2. táblás gyakorlat – specifikáció, összegzésre visszavezetés

A struktogramban alkalmazható transzformációkról

Alapvetően a félév során arra törekszünk, hogy egy tételre hasonlító specifikációt tudjunk felírni, amit így már vissza tudunk vezetni a tételre, nyilván a struktogramjával együtt.

De mi van akkor, ha az utófeltételben olyan elemek jelennek meg, amelyekre nincs tétel?

Bonyolultabb, ciklust is igénylő feladatokat ebben a félévben csak tételre vezethetünk vissza, de bizonyos egyszerűbb értékadásokat, elágazásokat valahogy úgy, ahogy a józan eszünkkel is tennék, természetesen most is alkalmazhatunk.

Primitív értékadás:

Nézzük a következő utófeltételt:

$$uf = (a = b)$$

Ekkor nyilvánvaló, hogy a feladatot megoldja az alábbi értékadás:

$a \coloneqq b$

Általános eset:

Kicsit általánosabb eset, ha az utófeltétel így néz ki:

$$uf = \big(a = f(b)\big)$$

Ez a struktogram adódna:

$a \coloneqq f(b)$

Na jó, de mi az az f? Abban az esetben, ha ez egy megengedett függvény, akkor semmi gond, kész vagyunk.

„Egyszerű” függvények:

Példa, ha f mondjuk egy olyan függvény, ami 1-et ad a paraméterül átadott számhoz:

$$uf = (a = b + 1)$$

$a \coloneqq b + 1$

Esetszétválasztással definiált függvények:

Mi van akkor, ha f kicsit összetettebb, mondjuk egy esetszétválasztással definiált függvény, mint az abszolútérték vagy a szignum függvény?

$$uf = (a = |b|)$$

$a \coloneqq \lvert b \rvert$

Ez így egy nem megengedett értékadás. Hasonlóan:

$$uf = \big(a = sign(b)\big)$$

$a \coloneqq sign(b)$

se.

Pirossal jelöltem azokat a *kifejezés*eket az utófeltételben, amik így ebben a formában hiányosak, hiszen nincsenek kifejtve, és szintén pirossal a struktogramban azokat, amelyek hasonló okokból *nem megengedett értékadás*ok, hiszen senki se mondta meg, hogy hogyan kell kiszámolni őket (a gyakorlatokon ezeket karikázással jelölöm).

Akkor definiáljuk!

$$|.|: \mathbb{Z} \to \mathbb{N}, \text{ és}$$

$$\forall n \in \mathbb{Z}: |n| = \begin{cases} n, ha\ n \geqq 0 \\ -n, különben \end{cases}$$

Hála ennek a pontosításnak, most már az utófeltétel nevű *elsőrendű logikai kifejezés*ben immáron nem ismeretlen jel a $|b|$. És mivel azt is tudjuk, hogy ezt az abszolút érték függvényt *esetszétválasztás*sal lehet megadni, ezért a struktogramban elágazást fogunk használni:

$b \geqq 0$	
$a \coloneqq b$	$a \coloneqq -b$

A szignum (előjelfüggvény) esetében hasonlóan járhatunk el:

$$sign: \mathbb{Z} \to \mathbb{N}, \text{ és}$$

$$\forall n \in \mathbb{Z}: sign(n) = \begin{cases} 1, & ha\ n > 0 \\ 0, & ha\ n = 0 \\ -1, & kül. \end{cases}$$

$b > 0$	$b = 0$	$b < 0$
$a \coloneqq 1$	$a \coloneqq 0$	$a \coloneqq -1$

Általános „tétel”: esetszétválasztással definiált függvény helyettesítésére:

adott egy $a \coloneqq f(b)$ nem megengedett értékadás, ahol f egy így megadott függvény:

$$f(b) = \begin{cases} c_1, ha\ \beta(b) \\ c_2, különben \end{cases}$$

Ekkor a fenti nem megengedett értékadással ekvivalens az f-et már nem tartalmazó alábbi programrészlet:

$\beta(b)$	
$a \coloneqq c_1$	$a \coloneqq c_2$

Persze ha c_1, c_2, vagy β akármelyike nem megengedett, akkor szükség van további lépésekre is.

Nyilván ha a feltétel több ágú (mint a szignumnál), akkor annak megfelelően az elágazás is több ágú lesz.

