Korespondenční Seminář z Programování

37. ročník **KSP** **Listopad 2024**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám druhé 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. prosince 2024 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 vysloužíte, kdo **ho najde v HTTP bludišti**.

Druhá série třicátého sedmého ročníku KSP

37-2-1 Urychlovač částic — 10 bodů

Sára připravuje demonstraci na den otevřených dveří v urychlovači částic. Ráda by návštěvníkům předvedla, jak je možné v urychlovači jeden prvek přeměnit v druhý. Sára během svého výzkumu našla N funkčních nastavení urychlovače, které pro přehlednost očíslovala od 1 do N. Každé z nich dokáže z jednoho specifického prvku vyrobit jiný specifický prvek.

Při přeměně urychlovač různě svítí, bliká a hučí. Sára proto každé nastavení ohodnotila kladným, celým číslem, které nazývá *úžasnost*. Úžasnost jednoduše vyjadřuje, jak moc se bude daná přeměna návštěvníkům líbit. Návštěvníci nemají až tak dobrou paměť, nevadí jim jednu přeměnu vidět vícekrát.

Během demonstrace bude Sára mít čas na přesně K přeměn. Navíc nebude čas na to, aby v urychlovači vyměňovala vzorky, takže na sebe přeměny budou muset navazovat. Na začátku demonstrace do urychlovače načerpá vodík, kterého mají v ústavu hodně. Chtěla by, aby na konec v urychlovači zbylo hélium, které dětem plánuje napustit do balónků.

Ráda by přeměny naplánovala tak, aby součet jejich úžasnosti byl co největší. Pomůžete jí?

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 dostane tři čísla P, N a K: protonové číslo nejtěžšího prvku, se kterým umí urychlovač pracovat, počet nastavení urychlovače a počet přeměn, na které má Sára čas. Na každém z dalších N řádků bude popsáno jedno nastavení pomocí tří čísel a_i, v_i a u_i: protonové číslo prvku, který musí být v urychlovači před přeměnou, protonové číslo prvku, který v něm zůstane po přeměně, a úžasnost této přeměny.

Vodík má protonové číslo 1, hélium 2.

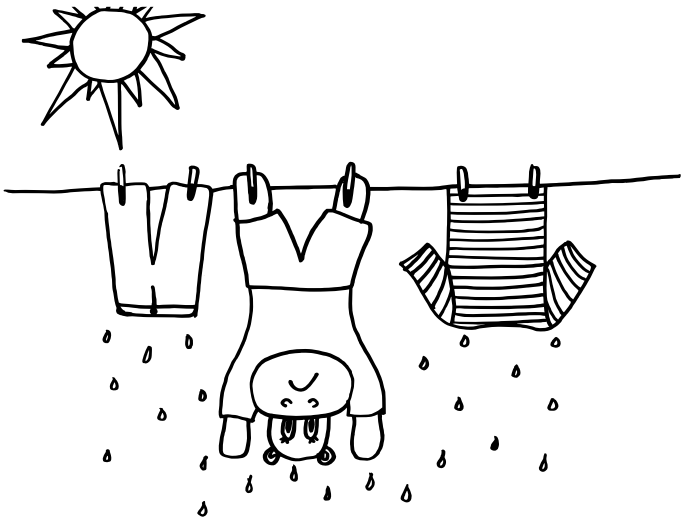
Formát výstupu: Vypište K řádků, na i-tém číslo i-tého použitého nastavení urychlovače.

Ukázkový vstup:	*Ukázkový výstup:*

`3 4 2`
`1 2 100`
`1 3 10`
`3 2 20`
`3 2 50`
	`2`
	`4`

Vysvětlení ukázkového vstupu: První nastavení sice vypadá lákavě, ale po jeho provedení bychom už nemohli provést žádnou druhou přeměnu. Proto nám nezbývá než nejprve vyrobit lithium pomocí druhého nastavení. Udělat z lithia helium umí dvě různá nastavení, nejvyšší úžasnost má to s číslem čtyři.

37-2-2 Pohoří plné jezer — 11 bodů



Kevin má spousty snů a jeden z nich je objevit všechny divy světa. Tak si počkal na prodloužený víkend a naplánoval si výlet na bájné Jezerní pohoří. Jezerní pohoří je známé pro svůj nekončící déšť, který se v slunečním svitu leskne jako perličky. Ale hlavní půvab této oblasti jsou všechna nejrůznější údolí, ve kterých se tvoří jezera.

Bylo potřeba spoustu příprav. Kevin si s sebou vzal pláštěnku, deštník a ještě k tomu nepromokavé oblečení a po dlouhé túře tam konečně dorazil. Bohužel si Kevin nepřečetl dodatečnou informaci, že všechna jezera vysychají mimo sezónu, takže místo překrásných jezer měl před sebou jen nejrůznější prázdná údolí. Kevin si tohle nenechal líbit a rozhodl se, že naplní všechna jezera sám, a aby z toho měl pořádný zážitek, tak je všechny naplní na maximum.

Pohoří si můžeme představit jako mřížku $R \times S$, kde každé políčko má zadanou výšku. Naším úkolem je zjistit, kolik nejvíce vody může celá krajina zadržet. (Jednotku vody můžeme změřit jako zatopení políčka na hladinu o jedna vyšší než jeho výška). Vlivem unikátního reliéfu teče voda pouze směry nahoru, dolů, doleva a doprava. Z krajních políček steče všechna voda do nížin a nedrží se tam. Tedy voda se udrží v nějakém políčku pouze v případě, kdy nedokáže nějakou cestou přetéct přes okraj pohoří.

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 čísla R a S: počet řádků a počet sloupců v mřížce pohoří. Na každém z dalších R řádků bude S čísel oddělených mezerou označujících výšky políček. Výška políčka je nezáporné číslo, které se vejde do 32-bitového integeru.

Formát výstupu: Výstupem je jediné číslo: počet jednotek vody, které jsou potřeba, aby se naplnila všechny jezera. Pozor, výsledek se nemusí vejít do 32-bitového integeru.

Ukázkový vstup:

```
5 5
0 3 6 6 7
2 2 0 1 3
3 6 6 5 7
6 3 0 7 4
5 6 8 3 5
```

Ukázkový výstup:

```
12
```

[1] `https://xkcd.com/571/`

Vysvětlení ukázkového vstupu: Ve vstupu si můžeme všimnout dvou oblastí, ve kterých nevyteče voda z pohoří ven. První z nich se nachází na pozicích $(1,2)$ a $(1,3)$, kde jsou hodnoty nula a jedna. Výška hladiny této oblasti bude dvě, protože jakékoliv větší množství vody přeteče přes políčka $(1,1)$ a $(1,0)$ mimo pohoří. Podobnou úvahu provedeme pro oblast na pozicích $(3,1)$ a $(3,2)$, které je obklopené políčky, které přetečou až nad výškou šest. Ve finále jsme pro naplnění potřebovali dvanáct jednotek vody $(0 \to 2, 1 \to 2, 3 \to 6, 0 \to 6)$.

