

KAJ JE PROGRAM IN KAJ JE PROGRAMIRANJE

Uporabniška programska oprema je že tako razvita, da omogoča reševanje skoraj vseh problemov, na katere naletijo računalniški uporabniki različnih področij. Za pomembnejša tehnična ali poslovna področja je napisanih tudi po več različic programske opreme. Toda za drobnejše, zelo specifične naloge je še vedno potrebno pripraviti poseben program. Če ga hočemo narediti, moramo poznati določene postopke in imeti vsaj osnovno znanje.

Program: Kaj je program? To je niz logično povezanih ukazov, inštrukcij, ki računalniku zapovedujejo, kako naj obdela določene podatke, da bo rešil neki problem. Preprosta definicija programiranja je pa je pisanje povezanih ukazov, inštrukcij, ki dejansko računalniku zapovedujejo kako naj obdela določene podatke. Z programiranjem nastane program.

Med programiranjem naletimo na več stopenj: najprej je treba nalogo opredeliti, jo razčleniti in najti rešitev; zatem jo preverjeno zapišemo v obliki in jeziku, ki ga razume računalnik. Seveda delo s tem še ni končano, saj so potrebna še dodatna preverjanja, pregled in testiranje programa; šele nato ga lahko varno shranimo na določeni element zunanega spomina. Glede na vse to je za programiranje potrebno poznati simbolične ukaze, pravila pisanja programa, nalogo je pa potrebno tudi analizirati. Programiranje obsega torej več stopenj, ki jih je treba podrobneje spoznati.

Opredelitev in razčlenitev problema

Prvi pogoj, za reševanje problema z računalnikom je, da problem oz. Nalogo v celoti in popolnoma razumemo; to pomeni, da jo znamo na kratko z besedami podrobno opisati. Zato je prvi korak pri reševanju kakega problema kar začetni opis, v katerem povemo predvsem, KAJ je potrebno narediti (torej jasno oblikujemo bistvo problema, ne pa šele premišljujemo, kako ga bomo rešili).

Takoj zatem je treba nalogo opredeliti, to je določiti, katere podatke potrebujemo in jih obdelujemo, kje jih dobimo, kako si jih zapomnimo, kako jih prepoznamo, kam jih shranimo, koliko jih je, kakšne operacije bomo z njimi opravljali (računske ali druge), kako bomo izrazili rezultat in kako ga bomo oblikovali.

Šele ko je začetna opredelitev problema končana, ko je torej povsem jasno, KAJ je potrebno narediti, pričnemo postopek razčlenjevanja, to je, KAKO narediti. Imenujemo ga načrtovanje rešitve problema. Bistvo tega dela je razgrajevanje problema na niz zaporednih majhnih korakov – postopkov, tako imenovanih neodvisnih problemov. Seveda je potrebno vsak tak korak sproti preverjati, da se izognemo napakam, ki se rade vrivajo.

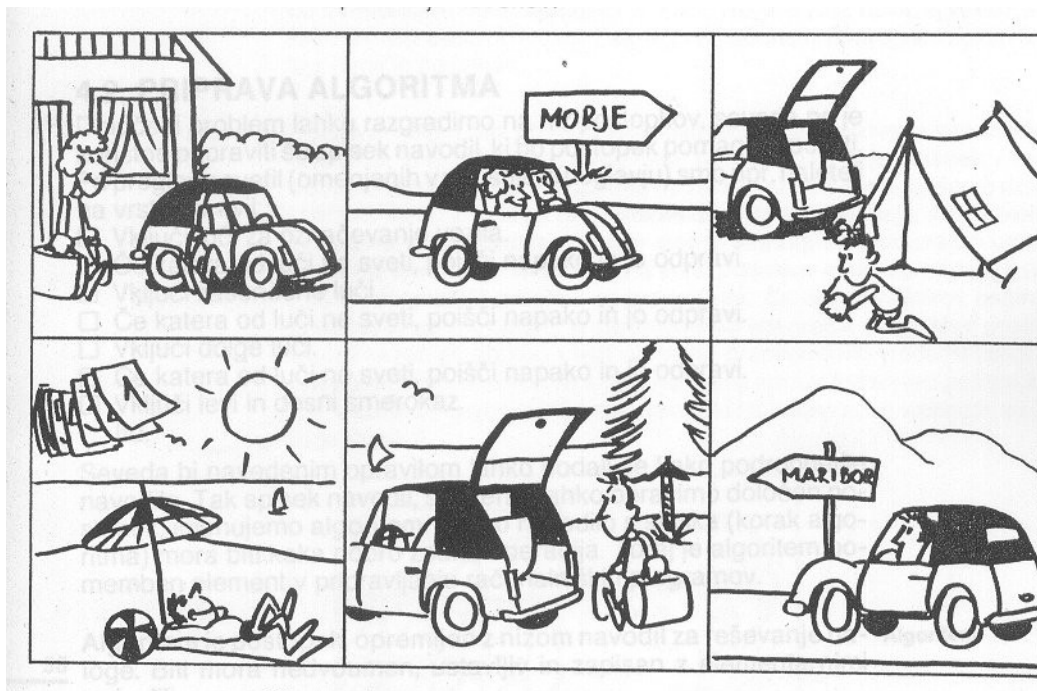
Ko opravimo razgrajevanje in ko menimo, da že poznamo rešitev problema, posamezne neodvisne podprobleme sestavimo v celoto in preverimo, če so ustrezni, prilagojeni reševanju z računalnikom, in zares najboljši. A ves postopek poteka še v okviru besednega izražanja, čeprav si lahko pomagamo tudi z dogovorjenimi (ali tudi lastnimi) grafičnimi simboli (o tem več kasneje).

Opisani postopek razreševanja problema ima v računalništvu tudi svoje ime; pravimo mu strukturirano programiranje ali sistematično razgrajevanje, ki je nepogrešljivo zlasti pri reševanju zapletenejših problemov. Tak način dela je dokaj razširjen v vsakodanem življenju, saj predstavlja najenostavnejši način reševanja problemov oziroma nalog. Vsaki nalogi lahko določimo še manjše in ožje podnaloge. To delamo toliko časa, dokler ne pridemo do nalog, ki jih lahko razrešimo brez težav (primeri, zidanje hiše – razgradimo na zidanje plošče, zidanje sten, zidanje strehe, zidanje pregradnih sten, ..., košnja trave – razgradimo na priprava kosilnice, košnja, pospravimo, kosilnico, itd..).

Vsak podproblem ali podnaloga torej sestoji iz enostavnejšega niza inštrukcij. Sistematično razgrajevanje teče po tako imenovani drevesni razgradnji “od zgoraj navzdol”, to je od splošnega in zapletenejšega k ožjemu in enostavnejšemu.

Zgled: dopust na morju

- pakiranje kovčkov
- potovanje na morje
- razpakiranje kovčkov
- kopanje in zabava
- priprava avtomobila
- pakiranje kovčkov
- potovanje domov
- razpakiranje kovčkov.