Egyéb függvények:

Ha f egy bonyolultabb függvény, akkor megpróbálunk belőle is egy progtételt kihozni, erről majd később tanulunk bővebben.

Még akad egy-két variáció.

f lehet egy rekurzív függvény, erre majd rátérünk később.

f lehet összetett függvény.

Valamint az utófeltétel lehet egy *konjunkció* (azaz mindenféle feltételek összeéseléséből született összetett feltétel)

Konjunkció:

Nézzük most ezt a legutóbbi esetet.

Mi van akkor, ha azonos intervallumon azonos jellegű műveletet kell végezni, egymástól függetlenül?

Legyen az az utófeltétel, hogy számítsuk ki egy tömb elemeinek összegét és szorzatát is.

$$uf = \left(s = \sum_{i=1}^{n} t[i] \wedge p = \prod_{i=1}^{n} t[i] \right)$$

Ekkor ez nyilvánvalóan két összegzés tétel, de ha az intervallumok azonosak és a két számítás nem *interferál* egymással (azaz nem hatnak egymásra), akkor hatékonyabb egy ciklusban lerendezni a kettő számítást:

$s, p \coloneqq 0,1$	
$i = 1..n$	$i : \mathbb{Z}$
$s, p \coloneqq s + t[i], p \cdot t[i]$	

Persze, az nem szükséges, hogy a két tételbeli f függvény azonos legyen (fent egyaránt $t[i]$, de lent most $-i$, ill. $t[i] + 2$)

$$uf = \left(s = \sum_{i=1}^{n} -i \wedge p = \prod_{i=1}^{n} t[i] + 2 \right)$$

$s, p \coloneqq 0,1$	
$i = 1..n$	$i : \mathbb{Z}$
$s, p \coloneqq s - i, p \cdot (t[i] + 2)$	

stb.

Ha s és p kiszámítása nem megy „párhuzamosan”, akkor valamilyen sorrendben egymás után kell elvégezni a két számítást.

A másik lehetőség, amikor egy már kiszámolt értékből kell valami újat kiszámolni (ekkor sem független a két összeéselt kifejezés). Most, ha tudjuk a függőségeket (azaz, hogy melyiket kell előbb kiszámolni, és abból hogy jön ki a másik adat), akkor egyszerűen egy szekvenciába rendezhetjük a két részszámítást.

Egyik példánk legyen az, hogy egy tömb elemeinek szorzatát és összegét adjuk össze.

$$uf = \left(s = \sum_{i=1}^{n} t[i] \wedge p = \prod_{i=1}^{n} t[i] \wedge e = s + p \right)$$

Nyilvánvalóan előbb kiszámoljuk s-t és p-t (valamilyen sorrendben – akár egyszerre, ahogy az előző példában volt), majd ebből e-t.

s, p kiszámítása
$e \coloneqq s + p$

Összetett függvény kiszámítása:

És mi van akkor, ha az nem feladat, hogy s-t és p-t számítsuk ki?

$$uf = \left(e = \sum_{i=1}^{n} (t[i]) + \prod_{i=1}^{n} (t[i]) \right)$$

Ekkor nem változik semmi az előzőhöz képest, csupán annyi, hogy most fel kell ismernünk azt, hogy ketté kell szedni a feladatot, előbb meg kell oldani az összetett kifejezés bal, majd a jobb oldalát, és végül összegezni. Ehhez teljesen ugyanazt csináljuk, mint az előbb, csak s-t és p-t a végén eldobhatjuk, mert ezek most csak segédváltozók, mint ahogyan egy `for` ciklusbeli i is az.

s, p kiszámítása	s, p: segédv.
$e \coloneqq s + p$	

Nézzünk még egy példát. Tudjuk, hogy egy hallgató eddig k kreditnyi tárgyat végzett el. Egy tömbben tároljuk az idén elvégzett tárgyainak kreditértékeit. Mennyi kreditet végzett el összesen?

$$uf = \left(s = \sum_{i=1}^{n} (t[i]) + k \right)$$

$s \coloneqq 0$	
$i = 1..n$	$i: \mathbb{Z}$
$s \coloneqq s + t[i]$	
$s \coloneqq s + k$	

Hasonlóan állunk hozzá, mint az előbb. Gondolatban két részfeladatra bontjuk a feladatot. Először kiszámoljuk az összeget egy összegzés tétellel (még segédváltozó se kell, mert mehet a kimenő változóba), majd ehhez hozzáadjuk az eddigi krediteket. Ugyanúgy, mintha egy éseléses utófeltétel lett volna, valahogy így:

$$uf = \left(ideiglenes = \sum_{i=1}^{n} (t[i]) \wedge s = ideiglenes + k \right)$$

Nézzünk rá kicsit általánosabban!

