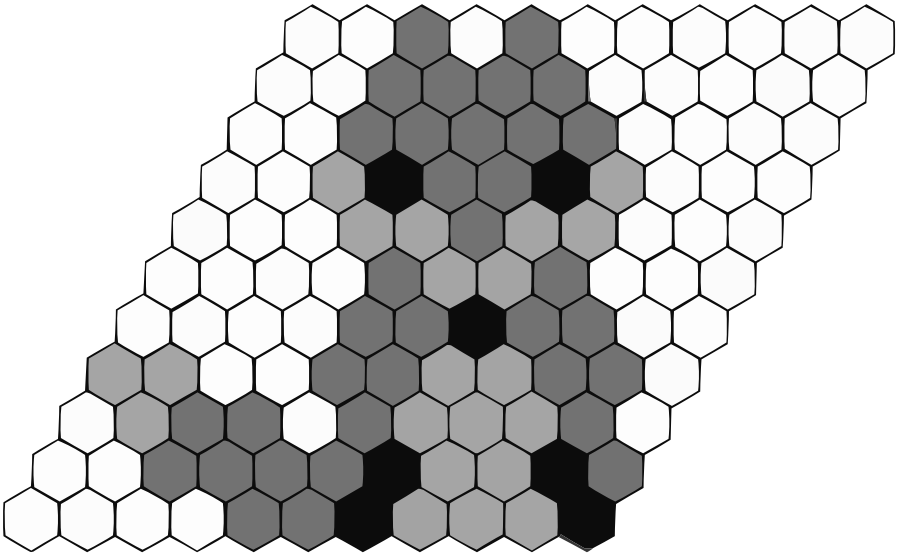
STUDENTSKÝ ČASOPIS A KORESPONDENČNÍ SEMINÁŘ

Ročník XXX
Číslo 1

MATEMATIKA FYZIKA INFORMATIKA



Uvnitř najdete několik témat a s nimi souvisejících úloh. Zamyslete se nad nimi a pošlete nám svá řešení. My vám je opravíme a ta nejzajímavější z nich otiskneme. Nejlepší řešitele zveme na podzim a na jaře na soustředění.

Milý čtenáři,

právě čteš první číslo 30. ročníku časopisu M&M. Možná jsi časopis dostal/a na nějaké z akcí Matfyzu, možná ti časopis dal nějaký kamarád či kamarádka a možná jsi ho našel/a na stolku s letáky. Na tom, jak se k tobě časopis dostal, však vůbec nezáleží. Důležité je, že už otevřít tento sešit vyžadovalo jistou dávku odvahy a zvídavosti. Máš-li ještě nějakou stále v zásobě, můžeš nyní s její pomocí časopis prolistovat a ponořit se do řešení. Pokud nevíš, jak na to, prohlédni si letáček „Jak řešit“[1], nebo navštiv náš web `mam.matfyz.cz`.

Co na tebe tento rok čeká? V prvním čísle jsme si připravili tři témátka. První z témátek je fyzikální a zabývá se chováním tepla a vnitřní energie. Postupně si projdeme všechny čtyři termodynamické zákony a pochopíme jejich důsledky v reálném světě. V témátku o Leanu se seznámíme s funkcionálním programováním, což je programování bez příkazů, a taky se dozvíme, co je to formální verifikace. A konečně v témátku o hře Hex se podíváme na hru podobnou té z AZ kvízu. Zkusíme zjistit, jak závisí výherní strategie na tvaru herního plánu, a taky si ji samozřejmě zahrajeme.

Pokud by ses o M&Mku rád/a dozvěděl/a něco dalšího, případně co je u nás nového, můžeš kromě webu `mam.matfyz.cz` navštívit i naše sociální sítě. S jakýmkoliv dotazem se na nás neváhej obrátit buď na mailu `mam@matfyz.cz`, nebo na našem Discordovém serveru[2]. Připomínáme, že řešit úlohy a témátka lze i ve skupinách, a právě Discord může být to místo, kde se s někým na spolupráci domluvit.

Budeš-li v řešení úspěšný/á, budeš se moci zúčastnit spolu s dalšími nejlepšími řešiteli podzimního nebo jarního soustředění. Pokud by tě tato představa nelákala, nemusíš se bát, potěšíme tě i tak, například odpuštěním přijímaček na Matfyz či nějakou odměnou. Pokud by tě však tato představa lákala, věz, že část řešitelů podzimního soustředění vybíráme dle výsledků prvního deadlinu prvního čísla, pokud tedy pošleš řešení do 12. září a budeš mezi nejlepšími, tak tě na soustředění pozveme!

Těšíme se na tvá řešení.

Tvoji organizátoři

Obsah

[1] Elektronickou verzi letáčku také najdeš na webu: `https://mam.mff.cuni.cz/jak-resit/`

[2] Adresa Discordového serveru: `https://discord.gg/MxPjKsDFWG`

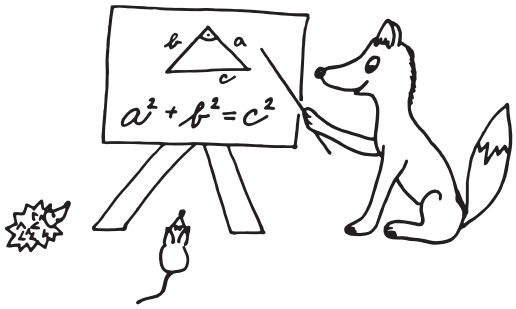
Zadání témat

1. deadline: 12. září 2023 | 2. deadline: 10. října 2023
Řešení odevzdaná do 12. září se započítají pro účast na soustředění.

Téma 1 – Termodynamika

Vítej u prvního dílu tématka, ve kterém se budeme zabývat energií a termodynamikou. Termodynamika je obor fyziky, ve kterém pomocí fyzikálních zákonů popisujeme chování jednotlivých molekul látek. Termodynamika bere jako alfu a omegu čtyři termodynamické zákony, které slouží jako návod popisu světa. Pokud jejich znění neznáte, podívejte se na internet[3]. Hlavními veličinami objevujícími se v termodynamice jsou různé formy energie (práce, teplo, termodynamické potenciály).

Pokud máš spoustu otázek ohledně těchto témat, tak zde je správné místo na to pokládat otázky a číst odpovědi. Vítáme dotazy, které ve tvé mysli vyvstanou ať už při čtení textu tématka, řešení problémů a úloh, měření experimentů nebo i kdykoliv jindy. Dotazy piš na náš Discordový server[4] nebo na e-mail na konci tohoto textu. Piš nám také o tom, co bys rád/a v tomto tématku četl/a v příštích dílech a my se ti pokusíme vyhovět a zakomponovat to do textu. V tématku postupně projdeme všechny čtyři termodynamické zákony a nějaká pozorování ve světě kolem nás, která z nich vyplývají. Text bude obsahovat i nějaké početní úlohy pro ty, kdo rádi počítají a přemýšlí nad výsledky úloh.



Cílem autorů je vám předat co nejvíc pro vás zajímavých informací. Tím, že se budeš ptát, můžeš ovlivnit, jakým směrem se toto tématko bude ubírat. Budeme se snažit zakomponovat různě obtížné úlohy, aby si každý něco našel. Pro každé tématko platí, že nemusíš řešit všechny úlohy (nebo všechny části jedné úlohy), abys mohl své řešení odevzdat.

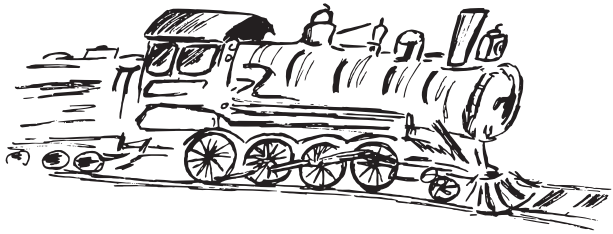
[3] Třeba na Wikiskripta: `https://tinyurl.com/4drbsj77`
[4] Pro připomenutí odkaz: `https://discord.gg/MxPjKsDFWG`

K čemu je taková termodynamika dobrá?

Abychom si mohli říct, k čemu je termodynamika dobrá, zavedeme na začátek několik pojmů: stav, systém s jeho hranicemi a okolí.

Systém je nějaký ohraničený prostor, který je předmětem našeho zájmu. Většinou jde o objekt, jenž je středobodem slovních úloh: termoska s čajem, rampouch, plyn v balónku apod. Každý systém má hranici a za tou se nachází *okolí* a další systémy, se kterými může interagovat. V jednom systému si můžeme vymezit další, menší (pod)systém: termoska s čajem bude mít podsystém čaj, další podsystém může být pára nad hladinou čaje v termosce; další příklad může být kovová kulička vhozená do vody v hrnci na sporáku: jeden systém je voda s kuličkou, druhý podsystém je samotná kulička. Hranice systém–okolí nebo systém–systém může být pohyblivá (stačí si vzpomenout na to, že systém můžeme stlačit a změnit jeho tvar a/nebo objem nebo z něj „odkrojit“ hmotu).

