IB015 – Domácí úkol 12: paketový filtr

Termín: 5. prosince 23.59; 5 pokusů, způsob odevzdání je popsán níže.

Jak možná víte, data mezi počítači v síti (např. Internetu) putují ve formě krátkých kusů informací – paketů. Aby síť byla schopna tuto výměnu dat zajistit, je každý paket vybaven jistými adresními údaji. Protože na koncových stanicích (či v podsítích) zpravidla není žádoucí povolovat veškerou komunikaci, používají se tzv. firewally, které se snaží nežádoucí a potenciálně škodlivou komunikaci zakazovat. Základem firewallu je pak tzv. *paketový filtr*, který si prohlédne každý příchozí i odchozí paket a na základě adresních údajů u každého z nich rozhodne, zda jej dovolí předat dál, nebo jej bez milosti zahodí.

V této úloze není potřeba vědět o sítích prakticky nic. Pracujeme s extrémně zjednodušeným modelem, kde každý paket sestává z **hlavičky** a přenášených **dat** (tzv. *payload*). Hlavička identifikuje **zdroj** (zkratka *src* od *source*) a **cíl** (zkratka *dst* od *destination*) a obsahuje čtyři čísla:

- **zdrojovou adresu** identifikující odesílající počítač,
- **zdrojový port** identifikující konkrétní službu/aplikaci na odesilateli,
- **cílovou adresu** identifikující počítač příjemce, a
- **cílový port** identifikující konkrétní službu/aplikaci na příjemci.

Payload může vypadat jakkoli; do procesu filtrování ale nevstupuje, takže s ním nebudeme příliš pracovat.

Úloha vám dává jisté množství volnosti v řešení spolu s tipy, jak této volnosti využít. Vašimi úkoly bude:

1. navrhnout datové typy pro representaci paketů,
2. poskytnout funkce pro kódování do/z těchto typů,
3. implementovat funkci filtrující pakety podle sady pravidel, a
4. implementovat sadu funkcí na „estetické“ zadávání pravidel.

Kostru řešení naleznete ve studijních materiálech předmětu v Informačním systému. Povšimněte si, že obsahuje zakomentované příklady, které si můžete odkomentovat, až budete považovat příslušnou část úlohy za hotovou.

Representace paketů

Zadány máte typy na representaci 32bitové adresy IPv4 a 16bitového portu:

```
import Data.Word ( Word16, Word32 )

type Address = Word32
type Port    = Word16
```

Typy `Word*` representují přirozená čísla omezeného rozsahu. Jsou instancemi `Num` a `Ord`; můžete s nimi tudíž pracovat jako s jinými čísly: provádět aritmetické operace, porovnávat je a vytvářet v kódu konstanty podobně jako u čísel typu `Int`.

Povšimněte si použití klíčového slova `type`: naše typy jsou jen aliasy (podobně jako `String` je aliasem pro `[Char]`), které se dají libovolně zaměňovat se svými „základními“ typy. Použitím aliasů však podstatně zvyšujeme čitelnost a sebedokumentaci kódu. Očekáváme, že je budete používat i při typování svých funkcí.

S využitím těchto aliasů navrhněte typy nebo typové aliasy[1]

```
data Packet header payload -- paket parametrisovaný hlavičkou a payloadem
data PacketHeader          -- jednoduchá hlavička s adresními údaji dle zadání výše
```

a poskytněte funkce

```
getPayload :: Packet header payload -> payload
getHeader  :: Packet header payload -> header
mkPacket   :: header -> payload -> Packet header payload
mkHeader   :: Address -> Port -> Address -> Port -> PacketHeader
```

Naše testy je budou používat na výrobu a čtení paketů. První dva argumenty `mkHeader` specifikují zdroj (odesilatele), další dva cíl (příjemce).

Tip: Před návrhem typu `PacketHeader` si nejprve přečtěte celé zadání. Zjistíte, že bude vhodné jej vybavit vnitřní strukturou, která umožní jednoduché psaní přístupových funkcí jako `getPort` či `getDst`.

[1] Chcete-li definovat alias, nezapomeňte změnit klíčové slovo `data` na `type`.