Tegyük fel, hogy ezt meg tudjuk már oldani:

$a \coloneqq f(b)$

Ekkor ezt továbbra sem:

$a \coloneqq f(b) + 1$

Miért nem?

Tegyük fel, hogy az $f(b)$ definíciója ismert, és ezáltal az $a \coloneqq f(b)$ értékadást is át tudjuk alakítani megengedett módon. Például ha f egy tételre visszavezethető, vagy f egy esetszétválasztás, stb.

A probléma az, hogy amikor mi „megírjuk” az $f(b)$ definíciója alapján az azt kiszámoló kódot, akkor mi nem az $f(b)$ kifejezést oldjuk meg, hanem az $a{:=}\, f(b)$ értékadást. Előbbit hogy is tudhatnánk? Hiszen az nem egy utasítás, nem értelmezhető alprogramként. Programot csak úgy tudunk írni, hogy adott egy vagy több változó, legyen ezek értéke valami. Például legyen a a már jól definiált f függvény értéke a b helyen! Tehát az $a{:=}\, f(b)$ egy korrekt alprogram, de önmagában az $f(b)$-ből nem tudunk programot írni.

Azzal, amikor leírunk egy nem megengedett értékadást, azzal tulajdonképpen egy *alfeladat*ot definiálunk. Legyen a változó értéke egy kifejezés értéke! Ez egy utasítás, valami olyan, amire programot lehet írni. Felfogható úgy, hogy ennek az alfeladatnak van egy külön állapottere, amelyben a bemenő változók a paraméterül megadottak (most b), a kimenők pedig, akik értéket kell, hogy kapjanak (most a), az utófeltétel pedig maga a kiszámítandó kifejezés ($a = f(b)$).

Szóval lehet, hogy ismerjük az $f(b)$ definícióját, lehet hogy azt is tudjuk, hogy $f(b) + 1$ az „eggyel több, mint $f(b)$”, de attól még az $a \coloneqq f(b) + 1$ egy nem megengedett értékadás marad, mert az $f(b)$ önmagában nem értelmezhető utasítás, és hiába tudnánk ahhoz egyet hozzáadni, nem tudjuk mi az, amihez hozzá kell adni.

Viszont – ha ezt a kifejezést összetett függvényként fogom fel – akkor két lépésre tudom bontani a megoldást, éppúgy mint az éselésnél. Fel is tudom írni konjunkciós alakban:

$$sv = f(b) \wedge a = sv + 1$$

És innen a megoldó program már adott, hiszen azt feltettük, hogy az $a{:=}\, f(b)$ értékadást már ki tudjuk számolni, és nyilván az $a \coloneqq\ b + 1$ értékadás is megengedett.

$sv \coloneqq f(b)$	$sv\colon segédv.$
$a \coloneqq sv + 1$	

Az összetett függvényként való felfogás nagy előnye az éseléshez képest az, hogy míg utóbbinál már a specifikációban is kénytelen voltunk az utófeltételben olyan változót emlegetni ami nincs az állapottéren (sv), ha összetett függvényként írjuk fel, akkor ez a jelölés elhagyható. Tehát mostantól az előző feladat utófeltételét felírhatjuk így:

$$uf = (a = f(b) + 1)$$

Megoldása pedig az alábbi program:

$sv \coloneqq f(b)$	$sv: segédv.$
$a \coloneqq sv + 1$	

Következő tételünk tehát, a lehető legáltalánosabban:

Ha adott az alábbi alakú utófeltétel:

$$uf = \Big(a = f\big(g(b)\big)\Big)$$

Akkor azt úgy lehet lekódolni, hogy előbb egy segédváltozóba kiszámítjuk a belső, g függvény értékét a b helyen, majd amit kaptunk, arra rázúdítjuk a külső, f nevű függvényt.