Okolí je tedy zbytek světa, který je mimo náš vybraný systém. Systém si většinou vybíráme tak, aby bylo jednoduché v něm pozorovat nebo měřit fyzikální veličiny, které potřebujeme k výpočtům nebo k úvahám o procesech, jež se v systému dějí. Kdybychom na chvíli zastavili čas a zapsali si hodnoty všech veličin, tak bychom získali hodnoty specifické pro daný *stav*: tlak, teplotu, hustotu, látkové množství a jiné. Proto se takovým veličinám říká stavové. Všimněte si, že některé veličiny v takovou chvíli nezměříme: jde dost často o formy energie jako práce a teplo. Hodnoty energie se často nedají změřit přímo, ale musíme je dopočítávat z rozdílů hodnot naměřených stavových veličin pro dva různé stavy. V tomto díle se podíváme na to, jak se přepočítávání dělá pro práci, vnitřní energii a trochu i pro teplo, o kterém bude převážně další díl.



Jeden příklad systému typický pro termodynamiku je plyn v nádobě. Systém obsahuje náš plyn a nádobu; okolí je všechno mimo nádobu: stůl, na kterém stojí nádoba, vzduch okolo nádoby, vědec, který nádobu zkoumá… Pokud vědec ve stejný moment změří teplotu a tlak v nádobě, získá hodnoty těchto veličin pro jeden stav. Teplota a tlak jsou tedy stavové veličiny, protože je vědec umí přiřadit jím vybranému stavu. Nás ale většinou zajímá energie, ne hodnota teploty a tlaku. Energie má totiž tu výhodu, že se přelévá z jedné své formy do druhé a jakmile ji vypočteme z jedněch veličin, můžeme z její hodnoty odvodit veličiny druhé. To je nejdůležitější přístup, který termodynamika využívá. Náš vědec může systém ohřát (zvedne se teplota), a pokud je nádoba uzavřená, vzroste v ní tlak. Vidíte, že tepelná energie se „přelila“ do jiné formy – tlakové.

Teplo a práce jako nestavové veličiny

Práce, jak ji chápe obecná fyzika, je množství energie, která se během fyzikálního děje přemění z jedné formy na druhou. Spousta definic říká, že pokud má těleso energii, tak je schopné konat práci. Takže jinými slovy, když něco nebo někdo vykonává práci, tak přeměňuje formy energie. To se projeví jako změna hodnoty nějaké stavové veličiny. Stlačíte nafouklý balónek a změní se v něm hodnota objemu a tlaku. Doktor natáhne do injekční stříkačky vakcínu – změní se hodnota látkového množství a velikost objemu systému (což je injekční stříkačka). Proběhne chemická reakce – změní se složení směsi v systému. Můžete si to představit tak, že každý systém má svůj energetický účet, ze kterého může převádět část energie z jedné formy na druhou. Přeměna forem energie se mimo jiné projevuje jako změna hodnot měřitelných veličin – poklesne tlak systému, zvětší se látkové množství, vzroste objem systému atp. Přitom však neporušuje znění termodynamických zákonů. Může se tak dít samovolně nebo i pod tlakem podmínek a faktorů vně systému. Stejně funguje i příjem/odevzdání energie systémem od/do jiného systému.

Nabízí se tím pádem otázka: Jak vypadá a jak se chová systém, co nemá energii? Můžeme nějaké takové systémy v realitě najít? Už umíme pomocí definice říct, že takový systém nekoná práci. Kde se ta energie v systému vlastně schovává?

Problém 1 [max 6b]: *V jakých formách jde uložit energii do systému? Pošlete nám vaše nápady. Nebojte se vzít v potaz různě velké i malé objekty, systémem se může rozumět i něco živého. Zkuste ve svém řešení popsat, kde je energie uložena a také jak vypadá proces, při kterém se daná forma energie uvolňuje/přeměňuje na jinou. Za každý správný originální princip a popis přeměny, který nikdo jiný neuvedl, dostanete dohromady jeden bod. (Tato úloha se vztahuje k 1. i 2. deadlinu. Pokud něco pošlete k 1. deadlinu a bude to originální, dostanete půlbod, pokud budete originální i mezi řešiteli 2. deadlinu, dostanete druhého půl bodu.)*

Není nutné hned zabíhat do částicové fyziky (ale samozřejmě to není zakázané, i tam je energie třeba), nejste omezeni ani na planetu Zemi, ale myslíme si, že podívat se po vašem domě nebo okolí vám bude stačit. Zkuste si všímat všeho, kde se děje nějaká činnost, kterou vnímáte vašimi smysly. Pamatujte také na to, že v systému se může měnit i počet částic (tedy hmotnost). V dalším čísle zveřejníme nejzajímavější a nejpodnětnější příklady, které pošlete.

Přeměníme energii na energii

Máme kontrolu nad tím, jaké formy energie budeme v systému přetvářet na jaké? Ano i ne. Regulovat jak velká část energie se přemění konkrétním dějem je už složitější. Neexistuje žádný proces, při kterém by nebyly ztráty, takže plnou kontrolu nad přeměnou energie mít nikdy nebudeme. Když nabíjíme telefon, ne všechna elektrická energie z elektrické sítě se uloží do baterky telefonu, už jen proto, že se ohřeje nabíječka.

Ve zbytku tématka se nám bude hodit rozlišovat teplo a práci v termodynamickém smyslu. Tyto dva koncepty se svou podstatou liší. Teplo popisuje energii vyměněnou tepelnou výměnou (tj. ohříváním systémů od sebe navzájem); práce pak ostatní vyměněnou energii – nezmění se hodnota teploty, ale jiných veličin. Ve výsledku je práce ta všechna energie, která se při interakci s jiným systémem nepřemění na teplo. Pod pojmem „konání práce“ se schovává samotný akt přeměny energie.

Otázka k zamyšlení pro vás, na které můžete založit nějaký zajímavý článek a poslat nám ho: jak tedy od sebe poznáme, co je teplo a co práce? Kde přesně rozřízneme balíček energie, který teče od jednoho objektu ke druhému, na tyto dvě formy? A má smysl práci a teplo vůbec rozlišovat? Pro nás ano, dost často potřebujeme znát velikost jen určitého typu energie (třeba při nabíjení telefonu nebo u táborového ohně).

Někdy potřebujeme vědět, jakou hodnotu energie má náš objekt k dispozici na svém energetickém kontě celkem – to spočteme tak, že sečteme hodnoty všech typů energie, které těleso má k dispozici. Na to, abychom mohli tento výpočet provést, musíme znát příslušné vzorečky a veličiny v nich, které jsou pro různý zdroj energie různé. Tento součet nám dá hodnotu veličiny, které se říká *vnitřní energie*. Značíme ji U a je to součet dodaného/odevzdaného tepla a vykonané práce. Jde si ji také představit jako celkovou hodnotu energie, kterou má systém na svém energetickém kontě, podobně jako vy máte peníze uložené v bance a poté je distribuujete na různé výdaje.

Ukažme si to na příkladu: fotbalista (první systém) nakopne míč (druhý systém). Před kopnutím má míč nulovou kinetickou energii, ale po srážce s fotbalistovou nohou od ní přijme energii – míč se nepatrně ohřeje v místě kontaktu a dá se do pohybu. Hodnota, o kterou naroste vnitřní energie míče (našeho systému), je tedy předané teplo a nárůst energie mezi stavy *míč leží na zemi před nakopnutím – míč letí vzduchem po nakopnutí*. Matematicky můžeme nárůst vnitřní energie napsat jako

$$\Delta U = U_2 - U_1 = U_{\text{po nakopnutí}} - U_{\text{před nakopnutím}}$$

U_2 a U_1 jsou hodnoty vnitřní energie pro stav na konci a na začátku a ΔU říká, o kolik se tyto dvě hodnoty liší. Zdůrazňujeme, že vnitřní energie je veličina stavová, ve které jsou schovány teplo a vykonaná práce.

Odbočka pro zvídavé matematiko-fyziky

Všimněte si, že práce je skalární veličina. To znamená, že nemá směr působení nebo orientaci v prostoru tak jako třeba magnetická indukce nebo síla, které jsou obě veličiny vektorové. Energie a teplo také nejsou vektorové, ale skalární[5]. Můžete si vzpomenout, že jste někdy slyšeli, že teplo se předává z jednoho objektu na druhý, tak jak to, že teplo nemá směr? Tuto otázku si schováme do druhého dílu, aby tento díl nebyl moc dlouhý :).

Úloha 2 [2b]: *Zamyslete se nad tím, jak je možné, že mechanická práce je skalární veličina, když se počítá jako součin veličin síla a dráha, kde síla i dráha jsou veličiny vektorové. Své zdůvodnění sepište.*

První termodynamický zákon (konečně)!

Nyní už máme trochu představu o tom, jak se chová energie, takže si můžeme představit první termodynamický zákon (1. TDZ). Matematicky jde zapsat takto: $\Delta U = W + Q$ a pro milovníky diferenciálů, derivací a integrálů takto: $\mathrm{d}U = \delta Q + \delta W$.

1. TDZ je axiom, tedy nedokazatelné tvrzení, které se bere jako fakt[6]. Veličiny v tomto vzorci se nazývají vnitřní energie systému U, resp. rozdíl energií před a po nějakém ději $\Delta U = U_2 - U_1$. Práce W a teplo Q jsou pro vás už známé veličiny. Zda hodnoty Q a W budou kladné nebo záporné závisí na tom, zda jsou do systému dodány (kladné teplo) nebo z něj odchází (záporné teplo). V tomto díle tématka se podíváme na práci, o teple si povíme něco v příštím díle spolu s nultým termodynamickým zákonem.

