Binarni konstruktor P je v 1. kontra v 2. kova.
Na typech je usporadani A <=: A' <=: A"
a) P(PAA')A' <: P(PA"A)A" (Spravne)
b) P(PA'A')A <: P(PA"A")A" (Blbe)
atd.

Jak na to opravdu obecne...

_____
a) P ma dva parametry PAA' a A' na druhe strane PA"A a A"
Vzhledem k tomu ze je v 2. kova musi platit ze
A' <: A"
to je ok takze jdeme dale.
V 1. je kontra takze musi platit
PAA' >: PA"A (proste obratim co je podtyp)
proto musi platit ze v 2. je tedka kova, takze
A' >: A
OK
a v 1. kontra takze
A <: A"
OK
dostali jsme se az uplne nejdal a vsecko plati, takze celkove plati
(pripadne bych to aplikoval porad dal a dal kdybych tam mel jeste P).

_____
b) P(PA'A')A <: P(PA"A")A"
jedeme uplne stejne: v druhem kova
A <: A" OK
v 1. kontra:
PA'A' >: PA"A"
a zase
v 2. kova
A' >: A"
BLBE KONCIM
==========
polymorfizmus (obecne) je keď sa jedna metoda správa rôzne podľa toho ako ju použijeme (napríklad predok aj potomok majú rovnakú metodu, a podľa toho z koho ju voláme sa vie ktorá sa použije)

preťaženie (obecne) je keď sa viac metód volá (volá v zmysle že majú rovnaký názov a nie že sa volajú z nejakého miesta v programe) rovnako, a podľa parametrov alebo návratovej hodnoty sa vie rozhodnúť ktorá z nich sa má použiť.

Keďže reťazce sa porovnávajú inač ako čísla, ide o implementáciu dvoch rôznych metód. Obe sú ale ==. Tým pádom to je určite nejaké preťaženie. Je už jednoduché rozhodnúť sa medzi tými dvoma, a správna odpoveď je teda kontextovo nezávislé preťaženie.
==========
Jsou definovány typy: A = 1..3, B = 2..4, C = union {a:A | b:B}, D = record {c:C
| D:Bool}, E = array Bool of D. Počet všech hodnot typu E je:

Prikladam opat svoj cheat sheet:

Union (A | B) = A+B
Record (A | B) = A*B

Set of A = 2^A
Array A of B = B^A
A ------> B = (B+1)^A
A ------> B B^A (sipka je spojita)

A = 1..3
B = 2..4
C = union {a:A | b:B}
D = record {c:C| D:Bool}
E = array Bool of D

A - 3 hodnoty
B - 3 hodnoty
C - 6 hodnot lebo 2 z A je ina ako 2 z B
D - 12 hodnot
E - 12^2
==========
Nechť sigma je stav v imperativním jazyce, A, B, C, D nějaké jeho příkazy. Který z násldujících stavů se může lišit od ostatních?

(A) [|begin A;B;C;D end|]sigma
(B) [|D|]([|C|]([|B|]([|A|]sigma)))
(C) [|D|]([|begin C;B;A end|]sigma)
(D) [|begin C;D end|]([|begin A;B end|]sigma)
(E) [|begin C;D end|]([|B|]([|A|]sigma))

Podle mého názoru je správná odpověď C. Protože v ostatních se vyhodnucují příkazy postupně A, pak B, pak C, pak D. Ale v odpovědí C se vyhodnocují C, pak B, pak A, pak D.
==========
V jistém jazyce je příkaz cyklu s podmínkou uvnitř, do P; when e break; Q od kde P, Q jsou příkazy a e je boolovský výraz. Opakování cyklu je ukončeno jakmile je v místě when splněna podmínka e. Vyhodnocení výrazu e může mít vedlejší efekty, tj. může způsobit změnu stavu. Nechť sigma je stav, v němž se začne tento příkaz cyklu provádět. Při prvním průchodu cyklem bude mít výraz e hodnotu false, při druhém průchodu bude mít výraz e hodnotu true.
pak platí, že [|do P; when e break; Q od|] sigma je rovno

(A)[|begin P;Q;P end|]([|e|] sigma)
(B)[|e|]([|begin P;Q;P end|] sigma)
(C)[|e|]([|begin Q;P end|]([|e|]([|P|] sigma)))
(D)[|begin P;Q;P end|] sigma
(E)[|begin Q;P end|]([|e|]([|P|] sigma))

Moje řešení:
1) vyhodnoti se prikaz P ve stavu sigma ..... [|P|]sigma
2) vyhodnoti se podminka e ve stavu [|P|]sigma ..... [|e|]([|P|]sigma)
3) jelikoz je e false tak se vzhodnoti prokazy Q,P ve stavu [|e|]([|P|]sigma)..... [|begin Q;P end|]([|e|]([|P|]sigma))
4) znovu se vyhodnoti podminka e ve stavu [|begin Q;P end|]([|e|]([|P|]sigma))
..... [|e|]([|begin Q;P end|]([|e|]([|P|]sigma)))

Takže podle mě je správně odpověď C.
==========
[[něco]] je prostě nějaká funkce stav -> stav (tedy dostane stav, provede ono něco a vyplivne změněný stav).

