To se cestujícím nebude líbit, aby první třída byla na špatné straně vlaku!

Depo sice má k dispozici místo, kde může točit lokomotivy, ale tato souprava se tam nevejde. Musíme tedy soupravu vypustit bez cestujících na železnici, aby se nám někde otočila a přijela zpět do depa správným koncem.

Železnici máme popsanou jako neorientovaný graf, kde vrcholy tvoří uzly (ať už stanice nebo třeba jen rozbočky) propojené tratěmi. Prozíravý správce železnice dokonce v každém uzlu postavil koleje tak, že se lze z uzlu dostat mezi libovolnými dvěma různými tratěmi z něj vedoucími bez úvratě – obrácení směru cesty soupravy. Z depa, taky uzlu, vede do zbytku železnice právě jedna trať (jde tedy o list).

Centrální dispečink Hrošín však má plné ruce práce s dodržováním jízdních řádů ostatních vlaků, proto této soupravě jakožto mimořádnému vlaku bez cestujících zakázal kdekoliv provést úvrať – ta z komplikovanosti železničních předpisů vyžaduje větší pozornost dispečinku. Projet stejnou trať vícekrát bez úvratě mu však nevadí.

Nám tedy nezbývá, než navrhnout centrálnímu dispečinku trasu z depa do depa bez úvratě. Protože každý kilometr cesty stojí peníze, chceme, aby naše souprava cestovala po trati nejkratší trasu. Pomozte ji najít.



Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu:

Na prvním řádku dostanete tři čísla N, M a D oddělené mezerou, kde N je počet železničních uzlů číslovaných $0, \ldots, N-1$, M značí počet tratí a D je číslo uzlu patřící depu.

Na následujících M řádcích najdeme tři čísla u_i, v_i a ℓ_i, popisující i-tou trať mezi uzly u_i a v_i o délce ℓ_i kilometrů.

Formát výstupu:

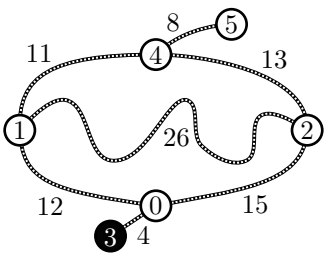
Na první řádek vypište K – počet uzlů, kterými souprava projede na nejkratší trase, kde první a poslední uzel je D. Na dalším řádku pak postupně vypište jednotlivé uzly, kterými souprava projede, oddělené mezerou.

Ukázkový vstup:

```
6 7 3
0 3 4
0 1 12
2 0 15
2 4 13
1 4 11
1 2 26
5 4 8
```

Ukázkový výstup:

```
7
3 0 1 4 2 0 3
```



Protože přímá trať mezi uzly 1 a 2 je klikatá, je kratší trasa přes uzel 4. Pokud by depo bylo v uzlu 5, už by se vyplatilo využít klikatou trať.

37-4-2 Bomby a tunel — 12 bodů

Stavba nové linky D pražského metra vyžaduje vyhloubení tunelu dlouhého M metrů. Výstavba bohužel trvá na náš vkus již příliš dlouho, a tak jsme se rozhodli ji trochu urychlit a vzít hloubení tunelu do vlastních rukou.

Jelikož však nemáme k dispozici profesionální techniku, budeme se muset spokojit s naší různorodou zásobou B bomb. Každá z těchto bomb má danou svou hmotnost w_i a délku tunelu d_i, kterou vyrazí. Při každém odpalu je nutné nejprve dopravit bombu až na konec zatím vyhloubeného úseku tunelu a teprve tam ji odpálit. Dopravou bomby spotřebujeme množství energie odpovídající součinu hmotnosti bomby a vzdálenosti, kterou jsme s ní museli urazit (tedy délky dosavadního úseku).

Naším cílem je odpálit několik bomb v takovém pořadí, aby vynaložená energie byla co nejmenší. Výsledný tunel, který tak vyhloubíme, musí mít délku přesně M (metrů).

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku dostanete dvě celá čísla M a B, tedy žádanou délku tunelu a počet bomb, které máme k dispozici. Na dalších B řádcích dostanete celá čísla w_i a d_i – hmotnost i-té bomby a vzdálenost, kterou prorazí.

Formát výstupu: Na první řádek vypište počet bomb V, které vaše řešení potřebuje. Na následujících V řádcích očekáváme čísla bomb, které použijeme, v pořadí, jak je do vznikajícího tunelu budeme dopravovat. Bomby číslujeme od 1 do B (pozor, ne od 0 do $B-1$).