Pravidla a jejich řetězy

Zjednodušený paketový filtr, který budete vytvářet, je inspirován skutečnými implementacemi v linuxovém jádře[2]. Princip jeho činnosti je následující:

- Každý paket je prozkoumán poskytnutým *řetězem* (angl. *chain*).
- **Řetěz** sestává ze seznamu *pravidel*, která se postupně filtr snaží na paket aplikovat, než najde vyhovující.
- **Pravidlo** je tvořeno sadou *podmínek* a jednou *akcí*. Pokud paket splňuje všechny podmínky, pak se provede *akce*. Pokud některou podmínku paket nesplňuje, pak toto pravidlo nevyhovuje a pokračuje se dalším pravidlem v řetězu.
- **Podmínka** je tvrzení o hlavičce paketu. Například:
  - „zdrojová adresa je 172.0.0.1“
  - „zdrojový port je z rozsahu 5000–5200“
  - „cílová adresa leží v síti 147.251.48.0/24“[3]
  - „cílový port je 80, 443 nebo 8080“
- **Akce** říká, co má filtr s paketem provést, pokud pravidlo vyhovuje. Budeme podporovat tyto:
  - `Accept` – paket je přijat a má se objevit na výstupu.
  - `Reject` – paket je zahozen a dále se nezkoumá.
  - `Call` ***`chain`*** – akci zkusíme zvolit pomocí jiného řetězu. Pokud ten nevede k přijetí či zamítnutí paketu (byť třeba skrze další `Call`), pokračujeme dalším pravidlem.
  - `Return` – provádění aktuálního řetězu je ukončeno a pokračuje se v nadřazeném řetězu. Efekt je stejný, jako by pravidlo obsahující `Call` aktuálního řetězu vůbec nevyhovovalo.
- Pokud se v řetězu nerozhodne o osudu paketu, pak se na jeho konci provede implicitní akce `Return`.
- V prvním (nejvyšším) řetězu je `Return` ekvivalentní s `Reject`.

Příslušné datové typy vypadají následovně:

```
type Chain  header = [Rule header]
type Rule   header = ([Match header], Action header)
type Match  header = header -> Bool
data Action header = Accept | Reject | Call (Chain header) | Return
```

Jak je vidět, celé soustrojí je parametrisováno typem hlavičky paketu. To znamená, že náš paketový filtr půjde použít s rozličnými druhy paketů, pokud k nim budeme schopni tvořit pravidla. V této úloze sice navrhujete jen jeden typ, `PacketHeader`, nicméně takto obecné otypování nutí programátora oddělit logiku zpracovávání řetězu od logiky kontroly podmínek.

Takové vynucení dekomposice je zcela v duchu funkcionálního programování, zvýší čitelnost a umožní případné rozšíření a znovupoužití kódu. Navíc umožňuje testovat implementaci filtru nad mnohem jednoduššími daty. Na další straně například uvidíte řetěz pro filtrování obyčejných celých čísel.

Filtrování

Srdcem celého řešení bude funkce

```
packetFilter :: Chain header -> [Packet header payload] -> [Packet header payload]
```

Ta každý paket na vstupu prožene dodaným řetězem a přijaté pakety (ty, které vedou k akci `Accept`) vloží i do výstupního seznamu. Protože paketový filtr by nám nebyl moc platný, kdyby nezvládal rozhodovat o paketech hned, jak přichází na vstup, musí tato funkce produkovat pakety postupně (líně) a tím zvládat zpracovávat i nekonečné seznamy paketů.

Jinými slovy, `take n (packetFilter chain packets)` se musí vyhodnotit v konečném čase pro libovolné číslo `n`, libovolný konečný `chain` a libovolný i nekonečný seznam `packets` obsahující alespoň `n` paketů, které mají být dle řetězu přijaty. Předpokládejte, že podmínky v řetězu jsou totální, tedy jejich ověření samo nezpůsobí nekonečný výpočet ani chybu.

Tip: Napište si pomocnou funkci, která na základě řetězu a hlavičky vrací podle zvolené akce `Maybe Bool`. Volání pomocného řetězu bude odpovídat rekursivní aplikace této funkce, akci `Return` pak výsledek `Nothing`.

Za správnou implementaci této funkce již můžete obdržet část bodů. Menší část bodů si vysloužíte, i když nebude správně fungovat `Call` či filtrování nekonečných seznamů.

[2] V případě zájmu se můžete pokochat IPTables a NFTables.