$sv \coloneqq g(b)$	$sv: segédv.$
$a \coloneqq f(sv)$	

És ezt hívjuk *összetett függvény kibontásá*nak.

A fejezet elején látott

$$uf = \left(e = \sum_{i=1}^{n}(t[i]) + \prod_{i=1}^{n}(t[i])\right)$$

kifejezés is egy összetett függvény, hiszen tulajdonképpen

$$uf = \Big(e = h\big(f(t), g(t)\big)\Big)$$

alakú, ahol a h az összeadás, az f a szumma és a g a produktum.

És akkor innen már gondolkodás nélkül adja magát a helyes megoldás:

$szumma \coloneqq h(t)$	$szumma: segédv.$
$prod \coloneqq g(t)$	$prod: segédv.$
$e \coloneqq h(szumma, prod)$	

Előbb kiszámoljuk a *belső függvény*(eke)t, majd erre/ezekre alkalmazzuk a *külső függvény*t.

Röviden a tanulság: minden olyan utófeltételben, ahol éselést látunk, ahol egymásba ágyazott függvényeket látunk, ahol részszámításokat látunk ott ezeket a részszámításokat egymás után elvégezve kapjuk a megoldó programot, miközben azt mondjuk itt a fenti, összetett függvényes eljárást alkalmaztuk.

Nem megengedett kifejezések:

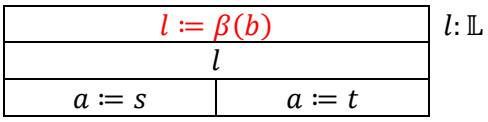
Mi a helyzet akkor, amikor egy nem megengedett *feltétel* kerül egy elágazásba vagy ciklusba?

$\beta(b)$	
$a \coloneqq s$	$a \coloneqq t$

Legyen ebben a példában $\beta(b)$ egy *nem megengedett kifejezés*, ezért ez így egy *nem megengedett feltétel*.

Függvény helyettesítése változóval:

De mint láttuk „$\beta(b)$”-ből direktben nem tudunk programot írni, hiszen a programok utasítások, $\beta(b)$ pedig csupán egy logikai kifejezés. De azt már meg tudjuk csinálni, hogy bevezetünk egy logikai változót, amibe kiszámoljuk $\beta(b)$ értékét, majd eszerint a logikai változó szerint ágazunk el:



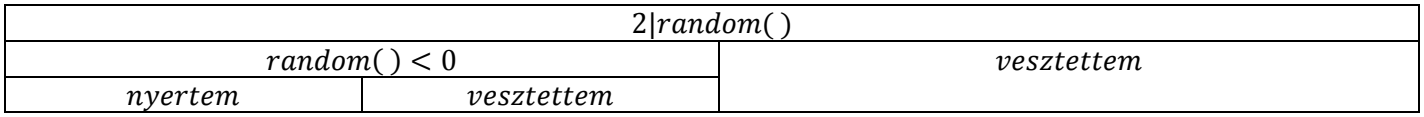
Tehát behoztunk egy $l := \beta(b)$ nem megengedett programot, amit persze a konkrét β definíciója alapján ki tudunk számolni. És az l változó értéke szerinti elágazás már megengedett program.

Ezt a fenti átalakítást hívjuk *függvény helyettesítése változóval*-nak.

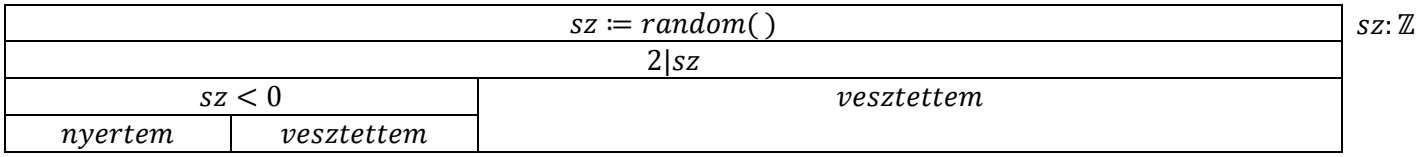
Ezt az átalakítást a gyakorlatban általában megússzuk. A következő fejezet szól arról, hogy miért. Viszont ez a transzformáció igen is hasznos tud lenni más esetekben, mégpedig olyan esetekben, ami valódi programozási gyakorlatban is előkerül!

