Pisati čitko – nečitak odgovor ne donosi bodove.

1. (2) Napisati makro `FUNKCIJA(x,y,z)` koji će za tri ulazne vrijednosti x, y i z vratiti vrijednost funkcije: $f(x, y, z) = x + y \cdot z$.

```
- samo dodati zagrade oko svega, pojedinačno i zajedno

#define FUNKCIJA(x,y,z) ( (x) + (y) * (z) )

- obzirom da se svaki argument koristi samo jednom nije potrebno
   dodavati typeof(x) _x_ = (x) i slično
- krivo je koristiti do-while(0) !!!
```

2. (2) Napisati makro `SREDNJA(x,y,z)` koji će za tri ulazne vrijednosti x, y i z vratiti onu ulaznu vrijednost koja nije ni najmanja ni najveća. Pretpostaviti da se taj makro može pozivati i sa složenim argumentima (npr. `SREDNJA(i++, b=fun1(c), fun2())`).

```
#define SREDNJA(x,y,z)                                                  \
({                                                                      \
    typeof (x) _x_ = (x);                                               \
    typeof (y) _y_ = (y);                                               \
    typeof (z) _z_ = (z);                                               \
                                                                        \
    (_x_ > _y_)?( _y_ > _z_ ? _y_ : ( _x_ > _z_ ? _z_ : _x_ ) ) : \
                ( _x_ > _z_ ? _x_ : ( _y_ > _z_ ? _z_ : _y_ ) )    \
})

- ovdje je potrebno koristiti typeof
  (jer se puno puta koristi isti argument)
- krivo je koristiti do-while(0)
- krivo je koristiti "return"
```

3. Student svoj zadatak za laboratorijsku vježbu predaje preko git repozitorija. Prvu vježbu je predao ("commit" i "push") i započeo je rad na drugoj (samo "commit"). Nastavnik je pregledao prvu vježbu i stavio svoje komentare u repozitorij (promjenio neke datoteke te napravio i "commit" i "push"). Što mora student napraviti da bi nastavio rad na drugoj vježbi i onda i to objavio:

   a) (2) ako su datoteke za drugu vježbu u drugoj mapi (npr. u lab2) ili

   b) (2) ako su datoteke za drugu vježbu iste kao i za prvu (prva se nadograđuje)?

```
a) prije "push" treba napraviti "pull" (on radi "automatski merge" i
   nema konflikata)
b) git pull + pogledati konflikte i popraviti ih, onda add/commit/push
```

4. (2) Dinamičko upravljanje spremnikom u nekom je sustavu ostvareno korištenjem metode "prvi odgovarajući" (npr. kao u Benu). Navesti sve razloge "neproduktivnog" korištenja spremnika zbog toga, tj. na što se sve troši/gubi spremnički prostor, osim za pohranjivanje podataka potrebnih programu.

```
- zaglavlja za blokove
- poravnanje na veličinu zaglavlja
- dodjela cijelog bloka kad je ostatak premali
- fragmentacija
```

5. (3) Radi upravljanja nekim sustavom treba generirati prekid **svakih** 100 mikrosekundi i pozvati funkciju `ažuriraj()`. Također, **jednom** unutar svakih 330 mikrosekundi (počevši od t=0) treba pozvati `proračunaj()` (obje funkcije postoje i njihovo izvođenje je kratko, do nekoliko mikrosekundi). Za to na raspolaganju stoji 16-bitovno brojilo (na adresi `CNT`) koje odbrojava frekvencijom od 25 MHz od zadnje učitane vrijednosti do nule, izaziva prekid i ponovno kreće s odbrojavanjem od iste vrijednosti. Ostvariti zadani sustav – funkciju `inicijalizacija()` koja se poziva na početku, funkciju `prekid_brojila()` koja se poziva na svaki prekid brojila, te opisati dodatno potrebnu strukturu podataka. Radi uštede energije prekid brojila se treba javljati što rijeđe moguće, ali opet da se zadovolje navedeni zahtjevi.

```
za prekid svakih 100 us, u brojilo treba staviti:
100 us / (1/25 MHz) = 2500 (stane u 16 bita!)

varijable:
long t; // vrijeme u us
long t330; // praćenje intervala od 330 us

void inicijalizacija() {
    t = t330 = 0;
    *CNT = 2500;
}

void prekid_brojila() {
    t += 100;
    ažuriraj();
    if ( t330 <= t ) {
        t330 += 330;
        proračunaj();
    }
}
```