Pro lepší představu tu máme obrázek zpracovaného vstupu. Šedá políčka znázorňují políčka, která zůstala bez vody a modrá políčka mají vlevo nahoře jejich původní výšku a vpravo dole hladinu vody, kterou udrží.

0	3	6	6	7
2	2	0 2	1 2	3
3	6	6	5	7
6	3 6	0 6	7	4
5	6	8	3	5

37-2-3 Chov ovcí — 12 bodů

Říša se rozhodl skončit s programováním a dát se na chov ovcí. Přestěhoval se na Balkán, koupil si kus půdy a začal zkoumat, jak se ovce chovají. Na důvěryhodném zdroji[1] se dozvěděl, že nejlepší je ovcím vyhradit čtvercový výběh s K keři. Koupil si ale celkem rozlehlý kus půdy, a tak se ptá vás, jestli byste mu správný výběh našli?

Pro N bodů v rovině najděte čtverec o velikosti $A \times A$ zarovnaný s osami, ve kterém je právě K bodů. (Nebo řekněte, že neexistuje.) Můžete předpokládat, že $K \ll N$ (K je řádově menší než N).

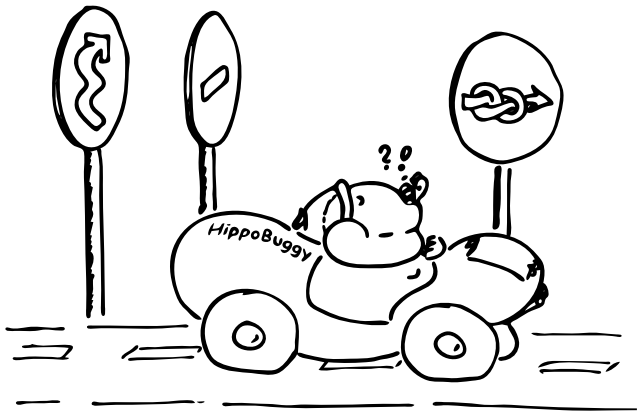
Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

37-2-4 HTTP bludiště — 12 bodů

Vítejte v bludišti! To naše se ukrývá uvnitř webového serveru. Komunikujete s ním po síti protokolem HTTP, jehož popis najdete v kuchařce této série. Vaším cílem je najít *klíč* – to je nějaký 16-znakový řetězec.

Bludiště má několik částí, které se chovají jako testy open-data úlohy. Když si necháte vygenerovat vstup, dozvíte se URL vchodu do bludiště. Jako výstup pak odevzdáte klíč. Na nalezení klíče máte hodinu, pak si musíte vygenerovat nový vchod.

Pečlivě sledujte, co vám server posílá, a zkoumejte i slepé uličky. Třeba se v nich dozvíte něco, co se bude později hodit.



37-2-X1 Intervalové většiny — 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 v poslední době začal zajímat o kompresi obrázků a dokonce si již navrhl svůj vlastní algoritmus.

Je však aktuálně příliš pomalý. V jeho průběhu totiž Kevin potřebuje opakovaně pro různé úseky obrázku rychle zjistit, zda v něm má nějaká barva nadpoloviční většinu a to Kevin neumí lépe, než průchodem celého úseku.

Hodila by se mu proto nějaká datová struktura, která by zvládla pro zadaný úsek obrázku rychle odpovědět, zda v něm má nějaká barva nadpoloviční většinu a případnou barvu vrátit.

To bude úkol pro vás.

Na vstupu dostanete obrázek. Ten má pro jednoduchost jen jeden řádek a lze si jej představit jako posloupnost barev. Každá taková barva je v posloupnosti reprezentována jako celé číslo. Upozorňujeme, že čísla barev mohou být poměrně vysoká, takže nemusí být například dobrý nápad s nimi indexovat pole.

Váš program nejprve na základě této posloupnosti postaví potřebnou datovou strukturu. Následně s její pomocí začne vyřizovat příchozí dotazy na jednotlivé úseky posloupnosti. Nelze předpokládat, že je program obdrží najednou, takže musí být schopen na dotazy odpovídat online.

Tím nicméně Kevinovy požadavky nekončí. Je nezbytné, aby výsledná datová struktura vyřizovala jednotlivé dotazy v čase $\mathcal{O}(1)$. Vaše řešení tak budeme posuzovat výhradně podle času stráveného předvýpočtem. Ten by se měl v ideálním případě vejít do $\mathcal{O}(N \log^c N)$, kde N je délka posloupnosti a c nějaká konstanta.

37-2-S Lexery útočí — 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.

Lexery útočí

V dnešním díle se vrhneme na sestrojení první analýzy zdrojového kódu – konkrétně lexikální analýzy. Lexikální analýza se provádí v části překladače, které se říká lexer nebo scanner. Jejím úkolem je vzít text, který napsal programátor jako pole znaků, a pospojovat znaky, které spolu patří, do skupinek nazývaných tokeny. Jeden token může reprezentovat nějaké slovo jazyka nebo interpunkci.

Lexer je relativně jednoduchá část překladače. V dnešní době existují šikovné nástroje, které dokážou vygenerovat kvalitní lexer z deklarativního popisu jazyka, ale cílem našeho seriálu je ukázat si, jak tyto věci uvnitř fungují, takže si lexer napíšeme sami.

Příklad tokenu může být `PLUS` pro operátor `+`, `FOR` pro klíčkové slovo `for` nebo `GREATER_EQUAL` pro operátor `>=`. Některé tokeny jsou ale složitější. Například budeme mít token pro jména proměnných nebo pro číselné konstanty použité v programu. Oba tyto typy tokenů si budou muset pamatovat znaky, ze kterých jsme je vytvořili, abychom později dokázali rekonstruovat hodnotu čísla nebo jméno proměnné.

Typy tokenů si zapíšeme jako `enum`:

```
enum TokenType {
  // Operátory
  TK_NOT, TK_EQUAL, TK_GREATER, TK_LESS, TK_MINUS,
  TK_PLUS, TK_SEMICOLON, TK_SLASH, TK_STAR,
  TK_NOT_EQUAL, TK_EQUAL_EQUAL, TK_GREATER_EQUAL,
  TK_LESS_EQUAL, TK_LBRACE, TK_LPAREN, TK_RBRACE,
  TK_RPAREN,

  // Klíčová slova
  TK_ELSE, TK_FOR, TK_IF, TK_PRINT, TK_VAR,
  TK_WHILE,

  // literály
  TK_NAME, TK_NUMBER,
};
```

Tokeny `TK_NAME` a `TK_NUMBER` si zároveň musí pamatovat znaky, ze kterých jsme tento token vytvořili. Proto tento enum zabalíme do malého `structu`.

```
struct Token {
  TokenType type;

  // pouze u TK_NAME a TK_NUMBER, jinak ""
  std::string value;

  Token(TokenType t, std::string v = "")
      : type{t}, value{v} {}
};
```

Tokeny chceme umět vypisovat do konzole, takže si hned napíšeme jednoduchou funkci, která převede `TokenType` na `std::string`:

```
std::string token_type_to_str(TokenType t) {
  switch (t) {
  case TK_ELSE: return "ELSE";
```

```
  case TK_EQUAL: return "EQ";
  case TK_EQUAL_EQUAL: return "EQUAL_EQUAL";
  ...
  case TK_WHILE: return "WHILE";
  }
  return "<invalid token value>";
}
```

Jeden tokenizovaný řádek bychom pak mohli takto vypsat:

```
var a  =  33+2;
```

```
VAR NAME(a) EQ NUMBER(33) PLUS NUMBER(2) SEMICOLON
```

Všimněte si, že pole tokenů nedrží informace o mezerách mezi tokeny.