Problém 3: *Může existovat práce záporná? Zdůvodněte své vysvětlení.*

Různě značené diferenciály veličin značí rozdíl mezi vnitřní energií a teplem/prací: rozdíl hodnoty vnitřní energie jakožto stavové veličiny mezi dvěma stavy nezávisí na tom, jakým způsobem přejdeme ze stavu počátečního do stavu konečného. Q a W ve vzorci nepopisují hodnotu danou pro stav, ale hodnotu energie získanou/ztracenou během nějakého děje – dodání nebo odebrání tepla či příjem nebo konání práce. Proto se práce a teplo označují za nestavové veličiny. Je celkem zajímavé, že ze dvou nestavových veličin umíme jejich součtem udělat jednu veličinu stavovou, pomocí které už umíme popsat, jak systém vypadá v námi vybraných momentech za nějakých podmínek: třeba kostka ledu před táním a po něm, plyn před stlačením a po něm a jiné další stavy a systémy. Všimněte si, že jsme nepopsali, jakým způsobem kostku ohřejeme nebo v jakém zařízení budeme plyn stlačovat. To je právě to kouzlo vnitřní energie – nezávisí na tom, jak vypadá vnější prostředí pro náš studovaný systém, pouze na tom, co okolí systému

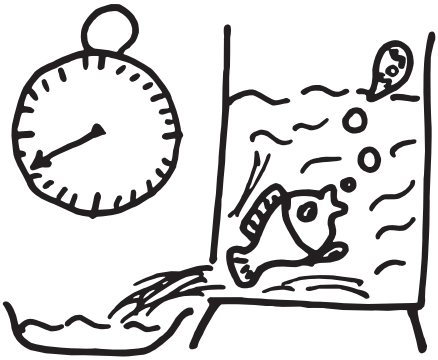
[5] Existují ještě veličiny tenzorové, což je obecný případ jak vektorové, tak skalární veličiny, které pro naše povídání o termodynamice nebudeme potřebovat.

[6] Tedy alespoň dokud ho někdo prokazatelně nevyvrátí :).

způsobuje za změny v našem systému. Toto samé jste mohli slyšet jako poučku, že rozdíl dvou hodnot stavové veličiny „nezávisí na cestě“.

Vraťme se na chvíli k analogii energetického konta: vnitřní energie je množství energie, které má objekt (nebo také jinak řečeno systém) na svém energetickém kontě. Z tohoto množství může systém vytvářet ony balíčky, se kterými se pak něco děje (balíčky jsou ΔU). U je tedy hodnota na účtě a závisí jen na složení a podmínkách, které jsou uvnitř systému v daný moment (stav). Slovy jednoho mého učitele, padající rampouch měl stejnou vnitřní energii i v moment, kdy ještě visel na okapu, akorát tehdy měl navíc potenciální energii na to někoho při pádu zranit.

Popis systému vnitřní energií je trumf v rukávu, který může fyzikům dovolit zahodit nějaké informace o okolí a nebrat je v úvahu během výpočtů, což jim zjednodušuje práci. Zkoumání procesů jako chování plynů, skupenské přeměny látek, směšování substancí, tavení slitin a chemické reakce se stává mnohem jednodušším. Na některé z těchto odvětví fyzikální chemie se podíváme v dalších dílech tohoto témátka.



Vlastnosti systému: stereotyp (ne)ideálního plynu

Pokud pracujeme nebo počítáme s plyny, tak se jako nejčastější práce jeví práce objemová $W = -p\Delta V$, pro milovníky diferenciálů $W = -\int_{V_1}^{V_2} p\,\mathrm{d}V$. Máme i jiné druhy práce, určitě na ně narazíte při řešení problému 1. Veličina p je tlak působící z vnějšku během děje od počátečního do konečného stavu, V je objem systému, $\Delta V = V_2 - V_1$ je rozdíl objemů ve stavech před a po ději, který pozorujeme. Pozor, tlak během děje nemusí být konstantní. Změnu hodnoty tlaku během přechodu mezi dvěma stavy můžeme popsat nějakou matematickou funkcí. Dost často se na to používají stavové rovnice pro chování plynů[7] ve tvaru $p = \ldots$ Toto obecné tvrzení ukážeme na stavové rovnici ideálního plynu $pV = nRT$, ze které vyjádříme tlak jako $p = nRT/V$. Tlak je tedy pro případ ideálního plynu funkcí počtu částic (resp. látkového množství n), teploty T a objemu systému V. $R = 8{,}314$ J/K/mol je univerzální plynová konstanta.

[7] Proto se těmto rovnicím říká stavové – popisují stavy systémů.

Na závěr tohoto dílu uvedeme jeden příklad, na kterém si můžete ověřit znalosti, o kterých jsme psali v tomto díle, a také prozkoumat kesonovou nemoc.

Úloha 4 [2,5b + 1,5b + 4b]: *Potápěči trpí tzv. kesonovou nemocí. Tkáně a tělní tekutiny potápěče (zejména tuk a krev) mají schopnost rozpustit v sobě plyny. Pokud dojde k poklesu tlaku kolem těla potápěče, plyn má tendenci z tkání unikat do kapilár a vytvářet tam bublinky, čímž může docházet i k embolii. Embolie je efektivně ucpání cévy, což zamezí prokrvení nějaké části těla. Pro nás je zajímavé, že i vzduchová bublina umí cévu ucpat. Více se můžete dozvědět třeba na Wikiskriptech*[8].

1. *Jedno z potenciálních nebezpečí vzniku embolie může být zvětšování těchto bublin. Kolikrát se zvětší poloměr bubliny dusíku v krvi potápěče, když potápěč rychle vystoupá z hloubky 22 m na hladinu vody? Tlak u hladiny považujte za atmosférický a teplota vody je všude v našem potřebném rozmezí stejná, odpovídá tělu potápěče – 36 stupňů Celsia. Zamyslete se, zda v této modelové situaci přispívá komprese/expanze plynu nějak výrazně k embolii.*

2. *Uvažujte stejnou situaci jako v případě 1 s jediným rozdílem: periferie těla potápěče v hloubce 22 m mají teplotu 35 stupňů a na hladině se již ohřejí na 37 stupňů. Porovnejte výsledky bodu 1 a 2.*

3. *Jak velkou práci musí výše zmíněná bublina vykonat, pokud je její průměr u hladiny vody roven průsvitu (vnitřnímu průměru) kapilární cévy – 7 μm? Vzduch v krvi lze považovat za ideální plyn ($V_m = 22{,}4$ l/mol) a teplotu vody považujte za konstantní jako v případě 1. Zamyslete se nad tím, zda by mohl být tento děj v realitě izotermní a proč ano/ne. Okomentujte také, co znamená, že vám práce vyšla kladná/záporná.*

Závěrem

Podívali jsme se na to, proč je dobré rozumět konceptu energie a zkusili jsme hlouběji proniknout do prvního termodynamického zákona. V posledním příkladu jsme se zkusili podívat na příklad expanze plynu. Co nás čeká v dalším díle? Podíváme se na teplo a nultý termodynamický zákon a čeká nás experiment s tepelnou vodivostí. Zjistíme, že zachytit teplo tak, aby neunikalo ze systému, je docela náročný úkol, který řeší jen *perpetuum mobile*.

Pája a Pavel; `termodynamika@ledoian.cz`
odevzdávejte do odevzdávátka

[8] `https://tinyurl.com/embolie`

Téma 2 – Programování a dokazování v Leanu

Díl 1: Výpočty

Nejdřív vám řeknu, proč je L∃∀N (dále psáno jako „Lean“) senzační jazyk, který stojí za to se naučit! Uvedu to na příkladu prvočísel.

1. V Leanu můžeme napsat program, který vygeneruje všechna prvočísla od jedné do tisíce. To asi jako v každém programovacím jazyce.

2. V Leanu můžeme napsat důkaz, že náš program vygeneruje všechna prvočísla od jedné do tisíce a nic jiného. Počítač zkontroluje, že je ten důkaz správně. Poté můžeme říci, že je náš program tzv. formálně zverifikovaný.

3. V Leanu také můžeme napsat důkaz, že existuje nekonečně mnoho prvočísel. To je abstraktní (matematické) tvrzení, které není možné dokázat tím, že prvočísla generujeme. Lean však nabízí nástroje, pomocí kterých lze takováto tvrzení zformulovat, a poté je matematicky dokázat. Počítač následně zkontroluje, že je ten důkaz správně.

Je užitečné, že ve stejném jazyce můžeme dělat výpočty i důkazy. V tomto tématku se budeme soustředit na úkoly typu 1 a 3.

Nebudete potřebovat žádné předchozí znalosti programování. Budete akorát potřebovat základní znalosti matematiky a schopnost psát na klávesnici (se zprovozněním Leanu na vašem počítači můžete někoho požádat o pomoc – není úkolem porozumět tomu, jak se Lean instaluje).

Počáteční díly tématka budou kurzem do základů funkcionálního programování[9]. To by mělo být zajímavé zejména pro čtenáře, kteří doposud znali jen imperativní programování[10] a rádi by ochutnali něco z úplně jiného světa. Také je však možné použít toto tématko jako úvod do programování pro úplné začátečníky (v takovém případě bude Lean vaším prvním programovacím jazykem; to, že se s funkcionálním programováním setkáte dříve než s imperativním programováním, ničemu nevadí).

To, že tématko nevyžaduje předchozí znalosti, však neznamená, že tématko bude snadné. Uvidíte, že obtížnost úloh bude podstatně narůstat. Důrazně doporučujeme, abyste si průběžně zkoušeli řešit příklady na počítači a nečetli tento text „na jedno posezení“.