Vsak od naštetih podproblemov je videti enostaven, toda vsaka aktivnost potrebuje podrobnejšo razlago ali navodila. Priprava avtomobila obsega naprimer vrsto stopenj:

- tehnični pregled pri mehaniku
- odpravo večjih in manjših napak
- plačilo popravila
- notranje in zunanje pranje avtomobila
- polnenje goriva
- pregled gum

Vsak tak podproblem bi lahko razgradili še na niz opravil (inštrukcij):

tehnični pregled pri mehaniku obsega namreč:

- pregled zavornega sistema
- pregled sistema za vžig (svečk, vplinjač,...)
- pregled akumulatorja
- pregled svetil

in še vsak podproblem bi lahko razgradili še na niz podopravil... V računalništvu imenujemo nižje in dokaj zaključene podprobleme postopki.

Podrobnejši pregled osnovne razgradnje "problema" letnega dopusta prikaže, da se določeni postopki ponavljajo (pakiranje in razpakiranje kovčkov...); podobna aktivnost se ponavlja na več kot enkrat. Dolgočasno bi bilo navajati posamezne korake večkrat, zato je prednost takega dela tudi

ta, da že razčlenjene in opredeljene postopke lahko uporabimo večkrat oz. V različnih stopjah reševanja problema.

Prednost takega ravnanja v zvezi s pripravo programov so očitne:

- omogočajo enostavnejšo definicijo glavne naloge oz. problema
- vsak posamezen postopek in korak se lahko zapiše in testira samostojno
- postopki oz. podproblemi se lahko v glavnem poteku naloge uporabljajo večkrat.

Vprašanja:

- kako razumeš pojma: program in programiranje?
- Kaj je značilnost strukturiranega programiranja?
- Zapiši primer iz vsakdanjega življenja, v katerem boš lahko prikazal značilnosti strukturiranega programiranja in pojmov, kot sta: podprogram in postopek.

PRIPRAVA ALGORITMA

Določeni problem lahko razgradimo na niz postopkov, seveda pa je koristno pripraviti še spisek navodil, ki bo postopek pomagal izpeljati. Pri pregledu svetil (omenjenih v prejšnjem poglavju) smo npr. naleteli na vrsto opravil:

- vključi luči za označevanje vozila
- če katera od luči ne sveti, poišči napako in jo odpravi
- vključi zasenčene luči
- če katera od luči ne sveti, poišči napako in jo odpravi
- vključi dolge luči
- če katera od luči ne sveti, poišči napako in jo odpravi
- vključi levi in desni smerokaz
- itd....

Seveda bi navedenim opraviлом lahko dodali še kako podrobnejše navodilo (npr. kako odpravimo napako!). Tak spisek navodil, s katerim lahko opravimo določen postopek, imenujemo **algoritem**. Vsako navodilo s spiska (korak algoritma) mora biti kaka dobro znana operacija. Torej je algoritem pomemben element v pripravljanju računalniških programov.

Algoritem je postopek, opremljen z nizom navodil za reševanje naloge. Biti mora nedvoumen, ustavljiv in zapisan z elementarnimi navodili.

Nedvoumen je takrat, ko je izvajanje navodil v vseh okoliščinah natančno določeno. Tudi ustavljivost je njegova nujna lastnost (in to algoritmov, v katerih se delovanje ponavlja); ustavljiv e, če se v vseh okoliščinah konča v merljivem času. Algoritem mora biti sestavljen iz samih elementarnih navodil; elementarno navodilo je tisto, ki ga oblikovalec algoritma razume in zna izpolniti.

Sestavljanje algoritma ni vedno preprosto predvsem zato, ker algoritmično razmišljanje človeku ni domače in se ga mora šele priučiti. Prav preskok od tega, da nam je povsem jasno, KAJ je treba storiti, do tega, da znamo nedvoumno povedatu tudi, KAKO to storiti, zahteva dokaj truda.

Algoritem lahko zapišemo:

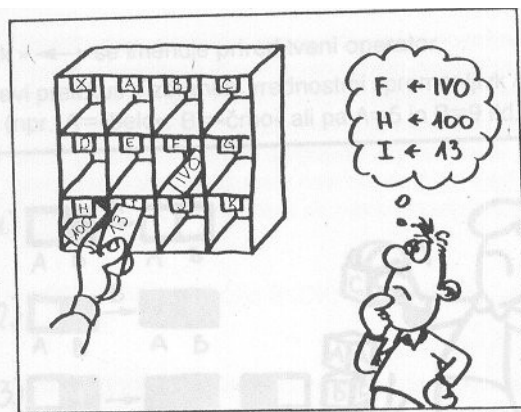
- z besednim opisom in
- z grafično predstavitvijo.

Sestavine besednega opisa algoritma so:

- naslov oz. ime algoritma;
- podatki, ki jih algoritem potrebuje in rezultati, ki nam jih vrne;
- opis postopka oz. algoritem sam

Računalniku je potrebno poleg postopka posredovati tudi **podatke**, s katerimi bo izvedel kako delo. Opredeliti je treba vse podatke, določiti kakšni so in kako so med seboj povezani. Zato je vsak računalniški program sestavljen iz algoritma in podatkov.

S pisanjem algoritma v bistvu sestavljamo navodila, ki povedo, kako s podatki ravnati, in to največkrat **neodvisno od vrednosti, ki jo podatek v tistem trenutku res ima**. Zato v algoritmu na zahtevanem mestu zapišemo oznako količine, njeno ime. **Med preizkusom algoritma in pozneje, med potekom programa, na vsakem mestu, kjer smo to ime zapisali, računalnik uporabi njeno trenutno vrednost. Ker se ta vrsta podatkov spreminja, ji (tako tudi v matematiki) pravimo spremenljivka**. Če si računalnikov pomnilnik predstavljamo kot omaro, si spremenljivko predstavljamo kot predal v njej. Med potekom algoritma (ali programa) v tem predalu hranimo dano vrednost. Ime spremenljivke je oznaka predala, njena trenutna vrednost pa tisto, kar v tem predalu shranimo. Če spremenljivki priredimo novo vrednost, se prejšnja izgubi.



Slika 48. Poenostavljena predstava spremenljivke v računalniku

Torej samo ime spremenljivke označuje mesto v pomnilniku, kjer poiščemo vrednost, ko spremenljivko pri delu potrebujemo. Na začetku dela so vrednosti spremenljivk nedoločene, zato jim moramo (vsaj enkrat) vrednosti predhodno prirediti.

Za ponazoritev sestavljanja besednega opisa algoritmov in dela s spremenljivkami je prav, če si ogledamo primere, iz katerih je mogoče razbrati princip delovanja računalnika.

1. naloga: povečaj število!

Postopek povečanja števila za B zahteva te korake:

- vzemi število (A)
- prištej mu število (B)
- shrani rezultat na mesto prvega števila (vrednost prvega števila se izgubi)

Sestava algoritma:

Ime algoritma: Povečaj število

Podatki: a, b

Rezultat: a

Postopek:

$a \leftarrow a + b$

konec

Rezultat je seveda matematični nesmisel, ni pa računalniški, saj se podobni postopki v računalniku res dogajajo. Opravi še preizkus za nekaj poljubnih vrednosti (številskih in nizovnih)!

2. naloga: zamenjaj vrednost dveh količin!

Vzemimo dve spremenljivki, npr A in B, katerih vrednosti želimo zamenjati, torej: kar je bilo

prvotno v spremenljivki A, naj bo na koncu v spremenljivki B in obratno. V spremenljivki se lahko skriva število, znak, beseda,... kar pa nas sedaj ne zanima. Če ti pojem spremenljivke še ni jasan, poiskusi problem rešiti s primerom dveh lončkov: v enem so kroglice bele barve, v drugem lončku pa kroglice modre barve. Naloga zahteva, da zamenjamo vsebino kroglic (obeh lončkov) v obeh lončkih!