Mi van akkor, ha β kiszámítása bonyolult? Netán mellékhatásos? Netán nemdeterminisztikus? Ezek mind olyan esetek, amikor meg kell gondolni, hogy β-t újra és újra ki akarom számolni, vagy megelégszem azzal, hogy egyszer kiszámolom, és onnan már csak a kiszámolt értéket használom. Mellékhatásos esetben is lehet, hogy az a cél, hogy sokszor lefusson, és a nem determinisztikus esetben is lehetséges… Mégis most lássunk két példát arra, amikor célszerű használni a transzformációt:

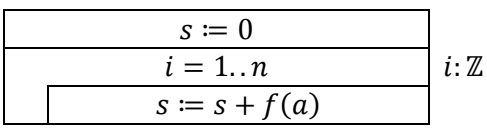
Legyen az a feladat, hogy sorsolok egy véletlen számot, ha ez a szám páros, akkor ha negatív akkor nyertem, ha pozitív, akkor vesztettem, valamint ha páratlan, akkor vesztettem:



Itt nyilvánvaló, hogy nem szabad a belső elágazásban újrasorolsni a számot, tehát be kell vetni a FHV transzformációt:



Nézzünk egy példát a műveletigényes függvényre is:



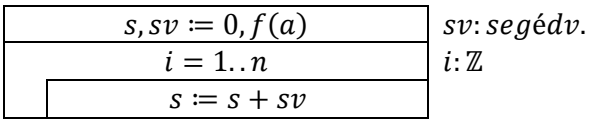
Láthatjuk, f függvény csak a-tól függ, amit pedig tegyük fel, hogy a ciklus nem módosít. Azaz, $f(a)$ értéke a ciklus előtt, a ciklus elején, a ciklusban, a ciklus végén és a ciklus után is

ugyanaz. Azt is tegyük fel, hogy f egyébként egy bonyolult, számításigényes függvény és/vagy n nagy.

Az egyik ok tehát egyszerűen a hatékonyság. Ha f mindig ugyanazt adja (amíg nem módosítom a-t), akkor minek számoljam ki n-szer, nem elég egyszer?

A másik ok a potenciális mellékhatásosság. Lehet, hogy f-et úgy oldottuk meg, hogy minden hívásakor növeljen mondjuk egy globális számlálót. Az én szándékom mondjuk legyen az, hogy csak egyszer növelődjön ez a számláló, de mégis n-szer fog. Vagy f a számolás során egy külső adatbázisból nyer adatokat. Minden alkalommal, amikor f-et meghívom, az csatlakozik az adatbázishoz, lekéri a szükséges adatokat, majd bontja a kapcsolatot. Kell ez nekem? Minden alkalommal? Tegyük fel, hogy az adatbázis naplózza az őt ért műveleteket, és azt látta, hogy az én programom ugyanazokat az adatokat gyors egymásutánban többször is lekérdezte, mikor láthatóan elég lett volna egyszer is. Ez legalábbis furcsa működés.

Nos, ez a két ok vihet rá minket, hogy mégis kiszervezzük egy változóba az alprogramunk által visszaadott értéket, és ezt a fenti ismeretek birtokában igen könnyen meg is tudjuk írni:



Persze, ha nekem kifejezetten célom, hogy f n-szer hívódjon meg, mert ezzel akarok valami kifejezni (ezt is el lehet képzelni), akkor nyilván ezt nem tehetjük meg. Viszont akkor már a terv hibás, mert elég életszerűtlen hogy a bonyolult f függvény legyen arra való, hogy számlálóként működjön közre. De ez már messzire vezet.

Most nézzük meg, hogy hogyan lehet egyszerűsíteni mindezeken!

Függvényszerű alprogram-hívás:

Hogyan kódolnánk C++-ban az előző fejezet β feltételes példáját?