[9] programování pomocí funkcí v matematickém slova smyslu – funkce (neboli zobrazení) je předpis, který přiřazuje vstupům výstupy; nic jiného nemůže dělat

[10] programování pomocí příkazů (tj. kroků, které se postupně vykonávají a mohou měnit stav programu)

Instalace

Existuje řada možností, jak používat Lean 4. Nejrychlejší (ale nejméně dokonalou možností z těch, co tu doporučujeme) je editor ve webovém prohlížeči, o který se stará Alexander Bentkamp z Universität Düsseldorf. Editor najdete na adrese: `https://lean.math.hhu.de`

Výhodou je, že si nemusíte nic instalovat na vlastní počítač a není potřeba se ani registrovat; jen kliknete na odkaz a píšete. Nevýhodou je, že editor neukládá vaši práci (takže před zavřením prohlížeče nebo záložky musíte zkopírovat zdrojový kód a uložit ho do textového souboru). Tento editor taktéž neumožňuje mít více souborů se závislostmi mezi nimi, takže pokud budete využívat něco z toho, co se objevilo v časopise, místo řádku `import mam.Cislo1` budete muset zkopírovat příslušné části našeho kódu na začátek vašeho kódu v prohlížeči (včetně funkcí a importů, na kterých příslušný kód závisí), což je značně nepohodlné. Poslední potenciální nevýhoda je, že se stránka automaticky aktualizuje, takže se může stát, že náš nebo váš kód začne zničehonic dělat jiné věci, pokud se změní důležitá část externí knihovny. Navzdory těmto nevýhodám, pokud si chcete Lean 4 jen rychle zkusit, tohle je nejsnazší možnost.

Druhou možností je použít GitHub Codespaces. To je taktéž editor v prohlížeči, ale jedná se o plnohodnotné vývojové prostředí se vším všudy. K jeho použití je však potřeba založit si GitHub účet. Automaticky dostanete 60 hodin měsíčně[11] pro práci v GitHub Codespaces. Po přihlášení do vašeho GitHub účtu jděte na: `https://github.com/madvorak/lean-mam`

Klikněte na zelené tlačítko „<> Code“, poté na pravou záložku „Codespaces“ a nakonec na zelené tlačítko „Create codespace on main“. Nyní budete muset několik minut počkat. Když se načte vývojové prostředí a vy otevřete libovolný soubor s Lean kódem (například soubor `Cislo1.lean` ve složce `mam`), měl by se v pravé polovině obrazovky objevit panel „Lean Infoview“. Nyní klikněte například na řádek `#eval obvod_obdelnika 3 2` a v tom „Lean Infoview“ by se měl objevit[12] výsledek `10`. Od teď můžete editovat svojí vlastní kopii kódu; ostatní uživatelé (ani já) ji neuvidí.

[11] Limit byste si mohli navýšit na 90 hodin tím, kdybyste si nechali ověřit, že jste studenti: `https://education.github.com/discount_requests/application`

[12] Pokud ani po načtení všech věcí nebude vidět v „Lean Infoview“ výsledek, je něco špatně; zkuste jiný webový prohlížeč.

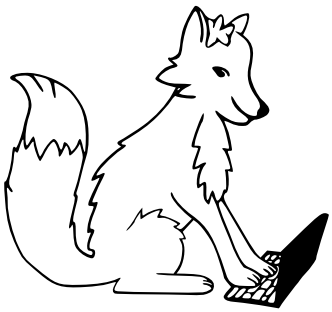
Třetí možností je nainstalovat si Lean 4 na vlastní počítač. Existují různé způsoby, jak Lean 4 a s ním spojené programy získat; my doporučujeme následující postup...

Abyste mohli používat Lean 4 standardním způsobem, je potřeba mít na svém počítači nainstalován Git. Pokud máte Windows, stáhněte[13] si Git na adrese: `https://gitforwindows.org/`

Proklikejte se instalátorem; kdykoliv se na něco zeptá, nechejte zvolenou výchozí možnost. Pak si nainstalujte prostředí Visual Studio Code od Microsoftu: `https://code.visualstudio.com/download`

Restartujte počítač. Otevřete Visual Studio Code a v levém panelu klikněte na „Extensions“ (má to ikonku čtyř malých čtverečků). Napište „lean4“ a najde vám to plugin „lean4“ s popisem „Lean 4 language support for VS Code“. Klikněte na něj a zvolte „Install“. Poté se přepněte do okna internetového prohlížeče a jděte na náš repozitář: `https://github.com/madvorak/lean-mam`

Po otevření odkazu můžete kliknout na zelené tlačítko „<> Code“ a poté na „Download ZIP“[14]. Rozbalte stažený soubor `lean-mam-main.zip` do nějaké složky na vašem počítači. Teď ve Visual Studio Code klikněte vpravo nahoře na „File“ a poté na „Open Folder...“. Najděte (rozbalený) projekt ve vašem počítači a otevřete ho. V levém panelu najděte soubor s Lean kódem a otevřete ho (například soubor `Cislo1.lean` ve složce `mam`). Visual Studio Code by se vás teď mělo zeptat, zda chcete nainstalovat „Elan“ (vyskočí zpráva v pravém dolním rohu, kde se nejspíš bude na prvním řádku psát „Failed to start 'lean' language server“ a pod ní budou dvě tlačítka). Klikněte na modré tlačítko „Install Lean using Elan“. Tím nainstalujete Lean 4 spolu s balíčkovacím systémem Lake a dalšími programy. Nyní budete muset počkat několik minut, než instalace doběhne. Když poté kliknete dovnitř okna s kódem, mělo by se napravo objevit „Lean Infoview“. Nyní klikněte například na řádek `#eval obvod_obdelnika 3 2` a v tom „Lean Infoview“ by se měl objevit výsledek `10`.



[13] Máte-li jiný operační systém, najděte si instalátor Gitu pro váš operační systém.

[14] Pokud znáte Git, můžete použít `git clone https://github.com/madvorak/lean-mam.git` namísto stahování ZIP. V každém případě je však pro kompilaci kódu v Leanu potřeba mít Git nainstalovaný, i když ho nebudete aktivně používat.

Pokud vám cokoliv z toho nefunguje, ozvěte se nám prosím přes e-mail nebo Discord. Pokusíme se vám udělat technickou podporu přes online hovor se sdílením obrazovky. Budu-li mít při hovoru zapnutou webkameru, možná vběhnou do záběru morčata.

Syntax

Základem programování v Leanu jsou funkce. Někoho možná překvapí, že se při volání funkce nedávají argumenty do závorky a neoddělují se čárkou. V matematice na papíře píšeme $f(x,y)$. V Leanu píšeme `f x y` bez závorky. Je potřeba nezapomínat na mezery, neboť identifikátory[15] mohou být víceznakové (jak je zvykem v programovacích jazycích), takže `fxy` by bylo jedno jméno.

Možná se ptáte, když nepíšeme argumenty do závorky a neoddělujeme je čárkou, jak se v Leanu napíše třeba $f(1, g(2, 3), h(4, 5), 6)$. Zde jsou už závorky potřeba, ale píší se okolo celého volání. Napíšeme `f 1 (g 2 3) (h 4 5) 6` a tím to je jednoznačné. Všimněte si, že při zanořování funkcí do sebe máme v Leanu o závorku méně (v tom základním případě). V matematice na papíře píšeme $f(f(f(x)))$. V Leanu píšeme `f (f (f x))`.

Podobně jako v matematice na papíře mají i v Leanu některé standardní funkce infixovou notaci[16]. V matematice na papíře píšeme $2f(x+1, y)$. V Leanu píšeme `2 * f (x + 1) y`. Symbol pro násobení (hvězdičku) není možné vynechávat, neboť dva termy[17] oddělené mezerou by byly dva argumenty (vstupující do nějaké funkce napsané nalevo od nich) nebo funkce a její argument (byť zde je jasné, že `2` není jméno funkce), a jak už víte, bez mezery bychom měli jeden identifikátor či syntaktickou chybu (neexistující identifikátor).

[15] Identifikátory jsou názvy funkcí, argumentů, typů, ... Říkáme jim tak, protože identifikují (jednoznačně), na co se odkazujeme.

[16] Například pro označení sčítání píšeme $1 + 2 = 3$ namísto $+(1, 2) = 3$, což by sice podle některých lidí bylo také správně, ale hůře by se to četlo i psalo. Slovo „infix“ označuje, že se operátor (zde „+“) píše doprostřed mezi argumenty (zde „1“ a „2“). Psaní názvu funkce před její argumenty, jako se v Leanu píšou všechny funkce „pojmenované písmenky“, se nazývá prefixová notace. Pro zajímavost – staré kalkulačky používaly tzv. postfixovou notaci, kde by se zadalo `1 2 +` a objevil by se výsledek `3`.

[17] Slovo „term“ může být matoucí. Úplně cokoliv je totiž term. Číslo je term. Součin dvou čísel je term. Mnohočlen je term. Lomený výraz je term. Textový řetězec je term. Funkce je term. Množina je term. Typ je term. Typ typů je term. Nerovnice je term. Soustava rovnic je term. Matematická věta je term. A její důkaz je také term. Nejpodobnější slovo, který by vystihovalo, co znamená „term“, je asi „objekt“, ale tento pojem má v programování už jiný ustálený výraz, takže slovo „objekt“ nebudeme v tomto tématku používat.