Tedy máme-li [[begin A; B; A; C end]]o, tak to znamená, že stav sigma změníme provedením bloku <A,B,A,C>, tedy prvně provedeme A, poté provedeme B, pak opět A na závěr C, tedy je to stejné jako [[C]]([[A]] ([[B]] ([[A]]o))).

Když se podíváme na [[begin A; C end]] ([[B]]([[A]]o)), tak vidíme, že se také nejprve provede A, pak B, a potom blok <A,C>, tedy nejprve A, pak C -- celkem tedy opět v pořadí A B A C
==========
"Prirazeni x+=e v jazyce C funguje tak, ze se nejdrive vyhodnoti vyraz e a potom se k hodnote, ktera se momentalne nachazi v x, pricte hodnota vyrazu e.
V tomto priklade uvazujeme jen pripad, ze x na leve strane prirazeni je jen jednoducha promenna bez indexu - pri vyhodnocovani leve strany tedy nemusime uvazovat zadne vedlejsi efekty.
Pro libovolny stav "sigma" (budu to oznacovat "o"), [x+=e]o = o`, a pro vsechny prepisovatelne promenne y, u!=x, plati:

A) o`x = o x + M[e]o, o`y = [e]o y
B) o`x = o x + M[e]o`, o`y = [e]o y
C) o`x = [e]o x + M[e]o, o`y = o y
D) o`x = o x + M[e]o, o`y = o y
E) o`x = [e]o x + M[e]o, o`y = [e]o y

Problem je v tom, ze nevim co se povazuje za "vyhodnoceni toho e", M[e]o, anebo [e]o ? Vzhledem k tomu, ze to y != x predpokladam, ze by melo byt spravne bud` to C) anebo D), ale si s tim vubec nejsem jisty :). Pomozte prosim :)

[e]o se míní stav po vyhodnocení e, M[e]o se míní hodnota e po vyhodnocení e. Jinak pokud se první vyhodnocuje e tak musíme vzít o`x = [e]o x + M[e]o (vyhodnocení e nám může změnit x) o`y = [e]o y (vyhodnocení e mohlo změnit y). Takže E /s kamarádem z předtermínu jsme nad tím seděli asi patnáct minut/ https://is.muni.cz/auth/cd/1433/podzim2008/PB006/7122626/7141003
==========
Máme dán jazyk zjednodušených aritmetickývh termů, v nichž se mohou vyskytovat jen proměnné x, y, z a operace umocňování ^. Abstraktní syntax těchto termů je popsána gramatikou

T::= V | T ^ T
V::= x | y | z

Navrhněte odpovídající konkrétní syntax tak, aby její gramatika byla jednoznačná, obsahovala další dva (terminální) symboly - závorky (,), ale umožnila minimalizovat počet závorek v termech podle pravidla, že umocňování implicitně sdružuje zprava. Tedy aby například vedle ((x^(y^z))^((y^x)^z)) byl i (x^y^z)^(y^x)^z term jazyka se stejným významem.

Podle mě by to šlo vyřešit přidáním pravidla T::=(T). Nejsem si ale jist, jestli

to splňuje tu předepsanou minimalizaci. Máte nějaké lepší řešení?

Už to začínám chápat. Moje řešení má problém v tom, že povoluje derivaci:
T -> T ^ T ->
T^T ^ T,
kde jakoby povoluju sdružování zleva. Tomu se může zabránit tímto způsobem:

T::= F ^ T | F
F::= (T) | V
V::= x | y | z
https://is.muni.cz/auth/cd/1433/podzim2008/PB006/7153314
==========
u polymorfismu má např. jedna funkce nějaký obecný typ, který implicitně zastupuje celou množinu všech možných myslitelných i nemyslitelných odvoditelných (pod)typů
např. IsEmpty: List -> Bool
Bool IsEmpty(List list) { return list[0] == null }
IsEmpty je potom zároveň typu IntList -> Bool, FloatList -> Bool, StringList -> Bool, ...

kdežto u přetížení je několik funkcí se stejným jménem, ovšem různých (explicitně uvedených) typů a různých implementací
např. Default: Void -> Int, Default: Void -> Float, Default: Void -> String
Int Default() { return 0 }
Float Default() { return 0.0f }
String Default() { return "" }
==========
Typy T, U su rekurzívne definované