Variante postopka zamenjave bi bile:

- vrednost v A vstavimo v B: vrednost iz A prekrije vrednost v B in tako imamo v obeh spremenljivkah vrednosti iz A. Izgubili smo podatek v B.
- Vrednost v B vstavimo v A: rezultat je podoben zgornjemu primeru, saj sva prav tako izgubili podatek iz A
- Za pravilno rešitev tega problema potrebujemo novo (začasno) spremenljivko C, v katero začasno prenesemo eno od vrednosti spremenljivk A oziroma B.

Sestavimo algoritem po dogovorjenem postopku:

Ime algoritma: Zamenjava

Podatki: a,b

Rezultat: zamenjane vrednosti a,b

Postopek:

c <= a

a <= b

b <= a

konec

znak <= se imenuje **prireditveni operator**.

Opravi preizkus z izbranimi vrednostmi spremenljivk A in B (npr. A=belo, B=črno ali pa A=5 in B=9 itd...)!

3. naloga: štetje od 1 do 20!

V tej nalogi bomo uporabili znanje sestavljanja algoritmov iz prejšnjih dveh primerov.

Spremenljivki I bomo prištevali 1 tako dolgo, dokler ne bo dosegla številke 20.

Ime algoritma: Štetje od 1 do 20

Podatki: I

Postopek:

I <- 1;

izpiši I

če je I < 20 potem I <- I + 1 in nazaj v stavek: izpiši I, sicer konec

konec

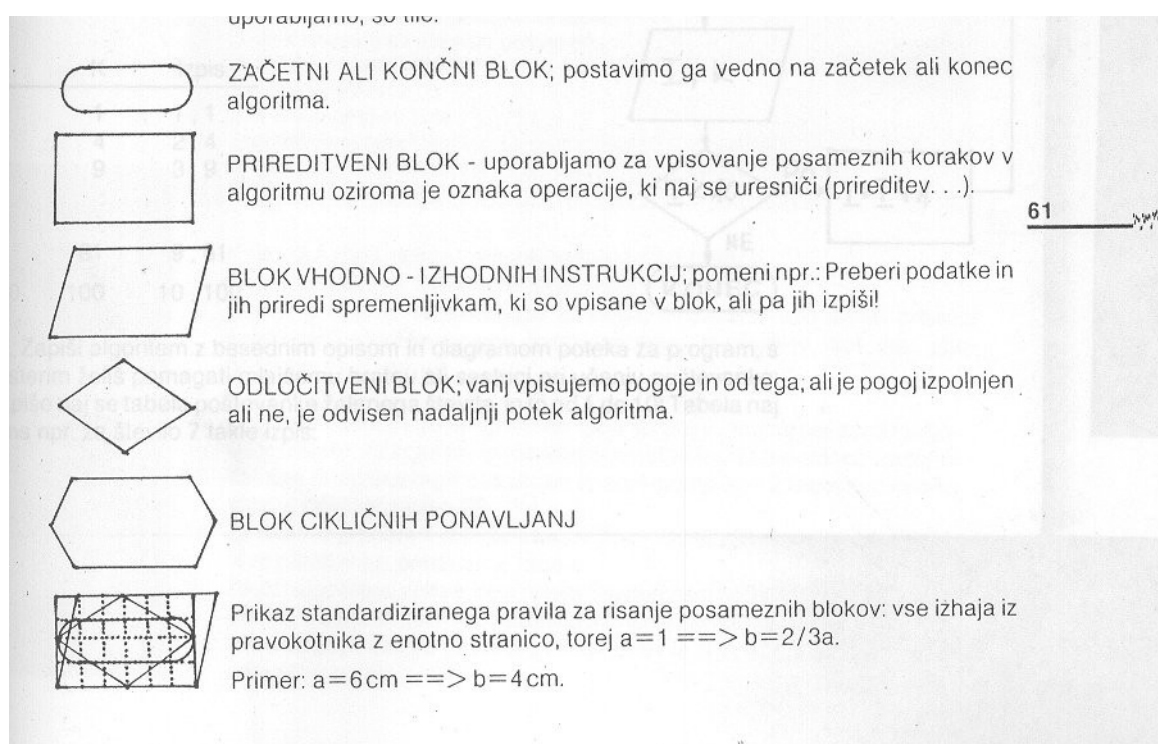
Problem sicer ni težak, a poudariti velja člen I <- I + 1, ki v bistvu opravlja štetje. Štetje je pogosta naloga v programiranju. Običajno jo rešimo tako, da uvedemo posebno spremenljivko, ki ji pravimo števec. Ta spremenljivka (v našem primeru I) se obnaša podobno kot resnični števec (npr. števec kilometrov v avtomobilu, električni števec itd...). Vsakič, ko se izpiše nova številka, se števec poveča za 1; to pomeni, da se števeni spremenljivki poveča vrednost za 1.

Opravimo preizkus. Za to je zelo pripravna t.i. **tabela rezultatov ali preizkusna tabela**.

I	Izpis
1	1
2	2
3	3
4	4
5	5
6	6

7	7
8	8
...	...
19	19
20	20

Besedni opis algoritma je največkrat (še posebno, če je daljši) nepregleden. V takih primerih se je uveljavilo zapisovanje algoritmov v drugih, nazornejših oblikah, med katerimi je grafični zapis algoritma z dogovorjenimi bloki; po tem se ta način imenuje zapisovanje algoritma z blok diagramom ali diagramom poteka. V takem diagramu s posebnimi (standardiziranimi) simboli označimo posamezne vrste navodil, simbole med seboj povežemo in opredelimo vrstni red, po katerem se navodila izvršujejo. Simboli, ki jih največkrat uporabimo, so:



Za rešitev problema je dovolj, če algoritem zapišemo na enega od naštetih načinov: z besednim zapisom ali z diagramom poteka.

1. Zapiši diagram poteka za vse tri primere algoritmov, ki smo jih reševali.
 2. Zapiši algoritem z besednim opisom in diagramom poteka za program, ki naj izpiše števila od 1 do 10 in sicer tako, da ob številu izpiše še kvadrat tega števila. Preveri algoritem s preizkusno tabelo!
- Ime algoritma: Števila in kvadrati števil od 1 do 10
 Podatki: I,K
 Rezultati: I,K
 Postopek:
 $I \leftarrow 1$
 $K \leftarrow I * I$
 izpiši I in K

če je $I < 10$ potem $I \leftarrow I + 1$ in nazaj v stavek $K \leftarrow I * I$ sicer konec
konec

Preizkusna tabela:

I	I	Izpis
1	1	1,1
2	4	2,4
3	9	3,9
...		
9	81	9,81
10	100	10,100

3. Zapiši algoritem z besednim opisom in diagramom poteka za program, s katerim želiš pomagati mlajšemu bratcu ali sestrici pri učenju poštevanke:
Izpiše naj se tabela poštevanke željenega števila in to od 1 do 10! Tabela naj ima npr. za število 7 naslednji izpis:

$7 * 1 = 7$

$7 * 2 = 14$

$7 * 3 = 21$ itd...

Algoritem preveri z preizkusno tabelo!

