Korespondenční Seminář z Programování

37. ročník KSP Duben 2025

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám páté 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 čtvrté 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: **neděle 15. června 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

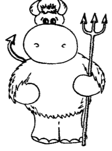
Odměna série: **Odznáček do profilu na webu** si vyslouží ten, kdo z každého dílu seriálu získá alespoň 2 body.

Pátá série třicátého sedmého ročníku KSP

37-5-1 Pekelná cesta — 10 bodů

Čertík Bertík má za úkol strašit spoustu domácností. Každá má svůj krb, mezi kterými se může pekelně přenášet. Kvůli šetření na meziprostorových cestách se ale z každého krbu může přenášet pouze mezi některými dvojicemi krbů – ne nutně mezi všemi. Aby si Bertík krby lépe zapamatoval, přiřadil každému z nich pětici velkých písmen z anglické abecedy. Tím se ale – jako správný čertík – moc chvástal na pekelných úřadech. Ty jsou proslulé svojí pekelností a nepochopitelností. Kvůli tomu mu teď zavedly novou pekelnou direktivu: musí mezi krby cestovat po *lexikograficky nejmenších cestách*. Bertík je z toho celý nesvůj, pomůžete mu?

Konkrétně, *cesta* mezi krby x a y je nějaká posloupnost na sebe navazujících krbů začínající na x a končící na y, kde se žádný krb neopakuje. Představíme-li si všechny možné cesty z x do y, pak ta *lexikograficky nejmenší* z nich je ta, která se, napíšeme-li za sebe názvy jejích krbů bez mezer, v řazení podle slovníku vyskytuje nejdříve.



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 vstupu dostanete neorientovaný graf krbové sítě pomocí hran. Na prvním řádku je počet hran M, na každém dalším dvě pětice velkých písmen oddělených spojovníkem – hrana vede mezi dvěma těmito krby.

Formát výstupu: Vypište, jak se má Bertík dostat z krbu `ALICE` do `BOBER`. To zahrnuje i tyto krby. Mezi krby pište `->`.

Ukázkový vstup:

```
8
ALICE-BOBER
ALICE-ALLAN
ALICE-ALBUS
ALBUS-EIDAM
ALBUS-ALLAN
EIDAM-BOBER
ALBUS-BOBER
ALLAN-BOBER
```

Ukázkový výstup:

```
ALICE->ALBUS->ALLAN->BOBER
```

Vysvětlení ukázkového vstupu: Bertík by se sice mohl z krbu `ALICE` rovnou přesunout do krbu `BOBER`, to by ale jeho cesta nebyla lexikograficky nejmenší, jelikož `ALICEA`... je ve slovníkovém řazení před `ALICEB`... Z podobného důvodu Bertík nemůže použít cestu `ALICE->ALLAN->BOBER`.

37-5-2 Fantomas se zlobí — 11 bodů

Tajným službám se konečně podařilo objevit Fantomasovo doupě – systém tunelů a chodeb kdesi ve francouzských Alpách. Než se do nich ale vydají Fantomase dopadnout a konečně ho dohnat ke spravedlnosti, chtějí si jeskyně pořádně zmapovat.

Fantomasovo doupě se skládá z N místností propojených $N-1$ chodbami, přičemž z každé místnosti se dá dojít do každé jiné. Informatik by řekl, že doupě je (neorientovaný) strom.

Naštěstí pro tajné služby má Fantomas mnoho poskoků a ne všichni se ve spletité síti chodeb orientují. Proto Fantomas nechal do každé místnosti umístit *rozcestník*. Rozcestník je nadepsaný názvem aktuální místnosti a pro každou z $N-1$ ostatních místností je na něm napsáno, kterou chodbou vycházející z aktuální místnosti se máme vydat, abychom došli do dané místnosti.

Tajným službám se podařilo získat všech N rozcestníků. Pomůžete jim zrekonstruovat mapu Fantomasova doupěte?

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 je číslo N – počet místností. Následuje N řádků. Na i-tém z nich je popis i-té místnosti zadaný seznamem N čísel. Konkrétně j-té číslo říká, jakým východem se vydat do místnosti j. (Pokud $i = j$, pak je odpovídající číslo 0.) Východy z každé místnosti mají v nějakém pořadí přiřazena čísla $1, \ldots, d$ kde d je počet chodeb vedoucích z této místnosti.

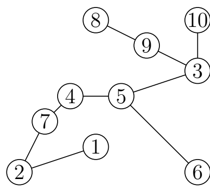
Formát výstupu: Vypište $N-1$ řádků popisujících chodby Fantomasova doupěte. Konkrétně, pro každou chodbu vypište na na vlastní řádek čísla dvou místností, které spojuje, oddělená mezerou. Chodby můžete vypsat v libovolném pořadí. Místnosti číslujeme od jedné.

Ukázkový vstup:

```
10
0 1 1 1 1 1 1 1 1 1
1 0 2 2 2 2 2 2 2 2
2 2 0 2 2 2 2 3 3 1
1 1 2 0 2 2 1 2 2 2
1 1 3 1 0 2 1 3 3 3
1 1 1 1 1 0 1 1 1 1
1 1 2 2 2 2 0 2 2 2
1 1 1 1 1 1 1 0 1 1
1 1 1 1 1 1 1 2 0 1
1 1 1 1 1 1 1 1 1 0
```

Ukázkový výstup:

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



Podíváme-li se na místnost 5 ve vyobrazeném ukázkovém vstupu, můžeme si všimnout, že východ směrem k místnosti 4 dostal číslo 1, východ směrem k místnosti 6 číslo 2, a východ směrem k místnosti 3 dostal číslo 3. Abychom se dostali z místnosti 5 například do místnosti 10, musíme použít východ 3, proto je v pátém řádku na desáté pozici číslo 3.

37-5-3 Ledová socha — 12 bodů

V království hrochů je dnes obrovská slavnost. Celá země se sešla v rozpáleném paláci uprostřed savany, aby zde oslavila sňatek královy dcery Hrochtenzie s princem Robeartem z Království ledních medvědů na dalekém severu. Svatba je vrcholem desetiletí úsilí diplomatů z obou království – všichni očekávají, že spojí dva znepřátelené národy do jednoho impéria táhnoucího se od rovníku až k pólu, a jednou provždy ukončí zbytečné války a podezíravost, která mezi obyvateli království po staletí panovala.

A ještě víc než kdy jindy teď záleží na detailech – každá zpackaná drobnost by mohla být chybně interpretována jako urážka a způsobit okamžité rozbroje přímo na místě. A to už nikdo nechce zažít. Opravdu.

Jak to ale bývá, chybičky se občas vloudí, a tak se dodávka nadživotní ledové sochy zachycující ve všech detailech nadpozemskou krásu hroší princezny – mimochodem, z hlediska výroby mimořádně nákladného svatebního daru – opozdila. Dokonce až tak moc, že všechny stoly a židle (s netrpělivě čekajícími hrochy a ledními medvědy) jsou již rozestavěny na mřížce paláce a náklaďák se sochou se přes ně nemůže dostat k podstavci, kde by socha měla stát.

A to je větší problém, než se zdá. Podstavec je speciálně chlazený a stíněný tak, aby na něm socha vydržela v původním stavu. Jenže od momentu, kdy ji vyloží z bezpečí náklaďáku, budou všechny její detaily vydány napospas spalujícímu polednímu slunci. Sochu je tedy nutné dostat na podstavec co nejdříve.

A to nebude snadné. Za prvé, socha je, jak jste si patrně všimli, z ledu. Takže klouže. Hodně. A taky je opravdu těžká (tady se poznámky, z úcty k princezně, zdržíme). S vypětím všech sil je ji možné roztlačit na určitou rychlost. Pak si všichni zainteresovaní musejí na nějaký čas oddechnout, zatímco socha vesele klouže dál konstantní rychlostí. Pak je možné sochu znovu více zrychlit, nebo zpomalit, ale opět jenom o stejnou hodnotu rychlosti (aplikací stejné síly).

Konkrétně to vypadá tak, že v každém *kroku* je možné sochu zrychlit nebo zpomalit o hodnotu rychlosti *jedno pole za krok* v horizontálním nebo vertikálním směru (nebo v daném kroku rychlost neměnit). Socha se pak posune o tolik polí, jaká je její aktuální rychlost. Pak následuje další krok. Jelikož je socha velice křehká, nesmí se stát, že by po cestě narazila do jakékoliv překážky. Pro změnu směru pohybu je nutno sochu nejdříve zastavit (jinak by hrozilo, že se začne nekontrolovatelně točit, a to nemůže dopadnout dobře) – tedy pokud se socha pohybuje směrem nahoru rychlostí 3, ale chceme ji posunout směrem doleva, musíme jí nejdříve třikrát zpomalit (přičemž se stihne poklouznout nejdřív o dvě a pak ještě o jedno pole) a až poté jí můžeme potlačit doleva.

Všechny zraky se upírají na vás. Máte k dispozici plánek paláce (rozdělený na volná políčka a políčka s překážkami), aktuální souřadnice sochy a souřadnice podstavce, kde musí socha skončit (s nulovou rychlostí, samozřejmě). Odpovězte takovou sekvencí zrychlení a zpomalení, která jí tam dostane v co nejmenším počtu kroků.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

37-5-4 Užitečné mnohočleny 12 bodů

Kevin si letos koledováním přišel na rekordní množství vajíček, která následně výhodně prodal. Za utržené peníze si pořídil novou kalkulačku – ten nejdražší a nejmodernější model se spoustou různých funkcí.

Kevina z nich nejvíce zaujala funkce rychlého násobení mnohočlenů s celočíselnými koeficienty.

Připomeňme, že násobení mnohočlenů je definováno následovně:

Máme-li dva mnohočleny:

$A(x) = a_0 + a_1x + \ldots + a_mx^m$

$B(x) = b_0 + b_1x + \ldots + b_kx^k$

pak jejich součin:

$C(x) = A(x) \cdot B(x)$

je opět mnohočlen, jehož koeficienty c_i jsou dány vztahem:

$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$

kde chápeme $a_j = 0$ pro $j > m$ a $b_{i-j} = 0$ pro $i - j > k$.

Kevina při čtení návodu překvapilo, že umí kalkulačka nalézt součin rychleji, než to umí klasický školní algoritmus.

Napadlo jej proto, že by mohlo jít kalkulačku využít k řešení několika problémů, nad kterými si již dlouho láme hlavu.

To bude vaším úkolem. Na konci zadání naleznete čtyři různé problémy a budete mít za úkol k nim najít efektivní řešení.

K tomu můžete využít Kevinovu kalkulačku. Tu má váš algoritmus k dispozici jako funkci, která na vstupu přijímá dva mnohočleny s celočíselnými koeficienty reprezentované jako seznamy jejich koeficientů (od nultého koeficientu po poslední nenulový), a vrací jejich součin ve stejné reprezentaci. Tuto funkci můžete ve vašem algoritmu zavolat libovolněkrát.

Při určování výsledné časové složitosti můžete předpokládat, že má tato funkce časovou složitost $\mathcal{O}(N \log N)$, kde N je součet délek obou vstupních seznamů koeficientů. Koeficienty v seznamech zadaných kalkulačce by měly být polynomiálně velké k velikosti vstupu celého algoritmu.

Nyní již k samotným problémům. Ty jsou následující:

Úkol 1 – Překryv jedniček [2b]:

Máme dva binární textové řetězce (složené z nul a jedniček) s celkovou délkou N. Na počátku leží tyto řetězce naproti sobě a jejich začátky jsou zarovnány. Cílem je jeden z řetězců posunout o libovolný počet znaků doleva či doprava vůči druhému tak, aby naproti sobě leželo co nejvíce jedniček. Znaky, které po posunu neleží naproti jinému znaku, ignorujeme.

Například v případě řetězců `10110` a `001010` je optimální posun takový, že třetí znak druhého řetězce leží naproti prvnímu znaku prvního řetězce. V tomto uspořádání jsou naproti sobě dva páry jedniček.

Úkol 2 – Nejlepší shoda [3b]:

Jsou dány dva binární textové řetězce s celkovou délkou N, jeden delší (označme jej seno) a jeden kratší (jehla). Najděte takový podřetězec sena délky rovné jehle, který má s jehlou nejvíce shodných znaků na odpovídajících pozicích.

Například pro seno `00100100` a jehlu `1011` je nejlepší shoda s podřetězcem začínajícím na 3. pozici, který se od jehly liší v jediném znaku.

Úkol 3 – Výskyt s otazníky [3b]:

Opět máme seno a jehlu s celkovou délkou N, přičemž seno je binární řetězec. Jehla však kromě nul a jedniček může obsahovat i otazníky, které reprezentují libovolný znak – nulu i jedničku. Různé otazníky mohou být nahrazeny různými znaky. Vaším úkolem je najít všechny výskyty jehly v seně.

Například pro seno `110011110` a jehlu `?11?1` se jehla v seně vyskytuje pouze jednou, a to od 4. pozice.

Úkol 4 – Regulární výrazy [4b]:

Tentokrát řešíme obdobný problém, jako v předchozím úkolu, ale seno i jehla jsou tvořeny znaky z obecné abecedy, nikoliv jen z nul a jedniček. Jehla může opět obsahovat otazníky, které zastupují libovolný znak této abecedy.

Mějme například seno `abcbbcbc` a jehlu `?bc`. Správným řešením je vypsat pozice 1, 4 a 6. Všimněte si, že se výskyty částečně překrývají.

Váš algoritmus by měl být na plný počet bodů efektivní i v případě, že má abeceda velikost $\mathcal{O}(N)$.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

37-4-X1 Mehrdeutige Wortbildung 10 bodů

Protože se nikomu nepodařilo X-kovou úlohu z čtvrté série vyřešit, rozhodli jsme se, že prodloužíme její deadline do konce páté série. Navíc vydáváme následující nápovědu, která vám snad pomůže k řešení.

U této úlohy vám pomůže známá nápověda: Sestrojte si vhodný graf. Na řetězce se pak můžete dívat jako na procházky v tomto grafu.

37-5-S Vzestup zdrojového kódu 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.

V posledním díle seriálu nejen prozkoumáme funkce, ale hlavně vás necháme naimplementovat druhý datový typ do našeho překladače.

Funkce

Pro funkce jsme si vybrali následující syntaxi:

```
fn main(a, b) {
        print(a + b)
        return a * b
}
```

Pamětníci jistě vzpomenou na jeden z minulých dílů, ve kterém jsme vysvětlovali, že všechny syntaktické struktury by měly začínat klíčovým slovem, aby je bylo jednoduché rozpoznávat. Funkce v tomto ohledu nejsou výjimkou.

Jedním z novodobých vynálezů je povolování extra čárky za posledním argumentem. Pokud má nějaká funkce mnoho argumentů, můžeme se rozhodnout je rozřezat na jednotlivé řádky a čárka za posledním argumentem pak umožňuje jednoduché prohazování argumentů:

```
fn main(
    a,
    b, # Můžu prohazovat!
    c,
    d, # všechny řádky stejné!
) {
        print(a + b)
        return a * b
}
```

Pokud se vám tato syntaxe nelíbí, můžete si vymyslet vlastní :D Rádi si přečteme vaše výmysly. Kromě kódu nám ale dodejte i popis, jak vaše řešení funguje.

Pro implementaci naší syntaxe budeme potřebovat přidat 3 nové tokeny: `FN`, `RETURN` a `COMMA`. Mnohem zajímavější bude úprava gramatiky a parseru. I v parseru nám přibude několik nových typů:

`ET_ARG_LIST` je seznam výrazů oddělených čárkami. Ten bude sloužit buď jako argumenty při volání funkcí, nebo jako argumenty u deklarace funkcí. V druhém případě navíc vyžadujeme, aby výrazy v seznamu byla pouze jména.

`ET_CALL` je další nový typ, který označuje volání funkce. Jeho první prvek je jméno funkce, kterou voláme, druhý prvek je seznam argumentů jako `ET_ARG_LIST`.

Dále nadefinujeme typ `ET_FN`, který popisuje definici funkce jako trojici: jméno (`ET_NAME`), argumenty (`ET_ARG_LIST`) a tělo (`ET_BLOCK`).

Poslední nový typ je `ET_FUNCTION_LIST`, což bude nový kořen našeho AST, který obsahuje seznam všech definovaných funkcí. Nově budeme totiž vyžadovat, aby všechen kód byl uvnitř nějaké funkce. Toho můžeme nejjednodušeji docílit tím, že přepíšeme funkci `parse`:

```
Expr parse(TokenScanner &ts) {
  std::vector<Expr> functions;
  while (!ts.isAtEnd()) {
    functions.push_back(function(ts));
  }
  return Expr(ET_FUNCTION_LIST, functions);
}
```

Funkce `parse` nyní místo seznamu příkazů vrací seznam funkcí. Další na řadě je funkce `function`:

```
static Expr function(TokenScanner &ts) {
  ts.consume(TK_FN,
      "expected fn keyword to start a function");

  auto token = ts.peek();
  ts.consume(TK_NAME,
      "expected function name after 'fn'");

  auto name = Expr(ET_NAME, token.value);

  ts.consume(TK_LPAREN,
      "expected '(' after function name");

  auto args = arguments(ts);

  ts.consume(TK_RPAREN,
      "expected ')' after function arguments");

  ts.consume(TK_LBRACE,
      "expected '\{' to start function body");
  auto body = block(ts);

  return Expr(ET_FN, std::vector{name, args, body});
}
```

Měla by vám připomínat parser smyček a ifů. Jsou si všechny velmi podobné. Z parseru funkce se eventuálně dovoláme do funkce `block`, která konečně začne parsovat jednotlivé příkazy pomocí existujícího potrubí.

Úkol 1 – Instalatéřina [4b]:

Upravte zbytek potrubí parseru, aby přijímalo nový výraz volání funkcí a příkaz vracení z funkce. Příkaz `return` vypadá stejně jako příkaz `print`. Bude mít pouze jiné chování uvnitř virtuálního stroje.

Volání funkcí je složitější. Vypadá takto: `foo(a, b, c)`. Podporujeme zbytečnou čárku za posledním argumentem. Argumenty mohou být libovolné výrazy. Výraz volání může být součástí větších výrazů. Detaily ale necháváme na vás. Pokud stojíte o nápovědu, můžete si připomenout, jak vaše implementace parsuje binární operátory. Ty totiž vlastně řeší podobný problém: „Naparsoval jsem nějaký výraz a kontroluji, jestli ihned za ním není napsaný tento konkrétní token, který se mi líbí.“

Další na řadě je převod AST do bytekódu. Spolu s tím budeme muset změnit, co pro nás znamená „program“. Program byla do teď sekvence instrukcí. S funkcemi se nám bude hodit dívat se na program jako na množinu funkcí. Funkce má jméno, jména argumentů a instrukce.

```
struct Function {
  std::vector<std::string> args;
  std::vector<Instruction> code;
};
```

Jméno funkce nebudeme mít uloženo přímo v této struktuře. Místo toho si budeme udržovat mapu ze jmen na tyto struktury, což je právě ten typ, který bude nově vracet funkce `emit_program`:

```
std::map<std::string, Function> emit_program(Expr &expr) {
  auto program = std::map<std::string, Function>{};
  assert(expr.type == ET_FUNCTION_LIST);

  for (auto function : expr.children) {
    assert(function.type == ET_FN);
    auto function_name = function.children.at(0);
    auto function_arguments = function.children.at(1);
    auto function_body = function.children.at(2);

    assert(function_name.type == ET_NAME);
    assert(function_arguments.type == ET_ARG_LIST);
    assert(function_body.type == ET_BLOCK);

    auto function_instructions = std::vector<Instruction>();
    emit(function_instructions, function_body);

    std::vector<std::string> args;
    for (auto const &arg_expr : function_arguments.children) {
      args.push_back(arg_expr.value);
    }

    program[function_name.value] =
        Function{.args = args, .code = function_instructions};
  }

  return program;
}
```

Kontrolovat typy vrcholů v první vrstvě AST je trochu zbytečné a možná vás napadlo, že bychom mohli podobně upravit náš pohled na AST. Místo jednoho AST bychom měli množinu AST, kde každé AST by mělo vlastní jméno. Pokud vás toto napadlo, dobrá práce! :D Rozhodli jsme se zachovat jedno AST, jelikož to zjednoduší potenciální přechod na podporu funkcí uvnitř funkcí, k čemuž se bohužel v dnešním díle nedostaneme.

`emit_program` volá funkci `emit` kterou už máme z minulých dílů. Do té musíme přidat podporu pro emitování volání funkcí a vracení z funkcí. Emitování vracení z funkcí je jednodušší – funguje totiž i tady úplně stejně jako příkaz print, jednoduše převedeme příkaz na odpovídající `OP_RET`. Emitování volání funkcí je o něco málo zajímavější. Musíme rekurzivně emitnout všechny argumenty funkce a poté instrukci `OP_CALL`.

```
void emit_call(
 std::vector<Instruction> &program,
 Expr const &expr
) {
  assert(expr.type == ET_CALL);

  auto name = expr.children.at(0);
  auto args = expr.children.at(1);

  assert(name.type == ET_NAME);
  assert(args.type == ET_ARG_LIST);

  for (auto const &arg : args.children) {
    emit(program, arg);
  }

  program.push_back(
    Instruction{.op = OP_CALL, .value = name.value}
  );
}
```

S touto změnou nám už zbývá jen samotný virtuální stroj. Ten teď bude muset počítat s novou definicí programu:

```
int interpret(
  std::map<std::string, Function> const &program,
  std::string name,
  std::vector<int> &zasobnik,
  std::unordered_map<std::string, int> &promenne);
```

Funkce `interpret` nově přijímá všechny funkce uvnitř programu a jméno funkce, kterou má spustit. Dále, jako dříve, zásobník a lokální proměnné. Opět bude příkaz `return` jednodušší, takže s ním začneme:

```
case OP_RET: {
  assert(!empty(zasobnik));
  return zasobnik.back();
} break;
```

Příkaz `return` vrací hodnotu z celé funkce `interpret`.

Úkol 2 – instrukce OP_CALL [4b]:

Naimplementujte instrukci `OP_CALL`. Ta má uvnitř uloženo jméno funkce, kterou volá, a kolik hodnot ze zásobníku má použít. Stačí tedy sebrat tyto hodnoty a předat je rekurzivnímu volání funkci `interpret` jako lokální proměnné. Hodnotu, kterou vrátí funkce `interpret`, pak instrukce `OP_CALL` strčí na zásobník.

Pospojujte vše dohromady a ozkoušjte jak to funguje. Například můžete naprogramovat výpočet faktoriálu pomocí rekurze ;)

Datové typy

Poslední úkol letošního seriálu bude velmi kreativní. Úkolem bude přidat do vaší implementace podporu pro datový typ schopný reprezentovat libovolné množství dat. Jediný datový typ, se kterým jsme do teď počítali, byla čísla, která jsou v tomto ohledu celkem omezující. Proto bychom chtěli přidat podporu pro buď stringy nebo pole. Necháme na vás, které si vyberete.

Úkol 3 – Stringy [7b]:

Podpora pro stringy má několik částí, které zasahují do všech fází překladu. Určitě potřebujeme umět vyrobit stringový literál a uložit ho do proměnné. Necháme na vás si rozmyslet, jak se mají chovat všechny aktuální operátory, když dostanou nějakou kombinaci stringů a čísel.

Spousta jazyků přetěžuje existující operátory s chováním pro stringy a kombinace stringů a čísel. Například Python umí násobit string číslem nebo v JavaScriptu jde sčítat dva stringy normálním operátorem `+`. Jak se ale pak má chovat například operátor `+`, pokud dostane na jedné straně string a na druhé číslo?

Lua má elegantní řešení. Pro sčítání čísel se používá operátor `+` a pro konkatenaci stringů se používá operátor `..`. To řeší všechny problémy s podivným chováním operátoru `+` pro různé kombinace intů a stringů. `+` stále funguje i když jsou oba nebo jeden operand string. V takovém případě se je pokusí převést na čísla, protože přeci provádíte sčítání čísel. Operátor `..` funguje analogicky, ale pro stringy.

Jak upravíte existující operátory nebo jestli přidáte nové, necháme na vás. I když se ale rozhodnete, že nebudete podporovat některý operátor s nějakou kombinací typů operandů, vypište alespoň chybovou hlášku do konzole.

Všude, kde jsme do teď ve VM používali `int`, budete teď muset přidat nějaký union type. V C++ doporučujeme `std::variant<int, std::string>`.

Stringy určitě také potřebujeme umět vypisovat pomocí příkazu print. Můžete také vyrobit pomocné funkce pro manipulaci stringů. Třeba funkce `startswith`, `endswith`, `contains`, atd. Necháme na vás, jestli tyto funkce budou implementované jako instrukce virtuálního stroje nebo jako funkce v samotném jazyce.

Úkol 4 – Pole [7b]:

Pokud se vám nechce podporovat druhý datový typ, nebo vám stringy přijdou příliš nudné, máme alternativní řešení! Změníme datový typ na pole a skalární čísla prostě zrušíme. Pole přeci umí být i jednoprvkové.

Většina operací, které náš jazyk umí, dávají na polích perfektní smysl: sčítání je prostě sečte po prvcích jako vektory, porovnávání bude fungovat jako filtr a podmínky se budou rozhodovat podle toho, jestli nám ten filtrovací operátor vrátil alespoň jeden prvek, nebo vrátil prázdné pole.

Zní to praštěně, a také trochu je (viz ukázkové použití níže). Nicméně je to funkční způsob, jak si pořídit jazyk, ve kterém lze efektivně napsat víceméně libovolný algoritmus, který znáte.

Do parseru nám stačí přidat syntax pro:

1. Výrobu pole: `[x, y, z]`
2. Indexování pole: `x[i]`.
3. Zjištění velikosti pole: například zavedeme předdefinovanou funkci `length(x)`, která vrátí počet prvků jako jednoprvkové pole :)

Především ale musíme upravit sémantiku jazyka, aby všechny vlastnosti jazyka fungovaly s hodnotami typu pole.

Číselný literál vrací jednoprvkové pole. Například `1` je ekvivalentní `[1]`. Příkaz `print` vypíše všechny prvky pole. Výraz „výroby pole“ `[x, y, z]` ve skutečnosti dělá slepení polí: `x`, `y` i `z` už pole jsou, takže je spojíme do nového, většího pole.

Aritmetické binární operátory se aplikují po prvcích a vracejí pole o stejné velikosti jako vstup. Pokud dostaneme dvě stejně velká pole, tak je to jasné: `[1, 2, 3] + [1, 1, 0]` se rovná `[2, 3, 3]`. Chtěli bychom ale, aby fungovala i notace `[1, 2, 3] + 1`, kde jedničku přičteme ke všem prvkům. Zavedeme tedy pravidlo, že pokud je jeden z operandů jednoprvkový, tak tento prvek spárujeme se všemi prvky druhého pole. Tomu budeme říkat broadcasting.

Jako boolean budeme místo 0 a 1 používat prázdná a neprázdná pole. `if (a) { ... }` se provede, pokud je pole `a` neprázdné.

Srovnávací operátory se také budou aplikovat po prvcích a podporovat broadcasting. To znamená, že místo jedniček a nul budou vracet pole indexů, kde podmínka platí. Například `1 > 2` vrátí prázdné pole, `[10, 20] > 10` vrátí `[2]`. Pravděpodobně budete muset přidat instrukce pro všechny druhy porovnávání, tentokrát je negacemi nevyřešíme přes `OP_LT`.

Indexování pole `x[i]` se také musí vypořádat s tím, že `i` je pole. Uděláme to tak, že pro každý index z `i` vybereme prvek z `x` a zkonstruujeme z nich nové pole o stejné velikosti jako `i`. Například `[7, 4][[2, 1, 1, 2]]` se bude rovnat `[4, 7, 7, 4]`. Opět se inspirujeme Luou a budeme indexovat od 1. Je to ale jedno, můžete si to změnit ;), jen si pak upravte testovací program.

Přiřazení by mělo podporovat indexování pole jako levý výraz (včetně podpory indexování polem), budete na to muset přidat další instrukci. Můžete podporovat i broadcasting na hodnoty (`a[[1, 2, 3]] = 0`).

Sami si rozmyslete:

1. Jestli chcete pole předávat hodnotou nebo referencí.
2. Jak se má chovat negace (`OP_NOT`).
3. Ve kterých okrajových případech vypisovat chybu a které nějak implicitně pořešit.

Implementace je na vás. Nebojte se ale říci si o radu pokud něco nebude jasné nebo se v něčem ztratíte. Pro C++ programátory přikládáme obvyklou radu: kompilujte program s `-fsanitize=address,undefined`.

Abyste měli na čem vyzkoušet funkčnost, přikládáme pár ukázkových funkcí, které by váš interpretr měl zvládnout. Můžete si je i poupravit, pokud nějaké chování implementujete trochu jinak než náš kód předpokládá (např. indexování od 1). Své úpravy zadání jen prosím zdokumentujte, ať se v tom úplně neztratíme.

```
# alokuje pole velikosti n s iniciální hodnotou v
fn alloc(v, n) {
  var res = v;
  while (length(res) < n) {
    res = [ res, res ];
  }
  var index = res == res;
  return res[index <= n];
}
# absolutní hodnota
fn abs(val) {
  val = [ val ];
  val[val < 0] = -val[val < 0];
  return val;
}
# vrátí pole čísel od from do to včetně
# (pokud indexujete od nuly tak nezahrnujte konec)
fn range(from, to) {
  var len = abs(from - to) + 1;
  var inc = length(from < to) - length(from >= to);

  var res = alloc(0, len);
  var index = 1;
  var current = from;
  while (index <= len) {
    res[index] = current;
    index = index + 1;
    current = current + inc;
  }
  return res;
}
fn sort(arr) {
  if (length(arr) <= 1) return arr;
  var pivot = arr[length(arr) / 2 + 1];
  return [
    sort(arr[arr < pivot]),
    arr[arr == pivot],
    sort(arr[arr > pivot])
  ];
}
fn uniq(a) {
  if (!a) return [];

  var ix = range(1, length(a) - 1);
  var dedup = a[a[ix] != a[ix + 1]];
  return [ dedup, a[length(a)] ];
}
fn union(a, b) {
  return uniq(sort([a, b]));
}
fn intersect(a, b) {
  var c = sort([ uniq(sort(a)), uniq(sort(b)) ]);
  var ix = range(1, length(c) - 1);
  var dup = c[ix] == c[ix + 1];
  return c[dup];
}
```

Kde se dozvědět víc?

Pokud se vám letošní téma seriálu zalíbilo a chtěli byste se o překladačích a programovacích jazycích dozvědět více, máme pro vás pár tipů na dobré zdroje.

Především bychom chtěli doporučit knihu Crafting Interpreters od Roberta Nystroma. Při psaní seriálu jsme se touto knihou inspirovali. Příklady vám určitě budou připadat povědomé.

Kvalita moderního překladače se většinou odvozuje od kvality optimalizací, které dokáže provádět. Optimalizace jsme z našeho seriálu pro zjednodušení zcela vynechali. Jde o fázi překladu, která stojí mezi parsováním a generováním strojového kódu. Většinou využívá speciální datové struktury pro reprezentaci kódu a moderní překladače zde tráví většinu času. Pokud by vás zajímalo, jaké optimalizace moderní produkční překladač provádí, doporučujeme moc pěknou prezentaci o LLVM pro začátečníky od Chandlera Carrutha.[1]

Několik z vás nám psalo, že byste rádi zkusili naprogramovat překlad do strojového kódu. Bohužel jsme se tomu nakonec nevěnovali – rozsahově to přesahuje možnosti jednoho dílu seriálu.

Jedním z nejlepších způsobů, jak to udělat, je využít projekt LLVM. Program by se přeložil do LLVM kódu, na kterém by LLVM provedlo optimalizace a vygenerovalo efektivní strojový kód.

Naopak jeden z nejjednodušších způsobů je udělat totéž, ale s použitím knihovny QBE,[2] která je vhodnější pro začátečníky, jako jsme my.

Také vám určitě doporučujeme prozkoumat fungování překladačů a interpretů vašich oblíbených jazyků. Kromě teoretických znalostí o kompilátorech získáte i spoustu praktických informací o daném jazyce a často odhalíte, proč určité aspekty fungují tak, jak fungují. Wren, Lua a Umka jsou interpretované jazyky s jednoduchou implementací, které bychom vám pro tyto účely doporučili.

Prokop Randáček, Standa Lukeš & Ondra Machota

[1] https://youtu.be/FnGCDLhaxKU
[2] https://c9x.me/compile

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/*