Dospělé lexery si zachovávají více informací, jako třeba lokaci ve zdrojovém textu, ze které každý token pochází. To se pak používá pro generování přesnějších chybových hlášek, ale přichází se zajímavým problémem: lokace zdroje každého tokenu nafoukne velikost každého tokenu, i když se zdrojová lokace každého tokenu jistě nepoužije. Řešení je například ukládat si lokace pouze každého N-tého tokenu a až při generování chybových hlášek znovu tokenizovat zdrojový kód pro spočítání přesné lokace potřebného tokenu. To sice znamená, že musíme nějaké informace zbytečně počítat znovu, ale ušetřená paměť nám celkový proces zrychlí.

Pro lexování se nám bude hodit pomocná struktura `Scanner`. Obsahuje jednoduchou abstrakci nad procházením pole znaků. Nejdůležitější je funkce `match`, která otestuje, jestli je daný znak první na vstupu, a pokud ano, přesune se přes něj a vrátí `true`, jinak vrátí `false`.

```
#include <cctype>
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Scanner {
  std::string source;

  Scanner(std::string s) : source{s} {}

  void advance(size_t i = 1) { source.erase(0, i); }
  bool is_at_end() { return source.empty(); }

  char peek() {
    return source.empty() ? '\0' : source[0];
  }

  bool match(char c) {
    if (peek() == c) {
      advance();
      return true;
    }
    return false;
  }
};

std::vector<Token> lex(std::string source) {
  Scanner sc = Scanner(source);
  std::vector<Token> ts;

  while (true) {
    auto t = lex_one(sc);
    if (!t.has_value()) { break; }
    ts.push_back(t.value());
  }

  return ts;
}

int main() {
  std::string source = "var a  =  33+2;";
  std::vector<Token> ts = lex(source);
  for (auto t : ts) {
    std::cout << token_type_to_str(t.type) << "("
              << t.value << ") ";
  }
  std::cout << '\n';
}
```

Tohle je další běžný tvar programu, který člověk najde v překladačích. Máme (budeme mít) funkci `lex_one`, která se podívá na prvních několik znaků vstupu, které jsme ještě nepřečetli, a usoudí, jaký token se na tomto místě nachází. Tento token nám vrátí a přesune se ve vstupu přes tento token. Pokud chceme tokenizovat celý vstup, stačí nám opakovaně volat tuto funkci, dokud nám neoznámí, že jsme dorazili na konec souboru.

Klasický design překladače volá tuto funkci líně. Až když je příští krok v překladu plně hotový s procesováním aktuálního tokenu, si překladač řekne o další token, který opět ihned předhodí dál a čeká, dokud se plně nezpracuje. Tohle rozhodnutí plynulo z omezení tehdejších počítačů, které měly problém načíst celý zdrojový kód do paměti! To ale rozhodně není naše situace, takže si vyrobíme pole pro naše tokeny a všechny si je tam vesele nastrkáme. Zpátky k `lex_one`.

První věc, kterou musí `lex_one` udělat, je přeskočit bílé znaky jako třeba mezery, tabulátory a zalomení řádku:

```
std::optional<Token> lex_one(Scanner &sc) {
  // přeskoč whitespace
  while (std::isspace(sc.peek())) {
    sc.advance();
  }
  ...?
}
```

Dalším úkolem je rozpoznat 3 skupiny tokenů: operátory (`+`, `{`, ...), čísla a jména. Nejjednodušší je rozpoznávat operátory, takže s tím i začneme. Vyrobíme si na ně vlastní funkci:

```
std::optional<Token>
match_operator_token(Scanner &sc) {
```

Pro rozpoznávání jednoznakových operátorů stačí použít funkci `match`:

```
  if (sc.match('(')) return TK_LPAREN;
```

Dvouznakové operátory jsou trochu složitější. Když na začátku vstupu vidíme znak `<`, nestačí pouze oznámit `TK_LESS` a posunout se dál. Může se totiž stát, že hned další znak je `=` a správný token je `TK_LESS_EQUAL`.

```
  if (sc.match('<')) {
    if (sc.match('=')) {
      return TK_LESS_EQUAL;
    } else {
      return TK_LESS;
    }
  }
```

Pokud žádný operátor nenajdeme, vrátíme z funkce null:

```
  return std::nullopt;
}
```

Úkol 1 – Operátory [3b]:

Váš první úkol je dopsat funkci `match_operator_token`. Můžete použít naše kousky kódu, které jsme vám právě ukázali.

Po tomto úkolu stačí zavolat funkci `match_operator_token` z `lex_one`, a pokud nevrátí `std::nullopt`, tak máme hotovo a vrátíme tento token. Jinak se musíme zeptat další pomocné funkce, což bude `match_digit_token`.

Funkce `match_digit_token` funguje velmi podobně jako funkce `match_operator_token`. Důležitá vlastnost všech těchto pomocných funkcí je, že nekonzumují žádný vstup, dokud si nejsou jisté, že na vstupu je skutečně token, který rozpoznávají.

Úkol 2 – **Čísla** [4b]:

Druhý úkol spočívá v napsání funkce `match_digit_token`. Ta by měla zkontrolovat, zda na začátku vstupu je číslo, a pokud ano, celé ho přeskočit. Zároveň si ale musí toto číslo uložit uvnitř tokenu, který pak vrátí. Později budeme potřebovat převést toto číslo na jeho hodnotu. Nebudeme řešit `+` ani `-` před číslem. Pro `-` máme (budeme mít) unární mínus operátor, a unární plus operátor nebudeme potřebovat.

Pokud ani `match_digit_token` nic nevrátí, pak poslední možnost je, že na vstupu je jméno nebo klíčové slovo. Na ty si pořídíme další funkci `match_keyword_or_name_token`.

Úkol 3 – **Jména a klíčová slova** [4b]:

Váš třetí úkol je funkce `match_keyword_or_name_token`. Jejím úkolem je rozpoznávat jména. Konkrétně nepřerušované sekvence malých a velkých písmen abecedy nebo číslic, které nezačínají číslicí. Stačí nám, když funkce bude pracovat s ASCII, ale pokud se cítíte odvážně, můžete podporovat i UTF-8.