Kezdetben van ez a programunk:

```
if(beta(b))
  a = s;
else
  a = t;
```

Aztán átalakítottuk így:

```
bool l = beta(b);
if(l)
  a = s;
else
  a = t;
```

És ezáltal *definiáltunk* is egy *alprogram*ot, egy `bool` visszatérési értékű `beta` függvényt kell írnunk, melynek egy paramétere van, `b`. (legyen ebben a példában mondjuk `b` egy `int`)

Ekkor tehát megírjuk ezt a függvényt:

```
bool beta(int b)
{
  …
  return …;
}
```

Most ez a program így, ebben a formában működőképes, de a gyakorlatban senki se ír ilyen programot, hiszen felesleges egy `l` változóba elmenteni a `beta` által adott értéket a fő programunkban… Ehelyett simán alkalmazzuk így, mint ahogy először is tettük:

```
if(beta(b))
  a = s;
else
  a = t;
```

Ezt szeretnénk a struktogramban is kifejezni, erre találták ki az ún. *függvényszerű alprogram-hívás*t.

A példabeli `beta` függvény egy alprogram. Ezt, hogy használni is tudjuk meg kell hívni a főprogramban. Erre két fajta mód van, az egyik az *eljárásszerű* megoldás, ami gyakorlatilag a függvény helyettesítése változóval transzformációt jelenti, a másik pedig a *függvényszerű*, aminek az a lényege, hogy ha leírok egy $f(b)$ kifejezést, az alatt az ő visszatérési értékét értem. Tehát felesleges pluszban bevezetni a logikai változót, ha azt írom le a feltételbe egyszerűen hogy `beta(b)`, akkor ha ez egy függvényszerű alprogram-hívás, akkor itt voltaképpen az `l = beta(b)` értékadás által `l`-be visszaadott értékre gondolok, megspórolva a segédváltozót.

A gyakorlatban tehát:

Amikor egy feltétel nem megengedett ($\beta(b)$), vagy egy utasítás nem megengedett rész kifejezést tartalmaz, akkor NEM kell mindenáron a FHV transzformációt alkalmazni, elég ha azt mondjuk , hogy az ominózus kifejezés helyén egy függvényszerű hívást hajtottunk végre. Ugyanakkor az a program ami végül megengedetté teszi a még nem megengedett $f(b)$ hívást, az sosem egy „$f(b)$”, hanem egy $a := f(b)$ alakú program lesz. Remélem a fentiek megmagyarázták, hogy miért is.

Egy összetett példa:

Legyen egy szimpla összegzéses feladat, egy intervallumon lépkedjünk végig és a páros számok esetében saját magukat, a páratlanak esetében az ellentettjüket adjuk hozzá az összeghez.

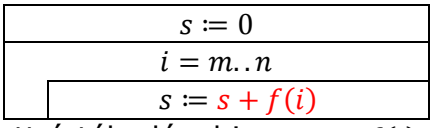
$$A = (m: \mathbb{Z}, n: \mathbb{Z}, s: \mathbb{Z})$$

$$ef = (m = m' \wedge n = n')$$

$$uf = \left(ef \wedge s = \sum_{i=m}^{n} f(i) \right)$$

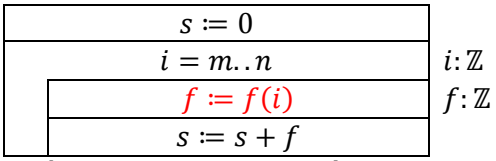
ahol $f: \mathbb{Z} \to \mathbb{Z}$, és $\forall a \in \mathbb{Z}: f(a) = \begin{cases} a, & ha\ 2|a \\ -a, & kül. \end{cases}$

(1)



Belül van egy nem megengedett értékadás, hiszen $s + f(i)$ egy *összetett függvény*. Bontsuk ki!

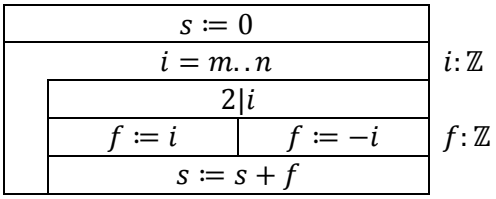
(2)



Most gyakorlatilag ugyanazt csináltuk, mint a *függvény helyettesítése változóval* esetében.

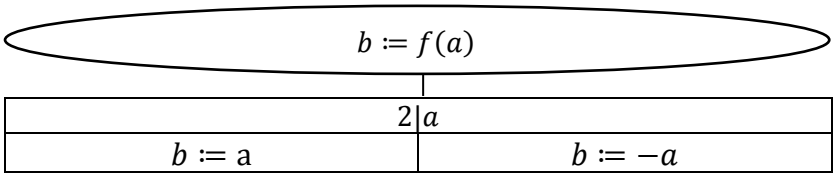
Fejtsük ki az *esetszétválasztásos függvény*t:

(3)



Ez egy helyes megoldás. Mégse fogjuk többé így csinálni, mert felesleges ennyire elbonyolítani.