Deklarace funkce

Teď si ukážeme, jak se definují nové funkce. První ukázkou programování v Leanu bude výpočet obvodu obdélníka:

```
def obvod_obdelnika (a b : Nat) : Nat := 2 * (a + b)
```

Začneme vždy klíčovým slovem `def` následovaným jménem funkce, kterou právě definujeme.

Poté uvedeme argumenty této funkce a jejich typ. Každý argument se píše do závorky jako `(jmeno : Typ)`. Pokud máme více argumentů stejného typu, tak je můžeme oddělit mezerou a jejich typ napsat jen jednou. Uvedená deklarace `(a b : Nat)` je tedy zkratka za `(a : Nat) (b : Nat)` akorát. Po uvedení všech argumentů napíšeme dvojtečku a typ výstupu. Zde je přehled základních datových typů:

typ	popis	příklady hodnot
`Nat`	přirozená čísla ($\mathbb{N}$)	`0, 1, 42`
`Int`	celá čísla ($\mathbb{Z}$)	`-9, 0, 5`
`Float`	desetinná čísla	`-0.5, 0.0, 8.63`
`Bool`	logická hodnota	`false, true`
`String`	textový řetězec	`"", "dvě slova"`

Doteď jsme mluvili o typu. Po napsání „`:=`“ přichází na řadu hodnota. Zde napíšeme výraz (dle syntaktických pravidel, které jsme si už vysvětlili) správného typu. Lze se odkazovat na argumenty a na dříve definované funkce.

Pokud si chceme zkusit, co naše nová funkce dělá, napíšeme (kamkoliv pod místo deklarace) `#eval obvod_obdelnika 3 2`. Zobrazí se odpověď `10`.

Úloha 1 [0,5b]: *Naprogramujte výpočet povrchu kvádru:*

$$S = 2 \cdot (ab + bc + ca)$$

Hlavička vaší funkce zní:
```
def povrch_kvadru (a b c : Nat) : Nat :=
```

Úloha 2 [1b]: *Naprogramujte výpočet obsahu trojúhelníku podle Heronova vzorce:*

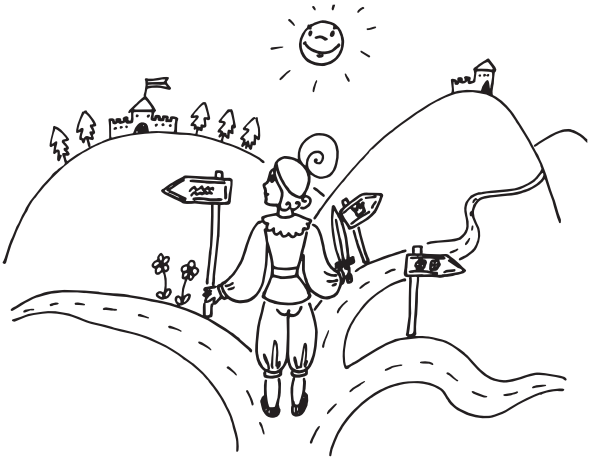
$$S = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)} \qquad \text{kde} \qquad s = \frac{a+b+c}{2}$$

Pro výpočet odmocniny používejte funkci `Float.sqrt` *pracující s desetinnými čísly. Hlavička vaší funkce zní:*
```
def obsah_trojuhelniku (a b c : Float) : Float :=
```

Jak řešit úlohy

Většinou je u úlohy uvedena hlavička funkce, kterou máte implementovat. Nemusíte ji ručně opisovat; podívejte se do souboru `Zadani1.lean` a uvidíte tam funkce (bez jejich implementace) a ukázky jejich volání. Občas je v komentáři (hned za `#eval` volání funkce) napsáno, jaký je vzorový výstup. Pro správné vyřešení úlohy je potřeba, aby vaše funkce dávaly správný výsledek na všech možných vstupech, nejen na těch ukázkových.

Vaše řešení doporučujeme psát přímo do souboru `Zadani1.lean` na místa označená komentářem `-- TODO`. Při řešení můžete používat (tj. odkazovat se na) cokoliv, co bylo v časopise. Kódy z časopisu (občas i s něčím navíc) najdete v souboru `Cislo1.lean`, který je souborem `Zadani1.lean` již naimportován. Můžete využívat i „standardní knihovny" (`core`, `std4`, `mathlib4`), ale vězte, že vše, co je k řešení úloh potřeba, je již naimportováno. Například soubor `Cislo1.lean` začíná řádkem `import Std.Data.Nat.Basic`, díky čemuž můžeme později využívat funkci `Nat.sqrt` pro výpočet odmocniny (z přirozeného čísla, zaokrouhlené dolů; viz následující podkapitola). Protože soubor `Zadani1.lean` importuje soubor `Cislo1.lean` na svém první řádku, funkce v souboru `Zadani1.lean` mohou používat `Nat.sqrt` taktéž.



Podmínka

Pro případy, kdy chceme dělat různé věci v různých situacích, máme v Leanu konstrukci if-then-else. Není to však stejný if-then-else, jako někteří znáte z imperativních programovacích jazyků. V Leanu neexistuje žádná posloupnost příkazů, která by se mohla podmíněně (ne)vykonávat; v těle funkce pouze deklarujeme, jaká je výstupní hodnota (a ta musí být definována pro všechny vstupy, takže nelze vynechat „else část"). Proto je zdejší if-then-else ve skutečnosti tzv. podmíněný výraz (funguje stejně jako ternární operátor „`?:`" v jazyce C).

Podmíněný výraz si uvedeme na (zcela nepraktickém) příkladu určení parity přirozeného čísla, kde výstup bude v podobě textového řetězce; buď napíšeme `sude`, nebo napíšeme `liche`.

```
def parita (n : Nat) : String :=
if n % 2 = 0
then "sude"
else "liche"
```

Část za `then` i část za `else` musí být výraz stejného (správného) typu. Tu část za `if` budeme nyní chápat spíše intuitivně. Můžeme zde používat relační operátory $=$, $\neq$, $<$, $>$, $\leq$, $\geq$ a další ($\in$, $\notin$, $\subseteq$, $\supseteq$, ... záleží na typu výrazů, které „porovnáváme“), nebo tady můžeme zavolat existující funkci vracející logickou hodnotu. Ve výše uvedeném příkladu porovnáváme výraz `n % 2`, tj. zbytek z čísla `n` po dělení dvěma, s přirozeným číslem nula.

Úloha 3 [1b]: *Objasněte, proč* `#eval parita (2 - 3)` *oznamuje* `sude`.

Jako příklad funkce vracející (tj. oznamující na výstupu) logickou hodnotu bude určení, zda je zadané přirozené číslo „čtverec“ (druhá mocnina nějakého přirozeného čísla).

```
def je_ctverec (a : Nat) : Bool := (Nat.sqrt a) ^ 2 = a
```

Hlavičku funkce snad není třeba komentovat. Pojďme si vysvětlit, jak funguje její tělo `(Nat.sqrt a) ^ 2 = a`. Druhá odmocnina se anglicky řekne „square root“, proto zkratka `sqrt` jako název funkce. Prefix `Nat.` označuje, že se uvažuje odmocnina nad přirozenými čísly; konkrétně je to dolní celá část druhé odmocniny z argumentu. Výraz `(Nat.sqrt a)` bychom v matematice na papíře zapsali jako $\lfloor\sqrt{a}\rfloor$. Následuje symbol „`^`“ (stříška), což je infixový operátor umocnění.

Například zadáním `a = 8` se vypočítá `(Nat.sqrt 8) ^ 2 = 4` a v porovnání se zadanou hodnotou `8` to dá výstup `false` (tj. logická hodnota nepravda).

Úloha 4 [1b]: *Naprogramujte funkci, která určí, zda je zadané číslo čtvrtou mocninou nějakého přirozeného čísla. Hlavička vaší funkce zní:*
`def je_ctvrta_mocnina (a : Nat) : Bool :=`

Seznam malé velikosti

Řekněme, že na vstupu máme dvě celá čísla, která chceme vrátit jako rostoucí posloupnost. Tento příklad demonstruje, jak se zapisují v Leanu seznamy malé velikosti (zde konkrétně seznamy velikosti jedna nebo dva); píší se do hranatých závorek; je-li více než jeden prvek seznamu, oddělujeme je čárkou, která se však nepíše za poslední prvek seznamu. Všimněte si také použití „zanořeného“ příkazu `if` bez nutnosti závorek navíc:

```
def dvojice_rostouci (x y : Int) : List Int :=
if x = y
then [x]
else if x < y
     then [x, y]
     else [y, x]
```

Typ výstupu `List Int` značí seznam celých čísel. Seznamům v obecnosti budeme věnovat celý příští díl.

Úloha 5 [2b]: *Naprogramujte řešení kvadratické rovnice $ax^2+bx+c=0$. Hlavička vaší funkce zní:*

```
def reseni_kvadraticke_rovnice (a b c : Float) : List Float :=
```

Z technických důvodů, které tu nechceme rozebírat, nelze desetinná čísla porovnávat operátorem „`=`“. Pro vyhodnocení, zda se dvě desetinná čísla rovnají, je potřeba použít operátor „`==`“. Tato operace je nebezpečná a obecně se nedoporučuje dělat, neboť počítač má při práci s desetinnými čísly zaokrouhlovací chyby, které není snadné předvídat. My však pro tentokrát slibujeme, že při používání relačních operátorů (`==`, `<`, `>`) nad typem `Float` v této úloze nenastanou žádné podivnosti (není to chyták).

Rekurze

Rekurze, tj. schopnost funkce odkazovat se sama na sebe, je srdcem funkcionálního programování. Vyjadřují se pomocí ní cykly a další věci. Pojďme si rekurzi ukázat na oblíbeném příkladu, kterým je faktoriál.

Funkce faktoriál, značená pomocí vykřičníku za přirozeným číslem, je součin přirozených čísel od 1 po zadané číslo. Například faktoriál pěti je:

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

Obecně to můžeme zapsat takto:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

Všimněme, co se stane, když se číslo, ze kterého počítáme faktoriál, zvýší o jedničku:

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 4 \cdot (1 \cdot 2 \cdot 3) = 4 \cdot 3!$$

Pro obecné n pak platí

$$(n+1)! = (n+1) \cdot n!$$

jak dokážeme nahlédnout jednoduchým přeuspořádáním součinu.

Udělejme teď myšlenkový obrat! Šlo by výše uvedený vztah použít, místo pozorování o vlastnosti faktoriálu, jako definici faktoriálu? Vskutku, pokud bychom již věděli, čemu se $n!$ rovná, mohli bychom zadefinovat hodnotu $(n+1)!$ pomocí výše uvedeného vztahu.

Dodefinujme si teď hodnotu $0! = 1$ (to se může zdát na první pohled divné, ale kdybychom tak neučinili, rozbila by se tím spousta matematiky). Teď budeme počítat:

$$\begin{aligned}
1! &= (0+1)! = (0+1) \cdot 0! = 1 \cdot 1 = 1 \\
2! &= (1+1)! = (1+1) \cdot 1! = 2 \cdot 1 = 2 \\
3! &= (2+1)! = (2+1) \cdot 2! = 3 \cdot 2 = 6 \\
4! &= (3+1)! = (3+1) \cdot 3! = 4 \cdot 6 = 24 \\
5! &= (4+1)! = (4+1) \cdot 4! = 5 \cdot 24 = 120 \\
6! &= (5+1)! = (5+1) \cdot 5! = 6 \cdot 120 = 720
\end{aligned}$$

Vidíme, že funkce faktoriál roste rychle. A taky vidíme, že lze všechny hodnoty faktoriálu určit pomocí těchto dvou vztahů:

$$\begin{aligned}
0! &= 1 \\
(n+1)! &= (n+1) \cdot n!
\end{aligned}$$

Není tu žádná nejednoznačnost v tom, který vztah máme použít; horní vzorec určuje hodnotu pro nulu, zatímco dolní vzorec určuje hodnotu právě pro všechna nenulová čísla.

Takže to je plnohodnotná definice faktoriálu. Naprogramujme ji teď v Leanu:

```
def faktorial : Nat → Nat
| 0   => 1
| n+1 => (n+1) * (faktorial n)
```

Možná jste si v ukázce výše všimli symbolu „→“ v deklaraci typu funkce. To není tisková chyba. V kódu se nachází Unicodový symbol šipky. Až ho někdy budete potřebovat napsat sami, mohli byste samozřejmě jít do naší ukázky a zkopírovat ho, ale lepší bude, když se naučíte „→“ psát na klávesnici. Ve vývojovém prostředí ho vytvoříte tak, že napíšete `\r` a stisknete tabulátor. Pokud chcete napsat symbol „→“ následovaný mezerou (což většinou budete chtít), můžete tuto dvojici symbolů snáze vytvořit tak, že napíšete `\r` a stisknete mezerník (tedy nedojde ke stisknutí tabulátoru).

Pokud narazíte na jiný speciální symbol (na ten vyžadující Unicode, jako třeba ta šipka doprava), který byste chtěli umět sami napsat, můžete nad ním (ve kterémkoliv z doporučených vývojových prostředí) podržet myš a zobrazí se,

co musíte napsat na klávesnici (uvádí se to tam včetně toho počátečního zpětného lomítka, ale bez uvedení následného tabulátoru/mezery). Občas tam uvidíte více možností (třeba ta šipka „→“ jich tam má hned šest), ale je lepší se dívat jen na tu první uvedenou možnost a tu si zapamatovat.

Po hlavičce následuje tzv. pattern matching. Před každý pattern se píše „|“ (svislítko). Mezi patternem a přiřazeným výstupem se píše dvojice symbolů „=>“ (rovnítko a většítko). To jsou standardní ASCII znaky.

Podobným způsobem, jako jsme rekurzivně naprogramovali faktoriál, zadefinujeme Fibonacciho čísla (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...). To je posloupnost přirozených čísel začínající nulou a jedničkou, ve které je každé další číslo rovno součtu předchozích dvou čísel:

```
def fibonacci : Nat → Nat
| 0   => 0
| 1   => 1
| n+2 => fibonacci n + fibonacci (n+1)
```

Všimněte si, že je opět jednoznačné, který pattern se použije.

Doteď pro nás byly termy typu `Nat`, tj. přirozená čísla, atomická; nedívali jsme se na ně jako na sekvenci číslic. Lze v takovém rámci definovat ciferný součet?

Ano, lze. Všimněme si třeba, že poslední cifru přirozeného čísla získáme jako zbytek po dělení deseti. Předchozí cifry posčítáme tak, že předhodíme celočíselnou desetinu ze zadaného čísla rekurzivnímu zavolání. Ta rekurzivní volání musí však někde skončit, jinak bychom se výsledek nikdy nedozvěděli. Stane se tak, když dojdeme k jednocifernému číslu. Pro teď ignorujte klíčové slovo `partial` na začátku hlavičky funkce:

```
partial def ciferny_soucet (a : Nat) : Nat :=
if a < 10
then a
else (a % 10) + ciferny_soucet (a / 10)
```

Úloha 6 [2b]: *Ciferace je ciferný součet opakovaný tak dlouho, až vznikne jednociferné číslo. Například ciferace čísla 919 je 1, protože ciferný součet 919 je 19, ciferný součet 19 je 10, ciferný součet 10 je 1. Naprogramujte funkci pro výpočet ciferace. Hlavička vaší funkce zní:*

```
partial def ciferace (a : Nat) : Nat :=
```

Teď se zaměříme na technickou úlohu, v níž dostaneme funkci $f : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ a máme za úkol spočítat součet jejích výstupních hodnot pro všechny vstupy (i, j), kde i a j jsou přirozená čísla menší než zadané n. Jinými slovy, na vstupu je funkce f a číslo n a my máme za úkol vypočítat hodnotu:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f(i, j)$$

Naši funkci pojmenujeme `suma_pres_ctverec` (to není název vstupní funkce – ta se jmenuje f) a naprogramujeme ji následovně:

```
private def suma_pres_radek (f : Nat → Nat → Float) (m : Nat) :
    Nat → Float
| 0   => 0.0
| n+1 => (f m n) + (suma_pres_radek f m n)

private def suma_pres_obdelnik (f : Nat → Nat → Float) (n : Nat) :
    Nat → Float
| 0   => 0.0
| m+1 => (suma_pres_radek f m n) + (suma_pres_obdelnik f n m)

def suma_pres_ctverec (f : Nat → Nat → Float) (n : Nat) : Float :=
suma_pres_obdelnik f n n
```

Všimneme si několika věcí:

- Funkce mohou být argumenty funkcí. To by nemělo být velkým překvapením; funkce je term a každá funkce si může libovolně specifikovat, jaké typy termů bere jako svoje argumenty. Říká se tomu, že `suma_pres_ctverec` je tzv. funkce vyššího řádu.
- Je tu použit tzv. currying. Funkce typu $A \times B \to C$ vyjadřujeme jako funkce $A \to (B \to C)$ a díky implicitní asociativitě „$\to$“ to zapisujeme jako $A \to B \to C$ ve stejném významu. Ve funkcionálním programování je zvykem používat currying při deklaraci všech funkcí, které by konceptuálně braly víc než jeden argument.

  Výhodou je, že takto definované funkce můžeme tzv. částečně aplikovat. Když `f` je funkce typu `Nat → Nat → Float`, tak po dosazení hodnoty (například čísla 42) za první argument (formálně za její jediný argument) vznikne funkce `f 42 : Nat → Float` a po dosazení dalšího přirozeného čísla dostaneme desetinné číslo na výstupu.
- Kýžená funkce `suma_pres_ctverec` má pouze dva argumenty, ale k jejímu výpočtu jsme zadefinovali pomocnou funkci, která bere tři argumenty. Takové věci se často hodí udělat!

- Pomocné funkce kombinují pojmenované argumenty s pattern matchingem na posledním argumentu (ten není pojmenovaný v hlavičce funkce, nýbrž jeho jméno plyne z výrazu napsaného mezi „|“ a „=>“).
- Pomocné funkce mají klíčové slovo `private` před svou deklarací. To určuje, že když soubor `Cislo1.lean` naimportujeme v jiném souboru, pomocné funkce `suma_pres_radek` a `suma_pres_obdelnik` tam nebudou viditelné (nepůjdou z ostatních souborů zavolat).