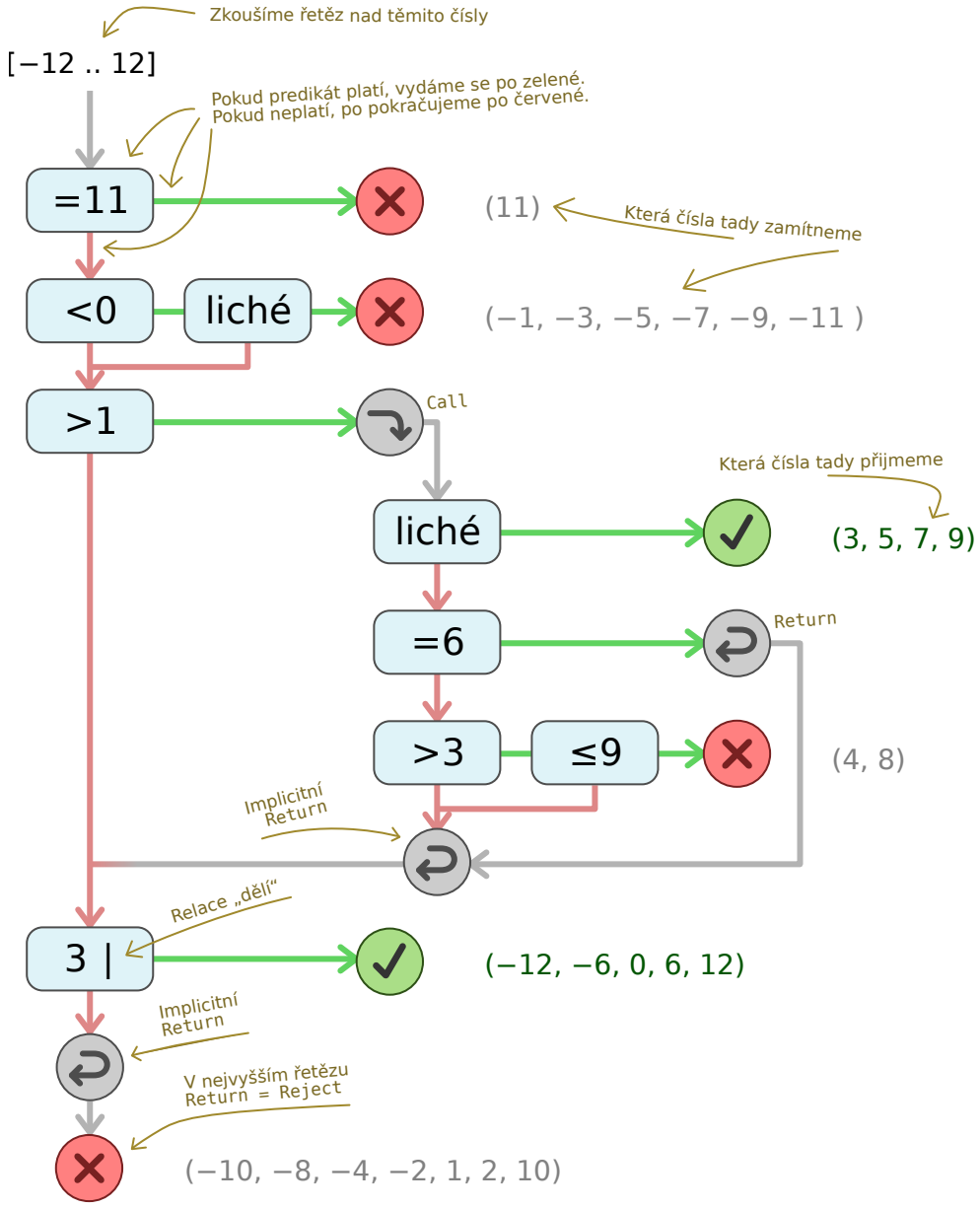
[3] Je zde použita tzv. *notace CIDR*. Pro vyřešení úlohy ji není potřeba chápat (potřebnou funkci vám poskytujeme v kostře), ale pro úplnost uvedeme, že číslo za lomítkem značí počet horních bitů v 32bitové adrese, které označují síť. Zbylé (spodní) bity pak identifikují počítač v této síti. Do sítě v příkladu tak spadají všechny adresy tvaru 147.251.48.X pro X mezi 0 a 255.