Ukázkový vstup:

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

Ukázkový výstup:

```
2
1
3
```

Jako první volíme pětikilovou bombu, kterou však (podle zadání) nemusíme nikam nosit, takže nespotřebujeme žádnou energii. Tím máme vyhloubený tunel délky 3 a dokončíme jej jednokilovou bombou s číslem 3. Celkem jsme tak spotřebovali 3 jednotky energie (1 kilo dopravené 3 metry). Kdybychom jako první odpalovali čtvrtou bombu místo první, byl by výsledek stejný, takže je to také validní řešení.

Lehčí varianta (za 3 body): U prvních dvou vstupů slibujeme, že součet délek d_i všech bomb se sečte přesně na M.

37-4-3 Detekce (3/4)-cyklů — 12 bodů

Koťátko Fousek se ubytovalo na chatě v malebných zimních Jizerských horách. Všude je spousta zajímavých míst – vrcholky, chaty, lázně, jezera… Fousek si pořídil mapu okolí, takže ví, odkud kam vedou přímé cesty a rád by si udělal krásnou zimní procházku sněhem. Jelikož by Fouskovi jinak byla zima, tak chce udělat maximálně 3 zastávky. Také se ale nechce vracet stejnou cestou jakou přišel, protože Fousek je toulavý dobrodruh a nikdy se neohlíží za sebe. Kvůli krásám okolí si ale vůbec nevšiml, že si pořídil mapu, kde jsou stezky zadané maticí sousednosti – tabulkou, kde pro libovolné i, j je $A_{ij} = 1$, pokud mezi vrcholy i a j vede přímá stezka, jinak jsou tam samé nuly. To ale Fouska přece nezastaví! Pomozte Fouskovi v neorientovaném grafu zadaném maticí sousednosti najít 3-cyklus nebo 4-cyklus.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N – počet zajímavých míst na mapě. Na dalších N řádcích dostanete vždy N nul či jedniček oddělených mezerou, reprezentující řádky matice sousednosti neorientovaného grafu. Graf může obsahovat 3-cykly i 4-cykly, nebo může obsahovat 3-cykly ale žádné 4-cykly či naopak, nebo nemusí obsahovat vůbec žádný 3-cyklus ani 4-cyklus.

Formát výstupu: Vypište nalezený 3-cyklus nebo 4-cyklus. Při nalezení 3-cyklu, vypište na jednom řádku `TRICYKLUS` u v w, kde u, v, w jsou vrcholy na třícyklu. Jinak při nalezení 4-cyklu, vypište na jednom řádku `CTYRCYKLUS` u v w x, kde u, v, w, x jsou vrcholy na čtyřcyklu v nějakém cyklickém pořadí. Vrcholy indexujeme od 0. Pokud graf neobsahuje ani třícyklus, ani čtyřcyklus, vypište `NIC`.

Ukázkový vstup:

```
6
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 1 0 0
0 1 1 0 1 0
0 1 0 1 0 1
0 0 0 0 1 0
```

Ukázkový výstup:

```
CTYRCYKLUS 0 1 3 2
```

Ukázkový vstup:

```
6
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 1 0 0
0 1 1 0 1 0
0 1 0 1 0 1
0 0 0 0 1 0
```

Ukázkový výstup:

```
TRICYKLUS 4 3 1
```

Lehčí varianta (za 4 body): Můžete předpokládat, že se v grafu nevyskytuje 3-cyklus.

37-4-4 Asteracer — 12 bodů

"Tři!"

Kevin nervózně svírá joystick. Sedí na můstku své ostřílené vesmírné lodi AstroŠutr.

"Dva!"

"Reaktor na plný výkon!" zařve Kevin do lodního telefonu. Ručičky budíků se roztřesou, megawatty narůstají. Za zády se mu ozývá hrozivé dunění.

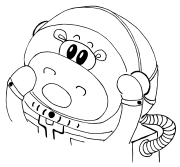
"Jedna!"

Dokáže doletět do cíle rychleji než jeho mnoholetá nemesis, loď Erebium?

"Start!"