Ime algoritma: Poštevanka poljubnega števila od 1 do 10

Podatki: N, I, R

Postopek:

Vstavi število (N) za željeno poštevanko:

$I \leftarrow 1$

ponavljaj

$R \leftarrow N * I$

izpiši v obliki: $N * I = R$

$I \leftarrow I + 1$

dokler ne postane $I > 10$

konec

Pojmi, ki si jih velja zapomniti:

- algoritem
- podatki
- spremenljivka
- tabela rezultatov ali preizkusna tabela
- začetni in končni blok
- prireditveni blok
- vhodno-izhodni blok
- odločitveni blok
- blok cikličnih ponavljanj

Vprašanja:

1. Kako razumeš pojma program in programiranje?
2. Kaj je značilnost strukturiranega programiranja?
3. Zapiši primer iz vsakdanjega življenja, v katerem boš lahko prikazal značilnosti strukturiranega programiranja in pojmov kot sta: podprogram in postopek.
4. Kaj je algoritem?
5. Oglej si enostavnejši program na računalniku in skušaj prikazati njegovo strukturo. Ali znaš zapisati tudi njegov algoritem z besednim opisom?

6. Kakšna je razlika med podatkom in spremenljivko? Zahtevano razliko skušaj razložiti.
7. Zapiši algoritem z besednim opisom in diagramom poteka za pripravo programa, s katerim želimo poljubno izbranemu številu zapisati predhodno in naslednje število (Izbrali smo število 5 --> predhodnjik je 4, naslednjik pa 6).
8. Zapiši algoritem z besednim opisom ali diagramom poteka za primer, ki bi ti pomagal izračunati srednjo vrednost ocen, ki jih imaš trenutno v redovalnici za vsak predmet.
9. Pripravi algoritme z besednim opisom ali diagramom poteka za primere programov, s katerimi želimo določiti:
 - a) vsoto naravnih števil od 1 do 100
 - b) vsoto sodih števil od 1 do 100
 - c) vsoto naravnih števil od m do n
 - d) vsoto sodih števil od 1 do n
10. V datoteki računalnika se nahaja N imen varčevalcev neke banke in višina njihove hranilne vsote. Zapiši algoritem, ki bo izpisal poročilo s takim vrstnim redom podatkov: zaporedno število, ime in priimek, hranilna vsota, obresti in skupna vrednost hranilne vloge. Obrestna mera je vhodni podatek. Opravi preizkus algoritma!
11. Zapiši algoritem za rešitev problema, ki bo preveril in izpisal, ali je poljubno število N parno ali neparno.
12. Pripravi algoritem za pripravo programa, ki naj določi in izpiše vsa naravna števila od 1 do 3000, ki so deljiva z 3. Opravi preizkus algoritma.
13. Zapiši algoritem, ki bo omogočal zaokrožitev večmestnega decimalnega števila na dve decimalni mesti.
14. Pripravi algoritem, s katerim izračunaš porabo goriva osebnega avtomobila z naslednjimi vhodnimi podatki: datum polnjenja, količina goriva in število prevoženih kilometrov. Izračun povprečne porabe goriva želiš dobiti po vsakih 10 polnjenjih in na koncu vsakega leta.
15. Joule je enota za merjenje energije (tudi za energijo, ki jo vsebuje hrana). Včasih je za to uporabljena enota Kalorija (cal). Pripravi algoritem, ki bo dekletom preračunaval kalorije v joule in obratno ($1 \text{ cal} = 4,2 \text{ joula}$).
16. Pripravi algoritem, ki prebere celo pozitivno število (do 5000) in ga izpiše z rimskimi številkami!
17. Zapiši algoritem, ki bo prebral tri realna števila in zapisal produkt največjega z najmanjšim!
18. Smučarske skoke ocenjuje 5 sodnikov z ocenami od 1 do 20. V skupni oceni se največja in najmanjša ocena ne vpoštevata, iz ostalih pa se izračuna povprečna vrednost. Pripravi algoritem, ki prebere vse ocene in izpiše skupno oceno skoka!
19. Leta 1000 je bila višina kapnika 3mm, nato pa se je vsakih 10 let povečeval za 6mm. Sestavi algoritem, s katerim lahko izračunaš, kolikšna bo višina kapnika leta 2000 in katerega leta bo dosegel višino 1,5m!

PROGRAMSKI JEZIKI

V dosednji obravnavi še ni bil potreben noben programski jezik, temveč le strokovno znanje za razčlenitev problema in znanje za sestavo algoritma. Algoritem v bistvu že predstavlja rešitev problema, njegove stavke moramo le prekodirati oz. prevesti v ukaze izbranega jezika. Kdor pozna kak računalniški jezik (npr. pascal ali basic), ve, da je potrebno določene besede oz. stavke algoritma le prevesti v ustrezne angleške ukaze in program je hitro pripravljen za uporabo. Seveda celoten postopek ni tako preprost, še zlasti pa ni bil v prvih letih računalništva. Torej program zapisan v programsjem jeziku je tudi algoritem!

Računalnik se odziva le na dva simbola (0 in 1 oz. DA in NE). A za sporočanje, kakršnega je navajen človek, sta dokaj nenavadna in nesprejemljiva. Zato bi bilo najbolje, če bi računalnik lahko naučili človeškega jezika. A do tega je še dolga pot (ali pa tudi ne). Znanstveniki so namreč že dosegli precejšen napredek. Doslej so pripravili na ducate tako imenovanih **programskih jezikov**, ki se jih je mogoče razmeroma zlahka in hitro naučiti in so človeku bližji kot računalnikov strojni

jezik. To pomeni, da so lažje razumljivi, hkrati pa bliže strojnemu jeziku kot naravni jezik.

Z računalnikom lahko torej komuniciramo le s posebno vrsto jezika, ki ga imenujemo **strojni jezik**. Določene naloge opravi le, če mu damo napotke v tem jeziku. In tako je bilo tudi vrsto let, še posebno v začetni dobi uporabe računalnikov, ko so bili vsi programi pisani v tem osnovnem programskem jeziku. To je bilo za programerje, pa tudi za druge uporabnike (katerih krog je bil tudi zato tako ozek) zelo zahtevno, težavno in zapleteno opravilo. Kaj kmalu je postalo jasno, da pisanje programov v strojnem jeziku ne pomaga širiti uporabe računalnikov. Razloga sta predvsem dva:

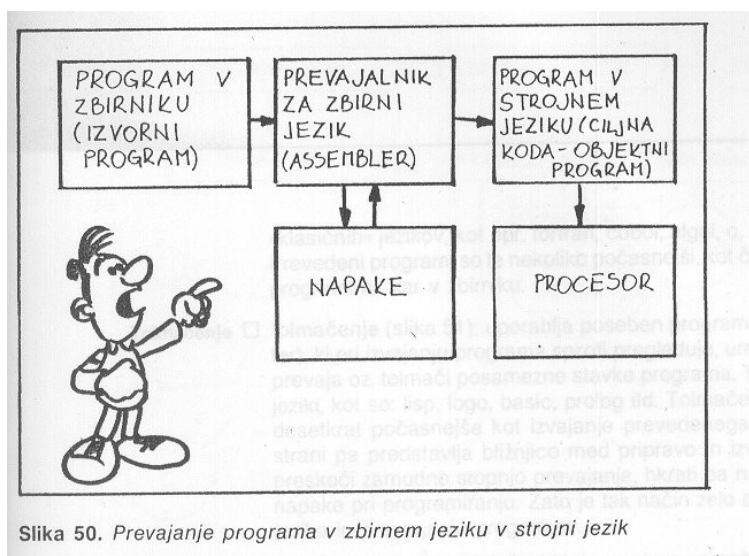
- pisanje programov z zaporedjem ničel in enic je tako nepregledno, da na tak način lahko rešujemo le najpreprostejše probleme;
- strojni jezik se pogosto razlikuje glede na vrsto računalnikov, kar pomeni nezdržljivost (neprilagojenost, medsebojno neusklajenost, neprenosljivost) narejenih programov.