Úloha 7 [3b]: *Nechť je dána funkce g, která trojici přirozených čísel přiřazuje přirozené číslo. Naprogramujte hledání maximální hodnoty funkce g na množině $\{0,\ldots,n-1\}^3$ neboli:*

$$\max_{i<n}\max_{j<n}\max_{k<n} g(i,j,k)$$

Celkem tedy musíte projít n^3 hodnot. Hlavička vaší funkce zní:

```
def maximum_z_krychle (g : Nat → Nat → Nat → Nat) (n : Nat) :
  Nat :=
```

Úloha 8 [2b]: *Naprogramujte funkci, která určí, zda zadané přirozené číslo je prvočíslo. Hlavička vaší funkce zní:*

```
def je_prvocislo (a : Nat) : Bool :=
```

Úloha 9 [2b]: *Naprogramujte funkci, která určí, zda zadané přirozené číslo je dokonalé číslo. Číslo je dokonalé, pokud je rovno součtu všech vlastních dělitelů. Například* $6 = 1 + 2 + 3$ *je dokonalé číslo. Oproti tomu* 8 *není dokonalé číslo (má moc nízký součet vlastních dělitelů). Stejně tak* 12 *není dokonalé číslo (má moc vysoký součet vlastních dělitelů). Hlavička vaší funkce zní:*

```
def je_dokonale_cislo (a : Nat) : Bool :=
```

Technické detaily

Tuto sekci můžete klidně přeskočit. Budeme se tu bavit o tom, co je povoleno dělat při pattern matchingu a co se zhruba tak nachází za kódem, který píšeme. Pokud se vám při definování rekurzivních funkcí bude odmítat zkompilovat kód, který jste napsali, můžete se pokoušet hledat odpověď na váš problém v této sekci. Jednodušší ale asi bude, pokud mi napíšete e-mail s dotazem a přiložíte kód, který vám nefunguje.

Typ `Nat` má dva konstruktory (ty společně tvoří konstruktivní definici přirozených čísel):

- `Nat.zero : Nat` „nula“
- `Nat.succ : Nat → Nat` „následník“

Pojďme se podívat, co se dělo, když jsme definovali rekurzivní funkci na přirozených číslech. Když jsme naprogramovali faktoriál

```
def faktorial : Nat → Nat
| 0   => 1
| n+1 => (n+1) * (faktorial n)
```

Lean přeložil náš pattern matching na něco jako následující definici (toto není platná deklarace v Leanu; následující dva řádky chápejte jen jako přirovnání):

```
faktorial (Nat.zero)   := (Nat.succ Nat.zero)
faktorial (Nat.succ n) := (Nat.succ n) * (faktorial n)
```

Napsáním pattern matchingu

`| 0 =>` konstanta	obraz `Nat.zero`
`| n+1 =>` výraz závisející na `n`	obraz `(Nat.succ n)`

dáváme kompilátoru návod na to, jak má „obsloužit“ oba konstruktory. Podobně pattern matching se třemi větvemi

`| 0 =>` konstanta	obraz `Nat.zero`
`| 1 =>` konstanta	obraz `(Nat.succ Nat.zero)`
`| n+2 =>` výraz závisející na `n`	obraz `(Nat.succ (Nat.succ n))`

opět určuje, jak se zobrazí všechna čísla.

Kompilátor zkontroloval, že je výčet vyčerpávající. Můžete si zkusit, že pokud bychom to změnili na

```
| 0   => ...
| 1   => ...
| n+3 => ...
```

kompilátor by si postěžoval

```
missing cases:
(Nat.succ (Nat.succ Nat.zero))
```

aby dal najevo, že funkce není definovaná na všech přirozených číslech (chybí zde obraz dvojky). Výrazy jako `n+1` mají v pattern matchingu speciální roli, protože jsou syntaktickou zkratkou za konkrétní konstruktor přirozených čísel. Nemůžeme napsat `n*2` na levou stranu od „`=>`“ v naději, že tenhle pattern matchne všechna sudá čísla – na to bychom museli použít podmínku (na pravé straně od „`=>`“ nebo úplně bez pattern matchingu).

Když definujeme rekurzivní funkci, kompilátor se automaticky snaží dokázat, že rekurze někdy doběhne pro každý vstup (je to tzv. well-founded recursion). Pokud se mu to nepodaří, odmítne funkci zkompilovat a řekne nám, že od nás požaduje důkaz terminace, což zatím nevíme, jak by se dělalo. Klíčové slovo `partial` vypíná toto chování. Automatické odvození terminace uspěje v základních případech, například když `f (n+1)` volá `f n` jako jediné rekurzivním voláním v deklaraci funkce `f`. V dílech o programování se tím nebudeme trápit – kdykoliv kompilátor nebude umět odvodit, že funkce doběhne, přidáme `partial` před její definici.

Pattern matching lze taky dělat na více argumentech. V takovém případě se oddělují čárkou. Opět je potřeba pokrýt všechny kombinace konstruktorů. Příkladem využití pattern matchingu na dvou argumentech je Ackermannova funkce[18]:

```
partial def ackermann : Nat → Nat → Nat
| 0  , n   => n+1
| m+1, 0   => ackermann m 1
| m+1, n+1 => ackermann m (ackermann (m+1) n)
```

[18] `https://cs.wikipedia.org/wiki/Ackermannova_funkce`

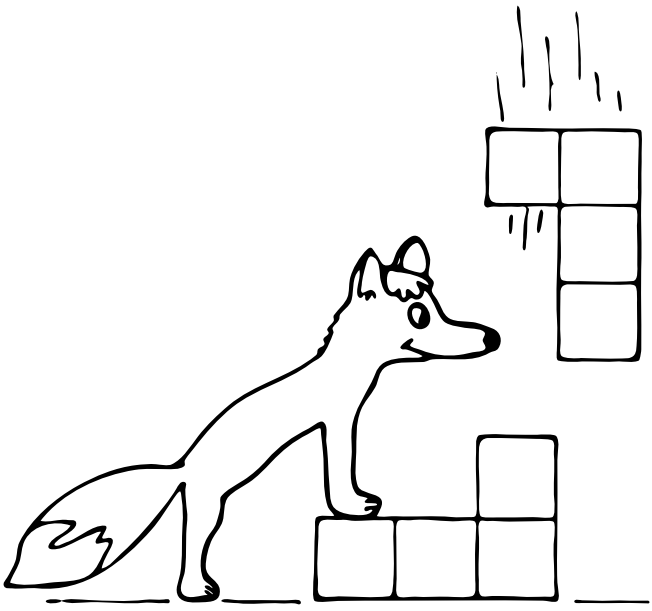
Výzva

Tohle zatím bylo programování. Pokud se nemůžete dočkat, až začneme s dokazováním matematických tvrzení, můžete si zatím zahrát Natural Number Game: `https://adam.math.hhu.de/#/game/nng`

Za hraní Natural Number Game vás nemůžeme odměnit body do M&M, ale můžete o tom napsat článeček, který otiskneme v našem časopise, a za to už body dostanete! Níže jsou uvedené náměty na článeček, který můžete napsat. Natural Number Game je relevantní pro první dva náměty:

- Dojmy z hraní Natural Number Game
- Axiomatizace přirozených čísel
- Jak zadefinovat vlastní infixový operátor
- Rozdíly mezi imperativním a funkcionálním programováním
- Co je to lambda kalkulus
- Co je to teorie typů
- Co je to intuicionistická logika

Martin Dvořák; `martin.dvorak@matfyz.cz`
odevzdávejte do odevzdávátka



Téma 3 – Hex

Piškvorky vás (už) nebaví? Na šachy / dámu / „Člověče, nezlob se!“ se vám nechce tahat figurky a herní plochu? Je čas na novou a lepší hru!

Hex je kompetitivní desková hra pro dva hráče s jednoduchými pravidly, kterou lze hrát pouze s papírem a tužkou. Spočívá v pokládání kamenů (vybarvování políček) s cílem spojit své strany / zabránit protihráči ve spojení jeho stran.

My se v tomto tématku podíváme na to, kdy vyhrává který hráč, a celkově na to, jak Hex hrát co nejlépe. Tématko bude hravé, ale stejně jako v matematice každý výsledek musíme řádně odůvodnit.

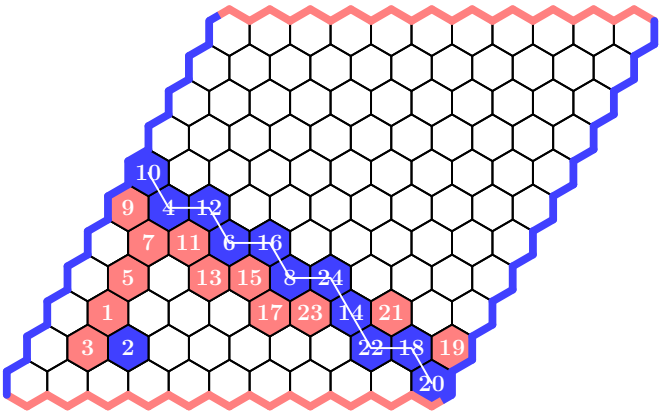
Díl 1: Pojďme (si) hrát

Pravidla

Hex se hraje na šestiúhelníkové síti, kde je vyznačeno, které části má který hráč spojit. V základní hře je tvar plochy kosočtvercový (viz obrázek 1), většinou 11×11 políček, a hráči mají spojit protější strany tohoto kosočtverce. Jiné hrací plochy však budou hlavním předmětem našeho zkoumání.

Počínaje prvním hráčem, BÚNO červeným[19], se hráči střídají ve vybarvování políček svými barvami (případně pokládání kamenů své barvy), s tím, že jakmile je políčko jednou vybarveno (hrací kámen položen), již jej nelze znovu obarvit.