Funkce si musí všimnout, pokud je na vstupu jedno z klíčových slov (`else`, `for`, `if`, `print`, `var`, `while`), a v takovém případě místo tokenu `NAME` vrátit token tohoto klíčového slova.

Po dokončení prvních tří úkolů by finální funkce `lex_one` měla umět rozpoznávat operátory, čísla, jména a klíčová slova a měla by vypadat přibližně takto:

```
std::optional<Token> lex_one(Scanner &sc) {
  // přeskoč whitespace
  while (std::isspace(sc.peek())) {
    sc.advance();
  }

  auto o = match_operator_token(sc);
  if (o.has_value()) { return o; }

  auto d = match_digit_token(sc);
  if (d.has_value()) { return d; }

  auto k = match_keyword_or_name_token(sc);
  if (k.has_value()) { return k; }

  // jinak jsme museli narazit na konec
  assert(sc.is_at_end());
  return std::nullopt;
}
```

Co jsme zatím úplně vynechali je ošetřování chyb. Náš lexer si vůbec neporadí s nevalidním vstupem, což je trochu smutné, jelikož kód je většinu času v nevalidním stavu, protože ho programátor zrovna píše a očekává chytré napovídání od svého editoru. Produkční překladače obsahují algoritmy, které se snaží šikovně ignorovat nesmyslné znaky a domýšlet si chybějící strukturu programu, aby mohly generovat nápovědy nebo podtrhávat sémantické chyby. Generované lexery jsou v této oblasti mnohem horší než ručně psané lexery, takže v praxi se člověk stále setká se spoustou ručně psaných lexerů.

Nám bude stačit velice jednoduché ošetřování chyb. Pokud náš lexer narazí na něco, čemu nerozumí, vypíše chybu a ukončí překlad. Pokud bychom ale pouze vypsali `Chyba.` a ukončili překlad, tak by z toho programátor byl velice smutný. Proto mu prozradíme, na který znak vstupu se překladač koukal, když vše přestalo dávat smysl.

Úkol 4 – **Chyby a lokace tokenů** [4b]:

Váš poslední úkol je naimplementovat vypisování pěkných zpráv při nesmyslném vstupu včetně řádku a znaku. Budete muset rozšířit `Scanner`, aby si pamatoval aktuální řádek a sloupec. Když už tuto informaci budete mít, uložte ji do každého tokenu. Bude se nám hodit pro podobně pěkné oznamování chyb v pozdějších fázích překladu. Po vypsání chybové hlášky z lexeru program ukončete.

Všechny 4 úkoly odevzdejte najednou jako jeden ZIP soubor obsahující všechen váš zdrojový kód. Oceníme i slovní popis zvoleného řešení, bohatě by měly stačit komentáře v kódu. Když budete mít jakýkoliv dotaz nebo problém, tak se nebojte zeptat na našem Discordu či pomocí emailu.

Prokop Randáček, Standa Lukeš & Ondra Machota

Recepty z programátorské kuchařky: Protokol HTTP

HTTP (Hypertext Transfer Protocol) je jedním z nejpoužívanějších síťových protokolů. Původně byl vymyšlen pro stahování webových stránek a odesílání webových formulářů, ale postupně si našel spoustu jiných aplikací. Povídají si po něm nejrůznější síťová API, nebo třeba servery a klienti chatovacího protokolu Matrix.

HTTP existuje ve více verzích. My si budeme povídat o nejběžnější verzi 1.1. Už existují i verze 2 a 3, které fungují podobně, ale snaží se o vyšší efektivitu, což je komplikuje.

Po HTTP si povídají dvě strany: *server* a *klient.* Server poskytuje nějakou službu (třeba webové stránky), klienti tuto službu využívají. Když se klient chce připojit k serveru, nejdříve si pomocí DNS přeloží jméno serveru na IP adresu. Pak s ní naváže spojení přes TCP typicky na portu 80. TCP spojení si můžeme představit jako obousměrnou trubku mezi serverem a klientem, která umí spolehlivě přenášet bajty – když do ní na jednom konci nějaké vložíme, za nějakou dobu vypadnou z druhého konce.

Pokud nám záleží na bezpečnosti, chceme data šifrovat a podepisovat – o to se typicky stará protokol TLS, který z TCP udělá „bezpečnou trubku.“ Skrz ni pak proudí obyčejné HTTP. Této kombinaci se říká HTTPS a místo portu 80 bydlí obvykle na 443.

HTTP je protokol košatý, takže se naše kuchařka zaměří hlavně na základy. Kdyby vás zajímaly detaily, najdete je ve standardech RFC 9110 a RFC 9112.

Objekty a URL

HTTP popisuje operace na nějakých *objektech* (resources). V nejjednodušším případě může být objektem soubor na disku serveru a operací „chci stáhnout obsah souboru.“ Nebo je objektem meteostanice na střeše budovy a operací „chci naměřené hodnoty ve formátu JSON.“

Síťové protokoly se obecně na objekty odkazují pomocí jejich URL (Uniform Resource Locator). Typické URL pro HTTP vypadá takto:

```
http://ksp.mff.cuni.cz/h/ulohy/37/reseni1.html
```

Skládá se ze *schématu* (názvu protokolu) `http:`, za ním následuje `//`*adresa serveru* (IP adresa nebo doménové jméno) a případně `:`*port,* pokud máme použít jiný TCP port než běžný 80. Zbytek URL je tvořen *cestou* v rámci serveru. Na další, méně obvyklé součástky URL narazíme později.

Někdy se stane, že do URL potřebujeme zapsat znak, který by jinak měl speciální význam – třeba mezeru. V takovém případě ho zakódujeme jako `%`*xy,* kde *xy* je hexadecimální kód znaku v ASCII. Takže mezera je `%20` a procento `%25`. Nechce-li se nám studovat v RFC 3986, které znaky jsou speciální, kódujme vše kromě písmen, číslic a tečky. a `-` `_` . `~`. Ještě dodejme, že občas potkáváme i zkratku URI (Uniform Resource Identifier), což je obecnější pojem, ale v kontextu HTTP mezi nimi není potřeba rozlišovat.

Požadavek a odpověď

Podívejme se, co se děje, když si klient chce stáhnout stránku z uvedeného URL. Spojí se po TCP s `ksp.mff.cuni.cz` na portu 80 a pošle požadavek:

```
GET /h/ulohy/37/reseni1.html HTTP/1.1
Host: ksp.mff.cuni.cz
Connection: close
```

Požadavek má tvar několika řádků textu (pozor, řádky se ukončují CR+LF). První řádek obsahuje *metodu* `GET` (to je ta operace, která se má s daným objektem provést – v tomto případě stáhnout si jeho obsah), *cestu* k objektu v rámci serveru a *verzi* HTTP, kterou se bavíme.