Kako je sestavljen strojni jezik? Sestavlja ga niz strojnih inštrukcij, a vsaka izmed njih sestoji iz niza dvojiških števil (cifer) določene dolžine. Ta niz se deli na dva dela: operacijskega in naslovnega. Operacijski del predstavlja kodirano obliko določene elementarne operacije, naslovni del pa kodirano obliko učinkovitih naslovov v spominu ali/in v registrih, na katerih vsebinah se želi opravljati določena elementarna operacija.

Programer, ki je uporabljal strojni jezik, je bil resnično zelo blizu računalniku; z njim sta govorila isti jezik. Uporabljal se je vse dotlej, dokler se uporabniki niso domislili, da bi vsaki inštrukciji določili črkovno kratico (simbol). Tako smo dobili t.i. simbolične programske jezike, ki jih delimo v dve skupini:

- nižje programske jezike
- višje programske jezike

Značilen predstavnik **nižjih programskih jezikov** je t.i. **zbirni jezik** (ang. assembly language), ki strojne inštrukcije predstavlja v simbolični in (ali) številčni obliki in se ne more neposredno izvajati na računalniku, saj ga je potrebno prej prevedi v strojni jezik. Kako poteka pisanje programov v teh jezikih? Program najprej napišemo v zbirnem jeziku (ki je navadno različen glede na tip računalnikov). Uporabi se kot vhodni t.i. izvirni program v zbirniku (ang. assembler); zbirnik je program, ki prevede program v zbirnem jeziku v strojni jezik (strojno kodo). Zbirnik preveri, ali so uporabljeni pravilni nazivi (simboli) in oblike ukazov. Če program nima napak, zbirnik oblikuje izhodno datoteko, ki jo imenujemo ciljna koda; to je sedaj program, preveden v strojni jezik, ki se lahko izvede na računalniku. Vsak simboličen ukaz se v zbirniku prevede v en strojni ukaz, torej je v tem pogledu odnos še vedno 1:1; vsakemu strojnemu ukazu ustreza en simboličen ukaz (ukaz zbirnika).

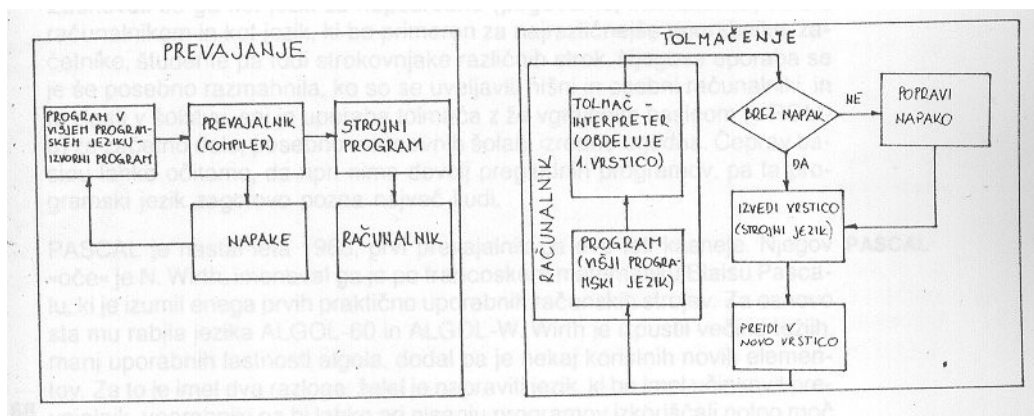


Vsekakor pa je pisanje programov v zbirnem jeziku enostavnejše kot pisanje s strojnim jezikom. Tako kot vsi programerski jeziki ima tudi ta svoje pomanjkljivosti in prednosti. Med prednostmi štejemo: izkušenemu programerju nudi veliko možnosti za natančen nadzor nad uporabo, za ukaze in seveda za doseganje hitre izvedbe oz. Poteka. Ob tem je treba omeniti še eno izmed dokaj uspešnih izboljšav te vrste jezikov, to je uvedbo t.i. makrojev oz. makroukazov (ang. macro instructions). Makroji so okrajšave za standardne nize ukazov; programerjem olajšujejo delo v zbirniku tam, kjer bi se moral niz kakih ukazov ponoviti na več mestih programa. Makroukaz se vključi v program v zbirniku skupaj z normalnimi ukazi, pri čemer jih prevajalnik za zbirni jezik prevede v niz normalnih ukazov. To rešuje programerje, kadar ponavljajo pisanje enih in istih skupin ukazov in zmanjšuje možnost, da bi delali napake.

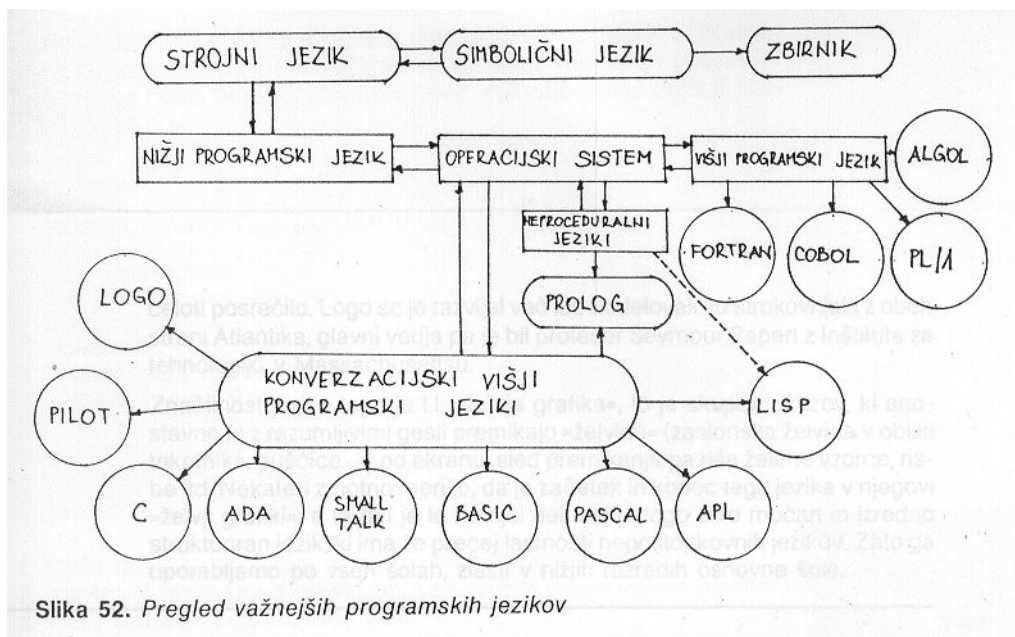
Okoli leta 1950 so veliko razpravljali o uporabi ustreznih programskih jezikov; mnogi uporabniki so zagovarjali nadaljnje izpopolnjevanje in uporabo programskih jezikov izključno na podlagi zbirnika (nižjih simboličnih programskih jezikov), drugi so pa menili, da se mora programiranje poenostaviti in približati človeku. Ta skupina je tudi prevladala, zato so se pojavile nove vrste jezikov, t.i. **višji programski jeziki** – problemsko orientirani jeziki.