T = union { M : Int | A : Int x U}
U = union { N : Char | B : Char x T}

(Tj. A je typu Int x U -> T , B je typu Char x T -> U , M je typu Int -> T
, N je typu Char -> U) Typ T popisuje :

Spr8vna odpoveď je "množ. všetkých neprázdnych zoznamov, kde sa pravidelne striedajú celé čísla a znaky"
https://is.muni.cz/auth/cd/1433/podzim2008/PB006/7157183
==========
Přiřazení el := er se vyhodnocuje tak, že se nejdříve vyhodnotí pravá strana a její hodnota v se uloží. Potom se vyhodnotí levá strana a pak se na paměťové místo levé strany uloží hodnota v.
Operátor el := er sdružuje zprava. Jaká bude hodnota u = M[e1:= e2 := e3]?

u = M[e3]s
Protože se ti prvně to e3 spočítá jako hodnota v, která se přiřadí do toho, na co se vyhodnotí e2, následně do e1. Vyhodnocení toho e2 a e1 ti už může ale změnit hodnotu nějaké proměnné, na které je závislá hodnota e3. To, co jsi uvedl, by znamenalo to, že vyhodnotíš e3, pak e2, pak e1 a nakonec to znovu spočítá e3 (z již změněném stavu) a ten teprve přiřadí do e1.
https://is.muni.cz/auth/cd/1433/podzim2008/PB006/7167563
==========

Prirazeni el := er v jistem jazyce funguje tak, ze se vyhodnoti nejdrive prava strane er, potom leva strana el, a pak se hodnota prave strany ulozi do prepisovatelne promenne dane levou stranou el. Vyhodnocovani leve i prave strany muze mit vedlejsi efekty. Necht s je libovolny stav a ozancime s´= [el := er]s.
Plati
A: s´(M[el]s) = M[er]s´
B: s´(Mp[el]([er]s))=M[er]s
C: s´(M[el]([er])s)=M[er]([el]s)
D: s´(M[el]s) = M[er]([el]s)
E: s´(M[el]s)=M[er]s

Vyhodnotí sa pravá strana, môže mať vedľajšie efekty (môže zmeniť hodnoty v iných premenných a podobne), následne sa vyhodnotí ľavá strana - na jej hodnotu má vplyv aktuálny stav (ten po vyhodnotené pravej strany).
M[el]([er]s) je teda hodnota, ktorá bude po vyhodnotení oboch strán v prepisovateľnej premennej el a do tejto sa priradí hodnota premennej er v stave s, čiže M[er]s. Takže správna možnosť by mala byť B.

Mohl by mi, prosím, někdo vysvětlit, co přesně znamená zápis "s´(M[el]s)" ? Já tomu asi úplně nerozumím.

s' je stav, což je parciální funkce z proměnných do hodnot
M[_]_ je taky parciální funkce : (P x S) -> V, tedy vrací hodnotu

Co znamená aplikace stavu na hodnotu?

Ta hodnota predstavuje adresu prepisovatelne promenne. Jak ovsem stoji v zadani, leva strana se nejprve musi vyhodnotit, proto nemuzeme el primo pouzit jako prepisovatelnou promennou.
https://is.muni.cz/auth/cd/1433/podzim2007/PB006/4804620
==========
• [blabla]σ říká, do jakého stavu se dostaneme ze stavu σ po provedení blabla.
• M[blabla]σ říká, jakou hodnotu má ve stavu σ výraz blabla.
• σ(x) říká, jakou hodnotu má proměnná x ve stavu σ (protože stav je vlastně seznam, který říká, které proměnné mají jaké hodnoty)

Jestli znáš céčkovou konstrukci x++ (která nejdřív vrátí x, a potom ho zvýší o jedničku), tak slidy obsahují část, která to názorně ukáže:

[x++]σ = σ', kde σ'(x) = σ(x) + 1
\all y, y ≠ x: σ'(y) = σ(y)
(Lidsky: ze stavu σ se provedením x++ dostaneme do nějakého jiného, nazvěme si ho σ'. V něm bude mít x přiřazeno o 1 vyšší hodnotu než ve stavu σ, zatímco všechno ostatní zůstane na místě.)

M[x++]σ = σ(x)
(Lidsky: výraz x++ má ve stavu σ přesně takovou hodnotu, jaká je ve stavu σ uložená v proměnné x.)
https://is.muni.cz/auth/cd/1433/podzim2007/PB006/4899503
==========

http://dictionary.reference.com/browse/call-by-name

call by name (volanie menom) - preda hodnotu premmenej, ktora ale nieje vyhodnotena, vyhodnoti sa az ked je volana v procedure, a to aj viackrat

http://everything2.com/title/call+by+value
call by value (volanie hodnotou) - do procedury sa preda iba hodnota vyrazu, teda globalne sa nic nezmeni, hodnota je vyhodnotena pred odoslanim do procedury