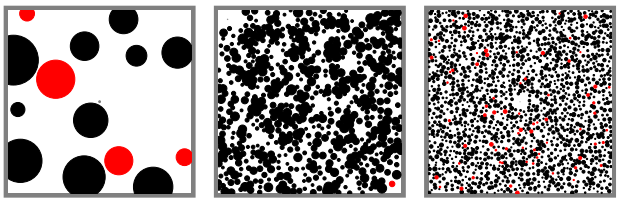
Z motorů vyšlehla plasma a loď se dala do pohybu. Jen co nabere trochu rychlosti, musí ale hned přibrzdit a změnit směr. Závody se totiž odehrávají ve zrádném poli asteroidů, je třeba se jim opatrně vyhýbat, ale přitom stále udržovat co největší rychlost. Kevin se proplétá skulinkami mezi vesmírným kamením. Závod pokračuje a na lodi začíná být pořádné vedro, když v tom se za zatáčkou objeví přímo v trase lodi skalní stěna. "U Jupiterových měsíců, tomu se už nevyhnu" pomyslí si Kevin. Loď vrazí plnou rychlostí do skály a odrazí se v mračnu prachu. Za zády se Kevinovi rozezní pronikavý alarm. Loď poté setrvačností vrazí do dalšího asteroidu, a proces se ještě několikrát opakuje, než loď téměř zastaví. Před okny kokpitu prolétá kamení, utržený solární panel a několik trubek. To bude mít opravář zase radost.

Kdyby si jen Kevin svou trasu lépe předpočítal…



V této úloze budete odevzdávat instrukce pro vaši vesmírnou loď, která závodí v poli asteroidů. Každá instrukce kóduje „tah motorů“, tedy požadované zrychlení lodi v daném okamžiku závodu. Odevzdávátko odsimuluje průběh závodu pomocí vašich instrukcí, vyhodnotí, zda byly splněny podmínky závodu a umístí vás do výsledkové tabulky. Čím méně instrukcí potřebujete k dokončení závodu, tím lepší máte „čas“ a tím lépe se umístíte!

Cílem vesmírného závodu je proletět všemi góly za co nejmenší počet instrukcí a vyhýbat se asteroidům, které loď zpomalují. Závodní okruhy jsou celkem 3: jednoduchý (`test`), na kterém je vhodné testovat řešení, krátký (`sprint`) s jedním gólem a spoustou asteroidů, a dlouhý (`marathon`) s větším počtem gólů a menším počtem asteroidů.



Instrukce jsou celočíselné dvojice hodnot (d_x, d_y) a určují, jakým směrem má loď zrychlovat. Aby se loď nepřehřála, tak je maximální platná délka instrukce $d = \sqrt{d_x^2 + d_y^2} \leq 127$. Libovolná kratší instrukce je platná, loď však poletí pomaleji než by mohla.

Pohyb lodi je rozdělen na tři části:

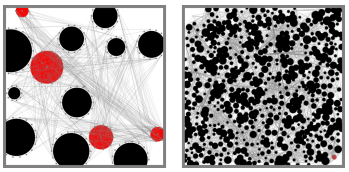
- **odpor vzduchu** – rychlost lodi je *snížena na 90% aktuální rychlosti*,
- **zrychlení** – k rychlosti lodi je *přičtena aktuální instrukce*,
- **posun** – k pozici lodi je *přičtena aktuální rychlost*.

Všechny objekty okruhu (asteroidy, góly a loď) jsou kruhy, díky čemuž je test kolize jednoduchý – pokud je vzdálenost středů menší než součet poloměrů, došlo ke kolizi. Kolize navíc kontrolujeme pouze proti finální pozici lodi na konci kroku – zanedbáváme tedy případné kolize, ke kterým by došlo na trase ze staré pozice lodi do nové pozice. V případě nárazu s asteroidem je *loď zpomalena o 50%* a *vysunuta z asteroidu*, v případě nárazu s gólem je gól *označen za splněný*. Okruh je ohraničen obdélníkem, náraz funguje stejně jako asteroid (loď je zpomalena a posunuta). Okruh je dokončen, pokud jsou splněny všechny góly.

Aby pro vás bylo řešení snadnější, závod jsme pro vás implementovali v Pythonu a Rustu.[1] Také je k dispozici webová vizualizace trasy lodi[2] s možností stažení mezikroků simulace pro snadnější debugovaní.

Pro jednoduchost jsme přeskočili určité detaily závodu, například jak přesně fungují nárazy, které se hodí především k implementaci a pro samotné řešení nejsou důležité. Pokud byste si přesto rádi pročetli technickou specifikaci podrobněji, nachází se v repozitáři.