Ti so precej neodvisni od strukture določenega računalniškega sistema in so tudi bližji človekovemu načinu zapisovanja. So problemsko usmerjeni, kar pomeni, da je njihova struktura prirejena področju, za katerega so namenjeni. Tudi ti potrebujejo prevajalnike, ki skrbijo za "razumevanje" med zapisanim programom v višjem programskem jeziku in strojnim jezikom, ki ga razume računalnik. Ne vsebujejo posameznih strojnih ukazov, temveč pomensko simbolične ukaze, ki pomenijo niz strojnih ukazov izvedbe (razmerje ni več 1:1, temveč 1:n – en ukaz v višjem jeziku se prevede v več ukazov v strojnem jeziku). Za pretvorbo višjega programskega jezika v strojni jezik se v glavnem uporabljata dva načina:

- **prevajanje**: uporablja se poseben program, imenovan **prevajalnik** (ang. compiler), ki celoten program prevede v strojni jezik računalnika. Šele v prevedeni obliki se lahko izvede. To pot uporablja večina "klasičnih" jezikov, kot naprimer fortran, cobol, algol, c, ada, pascal,...Prevedeni programi se le nekoliko počasnejši, kot če bi rešitev problema programirali kar v zbirniku.
- **tolmačenje**: uporablja poseben program **tolmač** (ang. interpreter), ki pri izvajanju programa **sproti** pregleduje, ureja, javlja napake in prevaja oz. tolmači posamezne stavke programa. Tak pristop uporabljajo jeziki kot: lisp, logo, basic, prolog,... Tolmačenje je ponavadi okrog desetkrat počasnejše kot izvajanje prevedenega programa. Po drugi strani pa predstavlja bližnjico med pripravo in izvajanjem programa – preskoči zamudno stopnjo prevajanja, hkrati pa nas sproti opozarja na napake pri programiranju. Zato je tak način zelo dobrodošel pri učenju jezika in preverjanju programov.



Katere jezike in prevajalnike bi bilo smotno uporabljati v določenih okoliščinah, če bi se znašli v vlogi programerja? Odgovor je odvisen predvsem od področja in narave programa. Pri pustolovskih igrah, kjer hitrost ni zelo pomembna in kjer večji del obdelave poteka v obliki niza besed, lahko uporabimo višji programski jezik basic; posamezne ukaze bo tolmač oz. interpreter prevajal sproti. Probleme, ki potrebujejo hitrejšo obdelavo, kot npr. poslovne programe, kjer bi bilo množico matematičnih operacij težko napisati v strojnem jeziku, bi bilo primerno pisati v enem izmed višjih programskih jezikov in potem prevesti s prevajalnikom, saj bi bil tolmač v tem primeru prepočasen. Za hitre in razburljive arkadne igre, ki imajo tudi obsežno grafično podporo, pa bi strojni jezik ustrežal, saj bi bil tudi prevajalnik prepočasen.



Slika 52. Pregled važnejših programskih jezikov

Primer kode v zbirnem jeziku (del kode Quake 3 arena), da se še danes uporablja zbirni jezik za hitre arkadne igre:

-----datoteka-----

```

;=====
;Copyright (C) 1999-2005 Id Software, Inc.
;
;This file is part of Quake III Arena source code.
;
;Quake III Arena source code is free software; you can redistribute it
;and/or modify it under the terms of the GNU General Public License as
;published by the Free Software Foundation; either version 2 of the License,
;or (at your option) any later version.
;
;Quake III Arena source code is distributed in the hope that it will be
;useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
;MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
;GNU General Public License for more details.
;
;You should have received a copy of the GNU General Public License
;along with Foobar; if not, write to the Free Software
;Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
;=====

;
; qftol -- fast floating point to long conversion.
;

segment .data

temp    dd 0.0
fpucw   dd 0

; Precision Control Field , 2 bits / 0x0300
; PC24 0x0000   Single precision (24 bits).
; PC53 0x0200   Double precision (53 bits).
; PC64 0x0300   Extended precision (64 bits).

```

```

; Rounding Control Field, 2 bits / 0x0C00
; RCN 0x0000 Rounding to nearest (even).
; RCD 0x0400 Rounding down (directed, minus).
; RCU 0x0800 Rounding up (directed plus).
; RC0 0x0C00 Rounding towards zero (chop mode).

```

```

; rounding towards nearest (even)
cw027F dd 0x027F ; double precision
cw037F dd 0x037F ; extended precision

```

```

; rounding towards zero (chop mode)
cw0E7F dd 0x0E7F ; double precision
cw0F7F dd 0x0F7F ; extended precision

```

```

segment .text

```

```

;
; int qftol( void ) - default control word
;

```

```

global qftol

```

```

qftol:
    fistp dword [temp]
    mov eax, [temp]
    ret

```

```

;
; int qftol027F( void ) - DirectX FPU
;

```

```

global qftol027F

```

```

qftol027F:
    fnstcw [fpucw]
    fldcw [cw027F]
    fistp dword [temp]
    fldcw [fpucw]
    mov eax, [temp]
    ret

```

```

;
; int qftol037F( void ) - Linux FPU
;

```

```

global qftol037F

```

```

qftol037F:
    fnstcw [fpucw]
    fldcw [cw037F]
    fistp dword [temp]
    fldcw [fpucw]
    mov eax, [temp]
    ret

```

```

;
; int qftol0F7F( void ) - ANSI
;

```

```

global qftol0F7F

```

```

qftol0F7F:
    fnstcw [fpucw]
    fldcw [cw0F7F]
    fistp dword [temp]
    fldcw [fpucw]
    mov eax, [temp]
    ret

```

```

;
; int qftol0E7F( void )
;

```

```

global qftol0E7F

```

```

qftol0E7F:
    fnstcw [fpucw]
    fldcw [cw0E7F]
    fistp dword [temp]
    fldcw [fpucw]

```

```

        mov eax, [temp]
        ret

;
; long Q_ftol( float q )
;

global Q_ftol

Q_ftol:
        fld dword [esp+4]
        fistp dword [temp]
        mov eax, [temp]
        ret

;
; long qftol0F7F( float q ) - Linux FPU
;

global Q_ftol0F7F

Q_ftol0F7F:
        fnstcw [fpucw]
        fld dword [esp+4]
        fldcw [cw0F7F]
        fistp dword [temp]
        fldcw [fpucw]
        mov eax, [temp]
        ret

```

-----datoteka-----

Oglejmo si nekaj pomembnejših predstavnikov višjih programskih jezikov (v oklepaju je angleška beseda, iz katerih je sestavljena kratica) in njihovo delitev!

FORTRAN (FORmula TRANslation) je nastal okrog leta 1955 kot program za prevajanje matematičnih postopkov v računalniške programe. Sprva zelo enostavni verziji so se kmalu pridružile izpeljanje, kot npr. fortran II, III, IV, V, fortran 77, fortran 90 in cela vrsta strukturiranih fortranov. Uporablja se predvsem za reševanje znanstvenih in tehničnih problemov.