Příklad s náčrtkem

Následuje visualisace jednoduchého řetězu na filtrování čísel, respektive paketů, jejichž hlavičkou je pouze jedno celé číslo. Jeho haskellová podoba je následující a naleznete ji také v kostře řešení.

```
ex_simpleChain :: Chain Int
ex_simpleChain = [([ (== 11) ],    Reject)
                 ,([ (< 0), odd ], Reject)
                 ,([ (> 1)],       Call
                                [([ odd ],           Accept)
                                ,([ (== 6) ],        Return)
                                ,([ (> 3), (<= 9) ], Reject)
                                ])
                 ,([ (== 0) . (`mod` 3) ], Accept)
                 ]
```

Pro vstup od −12 do 12 (v kostře `ex_simplePackets`) má filtrování tímto řetězem přijmout pakety −12, −6, 0, 3, 5, 6, 7, 9 a 12 (v kostře `ex_simpleExpected`).



Definice pravidel

Způsob zápisu pravidel (resp. podmínek) bude trochu netypický. Pokusíme se využít syntaxe jazyka Haskell k tomu, aby popis podmínek vypadal esteticky. Například budeme chtít mít sadu podmínek popisující spojení na webový server Aisa ze známých portů počítačů uvnitř sítě Masarykovy university, kterou zapíšeme takto:

```
httpToAisaFromWellKnownMU :: [Match PacketHeader]
httpToAisaFromWellKnownMU = [ dst addr equals "147.251.48.1"         -- Aisa
                            , dst port member [80, 443, 8080]        -- webové porty
                            , src port within (0, 1023)              -- známé porty
                            , src addr subnet ("147.251.0.0", 16) ] -- síť MU
```

To vypadá vcelku čitelně, zaplatíme za to však složitějším návrhem toho, co (a jakých typů) se skrývá za jmény `dst`, `port` či `subnet`. Takovým ohýbáním možností jazyka (zde Haskellu) k vytvoření jakéhosi menšího specifického jazyka (zde na popis podmínek) vzniká tzv. eDSL – *embedded domain-specific language*.

Povšimněte si, že zde už nemůžeme pracovat s obecnými pakety, o jejichž hlavičkách nic nevíme. Pravidla budou ušitá na míru vámi navrženému typu `PacketHeader`.

V našem maličkém *zabudovaném jazyce* bude mít každá podmínka čtyři mezerou oddělené části v tomto pořadí:

1. **Směr**: `src` nebo `dst`, který říká, zda nás zajímá odesilatel, nebo příjemce.
2. **Údaj:** `addr` nebo `port`, který říká, zda se díváme na adresu IP, nebo port.
3. **Porovnání:** `equals`, `member`, `within` nebo `subnet`, které říká, jakým způsobem budeme zvolený údaj porovnávat se čtvrtou položkou – *parametrem*:
   - `equals`: podmínka platí, je-li údaj přesně roven *parametru*,
   - `member`: podmínka platí, je-li údaj jedním ze seznamu *parametrů*,
   - `within`: podmínka platí, je-li údaj v rozsahu zadaném *parametrem*, a
   - `subnet`: podmínka platí, patří-li adresa IP do podsítě zadané *parametrem*.
4. **Parametr:** S čím se zvolený údaj porovnává; závisí přímo na předchozí položce:
   - `equals`: parametrem je číslo portu nebo adresa IP,
   - `member`: parametrem je seznam čísel portů nebo adres IP,
   - `within`: parametrem je dvojice (od, do)[4] zadávající rozsah (včetně) portů či adres IP,
   - `subnet`: parametrem je dvojice (adresa IP, prefix typu `Subnet`) zadávající adresu IP a počet nejvyšších bitů adresy, které tvoří adresu sítě.

Pro účely zadávání podsítí zavádíme další typový alias. Také vám dáváme k disposici predikát, který o adrese rozhodne, zda patří do podsítě.

```
import Data.Word ( Word8 )
type Subnet = Word8
isInNet :: Address -> (Address, Subnet) -> Bool
```

Možná jste si všimli, že zatímco porty uvádíme číselně, adresy IP zapisujeme tečkovou notací do řetězce. Jistě vás potěší, že převodní funkce je součástí kostry:

```
ip :: String -> Address
```

Co to však znamená, je, že *parametr* porovnání závisí nejen na samotném *porovnání*, ale i na *údaji* – zajímá-li nás `port`, pak můžeme pracovat všude rovnou s čísly a porovnání budou typově sedět; zajímá-li nás však `addr`, je nutné všechny adresy v *parametru* převést na číslo typu `Address`. Která komponenta by tento převod měla zajišťovat, je nasnadě – ta, která ví, jak přesně bude *parametr* vypadat a kde v něm hledat adresy.