6. Izvorni kod nekog sustava sastoji se od datoteka s izvornim kodom u C-u koje se nalaze u mapama (direktorijima) *boot*, *arch*, *kernel* i *progs*. Popis datoteka u svakom direktoriju (tj. objekata koji od njih nastaju) zadan je u zasebnim datotekama (npr. *arch/files.txt*) u obliku `dir_OBJS=dat1.o dat2.o ...` (`dir` je ime mape, `BOOT`, `ARCH` ...). Prilikom prevođenja datoteka potrebno je koristiti zastavice `Z1=5`, `Z2=1` i `Z3=15`. Dodatno, za prevođenje datoteka iz mapa *boot*, *arch* i *kernel* potrebno je postaviti zastavicu `KERNEL`. Uređaj za koji se sustav priprema ima ROM na adresi 0x10000 i RAM na adresi 0x100000. Ugrađeni program pokretač (boot loader, koji nije dio navedena izvorna koda) će odmah pri pokretanju kopirati sve iz ROM-a u RAM te započeti s izvođenjem koda na adresi 0x100000. Stoga treba program tako pripremiti da napočetku budu početne instrukcije (instrukcije jedine datoteke iz mape *boot*).

   a) (3) Napisati skriptu povezivača (*boot/ldscript.ld*).

   b) (4) Napisati *Makefile* (ili više njih, za svaki direktorij zaseban) za prevođnje sustava.

   Pri povezivanju, skripta povezivača se dodaje uz zastavicu `-T`:
   `ld -o ime_slike popis-objekata -T ime_skripte dodatne-zastavice`
   Koristiti recepte za prevođenje više datoteka u obliku:
   `mapa/%.o: mapa/%.c` (znak % mijenja više znakova)
   `$(CC)-c $< -o $@ <+dodatne zastavice>`.
   Za gornji primjer, varijabla `$<` bi sadržava samo ime datoteke (npr. `mapa/dat1.c`), a varijabla `$@` ime cilja (npr. `mapa/dat1.o`).

a) boot/ldscript.ld:

```
ENTRY(0x100000) /* nije nužno */
SECTIONS {
    .sve 0x100000 : AT ( 0x10000 ) {
        *boot* (.text)
        * (*)
    }
}
```

(može i drukčije)

b)
U tekstu se nespretno spominju "zastavice" umjesto "makroi" ili "vrijednosti" pa su i rješenja s -Z1=5 i slično ispravna iako je zamišljeno da se to dodaje sa zastavicom -D (-D Z1=5 -D Z2=1 itd.)

i) jedan Makefile u početnom direktoriju:

```
include boot/files.txt arch/files.txt kernel/files.txt progs/files.txt
CFLAGS1 = -D Z1=5 -D Z2=1 -D Z3=15 -D KERNEL
CFLAGS2 = -D Z1=5 -D Z2=1 -D Z3=15

bOBJS := $(addprefix boot/,$(boot_OBJS))
aOBJS := $(addprefix arch/,$(arch_OBJS))
kOBJS := $(addprefix kernel/,$(kernel_OBJS))
pOBJS := $(addprefix progs/,$(progs_OBJS))
# iako su ove transformacije potrebne, priznata su i rješenja bez njih

image.elf: $(bOBJS) $(aOBJS) $(kOBJS) $(pOBJS)
    ld -o $@ $^ -T boot/ldscript.ld

boot/%.o: boot/%.c
    gcc -c $< -o $@ $(CFLAGS1)
arch/%.o: arch/%.c
    gcc -c $< -o $@ $(CFLAGS1)
kernel/%.o:kernel/%.c
    gcc -c $< -o $@ $(CFLAGS1)
progs/%.o:progs/%.c
    gcc -c $< -o $@ $(CFLAGS2)
```

ii) po jedan Makefile u svakom direktoriju:

Makefile (početni):

```
image.elf: boot/boot.o arch/arch.o kernel/kernel.o progs/progs.o
    ld -o $@ $^ -T boot/ldscript.ld

%.o: # jedno pravilo za sve
    make -C $(dir $@)
```

```
# ili raspisati za svaki zasebno:
# boot/boot.o:
#     make -C boot
# ...
```

boot/Makefile:

```
include files.txt
CFLAGS = -D Z1=5 -D Z2=1 -D Z3=15 -D KERNEL

boot.o: $(boot_OBJS)
    gcc -r $^ -o $@

# implicitna pravila su dovoljna za izgradnju pojedinih .o datoteka
# ili koristiti recept:
# %.o: %.c
#     gcc -c $< -o $@ $(CFLAGS)
```

slično za arch i kernel;
za progs samo zadnju zastavicu maknuti (bez -D KERNEL)