Následující řádky tvoří *hlavičku* požadavku složenou z *polí* tvaru *klíč*: *hodnota.* Pole `Host` je povinné a obsahuje doménové jméno serveru (posílá se proto, aby na jedné IP adrese mohly běžet servery pro více domén). Jelikož chceme být slušní, tak pomocí `Connection: close` řekneme serveru, že po tomto spojení už nechceme posílat další požadavky.

Požadavek je ukončen prázdným řádkem.

Server pak odpoví třeba takto:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 38771
Connection: close
```

Na prvním řádku nám server sděluje, že také hovoří verzí 1.1 a že operace byla provedena úspěšně. To říká jednak stavovým kódem 200 (určeným pro stroje), jednak zprávou `OK` určenou pro lidi.

Následující řádky obsahují hlavičku odpovědi:

- `Content-Type` – typ dat, která server posílá. Jedná se o text ve formátu HTML, tedy webovou stránku. A znaky má zakódované v UTF-8 (jelikož software, kterým se na odpověď díváme, nejspíš také používá UTF-8, nemusíme se o kódování starat).
- `Content-Length` – délka dat v bajtech.
- `Connection: close` – server potvrzuje, že po této odpovědi spojení zavře.

Hlavičku opět ukončuje prázdný řádek. Za ním následuje *tělo* odpovědi – data, která si stahujeme.

Metody

Náš příklad používá metodu `GET`, ale existují i jiné. Pojďme se podívat na nejběžnější metody. Požadavky s metodou `GET` a `HEAD` nemají žádné tělo, s ostatními metodami ho mít mohou.

- `GET` – vyžádá si data daného objektu (přesněji řečeno *reprezentaci* objektu v nějakém formátu; časem uvidíme, že jich může být na výběr víc). Pokud webovému prohlížeči zadáme URL stránky, stáhne si ji touto metodou. Reprezentace může být soubor na disku, ale často ji server vytváří v okamžiku vyřizování požadavku.
- `HEAD` – jako `GET`, ale vyžádá si od serveru jen hlavičku odpovědi bez těla. To se hodí, pokud nás zajímají informace o objektu, aniž bychom si ho chtěli celý stáhnout.
- `POST` – řekne objektu, ať zpracuje nějaká data. Může to být třeba odeslání webového formuláře (viz dále), vytištění něčeho na tiskárně nebo přihlášení uživatele.
- `PUT` – nahrazuje už existující objekt novým obsahem.
- `PATCH` – mění jen některé části daného objektu.
- `DELETE` – odstraňuje daný objekt.

Kromě nich ještě existují `CONNECT`, `OPTIONS` a `TRACE`, které mají poměrně specifické použití a nebudeme se jimi zabývat.

Metody `GET` a `HEAD` nesmí měnit stav objektu na serveru – prohlížeč se tedy může sám od sebe rozhodnout načíst na pozadí stránku, na níž někde viděl odkaz, a nic tím nezkazí. Takovým metodám se říká *bezpečné.*

Metody `PUT` a `DELETE` sice mění stav, ale jsou *idempotentní* – pošleme-li stejný požadavek vícekrát, dopadne to stejně jako jednou. (Všimněte si, že smazat nebo nahradit něco dvakrát má stejný efekt, jako to udělat jednou.) Pokud tedy klient zjistí, že se mu spojení se serverem rozpadlo, může `GET`, `HEAD`, `DELETE` nebo `PUT` automaticky zopakovat; s `POST`em by to ovšem dělat neměl (aby vám neobjednal tři krabice zmrzliny místo jedné).

Metoda `PATCH` nemusí být ani bezpečná, ani idempotentní.

Data požadavku/odpovědi

Jak už víme, požadavek i odpověď mohou obsahovat tělo s daty. Jak se pozná, kde data končí? Jestliže dopředu víme, jak jsou data velká, pošleme `Content-Length` a problém je vyřešen. Pokud to ale nevíme, pomůžeme si *nakouskováním* dat. Do hlavičky přidáme `Transfer-Encoding: chunked` a tělo pošleme jako posloupnost *kousků.* Každý kousek vypadá takto: délka dat zapsaná hexadecimálně, CR+LF, data kousku, CR+LF. Poslední kousek má délku 0.

Historické verze HTTP po skončení těla zavíraly celé TCP spojení. To bylo zbytečně neefektivní, protože pro stažení každého objektu se muselo navázat spojení nové. Dnes lze poslat po stejném spojení další požadavek a přijmout další odpověď, dokud se jedna ze stran nerozhodne, že chce spojení ukončit. Pokud chceme poslat jen jeden požadavek, dá se toto chování předem zakázat pomocí `Connection: close`, jak už jsme viděli výše.

Formát dat je specifikován v hlavičce požadavku/odpovědi:

- `Content-Type` jsme již potkali. Typů dat je mnoho a definuje je standard MIME, proto se jim říká MIME-typy. Běžný je třeba `text/plain` pro čistý text, `text/html` pro webovou stránku, `image/jpeg` a `image/png` pro obrázky, `application/zip` pro ZIPové archivy atd. Za typem mohou být další parametry oddělené středníkem: pro textové typy je to třeba `charset`.

  Podle `Content-Type` se prohlížeče rozhodují, jestli data zobrazí (případně jak), nebo je nabídnou ke stažení.
- `Content-Encoding` popisuje transformaci dat, nejčastěji kompresi, která se stala ještě před odesláním dat. To znamená, že třeba `Content-Length` už bude udávat velikost po zkomprimování, ale `Content-Type` se nezmění. Tedy například zkomprimovaná webová stránka může mít `Content-Type: text/html` a `Content-Encoding: gzip`. V kontrastu s tím dříve zmíněný `Transfer-Encoding` popisuje transformaci, která se stala až při přenosu. Klienti se obvykle k obojímu chovají stejně: před dalším zpracováním data dekódují.
- `Content-Language` říká, jakým (lidským) jazykem je dokument napsán, přesněji řečeno pro mluvčí jakého jazyka je určen. Kupříkladu `cs` pro text psaný česky, `cs, en` pro dvojjazyčné vydání knížky, ale učebnice arabštiny psaná anglicky by měla jen `en`, ač obsahuje i arabské věty.

Kromě `Content-Type` nejsou tato pole povinná.

Další pole hlavičky

Uvedeme ještě pár zajímavých polí hlavičky, všechna jsou nepovinná. V požadavcích:

- `User-Agent` – software klienta (`produkt/verze`, případně více oddělených čárkou; může obsahovat komentáře v závorkách). Hodí se, aby servery mohly obcházet chyby v konkrétních klientech. Ale také útočníkům, aby poznali, kdo má nezabezpečeného klienta :)
- `Referer` – pokud požadavek vznikl následováním nějakého odkazu (třeba klikacího na webové stránce), můžeme uvést URL, kde se odkaz nacházel. To se hodí správcům serveru k ulovení chybných odkazů.
- `Cookie` – server může pomocí pole `Set-Cookie` v odpovědi předat klientovi „sušenku“. To je libovolný řetězec, který si klient zapamatuje a pak ho posílá v hlavičkách dalších požadavků. Cookies se používají pro udržování stavu webových aplikací (třeba přihlášení uživatele nebo nákupní košík v obchodě). Jsou specifikované v RFC 2965, my si odpustíme jak detaily, tak úvahy o důsledcích pro soukromí uživatelů.

A v odpovědích:

- `Server` – software serveru (formát jako `User-Agent`).

Ještě dodejme, že ve jménech polí se nerozlišují velká a malá písmena – psát velká počáteční je jenom (dobrý) zvyk. Kromě standardních polí si každý může přidat svá vlastní, jejichž jména začínají na `X-`. A lidé někdy nepořádně říkají „hlavička“ jednomu poli a „hlavičky“ množině všech polí.

Stavové kódy

Jak už víme, server začíná odpověď řádkem se stavovým kódem. Pojďme se podívat, co kódy znamenají. Uvádíme obvyklé názvy stavů, server ovšem může být kreativní. Méně běžné kódy vynecháváme.

- `1xx` – operace stále probíhá (to nenastane v žádné ze situací, které popisujeme).
- `2xx` – operace byla provedena úspěšně:
  - `200 OK` – nic zvláštního se nestalo
  - `201 Created` – operací (typicky `PUT`em) vznikl nový objekt, `Location` v hlavičce říká jeho URL.
  - `202 Accepted` – operace byla přijata ke zpracování a provádí se na pozadí. HTTP nedefinuje, jak se časem dozvědět výsledek.