Jakmile některý z hráčů spojí[20] své dvě vyznačené části herní plochy (v základní hře dvě protější strany kosočtverce), hra končí a tento hráč vyhrává. Jednu takovou hru si můžete prohlédnout na obrázku 1.

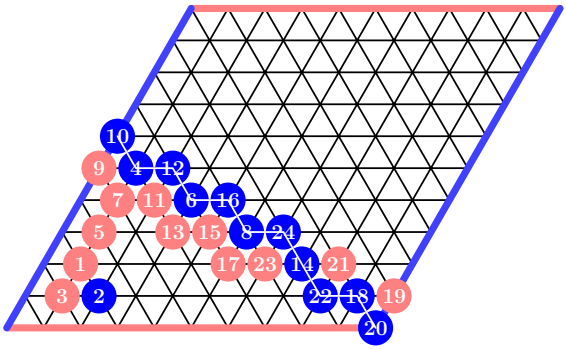


Obrázek 1: Odehraná hra na základní herní ploše, jak hra probíhala je naznačeno čísly, vyhrál modrý hráč

[19] V černobílé verzi světlejší.

[20] Spojí = vytvoří posloupnost políček, kde vždy dvě následující políčka spolu sousedí (políčka spolu sousedí, sdílejí-li stranu) a první a poslední políčko leží u daných částí.

Šestiúhelníkovou síť si můžete stáhnout na `https://mam.matfyz.cz/media/prilohy/30-1-a4.pdf` a vytisknout. V případě, že byste si chtěli hru zahrát a nemáte u sebe zrovna šestiúhelníkovaný papír, můžete si všimnout, že obarvovat šestiúhelníky v šestiúhelníkové síti je to samé jako obarvovat vrcholy v trojúhelníkové síti, která se rýsuje rozhodně snadněji. Viz obrázek 2.



Obrázek 2: Stejně odehraná hra, tentokráte na trojúhelníkové herní ploše

Zadání

Při řešení tohoto dílu si nic nedohledávejte na internetu, na to přijde čas později! U problémů se boduje jak množství, tak „kvalita“ (zajímavost, použitelnost, kreativita, ...).

Za vítěznou strategii považujeme postup pro konkrétního hráče, který vede k vítězství, ať bude hrát protihráč jakkoliv. Je celkem jedno, zda takový postup popíšete formálně (např. zobrazení z pozic do konkrétních tahů) nebo slovně (např. budeme hrát symetricky jako protihráč), vždy by ale mělo být jasné, co má hráč držící se této strategie hrát. Kromě toho vždy pečlivě zdůvodněte, proč tímto postupem dosáhneme výhry, ať bude protihráč dělat cokoliv.

Úloha 1 [4b]: *Dokažte, že v základní hře (na kosočtvercové* 11×11 *herní ploše), ať budou hrát hráči jakkoliv, musí jeden z hráčů vyhrát.*

Ukažte navíc, že vítěznou strategii má červený (začínající) hráč. (Není třeba najít konkrétní vítěznou strategii, jen dokázat, že ji červený má. Nápověda: Může se hodit první část úlohy.)

Plný počet bodů dostanete pouze za důkaz, který opravdu nic neopomíjí. Za zobecnění na další herní plochy můžete získat bonusové body navíc.

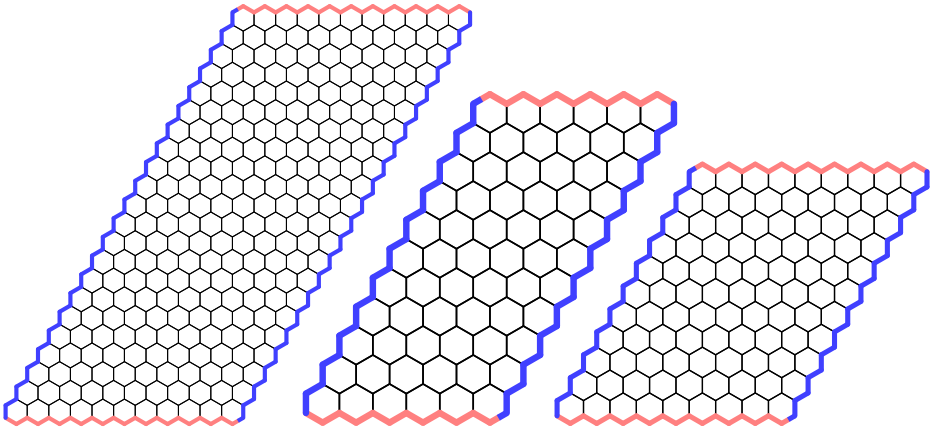
Pro základní hru (11×11) není vítězná strategie známa, ale pro herní plochu o jednom políčku máme jasnou vítěznou strategii: prostě zahrajeme svoji jedinou možnost. Jak moc můžeme hru zvětšit, abychom ještě uměli vyhrát za začínajícího hráče? A jsou nějaké jiné tvary herních ploch, kde máme nějakou zajímavou vítěznou strategii?

Problém 2: *Nalezněte herní plochy, na kterých umíte ukázat konkrétní vítěznou strategii za jednoho z hráčů.*

Pro základní hru platí, že vyhraje začínající hráč. Co když ale změníme herní plochu na nějakou jinou? Například:

Problém 3: *Nalezněte herní plochy, kde má modrý hráč tak velkou výhodu, že vyhraje, ať bude červený hrát jakkoliv, přestože nezačíná.*

Můžete například zkusit zjišťovat, pro které rozměry kosodélníků (obrázek 3) vyhrává červený a pro které vyhrává modrý. (Není nutné toto rozhodnout hned pro všechny, ale např. „Když to bude mít modrý 42krát blíže, tak už vyhraje, a když to bude mít o jedna blíže, tak vyhraje červený[21]. Nic mezi nevím.“ Ale samozřejmě se boduje, které všechny kosodélníky umíte rozhodnout.)



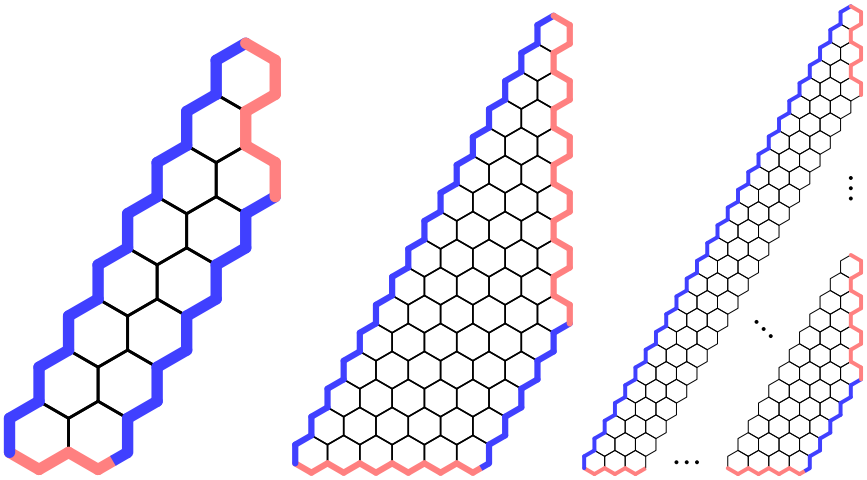
Obrázek 3: Různé tvary kosodélníkové hrací plochy

Problém 4: *Vymyslete zajímavé herní plochy nebo úpravy pravidel, které bychom mohli zkoumat. Než řešení tohoto problému odevzdáte, zamyslete se, zda byste vůbec takto upravenou hru chtěli hrát za libovolného hráče (např. „vždy hraje červený a modrý se jen kouká“ není moc zajímavá úprava).*

U herních ploch není nutné, aby měly dvě a dvě vyznačené oblasti, které mají hráči spojit. Můžete si s tím vyhrát (můžou spojovat stejné části, kus okraje plochy lze nechat jako neutrální území, atd.). V takovém případě ale uveďte, co tedy musí hráč spojit pro výhru (např. pokud bude vyznačených částí více, tak jestli stačí spojit libovolné dvě, nebo chcete spojit všechny naráz, nebo nějaké konkrétní dvojice).

[21] Toto tvrzení nemusí být pravda, nevím, kdo vyhraje, když to bude o jedna.

Pro odvážlivce nabízíme na obrázku 4 i pravděpodobně velmi obtížný problém.



Obrázek 4: Ultimátní otázka: Mějme lichoběžník vzniklý z pravoúhlého trojúhelníku uříznutím vrcholu s pravým úhlem. Nechť jsou ramena lichoběžníku červená a základny modré. Existuje pro každou délku menší základny dostatečná výška lichoběžníku, při které už vyhraje červený?)

A nyní už s chutí do toho!

Bětka; betka.n@centrum.cz
Jidáš; jonas.havelka@volny.cz
odevzdávejte do odevzdávátka

Časopis M&M je zastřešen Matematicko-fyzikální fakultou Univerzity Karlovy. S obsahem časopisu je možné nakládat dle licence CC BY 4.0. Autory textů jsou, není-li uvedeno jinak, organizátoři M&M. Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy. Pokud si časopis nepřejete dále dostávat v tištěné podobě, zrušte si prosím jeho odběr v nastavení svého účtu na webu.

Kontakty:

M&M, OPMK, MFF UK
Ke Karlovu 3
121 16 Praha 2

E-mail: mam@matfyz.cz
Web: mam.matfyz.cz
FB: casopis.MaM