call by reference (volanie odkazom) - do procedury sa predava odkaz na pametove miesto

call by value-return (volanie hodnotou vysledok) - rovnake ako volanie hodnotou spolu s volanim odkazom, iba po ukonceni procedury bude zmenena hodnota zapisana globalne

call by need (volanie potrebou) - podobne ako volanie menom, argumenty sa vsak vyhodnocuju maximalne raz a to iba vtedy ak je ich treba, teda a+b sa raz spocita a potom sa dosadzuje iba konkretna hodnota, pouziva sa iba u referencne transparentych jazykov, teda jazykov bez vedlajsich efektov

call by return (volanie vysledkom) - hodnota skutecneho parametru v dobe volani neni pouzita, az pri opousteni se vysledek (hodnota) formalniho parametru ulozi do skutecneho parametru; skutecnymi parametry tedy smi byt jen adresovatelna pametova mista

out je velmi podobne jako in.
in ti vezme hodnotu nejake promenne a zkopiruje si ji k sobe. Pak uz pracuje jenom s touto hodnotou a je mu jedno, jestli se zmenila hodnota promenne.
out si pocita hodnotu svoji promenne uvnitr, neprojevuje se to NIJAK na globalni promenne, ktera byla argumentem procedury/funkce. Projevi se to az po skonceni vypoctu, kdy se do globalni promenne oznacene "out" zkoupiruje hodnota vypocitana v tele procedury/fce.
==========
Jsou definovány typy: A = 1..3, B = 2..4, C = union {a:A | b:B}, D = record {c:C | D:Bool}, E = array Bool of D. Počet všech hodnot typu E je:

Prikladam opat svoj cheat sheet:

Union (A | B) = A+B
Record (A | B) = A*B
Set of A = 2^A
Array A of B = B^A
A ------> B = (B+1)^A
A ------> B B^A (sipka je spojita)
==========
Výraz má hodnotu.
Příkaz hodnotu nemá (ve slajdech je napsáno, že lze považovat za speciální výraz typu Command).

Výraz obecně v imperativních jazycích může měnit stav (právě pokud má ty vedlejší efek
==========

0 až 18 F,
19 až 24 E,
25 až 29 D,
30 až 33 C,
34 až 37 B,
38 a více A.

```
height(Empty,Zero).
height(Node(l,_,r),Succ(n)) :- height(l,hl), height(r,hr), max(hl,hr,n).

max(x,y,x) :- ge(x,y), !.
max(x,y,y).
```

Vyska prazdnyho stromu je nula, to je jasny. Vyska neprazdnyho stromu je maximum z vysek obou podstromu + 1 (to "+1" vyjadruje Succ(n) nalevo).

Takze: pokud strom neni Empty, nahrajeme si pomoci height do promennych hl, hr postupne vysky stromu l, r. Maximum z nich ulozime pomoci predikatu max do n, k nemuz pak nalevo pricteme jednicku.

Predikat max(x,y) vraci x, pokud x >= y, jinak y. Aby se zabranilo dalsimu vyhodnocovani v pripade, ze x>=y, pouzije se rez (!).

==========

Napiste program v prologu. Predikat height(t,n) ktery se vyhodnoti na true pokud n je rovno poctu uzlu v nejdelsi vetvi. Mame Succ, Zero, Empty pro cisla a Node a Empty pro stromy, binarni fci ge():

```
ge(m,Zero).
ge(Succ(m),Succ(n)) :- ge(m,n)
```

mozne riesenie:

```
height(empty, zero).
height(node(X,Y), succ(Z1)) :- height(X,Z1), height(Y,Z), ge(Z1, Z).
height(node(X,Y), succ(Z1)) :- height(X,Z), height(Y,Z1), ge(Z1, Z).
```

riesenie mi fungovalo v SWI Prologu, ale funkcia ge() vyzerala nasledovne:

```
ge(_, zero).
ge(succ(X), succ(Y)) :- ge(X,Y).
```

https://is.muni.cz/auth/cd/1433/podzim2009/PB006/10351417

==========

Máme predikáty A a B a prázdný symbol 0. Predikáty zadávají slova, takže např. A(A(A(B(0)))) zadává slovo aaab. Napište Hornovy klauzule zadávající predikát Isrev(x,y) který je splněn, pokud je x reverzí slova y.

Program v prologu:

```
isrev(X,Y) :- reverse(X,0,Y).

reverse(a(X),Y,Z) :- reverse(X,a(Y),Z).
```

```
reverse(b(X),Y,Z) :- reverse(X,b(Y),Z).

reverse(0,a(X),a(Y)) :- reverse(0,X,Y).
reverse(0,b(X),b(Y)) :- reverse(0,X,Y).

reverse(0,0,0).
```

https://is.muni.cz/auth/cd/1433/podzim2008/PB006/7167563
==========