Jelikož i tak je úloha poměrně složitá, předpočítali jsme pro vás navíc pro každou mapu grafy, jejichž vrcholy odpovídají pozicím okolo asteroidů / gólů tak, že následováním cest můžete létat okolo asteroidů / do gólů bez nárazu. K řešení není nutné tyto grafy použít, ale jedná se o dobrý startovní bod, pokud si nevíte rady. Příklad použití, včetně grafů pro všechny mapy úlohy, je v každém z implementačních repozitářů.



Toto je speciální **soutěžní úloha se statickým vstupem** – všichni závodí na stejných mapách a přes Odevzdávátko pak odevzdají řešení s nejmenším počtem instrukcí, které se jim povede najít. Obodování úlohy provedeme až po konci série a to tak, že nejlepší účastnické řešení dostane plný počet bodů a ostatní řešení dostanou body odstupňovaně podle toho, jak byla dobrá oproti nejlepšímu. Zároveň slibujeme, že každé korektní řešení dostane alespoň jeden bod.

V průběhu série se můžete s ostatními porovnávat pomocí průběžné online výsledkovky.[3] Upozorňujeme, že se v ní mohou vyskytnout i řešení od organizátorů.

Formát okruhu: Na prvním řádku dostanete pozici a poloměr lodi x_l y_l r_l, na druhém hranice simulace $x_{\min}$ $y_{\min}$ $x_{\max}$ $y_{\max}$. Na třetím je počet asteroidů n, který je následován n řádky s jejich pozicemi a poloměry x_{ai} y_{ai} r_{ai}. Poté následuje počet gólů m, který je obdobně následován m pozicemi a poloměry x_{ci} y_{ci} r_{ci}.

[1] `https://gitea.ks.matfyz.cz/KSP/asteracer/`
[2] `https://kvaleya.gitlab.io/asteracer/`
[3] `https://ksp.mff.cuni.cz/h/ulohy/37/37-4-4/vysledky`

Formát výstupu: Na prvním řádku vypište počet instrukcí k, které má vaše řešení. Na dalších k řádcích vypište mezerou oddělené instrukce pro loď $dx_i\ dy_i$, po jejichž provedení jsou všechny góly splněny.

Odevzdávátko zkontroluje platnost řešení a přidá jej do průběžné výsledkovky. Upozorňujeme, že od každého řešitele bereme v potaz vždy jeho poslední odevzdané řešení, i kdyby si tím měl zhoršit skóre. Proto vám doporučujeme si svá řešení ukládat, abyste je případně mohli odevzdat znovu.

37-4-X1 Mehrdeutige Wortbildung — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Kevin se učí německy. Jako mnohé před ním ho fascinuje německá slovotvorba. Jak asi víte, v němčině je mnohdy možné vzít dvě nebo více podstatných jmen a spojit je do jednoho podstatného jména. Například spojením slova *Gast* (host) a *Raum* (pokoj) vznikne slovo *Gastraum* (pokoj pro hosty).

Kevinovi se ale zdá, že tento systém není příliš spolehlivý. Nemohlo by se stát, že by nějaké slovo vzniknout více způsoby? Skutečně, slovo *Gastraum* by mohlo vzniknout i složením slov *Gas* (plyn) a *Traum* (sen), a tedy znamenat *plynový sen*! Obdobně (smyšlené) slovo *Baumausgang* bychom mohli rozložit buď jako *Baum-ausgang* (východ ze stromu) nebo jako *Bau-maus-gang* (chodba pro stavební myši). Kevina by zajímalo, jaká další taková víceznačná slova existují. Pomůžete mu?

Napište program, který na vstupu dostane slovník německých podstatných jmen a najde a vypíše jedno libovolné nové slovo, které je možné rozložit na slova ze slovníku dvěma nebo více způsoby, nebo odpoví, že takové spovo neexistuje. Časovou složitost vašeho algoritmu určujte vzhledem k součtu délek všech slov ve slovníku L. Německá abeceda má 30 písmen, toto číslo můžete považovat za konstantu. Můžete předpokládat, že ve slovníku nebude žádné slovo, které je samo o sobě možné rozložit na kratší slova, ani v něm nebude slovo nulové délky.