COBOL (COmmon Business Oriented Language) je nastal okrog leta 1960, vpeljan pa je bil sedem let kasneje. Namenjen je predvsem pisanju programov za poslovno-ekonomsko problematiko. Strukturiran je tako, da skuša biti čim bolj podoben naravnemu (angleškemu) jeziku in razumljiv tudi neprogramerjem, zlasti vodstvenim delavcem.

BASIC (Beginner's All-purpose Symbolic Instruction Code) je nastal leta 1965 kot preprost jezik za učenje programiranja in kot stopnjička k fortranu. Zasnovali so ga kot jezik, ki bo primeren za najrazličnejše uporabnike: začetnike, študente pa tudi strokovnjake različnih strok. Njegova uporaba se je še posebno razmahnila, ko so se uveljavili hišni in osebni računalniki in to zlasti v šolstvu, saj je uporaba tolmača z že ugrajenim basicom (v ROMu) za začetno delo, posebno v osnovnih šolah, izredno ugodna. Čeprav basicu očitamo, da npr. nima dovolj preglednih programov, pa ta programski jezik zagotovo pozna največ ljudi.

PASCAL je nastal leta 1968, prvi prevajalnik pa dve leti kasneje. Njegov "oče" Niklaus Wirth; imenoval ga je po francoskem matematiku Blaisu Pascalu, ki je izumil enega prvih praktično uporabnih računalniških strojev. Za osnovo sta mu rabila ALGOL-60 in ALGOL-W. Wirth je izpustil večino težjih, manj uporabnih lastnosti algola, dodal pa je nekaj koristnih elementov. Za to je imel dva razloga: želel je napraviti jezik, ki bo imel učinkovit prevajalnik, uporabniki pa bi lahko pri pisanju programov izkoriščali polno moč jezika. Pascal se je hitro razširil po svetu in v nekaj letih postal eden pomembnejših programskih jezikov: je šolski primer za lepo izdelan in strukturiranemu programiranju naklonjen programski jezik. Ima vrsto koristnih prijemov, ki jih najdemo tudi pri drugih programskih jezikih in programerju naravnost ponuja možnost, da piše programe, kar se da strukturirano.

C je zelo sposoben višji programski jezik (avtor Dennis Ritchie), ki se je razvil ob operacijskem sistemu UNIX. Večina sistemskih programov drugih operacijskih sistemov je namreč napisanih v nižjih programskih jezikih (zbirnem jeziku). Programski jezik c je posebno primeren za pisanje operacijskih sistemov in sistemskih servisnih programov.

LOGO – Ime programskega jezika logo ne izhaja iz kratice, temveč je izpeljanka grške besedice LOGOS (gr. logos – misel, razum). Njegovi avtorji so želeli ustvariti jezik, ki bi povezal način programiranja in način človekovega razmišljanja. Začetnike, zlasti otroke, naj bi spodbujal, da bi probleme reševali z razstavljanjem na manjše in lažje rešljive enote. In to se jim je tudi v celoti posrečilo. Logo se je razvijal več let. Sodelovali so strokovnjaki z obeh strani Atlantika, glavni vodja pa je bil profesor Seymour Papert z Inštituta za tehnologijo v Massachusettsu. Značilnost jezika logo je t.i. "želvja grafika", to je skupek ukazov, ki enostavno in z razumljivimi geslo premikajo "želvico" (zaslonska želvica v obliki trikotnika, puščice,...) po ekranu, sled premikanja pa riše željene vzorce, risbe, itd. Nekateri zmotno menijo, da je začetek in konec tega jezika v njegovi "želvji grafiki"; a ta del je le manjši del, saj je logo zelo močan in izredno strukturiran jezik, ki ima že precej lastnosti nepostopkovnih jezikov. Zato ga uporabljamo po šolah, zlasti v nižjih razredih osnovne šole.

Višjih programskih jezikov je doslej že več kot 100. Delimo jih v dve vrsti:

- jezike, s katerimi predvsem opisujemo, kaj želimo narediti, in
- jezike, s katerimi tudi podrobneje povemo, kako to storiti.

Prva skupina jezikov je bližje naravnemu, druga pa strojnemu jeziku računalnika. Jezike prve vrste imenujemo nepostopkovne ali opisno usmerjene (neproceduralni jeziki), jezike druge vrste pa postopkovne ali ukazovalno usmerjene (proceduralni jeziki). Postopkovni jeziki opisujejo program podobno kot kuharska knjiga kuho: vzemi to ali to, stori prvo, drugo, tretje itd. Danes pa že obstajajo jeziki, ki se programiranja lotevajo na popolnoma drugačen način: program npr. opišemo s pomočjo lastnosti željene rešitve. Tako lastnost imajo nepostopkovni jeziki, jeziki pete generacije računalnikov, ki so po svoji strukturi in logiki še najbolj prilagojeni človeškemu načinu razmišljanja: ljudem se naj ne bi bilo treba prilagajati računalniški programski logiki. Pomemben predstavnik takih jezikov je PROLOG.

PROLOG (PROgramming in LOGic) – je bil prvi poskus oblikovanja jezika, ki naj bi programerjem omogočal definiranje rešitev s pomočjo logike. Zasnovali so ga že leta 1975 na Univerzi v Marseillu. To je preprost, vendar presenetljivo močan programski jezik, ki se vedno bolj uveljavlja in bo kmalu na široko prodril med uporabnike. Posebno učinkovit je pri simboličnem, nenumeričnem procesiranju in pri delu z bogatimi podatkovnimi strukturami. Med njim in postopkovnimi jeziki obstaja bistvena razlika: postopkovni jeziki opisujejo postopke, to je, kako pridemo od vhoda do izhoda oz. od danih podatkov do rezultatov, v prologu pa definiramo samo razmerja med podatki in rezultati, prevajalnik pa mora sam poiskati postopek operacij, ki prevedejo podatke v rezultate tako, da ustrezajo zahtevam.

In kako se bojo razvijali programski jeziki v prihodnje? Kaže, da programiranje nenehno napreduje in je ljudem vse bližje. Prvi začetki so bili v strojni kodi, nato so se pojavili zbirni jeziki, pred približno 35 leti pa so se začeli pojavljati višji programski jeziki, tipa fortran, cobol, basic itd. Zadnjih 15 let se poleg najnovejših postopkovnih jezikov tipa pascal in C pojavljajo in uveljavljajo nepostopkovni jeziki tipa prolog. Veliko študij in projektov kaže, da gre za razvoj jezikov in programiranja v to smer, torej k približevanju ljudem in njihovem načinu razmišljanja in sklepanja. Računalniki pete generacije, ki temeljijo na nepostopkovnih jezikih, so že taki, da je njihova strojna oprema prilagojena tem jezikom. Ti načrti imajo močno finančno podporo zlasti do takrat, ko so Japonci in Američani pričeli hud boj za prevlado na področju računalništva. Danes je večina aplikacij narejena v postopkovnih jezikih in le peščica v nepostopkovnih;

nepostopkovni se bodo morali še precej izpopolniti, če bodo hoteli pridobiti veljavo, kakršno imajo najbolj razširjeni jeziki.