Adresy a porty mají různé číselné typy. To je záměr a pomůže to jisté typové bezpečnosti. Například by tak nemělo být možné ptát se, zda port patří do podsítě. Není tedy vhodné nejprve všechny údaje převést na jednotný číselný typ. Místo toho by měly být všechny typy funkcí dostatečně obecné, aby ve výsledku bylo jedno, zda pracujeme s 16bitovým portem nebo 32bitovou adresou. To vyžaduje **psát od začátku explicitně typy**, jinak se neodvodí dostatečně obecný typ.

Možná vás napadlo, že všechny typy porovnání lze převést na `member`. Nedělejte to – podsítě nebo rozsahy mohou být velmi velké a rozbalovat je na seznam všech adres či portů v rozsahu by bylo velmi neefektivní.

[4] Je-li druhá složka menší než první, jedná se o prázdný rozsah, kterému žádná adresa či port nikdy nevyhoví.

Návod

Výše zadané eDSL je výsledkem iterativního návrhového procesu. Vymyslet typy a definici všech použitých slov rovnou by bylo silně netriviální. Tato sekce slouží k tomu, aby vás skrze několik mezikroků provedla tímto procesem. Začneme s o něco „méně estetickým“ eDSL, u kterého je ale více vidět, jak vnitřně pracuje, a postupně se propracujeme k výslednému jazyku.

Krok 1. Začneme něčím jednoduchým. Napište funkce tak, abyste byli schopni zadávat podmínky následujícím způsobem:

```
match getDst getPort (== 80)
match getSrc getAddr (== ip "127.0.0.1")
```

V tuto chvíli by `match` měla být jednoduchá funkce vyššího řádu, která jen vhodně poskládá své argumenty, ovšem postupně ji bude ještě potřeba upravovat. Výsledek těchto výrazů je typu `Match`, neboli **funkce** (!) `PacketHeader -> Bool`. Lambdou bychom měli být schopni specifikovat i další druhy porovnání:

```
match getDst getPort (\p -> p `elem` [80, 443, 8080])
match getDst getAddr (\a -> a `elem` [ip "127.0.0.1", ip "127.0.0.2"])
match getSrc getPort (\p -> p >= 1024 && p <= 2048)
match getSrc getAddr (\a -> a >= ip "147.251.0.0" && a <= ip "147.251.255.255")
```

Sami jistě určíte, jak bude vypadat porovnávání podsítě.

Krok 2. Vytáhneme porovnávací logiku do samostatných funkcí. Zamyslete se, kdo musí porovnávátkům předat jejich parametr.

```
match getDst getPort equals 80
match getSrc getAddr equals (ip "127.0.0.1")
match getDst getPort member [80, 443, 8080]
match getDst getAddr member [ip "127.0.0.1", ip "127.0.0.2"]
match getSrc getPort within (1024, 2048)
match getSrc getAddr within (ip "147.251.0.0", ip "147.251.255.255")
match getSrc getAddr subnet (ip "147.251.49.0", 24)
```

Krok 3. Už nás nebaví pořád psát `match`. Triviální úpravou převedeme příklad do kratší podoby. Neduplikujte při tomto kroku kód, místo toho využijte částečné aplikace funkcí.

```
dst getPort equals 80
src getAddr equals (ip "127.0.0.1")
dst getPort member [80, 443, 8080]
dst getAddr member [ip "127.0.0.1", ip "127.0.0.2"]
src getPort within (1024, 2048)
src getAddr within (ip "147.251.0.0", ip "147.251.255.255")
src getAddr subnet (ip "147.251.49.0", 24)
```

To už je pěkný kus práce, a pokud vám stačí část bodů, stačí přejmenovat `getPort` na `port`, `getAddr` na `addr` a prohlásit za hotovo.[5]

V plnotučné variantě je však `port` a `addr` něco mírně složitějšího. Trnem v oku je nám teď totiž to zbytečné explicitní používání `ip` u každé adresy. Je přece jasné, že když chceme porovnávat adresu, je potřeba ji nejprve převést z řetězce na typ `Address`.

Krok 4. Převaděč `ip` proto nějak šikovně schováme do `addr`. Při tom se nevyhneme také úpravě `port` a ostatních částí. Na použití převaděče navíc musíme uzpůsobit i porovnávátka. Nesnažte se však zneužít typovou třídu `Show`! Výsledný jazyk pak vypadá nějak takto (explicitní `ip` už fungovat nemá):

```
dst port equals 80
src addr equals "127.0.0.1"
dst port member [80, 443, 8080]
dst addr member ["127.0.0.1", "127.0.0.2"]
src port within (1024, 2048)
src addr within ("147.251.0.0", "147.251.255.255")
src addr subnet ("147.251.49.0", 24)
```

[5] Výstup (nikoli vstup!) testů vám ale věří, že se tady nezastavíte. Z technických důvodů v testech vypisované protipříklady uvádí `"0.0.0.0"`, i když v závislosti na tom, co vaše řešení podporuje, mu ve skutečnosti možná předávají `(ip "0.0.0.0")`.