  - `204 No Content` – operace se provedla, ale výsledek nemá tělo (častá reakce na `PUT`).
  - `206 Partial Content` – viz intervalové dotazy níže.
- `3xx` – přesměrování: místo uvedeného URL se máme obrátit na jiné, typicky uvedené v poli `Location`. Druhů přesměrování je vícero:
  - `300 Multiple Choices` – server nabízí víc reprezentací objektu (třeba obrázky v různých formátech). Tělo obsahuje odkazy na URL jednotlivých reprezentací (typicky jako HTML), `Location` říká, kterou reprezentaci server preferuje.
  - `301 Moved Permanently` – objekt se přesunul na jiné URL, zkuste to tam a zapamatujte si nové umístění. Prohlížeče z historických důvodů po přesměrování mění metodu `POST` na `GET`, ačkoliv to standard zakazoval.
  - `302 Found` – objekt dočasně najdete na jiném URL, ale příště zase zkuste původní URL. Prohlížeče opět mění `POST` na `GET`.

  - `303 See Other` – operace nemá výstup, ale uživateli chceme ukázat nějaká související data na jiném URL (z nějž se provede `GET`).
  - `304 Not Modified` – viz podmíněné dotazy níže.
  - `307 Temporary Redirect` – obdoba `302`, která nemění metody.
  - `308 Permanent Redirect` – obdoba `301`, která nemění metody.
- `4xx` – chyba klienta: operaci nebylo možné provést z důvodů na straně klienta:
  - `400 Bad Request` – kdykoliv chybu nejde popsat přesnějším kódem.
  - `401 Unauthorized` – viz autentikace níže.
  - `403 Forbidden` – tuto operaci nemáte právo provést.
  - `404 Not Found` – URL ukazuje na neexistující objekt.
  - `405 Method Not Allowed` – objekt nepodporuje použitou metodu; seznam podporovaných metod najdete v poli `Allow`.
  - `406 Not Acceptable` – objekt nemá požadovanou reprezentaci, viz domlouvání na formátu dat níže.
  - `410 Gone` – URL ukazuje na objekt, který už neexistuje, ač dříve existoval.
  - `411 Length Required` – server vyžaduje, abyste uvedli délku těla.
  - `413 Content Too Large` – tělo je moc dlouhé.
  - `414 URI Too Long` – URL je moc dlouhé.
  - `415 Unsupported Media Type` – poslali jste `Content-Type`, kterému server nerozumí.
  - `416 Range Not Satisfiable` – viz intervalové dotazy níže.
  - `418 I'm a Teapot` – jeden dávný aprílový vtip :)
  - `422 Unprocessable Content` – typu dat server rozumí, ale obsahu dat už ne (to by mohla být reakce na `PUT` nebo `POST` syntakticky chybného souboru).
- `5xx` – chyba serveru:
  - `500 Internal Server Error` – kdykoliv chybu nejde popsat přesnějším kódem.
  - `501 Not Implemented` – požádali jste o něco, co server vůbec nepodporuje. Třeba úplně neznámou metodu.
  - `503 Service Unavailable` – server momentálně neobsluhuje požadavky. Možná je vypnutý kvůli údržbě, možná jen přetížený. `Retry-After` může říci, za jak dlouho to máte zkusit znovu.
  - `505 HTTP Version Not Supported` – danou verzí protokolu server nehovoří.

Všechny stavové kódy mohou být doprovázeny tělem. Často to bývá chybová zpráva v HTML, hezky zformátovaná pro pohodlí uživatele.

Použití

Dotazy a předávání argumentů

`GET` nemusí nutně vracet celý dlouhý dokument. Server nad ním může umět provádět *dotazy* – například v seznamu vyučujících na webu školy najít položku s konkrétním jménem a příjmením.

Dotaz se specifikuje přidáním argumentů na konec URL (za cestu) ve tvaru `?jmeno=Tim&prijmeni=Berners-Lee`. Dotaz začíná otazníkem a pak následují dvojice *klíč*`=`*hodnota* oddělené ampersandy. Pokud hodnoty obsahují nějaké „divné“ znaky, kódují se už známým `%xy`; místo mezery můžeme poslat `+`.

Jak přesně přítomnost dotazu ovlivní operaci, je definované serverem. Stejně jako co se stane, když dotaz přidáme k jiné metodě než `GET`.

Webové formuláře

Součástí webové stránky může být formulář (`<form>`). Ten nechá uživatele zadat data do pojmenovaných políček a pak tato data odešle serveru na určené URL buď metodou `GET`, nebo `POST`. Výběr metody také určí, jakým způsobem se data formuláře odešlou.

Metoda `GET` je jednodušší. Klient využije syntaxi URL s dotazem a jako argumenty vyplní data formuláře. To se hodí třeba pro vyhledávací okénka nebo stránkování dlouhých seznamů. Server na `GET` z daného URL může poslat dynamicky generovanou odpověď (třeba podle zadaného vyhledávacího dotazu). Pozor na to, že `GET` je z definice bezpečná metoda, takže takto odesílané formuláře nesmí měnit stav světa.

Pokud odeslání formuláře má měnit stav (třeba někomu poslat zprávu), nebo pokud je dat tolik, že se do URL prakticky nevejdou, použijeme metodu `POST`. Tou pošleme tělo s `Content-Type: application/x-www-form-urlencoded` obsahující data zakódovaná stejně jako dotaz na konci URL. (Pokud by byl součástí formuláře upload souboru, posílá se jiný formát `multipart/form-data`, který nebudeme rozebírat.)

Autentikace

Pokud chcete po HTTP řídit svou vzducholoď, jistě nestojíte o to, aby jí mohl posílat příkazy každý. HTTP nabízí *autentikační* mechanismy (kterými může klient dokázat, kdo je) a servery pak na jejich základě klienty *autorizují* k provedení konkrétní operace. (Přestože HTTP takové věci umí, dnešní weby si obvykle přihlašování řeší samy přes formuláře a kryptograficky podepsané cookies. Ale u různých API je autentikace na úrovni HTTP naprosto běžná.)

Když se klient pokusí přistoupit na stránku vyžadující autorizaci, dostane stav `401 Unauthorized` a v hlavičce něco jako `WWW-Authenticate: Basic realm="Hrochovo"`. Z toho se dozvěděl, že má použít autentikační metodu jménem `Basic` v „říši“ `Hrochovo` (říše je nápověda pro uživatele, jaký druh účtu potřebuje).

Požadavek pak pošle znovu a do hlavičky přidá něco jako `Authorization: Basic aHJvY2g6aHVtcGY=`. Tutéž hlavičku pak posílá s dalšími požadavky na totéž URL a často i na URL „pod ním v hierarchii“.

Konkrétní podoba autorizačního řetězce je definovaná autentikační metodou. Pro `Basic` se řídíme RFC 7617, které nám řekne, že máme vzít řetězec `jmeno:heslo` a zakódovat ho do Base64. To není šifra, ale usnadní to přenášení divných znaků a také si nedopatřením nepřečtete cizí heslo. (V praxi samozřejmě chcete jakákoliv citlivá data přenášet přes HTTPS místo obyčejného HTTP.)

Server může klientovi nabídnout víc autentikačních metod (oddělených čárkou), klient si z nich jednu vybere.

Syntaxe URL dovoluje uvést jméno a heslo, která se mají použít, jako `http://`*jméno*`:`*heslo*`@`*doména*`/...` Jelikož aplikace málokdy předpokládají, že URL je citlivý údaj, nebývá to moudré používat.

Optimalizace

Domlouvání na reprezentaci dat

Server může data poskytovat ve více reprezentacích, které se hodí v různých situacích. Například u obrázku může nabízet běžný JPEG, pak JPEG XL (který je menší, ale zatím ho málokdo zná) a PNG (bezztrátové, takže kvalitnější, leč mnohem větší). Jak server pozná, o kterou reprezentaci má klient zájem?

Jedna možnost je využít stavový kód `300 Multiple Choices` a poslat klientovi seznam všech variant. Klient, který si vybírat neumí, si prostě z `Location` přečte default a následuje ho, jako u každého jiného přesměrování. Každý výběr varianty nás ale stojí poslání dalšího požadavku. A navíc jsou varianty popsané odkazy v HTML, takže se těžko zpracovávají strojově.

Dnes je běžnější, že klient pošle serveru své preference a server podle toho sám vybere vhodnou reprezentaci. Preference na MIME-typ dat můžeme poslat třeba takto:

```
Accept: image/jxl, image/*; q=0.5, */*; q=0.1
```

Nabízíme několik variant oddělených čárkami. Varianty můžeme ohodnotit *kvalitou* mezi 0 a 1 (s přesností na tisíciny). Neuvedená kvalita je rovna 1. Uvedeme-li `q=0`, explicitně říkáme, že o takový typ nestojíme. Náš příklad tedy říká, že preferujeme JPEG XL, spokojíme se s libovolným jiným obrázkovým formátem, a když není ani ten, přežijeme s čímkoliv.

Server nám pak pošle tu variantu, které vyjde nejvyšší kvalita, případně chybu `406 Not Acceptable`, pokud žádná dostupná varianta nesplňuje naše požadavky.

Podobně existují pole `Accept-Charset`, `Accept-Encoding` a `Accept-Language` pro domlouvání se na dalších vlastnostech reprezentace dat. Pokud preference neudáme, server si zvolí po svém.

Kromě toho má HTTP hlavičku požadavku s lehce obskurním názvem `TE`, která uvádí klientem podporované hodnoty `Transfer-Encoding`. Ty se ale nehodnotí kvalitou.

Server tedy může pro jedno URL posílat různým klientům různý obsah podle toho, jakou hlavičku požadavku pošlou. Aby bylo jasné, že se něco takového děje, server do odpovědi připíše pole `Vary`, kde vyjmenuje všechna pole hlavičky požadavku, kterými se řídila volba reprezentace. Třeba `Vary: Accept, Accept-Encoding`.

Podmíněné operace

Představte si, že máme stažený ohromný soubor a chceme zjistit, zda se mezitím na serveru změnil. K tomu můžeme použít metodu `HEAD`, jež nám v poli `Last-Modified` řekne datum a čas poslední modifikace souboru (pokud reprezentace není soubor, ale něco dynamicky generovaného, tak toto pole buď bude rovno aktuálnímu času, nebo bude úplně chybět). Tento čas pak můžeme porovnat s hodnotou z předchozího stažení.

Pozor na to, že hodiny serveru nemusí být synchronizované s našimi, takže nedává smysl porovnávat `Last-Modified` s naším časem modifikace souboru, do nějž jsme si data uložili. Server nám ovšem v poli `Date` řekne svůj čas, kdy odpověď vytvořil (takže umíme dopočítat stáří dat).

Časy bývají dost nepraktické identifikátory verzí – data mohou vznikat dynamicky kombinací různých zdrojů, data se mohou měnit vícekrát za sekundu atd. Proto server může verzi dat identifikovat také polem `ETag` (entity tag). To je nějaký řetězec, který si každý server může vytvořit po svém. Jediná povinnost je, aby jakákoliv změna reprezentace dat vyvolala změnu ETagu. (To striktně vzato není pravda, protože existují i slabé ETagy začínající na `W/`, které mohou zůstat stejné, pokud se nezměnil *význam* dat. Ale ty potkáme málokdy.)

Kombinace „zjistím, jestli se data změnila, a pokud ano, stáhnu si novou verzi“ je ovšem natolik častá, že pro ni existuje zkratka. Libovolný požadavek jde *podmínit*. Pokud například do hlavičky `GET`u připíšeme pole `If-Modified-Since` s nějakým časem, dostaneme data pouze tehdy, pokud je jejich `Last-Modified` pozdější než tento čas. V opačném případě server odpoví stavem `304 Not Modified`.

Podobně existuje:

- `If-Unmodified-Since` – čili opak `If-Modified-Since`. Typické použití: stáhli jsme si starou verzi, chceme ji `PUT`em vyměnit za novou, ale jen tehdy, když se na serveru mezitím data nezměnila.
- `If-Match` – ETag odpovídá některému z uvedených.
- `If-None-Match` – ETag neodpovídá ani jednomu z uvedených.

Dodejme, že server se může rozhodnout podmínku ignorovat.

Intervalové dotazy

Teď si představte, že si prohlížíme obrovský videozáznam. Tehdy se hodí stáhnout si jen prvních pár megabytů, abychom nemuseli čekat, až se přenese všechno. A pak postupně stahovat další data, třeba i na přeskáčku, pokud po videu chceme skákat.

HTTP tuto situaci řeší *intervalovými dotazy*. Nejdříve pošleme `HEAD` a server odpoví polem `Accept-Ranges: bytes`. Tím říká, že podporuje intervalové dotazy a že intervaly se uvádí v bajtech (specifikace připouští jiné jednotky, ale zatím žádné nedefinuje).

Intervalový `GET` pak v hlavičce specifikuje například `Range: bytes=1000-1999`. Tím si řekne o 1000 bajtů, na což server odpoví stavem `206 Partial Content` a v hlavičce dodá `Content-Range: bytes=1000-1999/8888`, tedy že posílá bajty 1000–1999 z celkem 8888 (pokud není celková délka známá, uvede se `/*`).

Kdybychom se zeptali na interval `-100`, dostaneme posledních 100 bajtů. Můžeme uvést více intervalů, ale pak odpověď dostaneme v poměrně komplikovaném formátu. Pokud se zeptáme na interval, který zčásti leží mimo soubor, dostaneme existující podinterval. A pokud leží celý mimo, dostaneme chybu `416 Range Not Satisfiable`.