37-4-S Proměnné se probouzí — 15 bodů

Letošním ročníkem vás bude provázet seriál. V každé sérii se objeví jeden díl, který bude obsahovat nějaké povídání a navíc úkoly. Za úkoly budete moci získávat body podobně jako za klasické úlohy série. Abyste se mohli zapojit i během ročníku, seriál bude možné odevzdávat i po termínu za snížený počet bodů. Vzorové řešení seriálu proto před koncem celého ročníku KSP uvidí jen ti, kdo už seriál odevzdali.

Máme nedokončenou práci. Je na čase dovést náš parser a překlad instrukcí z minulého dílu do jejich finální formy.

Začneme parserem. V minulém díle jsme vytvořili kostru, která dokáže parsovat matematické výrazy a příkaz print. Dnes rozšíříme schopnosti našeho parseru o příkazy if, for a while. Zajímavé je, že parsovat tyto výrazy v dobře navrženém jazyce je velmi jednoduché. Stačí rozšířit naši funkci `statement`. Nyní bude kromě tokenu PRINT kontrolovat i tokeny FOR, IF a WHILE:

```
Expr statement(TokenScanner &ts) {
  if (ts.match(TK_PRINT)) return print_statement(ts);
  if (ts.match(TK_VAR)) return var_statement(ts);
  if (ts.match(TK_FOR)) return for_statement(ts);
  if (ts.match(TK_IF)) return if_statement(ts);
  if (ts.match(TK_WHILE)) return while_statement(ts);
  if (ts.match(TK_LBRACE)) return block(ts);

  // když nic speciálního, prostě parsujeme výraz
  Expr expr = expression(ts);
  ts.consume(TK_SEMICOLON,
             "Za výrazem jsem očekával ';'.");
  return expr;
}
```

Tokenům print, for, if, while a dalším, které uvádějí syntaktický konstrukt, se někdy říká introducer. Jazyky, které introducery pro některé syntaktické konstrukty nemají (např. deklarace proměnných v C), musejí v této funkci více přemýšlet – nebo přesněji, odkládat rozhodnutí na později. Nejhůře je na tom asi C++, které v těchto případech musí provádět sémantickou analýzu nedoparsovaného programu, aby se rozhodlo, jak naparsovat příští token. Takže si važte toho, jak jednoduché to máme. :D

Stačí dopsat chybějící funkce na parsování příkazů. Jako první si ukážeme for, který má v našem jazyce stejný tvar jako v C. Céčkový for má v závorkách tři části – inicializátor, podmínku a inkrementer. Inicializátor kromě výrazů podporuje i definování proměnných (ale ne jiné příkazy) a všechny mohou být prázdné. Pokud dostaneme prázdný inicializátor nebo inkrementer, nahradíme ho v našem Expr stromě prázdným blokem. Prázdnou podmínku nahradíme jedničkou (vždy platnou podmínkou).

```
Expr for_statement(TokenScanner &ts) {
  ts.consume(TK_LPAREN, "expected '(' after 'for'");

  Expr init(ET_BLOCK, std::vector<Expr>{});
  if (ts.match(TK_SEMICOLON)) {
    // prázdný
  } else if (ts.match(TK_VAR)) {
    // definice proměnné (středník kontroluje sám)
    init = var_statement(ts);
  } else {
    init = expression(ts);
    ts.consume(TK_SEMICOLON,
               "expected ';' after the first "
               "expression inside a for statement");
  }

  Expr cond = ts.check(TK_SEMICOLON)
                  ? Expr(ET_LITERAL, "1")
                  : expression(ts);

  ts.consume(TK_SEMICOLON,
             "expected ';' after the second "
             "expression inside a for statement");

  Expr expr = ts.check(TK_RPAREN)
      ? Expr(ET_BLOCK, std::vector<Expr>{})
      ? expression(ts);
  ts.consume(TK_RPAREN,
             "expected ')' after the third "
             "expression inside a for statement");

  Expr body = statement(ts);

  return Expr(ET_FOR, {init, cond, expr, body});
}
```

Vlastně jsme nenapsali nic složitého. Jednoduše vyjmenujeme, které tokeny nebo syntaktické konstrukty jeden po

druhém očekáváme. Narážíme na nejčastěji zmiňovanou výhodu ručně psaných parserů: specifické chybové hlášky. Kde by generovaný parser pouze ohlásil neočekávaný token, my můžeme vypsat obstojné odůvodnění, co přesně se nám nelíbilo a v jakém kontextu se to stalo.

Všimněte si, že `forStatement` používá již existující funkci `expression` z minulého dílu a rekurzivně volá `statement` pro své tělíčko. Jelikož je `statement` rekurzivní, je možné vnořovat for smyčky do for smyček.

Poslední věc, která stojí za zmínku, je, že jsme se rozhodli reprezentovat for smyčky v AST jako čtveřici: inicializační výraz, podmínkový výraz, výraz a samotné tělo smyčky.

Všimněte si, že nikde neříkáme, že tělo smyčky musí být blok kódu. Povolujeme libovolný příkaz, například takto:

```
for (var i = 0; i < 10; i = i + 1) print i;
```

Většinou ale budeme chtít celý blok kódu uvnitř smyčky, takže si ještě ukážeme funkci `block`:

```
Expr block(TokenScanner &ts) {
  std::vector<Expr> statements;

  while (!ts.match(TK_RBRACE) && !ts.isAtEnd()) {
    statements.push_back(statement(ts));
  }

  return Expr(ET_BLOCK, statements);
}
```

Úkol 1 – příkazy while a if [4b]:

Tím by mělo být parsování for smyček hotové a vaším úkolem je teď obdobně doplnit zbývající funkce `whileStatement` a `ifStatement`. Budete se muset rozhodnout, jak je reprezentovat v AST. Nezapomeňte napsat pěkné chybové hlášky.

Příkazy `while` a `if` mají stejné tvary jako v C. Například byste měli být schopni naparsovat tento program:

```
if (a > b) {
  while (b < a)
    print b;

  if (123)
    if (321)
      while (231)
        print 213;
}
```

Převod do instrukcí

Parsování podmínek a cyklů je v porovnání s binárními operátory o poznání jednodušší, ale převod všech těchto konstrukcí na instrukci `OP_BRANCH` bude naopak o něco komplikovanější.

Pro připomenutí, instrukce `OP_BRANCH` má fixní parametr: index instrukce, na kterou má skočit, pokud číslo sebrané ze zásobníku není nula.

Ukážeme si, jak převést if/else, a naprogramovat while a for cyklus bude další úloha. Budeme potřebovat dva skoky – jeden přeskočí příkaz v `if`, pokud podmínka neplatí, a druhý naopak přeskočí příkaz za `else`, když podmínka platí.

Když vyhodnotíme podmínku a rovnou zavoláme instrukci `OP_BRANCH`, dostaneme opačné chování, než chceme – když podmínka platí, přeskočíme blok. Můžeme ale ve výstupním kódu jednoduše prohodit pořadí sekcí `if` a `else`.

Druhý skok umístíme na konec prvního bloku, abychom přeskočili blok druhý. Nepotřebujeme ho ani podmiňovat, protože celý blok se spustí jen tehdy, když podmínka neplatila.

Dohromady tedy chceme vygenerovat instrukce v tomto pořadí: `podmínka`, `OP_BRANCH` na `if(true)` větev, `if(false)` větev, `OP_PUSH 1`, `OP_BRANCH` na konec, `if(true)` větev.

```
void emit_condition(std::vector<Instruction> &program,
                    Expr &condition, Expr &if_true,
                    Expr &if_false) {
  // generujeme něco takového
  // * condition
  // * OP_BRANCH [if_true]
  // * if_false
  // * OP_PUSH 1
  // * OP_BRANCH [end]
  // * if_true
  // * end:
  emit(program, condition);

  auto condition_ix = size(program);
  program.push_back(Instruction{
      .op = OP_BRANCH}); // value přiřadíme později

  emit(program, if_false);

  program.push_back(
      Instruction{.op = OP_PUSH, .value = 1});
  auto endjump_ix = size(program);
  program.push_back(Instruction{.op = OP_BRANCH});

  // sem by měla skočit podmínka, pokud je true
  program[condition_ix].value = ssize(program);
  emit(program, if_true);

  // sem skáče nepodmíněný skok po ‘else‘ bloku
  program[endjump_ix] = ssize(program);
}
```

Kód pro `if` bez `else` větve můžeme jednoduše vygenerovat převedením na předchozí případ, stačí do parametru `if_false` předat prázdný blok. To nás ale přivádí k problému, který jsme zatím nediskutovali – jaké mají být návratové hodnoty příkazů? Uvnitř podmínky můžeme mít buď příkaz (například blok nebo `print x`), nebo výraz (například `a = 1`), a nebylo by vůbec dobré, kdyby jedna větev měla návratovou hodnotu a druhá ne.

Řešení se nabízí několik. Například si můžeme pořídit funkci `is_statement`, která určí, jestli je nějaký `Expr` opravdový výraz nebo příkaz. Podle toho za něj umístíme `OP_POP`, abychom se přebytečné hodnoty zbavili.

Druhá možnost je na to vyzrát tím, že necháme všechny příkazy vracet nějakou hodnotu, díky čemuž se k nim budeme moct chovat úplně stejně jako k výrazům. Většina příkazů může vracet nějakou smysluplnou hodnotu: `print` vrátí svůj argument, blok výsledek posledního příkazu, if výsledek provedené větve, atd. Jen cykly a prázdné bloky žádný intuitivní výsledek nemají, tak je necháme vracet nulu.

Na první pohled se to může zdát jako praštěný hack, ale u jazyků inspirovaných funkcionálním programováním je to celkem oblíbený přístup. Typicky jdou ještě o krok dál, a prostě vás nechají libovolně výrazy a „příkazy“ míchat, jako třeba `var abs = if (x < 0) -x else x`. To ale teď implementovat nemusíte.

My se vydáme touto cestou „zrušení příkazů,“ ale uznáváme libovolné řešení, které nepadá a neleakuje paměť v cyklech ;) Funkci `emit_condition` tím pádem nepotřebujeme měnit, ale zato musíme vyměnit implementaci `emit_block`:

```
void emit_block(std::vector<Instruction> &program,
                std::vector<Expr> &statements) {
  if (statements.empty()) {
```

```
    program.push_back(
        Instruction{.op = OP_PUSH, .value = 0});
  } else {
    for (size_t i = 0; i < size(statements); i++) {
      if (i > 0) {
        program.push_back(
            Instruction{.op = OP_POP, .value = 0});
      }
      emit(program, statements[i]);
    }
  }
}
```

Úkol 2 – **Generování kódu if, while, for** [5b]:

Naprogramujte funkce `emit_while`, `emit_for` a doplňte do funkce `emit` podporu pro `ET_IF`, `ET_WHILE` a `ET_FOR`. Nezapomeňte na to, že `if` může a nemusí mít else větev. Smyčka `for` má v kulatých závorkách tři příkazy: první se provede jednou na začátku, druhý je efektivně to samé jako `while` podmínka a třetí se provede vždy na konci těla cyklu.

Až budete mít naprogramováno, měl by vám seběhnout například následující program:

```
var a = 5;
while ((a = a - 1) >= 0) {
  print a;
}

for (var i = 10; i < 20; i = i + 1) {
  var delitelne2 = i / 2 == (i + 1) / 2;
  if (delitelne2) {
    print i;
  }
}
```

Až na to chybějící modulo ten jazyk začíná vypadat, ehm, jako programovací jazyk! K praktické použitelnosti tomu ještě chybí celkem dost, ale konečně jsme v bodě, kdy si můžeme začít vybírat, co půjdeme kódit dál :)

Úkol 3 – **Freestyle** [6b]:

Přidejte do jazyka podporu, pro co chcete.

Tentokrát nemáme v úmyslu vařit dort. :) Také tentokrát můžete své nápady libovolně diskutovat s ostatními řešiteli. Můžete například přidat vaši oblíbenou Ksplang instrukci jako prvotřídní operátor.

Doporučujeme se nepouštět do obřích projektů; ideálně se nepokoušejte přidat objektový systém. Jako vhodný rozsah nám přijde například přidání logických operátorů `&&` a `||`. Pozor, nejsou to jen další aritmetické operátory – jsou to spíš převlečené podmínky, protože druhý operand se nemá vyhodnotit, pokud je výsledek jasný už z prvního.

Rádi bychom na konci série zveřejnili vaše kompletní řešení; napište nám prosím, zda s tím souhlasíte.

Ještě než tuto epizodu seriálu ukončíme, chtěli bychom vás vyzvat, abyste s námi na Discordu sdíleli svá přání ohledně toho, co by vás potěšilo vidět v příštím, posledním díle.

Spolu s řízením toku jsme dokončili takové překladačové minimum a nyní si můžeme víceméně libovolně vybrat, které odvětví návrhu a implementace programovacích jazyků chceme prozkoumat.

Některé nápady, které nám létají hlavou, jsou: podpora funkcí, polí, řetězců, vláken, sum typů, error handlingu, x86 JIT, integrace s GCC/LLVM.

Prokop Randáček, Standa Lukeš & Ondra Machota

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*