Ezzel szemben, ha megnézzük a (2) struktogramot, láthatjuk, hogy „igény van” egy $f \coloneqq f(i)$ feladatot megoldó programra, hát írjuk meg:

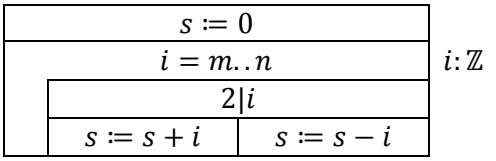


(láthatjuk, hogy az *aktuális* (függvény meghívása-beli) és *formális* (függvény definíció-beli) *paraméter*ek nevei eltérhetnek)

Ekkor most a (2) struktogram, így ezzel a megoldott alprogrammal helyes, és ami igazán fontos, ha *függvényszerű alprogramhívás*sal tekintünk erre a frissen megírt programra, akkor az (1) struktogram is! És ez az, amit leginkább használni fogunk.

Van még egy lehetőség, az, hogy az összetett függvényes gondolatmenetet csak fejben végezzük el, és segédváltozó bevezetése nélkül egyből úgy bontjuk szét az esetszétválasztást, ahogy ebben az utolsó példában:

(4)



Értelemszerűen ez is egy használható megoldás.

Miképpen az is megengedett transzformáció, hogy egy skippel egyenértékű utasítást kicseréljünk skipre (vagy eleve azt írjuk)

Azaz:

$$a \coloneqq a$$

helyett írhatjuk azt is, hogy

$$SKIP$$

Ez ilyen esetszétválasztásos szituációban elég sűrűn elő tud fordulni.

Megengedett műveletek és kifejezések:

ZH-ban megengedett kifejezések (előfeltételben, utófeltételben, ciklus- és elágazás-feltételben, logikai értékadás jobb oldalán levő kifejezésben):

- | (osztója)
- mod (maradékképzés)
- ∧, ∨ (és, vagy)
- ¬ (tagadás)
- → (implikáció – bár ezt visszavezethetjük vagyolásra)
- ∈ (eleme-e)
- =, ≠ (egyenlő-e), nem egyenlő-e
- <, >
- ≦, ≧
- +, −
- ·, /
- ^ (hatványozás – nyilván felső indexes jelöléssel is megengedett)
- tömb- vagy sorozatindexelés
- . (szelektorfüggvény rekordhoz)
- $perm(.)$ (permutáció)
- |.| (abszolút érték)
- ⌊.⌋, ⌈.⌉ (alsóegészrész, felsőegészrész)
- állapottér komponensei, adott konstansok, adott függvények
- már megírt alprogrammal rendelkező saját függvény szerepeltetése logikai kifejezés részeként (*függvényszerű alprogram-hívás*)
- ezekből készített összetett kifejezések
- speciális esetekben más is
- csak ef/uf: nagyoperátoros kifejezések, „vesszős” konstansok, kvantoros kifejezések, tételek rövidített „operátorai”
- csak ef/uf: χ függvény: $\chi: \mathbb{L} \to \{0,1\}$, úgy hogy: $\chi(\uparrow) = 1$ és $\chi(\downarrow) = 0$.
- utófeltételben összetett függvénynél megengedek olyan változót szerepeltetni, ami nincs az állapottéren
- csak ef/uf: χ függvény: $\chi: \mathbb{L} \to \{0,1\}$, úgy hogy: $\chi(\uparrow) = 1$ és $\chi(\downarrow) = 0$.

Megengedett utasítások:

- $SKIP$
- ≔ (értékadás, baloldalán változók, illetve bevezetett segédváltozók lehetnek)
- elágazás
- ciklus
- szekvencia
- „legyen eleme” (halmaznál) /normális nevén: nemdeterminisztikus értékkiválasztás/ jele: „$:\in$”
- már definiált függvény értékadása (*eljárásszerű alprogram-hívás*)
- már definiált saját függvény szerepeltetése értékadás jobboldalán levő kifejezésben (*függvényszerű alprogram-hívás*)
- ezekből és a megengedett kifejezésekből képzett összetett utasítások
- speciális esetekben más is