Nezapomeňme, že mezi intervalovými dotazy by se soubor mohl změnit. Často tedy přidáváme `If-Match`, případně `If-Range`, s nímž server otestuje, zda objekt odpovídá danému ETagu, a pokud nikoliv, ignoruje `Range` a pošle celý soubor.

Chování intervalů u jiných metod než `GET` není zatím definováno.

Proxies

Server se smí rozhodnout, že požadavek nevyřídí sám, ale předá ho někomu jinému. Předávajícímu serveru se říká *proxy* (zmocněnec). Proxies existují dvou základních druhů: dopředné (ty si vybírá klient) a reverzní (na klienta se tváří

jako cílový server, ale požadavky předávají back-endovým serverům, třeba kvůli rozdělování zátěže mezi více strojů).

Pro proxies platí speciální pravidla, které nebudeme rozebírat. Alespoň poznamenejme, že kdykoliv požadavek projde přes proxy, ta by se měla podepsat do pole `Via` a uvést verzi HTTP, kterou mluvila. U reverzní proxy se navíc hodí, aby předala adresu klienta. Pro to existuje nestandardní, ale běžné pole `X-Forwarded-For`.

Kešování

Proxy může kromě předávání požadavků i kešovat odpovědi – pamatovat si u požadavků, které přes ni prošly, jaká byla odpověď, a pokud se nějaký klient zeptá na totéž, rovnou mu poskytnout zapamatovanou odpověď. Keše mohou být sdílené (pro více klientů) nebo soukromé (pro jednoho, třeba zabudovaná keš webového prohlížeče).

Keš si ovšem musí dávat pozor, aby vždy poskytovala relevantní data. Proto pro chování keší existují poměrně přísná pravidla standardizovaná v RFC 9111 (interní keše v prohlížečích se jimi bohužel ne vždy řídí, což je častým zdrojem potíží). Především se obvykle kešují jenom odpovědi na `GET` a `HEAD` (přičemž negativní odpovědi jen velmi opatrně). Sdílená keš neukládá odpovědi na požadavky, které obsahovaly autorizaci. Aby se nerozbíjel mechanismus domlouvání na reprezentaci, musí keš při objevení pole `Vary` v zakešované odpovědi ověřit, že se nový požadavek shoduje s původním ve všech polích, jež byla ve `Vary` uvedena.

Keš si pro každou odpověď spočítá, jak dlouho ji může poskytovat dalším klientům. Po uplynutí této doby je odpověď *prošlá.* Prošlou odpověď může keš *revalidovat* – poslat serveru `HEAD` nebo podmíněný požadavek, aby si ověřila, že uložená odpověď je stále platná. Keše smí v některých případech revalidaci vynechat a dávat klientům prošlé odpovědi (například není-li server zrovna dostupný).

Server může v poli `Cache-Control` sdělit instrukce pro kešování (oddělené čárkami):

- `max-age=N` – odpověď se smí kešovat nejvýše `N` sekund.
- `must-revalidate` – je zakázáno vynechat revalidaci.
- `no-cache` – navzdory názvu je kešování povoleno, ale musí se revalidovat pokaždé. V praxi se implementuje jako zákaz kešování.
- `no-store` – data není povoleno ukládat na permanentní médium (disk).
- `private` – data smí ukládat jen soukromé keše.

Kromě `max-age` může server poslat též pole `Expires`, kterým uvádí čas (serverový), do kdy je odpověď platná. Pokud nepošle ani jedno, smí keš dobu platnosti odhadnout.

Klient může také instruovat keše pomocí `Cache-Control`, konkrétně:

- `max-age` a `no-store` – jako u serveru.
- `no-cache` – zde opravdu zakazuje kešování.
- `max-stale=N` – klient je ochoten akceptovat prošlá data (která neprošla revalidací), pokud jsou prošlá nejvýše `N` sekund.

Ještě doplníme, že odpověď z keše se doprovází polem `Age`, jehož hodnota říká, kolik sekund uplynulo od zakešování nebo revalidace odpovědi.

Knihovny

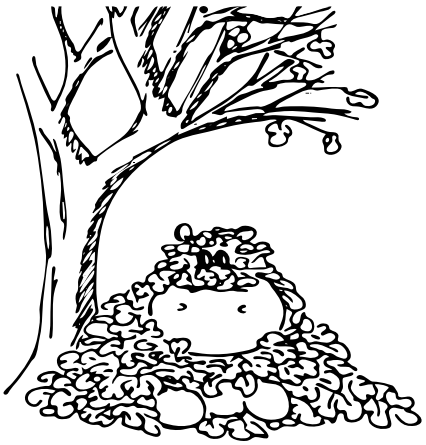
HTTP je docela rozsáhlé a implementovat vlastní server nebo klienta dá dost práce a znamená to strávit pár dní detailním studiem specifikací. Proto nejspíš chcete použít nějakou existující knihovnu. Můžeme doporučit například:

- requests pro Python
- libcurl pro C/C++.
- curl, což je program použitelný z příkazové řádky a ve skriptech, velmi praktický při ručním experimentování.
- httpie je další podobný nástroj.

Také vás mohou inspirovat vývojářské nástroje webových prohlížečů, které umí ukazovat hlavičky HTTP požadavků odeslaných prohlížečem i odpovědí na ně.

Dnešní menu servíroval

Martin Mareš



matfyz

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/*