Výzva: syntakticky krásná negace

Nemáte-li stále dost, můžete si jako nepovinné procvičení zkusit napsat operátor vykřičník sloužící k negaci podmínky, který se dá používat následujícím způsobem. Není to až *tak* obtížné a odměnou vám bude nehynoucí respekt autora úlohy.

```
dst port ! equals 80
src addr ! member ["127.0.0.1", "127.0.0.2"]
```

Poznámky a tipy

- Importovat smíte moduly z balíku `base`.[6]
- Nevynalézejte znovu kolo! Hledejte mezi knihovními funkcemi fakultní instancí Hoogle.
- Řiďte se tipy k psaní dobrého kódu ze Sbírky! Zejména neduplikujte kód, ale tvořte abstrakce.
- Nejste-li si jisti nějakou částí zadání, zeptejte se v diskusním fóru.
- Pište si vlastní testy. Příklady v kostře pokrývají jen zlomek řešení.
- Nezapomeňte, že **opisování je zakázáno** a bude postihováno podle disciplinárního řádu.
- Přebíráte-li kód odjinud, uveďte zdroj, jinak bude na vaši práci pohlíženo jako na plagiát.

Odevzdání

Odevzdávání a vyhodnocování se řídí stejnými pravidly jako předchozí velké úlohy (osmá a desátá), podrobnosti proto hledejte u nich.

Rekapitulace nejdůležitějších pravidel:

- Svůj jediný soubor s příponou `.hs` nahrajte do odevzdávárny v Informačním systému.
- Výsledky testů se po odevzdání objeví v poznámkovém bloku (nejvýše v řádu desítek minut).
- Všechny globální funkce opatřete typovými signaturami.
- Máte pět pokusů na splnění testů; nepřeložitelná řešení se do limitu nepočítají.
- Neimplementované funkce mohou v řešení zcela chybět.

Protože eDSL je poměrně křehká záležitost, co se otypování týče, věnujte pozornost následujícím bodům:

- Nepoužívejte v otypování funkcí eDSL **žádné typové třídy kromě `Eq` a `Ord`**.
- I pokud se vám nepodaří dostat eDSL do funkčního stavu, je z technických důvodů nutné, aby výrazy použité při tvorbě pravidel[7] byly **otypovatelné**, nebo aby jednotlivé funkce úplně **chyběly**. Triviální implementace v kostře splňuje to první, zakomentování celého kódu týkajícího se eDSL splní to druhé.

Hodnocení

Úloha má několik automaticky testovaných aspektů, které jsou samostatně bodově hodnoceny:

Funkcionalita	Body
`packetFilter` (řetěz bez `Call`)	0,4
`packetFilter` (řetěz vč. `Call`)	0,5
`packetFilter` funguje líně	0,1
eDSL na zápis pravidel (po krok 3 v návodu)	1,0
eDSL nemá explicitní `ip` (krok 4 v návodu)	0,5

Celkově můžete od automatických testů získat **až 2,5 bodu**. Úloha je následně hodnocena ještě cvičícími, kteří vám za čitelnost a eleganci řešení mohou udělit navíc **až 0,5 bodu**. Dejte si proto záležet na úhlednosti a dobré dekomposici.

Cvičící bude hodnotit vždy poslední odevzdání s maximem bodů ze všech vašich odevzdání. Pokud vám tedy po splnění všech automatických testů ještě zbývají pokusy, můžete bez obav odevzdat ještě uhlazenou versi. Na druhou stranu, pokud si řešení nejprve rozmyslíte a psát hezky ho budete od začátku, bude se vám tvořit příjemněji a s menším množstvím chyb. Přístup „nějak to uplácám a pak případně uklidím“ tedy obecně nedoporučujeme.

Celkově můžete za úlohu získat **až 3 body**.

[6] Výjimku tvoří moduly, které jsou „`Unsafe`“, ty však určitě nebudete potřebovat.

[7] Myšleno všechny přípustné kombinace jako `src port equals 80`, `dst addr subnet ("0.0.0.0", 32)` apod.