Pojmi, ki si jih velja zapomniti!

programski jeziki, nižji – višji programski jeziki, strojni jezik, simbolični programski jezik, zbirni jezik, zbirnik (assembler), makro-instrukcije, prevajanje in tolmačenje, nepostopkovni – postopkovni jeziki

Naloge:

- opiši razliko med nižjim in višjim programskim jezikom?
- kakšna je razlika med strojnim in simboličnim programskim jezikom?
- kako razumeš pojem strojni jezik?
- v zadnjem času se vse bolj uporabljajo jeziki z makro instrukcijami. Kaj ti pove ta pojem?
- Opiši razliko med tolmačenjem in prevajanjem.
- Poveži levo in desno stran in s tem prikaži področje uporabljenosti posameznega programskega jezika: FORTRAN (tehnični in matematični problemi), COBOL (poslovno-ekonomski problemi), BASIC (slabše strukturiran jezik začetnikov), PASCAL (strukturirani jezik za) LOGO (strukturirani jezik začetnikov in otrok), C (pisanje operacijskih sistemov).
- naštej pomembne razlike med postopkovnimi in nepostopkovnimi jeziki!
- zakaj imenujemo nepostopkovne jezike tudi jezike umetne inteligence?

VNOS, TESTIRANJE IN OPREMA PROGRAMA

Računalnik sam po sebi ne naredi ničesar: če mu zapovemo, naj naredi neumnost, jo bo zagotovo zagrešil; če mu narekujemo napačen način reševanja problema, ga bo po napačni poti tudi izpeljal. Če želimo torej z računalnikom rešiti kak problem, moramo postopke reševanja vnesti natančno in strokovno pravilno.

Ko z algoritmom rešimo določen problem, je treba rešitev kot delovno nalogo prenesti v računalnik, seveda v izbranem programskem jeziku.

Ob sestavljanju algoritmov je treba upoštevati nekaj značilnih algoritemskih navodil oz. tako imenovanih osnovnih stavkov. V vsakem algoritmu oziroma v programu najdemo izvršilni stavek, ki označuje izvršitev določenih nalogin stavek z odločitvijo, ki predstavlja skoke na določene ukaze. Seveda pa ni nujno, da teče program ves čas le v isti smeri. Lahko se vrne na stavek, v katerem je že bil; tako dobimo priljubljeno programsko strukturo – zanko.

Ko je program v izbranem programskem jeziku napisan, ga torej vpišemo v računalnik. Nato ga poskusno pregledamo in hkrati popravimo glavne napake. Začetniki so seveda presenečeni zaradi številnih in nenavadnih napak, ki jih računalnik sporoči med prevajanjem, tolmačenjem ali izvajanjem programa. Obravnava napak v programu je za računalnik zahteven in zapleten postopek, v katerem žal ni mogoče predvideti vseh napak, ki jih zagreši programer. Tako lahko manjkajoči ali odvečni znak v programu povzroči, da računalnik sporoči napako, s katero najustrezneje opiše nastalo situacijo. Če je situacija taka, da računalnik ne najde praviga zapisa zanjo, javi neustrazno sporočilo ali prekine nadaljnji potek dela. Zato je že na začetku potrebno poskrbeti, da je program napisan pravopisno pravilno, saj ni tehtnega opravičila za manjkajoče vejice, podpičja, dvopičja in nepravilno napisane ključne besede programskega jezika.

Programske napake delimo v:

- sintaktične (pravopisne)
- logične in
- semantične (snovalske)

Sintaktične napake nastanejo zaradi slovnično nepravilnih zapisov ukazov, zato jih je še mogoče odpraviti. Logične napake so že zahtevnejše, saj zaradi njih program ne deluje pravilno; so v bistvu taktične napake (pojavi se naprimer pri prekoračitvi v tabeli, itd...). Semantične napake so

strateške. Te so najhujše, saj zahtevajo veliko premišljevanja in popravljanja (če v zasnovi uporabimo nepravilen algoritem ali algoritem z napakami ali zgrešeno zasnovo programa,...). Sintaktične napake odkrivamo brez težav s pomočjo sporočil prevajalnika, logične s premišljevanjem in posebnimi programi za odpravljanje napak (razhroščevalnik – debugger).

Pri semantičnih napakah pa je odpravljanje najtežje, saj je zasnova najpomembnejša in najzahtevnejša stopnja dela. Zato velja: Če napišemo program, ki je zgrešen v zasnovi, je najbolje, če začnemo delo z vsemi koraki programiranja znova.

Da bi bilo napak kar najmanj, se moramo (ne glede na to, kateri jezik bomo uporabili za zapis programa) naučiti slovničnih pravil: poznati moramo zgradbo in pomen stavkov. Za popoln zapis jezika je torej potrebno dvoje:

- opis zgradbe jezika (jezikovna pravilnost – sintaksa); vedeti moramo, kako so zgrajeni pravilni programi, to je, katera zaporedja znakov pomenijo smiselni program;
- opis pomena (semantika), ki pove, kaj naj sintaktično pravilen program počne.

Semantiko jezika navadno opišemo z besedami, sintakso pa je z besedami težko opisati, zato navadno uporabimo grafični prikaz; to pomeni, da jo prikažemo v obliki t.i. pravopisnih (sintaktičnih) diagramov.

Ko so osnovne programske in pravopisne napake odpravljene, je najbolje, če program shranimo na ustrezen element zunanega spomina (disketo, cd,...). Tako dobimo prvo verzijo programa, katerega uporabnost je potrebno še preveriti. Nadaljnja stopnja programiranja je testiranje oz. preverjanje pravilnosti in celovitosti rešitve določenega problema. Postopek testiranja navadno poteka tako, da v program vnašamo izmišljene podatke; seveda težavnost, zapletenost in različnosti podatkov stopnjujemo. Na ta način ugotavljamo, ali so vrednosti, ki jih dobimo kot rezultat procesa, pravilne in predvsem logične. Zato ni odveč, če vnašamo za preskus programa tudi nelogične podatke.

Testiranje programa je torej logično preverjanje pravilnosti programa, druge napake smo namreč že popravili v prejšnji stopnji. Dobro je, če program testira še kdo, ki ga sicer ni pripravil.

Na koncu je treba programsko dokumentacijo dodelati, opraviti poskusno dobo in poskrbeti za trajno hrambo originala programa. Program, ki je testiran, moramo še nekaj časa preskušati in ga s tem preverjati ob manjših in večjih količinah podatkov, pa tudi pri različnih uporabnikih.

Program nato shranimo na element zunanega spomina (disketa, cd,...) in to vsaj v dveh verzijah. Če ga imamo možnost prevajati v strojni jezik (hitrejši potek in uporaba programa), to tudi opravimo, original, pisan v izbranem jeziku (basic, pascal,...) pa hranimo, saj ga bomo uporabljali za poznejše spremembe, dopolnitve, razširitve itd...

Dober in skrben programer zbrano dokumentacijo (idejno skico, algoritem, diagram poteka posameznih podprogramov in glavnega programa) na koncu uredi, nato pa napiše še podrobna navodila za delo in uporabo programa, saj s tem zelo pomaga bodočim uporabnikom. Ni odveč, če pripravi tudi demonstracijski program s karakterističnimi pozitivnimi in negativnimi primeri uporabe.

Pomembnejše faze nadzorneje prikazuje fazni diagram:

