TARTU ÜLIKOOL
Matemaatika-informaatikateaduskond
Arvutiteaduse instituut

Ain Isotamm

PROGRAMMEERIMIS-KEELED

TARTU 2007

Matemaatika-informaatikateaduskond

Arvutiteaduse instituut

Ain Isotamm

PROGRAMMEERIMISKEELED

Tartu 2007

Käesoleva raamatu väljaandmist on toetanud Eesti Infotehnoloogia Sihtasutus projekti "Tiigriülikool" raames.

Toimetaja: Varmo Vene

ISBN 978–9949–11–553–2

Tartu Ülikooli Kirjastus
www.tyk.ee
Tellimus nr. 28

SAATEKS

„Programmeerimiskeeled" on temaatika, kus on kirjutatud sadu raamatuid (sh. õpikuid). Me ei üritagi neid ei ületada ega ka kokku võtta; pigem õigustab järjekordse teemakohase raamatu ilmumist kasvõi pisutki uus rakurss. Taotleme anda lugejale mingil moel korrastatud pilt programmeerimiskeelte ja nende realiseerimist võimaldanud masinate vastastikustest seostest. Masinad on konstrueeritud läbi aegade rahuldamaks arvuteid vajavate spetsialistide vajadusi. Ent samas on riistvara eelisareng (tarkvara arenguga võrreldes) pakkunud uusi võimalusi programmeerimiskeeltele (mõelgem kasvõi „pesamasinatelt" üleminekut „baitmasinatele", aparatuurset magasini või kiirete registrite evitamist).

Arvutite ainus ülesanne[1] on algusest peale olnud intellektuaalse töö viljakuse tõstmine. On ju nii, et masin ei ole võimeline „ise" lahendama ainsatki probleemi, millega inimene „käsitsi" hakkama ei saaks (olgu, internetipäringud tunduvad olevat selline valdkond, ent ainult tundub: ideaalse raamatukogu kartoteegi abil saaks lõpmatu aja jooksul leida ikkagi kõik vajalikud viited).

Pool sajandit tagasi programmeeriti masinkoodis (see on tänini *ainus* keel, mida on suutelised interpreteerima protsessorid). Programmeerimiskeeled disainiti ainsa eesmärgiga: lihtsustada programmeerimist ja mehhaniseerida rutiinseid töid. Esmasjoones peame siinjuures silmas masinast sõltuvaid keeli, aga mitte ainult: primitiivsete arvutite jaoks tuli kõik käskhaaval programmeerida, ent mida enam arenesid arvutid ja nende tarkvara, seda enam hakkas programmeerijat toetama *süsteemne tarkvara*, mis oli installeeritud arvutite välismällu ja mida oli võimalik kasutada rakendusprogrammides. Esialgu moodustus konkreetse arvuti süsteemne tarkvara varemprogrammeeritud ülesannete (süsteemsetest) alamprogrammidest, ja kui need olid vastavalt masina võimalustele vormistatud, siis olid nad *moodulitena* kasutatavad järgmistes projektides. Tehniliselt tähendas see käskude (so., protsessori) tasemel alamprogramm*ide* toetamist[2] Noist esialgu minimaalsetest tugivahenditest (näieks teisendused *sümbol*→ sisekuju, matemaatilised elementaarfunktsioonid, primitiivsete välisseadmete juhtimine) arenesid aja jooksul välja kaasaegsed (tänase hinnangu kohaselt võimsad) operatsioonisüsteemid.

Meie raamatu teine peatükk püüab lugejat sisse juhatada *masinorienteeritud* keskkonda: alustame mängu- ja õppearvutitest ning lõpetame tänapäevase *Intel*-tehnoloogia lihtsaima versiooniga. Tutvustame seal põgusalt nelja erinevat reaalset masinat; lugeja tähelepanu säilitamise eesmärgil on nood järjestatud ühelt poolt ajaliselt, aastaarvude järgi, ent samas operatsioonisüsteemide arengu järgi: *Razdan-3* oli praktiliselt „paljas masin", ent *IBM* (tolle op-süsteem oli omal ajal suurim realiseeritud tarkvarasüsteem üldse) või *Intel*ile baseeruvad *MS-DOS* või (eriti) *Windows* ei jää mahult *IBM*ile alla, sj. uute ülesannete tõttu ületavad nad oma eelkäijat nii funktsionaalsuses kui ka koodi mahus. Vaheetapina kasutame Valgevene „*Minsk-32*", see peaks demonstreerima evo- ja mitte revolutsiooni.

[1] Kui mitte arvestada tänapäevaste multimeedia rakendustega.

[2] Esimeses peatükis püüame sellele valdkonnale pöörata enim tähelepanu – ent vastavalt vajadusele ka hiljem: programmide struktureerimine on esmatähtis programmeerimise tehnoloogia objekt.

NL masinad on näideteks valitud mitte sellepärast, et nad olnuks paremad Lääne pesamasinatest; põhjused on proosalisemad esiteks, kaasaegsete Ameerika või Euroopa masinkoodide ja arvutiarhitektuuride kohta on vähe kättesaadavat materjali, ja teiseks − nende ridade autor on programmeerinud ligi aasta *Razdan*il ja üsna mitu aastat *Minsk-32* assembleris. Mainida tuleb ka seda, et kuni *IBM*i kopeerimiseni (üleminekuni *EC*-seeriale) olid NL masinad originaalsed ja täiesti konkurentsivõimelised. Ja et jutt juba autori kogemustele läks, siis tal on need lisaks mainitud masinatele *IBM OS/360/370* assembleri ja makroassembleri, *Malgol*i, *FORTH*i, *C* ja *C++*-ga, põgusamalt keeltega *Pascal* ja *Modula-2* ning *Intel*i assembleriga.

Kolmandas peatükis tutvustame põgusalt kaht kardinaalselt erinevat programmeerimiskeelt − *FORTRAN*i ja *ALGOL-60*t; esimene loodi mahukate teadusarvutuste võimalikult efektiivseks programmeerimiseks, teine aga nende arvutuste algoritmide matemaatiliselt elegantseks ja ühemõtteliseks publitseerimiseks. Kusjuures pole juhus, et mõlemi projekti üheks juhtfiguuriks oli *John Backus*. *FORTRAN* käis uute masinate ja nende võimalustega „ühte sammu" ning on oma valdkonnas tänini tunnustatud ja kasutatav keel. *ALGOL*i mõju on vägagi tuntav enamuse tänapäevaste programmeerimiskeelte süntaksi ja stiili puhul.

Arvutite ja programmeerimise arenguga kaasnes *süsteemse tarkvara* evolutsioon. Esialgu moodustus see lihtsate arvutitehases programmeeritud funktsioonide ja arvutuskeskustes programmeeritud moodulite teekidest, ent üsna ruttu arenesid esimestest üha võimekamad operatsioonisüsteemid ning üha rohkem loodi uusi programmeerimiskeeli, hõlbustamaks (alam)programmide kirjutamist. Esialgu programmeeriti süsteemne tarkvara masinorienteeritud keeltes, ent see protsess oli nii aeglane kui ka tööjõumahukas. Tarkvaraturul tekkis nõudmine kõrgtaseme-keelte järele, milles oleks võimalik efektiivselt programmeerida nii operatsioonisüsteeme kui ka translaatoreid. Soodsa võimaluse nende ilmumiseks andis miniarvuti *PDP-11* konstrueerimine (selle masina innovaatiline arhitektuur oli ilmselt suuresti eeskujuks mikroprotsessorite omale). Neljandas peatükis antakse lühiülevaade *PDP-11*-st ning kahest süsteemprogrammeerimise keelest − neiks on *FORTH* ja *C*.

Valdav enamus tänapäevastest programmeerimiskeeltest kuulub *protseduurorienteeritute* klassi, ja see tingib vajaduse omakorda nende keelte klassifitseerimiseks, seejuures võib lähtuda näiteks keele orientatsioonist mingile ülesannete klassile või programmeerimisstiilile (noid esindavad nn. *paradigmad*), samas on masinastsõltumatud keeled siiski tihedalt seotud arvutite arhitektuuriga (so., nad sõltuvad siiski). Neid teemasid püüame käsitleda peatükkides 4–6.

Ülalpool osutasime seikadele, mis põhjustasid süsteemprogrammeerimise keelte „tekke". Ent need keeled ei suutnud siiski likvideerida nn. *tarkvarakriisi*, mille üheks ilminguks on erinevate protsessorityyüpide ja programmeerimiskeelte paljus: lühidalt, kui meil on n protsessorit ja m keelt, siis me peame kirjutama $m \times n$ translaatorit. See on ilmselgelt kallis. Juba pool sajandit tagasi otsiti teid, kuidas seda ülesannet lahendada $m+n$ translaatori-

ga; pisut lihtsustatult nimetagem seda probleemi *tarkvara mobiilsuse* suureks projektiks. Seda teemat käsitleme 7. peatükis.
Viimases, 8. peatükis antakse lühiülevaade (programmeerimis)keelte teoreetilistest kontseptsioonidest: leksika, süntaks ja semantika. Püüame võimalikult lühidalt näidata, kuidas saadakse kõrgtaseme keeles kirjutatud programmist protsessori jaoks aktsepteeritav masinkoodi käskude jada ja kuidas saab aparatuurset interpretatsiooni vältida programse interpretatsiooniga. Programmeerimiskeeled on vahendid tarkvara loojate töö viljakuse tõstmiseks ning nonde keelte ülesanne on võimaldada (või soodustada) tarkvara akumulatsiooni ja mobiilsust, säilitades efektiivsuse (so, ressursisäästlikkuse). Selles peatükis on seetõttu põgusalt peatatud ka lihtsamatel koodi optimiseerimise võtetel.

Lõpuks tuleb mainida, et meie raamat on orienteeritud eeskätt Tartu Ülikooli informaatika ja infotehnoloogia õppekavadele. Nois kavades on omaette kursused programmeerimisest (praegu *Java*-keeles), objektorienteeritud programmeerimisest, operatsioonisüsteemidest, funktsionaalsest programmeerimisest ning loogilisest programmeerimisest. See seik muutis meie töö lihtsamaks: meie kontekstis tulnuks neid teemasid (möödaminnes või nende kaante vahel praegusest põhjalikumalt) kindlasti käsitleda, ent tänu kolleegide raamatutele ja kursuste materjalide kättesaadavusele võime piirduda kitsama temaatikaga. Ning programmeerimiskeelte käsitlus on sisuldasa lahutamatult seotud nende *realiseerimisega*, see on, transleerimisega. Meie raamatus on sel temaatikal vaid põgusalt peatatud, põhjuseks on siingi vastavate spetsiaalkursuste olemasolu: „Formaalsed keeled", „Automaadid, keeled ja translaatorid" ning „Transleerimismeetodid".

Ja ärgem unustagem, „programmeerimiskeel" pole mingilgi määral võrreldav objekt loomulike (so, elavate inimkeeltega) või loodusteaduste uurimisobjektidega – need viimased on objektid olemuslikult ja objektiivselt. Programmeerimiskeeled on täiesti tehislikena subjektiivsed ning tundub, et ainus kriteerium nende hindamiseks on pragmaatika: mil määral tõstab „antud keel" programmeerijate töö viljakust „antud valdkonna" ülesannete lahendamisel (püüdes säilitada efektiivsust ja turvalisust).

Nende ridade autor on siiralt tänulik kõigile, kes osutasid lahkelt abi meie raamatu koostamisel: need on kolleegid *Anne Villems, Mati Tombak*, *Ahti Peder*, *Mare Koit*, *Jüri Kiho* ja paljud teised, näiteks *IBM*i Eesti esinduse spetsialist *Andres Sirel*, ent eeskätt kõik meie loengukursuse aktiivsed kuulajad (näiteks *Siim Karus* või *Margus Niitsoo*). Ja muidugi, eritänu kuulub toimetaja *Varmo Vene*le.

Loomulikult ei tähenda tänu avaldamine vastutuse jagamise katset − kõikide võimalike sisuliste vaieldavuste ja leidmata trüki- jm. vigade autorlus kuulub jagamatult nende ridade kirjutajale (programmeerijaina teame, et kirjutades keeles, kus puuduvad silumisvahendid, on vead vältimatud).

SISUKORD

1. PROGRAMMEERIMISKEELED: SISSEJUHATUS

Programmeerimiskeeled on *tehiskeeled* programmide kirjutamiseks. Programme on lakooniliselt defineerinud *Niklaus Wirth*: **algoritmid+andmestruktuurid=programmid**. [62]

Tehiskeele mõiste on vanem kui programmeerimiskeele oma ─ loomulike keelte üldistusena on konstrueeritud palju rahvastevahelise suhtlemise ühiskeeleks mõeldud keeli, tuntuim neist on ehk *esperanto*.

Algsed programmeerimiskeeled (1950-ndate keskpaigani erinevate, unikaalsete arvutimudelite jaoks konstrueeritud masinkoodid ja mõnevõrra hiljem assemblerkeeled) olid absoluutselt tehislikud, olemata millegigi üldistused. Sama võib öelda esimeste kõrgtaseme keelte (eeskätt *FORTRAN*i ja *ALGOL*i, aga ka muude varajaste keelte, näiteks *COBOL, SNOBOL, Lisp* või *APL*) kohta. Need keeled, mida järgnevatel kümnenditel on lisandunud märkimisväärsel hulgal, on nii või teisiti võtnud eeskujuks protseduurorienteeritud „esmaskeeled".

Mõistagi, masinkoodid (ja neid toetavad arvuti-arhitektuurid) polnud absoluutne inseneride „vaba looming", vaid pidid võimaldama efektiivselt programmeerida ja lahendada aktuaalseid ülesandeid, algselt mahukaid arvutusmatemaatika ülesandeid (ballistika, koodi murdmine jmt.) ning tellimus määras tehnilise lahenduse: töödeldavate andmete formaadid ─ arvutüübid *real, integer* ja näiteks *FORTRAN*is ka *complex* ─ ning käskude repertuaari, aga ka *pesa-masina* kontseptsiooni.

Programmeerija jaoks pole midagi raskemat kui programmeerimine masinkoodis. Normaalne, et programmeerijad otsisid programseid vahendeid oma töö lihtsustamiseks[1], ja resultaadiks oli *assemblerkeele* leiutamine. See keel võimaldas 8-ndarvude jadade asemel programmeerida suhteliselt inimlikus keeles, ja jätta translaatori hooleks tülikaima osa ─ mälujaotuse; teiseks avanes võimalus teksti redigeerimiseks (käskude kustutamine, asendamine ja lisamine ilma aadresside käsitsi modifitseerimiseta).

Erinevaid programmeerimiskeelte klassifikatsioone on väga (*väga* ─ ebasoovitav sõna teaduskirjanduses, ent loodetavasti lubatav õppevahendis) palju[2], ent enamasti on nad kasutatavad *protseduurorienteeritud keelte* liigitamiseks, ja mitte kõikehõlmavaks liigiteluks. Nende ridade autorile sümpatiseerib klassifikatsioon, mille aluseks on *keele kaugus* masinkoodist, mis on protsessori poolt interpreteeritava koodi viimane „nähtav osa" (vt. [32], lk. 13 ─ 18), tinglikult 0-taseme keel.

---

[1] Nende otsingud olnuksid viljatud, kui poleks evitatud tähelis-numbrilisi sisend-väljundseadmeid ja kui masinkoodid poleks hakanud toetama sümboltöötlust. Ent oletetavasti tulid insenerid siin vastu programmeerijatele.

[2] Toome ainult ühe alternatiivse klassifikaatori näite, tuginedes *Herbert Schildt*i raamatule [52, lk. 16 jj], kus liigitatakse programmeerimiskeeled kolme klassi: *madala* taseme keeled on assemblerid, *keskmise taseme* omad makroassemblerid, *FORTH*, *C* ja *C++* ning kõrgtaseme keeled on *BASIC*, *FORTRAN*, *COBOL*, *Pascal*, *Modula-2* ja *Ada*.

Järgnev tabel (1a) püüab sisse juhatada programmeerimiskeelte klassifikatsiooni puutuva.

tase	**Klass**	**orientatsioon**
-1	Mikrokood: masinkoodi interpretaatori keel	Masinast sõltuv
0	Masinkood: programmeerija jaoks madalaim tase	Masinast sõltuv
1	Assembler: ≈ 1:1- suhe masinkoodiga, sama võimsus	Masinast sõltuv
2	Makroassembler: assembleri pealisehitus	Masinast sõltuv
3	Protseduurorienteeritud keeled (*3GL*)	Masinast sõltumatud
4	Probleemorienteeritud keeled (*4GL*)	Masinast sõltumatud

.
Tabel 1a. Programmeerimiskeelte klassifikatsioon.

Niisiis, 3 ja 4. taseme keeled on masinast sõltumatud, mis tähendab, et neid saab realiseerida erinevate protsessorite (ja masinkoodidega) arvutitel. Keele *realiseerimise* all mõistame objektarvuti jaoks kättesaadavas keeles translaatori kirjutamist; objektarvuti on see masin, mille jaoks keel realiseeritakse.

Valdav enamus programmeerimiskeeli kuulub **protseduurorienteeritud** keelte klassi (vahel nimetatakse neid ka 3. põlvkonna keelteks, ingliskeelne lühend on *3GL*). Protseduurorienteeritus tähendab, et programmis tuleb defineerida kasutatavad andmestruktuurid ning esitada ülesande täpne lahendusalgoritm (instruktsioonide jada). Sellest aspektist on täpselt samuti orienteeritud ka masinast sõltuvad keeled. Üldjuhul töötab selle klassi keelte translaator kompilaatorina (genereerib paigaldatava masinkoodifaili), harvem genereerib interpretaatori, mis „jooksutab" antud keeles kirjutatud programme. Ja need keeled on universaalsed selles mõttes, et enamik „üldotstarbelistest" algoritmidest on programmeeritavad igas protseduurorienteeritud keeles.

Ja veel, kui teil on mingi (kasvõi tuntud) ülesande lahendamiseks *oma* algoritm, siis protseduurorienteeritud keeled võimaldavad (ja ei takista) selle programmeerimist.

Esimene selle klassi keel oli *FORTRAN* (*FORmula TRANslator*), mille loomisega tegeles *John Backuse* juhitud töörühm alates 1954. aastast. Teade tööst publitseeriti 1956. aastal ja esimene realisatsioon pärineb 1957. aastast. Me käsitleme seda keelt tagapool detailsemalt, siinkohal nentigem, et üsna paljude tunnuste poolest oli see keel lähedasem makroassemblerile kui *ALGOL*ist alanud standardile. Keelt iseloomustasid järgmised seigad:

- Orienteeritus arvutusmatemaatika ülesannete lahendamisele.
- Direktiivide formaat järgis pigem (makro)assembleri stiili, sj. orienteeritust perfokaardi-formaadile (st. poolfikseeritud formaadile).
- Assemblerile omaselt toetas *FORTRAN* moodulprogrammeerimist masinorienteeritud keelte stiilis: moodulid olid eraldi transleeritavad iseseisvad programmid, mis ühendati alles komplekteerimise käigus. Komponentmoodulite võimalikke vigu translaator ei saanud *põhimõtteliselt* avastada.


John W. Backus (3.12.24), USA, *FORTRAN* (1954─1957) ja *BNF* (1959).

Protseduurorienteeritud keelte arengut on enim mõjutanud *ALGOL*i (*=ALGOrithmic Language*)[1], mille esmasversioonist teatati 1958. aastal (*ALGOL-58*) ja klassikaliseks kujunenud versioonist 1960. aastal (*ALGOL-60*). Teate autoriteks olid *John W. Backus*, *F. L. Bauer*, *P. Naur*, *A. van Wijngaarden* jt. Keel polnud mõeldud *programmeerimiskeeleks,* vaid *algoritmiliseks* ─ algoritmide publitseerimise keeleks, ent siiski realiseeriti ta hoolimata standardis puuduvast sisend- väljundplokist väga ruttu. Ja et *realiseerimis*standard puudus, siis realiseerijatel olid vabad käed nii keele „kohendamiseks" kui ka oma versiooni nimetamiseks. N. Liidu tarkvaraloomingu hulka kuuluvad meile olulised *ALGOL*i translaatorid:

- *Malgol*[2] arvutile *Minsk-22*, mida kasutati (vist) kõigis sots-maades, autor *Malle Kotli*.


Malle Kotli

- *Velgol* Minsk-tüüpi arvutitele, orienteeritud mahukale andmetöötlusele, autor *Vello Kuusik.*
- *АЛГАМС,* objektmasin Minsk-32, tehase tellimusel tehtud TPI Arvutuskeskuses (projektijuhid *Tõnu Lume* ja *Leo Prisk*, juhendaja *Leo Võhandu*).

ALGOL-60 oli mitmeti teedrajav keel. Nii oli ta esimene, mille *süntaksit* kirjeldati just selleks otstarbeks loodud metakeele vahenditega, selleks oli *BNF* (eri allikates dešifreeritakse seda akronüümi kui *Backus-Naur Form* või *Backus Normal Form*), nondega tutvume hiljem. Formaliseeritud süntaks andis võimaluse järgmiste keelte *süntaksorienteeritud* transleerimiseks ja *eelnevusmeetodi* väljatöötamiseks (*Niklaus Wirth*). *ALGOL-60* realiseerimisel kasutati programmiteksti kompileerimiseelse vormina *pööratud Poola kuju*, ja algoritmi selle kuju saavutamiseks töötas välja *Edsger W. Dijkstra.* Me käsitleme noid asju tagapool.

[1] Eesti keeles on pädev tutvustus publitseeritud TRÜ kirjastuses [25], autorid *Ülo Kaasik* ja *Ivar Kull.*
[2] ametlikult: *Malgol=Minsk-Algol* või *Moderniseeritid Algol*, ent kaasaegsed dešifreerisid nime kui *Malle Algol* – mis võib ka õige olla, eriti, kui püüda *Vello Kuusiku* keele *Velgol* nimele šifrit leida.

Edsger Wyte Dijkstra, Holland (Rotterdam), 1930 ─ 2002 [66].

*ALGOL*i roll on mõneti sarnane ladina keele omale. Tänaseks on nad mõlemad surnud keeled, ent *ALGOL*i mõju programmeerimiskeeltele ja ladina keele mõju Euroopa keeltele (kui mõtleme kasvõi mitte niivõrd romaani keeltele kui inglise keelele) on vägagi tuntav. *FORTRAN* on selles kontekstis midagi eesti või keldi keel(t)e taolist ─ elavad eirates objektiivseid prognoose.

Niklaus Wirth (veebruar 1934), Šveits.

Probleemorienteeritud keeled *pole* universaalsed, nad võimaldavad lahendada ainult oma spetsiifilise klassi ülesandeid. Seejuures täpse lahendusalgoritmi esitamise võimalused on kas piiratud või puuduvad sootuks. Selle klassi keeled realiseeritakse üldjuhul interpretaatorite abil. Toome paar näidet. Enne baitmasinate ilmumist (ja sellega seoses andmebaasisüsteemide ilmumist) olid populaarsed *aruannete generaatorid* (*RPG= Report Program Generator* (vt. näit. [87])), tänapäeval on lisandunud infosüsteemide *päringukeeled* (näiteks, *SQL = Structured Query Language*); aruannete generaatorid (nüüd lihtsalt *RG*) on teisenenud, ent mitte kadunud.

Ja veel: kasutajal puuduvad võimalused „sisseprogrammeeritud" lahenduskäigu muutmiseks.

2. MASINORIENTEERITUD KEELED

2.1. Mänguarvuti

2.1.1. Arhitektuur

Arvuti kui masin koosneb põhimõtteliselt neljast osast: need on juhtimisseade, mäluseade, aritmeetika-loogikaseade ja sisend-väljundseade. **Juhtimisseade** loeb mälust ükshaaval käske ja täidab neid; kui vaja, siis annab ta ülesandeid ülejäänud seadmetele. **Mälu** on seade, kus paikneb käskudest koosnev programm ja töödeldavad andmed. **Aritmeetika-loogikaseade** sooritab aritmeetilisi ja loogilisi tehteid. **Sisend-väljundseade** on ette nähtud andmete toomiseks mällu ja väljastamiseks mälust.

Juhtimisseade ja aritmeetika-loogikaseade üheskoos moodustavad **tsentraalprotsessori** (ehk lihtsalt protsessori, kui tegemist on primitiivse arvutiga). Mingi töö tegemiseks arvuti mällu viidud käskude jada on **programm**. Selle täitmiseks tuleb protsessorile teatada, kus on mälus programmi esimene käsk; arvuti eeldab, et käsud paiknevad mälus üksteise järel ja ta püüab neid selles järjekorras ka täita.

Illustreerimaks arvuti töötamise printsiipe konstrueerime mänguarvuti. Idee on laenatud *Leo Võhandult,* kes avaldas 1969. aasta „Horisondi" kuues esimeses numbris artiklisarja pealkirjaga „Õpime programmeerima!", mille sissejuhatuses öeldi arvutihirmu peletamiseks, et arvuti tööd võib vabalt imiteerida kasvõi arvelaua abil. Sellest inspireerituina tegi nende ridade autor koos *Anne Villemsi* ja mõne tudengiga üleliidulises noorte teadlaste ja spetsialistide koolis Narva-Jõesuus, mille teemaks oli „Arvutid õppeprotsessis", 1. aprillil 1987 ettekande „Урок программирования на машинном уровне в безмашинном варианте обучения[1]". Nimetagem seal demonstreeritud installatsiooni mänguarvutiks; ainus erinevus allpool esitatavaga on arvelaudade arvus: Narva-Jõesuus oli meil neid kaasas 30 tükki (saadud majandusteaduskonna kolikambrist). Õppearvuti on järgmine:

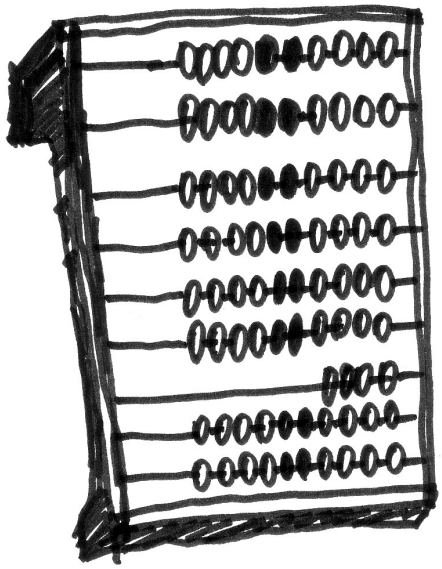
- Mälu koosneb sajast arvelauast: Oletagem, et nad on meil väiksemat sorti, 9 traadiga, nagu näha järgmisel leheküljel paikneval joonisel 2.1.1a. Arvelauad paneme laudadele ritta ning kirjutame igale järjekorranumbri (0 kuni 99), mida edaspidi nimetame **aadressiks.**

Ülejäänud seadmeteks peavad hakkama neli inimest; kolmel neist on veel oma isiklik arvelaud, mis ei kuulu mälu hulka.

- Juhtimisseadme arvelaua nimi on **käsuloendaja** ja seadmel on kolm alluvat.
- Esimene neist on aritmeetika-loogikaseade, kellel on arvelaud liitmiseks ja lahutamiseks ning paber ja pliiats (või midagi muud) korrutamiseks ja jagamiseks. Tehte tulemuse lööb ta oma arvelauale.
- Teine alluv täidab sisendseadme ülesandeid; temalgi on oma arvelaud nimega **puhver,** millele ta lööb etteöeldud arve.

[1] „Masintasemel programmeerimistund ilma arvutita õpetamisvariandi puhul"

- Kolmas abimees on väljundseadme rollis; talle anname kriidi, millega saab − kui kästakse − tahvlile arve kirjutada.



Joonis 2.1.1a Arvelaud.

2.1.2. Käskude süsteem

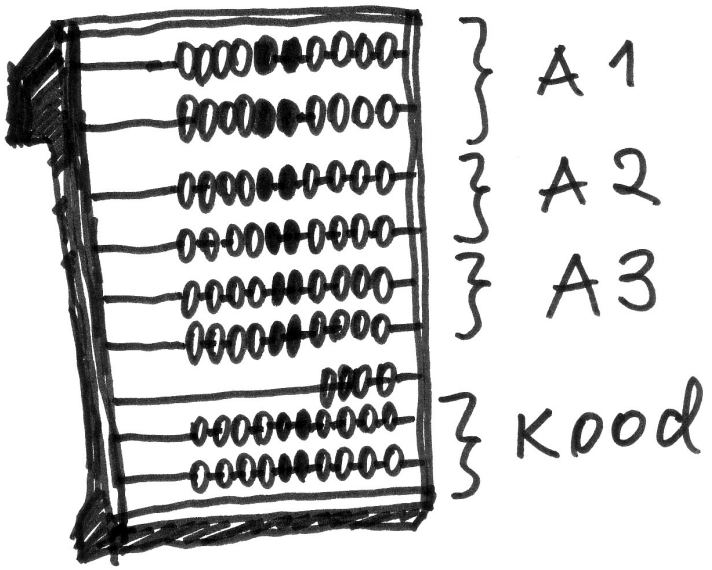
Jäänud on veel kokku leppida käskude esitusviisis. Iga käsk peab ära mahtuma ühele 9-traadisele arvelauale; rohkem kitsendusi meil esialgu ei ole. Mõtelgem, kuidas seda ruumi oleks hea ära kasutada. Kõigepealt, käsus tuleb ruumi jätta käsu **koodile,** mis näitab, milline tehe tuleb (üldjuhul operandidega) sooritada. Edasi tuleb käsus näidata, kus paiknevad operandid ja kuhu tuleb panna tehte tulemus. Seega peame käsu kirjutamiseks jagama arvelaua neljaks osaks. Igale neist jääb kaks traati. Nelja nupuga traadi jätame esialgu tagavaraks. Sellist jaotust nimetame **käsu formaadiks** (vt joonis 2.1.2a).

Operandide ja resultaadi näitamiseks kasutame nende aadresse, iga jaoks on meil 2 traati, kuhu saame kanda arve vahemikust 0..99, ja et meie arvuti mälu koosneb parajasti 100 arvelauast, siis saame igas käsus adresseerida suvalist neist. Edaspidi nimetame arvelauda mälupesaks ehk lihtsalt **pesaks.**

Käsukoodi jaoks võtame kaks viimast traati, seega võime kokku leppida kuni sajas erinevas käsus, mida meie arvuti võiks teha. Alustagem järgmistest käskudest:

Käsu kood	Aadressid	Toime
00	Ei kasuta	Stopp
10	(A1)+(A2)→A3	Liida
11	(A1)-(A2)→A3	Lahuta
12	(A1)x(A2)→A3	Korruta
13	(A1)/(A2)→A3	Jaga
80	A3	Loe sisendist arv → A3
90	A3	Arv (A3) → väljundisse

Tabel 2.1.2a. Esimesed käsud.



Joonis 2.1.2a Käsu formaat.

Mainigem, et nt. A2 tähistab aadressi, aga (A2) sellel aadressil olevat arvu.

Meie mänguarvuti suudab tehteid teha vaid kümnendsüsteemi reaalarvudega, milledel on maksimaalselt 6 kohta enne ja 2 pärast koma; täisosa paikneb 6 esimesel ja murdosa 2 viimasel traadil. Visuaalseks „komakohaks" on nelja nupuga traat. Anname talle ka sisulise funktsiooni: kui selle traadi esimene nupp on lükatud vasakule, on pesas olev arv negatiivne.

2.1.3. Esimene programm

Asetame mällu programmi, mis sooritab omistamistehte *Y:=A+B*. Programm peab tegema järgmised tööd:

1. Lugema mällu muutuja *A* väärtuse.
2. Lugema mällu muutuja *B* väärtuse.

3. Leidma summa *A*+*B*.
4. Omistama summa *Y*-le.
5. Väljastama *Y* väärtuse.
6. Andma lõpusignaali.

Käsud peame mällu paigutama suvalisest aadressist alates järjestikustele arvelaudadele; andmed võivad paikneda juhuslikel aadressidel. Reserveerime neile järgmised pesad:

1. *A* ─ aadress 10.
2. *B* ─ aadress 19.
3. *Y* ─ aadress 40.

Programmi alustame 3. arvelaualt. Loobudes arvelaudade joonistamisest näeb meie „abstraktne" programm välja selline:

Aadress	**A1**	**A2**	**A3**	**Kood**	**Kommentaar**
3	00	00	10	80	Loe *A*
4	00	00	19	80	Loe *B*
5	10	19	40	10	*Y:=A+B*
6	00	00	40	90	Kirjuta *Y* tahvlile
7	00	00	00	00	Stopp

Tabel 2.1.3a. Esimene programm.

Juhime tähelepanu tõigale, et käsud (arvelaudadel, so reaalses arvutis) ei sisalda ei endi aadresse ega ka kommentaare! Vaadelgem sammhaaval, kuidas meie arvuti seda programmi täidab.

1. Teatame juhtimisseadmele, et programm algab aadressilt 3 ja palume tal seda täitma hakata. Juhtimisseade lööb käsuloendajale arvu 3 ja astub kolmanda laua juurde.
2. Kolmandal laual on lugemiskäsk. Juhtimisseade ütleb meile, et lugemisseade ootab arvu. Ütleme talle (näiteks) „seitse". Lugemisseade kannab tolle oma arvelauale (puhvrisse) ja kannab juhtimisseadmele ette, et viimase korraldus on täidetud. Juhtimisseade loeb puhvrist arvu 7 ja kannab selle aadressile 10. Käsk on täidetud. Juhtimisseade lööb käsuloendaja seisule 1 juurde ja astub 4. laua juurde.
3. Sealgi on lugemiskäsk ning selle täitmine käib samuti nagu eelmisel sammul. Oletagem, et nii kanti aadressile 19 arv 319, ja käsuloendaja seisuks saab 5.
4. Viiendal laual on liitmiskäsk. Juhtimisseade annab käsu aritmeetika-loogikaseadmele: liida arvud arvelaudadelt aadressidega 10 ja 19! Käsutatav kasutab oma arvelauda ja peale liitmist on seal arv 326. Juhtimisseade ootab ära raporti tehte sooritamise kohta ning kirjutab tulemuse (326) aadressile 40. Käsuloendaja seisuks saab 6.
5. Kuuendal laual on kirjutamiskäsk. Juhtimisseade loeb aadressilt 40 arvu 326 ning käsib väljundseadmel see tahvlile kirjutada. Ära oodanud raporti käsu täitmise kohta suurendab juhtimisseade käsuloendajat, uueks seisuks saab 7.

6. Seitsmendal laual on stopp-käsk: juhtimisseade teatab meile (kasutajale), et programm on täidetud.

Ja veel, pesade sisusid võime interpreteerida ka arvkonstantidena, näiteks pesas aadressiga 4 on arv 19,80.

2.1.4. Võrdlemine ja suunamine

Järgmise programmi teeme pisut keerulisema: laseme tal lugeda arve ja leida lõpuks nende väärtuste aritmeetilise keskmise. Arve loeme ükshaaval kuni lõputunnuseni, milleks on arv väärtusega 0. Sammhaaval tuleb teha järgmised tööd:

S1: *S:*=0*; I:*=0*; K:*=0;
S2: Loe arv *A*;
Kas *A=0*? Kui **jah**, siis mine S3;
S:=S+A; *I:=I*+1;
Mine S2;
S3: Kas *I*=0? Kui **jah**, siis mine S4;
K:=S/I;
S4: Väljasta *K*; Stopp.

Algoritmiga tutvumine näitab, et meie arvuti pole võimeline seda interpreteerima, kuivõrd puuduvad vahendid arvude võrdlemiseks ja juhtimise üleandmiseks. Viimane tähendab käskude täitmise loomuliku järjekorra rikkumist: juhtimisseade peab käsuloendajasse kandma käsust saadud aadressi.

Ilmselt peame täiustama käskude süsteemi. Võrdlemiskäsu formaat võiks olla järgmine:

Kood=20, A1 on esimese ja A2 teise operandi aadress.

Kolmandat aadressi see käsk ei kasuta. Käsu täitmisel sooritab protsessor (mäletatavasti on see juhtimisseadme ja aritmeetika-loogikaseadme ühine nimi) tehte

$$R:=(A1)-(A2)$$

R väärtust kasutame suunamiskäsus (so juhtimise üleandmise käsus), seega tuleb see kuskil meeles pidada. Kohaks sobib juhtimisseadme käsuloendaja-arvelaud, kus seni oleme kasutanud ainult kaht alumist traati. Resultaadi *R* jaoks võtame kõige ülemise traadi, kuhu protsessor paigutab järgmised signaalid (nimetagem signaali „pärisarvuti" eeskujul *CC = condition code* ─ tingimuskood):

CC=0, kui *R*=0;
CC=1, kui *R*>0;
CC=2, kui *R*<0.

Pisut ette rutates lepime kokku, et signaali *CC* töötavad välja kõik käsud peale suunamiskäskude; võrdlemiskäsk teeb seda ilmutatud kujul, teised varjatult.

Valime suunamiskäsu koodiks 60. Käsk kasutab signaali *CC* ja annab juhtimise üle käsule aadressiga *A[CC+1]*, so kui *CC*=0, siis mine *A*1, kui *CC*=1, siis *A*2, ja kui *CC*=2, siis A3.

Vaadelgem veel algoritmi. Plokis S3 on kontroll: kas *I*=0? Nii püüame end kaitsta kasutaja eest, kes annab esimese arvuna lõputunnuse 0. Nulliga jagamist tuleb iga hinna eest vältida, kuivõrd selle tehte resultaat pole määratud; tavaliselt annab aritmeetikaseade tulemuseks arvu, mis ei mahu mistahes pessa. Selliseid „mittemahtuvaid" arve võime saada ka „legaalsete" operandide puhul, näiteks tehe 999900+231,12=1000131,12 ─ mis ka ei mahu arvelauale. Sellist situatsiooni nimetatakse **ületäitumiseks** (*overflow*, переполнение) ning protsessor pole võimeline programmi edasi täitma ilma kõrvalise abita, aga selle saamiseks tuleb signaliseerida. Reserveerime ületäitumise signaali jaoks käsuloendaja-arvelaua ülalt teise traadi: normaalselt on seal arv 0, ületäitumise korral aga 1. Nimetame seda signaali "OF".

Asume programmi arvelaudadele panema. Paigutame 4 muutujat mälu algusse:

0) *A*;
1) *S*;
2) *I*;
3) *K*.

Konstandi 1 paneme aadressile 4, konstandi 0 rollis kasutame stopp-käsku. Algoritmi sammus S1 on kolm omistamiskäsku: väärtus 0 tuleb kirjutada aadressidele 1, 2 ja 3 (vastavalt *S. I* ja K). Kasutame siin väikest kavalust: teeme tehted *S:=S-S,I:=I-I* ning *K:=K-K*. Mõistagi võinuks me arvelaudadele aadressidega 1, 2 ja 3 lüüa nullid[1]. Programm arvelaudadel näeb tinglikult välja nii:

Aadress	**A1**	**A2**	**A3**	**Kood**	**Kommentaar**
4	00	00	01	00	Konstant 1
5	03	03	03	11	S1: sisendpunkt; *K:*=0
6	01	01	01	11	*S*:=0
7	02	02	02	11	*I*:=0
8	00	00	00	80	S2: Loe *A*; signaal *CC*
9	13	10	10	60	Kui *A*=0, mine 13
10	01	00	01	10	*S:=S+A*
11	02	04	02	10	*I:=I*+1
12	08	08	08	60	Tingimusteta suunamine
13	02	17	00	20	S3: Kas *I*=0?
14	16	15	15	60	Kui on, mine 16
15	01	02	03	13	*K:=S/I*
16	00	00	03	90	S4: Väljasta *K*
17	00	00	00	00	Stopp

Tabel 2.1.4a. Teine programm.

[1] Nüüdne variant on eelistatavam, kuivõrd nii saame **taaskasutatava** (*reenterable* või *reusable*) programmi, mis ei vaja kahe seansi vahelist häälestamist. Tagapool kasutame ka terminit "re-enteraabel".

Protsessorile teatame programmi algusaadressiks mõistagi 5

Loodetavasti õnnestus meil näidata, et arvutit võib imiteerida .tõepoolest arvelaudade abil. Muidugi, tegime igati primitiivse masina, mis pealegi opereerib ainult ühte tüüpi arvandmetega, ent tõelise elektronarvuti erinevused meie mängumasinaga võrreldes on pigem kvantitatiivset kui kvalitatiivset laadi. Muide, mängu mänguarvutiga võiks jätkata. Näiteks nii, et väljundseade töötaks autonoomses režiimis, so juhtimisseade ei pruugi välisseadme raportit ootama jääda; ootama peaks ta ainult siis, kui uue väljastamiskäsu täitmise ajaks pole eelmine veel lõppenud. Kõiki seadmeid peale mälu võiks olla rohkem kui üks: juhtimisseadmed koordineeriksid tööd omavahel ise (ja ka programmist sõltuvalt) ning jaotaksid ülesandeid muude seadmete vahel jne.

2.2. Arvuti ja kahendsüsteem

Tänapäeva arvutite teoreetilised alused töötas välja aastatel 1945 kuni 1947 *John v. Neumann*), USAs. Pisut lihtsustatult on tema printsiibid järgmised:

John von Neumann sündis Ungaris 28.12.1903, suri 8.02.1957 USAs.

- Nii programm kui ka andmed paiknevad ühtses operatiivmälus.
- Andmed (programm ja „pärisandmed") kujutatakse kahendsüsteemi arvudena.
- Suvalised mälupesad võivad olla tehete operandideks, sh. ka käske sisaldavad. Mis pesas on ─ andmed või käsk ─ on interpretatsiooni asi.

Niisiis, **andmete** all mõistame nii programmi kui ka viimase poolt töödeldavaid objekte (arve, teksti, heli, pilti jne.). Selle nõude järgimine dikteerib kogu arvuti kui masina konstruktiivse lahenduse.

Kõik see, mida masin „mõistab", koosneb kahendarvude jadadest: suvaline „aatom" evib väärtust 0 või 1 (sellise ühiku nimi on „bitt"). Varasemates arvutites agregeeriti neid **pesadeks** nagu meie mänguarvuti puhul ─ kogu mälu koosneb pesadest, millel on interpreteeritav struktuur, ent pesa[1] on minimaalne adresseeritav mäluühik. Kõik pesad on ühepikkused, so koosnevad samast arvust bittidest.

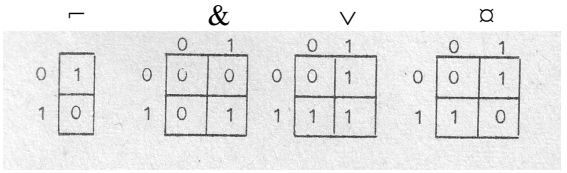
[1] Mõistega *pesa* kasutatakse mõnikord paralleelselt mõistet *sõna.* Võiksime soovi korral neid mõisteid eristada nii: pesa on mäluväli, sõna aga tollel mäluväljal kujutatud andmeühik (käsk, ujupunktarv vmt.). Ehk, kasutades üldtunnustatud metakeelt, *sõna=(pesa). Sõna* on i.k. *word,* vene k. *машинное слово*. *Pesa* on vastavalt *cell* ja *ячейка*.

Inimlikust seisukohast on kahendkoodi raske kirjutada ja lugeda, sestap kasutatakse masinkoodi esitamiseks kaheksand- või kuueteistkümnendarve. Näiteks, arv 14 on

- Kahendsüsteemis 001110 (tähistatakse 001110_2 või 001110*b*[*inary*]);
- Kaheksandsüsteemis 16 (16_8 või 16o[*ctal*]);
- Kuueteistkümnendsüsteemis E (E_{16} või E*h*[*exadecimal*]).

Aritmeetika-loogikaseade ehitatakse nii, et ta suudaks realiseerida tehteid kahendarvude vahel. Ühepikkuste täisarvuliste operandidega aritmeetikatehete realiseerimine on üsna sarnane sellele, kuidas me neid paberi ja pliiatsi abil teeme, ehkki mitte päriselt: mõnevõrra erinevalt toimuvad ülekanded, negatiivseid täisarve kujutatakse tavaliselt täiendkoodis (näit. arvu 011010 täiendkood on 100101) jmt.

Loogikatehete realiseerimisel on kahendsüsteem eriti soodne, kuivõrd see annab võimaluse kasutada Boole'i algebrat (*George Boole,* 1815 ─ 1864). Boole'i muutujad evivad väärtusi 0 ja 1 ning defineeritud on üks unaarne tehe ─ loogiline eitus (negatsioon ¬) ning komplekt binaarseid tehteid, nt. loogiline korrutamine (konjunktsioon &), loogiline liitmine (disjunktsioon ∨), loogiline mittesamaväärtustamine (tähistame seda näit. ¤) jt. Need tehted on defineeritud järgmiselt:



Loogikatehted kahendarvude vahel toimuvad bitikaupa. Näiteks

111 ∨ 010 = 111
111 & 010 = 010
111 ¤ 010 = 101

2.3. Õppearvutid

2.3.1. Kolmeaadressiline arvuti

Eelmises peatükis konstrueeritud mänguarvuti on **kolmeaadressiline,** kuivõrd käsus on ilmutatud kujul kohad kolmele mäluaadressile. Mainigem, et see masin on loomulik. Aritmeetikatehetes osalevad kaks operandi ja resultaat, suunamiseks tingimuskoodi järgi on kolm võimalust. Allpool püüame sama arvuti realiseerida elektronarvutina, järgides juba kõiki *v. Neumanni* printsiipe.

Kõigepealt tuleb meil kokku leppida arvuti mälu mahus (mälupesade arvus). Arusaadavatel põhjustel on hea, kui pesade arv on „ümmargune". Mänguarvutil oli 100 pesa aadressidega 0..99 ning suvalise aadressi kujutamiseks piisas kahest traadist. Olnuks meil mälus 101 pesa, tulnuks lisada kolm traati, mis olnuks ilmselt ebaratsionaalne lahendus. Kuivõrd meie õppe(elektron)arvuti töötab kahendsüsteemis, peame mälu mahuks võtma „ümmarguse" kahendarvu, milleks on kõik arvu 2 astmed. Teeme oma arvuti 64-pesalise-

na: see on 2^6, ja seega suvalise aadressi väljendamiseks on vaja 6 bitti. Võimalik aadressruum on vahemik $0..77_8$.

Käsukoodi jaoks oli meil arvelaual sama palju ruumi nagu iga aadressi jaoks, talitame siin samuti. Et käsukood tuleb nüüd esitada kahendarvuna (*resp.* kaheksandarvuna), siis peame koodid 80 ja 90 asendama, näiteks koodidega 50 ja 70. Niisiis, käsu kujutamiseks on vaja kokku 24 bitti ─ see saabki meie arvuti pesa pikkuseks. Kokku koosneb mäluseade järelikult 1536 bitist (64 pesa á 24 bitti). Käsu formaadi võime valida näiteks sellise:

Kood	A1	A2	A3

Käsud ise (järgigem eelmise arvuti käskude süsteemi!) on toodud tabelis 2. Märk ¤ tähistab võrdlustehet. Lihtsuse mõttes võimaldame opereerida vaid märgiga täisarvudega. Pesa vasakpoolseim bitt on arvu märk: 0 positiivse ja 1 negatiivse arvu puhul. Seega saame kujutada arve, mille absoluutväärtus ei ületa 2^{23}-1 (so. kaheksandarvu 37777777). Jagamistehte tulemuseks on jagatise täisosa.

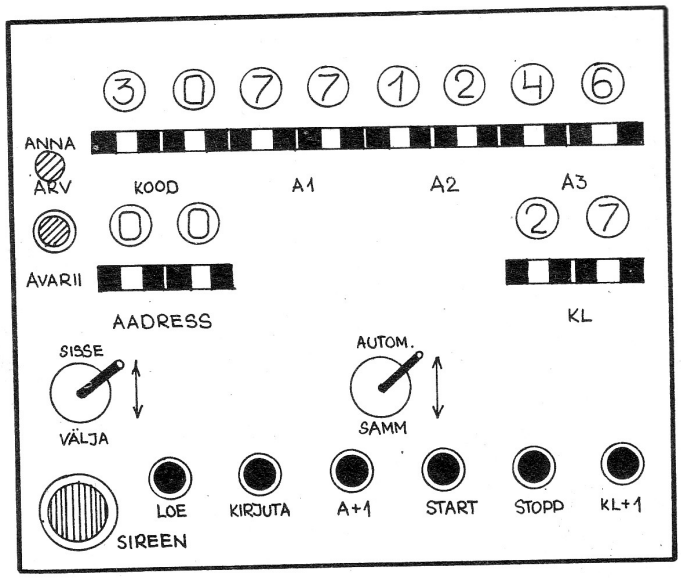
Kood	**Aadressid**	**Semantika**	**Signaalid**
00	Ei kasuta	Stopp	pole
10	A1, A2, A3	(A1)+(A2)→ A3	CC, OF
11	A1, A2, A3	(A1)-(A2)→ A3	CC, OF
12	A1, A2, A3	(A1)x(A2)→ A3	CC, OF
13	A1, A2, A3	(A1)/(A2)→ A3	CC, OF
20	A1, A2	(A1)¤(A2)	CC
50	A3	Sisend→A3	CC
60	A1, A2, A3	Mine A1, A2 või A3	Ei muuda
70	A3	(A3)→väljund	CC

Tabel 2.3.1a. Kolmeaadressilise masina käsud.

Arvutiga suhtlemiseks kasutame nn. juhtimispulti (vt. joonis 2.3.1a). Puldil on kolm sissevajutatavate klahvidega klaviatuuri, igal neist saab valida kolmekaupa grupeeritud kahendarve. (000..111). Käskude ja arvude sisestamiseks on ülemine, 24-nupune klaviatuur. Pesa aadress, kuhu kirjutada arv, tuleb määrata klaviatuuril „aadress", vahetult järgmise aadressi saamiseks võib kasutada nuppu „A+1" (alumises reas). Arvu (või käsu) kirjutab mällu nupule „kirjuta" vajutamine. Programmi sisendpunkt valitakse klaviatuurilt „KL" (käsuloendaja) ja programm käivitatakse nupu „Start" abil.

Kõigi kolme klaviatuuri kohal on valgustabloo aknad, kus näidatakse klaviatuuridega määratud arve kaheksandsüsteemis. Programmi töö võib katkestada nupuga „Stopp" (et kontrollida näiteks mingis pesas leiduvat infot tuleb seejärel valida aadressiklaviatuurilt vajalik aadress ning vajutada nupule „Loe": pesa sisu näidatakse pikal tablool). Jätkata saab nupu „Start" abil (käsuloendaja seisuks on katkestusmomendil aktiivse käsu aadress).

Sisestamiskäsk paneb vilkuma lambi „Anna arv”. Arvuti ootab, kuni arv on valitud ja sisestatud (nupu „kirjuta” abil).



Joonis 2.3.1a. Juhtimispult.

Ületäitumise korral süttib lamp „avarii” ja hakkab undama sireen.Puldil on veel kaks tumblerit. „Sisse-välja” lülitab arvuti vooluvõrku või võrgust välja; teisega saab määrata töörežiimi. Automaatrežiimil täidab arvuti käske ilma katkestusteta, sammrežiimil katkestatakse peale iga käsu täitmist töö (võimaldamaks nt. mälus olevaid andmeid kontrollida vms.) ning järgmise käsu täitmiseks tuleb vajutada nupule „KL+1”. See režiim on mõeldud silumiseks.

Vaatame nüüd, kuidas näevad välja mänguarvuti programmid uuel arvutil. Esimene programm sisestas arvud *A* ja *B*, omistas nende summa muutujale *Y* ja trükkis selle välja. Kaheksandsüsteemis näeb programm välja nii (esimene käsk on aadressil 0 ning muutujate aadressid on vastavalt 10_8, 11_8 ja 12_8):

0) 50000010 1) 50000011 2) 10101112 3) 70000012 4) 00000000

Pilt vastab peaaegu reaalsele mäluseisule, seal pole ainult ilmutatud kujul käskude aadresse. Aga „päris tegelik” programmiga hõivatud mälu seis on (alates pesast aadressiga 0) selline:

10100000000000000001000
10100000000000000001001
00100000100000100100101 0
11100000000000000001010
00000000000000000000000

Joonis 2.3.1b (pärit [74]). *Minsk-22* juhtimispult, mida matkis meie eelmine joonis.

Aadr.	Kood	A1	A2	A3	Kommentaar
0	50	0	0	10	Loe A
1	50	0	0	11	Loe B
2	10	10	11	12	Y:=A+B
3	70	0	0	12	Trüki Y
4	0	0	0	0	Stopp

Tabel 2.3.1b. Esimene programm.

Programmi sisestamiseks arvutisse valime aadressi 00, paneme ülemisele klaviatuurile käsu 50000010 ja vajutame „kirjuta", seejärel vajutame „A+1", sisestame teise käsu ja nii edasi, kuni stopp-käsk on sisse viidud. Seejärel valime klaviatuuril „KL" aadressi 00 ning anname stardi. Arvuti hakkab käske interpreteerima:

0) Kood on 50. Signaaltuli „Anna arv" süttib. Valime klaviatuurilt *A* väärtuse kahendarvu 000 000 000 000 111 (so, arvu 7) ja vajutame nupule „kirjuta". Protsessor kirjutab aadressile 10 arvu 7.
1) Kood on taas 50, sisestame *B* väärtuse 477_8 (kümnendarvu 319) aadressile 11.
2) Liidetakse aadressidel 10 ja 11 olevad arvud, resultaat 506_8 kirjutatakse aadressile 12.
3) Aadressilt 12 loetakse arv 506_8 ja trükitakse paberilindile.
4) Protsessor seiskub („KL"-tabloo ei vaheta enam aadresse, vaid seis 04 jääb põlema).

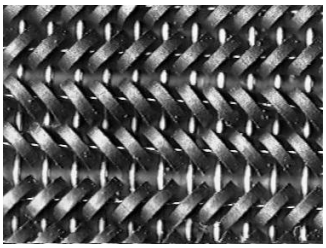
Meie teine programm võib programmeerija jaoks välja näha järgmiselt:

Aadress	**Kood**	**A1**	**A2**	**A3**	**Kommentaar**
4	00	00	00	01	Konstant 1
5	11	03	03	03	**S1**: sisendpunkt; *K:*=0
6	11	01	01	01	*S*:=0
7	11	02	02	02	*I*:=0
10	50	00	00	00	**S2**: Loe *A*; signaal *CC*
11	60	15	12	12	Kui *A*=0, mine S3
12	10	00	01	01	*S:=S+A*
13	10	02	04	02	*I:=I*+1
14	60	10	10	10	Tingimusteta suunamine S2
15	20	02	21	00	**S3**: Kas *I*=0?
16	60	20	17	17	Kui on, mine S4
17	13	01	02	03	*K:=S/I*
20	70	00	00	03	**S4**: Väljasta *K*
21	00	00	00	00	Stopp

Tabel 2.3.1c. Teine programm.

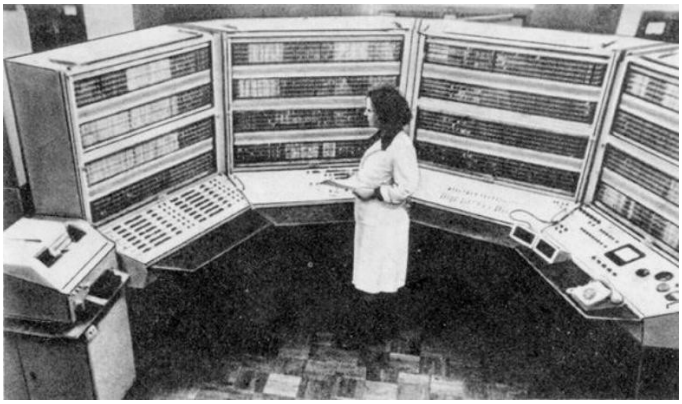
Mainisime juba, et kolmeaadressilise arvuti ilmne eelis on käsu loomulik struktuur, mis hõlbustab nii programmi kirjutamist kui ka lugemist. Siiski polnud sellised masinad prevaleerivad, vaid pigem marginaalsed. N. Liidus oli tuntuim kolmeaadressilise arvuti *БЭСМ-4* (Большая Электронная Счетная Машина); selle perekonna tuntuim mudel, *üheaadressiline БЭСМ-6* oli omal ajal võimsaim Euroopas (1 miljon tehet sekundis), ja võrreldav suurimate USA mudelitega.
Kolmeaadressilise arhitektuuri puuduseks oli pikk pesa, kuivõrd noil aegadel oli operatiivmälu kalleim ressurs (näiteks esialgu elektronlampidest või hiljem ferriitrõngakestest koosnev: iga bitt vajas ühte lampi või rõngakest).

Oletagem, et meie kolmeaadressilise arvuti projekt kukkus just ülemäärase pesa pikkuse tõttu läbi ning meile tehti ülesandeks konstrueerida samade võimalustega, ent 11-bitise pesaga arvuti. Selle masina projekt on esitatud järgmises jaotises.



Jooni 2.3.1c. Ferriitmälu fragment[1] (vt. [79]).

[1] Selle realisatsiooni leiutas *Jay Forrester* (*Massachusettsi Tehnoloogiainstituudis*, *MIT*) 1951. aastal.



Joonis 2.3.1d [75]. Arvuti *БЭСМ-6* inseneripult.

2.3.2. Üheaadressiline arvuti

Nagu juba selle masina nimetus (“üheaadressiline”) ütleb, on selle arvuti käsu põhimõtteline formaat järgmine:

Kood	Aadress

Mälu maht jääb endiseks ─ 64 pesa. Järelikult, aadressosa pikkuseks piisab endiselt 6 bitist ning käsukoodile jääb 5 bitti.

Loomulikult tekib küsimus, kuidas tuleb realiseerida binaarseid tehteid. See lahendatakse nii, et protsessori koosseisu tuuakse juurde veel üks pesa (mis ei kuulu operatiivmälu hulka), milles tehte sooritamise ajal hoitakse ühe operandi väärtust ja kuhu pärast tehte sooritamisest kirjutatakse selle tulemus. Seda pesa nimetatakse **summaatoriks.** Tähistagem teda edaspidi tähega S.

Ilmselt peame lisama kaks käsku: lugeda mälust arv summaatorisse ja kirjutada summaatorist arv mällu. Tähistagem neid tehteid nii:

(A)→S ja (S)→A.

Uue masina käskude formaadid on tabelis 2.3.2.a.

Kood	**Tegevus**	**Semantika**	**Signaalid**
00	Stopp		pole
01	(A)→S	Loe summaatorisse	CC
02	(S)→A	Kirjuta summaatorist mällu	CC
10	(S)+(A)→S	Liida	CC, OF
11	(S)-(A)→S	Lahuta	CC, OF
12	(S)x(A)→S	Korruta	CC, OF
13	(S)/(A)→S	Jaga	CC, OF
20	(S)¤(A)	Võrdle	CC

21	Mine A	Kui CC=0	Ei muuda
22	Mine A	Kui CC=1	Ei muuda
23	Mine A	Kui CC=2	Ei muuda
24	Mine A	Tingimusteta	Ei muuda
26	Sisend→A	Lugemine	CC
27	(A)→väljund	Kirjutamine	CC

Tabel 2.3.2a.. Üheaadressilise arvuti käsud.

Allpool esitame meie kaks näiteprogrammi üheaadressilise masina koodis nii, nagu programmeerija seda võiks kirja panna.

Aadress	**Kood**	**A**	**Kommentaar**
00	26	10	Loe *A* aadressile 10
01	26	11	Loe *B* aadressile 11
02	01	10	(10) → S
03	10	11	(S)+(11) → S
04	02	12	(S) → 12
05	27	12	(12) → väljund
06	00	00	Stopp

Tabel 2.3.2b. Esimene üheaadressilise masina programm.

Teine programm ─ kui järgida mälujaotust, et $A,\sum$ (tähistame nii summat, vältimaks summaatoriga ära vahetamist), I ja K aadressid on endiselt 0, 1, 2 ja 3 ning konstant 1 on aadressil 4 ─ võib saada selline:

Aadress	**Kood**	**A**	**Kommentaar**
00			*A*
01			∑
02			*I*
03			*K*
04	00	01	Konstant 1
05	01	31	**Sisendpunkt:** 0 → S
06	02	01	∑:=0
07	02	02	*I*:=0
10	02	03	*K*:=0
11	26	00	**S1**: Loe *A*
12	21	22	Kui *A*=0, mine **S2**
13	01	01	(∑) → S
14	10	00	(S)+(*A*) → S
15	02	01	(S) → ∑
16	01	02	*I* → S
17	10	04	(S)+1 → S

20	02	02	(S) → *I*
21	24	11	Mine **S1**
22	01	02	**S2**: *I* → S
23	20	31	Kas *I*=0?
24	21	30	Kui jah: mine **S3**
25	01	01	(∑) → S
26	13	02	(S)/*I* → S
27	02	03	(S) → K
30	27	03	**S3**: Trüki *K*
31	00	00	Stopp (üksiti konstant 0)

Tabel 2.3.2c. Teine programm.

Kommenteerigem vaid 7. ja 10. käsku. Võib tekkida küsimus, mis on summaatoris pärast selle kirjutamist mällu 6. käsus. Arvutid ehitatakse nii, et mälust lugemine ei riku mälu seisu ja täpselt samuti ei riku summaatori seisu selle kirjutamine mällu. Pesas olev info muutub ainult siis, kui sinna midagi **kirjutatakse**.

Võime nentida, et meie üheaadressilise ainus silmnähtav eelis on lühem pesa (ja oluliselt odavam mäluseade), ent selle eest peame maksma pikema programmi ja kujutatavate arvude väiksema diapasooniga: nende maksimaalne absoluutväärtus on nüüd 2^{10}-1 (=1023 või 1777_8). Ent need on silmnähtavad seigad, eelised on varjatud.

Esiteks, pesa pikkus pole sisuline arvude kujutamisvõimaluse piiraja: me võime arve kujutada ka nii, et nad asuvad *n* järjestikuses pesas (tavaliselt *n*=2 või 4) ja mõistagi tuleb sel juhul lisada käskude nomenklatuuri aritmeetika- ja võrdlustehted kõigi pikkusklasside jaoks.

Teiseks, iga käsu täitmise kiirus (sooritamiseks vajalike taktide arv) on üheaadressilise arvuti puhul ca 3 korda väiksem kui kolmeaadressiline masin vajab, seega programmide täitmise kiirus ei pruugigi olla kolmeaadressilise masina kasuks. Käsukoodi interpreteerimine ja summaatoriga suhtlemine „maksavad" vähe, „kallis" on mälu poole pöördumine.

Kolmandaks, meie teise programmi pikkus kolmeaadressilise arvuti jaoks on 18 pesa ja üheaadressilisel arvutil 26 pesa ja võib näida, et üheaadressiline „raiskab mälu", ent esimesel juhul on programmi pikkus 432, teisel aga 312 bitti. Ja et bitt oli pool sajandit tagasi küllaltki kallis ressurss (väärtuse säilitamiseks näiteks raadiolamp), siis tuleks anda plusspunkte üheaadressilisele masinale.

Mainigem, et **pesamasinate** ajal prevaleeris kaheaadressiline arhitektuur, ent järgmine tase ─ **baitmasinad** ─ täidavad lõviosa käskudest üheaadressilises režiimis.

2.3.3. Kaheaadressiline arvuti

Kaheaadressilise arvuti käsk peab olema ilmselt järgmise struktuuriga:

Kood	A1	A2

Jättes mälumahu samaks nagu ta oli eelmistes arvutites ($64 = 100_8$), on pesa aadressosa pikkuseks 12 bitti. Kui käsukoodi jaoks reserveerime kolmeaadressilise masina 6 bitti, saame pesa pikkuseks 18 bitti.

Nii nagu üheaadressilise arvuti puhul, tuleb ka siin otsustada, kuhu salvestada aritmeetikatehete resultaat. Võime teha nii, et tulemus kirjutatakse ükskõik kumma operandi aadressile, ent siis jääme ilma võimalusest tolle väärtuse taaskasutamisest. Sobiv lahendus on **summaatori** ülevõtmine üheaadressilise arvuti projektist. Nii saame huvitava võimaluse aritmeetiliste tehete formaatide varieerimiseks (tähistagem tehet üldistatult sümboliga ¤):

- (A1)¤(A2) → S
- (A1)¤(A2) → S, A2
- (S)¤(A1) → S, A2
- (S)¤(A1) → S

Esitame kaheaadressilise masina esialgse käskude komplekti tabelis 2.3.3a.

kood	**aadressid**	**tegevus**	**signaalid**
00	Ei kasuta	Stopp	pole
01	A2	(A2)→S	CC
02	A2	(S)→A2	CC
03	A2	sisend→A2	CC
04	A2	(A2)→väljund	CC
10	A1, A2	(A1)+(A2)→S	CC, OF
11	A1, A2	(A1)+(A2)→S, A2	CC, OF
12	A1, A2	(S)+(A1)→S,A2	CC, OF
13	A1	(S)+(A1)→S	CC, OF
20	A1, A2	(A1)-(A2)→S	CC, OF
21	A1, A2	(A1)-(A2)→S, A2	CC, OF
22	A1, A2	(S)-(A1)→S,A2	CC, OF
23	A1	(S)-(A1)→S	CC, OF
30	A1, A2	(A1)x(A2)→S	CC, OF
31	A1, A2	(A1)x(A2)→S, A2	CC, OF
32	A1, A2	(S)x(A1)→S,A2	CC, OF
33	A1	(S)x(A1)→S	CC, OF
40	A1, A2	(A1)/(A2)→S	CC, OF
41	A1, A2	(A1)/(A2)→S, A2	CC, OF
42	A1, A2	(S)/(A1)→S,A2	CC, OF
43	A1	(S)/(A1)→S	CC, OF

50	A1, A2	(A1) ¤ (A2)	CC
60	A2	Mine A2	Ei muuda
61	A2	Kui CC=0, mine A2	Ei muuda
62	A2	Kui CC=1, mine A2	Ei muuda
63	A2	Kui CC=2, mine A2	Ei muuda
64	A1,A2	Kui CC=0, mine A1, muidu A2	Ei muuda
65	A1,A2	Kui CC=2, mine A1, muidu A2	Ei muuda
70	A1, A2	(A1) → A2, S	CC
71	A1, A2	(A1) ↔ (A2)	Ei muuda

Tabel 2.3.3a. Kaheaadressilise arvuti käskude lähtekomplekt.

Juhime tähelepanu kaheaadressilise arvuti tüüpilistele käskudele: need on „suunamine nulli järgi" (kood 64: kui signaali andnud tehte resultaat on 0, siis mine A1, muidu A2) ja „suunamine märgi järgi" (kood 65: miinuse korral mine A1, muidu A2) ning „vahetamine" (kood 71: vaheta arvud (A1) ja (A2)).

Meie eelmiste masinate näiteprogrammid võivad kaheaadressilisel arvutil olla sellised:

Aadr.	**Kood**	**A1**	**A2**	**Kommentaar**
00	03	00	10	Loe *A*
01	03	00	11	Loe *B*
02	11	10	11	(*A*)+(*B*)→*B*, summaator
03	04	00	11	(*B*)→väljund
04	00	00	00	Stopp

Tabel 2.3.3b. Programm B:=A+B.

Paar kommentaari. Pärast 2. käsu liitmistehet pole meil enam *B* väärtust vaja, seega võime avaldise *Y:=A+B* asemele programmeerida *B:=A+B*. Teise näiteprogrammi üks võimalikest tekstidest on järgmine:

Aadr.	**Kood**	**A1**	**A2**	**Kommentaar**
00	00	00	00	Muutuja *A*
01	00	00		Muutuja ∑
02	00	00	00	Muutuja *I*
03	00	00	00	Muutuja *K*
04	00	00	01	Konstant 1
05	00	00	00	Konstant 0
06	70	05	01	0→∑, S
07	02	00	02	*I:*=0
10	03	00	00	Sisend→00 (loe *A*)
11	61	00	15	Kui CC=0, mine 15
12	11	00	01	*A*+∑→S, ∑
13	11	04	02	1+*I*→S, *I*

14	60	00	10	Mine lugema (10)
15	50	02	05	Võrdle *I* ja 0
16	61	00	22	Kui CC=0, mine 22
17	40	01	02	$\sum/I \rightarrow$S
20	02	00	03	(S)$\rightarrow K$
21	04	00	03	(03)→väljund (trüki *K*)
22	00	00	00	Stopp

Tabel 2.3.3c. Kaheaadressilise masina teine programm.

Võrrelgem: kui see programm vajas kolmeaadressilisel arvutil 432 bitti ja üheaadressilisel 312 bitti, siis nüüd 442 bitti. Tõsi, kui hoiaksime kokku konstant 0 pealt, oleks käesoleva programmi pikkus 424 bitti.

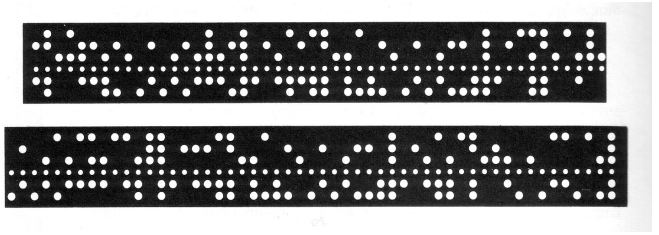
2.3.4. Massiiv ja tsükkel

Mõlemad näiteülesanded, mille jaoks me programme oleme koostanud, opereerisid lihtmuutujatega: igal neist on korraga üksainus väärtus selleks ette nähtud aadressil. Tihti tuleb aga lahendada ülesandeid, kus mõnel muutujal on korraga terve rida erinevaid väärtusi. Meie teise näiteülesande võiksime püstitada nii, et *A* pole lihtmuutuja, millele omistame järjest sisendist loetud väärtusi, vaid **vektor** e**. ühemõõtmeline massiiv**, mille võiksime sisestada ühe käsuga Loogilisel tasemel võiksime vektorit tähistada *A*[*N*], kus *N* on elementide arv ja *A*[*i*] on vektori *i*-s element ($1 \le i \le N$) ning *i* on **indeks.**

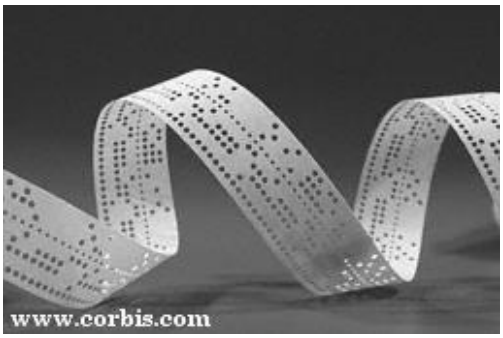
Modifitseerime oma kaheaadressilise arvuti sisestamise ja väljastamise käske selliselt:

03	A1	A2	Kui A1=0, siis loe 1 arv, muidu aga (A1) arvu ja kirjuta alates A2
04	A1	A2	Kui A1=0, siis kirjuta 1 arv, muidu aga (A1) arvu alates A2

Arvutile peame lisama uue välisseadme ─ kas perfokaart- või perfolintsisendi.



Joonis 2.3.4a. Viiebitine perfolint [4, lk. 1]



Joonis 2.3.4b. Kaheksabitine perfolint (internet).

Vektori kui andmestruktuuriga on olemuslikult seotud operatsioon **tsükkel** ─ selle käigus korratakse mingit tegevust järgemööda kõigi vektori elementidega. Esitame allpool algoritmi vektori elementide summa leidmiseks.

```
Ette on antud vektor P[N]
Start: I:=0; S:=0;
     S1: S:=S+P[I];
         I:=I+1;
         Kui I=N, mine S2;
         Mine S1;
     S2:
```

Tsükli „tegevus" on summeerimiskäsk, sellele järgnevad kolm käsku moodustavad tsükli juhtimise ploki. Indeksi muutmise realiseerime kaheaadressilise masina olemasolevate vahenditega, ülejäänu jaoks lisame uue käsu („tsüklikäsk"):

07	A1	A2

Käsk töötab järgmiselt: aadressil A1 paikneb **tsükliloendaja** (meie näites *N*) Käsu 07 täitmisel tehakse tehe *N:=N*-1; kui *N*=0, siis antakse juhtimine järgmisele käsule, muidu käsule aadressil A2.

Nüüd võime esitada oma teise näiteprogrammi uue versiooni. Lihtsustamise huvides loobume kontrollimistest kas etteantav arv *N*>0? Kas vektor mahub mällu? Indekseerimisvahendeid meil veel pole; sestap kasutame vektori adresseerimiseks üht *v. Neumanni* printsiipi, mis lubab käske käsitleda nagu tavalisi andmeid ─ vt. käsku aadressil 06.

Aadr.	**käsk**	**kommentaar**
00	03 00 13	Sisesta N
01	03 13 20	Sisesta vektor A[N]
02	70 16 14	S:=0
03	70 13 15	M:=N
04	10 14 20	S+A[0]→summaator
05	02 00 14	summaator→S

06	11 17 04	4. käsu modifitseerimine
07	07 15 04	Tsüklikäsk
10	41 14 13	N:=S/N
11	04 00 13	Trüki N
12	00 00 00	stopp
13	00 00 00	N
14	00 00 00	S
15	00 00 00	M
16	00 00 00	Konstant 0
17	00 00 01	Konstant 1
20	00 00 00	P[0]
21	00 00 00	P[1]
jne	jne	jne

Tabel 2.3.4a. Tsüklikäsu kasutamine.

Vaatleme seda programmi käskhaaval..

0) Aadressile 13 loetakse vektori elementide arv *N*.
1) Alates aadressist 20 loetakse mällu *N* arvu ─ vektor *A*.
2) Aadressile 14 (sinna kogume summat *S*) saadetakse aadressilt 16 arv 0.
3) Aadressile 15 (*M*) saadetakse *N* väärtus (*M* kasutame tsükliloendajana).
4) Liidetakse arvud aadressidelt 14 (*S*) ja 20 (*A*[0]).
5) Resultaat kirjutatakse summaatorist aadressile 14 (*S*).
6) Liidetakse arvud aadressidelt 17 ja 04 ja resultaat kirjutatakse aadressile 04; nüüd on seal käsk 10 14 21.
7) Tehakse tehe *M:=M*-1. Kui *M*≠0, antakse juhtimine käsule aadressiga 04, muidu aga järgmisele käsule.
10) Leitakse jagatise *S/N* täisosa ja kirjutatakse see aadressile 13.
11) Trükitakse *N*.
12) Stopp.

Mõistagi võinuks me 4. käsu kirjutada ka nii: 04) 11 20 14, ent sel juhul peab olema aadressil 17 konstant 00 01 00.

Jääme selle variandi juurde ja vaatame, mis juhtub, kui *N* väärtuseks sisestada (eksikombel muidugi) 0. Ilmselt tsüklikäsk ei lõpetagi tööd, kuivõrd lõputingimust *M*=0 ei saabu. Neljas käsk teeb ettenähtud tööd viimati veel kujul

04) 11 77 17,
seejärel tehakse 6. käsu toimel liitmistehe

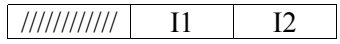
11 77 17+00 01 00=12 00 17

ja see resultaat kirjutatakse 4. käsuks, mis teeb aga sootuks midagi muud: summaatori sisule liidetakse aadressil 0 olev arv 030013 ning kirjutatakse tulemus aadressile 17. Arvuti jatkab tööd seni, kuni 4. või 6. käsk saavad ületäitumise või satub juhtimine pesale,

mille 6 esimese biti väärused on nullid ─ seda interpreteeritakse stopp-käsuna, või modifitseeritakse 4. käsku olematu koodi tekkimiseni… Millal just lõpp saabub, on raske ennustada.

2.3.5. Indekseerimine

Eelmises jaotises nägime, et vektorit saab indekseerida käsku modifitseerides; mitmetel põhjustel pole see aga hea variant. Et see kätkeb raskestiavastatavate vigade ohtu, nägime äsja. Lisaks pole modifitseeritav programm taaskasutatav (*reenterable*), taaskasutatavus on aga üheks üldiselt aktsepteeritud nõudeks heale programmile. Tegelikes arvutites kasutatakse indekseerimiseks kiireid (so, protsessori koosseisu kuuluvaid) **indeksregistreid** (summaatoritki võime vaadelda kui üht spetsiaalset kiiret registrit). Lisagem oma õppearvutile 7 indeksregistrit numbritega 1..7; iga neist on sama pikk kui mälupesa ja järgmise formaadiga:



Käsu formaati täiendame 3-bitise väljaga „indeks":

Osad „Kood", „D1" ja „D2" on endiselt 6-bitised, „I" aga 3-bitine; seega on meie masina pesa pikkus nüüd 21 bitti. Kui käsus I=0, siis käsku ei indekseerita, kui aga $1 \le I \le 7$, siis tuleb käsku indekseerida I-nda indeksregistri abil. Indekseerimisel leitakse operandide tegelikud aadressid komponentide liitmise teel:

A1=D1+I1 ja A2=D2+I2.

Näiteks, kui 3. indeksregistris on indeksid I1=0 ja I2=2, siis käsk 71 3 45 45 vahetab arvud aadressidelt 45 ja 47. Indeksregistris I **nn. indekskonstandi** kandmiseks lisame käsu koodiga 05:

05 I 00 D2

Indekskonstandi (siin aadressil D2) formaat kattub indeksregistri omaga: esimest 9 bitti ei kasutata, järgnevad kaks kuuebitist arvu X1 ja X2. Indeksi modifitseerimiseks lisame käsu

06 I 00 D2,

kus aadressil D2 paikneb indekskonstant. Käsu toimel muudetakse I-ndas indeksregistris indeksite I1 ja I2 väärtused:

I1:=I1+X1 ja I2:=I2+X2.

Eelmises punktis toodud näite võime uusi võimalusi kasutades ümber kirjutada järgmiselt:

Aadr.	**käsk**	**kommentaar**
00	03 0 00 13	Sisesta N
01	03 0 13 20	Sisesta vektor A
02	70 0 16 14	∑:=0
03	70 0 13 15	M:=N
04	05 1 00 16	0;0→indeksregister 1
05	11 1 20 14	∑:=∑+A[i]
06	06 1 00 17	I:=i+1
07	07 0 15 05	tsüklikäsk
10	41 0 14 13	N:=∑/N
11	04 0 00 13	Trüki N
12	00 0 00 00	stopp
13		N
14		∑
15		M
16	00 0 00 00	Konstant 0
17	00 0 01 00	Indekskonstant 1;0
20		A[0]
21		A[1]
Jne		

Joonis 2.3.5a. Indeksregistrite kasutamine.

2.3.6. Baseerimine

Meie kaheaadressilise arvuti mäluseade koosneb 64 pesast. Sellega arvestades on valitud käsu formaat, see omakorda määrab pesa pikkuse. Mälu mahu suurendamine olemasoleva variandi puhul on võimalik ainult pesa pikkuse suurendamise teel. Ent oletagem nüüd, et saame ülesande hakata tootma oma masinat erinevate mälumahtudega: odavaimal versioonil olgu endiselt 64 mälupesa, ent vastavalt nõudlusele tuleks toota ka 128, 192, 256 jne mälupesaga ─ mälu võiks koosneda *n* mäluplokist á 64 pesa ning peab olema võimalus, et klient saab hõlpsalt mälu(plokke) lisada ($n \leq 7$).

Meie jaoks tähendab see, et peame adresseerima operande, mille aadress ei mahu käsus aadressväljale. Lahendame ülesande nii, et lisame kiiresse mällu veel 8 registrit ─ nimetagem neid **baasregistriteks,** numbritega 0..7. Iga baasregistri sisu interpreteeritakse kui üht arvu (erinevalt indeksregistrist!) ja see kujutab endast mingi mäluosa tegelikku aadressi e. **absoluutaadressi.** Operandi tegeliku aadressi leiab protsessor nüüd alati vähemalt kahe komponendi ─ baasaadressi ja käsku kirjutatud suhtaadressi summana. Sellele võib lisanduda veel indekskomponent. Käsu formaadis lisandub kummalegi aadressile 3-bitine osa „baas": baasregistri number. Uus formaat on järgmine:

kood	I	B1	D1	B2	D2

Väljad „kood", „D1" ja „D2" on kuue-, ülejäänud aga kolmebitised. Pesa pikkus on nüüd 27 bitti.

Protsessori võime ehitada nii, et programmi baasaadress on alati baasregistris 0; programmi mällu tuues laaditakse ta tegelik aadress just sellesse registrisse. Muuseas ei sõltu programmid tänu baseerimisele enam nende asukohast mälus.

Käskude repertuaari toob baseerimine veelgi lisa:

- Käsk absoluutaadressi baasregistrisse kirjutamiseks: arvutatakse A=(B)+(I)+D ja A kantakse käsus näidatud registrisse.
- Käsud baasaadressi muutmiseks.

Arvestagem sellega, et D1 ja D2 pole enam pesade tegelikud aadressid, vaid suhtaadressid baasaadresside suhtes.

2.3.7. Andmete kujutamine

Loobugem selles punktis oma õppearvutist ja tutvustame kaheaadressilise arvuti *Minsk-32* (pesa pikkus 37 kahendkohta) poolt toetatavaid andmetüüpe (vt. [40], lk. 11-14):

- **Fikseeritud komaga kahendarvud**. Vasakpoolseim bitt on arvu märgi jaoks (negatiivse arvu puhul 1) ning numbriosa kahendkohtadel 1..36. Tüübi nimetuses mainitud komakoht on kohe märgi järel. Laskumata tehnilistesse üksikasjadesse mainigem vaid, et selline interpreteering vähendab ületäitumise ohtu. Seda tüüpi arvude diapasoon on

  $$1 - 2^{-36} \geq |X| \geq 2^{-36}$$

  Täisarvude diapasoon on $2^{36} - 1 \geq |X| \geq 0$

- **Ujupunktarvud** kujutatakse pesas nii: 0. bitil on mantissi märk, kahendkohtadel 1..28 on mantiss M, 29. kahendkohta arvu esitamisel ei kasutata (seda teeb aritmeetikaseade ümardamisel) ning kahendkohtadel 30..36 on arvu järk P. Arv on kujul $X=M\cdot 2^P$. Ujupunktarvud võivad olla *normaliseeritud* (1. bitil on 1), sel juhul $1>M\geq ½$. *Normaliseerimata* arvu puhul $M<½$. Normaliseeritud arvude diapasoon on $(1 - 2^{-28})\cdot 2^{63} \geq |X| \geq 2^{-64}$ ning normaliseerimata arvudel on see $(1 - 2^{-27})\cdot 2^{62} \geq |X| \geq 2^{-91}$.
- **Kümnendarvud** kujutatakse nn. tetraadidena, iga number neljal bitil. Sõna 0. bitt on siingi märgi jaoks ning pessa mahub kuni 9-kohaline kümnendarv. Numbri 8 kahend-kümnendkood on 1000 ja 9 oma ─ 1001
- **Teksti** mahub pessa viis 7-bitist koodi ГОСТ 10859─ 64 sümbolit, näiteks sümboli ≠ kood on 0011100, Ж ─ 0100110 ja W ─ 1001011.

Mõistagi on iga tüübi jaoks oma käskude komplekt.

2.4. Õppearvuti assembler

2.4.1. Programmeerimine masinkoodis

Arvuti on võimeline interpreteerima ainult kahendarvudena esitatud infot, nii käske kui ka operande. Kahendarvudena (*resp.* kaheksand- või kuueteistkümnendarvudena) esitatud programmi nimetatakse masinkoodi-programmiks. Paraku on see keel inimese jaoks küllaltki tülikas ─ ehkki sellegagi on võimalik harjuda. Tänapäeval kirjutatakse masinkoodis äärmiselt tühine osa programmidest, seda oskust läheb vaja näiteks olemasolevatest sootuks erineva käskude süsteemiga arvuti „tuumtarkvara" loomiseks, või siis pahatahtlikel häkkeritel mingi uue toote loomisel. Seevastu arvutite kasutamise algaastail oli programmeerimine masinkoodis programmeerijate jaoks täiesti tavaline töö; nende ridade autor tegeles sellega näiteks 1970. aastal, osaledes Eesti Raadio Arvutuskeskuses süsteemi „SODI[1]" tegemisel masinale *Razdan-3* (bossiks oli *Leo Võhandu*, juhtprogrammeerijaiks *Mati Räbovõitra* ning hilisemad professorid *Toomas Mikli*, *Mati Tombak* ja *Jaak Tepandi*. See polnud mingi erand, just masinkoodis kirjutati meil ka enamus *Ural*-seeria arvutite tarkvarast.

Programmeerimiskeelena on masinkoodil üks vaieldamatu eelis kõigi muude võimaluste ees. Nimelt: masinkoodis programmeerides saab programmeerija kasutada täielikult kõiki arvuti võimalusi, tal tekib täiuslik masinatunnetus, mis on üsna lähedane arvutiinseneri omale ─ nende tööks oli kümneid kordi vahetuse jooksul leida ja parandada aparatuurseid vigu.

Puudusi on paraku rohkem. Loetlegem neist mõningaid.

- Programmeerija peab *ise* tegelema oma programmi objektide adresseerimisega. Vaatame näiteks kaheaadressilise õppearvuti teise programmi teist varianti (tabel 2.3.4a lk 34): selle kirjutas nende ridade autor kahes jaos: esiteks kirjutas ta käskude operandide aadresside asemel nende enda jaoks arusaadavad tähised, näiteks käsk aadressil 0 nägi algselt paberil välja nii:

  00) 03 00 N

  Ja alles siis, kui programmi viimane käsk oli kirjutatud, selgusid operandide tegelikud suhtaddressid (*N*-i omaks sai 13). Viimased tuli kirjutada käskudesse tähiste asemele. Vähegi keerukama loogikaga programmi ei õnnestu peaaegu kunagi kirjutada nii, et seal ei tuleks midagi parandada. Tavaliselt tuleb mõni käsk vahele panna või vastupidi, kõrvaldada. Usutavasti pole vaja pikemat seletust, kuidas see mõjub aadressidele. Masinkoodis programmeerijatel oli terve arsenal teravmeelseid võtteid, kuidas sellistest olukordadest võimalikult puhtalt välja tulla.
- Suunamiskäskdega kaasnevad samasugused probleemid. Ettepoole suunates ei tea me veel, millisele suhtaadressile tuleb juhtimine anda, käskude lisamine või eemaldamine põhjustab ka suunamiskäsu parandamist.

[1] Система Обработки Дискретной Информации.

- Võõrast (aga reeglina ka oma vana) programmi on väga raske lugeda: puuduvad autori kavatsusi mõista hõlbustavad kommentaarid ─ nende jaoks pole lihtsalt kohta. Programmi mõista on aga vaja, et parandada mõnda avastamata jäänud viga või modifitseerida algoritmi.
- Programmeerija keskmine tööviljakus oli väga madal. Tolleaegse folkloori järgi oli normaalne tootlikkus kaks siledat masinkoodi käsku päevas. Tuleme selle väite juurde tagasi veidi hiljem, kui käsitleme *moodulprogrammeerimist*.

Masinkoodis programmeerimise tehnikast. Programmid valmistati ette ja kirjutati eranditult paberil, käsud perforeerimiseks tüüpiliselt spetsiaalsetel plankettidel. Programmeerija pani esmalt kirja programmi üldise skeemi (tavaliselt joonistas plokk-skeemi), selle koostamise käigus pani oma objektidele nimed ning esimeses masinkoodi-versioonis opereeriski nonde nimedega; enne perforeerimisvalmi versiooni kirjutamist pandi paika objektide aadressid ning kirjutati need käskudesse.

Näiteks jooniselt 2.3.5a tuttava programmi tööversioon võis välja näha selline:

Aadr.	**käsk**	**kommentaar**
00	03 0 00 N	Sisesta N
01	03 0 N A	Sisesta vektor A
02	70 0 +0 ∑	∑:=0
03	70 0 N M	M:=N
04	05 1 00 0,0	0;0→indeksregister 1
05	11 1 A ∑	Tsükkel:∑:=∑+A[i]
06	06 1 00 1,0	I:=i+1
07	07 0 M Tsükkel	tsüklikäsk
10	41 0 ∑ N	N:=∑/N
11	04 0 00 N	Trüki N
12	00 0 00 00	stopp
13		N
14		∑
15		M
16	00 0 00 00	Konstant 0
17	00 0 01 00	Indekskonstant 1;0
20		A[0]
21		A[1]
Jne		

Joonis 2.4.1a. Masinkoodi tööversioon.

Mõistlik on oletada, et esialgu said kirja käsud 0..12 ja seejärel tehti mälujaotus ning pärast seda kirjutati tekst perforeerimiseks, asendades käskudes nimed aadressidega. On lihtne näha, et viimase tööga saab näpuvigasid tegemata sootuks paremini hakkama spetsiaalne programm kui programmeerija ise. Ja loomulikult selliseid programme hakati looma; nende üldtunnustatud nimetus on *assembler* ja etteantav tekst vormistatakse *assemblerkeeles*.

Enne assembleri tutvustamist näitame (demonstreerimaks probleemi tõsidust) viimase näite perforeeritavat ja seejärel sisestatavat programmi.

```
0300013
0301320
7001614
7001315
0510016
1112014
0610017
0701505
4101413
0400013
0000000
```

2.4.2. Assemblerkeel

Alustagem näitest. Tabelis 2.4.2.a on kõrvuti osast 2.3.5 tuttav masinkoodiprogramm ja talle vastav assemblerprogramm. Assembler on programmeerimissüsteem, mis koosneb

Aadr.	**masinkood**	**assembler**			
00	03 0 00 13	KESK	SISSE1	N	Loe N
01	03 0 13 20		SISSEN	N,A	Loe A[1:N]
02	70 0 16 14		SAADA	K0,S	S:=0
03	70 0 13 15		SAADA	N,M	M:=N
04	05 1 00 16		SAADI	:1,K0	I:=0
05	11 1 20 14	RING	PLUSSK	:1,A,S	S:=S+A[I]
06	06 1 00 17		PIND	:1,I10	I:=I+1
07	07 0 15 05		TSÜKK	M,RING	
10	41 0 14 13		JAGAK	S,N	N:=S/N
11	04 0 00 13		TRÜKI1	N	Trüki N
12	00 0 00 00		STOPP		
13	00 0 00 00	N	RES	1	vekrori pikkus
14	00 0 00 00	S	RES	1	summa
15	00 0 00 00	M	RES	1	tsükliloendaja
16	00 0 00 00	K0	AK	0	null
17	00 0 01 00	K10	IK	1,0	indekskonstant
20	00 0 00 00	A	RES	40	vektor A[1:40]
			LÕPP		

Tabel 2.4.2a. Masinkood ja assembler.

programmeerimiskeelest ja translaatorist. Mingi arvuti (või protsessoritüübi) assemblerkeel on selle arvuti masinkoodiga peaaegu üksüheses vastavuses: ühele assemblerprogrammi reale (direktiivile) vastab tavaliselt üks rida masinkoodis kirjutatud programmis. Assemblerkeel on *masinorienteeritud* keel. Assembler-translaator on programm, mis „tõlgib" assemblerkeelse programmi masinkoodi.

Assembleri mõte on hõlbustada programmeerija tööd nii, et masinkoodis programmeerimisega võrreldes võimalused oluliselt ei väheneks[1], ent kõige tülikamad tegevused (eelkõige mälu jaotamine) jääksid arvuti kanda. Assemblerkeeltele omaseid peamisi vahendeid käsitleme pisut hiljem (osas 2.6), enne seda aga projekteerime oma õppearvutile assembler-translaatori.

2.4.3. Õppearvuti assembler-translaator

Masinkoodi-versiooniks võtame eelviimase, kus veel polnud baseerimist ning assemblerkeele fikseerime minimaalsete võimalustega: võime anda programmi objektidele nimesid, kasutada käsukoodide asemel nn. mnemokoode, kirjutada operandide aadresside asemel nimesid ning kasutada mälu reserveerimise ja konstantide defineerimise „pseudodirektiive".

Mäletatavasti on õppearvuti mälu maht 64 pesa. Lihtsate vahenditega pole võimalik nii vähese mäluga masinale translaatorit teha: translaator ise ei mahu mällu, rääkimata sellest, et sinna peab mahtuma veel vähemalt üks rida lähteprogrammist ja mingi osa väljundprogrammist (so., transleerimise käigus tehtavast masinkoodi-programmist). Lisaks pole õppearvutil teksti töötlemise vahendeid. Käsitlegem seetõttu oma õppearvutit masinana, mille tööd imiteeritakse pärisarvutis ja et viimase mälus on pärisarvuti vahenditega töötav translaator, mis transleerib õppearvuti assemblerprogrammi õppearvuti masinkoodi[2].

Assembler-programmid kirjutatakse tavaliselt spetsiaalsetele plankettidele. Olgu meie assemblerprogrammi formaat selline, nagu on esitatud tabelis 2.4.3a.

Meie translaator teeb oma töö kahes etapis (töötab kahe läbivaatusega): esimesel neist vaadatakse läbi lähtetekst ja moodustatakse etikettide tabel ning masinkoodi-programmi „toorik" ─ nn. *vahekeelne programm*. Teisel etapil „vaadatakse läbi" vahekeelne programm ja formeeritakse *objektprogramm* ─ programm masinkoodis.

[1] Enamik multikasutusrežiimiga masinate assembleritest „peidab" kasutaja eest mitmeid masinkoodi käske, kaitsmaks süsteemi julgeolekut kasutaja ettevaatamatusest või pahatahtlikkusest tingitud toimingute eest.

[2] See idee on laenatud *Jüri Kiholt*: enne arvuti *EC-1022* saamist oli vaja üliõpilasi tööks tolle masinaga ette valmistada. Kaks üliõpilast tegid diplomitööd, üks kirjutas *Minsk-32* assembleris EC-arvuti assembler-translaatori, teine aga EC-arvuti masinkoodi interpretaatori. Muide, sama töö tegi *Silver Aabjõe* 2004. a. kevadel selles raamatus esitatud masinkoodi ja assembleri jaoks.

Translaatori tabelid. Vajame kaht tabelit: „sisseehitatud" tõlketabelit ning töö käigus moodustatavat etikettide tabelit. Meenutagem, et *tabel* on *kirjetest* koosnev *abstraktne andmestruktuur,* kus kirje koosneb loogilisel tasemel *võtmest* ja sellega seotud informatsioonist. Tõlketabeli võtme rollis on käsu mnemokood ning etikettide tabelil ─ etikett.

nr	**etikett**	**kood**	**indeks**	**A1**	**A2**	**kommentaar**
1	KESK	SISSE1			N	Loe N
2		SISSEN		N	A	Loe A[1:N]
3		SAADA		K0	S	S:=0
4		SAADA		N	M	M:=N
5		SAADI	:1		K0	I:=0
6	RING	PLUSSK	:1	A	S	S:=S+A[I]
7		PIND	:1		I10	I:=I+1
8		TSÜKK		M	RING	tsüklikäsk
9		JAGAK		S	N	N:=S/N
10		TRÜKI1			N	Trüki keskmine
11		STOPP				
12	N	RES	1			Massiivi pikkus
13	S	RES	1			summa
14	M	RES	1			tsükliloendaja
15	K0	AK	0			Arv null
16	I10	IK		1	0	indekskonstant
17	P	RES	40			Vektor, kuni 40 elementi
18		LÕPP				

Tabel 2.4.3a. Assemblerprogramm.

Tõlketabeli kirje formaat on järgmine:

mnemokood	käsukood	P	I	OP1	OP2
1	2	3	4	5	6

P on direktiivi tunnus: P=0 käskdirektiivi ja P>0 pseudodirektiivi puhul. Indekseerimistunnus **I**=0, kui käsku ei indekseerita, I=1, kui käsku võib indekseerida ja I=2, kui käsu üks operand on indeksregister. Väljadel **OP1** ja **OP2** on nullid, kui käsk ei kasuta vastavalt komponenti D1 ja D2 ning ühed, kui kasutab. Mäletatavasti oli masinkäsu formaat selline:

Kood	I	D1	D2

Tabelis 2.4.3b on esitatud tõlketabeli „asjassepuutuv" osa, tegemata oletusi või vihjeid tabeli tüübile (läbivaadatav, järjestatud või paisktabel). Etikettide tabeli kirjed on pisut keerulisema struktuuriga: kui kirje võti (etikett *resp.* märgend) esineb assemblertekstis rohkem kui üks kord, siis kõigi „lisaesinemiste" jaoks moodustatakse ahel. Kirje formaat on järgmine:

etikett	V	väärtus	Viit/Ø

Kirjest alata võiva ahela lüli formaat on kolmik:

aadress	S	Järgmine/Ø

Etiketi *väärtuse* all mõistame talle vastavat suhtaadressi objektprogrammis. **V**=0, kui väärtust (veel) pole, ja 1, kui (juba) on. *Aadress* ahela lülis on formeeritava objektprogrammi pesa aadress, mille aadressosas kasutatakse etiketti. **S**=1, kui see on esimese ja 2, kui teise operandi rollis.

Jrk-nr	**mnemokood**	**käsukood**	**P**	**I**	**OP1**	**OP2**
1	SISSE1	03	0	1	0	1
2	SISSEN	03	0	1	1	1
3	SAADA	70	0	1	1	1
4	SAADI	05	0	2	0	1
5	PLUSSK	11	0	1	1	1
6	PIND	06	0	2	0	1
7	TSÜKK	07	0	1	1	1
8	JAGAK	41	0	1	1	1
9	TRÜKI1	04	0	1	0	1
10	STOPP	00	0	0	0	0
11	RES		2			
12	AK		3			
13	IK		4			
14	LÕPP		7			

Tabel 2.4.3b. Tõlketabeli fragment.

Esimene läbivaatus. See translaatori alamprogramm saab ette assemblerteksti *LP*, mida võime käsitleda kui ridadest koosnevat *N*-realist läbivaadatavat tabelit reanumbritega 1..*N.* Iga selle kirje (=assembler-programmi rida indeksiga *I*) *R* koosneb 6 väljast, viimaseid võime tähistada kui *R.etikett, R.kood, R.indeks, R.A1* ja *R.A2*; välja *R.kom-*

mentaar translaator ei kasuta. Formeeritav objektprogramm *OP* on ühepikkuste ridade (pesade) *O* hulk järjekorranumbritega 0..*M* (piir on esialgu lahtine, jooksva rea nr. on *K*) ning iga pesa väljad on *O.kood, O.I, O.D1 ja O.D2*. Tõlketabeli kirje välju tähistame *TT.mnemokood, TT.käsukood, TT.P, TT.I, TT.OP1 ja TT.OP2*. Analoogiliselt nimetame etikettide tabeli *ET* kirje *E* välju *E.etikett, E.V, E.väärtus* ja *E.viit* ning noist kirjeist viidatavate ahelate lülide *L* välju tähistame *L.aadress, L.S ja L.järgmine.* Järgides noid kokkulepitud tähistusi, võime esitada esimese läbivaatuse algoritmi:

algus: *I:=0, K:=0*; *J*:=0;

ring: $O:=OP_K$; $R:= LP_I$**;** Kui *R.etikett*= Ø, siis mine **kood**;

Kas etikettide tabelis on kirje *E*, kus *R.etikett= E.etikett*? Kui pole, siis

{lisa tabelisse uus kirje E_J, *E.etikett*:=*R.etikett*, *E.V*:=1, *E.väärtus*:=*K*; *J*:=*J*+1; mine **kood**;}

Kui *E.V*=1, siis {**viga**: „etikett *R.etikett* kordub real *I"*; mine **kood**;}

E.V:=1; *E.väärtus*:=*K*;

kood: Otsi tõlketabelist kirje *TT*, kus *TT.mnemokood=R.kood.* Kui sellist kirjet pole, siis **viga** „real *I* olematu mnemokood *R.kood*", mine järgmine;

Kui *TT.P*=0, siis

{*O.kood*:=*TT.käsukood*;

Kui *TT.I*=0 ja *R.indeks*≠0, siis **viga** „real *I* indekseeritud keelatud käsku";

mine **o1**;

Kui *TT.I*=1 ja *R.indeks*=0, siis mine **o1**, muidu kui *R.indeks*≠0, siis

index:{teisenda *R.indeks* kaheksandarvuks x; Kui 1≤x≤7, siis *O.I*:=x, muidu **viga** „real *I* liiga suur indeksregistri number"; mine **o1**}

Kui *TT.I*=2 ja *R.indeks*>0, siis mine **index**, muidu **viga** „real *I* puudub indeksregistri number"

o1: Kui *TT.OP1*=0, siis

{kui *R.A1*≠Ø, siis **viga** „real *I* ei või kasutada A1";

mine **o2**;}

muidu *s*:=1; *A*:=*R.OP1*; Kui *A*=Ø, siis {**viga „**real *I* puudub A1"; mine **o2**;}

aadress: otsi etikettide tabelist kirje *E*, kus *E.etikett=A.* Kui sellist pole, siis

{lisa etikettide tabelisse uus kirje *E*, *E.etikett*:=*A,*

e-lüli: *E.viit*:=uus lüli *L*;

lüli: *L.aadress*:=*K*; *L.S*:=*s*; kui s=1, siis mine **o2**, muidu mine **järgmine**;}

Kui *E.viit*=Ø, siis mine e-**lüli**;

L:=*E.viit*;

ahel: kui *L.järgmine*≠Ø, siis {*L*:=*L.järgmine*; mine ahel;

L.järgmine:=uus lüli *L*; mine lüli;

o2: Kui *TT.OP2*=0, siis

{kui *R.A2*≠Ø, siis **viga** „real *I* ei või kasutada A2";

mine **järgmine**;}

muidu *s*:=2; *A*:=*R.OP2*; Kui *A*=Ø, siis {**viga „**real *I* puudub A2"; mine **järgmine**;}

mine **aadress**;

Kui *TT.P*=2, siis teisenda *R.A2* kaheksandarvuks x; Kui x>0, siis *K*:=*K*+x,
muidu **viga** „reserveeritud 0 pesa";
mine **järgmine**;

Kui *TT.P*=3, siis teisenda *R.A2* kaheksandarvuks x; *O*:=x; mine **järgmine**;

Kui *TT.P*=4, siis teisenda *R.A1* kaheksandarvuks x ja *R.A2* kaheksandarvuks y;
O.D1:=x; *O*.D2:=y; mine **järgmine**;

Kui *TT.P*=7, siis on 1. läbivaatus lõppenud, objektprogrammi pikkus *M*:=*K* välju programmist.

järgmine: *K*:=*K*+1; *I*:=*I*+1; mine **ring**;

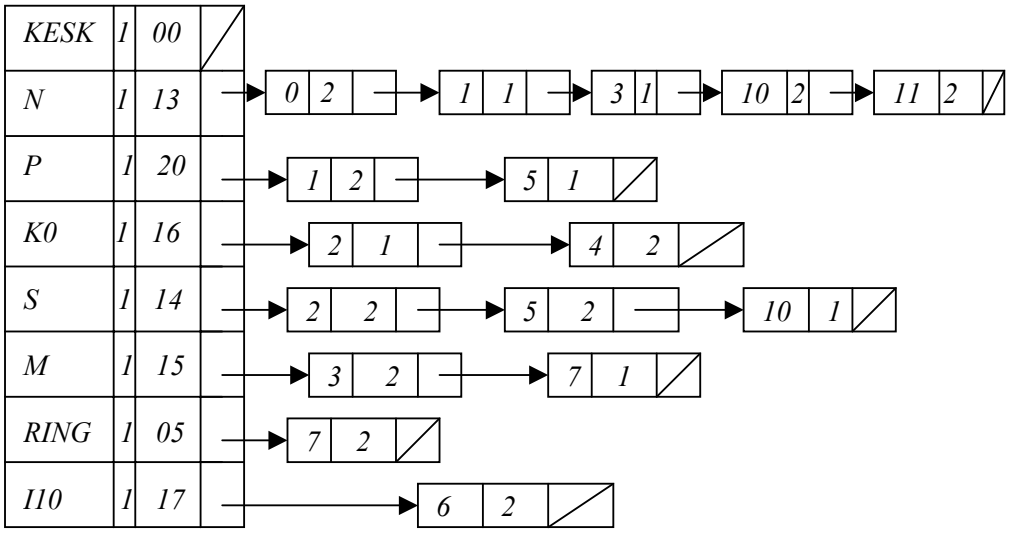
Niisiis, esimese läbivaatuse lõppedes on mälus *vahekeelne objektprogramm* ja etikettide tabel. Meie näiteprogrammi puhul on nood sellised, nagu näha tabelis 2.4.3c ja joonisel 2.4.3a.

Teine läbivaatus „vaatab läbi" etikettide tabeli ning kirjutab käskudesse operandide aadressid (so., etikettide väärtused). Kasutagem juba esimesest läbivaatusest tuttavaid tähistusi. Algoritm on järgmine:

I:=0;
uus: $E:=ET_I$;
Kui *E.V*=0, siis {**viga** „etiketil *E.etikett* pole väärtust"; mine **järgmine**;}
L:=*E.viit*;
lüli: kui *L*=Ø, siis mine **järgmine**;

$O:=OP_{L.aadress}$; Kui *L.S*=1, siis *O.D1*:=*E.väärtus*, muidu *O.D2*:=*E.väärtus*;

L:=*L.järgmine*; mine **lüli**;
järgmine: *I*:=*I*+1; Kui *I*<*J,* siis mine **uus**;
Teise läbivaatuse lõpp.



Joonis 2.4.3a. Etikettide tabel.

aadr.	**kood**	**I**	**op1**	**op2**
00)	03	0	00	00
01)	03	0	00	00
02)	70	0	00	00
03)	70	0	00	00
04)	05	1	00	00
05)	11	1	00	00
06)	06	1	00	00
07)	07	0	00	00
10)	41	0	00	00
11)	04	0	00	00
12)	00	0	00	00
13)	00	0	00	00
14)	00	0	00	00
15)	00	0	00	00
16)	00	0	00	00
17)	00	0	01	00
20)	00	0	00	00

Tabel 2.4.3c. Vahekeelne programm.

Järgneb transleerimisprotokolli trükk ekraanile, paberile või faili ning ─ kui vigu ei olnud ─ siis objektprogrammi kirjutamine kettale.

Protokoll võib välja näha näiteks nii, nagu tabelis 2.4.3.d esitatud.

aadr.	**kood**	**I**	**op1**	**op2**	**etikett**	**kood**	**I**	**OP1**	**OP2**
00)	03	0	00	13	KESK	SISSE1			N
01)	03	0	13	20		SISSEN		N	P
02)	70	0	16	14		SAADA		K0	S
03)	70	0	13	15		SAADA		N	M
04)	05	1	00	16		SAADI	:1		K0
05)	11	1	20	14	RING	PLUSSK	:1	P	S
06)	06	1	17	04		PIND	:1		I10
07)	07	0	15	05		TSÜKK		M	RING
10)	41	0	14	13		JAGAK		S	N
11)	04	0	00	13		TRÜKI1			N
12)	00	0	00	00		STOPP			
13)	00	0	00	00	N	RES			1
14)	00	0	00	00	S	RES			1
15)	00	0	00	00	M	RES			1
16)	00	0	00	00	K0	AK			0
17)	00	0	01	00	I10	IK		1	0
20)	00	0	00	00	P	RES			40

Süntaksivigu ei leitud.

Etikettide tabel

etikett	**väärtus**	**viidad**
I10	17	6
K0	16	2, 4
KESK	00	
M	15	3, 7
N	13	0, 1, 3, 10, 11
P	20	1, 5
RING	05	7
S	14	2, 5, 10

Programm KESK pikkusega 71_8 pesa on kirjutatud kettale.

Tabel 2.4.3d. Transleerimisprotokoll

2.5. Reaalsed masinkoodid

Täpsustagem sissejuhatuseks masinkoodi enda olemust: nimelt eksisteerib mõnede „hilisemate" (alates arvutite kolmandast põlvkonnast) arhitektuuride puhul veel madalam tase kui masinkood ─ see on **mikrokood**.

Refereerigem *Microsoft Press*i väljaannet „*Computer Dictionary*" [11, lk.227 ja 228]: mikrokood on mikrokeele käskude jada, millede abil interpreteeritakse masinkoodi käsku. Võime asja ette kujutada nii, et mikroprogrammis on üldjuhul instruktsioonid käsukoodi tuvastamiseks, suhtaadresside eraldamiseks, indeks- ja baasregistrite poole pöördumiseks, absoluutsete aadresside arvutamiseks ning käsu enda täitmiseks ─ see viimane ei pruugi ka olla tehtav ühe takti abil. Mikroprogrammi kirjutamine protsessori juhtimiseks on **mikroprogrammeerimine**. Üldjuhul „kirjutatakse mikroprogrammid rauda", ent on näiteid (suur- e. *meinfreim*-arvutite ja miniarvutite klassidest ─ viimase näiteks sobib meilgi kasutatud Armeenia päritolu arvuti *Nairi-2*), kus installeeritud protsessor on avatud mikroprogrammide muutmiseks, asendamiseks või lisamiseks.

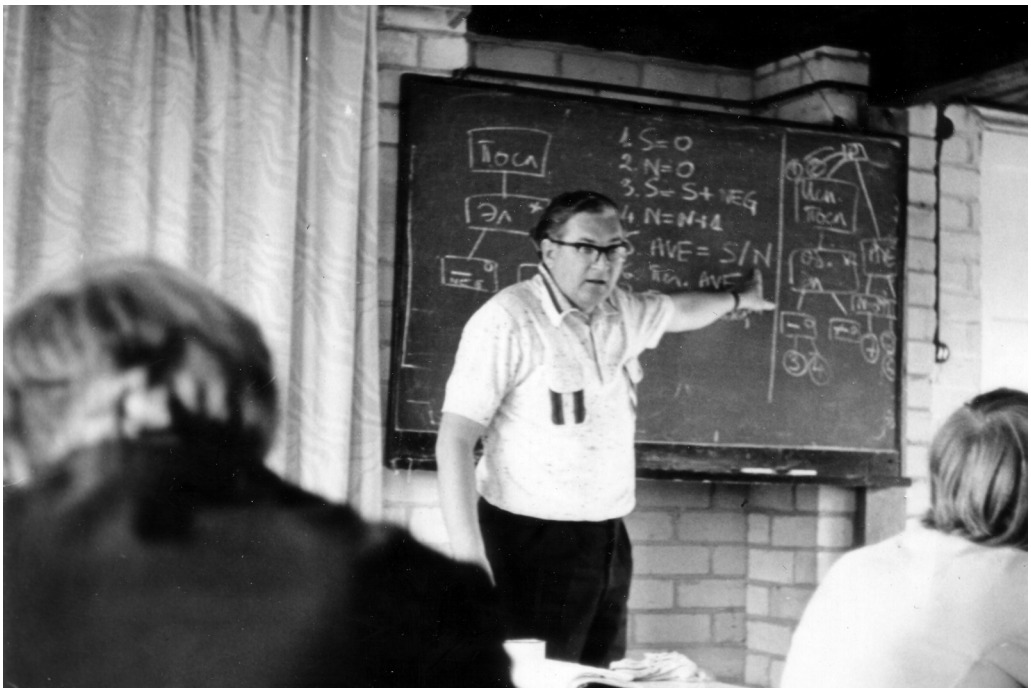
Tänapäevaste *RISC*-tehnoloogia (*Reduced Instruction Set Computer*) protsessorite puhul mikrokoodi tase puudub tänu sellele, et käsu struktuur on võimalikult lihtne. Intuitiivselt tundub *RISC*-tehnoloogia sobivat parimini üheaadressilise protsessori jaoks. Ent ─ naaskem masinkoodi „enda" juurde..

Eelmises peatükis tutvustatud õppearvutid on küll primitiivsed, ent siiski piisavalt representatiivsed, illustreerimaks pesaarvutite arhitektuuri ja võimalusi. Olgu kohe tehtud üks märkus (hiljem käsitleme seda teemaderingi põhjalikumalt): meie masina(te) sisestuskäsk töötab ainult siis, kui sisestatakse kaheksandarve seadmelt, mis ei edasta sümbolkoodi, vaid numbrikoode (0=000, 1=001,…, 7=111). Ja üks asi veel, pesatäis „nulle" stopp-käsuna on ettevaatamatu: üsna mitmel põhjusel on parem, kui käsku koodiga „0" interpreteeritakse käsuna „*no operation*", mis on võrdväärne tingimusteta suunamisega järgmisele käsule.

Demonstreerimaks tegelikke pesamasinate masinkoode vaatleme järgnevas kaheaadressiliste masinate *Razdan-3* ja *Minsk-32* käsustikke. Alustagem *Razdan*ist – see oli masin üpris keerulise (ja võimalusterohke) käsustiku, ent praktiliselt puuduva op-süsteemiga (sh. polnud ka assemblerit).

2.5.1. *Razdan-3*

Selle masina tutvustamisel tugineme Leedus trükitud rotaprindibrošüürile [43]; masin oli konstrueeritud ja toodetud Armeenias. Ainuke Eestisse soetatud masin kuulus Eesti Raadio Arvutuskeskusele ja tolle masina süsteemse tarkvara tegi TPI (nüüdse Tallinna Tehnikaülikooli) Arvutuskeskuse meeskond (juht *Leo Võhandu*, peategijad *Toomas Mikli*, *Mati Tombak*, *Mati Räbovõitra* ja *Jaak Tepandi*, ühena paljudest programmeerijatest ka nende ridade autor).



Joonis 2.5.1a. *Leo Võhandu* 1977. a. Elbi suvekoolis.

Joonis 2.5.1b. *Mati Tombak*, *Ain Isotamm*, *Mati Räbovõitra*, *Jaak Tepandi* (1971)

Masin on kaheaadressiline, töökiirus on 20..25 tuhat tehet sekundis, sõna pikkus on 48 bitti, opereeritakse ujupunktarvudega, mälumaht on kas 16384 või 32768 pesa. Täistsükkel kestab kuni 10 mikrosekundit. Välismäluks on magnetlindid, igale mahub 390 000 või rohkem 56-bitilist sõna. Muud välisseadmed on perfolint- ja perfokaartsisendid, puldikirjutusmasin, laitrükkal (128 sümbolit reas, kiirus 7 rida sekundis), perfolint- ja perfokaartväljundid. Sisestamine toimub autonoomselt, so. samal ajal jatkab aritmeetikaseade oma tööd.

Käsu struktuur on järgmine:

μ_1	μ_2	μ_3	modifikatsioon	kood	δ	indeks	A1	A2
48	47	46	45-43	42-37	36	35-31	30-16	15-1

Aadressruum jagub kahe ploki vahel: esimese aadressid on vahemikus $0..37777_8$ ja teisel $40000..77777_8$. Indeksregistrite jaoks on eraldatud kummastki plokist 16 esimest pesa. Käsus on indekspesa aadressi näitamiseks 5 bitti; kui neil on nullid, siis käsku ei indekseerita, muidu määrab 31. bitt, kas indekspesa on esimese (väärtus 0) või teise ploki (1) alguses ning ülejäänud 4 bitti annavad indekspesa (suht)aadressi ploki alguse suhtes. Nullindat pesa indeksregistrina ei kasutata. Lipp δ=0, kui käsk pole tsüklis, ja 1, kui on. Kolm esimest bitti (μ_1.. μ_3) näitavad, kuidas ja milliseid aadresse tuleb indekseerida. „Indeksregistrite" (jutumärgid vihjavad, et nood pesad on tavalises mälus) formaat on järgmine:

tsükliloendaja	I_1	I_2
48-31	30-16	15-1

Tsükliloendaja vasakpoolseim bitt on märk. Bitid μ_1, μ_2 ja μ_3 määravad indekseerimise (vt. tabel 2.5.1a)

μ_1	**μ_2**	**μ_3**	**A_1**	**A_2**
0	0	0	A_1	A_2
0	0	1	A_1	A_2+I_2
0	1	0	A_1+I_1	A_2
0	1	1	A_1+I_1	A_2+I_2
1	0	0	A_1+I_2	A_2
1	0	1	A_1+I_2	A_2+I_2
1	1	0	A_1+I_1	A_2+I_1
1	1	1	A_1	A_2+I_1

Tabel 2.5.1a. Indekseerimine.

Käskude süsteem on järgmine:

I Aritmeetikatehted arvudega (liitmine, kaks lahutamisvarianti, moodulite lahutamine, korrutamine, jagamine, järkude liitmine ja lahutamine);

II Loogilised tehted ja tehted koodidega (loogiline liitmine, korrutamine ja mittesamaväärtustamine, loogiline nihutamine, tsükliline nihutamine, saatmine, käskude liitmine ja lahutamine, tsükliline liitmine ja lahutamine (viimaseid kahte kasutatakse eeskätt kontrollsummade leidmiseks));
III Juhtimine (tingimusteta suunamine, tingimuslik suunamine, sisend- ja väljundoperatsioonide lõpu kontroll, grupioperatsioon (sisuliselt tsükli juhtimine), peatumine ja suunamine puldilt antud võtme järgi);
IV Muud operatsioonid (infovahetus magnetlindiga, mäluplokkide vahel, perfosisend, väljund trükki ja perfokaartidele).

Käsukoodi modifikatsioon 6 määras muudest (intuitiivselt mõistetavatest) erineva interpretatsiooni: (A1) tõlgendatakse kui kahte mäluaadressi A_1^* (kohad 1..15) ja A_2^* (kohad 16..30). Operatsioon täidetakse eelmise käsu resultaadi (S) ja (A_1^*) vahel ning tolle tehte resultaat salvestatakse aadressile A_2^*. Allpool tutvustame valikuliselt mõningaid käsugruppe; valik on tehtud johtuvalt noist asjust, mida teame juba õppearvutite repertuaarist. Razdan-3 võimalused olid sootuks rikkamad. Aritmeetika- ja loogikatehted töötavad välja 4 tüüpi signaale:

ω=1, kui aritmeetikatehte resultaadi mantissi märk on negatiivne;
υ=1, kui tehte resultaat=0;
γ=1, kui tehte resultaat on negatiivne;
φ=1, kui tehe andis ületäitumise.
Tehete võimaluste illustreerimiseks esitame tabelites 2.5.1b ja 2.5.1c aritmeetikatehete ja suunamiskäskude variandid. *S* tähistab siingi summaatorit (kiire register).

Modif.	**Tegevus**
0	$(A_1)¤(A_2)\rightarrow A_2$, S
1	$(A_1)¤(A_2)\rightarrow$ S
2	$[(A_1)¤(S)]+(A_2)\rightarrow A_2$, S
3	$[(A_1)¤(S)]+(A_2)\rightarrow$ S
4	$(A_1)¤(S)\rightarrow A_2$, S
5	$(A_1)¤(S)\rightarrow$S; mine A_2, kui ω või γ =1
6	$(A_1^*)¤(S)\rightarrow A_2^*$, S; mine A_2, kui δ=1 ja tsükliloendaja>0
7	$(A_1)¤(S)\rightarrow$S; mine A_2, kui ω või γ =0

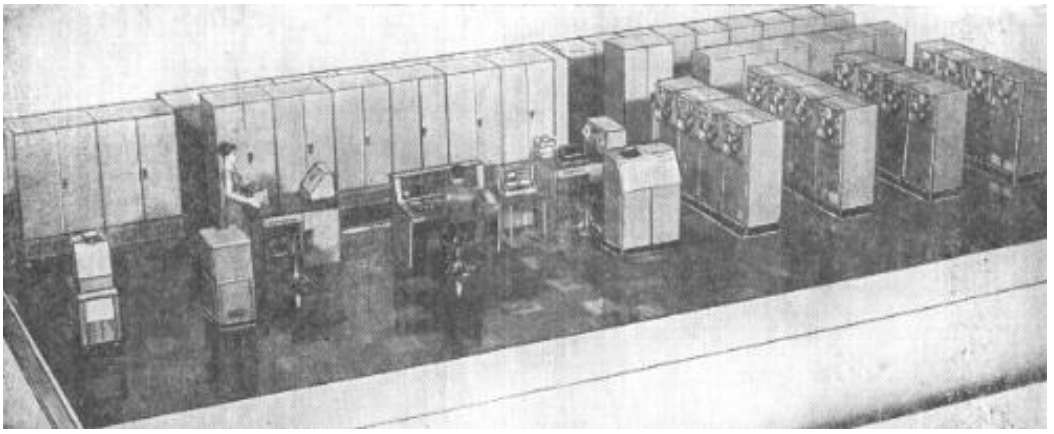
Tabel 2.5.1b. Aritmeetikatehete variandid.

Modif.	**Tegevus**
0	Mine tingimusteta A_2 ilma tagasipöördumiseta
1	Mine tingimusteta A_2 koos tagasipöördumisega
2	Mine A_2 koos tagasipöördumisega, kui ω=1
3	Mine A_2 koos tagasipöördumisega, kui ω=0
4	Mine A_2 koos tagasipöördumisega, kui γ=1
5	Mine A_2 koos tagasipöördumisega, kui γ=0
6	Mine A_2 koos tagasipöördumisega, kui υ=1
7	Mine A_2 koos tagasipöördumisega, kui υ=0

Tabel 2.5.1c. Suunamise variandid.

Tagasipöördumiseks kasutatakse suunamiskäskudes aadressi A_1, kuhu tuleb kirjutada selle käsu aadress, kuhu suunamiskäsk aadressil *k* kirjutab käsu „mine *k+1*". Mõistagi tuleb programmis aadressile (A_1) kodeerida 0, ja hoolitseda, et $A_1 > A_2$.

Juhime lugeja tähelepanu seigale, et *Razdan-3* käskude repertuaaris puudub alamprogrammi poole pöördumise käsk. Ühe tükina kirjutatud programmi korduvalt kasutatavate lõikude (näiteks, mingite funktsioonide väärtuste arvutamiseks, sisestamiseks ja sisestatu teisendamiseks vms) jaoks on hea vahend naasmisega (tagasipöördumisega) suunamiskäsud. Ent vähegi suuremate tööde programmeerimisel on vältimatult vajalik võimaldada eraldi kirjutatud ja silutud programmide (alamprogrammide) kasutamist, mis eeldab esiteks komplekteerimisprogrammi olemasolu ja teiseks, kokkuleppeid alamprogrammide vormistamiseks, parameetrite edastamiseks ning alamprogrammi pöördumiseks ning naasmiseks. Neid vahendeid tolle masina tehaseversioon üldistatud kujul ei pakkunud (Eesti Raadio Arvutuskeskuse masinale kirjutas nood *Toomas Mikli*)[1], selle



Joonis 2.5.1c.*Razdan-3* (vt. [83])

asemel oli spetsiaalne „interpretaator", mille poole sai programmi täitmise ajal pöörduda ning mis paigaldas välismälust („süsteemilindilt") alamprogramme, hoolitses nende täitmise eest ning vabastas vajadusel nende alt mälu. Süsteemilindil oli ca 40 moodulit, näiteks teisendused 10→2 ja 2→10, trigonomeetrilised funktsioonid, ruutjuure leidja, perfoväljund ja trükkija, maatriksite korrutaja jmt.

[1] Põhimõtteliselt toimis järgmine „mehhanism": meeskonnal oli paks kaustik, kuhu pandi kirja kõik moodulid: identifikaator (4-kohaline kaheksandarv), nimi (meeskonnasiseseks suhtlemiseks), autor, sisendi ja väljundi spetsifikatsioon (sh. sisendparameetrite pesade arv ja pakkimisskeem) ning võimalikult täpne (informaalne) algoritmi kirjeldus. Üks naasmisega suunamiskäsu modifikatsioonidest reserveeriti alamprogrammi pöördumiseks (seda ei võinud „tavakohaselt" kasutada; programmis kirjutati A1=mooduli identifikaator ja A2=sisendparameetrite arv. *Mikli* komplekteerija jaoks oli süsteemilindil moodulite info (identifikaator ja pikkus) ning komplekteerimise käigus asendati identifikaator mooduli sisendpunkti aadressiga (+ sisendparameetrite arv) komplekteeritud programmis ning mooduli pikkuse järgi kirjutati õigesse kohta naasmisaadress. Mõistagi, mooduli viimane käsk pidi olema tühidirektiiv tollele naasmisaadressile suunamiseks vajaliku käsu kirjutamiseks.

Joonis 2.5.1d.. *Marika* ja *Toomas Mikli* suvel 1970, Siberis, Keti jõe ääres.

Tolle „interpretaatori" kasutamiseks tuli täita terve hulk nõudmisi. Näiteks, põhiprogrammis ei tohtinud kasutada pesi aadressidega 00020..0042, 321..325 ja 260..325 ─ nood olid „interpretaatori" tööväljad.

Kuivõrd **tsükli juhtimine** erineb üsna palju õppearvutist tuntust, siis vaadelgem, kuidas see teema on kaetud *Razdan-3* puhul. Tsüklikäsu kood on 32 ja selgi on 8 modifikatsiooni. Nagu juba öeldud, võib indeksregistri rollis kasutada tavalist mälupesa ploki algusest (selle aadress on käsus A_1); käsk ise võib paikneda nii tsükli ees kui ka taga (tagapool tuletame seda seika meelde *for*- ja *while*-tsüklite juures). Esimesel juhul tuleb tsükliloendajaks L kirjutada n-1, teisel aga n (n on korduste arv). Igal tsüklisammul vähendatakse tsükliloendajat L 1 võrra ja sõltuvalt tolle tehte resultaadist kas jätkatakse tsükli täitmist või ei. Samuti modifitseeritakse igal sammul indekseid ühe võrra.

Modif.	**Tegevus**
0	Kui $L \geq 0$, siis I_1+1, I_2+1 ja mine A_2
1	Kui $L < 0$, siis I_1+1, I_2+1 ja mine A_2
2	Kui $L \geq 0$, siis I_1 -1, I_2-1 ja mine A_2
3	Kui $L < 0$, siis I_1-1, I_2 -1 ja mine A_2
4	Kui $L \geq 0$, siis I_1+1 ja mine A_2
5	Kui $L < 0$, siis I_1+1 ja mine A_2
6	Kui $L \geq 0$, siis I_2+1 ja mine A_2
7	Kui $L < 0$, siis I_2+1 ja mine A_2

Tabel 2.5.1d. Tsüklikäsu variandid.

Vaadelgem veel, milliseid võimalusi pakub protsessor sisestamiseks perfokandjatelt. Kõikide modifikatsioonide korral A_1 on sisestatavate andmete algusaadress ning teist aadressi A_2 kasutab ainult modifikatsioon 1: see on aadress, kuhu antakse juhtimine vigade korral. Variandid on järgmised (kood on 36):

Modif.	Tegevus
0	Perfolindi tagasikerimine (*reverse*)
1	Kui viga, siis mine A_2
2	8-ndarvude või kahend-kümnendarvude sisestamine perfolindilt
3	Sümbolkujul arvude sisestamine perfolindilt koos teisendamisega sisekujule
4	Sümbolite sisestamine perfolindilt
5	Kahendarvude sisestamine perfokaartidelt
6	Kümnendarvude sisestamine perfokaartidelt
7	Sümbolite sisestamine perfokaartidelt

Tabel 2.5.1e. Perfosisendi juhtimine.

Razdan-3 oli mõneti erandlik arvuti: esiteks, valmistajatehas (ega keegi muugi) ei varustanud teda *assembleriga* (ainus programmeerimiskeel oli masinkood), teiseks ─ nagu võisime juba usutavasti veenduda ─ oli ta masinkood ootamatult võimalusterohke ja „kaval".

Tehkem mõningaid järeldusi *Razdan-3* protsessori kohta; seni on meil ainuke võrdlusbaas õppearvuti.

1. Käskude struktuur ja repertuaar on oluliselt keerulisem ja rikkam;
   ainus „kiire register" on summaator; indeksregistritena kasutatakse operatiivmälu plokkide algusvälju, baseerimist süsteem ei toeta (mingi kauge analoog on kahe mäluploki aparatuurne toetus);
2. indekseerimine ja tsükli juhtimine on oluliselt nüansirikkamad;
3. *Razdan* annab mingeid vihjeid programmi struktureerimiseks (naasmisega suunamised, „interpreteerimismoodul"), õppearvutile seda probleemi ei püstitatud;
4. Õppearvuti sisend-väljundkäsud olid spetsifitseerimata (abstraktsed), tegelik arvuti on hoopis konkreetsem ja keerulisem;
5. Kumbki versioon ei toeta sümbolitöötlust; *Razdan*i trükimoodulis on minimaalvõimalused olemas (me ei tutvustanud käsku koodiga 76 ─ suunamine puhvrist trükki ─ , kus 3. modifikatsioon on teksti väljastamiseks, igal sümbolil on 6-bitine kood ja järelikult mahub ühte pessa 8 sümbolit;
6. Kumbki versioon ei anna mingeid vahendeid mälujaotuseks ega vaba mälu kontrolliks.
7. On toiminguid, mille jaoks pole masinkoodis interpretatsiooni, ja nood valdkonnad kaetakse *kokkulepetega*. Täpsemalt ja rohkem sellest võib lugeda näideteks valitud masinate tutvustusest.

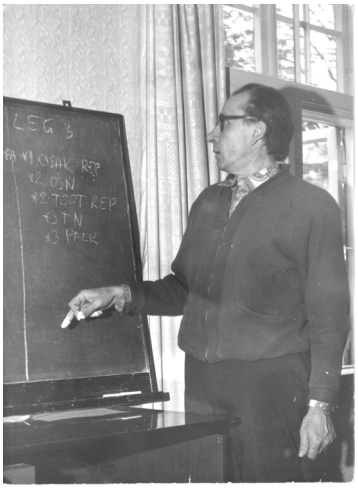
Järgmise masinana vaatleme arvutit *Minsk-32*, mille käskude süsteemil oli tuntav operatsioonisüsteemi tugi, alamprogrammide ilmutatud toetus ning assemblerkeel.

2.5.2. *Minsk-32*

2.5.2.1. Üldandmed

Tsiteerigem *Ü. Kaasikut*[1] jt [26]: ”Universaalne elektronarvuti „*Minsk-32*” (konstrueeritud 1968. a.) on sobiv nii teaduslik-tehniliste, majanduslike kui ka statistiliste ülesannete lahendamiseks”. See mudel oli TRÜ Arvutuskeskuse uue maja esimene arvuti (eelmised olid eelmistes majades *Ural-1* ja *Ural-4*, milledele programmeeriti masinkoodis), masin oli valdavalt kaheaadressiline (mõnede käsurühmade jaoks üheaadressiline) ja programmeerimiskeeleks oli *Minsk-32* assembler ЯСК (Язык Символического Кодирования). Jaotises tugineme lisaks juba tsiteeritud allikale ka *K. S. Nesluhhovski* [40] ning *Mark Nemenman*i (kes oli selle masina peaideoloog ja -konstruktor) jt [6] raamatutele.

Meie kontekstis esindab *Minsk-32* Eesti jaoks esimest (mis siis, et lihtsa) operatsioonisüsteemi ja assembleriga masinat.



Ülo Kaasik Vellaveres, 1978. a.

Ainus „kiire register” oli summaator. Kaheaadressilise käsu struktuur on järgmine:

Kood (0..6)	**Ip** (7..10)	**B1** (11..12)	**D1** (13..23)	**B2** (24..25)	**D2** (26..36)

Sulgudes on bitirajad. Väljade semantika on selline: **Ip** on indekspesa järjekorranumber (vahemikust 0..15), **B1** ja **B2** on vastavalt esimese (**D1**) ja teise aadressi (D**2**) baasaadressid. Niisiis, kuivõrd suhtaadresside väärtusvaru on 0..2047, siis baasaadressid ja indekspesade sisud on mõeldud võimaldamaks adresseerimist üle kogu operatiivmälu. *Baasaadressid* on 16-bitised (võimalikud väärtused $0..2^{16}$=64K)

[1] *Ülo Kaasik* rajas (koos *Leo Võhandu*ga) Ülikooli arvutuskeskuse ja on jatkuvalt üks pädevamaid IT-valdkonna autoriteete.

Indekspesa formaat on järgmine:

0..4 vaba **Δ1** (5..20) **Δ2** (21..36)

Mark Nemenman, „Minsk-32" projekti juht *(M.N.* erakogust).

Üheaadressilise käsu struktuur:

Kood (0..6)	**Ip** (7..10)	**Määraja** (11..23)	$\mathbf{B_{Ip}}$ (24..25)	**D2** (26..36)

„Määraja" (определитель) täiendab käsu koodi; üheaadressilised on näiteks indekspesadesse kirjutamise ja nende sisu modifitseerimise käsud (**Ip** osutab indekspesa numbrile ja $\mathbf{B_{Ip}}$ on indeksvälja baasaadress), välisseadmete käsud jmt.

Minsk-32 oli loogiline järg masinatele *Minsk-2* , *Minsk-22* ja *Minsk-22M*, milledest ükski polnud kvalitatiivselt üle *Razdan*ist (minimaalne opsüsteem, siiski ─ *Minsk-22* standardvarustuses olid *Malgol*-translaator, ent assemblerit ikka veel polnud). Uus mudel oli uus sõna kolme seiga poolest: esiteks, tal oli üpris arenenud *operatsioonisüsteem* („Dispetšer" [6]), teiseks, aparatuurselt toetati *tekstitöötlust*, ja kolmandaks, programmeerimissüsteem oli orienteeritud *moodulprintsiibile.*

Too viimane printsiip võib oluliselt võimenduda, kui süsteem toetab *dünaamilist mälujaotust*, so. võimalust masinkoodi- või assemblerprogrammis küsida, jaotada ja kasutada vaba mälu. *Minsk-32* käskude süsteemis oli assembler-direktiiv ПАУП[1] , mille abil sai kätte aadressi, mis viitas pesale vaba mälu algus- ja lõpuaadressidega, ja nendega oli võimalik manipuleerida. Näite esitame jaotises 3.7.

[1] Переслать Адрес Управления Памятью (saata mälujuhtimise aadress).

2.5.2.2. Dispetšer

Dispetšer kui operatsioonisüsteem toimib paljude kokkulepete tõttu. Vaadelgem neid põgusalt.

Esiteks, pesad aadressidega 000000 kuni 000377_8 on reserveeritud **tasemete juhtpesadele,** numbritega 0..13. Igale tasemele on reserveeritud 8 masinsõna ja tasemete töödjaotus on (näiteks ja valikuliselt) järgmine: tasemed 0..2 esitavad tõrgete töötlemise programmide infot, 3..5 sidet välisseadmetega, 6 ─ nn. ekstrakoodide töötlejat, $7..12_8$ kasutajaprogrammide baasaadresse ning 13_8 ─ residentse paigaldusprogrammi parameetreid. Tasemetel 0..6 on kõrgem prioriteet kui teistel. Taseme juhtpesa aadressist algab 8-pesaline väli järgmise informatsiooniga:

0) kohad 5..20 ja 21..36 **baasaadress 1** ja **baasaadress 0**;
1) kohad 5..20 ja 21..36 **baasaadress 2** ja **baasaadress 3**;
2) tühi (so., reserveeritud täitmisaegsele informatsioonile);
3) 5..11 **vaba mälu** absoluutne algus- ja 21..27 lõppaadress täpsusega 512 pesa;
4) 21..32 **indeksvälja** baasaadress (16-kordne);
5) 21..36 (alam)programmi **baasaadress,** ent katkestuse korral ─ **jatkuaadress**; 13..18 katkestuse momendi **indikaatorid**, 5..11 ─ aritmeetikaseadme tõrgete **lipud. Lipud** signaliseerivad alamprogrammi täitmise lõpetamisest, ületäitumisest, aktsepteerimatust käsukoodist (juhtimine sattus tõenäoliselt väljapoole programmi), aritmeetikatehete lubamatust operandist, adresseerimisest väljapoole programmile lubatud mäluruumist ja paarist aparatuursest (riistvara-)veast. **Indikaatorid** võimaldavad blokeerida ümardamist, normaliseerimist ja katkestusi, teatada protsessori töötamisest *Minsk-22* režiimis jmt.
6) katkestatud programmi **jooksvad resultaadid** (eeskätt summaatori seis);
7) 14..17: **katkestatud programmi** taseme number.

Iga taseme juhtpesaga võib olla seotud *programm*, seega võib töövalmis olla neid korraga 12, ning juhtimise üleandmine võib toimuda kas programselt (pöördumine alamprogrammi), tasemete vahetamise erikäskudega või katkestussignaalide toimel. Viimaseid oli üpris palju, näiteks andsid neid aritmeetikaseadme tõrked (ületäitumine jmt.), välisseadmete tõrked jne.

Dispetšeri seisukohalt jagunevad programmid *süsteemiprogrammideks* (dispetšer ise ja teiste programmide käivitamiseks ja teenindamiseks vajalikud programmid, näiteks translaatorid, siluja, sisend-väljundprogrammid jmt.) ─ need asuvad *süsteemi(magnet)lindil* ja sisestatakse, paigaldatakse ning käivitatakse vastavalt vajadusele, so. siis, kui neid vajab *kasutajaprogramm* (võimalik, et dispetšeri initsiatiivil).

2.5.2.3. Tekstitöötlus

Enam-vähem kõik maailma arvutid olid 60-ndate aastate lõpuni konstrueeritud *arvutamiseks*, so. teadusarvutusteks. Pesamasinad sobisid selleks otstarbeks väga hästi, pesa oli piisavalt „pikk" kujutamaks nii *integer*- kui ka *real*-tüüpi arve; kahe- ja neljapesaliste arvude kasutamiseks tuli vaid lisada nondega opereerimise käsud.

Kasutusala esimeseks laienduseks sai raamatupidamine ja selle aruandlus, aga ka (tänapäevases vaates primitiivsed) info-otsisüsteemid. See tingis tekstitöötlusvahendite arendamise, nii protsessori kui ka välisseadmete aspektist.

Minsk-32 kasutas sümbolite esitamiseks seitsmebitilist koodi ГОСТ-10859-64; pessa mahtus viis sümbolit (infovahetusel välisseadmetega kasutati 8-bitilist koodi: iga sümboli vasakpoolseim bitt oli info korrektsust kontrolliv nn. paarsusbitt). Sümbolid järjekorranumbritega 0..4 paiknesid 37-bitilises pesas kohtadel 0..34, bitte 35 ja 36 ei kasutatud.

Pikemate tekstide sümbolite adresseerimist toetati aparatuurselt; indekspesa formaat sümbolkäskude jaoks oli järgmine:

0..17 ei kasutatud	18.20 sümboli nr. pesas	21..36 pesa suhtaadress

Aparatuurne toetus seisnes selles, et „sümboli nr." sai muutuda piires 0..4, ja kui seda suurendada nii, et ta väärtus ületab ülapiiri, siis korrigeeriti nii „pesa suhtaadressi" kui ka „sümboli numbrit".

2.5.2.4. Moodulite toetus

Minsk-32 programmeerimissüsteem oli orienteeritud *moodulprintsiibile*, so. toetas eraldi kirjutatud ja transleeritud assemblerkeelsete moodulite süsteemi. Kõik dispetšeri „süsteemilindi" standardprogrammid olid moodulid, mida sai kasutajaprogrammist vajadusel välja kutsuda, nagu ka kasutaja enda kirjutatud mooduleidki. Mõistagi pidi operatsioonisüsteem toetama moodulitest koosneva programmi komplekteerimist ja paigaldamist, aga ka infovahetust moodulite vahel. Seetõttu sisaldas selle arvuti masinkood palju erinevaid *pseudokoode*, mis juhivad komplekteerija (Сборщик) ja paigaldaja (Загрущик) tööd.

Kui (alam)programm kirjutatakse masinkoodis, siis tuleb ta *päis* maketina kirjutada käsitsi, kui assembleris, siis teeb seda translaator. Paigaldatud alamprogrammi *päis* oli järgmise struktuuriga:

Pesa 0: päise pikkus;
Pesa 1: alamprogrammi nimi (5 sümbolit);
Pesa 2: indeksvälja pikkus ja algus-suhtaadress (teeb komplekteerija) ja töövälja pikkus ja algus-suhtaadress (teeb komplekteerija):
Pesa 3: indekspesade arv, segmendi (mooduli kõigi piirkondade) pikkus ja piirkondade arv *n*;

Iga piirkonna kohta, alates *4. pesast*; 2 pesa: piirkonna pikkus ja nimi;

Paigaldaja kirjutab *järgnevatesse* pesadesse iga kasutada tahetava taseme (0. tase oli programm ise) välja pikkuse ja absoluutsed algus- (so, baas-)aadressid.

Toogem ainult paar **pseudokoodide** näidet.

- Alamprogrammi *kirjeldus*: kood -060, 27. bitt komplekteerija käsutuses (0: pole liidetud, 1: on liidetud), enne komplekteerimist oli pesas alamprogrammi nimi (5 sümbolit), pärast komplekteerija tööd oli bittidel 28..36 alamprogrammi suhtaadress;
- *Aadresskonstant*: kood 000, kohtadel A_1 ja A_2 olevad suhtaadressid asendab paigaldaja absoluutsetega (piirkonna baasaadress+suhtaadress piirkonnas).

Alamprogrammide tugi on tagatud *eribaseerimisega* käskudega; nende puhul moodustatakse operandide absoluutsed aadressid järgmiselt: esimese operandi aadress määratakse tavalises korras:

A1=D1+B1+Δ1

A1-l on kas aadresskonstant või pesa alamprogrammi 0. ja 1. baasaadressiga. Ning seda kasutatakse teise operandi baseerimiseks.
Eribaseeerimisega käske on neli:

- Lugemine eribaseerimisega (-05, assembleris ЧСП);
- Kirjutamine eribaseerimisega (-06; ЗСП);
- Alamprogrammi pöördumine (31; ИП);
- Alamprogrammist väljumine (-21; ВЫХ).

Vaadelgem neist käskudest kaht viimast; vältimaks liigseid masinkoodi-nüansse, kasutagem lihtsustuseks ka assembleri vastavaid direktiive.

Kõik alamprogrammid peavad olema kirjeldatud kahel assemblerireal, näiteks:

```
ОППР  ОПР СУМ
        НОП
```

Kood ОПР on alamprogrammi kirjeldamise pseudodirektiiv, СУМ on alamprogrammi tegelik nimi ning etiketti ОППР kasutatakse alamprogrammi pöördumiseks (Mõistagi võis kirjutada ka СУМ ОПР СУМ). НОП on tühidirektiiv, millele vastab masinkoodis 0-dega täidetud pesa; sinna kirjutab paigaldaja alamprogrammi СУМ kaks baasaadressi. Selle alamprogrammi poole pöördutakse näiteks nii:

```
ИП  ОППР;2
КА  АМАС;СУММА
```

Teise aadressi kohal näidatakse parameetrite arv (≥0) ja järgmis(t)el real/ridadel paiknevad parameetrid ise (mis meie näites on antud aadresskonstandina). Käsk ИП täidetakse järgmiselt:

- Formeeritakse absoluutne aadress A1 (kasutades etiketti ОППР) ning sellel aadressil oleva arvu (A1) kahendkohtadel 21..36 on alamprogrammi baasaadress (baas 0) ─ A_P.

- Assembleris vormistatakse alamprogrammi algus järgmiselt:

```
        ЗАГЛ  pealkiri (kommentaar)
        БАЗ   0
СУМ     РЗВ   3
П1      РЗВ   1
        РИП   16
        СУ    4;+16
```

  „РЗВ" on mälu reserveerimise pseudodirektiiv; 3 esimest reserveeritud pesa on kohustuslikud, järgneb vajalik arv reserveerimisi sisendparameetrite jaoks. Pseudodirektiivid "РИП" ja "СУ" on indeksvälja moodustamiseks ja adresseerimise tagamiseks.

  Esimesse reserveeritud pessa aadressiga (A_P) kirjutatakse *käsuloendaja seis*+1 so. alamprogrammi pöördumise käsule järgneva pesa aadress ning selle pesa esimesse poolde lippude ja indikaatorite seis (sh. lipp „hõivatud", mis välistas sellesse alamprogrammi sisenemise enne ta töö lõppu; seega polnud võimalik kasutada rekursiooni[1]).
- Kahte ülejäänud reserveeritud pessa kirjutatakse alamprogrammi pöördunud programmi taseme juhtpesade (numbritega 0..3) sisud.
- Alamprogrammi kirjeldava pseudodirektiivi ning talle järgneva pesa sisud kantakse taseme 0. ja 1. pessa.
- Vajadusel kantakse üle alamprogrammi parameetrid.
- Modifitseeritakse käsuloendaja seis adresseerimaks alamprogrammi esimest täitmisele kuuluvat käsku.

Alamprogrammist väljudes, näiteks täitmaks käsku «ВЫХ СУМ;2» tehakse järgmised tööd:

- Esimene aadress on programmi keha alguse etikett, teine aga sisendparameetrite arv. Kasutades eribaseerimist formeeritakse naasmisaadress, millele liidetakse parameetrite arv
- Taastatakse ülemise taseme lippude ja indikaatorite seisud ning taseme juhtpesade seisud.
- Juhtimine antakse sisendparameetrite arvuga korrigeeritud naasmisaadressile.

Nentigem, et alamprogrammi pöördumise ja sellest väljumise käske interpreteerivad mikroprogrammid pidid olema üsnagi mahukad.

Niisiis, infovahetus alamprogrammide vahel võib toimuda parameetrite abil. Teine võimalus on ühisväljade kasutamine, selleks peavad alamprogrammid deklareerima samanimelised ühisväljad ja ühiskasutatavad muutujad (või konstandid) peavad paiknema sama-

[1] Põhjus peaks olema läbinähtav: komplekteerija ja paigaldaja tekitasid staatilise struktuuri alamprogrammide jaoks (mis seejuures ei välistanud alamprogrammide objektide jaoks dünaamilist mälukasutust). Seega sai iga alamprogrammi süsteemne informatsioon (väljad ülemise taseme alamprogrammide baasaadresside ja indikaatorite säilitamiseks, naasmisaadressile jmt.) olla vaid *ühes eksemplaris* ja ettenähtud kohas.

del suhtaadressidel ja on üldjuhul sama tüüpi; nende *nimed* ei pruugi olla eri moodulites ühesugused.

2.5.2.5. Käskude süsteemist

Selles jaotises kasutatakse *Ü. Kaasik*u jt. [26] materjale. Kuivõrd assembler on masinkoodiga peaaegu üksüheses vastavuses, ent sootuks paremini loetav, siis kasutame lühiülevaate andmiseks just assemblerit (CCK).

Käske võib grupeerida järgmiselt:

- Tehted fikseeritud koma arvudega.
- Tehtud liikuva koma (ujukoma-) arvudega.
- Tehted kümnendarvudega ja loogilised tehted.
- Salvestamiskäsud.
- Suunamiskäsud.
- Tehted indekspesades paiknevate arvudega.
- Tehted baasaadressi muutmisega (eribaseerimine).
- Tehted juhtpesades paiknevate arvudega.
- Abitehted ja ekstrakoodid.

Aritmeetika ja loogika.

Enamikul aritmeetika- ja loogikatehetel on neli modifikatsiooni (*xy* on mnemokoodi prefiks, A1 ja A2 operandide absoluutsed aadressid (suhtaadress+baasaadress+(vajadusel) indeks, ¤ on tehe ning *S* on summaator):

1. x3: (A2)¤(A1)→A2, S
2. x (A2)¤(A1)→ S
3. xB (S)¤(A1)→A2, S
4. xP (S)¤(A1)→S

Aritmeetika- ja loogikatehete *prefiksite* komponent *x* määrab tehte:

- С ─ liitmine;
- В ─ lahutamine;
- ВМ ─ arvude moodulite lahutamine;
- У ─ korrutamine;
- Д ─ jagamine.
- АСД ─ aritmeetiline nihutamine;
- ЛС ─ loogiline liitmine;
- ЛУ ─ loogiline korrutamine;
- ПС ─ loogiline mittesamaväärsus;
- ЛСД ─ loogiline nihutamine.

Prefiksikomponent *y* osutab arvu tüübile, kui tegemist on *aritmeetikatehtega* (nihutamis- ja loogikatehete korral see komponent puudub):

- Ф ─ fikskoma-arvud;
- П ─ ujukoma-arvud.

Lisaks on mõned spetsiifilised (aritmeetika)käsud:

- УЦ ─ täisarvude korrutamine: (A2)•(A1)→S (tulemusest säilib 36 viimast kahendkohta);
- ДФО ─ fikskoma-arvude jagamine jäägi leidmiseks (viimane kantakse summaatorisse);
- СЦКЛ ─ kontroll-liitmine, kasutati *kontrollsumma* leidmiseks: (A1)+(A2)→A2, S;
- СЕ ─ ühtede arvu leidmine: (A1) 1-väärtusega bittide arv kantakse A2 ja S;

Kümnendarvude vahel saab sooritada liitmis-, lahutamis- ja korrutamistehteid (СФДР, ВФДР ja УФДР) formaadiga (S)¤(A1)→S. See arvutüüp oli mõeldud raamatupidamisrakenduste programmeerimiseks: jagamistehet selles valdkonnas ei kasutata (majandustegevuse analüüs pole raamatupidamise komponent, raamatupidamine annab analüüsile lähteandmeid).

Lugemine ja salvestamine.

Esitame allpool valiku selle rühma käskudest.

- Ч ─ lugemine: (A2)→ S;
- З ─ kirjutamine: (S)→ A2;
- П ─ saatmine: (A1)→ A2,S;
- О ─ vahetamine: (A2)→ A1, (A1)→A2,S;
- ГРУП ─ tsükliloendajasse salvestamine: (A2)→ loendaja (süsteemne pesa). Järgmisel real asuvat direktiivi täideti ГРУПi argumendiga määratud arv kordi;
- ИЧ ─ suunamine koos lugemisega: mine A1, (A2)→ S;
- ИЗ ─ suunamine koos salvestamisega: mine A1, (S)→ A2;

Suunamine.

Minsk-32 ei paku siin põhiosas midagi uut: repertuaari kuuluvad kõik tuntud tingimuskoodile põhinevad käsud: märgi järgi suunamine (ИЗН), plussi järgi suunamine (ИПЛ), miinuse järgi suunamine (ИМН), nulli ja mitte-nulli järgi suunamised (ИН ja ИНРН), mittevõrdumise järgi suunamine (ИНС), loendaja järgi suunamine (ИС ─ (A2) tsükliloendaja, A1 tsükli algusaadress) ning ületäitumise järgi suunamised (ИПЕР0 ja ИПЕР1). Tingimusteta suunamiseks saab kasutada suvalist kaheaadressilist suunamiskäsku, kirjutades *A1=A2*.

Meie kontekstis on uued suunamine alamprogrammi (ИП) ja väljumine alamprogrammist (ВЫХ).

Tehted indekspesades ja juhtpesades paiknevate arvudega, abitehted ja ekstrakoodid.

Indekspesadega manipuleerimiseks olid *Minsk-32*l enam-vähem täiuslik komplekt vahendeid kaheaadressilise pesamasina jaoks: indekspesaga liitmine, lahutamine ja korrutamine, loogiline liitmine ja korrutamine, järgukaupa võrdlemine, indekspessa saatmine aadressilt või salvestamine summaatorist, indekspesast saatmine aadressile, indekspesa lugemine ning neli käsku operandi aadressosa salvestamiseks indekspessa; selle grupi näitena esitame need neli (üldkujuga *kood* :I; A1) siinkohal:

- ПА1И1 ─ Δ1(A1) → I(5..20);
- ПА2И1 ─ Δ2(A1) → I(5..20);
- ПА2И2 ─ Δ2(A1) → I(21..36);
- ПА1И2 ─ Δ1(A1) → I(21..36);

Juhtpesades olevate arvudega sai teha nii aritmeetilisi kui ka loogilisi tehteid, juhtpesi sai lugeda ja neisse kirjutada, ent kuivõrd need arhitektuurielemendid olid omased (vähemalt meie kontekstis) vaid *Minsk-32*le, siis me neil pikemalt ei peatu.

Abitehetega sai määrata mõnede indikaatorite seisu (näiteks, blokeerida ümardamist või normaliseerimist); kirjutada tühikäsu koodiga 00 (НОП; juhtimine antakse üle järgmisele käsule, signaalid ja summaatori seis säilivad) jmt.

Ekstrakoodide interpreteerimine tekitas kasutajaprogrammi katkestuse, juhtimise üleandmise operatsioonisüsteemile (dispetšerile) ja ─ kui lubatud ─ kasutajaprogrammi jatkamise pärast katkestuse töötlemist. Näiteks, ekstrakoodid -65 ja -67 olid seotud välisseadmete tööga.

Sisend ja väljund.

Sisendit ja väljundit juhtis arvuti *vahetusseade,* mis „organiseerib programmide katkestamisega seotud ümberlülitamisi. Vahetusseadmes on peale juhtsõnaregistri veel eraldi register automaat-katkestuse põhjuste salvestamiseks. Nendeks põhjusteks võib olla: ekstrakoodi ilmumine; arvuti, vahetusseadme või mäluseadme viga; välisseadme töö lõpp. Mingi põhjuse esinemine nendest toob endaga kaasa programmi täitmise automaatse katkestamise ja juhtimise üleandmise HALDURI (meil *dispetšeri* ─ A.I.) vastavale osale.” [26, lk. 6]

Ainuke välisseade, mida on arvutil parasjagu üks, on operaatori *juhtimispult* e. kirjutusmasin, sisend-väljundseade infovahetuseks masinaga. Ülejäänuid võis olla rohkem kui üks, ja need olid *perfolint-* ja *perfokaartsisend* ja *-väljund*, *magnetlintseade* ja *magnet-*

trummel (nii lugemiseks kui ka kirjutamiseks*), teletaip* ja *laitrükkal* (mõlemad väljastamiseks).

Käskude repertuaaris olid kõik loomulikult vajalikud käsud infovahetuseks. Nende hulgas olid käsud välisseadme identifitseerimiseks (КНВУ V,K :<tüüp; nr>, näiteks osutamaks trükkalile numbriga 1 tuli kirjutada КНВУ ПЧ;1), kinnistamiseks ja vabastamiseks (näit. ЗАКР ПЧ;1 ja ОСВ ПЧ;1), välisseadme töö lõpu ootamiseks (Ж :I;V,K) ja operaatori vastuse ootamiseks (ЖОО :I;V,K).

Enamus vahetuskäske olid kaherealised: esimesel määrati tegevus, teisel aga direktiiv juhtsõna moodustamiseks, viimaseid võimalusi oli 8 ja kõigi nende direktiivide mnemokoodi prefiks oli КОС (vahetuskonstant sõna või sümbolini ─ до Слова/Символа).

Tegevusi määravates vahetusdirektiividest toogem ainult paar näidet. Kõigil noil direktiividel on ühesugune formaat:

Kood :I;VL;V,K

VL on vahetusoperatsiooni liik, *V* ja *K* on seadme tüüp ja kinnistatud number. Niisiis, näited.

В ─ sisesta;
ВЖ ─ sisesta ja oota operatsiooni lõppu;
ВЗУЖ ─ sisesta, kirjuta ja küsi indikaatorite seis enne ja pärast operatsiooni.

Magnetlintseadme töö juhtimiseks oli lisaks samuti „täiskomplekt" käskusid, sama formaadiga nagu juba tutvustatud, näiteks:

ПР ─ jäta tsoon vahele;
ВРИЮ ─ keri tsoon tagasi ja mine indikaatorite järgi;
СНУЗЖ ─ lõpeta magnetlint, oota, küsi ja kirjuta indikaatorid.

Kokku oli 37 selle valdkonna tarbeks realiseeritud käsku.

Vahetusmoodulid on süsteemsed standardprogrammid, mille poole sai kasutajaprogrammist pöörduda nagu suvalise muu alamprogrammi poole. Nood võimaldasid sisestada ja väljastada kirjetest koosnevaid *massiive.* Olenemata konkreetsest seadmest oli vahetusmooduli väljakutsumine standardne:

ИП <vahetusmoodul>,1
КА Д;А

Д (дескиптор) on *massiivi kirjelduse* aadress; kirjeldus paiknes 8 pesas, elementideks näiteks massiivi nimi, vahetuse liik (5 või 6 sümbolit), välisseadme tüüp ja number jmt, aga ka töövälju vahetusprotsessi jooksva seisu hoidmiseks. Parameeter *A* viitas „infotabelile":

A КА АН;АК massiivi vahetusvälja algus- ja lõpuaadressid
КА В;С *B*: aadress, kus paikneb eriolukordasid (nt, loeti lõpuplokk) lahendav kood, *C:* tõrgete töötlemise bloki aadress
КА ЗН;ЗК kirje vahetusvälja algus- ja lõpuaadressid
КА ОФ;АС trüki korral lehekülgede vormistusinfo aadressid

Toome mõned vahetusmoodulite näited: avada sisendmassiiv magnetlindil(ОТВМЛ), avada väljundmassiiv perfokaartidel, perfolindil või trükkalil(ОТЫ), sisestada perfolindilt (ВПЛ), väljastada magnetlindile (ЫМЛ) või trükkalile (ЫПЧ), lugeda (ЧТЗ) ja kirjutada (ЗПЗ) kirje, sulgeda väljundmassiiv (ЗАЫ), kujundada trükitav tekst (ОФОРМ).

2.5.2.6. Näiteprogramm

Esimeseks näiteks on ühe alamprogrammi transleerimisprotokoll, mis on lisas 7. See programm väljastab trükiseadmele sisseprogrammeeritud ja masinkujul esitatud kümnendarvu tekstikujul [40, lisa 3].

Palume meeles pidada assembleri direktiivid ridadel 010 171 ja 010 190: esimene on väljastuskäsk, mis näitab, et seade on trükkal ja trükitakse üks rida, teine aga määrab ülekande formaadi (5 sümbolit pesas, so. tekst), milliselt väljalt trükkida, alates mitmendast ja lõpetades mitmenda sümboliga. Tuletagem seda meelde, kui jõuame *FORTRAN*i sisend- ja väljundkäskudeni.

Eribaseerimisega lugemis- ja kirjutamiskäskude näitena esitame allpool programmifragmendid, kus esiteks reserveeritakse dünaamiliselt jaotatavast mälust *N* pesa ja teiseks, vabastatakse reserveeritud mälulõik. Tekst pärineb raamatust [40], lk. 140.

```
ВХ    БАЗ    0
      РЗВ    3
- - - - - - - - - - -
      КОММ  ВЫДЕЛЕНИЕ ДИН. ПАМЯТИ    kommentaar
      ЧСП    АУЗП;0                  A1;B1→ summaator
      ВФВ    ДЛИН;АМАС              A1;B1─N→АМАС
      ВФВ    +1;УЗП                 A1;B1─N─1→ УЗП
      ЛУЗ    +177777В;АМАС           0;B1─N→АМАС
      ЛСУ    1;АМАС                  B1─N→АБ2
      ЧСП    АУЗП;0                  A1;B1→ summaator
      ЗСП    УЗП;0                   A1;B1→B1─ N─1
      Ч      УЗП                     A1;B1─ N─1 → summaator
      ЗСП    АУЗП;0                  A1;B1─ N─1 → A
- - - - - - - - - - - -
      КОММ  ОСВОБОЖДЕНИЕ ДИН. ПАМЯТИ   kommentaar
      ЧСП    АУЗП;0                  A1;B1─N─1→ summaator
      З      УЗП                     A1;B1─N─1→УЗП
      ЧСП    УЗП;0                   A1;B1→ summaator
      ЗСП    АУЗП;0                  A1;B1→ A
```

```
- - - - - - - - - - - -
        ВЫХ     ВХ;0
АУЗП    ПАУП                            (0;А)
        БАЗ     1;РАБ
АМАС    РЗВ     1
УЗП             1
        БАЗ     2;ОБЩ
ДЛИН    РЗВ     1
```

2.5.2.7. Resümee

Tuletagem meelde, et *Razdan-3* jaoks sai kirjutada tervikliku masinkoodi-programmi, kasutades ainult masinkäske, sisestada see klaviatuurilt valitud aadressist alates mällu, anda juhtimine klaviatuurilt valitud aadressile, ja „asi toimis". *Minsk-32* puhul pole see üldjuhul võimalik: programmi saab käivitada ainult pärast *paigaldaja* tööd, too aga eeldas üsna paljude *ekstrakoodide* interpreteerimist, neid aga (ehkki sai ka „puhtasse" masinkoodi lisada) genereerisid assembler-translaator ja komplekteerija.

Minsk-32 normaalne masinataseme programmeerimiskeel polnud enam masinkood, vaid oli assembler. Ja assembler toimis ainult koos operatsioonisüsteemiga, mis „peitis ära" kasutaja eest palju sellele kasutajale (programmeerijale) eluliselt mittevajalikku informatsiooni (juba mainitud ekstrakoodid, näiteks) ja tegid programselt kättesaadavaks objekte, mida masinkoodis olnuks kas raske või võimatu saada (aadresskonstandid, näiteks).

Assembleris programmeerija peab normaalseks toimetulemiseks meeles pidama ainult mõned kokkulepped (sh. pseudodirektiivid) ja üsna palju tööd teeb tema eest (operatsiooni)*süsteem.*

Protsessor suudab *aparatuurselt* interpreteerida ainult „päris"-masinkoodi. Kogu ülejäänu (ekstrakoodid, näiteks) on masinkoodi-sarnane info, mida interpreteerivad *programselt* komplekteerija ja paigaldaja. Ent programmeerija jaoks on see kõik toimiv „masin". Siit tuleneb programmeerimiskeelte-alases kirjanduses tihti kasutatav mõiste „*keele virtuaalmasin"* (vt. [44], lk. 42): mingi programmeerimiskeele kasutaja jaoks toimib kogu masin näiliselt nagu tolle keele *aparatuurne interpretaator.*

Lõpuks, rõhutagem veel kord olulisi seiku. *Minsk-32*l oli oma aja kohta arenenud operatsioonisüsteem, vaikimisi-stiiliks oli moodulprogrammeerimine, oli aparatuurne tekstitöötluse-toetus (tõsi küll, kohmakavõitu), sisend- ja väljundoperatsioonidele oli nii aparatuurne kui ka programne toetus.

Minsk-32 võis töötada *Minsk-22* režiimis: täita eelmise mudeli programme, sh. *Minsk-22* jaoks koostatud *Malgol*-programme. Masina tarkvarasüsteemi kuulusid translaatorid keeltest *ALGOL-60* (*АЛГАМС*), *FORTRAN* ja *COBOL*.

2.5.3. *IBM 360/370*

2.5.3.1. Üldandmed

„Pesamasinad" olid orienteeritud arvutustöödele. Ballistika, kosmosetehnika, ilmateenistus, statistika, raamatupidamine...Ent lihtsa aparatuurse toeta oli infotöötlus, kus normaalne informatsioon on tekstikujul, so. sümbolkujul, igale sümbolile kulub meie kultuuriruumis 1 bait. See valdkond hõlmab infosüsteeme (aluseks andmebaasisüsteem jäiga andmemudeli ja fikseeritud päringukeeltega) ja teadmiste baase (teadmised on näiteks interneti-saitidel või ekspert-süsteemide tarbeks struktureeritud, ent siiski peamiselt tekstipõhisel kujul), kirjastamistöid ja palju muud.

Pesamasinate asemele tulid „bait-masinad, kus minimaalse pikkusega ja adresseeritav mäluväli oli pikkusega 1 bait (=8 bitti). Muidugi, raske on ütelda, kas nad ilmusid tekstitöötlusenõudluse tõttu või tegi hüppe tekstitöötlus bait-masinate tõttu. Toogem siinkohal mõned seigad, mis kaasnesid uuele mudelile üleminekuga.

- Käsu formaat muutus muutuvaks; käsk võib olla 0..2-aadressiline.
- Tekstitöötlus muutus loomuliku(ma)ks.
- Operandide pikkused polnud enam jäigalt piiratud. Seejuures, täisarvulise aritmeetika (ja loogika) jaoks tekkis 8-bitine arvutüüp. Spekulatsiooni korras ─ ehk sai siit alguse idee teha 8-bitine protsessor, ja siit edasi kõik mikromasinad (*Apple*, *Yamaha* jpt). Mõistagi, 8-bitine aadressruum vajab baseerimis-, indekseerimis- jm. vahenditega laiendamist.

IBM/360 kloon TRÜ Arvutuskeskuses oli masin *EC-1022* ning järgmise põlvkonna, *IBM/370* kloon üks Baltimaade võimsamaid, *EC-1060* (see masin mahtus umbes korvpalliväljaku-suurusele pinnale, op-mälu maksimaalne maht oli 8MB, ja iga 8st *HD*-seadest (suuruses nagu võimas pesumasin) mahutas kahte kahe-käe-ketast á 100 *MB* ja töökindlus oli nullilähedane ─ tavaline oli, et ca 5-minutilise töö järel pidi masina tõrke likvideerima valveinsener[1]).

IBM-masinal oli 16 kiiret neljabaidilist üldregistrit *R0..R15* ja 4 kiiret kaheksabaidilist ujupunktregistrit *R0, R2, R4* ja *R6* . Kõiki üldregistreid võis kasutada nii summaatori kui ka baas- või indeksregistri rollis. Registrid polnud struktureeritud (näiteks olid sellised kaheaadressilise masina indeksregistrid). Käskude operandid võisid paikneda registrites või mälus: nende asukoha järgi eristatakse käsuformaate. Viimaste nimetused pärinevad inglisekeelsetest lühenditest: R=*Register*, S=*Storage* (mälu), X=*indeX*, I=*Immediate operand* (vahetu operand). Käsuformaatide esitamisel kasutame osaliselt juba tuttavat sümboolikat: R_i (4 bitti) on register, D (12 bitti) on suhtaadress (e. nihe baasaadressi suhtes), B (4 bitti) on baasregister, X (4 bitti) on indeksregister, L_i (4 bitti) on i-nda operandi pikkus, L (8 bitti) on pikkade operandide võrdne pikkus ning I (8 bitti) on vahetu operand. Lubatud operandid olid erinevate pikkustega: 8 bitti (sümbol), 16 bitti (poolsõ-

[1] Mõistagi polnud selles süüdi *IBM*, vaid tõik, et N. Liidu jaoks kehtis strateegiliste toodete (sh. arvutid) embargo ning plagieerimine (viisakamalt "adapteerimine") polnud edukas. *Rein Jürgenson* kirjutas läbi väheste lillede: "…ühtsussüsteemi arvutid erinevad loogilise struktuuri ja tarkvarasüsteemi ülesehituse poolest eespoolnimetatud arvutitest (eriti *IBM*-arvutitest) äärmiselt vähe…" [24, lk. 10].

na (2 baiti) ─ *int*-tüüpi arv, 4-baidine sõna (*int* või *real*), 8 baiti (10ndarv, sümbolid või *real*), 16 baiti (10ndarv või sümbolid) ning kuni 256 baiti (sümbolid).

2.5.3.2. Käsuformaadid

Enamiku *IBM*-assembleri ja masinkoodikäskude vahel oli 1:1-vastavus. Käsuformaadid olid järgmised:

RR (Register ─ Register): Kuivõrd käsu operandide hulgas pole mäluaadresse, on see 0-aadressiline käsk: mõlemad operandid on kiiretes registrites. Käsu pikkus on 2 baiti.

Kood	R_1	R_2
1	2	

Näiteks, käsk 18 3 7 kirjutab üldregistrist R7 sõna üldregistrisse R3 (so, (R7)→ R3) Assembleris kirjutatakse LR 3,7

RX (Register ─ indeXed storage). Tehe toimub registri ja mäluväljal E oleva operandi vahel, kusjuures E=(B)+(X)+D. B on baas- ja X indeksregistri rollis; kui kirjutada B või X kohale nullid, siis nihet D vastavalt ei baseerita ja/või indekseerita. Käsu pikkus on 4 baiti.

Kood	R	X	B	D
1	2	3	4	

Näiteks, käsk

48	3	4	5	1F0

Kirjutab mäluaadressilt E=(4)+(5)+1F0 sõna üldregistrisse R3. RX-formaadi käsud on ilmselt üheaadressilised (nagu üldse enamik selle masina käske). Assembleris näeb see käsk välja nii: L 3,OBJEKT(4,5) ─ ilmselt on etiketi OBJEKT väärtus 1f0. Indekseerimise puudumist väljendatakse assembleris meie näites nii: OBJEKT(,5). Registrit R0 saab kasutada ainult *R* rollis, mitte aga *X* või *B* kohal, *R* aga täidab „ilmutatud summaatori" rolli.

RX' sarnaneb RX-põhiformaadile:

Kood	M	X	B	D
1	2	3	4	

Seda formaati kasutatakse suunamiskäskudes. *M* on kuueteistkümnendarv $0 \le M \le f$, mille väärtus näitab, missuguse signaali puhul tuleb anda juhtimine aadressile A=(B)+(X)+D.

Näiteks, käsk

47	B	0	C	148

annab juhtimise aadressile (C)+148 kahel juhul: esiteks siis, kui signaali andis võrdluskäsk ning operand1>operand2 ja teiseks siis, kui signaali andis aritmeetikakäsk: tehte tulemus oli mittenegatiivne. Assembler: kas BH SIIA(0,12) või BNM SIIA(0,12); etiketi SIIA väärtus on 148. Nagu näeme, vastab ühele masinkoodi käsule kaks erinevat assembler-direktiivi.

RS (Register ─ Storage)

Kood	R1	R2	B	D
1	2		3	4

Näiteks, kui neljandas üldregistris on absoluutne aadress 47000, siis saab käsuga

86	1	5	4	0

Kanda registritesse 1, 2, 3, 4 ja 5 sõnad aadressidelt 47000, 47004, 47008, 4700C ja 47010. Assembleris näeb see välja nii: LM 1,5,0(4).

RS' ei kasuta formaadi-registrit R2: see kodeeritakse alati kui 0. See on *nihutamiskäskude* formaat. Oletagem näiteks, et meil on üldregistris R3 arv 16 (kahendarv 10000). Käsuga

88	3	0	0	2

Nihutatakse kolmanda registri sisu kahe kahendkoha võrra paremale: vasakult lisatakse kaks nulli, paremalt läheb kaks kahendkohta kaduma: avaldise E=(B)+D väärtust interpreteeritakse nihutamiskonstandina. Tehte tulemusena on (R3)=4, kahendarvuna 100. Assembleris näeb see näide välja nii: SRL 3,2(0) või SRL 3,2.

SI (Storage ─ Immediate operand) evib järgmist formaati:

Kood	I	B	D
1	2	3	4

Vahetu, 1-baidine operand *I* paikneb käsus endas. Näiteks, käsuga

95	6F	3	0

kontrollitakse, kas mäluaadressil (3)+0 on sümbol „?" (koodiga 6F) ning töötatakse välja signaal „võrdub" või „ei võrdu". Assembleris: CLI 0(3),C'?'

SS (Storage ─ Storage). Selle formaadiga käskude pikkus oli 6 baiti:

Kood	L1	L2	B1	D1	B2	D2
1	2		3	4	5	6

Käsk on kaheaadressiline: mõlemad operandid paiknevad mälus. Esimese operandi pikkus on L1+1 baiti ja aadress A1=(B1)+D1, teisel operandil vastavalt L2+1 ja A2=(B2)+D2. Näiteks, kui esimeses registris on aadress 17A30 ja teises 23FC0, siis käsuga

CP	5	5	1	0	2	4

võrreldakse aadressil 17A30 paiknevat kuuebaidilist kümnendarvu aadressil 23FC4 asuva sama pika kümnendarvuga. Assembleris näeb see käsk välja nii: CP 0(5,1),4(5,2)

SS' on samuti 6-baidine, ent tehe sooritatakse kahe L-baidise mäluvälja vahel ($1 \leq L \leq 256$). Formaat:

Kood	L-1	B1	D1	B2	D2
1	2	3	4	5	6

Näiteks, käsuga

D2	FF	7	0	9	0

kantakse registris 9 olevast aadressist alates 256 baiti mäluväljale, mille aadress on registris 7. Assembler: MVC 0(256,7),0(9). Assembleris kirjutatakse välja tegelik pikkus, seda lühendab 1 võrra translaator.

2.5.3.3. Käsugrupid

Eelmises jaotises esitasime põgusa ülevaate *IBM*i põhilistest käsuformaatidest. Tegelikult oli formaate mõnevõrra rohkem (igal „lisaformaadil" vaid vähesed rakendused), nii oli RR-formaadil veel 3 modifikatsiooni. Järgnevas tabelis (tabel 3.5.3.3a) näitame, milliste valdkondade jaoks olid eri formaadid mõeldud (RR ja tema modifikatsioonid on kõik veerus RR*)

Operandid võivad olla kas 1-baidised, 2-baidised (poolsõna), tavaliselt 4-baidised (sõna) või 8-baidised (topeltsõna). SS-formaadid opereerivad kuni 256-baidiste väljadega. Tsiteerigem [41, lk 55]: „Registrite pikkus on 4 baiti, poolsõna puhul võtab operatsioonist:osa registri parempoolse madalama järgu 2 baiti, kahekordsete sõnade puhul aga 2 registrit. Ka mõningate korrutamis- ja jagamisoperatsioonide puhul võtavad sellest osa kaks registrit."

Käsurühm	RR*	RX	RX'	RS	RS'	SI	SS	SS'
Laadimine ja salvestamine	*	*		*		*	*	*
Aritmeetika	*	*					*	
Loogika	*	*				*		*
Võrdlemine	*	*				*	*	*
Nihutamine					*			
Suunamine	*	*	*	*				
Teisendamine		*					*	*
Muud käsud	*	*				*		*

Tabel 2.5.3.3a. Käsugrupid ja formaadid (vt. [22], lk. 48).

Esitagem allpool väljavõtte *IBM*i assemblerkäskudest (seegi pärineb [41], lk 57..61, lisaks [13], lk. 764..777).

Fiks-koma aritmeetika.

Kood ja mnemokood	Semantika	Formaat	Tegevus
5A A	Liitmine	RX	(R1)+(E2)→ R1
4A AH		RX	poolsõna
5B S	Lahutamine	RX	(R1)-(E2)→ R1
4B SH		RX	poolsõna
5C M	Korrutamine	RX	(R1+1)x(E2)→ R1, R1+1
5D D	Jagamine	RX	(R1,R1+1)/(E2)→ R1+1
1A AR	Liitmine	RR	(R1)+(R2)→ R1
1B SR	Lahutamine	RR	(R1)-(R2) → R1
1C MR	Korrutamine	RR	(R1+1)x(R2) → R1, R1+1
1D DR	Jagamine	RR	(R1, R1+1)/(R2) → R1+1

Tabel 2.5.3.3b. Fiks-koma aritmeetika.

Kümnendaritmeetika. See käsugrupp on oluline raamatupidamise jaoks: bilanss peab klappima sendipealt, summad on sentides ja ümardamist tuleb vältida[1].

Kood ja mnemokood	Semantika	Formaat	Tegevus
FA AP	Liitmine	SS	(E1)+(E2)→ E1
FB SP	Lahutamine	SS	(E1)-(E2)→ E1
FC MP	Korrutamine	SS	(E1)x(E2)→ E1
FD DP	Jagamine	SS	(E)/(E2)→ E1

Tabel 2.5.3.3c. Kümnendaritmeetika.

[1] Teadaolevalt esimene trellide taha saadetud programmeerija mõisteti süüdi pangas intresside ümardamis-vahede (+-suunas) oma arvele kandmise eest. Iga operatsioon andis 1 sendi, ent „kes kopikat ei korja, see rublat ei saa".. Tema sai väidetavalt miljoni, lisaks aastaid.

Mälukäsud.

Kood ja mnemokood	Semantika	Formaat	Tegevus
58 L	Laadimine	RX	Sõna (E2)→ R1
48 LH		RX	poolsõna
43 IC		RX	bait
41 LA		RX	E2 abs-aadress või konstant → R1
98 LM		RX’	(E2)→ alates R1.. R3
18 LR		RR	(R2)→ R1
12 LTR	testimisega	RR	(R2)→ R1, signaalidega
50 ST	Kirjutamine	RX	(R1)→ E2
42 STC		RX	(R1)→ E2 (bait)
40 STH		RX	(R) → E2 (poolsõna)
90 STM		RX’	(R1..R3) → alates E2st
D2 MVC		SS’	L baiti (E2)→ E1
92 MVI		SI	I2→ E1 (1 bait)

Tabel 2.5.3.3d. Mälukäsud.

Lühikeste ja pikkade ujupunktarvude jaoks olid tähed „E” ja „D”., näiteks LTER ja LTDR (operandid ujupunktregistrites), LER ja LDR, RX-formaadi käsk MXD ─ korrutamine ülipika resultaadiga (kahes järjestikuses ujupunktregistris) jpt.

Võrdlemis- ja juhtimiskäsud.

Kood ja mnemokood	Semantika	Formaat	Tegevus
F9 CP	Võrdlemine, 10ndarvud	SS	(E1) ¤ (E2) sümbolid
D5 CLC	Võrdlemine, sümbolid	SS’	Kuni 256 baiti
55 C	Võrdlemine, sõnad	RX	(R1)¤(E2)
49 CH	Võrdlemine, poolsõnad	RX	(R1)¤(E2), poolsõna
19 CR	Võrdlemine	RR	Signaal
95 CLI	Võrdlemine, sümbol	RI	(E2) ¤ I2→ signaal
47 BC	Tingimuslik suunamine	RX	Mine E2, kui tingimuskood on tõene
07 BCR	Tingimuslik suunamine	RR	Mine (R2), kui tingimuskood on tõene
45 BAL	Suunamine naasmisega	RX	Järgmise käsu aadress → R1, juhtimine → E2
46 BCT	Tsüklikäsk	RX	Kui (R1) on pärast 1 lahutamist 0, siis mine E2, muidu järgmisele käsule.
05 BALR	Suunamine naasmisega	RR	Nagu BAL, juhtimine aadressile (R2).

Tabel 2.5.3.3e. Võrdlemis- ja juhtimiskäsud

Sellesse gruppi kuuluvad ka suunamiskäsu (kood 47) ülejäänud modifikatsioonid: BE (mine, kui „võrdub": =), BL (<), BH (>), BNE (≠), BNL (≥), BNH (≤), BP (+), BM (-), BO (ületäitumisel), BZ (0), BNP (mitte +), BNM (mitte -), BNO (mitte-ületäitumine) ja BNZ (mitte 0) ja B (tingimusteta suunamine)

Kui kirjutada BALR kujul R2=0, siis R1 salvestatakse ikkagi järgmise käsu aadress, ent suunamist ei toimu.

Rõhutagem, et esitasime ainult *väljavõtte* suurest hulgast käskudest; tutvustamata jäid loogikakäsud (RX ja RR-formaatides), teisendamiskäsud (10→ 2 ja 2→ 10), nihutamiskäsud, kümnendarvude pakkimine ja lahtipakkimine jpm.

Masinkoodi tasemel toetatakse aparatuurselt andmetüüpe *sümbol, int poolsõna, int sõna, real sõna, real topeltsõna, kümnendarv* ja kuni 256-baidiseid *stringe.*

2.5.3.4. Naasmisega suunamine

Nagu meile juba tuttavas *Minsk-32* programmeerimissüsteemis, tuleb ka *IBM* süsteemis kõik programmid vormistada *alamprogrammidena*; nende kirjutamise kokkuleppeid käsitleme järgmises jaotises.

Ent on situatsioone, kus mingit koodi pole otstarbekas vormistada alamprogrammiks, näiteks siis, kui too kood on suhteliselt lühike (alamprogrammi pöördumine on suhteliselt suurt ressurssi nõudev, vaja on nii ruumi kui ka aega).

Allpool toome näited käskude BAL ja BALR kasutamisest, mis pärinevad [51, lk.209 ja 211], illustreerimaks „lühikest koodi". Esimene näide:

```
ROUTE1      ZAP   PAKANSR,PACK1   nullib PAKANSR ja kannab sinna PACK1
            BAL   10,DOMULT       mine DOMULT
            AP    CUMULAT,PAKANSR  CUMULAT:=CUMULAT+PAKANSR
- - - - - - - - - - - - - - - - - - - -
             ZAP  PAKANSR,PACK2
            BAL   10,DOMULT
            AP    CUMULAT,PAKANSR
- - - - - - - - - - - - - - - - - - - -
            ZAP   PAKANSR,PACK3
            BAL   10,DOMULT
            AP    CUMULAT,PAKANSR
- - - - - - - - - - - - - - - - - - - -
DOMULT      MP    PAKANSR,PAKMULT  PAKANSR:=PAKANSR*PAKMULT
            BCR   15,10             mine (R10)
- - - - - - - - - - - - - - - - - - -
```

```
PACK1    DC   PL3`0´      konstandid on pikkusatribuudiga kümnendarvud
PACK2    DC   PL3`0´
PACK3    DC   PL3`0´
CUMULAT  DC   PL6`0´
PAKANSR  DC   PL4`0´
PAKMULT  DC   PL1`5´
```

Ja teine näide:

```
PRELOAD  LA   11,DOMULT
ROUTE1   ZAP  PAKANSR,PACK1   nullib PAKANSR ja kannab sinna PACK1
         BALR 10,11           mine DOMULT
         AP   CUMULAT,PAKANSR  CUMULAT:=CUMULAT+PAKANSR
- - - - - - - - - - - - - - - - - - - -
         ZAP  PAKANSR,PACK2
- - - - - - - - - - - - - - - - - - - -


         BALR 10,11
         AP   CUMULAT,PAKANSR
- - - - - - - - - - - - - - - - - - - -
         ZAP  PAKANSR,PACK3
         BALR 10,11
         AP   CUMULAT,PAKANSR
DOMULT   MP   PAKANSR,PAKMULT  PAKANSR:=PAKANSR*PAKMULT
         BCR  15,10
```

Konstandid on deklareeritud eelmises, so. „esimeses" näites.

Niisiis, alamprogrammide asemel on võimalik kasutada lihtsamat varianti. Ent leidub ka ülesandeid, kus alamprogrammide asemel on otstarbekam kasutada teistsugust süsteemi, mida on hõlbus realiseerida just naasmisega suunamiskäskude abil. Tuletagem meelde, et alamprogrammid moodustavad alati *hierarhilise* süsteemi: välja kutsub „ülemus" (kõrgema taseme programm) ja välja kutsutakse „alluv". Alternatiiv on *kaasprogrammide* (*coroutins* või *coprogramms* inglise keeles, *сопрограммы* vene keeles) kirjutamine.

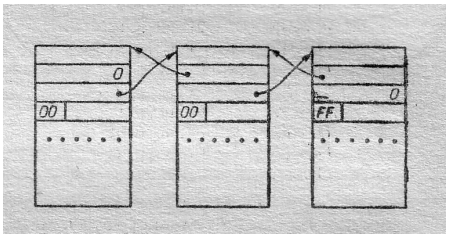
T. Pratt [44, lk. 178..182] selgitab kaasprogrammide olemust järgmiselt. Kaks (või enam) programmi on üksteise jaoks kaasprogrammid, kui igaüks neist saab oma suvalises punktis anda juhtimise (mõne) teise kaasprogrammi määratud punkti: esimesel pöördumisel algusse, järgnevatel kordadel aga sinna, kuhu väljakutsutav kaasprogramm eelmisel korral fikseeris. Kui eirata realiseerimisnüansse (registrite säilitamine, parameetrite ülekanne ─ siin tundub loomulikuna ühisvälja kasutamine, jmt), siis sobib ilmselt sellise koosluse tekitamiseks kasutada naasmisega suunamise käske *á la* BAL ja BALR.

2.5.3.5. Alamprogrammid

*IBM*i programmeerimissüsteemis tuleb masinkoodi- ja assemblerprogrammid vormistada alamprogrammidena. Masinkoodis programmeerides tuleb järgida *kokkuleppeid* ning hoolitseda komplekteerija ja paigaldaja tööks vajalike pseudokäskude eest. Assemblerprogrammide puhul jääb see translaatori tööks. Mainitud kokkulepped on järgmised:

Registritel 0, 1, 13, 14 ja 15 on spetsiifiline roll. Nende kasutamist programmi „tööregistritena" tuleks vältida, kuna reeglina rikuvad nende seisu süsteemsed makrokäsud . Registreid 0 ja 1 kasutatakse alamprogrammi parameetrite ülekandmiseks (R0 ainult süsteemsete makrode puhul). Registris 1 on viit *n*-viidasele vektorile (*n* on parameetrite arv) ja iga selle vektori i-s element viitab *i*-ndale parameetrile ($0 \le i \le n$).

Registrisse 13 tuleb kirjutada väljakutsuva alamprogrammi *säilituspiirkonna* (i.k. *save area*) aadress. Väljakutsutava (madalama taseme) alamprogrammi esimeseks tööks peab olema registrite jooksva seisu salvestamine; see tuleb taastada enne alamprogrammist väljumist. Säilituspiirkonnad tuleb ühendada viitade abil *topeltseotud ahelasse*, nagu illustreerib joonis 2.5.3.5a ([49], lk. 101). Lipp *00* 3. sõna alguses tähistab aktiivset piirkonda ja *FF* kasutamiskõlbmatut. Neid lippe haldab makrokäsk *RETURN* (välju alamprogrammist).



Joonis 2.5.3.5a. Säilituspiirkondade ahel.

Säilituspiirkonna moodustamise lihtsaim variant on 18 sõna reserveerimine programmi muutujate ja konstantide väljas, ent see pole soovitatav variant, kuivõrd ta on vastuolus *re-enteraabluse* printsiibiga (i.k. *reenterable*). See printsiip nõuab, et alamprogrammi suvaline väljakutse leiaks eest alati ühesuguse keskkonna (mis välistab lokaalsete muutujate kasutamise). See põhimõte on hõlpsasti arusaadav, kui mõtleme *rekursiivsetele* alamprogrammidele[1]. Niisiis on heaks tooniks võtta säilituspiirkonnale (ja lokaalsetele muutujatele) mälu dünaamiliselt (näiteks makrokäsuga *GETMAIN*; me toome asjakohase näite *IBM*i makroassemblerit käsitledes). Kurioosumina pole sugugi mitte kõik OS makrokäsud re-enteraablid.

[1] Teine näide: ajajaotusega (so, paljude „kasutajatega") süsteemides on oluline, et erinevad kasutajad saaksid oma ajakvandi vältel kasutada samu (jagatavaid) ressursse, sh. mooduleid. Ükski kasutaja ei tohi sellist moodulit mingilgi moel modifitseerida ja iga kasutaja peab saama oma tööd moodulis jatkata samast seisust, kus ta eelmine seanss katkestati.

Ülemise taseme alamprogrammi registrite seis taastatakse enne alamprogrammist väljumist. Ja kui alamprogrammi säilituspiirkond on saadud dünaamilisest mälust, siis tuleb see tagastada.

Säilituspiirkonna struktuur (*sa*=suhtaadress) on järgmine:

sa	**Sisu**
0	Kasutab PL\I virtuaalmasin
4	Viit eelmisele säilituspiirkonnale („teab" ülemise taseme programm)
8	Viit järgmisele säilituspiirkonnale (määrab „jooksev" alamprogramm)
12	R14: naasmisaadress
16	R15: sisendpunkti aadress
20	R0
24	R1
28	R2
32	R3
36	R4
40	R5
44	R6
48	R7
52	R8
56	R9
60	R10
64	R11
68	R12

Tabel 2.5.3.5a. Säilituspiirkonna formaat.

„Lõdva" kokkuleppena kasutatakse R15 naasmiskoodi edastamiseks (täisarv 0 viitab normaalsele lõpule, koodid 4, 8, … aga mingile analüüsimist vajavale situatsioonile, nood koodid on „vabad" selles mõttes, et neid võib analüüsida *ainult* väljakutsuv kasutajaprogramm.

Tulgem veel kord tagasi mittestandardsete lahenduste juurde. *Kaasprogramme* võiks realiseerida nii, et väljakutsuv programm hoolitseb ise oma registrite säilimise eest, väljakutsutav samuti.

Veel üks alamprogrammi variant kannab nimetust *ülesanne* (i.k. *task*): see käivitatakse (näiteks *C*-keskkonnas „käsurealt" funktsiooniga *system*) ning ta töötab põhiprogrammiga paralleelselt (saab aega siis, kui põhiprogramm tegeleb kursori jälgimisega menüüdes või sisend-väljundoperatsioonidega) ja töö lõpetades ei naase mingile kindlale aadressile, vaid annab mingi kokkuleppelise signaali, mida põhiprogramm „teab" ja oskab kontrollida. Näiteks võib selline alamprogramm tööd alustades kirjutada kettale kokkuleppelise nimega faili, mille kustutab siis, kui töö on tehtud. See tüüp on oluline reaalaja ülesannete puhul.

Üks variant selle tehnika rakendamiseks on *planeeritavate alamprogrammide* kirjutamine. Sel juhul pöördub alamprogrammi ja saab sellelt juhtimise spetsiaalne moodul ─ *planeerija*, mis kontrollib teatud tingimuste või sündmuste täidetust/saabumist. *Pratt*i järgi [44] on neli levinumat varianti („üldarusaadavas" programmeerimiskeeles) sellised:

- CALL B AFTER A
- CALL B WHEN X=5&Z>0
- CALL B AT TIME=25 või CALL B AT TIME=CURRENT-TIME+10
- CALL B WITH PRIORITY 7 (käivitatakse kohe, kui pole ühtegi seitsmest kõrgema prioriteediga protsessi).

Planeerija töö sisaldab palju keerulist (sj. programmeerija eest varjatud) tööd. Esimene seda tehnikat kasutanud keel (loe: selle keele translaator koos vastava virtuaalmasinaga) oli *O.-J. Dahl*i ja *K. Nygaard*i *SIMULA-67*, kus planeeritavad alamprogrammid võisid olla kasutatavad *kaasprogrammidena.*

Intuitiivselt võime ette kujutada, et planeerijat kasutav virtuaalarvuti pidi evima mingit katkestusi kontrollivat alamsüsteemi (mida pole keeruline realiseerida *interaktiivses* keskkonnas ─ näiteks võib planeeritud alamprogrammide juht-andmestruktuuri üle vaadata iga kord, kui avatakse näiteks dialoogiaken), või muid vahendeid, näiteks käivitada sobivas punktis „oma kell" ja kontrollida sobivates kohtades selle jooksvat seisu ning vastava tingimuse täidetuse korral saab planeerija pöörduda tolle tingimusega seotud alamprogrammi. Nii või teisiti, põhiprogramm peab aeg-ajalt pöörduma planeerija (mis on ju ka alamprogramm) poole.

Analoogiline situatsioon tekib *mitmeprotsessoriliste* masinatega, kus on mingis programmi punktis otstarbekas jagada töö eri protsessorite vahel, igal neist käivitatakse alamprogramm (või selle eksemplar) ─ seda teeb tavaliselt operaator *fork* („kahvel") ─ ja põhiprogramm peab tavaliselt töö jatkamiseks ootama (tavaliselt operaatoriga *join,* „ühine") kõigi paralleelselt töötanud harude töö lõppu. Sel juhul ei lõpeta paralleelselt töötav alamprogramm naasmiskäsuga, vaid signaaliga „tehtud", millest peab ülemise taseme programm „aru saama". Näite toome *PL\I*-keelt tutvustades.

Rekursiivse alamprogrammi näitena toome kahendpuu läbija lõppjärjekorras (*vasak* → *parem* → *juur*). Parameetrite väli edastatakse R1-s (0: viit *vasak*, 4: viit *parem*, …), R2 sisaldab tipu *töötlemisprogrammi* aadressi. *WRF* ja *WRL* ja *CALL* (alamprogrammi pöördumine) on makrokäsud, mida tutvustame *makroassembleri* jaotises.

```
POST    CSECT               Code SECTion
        WRF     3           (1) → 3
        L       1,0(3)      vasak
        LTR     1,1         vasak=0?
        BZ      PAREM       kui jah, siis mine „parem”
        CALL    POST        rekursioon
PAREM   L       1,4(3)      parem
        LTR     1,1
        BZ      LABOR
        CALL    POST
LABOR   LR      15,2
        LR      1,3         tipu aadress → R1
        BALR    14,15       mine töötlusprogrammi
        WRL
```

Kokkuvõtte IBM/360/370 süsteemist teeme pärast makroassembleri tutvustamist.

2.5.4. Mikroarvutid: *Intel*-protsessor ja selle programmeerimine

2.5.4.1. Protsessor

Selles peatükis tutvume põgusalt mikroarvuti protsessori arhitektuuri ning masinorienteeritud programmeerimiskeeltega. Mikroarvuti all mõtleme *ränikiibi* ümber ehitatud arvutit: nii baseerub *IBM PC Intel*i protsessorile, *Apple][* tugines *Rockwell 6502*-le, *Apple Macintosh Motorola* protsessorile jne [57]. Esimene *Intel*i mikroprotsessor oli 8-bitine mudel 8080, mis tuli turule juba 1971. aastal. Järgnesid 16-bitine 8086 (80-ndate alguses), 32-bitine 80386 (1986) ja üsna hiljuti 64-bitine *Pentium*[1] 4 660 [137].

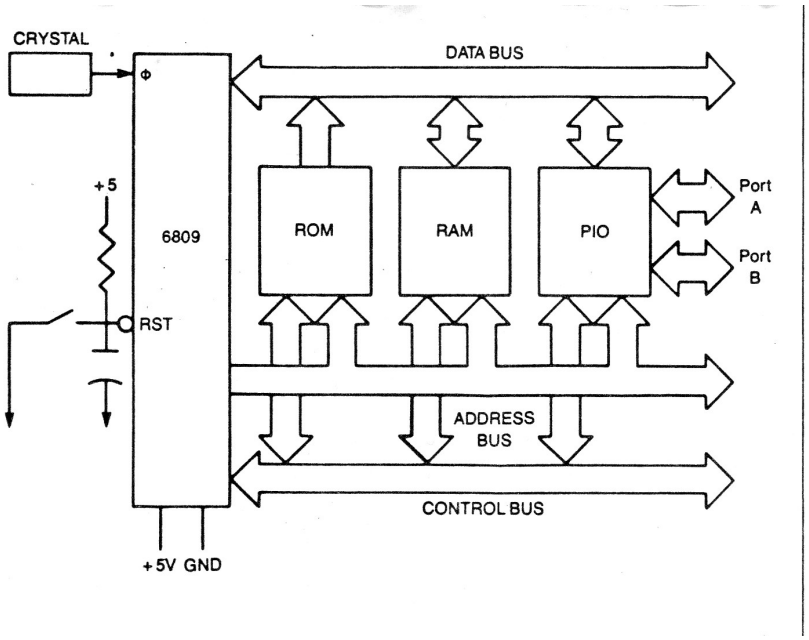
Mitmel põhjusel vaatleme siin 16-bitilist protsessorit *Intel*-8086. Põhjused on eeskätt järgmised:

- See variant on oluliselt lihtsam hilisematest, ent tööpõhimõtted on samad; liiatigi on kõik nood protsessorid ehitatud nii, et võimsam mudel suudab interpreteerida eelnevate mudelite koode.
- Protseduurorienteeritud keele realiseerimise täistsükli näitena kasutame keelt *Trigol,* selle programmid transleeritakse esmalt assemblerkeelde ja sealt masinkoodi, lõppresultaadiks on *.exe*-fail. Assemblerkeele ja selle translaatorina kasutatakse 16-bitist koodi toetavat *Borland*i Turboassemblerit [56], asjakohasest illustreerivast materjalist aru saamiseks tuleb 8086-süsteemist ettekujutuse omamine loodetavasti kasuks.

Lisaks [56] tuginetakse allpool peamiselt raamatule [10]. Niisiis, mikroarvuti. Kui siiani (meie raamatu kontekstis) oli masin igas komponendis (läbi)nähtav ja vajadusel sai õppinud arvutiinsener kruvikeeraja ja jootekolviga parandada nii protsessorit kui ka ope-

[1] Miks *Pentium*? Venemaa pakub välja üsna tõepärase seletuse: N. Liidu superarvuti *Elbrus* juhtiv konstruktor *Vladimir Mstislavovitš Pentkovski* kirjeldas aastaid enne *Intel Pentium*i loomist sellise tasemega kiibi tööpõhimõtteid ja disaini, ja pärast seda, kui uus Venemaa vähendas *Pentkovski* ja *Elbrus*e finantseerimist *40* (*nelikümmend*) *korda*, läks *Pentkovski* tööle USAsse, *Intel*isse. *Pent*ium on enamvähem see, mille kavandas *Pent*kovski. Huviline saab hulgaliselt asjakohast materjali vene otsingusüsteemi *Rambler* abil − piisab, kui otsida *Pentkovski*t.

ratiivmälu (või vahetada välja nonde üksikuid detaile), siis nüüd on olukord kardinaalselt teistsugune. Põhimõtteliselt jääb arvuti ikkagi samasuguseks masinaks nagu seni, ent sootuks erineb ta insenerliku lahenduse poolest.



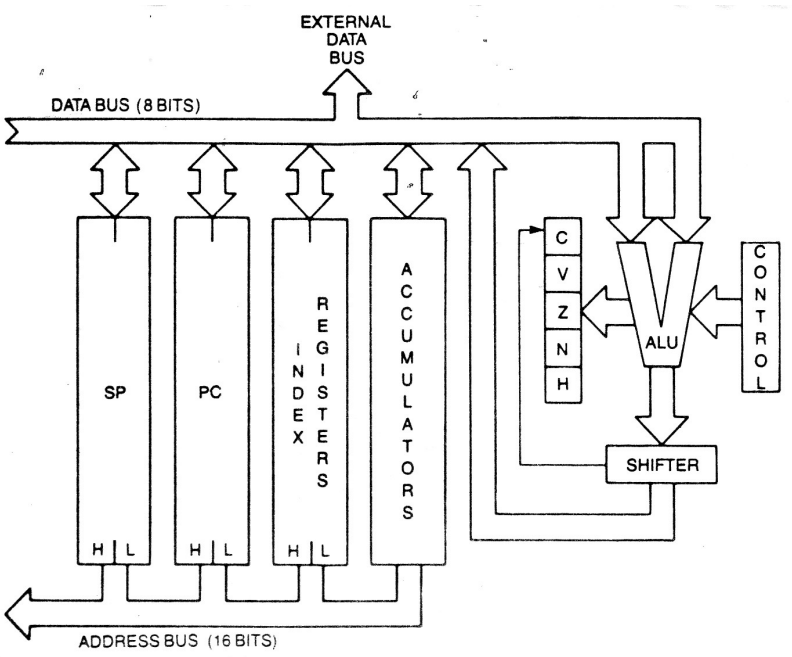
Joonis 2.5.4.1a. Masina üldine skeem.

Joonisel 2.5.4.1.a [10, lk. 28] on kujutatud 8086-protsessorile tugineva masina üldine skeem. Joonise vasakul äärel asub protsessori*kiip*[1] (tavaarusaamise tasemel kivi emaplaadil); seal sisalduvad aritmeetika-loogikaseade, juhtimisseade ja kiired registrid. Protsessoril on kolm *siini*[2] (i.k. *bus*): andme-, aadressi- ja juhtimissiin (joonisel vastavalt *data bus, address bus* ja *control bus*). Nood siinid ühendavad protsessorit kolme ülejäänud tähtsama kiibiga: *ROM* (*Read-Only Memory*, püsimälu, mis sisaldab süsteemiprogramme ─ näiteks algpaigaldusprogrammi, ja mille seis säilib ka siis, kui arvuti on välja lülitatud), *RAM* (*Random Access Memory*, operatiivmälu) ja *PIO* (*Parallel Input/ Output*); viimane on üks sagedaminikasutatavaid välisseadmetega suhtlemist juhtivaid kiipe, selliseid võib olla lisaks palju ja erinevaid

Aadressisiin edastab temaga seotud seadmetele protsessori poolt arvutatud aadresse (et lugeda andmeid või käsku) või kirjutada, juhtimissiin edastab käske (*ROM*is või *RAM*ist) ja/või signaale.

[1] Kiip on originaalis *chip* ─ ühte (enamasti räni-)kristalli pakitud elementide, eeskätt transistoride ja resistoride kogum [11, lk. 190].
[2] Siinid on arvuti komponente (näit. protsessor, draiverite juhtimisseadmed, mälu ja sisend/väljund) ühendavad juhtmed, millede kaudu edastatakse bitijadasid. Siine haldab tavaliselt protsessor. Siinid on tavaliselt laiendatavad, so., nende külge saab ühendada täiendavaid seadmeid [11, lk. 49].

Joonis 2.5.4.1b tutvustab skemaatiliselt *mikroprotsessori* struktuuri ja tööjaotust [10, lk. 30].



Joonis 2.5.4.1b. Mikroprotsessori arhitektuur.

Alustagem sedapuhku skeemi paremast servast. *CONTROL* on juhtimisseade, mis juhib lisaks käskude täitmise järjekorrale ka *taimeri* (süsteemne kell) tööd ja sünkroniseerib riistvara tegevust.

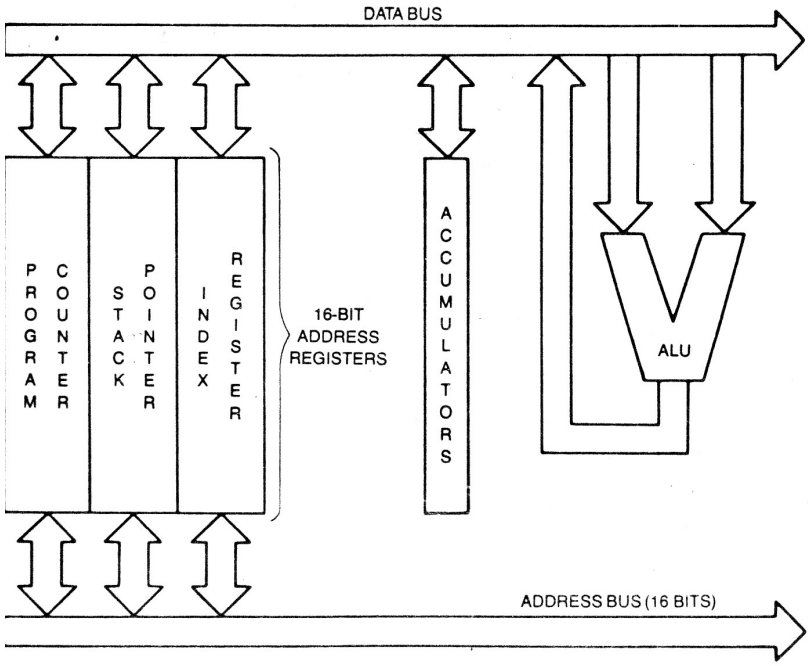
ALU (*Arithmetical and Logical Unit*) on aritmeetika-loogikaseade, mis interpreteerib käske ja ─ tulenevalt 8- või 16-bitilistest operandidest ─ tegeleb ka nihutamisega (*SHIFTER*).

Veel vasemal on *aadressregistrid,* mis on *ALU*-ga ühendatud aadressisiini abil (tagasiside käib andmesiini vahendusel). Neid käsitleme ─ sama põgusalt nagu eelnenud detailegi ─ joonisele 2.5.4.1c ([10], lk. 32) tuginedes.

PROGRAM COUNTER on meie senist terminoloogiat järgides käsuloendaja. *STACK POINTER* (magasiniviit) vajab kommentaari: nimelt toetavad mikroarvutid *aparatuurset magasini* (LIFO-tüüpi: *Last In First Out*). Kusjuures operatiivmälu on ikkagi lineaarne baidijada, ent mingi piirkond sellest on eraldatud magasinile ning *LIFO*-operatsioone *push* ja *pop* toetatakse aparatuurselt (so., noid suudab interpreteerida protsessor)[1].

[1] See mehhanism on vanem kui mikroarvutid. Nii toetas sellist adresseerimist miniarvuti *PDP-11* ([15], lk. 35).

INDEX REGISTER on indeksregister ja *ACCUMULATORS* on ühine nimetus neljale 16-bitisele registrile. Igaüks neist on kasutatav kas tervikuna või siis kahe eraldi 8-bitise registrina.



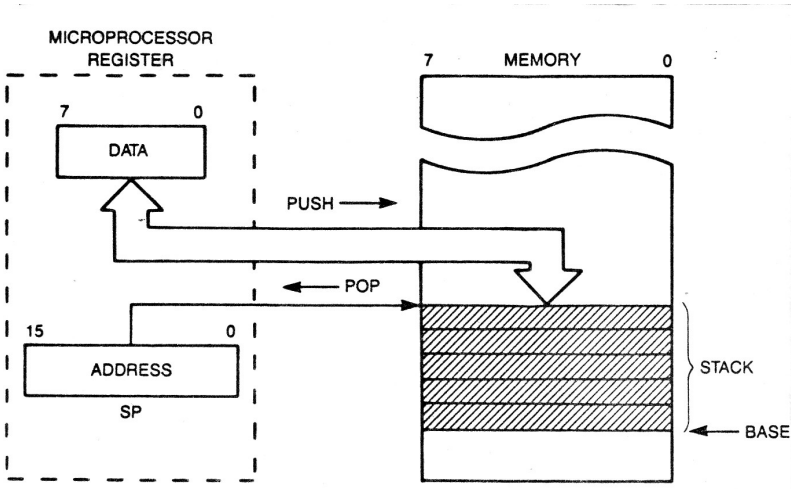
Joonis 2.5.4.1c. Mikroprotsessori registrid.

Magasini aparatuurset toetust illustreerib joonis 2.5.4.1d ([10], lk. 32). Käsk *PUSH* (tõuka, lükka, suru) kirjutab andmeannuse registrist magasini tippu, *POP* (üks tähendus on „äkki väljuma, välja kargama" ─ meie kontekstis sobiks „võta välja") viib andmeannuse magasini tipust registrisse. Magasini tipule viitab *SP* (magasiniviit). Ning magasin ise on sidus piirkond operatiivmälus (*RAM*).

Joonisel 2.5.4.1e on kujutatud *Intel8080* registrid. Neli esimest (*AX, BX, CX* ja *DX*) on 16-bitised, kui kasutada sufiksit *X* või kasutatavad kahe 8-bitise registrina (näit. *CL* või *DH*). Registrinimede sufiksite mnemoonika tähistab kahebaidise arvu „madalamat" (*Lower*) ja „kõrgemat" (*Higher*) baiti. Näiteks, kui registris AX on 16-ndarv *ABCD*h, siis on registris *AH* arv *AB*h ja registris *AL* arv *CD*h (meenutagem: h = *hexadecimal*).

AX-registrit kasutatakse operandina aritmeetika-, loogika- ja salvestamistehete puhul (alati korrutamis- ja jagamistehete korral); siit ka mnemooniline prefiks *A* ─ akumulaator.
BX-registrit kasutatakse mälus paikneva operarandi aadressi nihke (andmesegmendiregistri *DS* väärtuse suhtes) hoidmiseks.

CX-registrit kasutatakse *tsükliloendaja* hoidmiseks.
DX-registril on kaks funktsiooni: esiteks, selle abil saab adresseerida sisend-väljundpuhvreid, teiseks ─ ta osaleb koos *AX*-iga korrutamis- ja jagamistehete täitmisel.



Joonis 2.5.4.1d. Aparatuurne magasin.

Ülejäänud registrid pole struktureeritud. Register *SP* sisaldab jooksvat viita magasini tipule ja seda kasutavad ning muudavad käsud *PUSH* ja *POP*. Ehkki tehniliselt on see register kasutatav ka muudeks otstarveteks, tuleb seda kindlasti vältida. Registrid *SI* (ressursi indeks) ja *DI* (tulemuse indeks) on mõeldud pikkade stringide indeksregistriteks. *BP*-register on nagu *BX, SI* ja *DI* mäluviida jaoks, ent kui viimased positsioneerivad nihet *DS*-registri suhtes, siis *BP* teeb seda registri *SS* seisu suhtes; viimast kasutavad käsud *PUSH* ja *POP*.

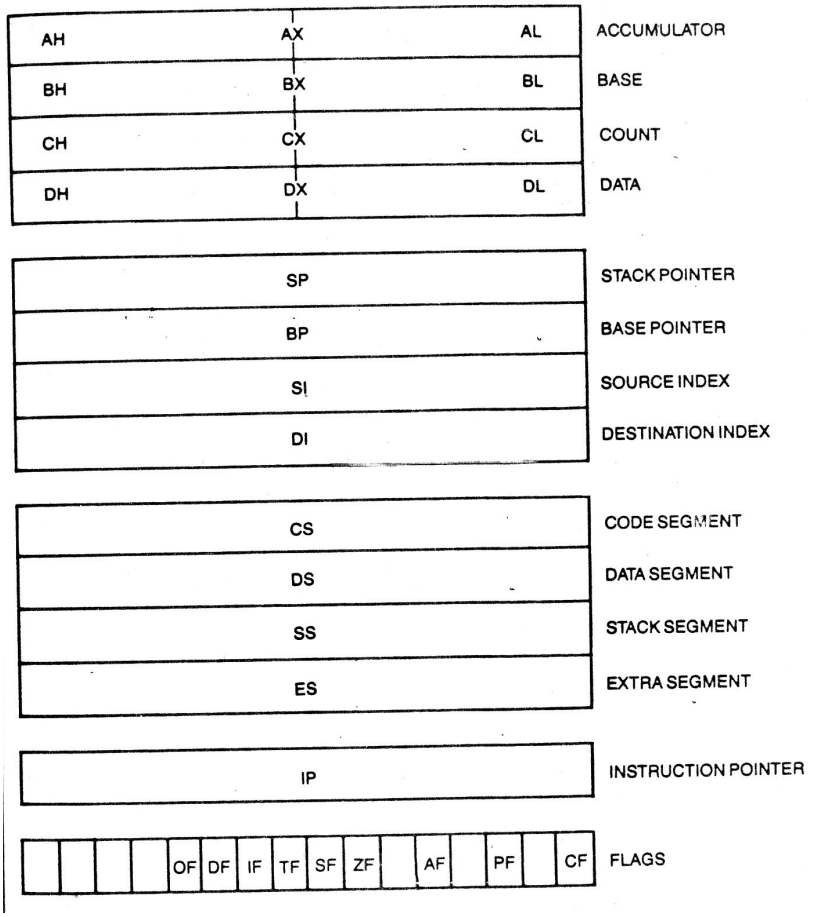
Kõik segmendiregistrid sisaldavad viitu 64KB-ste mälupiirkondade algusse Koodisegmendi register *CS* viitab plokile, kus paiknevad programmi käsud; järgmisena täitmisele tuleva käsu aadress on käsuviida-registris *IP*. *SS* viitab magasini algusele. *DS* viitab piirkonnale, kus paiknevad töödeldavad andmed. Lisasegment *ES* on kasutusel stringide programseks salvestamiseks.

Joonise 2.5.4.1e viimasel real on 16-bitine *lippude* register. Neist *SF* (märk) ja *ZF* (null) on analoogilised meie eelmiste masinate omadele. *OF* (ületäitumine) ja *CF* (*carry flag*, ülekanne) on omavahel seotud. Toome näite.

Olgu meil kaks täisarvulist operandi, *a* ja *b* ja tuleb teha operatsioon $a:=a+b$. Olgu $a=127$ ja $b=1$ (kahendarvud 01111111 ja 00000001). Resultaat on 10000000, see aga on -1 ─ seega vale ning $OF:=1$. Kui aga *a* on kahebaidine (mälus on „madalam bait" esimeses ja „kõrgem bait" teises baidis) kujul 01111111 00000000, siis liitmise käigus ─ seal, kus enne saime ületäitumise ─ „heisatakse" *CF* ning seda kasutades saadakse resultaadiks 00000000 00000001.

Paarsuslipp *PF* on kasutusel sisend-väljundoperatsioonides andmete formaalse õigsuse kontrollimiseks. Suunalipp *DF* ─ (*direction flag*) määrab stringioperatsioonide suuna (vasakult paremale või vastupidi), katkestuslipp (*IF* ─ *interrupt flag*) signaliseerib näiteks,

kas klaviatuurilt on valitud sümbol, *TF* (*trap flag*) on kasutatav ainult süsteemsetes silumisprogrammides (*debugger*).



Joonis 2.5.4.1e. Registrid.

2.5.4.2. Masinkood

*Intel*8086 käsukood on struktureeritud. Allpool on tuginetud [10] 3.ptk. materjalidele.

Kui käsk opereerib *vahetu operandiga,* siis koosneb käsukood kolmest komponendist ─ esimesed 4 bitti on „käsk ise", bitt *W* näitab, kas aadressosas on ühe- või kahebaidine operand (*W*=0, kui operand on ühebaidine) ning 3 järgmist bitti (*REG*) näitavad, milline register on esimese operandi rollis. Variandid on järgmised:

REG	W=1	W=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Tabel 2.5.4.2a. Käsukoodi komponent *REG*.

Kui *W*=1, siis käsukoodile järgnevates kahes baidis on vahetu operand: esimeses baidis „madalam" ja järgmises „kõrgem" bait. Näiteks, käsuga B8 7F 10 kantakse registrisse AX 16-ndarv 107F. Ja käsuga B0 7A kantakse registrisse AL arv 7A.

Kui teine operand *pole vahetu aadress,* siis on käsukoodi 6 esimest bitti „käsk", 7. bitt on *D* (*D*=0, kui operandi määravad väljad *MOD* ja *R/M* operandi-baidis ja *D*=1, kui selleks kasutatakse operandi-baidi välja *REG*, mille semantika on tuttav jooniselt 2.5.4.2a). 8. bitt on *W* (operandi pikkus on 8 või 16 bitti nagu vahetu operandiga opereerivas koodiski). Käsu teine bait on järgmise struktuuriga: 2 bitti *MOD*, 3 bitti *REG* ja 3 bitti *R/M*. *MOD* näitab käsu pikkust baitides: kui *MOD*=00, siis on käsu pikkus 2 baiti, *MOD*=01 näitab, et käsu pikkus on 3 ja 10, et 4 baiti. *R/M*-välja semantika on esitatud tabelis 2.5.4.2b.

R/M	baasregister	indeksregister
000	BX	SI
001	BX	DI
010	BP	SI
011	BP	DI
100	pole	SI
101	pole	DI
110	BP	pole
111	BX	pole

Tabel 2.5.4.2b. Teise operandi määrajad.

Toome neljabaidise käsu näite ─ assemblerkäsk mov cx,count[bx][si] on masinkoodis 8B 88 45 03 (vt. [10], lk. 62).

2.5.4.3. Katkestused, *DOS* ja *BIOS*

Meenutagem: *Minsk-32* programmeerimissüsteem võimaldas ИП-direktiiviga pöörduda süsteemsete alamprogrammide (mis komplekteeriti vaikimisi süsteemilindilt) või kasutaja-alamprogrammide poole, millede asukohalint tuli komplekteerijale teatavaks teha.

*IBM*i keskkond pakkus seda võimalust assembleri tasemel samuti pöördumisega alamprogrammi, ja nagu makroassembleri käsitluses tagapool näeme, tegi makrokäskude abil kasutatavaks peaaegu kõik *OS*i makroteegis olevad „programmid".

*Intel*i assembler (ja masinkood) ei toeta programmeerijat vähem kui *IBM*i vastavad keeled. Komplekteerimisel liidetud alamprogrammi saab pöörduda *call*-mnemokoodiga ning operatsioonisüsteemi (meie valitud platvormi jaoks *MS-DOS*i ─ *MicroSoft Disk Operating System*) ning välisseadmetega suhtlemist madalaimal tasemel toetavad *BIOS*i (*Basic Input-Output System*) funktsioonid on kättesaadavad *programsete katkestuste* abil.

Arvutileksikon ([11], lk. 194) selgitab *katkestuse* (*interrupt*) olemust (meie vabas tõlkes) järgmiselt: see on kas aparatuursete või programsete vahenditega saadetav signaal, mille toimel protsessor salvestab oma jooksva oleku ja annab juhtimise katkestusi töötlevale protseduurile (*interrupt handler*). Katkestus ise võib olla põhjustatud mitmeti: nii normaalse töö käigus „tahtlikult", avariisituatsioonis, välisseadmete teenindamise tellimuse poolt, käsuloendaja seisu sattumisel väljapoole programmi, andmete adresserimisel väljaspoole lubatud piire jne. Kui protsessor saab lühikese aja vältel palju katkestussignaale, ja neid on järge ootamas „liiga palju", siis saab ta nood blokeerida ja püüab olukorra „ise lahendada".

James W. Coffron ([10], lk. 196) selgitab protsessori võimalikke reageeringuid katkestustele järgmise analoogiaga: „Te vestlete ühe inimesega. Teine inimene astub juurde ja kõnetab teid nimepidi, pälvimaks teie tähelepanu." Edasi, teil on „neli võimalikku käitumisvarianti" (taas vabas tõlkes, A.I.):

1. Te võite ligiastunut täielikult ignoreerida.
2. Te võite viia oma jutuajamise lõpuni ja pöörata siis tähelepanu juurdeastunule.
3. Te võite kohe lõpetada suhtlemise esimese vestluskaaslasega ja alustada vestlust juurdeastunuga.
4. Te *katkestate* suhtlemise esimesega, ent pärast teise lahkumist te jatkate juttu esimesega.

Nonde situatsioonide puhul olete Teie protsessori rollis ja „esimene inimene" on täidetav põhiprogramm. „Teine inimene" on katkestussignaal (mis püüab protsessori tähelepanu)." *Intel-8086* katkestusi on 255 tüüpi. Põhimõtteliselt, peamised katkestussignaalide saatjad on:

- Välisseadmed (riistvara).
- Interpreteerimise käigus selguvad vead (näit. jagamine nulliga, ületäitumine).
- Programmid ─ silumisprogrammid või kasutajaprogrammid, viimased kutsuvad nii välja *DOS*i ja *BIOS*i funktsioone.

Tavaliselt ─ kui katkestussignaali ei eirata ─ salvestatakse magasinis registrite jooksev seis, kutsutakse välja vajalik katkestusprogramm ja selle töö lõppedes käitutakse just samamoodi nagu suvalisest muust alamprogrammist naasmisel (võetakse midagi ette väljundandmetega, taastatakse registrite seisud jms).

Infovahetus põhi- ja katkestusprogrammide vahel toimub registrite vahendusel. ***MS-DOS*i** funktsioone välja kutsuvad katkestused kasutavad direktiivi ***int 21h***, kusjuures funkt-

siooni identifikaator (16-ndarv) edastatakse registris *AH* (tihti lisaks *AL*is, näiteks funktsiooni „anna faili moodustamise aeg" väljakutsumiseks peab (*AH*)=57 ja (*AL*)=00) ja võimalikud parameetrid ilmutatud kujul muudes registrites; mis ja kuidas, on paika pandud nende katkestusprogrammide kirjeldustes. Toome ühe sellise kirjelduse näite, kus funktsiooni identifikaator (2A) pannakse registrisse *AH* ning väljund on registrites *AL, CX, DH* ja *DL*.

```
INT 21, 2A : Get Date
      Returns current system date
```

AH = 2Ah	AL = WeekDay (0 = sunday, 1 = monday, .., 6 = saturday) CX = Year (1980..2099) DH = Month (1 = january, .., 12 = december) DL = Day (1..31)

Int21 [64] abil käivitatavad funktsioonid võib valdkonniti rühmitada järgnevalt: failide haldus (loomine, ümbernimetamine, kustutamine, atribuutide lugemine ja muutmine jne), sisend-väljundseadmete haldamine, programmide haldamine (näiteks alamprogrammist väljumine, täitmise lõpetamine, programmi laadimine ja käivitamine jmt), mõned funktsioonid võrguga suhtlemiseks, teegifunktsioonid, välisseadmete haldamine, funktsioonid info-vahetuseks konsooli abil, mäluhaldus (vaba mälu küsimine ja tagastamine jm.) ja süsteemi halduse valdkonna mõned funktsioonid (katkestusvektori seadmine, kuupäeva- ja ajateenistus jm).

BIOS on programmide komplekt, mis *IBM*i arvutites[1] paikneb *püsimälus* (*ROM*) ning selle ülesandeks on arvuti erinevate komponentide (nt. mälu, kettad või monitor) vaheline infovahetus. Samuti teostab see komplekt algpaigaldust. *BIOS*i koostisse kuulub suur hulk funktsioone, millede väljakutse toimub katkestuste vahendusel ning need on assemblerprogrammile kättesaadavad.

Näitena esitame ülevaate *int 13* abil kättesaadavatest madalatasemelistest kettateenustest [65]. Register *DH* määrab kettaseadme:

1. *DH*=00h: 1. flopiketas („A"-draiv);
2. *DH*=01h: 2. flopiketas („B"-draiv);
3. *DH*=80h: 1. kõvaketas;
4. *DH*=81h: 2. kõvaketas.

Alljärgnevas tabelis on toodud mõningad näited.

AH = 00h		Reset Disk Drives
AH = 01h		Check Drive Status
AH = 02h		Read Sectors From Drive
AH = 03h		Write Sectors To Drive

[1] *Robert Lafore* ([31], lk. 322) märgib, et *IBM*-arvutite *ROM BIOS* kujutab endast "sisseehitatud" rutiinsete koodide kogumit, mis on masina lahutamatu osa ja seetõttu on ta pigem riist- kui tarkvara.

AH = 04h		Verify Sectors
AH = 05h		Format Track
AH = 08h		Read Drive Parameters
AH = 09h	HD	Initialize Disk Controller
AH = 0Ah	HD	Read Long Sectors From Drive
AH = 0Bh	HD	Write Long Sectors To Drive
AH = 0Ch	HD	Move Drive Head To Cylinder
AH = 0Dh	HD	Reset Disk Drives
AH = 10h	HD	Test Whether Drive Is Ready
AH = 11h	HD	Recalibrate Drive

Tabel 2.5.4.3a. Katkestuse *int 13* funktsioone.

2.5.4.4. Assembler

*Intel*i assembleri tutvustamiseks valis nende ridade autor eelmiste masinate käsitluste analoogilistes jaotistes kasutatust erineva tee. Allpool me ei püüa anda ülevaadet ei assembleri võimalustest ega direktiivide formaatidest; selle asemel näitame ühe lihtsa programmi transleerimise ja komplekteerimise (linkimise) täistsüklit. Programm *P6.tri* on kirjutatud *Trigol*-keeles ─ see on õppekeel aine „Automaadid, keeled ja translaatorid" kursuse jaoks. Selle keele süntaksist tuleb juttu tagapool, siinkohal ütleme vaid, et tegemist on väga lihtsa *ALGOL*-tüüpi keelega, mille programme saab lahendada kas interpretaatori abil või käivitades kompileeritud *.exe*-faile. Kompilaator genereerib *Intel*8086 assemblerteksti, mis antakse ette *Borland*i „*Turbo Assembler*ile" (*TASM*). Kui assemblertranslaator formaalseid vigu ei leia, siis kirjutatakse kettale (meie näites) failid *P6.map* ja *P6.obj* ning kutsutakse välja komplekteerija *TLINK*, millele antakse käsurealt ette ka juurdelisatavate moodulite nimed. Resultaadina kirjutatakse kettale (taas meie näites) fail *P6.exe*. Trigol-programm ise on järgmine:

P6.tri

```
READ n; F:=1; I:=0;
M1: I:=I+1;
IF I>n THEN GOTO M2;
F:=F*I;
GOTO M1;
M2: WRITE F
```

Usutavasti on lihtne näha, et programm loeb klaviatuurilt arvu *n,* arvutab selle faktoriaali *n!* ning trükib resultaadi ekraanile. Sisestamiseks ja väljastamiseks kasutatakse funktsioone (alamprogramme) *READ* ja *WRITE*.

Tekstifail *P6.asm* on kompilaatori töö vaheresultaat, mis näeb välja peaaegu nii, nagu allpool esitatud. (“Peaaegu” sellepärast, et siin on lisatud ridade kursiivis antud kommentaarid; assembleri enda kommentaar algab semikooloniga “;” ja kehtib rea lõpuni.)

P6.asm

```
; #READ n; F:=1; I:=0;  kommentaarina esitatakse lähteprogrammi tekst
; M1: I:=I+1;
; IF I>n THEN GOTO M2;
; F:=F*I;
; GOTO M1;
; M2: WRITE F#
;
; Program P6.asm
      .MODEL      small programmisegmendi ja andmesegmendi maht on kuni
      .STACK      100h  64 KB, magasini maht on 256 baiti
      EXTRN readint:PROC      väilisviidad alamprogrammide readint ja
      EXTRN bin2dec:PROC      bin2dec nimedele, kasutab linker
      .DATA       andmesegment
n     DW    0     muutujate n, F ja I kirjeldamine DW=Data Word (2B)
F     DW    0
I     DW    0
dTv0  DW    0     töömuutuja (translaatori genereeritud)
Sisse DB    'Input the variable ','$'  Data Byte=string, $ on lõputunnus
Trykk DB    'Variable ','$'            mida ei väljastata
n_S   DB    'n=','$'
F_S   DB    'F=','$'
      .CODE       koodisegment
ProgramStart:     algusmärgend, kohustuslik
      mov   ax,@data    andmesegmendi aadress → AX
      mov   ds,ax       (AX) → DS
      mov   ah,9h       int21-09: string ekraanile
      mov   bx,1        1 rida
      mov   cx,17       ekraanile 17 sümbolit, täiend. tühikutega
      mov   dx,OFFSET Sisse   stringi aadress → DX
      int   21h         mine katkestus-alamprogrammi
      mov   ah,9h       eelnev kordub: ekraanile "n="
      mov   bx,1
      mov   cx,2
      mov   dx,OFFSET n_S
      int   21h
      call  readint     alamprogrammi pöördumine (tekst teek.asm-is)
      mov   n,ax        AX-s on readint'i loetud int-arv, see → n
      mov   ax,1        1 → AX
      mov   F,ax        (AX)→F
      mov   ax,0        0 → AX
      mov   I,ax        (AX) → I
M1:   mov   ax,I        (I) → AX
      add   ax,1        (AX)+1 → AX
      mov   dTv0,ax     (AX) → dTv0
      mov   ax,dTv0     (dTv0) → AX
      mov   I,ax        (AX) → I
      mov   ax,I        (I) → AX
      cmp   ax,n        võrdlemine: (AX)¤(n)
      jg    M2          kui (AX)>(n), mine M2
      mov   ax,F        (F) → AX
      mov   dx,I        (I) → DX
```

```
        mul   dx          korrutamine: (AX)*(DX) → AX
        mov   dTv0,ax     (AX) → dTv0
        mov   ax,dTv0     (dTv0) → AX
        mov   F,ax        (AX) → F
        jmp   M1          mine M1
M2:     mov   ah,9h       taas stringi väljastamine: int21:9
        mov   bx,1
        mov   cx,8
        mov   dx,OFFSET Trykk
        int   21h
        mov   ah,9h       veel väljastamine: string F_S
        mov   bx,1
        mov   cx,2
        mov   dx,OFFSET F_S
        int   21h
        mov   ax,F        (F) → AX
        mov   dx,0        0 → DX
        cmp   ax,0        (AX) = 0?
        jg    s1h2o3w     kui ei, siis mine trükkima
        mov   dx,1        faktoriaal:=1 (0!=1)
s1h2o3w:
        mov   ch,1        minimaalne trükitava arvu pikkus (parameter)
        call  bin2dec     teisendab faktoriaali sümboliteks ja väljastab
        mov   ah,4ch      int21:4c väljub alamprogrammist
        int   21h
        END   ProgramStart      pseudodirektiiv: asm-teksti lõpp
```

Tähelepanelik lugeja leiab P6.asm-i tekstist üsna palju liigset koodi ─ sellist, mida ta ise ei kirjutaks. Asi on selles, et kood on genereeritud kompilaatori režiimis, mis ei püüa optimeerida. Trigol-kompilaatori optimeerivat režiimi käsitleme hiljem.

Teine assemblertekstifail on lisas 1. See sisaldab ülalpoolkasutatud sisend- ja väljundmoodulite tekste. Sisendmoodul *bin2dec* on laenatud *asm*-tekstis viidatud raamatust, teine aga (*readint)* on nende ridade autori seni ainuke *Inteli* assembleris kirjutatud kood. *Teek.asm* on samuti *TASM*i abil transleeritud ning selle tegevuse resultaatfail on *TLINK*i poolt lisatud *P6.exe* koodi.

Niisiis, moodulid P6.asm ja teek.asm on transleeritud eraldi (resultaadid on *P6.obj* ja *teek.obj*); translaatorilt võib tellida täiendavalt ─ näiteks silumise huvides ─ muidki faile, näiteks nimelaienditega *.map*, *.lst* ja *.xrf*. Komplekteerija teeb noist *.obj*-failidest paigaldatava faili *P6.exe*. Et mitte minna liiga detailseks, ärgem vaadelgem .obj- ja .exe-failide füüsilist esitust (seda saab lugeja hõlpsasti ise näha, kasutades näiteks programmi *gVim* teeneid) ning esitame lisas 2 *Borg*i disassembleri (vabavarana saadav[1] programm *BORG.EXE*) abil P6.exe-st saadud programmi koodi puudutava osa. *Disassembler* on retransleeriv programm, nood on programmid, mis saavad ette translaatori väljundi ning taastavad sisendi. Kuivõrd *P6* nimede tabel pole disassemblerile kättesaadav, siis on kõik etiketid asendatud genereeritud märgenditega. Meie poolt lisatud kommentaarid on esitatud kursiivis (veelkord, vt. lisa 2).

[1] Vt. [68]

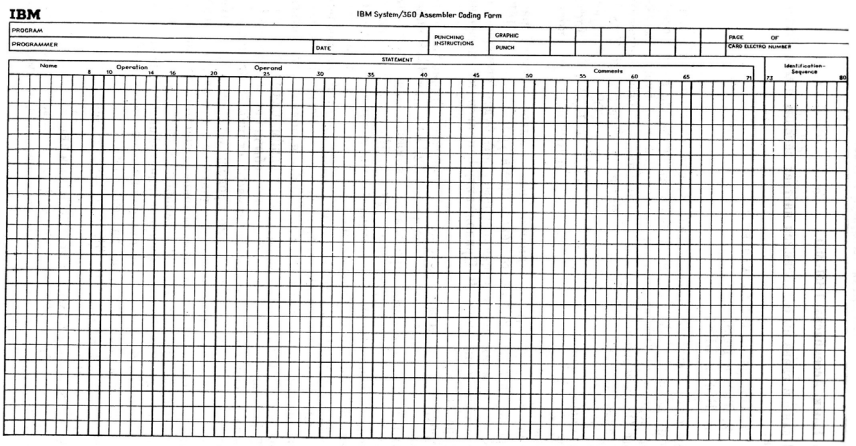
2.6. Assembler ja makroassembler

2.6.1. Assembler

Allpool püüame anda põgusa ülevaate assemblerkeelte[1] ühisjoontest.

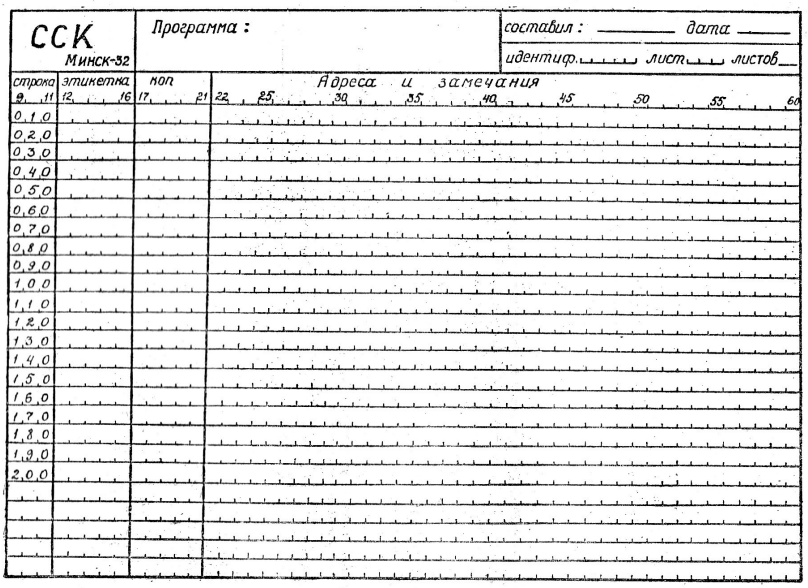
- *Formaat*. Kuivõrd enamik assemblereid olid mõeldud *mainframe*-arvutitele, kus programmeerija ja masina vahel olid vahendajad: perforeerija ja operaator eeskätt, siis oli loomulik, et assemblerprogramm kirjutati spetsiaalsetele plankettidele (vt. jooniseid 2.6.1a ja 2.6.1b). Tänapäeval kasutame *plain*-teksti redaktoreid ja enda huvides tabulatsiooni. Formaaditi võib assembler olla kas fikseeritud, vaba või kombineeritud formaadiga. Suvalise variandi puhul jaguneb iga direktiiv viieks osaks: *märgend* (etikett), *mnemokood, aadressosa, kommentaar* ja rea *identifikaator.* Iga konkreetne assembler paneb paika osade järjekorra ja erinevused seisnevad väljadevaheliste eraldajate ja väljade pikkuste paikapanemises. Vaba formaat kasutab eraldajaid, fikseeritud formaadi puhul on fikseeritud iga välja algus- ja lõpp-positsioonid ning kombineeritud variandi puhul on direktiivi algus fikseeritud, ent lõpuosa on vaba formaadiga (tavaliselt on selles osas eraldajaks tühik). Rea identifikaator on tavaliselt kümnendarv ja tavaline soovitus on, et kahe identifikaatori vaheline samm poleks 1, vaid sootuks rohkem ─ nii on hõlpus käske vahele lisada, aga silumise käigus programmi parandamine on vältimatu.
- *Kommentaar* on oluline komponent (meenutame: masinkood seda ei võimalda). See on (soovitavalt) asjakohane tekst, mis hõlbustab programmist arusaamist: nii autoril, kui ta peab juba unustatud koodi parandama, kui ka programmeerijal, kes võtab üle „vana toote" saatmise. Kommentaar ei satu masinkoodi-faili ja näha on teda ainult assemblerprogrammi listingus. Rusikareegel: sisulisi kommentaare pole kunagi liiga palju! Enamik assemblereid lubavad mitmerealisi kommentaare; tavaliselt tähistab kommentaaririda sümbol „*" rea alguses. *Intel*i assembleri kommentaaririda algab sümboliga „**;**".
- *Märgend* (etikett) on vist parim, mida assembler pakub masinkoodi-mehele: mälujaotus jääb translaatori kanda. Viimane asendab etiketi objektprogrammis märgendatud objekti suhtaadressiga. Enamik assemblereid lubab aadressväljas kasutada aritmeetilisi avaldisi *á la* A+16, ja spetsiaalsümbolit (tavaliselt *) tähistamaks antud käsu aadressi. Näiteks, võrdväärsed on direktiivid SIIN MINE SIIN + 12 ja MINE *+12.
- *Literaal* on käsku kirjutatud vahetu operand niisuguses käsuformaadis, mis vahetut operandi ei luba. Niisugune „libaoperand" kirjutatakse translaatori poolt objektprogrammi konstantide välja ning käsku kirjutatakse tema aadress. Selline operand nõuab erikohtlemist, so hoiatust translaatorile. Minsk-32 hoiatas arvliteraalide eest märgiga (meenutagem: СУ 4,+16), IBM aga markeriga „=", näiteks C 7,=F'11'.

[1] Vahepalana: kas saab, ja kui, siis kuidas defineerida assemblerit? Selle õppevahendi koostaja luges kunagi ühest raamatust (mille autor on paraku ununenud), et assembleriga on lood umbes samuti nagu koertega: välimuselt erinevad nad "seinast seina", ent ometi ei teki mingit raskust otsustamisel, et see siin on koer ja see seal pole. Umbes nii, et buldog *on* koer ja hunt *ei ole.* Ja et *FORTRAN ei ole* assembler (saksa lambakoer võrdluses hundiga..).



Р и с. 17.2. Бланк программирования на языке Ассемблера.

Joonis 2.6.1a. *IBM* assembleri plankett ([13], lk. 586)



Joonis 2.6.1b. *Minsk-32* assembleri plankett ([40], lk. 54).

- *Mnemooniline* („meeldejääv" ?) käsukood. Näiteks, arvu *47* on meil arvatavasti üpris raske seostada lähetamisega, aga sõna *branch* (*ik*. hargnema, harunema jmt) lühend *BR* tekitab tollest keelest aru saajates vististi assotsiatsioone. Või *SRL* ─ *Shift Right Logical.*
- *Operandide kirjutamine.* Reeglina järgib assembler oma masinkoodi formaati. Näiteks, *IBM*i masinkoodi RX-formaat lubas operandi aadressi kirjutamiseks kasutada ülimalt kolme komponenti ─ baasaadress *B*, indeksregister *X* ja nihe *D.* Komponent *D* võib olla kas kümnendarv, sümbol „*" või etikett, millele võib lisanduda +/- nihe. Kui *D* on kümnendarv, siis on ta ainus kohustuslik komponent. Näiteid:

```
LR    3,7          (R7) → R3
L     3,16(4,5)    X=4,  B=5, D=16
LA    4,7          arv 7→ R4
```

- *Pseudodirektiivid.* Nende mnemokoodidele ei vasta programselt interpreteeritavaid masinkoodikäske; neid kasutatakse mitmesugustel eesmärkidel:
    1. *Mälu reserveerimiseks* muutujatele ja vektoritele. *Minsk-32* assembleris näiteks A P3B 1 või B P3B 44. Või IBMi assembleris Y DS CL7 või X DS 44F (vektor 44st *int*-arvust).
    2. *Konstantide defineerimiseks*. Arenenud assemblerid võimaldavad kirjeldada arvkonstante (lubatud on kõik aparatuurselt toetatavad tüübid), stringe ja aadresskonstante (mille väärtuse paneb paika *paigaldaja*). IBMi standardis tuli konstante esitada järgmises järjekorras: *kordsus, tüüp, pikkuse modifikaator* ja *konstant* (või *konstandid*). Aadresskonstandid paigutati sulupaari ( ) vahele, muud apostroofide vahele. IBMi konstanditüübid olid *F,H* (*int*), *E,D* (*real*), pakitud ja pakkimata kümnendarvud (*P,Z*), *binary* (*B*), stringid (*C*) ja aadresskonstandid (*A,Y,S,V*).
    3. *Translaatori töö juhtimiseks.* Sellesse rühma võib (olenevalt arvuti võimalustest ja assembleri lahendusest) kuuluda üsna palju erinevaid direktiive. IBMi assemblerit näiteks tuues: *USING *,12* teatas, et programmi baasaadress on R12-s, *CSECT* tähistas programmisektsiooni ja *DSECT* fiktiivse piirkonna algust, *COM* ühispiirkonna algust, *COPY <nimi>* kopeeris teegist assemblerteksti-faili transleeritavasse programmi ja *END* tähistas assemblerprogrammi teksti lõppu.

2.6.2. Makroassembler

2.6.2.1. Makrovahendid

Makrovahendite all mõistame *makrokeelt* ja *makrogeneraatorit* ─ makrokeele interpretaatorit, seejuures ei pruugi makrokeel olla üldse programmeerimiskeel (vt. [23]). Makrokeel on vahend, mis võimaldab etteantavate parameetrite abil genereerida tekste. Selle raamatu kontekstis huvitavad meid programmeerimiskeelte, eeskätt assembleri makrovahendeid.

Alustagem kolmest tähtsast (makro*assembler*i) mõistest:

- *Makromäärang* on makrokeeles kirjutatud *makrokäsu* kirjeldus koos genereerimiseeskirjadega.
- *Makrokäsk* kirjutatakse assemblerprogrammi võrdväärselt tavaliste assembleri direktiividega; makrokäsu formaadi määrab makromäärang.
- *Makrolaiend* on makrokäsu asemele genereeritud assemblerdirektiivide jada.

Assembleri makrogeneraator töötab tavaliselt *preprotsessorina*: assemblertranslaator teeb enne oma tavapärast tööd teksti läbivaatuse, mille käigus asendatakse leitud makrokäsud nende laienditega ning edasi on kogu töödeldav tekst translaatori jaoks tavaline assemb-

lerkeelne tekst. Allpool tutvume[1] eeskätt *IBM*i suurarvutite makrovahenditega, hiljem vaatleme põgusalt *Inteli* assembleri vastavaid võimalusi. *IBM*i eelistame kahel põhjusel: esiteks, *IBM/360* makrokeel on (vähemalt nende ridade autorile) teadaolevalt võimsaim ja teiseks, opsüsteem *OS/360* oli mõningatel hinnangutel senitehtutest suurim tarkvarakomplekt ning märkimisväärse osa sellest moodustasid süsteemsed *makrod* (nii nimetatakse programmeerijate seltskonnas (aga ka kirjanduses) tavaliselt kontekstist sõltuvalt nii makromääranguid, makrokäske kui ka neid mõlemaid üheskoos; makrolaiendid ei paku tavaliselt erilist huvi).

2.6.2.2. *OS/360* makrovahendid

OS/360 makrod on jaotatavad kasutaja seisukohalt *süsteemseteks* ja *kasutaja-makrodeks. Süsteemsed makrokäsud* kuuluvad operatsioonisüsteemi koostisse; viimane liigitub funktsionaalseteks alamsüsteemideks. Enamusega neist on võimalik assemblerprogrammi tasemel suhelda vastavate makrokäskude abil. Tulenevalt sellest rühmitatakse makrokäsud järgmiselt:

- Superviisori makrod (programmkatkestuste eratöötlus, dünaamiline mälujaotus, ajateenistus, alamprogrammide poole pöördumine (sh. programmide dünaamiline väljakutsumine), avariisituatsioonide töötlus jm.).
- Andmeohjamise makrod (andmekogumite kirjeldamine, infovahetus standardsete välisseadmetega).
- Assemblerprogrammide silumise süsteemi *Testran* makrokäsud.
- Tekstikuvarite makrod.
- Graafiliste kuvarite makrod.

Loetelu pole ammendav.

Pisut teisest aspektist ─ millist tüüpi on genereeritav makrolaiend ─ võime süsteemsed makrod jagada kolme rühma:

- Käsud superviisori poole pöördumiseks (*SVC* = *SuperVisor Call*). Need valmistavad ette pöördumisparameetrid ning annavad üle juhtimise väljakutsutavatele spetsiaalsetele *SVC*-programmidele; siia rühma kuulub enamus superviisori makrodest.
- Makrod alamprogrammide poole pöördumiseks. Needki valmistavad ette parameetrid ja annavad juhtimise üle programmidele, mis ei kuulu *SVC*-programmide hulka. Siia rühma kuuluvad mh. andmeohjamise makrod.
- Makrod tabelite moodustamiseks ─ need on tavaliselt ühised parameetrite tabelid makrokäskudega kaetud programmide jaoks (näiteks andmeohjamise programmid kasutavad *DCB*-plokki (*Data Control Block*), mis genereeritakse samanimelise makrokäsu abil).

[1] Selles jaotises tugineme brošüürile [23] ning teemaga põhjalikumalt tutvuda soovivad huvilised võiksid seda lugeda; ka on seal nende materjalide loetelu, mis olid brošüüri koostamisel abiks. Suures osas tugines tolle autor enda kogemustele.

Kuivõrd operatsiooni *genereerimise* ajal konkreetsele arvutile (sedagi tehakse nn. genereerimis*makrode* abil) on võimalik osa vabasid *SVC*-numbreid reserveerida arvutuskeskuse kohalike *SVC*-programmide jaoks, siis võivad ka *kasutaja-makrod* kuuluda suvalisse ülaltoodud rühma.

Kasutaja-makrokäsud võivad olla kirjutatud kas individuaalseks või siis üldiseks, suurema meeskonna poolt kasutamiseks. Esimesel juhul hõlbustavad nad vaid nende kirjutaja tööd, teisel juhul aga pakuvad huvi kasvõi tervele arvutuskeskusele (mis noil aegadel oli koondunud ühe suurarvuti ümber). Lisaks *SVC*-võimalustele kirjutati kasutajamakrosid peamiselt

- Rutiinsete tööde vältimiseks.
- Makromäärangute kirjutamise hõlbustamiseks (näiteks omaloominguline *W*).
- Fiktiivsete sektsioonide kirjeldamiseks.

Mõistagi pole seegi loetelu ammendav, näiteks võib makrovahendeid kasutada viitade tabeli genereerimiseks, silumisandmete tekitamiseks, funktsiooni väärtuse arvutamiseks jne. jne.

Selle raamatu eesmärkide hulka ei kuulu *OS/360* makrokeele õpikuks olemine, sestap toogem pelgalt näiteid tolle süsteemi makrodest. Makromäärangu formaat järgis assemblerit: erandina võib käsukood olla pikem kui assembleri 5 sümbolit, seega tühikutega eraldatud „vaba formaat" algab juba koodiväljast. Näidetes on kommentaarid esitatud *kursiivis*.

Makro *W*.

See makro oli mõeldud kasutamiseks eeskätt makromäärangutes ─ selle abil genereeritakse parameetrite väärtuste edastamise direktiivid parameetrite suvalise assembleri poolt lubatud esitusviisi puhul[1]. Selle käsu formaalne kirjeldus on järgmine (fiktiivsed parameetrid on kandilistes sulgudes ning makrokäsu parameetri marker on „&"):

[märgend] W &P,&K[,&ST]

Makrokäsk *W* kannab parameetri *P* väärtuse aadressile *K*. *P* esitamiseks on järgmised võimalused:

- Parameetri väärtus on suvalises üld- või ujupunktregistris, näiteks (1), (0) jne.
- Parameetri väärtuse aadress on kirjutatud *RX*-formaadis, näiteks PEE(8) või 4(6,1). Või A+12.
- Parameetri väärtus on esitatud märgita kümnendtäisarvuna vahemikust 0..4095.

[1] Siinkirjutajat ajendas *W*d kirjutama reaalne vajadus − millegipärast polnud *OS/370* makroteegis selle töö jaoks midagi, ja jahmatas tutvumine makro *DCB* määranguga (väljatrükis ca 50 laitrükilehekülge): viimane kui üks parameeter oli rutiinselt „lahti kaevatud" (NB! *Copy-paste-meetod*it veel polnud). Selmet kirjutada (W-ga) analoogiline makro − sel juhul piisanuks *IBM*il *DCB* makromääranguks paarist leheküljest.

- Parameetri väärtus on kirjutatud makrokäsku literaalina, näiteks =’XL4’F’ või =A(B).
- Parameetri väärtus on antud *SI*-tüüpi (so., vahetu) operandina, näiteks C’?’, X’FF’ või B’10011100’.

K (sihtkoha) esitamiseks on kaks võimalust, see võib olla kas
- Registri number sulgudes või
- *RX*-tüüpi aadress.

Fakultatiivne parameeter *ST* näitab ülekantava parameetri väärtuse pikkust baitides (vaikimisi on selleks 4 baiti). Selle kodeerimisvõimalused on järgmised:
- *C* ─ 1 bait.
- *H* ─ 2 baiti.
- *E* ─ ujupunktarv, 4 baiti.
- *D* ─ ujupunktarv, 8 baiti.

Tabelis 2.6.2.2a on toodud mõned *W* kasutamise näited.

Makrokäsk	**Makrolaiend**
W (1),(0)	LR 0,1
W (1),4(2),H	STH 1,4(2)
W 8(3,4),12(5),D	LD 0,8(3,4) STD 0,12(5)
W 3(6),A,C	IC 0,3(6) STC 0,A
W 0(5),B+8	L 0,0(5) ST 0,B+8
W 0,A	LA 0,0 ST 0,A
W =V(PROG),(9)	L 9,=V(PROG)
W =V(PROG),A	L 0,=V(PROG) ST 0,A
W =C’+’,C,C	IC 0,=C’+’ STC 0,C
W (0),(2),E	LER 2,0
W (0),(2),D	LDR 2,0
W X’FC’,3(5),C	MVI 3(5),X’FC’
W 25(7),(1),C	XR 1,1 IC 1,25(7)

Tabel 2.6.2.2a. Makrokäsu *W* näited.

W **makromäärang** on järgmine *IBM/360* makroassemblerkeelne tekst:

```
        MACRO       makromäärangu algus
&N      W       &P,&K,&ST   &N näitab, et käsku tohib märgendada
.*  MÄRGENDATUD TYHIDIREKTIIVI GENEREERIMINE JUHUL; KUI „&N"
.*  POLE TYHI   kommentaarirea marker on " .*"
        AIF     (N'&N EQ 0).YLE   makrokeeles: kui &N kohtade arv=0, mine .YLE
&N      EQU     *           asm-direktiiv (mudeloperaator)
.YLE    ANOP                makrokeeles: sisemärgend+tühidirektiiv
.* KUI P VQI K PUUDUVAD, SIIS MAKROLAIENDIT EI SAA!
        AIF     (N'&P NE 0 AND N'&K NE 0).GO
        MEXIT               lõpeta laiendi genereerimine
.* KUI OPERAND AGAB SULUGA, SIIS AINUVÕIMALIK ÕIGE KIRJAPILT ON
.* (R)
.GO     AIF     ('&K'(1,1) EQ  '(').REGK   kui K  algussümbol on (, mine .REGK
        AIF     ('&P'(1,1) NE  '(').RX     kui P  algussümbol pole (, mine .RX
        ST&ST   &P(1),&K    mudeloperaator (R)→A
        MEXIT               lõpeta laiendi genereerimine
  .RX   AIF     ('&P'(K'&P,1) NE  ')').OTSE  kui P lõpusümbol pole ), mine .OTSE
.* KUI P ALGUSSYMBOL POLNUD"(„ LÕPUSYMBOL AGA ON „)", SIIS ON TA
.* KIRJUTATUD RX-FORMAADIS; NÄIT. A(4), 4(5,8) VÕI A+8(6).
.STORE  AIF  (N'&ST EQU  0).LST    kui ST on tühi, mine .LST
        AIF     ('&ST'(1,1) EQ  'C').IC  kui ST=C, mine .IC
.* KUI  ST  ON TYHI; SIIS GENEREERITAKSE „L" JA „ST"; MUIDU AGA NÄI-
.* TEKS „LH" JA „STH" JNE.
.LST    L&ST  0,&P    mudeloperaator
.ST     ST&ST  0,&K   mudeloperaator
        MEXIT         lõpeta
.IC     IC      0,&P  mudeloperaator
        AGO   .ST     makrogeneraatori suunamine
.* KONTROLL: KAS „P" ON LITERAAL (ALGAB „="-GA)?
.OTSE AIF     ('&P'(1,1) EQ  '=').STORE   kui on literaal, mine .STORE
.* KUI „P" LÕPUSYMBOL POLE „'", SIIS MINE .ARV
        AIF     ('&P'(K'&P,1) NE  '''').ARV
        MVI   &K,&P   apostroofiga võib lõppeda ainult vahetu operand; mudeloper.
        MEXIT         lõpeta laiendi genereerimine
.* KONTROLL: KAS"P" ALGAB TÄHEGA? KUI JAH; SIIS RX-FORMAAT
.ARV  AIF     ('&P'(1,1) GE  'A' AND '&P'(1,1) LE  'Z').STORE
        LA      0,&P  mudeloperaator, eeldame, et P on täisarv
        AGO   .ST     makrogeneraatori suunamine
.* KUI „P" ALGAB SULUGA; SIIS PEAB TA OLEMA REGISTER (R)
.REGK  AIF    ('&P'(1,1) NE  '(').RXP
        AIF     (N'&ST EQU  0).LR   kui ST on tühi, siis mine .LR
.* UJUPUNKTREGISTRITE RR-TEHE ON „LER" VÕI „LDR"
        AIF     ('&ST'(1,1) NE  'E' AND '&ST'(1,1) NE 'D').LR
        L&ST.R  &K(1),&P(1)  mudeloperaator, kood tehakse konkatenatsiooniga
```

```
        MEXIT       lõpeta laiendi genereerimine
.*  YLDREGISTRITE RR-TEHE
.LR     LR      &K(1),&P(1)  mudeloperaator
        MEXIT       lõpeta laiendi genereerimine
.* KUI „P" LÕPUSÜMBOL POLE „)", SIIS MINE .OP
.RXP    AIF     (′&P'(K'&P,1) NE  ')').OP
.LOAD           ANOP    makroassembleri tühidirektiiv märgendi jaoks
        AIF     (N′&ST EQU  0).LP
.       AIF     (′&ST'(1,1) EQ  'C').PIC
.LP     L&ST  &K(1),&P      mudeloperaator: (mälu)→R
        MEXIT       lõpeta laiendi genereerimine
.PIC    XR      &K(1),&K(1)  mudeloperaator: 0→R
        IC      .&K(1),&P      mudeloperaator: sümbol (mälu)→R
        MEXIT       lõpeta laiendi genereerimine
.* KUI „P" ON LITERAAL, SIIS TÖÖTLEME TEDA NAGU RX-OPERANDI
.OP     AIF     (′&P'(1,1) EQ  '=').LOAD
.* „P" POLE LITERAAL; KUI LÕPUSÜMBOL ON „'",SIIS PEAB TA OLEMA
.* VAHETU OPERAND, REGISTRISSE LOEME TA LITERAALINA
        AIF     (′&P'(K'&P,1) NE  ''''').ETIK
        IC      &K(1),=&P    mudeloperaator
        MEXIT       lõpeta laiendi genereerimine
.* KUI „P" ALGAB TÄHEGA, SIIS ON TA RX-OPERAND; MUIDU EELDAME
.* KÜMNENDTÄISARVU
.ETIK  AIF     (′&P'(1,1) GE  'A' AND ′&P'(1,1) LE  'Z').LOAD
        LA      &K(1),&P      mudeloperaator
        MEND        makromäärangu lõpp
```

2.6.2.3. *IBM/360* (ja */370*): kokkuvõte

Meie kontekstis on *IBM*i *mainframe*-masina arhitektuur üleminek pesamasinatelt mikroprotsessoritele. Selle arvutite „perekonna" masinkood ja assembler on lihtsad, loogilised ning läbipaistvad, makroassembler on võimsaim kõigist senituntuist (vististi ka järeltulijatest) ning assembleri ja riistvara-taseme (katkestused, alamprogrammid, mälujaotus, välisseadmed jne.) vahendajaiks olid *makrod.*

Selle klassi masinad olid mõeldud töötamaks *tööjaotusrežiimis*: op-süsteem pidas kasutajate järjekorda, arvestas kasutajate prioriteetidega ning jagas kasutajatele järgemööda ressursse; tähtsaimad neist olid op-mälu ja protsessoriaeg. Välisseadmete järele võis oodata.

Nentigem, et nende masinate *süsteemse tarkvara* tegemise keel oli (makro)assembler. Firma toetatavad keeled ─ *FORTRAN, COBOL* ja *PL\I* ─ ei sobi(nud) ükski selle valdkonna jaoks. Süsteemse tarkvara kirjutamise kõrgtaseme keeled (eeskätt *Forth* ja *C*) tulid teistelt platvormidelt ja firmadelt, aga neid käsitleme hiljem.

2.6.2.4. *Intel*i makrovahendid

*Intel*i assembler ja makroassembler pole enam tarkvara kirjutamise olulised vahendid. Assembler võib olla oluline kompilaatori programmeerimisel (tänapäeval on tavaline, et kõrgtaseme keele kompilaator genereerib esmalt assemblerteksti, mis antakse ette assemblertranslaatorile ja too omakorda linkerile .exe-faili tegemiseks). Ja assembleri makrovahendid ei evi enam sellist tähtsust nagu *IBM*-arhitektuuri kõrgajal. Veel kord, ilmusid keeled, mis võimaldavad kirjutada efektiivset „koodi" kõrgtaseme keeles.

Sestap on *Intel*i makrovahendid suhteliselt tagasihoidlikud, selle makrovõimalusi võib grupeerida järgmiselt [37, lk. 213 jj]:

- *sünonüümid* (*equates,* EQU), näiteks:

```
        columnEQU   80
        row     EQU   25
        screenful  EQU  column*row
        line      EQU  row
        .DATA
Buffer DW     screenful
        .CODE
…
        mov    cx,column
        mov    bx,line
```

- *Makrod*

  Nende esitusviis on järgmine: **MACRO** avaldised **ENDM**
  Näiteks: [37, lk. 219]:

```
        Addup MACRO        ad1,ad2,ad3
                mov    ax,ad1
                add     ax,ad2
                add     ax,ad3
                ENDM
Makrokäsk on näiteks
                addup  bx,2,count
ja makrolaiend:
                mov    ax,bx
                add     ax,2
                add     ax,count
```

- *Korduvad blokid*
  Siin on kaks varianti, esiteks selline, et mingi *avaldis* määrab, mitu korda mida tuleb assemblertekstis korrata. *REPT*-variandi näide on järgmine [37, lk 224]:

```
Alphabet    LABEL    BYTE
  X         =        0
            REPT     26
            DB       'A'+X
  X         =        X+1
            ENDM
```

Teine variant on *IRP*- direktiiv. Näiteks:

```
numbers  LABEL byte
         IRP   x,<0,1,2,3,4,5,6,7,8,9>
         DB    IO DUP(x)
         ENDM
```

Makrolaiendisse läheb etiketile *numbers* 10 baiti väärtustega '0', '1' jne.

- *Makrooperaatorid*
  Nood on avaldised, kus on kasutatavad järgmised tehted (operaatorid):
  & ─ formaalse parameetri asendamine jooksva väärtusega;
  <tekst> ─ see on sisuliselt mingi parameetri tegelik väärtusvaru, mida saab kasutada muudel otstarvetel.

- On veel *struktureeritud, rekursiivsed* ja *üledefineeritud* makrod; Lugeja võib tutvuda detailidega nt [37, lk. 230.]. Ent, ei midagi sellist, mis ületaks *OS* makroassembleri võimalusi.

2.7. Masinorienteeritud keelte kokkuvõtteks

2.7.1. Aparatuurne ja programne interpretatsioon

Masinkoodi interpreteerib *aparatuurselt* protsessor (juhtimisseade ning aritmeetika- loogikaseade). Põhimõtteliselt võibki masinkoodis programmeerida, silmas pidades ainult aparatuurset interpretatsiooni, kirjutades kuitahes pikki tervikprogramme, kus puudub hierarhiline struktuur, mis tekkib alamprogrammide lubamise ja kasutamisega (naasmisega suunamiskäsud alamprogrammide puu tekitamiseks päris hästi ei sobi). Alamprogrammi all mõeldakse ikkagi eeskätt autonoomselt kirjutatud ning põhiprogrammi juurde komplekteeritud koodi, mille kirjeldus määrab, millised ja kuidas edastatud peavad olema sisend- ja millised väljundparameetrid (ja lisaks ühisväljade võimalused).

Alamprogrammide (nii süsteemsete kui ka kasutaja kirjutatute) kasutamiseks ei piisa koodi aparatuursest interpreteerimisest; lisanduma peab *programne* interpretatsioon: masinkoodi-programmi kirjutatakse nn. *pseudokoodis,* mida ei anta protsessori kätte, ent mida interpreteerivad (eeskätt) *komplekteerija* ja *paigaldaja* ─ aga need on programmid, mis interpreteerivad masin- ja pseudokoodist koosnevat andmekogumit.

Komplekteeritavate ja paigaldatavate masinkoodi-programmide kirjutamiseks kehtestatakse reeglina teatud *kokkulepped,* mille järgimine tagab programse interpretaatori toimimise.

Assembler ei lisa selles kontekstis masinkoodile praktiliselt mitte midagi; lihtsustub ainult pseudokoodide meelespidamine ja kirjutamine. Makroassembler aitab *kokkulepetest* kinni pidada.

Ent korrakem: assembler ei võimalda midagi sellist, mida ei saaks masinkoodis kirjutada, ja makroassembler ei paku midagi, mida ei saaks assembleris kirjutada. Vahe on programmeerimise tehnilises keerukuses ja tööviljakuses.

2.7.2. Moodulid

Ehkki Arvutileksikon [11, lk. 230..231] peab *mooduliks* suvalist reeglipäraselt kirjeldatud alamprogrammi (fikseeritud peab olema *liides* ─ sisend- ja väljundparameetrid, töö ja (võimalikud) alamprogrammid), siis nende ridade autor toetab pisut kitsamat interpretatsiooni. Ja kitsendus on: mooduliks võime pidada ainult *autonoomselt transleeritud* ja suvalises keeles kirjutatud, põhiprogrammi koodi juurde komplekteeritavat iseseisvat (alam-)programmi.

Mooduli mõiste on ammu tuntud tehnikas, ja sama semantika on tal ka IT-valdkonnas: moodul on standardiseeritud ja rahvusvaheliselt tunnustatud detail, mida võib nagu legoklotsi sobitada kuhu iganes ─ kui ainult vastav valdkond on niisuguseks sobitamiseks avatud ja valmis.

Moodulprogrammeerimise ideoloogia „maaletooja" on Eestis suure tõenäosusega *Leo Võhandu*; just sellest printsiibist lähtudes loodi lühikese ajaga Eesti Raadio jaoks (masinale *Razdan-3*) ankeet-tüüpi informatsiooni töötlemise süsteem *СОДИ ─ Система Обработки Дискетной Информации*, millega tegeleti 60.-70.-ndate aastate vahetusel. Tema meeskonna arusaamised moodulitest olid (tegijate arvamusi üldistades) järgmised:

- Moodul on ainult ühte funktsiooni täitev lihtne, täpselt määratletud liidesega iseseisev programm[1] (mis võib piiramatult pöörduda „asja huvides" teiste moodulite poole) Käitumine veasituatsioonides peab olema ühene ja täpselt dokumenteeritud. Moodul ei tohi ennast täitmise käigus muuta. Sisend- ja väljundoperatsioonid peavad jääma selleks mõeldud moodulite hoolde.
- Moodul pole reeglina pikem kui 100 masinkoodikäsku (assemblerdirektiivi); mida lühem, seda parem. Kommenteerigem: kogemused näitasid, et selle pikkusega „juppe" võis korraga olla töös paar- kolmkümmend (suhtlemine arvutiga

[1] *Glenford J. Myers* [38, lk. 92] rõhutab, et moodulprogrammeerimine on traditsiooniline vahend võitluses programmsüsteemide keerukusega, ja et seejuures on kaks "ohupunkti" ja nende tulem:

─ moodulid täidavad omavahel (loogiliselt) seotud, ent paljusid erinevaid funktsioone (mis hägustab nende loogikat);

─ süsteemi projekteerides on „kahe silma vahele" jäänud paljud ühisosad, ja need on erinevates moodulites programmeeritud erinevalt (ehkki nood "ühisosad" tulnuks kirjutada moodulitena);

─ ja tulemus: moodulid töötlevad ühiskasutuses olevaid andmeid ettearvamatult erinevalt.

käis dispetšeri, perforeerijate ja operaatorite vahendusel) ja siledaks said nad kõik kokku mitte kauem kui päeva- paariga. Tuhandekäsuline programm võttis aega paarist nädalast paari kuuni, veel 10 korda pikemale programmile kulub ca paar aastat, ja miljonikäsune kompaktne programm ei saagi valmis.

Niisiis: moodul on autonoomne, *eraldi transleeritud* alamprogramm, mille poole saab pöörduda (üldistatult) *CALL*-operaatoriga ja mille kood pole ei põhiprogrammi translaatorile, komplekteerijale ega ka paigaldajale semantiliselt kättesaadav, ja see teeb suuresti võimatuks interaktiivse silumise ning *kõik-vead-avastava* transleerimise (ja paraku läbiva optimeerimise ─ kompilaator peab lootma eralditransleeritud moodulite optimeeritusele, või siis masinkoodi optimeerimisele − mis on paraku komplitseeritud).

Ent moodulprogrammeerimise võimsus seisneb moodulite *teekide* kumuleerumises: mingi ülesande lahendamise käigus kirjutatud moodulitest saab üsna suurt osa kasutada järgmiste projektide jaoks ja nii koguneb neid aja jooksul mingil töörühmal üha täielikum komplekt. Iga järgmise ülesande lahendamine on seega suhteliselt vähem töömahukas kui eelmiste puhul. Masinorienteeritud keeltes loodud tarkvara „elutsükkel" on paraku sõltuv objektmasina omast. Kõrgtaseme keeled arvutist ei sõltu ning moodulprogrammeerimise efektiivsuse hea näide on *FORTRAN*i moodulite teegid (ja nonde kasutamist toetavate keeleversioonide jätkuv elujõulisus).

*FORTRAN*i teegid on rahvusvahelised ja vabalt kättesaadavad[1] ja need sisaldavad väga palju mooduleid paljude arvutusmatemaatika valdkondade tarbeks. Viiskümmend aastat on märkimisväärne aeg, mille jooksul noid teeke on jätkuvalt täiendatud.

2.7.3. Operatsioonisüsteemi tugi

Oleme põgusalt käsitlenud nelja erineva arhitektuuriga arvutit (või arvutiperekonda): *Razdan-3, Minsk-32, IBM/360* ja „abstraktset arvutit", mis tugineb *Intel*-protsessorile. *Razdan*i operatsioonisüsteem oli peaaegu olematu, *Minsk-32* tagas piisava komplekti standardprogramme (vormistatud moodulitena): teisendusprogrammid, matemaatilised funktsioonid, sisend-väljundprogrammid jm., aga ka võimalused dünaamiliseks mälujaotuseks ning toe alamprogrammide kasutamiseks. Riistvara initsieeritud katkestusi töötles eeskätt protsessor, ent ka programmeerijal oli võimalus katkestuste privaattöötluseks.

IBM/360 operatsioonisüsteem *OS/360* toetas programmeerijat eeskätt *makrode* abil. Riistvara-katkestusi sai kinni püüda makrodega ning tarkvara-katkestusi sai esile kutsuda *SVC*-makrode (*Supervisor Call*) vahendusel. Operatsioonisüsteem toetas nii staatiliste alamprogrammide puu tekitamist kui ka dünaamiliselt paigaldatavaid alamprogramme ning andis paendliku dünaamilise mälu jaotamise vahendid.

[1] Näiteks vt. [47], mille esimeses köites oli valik statistikamooduleid (korrelatsioonanalüüs, mitmene lineaarne regressioonanalüüs, polünomiaalne regressioon, kanooniline korrelatsioon, faktoranalüüs, diskriminantanalüüs, aegridade analüüs, mitteparameetrilised meetodid, juhuarvude generaatorid, jaotusfunktsioonid jm.) ning teine köide esitas maatriks- ja lineaaralgebra programme.

*Intel*ile baseeruv arhitektuur suhtleb välisseadmete ja kasutajaprogrammidega *katkestuste* vahendusel Kasutaja jaoks on talle kättesaadavad katkestused sisuliselt pöördumised opsüsteemi kuuluvate alamprogrammide poole[1], mis paiknevad täitmise ajal kas alati (*BIOS*) või reeglina (*DOS*) sisemälus.

Võime nentida, et mida suuremaid võimalusi pakub operatsioonisüsteem, seda vähem teab programmeerija „täpselt", mida ja kuidas tema programm ikkagi teeb, ja seda isegi masinkoodis programmeerides. Masinorienteeritud keelte puhul ei pane see seik tavaliselt programmeerijat ennast ahistatuna tundma, ent selline tunne tekib üpris tihti, kui kirjutada näiteks *C++*-keeles *Windows*i keskkonna jaoks interaktiivset programmi.

2.7.4. Virtuaalarvuti

Terrence W. Pratt ([44], lk. 42) defineerib pedagoogilistel kaalutlustel „arvutit"[2] algoritmide ja andmestruktuuride integreeritud kogumina, mis on võimeline (mälus) hoidma ja täitma programme, ja selle ülesande täitmiseks on mitu erineva taseme võimalust:

1. Aparatuurne realisatsioon: kõik on esitatud füüsiliste seadmete abil, so kõiki käske interpreteerib vahetult protsessor.
2. Programne-aparatuurne realisatsioon: madalaim on mikroprogrammide tase, kus masinkoodi interpreteeritakse kui mikrokäskude (aparatuurselt interpreteeritavat) jada.
3. Programne modelleerimine: programmid ja andmestruktuurid on esitatud mingis muus (mikro- või masinkoodi suhtes kõrgema taseme) programmeerimiskeeles .
4. Eeltoodud võimaluste kombinatsioon: mingi osa „arvutit" on kaetud aparatuurselt, mingi osa programselt-aparatuurselt ja mingi osa modelleeritakse programselt.

Aparatuurne „arvuti" on *reaalne*. Masin, mida modelleeritakse kas osaliselt või täielikult programmide või mikroprogrammidega, on *virtuaalne* „arvuti". Konkreetsemalt, me saame rääkida mingi *programmeerimiskeele virtuaalarvutist,* kuivõrd peaaegu alati modelleeritakse selles keeles kirjutatud ja selle keele translaatoriga genereeritud objektprogramme mingis ulatuses *programselt* ja iga keel modelleerib semantiliselt ekvivalentseid teiste keelte objekte tavaliselt talle ainuomaselt, kuivõrd pole mingeid standardeid objektkoodile[3]. Võime seda mõista nii, et tavakasutaja võib uskuda, et tema kirjutatud programm transleeritakse üksüheselt masinkoodi ja interpreteeritakse aparatuurselt, ja et see pole üldjuhul sugugi nii lihtne.

Programmeerimiskeel ei määra üheselt keele virtuaalmasinat: viimane sõltub lisaks nii translaatori tegija suvast kui ka objektmasina (arvuti, millel programme „jooksutatakse")

[1] Tegelikult on asi pisut keerulisem (või lihtsam?): protsessor garanteerib täpselt kodeeritud katkestuste "kättesaadavuse", ent ei määra nende töötlemist; *BIOS* ja *DOS* on eeskätt mugavad „vaikimisi-variandid". Me tuleme selle juurde tagasi *FORTH*i tutvustamise käigus.

[2] Jutumärkideta arvuti on füüsiline masin ise ja mitte tema abstraktsioon.

[3] Siiski, tänapäeval on kaks tunnustatud abstraktse arvuti vahekeelt – *Java* baitkood ning *.NET*i versioon. Me käsitleme neid 8. peatükis.

arhitektuurist ning too virtuaalmasin on lahendamisaegne reaalne kood ja andmestruktuurid ─ ja sugugi mitte ainult programmeerija kirjeldatud, vaid ka translaatori tekitatud (näiteks deskriptorid, aga neist hiljem). Üldiselt, näib kehtivat seaduspärasus, et mida „kõrgema taseme" keel, seda väiksem on tolle keele virtuaalmasinas aparatuurse interpretatsiooni osatähtsus.

Kui unustame ära mikrokoodi, siis masinkoodi tasemel on programset modelleerimist vaja eeskätt *komplekteerijale* ja *paigaldajale* nende interpreteeritavate pseudokoodide puhul. Ent kui programmeerimiskeel suhtleb operatsioonisüsteemiga eeskätt *makrode* vahendusel, siis näiteks *OS/360* makroassembleri *virtuaalne* osa võib olla üsna tuntav. Selleni välja, et ainult süsteemseid makrosid kasutav programmeerija võib uskuda, et igale makrokäsule vastabki parasjagu üks masinkoodi-käsk ─ aga sellise näivuse tekitamine ongi heale programmeerimiskeelele iseloomulik.

2.7.5. Programmeerimiskeelte ühisosad ja masinorienteeritud keeled

Veebruaris 1977 toimus Otepää lähistel Veski spordibaasis *Enn Tõugu* organiseeritud ja TRÜ AK organisatsiooniliselt toetatud programmeerimiskeelte talvekool, kus lisaks eestlastele osalesid tegelikud tipptegijad N. Liidust. Sissejuhatavalt olid tahvli ees *Tõugu* ja NSVL TA Arvutuskeskuse üks liidreid, *Vladimir Kurotškin*, ja paika pandi järgmised orientiirid järgnevatele ettekannetele:

1. Andmed. Need jagunevad lihtandmeteks ja struktuurideks; lihtandmed liigituvad tüübiti: *arvud* (nt. täis- ja reaalarvud), *boolean*, *sümbolid, tekst* ja *viidad*;

Struktuurid on näiteks *massiivid, kirjed* või *programmi(tüki)d* ─ nagu *Lisp*is või *SIMULA*s, *puud* ja *graafid*

Masinorienteeritud keeled toetavad andmeid ainult aparatuurselt. *int* 2, 4 ja rohkem baiti, *ujupunktarvud* 4, 8 või rohkem baiti, pakitud (numbrikoht 4 bitti) või pakkimata (numbrikoht 8 bitti*) kümnendarvud* (tavaliselt piisavalt pikad), *boolean* {0 või 1}, *sümbolmuutujad* on kas ühebaidised või stringid (ilmutatud kujul alles baitmasinates). *Viidatüüpi* toetab ilmutatud kujul *OS/360* assembler (*Minsk-32* pakub selleks assembleri aadresskonstanti ─ aga ka juurdepääsu dünaamilisele mälujaotusele assembleri vahenditega).

Struktuure ei toeta ükski masinorienteeritud keel (ehkki indekseerimisvahendid võimaldavad hõlpsasti töödelda vektoreid, ja dünaamilist mälujaotust toetavad assemblerid tekitada ja töödelda viidastruktuure (ahelad, puud ja graafid)).


Enn Tõugu Elbi suvekoolis.

2. Tegevused: aritmeetilised ja loogilised avaldised, omistamine, tüübiteisendused, kirjeldused, protseduurid, funktsioonid, rekursioon jm.

Avaldisi masinorienteeritud keeled ei toeta. Omistamistehte jaoks on mitmeid võimalusi: näiteks salvestamine, saatmine ja vahetamine. Tüübiteisendused (näiteks 10 → 2 ja 2 → 10) on kaetud tavaliselt operatsioonisüsteemi vahenditega, mis on kättesaadavad ka masinorienteeritud tasemel. Kirjeldused on esitatavad alates assemblerist. Alamprogrammid (protseduurid ja funktsioonid, sh. rekursiivsed) on masinorienteeritud keeltes kaetud kas naasmisega suunamiskäskudega või ─ kokkuleppeid järgides ─ üldistatult *CALL*-operaatoriga.

3. Programmi struktuur. Variandid on järgmised: alamprogrammideks liigendamata programm, programmis defineeritud alamprogrammid, plokkstruktuur ─ staatiline või dünaamiline ─, sõltumatud alamprogrammid, jm.

Plokkstruktuure toetavad ainult mõned (masinorienteeritud keeltest) kõrgema taseme keeled (*ALGOL* ja tema otsesed järglased). Sõltumatud alamprogrammid on need, mis lisab põhiprogrammile *komplekteerija* ja teeb kasutatavaks *paigaldaja* ─ need on ka masinorienteeritud keeltes kasutatavad. Programmis defineeritud alamprogrammid on masinkoodi ja assembleri jaoks need koodilõigud, mille kasutamiseks on naasmisega suunamiskäsud.

4. Juhtimine: variandid on lineaarne käskude jada, hargnemine, valik, tsükkel, suunamine, tegevusest väljumine („*exit*"), sündmuste poolt juhitavus, paralleelsus jm.

Lineaarne käskude jada on juba protsessori jaoks loomulik programmistruktuur. Hargnemist ja valikut saab modelleerida võimsa (näiteks *OS/360*) makrokeelega, ent mitte masinkoodi või assembleri „sisseehitatud" tasemel, küll aga tavalise käskude jada abil. Suunamist (kõrgtaseme keelte jaoks tuntud kui *GOTO*-operaator) toetavad kõik masinkoodid ─ too ongi ainus selle taseme keelte võimalus käskude täitmise järjekorra muutmiseks. *Exit* on ka masinkoodi tasemel hõlpsasti realiseeritav. Sündmuste poolt juhitavus on programselt modelleeritav ka masinorienteeritud tasemel, paralleeltöötlust saab üheprotsessorilise arhitektuuri puhul programselt modelleerida, ent mitmeprotsessoriline arhitektuur eeldab üldjuhul ka aparatuurset tuge masinkoodi tasemel.

5. Sisend ja väljund. Selle klassi tööde tegemiseks on reeglina kasutatavad operatsioonisüsteemi vahendid, ent kui meenutada *Minsk-32*, siis ka üsna suur valik assembleri vahendeid. Baitmasinad pakuvad tavaliselt nondeks operatsioonideks tuge katkestuste vahendusel (sageli omakorda makrode vahendusel).

Resümeerigem. Ehkki me ei pidanud mingit arvestust (masinorienteeritud keeled *contra* kõrgtaseme keeled), tundub, et pole just palju asju, mida ei saaks väljendada vahetult masinkoodis või assembleris. Enamgi veel, peame silmas sellise väljendamise *lihtsust*, tegelikult ei saa protseduurorienteeritud (rääkimata probleemorienteeritutest) keeled võimaldada midagi enamat kui masinorienteeritud keeled; kõrgtaseme keelte eelised peituvad paremates väljendusvõimalustes ning nende kasutamisega kaasnevas kõrgemas tööviljakuses. Ja lisaks paljus muus, näiteks programmi loetavuses, muudetavuses, publitseeritavuses jne.

Niisiis, kui me nendime, et mingit konstruktsiooni ei toeta masinorienteeritud keeled, siis mõtleme seda, et puudub aparatuurne ja integreeritud programne (so, lakooniline) interpretatsioon, mida pakuvad näiteks kõrgtaseme keeltes populaarsed operaatorid *switch*, *case*, *for* jmt. Ja mitte seda, et neid ei saaks väljendada masinorienteeritud keeles (**kui see nii oleks, siis ei saaks noid konstruktsioone mitte kuidagi masinkoodi transleerida**)[1].

Miks oli vaja kõrgtaseme keeli. Loodetavasti õnnestus meil näidata, et kõik, mida arvuti võimaldab, on kindlasti programmeeritav masinorienteeritud keel(t)es. Ülalpool käsitlesime seiku, miks masinkoodi kõrvale tuli assembler ja miks mõeldi välja makroassembler: mõlemad rutiinse programmeerimise „mehhaniseerimiseks".

(Makro)assembler andis idee *avaldiste* programmeerimiseks oluliselt kõrgemal tasemel. Kuivõrd lugeja on mingil moel juba tuttav *OS/360* makrokeelega, võiks ta sportlikust huvist püüda kirjutada makromäärangu, mis genereeriks suvalise aritmeetilise (sulg)avaldise väärtuse arvutamise direktiivid, see töö pole just lihtne, ent on täiesti tehtav.

Lõpetagem masinorienteeritud keelte klassi käsitlus kahe olulise mõiste üleräkimisega: need on *interpreteerimine* ja *kompileerimine.*

Masinkoodi *interpreteerib* protsessor aparatuurselt. Seda arvestades on interpreteerimine primaarne, masinale omane. Assemblerprogrammi tekstist *kompileeritakse* interpreteeritav masinkood. Makrokäske interpreteerib kas preprotsessor või assemblertranslaator „ise", genereerides nende asemele makromäärangu baasil assemblerdirektiivide jadad.

Masinast sõltumatud keeled realiseeritakse kas interpretaatorite või kompilaatorite abil; esimesel juhul on programmi transleerimise resultaadiks mingi andmestruktuur (tavaliselt

[1] Täielikult kehtib see kompileeritavate keelte puhul. Interpreteeritavad keeled võivad kasutada muid vahendeid, mida kas ei saa või pole mõtet keskprotsessorile interpreteerida anda, näiteks, multimeedia-"protsessing" (vastavad kaardid (kiibid) on autonoomselt programmeeritavad ja nonde koode ei interpreteeri tavaline käsuprotsessor, vaid erivahendid), arvutivõrkudega seonduv jmt.

puu)[1], mille abil interpretaator „täidab" etteantud programmi; kompilaatori väljund on käsufail (meie jaoks tänapäeval *.exe*-fail). Kompileeritavad keeled on ─ interpreteeritavatega võrreldes ─ sootuks rohkem orienteeritud masina tasemele, so. nad püüavad maksimaalselt saavutada *aparatuurset* tuge, ja minimiseerida *programset* interpretatsiooni.

Interpreteeritavad keeled on objektmasina ja selle keel(t)ega tavaliselt võrdlemisi lõdvalt seotud. Hoopis rohkem huvitab neid kasutatav operatsioonisüsteem (dünaamiline mälujaotus, alamprogrammide kokkulepped, tänapäeval võrgu- ja multimeediavahendite liideste integreerimisvõimalused jmt.).

Translaator on talle orienteeritud programmi(teksti) „tõlkija" interpreteeritavasse või kompileeritavasse vormi. Kõige üldisemalt, translaator säilitab lähtekeele *semantika,* ent muudab reeglina nii leksikat kui ka süntaksit. Nii on semantiliselt võrdväärsed üldarusaadavad fraasid

- Ma armastan sind
- I love you
- Я тебя люблю,

ent muud komponendid on tuntavalt erinevad. Ja kui naasta programmeerimiskeelte juurde, siis on semantiliselt samaväärsed omistamisoperaatorid

- B:=A+B; (Algol)
- СФЗ A,B (ССК)
- B DUP @ A @ + ! (*FORTH*)
- L 1,A
A 1,B
ST 1,B (*IBM/360* assembler)

Varajaste arvutite *masinkoodid* olid sama unikaalsed nagu nood masinad isegi. Edasine areng on suuresti dikteeritud tarkvara akumulatsioonist: unikaalsele masinale kirjutati aja jooksul tähelepanu vääriv kogus programme, milledest mingi osa polnud ühe konkreetse ülesande lahendamiseks kirjutatud, vaid olid sootuks üldisemad, näiteks maatriksarvutuste või statistilise andmetöötluse „paketid". Aegunud masina väljavahetamine kaasaegsema vastu tähendas sageli kogu senini loodud tarkvara „üle parda heitmist" ja püsivamat huvi pakkunud programmide uuesti kirjutamist.[2]

[1] Programmi puu interpreteerimise asemel on kasutatud ka muid strateeegiaid, nii seletab *Schildt* [52, lk. 19] interpretaatorit kui programmi, mis "lihtsalt loeb pogrammi lähtekoodi, täidab jooksva programmirea instruktsioone ja läheb üle järgmisele reale". See seik tasuks meelde jätta, eriti, kui lugeda interpretaatorite (*versus* kompilaatorite) efektiivsusest, esmajoones tuleks selgitada, kas on mõeldud interpreteerimist teksti tasemel – see variant on konkurentsitult ebaefektiivseim – või midagi muud. (Vt. ka [20], lk. 17).

[2] Programmeerijate defitsiit oli väga aktuaalne juba 50 aastat tagasi. Ja kalambuurgi, et „arvuti on odav pakend kalli kauba (=tarkvara) ümber" on pea sama vana, ehkki noil aegadel oli see lihtsalt hea nali. Näiteks, enne totaalset inflatsiooni oli *IBM XT*-masina turuhind NL-is ca 100 000 rubla, parim (ostuloaga) saadav auto *„Žiguli-Luks"* maksis ca 10 000 rbl. (turul mõistagi paar korda rohkem), ja professori palk oli ca 500 rbl. kuus (ja see oli väga hea palk). Programmeerimisäss sai ca 150 (hinnanguliselt 1 rbl ≈ 35 EEK)

Kui uus mudel on taas unikaalne, siis tuli loobuda kõigist vanale mudelile orienteeritud programmidest, nii masinkoodis, assembleris kui ka makroassembleris kirjutatutest. Ent mida kauem oli tarkvara loodud, seda ilmsemaks sai väga olulise ressursi (programmeerijate töö) raiskamine.

Lahendusi otsides oli lihtsam jõuda *omavahel ühilduvate* arvutimudelite ideele, mis seisneb lihtsustatult selles, et sama firma järgmine (uuem ja võimsam) mudel konstrueeritakse nii, et ta on võimeline *emuleerima*[1] eelmis(t)e mudeli(te) koodi. Sellist lähenemist võib täheldada *IBM*i suurarvutite, *Siemens*i mudelite, *Minsk-22* ja *Minsk-32* puhul, aga ka *Intel*i protsessoriga mikroarvutite puhul ─ nii „jooksevad" ka 32-bitisel masinal 16-bitisest arhitektuurist pärit *.com*-failid. Võiksime nentida, et see oli *tarkvara mobiilsuse* inseneerlik lahendus.

Teist teed võiksime nimetada programmeerijalikuks lahenduseks ─ see seisnes sellise programmeerimiskeele disainimises, mis ei sõltuks konkreetsest arvutimudelist, vaid mis võimaldaks tolles keeles kirjutatud tarkvara kasutada ka uutel mudelitel. (Makro)assembler-keel on vahetult seotud konkreetse arvutiga ning selle sobitamine võimsama mudeliga on teoreetiliselt mõeldav, ent kaheldava efektiivsusega. Järelikult, tuli välja mõtelda keel, mis oleks piisavalt üldine, et selles kirjutatud programme saaks oluliste kadudeta transleerida uute mudelite (eeskätt sama firma omade) masinkoodi. Ja et selles programmeerimine oleks veel lihtsam kui makroassembleris ning programmid oleksid vähemalt sama loetavad kui makroassembleri omad. Koos sellega kasvaks ka programmeerijate töö viljakus ning tarkvara usaldusväärsus (kirjutatud tekst on loetavam ja testitavam).

Tuletagem meelde, et arvuteid kasutati 50 aastat tagasi just selleks nagu nende eestikeelne nimetus üheselt paika paneb ─ arvutamiseks, ja kõige tähtsam töö oli aritmeetiliste avaldiste lahendamine. Juba enne masinastsõltumatuid keeli loodi aritmeetiliste avaldiste lahenduskäigu programmeerimiseks (meie distantsilt vaadates üsnagi võimsaid) assemblereid, näiteks brittide *Manchester-Autocode*, 1956 (vt. [21, lk. 107]) võimaldas kodeerida omistamisoperaatorit

```
x=a+b*c+d*e
```

direktiivide jadana

```
v1=b*c
v2=d*e
v1=a+v1
x=v1+v2
```

Programmeerija ei pidanud tegelema tegelike aadressidega, summaatorite nullimisega jne., seega ei pidanud ta tundma reaalset masinkoodi. Veelgi enam, assembleris sai kasutada teegifunktsioone, näiteks

```
v2=sqrt(v1)          sqrt (square root)  on ruutjuure võtmise funktsioon
```

[1] Arvutileksikon [11, lk.127] seletab emulaator*it* (i.k. *emulator*) kui riist- või tarkvara, mis on loodud selleks, et üks seade (näiteks arvuti) käituks nii, nagu oleks ta midagi muud.

Sama allikas [21, lk. 93] toob meie teadvusse veel tõiga, et meie jaoks loomulik *vaikimisi*-variant sai alguse arenenud assemblerist *Mercury Autocode,* kus muutujate nimede esitähed määrasid vaikimisi muutuja tüübi, sj. just samamoodi nagu hiljem *FORTRAN*i kaudu üldtuntuks sai. „Vaikimisi"-printsiip ise paneb keele informaalses kirjelduses paika translaatori poolt eeldatud variandid, ja kui programmis pole öeldud teisiti, siis translaator käsitleb olukordi nii, nagu sisse programmeeritud.

Esimestena maailmas hakkasid globaalse resultatiivsusega sellise „üldistatud assembleri" loomisega tegelema *IBM*i programmeerijad *John Backus*e juhtimisel masina *IBM 704* jaoks 1954. aastal, novembris avaldati juba töörühma esimene teade projekti kohta, ja projekti nimeks sai „*The IBM Mathematical FORmula TRANslating system*", ehk tänaseks lihtsalt üldtuntud *FORTRAN*.

Ja nii või teisiti, ent *FORTRAN*ist sai maailma esimene masinast sõltumatu keel. Aeg on näidanud, et teda saab realiseerida mistahes senituntud protsessoriga universaalarvutil.

Teiselt poolt, *FORTRAN*i nimetatakse sageli *esimeseks protseduurorienteeritud keeleks*, ent see tundub olevat mõneti eksitav terminoloogia: *kõik* masinorienteeritud keeled on täielikult ka *protseduurorienteeritud* (e. *imperatiivsed*) keeled, kuivõrd nad võimaldavad esitada *ainult* täpset instruktsioonide hulka.

Pigem on kõrgtaseme keeled eristatavad masinorienteeritutest *masinast* (suhtelise) *sõltumatuse* poolest. Seega: keskendugem programmeerimiskeelte klassifikatsiooni esmasele jaotusele: *masinorienteeritud* ja *masinast sõltumatud* keeled.

Meie raamatu teine pool püüab käsitleda „kõrgtaseme" keeli alates *FORTRAN*ist. Ja loomulikult me ei üritagi neist kasvõi üht promilli käsitleda ─ neid keeli on lihtsalt nii (või liiga?) ─ palju.

3. PROTSEDUURORIENTEERITUD KEELED: FORTRAN JA ALGOL-60

3.1. *FORTRAN*

Meie õppevahendi üks eesmärk on näidata, et programmeerimiskeeled on arenenud (senini, so. *FORTRAN*ini) loogiliselt järjepidevalt, püüdes anda programmeerijatele võimalust alati sammu ees olnud riistvara[1] poolt pakutava võimalikult heaks ärakasutamiseks. Kõik senikäsitletu peaks näitama, et nii masinad kui ka keeled arenesid selles suunas, et anda parimaid võimalusi arvutusülesannete lahendamiseks. Selleks olid konstrueeritud „pesamasinad" koos nende käskude süsteemidega ning programmeerijate käsutusse anti adekvaatsed masinorienteeritud keeled.

Programmeerimiskeelte geneesi näitamiseks on meie arvates loomulik üritada tutvustada nonde keelte esimesi versioone. Hilisemad, konkurentsi jäänud versioonid erinevad esmasversioonidest üsnagi palju; lugejale soovitame võrrelda „alg-*FORTRAN*i" sellega, mis sellest keelest tänaseks on saanud [58].

Lugeja peaks olema juba raamatu alguses ja eelmise peatüki lõpus esitatud materjalide põhjal mingil moel tuttav oludega, mis viisid, vististi vältimatult, masinast sõltumatu keele ilmumiseni.

3.1.1. Assembler ja *FORTRAN*

FORTRAN **sarnaneb** (makro)assembleriga üsna mitme seiga poolest. Loetlegem neist mõningaid:

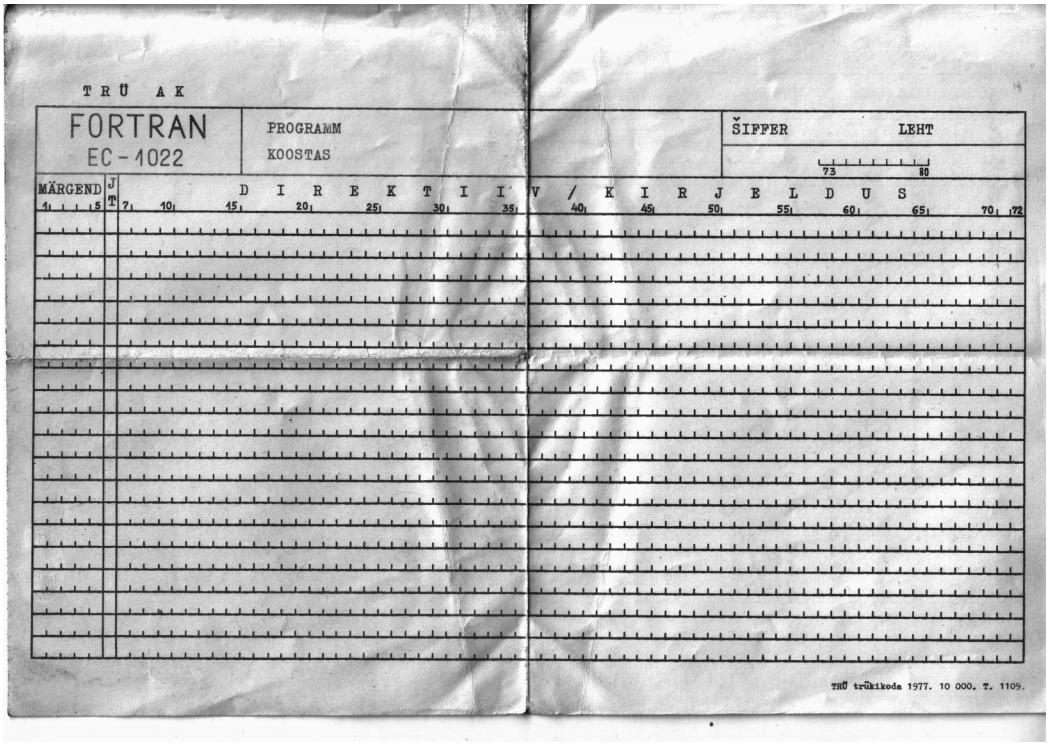
- Orienteeritus perfokaartidele.
- Poolfikseeritud formaat (vt. joonis 3.1.1a).
- Võtmesõnade (`DO, IF, GOTO` jne) paiknemine seal, kus makroassembleris on makrokäsk.
- Masinorienteeritud sisend-väljundformaat.
- Masinkoodist ja assemblerist tuttav eraldi transleeritud moodulite toetamine.
- Ühisväljad.

FORTRAN **ei sarnane** (tavalistele) assembleritele samuti mitme asjaolu poolest, näiteks:

- Võimalus esitada harjumuspärasel „matemaatilisel" kujul aritmeetilisi avaldisi.
- Võimalus deklareerida vektoreid ja massiive ilmutatud kujul.
- Esmakordselt lisandus *programselt interpreteeritav* andmetüüp ─ *FORTRAN*is konkreetselt *complex.*

[1] Insenerid naljatasid, et IT-valdkonnas kehtib "universaalne *Malthus*e seadus": riistvara areneb geomeetrilises, tarkvara aga aritmeetilises progressioonis. Pisut leebem on *Reiseri seadus*: "tarkvara muutub aeglasemaks kiiremini kui riistvara muutub kiiremaks" [63, lk.20].

- Võimalus tekitada käskude gruppe „normaalsel" kujul (tsükkel, IF-lause, grupiviisiline suunamine, loogilised tingimused jm.).
- *FORTRAN*-teksti võis transleerida suvalise (tolleaegse) pesamasina masinkoodi, selleks tuli ainult kirjutada antud masina jaoks translaator. See seik tagas esmakordselt keele *masinastsõltumatuse.* Ja kuivõrd esialgu puudus *FORTRAN*i standard, võis iga konkreetse masina jaoks realiseeritud *FORTRAN* olla üsna vabalt tõlgendatud.



Joonis 3.1.1a. *FORTRAN*i plankett.

3.1.2. Näiteprogramm

Näite laename raamatust [61, lk.22]: leida vektori elementide väärtuste aritmeetiline keskmine ja seejärel nende elementide arv, mille väärtus on keskmisest suurem.

```
C       FORTRAN PROGRAM TO FIND MEAN OF N NUMBERS AND NUMBER OF
C       VALUES GREATER THAN MEAN
        DIMENSION A(99)
        REAL MEAN
        READ(1,5),N
    5   FORMAT(I2)
        READ(1,10)(A(I),I=1,N)
   10   FORMAT(6F10.5)
        SUM=0.0
        DO 15 I=1,N
```

```
15  SUM=SUM+A(I)
    MEAN=SUM/FLOAT(N)
    NUMBER=0
    DO 20 I=1,N
    IF(A(I).LE.MEAN)GOTO 20
    NUMBER=NUMBER+1
20  CONTINUE
    WRITE(2,25)MEAN,NUMBER
25  FORMAT(8H MEAN = ,F10.5,5X,20HNUMBER OVER MEAN=,15)
    STOP
    END
```

3.1.3. Lühiülevaade

Alates sellest alajaotusest on suuresti tuginetud *Terrence W. Pratt*i raamatule [44].

FORTRAN loodi ajal, mil valitsesid defitsiidid: mäludefitsiit („füüsiline bitt" ─ raadiolamp ─ oli kallis) ja masinaaja defitsiit (ühte masinat jagasid paljud üsna suured töörühmad ja masin tarbis palju elektrit ning teenindava personali tööjõudu). Seega oli oluline, et translaator, komplekteerija ja paigaldaja töötaksid kiiresti ning et *FORTRAN*-programmide lahendamiskiirus ei jääks liiga palju alla assemblerprogrammidele.

Sellest johtuvalt disainiti keel nii, et kompilaator teeks võimalikult palju tööd ja teeks koodi, mida saaks võimalikult suures ulatuses aparatuurselt interpreteerida ning et programmi lahendamise ajal oleks minimaalselt vaja programset interpreteerimist. Esmajoones välistas see dünaamilise mälujaotuse ja rekursiooni.

FORTRAN-programm koosneb põhiprogrammist ning juurde komplekteeritud alamprogrammidest. Infovahetus alamprogrammidega toimub (kas) parameetrite ja/või ühisväljade abil. Andmetüüpideks on neli arvulist ja üks loogiline (*boolean*) tüüp, kasutada saab lihtmuutujaid ja kuni 3-mõõtmelisi staatilisi massiive. Andmed olid *kas* globaalsed *või* lokaalsed. Keel võimaldab tüübiteisendusi ja evib enam-vähem täielikku aritmeetiliste ja võrdlemisoperatsioonide kogumit.

Sisendi-väljundi programmeerimine oli kohmakas ja keeruline, ent tolleaegsetel masinatel puudusidki noiks töödeks mugavad võimalused.

3.1.4. Andmed

*FORTRAN*is on viis lihttüüpi ning ainsaks andmestruktuuriks[1] kuni 3-mõõtmeline massiiv, mille elemendid on kõik ühte ja sama lihttüüpi.

Lihttüübid on:

- *Boolean* väärtustega `.TRUE.` ja `.FALSE.`
- Täisarv, näiteks 498.

[1] See ei tähenda, et *FORTRAN* ei saaks hakkama viitasid kasutavate struktuuriga (ahelad, puud, magasinid jmt.), vt. osa 3.1.9.

- Reaalarv, mis peab sisaldame kas kümnendpunkti või astme määrajat *E*, näiteks `7.98, 33.1E12 või 66E7`.
- Kahe-pesa-reaalarv, mis esitatakse nagu *real*, ent *E* asemele tuleb kirjutada *D*.
- Kompleksarv, mis esitatakse sulupaari vahel kahe reaalarvuna, näiteks `(0.11, 2E3)`; tehteid nonde arvudega interpreteerib *FORTRAN* programselt.

Andmed tuleb enne kasutamist *kirjeldada. FORTRAN* järgis ühe ülalpool näiteks toodud arenenud assembleri pretsedenti (mida hilisemad tuntud keeled siiski ei järgi): andmeühiku aparatuurselt toetatava tüübi määrab vaikimisi identifikaatori esitäht: kui see on vahemikust *I..N*, siis tüüp on täisarv (*Integer* ja *Natural*)[1], muidu aga *real.* Kui semantilistel kaalutlustel (nagu näiteprogrammis, muutuja *MEAN*) soovitakse tüüpi muuta, tuleb seda teha ilmutatud kujul.

Sisendi-väljundi formaatide esitamisel võib kasutada ka *stringe* ─ nn. *Hollerith*i konstante[2] kujul *nH*string, kus *n* on stringi pikkus, näit. „`9HHOLLERITH`" (*n=9,* esimene „H" on marker ja string on „HOLLERITH").

Massiivi kirjeldus algab võtmesõnaga `DIMENSION`. Näiteks, `DIMENSION A(10,20), K(25)` defineerib kahemõõtmelise *real*-massiivi *A* (*ALGOL*i mõttes A[1:10,1:20]) ja *int*-vektori *K*.

Aritmeetilised tehted järjestab *FORTRAN* järgmiselt: kõrgeim prioriteet on *astendamisel* (**), järgnevad *korrutamine* ja *jagamine* (* , /) ja *liitmine* ja *lahutamine* (+ , -).

Võrdlustehete operandid peavad olema sama tüüpi (erandina saab omavahel võrrelda lühikesi ja pikki reaalarve); arvulisi operande saab võrrelda järgmiste operaatoritega:

- `.EQ.` (*equal to*), võrdub
- `.NE.` (*not equal to*), ei võrdu
- `.LT.` (*less than*), väiksem
- `.GT.` (*greater than*), suurem
- `.LE.` (*less than or equal to*) väiksem või võrdne
- `.GE.` (*greater than or equal to*) suurem või võrdne (vt. [45], lk. 40)..

Boolean-tüüpi muutujate jaoks on operatsioonid

- `.OR.` loogiline liitmine (∨).
- `.AND.` loogiline korrutamine (&).
- `.NOT.` loogiline eitus, unaarne operatsioon (¬).

[1] *FORTRAN* lubas kuni 3-mõõtmelisi massive, rajad ja indeksid on loomulikult täisarvud; harjumuspärane on indekseid tähistada *I, J* ja *K* ning rajasid *N, M* ja *L* (*I* muutub 1..*N* jne).

[2] *Herman Hollerith* patenteeris 1884. aastal, 24-aastasena perfokaartide sorteerimise masina; kaardi oli ta juba varem leiutanud. 1896. aastal asutas ta oma masinate (mida oli kasutatud edukalt 1890. a. USA rahvaloenduse kokkuvõtete tegemisel) tootmiseks firma, millest sai pisut hiljem *International Business Machines Corp.* ─ *IBM* [13, lk. 9 ─ 10].

3.1.5. Adresseerimine ja omistamine

FORTRAN oli kõrgtaseme programmeerimiskeelte süntaksi teerajaja ja moelooja. Ja adresseerimisel järgis ta stiili, mille algus oli juba masinkoodides: kas on mõeldud aadressil olevat väärtust või aadressi, kuhu salvestada mingi väärtus, oli määratud üksnes ja üheselt käsu formaadiga. Selles veendumiseks võiks vaadata kasvõi õppearvutite masinkoode. Ent kõrgtaseme (so, masinast sõltumatute, sh *FORTRAN*i) keelte puhul tekitas see stiil semantilisi probleeme (noist on vaba *FORTH*, mida käsitleme hiljem). Nii tähistab *FORTRAN*is „B(7,I)" nii tabeli 7. rea *i*-nda elemendi aadressi kui ka tolle elemendi väärtust ─ sõltuvalt kontekstist. Kui see aadressi väljendav avaldis on omistamisoperaatori vasakul poolel, on mõeldud aadressi, kui aga operandina selle operaatori paremal poolel, siis väärtust.

FORTRAN-kompilaator teostab tüübikontrolli: nt. *real*-tüüpi muutujale ei saa omistada *int*-tüüpi väärtust; kui seda programm „püüab teha", siis genereerib kompilaator madalama prioriteediga tüübi teisendamise vajalikule tasemele (kasutades töömuutujaid ja rikkumata programmis kirjeldatud andmeid). Ilmselt on teisendamise prioriteet *double* ← *real* ← *integer*. Sama taktikat järgitakse, kui aritmeetilise avaldise operandid pole sama tüüpi.

Lihtmuutujate ja massiivi elementide väärtuste muutmiseks on *omistamine* põhiline vahend. Eristatakse 3 liiki omistamisi:

- „aritmeetiline": näiteks `K(4,3)=X**X`
- „loogiline", näiteks `B=X.GE.3.AND.Y.EQ.0`
- „märgendile omistamine" *ASSIGN*, näiteks `ASSIGN 3 TO I`

Lisaks saab muutujaid *algväärtustada* kompileerimise ajal, näiteks lokaalsetele muutujatele X, Y ja K saab omistada algväärtusi järgmiselt:

```
DATA X/1.1/Y/0.86/K/99
```

Globaalsed muutujad paiknevad *ühisväljas* ja programmis näeb nende algväärtustamine välja näiteks nii:

```
BLOCK DATA COMMON/YHIS/PII/3.14/KILO/1000
```

Need väärtused paneb paika *paigaldaja*. (`PII=3.14, KILO=1000`).

Lihtmuutujate ja massiivi elementide väärtuste kehtestamiseks saab kasutada ka *sisend-väljundoperaatoreid*. *FORTRAN* toetab ainult jadafaile (nagu tolleaegsed masinadki); vahendid olid lugemine (READ), kirjutamine (WRITE) ja abivahendina jooksva kirje viida nihutamine (näit. REWIND *n* ─ välisseadmel nr. *n* asetatakse viit faili algusse, BACKSPACE *n* ─ eelmisele kirjele) Viita nihutasid automaatselt ühe võrra nii lugemine kui ka kirjutamine. Viimaste esitusviis oli

READ/WRITE(n,viit formaadile)

Parameetri *n* väärtusvaru oli fikseeritud masina riistvara poolt (näiteks nii, et 1 on perfokaartsisend ja 2 on trükiseade) ning „viit formaadile" ─ *FORTRAN*i kombe kohaselt täisarv ─ osutas reale, kus paiknes sisend-väljundoperatsiooni operandi mastaabi- ja tüü-

bikirjeldus. Formaate ei transleeritud, vaid säilitati kompileeritud programmis tekstikujul ning neid *interpreteeriti* programselt spetsiaalsete süsteemsete alamprogrammide abil täitmise ajal. Formaadiga sai mh. näidata, milliseid (vahe)pealkirju tuleb trükkida ─ kasutades *Hollerithi* konstante.

3.1.6. Alamprogrammid

Selles jaotises on tuginetud lisaks *T. Pratt*ile [44, lk. 355 jj] ka *Ü. Kaasik*u ja *H. Niiliski* koostatud käsiraamatule [45, lk. 92 jj.].

Esiteks, *FORTRAN* võimaldas kirjeldada (programmi alguses, pärast teisi kirjeldusi, ent enne esimest direktiivi) *lokaalseid funktsioone* ─ omamoodi *makrosid*. Näiteks, kui lokaalse funktsiooni kirjeldus on `FN(X,Y)=SIN(X)**2-COS(Y)**2`, võis seda kasutada programmis näiteks nii: `Y=FN(ALFA,BEETA)` ning translaator asendas selle enne masinkoodi genereerimist tekstiga `Y=SIN(ALFA)**2-COS(BEETA)**2`.

Teiseks, *FORTRAN* võimaldas kasutada eraldi transleeritud ning juurde komplekteeritud *funktsioone* (tagastavad ühe aritmeetilise või loogilise väärtuse; neid saab kasutada avaldistes operandidena) ja *protseduure* (pöördumine *CALL*-operaatoriga). Näiteks:

```
A=B+SQRT(C+D)
```

Välisfunktsioone pole vaja enne kasutamist kirjeldada, küll aga protseduure, programmis kirjelduste osas, näiteks `SUBROUTINE AP(A,B,C)` ning kasutada saab seda näiteks nii:

```
CALL AP(14.0,Y,W).
```

Kui näiteks *AP* peab tegema omistamise `C=A*B`, siis antud pöördumisega omistatakse `W=14.0*Y`. Ent programmi tekstis (kui just ei kommenteeri) puudub informatsioon selle kohta, et *AP*-sse pöördumine muudab muutuja *W* väärtust. Sellist seika nimetatakse *kõrvalefektiks* (*side effect, побочный эффект*): mingi funktsiooni või protseduuri täitmine muudab mälu seisu.

Alamprogrammide parameetrid kantakse üle ainult *viitadega*, seega on igal välisfunktsioonil või protseduuril potentsiaalne võimalus neid viitu kasutades tekitada (täiendavaid) kõrvalefekte. Peale parameetrite saavad alamprogrammid suhelda põhiprogrammiga ja omavahel samanimelis(t)e ühisvälja(de) abil, need moodustavad (viitade) globaalse keskkonna. Seejuures iga ühisvälja kasutaja võib ühiskasutatavaid objekte nimetada ja interpreteerida individuaalselt; seotud on kasutajad ainult nonde objektide suhtaadressidega. Mis on ka arusaadav, kuivõrd alamprogrammid transleeritakse autonoomselt ning komplekteerijal ja paigaldajal puudub informatsioon eraldi transleeritud alamprogrammide muutujate nimedest ja kirjeldustest (küll aga on see olemas kasutajatel ─ alamprogrammide kirjalikest spetsifikatsioonidest). *T. Pratt* toob järgmise näite ([44], lk. 361 ─ 362): alamprogrammid *SUB1* ja *SUB2* kasutavad ühisvälja nimega *BLK*, pikkusega 27 pesa. Seejuures võib tolle ühisvälja kirjeldus olla alamprogrammis *SUB1* näiteks

```
COMMON/BLK/X,Y,K(25)
```

ja alamprogrammis `SUB2`

```
COMMON/BLK/U,V,I(5),M(4,5)
```

Mälus näeb see välja nii:

SUB1	X	Y	K_1	K_2	K_3	…	K_6	…	K_{25}
SUB2	U	V	I_1	I_2…	I_3	…	$M_{1,1}$	…	$M_{4,5}$
	1	2	3	4	5		8		27

Pöörakem tähelepanu seigale, et mõlemad alamprogrammid aktsepteerivad väljade tüüpe (*X,Y* ja *U,V* on *real,* ülejäänud *int*)[1].

Alamprogrammist väljumiseks kasutatakse direktiive *RETURN* (funktsioonist) ja *STOP* (protseduurist).

Kordame veel üle, et staatilise mälumudeli[2] tõttu ei saa *FORTRAN*is kasutada rekursiivseid alamprogramme[3]. Ja sedagi, et eraldi transleeritud alamprogrammide tõttu pole kompilaatori kontrolli all kõrvalefektide võimalus/oht. Siluja ei pääse juurde alamprogrammide nimede tabelile jmt. infole, see raskendab sekundaarsete vigade avastamist, ent see probleem on „kasutaja enese asi".

3.1.7. Tegevuste järjekord

Avaldistes (aritmeetilised ja loogilised) on tehete täitmise järjekord paika pandud *prioriteetidega*, esitame nad kahanevas järjekorras:

1. ** *astendamine* (kõrgeim prioriteet, tehe sooritatakse võimaluse korral esimesena)
2. * ja / *korrutamine ja jagamine*
3. + ja − *liitmine ja lahutamine*
4. `.EQ. .NE. .LT. .GT. .LE. .GE.` *võrdlemised* $=,\neq,<,>,\leq,\geq$
5. `.NOT.` *eitus* $\neg$
6. `.AND.` *loogiline korrutamine* & või $\wedge$
7. `.OR.` *loogiline liitmine* $\vee$ (madalaim prioriteet, tehe sooritatakse viimasena)

Tehete sooritamise järjekorda (prioriteete) saab muuta sulgude „(" ja „)" abil.

Operaatorite (*statements, инструкций*) täitmise järjekord on vaikimisi loomulik, so., selline nagu ka juhtimisseade eeldab. Programsed võimalused toda järjekorda muuta on järgmised, **esiteks** suunamised:

---

[1] Korrakem veel kord: masinorienteeritud keelte toetatud ühisväljad on täpselt samuti autonoomsed; ühisvälja kasutaja teab ainult suhtaadressi ja ei enam.

[2] Tuletame lugejale meelde, et staatiline mälujaotus ei võimalda lahendamise ajal mälu ümber jaotada, so, kõik pannakse paika translaatori, komplekteerija ja paigaldaja poolt.

[3] Põhimõtteliselt oleks see võimalik, kui *FORTRAN* toetaks pseudoaparatuurset *magasini* (so, tal oleksid analoogid käskudele *PUSH* ja *POP*, mida täidaks *FORTRAN*i virtuaalmasin staatilises magasinis (mille mahtu võiks transleerimistellimuses näidata)).

- Suunamine märgendile (viimane on 1..5 kümnendnumbrit ja paikneb perfokaardi vasakul serval), sh.
- *Tingimusteta* suunamine, nt. `GOTO 15`
- *Valikuline* suunamine, üldkujul `GOTO K,(n1,n2,..np)`, kus *K* on muutuja, millele tuleb eelnevalt omistada üks täisarvuline väärtus hulgast $\{n_1,n_2,..n_p\}$ käsuga *ASSIGN* — näiteks `ASSIGN 11 TO K` ning kompilaator teeb tingimusteta suunamise `GOTO 11`
- *Hargnemine* kujul $GOTO(n_1, n_2,.. n_p), I$, kus *I* on *indeks* vahemikust 1..p. Juhtimine antakse *I*-ndale märgendile; *I* väärtustatakse lahendamise ajal. Kui $I>p$, siis mingit suunamist ei toimu ja täidetakse järgmine direktiiv.

Teine tehete sooritamise järjekorra muutmise võimalus on *tingimuslik operaator* üldkujuga *IF* …, siin on kaks võimalust: *aritmeetiline* ja *loogiline IF*. *Aritmeetilise* üldkuju on

IF(aritmeetiline avaldis) n_{miinus}, n_{null}, n_{pluss}

n-d on märgendid, kuhu antakse juhtimine olenevalt aritmeetilise avaldise väärtusest.

Lugejale, kes on jälginud kogu materjali: kas ei tuleta see tingimusoperaator meelde kolmeaadressilise masina koodi?

Loogiline IF on kujul

IF (loogiline avaldis) operaator,

kusjuures too „operaator" ei tohi olla ei teine loogiline *IF* ega ka *DO*-tsükkel — mis tähendab, et välistatud on operaatorite *puud.*

Kolmas tegevuste järjekorra muutmise vahend on *tsükkel.* Umbes nii nagu meie õppearvutite puhul algab ka *FORTRAN*is tsükkel algusaadressiga (mida peab meeles tsüklikäsk) ja tsükli keha algab *DO*-le järgmisest direktiivist. Tsükli jaoks on kaks võimalust:

DO märgend tsükliloendaja=algväärtus,lõppväärtus

Märgendiga tuleb varustada tsükli viimane direktiiv. *Ü. Kaasik* ja *H. Niilisk* toovad seda tüüpi tsükli kohta järgmise näite [45, lk. 53];

```
      DO 30 K=1,20
      IF(A(K)-B(K)) 5,40,40
5     A(K)=C(K)
      GOTO 30
40    A(K)=0.0
30    CONTINUE          tühidirektiiv (viimaseks ei tohi olla tingimus ega
                        suunamine!)
```

Teine variant on

DO märgend tsükliloendaja=algväärtus,lõppväärtus,samm

Seda versiooni kasutatakse siis, kui tsükliloendaja samm≠1.

Lubatud on tsüklid tsüklis, ent keelatud on tsükliparameetrite muutmine tsükli kehas. Kui arvuti arhitektuur seda võimaldab, siis püüab kompilaator panna tsükliparameetrid *kiiretesse registritesse.*

3.1.8. Süntaks ja transleerimine

Nagu me juba teame, on *FORTRAN*i süntaks orienteeritud perfokaartidele, so, eeldab poolfikseeritud formaati ning see seik tegi keele esimeste versioonide süntaksi formaliseerimise raskeks, kui mitte võimatuks. Esimene teadaolev *FORTRAN*i formaalne süntaks esitati *FORTRAN*-77 jaoks, seda *Wirthi skeemidega* (nii neid skeeme kui ka muid süntaksi formaalse esitamise variante tutvustame hiljem).

Kordame siin üle mõningaid iseloomulikke jooni: kõik operaatorid algavad *võtmesõnaga* (*DO, IF* jne), kasutatavad on konkreetse masina trükisümbolite standardkogum(id) ning iga programmirida on üldjuhul nii lugejale kui ka translaatorile lihtsalt ja üheselt mõistetav; *inimese* raskused tekkivad pikemate tekstide loogika jälgimisel (võrreldes hilisemate keeltega on väga palju *suunamisi GO TO*) ja seetõttu ka märgendatud operaatoreid ning nende vastavusseviimine võib osutuda tülikaks. Lisaks ei tule programmiteksti loetavusele kasuks identifikaatori piiratud pikkus (1..6 sümbolit). Ehk seegi, et tühikut *võib* kasutada, ent *ei pea* kasutama: näiteks pole vahet, kas kirjutada *GO TO* või *GOTO.*

Kompilaatorile tekitavad raskusi tsüklidirektiivid (*DO*): tuleb otsida tsükli lõpu kaarti ning kontrollida kitsenduste täidetust tsükli kehas. Muidu on programm transleeritav kaarthaaval[1].

Kompilaator teeb peaaegu kõik vajaliku; milleks ta pole suuteline, koondatakse paigaldaja interpreteeritavasse tabelisse. Tabelis on vajalik info *viitade* väärtustamiseks: *ühisväljadele, süsteemsetele programmidele* ja juurdekomplekteeritud alamprogrammidele.

Kompilaator suudab (tänu staatilisele mälujaotusele ja suhtelisele jäikusele) avastada ─ kui uskuda müügimehi ─ peaaegu kõik vead, ent ei saa vastutada paigaldamisel lisatavate programmide eest (liiati võivad nood olla kirjutatud mistahes keeles ─ paigaldaja lisab

[1] *John J Donovan* [12, lk 18] kirjeldab *FORTRAN*i algusaegade (kui puudusid tegusad operatsioonisüsteemid) programmeerija töötsüklit umbes nii: ta tuli masina juurde, vasakus käes „tavalist" värvi perfokaartide pakk programmi tekstiga, paremas käes oli roheliste kaartide pakk: *FORTRAN*-translaator **ise.** Edasi tegi ta mõningaid asju:

1. pani rohelised kaardid sisendseadmele ja vajutas masina puldil nuppu: masin luges sisse ja käivitas kompilaatori.
2. pani oma *FORTRAN*-programmi teksti kaardid perfokaartsisendisse; kompilaator luges need sisse ja perforeeris väljundperforaatoril (mille „taskus" olid punased kaardid) programmi „vahekeelse" (masinkood pluss info komplekteerijale ja paigaldajale) variandi
3. tõi hoolikalt valvatud toast paki roosasid perfokaarte ja sisestas need ─ laadis mällu ja käivitas *paigaldaja*.
4. pani punase paki uuesti seisendisse: paigaldaja loeb nad sisse.
5. perfokaartide sisestamise seadmele antakse lugemiseks kõikide kasutatavate alamprogrammide juba varem tehtud „vahekeelsed" (punased) kaardid. Paigaldaja liitis nad põhiprogrammiga.
6. Kui paigaldaja on omadega valmis, siis antakse juhtimine kasutaja-programmile, mis omakorda alustab reeglina tööd perfokaartide või nende pakkide lugemisega ─ noil on töödeldavad andmed.

peale nende standardse (opsüsteemi dikteeritud) vahekeelse koodi redigeerimist selle koodi täidetava programmi „kehasse")

Algaegade *FORTRAN* pööras suurimat tähelepanu *koodi efektiivsusele*: transleeritud *FORTRAN*-programm ei võinud võtta oluliselt rohkem mälu ega töötada oluliselt aeglasemalt kui masinkoodi-ässade programmid. Seetõttu oli nonde aegade kompilaatoritel reeglina kaks režiimi: silumiseks ja lahendamiseks; esimese efektiivsus küündis tavaliselt ca 10%-ni lahendamisversiooni omast, ent aeglane kompilaator andis tolle aja kohta maksimaalsed silumisvahendid ja -võimalused. Ja silutud programmide jaoks olid *optimeerivad* kompileerimisrežiimid.

Laename *T. Pratt*ilt pildi mingi *FORTRAN*-programmi täitmisaegsest mälujaotusest ([44], lk. 366]:

Süsteemsed andmed ja sisend- väljundpuhvrid
Põhiprogramm: lokaalsed muutujad ja põhiprogrammi käsud
COMMON BLK1 ühisväljad
COMMON DAT2 see samuti
SUB1 transleeritud alamprogramm ja tema lokaalsed andmed
SUB2 see on ka alamprogramm koos kõige vajalikuga
COMMON BLK2 ühisväli
SUB3 seegi on alamprogramm
.. . .
Sisend- väljundprogrammid
nimeta COMMON: siia resideerub *paigaldaja ise,* et mitte ennast üle kirjutada.

Joonis 3.1.8a. Täitmisaegne mälujaotus.

FORTRAN disainiti assemblerprogrammeerijate (s.o., professionaalide) töö lihtsustamiseks, ja kuivõrd ta oli esimene selle taseme keel, polnud tugineda varasematele kogemustele ning esmas-*FORTRAN*i süntaksis sisalduvad mitmed seigad, mis tekitasid lisaraskusi translaatorite kirjutajatele, aga ka tekstide lugejatele. Toogem mõned näited.

- Vaikimisi määras tüübi identifikaatori algustäht, ent tüüpi sai deklareerida ka ilmutatud kujul. Kas muutuja on üledefineeritud, selle tuvastamiseks pidi uurima programmi teksti algusest alates.

- Märgendid on kümnendtäisarvud ja nad ei pea olema järjestatud, seega pikema programmi tekstist tuli tolle tekstiga tutvujal otsida vajalikku märgendit tavaliselt alati teksti algusest alates (just nii, nagu käib otsimine korrastamata tabelist).
- Tolleaegsete masinate jaoks kasutatavate sümbolite hulk oli piiratud, nii olid ainsad võimalikud sulud paar „(,)". Siit tulenesid mitmed kahetimõistetavused.
- Ei tehtud vahet *võtmesõnade* ja *reservsõnade* vahel: esimesed identifitseerivad (keele autoritele heausklikult) eeskätt operaatoreid, teised aga moodustavad võtmesõnade alamhulga, mille elemente ei tohi kasutada tavaliste identifikaatoritena (muutujate ja alamprogrammide nimed). *FORTRAN* ei tunnistanud reservsõnu tegelikult üldse: nii olid legaalsed operaatorid

```
      7     IF(A*B-SIN(X))8,9,10
```

ja

```
      IF(J+1)=J
```

Kommenteerigem viimast näidet; esimene avaldis on tingimuslik suunamine, teine aga vektori *IF* elemendile indeksiga *J+I* uue väärtuse omistamine. See on, võtmesõnu (nagu *IF*[1]) võis kasutada muutujate nimedena, ja võtmesõna tuvastamiseks pidi kompilaator tegema täiendavat analüüsi. Näiteks niigi, et kas *SIN* on *real*-tüüpi vektor või teegifunktsioon *sin()*.

Arendagem viimaste näidete segadust pisut edasi:

```
17 IF((IF-J)/IF)22,33,44
```

See on legaalne, ent kas ka loetav?

3.1.9. *FORTRAN* ja viidasüsteemid

Alg-*FORTRAN* ei võimalda dünaamilist mälujaotust ega toeta viidatüüpi. Ent sellest hoolimata sai programmeerija ise tekitada (piiratud mahuga) viitadega seotud andmestruktuure; absoluutaadresside asemel tuli kasutada mõistagi suhtaadresse. Allpool toome näite lihtahelast ja *LIFO*-tüüpi magasinist.

Lihtahela (formaadiga *info, next*) kuni 50 lülile saab paigutada massiivi *M(50,2)* nii: *A(I,1)* on infoväli ning *A(I,2)* viit *J* järgmisele lülile ($0<J\leq 50$).

Magasinina võib kasutada näiteks massiivi *K(50)* ja magasin võiks alata näiteks aadressilt *K(50)*. Magasini tipu viida *I* algväärtuseks oleks seega 50 ning magasini kirjutamiseks tuleks täita käsud `K(I)=X I=I+1` ning lugemiseks `I=I−1 X=K(I).`

[1] Võtmesõnade tuvastamisega ei teki kompilaatoril üldjuhul probleeme, kuivõrd tänu poolfikseeritud formaadile paiknevad need assembleri tavasid järgides „koodiväljal" (ja nende võimalikud homonüümid „aadressväljal"). Assembleris on tavaline võte kasutada käsu etiketina ta mnemokoodi, kui see aitab paremini mõista programmi loogikat.

3.1.10. Kokkuvõtteks

Niisiis, *FORTRAN* konstrueeriti lihtsustamaks (makro)assembleris-programmeerijate tööd ning tegemaks tarkvara *mobiilseks* (kirjutades kompilaatoreid erinevatele protsessoritele). Oma perfokaart-orienteerituse ja assembler-stiilis direktiivide formaadiga oli ta programmeerijatele lihtsalt arusaadav ja õpitav. Tagantjärele tarkusena oli *FORTRAN*i püsimajäämise ja jätkuva konkurentsivõimalisuse võti assembleritest tuntud paindlikus alamprogrammide (eriti eraldi transleeritud juurdekomplekteeritavate „*CALL*"-alamprogrammide) säilitamises. Just see on võimaldanud poole sajandi jooksul akumuleerida arvutusmatemaatika mooduleid (vt. näit. [47]), ja on üpris ohutu ennustada, et *FORTRAN* ei kao iseenesest kuhugi.

FORTRAN on alati olnud orienteeritud efektiivsele[1] masinkoodile. Ja nii peab ka väljakutsutavate alamprogrammide kasutaja heauskselt arvama, et nood programmid on just sellise kompileerimisstiili produktid.

Edasi, põhiprogrammi kompilaator saab optimeerida ainult oma koodi, mitte *CALL*-operaatori(te)ga väljakutsutavaid, ja nimedega toetavat silumist saab pakkuda ainult põhiprogrammi osas. Ja sellest johtuvalt pole (suure) *FORTRAN*-teksti publikatsioon piisavalt informatiivne: me teame, mida teevad standardfunktsioonid (ruutjuur, logaritmid, trigonomeetria jne), aga ei tea, mida teeb protseduur X, mille poole pöördub publitseeritud programm käsuga *CALL X(Z,W)* ehkki me teame, mis objektid on *Z* ja *W*.

Ja siit on lihtne teha järeldust, et *FORTRAN* oli oma aja parim programmeerimiskeel, ent kaheldava väärtusega algoritmide publitseerimise keel.

3.2. *ALGOL*

3.2.1. Sissejuhatus

1950.-ndate aastate esimene pool oli aeg, kus töötati läbi ja konstrueeriti enam-vähem kõik see, mis on meie loengukursuse „Algoritmid ja andmestruktuurid" sisuks: keerukus, järjestamine, paisksalvestus[2], ahelad, magasinid, vektorid, massiivid, aga ka viidastruktuurid. Juba 1947. aastal oli asutatud tänini autoriteetne ajakiri *Communications of ACM* (*ACM=Association for Computing Machines*), mille üheks peaülesandeks oli algoritmide publitseerimine. Ent selleotstarbeline „normaalne" keel puudus (plokkskeemid jm. graafilised vahendid on küll head asjad programmi lähteülesande ja põhimõttelise lahenduskäigu kavandamiseks, aga mitte algoritmi täpseks esitamiseks). Üheselt mõistetavate algoritmide publitseerimiseks sobiva keele (*algoritmikeele,* ja mitte *programmeerimiskeele*) disainimiseks kutsuti kokku rahvusvaheline meeskond umbes samal ajal, kui *FORTRAN* oli ennast programmeerimiskeelena juba õigustanud, ja üheks initsiaatoriks oli *John Backus*, *FORTRAN*i „isa". Algoritmide publitseerimiseks mõeldud keele

[1] Mõtleme siingi selle termini all võimalikult vähe mälumahtu ja protsessoriaega nõudvat lahendust.
[2] Detsembris 1956 esimesed teated (Arnold I. Dumey), 1957 (W. W. Peterson, A. P. Jeršov), 1963 (ülevaade paiskfunktsioonidest, W. Buchholz), 1968 (R. Morris, ülevaateartikkel).

esimeseks nimeks sai *IAL* (*International Algebraic Language*), ent 58. aasta maikuul Zürichis leiti, et see nimi pole piisavalt atraktiivne ja publitseerimiskeele nimeks sai *ALGOL ─ ALGOrithmic Language*. „Teate" (*Revised Report on the algorithmic language ALGOL 60, Communication of the ACM, v. 6 n.1, p.1-17, Jan* 1963) allkirjastanud olid *J. W. Backus, F. L. Bauer, J. Green, C Katz, J. McCarthy, A. J. Perlis, H. Rutishauer, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger* ja *P. Nauer.* Niisiis, keel kavandati võimalikult heaks publitseerimiskeeleks, ja üldse mitte programmeerimiskeeleks. Sisend ja väljund (kui tol ajal enamvähem täielikult masinorienteeritud valdkond) jäeti keelest hoopis välja. Loomulikult loobuti *CALL*-operaatorist (väljakutsutava alamprogrammi kirjeldus ja algoritm peavad ka olema avalikud, seega alamprogrammid pidid olema kirjeldatud ja nende algoritmid esitatud koos põhiprogrammiga), ja ei saadud loobuda meile nii harjumuspärasest stiilist, et indekseid tähistame sümbolitega *i* ja *j,* muutujaid *a, b* ja *c,* tundmatuid *x* ja *y* jne. Ehk ka sellepärast on *ALGOL* plokkstruktuuriga keel ─ igas plokis võib deklareerida ja kasutada kui lokaalseid muutujaid just neid harjumuspäraseid „nimetusi" koos nende tavasemantikaga.

Meil pole mingit põhjust eristada *ALGOL-58* (vt. näit. [67]) ja *ALGOL-60*-keeli. Viimane oli lihtsalt esmasversiooni edasiarendus, ja kõik see, mis ülalpool *IAL*i kohta öeldud, kehtib ka üldtuntud *ALGOL*i kohta. Rõhutagem: *ALGOL* polnud mõeldud programmeerimiseks ja omas ajas oli seal üsna palju sellist, mida polnud võimalik efektiivselt realiseerida: dünaamiline mälujaotus (sh. dünaamilised massiivid), rekursioon, plokkstruktuur jm. Ja kui *ALGOL* realiseeriti programmeerimiskeelena, siis kaasnes see paratamatu ebaefektiivsusega ─ „virtuaalmasin" pidi (võrreldes *FORTRAN*iga) sootuks rohkem asju interpreteerima programselt ja samavõrra vähem aparatuurselt.

Ja veel. Et publitseeritud algoritmid oleksid *üheselt* mõistetavad, siis esitasid *John Backus* ja *Peter Naur ALGOL*i süntaksi formaalse definitsiooni metakeeles, mida tänapäeval tunneme kui *BNF*-notatsiooni[1]. Me käsitleme seda hiljem. Etteruttavalt: *BNF* inspireeris *süntaksorienteeritud translaatorite* konstrueerimist.

Enamvähem kõikide hilisemate programmeerimiskeelte süntaksi disain on tahes-tahtmata võtnud eeskuju *ALGOL*ist, ent keel ise jäi *programmeerimiskeelena* pigem marginaalseks, nii jäi ta täiesti tähelepanuta USA tarkvaratööstuses ja arvutuskeskustes (ehkki oli populaarne teema ülikoolides); mõnevõrra menukam oli ta Euroopas, ja eriti N. Liidus jm „sotsialismimaades", kus *Minsk*-tüüpi masinatel olid konkurentsivõimelised nii *Malgol* kui ka *Algams.*

3.2.2. Näiteprogramm

Näide on taas laenatud *L. B. Wilson*ilt ja *R. G. Clark*ilt ([61], lk. 26) ning esitatud on just sama algoritm, mis *FORTRAN*iski.

begin comment this program is the ALGOL-60 version of finding the mean
and the number of those greater than the mean;

[1] Seda akronüümi on dešifreeritud kui *Backus-Naur Form* või *Backus Normal Form.*

```
        integer n;
        read(n);
        begin real array a[1:n];
                integer i,number; real sum,mean;
                for i:=1 step 1 until n do read(a[i]);
                sum:=0.0;
                for i:=1 step 1 until n do sum:= sum+ a[ i];
                mean:= sum/n;
                number:=0;
                for i:=1 step 1 until n do
                if a[i]>mean then number:=number+1;
                write(„MEAN=”,mean, „NUMBER OVER MEAN=”,number);
        end
end
```

Üldiselt *ALGOL*i *plokk* paikneb võtmesõnade *begin* ja *end* vahel. Ploki alguses võivad olla lokaalsete (ja alama taseme plokkide(puu)le kättesaadavate) muutujate kirjeldused.

C-programmeerijad: *write* peaks pakkuma äratundmisrõõmu!

3.2.3. Plokkstruktuur

Me võiksime *ALGOL*i plokkstruktuuri käsitleda kui kompensatsiooni masinkoodi ja *FORTRAN*i *CALL*-puu hülgamise eest. Tõepoolest, iga *CALL*-operaatoriga välja kutsutud programm on täiesti sõltumatu väljakutsuja nimedest jms, ja kui ta peab neid teadma, siis antakse talle vajalik üle kas parameetrite või/ja ühisvälja(d)e vahendusel. Sama võime täheldada plokkides: iga plokk on *vaba* üle defineerima talle mittevajalikke ülemise taseme muutujaid ning alamprogramme ja nimetama oma objekte just nii nagu meeldib ─ hoolimata madalama taseme programmidest. Ja kui alumis(e/te) taseme(te) programm(id) tahab/tahavad kasutada ülemise taseme programmi muutujate väärtusi ja alamprogramme, siis tuleb ainult hoiduda nimede ümberdefineerimisest[1].

Üldjuhul *loodi* andmestruktuurid (sh. massiivid) plokki sisenemisel ja *hävitati* plokist väljudes (seda sai siiski keelata kirjelduses reservsõna **own** abil, mis aga tekitas probleeme: näiteks, mis juhtub, kui plokis oli kirjeldatud **own**-massiiv rajadega 1:*N*, aga too *N* oli mittelokaalne muutuja, mille väärtus oli vahepeal muudetud).

Tuleb rõhutada, et plokkstruktuur *pole* alamprogrammide puu. Püüame nende asjade erinevust seletada näitega: *C*-keeles saame programmi tekstis kirjeldada alamprogramme (nii funktsioone kui ka protseduure) ja iga neist võib *pöörduda* teiste samamoodi kirjeldatud alamprogrammide poole, aga alamprogrammi *kirjelduses* ei saa kirjeldada teisi alamprogramme. Ütleme, et *C* toetab alamprogrammide puud. *ALGOL*i plokid võivad see-

[1] Mõtlemiseks lugejale: kas, ja kui, siis millised *objektorienteeritud* programmeerimiskeelte printsiibid on ülaltoodutega analoogilised?

vastu *sisaldada* madalama taseme *plokkide kirjeldusi*, neist igaühes saab kirjeldada omakorda madalama taseme plokke jne.

Esmapilgul tundub, et *ALGOL*i variant ei paku sisuliselt midagi rohkem kui *C*, ent ─ kui meenutame taas, et *ALGOL* konstrueeriti algoritmide publitseerimiseks, siis on plokkstruktuur paremini loetav ja mõistetav. Nii on see „esmapilgul", tegelikult on *ALGOL*i ideoloogia just see kontseptsioon, millel baseerub kogu tänapäeval prevaleeriv *objektorienteeritud paradigma.* Lisagem, et *ALGOL* programmeerimiskeelena annab võimaluse koodi *optimeerimiseks*: kogu lähteprogramm on kättesaadav ja manipuleeritav (*FORTRAN*i *CALL* seda ei võimalda).

3.2.4. Andmed ja operaatorid

3.2.4.1. Andmed

Nood asjad (andmed ja operaatorid) ─ nagu lugeja teab juba „Algoritmide ja andmestruktuuride" kursusest ─ on omavahel lahutamatult seotud: operaatorid on funktsioonid, millede argumendid on tüüpidega muutujad ja andmestruktuurid. *ALGOL* aktsepteeris ainult kolme andmetüüpi: *integer*, *real* ja *Boolean*[1]. Meie distantsilt (50 aastat) vaadatuna tundub arusaamatuna, miks *ALGOL* ei kasuta *viidatüüpi*, ehkki kaasaegsed masinkoodid ja assemblerid võimaldasid täiesti loomulikult absoluutaadresside arvutamist ja kasutamist, ja *ALGOL* lubab dünaamilisi massiive. Põhjus peitub (vist) taas *ALGOL*i orientatsioonis ─ olla publitseerimise keel. Dünaamiline mälujaotus (ja viidatüüp) tundub olevat „tehniline probleem", ent ─ ainult *tundub.* Paljud (*CACM*is) publitseeritud algoritmid on just nii kohmakad nagu nad on ainult tänu viidatüübi puudumisele *ALGOL*is, piirdugem ainult ühe valdkonna näitega. *CACM* avaldas ja sajad teoreetikud publitseerisid ja esitasid artikleid ja (konverentsidel) ettekandeid paisksalvestusmeetoditest, mis likvideerivad põrkeid staatilises tabelis. Iseenesest oli see siis ja on praegu *pseudoprobleem,* kuivõrd dünaamilist mälujaotust kasutav (ent mittepublitseeritav: viidad, mida *ALGOL*i jaoks pole olemas) *välisaheldusmeetod* oli ja on konkurentsitult parim.

*ALGOL*i *massiivid* on homogeensed (kirjelduses fikseeritud tüüpi elementidega) ning raskestimõistetavate rajadega (raske on aru saada, mida annab indeksite muutumispiirkond „-10:20"). Vaikimisi luuakse massiiv plokki sisenemisel (dünaamika) ja kustutatakse plokist väljudes. Kui plokk tahab kahe pöördumise vahel oma massiivi seisu säilitada, siis seda saab teha märksõnaga *own*, näiteks:
own Boolean array B[10:M];

Siin peitub siiski dokumenteerimata oht: mis juhtub, kui M väärtust on vahepeal muudetud? Seetõttu enamus realisatsioone seda võimalust ei toeta.

[1] Nagu ka *FORTRAN*, ei toeta *ALGOL sümboli-* ja *tekstitüüpi*; noid saame kasutada ainult trüki vormistamiseks. Omamoodi on üllatav *complex*-tüübi puudumine, kuivõrd *ALGOL*is publitseeriti ju arvutusalgoritme ning eeskätt neid, mida realiseeriti *FORTRAN*is.

3.2.4.2. Operaatorid

Aritmeetilistel operaatoritel on üldjuhul aparatuurne toetus. Kehtestatud on tehete prioriteet: astendamine (↑), korrutamine ja jagamine (×, /) ning liitmine ja lahutamine (+, –). Ja nagu *FORTRAN*iski, teevad *ALGOL*i realisatsioonid vajadusel varjatud tüübiteisendused.

Võrdlus- ja loogikatehete prioriteedid on järgmised:
1. =, ±,<.>,≤,≥
2. ⌐, Λ, V, ⊃

ALGOL ei tee vahet (süntaksi tasemel) aadressil ja sel aadressil oleval arvul, nii võib kirjutada

A(a,b):=7 ja c:= A(a,b)+1;

ja seejuures esimesel juhul peetakse silmas massiivi A elemendi aadressi ja teisel ─ selle väärtust. Ent *ALGOL* oli esimene keel, kus tehti semantilist vahet *omistamise* (:=) ja *võrdlemise* (=) vahel. Samm algebrast eemale.

Avaldiste ja operaatorite näited laename *T. Pratt*ilt ([44], lk. 379 jj).

Aritmeetiline avaldis on näites omistamisoperaatori paremal poolel:
X:=Y+(**if** A=B **then** A+1 **else** A) ×B;

Loogilise avaldise näide on järgmine (muutujad on kirjeldatud **integer** X ja **boolean** B,C,D,E):
B:=(**if** X>0 **then** C **else** D) ΛE;

Tingimusliku operaator üldkuju on **if <**loogiline avaldis**>** **then** <tingimusteta operaator> **else** <operaator>;

Kui tolle operaatori *tõese* ja *väära* haru operaatorid pole suunamised (**goto**), siis „saavad mõlemad harud kokku" *else*-operaatori järel ─ ei assembler ega ka *FORTRAN* sellist ülevaatlikku võimalust ei paku. *Then*-operaatori lõpus pole vajadust suunamiseks „ühinemispunkti" (kompilaator lisab selle ise).

Tsükli (mis on programmeerimiskeeltes seotud eeskätt massiivi(de)ga) jaoks oli kolm põhivõimalust (ning lubatud olid nende kombinatsioonid):

- **for** <muutuja>:=<väärtuste loetelu> **do** <keha>; näiteks:
  for I:= 2,3,5,11 **do <**keha>;
- **for** <muutuja>:=<avaldis> **while** <loogiline avaldis> **do** <keha>; näiteks
  for K:= K+1 **while** K<N Λ X>0 **do <keha>;**
- **for** <muutuja>:=<algväärtus> **step** <samm> **until** <lõppväärtus> **do** <keha>; näit.
  for I:= 1 **step** 1 **until** N **do** <keha>;

Tsükli „keha" on tavaliselt plokk **begin** … **end**.

Väga keeruliste reebuste huvilistele pakume kurioosumina välja tsüklioperaatori ([44], lk. 381): mida see teeb, miks just niimoodi, ja kas kuidagi lihtsamalt ei saaks[1]?

for I:= 2,3,7,11,15 **step** 1 **until** 20, I+2 **while** I≤N ΛX<0 **do** <keha>;

Masinkoodi ainus (ja *FORTRAN*i peamine) operaatorite täitmise järjekorra muutmise vahend on *suunamisoperaator* (*FORTRAN*i *GO TO*, *ALGOL*i **goto**). *ALGOL*i orientatsioon tekitab kohati raskusi: mis juhtub (programmeerimis- ja mitte publitseerimiskeeles), kui suunamine toimub blokist välja (näiteks, kui *märgend* on alamprogrammi parameeter)? Kas tuleb enne suunamist desaktiveerida „jooksev plokk"?

ALGOL arendas *FORTRAN*i grupiviisilise suunamise vahendeid edasi, lisades *lüliti* (**switch**) kujul (näiteks) **switch** S:=L1,L2,SY,NEXT; ─ kompileerimisaegse *S* väärtuse järgi genereeritakse vajalikud **goto** operaatorid. Meie näite puhul: kui *S*=3, siis genereeritakse direktiiv **goto** SY; Erinevalt tingimusoperaatorist ei garanteerita harude „kohtumispunkti", selle peab (vajadusel) tekitama programmeerija[2].

3.2.4.3. Alamprogrammid

Kõigepealt, *ALGOL* toetab (realisatsioonides tänu dünaamilisele mälujaotusele) *rekursiooni* (rekursiivsest alamprogrammist saab väljuda nii *return-* kui ka *goto-operaatoriga*). Infovahetus alamprogrammide vahel on organiseeritud just nii, nagu eeldab algoritmide publitseerimine: *ühisvälja* ei aktsepteerita[3], ning parameetrite väärtusi saab edastada vaid kirjelduses deklareeritud *formaalsete parameetrite* kohtadel esitatavate *faktiliste parameetrite* väärtustega. *ALGOL* pakub siin kaks võimalust: *väärtuse järgi* (*by value*) ja *makro järgi* (*by name,* ent nende ridade vahendaja leiab selle otsetõlke „nime järgi" eksitava olevat).

Variandil „*väärtuse järgi (by value)*" oli palju erindeid, ent ühine oli, et parameetri väärtus (arvutatakse üldjuhul välja ja) edastatakse alamprogrammi pöördumisel; variandid on:

- arvud või tõeväärtused;
- alamprogrammide nimed;
- märgendid (*sic!*);
- muutujate või massiivide nimed;
- aritmeetilised või loogilised avaldised.

T. Pratt toob näite, kus parameetriteks on nii protseduur kui ka märgend ([44], lk. 248):

[1] Selle õppevahendi koostaja ei oska vastata kahele esimesele küsimusele, kolmandale ehk küll: kasutada tuleks programmeerijate vennaskonna vana printsiipi *KISS* (*Keep It Simply, Stupid!*).
[2] Vrd. *C switch*-operaatorit, kus nii iga *case*-operaatori kui ka kogu *switch*i mõjupiirkonnad „võetakse kokku".
[3] Siiski, me võime interpreteerida ühisväljadena kõrgema taseme blokis kirjeldatud ja madalama(te) taseme(te) blok(i/kide)s üledefineerimata muutujaid ja massiive. Visuaalselt on kõik kontrolli all!

```
B: begin integer N;
        procedure P(X,C); value C; procedure X; integer C; begin
                procedure Q(T); label T; begin
                        N:=N+C;
                        X(K);
                        goto T;
                end proc Q;
                J: if C>N then X(J) else P(Q,C+1);
                K: N:=N+C;
                goto L
        end proc P;
        procedure Q(L); label L; begin
                N:=N+1;
                goto L
        end proc Q;
        N:=3;
        P(Q,4);
        L: print(N)
end block B;
```

Makro järgi (by name) näiteks toome nn. *Jenseni võtte*[1] ([44], lk. 258): alamprogrammile antakse ette mingi (ühe või rohkema muutujaga, nende väärtused edastatakse eraldi *väärtuse järgi*) avaldis, näiteks:

```
real procedure SUM(EXPR,INDEX,LB,UB) value LB,UB;
        real EXPR; integer INDEX,LB,UB;
begin real TEMP; TEMP:=0;
        for INDEX:=LB step 1 until UB do
                TEMP:=TEMP+EXPR;
        SUM:=TEMP;
end  proc SUM;
```

Selle protseduuri kasutamise näiteid:
SUM(A[I],I,1,25) *leitakse massiivi A 25 esimese elemendi väärtuste summa*
SUM(A[I]×B[I],I,25) *leitakse vektorite A ja B 25 esimese elemendi korrutiste summa*
SUM(C [K,2],K,1,25) *leitakse massiivi C 2. veeru esimese 25 elemendi väärtuste summa.*
Kolmas parameetrite edastamise variant on *viida järgi* ─ ainuvõimalik *FORTRAN*is ja võimatu *ALGOL*is. Tuletagem meelde, et selle variandi korral edastatakse alamprogrammidele parameetrite *aadressid*, ent nende abil osutub alamprogrammile võimalikuks oma sisendparameetreid üle kirjutada (mäluseisu varjatud muutmise võimalusi nimetatakse tavaliselt *kõrvalefektideks*).

[1] See "võte" oli *ALGOL-60* järeltulija *Simula 67* arsenalis.

3.2.5. Realisatsioon

ALGOL realiseeriti kompilaatori(te)ga; ja kuivõrd ta polnud mõeldud programmeerimiskeeleks, siis translaatorite tegijatel olid vabad käed omapoolseks tõlgenduseks. Eeskätt käib öeldu sisendi ja väljundi kohta.. Tuletades meelde 50.-ndate aastate lõpu defitsiite (mälu ja masinaaeg), siis on arusaadav, et ka *ALGOL* püüti realiseerida võimalikult efektiivselt, seda soodustab eeskätt püüd maksimeerida aparatuurset interpretatsiooni (*á la FORTRAN*), ent see oli raskendatud: masinkood ei toeta vahetult ei dünaamilisi massiive (näiteks indekseerimiseks tuli kasutada *deskriptoreid*[1]), plokkstruktuuri viitade keskkonda ja ka üldisemaid ja loetavamaid juhtimisstruktuure. Ja kõik, mida ei saa interpreteerida aparatuurselt, tuleb virtuaalmasinal interpreteerida programselt. Ning tulemusena ei suutnud *ALGOL* efektiivsuses konkureerida *FORTRAN*iga.

ALGOL-programmid ei vaja komplekteerimist (käivitamiseks piisab paigaldajast). Loetavuse aspektist on see muidugi hea, ent *moodulprogrammeerimist* see stiil ei toeta.

Siin on omad head ja vead. Vaieldamatult *hea* on see, et programmeerijal on *kogu* tema kood nähtav ja silutav ning kontrollitav.

Paha on see, et tarkvara akumulatsioon on vähemasti pärsitud. Olgu, me saame näiteks *C*-keeles kasutada #*include*- makrosid ning kasutada lihtfunktsioone (näiteks neid, mida pakuvad <*string*>, <*math*> jne), ent need on pelgalt madalaima taseme vahendid (üldjuhul funktsioonid) ja meil puudub võimalus lihtsalt välja kutsuda suuri plokke, näiteks maatriksalgebra või faktoranalüüsi omi[2]. Teisisõnu, pole *CALL*-operaatorit, mis on omane assemblerile ja *FORTRAN*ile, ja mis on välistatud *ALGOL*is ja tema järgijates (*C* annab siiski võimaluse käivitada *käsurea* vahendusel sõltumatuid programme[3]).

Ja ikkagi, ärgem unustagem, et *ALGOL* polnud mõeldud realiseerimiseks, vaid lugemiseks. Algoritme realiseeriti *FORTRAN*is. Viimases pidi programmeerija usaldama *CALL*-käsuga väljakutsutavat koodi, *ALGOL* aga pidi publikatsiooni lugejat informeerima kõigest, mida täpselt alamprogrammides (sisuliselt *ALGOL*i plokkides) tehakse. Arvata võib, et kui *ALGOL*i autorid oleks aimanud, et nende keelt hakatakse realiseerima programmeerimiskeelena, ja et nende keel saab prototüübiks väga paljudele hilisematele programmeerimiskeeltele, oleks püütud säilitada *CALL*-operaatorit.

[1] Deskriptoreid käsitleme hiljem, vt. 4.3.8.

[2] Too „suur plokk" on loomuldasa *iseseisvalt kasutatav programm* ja mitte pelgalt *lihtne funktsioon*. *CALL*-tehnika võimaldab elegantselt lahendada ühiskasutatavate mälupiirkondade probleemi *ühisväljade* abil; et need oleksid tõepoolselt *ühised*, on komplekteerija ja paigaldaja „mure". *ALGOL*i, *C* ja teiste sama pere keelte kontseptsioon välistab tõepoolest *suurte ja iseseisvate* (väljakutsuja seisukohast) alamprogrammide integreerimise.

[3] Ent nondega on infovahetus mitmeti piiratud. Näiteks võib väljakutsuja enne „alamprogrammi" väljakutsumist kirjutada oma ühiskasutatavad andmestruktuurid kettale, ent see „ahistab" väljakutsutava autonoomsust (too tuleks kirjutada nii, et ta algandmed *ongi* kettal ja mitte enda genereeritud (või muul moel saadud)).

4. SÜSTEEMPROGRAMMEERIMISE KEELED

Kordame üle: *FORTRAN* disainiti arvutusmatemaatika ülesannete programmeerimise hõlbustamiseks ja *ALGOL* nonde ülesannete algoritmide publitseerimiseks. Tol ajal, millal nad loodi, piisas sellest täielikult. *FORTRAN* realiseeriti kas masinkoodis või assembleris, ja *ALGOL* polnud üldse mõeldud realiseerimiseks.

Tänaseks on keerulised teadusarvutused endiselt tähtis arvutite kasutamise valdkond, ent sedatüüpi ülesannete osatähtsus on kardinaalselt vähenenud. Enamgi veel, arvutite kasutamine *programmeerimiseks* on osatähtsuselt taandunud sama marginaalseks. Ent üks seik on püsinud muutumatuna: programmeerimine on endiselt elitaarne tegevus, millega tegelevad eeskätt eriettevalmistusega spetsialistid. Põhimõtteliselt tegutsevad programmeerijad kas rakendusprogrammide kirjutamisega (arvutusülesanded, mängud, toimetid, päringusüsteemid jmt.) või tarkvara loomisega rakendustarkvara tegijatele (translaatorid, operatsioonisüsteemid, hüperteksti(de) interpretaatorid jne.) ─ neid nimetatakse tavaliselt süsteem(i)programmeerijateks.

Rakendusprogrammeerijate tarbeks on alates *FORTRAN*ist ja *ALGOL*ist konstrueeritud hoomamatu arv programmeerimiskeeli.

Süsteemprogrammeerijatele sobivad sisuldasa parimini *masinorienteeritud* keeled ja nende abil luuakse ka tänapäeval kõige „sensitiivsemad" süsteemsed moodulid. Ent juba enne masinast sõltumatute programmeerimiskeelte, eritüübiliste masinate paljuse ja noile orienteeritud masinkeelte buumi sai selgeks, et ka süsteemprogrammeerijad vajavad just nende vajadustega arvestavaid keeli, ja need ilmusidki (nagu lihtne taibata, siis konstrueerisid nood keeled süsteemprogrammeerijad ise ja mitte teoreetikud). Selliseid keeli pole kuigi palju, kuivõrd nad peavad rahuldama spetsiifilisi tingimusi, sh. mitmeid taolisi, missuguste pakkumist rakendusprogrammeerijatele peeti mittevajalikuks või isegi „ohtlikuks". Ohtlikkuse näiteks: võimalus suhelda vahetult operatsioonisüsteemiga (sh. juurdepääs näiteks *Microsofti* puhul *DOS*i ja *BIOS*i katkestustele), vahetu juurdepääs videomälule ja kõvaketta „juhttabelitele", kontrollimatu (seeläbi ka kitsendusteta) juurdepääs kogu operatiivmälule jne.

Allpool käsitleme põgusalt kaht süsteemprogrammeerimise keelt: need on *FORTH* ja *C*. Kuivõrd nii *FORTH*i kui ka *C* mõningatest originaalsetest joontest on kahtlemata võimalik paremini aru saada, evides põgusat ettekujutust nende jaoks parimast masinast, *PDP-11*st, siis üritatakse allpool seda pakkuda, tuginedes eeskätt *Richard H. Eckhouse'*i ja *L. Robert Morris*e raamatule [15]. Nad mõlemad (*C* ja *FORTH*) olid loomuldasa orienteeritud (olemata muidugi masinorienteeritud keeled) miniarvutile *PDP-11*[1].

[1] Kompanii *DEC* (*Digital Equipment Coproration*) rajati 1957. a. ühe mahajäetud veski ruumides Maynardis (USA, Massachusetts, suur-Bostonist pisut läänes). Tootma hakkasid nad suhteliselt odavaid integraallülitusi (esimene kokkupandud arvuti, *PDP-1*, läks müüki ca 120 000$ eest, muude firmade masinate hinnad algasid miljonist). Akronüüm *PDP* on *Programmed Data Processor*. Nn „sotsialismimaad" kloonisid *PDP*-arvuteid suhteliselt edukalt arvutitüüpide *CM* (*Серия Малая*, mudelid *CM-4, CM-1420, CM-1600* jt), *ДВК, Электроника* jt. arvukate mudelite näol.

4.1. Miniarvuti *PDP-11*

PDP-11 oli *DEC*-i esimese 16-bitise protsessoriga masina *PDP-8* edukas edasiarendus. Vanemad mudelid kuni *PDP-7*-ni olid 18-bitised ja protsessori tasemel ühildatavad, so neil oli sama käsusüsteem, seega ka assembler. *PDP-11* oli arvutiehituse seisukohalt mitmeti innovaatiline ning sai paljuski eeskujuks mikroarvutitele, sh. *Apple][* ja *IBM PC*. Ja selle masina arhitektuur määras paljuski *C*-keele disaini. Mudelit müüdi möödunud sajandi 70.-ndatel ja 80.-ndatel aastatel (vt.[84], [86]).

Joonis 4.1a. Osa miniarvutist *PDP-11/20* [80].

Seda mudelit on, arvestades tõigaga, et lugeja teab juba üht-teist pesamasinatest, *IBM* suurarvutitest (baitmasinad) ja mikroarvutitest (tõsi, ainult *Intel*ist), suhteliselt lihtne tutvustada.

Niisiis, *PDP-11*. Arvuti on arhitektuurilt pigem bait- kui pesamasin. Sõna pikkus on 2 baiti ja operatiivmälu koosneb 64-kilobaidistest plokkidest (so, 32 kilosõna). Käskude tasemel on adresseeritavad nii baidid kui ka sõnad (nende aadressid on paarisarvulised). Kahebaidine sõna on mälus just samuti nagu mikroarvutites: väiksemal aadressil on „tagumine" ja suuremal „esimene" pool, näiteks 16-ndarv „AB" on mälus kujul „BA". Andmete kujutamiseks saab pesi integreerida (näiteks, kasutada 64-bitiseid välju reaalarvude jaoks).

Masinal on 8 üldregistrit (R0..R7), sj. R6 ja R7 on reserveeritud: R6 „alamprogrammide aparatuurse magasini" viidale (*Stack Pointer, SP*) ja R7 on üllatuslikult käsuloendaja jaoks (*Program Counter, PC,* ja see pole protsessori kiivalt kaitstud koosseisus!). Masina *jooksva seisundi* hoidmiseks on kasutusel *protsessori seisundi sõna* (*PSW = Program Status Word*), mis koosneb kahest osast: lisaks juba mainitud *PC*-le *PS ─ Processor Status.* Viimast muudavad *katkestused*, nii aritmeetika-loogikaseadme ja välisseadmete kui ka programselt tekitatud, ent nii, et enne *PSW* modifitseerimist säilitatakse vana *PSW*. Vähe tähtis pole seik, et programse katkestusega on võimalik viia programm seisundisse *süsteem*, millega tagatakse vahetu programne juurdepääs arvuti kõikidele ressurssidele.

PDP-l on *aparatuurselt toetatav* magasin alamprogrammide süsteemi võimaldamiseks (viit R6). Lisaks saab moodustada „eramagasine", mille viit on suvalises üldregistris (välja arvatud R6 ja R7) ja mida toetavad (indekseeritavad) *increment*- ja *decrement*-režiimid. Tuletagem meelde, et „aparatuurselt toetatav magasin" ei tähenda eriliselt ehitatud mälu, vaid protsessori aparatuurset toetust magasinidega (so, neiks määratud operatiivmälu piirkondadega) opereerivatele käskudele.

Käsud on formaaditi 0-, ühe- ja kaheaadressilised. Seega käsu pikkus on varieeruv. Erinevalt *IBM*i masinkoodist ja sarnaselt *Inteli* koodile pole *PDP* masinkood ilma dekodeerimiseta loetav. „Nullaadressiliste" käskude operandid on ilmutatud kujul register ja ilmutamata kujul magasini tipmine element. (käsu pikkus on 2 baiti).

Iga käsk on vähemalt kahebaidine (kahendkohad 15..0, tuletagem meelde sõna baitide *tegelikku* järjekorda).

4.1.1. Üheaadressilise käsu formaat

	k	ä	s	u		k	o	o	d	rež	iim	@	re	gis	ter
15									6	5	4	3	2		0

Nagu näeme, on bittidel 6..15 *käsu kood*. Bittidel 0..2 on käsu ühe operandina esineva re*gistri number*, bitid 3..5 näitavad, mis rollis too register esineb (*režiim*), ja seejuures 3. bitt näitab, kas kasutada tuleb otsest või kaudset adresseerimist.

Otseadresseerimise jaoks on *režiimid* 0,2,4 ja 6 (so, paarisarvulised, kaheksandnumbritena 000, 010, 100 ja 110, 3. bitt kõigil 0). Variandid on järgmised:

- **Režiim 0:** Register-adresseerimine: näiteks, assemblerkäsk *INC R3* (semantika: (R3) + 1 → R3) on masinkoodis 005203, dešifreeritult:
  0052 on operatsiooni kood
  0_8 (so, 00=režiim ja 0=@) määravad *registerrežiimi* ja otseadresseerimise. Kaheksandnumber 3 määrab registri R3 ─ käsu operandi. Näiteks, kui enne käsu täitmist (R3)=000005, siis pärast täitmist (R3)=000006.

  INC tähistab sõna *increment* ─ selle all mõistetakse „kasvavat" autoindsekseerimist: (ilmutatud) indeksregistri seisu suurendatakse, olenevalt operandi pikkusest, kas 1 või 2 võrra pärast tsükli keha täitmist; „kahaneva" analoogilise autoindekseerimisrežiimi nimetus on *decrement* (indeksregistri seisu vähendatakse enne tsüklisammu). Mõistagi, see režiim pole jäigalt seotud indeksregistritega.

  Selle režiimi käsud on 16-bitised (pikkuseks on 1 sõna)[1].

[1] Meie senise käsitluse kohaselt on selle grupi käsud 0-aadressilised, kuivõrd käsus puudub *mäluaadress.* Ent järgigem selles jaotises *PDP* terminoloogiat; selle kohaselt pole 0-aadressilisel käsul *üldse* operande (nagu meie õppearvuti STOP-käsul).

- **Režiim 2** (autoinkrementne) ja **4** (autodekrementne). Käsus näidatud register sisaldab *operandi aadressi.* Toda aadressi kas suurendatakse (pärast käsu täitmist) või vähendatakse (enne käsu täitmist) aadressil oleva operandi pikkuse[1] võrra. Näiteks, assemblerkäsud NEG (R5)+ ja NEGB (R5)+ on *autoinkrementsed*, esimese operand on registris 5 oleval aadressil asuv *sõna*, teisel aga *bait*. Käsk asendab operandi väärtuse tema täiendkoodiga (kuni 2-ni). Assemblerprogrammeerijal on suvalise tolle režiimi käsu kodeerimisel vabad käed, kas ta tahab aadressi muuta enne või pärast käsu täitmist, näiteks kui ta kirjutab NEG –(R5), siis määrab ta *autodekrementse* režiimi. Käsk NEG (R5)+ on masinkoodis 005425, dešifreeritult: kood on 0054, režiim on 010 (so, režiim 2) ja komponent @ on 0, mis määrab otseadresseerimise. Viimane kaheksandnumber „5" määrab operandi aadressi sisaldava registri.
- **Režiim 6**: indeksrežiim (110). Selle formaadi käsud on kahesõnalised, so, nende pikkus on 32 bitti. Käsukoodi esitava 1. sõnale järgneb meile harjumuspärases mõistes „indeksregistri sisu" 2. sõnas. Üldkuju (assembleris) on *Kood n(*R_i*)*, näiteks CLR 200(R4) on mälus 005064 000200 ja mille toimel registris R4 sisalduvale aadressile liidetakse indeks 200 (vt[2]).

Kaudse adresseerimise määravad režiimid 1, 3, 5 ja 7. Nende režiim kodeeritakse kui $xx1_2$, so. 001, 011,101 või 111. Käsus antud registri sisu interpreteeritakse kui operandi 16-bitilist aadressi, ning järgitakse *increment-* ja *decrement*-variante. Kaudne indeksrežiim tähendab, et enne leitakse operandi aadress ja siis modifitseeritakse see (käsu viimase komponendina antud) indekssõna väärtusega.

Assemblerkeeles osutab kaudsele adresseerimisele sümbol @ aadressvälja alguses. Toogem näiteid:

- **Režiim 1:** kaudne register-adresseerimine. Näiteks INC @R5 (masinkoodis 005215 ─ kood 0052, režiim 001) suurendab R5 väärtusena esitatud aadressil olevat arvu. Kui (R5)=001700 ja enne käsu täitmist (001700)=000100, siis pärast (001700)=000101.
- **Režiim 3:** kaudne autoinkrementne režiim. Näiteks NEG @(R2)+ (005432). Siinkohal toome ära Eckhouse'i ja Morrise täisnäite ([15], lk. 74):
  Enne käsu täitmist (R2)=010300 ja mälus aadressidel (001010)=000001 ja (010300)=001010.
  Pärast käsu täitmist (R2)=010302, (001010)=177777 ja (010300)=001010.
- **Režiim 5:** kaudne dekrementne režiim. Näiteks COM @ -(R0) (kood 005150) täidetakse nii: registri R0 sisu (mäluviit) vähendatakse 2 võrra ning resultaati interpreteeritakse kui selle operandi aadressi, mille väärtus asendatakse tema täiendkoodiga (kuni 1-ni). Kui enne käsu täitmist (R0)=010776, (010100)= 000000 ja (010774)=010100, siis pärast (R0)=010774, (010100)=177777 ja (010774)=010100.

[1] Operandi pikkuse määrab assembleris sufiks „B" (1 bait); masinkoodis on selleks 1 võrra erinevad koodid, vaikimisi (ilma sufiksita B) on operandi pikkuseks 1 sõna (2 baiti).

[2] Juhime lugeja tähelepanu huvitavale seigale: indekseeritava mälulõigu baasaadress määratakse kui (R)+ (indekssõna), ent tsüklis *ei muudeta* indekssõna sisu, vaid *registris* antud aadressi (inkrement- või dekrement-võttega).

- **Režiim 7:** kaudne indeksrežiim. Näiteks käsuga CLR @ 1000(R2) (005072 001000) summeeritakse (R2) ja indekssõna (=001000), resultaat on mäluaadress, mille sisu nullitakse. Kui (R2)=000100, (001050)=007777 ja (001100)=001050, siis pärast käsu täitmist (001050)=000000.

4.1.2. Kaheaadressilise käsu formaat

k	o	o	d	rež	iim	@	re	gis	ter	rež	iim	@	re	gis	ter
15			12	11	10	9	8		6	5	4	3	2	1	0

Esimest operandi (bitid 6..11) käsitletakse kui „allikat" (*source, SRC*) ja teist (bittidel 0..5) kui „vastuvõtjat" (*destination, DST*); kui käsk eeldab, et resultaat tuleb salvestada, siis kirjutatakse üle *DST*. Kumbagi operandi saab esitada täpselt samade võimalustega nagu tutvusime üheaadressiliste käskude ülevaates; niisiis, kui kumbagi operandi ei indekseerita, on käsu pikkus 1 sõna (16 bitti), kui indekseeritakse ühte operandi, siis on pikkus 2 ja kui mõlemit, siis 3 sõna. Mõned üldised näited (vt. [15], lk. 77):

tegevus	**kood**	**resultaat**
Saada (*Move*)	MOV	(SRC) → DST
Liida (*Add*)	ADD	(SRC)+(DST) → DST
Lahuta (*Subtract*)	SUB	(SRC)-(DST) → DST
Võrdle (*Compare*)	CMP	(SRC)-(DST), märk → lipp
Loogiline liitmine (*Bit set*)	BIS	(SRC)V(DST) → (DST)

Toogem äsjaviidatud leheküljelt ka ühe konkreetse näite:

assembler	**masinkood**	**kirjeldus**
ADD 30(R2),(R1)+	066221 000030	(R2)+(indekssõna)=allik-operandi aadress; seal olevale arvule liidetakse arv, mille aadress on R1-s, pärast liitmist (R1)=(R1+2)
Enne käsku: (001000)=066221 (001002)=000030 … (001130)=000001 … (001500)=000025	 (R1)=001500 (R2)=001100	**Pärast käsku:** (R1)=001502, (001000)=066221 (001002)=000030 (R2)=001100 (001130)=000001 … (001500)=000026

Suunamine (*BEQ, BNE, BPL, BMI* ja *BR*) on kahebaidine käsk; „vanema baidi" 7 bitti esitavad käsu koodi, 8. nihke märki ja „noorem bait" *nihet* suunamiskäsu enda aadressi suhtes (tagasisuunamiseks maksimaalselt 200_8 sõna või 400_8 baiti ja edasisuunamiseks vastavalt 177_8 või 376_8)[1].

[1] Mooduli või alamprogrammi piires suunamiseks pole need piirid oluliseks kitsenduseks.

Käsuloendaja: R7. Tuletagem meelde, kuidas protsessor manipuleerib käsuloendajaga[1]: mälust loetakse käsuloendajaga viidatud käsk, suurendatakse käsuloendaja seisu käsu pikkuse võrra, ja täidetakse loetud käsk.

Kuivõrd *PDP* kasutab käsuloendaja-registrina *üldregistrit* R7, siis saab seda piiranguteta kasutada just samuti nagu teisigi üldregistreid (so, R0..R6), ent ilma erilise vajaduseta R6 ja R7 „näppimine" pole siiski soovitav. Käsuloendajaga manipuleerimise aktsepteeritavad variandid on *asukohast sõltumatu koodi* kirjutamine ja töö *struktureerimata andmetega*. Neid variante toetavad neli režiimi: *vahetu, absoluutne, suhteline* ja *kaudselt suhteline.*

režiim	**nimetus**	**assembleris**	**funktsioon**
010	vahetu	#n	Operand *n* paikneb vahetult käsu järel
011	absoluutne	@#n	Operandi *n aadress* paikneb vahetult käsu järel
110	suhteline	A	Operandi suhtaadress (täidetava käsu suhtes) paikneb vahetult käsu järel
111	kaudselt suhteline	@A	Vahetult käsu järel paikneb aadress, millel on operandi suhtaadress

„Režiim" on sisuliselt identne üheaadressiliste käskude režiimidega, üldregistri numbrit pole näidatud ─ selles rollis on käsuloendaja-register R7 (*PC*). Toogem näited (vt.[15], lk. 80..82).

assembler	**masinkood**	**kirjeldus**
ADD #10,R0	062700 000010	(R0)+10→ R0
Enne käsku: (001000)=062700 (001002)=000010	 (R0)=000020 (PC)=001000	**Pärast käsku:** (R0)=000030, (001000)=062700 (001002)=000010 (PC)=001004

Tabel 4.1.2a. Vahetu režiim.

assembler	**masinkood**	**kirjeldus**
CLR @#1100 Mõistagi, suhtaad-ressi 001100 asemel võib olla etikett.	005037 001100	Puhastada pesa aadressiga 001100
Enne käsku: (001000)=005037 (001002)=001100 … (001100)=012345	(PC)=001000	**Pärast käsku:** (PC)=001004, (001000)=005037 (001002)=001100 (001100)=000000

Tabel 4.1.2b. Absoluutne režiim.

[1] PDP nimetab seda registrit *PC* (*Program Counter*).

assembler	masinkood	kirjeldus
INC A	005267 000074	Liidetakse arv 74 ja (PC), Tulemuseks on aadress A. A:=A+1.
Enne käsku: (001000)=005267 (001002)=000074 … (001100)=001200 *Märkus:* 1004+74=1100	(PC)=001000	**Pärast käsku:** (PC)=001004, (001000)=005267 (001002)=000074 (001100)=001201

Tabel 4.1.2c. Suhteline režiim.

assembler	masinkood	kirjeldus
CLR @A	005077 000020	(PC)+(indekssõna)=aadress A; (A)=pesa aadress, mis tuleb puhastada
Enne käsku: (001000)=005077 (001002)=000020 … (001024)=010100 … (010100)=135531 *Märkus:* 1004+20=1024	(PC)=001000	**Pärast käsku:** (PC)=001004, (001000)=005077 (001002)=000020 (001024)=010100 … (010100)=000000

Tabel 4.1.2d. Kaudselt suhteline režiim.

Toome viimase režiimi kohta näiteks kaks assembler-direktiivi, mis transleeritakse täpselt samamoodi (vt [15, lk. 83]):

HERE: ADD X,Y
HERE: ADD X-HERE-4(PC),Y-HERE-6(PC)

Niisiis, PDP adresseerimisviis erineb oluliselt sellest, millega oleme harjunud (operandi absoluutne aadress A=D+(B)[+(I)] ─ alamprogrammis adresseeritakse objekte baasaadressi asemel käsuloendaja (R7, *PC*) jooksva seisu suhtes: A=D+(PC), kusjuures *D* asub „indekssõnas" vahetult käsukoodi esitava pesa järel.

4.1.3. Moodulid

Selles jaotises tuleb meil paratamatult[1] teha *PDP* „alamprogrammide"[2] kasutamise võimalustest suunatud valik. Valiku kriteeriumiks on see, kuidas vastavad vahendid on mõjutanud keelte *FORTH* ja *C*, aga ka *Lisp*i vahendeid (või on nendega *seotud*?). Kõigil huvilistel, kes tahavad teada *PDP* kõiki võimalusi, soovitame lugeda [15] või otsida materjale veebist.

Tuletagem meelde, kuidas *PDP* interpreteerib magasini (*LIFO*-stack). Suhtlemiseks alamprogrammidega on *magasini viit* (*SP*=*Stack Pointer*=R6), ja *increment*- ja *decrement*-režiimid võimaldavad luua „eramagasine"[3]. Magasinina interpreteeritava mälulõigu algaadress (aken) on alati magasini „põhjas", so, suurima aadressiga element. Magasini lisamine toimub aknast väiksemale aadressile (*decrement,* enne vähendatakse magasini viita ja siis kirjutatakse), ja eemaldamine toimub nii, et loetakse element „aknast" ja pärast seda suurendatakse viida väärtust (*increment*), so pärast seda on magasin „lühem". Magasini viit osutab alati aknale.

„Eramagasini" kirjutamiseks („PUSH") saab kasutada *autodekrementset* meetodit kujul MOV ITEM, -(R_i) ja lugemiseks (loe: eemaldamiseks, „POP") *autoinkrementset* meetodit kujul MOV (R_i)+, ITEM[4].

4.1.3.1. Aparatuurne tugi

Käsk *JSR*:

k	ä	s	u	k	o	o	d	**re**	**gis**	**ter**	mee	tod	**re**	**gis**	**ter**
15							9	8		6	5	3	2		0

Käsk *RTS*:

k	ä	s	u						k	o	o	d	**re**	**gis**	**ter**
15												3	2		0

Käsuga *JSR* pöördutakse alamprogrammi ja käsuga *RTS* naastakse alamprogrammist. Käsu *JSR* kahendkohtadel 6..8 näidatud register on sama, mis *RTS*i kahendkohtadel 0..2. Näiteks, kui alamprogrammi *SUBR* pöördutakse käsuga

JSR R5,SUBR

[1] tingituna piiratud mahust ja üldisest suunitlusest ─ näidata, et ei masinad ega ka programmeerimiskeeled ei kujunenud *asjadena iseeneses*, vaid praktikute poolt tunnustatud keelte puhul puhtalt pragmaatiliste kaalutluste tulemusena

[2] „Alamprogramm" viitab üheselt hierarhilisele struktuurile; „ülemus-alluva" suhteta moodulid suhtlevad omavahel kui *kaasprogrammid*, neid käsitleme selles jaotises pisut hiljem (too mõiste tuli meil juba varem jutuks).

[3] Neis on mugav säilitada väljakutsuvate moodulite registreid ja edastada parameetreid (kui mingitel kaalutlustel ei taheta selleks kasutada R6 abil toetatavat süsteemset magasini).

[4] Kui magasinielemendi pikkus on 1 bait, siis tuleb meie näidetes kasutada direktiivi „MOVB".

siis too register on R5 ning etikett *SUBR* transleeritakse masinkoodi bittide 0..5 väärtuseks. Kui käsu *JSR* aadress on 1000 ja alamprogramm *SUBR* algab aadressilt 1064, siis enne käsu täitmist võib olla selline seis (vt. ([15], lk. 105):

(R5)=000132 *suvaline väärtus*
(R6)=001776 (=SP) *StackPointer*
(R7)=001000 (=PC) *käsuloendaja, JSR-käsu aadress*

(001772)=?????? (..001772..002000) *magasin*
(001774)=??????
(001776)=xxxxxx (← SP)
(002000)=yyyyyy,

siis pärast käsu täitmist on seis järgmine:

(R5)=001004 *naasmisaadress=001000+4*
(R6)=001774 *StackPointer*
(R7)=001064 (=PC) *käsuloendaja=AP algusaadress*

(001772)=??????
(001774)=000132 (← SP) *R5 vana seisu salvestus*
(001776)=xxxxxx
(002000)=yyyyyy

Käsuga RTS R5 tehakse järgmised tööd:
(R5) → (PC)
(SP)+ → (R5)

Tavaliselt kasutatakse JSR- ja RTS-käskudes registrina käsuloendaja-registrit, eelmise näite jaoks JSR PC,SUBR ja RTS PC. Sel juhul kirjutatakse JSR-ga magasini naasmisaadress, mida kasutab RTS. Ja, mõistagi, SP kasutamine nois käskudes registri rollis annab transleerimisvea.

Kui alamprogramm kasutab registreid, siis peab ta tööd alustama nende seisude kirjutamisega magasini ja lõpetama nende taastamisega, näiteks:

```
SUBR: MOV  (R0),-(SP)
      MOV  (R1),-(SP)
      …                 SUBR keha
      MOV  (SP)+,R1
      MOV  (SP)+,R0
      RTS  PC
```

4.1.3.2. Parameetrite edastamine

Triviaalne variant kasutab globaalseid muutujaid (sisuliselt ühisvälju) või registreid; huvitavamad on võimalused edastada parameetreid ilmutatud kujul koos *JSR*-käsuga ja viidaga parameetrite loetelule. Allpool on mõned näited (vt. [15,] lk. 107 jj).

Esiteks, koos käsuga edastatakse sisendparameetrite *väärtused*:

```
        JSR         R0,MULT
        .WORD       XVALUE,YVALUE
```

Alamprogrammis *MULT* võivad olla järgmised käsud:

```
MULT:   MOV  (R0)+,R1
        MOV  (R0)+,R2
        ….
        RTS  R0
```

Teiseks, edastatakse *viidad* parameetritele:

```
        JSR         R0,MULT
        .WORD       XADDR,YADDR
```

Sel juhul võib *MULT* parameetrite väärtusi lugeda nii:

```
MULT:   MOV  @(R0)+,R1
        MOV  @(R0)+,R2
        ….
        RTS  R0
```

Kolmandaks, parameetrite nimistu aadressi võib laadida registrisse (sel juhul pole edastatavate parameetrite arv piiratud näiteks (vabade ja kasutatavate) registrite arvuga). Näide:

```
MOV  #POINTER,R1
JSR  PC,SUBR
```

Kui alamprogramm *SUBR* peab summeerima kahe etteantud parameetri väärtused ja tagastama summa 2. parameetri väärtusena, siis võib seal olla järgmine käsk:

```
ADD  (R1)+,(R1)
```

Sama töö teeb käsk

```
ADD  (R1),2(R1)
```

Juhime lugeja tähelepanu seigale, et kahe viimase variandi puhul saab alamprogramm „kätte" parameetri *aadressi* ja seega on tal „vabad käed" etteantud väärtuse ülekirjutamiseks. Üldjuhul on see ülemise taseme programmi poolt aktsepteeritud tegevus (vt. meie viimast näidet), ent võib juhtuda, et pole. Heal juhul on alamprogrammi käitumine täpselt dokumenteeritud, ent on ka erandeid. Üldiselt, juhtu, kui alamprogrammi poole pöörduva mooduli mäluvälju muudetakse „ilmutamata kujul", nimetatakse *kõrvalefektiks*. See probleem tekkis alates *FORTRAN*ist (loe: kõrgtaseme keeltest) ja oli kontrolli all masinorienteeritud keeltes. Nois viimastes on terminoloogias väljendid „*by value*" (edastatakse parameetri väärtus ja mitte aadress) ja „*by address*", kui kõrvalefekt on reaalselt kasutatav. *FORTRAN* edastab parameetreid (reeglina) aadressi järgi.

4.1.3.3. Baseerimine

Nagu lugeja on loodetavasti tähele pannud, pole *PDP*-arhitektuuris baseerimisvõimalusi (so, vahendeid, mis võimaldaksid operandi aadressi arvutada kui suhtaadress (alamprogrammis) + („baas"= alamprogrammi algusaadress) + („indeks"). Baseerimist asendas nn *PIC*-tehnoloogia (*PIC=Position-Independet Coding*, vabas tõlkes „nihkestsõltumatu kodeerimine"). Et suvalise (alamprogrammi) käsu täitmisel on registris 7 (*PC*) tolle käsu absoluutaadress, siis *PIC* tähendab, et assembler adresseerib alamprogrammi objektid (märgendid, muutujad ja konstandid) *PC* suhtes (inimesele on see tüütu, ent assemblertranslaatorile rutiinne töö). Vahe on selles, et kui baseerimist toetav arhitektuur positsioneerib alamprogrammi lokaalsed objektid mooduli alguse suhtes (=baasaadress), siis siin tehakse seda alamprogrammi käsu absoluutse aadressi suhtes. Mis tähendab, et viit ühele ja samale mooduli objektile on selle mooduli eri käskudes erineva väärtusega[1].

4.1.3.4. Re-enteraablus, rekursiivsus ja kaasprogrammid

PDP-11 oli orienteeritud mitme-kasutaja-süsteemidele, nood töötavad ajajaotusrežiimis. Mälu ökonoomseks kasutamiseks oli oluline, et need moodulid, mille poole pöörduti mitmes ülesandes (näiteks *FORTRAN*-translaator) oleksid mälus ainult ühes eksemplaris, nende töö võib katkestada kõrgema prioriteediga ülesanne ja kasutada neid ise, aga seejärel saab madalama prioriteediga ülesanne jatkata mooduli täitmist punktist, kus see pooleli jäi. Et moodul oleks kasutatav simultaanselt, tuleb ta kirjutada *re-enteraablina*, so. ta koosneb ainult „puhtast koodist" ilma lokaalsete muutujateta ning tema kood ei modifitseeru täitmise käigus. Väljakutsuva mooduli registrid säilitatakse loomulikult magasinis.

Re-enteraablus on vältimatu omadus ka „ühe-kasutaja-programmis", kui osutub otstarbekaks kirjutada *rekursiivne* programm. Näite laename raamatust ([15], lk. 118): seal on kaks moodulit, *FACT,* mis peab arvutama etteantud naturaalarvu *N* faktoriaali, ja rekursiivne moodul *FAC*, mis arvutabki *N!*

[1] Tuletagem meelde, et assemblerkeeltes on tavaline adresseerimisvariant „* ± nihe"; sisuliselt on see *PIC*-programmeerimise kaudne analoog: „nihe" on muutuja. *PIC*i puhul muutub absoluutaadress.

```
FACT: MOV  N,-(SP)          ;INITIAL VALUE ON STACK
      JSR  PC,FAC           ;CALCULATE N!
      MOV  (SP)+,R0    ;GET ANSWER
      RTS  PC

FAC:  MOV  2(SP),R1     ;R1←N („2" INDEXES AROUND RETURN ADDRESS
ON                            ;STACK)
      BEQ  ZERO         ;N=0?
      DEC  R1           ;NO.R1←N-1
      MOV  R1,-(SP)     ;PUSH R1=N-1 ONTO STACK
      JSR  PC,FAC           ;RECURSIVE CALL
RET:  MOV  (SP)+,R1     ;POP (N-1)! INTO R1
      MUL  2(SP),R1     ;R1←(N-1)×N
      MOV  R1,2(SP)     ;N!=(N-1)×N REPLACES N
      RTS  PC
ZERO: MOV  #1,2(SP)     ;0!=1
      RTS  PC
```

Tuletagem meelde, et *kaasprogrammide* vahel pole tavalist alamprogrammide ülemuse-alluva suhteid, vaid juhtimise üleandmine käib „võrdsetel alustel". Seejuures toimib järgmine mehhanism: kaasprogrammid *A* ja *B* (piirdugem lihtsuse huvides kahega) on mõlemad seotud ühe ja sama protsessi menetlemisega. Oletagem, et alustab *A*, mis töötab momendini, kus tuleb juhtimine üle anda moodulile *B*. See alustab algusest, ja viib protsessi edasi kuni oma võimaluste ammendumiseni; seejärel tagastab ta juhtimise *A*-le (siiani käis kõik just nii nagu põhi- ja alamprogrammi puhulgi). *A* teeb jälle oma tööd ning mingil ajal tuleb kasutada taas *B* teeneid, ent sedapuhku *B* jatkab oma tööd punktis, kus see eelmisel korral pooleli jäi — *B* käitub nii, nagu oleks ta pöördunud *oma alamprogrammi A* poole. Selline võrdväärne partnerlus toimib kuni protsessi lõpuni. *PDP* arhitektuur soosib selliselt seotud moodulite kirjutamist. Laename asjakohase näite *Eckhouse'*ilt ja *Morris*elt (vt [15], lk. 119..120).

Koostöö alustamiseks tuleb kirjutada magasini *B* sisendpunkti aadress (#PC2). Allpool on tähistatud moodulit *A* kui programm #1 käsuloendajaga PC1 ning *B* — #2 ja PC2.

Töötab #1, seal täidetakse (mingil ajal) käsk
JSR PC,@(R6)+
ja selle toimel tehakse järgmised tegevused:
1. PC2 võetakse magasinist; magasini viita suurendatakse (so, magasin jääb „lühemaks");
2. SP vähendatakse ja PC vana väärtus (so. PC1) kirjutatakse magasini
3. Juhtimine antakse aadressile PC2, so. programmile #2.

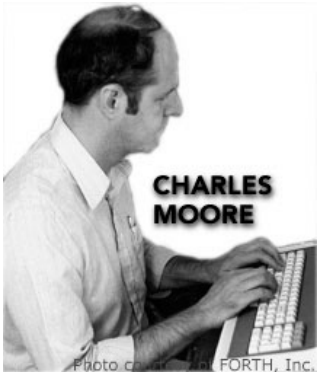
Töötab #2, ja seal täidetav käsk
JSR PC,@(R6)+
vahetab omakorda magasinis aadressid PC2 ja PC1 ning juhtimine antakse programmile #1.

4.2. *FORTH*

4.2.1. Sissejuhatus

Niisiis, neljandas peatükis käsitleme põgusalt kaht süsteemprogrammeerimise keelt, ja alustame *Charles „Chuck" Moore*'i *FORTH*ist, mis on sisuliselt rohkem kui pelgalt programmeerimiskeel.

Chuck Moore sündis Pennsylvanias (USA), õppis ülikooli(de)s *Lisp*i (*John McCarthy*lt), *FORTRAN II, ALGOL*i (kirjutas mh. translaatori *FORTRAN*ist *ALGOL*i) ja *COBOL*it. Oma unikaalse keele *FORTH* disainis ta 1968. a., arvutile *IBM 1130* Stanfordis. Esimene täisversioon leidis rakenduse 1971. a., arvutil *PDP-11*[1], Arizonas asunud 11-meetrise raadioteleskoobi juhtimiseks (objekti fikseerimine ja saatmine ning vaatlusandmete kuvamine ning salvestamine). Süsteem leidis kiiresti tunnustust esmalt astronoomide, ja varsti ka muude valdkondade rakendusteadlaste poolt; 1976. a. tunnistati *FORTH* Rahvusvahelise Astronoomialiidu (*International Astronomical Union*) standard-programmeerimiskeeleks [30].



Charles „Chuck" Moore (s. 1938), *FORTH*i autor.

Järgnevates lõikudes usaldame *Juri Semjonov*i [48] ja *P. Knaggs*i [30] andmeid. Kuivõrd *Moore* pidas oma keelt „neljanda põlvkonna"[2], teisisõnu „tuleviku"-keeleks ─ siis andis ta sellele nimeks *FOURTH, „a Fourth GL*". Et aga objektmasin ei võimaldanud pikemaid nimesid kui 5 tähte, siis ohverdati „*U*". Algusest saadik oli *FORTH*i konkurent *C* Ent *FORTH*i nišš on keeruliste objektide juhtimine reaalajas. Mõned näited:

- Kosmosesüstiku *Shuttle* tarkvara.
- Luuresatelliitide (näiteks 1802 ─ *Avco Inc*) tarkvara.

[1] *PDP* arhitektuur sobis *FORTH*i jaoks oluliselt paremini kui *IBM*i oma.

[2] Noil aegadel oli üldse kombeks erisatada „põlvkondi", nii keelte kui ka arvutite omi. Umbes selle ajani asi „töötas", hiljem vististi enam mitte. *Arvutipõlvkondi* eristati füüsilise lahenduse järgi. „Fossiilsed" olid elektromehhaanilised (eeskätt releedel baseeruvad), edasi elektroonilised: 1. põlvkond ─ raadiolambid, teine ─ transistorid, kolmas ─ suured integraalskeemid. Mikrokiipide tulekuga too klassifikatsioon hääbus. Keelte asjus võiksime tõdeda, et kuni 3. põlvkonnani (*3GL*) ─ protseduurorienteeritud keelteni (mõistagi, kaasa arvatud) oli asi lihtne, ent edasi juba vaieldav: probleemorienteeritud keeled pole lihtsalt võrreldavad protseduursetega ning et „järgmine põlvkond" peaks intuitiivselt olema eelmisega vähemalt sama võimsuse ja ligikaudu samade võimalustega, siis ka see liigitelu pole enam aktuaalne.

- Multifilmid „*Star Wars*", „*Battle Beyond*", „*Stars*" ja „*Star Track*" programmeeriti *FORTH*is.
- Er-Riadi lennujaama juhtimissüsteem (süsteemis oli 400 arvutit ja 36 000 andurit).
- Ekspertsüsteemide, robotnägemise, automaatsete meditsiinilise kontrolli süsteemide, masintõlke, arvutigraafika jpt. programmeerimiseks.

Kindlasti ei sobi *FORTH* rakendusprogrammeerimise keeleks ─ selleks on keel liiga erinev *FORTRAN*i ja *ALGOL*i mallidest ning eeldab „normaalselt programmeerijalt" harjumatult põhjalikke teadmisi masinatasemest. Ja võrreldes teiste kõrgtaseme keeltega on programmeerimine *FORTH*is rohkem tähelepanu ja aega nõudev. Ent, süsteemprogrammeerimiseks sobib *FORTH* igati.

FORTH on pisut rohkem kui programmeerimiskeel, pigem on ta *süsteem*: ta sisaldab vahendeid tööks välisseadmetega, failidega, katkestuste töötlemiseks, oma redaktorit jne. Ja veel, *FORTH* suudab töötada ilma operatsioonisüsteemi toeta (mõtlemise koht: *DOS*i katkestused on pelgalt protsessori genereeritud signaalid, ja see, et neile reageeritakse *DOS*i *int*-moodulite abil, on ainult üks, ehkki mugavaim võimalus). Liiati saab süsteemi kasutada nii interpreteerivas režiimis kui ka (eeskätt) kompileerimaks käsufaile (so., kasutada teda kompileerivas režiimis).

FORTH-süsteemi *tuum* (primitiivide kogum) on konkurentsitult väikseim; 16-bitise protsessori puhul vaid 5..8 *kilo*baiti (võrdluseks, *Pascal* vajab 48 KB, ([48], lk. 4). Ja veel, *FORTH*-programm ei vaja paigaldamist, kõik saab paika kompileerimisel. Lisaks, süsteemi kuulub oma *assembler*. *J. Semjonov*i andmetel kaotab *FORTH* seda kasutades pärisassemblerile kiiruses mitte rohkem kui paarkümmend protsenti ([48], lk. 4).

FORTH on avatud süsteem: iga kasutaja saab seda lihtsalt laiendada (täiendada) vastavalt vajadustele. See on, tuumaprimitiivide hulk pole lõplik ja selle laiendamiseks on olemas kõik võimalused süsteemi enda vahenditega.

FORTH-programmeerijaid on suhteliselt vähe, ent nad moodustavad hästi organiseerunud ühtse vennaskonna[1]. Paljudes riikides tegutsevad *FORTH*-programmeerijate organisatsioonid[2], regulaarselt korraldatakse selle sekti rahvusvahelisi konverentse.

Eestisse jõudis *FORTH* 1980-ndate alguses. *Leo Võhandu* „leidis" selle üles, tekitas *Mati Räbovõitra*s huvi, ja koos majandusküberneetikatudengist praktikandi *Reino Väinaste*ga tegid nad *BlackHill Forth*i (Mustamäe, eks ole), see võis olla 1981. aastal. Tartus algas *FORTH*iga tegelemine 1982. aastal, nimelt oli *Apple*'iga kaasas *fig-FORTH* ning *Mati*

[1] Üks esimesi näiteks *USA FIG ─ F[ORTH] Interest Group*.
[2] Aprillis 1991, esimese üleliidulise *FORTH*i konverentsi käigus Leningradis, asutati N. Liidu *FORTH*i assotsiatsioon. Meie ülikooli esindajatele tehti loomulikult ettepanek hakata asutajaliikmeteks, ent neiks me siiski ei saanud, asja lahendas *Jaanus Pöial*i üllatunud küsimus, et kas NL *FORTH*i-assotsiatsioon näeb tõepoolest ette ka välisliikmete olemasolu. Noiks saanuks me sama aasta sügisel.
Tänane Venemaa organisatsioon on nii arvukas kui ka teovõimeline (vt. näit. [71])

Tombak, *Jaanus Pöial* ja *Viljo Soo* kirjutasid selles *TTS*i[1]. Miniarvutile *CM-4* kandsid *BlackHill FORTHi* üle *M. Räbovõitra, R. Väinaste* ja *Aivar Juurik* 1984. aastal, pisut hiljem kirjutasid süsteemi *Forth-83/32* tuuma *R. Väinaste* ja *A. Juurik* ning *M. Tombak*, *J. Pöial* ja *V. Soo* modifitseerisid ülejäänud süsteemsed komponendid. Tegemist oli kättesaadava, 16-bitise *FORTH*i *Forth-83* tuunimisega 32-bitiseks. Tolles uues *FORTH*is kirjutati translaator *Modula-2 → FORTH* (*M. Tombak, J. Pöial, V. Soo, T. Saarsen, R. Väinaste, A. Juurik*, aga ka nende ridade kirjutaja). Ja Eesti *FORTH*i-ajalugu poleks aus, kui ei mainiks tihedaid sidemeid N. Liidu, ja eeskätt Leningradi *FORTH*-i entusiastidega.

4.2.2. *FORTH*-süsteemist

Ülalpool mainisime, et *FORTH* on *süsteem* ja mitte pelgalt programmeerimiskeel. Too süsteem koosneb oma tekstitoimetist (*Editor*), mis võimaldab luua, modifitseerida ja trükkida süsteemseid tekstifaile (nimelaiendiga *.scr = screen*), interpretaatorist ja kompilaatorist. Ja et ─ kui ta nii realiseerida ─ suudab ta asendada operatsioonisüsteemi, so. reageerida oma vahenditega katkestustele (sh. evida omi vahendeid tööks välisseadmetega). Et *FORTH*i tunnetada, tuleks käsitleda kõiki süsteemseid komponente, lisaks nonde realisatsiooni, mõistagi koos vastavate andmestruktuuridega. Ja ehkki kogu süsteemi maht on üllatavalt väike, ei saa me kahjuks nende kaante vahel sellist ülevaadet anda, sestap vaadelgem allpool ainult mõningaid aspekte, mis on seotud *FORTH*i kui eeskätt programmeerimisvahendiga.

Süsteem kasutab kahte *LIFO*-tüüpi magasini: need on *naasmismagasin* (*RS = Return Stack*) „alamprogrammide" naasmisaadresside jaoks (ja kuhu võib „kasutajaprogramm" omal riisikol salvestada lokaalsete muutujate väärtusi) ning *andmemagasin* (*DS = Data Stack*) nii sisend- kui ka väljundparameetrite ja (vajadusel) lokaalsete muutujate jaoks. Kui meenutada *PDP*-d, siis naasmismagasin on selle masina süsteemne magasin viidaga *SP* ning andmemagasin on *FORTH*-süsteemi „eramagasin".

„Ilmutatud" viitade keskkond on minimaalne: kasutaja saab defineerida ainult globaalseid muutujaid, neist mõned on defineeritud *vaikimisi* (näiteks, tsükliindeksid *I* (kõige sisemine või ainus) ja *J* (tase kõrgemal)). Nime omavaid lokaalseid muutujaid *ei ole* (nende väärtusi saab anonüümselt hoida andmemagasinis).

Mõiste *(alam)programm* on üle-eelmises lõigus jutumärkides; asi on selles, et toda mõistet *FORTH*is ei ole. Kõik sellised „programmid" on kas süsteemi loomisel defineeritud „sõnad" või kasutaja kirjeldatud, tema jaoks samaväärsed „sõnad". Infovahetus nonde sõnade vahel käib andmemagasini vahendusel. Kasutaja defineerib oma sõnu nn. *koolon-definitsiooniga*; tavaliselt algab see *spetsifikatsiooniga* (sulgudes kommentaar, kus on antud andmemagasini seis enne ja pärast sõna täitmist, nende vahel on eraldaja --- . Defineerime näiteks sõna, mis saab ette kaks aadressi, liidab neil olevad arvud kokku ja kirjutab resultaadi argumentide asemele magasini. Olgu selleks sõnaks ab+.

[1] *TTS* ─ Translaatorite Tegemise Süsteem (i.k. *Compiler Compiler,* v.k. *Система Построения Трансляторов*) on süntaksorienteeritud, kontekstivabadele grammatikatele tuginevate programmeerimiskeelte translaatorite kirjutamist oluliselt hõlbustav tehnoloogia.

(aadr1 aadr2 --- n)
: ab+ @ SWAP @ + ;

Vaatame, mis selle sõna „täitmisel" (vasakult paremale, algseis *DS*-is on spetsifikatsioonis) toimub:
@: (aadr2) → DS (*Data Stack*, so vahetati tipmise elemendi väärtus *aadress→arv*);
SWAP (vahetab magasini 2 tipmist elementi): magasinis on nüüd (aadr2) aadr1;
@: (aadr1) → DS (*Data Stack*), seal on nüüd (aadr2) (aadr1);
+: (aadr2)+(aadr1)→ DS; seal on nüüd tipmiseks elemendiks kahe arvu summa.

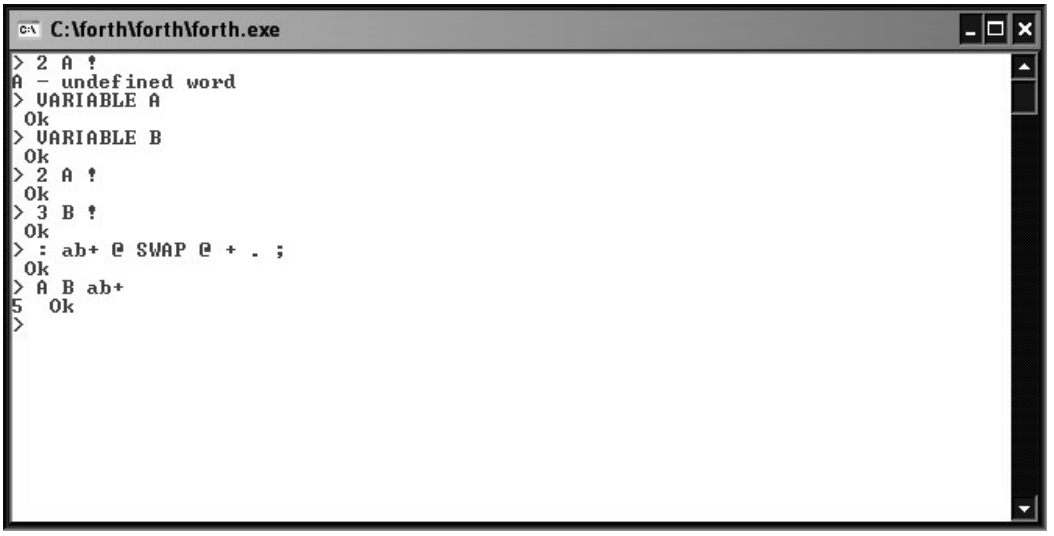
„Tuumaprogramm" *SWAP* võib olla *FORTH*-assembleris (*Intel*-masina jaoks) kirjutatud nii (vt. [48], lk. 223):

```
HEAD   204Q,'SWA',320Q,SWAP
POP    CX
POP    AX
PUSH   CX
PUSH   AX
NEXT
```

Kui toda sõna käivitada interpreteerivas režiimis, siis võib dialoog välja näha nii:

5 a ! 6 b ! a b ab+ .
11 ok

Mõistagi, nii a kui ka b peavad olema süsteemis kirjeldatud eelnevalt süsteemsete muutujatena. Sõna „." väljastab magasini tipmise elemendi ekraanile. Ja *a 5 !* tähendab Algoli-stiilis omistamist a:=5. „OK" tähendab, et süsteem arvab, et kõik nii tema poolt ongi. Joonisel 4.2.2a on dialoog *FORTH*-interpretaatoriga tolle eesmärgi saavutamiseks.



Joonis 4.2.2a. Dialoog interpreteerivas režiimis.

4.2.3. „*Wilson*i näide”

Allpool defineerime sõnad *SUM,* mis leiab etteantud vektori *A* (pikkusega *n*) elementide väärtuste summa ja *WILSON*, mis väljastab vektori elementide keskväärtuse täisosa *MEAN* ning nende elementide arvu, millede väärtus ületab keskmise täisosa (*#BIGGER*)[1].

```
0 VARIABLE  MEAN
0 VARIABLE  #BIGGER

(SUM  0 A n --- s )
: SUM 0 DO DUP I @ + @ ROT + SWAP
        LOOP
      DROP ;

(WILSON A n ---   )
: WILSON 2DUP 0 -ROT SUM OVER / MEAN !
0 #BIGGER !
0 DO
      DUP I @ + @ MEAN @ >
      IF #BIGGER @ 1+ #BIGGER ! THEN
  LOOP
DROP
." MEAN = " MEAN @ .
." NUMBER OVER MEAN = " #BIGGER @ .
;
```

Kirjutame need koolondefinitsioonid „lahti”, näidates, milline on andmemagasini seis enne ja pärast järjekordse sõna „täitmist”.

SUM:

```
      0 A n       (0 on summa „s” algväärtus)
0     s A n 0
DO    s A         (n 0 → naasmismagasini, I on tipmise elemendi
                  aadress, n="limit" ja 0="indeks"; viimast
                  suurendab „tsüklikäsk” LOOP)
DUP   s A A
I     s A A I
@     s A A i     (i=(I))
+     s A A+i
@     s A a       (a=(A+i))
ROT   A a s
+     A a+s
SWAP  a+s A
LOOP  a+s A       (tsükli lõppedes eemaldatakse naasmismagasi-
                  nist tsüklimuutujad „limit” ja „indeks”)
DROP  a+s
;                 (juhtimine antakse naasmismagasini järgi üle)
```

[1] Lisaks oletagem, et meil on defineeritud muutuja *VARIABLE A* ning sõnad *READ1(--- n) ja READVECTOR (A n ---);* esimene loeb (suvalisest) sisendist sümbolkujul täisarvu (see moodul võiks olla realiseeritud meile tuttava *readint*i abil, ja teine loeb sümbolkuju-arvude vektori ning teisendab need *int*-kujule; näiteks kasutades sõna *READ1* tsüklis. Sel juhul võiks meie programmi eelmäng olla näiteks selline:
A READ1 DUP >R 2* ALLOT A R> READVECTOR — sisestame vektori *A* pikkuse, reserveerime talle mälu ning sisestame *A* elemendid. Vektori pikkuse „hoiame meeles” ajutiselt naasmismagasinis.

WILSON:

```
            A n
2DUP        A n A n
0           A n A n 0
-ROT        A n 0 A n
SUM         A n s       (järgmise sõna OVER koodivälja aadress →
            naasmismagasini)
OVER        A n s n
/           A n täisosa(s/n)
MEAN        A n täisosa(s/n) MEAN
!           A n
0           A n 0
#BIGGER     A n 0 #BIGGER
!           A n
0           A n 0
DO          A
DUP         A A
I           A A I
@           A A i
+           A A+i
@           A a
MEAN        A a MEAN
@           A a k       (k=(MEAN))
>           A tf        (tf=TRUE (=1)/FALSE (=0))
IF          A
#BIGGER     A #BIGGER   (kui tf=1, siis täidetakse programmi
                        kuni sõnani ELSE (või selle puudumisel)
                        sõnani THEN)
@           A arv
1+          A arv+1
#BIGGER     A arv+1 #BIGGER
!           A
THEN        A
LOOP        A
DROP
." MEAN = " MEAN @ .
." NUMBER OVER MEAN = " #BIGGER @ .
;
```

4.2.4. Tuumaprimitiivid

Selles jaotises esitame valikuliselt lihtsamate sõnade spetsifikatsioone, rühmitatult otstarbe järgi. Tehkem üks (meie raamatu kontekstis ebaoluline) mööndus: allpool-esitatav ei pruugi olla realiseeritud täielikult mingis konkreetse *FORTH*i süsteemis; me jätame endale vabaduse „miksida" erinevaid variante (*FORTH 83/32, figFORTH* jt). Õigustagem end tõigaga, et nende kaante vahel *ei ole* programmeerimisõpik.

Allpool on järgitud *FORTH*i materjalides üldtunnustatud tähistusi, ja nimelt (tähestiku järjekorras):

addr mäluaadress
b bait (sõna „vanemates järkudes" nullid)
c ASCII-sümbol
d 64-bitine märgiga täisarv

f tõeväärtus: väär=0, tõene=1
n 32-bitine märgiga täisarv
u 32-bitine märgita täisarv
ud 64-bitine märgita täisarv

Magasinioperatsioonid.

DUP (n --- n n) *(1 --- 1 1)*
?DUP (n --- n *või* n n) *kui n=0, siis teda ei dubleerita*
2DUP (d --- d d) *(1 2 --- 1 2 1 2)*
DROP (n ---) *(1 ---)*
2DROP (d ---) *(1 2 ---)*
OVER (n1 n2 ---n1 n2 n1) *(1 2 --- 1 2 1)*
ROT (n1 n2 n3 --- n2 n3 n1) *(1 2 3 --- 2 3 1)*
-ROT (n1 n2 n3 --- n3 n1 n2) *(1 2 3 --- 3 1 2)*
SWAP (n1 n2 --- n2 n1) *(1 2 --- 2 1)*
PICK (n1 --- n2) *kopeerib magasini n1-nda elemendi tippu; 0 PICK = DUP, 1 PICK = OVER jne. Nt, kui algseis on (3 4 5 6), siis pärast 3 PICK on magasinis (3 4 5 6 3)*
ROLL (n ---) *magasini tippu tõstetakse magasinist kohal n paiknev element (loendamine algab nullist), n ise arvesse ei lähe. 0 ROLL ei tee midagi, 1 ROLL = SWAP, 2 ROLL = ROT jne. Nt, algseisust (3 4 5 6) käsu 3 ROLL toimel on magasinis (3 5 6 4).*
>R (n ---) *n kantakse naasmismagasini; kasutaja :-definitsioon peab kindlasti enne lõpetamist täitma sõna R>.*
R> (--- n) *n loetakse ja eemaldatakse naasmismagasinist*
R@ (--- n) *n koopia naasmismagasinist kantakse andmemagasini*[1]

Tüübid.

Sõnadega BINARY, DECIMAL, HEX ja OCTAL (ükski neist ei kasuta magasini) määratakse (järgmise tüübimääramiseni) sisend- ja väljundoperatsioonides osalevate arvude esitus (nood määravad süsteemse muutuja *BASE* väärtuse). Näeme, et baastüüpide hulgas pole ujupunktarvudele vihjavaid (nt. REAL, FLOAT vmt.) tüüpe, ent paljud *FORTH*i realisatsioonid pakuvad asjakohast „sõnastikku" (primitiivide hulka). Põhjus on ehk selles, et erinevalt tolle aja keelte suunitluselt oli *FORTH*-süsteem algusest peale orienteeritud *süsteemse tarkvara* (sh. operatsioonisüsteemid) kirjutamisele.

Aritmeetika ja loogika.

+ (n1 n2 --- n1+n2) *(3 5 --- 8)*
/ (n1 n2 --- jagatise-täisosa) *(33 7 --- 4)*
/MOD (n1 n2 --- jääk täisosa) *(33 7 --- 5 4)*
1+ (n1 --- n2) *(7 --- 8)*

[1] Kui naasmismagasini midagi kirjutada tsükli kehas, siis tuleb kirjutatu eemaldada enne tsükli lõpusõna *LOOP* või *+LOOP*. Põhjus on ilmne: naasmismagasini tipus on tsükli ajal tsüklimuutujate väärtused.

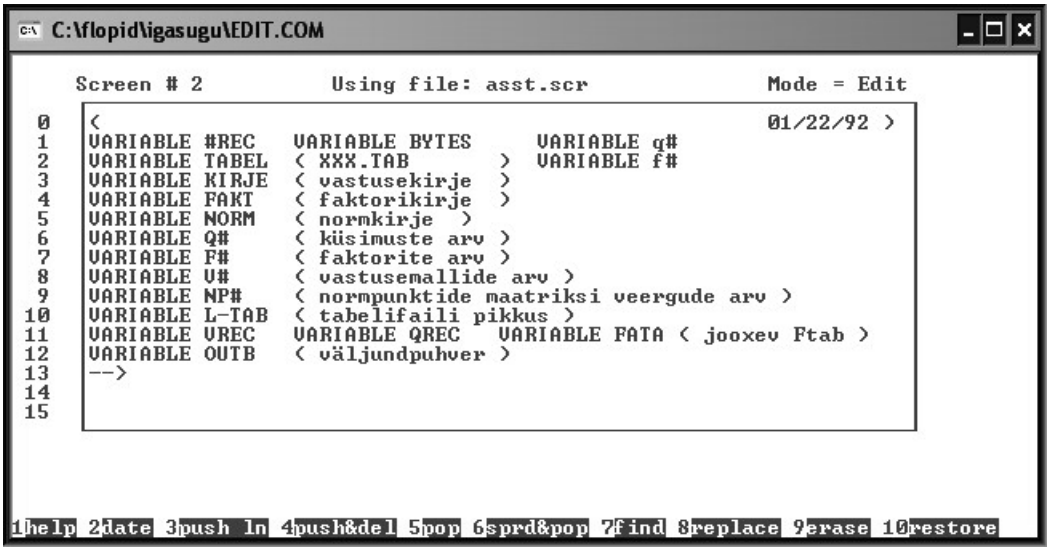
2* (n1 --- n2) *n2=n1*2: (14 --- 28)*
2/ (n1 --- n2) *n2=n1/2: (14 --- 7)*
>< (n1 --- n2) *n1=ab, n2=ba (baitide vahetus sõnas)*
AND (u1 u2 --- and) *u1&u2 → DS (0110 1100 --- 0100)*
NEGATE (n --- „-n") *märgimuutmine (444 --- -444)*
XOR (u1 u2 --- xor) *(0110 1100 --- 1010)*
S>D (n --- d) 32-bitine arv → 64-bitine

Võrdlemine.

0< (n --- f) *kui n<0, siis TRUE*
0<> (n --- f)
0= (n --- f)
0> (n --- f)
= (n1 n2 --- f) *kui n1=n2, siis TRUE*
< (n1 n2 --- f)
<> (n1 n2 --- f) *kui n1≠n2, siis TRUE*
D> (d1 d2 --- f) *topeltarvud*
U< (u1 u2 --- f) *märgita arvud*

Konstandid ja muutujad.

Nonde objektide vahe seisneb selles, et objekti „B" kasutamisel kirjutatakse andmemagasini *B aadress*, kui ta on muutuja, ja *väärtus*, kui ta on konstant. Ja veel, muutuja võib kirjeldada ilma algväärtustamata, aga konstant lihtsalt *tuleb* defineerida. Konstant evib alati ühte väärtust; muutuja võib seevastu olla kas lihtmuutuja või vektor (vt. joonis 4.2.4a).



Joonis 4.2.4a. Muutujad.

Näiteks, 7 CONSTANT PIUX kirjeldab konstanti PIUX (ASCII 07), mille saatmine väljundisse annab lühikese vile.

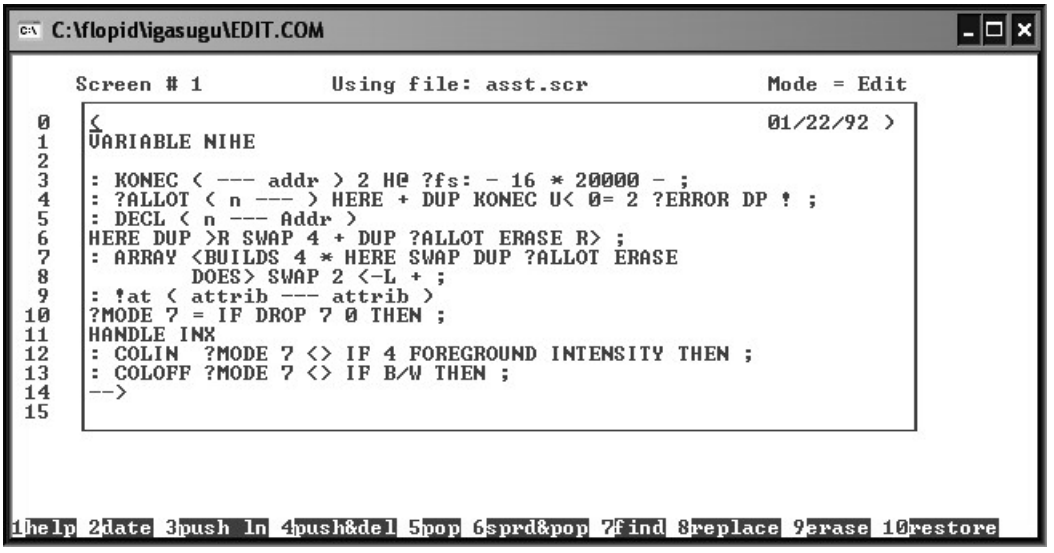
Muutujat VAR võime defineerida kas algväärtustamisega või ilma:
77 VARIABLE VAR
VARIABLE VAR
Vektorit saab (vist) kõigis *FORTH*-realisatsioonides defineerida *staatiliselt* näit. nii:

0 VARIABLE B 128 ALLOT — *muutujaga B seotakse 128-baidine vektor, mille elementide algväärtuseks on 0.*

Sõna ALLOT dünaamilist varianti pole näiteks *FORTH 83/32*-s; seal tuleb see ise koolondefinitsiooniga defineerida (vt joon. 4.2.4b). J. Semjonov [48, lk. 24] toob järgmise „dünaamilise *ALLOT*i" näite:

```
0 VARIABLE LL
…16*16 LL !
…0 VARIABLE BB LL @ ALLOT   — BB on nüüd 256-baidise vektori aadress.
```

Mainigem, et koolondefinitsioonis kasutatavaid arve (0, -1, 47 jne) tõlgendatakse samuti nagu assembler *literaale,* so, kui antud väärtusega konstanti sõnastikus juba varem polnud, siis see lisatakse sinna.



Joonis 4.2.4b**.** Dünaamilise mälujaotuse toe fragment.

Mälu.

@ (addr --- n) *loeb addr-lt magasini*
! (n addr ---) *kirjutab n→addr*
2@ (addr --- d)) *loeb addr-lt magasini topeltsõna*
C! (b addr ---) *kirjutab baidi väärtuse mällu*
2! (d addr ---) *kirjutab topeltsõna mällu*
+! (n addr ---) *(kui n=10 ja (addr)=444, siis käsu toimel (Addr)=454*[1]

Ketas

--> (---) *jatka järgmiselt .scr-ekraanilt*
;S (---) *.scr-faili --> abil seotud ekraanide viimase ekraani lõputunnus*
BLOCK (u --- addr*) kettablokiga u seotakse puhvri aadress (kui puhvrit veel pole, siis selleks vajaliku mälu saab sõna BUFFER (u --- addr) abil) ning sinna loetakse uus ketta-plokk. Tegelik mehhanism on komplitseeritum kui siinkirjeldatu. Eriolukordade jaoks on sõnad R/W, UPDATE, FLUSH, EMPTY-BUFFERS, SAVE -BUFFERS* jt.
EMPTY-BUFFERS (---) *vabasta kõik kettapuhvrid*
FLUSH (---) *kirjuta välja ja vabasta kõik kettapuhvrid*
LOAD (n ---) *laadi/käivita aktiivse .scr-faili screen n (sealt algavad (ja võivad jatkuda --> toimel) koolon-definitsioonid)*

OS-liides.

!DATE (n1 n2 --- *) sea süsteemne kuupäev (n1=aasta, n2=kuu ja päev)*
@DATE (--- n1 n2) *loe süsteemne kuupäev (n1=aasta, n2=kuu ja päev)*
@COM1 (--- c*) loe sümbol jadapordist*
?TERMINAL (--- f) *tõene, kui klaviatuuril on mingit klahvi vajutatud*
BYE (---) *väljumine FORTH-süsteemist*.
CHDIR [path] (---) *jooksva tee (teegini) muutmine*
PRINTER (---) *vali printer väljundseadmeks*
CONSOLE (---) *vali displei väljundseadmeks*
RUN (---) *Kasutatakse kujul RUN seade:kataloog\failinimi.suf, „suf" on kas com või exe ja käivitatakse süsteemiväline programm, sealt naastakse FORTHi.*
FDOS (funktsioon parameeter --- AX-kood BX-kood) *kasutatakse DOSi katkestust int 21. „Funktsioon" → AH, „parameeter" → DX (vt. Inteli katkestused)*[2].
ABORT (---) *Tühjendatakse andmemagasin ja antakse juhtimine sõnale QUIT*
QUIT (---) *Naasmismagasin tühjendatakse, süsteem viiakse täitmisolekusse ja alustatakse teksti interpreteerimist aktuaalselt seadmelt*

---

[1] *Juri Semjonov* ([48], lk. 22) „kirjutab lahti" tolle definitsiooni nii:
: +! DUP @ ROT + SWAP ! ;

[2] Juhime lugeja tähelepanu seigale, et *FDOS* (=*Function of DOS*) võimaldab kasutajal ilmutatud kujul suhelda (*Inteli* keskkonnas) MS DOSi funktsioonidega. Varjatud kujul kasutavad *DOS*i ja *BIOS*i vahendeid paljud *FORTH*i tuumaprimitiivid ja see variant on kättesaadav ka „lihtkasutajale", kui ta oma koolondefinitsioonid kirjutab (*FORTH*-)assembleris.

Sisend ja väljund.

Mitmed spetsiifilised formaadid (#, #>, #S) on mõeldud loetava, sümbolesituses arvu teisendamiseks digitaalkujule (arvestades süsteemse muutuja *BASE* jooksva väärtusega).

PCKEY (--- b või --- b 0) *funktsionaalse klahvile vastab kahebaidine kood 0b*
?TERM (--- b) *Kui sisendpuhvris on sümbol, siis kantakse see andmemagasini, kui aga pole,siis ei jääda ootama*[1].
KEY (--- b) *konsool → sümbol*
EXPECT (aadr pikkus ---) *terminalilt loetakse mällu string alates antud aadressist ja sisestamise lõpetab kas Enter-kood või „pikkuse" saavutamine. Lõputunnuseks pannakse 0*[2].
ASCII (--- c) *kasutatakse kujul ASCII c. Sümbol kirjutatakse magasini.*
CR (---) *väljastatakse koodid „kelk tagasi" ja „reavahetus"*
. (n ---) *trüki märgiga arv*
U**.** (u **---)** *trüki märgita arv*
." (---) *trüki string kuni sümbolini „jutumärgid", näiteks ." siin see on"*
. STACK *trüki andmemagasinimagasini sisu*
.FCB (addr ---) *trüki faili nimi FCB-st (=File Control Block)*
TYPE (addr n ---) *ekraanile väljastatakse aadressilt addr n-baidine string*
SPACES (n ---) *väljastatakse n tühikut*
EMIT (b ---) *aktuaalsele väljundseadmele väljastatakse sümbol koodiga b, süsteemse muutuja OUT väärtust suurendatakse ühe võrra. Aktuaalne seade määratakse sõnadega PRINTER, CONSOLE või COM1.*

Siia kuuluvad ka sõnad **.**CPU, **.**DATE, .TIME, .VERSION (kõik nad on parameetriteta) ja väljastavad vastavalt protsessori identifikaatori, süsteemse kuupäeva ja kellaaja ning *FORTH*i versiooni numbri.

String.

BLANK (aadr n ---) *alates aadressist addr kirjutatakse n tühikut*
CMOVE (aadr1 aadr2 len ---) *n baiti aadr1 → aadr2*
CMOVE> (aadr1 aadr2 len ---) *n baiti aadr1 → aadr2 pöördjärjestuses*
CONVERT (d1 aadr1 --- d2 aadr2) aadr1, d1 on tekst, see teisendatakse aadr2-le (pikkusega d2) int-kujule
S= (aadr1 aadr2 len --- f) *n-baidiste stringide võrdlus. Tõene, kui samad.*

Juhtimine.

DO … LOOP (limit start ---) *tsükkel vähemalt üks kord (LOOP (---))*
?DO … LOOP (limit start ---) *tsükkel, kui limit <> indeks*

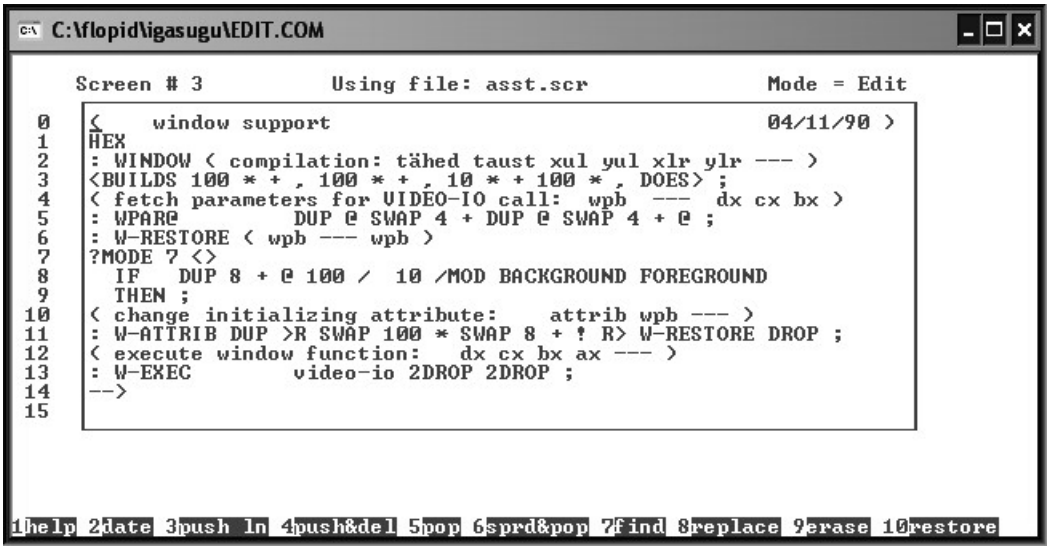
---

[1] *J. Semjonov* ([48], lk. 30) toob lõpmatust tsüklist väljumise näite (päästab klahvi *Z* vajutamine):
BEGIN XXX ?TERM ASCII Z = UNTIL
[2] J. Semjonovi ([48], lk. 31) näide: 0 VARIABLE HI 12 ALLOT
: GREAT HI 12 BLANK CR ." ENTER>" HI 12 EXPECT ;

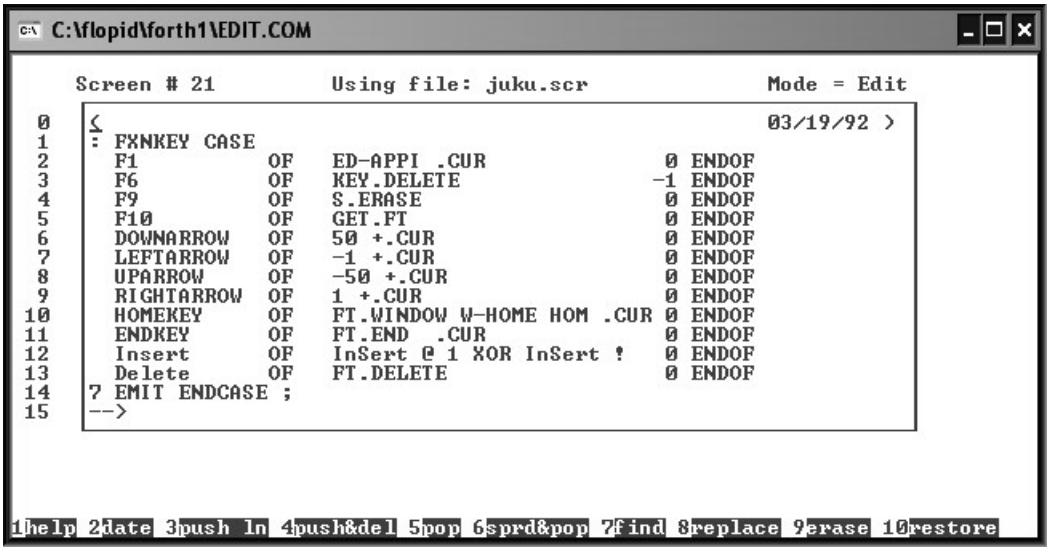
DO … +LOOP (limit start ---) *sammuga n (+LOOP (n ---))*
BEGIN … UNTIL *(UNTIL (f ---))*
BEGIN … AGAIN (---) *lõpmatu tsükkel, väljumiseks EXIT*
IF … [ELSE] … THEN (f ---) *tüene haru IF…ELSE või viimase puudumisel THEN; ELSE'le järgneb väär haru kuni THEN*
LEAVE (---) *katkestab DO─LOOP-tsükli*
EXIT (---) *lõpetab jooksva koolondefinitsiooni täitmise*
I (--- n) *sisemise (vüi ainsa) tsükli indeksi väärtus*
J (--- n) *välimise tsükli indeksi väärtus*
CASE (n ---) *alustab mitmeharulist valikukonstruktsiooni kujul n CASE … OF … ENDOF … ENDCASE. Paari OF … ENDOF võib korrata suvaline arv kordi. Parameeter n on selektor, mille abil toimub haru valik* (vt. joonis 4.2.4d).

Video
Selle rühma näited on toodud ainsal eesmärgil ─ näidata, et *FORTH* opereerib „masinatasemel" (vt.. joonis 4.2.4c). Niisiis:

B/W (---) *lülitab displei ümber; mustvalge*
BACKGROUND (n ---) *vali displei foonivärv*
FOREGROUND (n ---) *väli „teksti" värv*
GOTOXY (x y ---) *positsioneeri kursor*
SET-CURSOR (start end ---) *määrab kursori suuruse*



Joonis 4.2.4c. *Video*-sõnade näide



Joonis 4.2.4d**.** *Case*-operaator.

4.2.5. Realisatsioon

Üldistatult öeldes: *FORTH* realiseeritakse kui *tabel-orienteeritud süsteem*, kui peame silmas, kuidas *tabel* on defineeritud *algoritmide ja andmestruktuuride* kursuses: „tabel on *kirjetest* koosnev andmestruktuur, kus *kirje* koosneb loogilisel tasemel kahest väljast, võtmest ja sellega seotud infost". *FORTH*-süsteemis on *võti* „koolon-definitsiooniga" või primitiiviga määratud *nimi* ja „info" on tolle definitsiooniga määratud algoritm („sõnade" jada). Toda tabelit nimetatakse *FORTH*-süsteemis *sõnastikuks* ja ta kirje on struktureeritud vektor, mille konkreetne esitus sõltub eeskätt realisatsioonist, ent kus on 5 välja järjekorras „lipud, nimi, viit eelmisele lülile *lfa* (*last field address*), viit koodiväljale *cfa* (*code field address*) ja viit „parameetriväljale" *pfa* (*parameter field address*).

Wikipedias on toodud üks võimalikest kirjestruktuuridest (vt. [69], [76]), kasutades lugejaile loodetavasti üldarusaadavat *C*-keelset (või *Java*) kirjeldust (originaalitruuduse ettekäändel jätame ta tõlkimata)[1]:

```
struct forthword {
   byte _flag; /* 3bit flags + length of word's name. */
   char name[];  /* name's runtime length isn't known at compile time in
C */
   struct forthword *previous; /* backward ptr to previous word. */
   struct forthword *codeword; /* ptr to the code to execute this word.
*/
```

[1]Toodud kirjeldus on suuresti abstraktne ja ei pruugi kajastada täpselt ühtegi võimalikku realisatsiooni. Näiteks, kiiruse huvides võib *cfa* olla viit mitte kirjele, vaid tolle *cfa*le (või *pfa*), *pfa* võib olla viit väljale või väli ise (nagu tsiteeritud kirjelduses)

```
   byte parameterfield[]; /* unknown length of data, words, or opcodes.
*/
 };
```

Viit **previous* osutab eelmisena defineeritud sõna kirjele (süsteemne nimi on *lfa* ─ *last field address*).

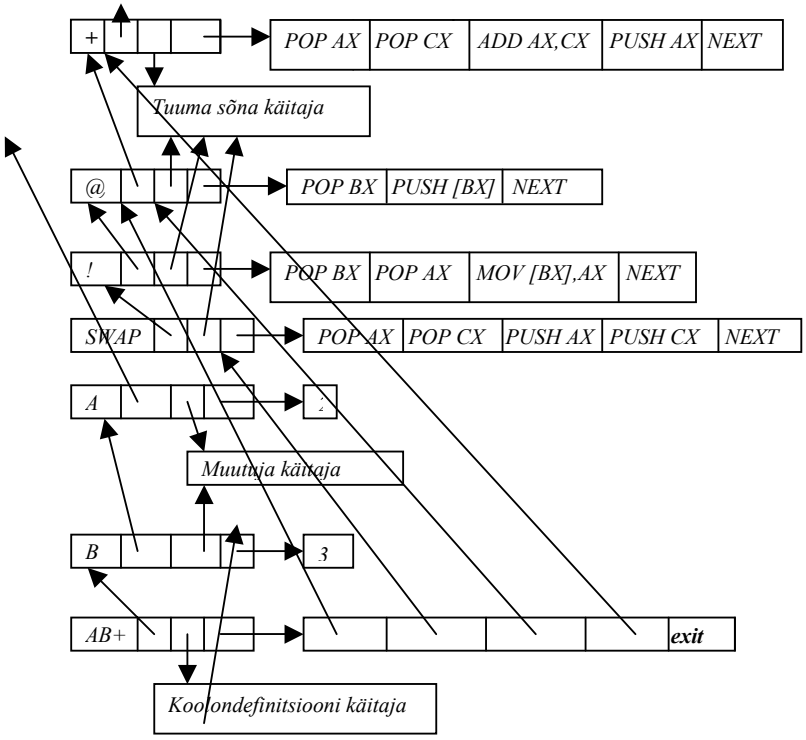
Kaks viimast struktuurielementi on *cfa* ja *pfa.* Neist esimene on viit sõnastiku kindlat tüüpi töötlevale sõnale, noid tüüpe on fikseeritud (väike) arv. Näiteks, üks ja ühine on kasutaja poolt defineeritud sõnade (*koolon-definitsioonide*) jaoks ─ sellega määratakse eeskätt naasmismagasiniga seonduv; tuumaprimitiivide jaoks üldse ja sh. sõnade CONSTANT ja VARIABLE jaoks eraldi jmt.

Kui primitiiv on CONSTANT-tüüpi, siis *pfa* (viitab või sellest algab) konstandi väärtus, kui aga tüüp on VARIABLE, siis seotakse *pfa*-ga muutuja algusaadress (ja pikkus). Nende puhul viitab *cfa* koodile, mis kirjutab konstandi väärtuse või muutuja aadressi andmemagasini ning see kood käivitatakse sõna täitmisel.

Ülejäänud tuumaprimitiivide *pfa* on seotud[1] primitiivi realiseeriva masinkoodiga Koolon-definitsiooni puhul on *pfa*-ga seotud viitade vektor tolle definitsiooniga määratud sõnadele (või nonde *cfa*-dele).

Joonisel 4.2.5a on kujutatud (võimalik) täitmisaegse sõnastiku fragment, mis illustreerib varemtoodud näite (sõna : ab+) esitamist. Tuumaprimitiivide kood (millele viitab *pfa*) on loetavuse huvides esitatud *FORTH*-assembleris (vt. [48], lk. 221 jj.), tegelikult on seal mõistagi masinkood. Muutujate *A* ja *B* väärtused on tol joonisel juba omistatud, vastavalt 2 ja 3. Seda, mida sõnastik esitab, nimetatakse *punutud koodiks* (*threaded code*).

[1] Viitab või on välja alguse suhtaadress



Joonis 4.2.5a . Sõnastiku fragment (…A B AB+ ..).

4.2.6. Kompilaator

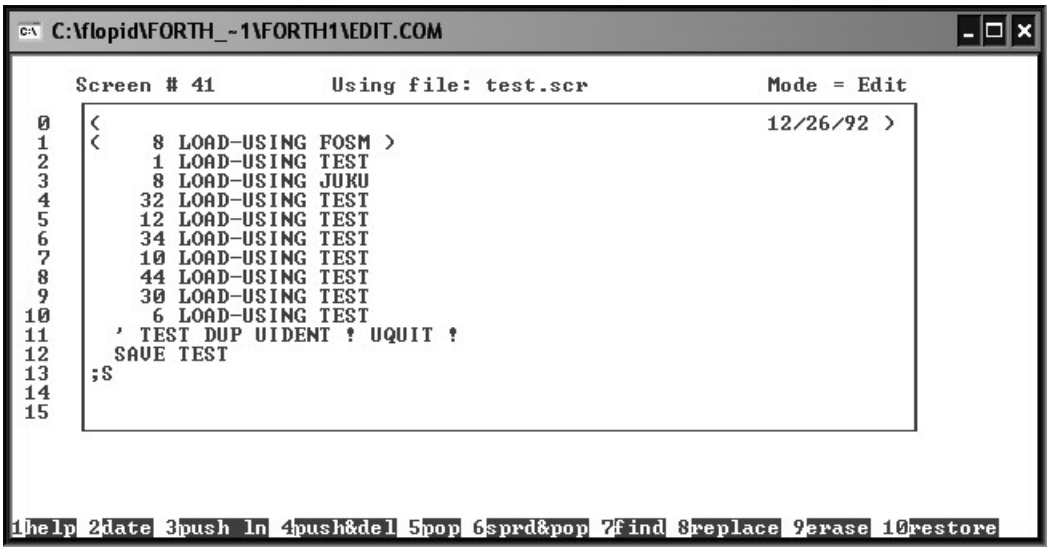
Kompilaatori „nähtav töö" seisneb tuumaprimitiivide sõnastikule etteantud *.scr*-failides defineeritud sõnade lisamises. Sõnastiku ehitamist tuuma ümber uue ülesande jaoks illustreerib joonis 4.2.6a, mille resultaat on spetsiaalse funktsiooniga[1] käivitatav fail *test.com.* [1] Ei toimu ei komplekteerimist ega ka paigaldamist (so. viitade korrigeerimist), kuivõrd märgendeid ja *goto*-operaatorit ei ole.

FORTH tunnistab ainult *globaalset* viitade keskkonda: kõik muutujad (nii liht- kui ka struktureeritud) on üldkasutatavas sõnastikus; sõnastikuobjekt (konstant, muutuja või *sõna*) on kasutatav alles pärast deklareerimist. Lokaalseid viitu pole. Viidatüüp on keele orgaaniline osa; *FORTH* on erandlik keel selle poolest, et suvalises avaldises on ühemõtteliselt näha, kas tegemist on operandi aadressi või väärtusega.

Keele esmasversioon toetas andmetüüpidest „normaalseid" ja topeltpikkusega (märgiga või ilma) täisarve, tõeväärtust ja stringi. Enamik hilisemaid realisatsioone toetab ka reaalarve (sh. ka topeltpikkusega).

[1] Spetsiaalne funktsioon on vajalik selleks, et "jooksutada" ka pikemaid kui 64K käsufaile (sellele kitsendusele viitab *.com*-laiend).

FORTH pakub „täiskomplekti" tsükli juhtmise versioonidest, sj aparatuurne tugi on kahetasemelisele tsüklile (välimise tsükli indeks on *J* ja sisemisel *I*) ning kasutatavad on nii *for*- kui ka *while*- versioon.



Joonis 4.2.6a. Eeskiri koodifaili *test.com* moodustamiseks.

4.2.7. *FORTH* kui programmeerimiskeel

*FORTH*i aritmeetilised ja loogilised avaldised pole tavapärased. Oleme harjunud, et nood avaldised kirjutatakse arvestades tehete prioriteetidega ning et tehete sooritamise järjekorda saab juhtida sulgudega. *FORTH*i avaldised ei võimalda sulge ega arvesta tehtemärkide prioriteetidega. Avaldised tuleb programmeerida *inverteeritud Poola kujul.*

Operaatorite *grupeerimise* võimalused *FORTH*is ei jää alla teistele keeltele. Üks taolisi võimalusi on see, mida pakub *tsükli keha,* ent seda võimaldab ka *hargnemine* (operaatorid *switch* ja *case*). Tingimuslik operaator pole hierarhiliselt piiratud ning ─ kui mitte pöörata liigset tähelepanu süntaksile ─ on täiesti ootuspärane.

Suunamisoperaaorit (GOTO märgend) *FORTH*is *pole.* Pole ei märgendeid ega ka avalikku suunamist. Sellega seoses võiks *FORTH*i pidada „süstemaatilise programmeerimise" paradigma teerajajaks, ent tundub siiski, et programmeerimismoodidel pole *FORTH*iga otsest seost.

Alamprogramm pole *FORTH*i termin. Meile harjumuspäraseks saanud ruumis on *FORTH*-süsteemis alamprogrammiks iga *sõnastikku lisatud sõna*. Infovahetus sõnade vahel käib üldjuhul andmemagasini abil ─ või siis globaalsete muutujate vahendusel. Parameetreid edastatakse nii *väärtuse* kui ka *viida* järgi (või abil). Ühisvälju pole (ja pole

ka vajadust, kuivõrd kõikide moodulite jaoks on üks ja ühine (globaalsete) viitade keskkond)

Kuivõrd *FORTH* ei tunnista (ega anna selleks võimalustki) lokaalseid muutujaid, siis on kõik alamprogrammid (sõnad) *á priori* re-enteraablid. Rekursiooni teeb pisut keerulisemaks nõue, et suvaline objekt (sh. koolondefinitsioon) peab olema enne kasutamist kirjeldatud, ja seega *rekursiivne* koolondefinitsioon ei saa oma „kehas" iseenda poole pöörduda, kuivõrd ta pole „kompileerimise" käigus veel sõnastikku lisatud. Siin aitab lihtne „kavalus": rekursiivseks kavandatud sõna defineeritakse eelnevalt „tühisõnana", näiteks faktoriaali arvutav sõna:

```
VARIABLE F
1 F !
: FACT NOOP ;          NOOP=NO OPeration
….
(n --- 1)
: FACT DUP F @ * F ! 1–  DUP 1  >  IF FACT THEN ;
```

„Tühi" *FACT* jääb ka sõnastikku, ent järgmistes koolondefinitsioonides, mis kasutavad sõna *FACT,* pannakse *pfa*-välja viit viimasele tolle sõna definitsioonile.

4.2.8. Kokkuvõtteks

FORTH on eeskätt süsteemiprogrammide kirjutamiseks mõeldud, eelkäijate ja (üldiselt tuntud) järgijateta keel. Programmeerida saab nii masina- kui ka kõrgeimal abstraktsioonitasemel (tsüklid, valik, loogiline tingimus). Keeles puudub näiteks **goto-**operaator (ent see ei tee teda *struktuurprogrammeerimise* keeleks). Tõik, et *FORTH* on „magasinorienteeritud" paigutub ta muude oma klassi protseduurorienteeritud keeltega võrreldes kõige lähemale masinorienteeritud (näiteks, *PDP-11*-le või *Intel*ile) keeltele. Seda muljet võimendab asjaolu, et *FORTH*i süntaksit pole võimalik formaliseerida.

Keel on „paradigmade-väline": ta ei mahu ühegi tuntud paradigma raamesse ning mõeldamatud on ta ─ näiteks ─ objektorienteeritud või funktsionaalprogrammeerimise variandid.

Kes võiks olla *FORTH*-programmeerija? *Á priori* mõistagi see, kes on programmeerinud sellise masina assembleris, kus on magasinikäsud ja kes valdab algoritme ja andmestruktuure. See aga tähendab, et *FORTH* pole üldjuhul hea (andeka programmeerija) esimeseks keeleks ja arvatavasti üldse mitte hea näiteks esmakursuslaste jaoks esimeseks keeleks. *Esiteks*, ta on liialt palju eelteadmisi eeldav ja algebralises mõttes ebatraditsionaalne, ja *teiseks*, kui see (*FORTH*) on omandatud, siis ei vii see oskus mitte kuhugile ─ sa kas kirjutad *FORTH*is või ei, *FORTH*i analooge pole[1]. *FORTRAN*i ja *Lisp*i kõrval on *FORTH* üks vanimaid tänini resultatiivseid „elavaid" programmeerimiskeeli.

[1] Siiski, *FORTH*ist inspireeritud keel on näiteks *PostScript*.

4.3. *C*

C on üldotstarbeline, protseduurorienteeritud programmeerimiskeel ─ kui peame silmas programmeerimiskeelte klassifikatsiooni. Keele peamine autor on *Dennis Ritchie* ja *C*-keele loomine on orgaaniliselt seotud nii *PDP-11* arhitektuuri kui ka *UNIX*-operatsioonisüsteemiga. Keel konstrueeriti puhtalt pragmaatilistel eesmärkidel: kanda üle *PDP-7*[1] jaoks assembleris programmeeritud operatsioonisüsteem *UNIX* (autor *Ken Thompson*) üle *PDP-7*-ga ühildamatule *PDP-11*-le. Jätkem meelde: *C* tehti programmeerijate (*Ritchie* ja *Thompson*) poolt nende endi ülesande võimalikult lihtsamaks lahendamiseks (nagu muide ka *FORTRAN* ja *FORTH*). Tulemus meeldis professionaalidele ning *C* on tänapäeval konkurentsitult eelistatavaim süsteemide programmeerimise keel[2] ning üks kasutatavaimatest rakendusprogrammeerimise keeltest. Samas, *C* pole kunagi meeldinud akadeemilistele ringkondadele[3] ning pole palju selliseid aktsepteeritavaid kõrgkoole, kus seda õpetataks (vt. [78]).

Ken Thompson ja *Dennis M Ritchie* [77].

C loodi 1969. aastal ning arendustöö kestis 1972. aastani. *UNIX*i tuum *PDP-11*-le kirjutati esialgu assembleris, hiljem aga asendati ka see *C*-keeles programmeerituga (aastal 1973).

[1] *PDP-7* oli 18-bitine masin, mis oli ühildatav *PDP-4* j a *PDP-9*-ga (valmis 1965), ent mitte *PDP-11*ga. *Ken Thompson* kirjutas *PDP-7*-le opsüsteemi „*Unics*" (nimi oli inspireeritud mängu „*Space Travel*" toetava opsüsteemi nimest „*Multics*", mis pakkus ohtralt võimalusi graafika programmeerimiseks), ent peagi muutis ta oma süsteemi nime: *UNIX*. *PDP-7* pilt on pärit Oslo arvutimuuseumi saidilt.
C-keel sai oma nime *Thompsoni* keele *B* järgi, mida loodeti ülekandmiseks kasutada, ent kuivõrd *B* ei suutnud kasutada *PDP-11* uusi võimalusi, siis tehti sellest keelest uus versioon. Nimi *B* pärineb omakorda eeskujuks olnud keele *BCPL* nime esitähest

[2] Suur osa *Windows*ist on kirjutatud just tavalises *C*-s. Süsteemiga kaasnevaid rakenduspakette on programmeeritud ka muudes keeltes, näiteks *Internet Explorer* on kirjutatud keeles *C++*.

[3] *N. Wirth* ütles ([63]), et „Tänapäeval kasutavad programmeerijad üldjuhul keelt C…C esindab väga madalat abstraktsioonitaset. See keel väldib neid andmestruktuure, staatilisi andmetüüpe ja staatiliselt kontrollitavate liidestega moodulite fundamentaalseid mõisteid, mis annavad keelele matemaatilise selguse... Tegelikult esitab C assembleri koodi, mis on peidetud ilmetu süntaksi varju ning mis on täis tipitud igasugu salamärke."

Joonis 4.3a. *PDP-7*.[86].

4.3.1. Näiteprogramm

Esitame allpool ka *C*-keelse versiooni „*Wilson*i näitest": sisestada *n* (*n<100*) ja *n*-elemendiline reaalarvude vektor, leida elementide aritmeetiline keskmine ning trükkida keskväärtust ületavate elementide arv (vt [61, lk. 51]).

```
main(){
       float a[100],mean,sum;
       int n,i,number;
       scanf(„%d",&n);
       for(i=0;i<n;i++) scanf(„%f",&a[i]);
       sum=0.0;
       for(i=0;i<n;i++) sum=sum+a[i]:
       mean=sum/n;
       number=0;
       for(i=0;i<n;i++){
              if(a[i]>mean) number++;
              }
       printf(„MEAN= %f\n",mean);
       printf(„NUMBER OVER MEAN= %d\n",number);
       }
```

4.3.2. Andmed

Elementaartüübid.

C-keeles on neli baas-andmetüüpi:

char ─ üks bait (tavaliselt ühe sümboli hoidmiseks);

int ─ *integer*-tüüpi arv;

float ─ ujupunktarv;

double ─ topeltpikkusega ujupunktarv.

Arvuvälja pikkus sõltub protsessorist ning need pikkused on fikseeritud päisefailis *limits.h* (näiteks, 16-bitisel masinal on konstandi INT_MIN väärtus −32767 ja UINT_MAX 65535).

Integer-tüüpi võib kitsendada sõnadega *short, long* või *unsigned.* „Need kolm tüüpi ei pea aga igas realisatsioonis tingimata esitama erineva pikkusega täisarve ─ pikkused sõltuvad konkreetsest arvutist. Üldnõue on siiski, et *short* ei oleks kunagi pikem kui *int* ja *int* pikem kui *long."* [28, lk. 61]

Char-tüüpi saab edukalt kasutada ka *int*-tüüpi väikeste arvude jaoks, näiteks määrangut
char n;
võime interpreteerida kui arvu *n* kirjeldust, kusjuures $127 \geq n \geq -128$. Kirjutades aga
unsigned char n;
on selle muutuja võimalikud väärtused vahemikus 0..255.

Lisaks on keeles *viidatüüp*, mille deklareerimiseks pole omaette võtmesõna, ent on vahendid viitade tekitamiseks ja kasutamiseks (sümbolite * ja & abil).

Konstandid (vt. [29], lk. 37 jj) võivad esineda deklareerimata kujul literaalidena aritmeetilistes avaldistes ja olla deklareeritud makroga *#define,* näiteks
#define HTL 79 (*int*-konstant) või
#define pii 3.14 (*float*-konstant: väärtus sisaldab kümnendpunkti).

Silmas pidades *C* protsessori-taseme võimalusi pole üllatav, et saame defineerida nii 10-ndsüsteemi (vaikimisi), kaheksandsüsteemi (´\oiii') kui ka kuueteistkümnendsüsteemi (´\xiii') ühebaidiseid konstante. Nii näiteks järgmiselt defineeritud konstandi
#define BELL ´\o007' saatmine konsoolile annab lühikese vile, võrdväärne definitsioon on *#define BELL ´*\x7'.

„Sisse ehitatud" on üle kümne 1-baidise konstandi, näiteks
\n ─ reavahetus;
\t ─ horisontaalne tabulatsioon või
\? ─ küsimärk.

Agregeeritud tüübid.

Keele tasemel on kasutatavad staatilised *vektor* (so, ühemõõtmeline *massiiv*) ja mitmemõõtmeline *massiiv*, näiteks:
char a[22]; kirjeldab 22-baidise vektori (juurdepääs näit. *a[i]*) ja
int B[4][10][2]; kirjeldab kolmemõõtmelise massiivi (juurdepääs näit. *B[i][j][1]*).

Dünaamilisele vektorile ja massiivile tuleb mälu küsida programselt. Selline vektor on just samamoodi indekseeritav kui staatilinegi, ent kahe- ja enamamõõtmelise massiivi indekseerimist tuleb programmeerijal endal modelleerida. Osas 4.3.8 esitame näite ühest võimalikust variandist selle töö tegemiseks.

C toetab andmestruktuurist *tabel* tuntud *kirjet*[1] tüübiga *struktuur* (*structure*). Tabeli saame, kui defineerime *struktuur*-tüüpi massiivi.

Alltoodud näide illustreerib nii struktuuri, viidatüüpi kui ka rekursiooni: esitame algoritmi kahendpuu läbimiseks lõppjärjekorras (*postorder*), kusjuures „tegevus" seisneb tipu märgendi väljatrükis.

```
struct tipp{
      char label;
      struct tipp *vasak;     //viit vasakule alluvale või tühi
      struct tipp *parem;     //viit paremale alluvale või tühi
      };

void post(struct tipp *t){
      if(t->vasak != struct tipp *NULL) post(t->vasak);
      if(t->parem != struct tipp *NULL) post(t->parem);
      printf(„%c\n",t–>label);
      }
```

Teine andmetüüpide agregeerimise vahend on *ühend* (*union*), selle näite leiame osas „*C* ja protsessori-tase".

4.3.3. Tehted

Omistamine.

„Tavaline" omistamismärk on nagu algebras või *FORTRAN*is „=", ent lisandub palju originaalset:

a Θ= b: *Algoli* notatsioonis oleks see *a:=aΘb;* kus *Θ* on kas +, −, *, /, %, ^,| , << või >>. Näiteks, *a*=b* tähendab *a:=a*b;* ja *a*=y+1,* tähendab *a=a*(y+1);*
Jagamine „/" annab *int*-tüübi korral resultaadiks jagatise täisosa ning „%" ─ jäägi (viimast tehet ei saa kasutada ujupunktarvude puhul). Tehe „|" tähistab loogilist bitikaupa liitmist (*OR*) ning „^" ─ loogilist bitikaupa välistavat liitmist (XOR). Märgid „<<" ja „>>" tähistavad vastavalt nihutamist vasakule ja paremale.

„Varjatud" omistamine seondub *PDP-11 increment*- ja *decrement*-režiimidega. Nii on avaldis *i++* samaväärne avaldisega *i=i+1* (või *i+=1*), ja *j −−* on sama, mis *j=j−1*. Nood režiimid on kasutatavad ka kujul *++i* ja *−−i*. *Kernighan* ja *Ritchie* toovad järgmise näite ([29], lk.46)[2]:
Kui *n*=5, siis
x=n++; on samaväärne *x=n; n=n+1;* aga
x=++n; on samaväärne *n=n+1; x=n;*

[1] *Tabel* on kirjetest koosnev andmestruktuur, kusjuures tabeli kirje koosneb loogilisel tasemel *võtmest* ja sellega seotud infost.
[2] Asi on tegelikult keerulisem. Näiteks. kui a=3, siis avaldise 3+(++a) väärtus on 7, aga 3+a++ väärtus on 6, seejuures pärast avaldise väärtuse arvutamist mõlemal juhul a=4.

Increment-režiimi näiteks esitame stringide[1] kopeerimise funktsiooni koodi [29, lk.105]. Operandideks olevad stringid on esitatud viidatüübi abil.

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t){
     while((*s++=*t++)!='\0');
     }
```

„Tingimuslik" omistamine: *z=(a>b) ? a : b;* Muutujale *z* omistatakse *a* väärtus, kui *a>b*, muidu aga *z=b*.

Aritmeetika.

C-keeles on 5 binaarset operaatorit: +, −, *, / ja %. Neli esimest on rakendatavad kõigi samatüübiliste operandide vahel, viimane aga ainult *int*-tüüpi operandidega. Nimelt „/" annab tulemuseks kahe *int*-tüüpi operandi jagatise täisosa ning „%" ─ jäägi. Näiteks:

a=5/2; annab resultaadiks 2 ning a=5%2; annab resultaadiks 1.

Aritmeetilistes avaldistes on prioriteetsemad korrutamine ja jagamine; tehete sooritamise järjekorda saab muuta sulgude abil.

Võrdlustehted ja loogika.

C võrdlustehted on järgmised:
== : võrdub,
!= : ei võrdu,
> : suurem,
>= : suurem või võrdne,
< : väiksem,
<= : väiksem või võrdne.
Nende tehete prioriteet on madalam kui aritmeetikatehetel, näiteks [29, lk.41]
i<lim−1; täidetakse nii, nagu oleks kirjutatud *i<(lim−1);*

Loogilistes tingimustes kasutatakse konjunktsiooni-märgina „&&" ja disjunktsiooni-märgina „||". Avaldistes kontrollitakse loogilisi tingimusi vasakult paremale ning see tegevus katkestatakse esimese *väära* tõeväärtuse tuvastamisel. Näiteks ([29], lk.41) funktsioonis *get-line* on tsüklioperaator

```
for(i=0; i<lim-1 && (c=getchar()) != '\n' && c!=EOF; ++i)
     s[i]=c;
```

Selle operaatori täitmine katkestatakse kohe, kui *i ==lim−1*.

[1] *C* string on *char*-tüüpi vektor, kus lõputunnuseks on bait '\0'.

Bitiviisilised operaatorid

Nende tehete operandideks võivad olla ainult *int*-tüüpi muutujad (so, *char, short, int* ja *long* ning nende *unsigned*-variandid) ja operaatorid on:
& : loogiline „ja" (*AND*), 110 & 011 = 010,
| : loogiline „või" (*OR*)[1], 110 | 011 = 111,
^ : loogiline mittesamaväärtustamine (*XOR*), 110 ^ 011 = 101,
<< : nihutamine vasakule, 011 << 1 = 110,
>> : nihutamine paremale, 011 >> 1 = 001.

4.3.4. Muud operaatorid

Tingimuslik operaator on täiskujul *if(avaldis) operaator else operaator;* ja lühendatud kujul *if(avaldis) operaator;* seejuures konstruktsioonis *else operaator* võib see *operaator* olla uus tingimuslik operaator.

Lüliti on kujul

```
switch (avaldis){
      case konstantavaldis: operaator
      case konstantavaldis: operaator
      ...
      default: operaator
      }
```

Tavaliselt järgneb *case*-operaatorile operaatorsulg *break,* mis annab juhtimise lüliti tegevuspiirkonnast välja, ent on juhtumeid, kus on otstarbekas täita ka algvalikule järgnev operaator ─ sel juhul *break*'i ei kirjutata. Märkusena mainigem, et „avaldis" on ka muutuja nimi ning „konstantavaldis" ─ konstant. „*Default*" on sundvalik: tegevus, mis käivitub, kui „avaldise" väärtus ei võrdu ühegi „konstantavaldise" väärtusega.

Tsüklioperaatoritest on kasutatavad nii **for**- kui ka **while**-variandid. Neist esimese kuju on *for(avaldis1;avaldis2;avaldis3) operaator,* mis vastab „klassikalisele" tsüklile: 1. avaldis algväärtustab tsükliindeksi, teine määrab lõputingimuse ning kolmas tsükli sammu. Teise variandi kuju on *while (avaldis) operaator;* ─ kuni *avaldise* väärtus on tõene, korratakse *operaatorit*, kusjuures *avaldise* argumentide väärtusi muudetakse *operaatoris.*

Suunamine on harjumuspärane operaator kujul *goto <märgend>*. Kuivõrd *Kernighan* ja *Ritchie* [29, lk.65] kirjutasid oma raamatu „süstemaatilise programmeerimise" paradigma tippajal, siis nendivad nad, et *goto* pole kunagi hädavajalik ja selle kasutamist on tavaliselt lihtne vältida, ent et siiski on mõned situatsioonid, kus võiks toda konstruktsiooni kasutada. Üks näide on mitmekordne tsükkel tsüklis, kus *break* katkestab ainult kõige sisesema tsükli, aga tarvis on lõpetada kogu tsükliline töö.

[1] Kui eksikombel kirjutada loogilises tingimuses && asemel & (või || asemel |), siis tulemus sõltub kompilaatorist. Mõned "annavad andeks", mõned mitte.

4.3.5. Alamprogrammid

„Minimaalne" *C*-programm on tekstifaili xxx.c kompileerimise resultaat xxx.exe. Seejuures saab selles tekstis kodeerida protseduure ja funktsioone, mida kasutavad hiljemdefineeritud alamprogrammid (viimati tuleb defineerida *main*-moodul)[1]. Niisiis, me saame kasutada ainult eelnevalt defineeritud muutujaid, konstante ja alamprogramme. Tavaliselt peab suvaline programm mingil moel saama sisendandmeid ja väljastama resultaate, so. kasutama *BIOS*i ja *DOS*i funktsioone (kui peame silmas *PC*-masinaid).

Niisiis, iga *C*-programm peab evima võimalust kasutada kompilaatori standardfunktsioone, mis on rühmitatud oma orientatsiooni järgi. Teek, mille funktsioone tahetakse kasutada, tuleb deklareerida *C*-teksti alguses, osas, kus tohib ja saab kirjutada *C*-makrosid. Näiteks, makro

#include <string.h>

toimel lisab *C* preprotsessor (mis genereerib makrokäskude makrolaiendeid) *.c*-tekstile päisefailist ümberkirjutatud funktsioonide kirjeldused, mis võimaldavad noid funktsioone kasutada.

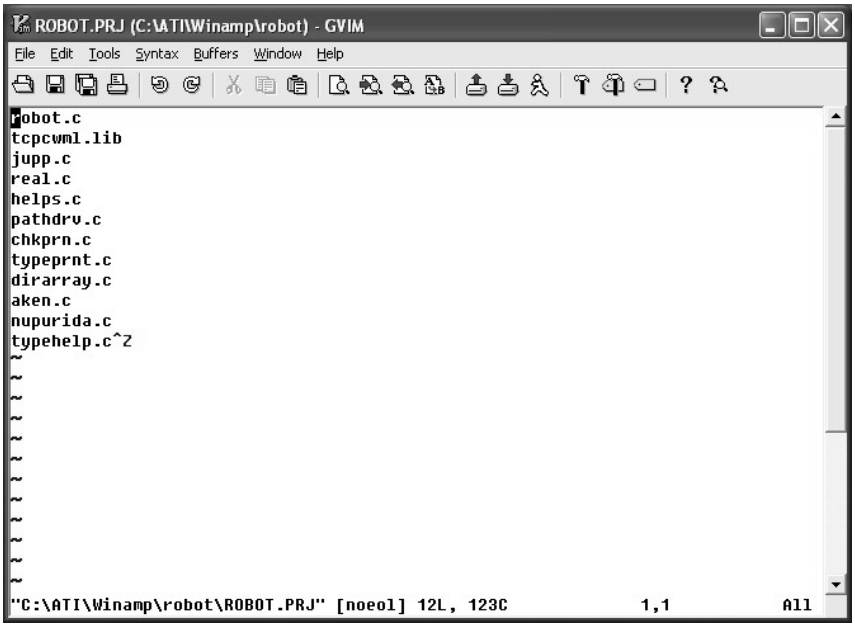
Iga iseseisvalt täidetavat algoritmi esitav *C*-tekst sisaldab reeglina järgmisi osi:

- makrod *#include* ja *#define*;
- globaalsete muutujate kirjeldused;
- alamprogrammide (protseduurid ja funktsioonid) koodid;
- peamoodul nimega *main*.

Kompilaatori standardfunktsioonide teekide nimed tuleb paigutada nurksulgude „<..>" vahele. Kasutaja saab samuti kirjeldada oma privaatseid mooduleid ─ nende teekide nimed on „jutumärkide" vahel. Muuseas on see võimalus kasutada *C*-keeles masinorienteeritud keeltest ja *FORTRAN*ist tuntud ühisvälju: tuleb kirjutada muutujate ja alamprogrammide kirjeldusi sisaldav päisfail (nimelaiendiga *.h*) ja sellega ühilduv tegelikele muutujatele mälu reserveeriv samanimeline *.c*-fail[2]. Viimane tuleb lülida kõrgema taseme programmi projekti.

„Peaprogramm" võib kasutada suvalise arvu „komponentprogrammide" alamprogramme, kusjuures nendevaheline infovahetus võib toimuda ühisvälja(de) vahendusel. Kõik need „komponentprogrammid" vormistatakse kui iseseisvad .c-tekstifailid. Nende „komponentprogrammide" (*x.c*) nimed tuleb (tavaliselt) paigutada komplekteerimiseeskirja esitava *.prj*- faili koosseisu. Allesitatud pildil on näha ühe sellise projektifaili tekst (*Turbo C*). Seejuures, põhiprogrammi sisaldav tekstifail peab olema esimesena deklareeritud ja tollest nimest tehakse ka *.exe*-faili nimi, näiteprojekti täitmine tekitab faili *robot.exe*.

[1] Suvalises järjekorras võime alamprogrammide tekste kirjutada siis, kui pärast globaalsete muutujate kirjeldusi esitada nonde alamprogrammide kirjeldused, nt. `void ap(int A, char *b);`
[2] Vt. lisa 5 ja 6.



Joonis 4.3.7a. Programmi *robot* projekt.

C erinevad realisatsioonid kasutavad tavaliselt erinevaid „komplekteerimiseeskirju".

Joonis 4.3.7b. Programmi käivitamine käsurealt.

C-programmis saab käivitada suvalist *.exe*-faili analoogiliselt sellele, kuidas see toimub *MS-DOS*i keskkonnas käsurealt, näiteks kui (pn)=p3, siis järgmiste käskude toimel pöördutakse käsuga *system* programmi *cmp32* poole ─ just nii, nagu näha ka joonisel 4.3.7b.

```
sprintf(MB,"cmp32 %s",pn);
system(MB);
```

Pärast *cmp32* töö lõppu antakse juhtimine *system*'ile järgnevale käsule.

4.3.6. Protseduurid ja funktsioonid

Nii protseduuridel kui ka funktsioonidel on ühesugune päiseformaat:

tüüp nimi(tüüp parameeter-1, ...,tüüp parameeter-n), kusjuures

protseduuri tüüp on *void*[1] ning funktsioonil tagastatava väärtuse tüüp. Kui viimane on viidatüüp, siis funktsiooni nimele lisandub prefiks „*". Funktsioonist väljutakse käsuga

return(avaldis, sh. ka *muutuja või konstant*);

Standardprogrammide teekide arv varieerub sõltuvalt keele realisatsioonist, ent tavaliselt on nende hulgas järgmised (vt. [29], lk. 241..258):

<*assert.h*> ─ diagnostika;
<*ctype.h*> ─ sümbolitestid;
<*float.h*> ─ ujupunktarvude karakteristikud;
<*limits.h*> ─ *int*-arvude karakteristikud;
<*math.h*> ─ kogum matemaatilisi funktsioone;
<*signal.h*> ─ katkestuste käsitlemine;
<*stdarg.h*> ─ alamprogrammi muutuva väärtuste arvuga parameetri töötlemine;
<*stdio.h*> ─ sisend- ja väljundfunktsioonid;
<*stdlib.h*> ─ „kasulikud" (*utility*) funktsioonid, näit. tüübiteisendused, dünaamilise mälu haldamine, *system* käsurea rollis jmt.;
<*string.h*> ─ stringifunktsioonid;
<*time.h*> ─ kuupäev ja kellaaeg.

Järgmises jaotises esitame näite, mis peaks illustreerima mitut ülalkäsitletud teemat: .c-faili üldkuju, päisefailide deklareerimine, muutuva väärtuste arvuga parameetri kasutamine, deskriptori tegemine ja kasutamine ning „normaalselt" indekseeritava dünaamilise *n*-mõõtmelise massiiviga operatsioonide kirjeldamine.

[1] Mõnikord on otstarbekas vormistada protseduur funktsioonina, mille tagastatavat väärtust ei kasutata ─ näiteks siis, kui protseduuril on mitu sõltumatut lõpetamispunkti; neis on lihtne kirjutada `return(n);` kus *n* on suvaline konstant.

4.3.7. Dünaamiline massiiv: *array.c*

```
/*   n-mõõtmeline dünaamiline int-massiiv. 20.04.00, VisualC++ 6.0  */
/*  see fail ei võimalda kompileerida .exe-faili (main() puudub, kood on
lisatav .prj-faili abil muudele rakendustele */

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <alloc.h>
#include <stdarg.h>

// massivi kirjeldus
struct descr{
      int *a;     // array: viit massiivile
      int dima;   // dimensioonide arv -- nt 3
      int *dim;   // nt 2 3 2; indeks[i]=1..dim[i]; i=1..dima
      int *DM;    // 0..dima-1: DM[i]=dim[i]*DM[i-1]; DM[0]=1
      };

      /*  massiivi  kirjeldamine,  mälueraldus.  n=dimensioonide  arv,
      k=dimensioonid.  */
      struct descr *array(int n,int k, ...){
      va_list ap;
      struct descr *D;
      int b,i;
      D=(struct descr *)malloc(sizeof (struct descr));
        //dünaamiline mälueraldus
      memset(D,'\0',sizeof(struct descr));
            //saadud mälutäitmine nullidega
      va_start(ap,k);  // initsialiseerib argumentide listi
      D->dima=n;
      D->dim=(int *)malloc(n*sizeof(int));
      memset(D->dim,'\0',n*sizeof(int));
      D->DM=(int *)malloc(n*sizeof(int));
      memset(D->DM,'\0',n*sizeof(int));
      D->dim[0]=k;
      for(i=1;i<n;i++) D->dim[i]=va_arg(ap,int);  /* tsükkel
                  üle argumentide */
      va_end(ap);  /* argumentide listi kustutamine --
            süsteemsed parameetrid! */
      printf("rajad: ");
      for(i=0;i<n;i++) printf("%d ",D->dim[i]); printf("\n");
      D->DM[0]=1;
      for(i=1;i<n;i++) D->DM[i]=D->DM[i-1]*D->dim[i-1];
      printf("DM: ");
      for(i=0;i<n;i++) printf("%d ",D->DM[i]);
      b=1;
      for(i=0;i<n;i++) b=b*D->dim[i];
      D->a=(int *)malloc(b*sizeof(int));
      memset(D->a,'\0',b*sizeof(int));
      printf("\n");
      return(D);
```

```
}

/* massiivi D elemendi lugemine: k on indeksite loetelu
   NB! indeks[i]=1..dim[i] */

int *val(struct descr *D,int k, ...){
      va_list ap;
      int i,n;
      int x;
      va_start(ap,k);
      n=D->dima;
      x=k-1;
      for(i=1;i<n;i++) x=x+D->DM[i]*(va_arg(ap,int)-1);
      va_end(ap);
      return(D->a[x]);
}

/* massiivi D kirjutamine: v on väärtus, k on indeksite loetelu */

void wri(int v,struct descr *D,int k, ...){
      va_list ap;
      int i,n;
      int x;
      va_start(ap,k);
      n=D->dima;
      x=k-1;
      for(i=1;i<n;i++) x=x+D->DM[i]*(va_arg(ap,int)-1);
      va_end(ap);
      D->a[x]=v;
      }
```

4.3.8. *C* ja protsessori-tase

Selles jaotises vaatleme põgusalt *C*-keele neid nüansse, mis teevad temast süsteemiprogrammeerimise keele. Loodetavasti on järjekindel lugeja adunud, et näiteks operatsioonisüsteemi programmeerimiseks peab olema võimalik (kui keskenduda *Intel*-protsessorile) evida juurdepääsu:

- aparatuursetele registritele
- vaba mälu piiraadressidele ja nende muutmisele;
- kui tegemist on *MS-DOS*-süsteemiga, siis kõigile *DOS*- ja *BIOS*-katkestustele:
- võimalusele katkestusi genereerida.

Lisades 3 ja 4 on toodud vabavaralise *GNU-C*[1] päisefailide *BIOS.H* ja *DOS.H* tekstid, millega tutvumine peaks andma ettekujutuse *C*-keele võimalustest tööks „madalaimal tasemel".

Tuleb rõhutada, et tänu masinorienteeritusele erinevad erinevate arvutite *BIOS*id üksteisest detailides, mistõttu võib ühel platvormil kirjutatud ja *BIOS*i funktsioone kasutatavate programmide ülekandmine teisele platvormile nõuda mõningat „kohendamist"; näiteks

[1] Copyright © 1989, 1991 Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

*IBM*i *PC-BIOS* ja selle funktsioonid pole üksüheses vastavuses *GNU-C* oma(de)ga, ent kui sellega oskame arvestada, pole lisatöö kuigi märkimisväärne.

Mõned *BIOS*i funktsioonid on dubleeritud *C* funktsioonidega, näiteks *IBM PC* funktsioonid *putch()* või *getche()*, ent enamik pole ning nende kasutamiseks tuleb neid kasutada „ilmutatud kujul". *IBM PC* võimaldab seda funktsiooni *int86(INT,&inregs,&outregs)* abil. Parameetrid on järgmised:

- *INT* on *int*-tüüpi katkestuse number
- *&inregs* on registrite sisendväärtused (vaikimisi kehtivad väärtused)
- *&outregs* registrite tagastatud väärtused

Robert Lafore [31, lk. 327] toob järgmise näite (kommentaarid on meie tõlgitud):

```
/* memsize0.c trükib mälu mahu */
#define MEM 0x12    /* BIOSi katkestuse number */
main(){
      struct WORDREGS {   /* 16-bitised registrid */
            unsigned int ax;
            unsigned int bx;
            unsigned int cx;
            unsigned int dx;
            unsigned int si;
            unsigned int di;
            unsigned int flags;
            };
      struct BYTEREGS {   /* 8-bitised registrid */
            unsigned char al,ah;
            unsigned char bl,bh;
            unsigned char cl,ch;
            unsigned char dl,dh;
            };
      union REGS          /* nii baidid kui ka sõnad */
            {
            struct WORDERGS x;
            struct BYTEREGS h;
            };
      union REGS regs;
      unsigned int size;
      int86(MEM,&regs,&regs);
      /* „sisendregistreid" ei muudetud */
      size=regs.x.ax;         /* mälu maht AX-registrist */
      printf(„Memory size is %d Kbytes",size);
}
```

Järgmises näites kasutatakse *IBM PC DOS*-funktsiooni peitmaks kursorit [31, lk. 331].

```
/* curoff.c */
#include <dos.h>  /* seal on defineeritud union REGS */
#define CURSIZE 1 /* teenus „set cursor size" */
#define VIDEO 0x10      /* BIOSi videokatkestuse number */
#define STOPBIT 0x20  /* see bitt lülitab kursori välja */
main()
{
```

```
        union REGS regs;
        regs.h.ch=STOPBIT;
        regs.h.ah=CURSIZE;
        int86(VIDEO,&regs,&regs):
        /* videokatkestuse väljakutse */
}
```

Meie õppevahendi piiratud maht ega ka eesmärgid ei võimalda esitada komplitseeritumaid näiteid; huviline võiks noist lihtsamatega tutvuda näiteks *Robert E. Marmelstein*i raamatu [39] abil.

4.3.9. Kokkuvõtteks

Niisiis, *Ritchie, Thompson* jt. (teistest eeskätt *Kernighan*) töötasid *C* välja utilitaarsetel eesmärkidel (kanda *UNIX* võimalikult kiiresti üle kardinaalselt uue arhitektuuriga masinale (*PDP-11*) . Polnud ei aega ega ka mõtet pöörata (liigset) tähelepanu keele „akadeemilistele aspektidele" nagu täitmisaegne turvalisus (näiteks indeksite ja dünaamilise mälu piiride kontroll, vaba juurdepääs protsessori-tasemele jmt), süntaksi selgus või kogu täidetava programmi lähteteksti tervikesitus. Tähtis oli funktsionaalsus ning efektiivsus. Ja tulemuseks oli keel, mis on tänaseks kõige populaarsem süsteemprogrammeerimise instrument ja mille leksika ning süntaks on olnud eeskujuks paljudele *C*-st sootuks erinevate orientatsioonidega keeltele (näiteks *Java* või *PHP*).

Joonisel 5a on näidatud, et *C* „esivanem" on *ALGOL-60*. Kahtlemata sarnaneb *C* rohkem *ALGOL*ile kui *FORTRAN*ile, *COBOL*ile või muudele kaasaegsetele teerajajatele (näit. *Lisp*ile, *APL*ile või *Snobol*ile, rääkimata juba *FORTH*ist). Johtuvalt *Thompson*i eelnevast kogemusest *ALGOL*i-sarnaste keelte *BCPL*i ja *B*-ga oli normaalne, et mitmes mõttes sai just *ALGOL* uue keele prototüübiks ─ kui peame silmas ranget tüpiseerimist (kõik kasutatavad objektid, nii lihtmuutujad, agregaadid kui ka alamprogrammid tuleb enne kasutamist täpselt kirjeldada)[1], operaatoreid (avaldised, loogilised tingimused, *for*- ja *while*-tsüklid, *switch*-tüüpi lüliti, märgendid ja suunamine, rekursioon) ning alamprogrammide kirjeldamise võimalust põhiprogrammi tekstiga samas „failis".

Ent samas loobusid *C* autorid plokkstruktuurist, alamprogrammide parameetrite edastamisest *by name* jmt. ning tõid juurde masinorienteeritud keeltest tuntud võimalused: eraldi kompileeritavad alamprogrammid, (varjatud) ühisväljad, dünaamilised andmestruktuurid (*struct* ja *union*, ent paradoksaalselt ei toeta ta dünaamilisi massiive), viidatüübi ning juurdepääsu protsessoritasemele.

C on suhteliselt väike (kui võrrelda *COBOL*i, *PL\I* või *Ada*ga) ja lihtsalt realiseeritav keel, ent tööriistana sobib ta pigem kogenud kui algajatele programmeerijatele[2].

[1] Samas puudub täitmisaegne tüübikontroll, mis annab head võimalused vigaste programmide kirjutamiseks [61, lk. 50].
[2] See kehtib kõigi süsteemprogrammeerimise keelte kohta.

Ja veel „inimlikust aspektist" andis sellele keelele hinnangu umbes 25 aastat tagasi TRÜ ja Küberneetika Instituudi ühise suvekooli (nood toimusid Elbis, Sindist pisut ülesvett, *Erik Meriste* juhitud spordibaasis) Moskvast tulnud noor lektor *Vladimir Serebrjakov*, kui talt küsisime (ise kogemust omamata), et kuidas *C* tundub. *Volodja* ütles: „Teate, see on midagi narkootikumi-sarnast… Alul on vastik, aga pärast ei saa lahti."

5. PROTSEDUURORIENTEERITUD KEELTE KLASSIFIKATSIOONID

Meenutagem: kõige üldisema programmeerimiskeelte klassifikatsiooni järgi jagunevad need keeled kahte suurde klassi: masinast sõltuvad keeled (mikro- ja masinkoodid, assemblerid ja makroassemblerid) ning masinast sõltumatud keeled ─ noid viimaseid saab suuremate või väiksemate efektiivsuskadudega realiseerida ka teiste protsessoritüüpide jaoks. Omakorda jagunevad „sõltumatud keeled" protseduur- ja probleemorienteerituiks: esimesed nõuavad nii või teisiti algoritmi samm-sammult esitamist ja teiste jaoks piisab sisendi ja väljundi spetsifikatsioonidest (mõnikord, näit. *RPG*[1], koos võimalusega anda näpunäiteid tee *sisend → väljund* läbimiseks). Käesolevas peatükis keskendume protseduurorienteeritud keeltele.

Protseduurorienteeritud keelte eesmärgid dikteerivad peaaegu alati nende realiseerimise viisi: kas keel on kompileeritav või interpreteeritav (siiski, näiteks *FORTH* evib mõlemat omadust).

Kompileeritav protseduurorienteeritud keel on ehedal kujul *protseduurorienteeritud*: kõik konstruktsioonid evivad ekvivalente masinkoodi ja op-süsteemi tasemel, ning nii palju kui võimalik, tõlgib translaator masinkoodi, ja ülejäänu (lahendamisaegsete dünaamiliste situatsioonide) jaoks lisab keele kompilaator vajalikud tolle keele virtuaalarvuti andmestruktuurid ja nondega opereerivad koodid. Kompileeritava keele virtuaalmasina maht küündib harva üle paarikümne protsendi reaalse arvuti vahenditega tehtust.

Interpreteeritav keel[2] pole kitsendatud ei protsessori ega ka op-süsteemiga, kuivõrd sel puhul on virtuaalmasina tehtu oluliselt suurem aparatuursest ─ viimane on sageli marginaalne. Pole juhus, et reeglina on arvutivõrkude, multmeedia või arvutigraafika (sh. arenenud tekstitoimetite) programmeerimise keeled interpreteeritavad.

Protseduurorienteeritud keeled on valdavalt *universaalsed*, so. (peaaegu) kõik algoritmid, mis on realiseeritavad suvalise selle klassi keeles, on (tavaliselt) realiseeritavad teistes selle klassi keeltes. Siiski, tavaliselt on keeled „spetsialiseerunud" teatud *ülesannete* klassidele nii, et nonde klasside algoritme on oluliselt hõlpsam programmeerida spetsialiseerunud keeles kui muude orientatsioonidega keelt abil.

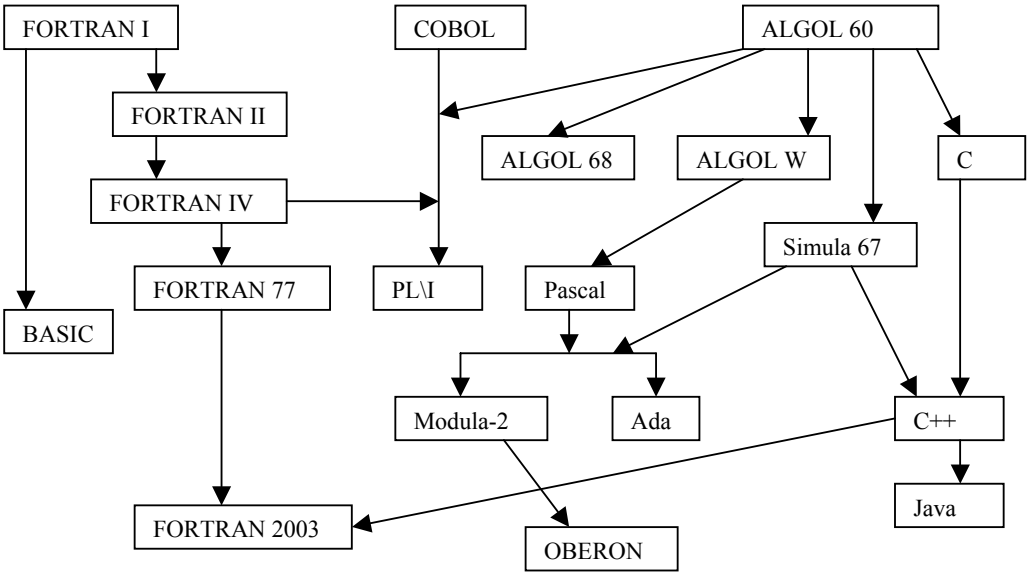
Protseduurorienteeritud keelte klassifikatsioone on palju. Näiteks, alternatiividena pakett-töötluse ja interaktiivsed keeled, paralleeltöötluse ja seda moodust mitte võimaldavad keeled või iseloomustab klassifikaator keelte orienteeritust erinevatele programmeerimisstiilidele (mida nimetatakse sageli *paradigmadeks*, me käsitleme neid pisut hiljem).

---

[1] *RPG ─ Report Programming Generator ─* aruannete generator (vt. näit. [87]).

[2] Esimeste interpretaatorite hulgas oli palju neid, mis interpreteerisid sisendprogrammi teksti tasemel ja operaatorhaaval ─ mis on ebaefektiivne. Normaalne interpretaator töötab programmi analüüsi puu peal nagu kompilaatorgi.

Enam-vähem neutraalne klassifikaator on liigitamine sobiva(ma)te ülesannete klasside-alusel. Joonisel 5a on toodud keeli enimmõjutanud keelte põlvnemis- ja sugulussuhted. Seal puudub üks kolmest suurest: *Lisp*. Põhjuseks on, et sel keelel (mis pani aluse funktsionaalse programmeerimise stiilile) pole mingeid „veresidemeid" joonisel esindatutega. Nagu muide ka *FORTH*il, mida me samuti sealt ei leia, ja paljudel muudel keeltel, näiteks *APL* või *Snobol*.



Joonis 5a. Esimeste kõrgtaseme keelte perekonnad (vt. [61, lk. 23 ja 27)

5.1. Arvutusülesannete (teadusarvutuste) keeled

Esimesed protseduurorienteeritud keeled ─ *FORTRAN* ja *ALGOL* ─ olid orienteeritud **arvutusülesannetele** (ehkki esimene nende lahendamisele ja teine nende algoritmide publitseerimisele). *FORTRAN* ja tema nimekuju säilitanud järeltulijad järgivad algset orientatsiooni ning ei evi protseduurorienteeritud keelte tasemel (üldtuntud) konkurente.

*ALGOL*il pole (ainult) teadusarvutustele orienteeritud järeltulijat.

5.2. Äri- ja raamatupidamiskeeled

Selle orientatsiooniga keeltest oli esimene *COBOL* (*COmmon Business-Oriented Language*), mis oli viimastel aastakümnetel näiteks USAs kõige rohkem kasutatav keel ja kuulub *IT*-erialade kohustuslikku õppeprogrammi. Keele töötas välja komitee nimega *CODASYL* (*COmmon DAta SYstem Languages*), möödunud sajandi 50-ndate aastate

teisel poolel. Mõningatel (kontrollimata) andmetel oli 70-ndatel aastatel 90% *kõigist* programmidest kirjutatud just selles keeles.

COBOL oli esimene keel, mis võimaldas kirjeldada objektmasinat (ja selle op-süsteemi poolt dikteeritud platvormi) ning võimaldas kirjeldada ja kasutada andmestruktuuri *kirje*, mis omakorda on abstraktse andmestruktuuri *tabel* element. Senini oli ainus andmetöötluse-andmestruktuur *homogeenne massiiv.*

COBOL on realiseeritud kompileeritava keelena. Ent algusest peale sattus üksmeelse kriitika alla *COBOL*i originaalne süntaks, mis püüdis järgida loomulikku (inglise) keelt ja seda igal tasemel, näiteks *Algoli* operaator

```
A:= B+C;
```

on *COBOL*is

```
ADD B TO C GIVING A.
```

Tulemuseks olid harjumatult pikad programmitekstid, raskused kompileerimisel ning akadeemiliste ringkondade üksmeelne kriitika[1]. Laename näite *T. Pratt*ilt [44, lk. 395]

```
IDENTIFICATION DIVISION.
PROGRAM-ID.SUM-OF-PRICES.
AUTHOR.T-PRATT
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER CDC6400.
OBJECT-COMPUTER CDC6400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT INP-DATA ASSIGN TO INPUT.
     SELECT RESULT-FILE ASSIGN TO OUTPUT.
DATA DIVISION.
FILE SECTION.
FD INP-DATA LABEL RECORD IS OMITTED.
01 ITEM-PRICE..
       02 ITEM PICTURE X(30).
       02 PRICE PICTURE 9999V99.
       02 FILLER PICTURE X(44).
FD RESULT-FILE LABEL RECORD IS OMITTED.
01 RESULT-LINE PICTURE X(132).
WORKING-STORAGE SECTION.
77 TOT PICTURE 9999999V99,VALUE 0,USAGE IS COMPUTATIONAL.
77 COUNT PICTURE 9999,VALUE 0,USAGE IS COMPUTATIONAL.
01 SUM-LINE.
       02 FILLER-VALUE  'SUM = ' PICTURE X(12).
```

[1] *E. Dijkstra* kirjutas näiteks, et *COBOL*i õpetamist kõrgkoolides tuleks karistada kriminaalkorras [33].

```
        02 SUM-OUT PICTURE $$,$$$,$$.999.
        02 FILLER-VALUE  'NO OF ITEMS = ' PICTURE X(21).
        02 COUNT-OUT PICTURE ZZZ9.99.
01 ITEM-LINE
        02 ITEM-OUT PICTURE X(30).
        02 PRICE-OUT PICTURE ZZZ9.99.
PROCEDURE DIVISION.
START.
        OPEN INPUT INP-DATA AND OUTPUT RESULT-FILE.
READ-DATA.
        READ INP-DATA AT END GO TO PRINT-LINE.
        ADD PRICE TO TOT.
        ADD 1 TO COUNT.
        MOVE PRICE TO PRICE-OUT.
        MOVE ITEM TO ITEM-OUT.
        WRITE RESULT-LINE FROM ITEM-LINE.
        GO TO READ-DATA.
PRINT-LINE.
        MOVE TOT TO SUM-OUT.
        MOVE COUNT TO COUNT-OUT.
        WRITE RESULT-LINE FROM SUM-LINE.
        CLOSE INP-DATA AND RESULT-FILE.
        STOP RUN.
```

Kommentaarina: see *COBOL*-programm koostab arve ja trükib selle välja.

Majandusülesannete programmeerimiseks mõeldud protseduurseid keeli oli teisigi (N. Liidus näiteks *АЛГЭК* ja *АЛГЭМ*), ent raamatupidamise (sh. materjali- ja palgaarvestus), majandustegevuse analüüsi ja muude aruannete ning tabelite koostamiseks kasutatakse tänapäeval kas spetsiaalpakette või andmebaasisüsteemide vahendeid ─ aga nende programmeerimise keeled on *probleemorienteeritud*, seejuures loodud oluliselt hiljem kui *COBOL*. Kaasaegne taolise orientatsiooniga keel oli samuti nagu nood hilisemad vahendidki samuti probleemorienteeritud keel (või süsteem) *RPG* (*Report Programming Generator*) ─ aruannete generaator. *COBOL*i (eeskätt süntaktilised) iseärasused ei võimaldanud tal *RPG*-d välja tõrjuda, ent tal olid kõik need eelised, mis on protseduursetel keeltel (eeskätt universaalsus) probleemorienteeritute ees ja samas see miinus, mis on probleemorienteeritud keelte ühine pluss: viimased on lakoonilisemad ja efektiivsemad oma kitsas ülesannete klassis. Konkreetselt, *RPG* aruannete genereerimisel.

5.3. Tehisintellekti (TI) programmeerimise keeled

Tehisintellekt (*Artifical Intelligence, AI*) on leksikonide ([11], lk. 21 ja [54], lk 27) järgi „arvutiteaduse selline haru, kus uuritakse võimalusi leida algoritme ja neid programmeerida valdkondades, mis imiteeriksid mõttetegevust ─ näiteks kõnetuvastus, deduktsion[1], loogiline tuletamine (näiteks: varblastel on suled, sest nad on linnud ja need on sulelised), kogemustest õppimine ja otsuste tegemine mittetäielikule informatsioonile tuginedes. *AI* valdkond hõlmab kaht omavahel seotud suunda: üks neist on elusolendite mõtlemisprotsesside uurimine ja teine ─ kuidas varustada arvutiprogramme analoogiliste võimetega. Mõned alul arvuti jaoks liiga rasketeks arvatud ülesanded, näiteks malemäng, osutusid ootamatult lihtsateks ja mõned esialgu lihtsateks peetud ülesanded, näiteks kõnetuvastus või masintõlge[2], osutusid tegelikult äärmiselt rasketeks. Selle valdkonna praktiliselt lahendatud ülesannete hulka kuuluvad (peale malemängu) *diagnostilised vahendid*, mida kasutatakse nii meditsiini- kui ka muude valdkondade *ekspertsüsteemides*" ─ ent tõenäoliselt ka (vähemalt osaliselt) sellised loogikaülesanded nagu *automaatne teoreemitõestamine* ja *programmide verifitseerimine* (vt. [53]**,** lk 298 jj.).

Tehisintellekti ülesanded tunduvad olevat sellised, kus tuleb salvestada uusi teadmisi nii, et ka senised säiliksid (programmeerimise aspektist: muutujate väärtusi ei tohi üle kirjutada ─ omistamine peaks olema välistatud, nii ilmutatud kujul kui ka kõrvalefektide toimel) ning programm saaks täitmise käigus genereerida uusi andmestruktuure ning neid kasutavaid funktsioone. On ju konstruktiivne mõtlemine (lihtsamalt, õppimine) protsess, kus senistele (mis siis, et tihtipeale lünklikele) teadmistele „ehitatakse peale" adekvaatsem versioon, unustamata seniõpitut või -kogetut.

5.3.1. *Lisp.*

Selle keele tutvustamisel on kasutatud järgmisi materjale: [36], [44, lk. 455 − 480], [61, lk. 257−265], [130], [131] ja [132].

Teadaolevalt on *John McCarthy* meeskonna aastal 1958 disainitud keel *Lisp* (*List Processing*[3]) *MIT*is (*Massachusetts Institute of Technology*) vanuselt teine kõrgtaseme keel pärast *FORTRAN*i (meenutagem, ka see loodi *MIT*is) ja esimene *funktsionaalprogrammeerimise*[4] keel.

---

[1] **Deduktsioon** on arutlemise viis, kus järeldus tehakse paljude erinevate andmetete (faktide) alusel. Liikumine toimub üldiselt üksikule, erinevalt induktsioonist, kus järelduste liikumine toimub üksikult üldisele [89] – meie märkus.

[2] „Tehniline tõlge" ─ näiteks, mingi tehnikaalase teksti või formaaljuriidilise teksti „tooriku" genereerimine pole eriti raske, ent senilahendamatu (võib olla, et sootuks võimatu) on luule adekvaatne masintõlkimine – meie märkus.

[3] *Wikipedia* andmetel on akronüümi *Lisp* interpreteeritud ka nii: *Lots of Irritating Superfluous Parentheses*, *Let's Insert Some Parentheses* või *Long Irritating Series of Parentheses* [130].

[4] Funktsionaalprogrammeerimise paradigmat käsitletakse *ATI* õppekavas kursuses "Funktsionaalprogrammeerimise meetod" (MTAT.05.047, eestvedaja on dots. *Varmo Vene*). Tänu sellele ei pea me nende kaante vahel üritama anda selle teema süvakäsitlust – seda teeb mainitud loengukursus. Liiati pole *Lisp* tänapäevase arusaama kohaselt "puhas" funktsionaalne keel, vaid evib palju "ebafunktsionaalset" [130].

John McCarthy (s. 4.09.27, Bostonis, Massachusetts, USA)

Programmeerimiseks oli 50-ndate keskpaigas kaks võimalust: kas kasutada assemblerit (*resp.* masinkoodi) või *FORTRAN*i. Assembler võimaldas programmeerida kõike seda, mida toetas protsessor (ja välisseadmed), sh. kirjutada interaktiivseid[1] ja/või rekursiivseid programme. *FORTRAN* neid võimalusi ei pakkunud; ta oli loodud arvutusmatemaatika ülesannete programmeerimiseks ning oli selle valdkonna jaoks piisavalt väljendusrikas ja efektiivne. Ent samas oli selge, et arvutiga saab teha sootuks rohkemat kui teadusarvutused, eeskätt sümboltöötlust (sh. tekstitöötlust[2]) ning programmeerida nn. "tehisintellekti ülesandeid" − esmased valdkonnad olid teoreemide tõestamine, robootika, masintõlge, mängud (esmajoones male) jmt.

McCarthy üritas esialgu (1956. a. suvel) kasutada *FORTRAN*i[3], võimaldamaks programmeerida noid rakendusi selle keele keskkonnas, masinale *IBM 704*[4]. Kuivõrd me oleme juba ülalpool tutvustanud keelt *FORTH* (autor *Chuck Moore*, *McCarthy* õpilane), siis ei saa nende ridade autor vastu panna ahvatlusele noid keeli pisut võrrelda.

- Mõlemad on *interaktiivsed*: me istume operaatori-kirjutusmasina taga, evime võimalust lisada teeki uusi sõnu (*FORTH*) või funktsioone (*Lisp*), kusjuures juhul, kui me neid üle defineerime, siis "vana variant" ei kao, vaid nihkub otsingutees koha võrra allapoole. Analoogiline liikumine toimub sõnade/funktsioonide kustutamisel. Ja veel: me võime kasutada uutes definitsioonides juba defineeritud objekte.
- Mõlemad kasutavad avaldiste *Poola kuju*, *FORTH* inverteeritud Poola kuju (*postfix*) ja *Lisp* − Cambridge'i Poola kuju (*CPF*, *prefix*).
- Mõlemi virtuaalmasin kasutab üldistatult puustruktuuri − *FORTH*il on see sisuliselt graaf, *Lisp*il "seotud ahelad" (mille puhul pole välistatud graafi moodustumine − juhul, kui mõnele (alam)ahelale moodustub rohkem kui üks viit).

---

[1] Ärgem arvakem, et see võimalus tekkis alles koos displeide (kuvaritega), palju varem ilmusid nn. puldikirjutusmasinad, mis võimaldasid operaatoril pidada kahepoolset loetavas keeles sidet täidetava programmiga. Mõned neist kirjutusmasinatest olid meie jaoks üpris eksootilised, näiteks *Iverson*i *APL*i pult või Armeenia minimasina *Nairi* oma.

[2] Sümboltöötlus on laiem mõiste kui tekstitöötlus, viimase all mõeldakse loomulikus (inim)keeles kirjutatut, ent esimene lisab sellele näit. male- või bridžipartii notatsiooni.

[3] Süsteemi nimeks sai *FLPL − F[ORTRAN] List Processing Language.*

[4] Kaheaadressiline pesamasin, 36-bitine pesa kahe 15-bitise aadressväljaga, esimene neist oli mõeldud operandi aadressi jaoks (*address*) ja teine − infovahetuseks 15-bitise indeksregistriga (välja "nimi" oli *decrement*).

- Mõlemi süntaks on raskesti formaliseeritav ning programmide kirjapilt ei evi varajasemaid analooge, on raskesti õpitav ja eriettevalmistuseta loetamatu. Ent kui nad on omandatud, on neis programmeerimine loogiline, lakooniline ja viljakas (viimane eeskätt assembleriga võrreldes, *C*-keelt kasutades on tööviljakus siiski kõrgem − mõistagi, peame silmas süsteemseid rakendusi).
- Mõlemad on efektiivseimad masinatel, kus on realiseeritud „aparatuurne magasin" (so, käskude süsteem toetab *push*- ja *pop*-operaatoreid). Nii *FORTH*i kui ka *Lisp*i loomulik realiseerimisvahend on *LIFO*-tüüpi magasin: esimesel andme- ja naasmismagasinideks, teisel rekursiooni jaoks. Meie andmetel pakkus seda võimalust esimesena *PDP*-arhitektuur (*J. McCarthy* kirjutab [36], et *PDP* tegi seda tema palvel).
- Siiski, keelte orientatsioonid on erinevad. *FORTH* sobib operatsioonisüsteemi programmeerimiseks (või asendamiseks) ja suurte reaalajasüsteemide (võimalik, et interaktiivsete) programmeerimiseks, *Lisp* aga eeskätt tehisintellekti ülesannete (sh. interaktiivsete reaalaja-ülesannete) programmeerimiseks.

*FORTRAN-Lisp*i programm (või interaktiivne seanss) nägi välja nii, et tekst algas funktsioonide defineerimisega (á la *FORTRAN*i lokaalsed alamprogrammid) ning järgnes noid kasutav põhiprogramm − mõistagi samuti funktsiooni kujul. Kuivõrd *McCarthy* kasutas *FORTRAN*-translaatorit, oli rekursioon välistatud, ent realiseerimiskeel ei olnud takistuseks kahe uue nüansi lisamiseks: neist esimene oli *if-then-else*-konstruktsioon (mis sai üldtuntuks *ALGOL 60* projektist[1]) ning teine oli funktsioonide kasutamine avaldiste argumentide rollis.

Tolle „kasutamise" võimaldamise idee sai *McCarthy Alonzo Church*i *Lambda-arvutuste* teooriast; lähemalt vt. [131] ja [53, lk. 376 jj]. *J. McCarthy* kirjutas: "Funktsioonide kasutamiseks argumentidena on vaja funktsioonide notatsiooni ning *Church*i (1941) λ-notatsiooni kasutamine tundus loomulikuna. Kuivõrd ma ei saanud raamatu ülejäänud osast aru, polnud minu jaoks ahvatlust kasutada tema funktsioonide defineerimise palju üldisemat mehhanismi. *Church* kasutab kõrgemat järku funktsionaalsümboleid selmet kasutada tingimuslikke avaldisi. Viimased on arvutitel palju hõlpsamini realiseeritavad." [36].

Andmestruktuurid.

Meenutagem üht *von Neumann*i printsiipi: nii programm kui ka programselt töödeldavad andmed paiknevad operatiivmälus, ja mis on programm ja mis on andmed, on interpretatsiooni küsimus. *Lisp* on selle valguses üks väheseid toda printsiipi järgivaid kõrgtaseme keeli: keele tasemel on programm esitatav sümbolavaldis(t)e (*S-expression=symbol expression*) abil, prefiks-kujul sulgavaldistena, ning samamoodi esitatakse ka andmeid. Ning *Lisp*-interpretaatori jaoks pole vahet, kas töödeldakse juba loodud andmeid või interpreteeritakse olemasolevat programmi, või genereeritakse uusi interpreteeritavaid andmeid, on need siis tõepoolest töödeldavad andmed või peatselt interpreteeritavad avaldised.

[1] Ärme jäta tähelepanuta, et *J. Mc Carthy* oli üks vähestest algoritmide publitseerimise keele *ALGOL 60* esmasteate (ja nähtavasti ka keele enda) autoritest.

Füüsilisel tasemel kujutatakse *Lisp*i andmeid (nii avaldisi kui ka "päris"-andmeid) seotud ahelatena (ik. *linked lists*, vk *связанные списки*). Ahela lüli formaat on järgmine:

Süsteemne info	Viit **CAR**	Viit **CDR**

"Süsteemne info" teavitab, kas tegemist on funktsiooni või tõepoolest andmetega; viimasel juhul, kas tegemist on arvu (kui nii, siis mis tüüpi) või sümbolitega, jne.

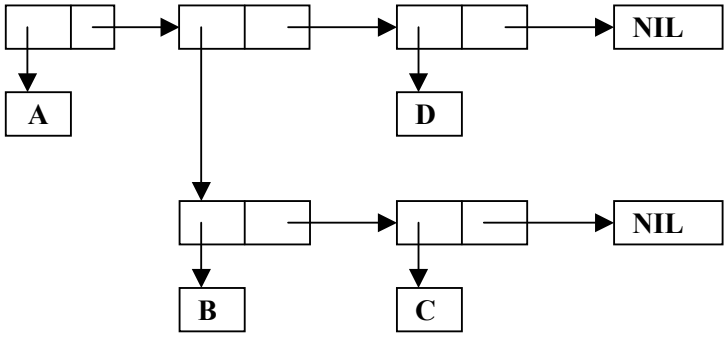
Viit *CAR* ("ajalooline" nimetus *IBM 704* ajast, *Contents of Address Register* ("aadressregistri sisu")); funktsiooni puhul on see viit funktsiooni nimele, andmete puhul − (alam)ahela esimesele elemendile.

Viit *CDR* ("ajalooline" nimetus *IBM 704* ajast, *Contents of Decrement Register* ("indeksregistri sisu")); funktsiooni puhul on see viit esimesele argumendile (mis võib olla funktsioon) või − kui tegemist on interpretaatori arvates andmetega − siis järgmisele andmeelemendile (või mõlemil juhul tühiviit − *NIL*).

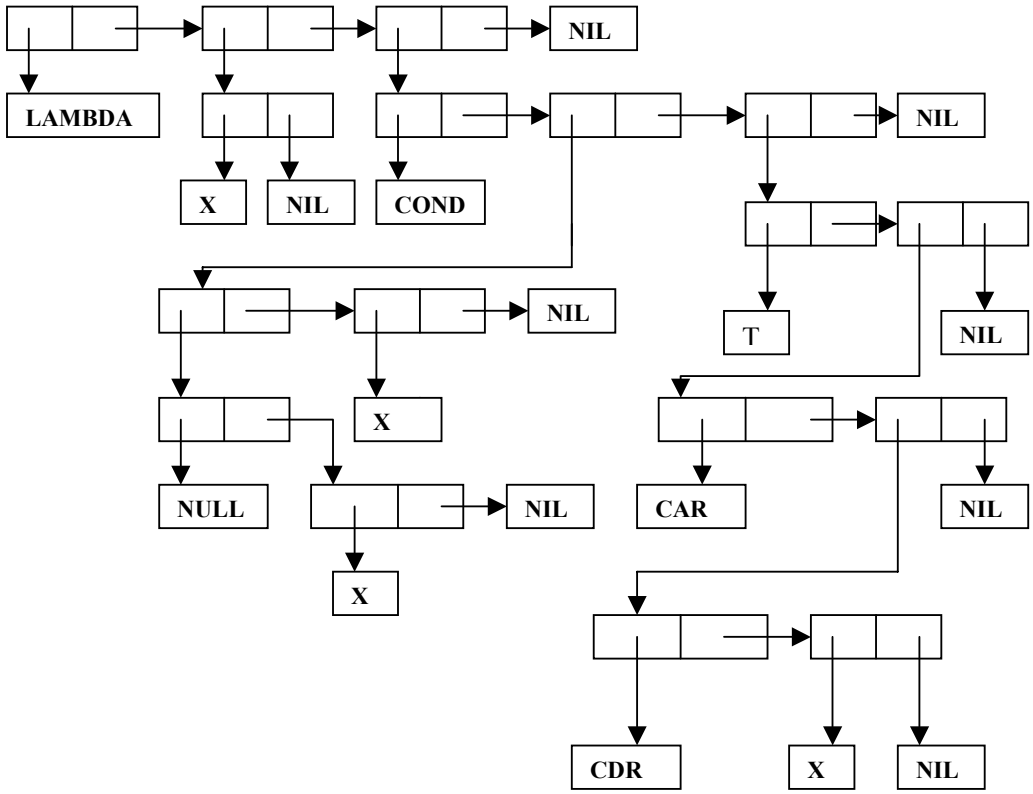
Toome esimese näite "päris"-andmete kujutamisest (vt. [44], lk. 457): ahel on kirjeldatud kui **(A(B C)D)**. Vastav andmestruktuur on kujutatud joonisel 5.3.1a. Teine näide on samuti laenatud *T. Pratt*ilt [44, lk. 464] illustreerimaks programmi (*resp.* avaldise või funktsiooni) kujutamist. Programmi tekst on järgmine:

```
(LAMBDA(X)(COND((NULL X)X)
(T(CAR(CDR X)))))
```

ning ta esitust illustreerib joonis 5.3.1b.

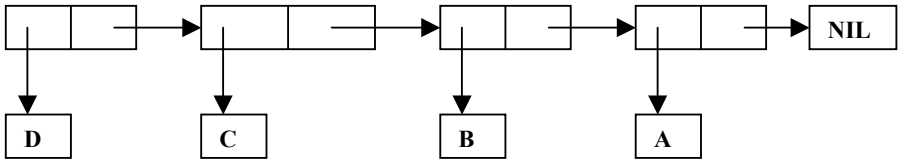


Joonis 5.3.1a. Andmestruktuur A(B C)D.



Joonis 5.3.1b. Programmi andmestruktuur.

Ka näiteprogrammi(d) laename *T. Pratt*ilt. Joonisel 5.3.1a kujutatud struktuure teisendavad lineaarseks ahelaks (vt. Joonis 5.3.1c) kaks programmi; esimene neist on iteratiivne (mis ei vasta *Lisp*i ideoloogiale, ent on tehtav) ja teine kasutab *Lisp*ile loomuomast rekursiooni; resultaat on ahel (D C B A).



Joonis 5.3.1c. Resultaatstruktuur.

Iteratiivne programm:

```
DEFINE(((LISTATOMS(LAMBDA(X)(PROG(RES)
LOOP(COND(NULL X)RETURN RES))
      ((ATOM(CAR X))(SETQ RES(CONS(CAR X)RES)))
      (T(SETQ RES(APPEND(LISTATOMS(CAR X))RES))))
      (SETQ X(CDR X))
      (GO LOOP)
      )))
      ))
```

Kui kasutame interaktiivset režiimi, siis võib seanss jatkuda *Pratt*i järgi nii:

```
LISTATOMS((A(B C) D E))
LISTATOMS(())
LISTATOMS(((A B)((C)D)E))
FIN
```

Sama töö (lineaarse ahela moodustamise) teeb järgmine rekursiivne programm:

```
DEF (LISTATOMS(LAMBDA(X)
      (COND(NULL X) NIL)
            ((ATOM(CAR X))(CONS(CAR X)(LISTATOMS(CDR X))))
            (T(APPEND(LISTATOMS(CAR X))(LISTATOMS(CDR X))))
      )))
```

Andmed.

Andmeid on kolme "tüüpi": *aatom* (tekst), *arv* (*int* või *real*, tüübideklaratsioone pole ning eritüübiliste arvude vahel võib sooritada tehteid − interpretaator hoolitseb vajadusel teisenduste eest) ja *ahel*.

Aatomid on harjumuspärases mõttes *identifikaatorid* ja nad on organiseeritud tabelisse (võimalik, et kujutatud ka ahelana) ning tabeli iga element on paar *<nimi*, *omadus>*; niisiis kõik samanimelised aatomid eri ahelates viitavad ühele ja samale unikaalsele objektile. "Omadus" (ik. *attribute* või *property*, vk. *свойство*) on struktureeritud väli, mis koosneb *deskriptori*st ja väärtusest. Kui identifikaator tähistab muutujat või konstanti, siis "väärtus" ongi ta väärtus, kui funktsiooni nime, siis omaduse *EXPR* nime ja viida funktsiooni kirjelduse, parameetrite jmt ahelale, kui märgendit, siis väärtus on viit märgendatud operaatorile jne.

Ahelate näiteiks sobivad selle peatüki joonised; kui ahelal on *nimi*, siis on ta sellenimelise aatomi *omadus*, kusjuures ahelalülide *car*-väljad esitavad sidet identifikaatorite tabeliga.

Identifikaatorite tabel on programmi täitmise ajal reeglina modifitseerimise objekt: muutujate väärtused muutuvad, ning tabel on avatud uutele kirjetele (muutujatele, konstantidele ja funktsioonidele). Seejuures on programmi täitmise käigus vabad käed süsteemsete primitiivide ülekirjutamiseks, näiteks võib anda mingil hetkel uue sisu funktsioonile *PLUS* (mis vaikimisi arvutab argumentide summa).

Ja, nagu tõdeb *T. Pratt* [44, lk. 465]: "…vajaduse korral saab andmete ahela muuta programmiahelaks ja vastupidi".

Operaatorid[1]**.**

Tähtsaimad on **ahelatega opereerimise** vahendid. Mõnedega neist oleme juba (põgusalt) tuttavad, ent korrakem siin tähtsamad (sj. tuntud) primitiivid üle.

- *CAR*-funktsiooni resultaat on viit ahelalüli esimeselt väljalt (tavaliselt viitab see identifikaatorite tabelisse);
- *CDR*-funktsiooni väärtus on viit järgmise alamahela algusele (või *NIL*);
- *CONS*(*a b*) loob uue lüli ahelasse, kusjuures *CAR*- ja *CDR*-väljade väärtusteks saavad *a* ja *b*, funktsiooni väärtuseks on tavaliselt viit vastloodud lülile, ent kui *b* on ahel, siis äsjaloodud lüli pannakse tolle ahela algusse ning tagastatakse uue ahela algusaadress. Ülalesitatud kolme primitiiviga saab luua suvalise uue ahela. Näiteks (`CONS(A(CONS B(CONS C NIL))` moodustab ahela aatomitest A, B ja C
- *LIST* on mõeldud asendama pikki *CONS*-konstruktsioonide jadasid − argumentide arv pole piiratud ning neist moodustatakse ahel. *QUOTE*[2] võimaldab suvalist nimega ahelat või aatomit käsitleda literaalina. Näiteks, kui `L=(B C)` on ahel, siis `(CONS (QUOTE A) L)` resultaat on ahel `(A B C)`;
- *APPEND* liidab kaks ahelat (*konkatenatsioon*), *COPY* kopeerib ahela, *EFFACE* eemaldab osutatud ahelalüli ning *SUBST* võimaldab lülide (*resp.* alamahelate) asendamist.

Identifikaatorite tabeliga (ik. *list of attributes* (*properties*), vk. *список свойств*) tööks on *Lisp*is kirje lisamise (*ATTRIB* ja *DEFLIST*), kustutamise (*REMPROP*[3]) ja lugemise (*GET*) funktsioonid (vt [44], lk. 467). Oletagem, et tabelis on aatom `JOE.` Et lisada ta atribuutide nimistusse paar `AGE, 25` tuleb kirjutada

```
(DEFLIST (((QUOTE JOE) 25))(QUOTE AGE))
```

Seejärel saame Joe vanust küsida järgmiselt:

`(GET (QUOTE JOE)(QUOTE AGE))` või lühemalt, `(GET (` ` JOE)(` ` AGE))`

Ning Joe atribuudi „AGE" saame kustutada nii:

```
(REMPROP (QUOTE JOE)(QUOTE AGE))
```

Omistamine on *Lisp*is ilmutatud kujul suhteliselt erandlik operatsioon, muutujate väärtusi muudetakse tavaliselt funktsioonide argumentidega ja rekursiooniga, ent keele efektiivsuse (kaasajal: töökiiruse) huvides realiseeritud *PROG*-sõna − mis võimaldab oma mõjupiirkonnas „harjumuspäraselt" (so, protseduurselt, sj. eeskätt iteratiivselt) programmeerida − lubab „ilmutatud kujul" omistamist. Selle ülesande täitmiseks on keeles primitiiv *SETQ*; avaldis `(SETQ X VAL)` tähendab (meie arusaamade kohaselt), et *X:= VAL*. Primitiiv *SET* erineb *SETQ*st selle poolest, et see konkreetne aatom, millele omistatakse juba arvutatud väärtus, võib selguda töö käigus.

[1] Tugineme siin *T. Pratt*ile [44, lk. 465-466].
[2] Uuemates realisatsioonides võib *QUOTE* asemel kasutada ampersandi (').
[3] Sufiks *PROP* (või *P*) vihjab terminile *property* (*attribute*).

Tehted.

"Originaal-*Lisp*is" on neli aritmeetikatehet: *PLUS* (+), *DIFFERENCE* (−), *TIMES* (×) ja *DIVIDE* (/). Taaskord *Pratt*: *ALGOL*i *A+B*C* on *Lisp*is
`(PLUS A(TIMES(B C)).`

Loogilistes avaldistes on kasutatavad tehted *AND*, *OR* ja *NOT* ning võrdlustehted *LESSP* (≤), *GREATERP* (≥), *ZEROP* (=0) ja *MINUSP* (<0).

Predikaat *ATOM* väljastab viida aatomile *T*(*rue*), kui *CAR* viitab muutujale või konstandile, *NUMBER* teeb sama, kui viidatakse arvkonstandile, predikaat *NULL* annab tõese väärtuse, kui viit on *NIL*, funktsioonid *EQ* (võrdleb sümbolkuju-aatomeid) ja *EQUAL* (võrdleb arve) viitavad aatomile *T*, kui operandid on identsed. Ning funktsioon *MEMBER* kontrollib, kas ta argument on etteantud ahela elemendiks.

***Lisp* ja TI-programmeerimine (*TI*=TehisIntellekt).**

*Lisp*i teeb *TI*-keeleks eeskätt seik, et nii interaktiivse seansi kui ka "normaalse" lahendamise ajal on võimalik lisada (või genereerida) uusi konstruktsioone.

Interaktiivsel kasutamisel käib töö käsurea abil; viimase nimeks on *Lisp*is *REPL* ("read-eval-print loop") [130, lk. 8]: funktsioon *READ* loeb *S-avaldise* ning teeb seejuures sisuliselt süntaksi analüsaatori (*parser*) tööd, teisendades avaldise seotud ahela(te)ks, *EVAL* interpreteerib toda struktuuri (mõnedes realisatsioonides interpreteerimine tähendab kompileerimist masinkoodi) ning *PRINT* väljastab avaldise väärtuse − kui selleks on aatomi väärtus − või väljastab sümbolkujul resultaadiks oleva uue sümbolavaldise. "*Loop*" tähendab, et süsteem on valmis vastu võtma ja töötlema järgmist *S-avaldist.*

Primitiivi *CONS* abil saab programmi täitmise käigus genereerida uusi "seotud ahelaid" ning neid saab interpreteerida funktsiooni *EVAL* abil.

Dünaamiliste andmestruktuuride kasutamine on *Lisp*ile loomuomane. Möödunud sajandi keskpaiga masinatel nappis alati operatiivmälu. Kui *Lisp* oleks ainult genereerinud oma seotud ahelaid juurde, saanuks mälu üldjuhul otsa "liiga vara" ning seetõttu tuli välja töötada paindlik mälujaotusskeem: vajadusel tuli välja segitada juba ebavajalikeks osutunud andmed (näiteks, ümberdefineeritud funktsiooni "eelmine keha") ja need kustutada.

Asja teeb keeruliseks see, et ühele ja samale mäluväljale võib olla rohkem kui üks viit ning mälu võib reaalselt vabastada ainult siis, kui kõik viidad sellele väljale on "kontrolli all" − neid saab kas ohutult kustutada või asuvad nad vabastatavas mälus. Esimese kõrgtaseme-keelena lahendas tolle ülesande (tuntud kui *prahikoristus*, ik. *garbage collection*) just *Lisp.*

Mälu vabastamine (teisisõnu, andmestruktuuride hävitamine) realiseeritakse (alates *Lisp*ist) reeglina nii, et keele "hävitamisoperaator" (näiteks, *delete*, *erase* vmt.) ei vabasta ise mälu, vaid keele virtuaalmasin fikseerib saadud signaali, ent ei anna "vabastatud" mälu kohe korduvkasutusse; seda teeb virtuaalmasin alles siis, kui vaba mälu on otsas. Miks

nii: suvalisele mäluväljale võib olla mitu aktiivset viita, ja neist ühe desaktiveerimine ei tohi muuta ülejäänud viitasid "rippuvateks" − nii nimetatakse viitu kustutatud andmestruktuuridele. Ning *praht* (ik. *garbage*) on mälu, mida pole taaskasutamisse antud, ent millele pole enam ainsatki aktiivset (so, kustutamata) viita. Asja teeb keeruliseks see, et (võimalik, et) nii rippuvad kui ka aktuaalsed viidad võivad olla nii tegelikult aktiivsetes andmestruktuurides kui ka "prahis".

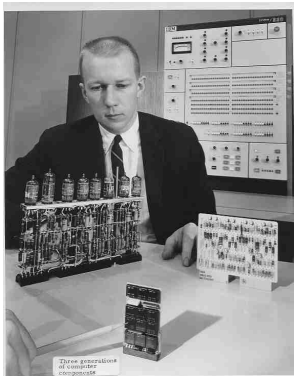
6. ARVUTITE PÕLVKONNAD JA PROGRAMMEERIMIS-KEELED

Üsnagi levinud arvamuse kohaselt on masinorienteeritud keeled jäigalt seotud masinaga, ja protseduurorienteeritud keeled pole sõltuvad arvuti masinkoodist. Seejuures me ei tea palju juhte, kus riistvara oleks konstrueeritud realiseerimaks (innovaatilist) keelt[1], ent samas on igale riistvarauuendusele reageerinud tarkvaratööstus. Ent alati on riistvara arenenud kiiremini kui tarkvara; irooniliselt on tuletatud *Malthus*e seaduse *IT*-versioon: tarkvra areneb aritmeetilises, riistvara aga geomeetrilises progressioonis[2]; pisut teisiti sõnastab sama tendentsi *Reiseri*[3] seadus: "tarkvara muutub aeglasemaks kiiremini kui riistvara muutub kiiremaks" [63, lk. 20].

Seega oleks pisut naiivne arvata, et programmeerimiskeeled võiksid olla tarkvara-arendamises suveräänsed. Pigem on nad üritanud (jatkuvalt) kasutada üha avarduvamaid võimalusi, neid, mida pakub riistvara areng. Ja, et "masinast sõltumatud keeled" (protseduurorienteeritud keeled eeskätt) polegi masinast absoluutselt sõltumatud – kui peame silmas arvutiarhitektuuri *trendi* ja mitte konkreetse arvutimudeli lahendust (*trendi* piirina – veelahkmena – võime teadvustada näiteks üleminekut pesamasinalt baitmsinale või *IBM*-ilt *Intel*ile).

Püüame järgnevas jälgida *arvutipõlvkondade* vaheldumist ja iga uue arvutipõlvkonna programmeerimiskeelte evolutsiooni.

Arvutipõlvkonna mõiste seondub oma ajajärgu tehnilise lahendusega – eeskätt sellega, millele baseerub protsessori ja operatiivmälu realisatsioon. Nende põlvkondade suhtes valitseb kirjanduses üksmeel esimese kolme suhtes, edasi võime täheldada eriarvamusi. Aga noist viimastest teeme juttu hiljem. Sissejuhatuseks reprodutseerime foto (vt. joonis 6a), mis näitab, kuidas on arvutite elementbaas muutunud väiksemaks (samas ei näita see pilt, et sama ruttu on too baas muutunud kiiremaks ja odavamaks).



Joonis 6a**.** Arvutite kolme põlvkonna elementbaasid.

[1] Väheste eranditena märkigem *FORTH*- või *Lisp*-protsessoreid ja *PDP* aparatuurset magasini.
[2] *Malthuse* (*Robert Malthus*, 1766 – 1834) seadus väidab, et inimkond suureneb geomeetrilises, ent toidu hulk aritmeetilises progressioonis.
[3] *Martin Reiser* oli *N. Wirth*i meeskonnas *Oberoni* projekti elluviimisel.

1. Vasakul, lähimal mehele on elektronlambid (esimene põlvkond).
2. Paremal ja suurelt on transistoridele baseeruv masin (1956), transistorid ise on pisut madalamal (teine põlvkond).
3. Kolmas põlvkond (integraallülitused ränikristallidel – kiipidel (*chips)*), 1964, on esiplaanil ja kõige väiksem.

Selle peatüki kirjutamisel oli üpris palju abi *I. Pottossin*i ja *M. Bežanova* õppevahendist [5].

6.1. Arvutite 1. põlvkond (1940 – 1956)

Esimese põlvkonna masinate elementbaas algas elektromagnetilistest releedest ja programmide "kommuteerimisest" spetsiaalsel tahvlil pistikute abil ning jõudis tänu *John v. Neumannile* tänaseni järgitavatel printsiipidel põhineva lahenduseni (kahendsüsteem, programm ja andmed paiknevad ühises operatiivmälus). Tuntava inertsiga (ja seega aeglased) releed asendati *elektronlampidega* (i.k. *Vacuum Tubes*), neist konstrueeriti *CPU* (protsessor) ja ka operatiivmälu (siin oli levinuimaks alternatiiviks magnettrumli kasutamine[1]). Masinad olid äärmiselt kallid, kui üldse, siis toodeti sama marki arvutit suhteliselt väikeses tiraažis ning kõikide markide masinkoodid olid unikaalsed – mistahes margi keel polnud ühildatav teiste markide (sh. sama firma järgmise mudeli) omaga.

Arvuti tegi kalliks elementbaas (elektronlambid), suured mõõtmed (väike masin mahtus heal juhul korvpalliväljakule ja koosnes keskmiselt paari meetrit kõrgustest "kappidest"), võttis palju elektrit ning genereeris märkimisväärselt soojust – see nõudis omakorda samuti kalleid jahutusseadmeid. Lisaks polnud nood arvutid kuigi töökindlad (lihtne arvutus: kui üks elektronlamp töötab keskmiselt 1000 tundi, siis kui tihti tuleb uue masina mingi aja töötamise – näiteks 800 tundi – järel igas minutis vahetada vähemalt üks elektronlamp (kui mälu on nende baasil, so. iga bitt vajab üht lampi ning operatiivmälu maht on 16 KB))? Esimese põlvkonna arvutid olid *pesamasinad.*

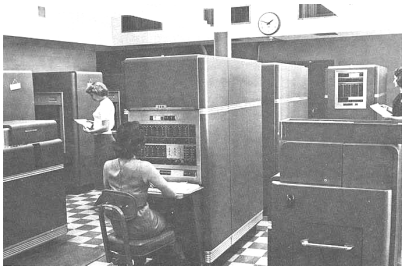
Esimese põlvkonna arvuteid kasutati eranditult mahukateks teadusarvutusteks (eeskätt sõjatööstus, sh. aatom- ja vesinikpommid, kosmosetehnika projekteerimine jmt.)[2] ning kogu töödeldav informatsioon oli numbriline (täis- või reaalarvud). Selliste andmete töötlemisele olid orienteeritud ka tolleaegsed sisend- väljundseademed: sisestada sai sümbolkujul arve ja samuti neid ka väljastada (teisendused tehti – meie mõistes draiveritega – ilma et protsessor sümboltöötlust käskude tasemel toetaks) ning programmeerija suhtles masinaga interaktiivselt juhtimispuldi vahendusel.

[1] Näiteks, Eestisse toodud esimese masina *Ural-1* (üksiti N. Liidu esimese ülikooli-arvutuskeskuse masina) op-mäluks oli magnettrummel (järgmise masina, *Ural-4*, magnettrummel osutus hiljem, peale mahakandmist, ideaalseks mururulliks). Trummel- *vs* lampmälu oli oluliselt odavam, ent sama oluliselt aeglasem lahendus (võrdluseks: kui kiired oleksid meie masinad, kui *RAM* asuks kõvakettal!).

[2] TRÜs programmeeriti näiteks statistilise andmetöötluse pakett, kangaste optimaalse „juurdelõikamise" pakett jmt.

Selle põlvkonna arvutite jaoks oli ainuvõimalik programmeerimiskeel *masinkood*, assembleri välistas sümboltöötluse võimatus. Joonisel 6.1a on kujutatud üks oma aja parimaid 1.põlvkonna arvuteid (töötas muide 10-ndsüsteemis!) – *IBM 650.*

Operatsioonisüsteemi varajase prototüübina kasutati sel ajal *alamprogrammide teeke*, mis toimisid tänu evitatud alamprogramme toetavatele käskude süsteemidele, komplekteerijatele ja paigaldajatele.



Joonis 6.1a. *IBM 650.*

6.2. Arvutite 2. põlvkond (1956 – 1963).

Tehnoloogiliselt ülikeerukad, energiamahukad ja suhteliselt aeglased elektronlambid asendusid poole sajandi eest *transistoridega* – need olid leiutatud küll juba 1947. aastal, ent elektroonikatööstusse (taskuraadiod, hiljem arvutid) jõudsid nad tuntava hilinemisega. Tänu uuele tehnoloogiale paranesid kõik arvuti parameetrid: ta muutus *kiiremaks, odavamaks, energiasäästlikumaks* ja *väiksemaks.*

Infovahetus arvutiga jäi põhiliselt samaks nagu esimese põlvkonna masinatelgi (perfosisend ja -väljund, arvude trükk), ent olulise täiendusega: numbritele lisati (trükitavad) märgid, sisend- väljundseadmed konstrueeriti neid aktsepteeritavatena, ja võimalikuks said assemblerkeeled ning esimesed kõrgtaseme keeled. Keelte *leksika* sõltus oluliselt masina sisend-väljundseadmete võimalustest[1]

Masinad olid endiselt *pesamasinad.* Arvutite operatiivmälu baseerus kas neile samadele transistoridele või siis *ferriitrõngastele* (vt. joonis 2.3.1c), võreldes lamp- või trummelmäluga oli see lahendus oluliselt kiirem, odavam ja energiasäästlikum. Lisandusid *välismäluseadmed* (magnetlintseadmed ja magnettrumlid − sedapuhku välisseadmetena).

Masinkoodi lisandusid sümboltöötluskäsud ning tehted aadressregistritega.

Masinate jõudlus kasvas hüppeliselt: endiselt olid prioriteetsed masinate militaarsed rakendused (näiteks, esimene rakendus oli USA aatomienergiatööstuses – ent see on otseselt strateegiline suund).

[1] Oli ka erandeid – näiteks *APL* orienteerus unikaalsele puldikirjutusmasinale, kus sai sisestada näiteks kreeka tähestiku tähti, aga ka seninähtamatuid sümboleid (näiteks,∇, $\leftarrow$, $\rightarrow$ jm.).

Programmeerija jaoks oli uus tehnoloogia mõneti ebameeldiv: üldjuhul kaotasid nad juurdepääsu masinale (sellega suhtlesid eriväljaõppege operaatorid nn. puldikirjutusmasina – operaatorpuldi vahendusel, juhtimispult jäi arvutiinseneridele), ent teisest küljest – nende tööd lihtsustasid *translaatorid* ja *standardprogrammide teegid.* Masin töötas *üheprogramses pakettrežiimis.*

Niisiis, keeled:

- ***FORTRAN***: esimene version novembris 1954, *FORTRAN I* oktoobris 1956, *FORTRAN II* 1957, *FORTRAN III* 1958. a. lõpus, *FORTRAN IV* 1962.
- ***B-O*** 1957, *Flow-Matic* 1958, *COBOL* 1959, *COBOL 61* 1961, *COBOL 61 Extended* 1962
- ***IAL*** 1958, *ALGOL-58* 1958, (selle edasiarendusena *JOVIAL* 1959, *JOVIAL I* 1960 ja *JOVIAL II* 1961), *ALGOL-60* 1960, selle baasil loodi 1963. a. *CPL.*
- ***Lisp*** 1958, *Lisp 1* 1959, *Lisp 1.5* 1962.
- ***APL*** 1960.
- ***SNOBOL*** 1962.

Võime täheldada arvuti abil lahendatavate ülesannete ringi märgatavat laienemist: esimese põlvkonna arvutite ülesandega (mahukad teadusarvutused) jätkasid *FORTRAN* ja *APL* (eeskätt maatriksarvutused) ja ka *ALGOL* (eeskätt küll arvutusalgoritmide publitseerimise keelena). Uuteks valdkondadeks olid majanduslike andmete töötlus (*COBOL*) ja tehisintellekti ülesanded (*Lisp*, mõnevõrra ka *SNOBOL*[1]).

6.3. Arvutite 3. põlvkond (1964 – 1971)

Võimaluse tõusta uuele tasemele andis 1950. aastal tehtud leiutis, siis tegid *Jack Kilby* (*Texas Instrument*) ja *Robert Noyce* (*Fairchild Semiconductors*) esimese *integraalskeemi* e. *kiibi* (i.k. *integrated circuit* e. *chip*): ühele ränikristallile mahutati suur hulk transistore. Tänapäeval tähistatakse kiipe nende suuruse järgi nii:

- *SSI* (*Small-Scale Integration*): ca 100 elektroonilist komponenti ühes kiibis.
- *MSI* (*Medium*(*level*)*-Scale Integration*): 100..3000 komponenti.
- *LSI* (*Largel-Scale Integration*): 3000…100 000 komponenti.
- *VLSI* (*Very Large-Scale Integration*): 100 000…1 miljon komponenti.
- *ULSI* (*Ultra Largel-Scale Integration*): üle miljoni komponendi ühel kiibil.

Kolmanda põlvkonna arvutid baseerusid väikestele või keskmistele kiipidele; neist ehitati nii protsessor kui ka operatiivmälu ning uus tehnoloogia tõi kaasa arvutite uue arhitektuuri: „pesamasinad" tõrjuti "baitmasinate" poolt välja. Programmeerijatele anti oluliselt suurem "mänguruum". Operatiivmälu maht kasvas seni kujuteldamatute megabaitideni (meenutagem, et näiteks korraliku II põlvkonna esindaja – "*Minsk-32*" – maksimaalne operatiivmälu maht oli 64 kilosõna (ümberarvestatult ca 320 kilobaiti). Tuntumate selle

[1] Vt. näiteks [17], [18] või [113].

põlvkonna masinatena mainigem mudeleid *IBM 360 seeria*[1] (mitu erinevat masinamarki) või *PDP 8*-t.

Sisend-väljund-andmekandjate hulgast taandus perfolint; perfokaardid jäid, eeskätt mitmeprogramse pakettrežiimi toetamiseks. Masinasaalis säilisid nii operaatoripult kui ka inseneripult. Välismäluseadmete hulgast kadusid magnettrumlid, jäid jadapöördust võimaldavad magnetlindid (pikkusega ca 700 m) ja uuendusena otsepöördusega magnetkettad (moodmahuga ca 200 MB, neid tuli tõsta umbes pesumasinassuurusesse seadmesse sisse ja sealt välja kahe käega). Üldjuhul töötasid 3. põlvkonna arvutid *ajajaotusrežiimis*: paketiga käivitatud programmid jagasid omavahel arvuti ressursse (eeskätt operatiivmälu ja protsessoriaega), arvestades tavaliselt operaatori sisestatud ülesannete prioriteetidega ja olulise uuendusena võimaldati taas kasutajatele otsesuhtlust masinaga: arvutiga ühendati töökohad (võimalik, et kaugel asuvad, näiteks TRÜ Arvutuskeskuse paljud "kliendid" istusid oma Tõravere kabinettides). Nood töökohad kujutasid endast displeid ja klaviatuuri ning ideaalsel juhul tundis kasutaja ennast sama hästi kui personaalarvuti taga (tõsi küll, viimaseid tol ajal veel polnud). Ja kui seni sai suuremaid andmehulki masinasse sisestada vaid perfokandjatelt, siis nüüd lisandus mugav võimalus "perforeerida" andmeid otse magnetlinti (mõistagi, auke tegemata, ent umbes samasuguse aparaadiga, millega perforeerija oli harjunud), ja muidugi sai andmefaili kirjutada kettale otse töökohal. Väljundi tavavariandiks jäi endiselt laitrükkal (2 × A4 - formaat, *ASCII*-tähestik – tõsi, NL-is ilma väiketähtedeta, neid asendasid kirillitsa suurtähed).

Operatsioonisüsteemide valdkonnas toimus midagi analoogilist riistvara-revolutsiooniga: *IBM* tegi enneolematult suure süsteemi *OS/360*[2]. Ühelt poolt, võimas operatsioonisüsteem hõlbustas oluliselt programmeerija tööd, ent teiselt poolt – tegi võimatuks programmeerimise ilma operatsioonisüsteemi iseärasusi tundmata (ja aktsepteerimata). See süsteem oli silmnähtavalt liiane: sisuliselt samade asjade tegemiseks oli reeglina palju alternatiivseid variante. *IBM*i järgmine versioon (*IBM/370*) oli veelgi võimalusterohkem. Võib tunduda, et võimaluste paljusus soodustab programmeerimist, ent üldjuhul on resultaat vastupidine: mida suurem ja alternatiivirohkem on operatsioonisüsteem, seda tülikam on teda kasutada. Pisut ette vaadates: *Microsofti MS-DOS* oli peaaegu kõigis versioonides "normaalne" (kui mitte meenutada ilmseid ebaõnnestumisi, näiteks versiooni 4.0), ent sama firma *Windows* on ilmselgelt liiane ning loobumine trükitud süsteemsest manuaalist jätab kasutajad *MSDN-Library* (mis on košmaarne, kui infot otsida) meelevalda.

Vaadakem, mis toimus sel ajal protseduurorienteeritud programmeerimiskeelte arenduses.

- *FORTRAN* jätkas: 1966. aastal standardiseeriti *FORTRAN IV* (*ANSI* poolt, *ANSI= American National Standard Institute*).
- *COBOL*ist tehti uuemad versioonid aastatel 1965 ja 1968. *COBOL* ja *COBOL 68 ANS.*

[1] Tõsi, selle seeria esimesed mudelid baseerusid transistoridele, ent järgmised integralskeemidele.
[2] Senini olid süsteemi tugi programmeerijale läbinähtav, minimaalne ja samas piisav, vt. näiteks *Minsk-32* dispetšerit [6].

- Aastal 1964 sünteesiti kolmest täiesti erinevast keelest (*FORTRAN*, *ALGOL* ja *COBOL*) ambitsioonikas *PL\I*[1], Neljandaks "lähtekomponendiks" võeti *OS/360* makrod.
- *ALGOL*i baasil arendati mitmeid uusi keeli: nime jatkajana *Algol-68* (sama aasta detsembris), 1963. aastal disainitud *CPL*ile järgnesid *BCPL* (juuli 1967), ja *B* (1969); 1964. aastal loodi objektorienteeritud keelte esiisaks peetud *Simula I* (1967. a. *Simula 67*), samal aastal kahe keele (lisaks *ALGOL*ile *FORTRAN*) baasil konstrueeritud *BASIC*, 1970. aastal disainis *N. Wirth Pascal*i.
- *SNOBOL 4* tuli 1967. aastal.
- *Lisp*i basil loodi 1968. aastal *LOGO.*
- Uued tulijad olid 1968. aastal *FORTH* ning 1970. aastal *PROLOG.*

6.4. Arvutite 4. põlvkond: 1971 – käesolev aeg

Neljanda põlvkonna[2] algusdaatum võib meid segadusse ajada: see ajastu kestab juba 35 aastat ja esimesed kolm kokku vaheldusid 30 aasta jooksul. Õnneks on see mulje petlik[3]; integralskeemidele põhineva arhitektuuri evolutsioon ülisuurte integralskeemideni on küll olnud sujuv, ent kiire ning toonud kaasa nii (varem mõeldamatuks peetud) gabariitide vähenemise, arvutuskiiruse kasvu kui ka hindade languse (töökindluse tõusust rääkimata). Lisaks on tänu uuele tehnilisele lahendusele osutunud võimalikuks ühelt poolt paljuprotsessoriliste *superarvutite* (näiteks, ameeriklaste *Cray*) loomine, teiselt poolt aga sama hästi (ent üldjuhul rutem) toimivate *mitmetuumaliste* protsessorite leiutamine[4].

Lohutagem ennast: kolmanda põlvkonna parimad masinad maksid rohkem kui mitu *Rolls Royce*'i, mahtusid vaevu kuhugile ära, nende töökindlus jättis soovida ning ühte masinat pidi teenindama terve tipptaseme-spetsialistide meeskond.

Tänapäeval on igal (teisel) tudengil kodus oma võrkulülitatud masin, mille ressursid ületavad omaaegse Eesti parima, TRÜ AK *EC-1060*, saadud aastal 1982, omi iga parameetri järgi (*EC-1060*: operatiivmälu oli 8 MB, kettamälu ca 1,5 GB. kiirus 1 miljon operatsiooni sekundis, võimsus 80 KW, vajalik pind 200 m^2: seal paiknesid järgmised kapid: protsessor, kanal, 2 operatiivmälu jaoks, 6 kettakappi ja 8 magnetlintseadmete kap-

[1] Kurioosumina seadustati see keel N. Liidus *Automatiseeritud Juhtimissüsteemide* realiseerimise keeleks, sj. polnud N. Liidul ei legaalseid *IBM*-arvuteid ega ka nende tarkvara, sh. *OS*i ja *PL\I* translaatorit..

[2] Viienda põlvkonnaga on lood pisut kummalised: esimestena kasutasid seda terminit jaapanlased, ent ei defineerinud masina tehnilist lahendust, vaid "rääkisid" üldsõnaliselt selle põlvkonna masinate tarkvarast: prevaleerima hakkavat tehisintellekti süsteemid, hääletuvastus, suhtlemine masinaga loomulikus keeles – sj. suuliselt, andmebaaside asemel teadmiste baasid jne. Me ei tea, kas see projekt oli tõsiselt mõeldud või oli provokatsioon, ent põlvkondadest tänapäeval enam tõsiselt ei räägita.

[3] Esimene suur integraalskeem – *Intel4004* – leiutati 1971. aastal. Esimese „kodutarbija-arvuti" evitas *IBM* 1981, aastal. 1984. aastal konstrueeris (koostöös firmaga *Macintosh Microprocessors*) firma *Apple* esimese kaasaegse lauaarvuti.

[4] Mõneti paradoksaalne on tõik, et suurimat paralleelprotsessingut võimaldavad tänapäeval videokaartide protsessorid (kuni 24 *tuuma* (=paralleelselt töötavat protsessorit)), mis võimaldavad ujukoma-arvutusi seninähtamatu kiirusega (*ujupunkt* sellepärast, et ekraanikoordinaatide teisendamise põhitehe on jagamine) – moodsad teadusarvutussüsteemid kasutavad just noid üllatavaid võimalusi, so. videokaartide protsessorit.

pi, lisaks topeltkomplekt esmaseid sisend-väljundseadmeid − laitrükkalid ja perfokaartide sisestamise ja väljundperforeerimise "kappe". "Kapp" pole eufemism.) [106].

Viimase kolmandiksajandi jooksul toimunud programmeerimiskeelte geneesi ja evolutsiooni püüame esitada jooniste abil (tuginedes [33]). Lugejat tuleb hoiatada ennatlike järelduste tegemise eest: ei keelte põlvnemine ega isegi sugulus pole nii ühesed mõisted, nagu noilt joonistelt järeldada võib (või võiks). Tavaliselt loetakse suguluse tuvastamise põhjuseks keelte ajaline järgnevus ja mingi (välise) pildi järjepidevus – ent viimane on eeskätt keele süntaksi, ja vähem semantika küsimus. Näiteks, *C*-keelt peetakse *ALGOL*-60 "lähisugulaseks" mõningate leksikaliste mööndustega (näiteks, *ALGOL*i operaatorsulud **begin…end** on asendatud sulupaariga **{…}**, ja ei pöörata tähelepanu tõigale, et esimesel juhul markeerivad nood sulud *plokki plokkstruktuuriga keeles*, ent *C* semantiline struktuur on pigem sugulane *FORTRAN*ile kui *ALGOL*ile. Ja teiselt poolt, alates *C* tulekust järgitakse reeglina süntaktiliste konstruktsioonide kujutamisel *C* leksikat ja süntaksit – ent see välise kirjapildi sarnasus ei pruugi tähendada sarnaste konstruktsioonide sama semantikat. Just samuti, nagu ladina tähestiku kasutamine ei tähenda, et seda kasutav keel oleks germaani või romaani oma või kirillitsa kasutamine, et keel on slaavi keel.

Järgnevatel joonistel (6.5a….6.5e) on püütud illustreerida programmeerimiskeelte geneesi, võttes aluseks *Éric Lévenéz*i graafi (vt. [34]). Neil joonistel tähistab märk "•" keelt, mis on antud pildi jaoks "võõras". Ülejäänud on nonde võõraste järeltulijad ja omavahel sugulased.

6.5. Kronoloogia

Tuginedes *Eric Lebherz*ile [33] esitame kronoloogilise loetelu[1] neist protseduurorienteeritud keeltest, mis said järgnevatele (aga neid on tuhandeid) eeskujuks või on tänaseks laialdaselt kasutusel. Valik pole mõistagi päriselt objektiivne, vaid kajastab paratamatult *Eric Lebherz*i eelistusi; temalt pärinevad ka lühitutvustused.

1957 *FORTRAN* (*John Backus*, alustas 1954, *IBM* 704)

1958 *ALGOL* (üks kolmest mõjukaimast *FORTRAN*i ja *Lisp*i kõrval)

1959 *Lisp* (*John McCarthy*, *MIT*[2], prefiks-stiili ja funktsionaalse programmeerimise rajaja. Vanuselt teine tänapäeval kasutatav keel; peamine rakendusvaldkond on tehisintellekti ülesannete programmeerimine.)

1960 *COBOL* (*COmmon Business Oriented Language*, *CODASYL*i toode. *Dijkstra*: „selle õpetamist tuleks käsitada kriminaalkuriteona".)

1962 *SIMULA* (*Dahl* ja *Nygaard*, Oslo. Esimene objektorienteeritud keel.)

1964 *PL\I*[3] (sel aastal ilmus teade keele esimese variandi *NPL* kohta; *PL\I* loodi aastatel 1963..1966 firmas *IBM*. Universaalne üldotstarbeline keel, mis sobib mh. ka süs-

[1] Mõned keeled jätame välja, näiteks *OPSS*, *CSP*, *FP*, *dBASE 11* jt.
[2] *Massachusetts Institute of Technology* ─ Massachusettsi Tehnoloogiainstituut, Cambridge.
[3] *Lebherz*i nimekirjas teda pole, ent keel polnud oma ajas sugugi väheoluline.

teemprogrammeerimiseks. *FORTRAN*i, *COBOL*it ja *ALGOL-60* võib vaadelda kui *PL\I* kitsaid alamhulki. Kasutada saab *OS*i makrosid.

1964 *BASIC* („tõsiseltvõetav mänguasi")

1966 *ISWIM* (λ-arvutuste funktsionaalne keel. *ISWIM=If you See What I Mean*)

1970 *PROLOG* (*PROgrammation en LOGique*: tehisintellektile ja teadmiste baasidele orienteeritud keel)

1972 *C* (*Kernighan* ja *Ritchie*, *UNIX* ja *PDP-11*. „Keel, mis kombineerib assembleri kogu elegantsi ja võimsuse assembleri selguse ja loetavusega")

1975 *Pascal* (*Niklaus Wirth*, loodud pedagoogilistel eesmärkidel, ent levis ka kui „päris"-keel. *Kernighan* (*C*) mainis, et ta sobib ainult pisitööde, ja mitte süsteemiprogrammide tegemiseks.)

1975 *Scheme* (*Lisp*i elegantne dialekt; Muuhulgas, kõik *integer*-arvud on ratsionaalarvud ja reaalarvud on kompleksarvud.)

1983 *SMALLTALK-80* (*Alan Kay*, *Xerox PARC*. *SIMULA 67* kontseptsiooni järgiv süsteem, lihtne süntaks, rõhk graafilisele displeile ja „teadete vahetamisele", objektorienteeritud keel. Esimene, mis toetas menüüsid ja hiirt.)

1983 *Ada* (võimas universaalne keel, orienteeritud paralleelprotsessingule ja tööks reaalajas, evib liideseid teiste keelte ja nende standardprogrammide teekidega ja

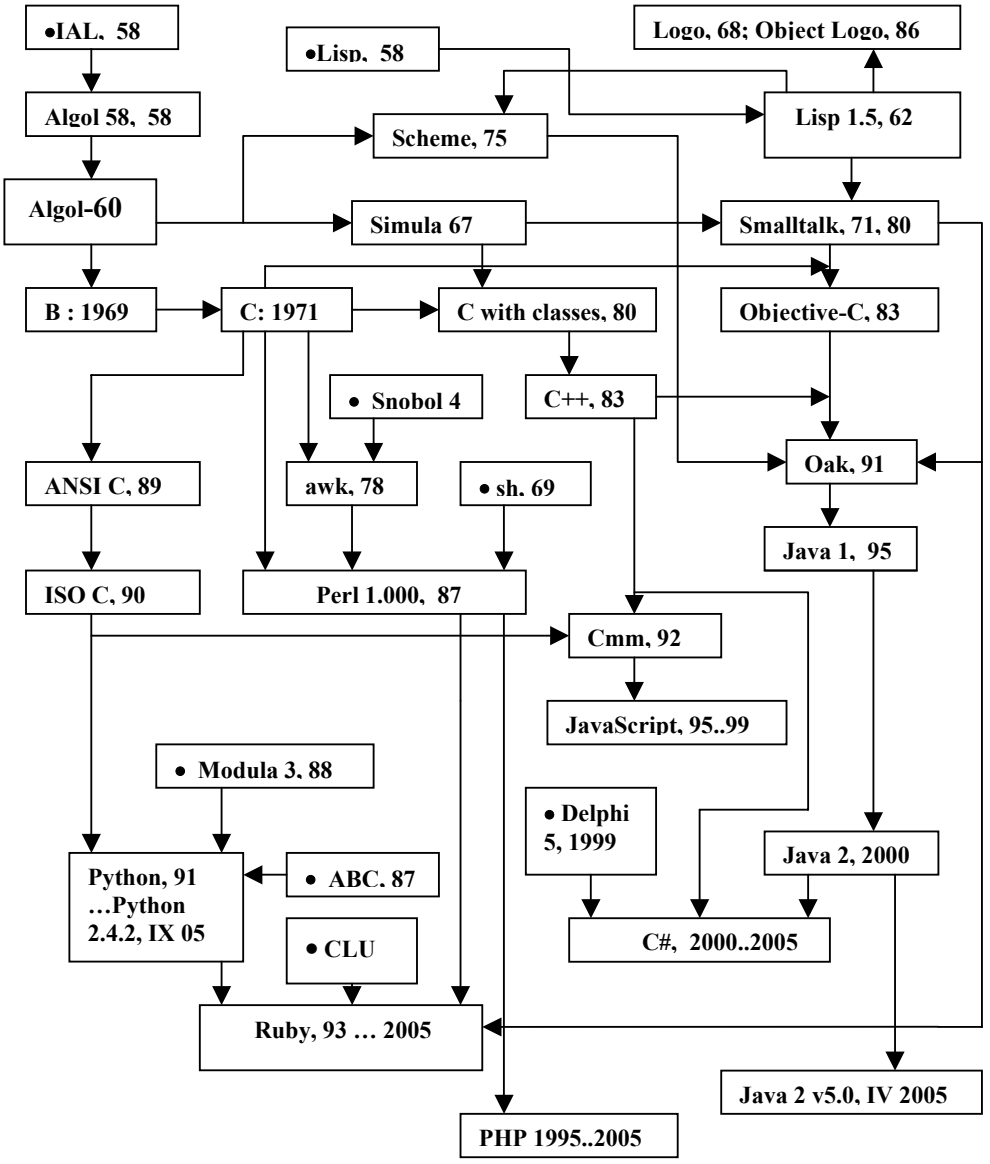
Ada Byron

nime sai keel *Ada Byron'i*, *Lady Lovelace*'i (10. 12. 1815 ─ 27. 11. 1852), *Babbage'*i masinale programme kirjutanu auks).

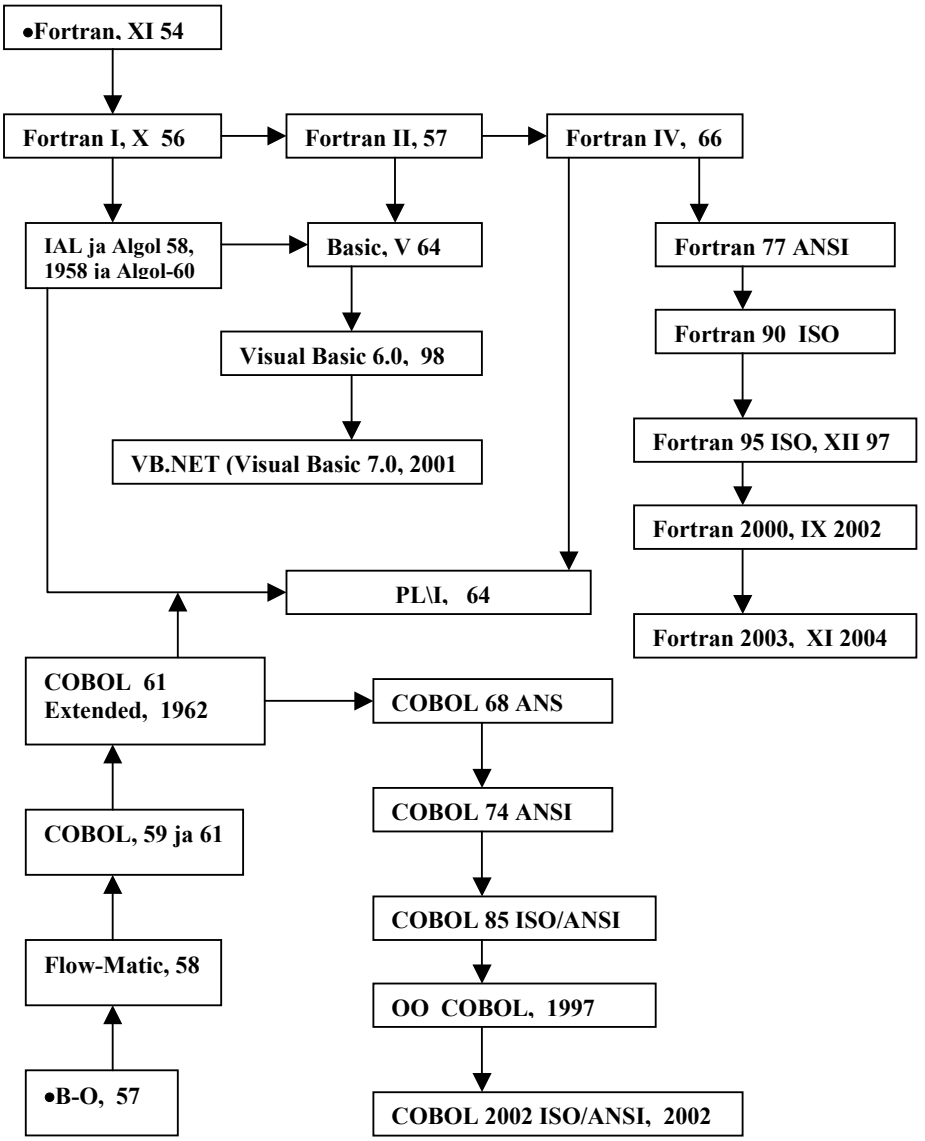
1986 *C++* (*C* objektorienteeritud versioon. Võimas ja kiire, ent raskestiõpitav. Autor on *Bjarne Stroustrup*. Realisatsioone: *Microsoft SDK Visual C++* ja vabavaralised *Borland Builder*i koosseisu kuuluv kompilaator 5.5 ning *GNU DJGPP*).

1988 *OBERON* (*Niklaus Wirth* ja *J. Gutknecht*. *OBERON* kujutab endast integreeritud tarkvarakeskkonda ühe-kasutaja-tööjaamale, keeleliselt järgib *Pascali/Modula* traditsiooni; *Oberon 2* on keele objektorienteeritud versioon.)

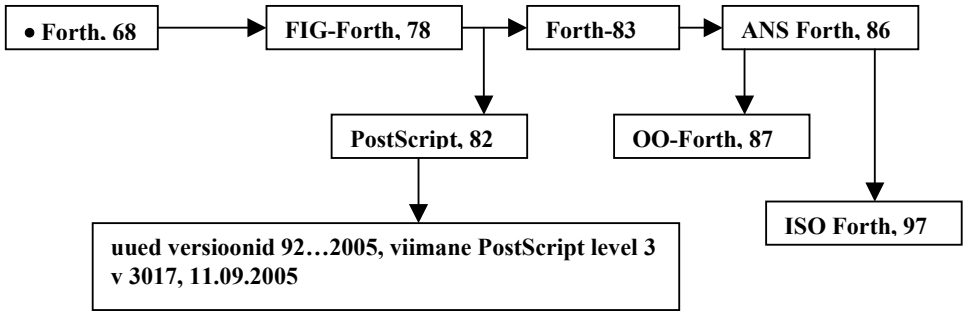
1990 *Haskell* (puhas funktsionaalprogrammeerimise keel. Programmide täitmisel asendatakse avaldised nende väärtustega ning programmid on komponeeritud vaid funktsioonidest. Keel baseerub *Lambda*-arvutusel. Nime sai keel loogiku *Haskell B. Curry* (1900 ─ 1982) auks).



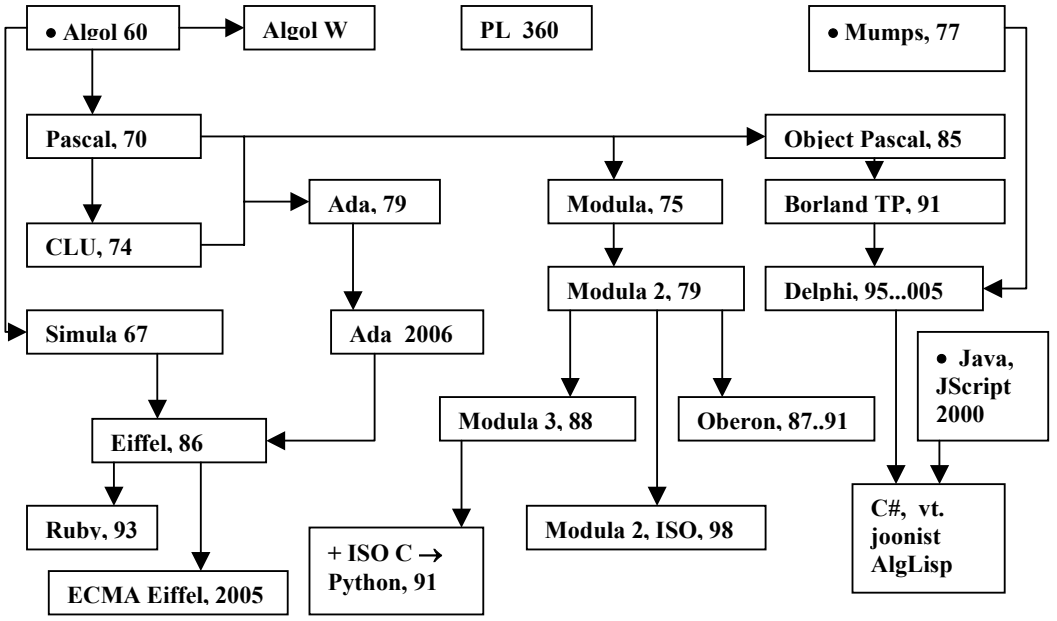
Joonis 6.5a. ALGOLi ja *Lisp*i "sugupuu" objektorienteeritusele orienteeruvalt. Marker "•" tähistab süsteemivälist mõjurit.



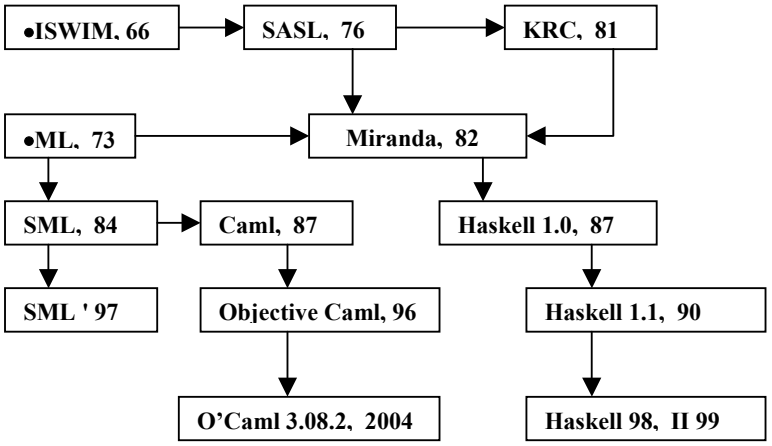
Joonis 6.5b. *FORTRAN* ja *COBOL*. FORTRANi ja *ALGOL*i "sugulus" on mitmeti mõistetav.



Joonis 6.5c. *FORTH*-süsteemi genees.



Joonis 6.5d. *N. Wirth*ist mõjutatud programmeerimiskeelte genees.



Joonis 6.5e. *Haskelli* "sugulaskeeled".

Enamiku selles jaotises mainitud keeltest leiab lugeja ülaltoodud joonistelt; rõhutagem veelkord: programmeerimiskeeled on *tehiskeeled*, nad on konstrueeritud lihtsustamaks kas masintasemel programmeerimist või (väga) suurte meeskondade koostööd. Vale oleks otsida „ideaalset masinastsõltumatut keelt", seda lihtsalt ei ole; olemasolevad keeled sobivad mingitesse kitsamatesse niššidesse, hõlbustamaks oma valdkonna ülesannete programmerimist. Protsessori tasemel tehakse ikkagi kõik masinkoodis ja keele virtuaalmasin määrab, kui mugav on vajalikke asju programmeerida antud keeles. Ja mõistagi, siin on eelis interpreteeritavatel keeltel, kuivõrd kompileeritavad peavad täpselt näitama, kuidas nad midagi teevad, ja genereerima adekvaatse masinkoodi (interpretaatorid võivad kasutada mida iganes, so, suvalist asjakohast antud keelest sõltumatut koodi).

6.6. Programmeerimiskeelte paradigmad

Paradigma on „*teadusloos* püsiv ja üldtunnustatud mõistete, seaduste ja meetodite süsteem, millel rajaneb teadusliku uurimise ja teadusdistsipliinide õpetamise traditsioon. P-sid on nii teadusharuti kui ka teaduslikus maailmapildis tervikuna. Ühe p. piires arenev normaalteadus jõuab lõpuks kriisiseisu, sellele järgneb üleminek uuele p-le." [14, lk. 184].

Seevastu „üldkeeles on see sõna hakanud tähistama mõtteraamistikku või asjade tegemise viisi mis tahes valdkonnas"[108].

Niisiis, selle mõiste teaduslik interpretatsioon pigem välistab kui võimaldab mitme paradigma kooseksisteerimise, ja „üldkeeleline" interpretatsioon sellist kitsendust ei sea. Kui vaadata programmeerimiskeelte paradigmade loetelu (näiteks [109]), siis tuleb nentida, et *paradigma* on selles valdkonnas just „üldkeelelises", hägusas tähenduses. Viidatud nimistus on tervelt 24 ühikut; pisut vanemas loetelus [110] on neid veel ainult 10, sealhulgas:

- Struktuurne programmeerimine („ilma *go to*'ta")
- Imperatiivne programmeerimine (*Algol, FORTRAN, C, Ada* jt.)
- Protseduurne programmeerimine (*vs* funktsionaalne programmeerimine)
- Funktsionaalne programmeerimine (*Lisp, Haskell, Scheme* jt.)
- Objektorienteeritud programmeerimine (*Smalltalk, Java, C++* jt..)
- Komponentorienteeritud programmeerimine (*OLE*)
- Loogiline programmeerimine (*Prolog*)
- Aspektorienteeritud programmeerimine (*AspectJ*)

Tuleb nentida, et *paradigma* on vahel seotud programmeerimisstiiliga, vahel keelte *klassiga*, üsna sageli esindab üks ja sama keel mitut paradigmat korraga ja nood paradigmad evivad tihtipeale omavahelisi ühisosi.

Nii tõdetakse näiteks [109], et enamik *protseduurse programmeerimise* keeli on üksiti *imperatiivsed* ja et mõned objektorienteeritud keeled (näiteks *C++*) kuuluvad ka *protseduursete* hulka. Konkreetsemalt, protseduursete keelte klassi kuuluvad selle allika järgi muuhulgas *Ada, ALGOL, BASIC, C, COBOL, FORTRAN, FORTH,* ja *Modula-2.*

Juhime tähelepanu tõigale, et meie raamatu alguses esitatud klassifikatsiooni kohaselt on kõik noid paradigmasid esindavad kõrgtaseme (so, masinast sõltumatud) keeled *protseduurorienteeritud.*

7. TARKVARA MOBIILSUS

Tuletagem meelde, et algusaegade masinamudelid olid kõik enam-vähem unikaalsed, so. ühildamatud, kusjuures ainus neid ühendav printsiip oli *von Neumanni* masin. Teisisõnu, masinkoode ja assemblereid oli pea samapalju kui erinevaid arvutimudeleid.

FORTRAN universaalse programmeerimis- ja *ALGOL* algoritmide publitseerimiskeelena andsid mõneks ajaks lootust, et on kokku lepitud programmeerijate *lingua franca* standardis (teisisõnu, programmeerijate *pidžin*-keeles). Nii see paraku ei läinud, nii *FORTRAŃ* kui ka *ALGOL* olid rõhutatult orienteeritud nn. teadusarvutustele (ja üldse mitte info-, sj. eriti tekstitöötlusele[1]) Juba enam-vähem samal ajal, kui nood keeled kindlustasid oma standardi-positsioone, ilmusid sootuks muude orientatsioonidega keeled (*COBOL*, *Lisp*, *Snobol*, *APL* jt ning pisut hiljem *C* ja *FORTH*). Iga konkreetse masinatüübi jaoks pidid erinevatest keeltest huvitatud tootjad (või sagedamini, arvutuskeskused) kirjutama translaatorid, ja üleminekul kaasaegsemale, võimsamale masinale tulid need tööd uuesti teha. Mis oli siiski valutum protsess kui kõrgtasemekeelte-eelneva perioodi oma, kus kogu senikirjutatud tarkvara tuli masinkoodis või assembleris uuesti programmeerida. Nii masina tasemel programmeerimine kui ka translaatorite kirjutamine on jõukohane kõrgeltkvalifitseeritud programmeerijaile, ent neid on olnud läbi aegade alati liiga vähe.

Loomulikult otsiti arvitite ja programmeerimise ala juhtivates riikides vastuvõetavaid variante ootamatult tekkinud probleemi lahendamiseks. Sellega tegelesid nii arvutikonstruktorid kui ka senikatmata valdkondade tarbeks projekteeritavate programmeerimiskeelte disainerid[2].

7.1. Insenerlikud lahendused

Inseneride poolt pakuti ühe esimese lahendusena välja protsessori töötamine nn. *ühildamisrežiimil*[3]: pesamasinaid tootva tehase järgmine arvutimudel oli reeglina pikema pesa ja rikkama käskude süsteemiga, ent ühildatavuse huvides püüti säilitada eelmise mudeli käsukoodid (lisati uusi koode ja vanade modifikatsioone) ning vana mudeli kood käivitati spetsiaalses *emuleerimisrežiimis*. Nii olid ühildatavad näiteks masinad *Minsk-22* ja *Minsk-32* (mis võimaldas näiteks "jooksutada" *Malgol*-programme). Selle variandi miinus oli emuleerimisrežiimi tuntavalt väiksem, kolm kuni neli korda aeglasem kiirus.

Üleminek baitmasinatele muutis olukorda. Konstrueeriti protsessori-tasemel tõepoolest ühilduvate masinate seeriad, kus „vana masina" käskude süsteemist sai „uue masina"

[1] Kurioosumina: TRÜ AK-s oli keelatud kasutada trükiseadet (АЦПУ) trüki- ja/või paljundustöödeks – masinkirjutaja (kvalifitseeritud daam) oli sootuks odavam. Samast rubriigist: esimestes arvutiklassides oli tudengitele kõige karmimalt keelatud *mängimine*: klaviatuur maksis õppejõu kuupalga.

[2] Siinkohal eirame mobiilsuse "pisiprobleemi": ühel platvormil (protsessor ja op-süsteem) kirjutatud mingi keele (näiteks *C*) programm ei pruugi olla probleemideta kompileeritav teisel platvormil. Reeglina on vajalikud mõningad „kohendamised", eeskätt *BIOS*iga seotud standardfunktsioonide tõttu (vt. ka [20], lk. 18).

[3] *Ühildatavus* on inglise keeles *compatibility*, vene k. *совместимость*.

käskude süsteemi alamhulk nagu pesamasinate puhulgi, ent emuleerida pole vaja: operandi formaadi – üks, kaks, neli või rohkem baiti – määrab käsu kood, uued võimalused pole lihtsalt vanadele kättesaadavad, ent vanad on probleemideta täidetavad. Nii on 16-bitine *Intel*i kood üldjuhul interpreteeritav 32-bitisel masinal (teoreetiliselt on seda ka 8-bitine kood)[1]; ja et see tihtipeale nii ei ole, pole protsessori, vaid *BIOS*i süü.

Nii olid/on ühildatavad *IBM/360/370*, *Siemens 4004*, NL *EC*-seeria jpt. arvutid, sh. *Intel*i protsessorid.

7.2. Tarkvara-lahendused

Möödunud sajandi kuuekümnendate alguses polnud kellelgi ettekujutust, mis suunas ja kui kiiresti kasvab arvutite kasutusvaldkond (meenutagem, paarkümmend aastat varem arvati, et maailmas pole vaja rohkem kui kümmekonda arvutit – ja ega maailm polekski suutnud suuremat arvu ehitada ja ülal pidada. Koormata ehk küll).

Niisiis, noil aastatel oli loomulik, et üldistati läbilöönud programmeerimiskeelte kontseptsioone ning üritati senistest suundumustest lähtudes sünteesida tõepoolest *universaalne* keel. Seda rolli täitma realiseeriti kaks mõneti vastandlikku projekti: „ümbrik"- või „kestkeel" (ik. *envelope language*, vk. *язык-оболочка*) – *PL\I* ja „tuumkeel" (*core language*, *язык-ядро*) – *Algol-68* (vt. näit. [32], lk. 22).

7.2.1. *PL\I*[2]

Ajal, mil arvuti- ja tarkvaraturul hegemoonitses *IBM* ja meil polnud muud valikut kui *EC*-masinad, kirjutas *Rein Jürgenson* [24, lk. 60]: „PL/I keele (ingl. k. *Programming Language One*; tähistatakse ka PL/1) põhikontseptsioonid töötati välja aastatel 1963… 1966 firma *IBM* arvutite kasutajate assotsiatsiooni poolt,. Materjalid keele esimese variandi *NPL* (*New Programming Language*) kohta avaldati 1964. aastal. Sellest ajast on loodud hulk üksteisest erinevaid keelevariante…PL/I-keel ei tekkinud tühjale kohale. Tema loomisel lähtuti mitme probleemorientatsiooniga (? A.I.) keele, eeskätt *FORTRAN*i ja *COBOL*i kasutamiskogemustest ning võeti arvesse enam-vähem kõik programmeerimisteaduse saavutused. Eesmärgiks seati tõeliselt universaalne keel, mis võimaldaks realiseerida enamikku praktikast pärit ülesannete lahendusalgoritme ja vastaks tänapäeva arvutustehnika võimalustele. PL/I-keele universaalsusmäär on niivõrd kõrge, et selliseid tuntud keeli nagu *FORTRAN, COBOL, ALGOL-60* ja *ALGAMS*, aga ka Eesti NSV-s väljatöötatud *MALGOL* ja *VELGOL*, võib vaadelda kui PL/I-keele kitsaid erijuhte. Täpsemalt väljendudes, PL/I-keelest saab hõlpsasti eraldada alamhulki ehk alamkeeli, mis võimsuse poolest vastavad äsjanimetatud keeltele ning erinevad neist küllalt vähe ka keelekonstruktsioonide poolest. Väga lihtne on näiteks keeles *ALGOL-60* koostatud suvalise

[1] Siit ka lisaseletus *Intel*i arvude esitamise stiilile: neljabaidine sõna *aabbccdd* kujutatakse masinas kui *ddccbbaa.* Ja "teoreetiliselt" seetõttu, et eeskätt *BIOS* ei pruugi enam vana versiooni täismahus toetada.
[2] Vt. ka [13, lk. 305 – 406], [50], [24].

programmi ümberkirjutamine PL/I-keelde. PL/I-keele universaalsus väljendub ka tõigas, et keele vahenditega on võimalik esitada enamikku algoritmidest, mis üldse on realiseeritavad kolmanda põlvkonna arvutite operatsioonisüsteemide abil."

Nentigem, et kuuekümnendate alguses oli see tõepoolest kõikhõlmav projekt; lisama peaksime veel tõiga, et *PL\I* kasutas *OS/360* makrosid – mis muutis *PL*-süsteemi konkurentsitult suurimaks tarkvaraprojektiks.

Ent paradoksaalsel moel ei saanud suurprojekt *PL\I* kunagi populaarseks USAs (kasutati endiselt – sõltuvalt ülesannetest – *FORTRAN*i ja *COBOL*it). Uus keel oli tähtsaimal positsioonil N. Liidus („*AJS"*-i[1] realiseerimise standard), ja oluliselt vähem populaarne mujal Euroopas.

L. B. Wilson ja *R. G. Clark* [61, lk. 35 – 38] seletavad projekti suhteliselt vähest edu järgmisega. *PL\I* autorite kollektiiv lootis luua laiahaardelist, lihtsalt õpitavat, õpetatavat ja kasutatavat, laiendatavat ning alamhulkadeks jaotatavat keelt. Ühelt poolt taotleti masinastsõltumatust, teiselt poolt aga võimaldada juurdepääsu võimalustele, mida suudavad pakkuda assembler ja operatsioonisüsteem – see eeldab, et programmeerijal on täisinformatsioon konstruktsioonide interpreteerimisest. Paraku varjati seda infot "vaikimisiprintsiibiga": programmeerija eest peideti süsteemi tegelik käitumisjoonis ja johtuvalt süsteemi suurusest puudus programmeerijatel lihtsaltsaadav ettekujutus alternatiivsetest võimalustest

*FORTRAN*i ja *COBOL*iga harjunud kasutajad teadsid täpselt, kuidas mingit konstruktsiooni täitmise ajal interpreteeritakse, ent uues situatsioonis kadus kindlus. *PL\I* oli eklektiline. *FORTRAN*ist võeti üle alamprogrammide parameetrite edastamise mehhanism (*parameter-passing*), sõltumatult transleeritavad alamprogrammid, formatiseeritud sisend ja väljund ning ühisväljad ("*COMMON*"). *ALGOL*ist (mis, tuletagem meelde, ei olnud edukas *programmeerimiskeelena*) – plokkstruktuur ja liitoperaatorid (näiteks *if-then-else* puud) ning *COBOL*ilt sisend/väljund-kirjed, *PICTURE*-tüübi ja heterogeensed andmestruktuurid.

Lisaks kolme "lähtekomponendi" võimalustele laenati mõndagi ka mujalt, näiteks ahelate töötlemine ning dünaamilise mäluhalduse mehhanism (so, pisut *Lisp*ilt). Üks originaalsetest uutest võimalustest oli sündmustest-sõltuv juhtimine (*ON*-tingimus).

PL\I-eelsed mõjukad keeled olid sunnitud valima täitmisaegse efektiivsuse (*FORTRAN* ja *COBOL*) või paindlikkuse (*Lisp* ja *SNOBOL*) vahel: mõlemit korraga ei saanud. *PL\I* suutis need alternatiivid peaaegu ühendada, ent sellega kaasnes oluline *keerukuse* kasv.

[1] *AJS – Automatiseeritud Juhtimissüsteem*, originaalis *АСУ – Автоматизированная Система Управления* – oli üks veidramaid asju N. Liidu lõppvaatuses. Mõiste oli defineerimata ja lahti kirjutamata, ja ometi tegelesid sellega käsu korras paljud teadlased ja teadusinstituudid ning sellenimeline aine oli ülikoolide (meie mõistes) IT-õppekavades kohustusliku ainena. Selle sildi all õpetati illegaalselt tavaliselt midagi kasulikku, näiteks andmebaasi- ja infosüsteeme.

Pangem tähele, et *PL\I* oli orienteeritud *IBM*i suurarvutitele (seeriad *360* ja *370* koos operatsioonisüsteemidega vastavalt *OS/360* ja *OS/370*) ning selleks ajaks prevaleerinud töövaldkondadele (mahukad teadusarvutused ja majanduslik andmetöötlus). Selle keskkonna ja ülesannete raames ehk polnudki mõttekas püüda leiutada uusi sama orientatsiooniga keeli. Ent samas olid juba miniarvutid (nagu *PDP*), tulemas olid mikroarvutid ja *IBM*i otsese toeta olid ka valdkonnad, mida programmeeriti näiteks *Lisp*is, *SNOBOL*is või *APL*is. Rääkimata tekstitöötlusest, infosüsteemidest, internetist ja multimeediast; nood asjad olid tol ajal kas marginaalsed või veel tulemata. Mis tähendab, et *PL\I* evis potentsiaali olla "kestaks" (suurele) osale kaasaegsetele rakendustele, ent mitte noile, mis kas juba terendasid või olid veel kaugemas perspektiivis tulemas[1].

PL\I püüdis asendada *FORTRAN*i ja *COBOL*it, ent edutult, kummagi keele kasutajad ei läinud *IBM*iga kaasa. *Wilson* ja *Clark* nendivad muide, et teine "kestkeele" disainimise katse oli USA kaitseministeeriumi *Ada*-projekt, mida kutsuti irooniliselt "kaheksakümnendate *PL\I*-ks".

7.2.2. *Algol-68*

Allpool on tuginetud *L. B. Wilson*ile ja *R. G. Clark*ile [61, lk. 27, 36 ja 337 – 338] ja kahele ainult *Algol-68* käsitlevale monograafiale ([60][2] ja [2]). Keele väljatöötamise usaldas *IFIP*i (*International Federation of Information Processing*) töögrupp 2.1 (*ALGOL*i grupp) rahvusvahelisele meeskonnale, keda juhtis *A. van Wijngaarden* ja kes avaldas 1969. aastal esimese teate uuest keelest; järgmise 6 aasta jooksul töötati projekti kallal ning lõpparuanne avaldati aastal 1975. Keeles endas tehti nende aastate jooksul suhteliselt vähe muudatusi, ent keele formaliseering oli harjumuspärasest *BNF*ist oluliselt detailsem ja raskestiloetavam: kasutati *van Wijngaarden*i formalismi[3] (mille interpreteerimine osutus keele realiseerijate jaoks tõeliseks õudusunenäoks [61, lk. 337]). Venekeelse tõlke saatesõnas kirjutas vene akadeemik, *A. Jeršov*, et keele realiseerijatele ja kasutajatele üheseks juhendmaterjaliks olema pidav aruanne on „kirjutatud kolmes keeles:

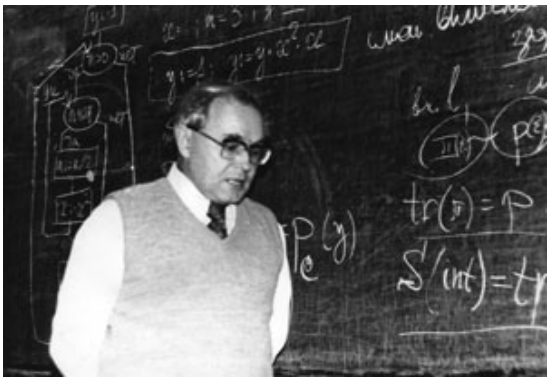
- keeles *ALGOL 68* tema kanoonilisel kujul;
- süntaksi kirjeldamise metakeeles (harjumuspärases, mitte *van Wijngaardeni* versioonis);
- ingliskeelses *proosas*, kus tuleb iseseisvalt eristada fragmente, kus on juttu:
  – rangest keelest;
  – *ALGOL 68*-st tema kanoonilisel kujul;

[1] "Kestkeele" kontseptsiooni sobib kommenteerima *Kozma Prutkovi* tõdemus: "Никто не обнимет необъятного." [http://www.anafor.ru/prutkov/prutkov01.htm], eesti keeles: "ei saa hõlmata hõlmamatut".

[2] viidatud raamat on omapärane: paarituarvulistel lehekülgedel on inglis- ja paarisarvulistel venekeelne (tõlke)tekst.

[3] *van Wijngaarden* oli üks silmapaistvamaist süntaksi ja semantika analüüsi ühendamise mooduseid otsiva koolkonna esindajatest. Süntaksorienteeritud translaatorite töö päädib antud süntaksile baseeruva programmeerimiskeele programmide analüüsi puude genereerimisega ja edasine kompileerimine tuleb "käsitsi" programmeerida iga keele jaoks Kompilaatori täieliku automatiseerimise suund püüdis leida formalismi, kuidas siduda süntaksireeglitega (esitatud näiteks *BNF*is või produktsioonide metakeeles) objektkoodi genereerimist. Paraku on see tänini (madalaimal, so. masinkoodi tasemel) vastuvõetava lahendita probleem.

– dokumendist (so, aruandest) endast;
– metakeelest ja
– sõnastikest.” [60, lk. 5 – 6]



Andrei Jeršov.

Niisiis, kuus aastat tegeles *ALGOL 68* meeskond süntaksi loetava kirjelduse tõlkimisega projekti juhi nõutud (kaasajal sugugi mitte perspektiivituna tunduvale) kujule. On arvatud [61, lk. 27], et see viivitus põhjustaski keele teoreetilise kontseptsiooni üksmeelse tunnustamise taustal tõiga, et kasutatava programeerimiskeelena keelt ei realiseeritudki (üks väheseid erandeid oli Leningradi Ülikooli katse [2] – tõsi, raamatu saatesõnas kirjutavad autorid, et töö on tegelikult pooleli).

Ent kontseptuaalselt oli keel muljetavaldav: püüti ekstraheerida miinimumkomplekt keelelisi vahendeid ning anda võimalused nonde vahendite baasil agregeerida mistahes uus keel.

Tuginedes *L. B. Wilson*ile ja *R. G. Clark*ile [61, lk. 337 – 338] on *Algol-68* iseloomustus järgmine: keel on plokk-struktuuriga (nagu *ALGOL-60*), võimalikud on nii suurte kompaktsete programmide kompileerimine (nagu *ALGOL-60*s) või alamprogrammide eraldi-kompileerimine (nagu *FORTRAN*is); baasandmetüübid olid *int*, *real*, *char*, *bool* ja *string*. Viidatüüpi saab ise defineerida, nagu ka *massiivi* ja (*C*-mõttes) struktuuri. Keel on *rangelt tüpiseeritud* (so, enne suvalise andmevälja või -struktuuri kasutamist peab too olema kirjeldatud koos tüübi näitamisega); objektid on *dünaamilised* (sj., vektori või stringi pikkust saab programmi täitmise ajal muuta). Juhtimisoperaatoritest on olemas *if*, *case*, *for* ja *while*. Keel toetab paralleelprotsessingut, mõistagi koos sünkroniseerimisega. Ja sisend ning väljund on täitmisaegselt üledefineeritavad.

Raske on oletada, millise rolli võinuks *Algol 68* omandada, kui 1969. aasta eelteade olnuks vormistatud lõppdokumendina, jäädes hõlpsaltrealiseeritava *BNF*-kirjelduse juurde. Ent tegelikkus on selline, et ka see keel ei andnud midagi praktilist süsteemse tarkvara mobiilsuse kriisi ületamiseks.

7.2.3. Universaalsed vahekeeled

Niisiis, kumbki äärmuslik täisuniversaalse kõrgtaseme keele projekt ei saavutanud loodetud edu. Paralleelselt nonde projektidega (tegelikult varemgi) otsiti muidki lahendusi. Näiteks, kasutades sel eesmärgil makrogeneraatorit [8], ent sootuks tulemuslikumaks osutus *virtuaalse arvuti* „vahekeele" kui erinevate keelte ja erinevate protsessorite omamoodi *puhvri* kontseptsioon. Allpool käsitletakse põgusalt viit projekti: *UNCOL*i, mis ei andnud pragmaatilist resultaati, ent käivitas idee, nõukogude *ALMO*t (mis oli sisuliselt *UNCOL*i realisatsioon) ning tänapäevaseid, toimivaid *Java* baitkoodi ning *Microsoft*i vahekeelt *CIL* (*Common Intermediate Language*)[1] ja *MONO*-projekti.

7.2.3.1. *UNCOL*

Projekti nime tuleb dešifreerida kui „*UNiversal Computer- Oriented Language*"[2] (universaalne masinorienteeritud keel). Andmeid tolle projekti kohta on alates 1954. aastast ning eesmärgiks oli luua keskkond *m* programmeerimiskeele realiseerimiseks *n* protsessoril, kirjutamata selleks $m \times n$ kompilaatorit. *UNCOL* pidi olema universaalne virtuaalse (tegelikult mitte eksisteeriva) masina *vahekood* (so, mitte lähte- ega ka mitte objektkeel), millesse transleeritakse *m* keelest (selleks on vaja *m* translaatorit) *n* masinkoodi (selleks on vaja *n* kompilaatorit), seega: mitte $m \times n$, vaid $m+n$ translaatorit. Ja iga uue keele jaoks tuleb teha ainult üks translaator *UNCOL*i ja iga uue masina lisandumisel projekti tuleb teha ainult üks kompilaator *UNCOL*ist tolle arvuti masinkoodi (vt.. [97]) .

Hea ülevaate *UNCOL*-kontseptsioonist annab Bath'i Ülikooli 3. aasta tudengi *Nolan Harley* [96] uurimustöö. Tuleb välja, et *UNCOL* polnud pelgalt üks projekt, pigem oli see „püha graal" [97], analoogiliste pürgimuste ühine nimetaja.

Üks esimesi sellesuunalisi projekte oli *SHARE*-komitee (*IBM*i kasutajate selts) projekt, mis oli tuntud kui *Kolme Taseme Kontseptsioon* (*Three Level Concept*) Viidatud tudengitööst pärineb 1958. aasta[3] joonis, seda originaalpilti üritab edastada joonis 7.2.3.1a**.** Nagu võime näha, olid projektiga hõlmatud 7 sisendkeelt, *UNCOL* ja 9 „objektarvutit".

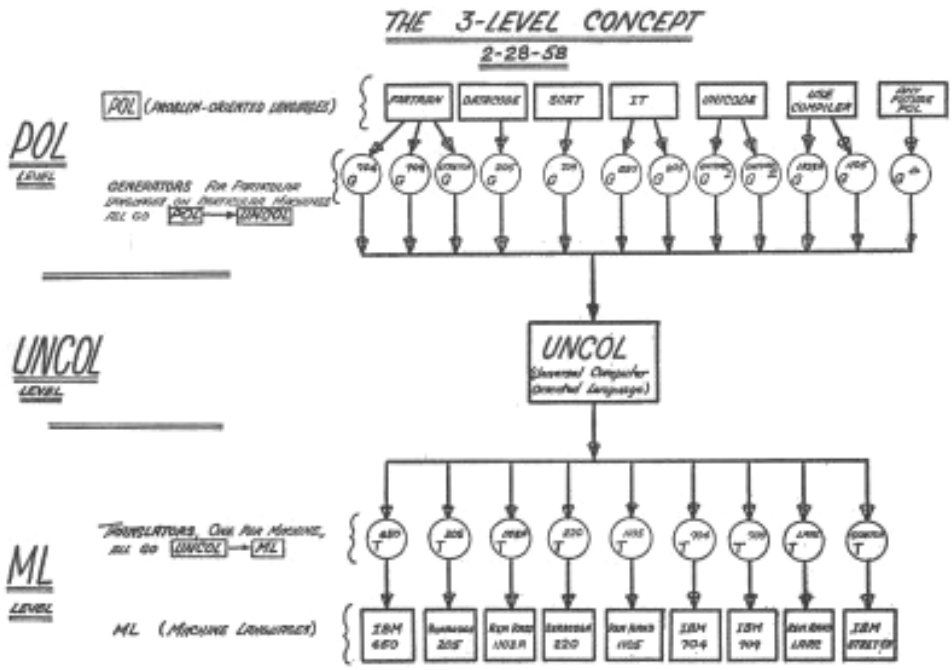
UNCOL-projekti esmasvariant polnud edukas. Teda ei spetsifiteeritud ega ei realiseeritud täismahus kunagi. Põhjuseks oli ilmselt see, et tol ajal oli otsustava tähtsusega kriteerium *efektiivsus* ja seda masinkoodis (või assembleris) kirjutatud programmide suhtes. Iga eraldi võetud keel realiseeriti iga eraldi võetud masina jaoks käsitsi vastuvõetava efektiivsusega (mälukasutus ja kiirus), ent raske oli sama taset säilitada vaheplatvormi kasutades. Meenutagem, et algusaegade arvutistsõltumatud keeled olid sõltumatud teatud piires: osa keelest ei sõltunud riistvarast, osa aga küll.

---

[1] Esialgne nimi oli *MSIL* (*MicroSoft Intrermediate Language*).
[2] Mõnedel andmetel ka "Universal Communication Oriented Language [97].
[3] artikkel "The Problem of Programming Communication with Changing Machines", ajakirjast "The Association for Computing Machinery Journal", august 1958. Wikipedia andmetel pärineb idee juba 1954. aastast [95].

Niisiis, ühelt poolt pidi *UNCOL* olema piisavalt rikas ja paindlik, et kadudeta säilitada lähtekeelte kontseptsioone, ja teiselt poolt olema orienteeritud üksteisest üsnagi palju erinevatele masinatele.

Projekti idee oli mõistagi edumeelne, ent efektiivsuse kadu polnud võimalik (ilmselt objektiivsetel asjaoludel) vältida. Võime nentida, et *UNCOL* tuli liiga vara: kaasajal olid masinad (ja eriti mälu) väga kallid ning programmeerimine masinatega võrreldes odav. Alates 80-ndatest aastatest on täheldatav vastupidine tendents, masinad muutuvad üha odavamateks, programmeerijad aga mitte, ning see seik tähendab, et *UNCOL* (mõtleme selle all ideed, mitte konkreetset esmasrealisatsiooni) on ka ligi 50 aastat hiljem aktuaalne.



Joonis 7.2.3.1a**.** *UNCOL*i esmasversioon.

7.2.3.2. *ALMO*[1]

V. N. Lebedev nendib, et analoogiline projekt – *UNCOL* – oli päevakorras ka 1960-ndate alguse Läänes, ent seda ei teostatud. N. Liidus seevastu projekt realiseeriti. Me ei tea, kas *UNCOL* (mõni selle konkreetne versioon) oli *ALMO* otsene prototüüp (mõtte sellele viib keele ingliskeelne leksika). Akronüüm *ALMO* dešifreeritakse kui *АЛгоритмический Машинно-Ориентированный язык* – „algoritmiline masinorienteeritud keel". Esmateade projektist publitseeriti *S. S. Kamõnini* ja *E. Z. Ljubimski* artiklis, NL Teaduste Aka-

[1] Siin on tuginetud peamiselt järgmistele allikatele: [32, lk. 21] ja [7] .

deemia Toimetistes 1967. aastal, samad mehed toimetasid 1976. aastal ilmunud brošüüri [27]. *Lebedev*i andmetel töötati vastav projekt välja 1966. aastal.

Projektiga olid hõlmatud arvutid *M-220*, *БЭСМ-4*, *БЭСМ*-6, *АСВТ*, *Минск-22, Минск-32, Урал-14, Весна, ЕС ЭВМ* jt. Näeme – kasvõi tuginedes meie teadmistele *Minsk*- ja *EC*-tüüpi arvutite arhitektuuri „totaalsest" erinevusest – et *ALMO* efektiivsuse võime panna kahtluse alla. Turumajanduse tingimustes tõdeti Ameerikas, et projekt pole majanduslikult kasulik, ent N. Liidus ei kehtinud majandus- vaid kehtisid Plaanikomitee seadused.

ALMO oli ühelt poolt projekteeritud universaalseks (transleerimis-)vahekeeleks, ent teiselt poolt iseseisvaks süsteemprogrammeerimise keeleks, so. keeleks, milles oleks hõlpus programmeerida näiteks translaatoreid ja operatsioonisüsteeme suvalise projekti kaasatud arvuti jaoks.

Projekti kaasatud keeltest mainib [27] *ALGOL*i, *FORTRAN*i, *ALGAMS*i, *Omega-60* jt.

Saamaks ettekujutust *UNCOL*i vahenditest[1] vaatleme mõningaid *ALMO* kontseptsioone. Näiteks jagunes *ALMO* virtuaalse masina mälu neljaks klassiks: indeksregistrid (*M*-mälu), üldregistrid (*R*-mälu), operatiivmälu (*V*-mälu) ja välismälu (*EX*-mälu).

Tuletagem meelde, et tolleaegsetel masinatel polnud üldjuhul „kiireid registreid" ja neid „simuleeriti" operatiivmälus (vt. näiteks *Razdan-3* või *Minsk-32*), ja samas olid nad olemas näiteks *EC*-seeria masinatel (so, *IBM*i kloonidel). *ALMO* oli orienteeritud *pesamasinatele* ning üldjuhul nähti ette registrite realiseerimist operatiivmälu pesade baasil (näiteks, indeksregistrite realiseerimist registritega nähti ette *БЭСМ-6,* ent üllatuslikult mitte *EC-1022* puhul (selle masina jaoks sätestas *ALMO* indeksregistrite interpreteerimise operatiivmälus [27], lk. 5))

.Üldjuhul jätab keel vabad käed objektmasina koodi translaatorile otsustamiseks, kas genereerida paralleelprotsessing või ei, ent võimaldab blokeerida alamprogrammi simultaankasutuse (kui lähtekood ei toeta re-enteraablust).

Teksti tasemel info vahetuseks kasutatakse *kanaleid* (*channels*, каналы). Üldjuhul on sisend ja väljund eraldatud, ent võimalikud on ka *konveierid* (ühe protsessi väljund võib olla teise sisendiks).
Andmetüüpe on kuus:

- *N*: arv suvalisel kujul (kehtib vaikimisi);
- *F*: fiks-koma-arv absoluutväärtusega alla ühe;
- *I*: täisarv;
- *B*: kahendkood;
- *M*: modifikaator (semantika on objektmasina koodi generaatori otsustada);
- *Q*: viit.

[1] *UNCOL*ist endast ei õnnestunud allakirjutanul leida ainsatki sisulist spetsifikatsiooni ega näidet.

Masinsõnade (pesade) pikkusatribuudid on järgmised: *H.* – standardne poolsõna, *W.* – sõna ja *D.* – topeltsõna. Noid saab täpsustada, näiteks *Fn* tähendab, et sõna peab mahutama fiks-koma arvu maksimaalse veaga 10^{-n} (näiteks, *F7* tähendab täpsuspiiri 10^{-7}. Ja veel, eeldatakse, et *W*-tüüpi sõna esitusse peavad mahtuma ka *M*- ja *Q*-tüüpi modifikaatorid. *ALMO sõna* võib olla erinevatel objektmasinatel evida suvalist formaati.

Noid instruktsioone kombineeritakse: näiteks, *I9.B31* tähendab, et objektmasina sõna peab mahutama täisarve diapasoonis -10^9 kuni $+10^9$ ja samas bitijada, mille maksimaalne pikkus on 31 kahendkohta.

ALMO-programm on *andmekirjelduste* ja *operaatorite* jada. Viimased võivad olla või on liitoperaatorid – viimasel juhul on operaatorsulud esitatud „sulgudes võtmesõnadega", näiteks *(BEGIN)…(END)*. Liitoperaatorite kehasid nimetatakse *ALMO*-projektis *plokkideks*, ent keel ei järgi plokkstruktuuriga keeli (nagu *ALGOL-60*) kõigis detailides.

ALMO-metakeel järgib tavaliselt traditsioone: indeksmuutuja on kujul *A[indeksid]* ja string *„string"*, ent reservsõnad on aktsentreeritult markerite *(..)* vahel, näiteks *(BEGIN)* ja *(END)* ning „vahelepaigutamine" (originaalis вставка, i.k. *insert*) on metakeeles /X/, kus X on vahelekirjutatav tekst. Ning kommentaarid on eraldatud markerite „*" vahele (NB! algus- ja lõpumarkerid kattuvad). „Tavalised" eraldajad, nagu „.", „," ja „:" on metakeelsed sümbolid (üldse, kasutatavad on 256-sümbolilisest ruumist ladina ja vene alfabeedi tähed (mõlemil nii suur- kui ka väiketähed), numbrid ja nii tava- kui ka matemaatilises kirjapildis kasutatavad sümbolid).

Identifikaatorid võivad koosneda ladina tähestiku tähtedest, araabia numbritest ja alakriipsust (näiteks *_123* või *ABS*) ja ei või alata numbriga.

Võtmesõnu oli 76. Toogem mõned näited:

- *(CONSTANT)* – konstant;
- *(COPY)* – kopeerimine *V*-mälu → *V*-mälu;
- *(CHANNEL)* – kanal;
- *(DVRG)* – hargnemine;
- *(DIV)* – jagamine;
- *(GE)* – suurem-võrdne;
- *(GO TO)* – suunamine;
- *(INCOPY)* – kopeerimine *EX*-mälu → *V*-mälu;
- *(OUTST)* – lugemine magasinist, *POP*;
- *(POW)* – astendamine;
- *(REPLACE)* – vahelepaigutamine;
- *(REF)* – viit;
- *(T)* – tüübiteisendus;
- *(TEXT)* – tekst;
- *(10)* – 10-ndsüsteem (veel on *(2)*, *(4)*, *(8)* ja *(16)*).

Muutujad on kas

- *M-muutujad*, mis paiknevad *M*-mälus ja neid tähistatakse *M.n*, kus *n* on märgita täisarv;
- *R-muutujad*, mis paiknevad *R*-mälus ja neid tähistatakse *R.n*, kus *n* on märgita täisarv. Nende tüüpide näiteid: *M.0, R.12*;
- *lihtmuutujad* paiknevad *V*-mälus. Meie arusaamade kohaselt on nad *vektorid* kujul *nimi*[*indeks*], kus *indeks* on *M*-muutuja, näiteks *A[11B13.]*, *CF[M.13-14]* või *PUF[M.0]*;
- *kaudne* adresseerimine on muutujale viitamine näiteks kujul *[R.0]* või *[SD]* – nurksulgudes on tegelike operandide *aadressid. M*-muutujat selles rollis kasutada ei saa;
- *muutujate algväärtustamine* toimus võtmesõna *(CONSTANT)* abil, näiteks *int*-vektor nimega *NUMBER* nullitakse nii: *I(CONSTANT)NUMBER:0,[1,9]*. Või viitade vektorit *SWITCH* saab defineerida kui *Q(CONSTANT)SWITCH:AB,MI,MA*.

ALMO tehted jagunevad tavapärasteks aritmeetilisteks ja loogilisteks, ja lisanduvad *eri-operatsioonid*, mis seonduvad kas viitade moodustamisega või tüübiteisendustega ilmutatud kujul[1]. Operandideks on muutujad, konstandid ja – mõnevõrra üllatuslikult – *märgendid.* Kui tehte resultaat pole üheselt läbinähtav, siis saab seda spetsifitseerida.

Aritmeetilistest tehetest on ainult liitmine ja lahutamine „ühemärgilised" (+ ja –), ülejäänud on kujul *(tehe)*, näiteks *(MULT), (DIV)* või *(ARCCOS)*. Samamoodi esitatakse ka kõik loogikaoperatsioonid, näiteks *(NOT)*, *(AND)* või *(SHIFT)*.

Omistamisoperaatoril on neli erivormi.

- *lihtne saatmine*: näiteks, 1101,*I*=:*Z* tähistab täisarvulise väärtuse *1101* omistamist muutujale *Z*. Pisut keerulisem näide on *A,B,+,A,B,–,(MULT),=:[C][M.1]* on tavapärasel kujul avaldise $(A+B)\times(A-B)$ väärtuse omistamine massiivi elemendile; massiivi algusaadress on muutuja *C* väärtus;
- *magasini kirjutamine ja magasinist lugemine.* Kirjutamiseks on operatsioon *(INST)*; näiteks magasini rollis olevasse massiivi *S* saab lisada „aknasse" (paljudes assemblerites on kirjutamisoperatsiooni nimeks *PUSH*) nii: *R.11,R.2,(MULT), (INST)=:S[M.15]*. Magasini tipmise elemendi lugemiseks (üksiti ka eemaldamiseks, tavapäraselt *POP*) kasutatakse operatsiooni *(OUTST)*, näiteks magasini rollis olevast massiivist loetakse tipmine element nii: *SS[M.10–1],(OUTST)=:Z1*. Sama resultaadi saaks kahe operaatori abil: *SS[M.10–1],=:Z1* ja *M.10,1M–,M=:M.10*;
- *arvu teisendamine tekstikujule*, näiteks *–252.084,(TEXT)8(DG)2(EXP)=:NA[1]* toimel kirjutatakse tekstivektorisse *NA* järgmised sümbolid: märk „miinus" aadressile *NA[1]*, 252084 aadressidele *NA[2]..NA[7]*, kaks järgmist baiti on sümbolid „0", *NA[10]* on järgu märk + ning kaks järgmist sümbolit on 0 ja 3 (see vastab teisendatava arvu normaliseeritud kujule –0.252084(10)3);
- *tekstikujulise arvu teisendamine sisekujule*: teisendamiskäsk ise on *(VALUE)*. Näide: olgu *M.1* väärtuseks 2 ja *C* väärtuseks 3 ning normaalkuju-arvu sümbolid on

[1] Näiteks, normaalkuju-arvu teisendamine bitikujule: *I(T)B20.B*

vektori *MIS* elemendid indeksitega *4..9*: *+473−2*. Omistamistehtega *MIS[M1+2], (VALUE)C(DG)1(EXP)=:NR* omistatakse muutujale *NR* väärtus *0.473(10)−2*; Ja eelmise „punkti" näite „tagasiteisendus" käib nii: *NA[1],(VALUE)8(DG)2(EXP) =:[R.3][2]*.

Juhtimisoperaatorid

- *Tingimusteta suunamine*: siin on kaks varianti: lihtsuunamine ja naasmisega suunamine (vt. masinkoode ja assemblereid). Lihtvariandi näiteks on toodud [8, lk. 52] kaks varianti: *(GO TO) SIN* annab juhtimise *märgendile SIN*, ent *(GO TO) [R]* annab juhtimise operaatorile, mille aadress on muutuja *R* väärtuseks. Seega, kui *R* väärtus on *SIN*, siis nood suunamised on võrdväärsed. Naasmisega suunamine on näiteks selline: *(GO TO)B(RET)C=:R1* tähendab, et juhtimine antakse märgendile *B* ning muutuja *R.1* väärtuseks saab samas märgendi *C* aadress;
- *Tingimuslik suunamine* on üldkujul *(IF)op₁,op₂,so(GO TO)m*: *op₁* ja *op₂* on võrdlustehte kaks operandi, *so* on operandide spetsifikatsioon[1] ning *m* on kas otsesel või kaudsel kujul esitatud märgend, kuhu antakse suunamine, kui *op₁* □ *op₂=tõene*. Näiteks: *(IF)R.1,1,<(GO TO)M*. Siia rühma on *ALMO* kirjeldajad kandnud ka *tsükli lõpu* operaatorid kujul *(FOR))op₁,op₂,so(GO TO)m(ELSE STEP)n* või *(FOR))op₁,op₂,so(GO TO)m(AND STEP)n* (vt. [7], lk. 55..57).
- *Paralleelprotsessingu* operaatoreid on kaks: *(DVRG)m* ja *(JOINT)m*. Siin on *m* selle protsessi märgend, kus algatatakse või lõpetatakse paralleliseerimine. Algatatud protsess võib omakorda algatada suvalise arvu paralleelseid protsesse. Sünkroniseerimiseks peavad *(DVRG)* ja *(JOINT)*-märgendid olema identsed. Ja *(JOINT)*-järgset operaatorit ei täideta enne kõigi paralleelharude lõpetamist. Ja iga *(DVRG)*-operaatorile peab vastama parajasti üks *(JOINT)*-operaator. Kui objektmasinal pole paralleelprotsessingu võimalusi, siis täidetakse protsessid järjestikku;
- kui objektmasin ei toeta *re-enteraableid mooduleid*, siis *ALMO* saab blokeerida nende simultaankasutamise atribuudiga *(SEQ)*. Piirangu lõppu tähistab võtmesõna *(REL)*;
- objektprogrammi täitmist saab ajutiselt *katkestada* operaatoriga *(STOP)*. Näiteks selleks, et suhelda „füüsilise operaatoriga",so. noore daamiga masina juhtpuldis (meenutagem *UNCOL*i ja *ALMO* juurutamisaega).

Vahetusoperaatorid on infovahetuseks eritüüpi mälude vahel. *ALMO*s olid operaato-rid *(INCOPY)* − välismälust sisemällu (*V*-mälu → *EX*-mälu), *(OUTCOPY)* sisemälust välismällu (*EX*-mälu → *V*-mälu) ja *(COPY)* ülekanneteks *V-mälus*.

Andmevoogude juhtimiseks kasutas *ALMO kanaleid* (*channel*, канал). Eristati andmete sisestamise (eraldi kanal teksti ja eraldi arvude jaoks[2]), väljastamise (samuti eraldi teksti ja arvude jaoks), arvandmete programmide vahelise vahetamise, operatsioonisüsteemiga

[1] Kas < (*LE*), = (*GE*) või > (*NE*).
[2] Eeldati, et arvandmete sisestamisega kaasneb nende teisendamine etteantud sisekujule.

side pidamise ja interpretaatoriga[1] side pidamise kanaleid. Kanalite identifikaatoritena kasutati nende numbreid. Kanalite seostamine konkreetsete seadmetega jäeti translaatorite *ALMO* → *konkreetne masinkood* hoolde.

Sisestamisoperaatorite üldkuju oli järgmine:
s(INDATA)v(CHANNEL)e_1*(NUMBER)*e_2 arvude sisestamiseks ja
(INTEXT)v(CHANNEL)e_1*(NUMBER)*e_2 teksti sisestamiseks, kus
s on sisendandmete formaat;
v on mäluaadress;
e_1 on muutuja, mille väärtus määrab kanali numbri ja
e_2 on avaldis, mille väärtus määrab sisestatavate sõnade arvu[2].

Arvandmete väljastamise operaatori üldkuju oli
s(OUTDATA)v(CHANNEL)e_1*(NUMBER)*e_2,
kus *s, v,* e_1 ja e_2 tähendused on samad nagu sisestamise puhulgi. Teksti väljastamise operaatori üldkuju oli pisut keerulisem:
(OUTTEXT)m:ngt,v(CHANNEL)e_1*(NUMBER)*e_2; lisandusid järgmised määrajad:
m – täisarv, ridade arv leheküljel;
n – täisarv, märkide arv reas;
g – kui on kodeeritud „G", siis trükitakse sümbolit „." kasutades graafik, kui seda ei soovita, jäetakse see parameeter vahele;
t – trükiseadme spetsifikatsioon: *D*, kui rida ei saa tagasi pööratja *DR*, kui ka sümboleid reas saab trükkida ainult vasakult paremale (tagasivõtuta). Kui neid kitsendusi pole, siis seda parameetrit ei kodeerita.

Programmidevaheliseks infovahetuseks olid ette nähtud kanalid numbritega 200..999. Info säilis neis kanalites seni, kuni neid polnud ilmutatud kujul üle kirjutatud. Näiteks,

B(OUTDATA)A[0](CHANNEL)201(NUMBER)1024
B(INDATA)B[0](CHANNEL)201(NUMBER)1024

esimene operaator väljastab 1024 pesa ja teine operaator sisestab nad endale kasutamiseks. Kanalite arv ja neis säilitatavate andmete maht jäi objektmasina koodi kompilaatori hooleks. Süsteemi autorid soovitasid ressursi nappimise korral kanaleid järjestikku ühendada.

Kasutajaprogrammi side süsteemsete programmidega polnud *ALMO*-projektis täpselt fikseeritud, kuivõrd see sõltus olulisel määral objektmasina operatsioonisüsteemist. Raamatus [7, lk. 77] on toodud näide *БЭСМ-6* baasil: kasutades *(OUTTEXT)*-formaadile lähedast esitust, näidatakse kanali numbrina väärtust 1000 – seega ei lähe käsk normaalselt täitmisele, juhtimine antakse operatsioonisüsteemile, mis interpreteerib määrajaid *m* ja *n* omamoodi, lugedes sealt välja vajaliku süsteemse alamprogrammi „numbri" ja saades „väljastatava teksti" väljast kätte parameetrid.

[1] *ALMO* autorite terminoloogias tähistas süsteemne *interpetaator* opsüsteemi seda osa, mis tegeleb teegiprogrammide dünaamilise laadimise ja täitmisega.
[2] Meenutagem, et pesamasinate puhul paigutati ühte masinsõnasse mitu sümbolit.

Näiteprogramm.

ALMO keel on *plokkstruktuuriga*, ligilähedaselt samamoodi nagu näiteks *ALGOL-60.* Alamprogrammide süsteem on üsna harjumuspärane, ühena vähestest omapäradest mainigem, et parameetrite hulgas (või ainsa parameetrina) tuleb näidata märgend, millele antakse juhtimine alamprogrammist väljumisel.

Programmi näitena reprodutseerime arvude *N=1,2,..10* faktoriaalide arvutaja [7, lk. 79]. Kommentaarid on originaalis vene keeles, need esitame siin tõlkes.

```
(BEGIN)
      *N! ARVUTAMISE PROGRAMM*
      I6.(VARIABLE)N,NFAK;
      1,I=:N;   *N ALGVÄÄRTUS*
      1,I=:NFAK;
M:NFAK,N,I(MULT)NFAK(CHANNEL)12(NUMBER)1;
      N,1,I+,I=:N;
      (IF)N,1 1,I<(GO TO)M;
      *ARVUTAMISE LÕPP*
(END)
```

Lõpetuseks. Nagu nähtub Venemaa Teaduste Akadeemia *M. V. Keldõš*i-nimelise Rakendusmatemaatika Instituudi teadusliku koolkonna „Programmeerimine"[1] koduleheküljelt [112], töötati *ALMO* välja just seal, *Mihhail Romanovitš Šura-Bura* juhtimisel . Sealt pärineb järgmine lõik: „Universaalsete (mobiilsete, mitut tüüpi arvutitel funktsioneerivate) kompilaatorite loomiseks töötati Instituudis välja spetsiaalne abstraktsele masinale orienteeritud algoritmiline keel *ALMO.* Teda kasutati nii kompilaatorite kirjutamise vahendi kui ka nonde kompilaatorite objektkeele rollis. Iga uue konkreetse arvutitüübi jaoks tuli vaid teha kompilaator *ALMO*st, kõik ülejäänu toimis automaatselt. Oma suunitluse ja instrumentaariumi poolest oli *ALMO* tunduvalt hilisema keele *C* prototüübiks." Nentigem, viimane lause on üllatav ja intrigeeriv!

7.2.3.3. Vahekokkuvõte

Teeme vahekokkuvõtte. Vajadus mobiilse tarkvara järele oli/on objektiivne. Pesamasinate algusaegadel olid nende mudelid reeglina unikaalsed ja programselt (so, masinkoodi ja assembleri tasemel) ühildamatud. *FORTRAN*i konstrueerimine polnud mitte ainult tõhus vahend programmeerijate töö hõlbustamiseks, see oli ka esimene samm tarkvara mobiilsuse probleemi lahendamise suunas. Ent teame, et *FORTRAN* sobis (ja sobib tänini) eeskätt nn. teadusarvutuste programmeerimiseks, arvutite rakendusvaldkonnad olid juba *FORTRAN*i tuleku ajal laiemad ja see „ring" laieneb tänini ning ükski optimist (või pessimist?) ei saa näidata, kus on arvutite kasutatavuse piirid. See laienemine tingis (taas objektiivselt) vajaduse üha uute, spetsifitseeritud programmeerimiskeelte järele. Ja teisalt, kuivõrd ükski kõrgtaseme keel pole absoluutselt „masinsõltumatu", siis riistvara areng põhjustas samuti vajaduse uute keelte disainimiseks.

[1] Научная школа «Программирование» Института прикладной математики (ИПМ) им. М.В. Кельдыша РАН.

Niisiis, masinatüüpide arv kasvas kiiresti ja programmeerimiskeelte arv veelgi kiiremini. Nägime, et katsed luua üks ja ainus tõepoolest universaalne protseduurorienteeritud keel polnud edukad.

*UNCOL*i idee oli põhimõtteliselt parem, see oli kompromiss niikuinii suureneva arvu arvutimudelite (*m* tükki) ja programmeerimiskeelte (*n* tükki)vahel: $m \times n$ translaatori asemel piisas $m+n$ translaatorist. Ent me teame, et *UNCOL* ise jäi idee tasemele ning *ALMO* küll realiseeriti, ent tegelikult polnud ta kuigi populaarne ja tema reaalsest kasutamisest puuduvad sisuliselt igasugused andmed – kui silmas pidada (teadus)publikatsioone või juhtivate arvutuskeskuste rakendusprojekte (näiteks, translaatorite tegemise vallas). Tundub, et ebaõnnestumise põhjustas tolleaegne arvutiarhitektuur: pesamasinad, ja arvutite vähene jõudlus. Erinevat tüüpi pesamasinate realisatsiooni (käskude süsteem, välisseadmetega suhtlemine jne.) ühisosa oli tõepoolest marginaalne.

Arvatavasti seetõttu *UNCOL* (*ALMO*) ei püüdnudki leida kompromissi erinevate masinkoodide või assemblerite vahel, vaid oli lähemal pigem universaalse kõrgtaseme keele projektile[1], mis sest, et osundamistega masintasemel realisatsioonile.

Ent idee – üldistada masinkoodi ja konstrueerida vastuvõetav (toimiv) kompromisskood oli endiselt hea ning meie ajal – mikroprotsessorite ajal – on see idee aktsepteeritaval kujul tegelikult realiseeritud. Ja kui me tavaliselt (õigusega, seejuures) taunime monopoolses seisundis suurtootjaid ja oleme tootjate paljususe poolt, siis mikroprotsessorite puhul on meie valdkond *Intel*i standardite taolisest seisundist üheselt võitnud. See, et *Intel*i kõrval on turul mõned muudki „tegijad" (*Apple'i* kasutatud *MacIntosh*, *Motorola PowerPC* jt.), pole mõistagi paha, situatsioon on oluliselt parem kui pesamasinate ajal unistadagi võis. *UNCOL*i tänapäevane idee on läinud poolprotseduurse vahekeele tasemelt assembleri tasemele. Erinevatele protsessoritele tuginevaid süsteeme nimetatakse selles kontekstis *platvormideks* ning mobiilsuse idee seisneb platvormideülese (ühisosale tuginеva) abstraktse masina assembleri kasutamises translaatori (vahe)objektkoodina. Toda koodi interpreteeritakse programmi lahendamiseks juba konkreetsele protsessorile programmeeritud interpretaatoriga (kusjuures interpretaatori väljundiks võib olla *.exe*-fail) konkreetse operatsioonisüsteemi keskkonnas.

7.2.3.4. *Java* baitkood

Interneti-allikas [133] kirjutab baitkoodi (*byte-code*) lahti vabas tõlkes järgmiselt: see on kompileeritult jooksutatav virtuaal- ja mitte reaalses arvutis. Ent lähtekood võib olla seejuures kompileeritud suvalisel platvormil ja „joosta" suvalisel platvormil[2], kui seda toetab vastav virtuaalarvuti. Baitkood on Java-programmide kompileeritud resultaatkuju. Baitkoodi interpreteerib[3] *Java* virtuaalmasin (*JVM*); seda tüüpi failide nime laiendiks on tavaliselt *.class*.

---

[1] Objektiivselt võttes polnudki tol ajal madalama taseme üldistus võimalik. idee oli omast ajast ees; meenutagem analoogiliste näidetena kasvõi *Babbage'*i masinat või *Boole'*i algebrat.

[2] *Java* (*Sun Misrosystems*) platvormideks on näiteks *Solaris, Windows* ja *Linux.*

[3] Pidagem silmas, et baitkood *ei ole* reaalse masina protsessori poolt interpreteeritav masinkood. Teda interpreteerib *protsessori* asemel *programne interpretaator.* Mis aga ei välista, et sellise interpretatsiooni

Allpool on tuginetud *Wikipedia*le [116]. Baitkood kujutab endast abstraktse arvuti masinkoodi ja assemblerit; nimetus viitab tõigale, et käsukood on ühebaidine (seega võimalike käskude arv on piiratud 256-ga). Baitkoodi saab põhimõtteliselt kasutada programmeerimiskeelena, ent selleks puudub praktiline vajadus, kuivõrd kompilaatorid (*Sun Microsystems*i *javac* – *Java* lähteprogrammide originaalteisendaja baitkoodi – ja teised baitkoodi genereerijad) suudavad väljastada piisavalt kompaktset, ratsionaalset koodi. *Java* baitkoodi genereerivad lisaks *javac*le näiteks:

- *Jython* – *Python*i *Java*s-kirjutatud interpretaator;
- *JRuby* – *Ruby* kompilaator;
- *JGNAT* – kompilaator *Ada*st baitkoodi;
- *GCJ* – *Gnu Java*-kompilaator (suudab genereerida ka „päris"-masinkoodi).

Äsjaosundatud materjal esitab näitena *Java*-programmi:

```
outter: for (int i = 2; i < 1000; i++) {
 for (int j = 2; j < i; j++) {
   if (i % j == 0)
     continue outter;
 }

 System.out.println (i);
}
```

Javac-kompilaator teisendab tolle teksti baitkoodi, mis on esitatud tabelis 7.2.3.4a. *Wikipedia* originaali on täiendatud veergudega "kood" ja „semantika". Veerg "aadr" ei sisalda käskude järjekorranumbreid, vaid nonde kümnendsüsteemis esitatud suhtaadresse[1].

---

resultaadiks võiks olla reaalne masinkood (või reaalse masina assemblerprogramm, millest saab assembler-translaatori ja linkeri abil kompileerida *.exe*-faili). Tehnika, kus masinkood genereeritakse baitkoodi interpreteerimise käigus, on tuntud kui *„just-in-time-compiling*" (*JIT*). Alternatiiv on *AOT*-tehnika (*ahead-of-time*), kus baitkood kompileeritakse masinkoodiks ning *JVM* töötabki kui kompilaator. Nii toimivad näiteks *Excelsior JET* ja *GCJ* (viimane on *GCC* (*Gnu Compiler Collection*) osa, mis käsitleb *Java*t *C++* alamhulgana, kuhu on lisatud *API*-komponendid). Igal juhul, baitkood on sõltumatu nii operatsioonisüsteemidest kui ka protsessoritest (siiski, viimased peavad olema „baitmasinad" ja toetama „aparatuurset" magasini).

[1] Asi on selles, et "operandiväli" on spetsifitseerimata, tegelikud formaadid pole need, mis joonisel paistavad. Ühebaidiste instruktsioonide aadressid järgnevad eelmistele intervalliga 1, kus see nii pole, seal on eelmise käsu formaat pikem.

aadr.	kood	mnemokood	operand	semantika
0	05	iconst_2		push +2[1]
1	3c	istore_1		pop int → muutuja #1
2	1b	iload_1		push muutuja #1
3	17	sipush	1000	push *int* 1000
6	a2	if_icmpge	44	mine 44, kui (R)>=1000
9	05	iconst_2		push +2
10	3d	istore_2		pop int → muutuja #2
11	1c	iload_2		push muutuja #2
12	1b	iload_1		push muutuja #1
13	a2	if_icmpge	31	mine 31, kui (m#2)>=(m#1)
16	1b	iload_1		push muutuja #1
17	1c	iload_2		push muutuja #2
18	70	irem		jääk((m#1)/(m#2))
19	9a	ifne	25	mine 25, kui ≠ 0
22	a7	goto	38	mine 38
25	84	iinc	2,1	m#2:=(m#2)+1
28	a7	goto	11	mine 11
31	b2	getstatic		#84; //Field/java/lang/System.out:Ljava/io/PrintStream
34	1b	iload_1		push muutuja #1
35	b6	invokevirtual		#85; //Method java/io/PrintStream.print.ln;(I)V
38	84	iinc	1,1	m#1:=(m#1)+1
41	a7	goto	2	mine 2
44	b1	return		

Tabel 7.2.3.4a. Baitkoodi olemust seletav illustratsioon.

Java baitkood pakub meile huvi eeskätt kui *masinorienteeritud keel* oma assemblerkeele ja masinkoodiga ja vähem saame huvituda baitkoodi interpretaatorist – *Java* virtuaalarvutist *JVM* (ehkki see on iseenesest huvitav, ent paraku nõuaks üsna palju lehekülgi ja meie raamatu maht pole piiramatu). Allpool (kui pole öeldud teisiti) on tuginetud põhiliselt *Bill Vennersi* artiklile [59]. Alustagem "naturaalse baitkoodi" näitest:

```
03 3b 84 00 01 1a 05 68 3b a7 ff f9
```

Selle baidijada "disassembleerimine" annab järgmise koodi:

```
// Dissassembly:
iconst_0    // 03
istore_0    // 3b
iinc 0,1    // 84 00 01
iload_0     // 1a
iconst_2    // 05
imul        // 68
istore_0    // 3b
goto -7     // a7 ff f9
```

[1] *iconst_2* semantikat on seletatud kui "*push* konstantide vektori element indeksiga 2", siintoodu on arvatavasti lugemist hõlbustav lihtsustus. Konstantide vektori teeb mõistagi *Java*-translaator teksti leksikalise analüüsi käigus ning muidugi võib vektori esimesed elemendid moodustada nii, et nende väärtusteks on indeksid..

JVM on "täielikult" orienteeritud *LIFO*-tüüpi magasini(de)le – nagu näiteks ka meile juba tuttav *FORTH*. Virtuaalmasinal puuduvad *registrid* (mis loomulikult ei tähenda, et *Java* interpretaator ei tohiks või ei saaks neid kasutada)[1].

*JVM*i magasini element on 32-bitine; *long*- või *double*-tüüpi muutujad (või konstandid) paigutatakse kahele järjestikusele magasiniväljale. *JVM*-käsukoodid jagunevad järgmistesse rühmadesse [136]:

- laadimine ja salvestamine;
- aritmeetika;
- tüübiteisendused;
- objektide loomine ja nendega manipuleereimine;
- operandide magasini tegevused (push / pop);
- suunamine;
- alamprogrammide väljakutsumine ja naasmine ("meetodid")

JVM toetab järgmisi andmetüüpe (vt. [59]):

- *byte* (ühebaidine märgiga täisarv);
- *short* (kahebaidine märgiga täisarv);
- *int* (neljabaidine märgiga täisarv);
- *long* (kaheksabaidine märgiga täisarv);
- *float* (neljabaidine ujupunktarv);
- *double* (kaheksabaidine ujupunktarv);
- *char* (kahebaidine märgita *Unicode*-sümbol).

Erinevalt mikromasinate (näiteks *Intel*i) masinkoodist pole baitkoodi-baite "ringi tõstetud", so. 256_{16} on kujul 01 00 ja mitte 00 01 (nagu *Intel*i standardis). *JVM* operatsioonikoodid viitavad *üheselt* operandide formaatidele; seetõttu pole vaja operande varustada tüübimarkeritega.

Lõpetame *Venners*i näitega *Java*-meetodist *Convert( )* ja *javac* poolt genereeritud baitkoodist (õigemini, selle koodi disassembleri väljatrükist).

```
class Diversion{
      static void Convert(){
            byte imByte = 0;
            int imInt = 125;
            while (true) {
                  ++imInt;
                  imByte = (byte) imInt;
                  imInt *= -1;
                  imByte = (byte) imInt;
                  imInt *= -1;
            }
      }
}
```

[1] *Venners* [59] nendib: *JVM* oli disainitud magasinipõhise arvuti ja mitte registripõhise arhitektuuri jaoks. „Registrivaese" masina näitena tõi ta *Intel 486*; vastupidiseks näiteks võiks olla *Intel*i *IA-64* arhitektuur (128 üldregistrit, 128 ujupunktregistrit, lisaks spetsiaalregistrid [138]).

Adekvaatne bait-mnemokood on järgmine:

```
iconst_0  // Push int constant 0
istore_0  // Pop the local variable 0, which is imByte: byte imByte=0
bipush 125  // Expand byte constant 125 to int and push
istore_1  // Pop the local variable 1, which is mInt: int imInt=125
iinc 1 1  // Increment local variable 1 (imInt) by 1: ++imInt
iload_1   // Push local variable 1 (imInt)
int2byte  // Truncate and sign extend top of stack so it has valid byte
          // value
istore_0  // Pop to local variable 0 (imByte): imByte=(byte)imInt
iload_1   // Push local variable 1 (imInt) again
iconst_m1 // Push integer -1
imul      // Pop top two ints, multiply, push result
istore_1  // Pop result of multiply to local variable 1 (imInt):
          // imInt *= -1
iload_1   // Push local variable 1 (imInt)
int2byte  // Truncate and sign extend top of stack so it has valid byte
          // value
istore_0  // Pop to local variable 0 (imByte): imByte=(byte)imInt
iload_1   // Push local variable 1 (imInt) again
iconst_m1 // Push integer -1
imul      // Pop top two ints, multiply, push result
istore_1  // Pop result of multiply to local variable 1 (imInt):
          // imInt *= -1
goto 5    // Jump back to the iinc instruction: while (true) {}
```

Ülaltoodud „assembler"-baitkood on "masin"-baitkoodis järgmine:

03 3b 10 fd 3c 84 01 01 1b 91 3b 1b 02 68 3c 1b 91 3b 1b 91 3b 1b 02 68 3c a7 05.

Kokkuvõtlik (paraku ka põgus) ülevaade *Java* baitkoodist on lisas 10.

Niisiis ("niisiis" nende jaoks, kes on tutvunud ka lisaga) – baitkood on masinorienteeritud ainult niivõrd, kuivõrd ta eeldab, et objektmasin on "baitmasin" ja enamat ta objektmasina "raualt" (*hardware*) ei oota ega eelda[1]. See seik annab vabad käed *JVM*-realiseerijale suvalisel riistvaraplatvormil. Ja pidagem silmas, et ehkki *Java*-baitkoodi suudavad genereerida paljud "sugulaskeelte" kompilaatorid, siis saab selle interpreteerimisega hakkama ainult *Java* virtuaalmasin (*JVM*).

Meie käesoleva peatüki („tarkvara mobiilsus") kontekstis on *Java* baitkood kahtlemata toimiv *UNCOL* (=*UNiversal Computer-Oriented Language*), ent – pidades silmas *Java* orientatsiooni (objektorienteeritud) – ei saa ta olla "päris" *universaalne*[2].

[1] Muidugi, „aparatuurne magasin" peab objektmasinal olema, see ongi (vist) kõigil mikromasinatel ja näiteks *PDP*-del, ent mitte *IBM*i suurarvutitel.

[2] Nii baitkoodi kui ka mõne muu universaalse vahekeele olemusest ja rollist vt. ka [19, lk. 40 – 42].

7.2.3.5. *Microsoft*i vahekeel

Usutavasti nägime, et *Java* baitkood on vastuvõetav vahekeel *C*-perekonna objektorienteeritud kloonide, eeskätt *Java*-keele mobiilsuse tagamiseks erinevate protsessorite (ja operatsioonisüsteemide) platvormide vahel. Toetatakse nii *Solaris*t, *Linux*it, *Windows*i ja muidki (näiteks *UNIX*it). Samas, olenemata platvormist, on see baitkood interpreteeritav ainult antud platvormi jaoks kirjutatud interpretaatoriga (*JVM*).

Microsoft on arendanud analoogilist projekti. Selle firma Eesti esinduse spetsialist *Andres Sirel* ja teine asjatundja, *Erki Savisaar* selgitasid tolle olemust nende ridade autorile järgmiselt: projekti "üldnimi" on *.NET* (loe: dot-net), nii nimetatud objekt on mitmekihiline toodete komplekt (*.NET framework*, eesti keeles *.NET- raamistik*); programmeerimiseks selle platvormi jaoks tuleb kasutada mõnda *.NET*-keelt[1]. Kõik nood keeled peavad aktsepteerima *CLR*-keskkonda (*Common Language Runtime*) ning realiseerima kõik *CLI* (*Common Language Infrastructure*)[2] poolt ette kirjutatu. *.NET* raamistik koosneb *CLR*ist ja kasutajatele mõeldud *klasside paketist* (viimane on ilmselt teegiprogrammide analoog).

*.NET*i rakenduste kompileerimisel läbitakse kaks faasi:

- transleeritakse kõrgtaseme keelest (näiteks, *C#*) süsteemsesse assemblerilaadsesse vahekeelde *MSIL* (*MicroSoft Immediate Language*, teised nimekujud on *CIL – Common Immediate Language,* ka *.NET CIL*, loe: "dot-net si-ai-el"). Nentigem, et assemblerteksti genereerimine pole obligatoorne, genereerida võib sel etapil ka baitkoodi (assembleri tase projektis on vajalik baitkoodi lugemiseks retranslaatori abil);
- masinkoodi genereerimine (kas *.NET*i assemblerist või baitkoodist).

TÜ ATI magistrandi *Janek Press*i andmetel toimub baitkoodi kasutamine järgmiselt:

- mällu laetakse platvormist sõltumatu baitkood;
- baitkood kompileeritakse ja resultaat salvestatakse vahemällu *Global Assembly Cash*;
- lingitakse juurde vajalikud teegid ja käivitatakse;
- järgmis(t)el käivitamis(t)el võib teise sammu (baitkoodi kompileerimise) vahele jätta.

Nagu *JVM*, nii ka *.NET* on orienteeritud täiesti magasinile (ja mitte ilmutatult registritele). Samas, kui *JVM* oli orienteeritud paljudele platvormidele, siis *Microsoft*i *.NET* orienteerus ainult *Windows*ile [121].

[1] Esimesed keeled, mille jaoks kirjutati *MSIL*-translaatorid, olid *C#*, *Visual Basic .NET* (lühemalt ka *VB .NET*) ja *Managed C++*.
[2] Nimetatu on *ECMA\ISO* standard.

A. Sirel ja *E. Savisaar* toovad tänapäevase *UNCOL*i eelised välja järgmiselt:

- programmeerimiskeelte loojad ei pea mõtlema realiseerimise peale;
- neil tuleb tagada kompileeritavus *MSIL*i[1];
- keelte disainerid ei pea muretsema kõikvõimalike riistvaraarenduste ja -seadistuste pärast;
- operatsioonisüsteemid (olles huvitatud *.NET*i toetamisest) tagavad kompileerimise konkreetsele protsessorile orienteeritud koodi, nii näiteks pole vahet, kas tegemist on *Intel*i või *AMD*ga, on ta 32- või 64-bitine jne. *JIT*-variandi puhul orienteerutakse konkreetsele platvormile alles töötluse ajal, *AOT*[2] puhul enne koodi käivitamist, õiges keskkonnas. Interpreteeritav (kompileeritav) baitkood on iga variandi jaoks sama.

Tänaseks on kirjutatud translaatoreid *lähtekeel* → *CIL* märkimisväärsel hulgal[3], kui tugineda magistrant *Siim Karuse* leitud andmetele (vt. joonis 7.2.3.5a).

Oluline erinevus *Java* virtuaalmasinaga (*JVM*) on *Microsoft*i versioonil asjaolu, et viimase koodi ei interpreteerita resultaatide saamiseks[4], vaid sellest kompileeritakse (tavaliselt *JIT*-tehnikaga) objektmasina masinkood. *.NET*i assemblerist ja baitkoodist huvitatuile soovitame tutvuda allikatega [124] ja [125]. Viimasest allikast on laenatud lisas 11 toodud fragment tabelist *.NET* − baitkood ning siinkohal piirdugem kõige triviaalsema näitega, muidugi on selleks programm, mis väljastab ekraanile sõnumi "tere maailm".

Nagu lubatud, esitame "*Hello world*"- programmi: esiteks keeles *C#* (mis ei erine oluliselt ei *C*-, *Java*- ega ka *C++*-keelest), seejärel *.NET*-versioonis ja lõpuks näitame, kuidas toda objektkoodi interpreteerib *.NET*-disassembler. Niisiis, *C#* :

```
class HelloWorld
{
  static void Main () {
     System.Console.Write("Hello World!");
  }
}
```

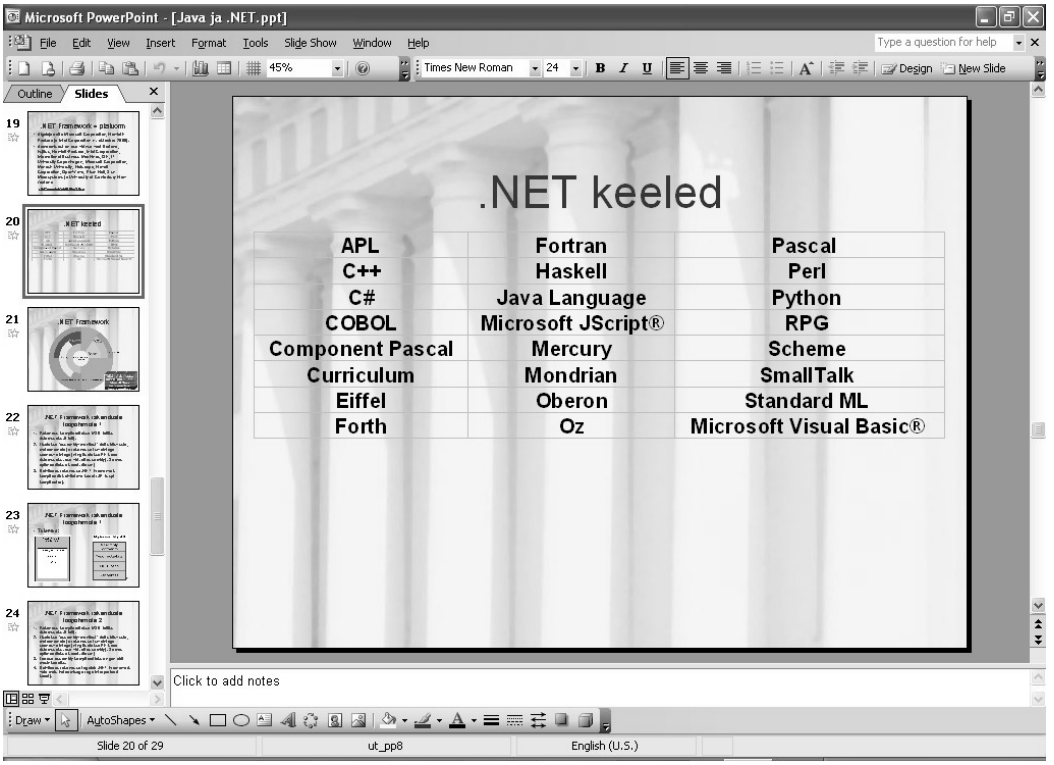
(vt. [122]. Ülejäänud näide pärineb allikast [123]. *MSIL*is käsitsi kirjutatult näeb see programm välja nii (kommentaarid on meie poolt, lisasime nad *Inteli* assembleri stiilis, *MSIL* ei pruugi seda aktsepteerida).

[1] Tundub, et kompileeritavate keelte jaoks on see nõue hõlpsasti täidetav, ent interpreteeritavate keelte jaoks ei pruugi. *MSIL*i direktiivide piirarv on 256 (selle dikteerib *bait*kood) ning keelele, mis orienteerub loetelu järgmise punkti (lisaks traditsioonilistele võimalustele) realiseerimisele võib neist väheks jääda. Seega, esimese ja teise punkti vahel võib olla vastuolu.

[2] *JIT=Just-In-Time* ja *AOT=Ahead-Of-Time.*

[3] Võib arvata, et kõik nood translaatorid on varustatud ka retranslaatoritega, mis suudavad tõlkida *CIL* baitkood → *keele keskkond* (*retranslaator* tõlgib originaaltranslaatori objektkoodist tagasi lähtekoodi).

[4] See oli *JVM* üks kahest võimalusest, teine oli kompileerimine masinkoodi, mis on *.NET*i ainus võimalus.



Joonis 7.2.3.5a. *.NET*-keeled (slaid pärineb *Siim Karus*e esitluselt 10.05.2006).

```
Example (informative):
.assembly extern mscorlib {}    ;viide teegile, kus on System.Console
                                 ;kirjeldus
.assembly hello {}              ;asm-programmi nimi
.method static public void main() cil managed  ;globaalne main-meetod
{                               ;meetodi keha on sulupaari { } vahel
  .entrypoint                   ;programmi sisendpunkt
  .maxstack 1
  ldstr "Hello, World!"         ;stringi aadress → magasin
  call void [mscorlib]System.Console::WriteLine(class System.String)
                                ;stringi trükk (ilmselt magasinist)
  ret                           ;meetodist väljumine
}
```

Baitkoodi saamiseks tuleb *MSIL*-assemblerprogramm transleerida *ilasm*i (*Intermediate Language Assembler*) abil. Baitkoodi "ennast" me siinkohal ei esita, küll aga toome *Visual Studio .NET*i abil transleeritud *C#*-programmi baitkoodi "retransleeritud" (kasutades *ildasm*i – *Intermediate Language Disassembler*) variandi :

```
.class private auto ansi beforefieldinit ConsoleApplication1.HelloWorld
       extends [mscorlib]System.Object
{
} // end of class ConsoleApplication1.HelloWorld
```

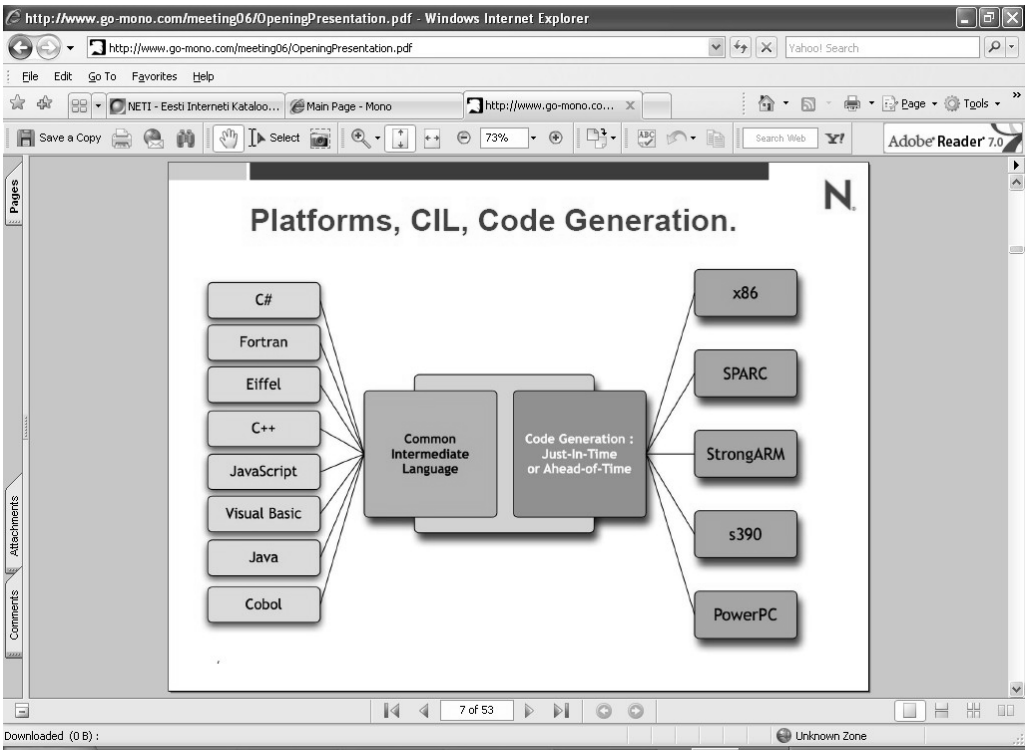
```
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method HelloWorld::.ctor


.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello, World!"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method HelloWorld::Main
```

7.2.3.6. *MONO*

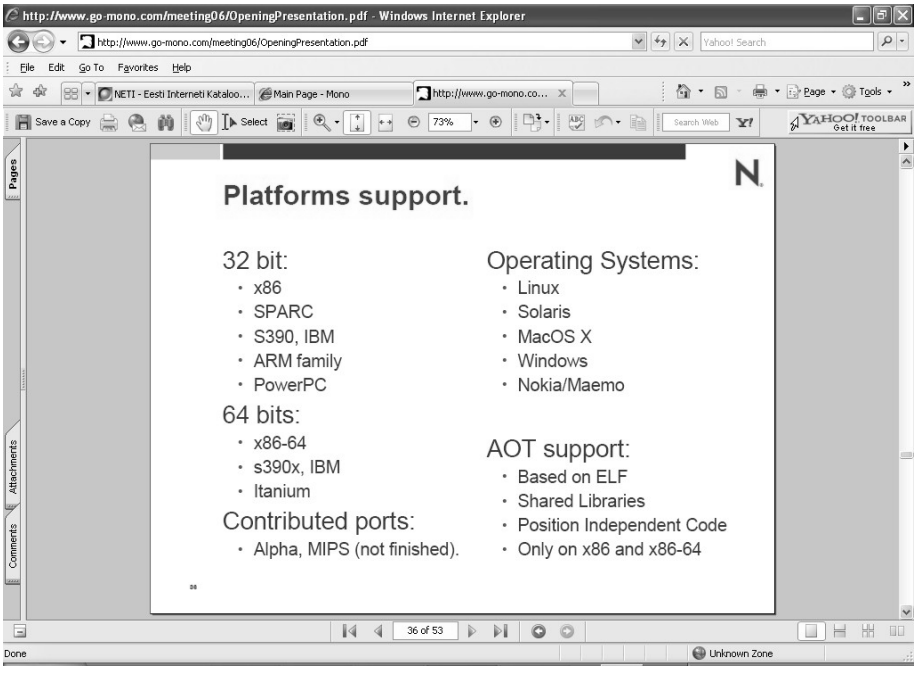
Niisiis, firmas *Sun Microsystems* töötati välja *Java* baitkood, *Microsoft*is *.NET*. Esimene on orienteeritud paljudele platvormidele (so., operatsioonisüsteemidele ja protsessoritele), ent keeleliselt eeskätt *Java*le, teine aga ühele platvormile (*Windows*), ent paljudele põhimõtteliselt erinevatele keeltele. Kontseptuaalselt on mõlemad projektid lähedased, neid ühendab abstraktse masina vahekeel. Ületamaks (eeskätt ärilist) ühildamatust konkureerivate projektide vahel lõid "entusiastid" (nagu nad end oma koduleheküljel [*http://mono-project.com/Main_Page*]) tutvustavad[1] vabavaralise projekti *MONO, laiendamaks .NET*-raamistikku ka teistele platvormidele (op-süsteemidest näiteks *Linux, Solaris, Mac OS X* ja *UNIX* ning erinevatele protsessoritele – *Intel* (*x86*), *PowerPC* jmt.). Noid taotlusi näitlikustavad joonised 7.2.3.6a, 7.2.3.6b ja 7.2.3.6c [126].

MONO on huvitav ning (näib, et) perspektiivikas projekt; huvilistel soovitame külastada *MONO* koduleheküIge ja tollega seotud materjale, ent eriti [126], [127] ja [128].

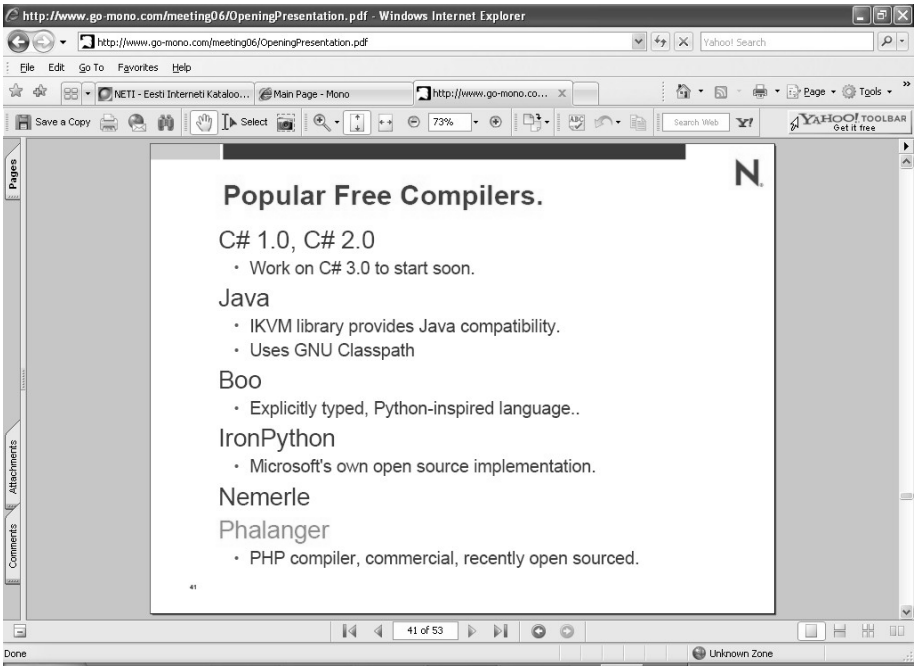
[1] Rahvusvaheline meeskond, osalejaid on nii Ameerikas kui ka Euroopas. Nende ridade kirjutamise ajal oli äsja lõppenud "entusiastide kohtumine" (*meeting*) Bostonis, 9. novembril 2006. Peateemad olid *MONO* integreerimine *Linux*i ja *MS API*ga.



Joonis 7.2.3.6a. Projekti *MONO* platvormid.



Joonis 7.2.3.6b: *MONO* platvormide toetused.



Joonis 7.2.3.6c. Esimesed *MONO*-projekti kaasatud keeled.

7.2.3.7. *.NET* ja *Java* baitkood

Paarid *Java* ja *JVM* (Java virtuaalmasin) ja *C#* ning *CLR* (*Common Language Runtime*) on täiesti võrreldavad: mõlemad baseeruvad virtuaalarvutile, mis on abstraheeritud objektmasina riistvarast, ja mõlemad on orienteeritud abstraktse masina assemblerile[1]. Mõlemad variandid toetavad tarkvara akumulatsiooni (tuletagem meelde: selle tee avas *FORTRAN* oma teekidega) *teekide klassi* aktsepteerimisega.

Nähtav erinevus kahe konkurendi vahel seisneb selles, et *Java* on avatud kõigile platvormidele (kui mõtleme nende all erinevaid op-süsteeme ja protsessoreid), *.NET* aga ainult *MS Windows*ile (ja eeskätt *Intel*ile)[2], ja *.NET* toetab (põhimõtteliselt) kõiki kõrgtaseme keeli, ent *Java* – ainult ennast ja oma "lähisugulasi".

*.NET*i arendamine on pälvinud ka kriitikat. Esiteks, uue kontseptsiooni evitamine põhjustas mitmete *Microsoft*i "toodete" uute versioonide ühildamatuse vanadega, teiseks – lihtne *C* on oluliselt efektiivsem kui üldine lahendus, ja kolmandaks, orienteeritus *Windows*ile pole pluss (vt. [135]).

[1] Meenutagem, et kõrgemal tasemel ei õnnestunud konkurentsivõimelist *UNCOL*i luua.
[2] Siiski, vt. eespoolt projekti *MONO* – ent see pole *Microsoft*i produkt.

Tundub, et programmeerimiskeelte *lingua franca* on seda reaalsem, mida madalamal tasemel seda otsitakse. *PL\I* ja *Algol-68* olid tüüpilised kõrgtaseme keeled (ehkki kontseptuaalselt diametraalselt erinevad) ja tarkvara mobiilsuse probleemi lahendajatena kukkusid mõlemad läbi. *UNCOL* (*resp. ALMO*) oli tasemelt midagi makrokeele ja protseduuriorienteeritud keele vahepealset ja oli ilmselt ka liiga „üldine" selleks, et püstitatud eesmärke saavutada. *Java* baitkood ning *Microsoft*i *.NET* on masinorienteeritud[1] taseme keeled ja – uues situatsioonis, kus konkureerivate protsessoritüüpide arv on läbi aegade väikseim – saavutasid nad lõpuks vististi selle rolli, mis avab perspektiivi tegelikule erinevate keelte ja platvormide ühildamisele. Lootkem, et me pole ülemäära optimistlikud[2].

[1] Pidagem silmas, et masin, millele orienteeruti, on *virtuaalne*: sõltumatu „rauast" ja operatsioonisüsteemist.
[2] Lootusi toetab tuntavalt *MONO*-projekt.

8. SÜNTAKS JA TRANSLEERIMINE

8.1. Semantika ja süntaks

Üldises plaanis on semantika hoopis avaram mõiste kui süntaks; viimane on kasutatav ainult *verbaalsete keelte* (loomulikud inimkeeled või inimeste loodud tehiskeeled) kontekstis, ent „semantika" laieneb (vähemalt selgroogsete) loomade kommunikatsioonile. Kõik me adume, mida tähendab koera rõõmus klähvimine, meie najale hüppamine või sabaliputamine või siis tige urin ja haukumine. Või inimeste kehakeel, ilmed ja intonatsioon (viimane neiski keeltes, millest me sisuldasa midagi ei taipa).

Inimkeeled on semantiliselt põhimõtteliselt võrdväärsed (kui mingis keeles puudub mingile mõistele täpne vaste, suudetakse see kas „lahti rääkida" või luuakse vastav omakeelne sõna või võetakse see lihtsalt teisest keelest üle). Ent grammatiliselt (süntaktiliselt) on maailma inimkeelete rühmad vägagi erinevad ─ võrrelgem kasvõi indoeuroopa ja läänemeresoome keeli.

Öeldu kehtib suuresti ka programmeerimiskeelte kohta, vajadusel saab mistahes universaalsele (protseduurorienteeritud) keelele lisada vahendeid uute riistvaravõimaluste kasutamiseks. Niisiis, ka üldotstarbelised (nii masin- kui ka protseduurorienteeritud) programmeerimiskeeled on *semantiliselt võrdväärsed*.

Üldiselt tunnustatakse seisukohta, et semantika kirjeldab paljuski sellist, mida süntaks ei suuda kirjeldada. Ärme naase loomariiki (me ei tea midagi süntaksist, mis ei tähenda, et seda ei võiks olla), näiteks eestikeelne lihtne küsimus „mida?" evib erinevat semantilist väärtust lillepoes (mida?) ja pimedas kangialuses (mida?!). Ent naaskem programmeerimiskeelte juurde: formaalne süntaks ei suuda väljendada *ilmutamata* (i.k *implicid*) operande, näiteks globaalseid muutujaid või ühisvälju, *kõrvalefekte*[1] (i.k. *side effect*, v.k. *побочный эффект*) *viitade keskkondi* jmt.

Süntaks sõltub ─ eeskätt või paljuski ─ keele autori(te) *suvast*; sama semantikaga konstruktsioon võib eri keeltes välja näha sootuks erinevalt, näiteks vektori 1. elemendi kasutamine [44, lk. 308]:

A[1] (*Algol*-60);
A(1) (*FORTRAN*);
(CAR A) (*LISP*);
FIRST OF A (*COBOL*);
A.FIRST (*PL\I*);
A 1 + @ (*FORTH*, lugemine) **ja …A 1 + !** (*FORTH*, kirjutamine).

[1] Triviaalne näide: omistamisoperaator *y:=a+b* kasutab kõrvalefekti, muutes *y* kui operandi väärtust.

8.2. Süntaksi elemendid

Formaliseerimata saame eristada järgmisi „ehituskive", mille abil saab kirjutada süntaktiliselt õiget teksti mingis keeles.

- *Tähestiku* moodustavad need märgid, mida saab kasutada antud keele teksti koostamisel. Näiteks, venekeelse teksti jaoks saame kasutada tähti *ŭ, ф, ы, р, я* jne., kreekakeelse jaoks η, θ, λ, μ, ξ, π jne või hieroglüüfkirjas 𓄿, 𓏏, 𓐍 jne. Kui räägime programmeerimiskeeltest, siis on ilmne, et keele autorid peavad lähtuma neist võimalustest, mida pakuvad sisend- väljundseadmed[1]. Kui viimaste võimalused on piiratud, siis sealt võib johtuda kaksitimõistetavuse oht — näiteks, *FORTRAN*is oli sümbol „." silmnähtavalt semantiliselt üle koormatud
- *Identifikaatorid.* Tavaliselt (traditsiooniliselt) kirjeldatakse nonde objektide esitamise eeskirjad *verbaalselt* (selmet defineerida nad regulaaravaldiste (vt. *Chomsky* klass 3) abil). Identifikaatorite hulga moodustavad objektide (muutujad, funktsioonid, alamprogrammid, mõnes keeles ka konstandid) nimed ja märgendid. *Nimede* kirjutamise reeglites on enamik programmeerimiskeeli üpris sarnased: reeglina algab nimi tähega ja võib jatkuna sisaldada tähti või numbreid; uuemates keeltes on jatkuosas lubatud kasutada ka alakriipsu „_". Vanemates, eriti (pool)-fikseeritud formaadiga („perfokaartorienteeritud")[2] keeltes oli nime maksimaalne pikkus rangelt piiratud, tavaliselt 5 või 6 sümboliga. Pikkuspiirangud on säilinud tavaliselt ka uuemates keeltes, ent harilikult nii, et kitsendus pole reaalselt tuntav, olles näiteks 31 sümbolit ($2^5 - 1$). *Märgendite* kohta kehtib üldjoontes sama, ent silmatorkava erandina meenutagem, et *FORTRAN*i (ja paljuski seda järgiva *BASIC*u) märgend on märgita täisarv. Nimede osas on omaette nähtus *FORTH*: märgendeid pole üldse ja nimi võib koosneda mistahes *ASCII* põhisümbolist[3] mistahes järjekorras. Mida vähem kitsendusi keel nime moodustamisele esitab, seda parem on see programmeerijale ja programmi inimesest lugejale, kuivõrd nii saab väljendada nimedega varustatud objektide semantikat ilma šifreerimata.
- *Tehtemärgid*, näiteks põhikoolist tuttavad sümbolid „+, −, × ja ∕". Ent on keeli, kus seda traditsiooni ei järgita, näiteks „+" on *Lisp*is *PLUS*( ) — selles keeles tuleb kõik avaldada (rekursiivsete) funktsioonidega, või *COBOL*, kus tehtemärke asendavad sõnad, näiteks „+″ asemel kasutatakse sõna „*ADD"*. Ent üldjuhul on masinast sõltumatutes keeltes aritmeetika- ja võrdlustehted kirjutatavad harjumuspärasel kujul ning erinevused on üpris väikesed, näiteks astendamistehte märk on *Algol-60*s (publitseerimiskeel!) „↑" ja *C*-s „**", korrutamine vastavalt „×" ja „*". Johtuvalt sisend-väljundseadmete piiratud märkide arvust on *FORTRAN*i võrdlustehted punktide vahel esitatud sõnad, näiteks ≠ on „.NE.", *C*-keeles ja selle keele stiili järgivates keeltes aga „!=".

[1] Tõsi, on erandeid: näiteks *APL* kasutas spetsiaalselt sellele keelele orienteeritud operaatori-kirjutusmasinat, kus olid klaviatuurilt valitavad näiteks sümbolid ω, ρ, ~, ⊃, ⊂, Δ, või Ô. *APL* (*Iverson*, 1970) on muide siiani "elus".

[2] Näiteks assemblerid, *FORTRAN* või *Snobol-4*.

[3] Mõeldud on trükipildis nähtavaid sümboleid, *FORTH*-sõnade kohustuslik eraldaja on *tühik* (*ASCII* kood 32).

- *Võtme-* ja *reservsõnad* (i.k. *key words, reserved words*, v.k. *ключевые и зарезервированные слова*) esitavad lekseemiklasse, milledel on üldjuhul suur ühisosa ja enamikus uuematest keeltest on *kõik* võtmesõnad üksiti reservsõnad. Üldiselt, reserv(eeritud)sõnade hulk on võtmesõnade hulga alamhulk: need on sellised võtmesõnad, mida ei tohi ega saa (translaator ei lase) kasutada muus kui ainult võtmesõna funktsioonis. Võtmesõnadega määratakse *operaatorite* funktsioonid; tavaliselt operaator algab võtmesõnaga (näiteks *for*, *array*, *variable*, *if* jne) ja võib omamoodi eraldajatena kasutada võtmesõnu, nagu *then*, *else* või *endcase*.Samuti markeeritakse võtmesõnadega operaatorsulgusid (näiteks paarid *begin…end*, *if …endif*) — seevastu näiteks *C*-koolkonnas asendab sõnalisi operaatorsulge tavaliselt eraldajate paar „{…}". Näitekeeleks, kus võtmesõnad pole reserveeritud sõnad, on *FORTRAN*. Translaatoril pole seal raskusi võtmesõnade tuvastamisega — reeglina algab operaator võtmesõnaga ja operaatori mõjupiirkonnas tolle võtmesõna esinemine translaatorit ei eksita, seda enam, et *FORTRAN* välistab nii rekursiooni kui ka hierarhilised liitoperaatorid. Nii võime selles keeles kirjutada näiteks *IF(IF.NE.0)IF=0.*

- *Kommentaarid.* Keele realiseerimise aspektist on kommentaar *tühi operaator*: translaator jätab ta lihtsalt vahele. Samas on kommentaaride tähtsust raske üle hinnata programmi teksti loetavuse (ja mõistetavuse) aspektist. Kommenteerimist toetavad pea kõik programmeerimiskeeled alates assembleritest. Süntaktiliselt eraldatakse kommentaar programmi transleeritavast tekstist operaatorsulgudega; tavaliselt on kasutusel paar „*algusmarker …reavahetus*" — *C*-keeles näiteks on algusmarkeriks paar „//", *Inteli*-assembleris aga sümbol ";". Pikemate kommentaaride jaoks on keeltes algusmarkerile sümmeetrilised lõpumarkerid, *C*-keeles näiteks paar „/* … */", *Modula-2*-keeles aga „(* … *).

- *Müra* (i.k. *noise*, v.k *шум*) on kasutatav vaid vähestes keeltes: siia klassi võime paigutada need võtmesõnad, mis võivad ingliskeelsed inimesed inglise keelele baseeruvas programmeerimiskeeles end harjumuspäraselt väljendada, ent mida translaator eirab. Nii piisab *COBOL*is ja *PL\I*s suunamiseks sõnast *GO*, ent kirjutada võib ka *GO TO* — siin on *TO* translaatori jaoks kommentaariga võrdväärne, ignoreeritav tekst.

- *„Tühikud".* Nende hulka loetakse nii „tühik ise" (*space*) kui ka tabulatsiooni- ja reavahetussümbolid (nende koodid on programmi tekstis varjatud, ent nende mõju on nähtav). Näiteks, *ASCII* 10-nd-koodis vastab kood 32 tühikule, 9 horisontaalsele tabuleerimisele, 10 reavahetusele ja 13 — „kelk tagasi", so. rea algusse. Üldjuhul pole neil sümbolitel semantilist kaalu, *FORTRAN*is on tühikul mõtet ainult *Hollerithi konstantides* (nendega formeeritakse väljundstringe), ent seesama tühik (*ASCII* 32) on — tsiteerides *Pratt*i [44, lk. 317] — keeles *SNOBOL 4* suurte segaduste allikas, sest ta tähistab nii eraldajat kui ka stringide konkatenatsiooni (liitmist). Ja *Haskell*is on mõnel juhul tabulatsioonil („treppimisel") operaatorsulu funktsioon — sisuliselt sama, mis *C*-keeles sulupaaril „{ … }".

- *Eraldajad ja sulud.* Siin peame ennast pisut kordama − *eraldajad* on nii tühikud kui ka tehtemärgid, neid kirjutatakse vajadusel muude süntaktiliste konstruktsioonide vahele. *Sulud* on esiteks juba ülalkäsitletud *operaatorsulud* (*begin …end,* või näiteks ka „{ … }".. Nendega pannakse paika algoritmi struktuur (plokid ja tegevused plokkides). Nende konstruktsioonide mõte on parandada programmi teksti loetavust ja (translaatorile –) üheseltmõistetavust. Ja teiseks, s*ulud* (reeglina paar „(…)" on ainus vahend, millega saab (aritmeetilistes ja loogilistes) avaldistes muuta tehete sooritamise (prioriteetidega paika pandud) järjekorda.

- *Avaldised* (i.k. *expressions*, v.k. *выражения*) – need on põhilised komponendid, millest konstrueeritakse *operaatoreid.* Me käsitleme neid järgmises alapeatükis – siia „pisiteemade-punktide" vahele mahutamiseks on see teema liiga mahukas (ja oluline).

- *Operaatorid* (i.k. *statements*, v.k. *инструкции*) võivad olla kas liht- või struktureeritud: esimesed ei sisalda, teised aga sisaldavad (võivad sisaldada) teisi operaatoreid. Ainult lihtoperaatoreid võimaldavad näiteks *FORTRAN* ja *APL*, tavaliselt lubavad protseduurorienteeritud keeled liitoperaatoreid, näiteks kujul *if…then…else.* Vaikimisi täidetakse operaatoreid selles järjekorras nagu nad programmi tekstis esinevad ning muuta saab seda järjekorda ainult spetsiaalsete operaatoritega: *suunamine, lüliti ja tingimus* (*C*-keeles vastavalt operaatorid *goto*, *switch* ja *if*-lause).

8.3. Avaldised

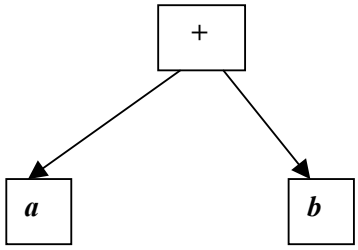
Avaldiste (aritmeetilised või loogilised või nende kombinatsioonid) puhul on olulisim, kuidas keeles on fikseeritud elementaaravaldis, võimalikud on 3 kuju (näitetehe on muutujate *a* ja *b* liitmine)[1]:

- *infiks* (tehtemärk paikneb operandide vahel – just nii nagu (kooli)matemaatikast harjunud oleme: *a* + *b*;
- *prefiks*: + *a b* (nii on kombeks funktsionaaltes keeltes: „+" on kahe argumendi – *a* ja *b* funktsioon;
- *postfiks*: *a b* +.

Neil kujudel on otsene seos avaldise kahendpuuga: suvalist avaldist saab rekursiivselt kujutada kahendpuu abil — *infiks*-notatsioonis esimesena sooritatava tehte märk on alampuu juure märgend ning nii vasak kui ka parem alampuu moodustatakse nii, et tehtest vasemale jäänud avaldise osast moodustatakse vasem alampuu ja paremale jäänud osast – parem alampuu.

[1] Vanas eesti koolimatemaatikas nimetati "liitmist" "kokkulöömiseks". Kui *a* on üks telliskivi, *b* teine telliskivi ja "+" – "löö kokku", siis *prefiks* annab eeskirja: "löö kokku, võta 1. kivi, võta 2. kivi", *infiks* "võta esimene kivi, löö kokku, võta teine kivi" ja *postfiks* – "võta esimene kivi, võta teine kivi, löö kokku".

Niisiis, „normaalse avaldise" $a + b$ kahendpuu on järgmine:



Kahendpuu läbimiseks on defineeritud 3 varianti (sulgudes samaväärsed alternatiivid), kasutame järgmist terminite süsteemi: ***juur*** on alampuu juur, joonisel +; ***vasak*** on juure vasak alluv (*a*) ning ***parem*** – parem alluv (*b*). :

- *eesjärjekord* (ik. *preorder*): *juur* → *vasak* → *parem* (*juur* → *parem* → *vasak*);
- *keskjärjekord* (ik. *inorder*): *vasak* → *juur* → *parem* (*parem* → *juur* → *vasak*);
- *lõppjärjekord* (ik. *postorder*): *vasak* → *parem* → *juur* (*parem* → *vasak* → *juur*).

Läbime ülalesitatud kahendpuu nois variantides, iga „aktiivse tipu" märgendi kirjutame välja. Eesjärjekorras läbides saame tulemuseks $+\ a\ b$ – avaldise *prefiks*-kuju, keskjärjekord annab *infiks*-kuju ($a + b$) ning lõppjärjekord – *postfiks*-kuju ($a\ b\ +$)[1].

8.3.1. *Infiks*-kuju

Enamik masinast sõltumatuid programmeerimiskeeli toetavad avaldiste *infiks*-kuju, kus-juures tehetele on keele kirjelduses tavaliselt[2] omistatud *prioriteedid*, *Algol-60* tehetel on need kahanevas järjekorras sellised:

priorit.	**tehted**
7	$\uparrow$
6	$\times$ / $\div$
5	+ –
4	$< \leq = \geq > \neq$
3	$\neg$
2	$\wedge$
1	$\vee$

Tehete sooritamise järjekorda saab tavaliselt muuta sulgude abil. Näiteks, kui me kirjutame $A+B * C- D$, siis leitakse kõigepealt $B * C$, ent korrutustehe sooritatakse viimasena,

[1] Ingliskeelses kirjanduses kasutatakse rööbiti siintooduduga ka termineid *preorder, postorder* ja *endorder*, vastavalt *ees-, kesk-* ja *lõppjärjekorra* tähenduses. Meie arvates on see terminoloogia vähem-süsteemne, kuivõrd ta ei haaku avaldise kolme kuju nimetustega (ik. *prefix, infix* ja *postfix*).
[2] Prioriteete pole näiteks *APL*-is.

kui kirjutame $(A+B)$ * $(C-D)$. Sama tavaliselt sooritatakse ühesuguse prioriteediga tehted järjekorras vasakult paremale; erand on taas *APL*, kus toimitakse vastupidiselt.

8.3.2. *Prefiks*-kuju

Niisiis, avaldise prefiks-kuju saame, läbides avaldise kahendpuu eesjärjekorras (*preorder*). Tuntud on kolm väikeste erinevustega varianti: *tavaline prefiks-kuju*, *Cambridge*'i *Poola kuju* ja *Poola kuju*[1]. Nimedes esinev *Poola* viitab tuntud Poola matemaatikule-loogikule *Jan Łukasiewicz*ile – prefikskuju autorile, kes avaldas oma idee 1920. aastal,

Jan Łukasiewicz (21.12.1878 Lvov – 13.02.1956 Dublin)[105].

olles Varssavi Ülikooli professor; 1946. aastal emigreerus ta Belgiasse, sealt edasi Iirimaale, kuhu ta kutsuti Dublini Ülikooli professoriks. Alates 1956. aastast on avaldiste esitamine Poola kujul või viimine sellele kujule programmeerimiskeelte ja translaatorite valdkonnas üheks fundamentaalsetest komponentidest (võrreldav näit. *Boole'*i *algebra* või *formaalsete grammatikatega*).

Infiks-avaldis $(A + B) \times (C - D)$ näeb tavalisel prefiks-kujul välja nii:
$\times (+ (A, B), - (C, D))$ ja Cambridge'i Poola kujul (*CPF*)
$(\times (+ \ A \ B)(- C \ D))$ ning selle suluvaba esitus on lihtsalt Poola kuju:
$\times + \ A \ \ B - \ C \ D$ [44, lk. 143].

Mainigem, et iga *LISP*-programm[2] kujutab endast üht *CPF*-avaldist.

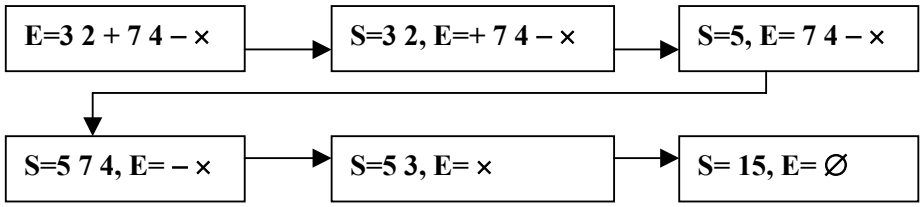
8.3.3. *Postfiks*-kuju

Siingi on kaks varianti: *prefiks-kujule* sarnane *postfiks-* e. *sufiks-kuju* ning *Poola kujule* analoogiline *inverteeritud Poola kuju* (i.k. *Inverted Polish Form*, v.k. *обратная Польская запись*). Meie näiteavaldis on esimesel kujul
$((A, B) + (C, D) -) \times$ ning teisel
$A \ \ B + C \ D - \ \times$.

[1] i.k. *Polish Form*, v.k. *Польская запись*
[2] *Lisp*i lõi 1958. a. *MIT*-s *John McCarthy* . Et *MIT* (*Massachusetts Institute of Technology*) asub *Suur-Bostoni Cambridge'i*-nimelises osalinnas, siis sai *Lisp*i süntaksi esitamise kuju nimeks *Cambridge Polish Form.*

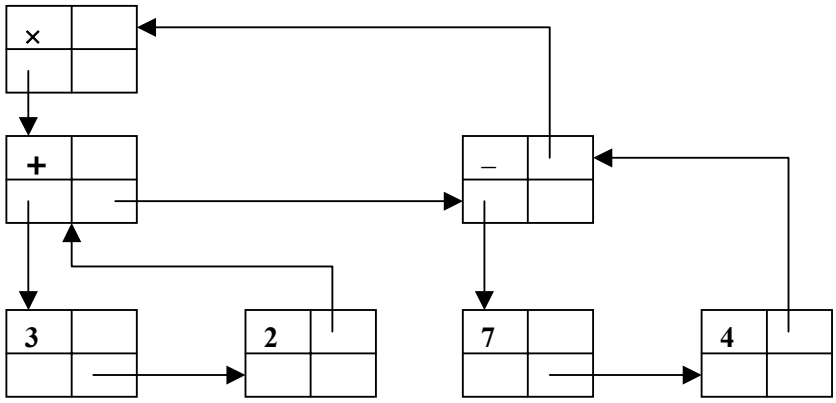
Postfiks-kuju ei kasutata tavaliselt programmeerimiskeelte avaldiste süntaksis (erand on *FORTH*), ent sellel kujul esitatud avaldisi on *LIFO*-tüüpi magasini abil väga lihtne lahendada. Näiteks, olgu *A*=3, *B*=2, *C*=7 ja *D*=4. Siis võime oma näite kirjutada kujul
3 2 + 7 4 – ×;

Magasini abil arvutatakse avaldise väärtus välja nii: avaldist töödeldakse vasakult paremale, operandid pannakse magasini, tehe sooritatakse magasini tipmisele eelneva elemendi ja tipmise elemendi vahel (lahutamine ja jagamine pole kommutatiivsed tehted) ning resultaat kirjutatakse magasini operandide asemele. Illustreerime seda järgmisel joonisel (avaldist tähistame *E* ja magasini *S*-ga), seisud on esitatud järjekorras vasakult paremale ja ülalt alla (nagu ka „plokkskeemiga" näidatud).



Joonis 8.3.4a. *Poola kuju*l avaldise väärtuse leidmine.

Niisiis, reeglina kasutavad masinast sõltumatud programmeerimiskeeled avaldiste *infiks*-kuju ja realiseerida (loe: interpreteerida väärtuse leidmiseks või käskude genereerimiseks, so. kompileerimiseks) on triviaalne *postfiks*-kulul esitatud avaldist . Süntaksorienteeritud transleerimismeetodid võimaldavad ehitada süntaksi analüüsi käigus avaldise kahendpuu, mille läbimine lõppjärjekorras annab inverteeritud *Poola kuju* – ent sel juhul pole viimast vajagi, kuivõrd me võime avaldist interpreteerida juba puu läbimise käigus. Näitame, kuidas seda võiks teha; kasutame selleks viimast avaldist, mille kahendpuu on joonisel 8.3.4b ning puu ise on kujutatud graafina (vältimaks magasini kasutamist puu läbimisel).



Joonis 8.3.4b**.** Avaldise (3+2)×(7–4) „puu".

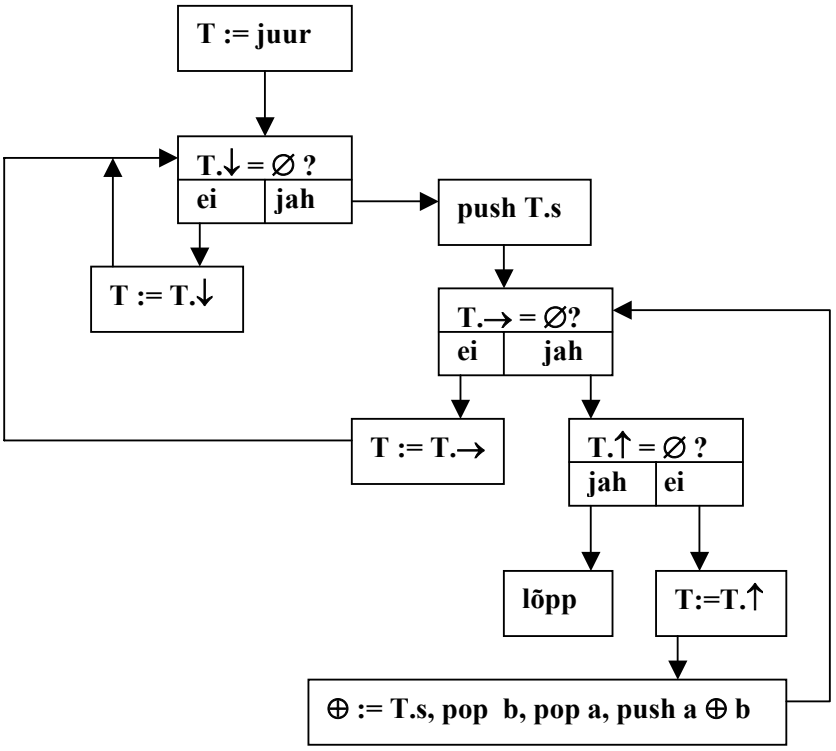
Joonisel 8.3.4c on esitatud plokkskeem aritmeetilise avaldise väärtuse leidmiseks avaldise kahendpuu läbimise käigus. Väärtuse arvutamiseks kasutatakse *LIFO*-tüüpi magasini; algoritmis me teda „nimepidi" ei kutsu, ent kasutame funktsioone *push* (pane magasini) ja *pop* (võta magasinist). „Aktiivse" tipu tähis on *T,* tipu-väljad on (vasakult paremale ja ülalt alla) T.s, T.↑, T.↓ ja T.→ ning jooksvad operandid on *a* ja *b*.

Meenutagem, et sel ajal, kui tehti esimesed *Algol*-translaatorid, ei olnud veel leiutatud süntaksorienteeritud transleerimise tehnikaid. *Infiks*- kujul esitatud avaldiste viimiseks *Poola kujule* pakkus *E. W. Dijkstra* välja teravmeelse algoritmi, mis kasutab prioriteetidega *LIFO*-tüüpi magasini ja mida saab hästi näitlikustada raudtee sorteerimisjaama analoogiga: paremalt tulevad vagunid, millede paiknemisjärjekorda tuleb rongi koostamisel muuta (tavaliselt selleks, et vahejaamades saaks osa lõpuvaguneid lihtsalt lahti haakida), otseteele koostatakse rong, vasakule keerab tupiktee, kuhu saab saata „sisendvaguneid" ja kust saab neid toimetada koostatava rongi sappa. Translaatori jaoks on rongi „lähtekoosseis" sulgavaldis *infiks*-kujul ning „lõppkoosseis" on lähteavaldise suluvaba *inverteeritud Poola kuju.*

Ülalpool, käsitledes *infiks*-avaldiste tehete prioriteete, tõime nende *Algol*-60s kasutatud tabeli (üldiselt kehtib see väheste eranditega kõigi nende keelte puhul, kus kasutatakse avaldiste esitamisel *infiks*-kuju). *Dijkstra* algoritmi kasutamiseks tuli prioriteetidega varustada lisaks tehtemärkidele ka *sulud*: aritmeetilistes ja loogilistes avaldistes tavaline tehete sooritamise järjekorda juhtiv paar „(,)", massiivi indeksite loetelu ümbritseva „[„ ja „]" ning operaatorsulud: avav *if* ning sulgevad *then* ja *else.* Kõik asjakohased prioriteedid on esitatud tabelis 8.3.4a, kus veerus *p* on prioriteet: mida suurem see number on, seda kõrgem on vastava sümboli magasiniprioriteet (vt. [32], lk.137).

p	**eraldajad**
0	([if
1	:=)] , then else
2	⊃
3	∨
4	∧
5	¬
6	> ≥ = ≠ ≤ <
7	+ –
8	× / ÷
9	↑

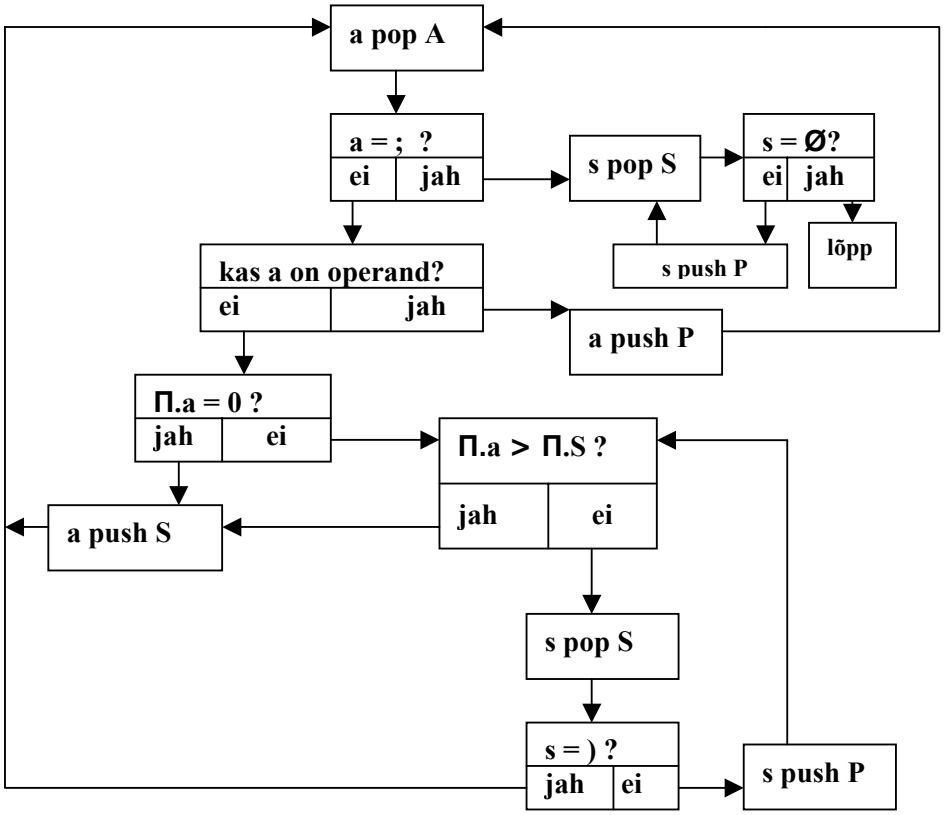
Tabel 8.3.4a**.** *Algol-60* avaldistes kasutatavate eraldajate prioriteedid.



Joonis 8.3.4c**.** Aritmeetilise avaldise väärtuse arvutamise algoritm.

Järgnevatel joonistel esitame *Dijkstra algoritmi* plokkskeemi (joonis 8.3.4d) ja seda algoritmi järgiva teisenduse pildi (joonis 8.3.4e). Kasutame kolme magasini: *FIFO*-tüüpi magasinis *A* on *infiks*-avaldis lõputunnusega „;" , *LIFO*-tüüpi magasini *P* kogutakse avaldise inverteeritud Poola kuju ning eraldajate töötluseks kasutatakse samuti *LIFO*-tüüpi magasini *S*. Operaator *a pop A* tähendab, et *A*-st eemaldatakse tipmine element ja selle väärtus omistatakse muutujale *a*, ning *a push S* tähendab, et *a* kirjutatakse magasini *S* tipmiseks elemendiks. ⊓.*a* on *a* prioriteet (kui *a* on eraldaja) ning ⊓.*S* on magasini *S* tipmise elemendi prioriteet.

Joonisel 8.3.4e on magasin *A* tupikust (magasin *S*) paremal ning magasin *P* vasemal; „vaguneid" liigutatakse paremalt vasakule – kui nad on operandid, ja tupikusse ning sealt välja vasakule, kui prioriteedid, lõpetav sulg või lõputunnus seda nõuavad.



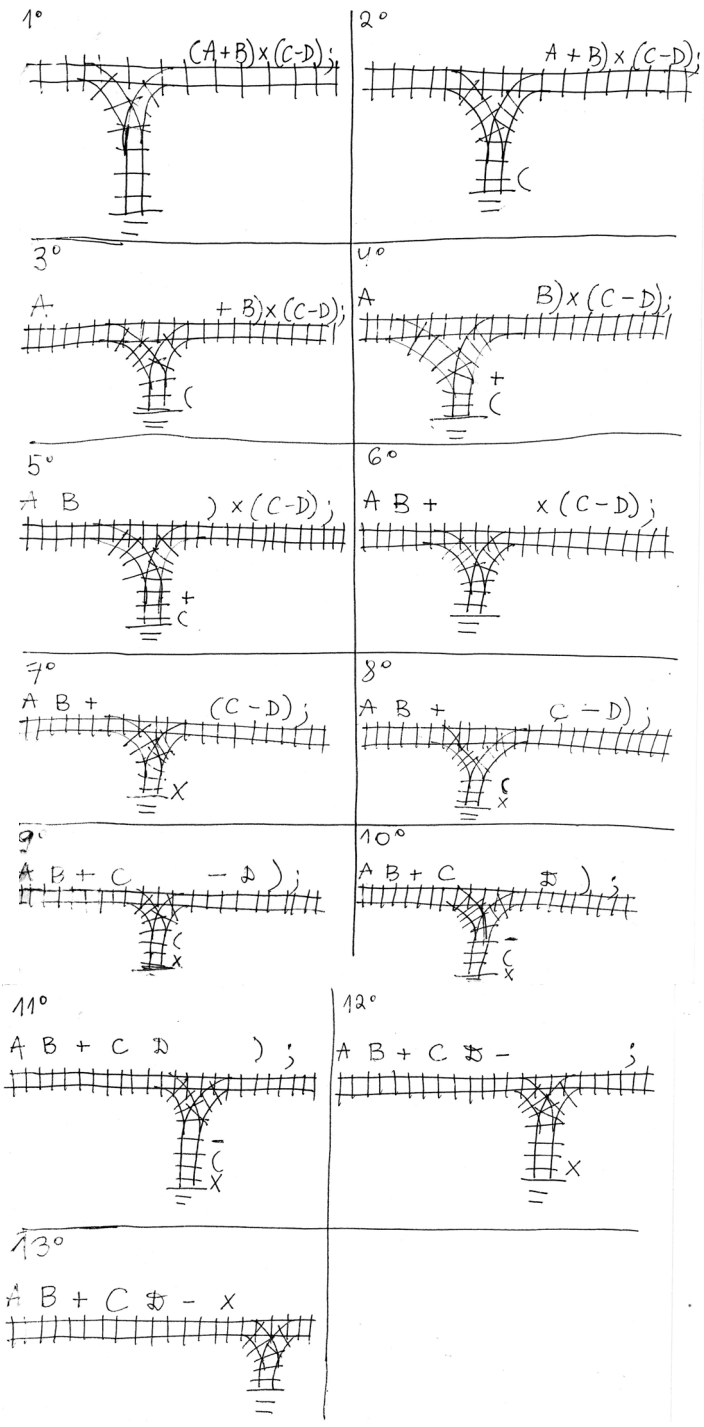
Joonis 8.3.4d. *Dijkstra* algoritm.

8.4. „Hea süntaksi" kriteeriumid[1]

Meenutagem, et üldotstarbelisi, universaalseid programmeerimiskeeli (masinkoodid, assemblerid ja protseduurorienteeritud keeled) on loodud palju, eriti möödunud sajandi 60-ndatest aastatest alates kuni mikroarvutite tulekuni. Juba 70-ndatel aastatel uuriti ja võrreldi protseduursete keelte süntaksit ning kujundati välja seisukohad, mis on süntaksi puhul oluline ja milliseid seiku tuleks arvestada uute keelte disainimisel. Oluliseks peeti:

Programmi teksti ***loetavus***. Ideaalne keel peaks olema *isedokumenteeriv*, so. selline, mille teksti kommentaare lisada polekski vaja. Üpris ideaalilähedaseks võib pidada *Algol-60*t – mis on ka mõistetav, kuivõrd ta loodi spetsiaalselt algoritmide publitseerimiseks. Siiski, ka *Algol*is on kohti, kus võib isegi kommentaaridest väheks jääda (vt. huvitavat tsüklioperaatorit *Algol*it tutvustavas jaotises). Seevastu on eriettevalmistuseta praktiliselt loetamatud näiteks *FORTH*, *APL* või *Lisp*. Arvamused *COBOL*i kohta lähevad lahku: matemaatikaharidusega inimeste ja muudes keeltes programmeerijate jaoks on ta raskesti loetav, humanitaaride (sh. majandustegelaste) ja elukutseliste *COBOL*-programmeerijate

[1] Selles jaotises on paljuski tuginetud *Terrence W. Pratt*i monograafiale [44, lk. 309 – 314].



Joonis 8.3.4e. Avaldise viimine inverteeritud *Poola kuju*le.

jaoks on see keel arusaadavam näiteks *C*-keelest. Loetavust soodustavateks loetakse järgmised seigad:

- *Operaatorite loomulikud formaadid ja struktuursus* Näiteks tähendab see, et kasutatavad võtmesõnad on (inglise keelest aru saavale lugejale) semantiliselt mõistetavad ja (ka liitoperaatori komponentide) mõjupiirkond on lihtsalt hoomatav; seda soodustab näiteks operaatorsulgude kasutamine (*ALGOL*i **begin…end**, *C* sulupaar „{…}" ning võimaluse korral **goto**-operaatori vältimine (pikema teksti korral raskendab see programmi loogika jälgimist), samuti see, et hargnevas programmis oleks lihtsalt tuvastatav harude kokkusaamispunkt (näiteks *if…then… else …endif* või *switch … case…endcase case…endcase…endswitch* mõnedes keeltes). „Loomulik formaat" peaks garanteerima ka seda, et raskete semantikavigadega ent süntaktiliselt õige programm ei tunduks korrektsena (nagu võib juhtuda *Lisp*is, kus sulgude arv küll klapib, ent kaks neist on eksikombel valedes kohtades).

- *palju võtmesõnu ja „müra"* Võtmesõnade paljususe kasulikkuse hea näite leiame (usutavasti üllatuslikult) *IBM*i assemblerkeelest: direktiivile *BC* (*Branch on Condition*, kood 47_{16}) vastab terve hulk mnemokoode, näiteks *B* (tingimusteta suunamine) , *BH* (mine, kui suurem), *BNE* (mine, kui ei võrdu) jne. „müra" (i.k. *noise*, v.k. *шум*) on allakirjutanu jaoks kaheldava väärtusega, see võib jätta ekslikku mulje „masina tarkusest" ja süntaksi „lõtvusest". Müra lubavad näiteks *COBOL* ja *PL\I*: võrdväärsed on näiteks *GO* ja *GO TO* (*TO* on müra). Semantiliselt sama konstruktsiooni esitamise erinevate süntaktiliste variantide võimaldamine on asi, mida tuleks keele disainimisel vältida. Näiteks, kui *C* autorid (kes tegid keele endi jaoks) lubavad kirjutada *i = i + 1*, *i++* või *i += 1*, siis on see lihtsalt *PDP-11* võimaluste maksimaalne ärakasutamine **–** omamoodi *masinorienteeritus*, ent **–** taas näiteks – operatsioonisüsteemides (kasvõi *OS/370* või *Windows XP*) võib võrdvõimalike variantide paljusus põhjustada liigset segadust.

- *Kommentaarid tekstis.* Kommentaaride (mis translaatori jaoks on müra) lisamist võimaldavad reeglina kõik keeled alates assembleritest, nende sisu pole ei süntaksi ega ka semantika valdkond; siiski näeb hea tava ette järgida kommenteerimisel *Ockhami reeglit*[1]: nii palju kui vajalik ja nii vähe kui võimalik – et kommentaarid ei muutuks sisuldasa müraks, mis raskendab programmi lugemist.

- *Identifikaatorite piiramata pikkus.* „Perfokaartorienteeritud" (poolfikseeritud formaadiga) keelte puhul oli paratamatu, et programmi objektide nimede pikkus pidi piirduma kas 5 või 6 sümboliga – mis tingis sisuliste nimede asendamist kodeeritutega ja lugeja pidi evima dešifraatorit saamaks aru nimede semantikast. Hilisemates keeltes on nimede pikkused samuti ülaltpoolt tõkestatud, ent ebaoluliselt (tavaliselt kuni 31 sümbolit (2^5–1)). Hea programmeerimistava näeb ette, et nimed peavad olema semantiliselt kaalukad – toetama teksti isedokumenteerimist.

[1] *William of Ockham* (1288 – 1348), inglise matemaatik ja filosoof, kelle reegli üks verioon on inglise keeles "It is vain to do with more what can be done with less".

- *Mnemoonilised võtmesõnad.* Kogu programmeerimise ajaloo vältel on keelte leksika järginud inglise keelt (väheste erandite hulka kuuluvad mõned N. Liidu programmeerimiskeeled, eeskätt assemblerid, aga ka näiteks *ALGOL*i realisatsioon *АЛГАМС* või *COBOL*i *КОБОЛ*). Seega, operaatorite nimed peavad olema hästi meeldejäetavad ja lugemisel semantiliselt arusaadavad. Seda printsiipi on järgitud juba assemblerkeeltest alates.

- *Vaba formaat* − teame, kuidas mõjutas *FORTRAN*i süntaksit 80-veeruline perfokaart ja kui palju paremini loetavad on hilisemate sisendseadmete ja andmekandjatega masinate aegsed keeled.

- *Täielik andmekirjeldus.* See tähendab, et kõik programmis nimepidi mainitud objektid peavad olema kirjeldatud (määratud tüüp, ja kui tegemist on alamprogrammiga, siis sisend ja väljund). Selle nõude järgimine lihtsustab translaatori tööd, aga ka programmi lugemist: teksti algusest saame üldjuhul (on keeli, mis lubavad objekte üle defineerida) teada, millega on tegemist.

Kirjutamise lihtsus. Tavaliselt on lugemise ja kirjutamise lihtsus enam-vähem kattuvad nõuded, ja seda juba masinkoodist alates. Hea masinkood on *süstematiseeritud*: sisuliselt sama grupi käskude koodid alluvad mingile loogilisele klassifikaatorile. Näite toome (taas) *IBM/360* masinkoodist ja assemblerist, kus nihutamiskäskude mnemoonikal on hoomatav semantika (akronüümid) ja masinkoodid kuuluvad samasse lõiku: *SRL* (*Shift Right Single Logical* = 88_{16}), *SLL* (*Shift Left Single Logical,* 89_{16}), *SRA* (*Shift Right Single Arithmetic*, $8A_{16}$) ja *SLA* (*Shift Left Single Arithmetic*, $8B_{16}$). Ja kui kõrgtaseme keele süntaks järgib loetavuse printsiipi, siis üldjuhul hõlbustab see võtmesõnade ja nendega seotud struktuuride meeldejätmist, seega ka programmeerimist. Ent need kaks nõuet – kirjutamise ja lugemise lihtsus – võivad olla omavahel vastuolus: pikka teksti (näiteks *COBOL*) võib olla hõlpus lugeda, ent tüütu kirjutada (eriti, kui pole *copy- paste*-vahendeid). Ja lakooniline *FORTH* ei pruugi olla kuigi loetav.

Transleerimise lihtsus. Selle nõude lisasid keelte realiseerijad (translaatorite kirjutajad) ja see on tavaliselt vastuolus nii lugemise kui ka kirjutamise lihtsuse nõudega. Näiteks, ei *FORTH*- kui ka *Lisp*-programm pole eriti loetav ega ka liiga lihtsalt kirjutatav. Ent transleerimise aspektist ei paku kumbki keel mingeid raskusi. Samas, *COBOL*i konstruktsioon *ADD A AND B GIVING C* (so., C*:=A+B*) pole ilmselgelt kuigi hästi transleeritav.

Kahemõttelisuse (i.k *ambiguous*, v.k *разночтение*) **puudumine**: suvalisel konstruktsioonil võib olla *ainult üks* võimalik interpretatsioon. See printsiip haakub sisuliselt põhimõttega, et suvalise „asja" programmeerimiseks peab olema parajasti üks süntaktiline võimalus. *ALGOL*i ja tema järeltulijate puhul on tavaline kahemõttelisuse allikas „muutuja *A*": keel ei spetsifitseeri, kas on mõeldud *A* aadressi või *A* väärtust; asi pannakse paika keele verbaalse kirjeldusega (üks väheseid erandeid on *FORTH*, kus *A* on aadress ja *A @* sel aadressil oleva operandi väärtus). Tulenevalt *FORTRAN*i piiratud kasutatavate sümbolite arvust on sellest keelest pärit n-ö klassikaline kahemõttelisuse näide: programmi tekstis on *A(I,J)*. Lugeja ei saa aru, kas tegemist on viidaga massiivi *A* elemendile

indeksitega *I* ja *J* või funktsiooniga *A* parameetritega *I* ja *J* (kui *A(I,J)* esines avaldise operandina).

8.5. Formaalsed grammatikad

Enamiku[1] masinast sõltumatute programmeerimiskeelte süntaks on formaalselt kirjeldatav *kontekstivaba grammatika* abil. Nood grammatikad moodustavad ühe neljast võimalikust formaalsete grammatikate (ja neile põhinevate keelte) klassist *Chomsky* klassifikatsiooni järgi. *Chomsky* kasutab kõikide klasside defineerimisel samu mõisteid ja sümboolikat:

- *Mõisted* on metakeelsed sümbolid, mis tuleb defineerida. Nende hulk moodustab *mitteterminaalse tähestiku* V_N (*Nonterminal* alphabet) ─ näiteks (kui räägime programmeerimiskeeltest, siis) mõisted *programm*, *tsükkel*, *avaldis* jmt.;
- *Terminaalne tähestik* V_T (*Terminal* alphabet) on (taas programmeerimiskeele puhul) nende sümbolite[2] hulk, mida on lubatud antud keele programmis kasutada. Sellesse hulka kuuluvad:
  ─ eraldajad, näiteks **. , : ; :=** + - * / ** < > = => <= | **& % @ $** jne, ka *tühik*;
  ─ *võtme- ja reservsõnad*, näiteks **for array goto if integer procedure** jne;
  ─ *identifikaatorid* ─ muutujate ja alamprogrammide nimed, märgendid jmt.;
  ─ *konstandid*,
  ─ *stringid* (*sõned*) ja
  ─ *kommentaarid.*
  Viimase nelja lekseemiklassi elementide kirjutamise reeglid on tavaliselt fikseeritud keele informaalses kirjelduses.
- Tähestik *V* on mõistete tähestiku ja terminaalse tähestiku ühend: $V=V_N \cup V_T$,
- V_T^* on tähestiku V_T baasil moodustatavate sõnade hulk (kui tegemist on taas programmeerimiskeele grammatikaga, siis V_T^* on kõigi selles keeles kirjutatavate leksiliselt korrektsete programmide hulk):
- V^* on tähestiku *V* baasil moodustatavate sõnade hulk, mida kasutatakse keele defineerimisel (ja selles keeles kirjutatud sõnade analüüsimisel).

Ameerika lingvist *Noam Chomsky*[3] (*MIT ─ Massachusetts Institute of Technology*) üldistas 1959. aastal formaalsete keelte kontseptsiooni. Formaalsed keeled baseeruvad *generatiivsetel grammatikatel* (mõistete definitsioone kasutades võimaldavad nad genereerida kõik antud keele "sõnad" (kui tegemist on programmeerimiskeelega, siis iga "sõna" kujutab endast ühte programmi). Kasutades klassifitseerimise kriteeriumina mõistete defineerimise vormi deklareeris *Chomsky* järgmised neli klassi:

---

[1] Erandi moodustab näiteks *FORTH*, seda põhimõtteliselt, ja poolfikseeritud formaadiga keeled, näiteks *FORTRAN* või *SNOBOL* ─ raskusi valmistab formaadi formaliseerimine.

[2] Terminaalse tähestiku elemente nimetatakse ka *lekseemideks*.

[3] *Chomsky* vanemad ─ Valgevene ja Leedu juudid ─ emigreerusid 1913. aastal Ameerikasse, Baltimore'i [101]. Venemaa traditsioonis kirjutatakse *Noam Chomsky* nime *Наум Хомский* (vt. näit. [32], lk. 223), seda hääldust peaks arvatavasti eelistama: „homski" ja mitte „čomski".

Noam Chomsky (s. 12.07.28, Philadelphias)

- ***0-klass:*** Kõik reeglid on kujul $\boldsymbol{a \rightarrow b}$, kus $a,b \in V^*$. Nii saab näiteks modelleerida loomulikke keeli.
- ***1. klass:*** *kontekstitundlikud* grammatikad; reeglid on kujul $\boldsymbol{pUq \rightarrow puq}$, kus $U \in V_N$ ja p, q ja $u \in V^*$ (seejuures võib nii p kui ka q olla *tühisõna* ─ Λ).
- ***2. klass:*** *kontekstivabad grammatikad* reeglitega kujul $\boldsymbol{U \rightarrow u}$, kus $U \in V_N$ ja $u \in V^*$ See klass sobib parimini programmeerimiskeelte formaalsete grammatikate kirjeldamiseks, kuivõrd ta võimaldab süntaksorienteeritud transleerimist. Oluline on seegi, et 1. klaasi grammatika saab teisendada keelt säilitades 2. klassi grammatikaks (vt. [32], lk. 224).
- ***3. klass:*** *regulaarsed grammatikad* reeglitega kujul $\boldsymbol{U \rightarrow u}$ või kujul $\boldsymbol{U \rightarrow uU'}$, kus U, $U' \in V_N$ ja $u \in V_T^*$. Neile baseeruvate keelte sõnadeks on *regulaaravaldised,* mida analüüsitakse *lõplike olekute hulkadega automaatide* abil. Programmeerimiskeeltes võib regulaarse grammatikaga kirjeldada lekseemiklasse *identifikaatorid*, *konstandid*, *stringid* ja *kommentaarid*.

8.6. Formaalne süntaks

Esimene teadaolev programmeerimiskeele süntaksi formaalne esitus pärineb aastast 1960, siis töötas *John Backus* (mõnedel andmetel koos *Peter Naur*iga) välja metakeele, mida meie tunneme akronüümi *BNF*-nimelisena. Seda nimelühendit tõlgendatakse tavaliselt kui *Backus ─ Naur Form*, ent varasematel aegadel ka kui *Backus Normal Form.*

Autori(te) eesmärk oli välja töötada täpne ja ühemõtteline programmeerimiskeele süntaksi esitamise meetod nii keele realiseerijatele kui ka selles keeles programmeerijate jaoks. Verbaalne kirjeldus võib olla kuitahes hästi läbimõeldud, ent mitmeti tõlgendamist on raske vältida. *Algol*-60 süntaksi formaliseeris *BNF*i abil *P. Naur* (1963), ühe fragmendi sellest esitame allpool *BNF*i näitena.

Tuletagem meelde, et *N. Chomsky* avaldas oma generatiivsete grammatikate klassifikatsiooni 1959. aastal; *BNF*i defineerimisvahendite sisuline ekvivalentsus kontekstivabade grammatikate klassiga on silmnähtav.

Ent *BNF* osutus õige pea sootuks tähtsamaks kui pelgalt keelte süntaksi ühemõttelise kirjeldamise vahend: see andis võimaluse *süntaksorienteeritud translaatorite* konstrueerimiseks, ning järgmise sammuna *kontekstivabade grammatikate klassile* orienteeritud *TTS*ide realiseerimiseks.

Süntaksorienteeritud translaatorid kasutavad tavaliselt originaal-*BNF*i asemel viimase erindit ─ *produktsioonide keelt*, mille töötlemine on hõlpsamini programmeeritav.

Süntaksi esitamise metakeeli on välja töötatud üsnagi palju; meie tutvustame lisaks *BNF*ile veel kahte: *CBL*i (*CoBol-Like grammar*) ja *Wirthi skeeme.* Esimest neist kasutati näiteks *COBOL*i ja *PL\I* , *IBM*i operatsioonisüsteemi makrode, andmebaasisüsteemide andmekirjeldamis- ja päringukeelte (näiteks *SQL*) jmt. süntaksi kirjeldamiseks. *CBL* on inimesele kahtlemata loetavam (seega ka arusaadavam) kui *BNF*, ent süntaksorienteeritud translaatori jaoks tuleb *CBL*-süntaks tõlkida produktsioonide keelde (sj. käsitsi, mitte programselt).

Wirthi skeemid on veelgi enam loetavam ja arusaadavam kui *CBL* ja veelgi vähem kasutav süntaksorienteeritud translaatorile kui *CBL*. *Niklaus Wirth* kasutas seda moodust 1973. aastal avaldatud monograafias, mis kirjeldas keelt *Pascal.*

8.6.1. *BNF*

*BNF*i metakeel on järgmine: *mõisted* (*Chomsky* järgi mitteterminaalse tähestiku sümbolid) kirjutatakse nurksulgude vahele ─ näiteks <aritmeetiline avaldis>, märki ::= loetakse „on definitsiooni kohaselt" ning alternatiivid eraldatakse püstkriipsudega „|" (loetakse „või"). Terminaalse tähestiku tähed kirjutatakse definitsiooni „loomulikul kujul".

Laename *Terrence Pratt*ilt näite *Algol*-60 omistamisoperaatori defineerimisest *BNF*i abil [44, lk. 335].

<omistamisoperaator> ::= <muutuja> := <aritmeetiline avaldis>

<aritmeetiline avaldis> ::= <term> | <aritmeetiline avaldis> ↑ <term> | <aritmeetiline avaldis> × <term> | <aritmeetiline avaldis> + <term>

<term> ::= <üksliige> | <term> ─ <üksliige> | <term> / <üksliige>

<üksliige> ::= <muutuja> | <arv> | (<aritmeetiline avaldis>)

<muutuja> ::= <identifikaator> | <identifikaator> [<indeksite loetelu>]

<indeksite loetelu> ::= <aritmeetiline avaldis> | <indeksite loetelu> , <aritmeetiline avaldis>

Produktsioonide keel on *BNF*i invariant: märk „::=" on asendatud sümboliga „→", mõisted võivad olla teisiti esile tõstetud kui nurksulupaari abil ning alternatiivid esitatakse sama mitteterminali korduva defineerimisega. Toome näiteks mõiste „term" defineerimise produktsioonide keeles:

<term> → <üksliige>

<term> → <term> ─ <üksliige>

<term> → <term> / <üksliige>

8.6.2. *CBL*

Selle variandi töötas välja USA komitee *CODASYL* (*Conference On DAta SYstems and Languages*) [9]; selle organisatsiooni produkt on ka *COBOL* (*COmmon Business-Oriented Language*).

Metakeel on järgmine: võtmesõnad kirjutatakse suurtähtedega, sj kohustuslikud on allajoonitud ja fakultatiivsed („müra") pole.

CBL kasutab sulupaare „( )", „[ ]", „{ }" ja „|| ||" ning kolm punkti „…" tähistavad neile eelneva konstruktsiooni (võimalikku) kordumist. Sulgude semantika on järgmine:

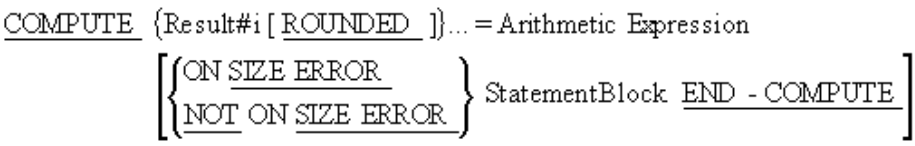
(1 2 3) … tähendab, et võib kirjutada „1 2 3" või „1 2 3 1 2 3" jne.

⌈1 ⌉ tähendab, et nii 1 kui ka 2 on fakultatiivsed: mitte midagi | 1 | 2
⌊2 ⌋

⎧1⎫
⎨2⎬ alternatiivid: 1 | 2 | 3
⎩3⎭

|| 1 || kas 1 2 | 2 1
|| 2 ||

Toome näiteks *COBOL*i operaatori *Compute* süntaksi [102]:



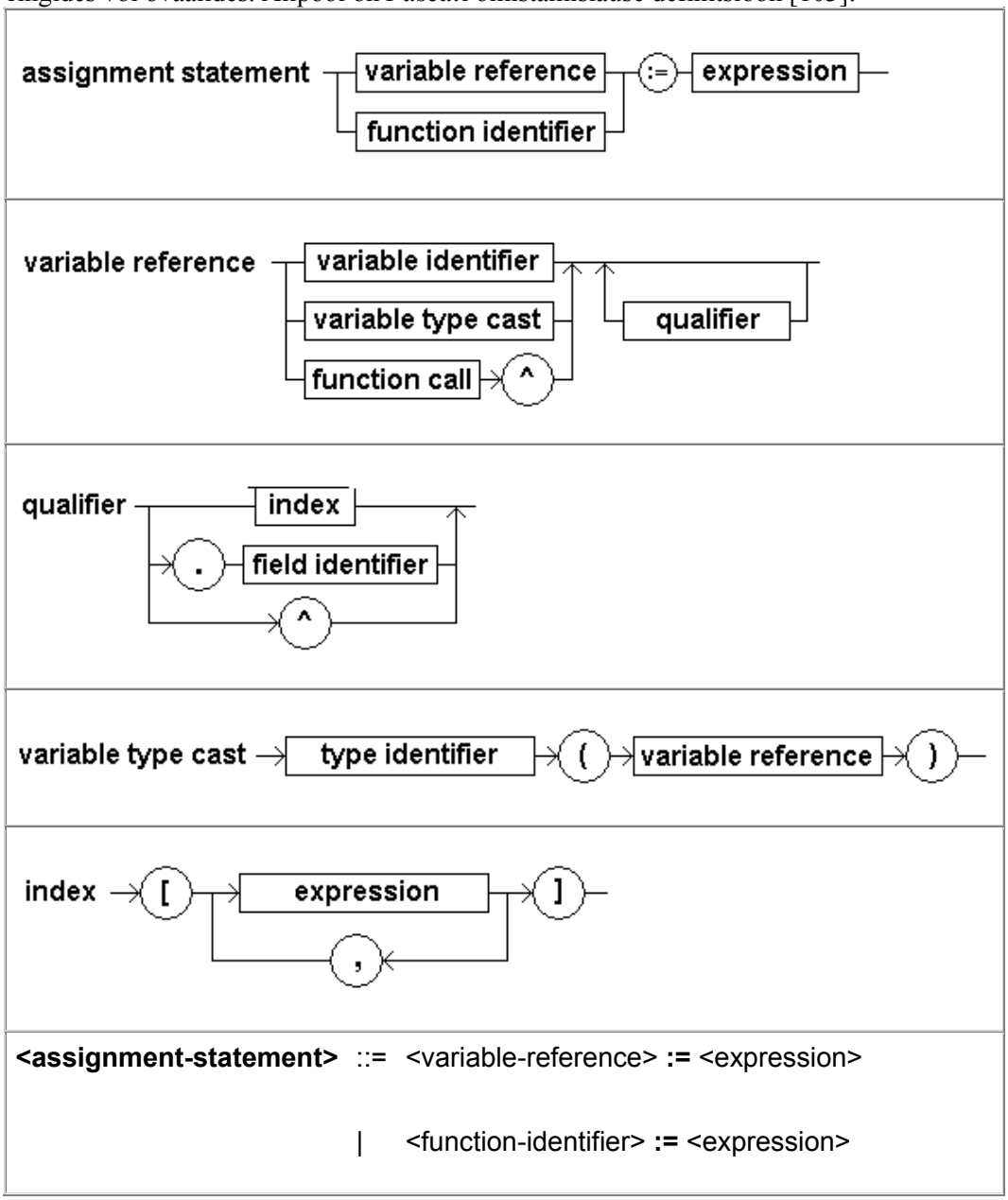
Programmis võib ülalkirjeldatud operaator välja näha järgmiselt:

```
COMPUTE Result1 ROUNDED, Result2 = ((9*9)+8)/5
```

Result1 väärtuseks saab 18 ning *Result2* väärtuseks 17.8.

8.6.3. Wirthi skeemid

Wirthi skeemide metakeel on järgmine: defineeritav mõiste on vasakul, definitsioon on esitatud plokk-skeemina; mõisted on ristkülikutes ning terminaalse tähestiku sümbolid ringides või ovaalides. Allpool on *Pascal*i omistamislause definitsioon [103]:



Samast pärineb asjakohane näide: 7! arvutamise programm.

```
PROGRAM Main (Output);

FUNCTION Factorial (N: Integer): Integer;
  BEGIN
    IF N > 0 THEN
      Factorial := N * Factorial (N - 1)
    ELSE
      Factorial := 1
  END;

VAR
  I: Integer;
  J: Integer;

BEGIN
  I := 7;
  J := Factorial (I);
  writeln ('I = ', I);
  writeln ('J = Factorial (', I, ') = ', J);
END.
```

Output

```
I = 7
J = Factorial (7) = 5040
```

8.7. Süntaksorienteeritud transleerimine

Esimesed andmed süntaksorienteeritud (seejuures orienteeritud mingile kontekstivabade grammatikate alamklassile) pärinevad 1960. aastast – *Tony Brooker* (Manchesteri Ülikool, Inglismaa, asut. 1824, umbes 35 000 üliõpilast) kirjutas teadaolevalt esimese süntaksorienteeritud translaatorite üldise süsteemi. Noil aegadel pidi transleerimisresultaat olema (meie mõistes) .exe-fail, seega täidetav masinkoodiprogramm. See oli ideaal, selle tulemuseni süntaksorienteeritud tehnikad (kuhu lisati ideaali saavutamiseks "semantilised atribuudid") arvestataval tasemel ei jõudnud. Ent *Translaatorite Tegemise Süsteemid* (i.k. *Compiler Compiler* – *Tony Brookeri* termin – , v.k. *Система Построения Трансляторов – СПТ*)[1] arendati järgmise paarikümne aasta jooksul välja (eeskätt teoreetilisel tasemel, tööstuslikud kompilaatorid on reeglina tehtud „põlve otsas", orienteerudes ühelt poolt keelele, teiselt poolt protsessorile) üpris arvestataval tasemel. Kuivõrd võrdväärseid analüüsimeetodeid oli palju: ülalt-alla-tehnikad, alt-üles-tehnikad, erinevad kontekstiarvestamise tehnikad jne[2]., siis mingit *TTS*-standardit ei kehtestatud. Tartu Ülikoolis on loodud mitu eelnevusgrammatikatele (*N. Wirth*) põhinevale alt-üles-realisatsioonile (mille eeliseid ülejäänute ees on raske tõestada) *TTS*i

[1] Hea translaatorite kui eriliste programmide käsitlus on *Jaak Henno* raamatus [19, lk. 42 – 44].
[2] Nende kaante vahele ei mahu nonde tehnikate käsitlus, sestap soovitame huvilistele otsida asjakohast informatsiooni iseseisvalt.

Kui esimesed *TTS*id üritasid kas objektkeelena väljastada masinkoodi (see ei õnnestunud normaalse aja- ja töökuluga, lihtsam oli *TTS*-väljund käsitsi kodeerida), siis tänapäevased *TTS*id on orienteeritud kasutama väljundkeelena (*objekt*keelena) kõrgtaseme keelt. Ülalesitatust on meil üks analoog: 80-ndate alguses tehti TRÜ AK-s translaator *Modula-2* $\rightarrow$ *FORTH*[1]. Interneti andmetel [115] on *kõik* tänapäevased *TTS*id tasemel *kõrgtaseme keel* $\rightarrow$ *kõrgtaseme keel* (á la *Modula-2* $\rightarrow$ *FORTH*), kusjuures reeglina kasutatakse vahekeelena *baitkoodi* (i.k. *byte-code*), omamoodi universaalse abstraktse arvuti (baitmasina) masinkoodi[2].

Niisiis, meenutagem, et transleerimine on programmi P (kirjutatud keeles L_1) teisendamine temaga ekvivalentseks programmiks keeles L_2, kusjuures traditsiooniliselt L_2 tase oli madalam kui lähtekeele oma. Möödanikus oli keeleks L_2 masinkood, hiljem reeglina *assembler*. Tänapäeva *TTS*i väljundkeelteks on pigem[3] *kõrgtaseme keeled*: *C*, *C++*, *C#*, *Java*, *Python*, *Delphi* jmt [115], millest saadakse interpreteeritavad andmestruktuurid või kompileeritud *.exe*-fail juba objektkoodi realiseerivate vahenditega.

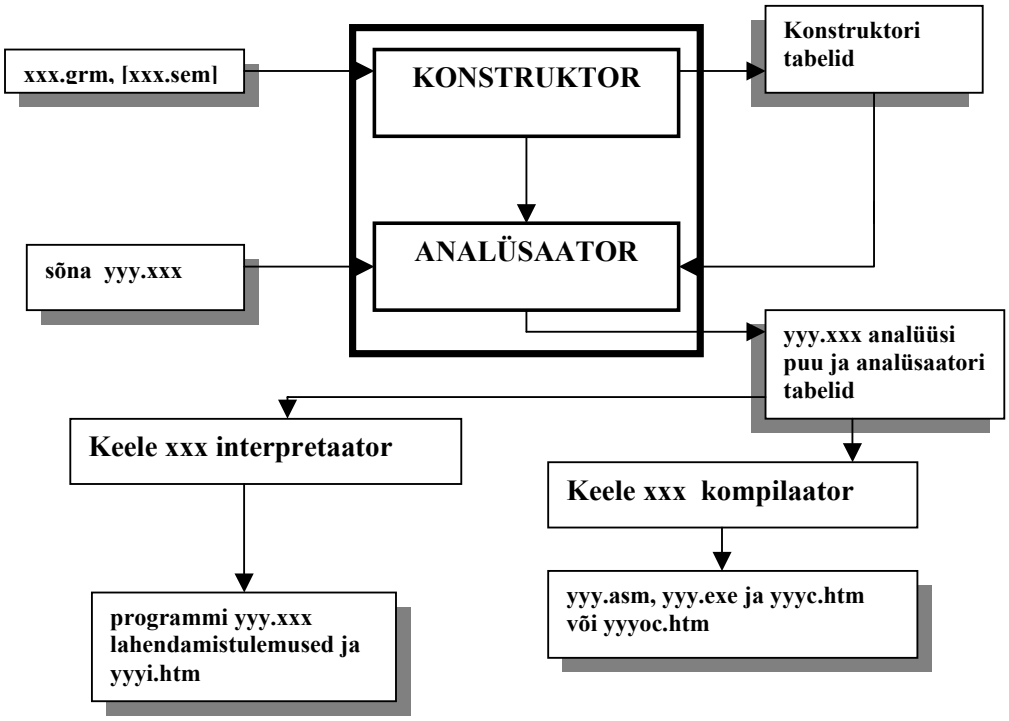
*M. Tombak*u eestvedamisel tehti T(R)Üs translaatorite tegemise süsteemid arvutite *Minsk-32, EC-1020, CM-4 ja PC XT* jaoks. Noile süsteemidele tuginedes kirjutas nende ridade autor *TTS*i kaasaegse personaalarvuti jaoks; süsteemi esimene versioon valmis 1998. a. *MS-DOS*i 16-bitises keskkonnas (kasutades graafilise liidese tegemiseks *Stan Milam*i vabavaralist menüüde tegemise süsteemi), teine versioon – *Windows*i keskkonnas – valmis paar aastat hiljem tollase tudengi *Gunnar Kudrjavets*i tungiva soovituse ja asjaliku teeleaitamise ajendusel (2001.a. esimene ja 2004.a. teine versioon). See *TTS* on kasutusel õppematerjalina loengukursuse "Automaadid, keeled ja translaatorid" (MTAT. 05.085) jaoks ning on kättesaadav (*e*-manuaal, programmid jpm) autori koduleheküljelt (http://www.cs.ut.ee/~isotamm).

Translaatori tegijale pakub süsteem "programmeerimisvaba" tuge kahe plokiga: need on *Konstruktor* ja *Analüsaator*. Esimene testib etteantud grammatikat (kas on eelnevusgrammatika? kas on pööratav? kas on *BRC*-redutseeritav?)[4], saades ette grammatika produktsioonide hulga P ning väljastades asjakohased tabelid. Ideaalis töötab *Konstruktor* üks kord iga uue grammatika jaoks; reaalselt läheb vaja rohkem "jooksutamisi". Sisulisel tasemel peab translaatori kirjutaja sekkuma ainult siis, kui derivatsioonist sõltuv kontekst ei eristu, kõik muu (välja arvatud näiteks produktsioonide hulga esitamisel tehtud näpuvead) teeb *Konstruktor* kui programm.

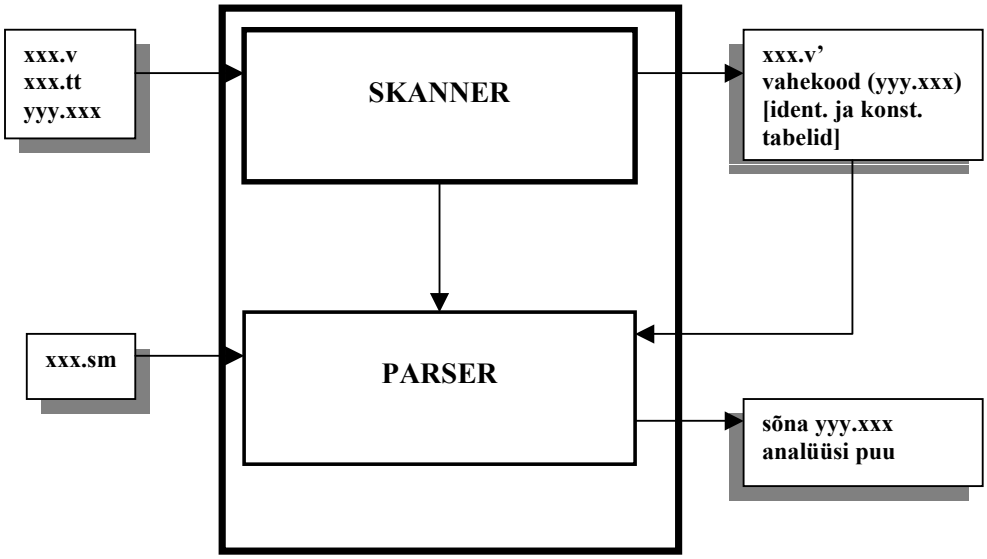
Teine plokk – *Analüsaator* – saab ette *Konstruktori* väljundtabelid ning lähtekeelse *programmi*, väljundiks on kas veateated või andmestruktuurid: analüüsi hõre puu ning identifikaatorite ja konstantide tabelid. Ka see plokk on tagatud *TTS*iga – programmeerija ei pea koodi kirjutama.

---

[1] Vt. [55].
[2] Me käsitlesime universaalseid vahekeeli 7. peatükis.
[3] Paratamatult on üldised vahendid ebaefektiivsemad. Konkreetse keele jaoks assembleris kirjutatud kompilaator on alati efektiivsem (töötab kiiremini ja genereerib kompaktsema masinkoodi) kui *TTS*i tehtu. Ent *TTS* hoiab kokku (tänapäeval juba) kalleimat ressurssi – kvalifitseeritud programmeerijate tööd.
[4] Eitavate vastuste korral leiab süsteem ise lahendused.



Joonis 8.7a. *TTS*i üldine skeem.



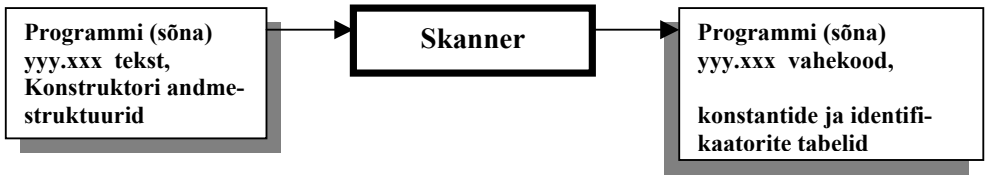
Joonis 8.7b. Analüsaatori plokid.

Kolmas plokk tuleb programmeerida: kirjutatakse kas interpretaator või kompilaator; mõlemad saavad ette analüsaatorilt programmi hõreda puu ning konstantide ja identifikaatorite tabelid.

8.7.1. Skanner

Meenutagem, et transleerimiseks antud programmi ("sõna") teksti võime vaadelda kui *lekseemide* (terminaalse tähestiku elementide) jada, kusjuures nood lekseemid on kas *reserv- või võtmesõnad*, *eraldajad* või *lekseemiklasside* (identifikaatorid, konstandid, stringid ja kommentaarid) esindajad.

Skanner on programm, mis peab esiteks tuvastama lekseemid ja teiseks, väljastama vektori, kus iga lekseem (algselt *tekst*) on asendatud *koodiga*, millega opereerivad kõik järgnevad algoritmid. Lisaks peab ta reaalse (programmeerimiskeele) translaatori koosseisus tuvastama lekseemiklasside objektid ning moodustama nende jaoks asjakohased andmestruktuurid (nagu näiteks identifikaatorite ja konstantide tabelid).



Joonis 8.7.1a. Skanneri töö.

Ühesed koodid paneb paika *Konstruktor* reserv- ja võtmesõnadele ning eraldajatele, ent mitte *lekseemiklassidele*[1]. Terminaalses tähestikus on fikseeritud *lekseemiklassi* kood (näiteks *Trigol*is, et suvalise *identifikaatori* kood on 4), ja relatsioonid on fikseeritud *klassi* jaoks. Samas, translaatorit huvitab, milline on konkreetse antud klassi kuuluva elemendi kirjeldus. Sestap on üheste lekseemide kood ühesümboliline, ent klassiesindajate oma kahesümboliline: esimene on klassi kood, teine aga "uue terminali" kood (see omistatakse alates "esimesest vabast koodist", so. viimase mitteterminali kood + 1).

Niisiis, *Skanner* peab tegema kaht kardinaalselt erinevat tööd: esiteks, tuvastama sisendtekstis (lähteprogrammis) terminaalse tähestiku konkreetseid sümboleid, ja teiseks – kui järjekordne lekseem *pole* vahetult äratuntav sümbol, siis peab *Skanner* otsustama, millisesse lekseemiklassi ta kuulub ning menetlema teda programmiga dikteeritud korras.

Klassikas ([1], [35] jt) soovitatakse *Skanner* alati häälestada kui automaat, mis interpreteerib *regulaarsete grammatikate* poolt defineeritud keeltes kirjutatud *regularavaldisi* (so, eeskätt identifikaatoreid ja konstante):

[1] Mõistagi, *kommentaarid* kuuluvad lihtsalt eiramisele.

8.7.2.Parser

Tuletame meelde, et *Parser* (sisuldasa süntaktiline analüsaator) on meie *Analüsaatori* teise ploki tinglik nimetus. Esimene plokk, *Skanner*, on leksika-analüsaator, mis teisendab põhitööna sisendprogrammi vahekoodi; mis on *Parser*i sisendkeeleks.

Analüsaator on võimeline leidma kõiki süntaksivigu ning veasituatsioonis tuleb valida mingi sisseprogrammeeritav strateegia töö jatkamiseks, et analüsaator vaataks läbi kogu sisendteksti ning püüaks avastada võimalikult palju olemasolevatest süntaksivigadest.

Ideaalne oleks, kui Analüsaator *parandaks* avastatud vead nii, et resultaadiks oleks nii süntaktiliselt kui ka semantiliselt korrektne programm, mis teeks seda, mida programmeerija *mõtles* ja mitte seda, mida ta ei soovinud (mis võib juhtuda, kui jätta süntaksivigade *parandamine* translaatori pädevusse). Ilmselt on ideaal siingi saavutamatu, enamgi veel, kõikide süntaksivigade programne *parandamine* tundub samuti mõeldamatuna. Erandiks on ortograafiavead reservsõnades: näiteks, kui *BEGIN* asemele on kirjutatud *PEGIN* või *BEGINN*, siis oleks mõeldav vea parandamine. *D. Gries* [16, lk. 354 – 356] refereerib *D. Freeman*i artiklit "*Error correction in CORC: the Cornell computing language. Thesis, Cornell University, 1963*", kus on osutatud neljale tüüpsituatsioonile:

1. üks täht sõnas on valesti kirjutatud (või perforeeritud);
2. üks täht on vahele jäänud;
3. üks täht on liigne;
4. kaks kõrvutiolevat tähte on vahetuses.

Gries möönab, et ortograafiavigade parandamine on tõenäosem reservsõnade ja eraldajate puhul ning kahtlasem identifikaatorite puhul. Tegelikult on ortograafiavigade võimaliku avastamise "koht" Skanneris ja konkreetsele keelele orienteeritud Analüsaatoril on seal kahtlemata võimalusi. Süntaksorienteeritud Analüsaatoril need paraku puuduvad[1].

Niisiis, Parseri ülesanne on fikseerida viga ja püüda vältida analüüsi jatkamisel sekundaarseid vigu – need väljenduvad tegelikult *olematute* süntaksivigade leidmises programmist ajal, mil Parser püüab neutraliseerida tegelikku viga.

8.7.3. Translaator

Translaator on programm, mis teeb sisendprogrammi *lahendatavaks.* Translaator saab ette Analüsaatori väljundi ning interpreteerib noid andmestruktuure; meie kontekstis tähendab see, et edasine tuleb iga uue keele jaoks programmeerida "käsitsi"[2] ning keele realiseerijatel on valida kahe variandi vahel: kas programmeerida *interpretaator* või *kompilaator.*

[1] Näiteks, oletagem et *Algol*-tüüpi keele Skanner leiab sõna *begun* ja parandab selle *begin*iks – ent programmeerija mõtles kasutada venekeelset muutujat tõlkenimega "jooksja". Segadust kui palju…

[2] Alternatiivi näitena meenutame translaatorit *Modula-2* $\rightarrow$ *FORTH.* Või transleerimine universaalsesse vahekoodi.

Et neid teid on eelnevalt juba korduvalt kirjeldatud ja võrreldud, siis tutvustame allpool variante meie *TTS*i näitekeele *Trigol*[1] baasil.

Niisiis, Translaator *interpreteerib* sisendprogrammi analüüsi puud, mille "ehitas" *süntaktiline Analüsaator*, mis garanteerib tolle programmi *süntaktilise* õigsuse, so. vastavuse produktsioonidega antud süntaksireeglitele. Ent peale nende reeglite on keele mitteformaalse kirjeldusega fikseeritud tavaliselt palju kitsendusi, mida süntaksiga ei saa esitada. Näiteks: märgendid peavad olema unikaalsed, suunata saab ainult olemasolevale märgendile või et muutujatele ei ole mõtet omistada väärtusi, mida programmis ei kasutata. Sellised seigad seonduvad mõistega *semantiline analüüs*, ja sellega tegeleb Translaator.

8.7.3.1. Interpretaator

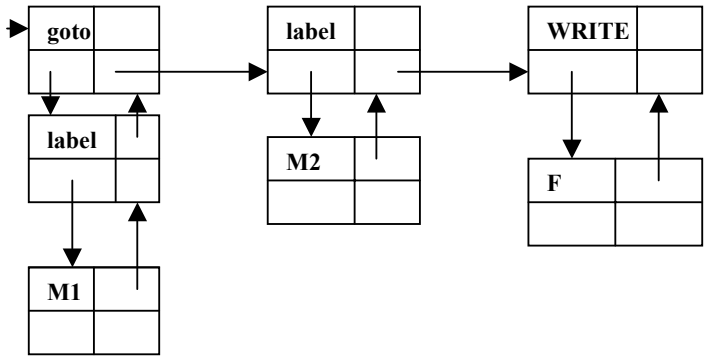
*Trigol*i interpretaatori põhitöö on sisendprogrammi analüüsi puu läbimine ja selle käigus resultaatide arvutamine. Ent sellele põhitööle eelneb märgendite ja suunamiste töötlus, mille käigus kontrollitakse märgendite unikaalsust ja seda, kas kõik suunamised on olemasolevatele märgenditele, modifitseeritakse identifikaatorite tabelit ning teisendatakse analüüsi puud. Võtkem näiteks programm *P6.tri* (mida kasutasime näitena juba osas 2.5.4.4. illustreerimaks *Intel*i assemblerit ja masinkoodi):

```
#READ n; F:=1; I:=0;
M1: I:=I+1;
IF I>n THEN GOTO M2;
F:=F*I;
GOTO M1;
M2: WRITE F#
```

Selle programmi kahele viimasele reale vastav analüüsi puu fragment on kujutatud joonisel 8.7.3.1a. Loodetavasti lugeja nõustub, et interpretaatori jaoks pole sel joonisel esitatud puu kuigi ratsionaalne: esiteks, "goto" alampuus pole midagi mõistlikku teha vahetasemel "label" ning tipus "M1" tuleks hakata tegelema nii märgendatud operaatori otsimisega, ja teiseks, "labeli" alampuu raiskab ainult interpretaatori aega[2]. Sestap teisendatakse analüüsi puud ning info märgendatud objektide kohta salvestatakse identifikaatorite tabelisse (vt. joonis 8.7.3.1b): puu (fragmendid) on joonise ülaosas ning identifikaatorite tabeli asjassepuutuv fragment on joonise alumises servas.

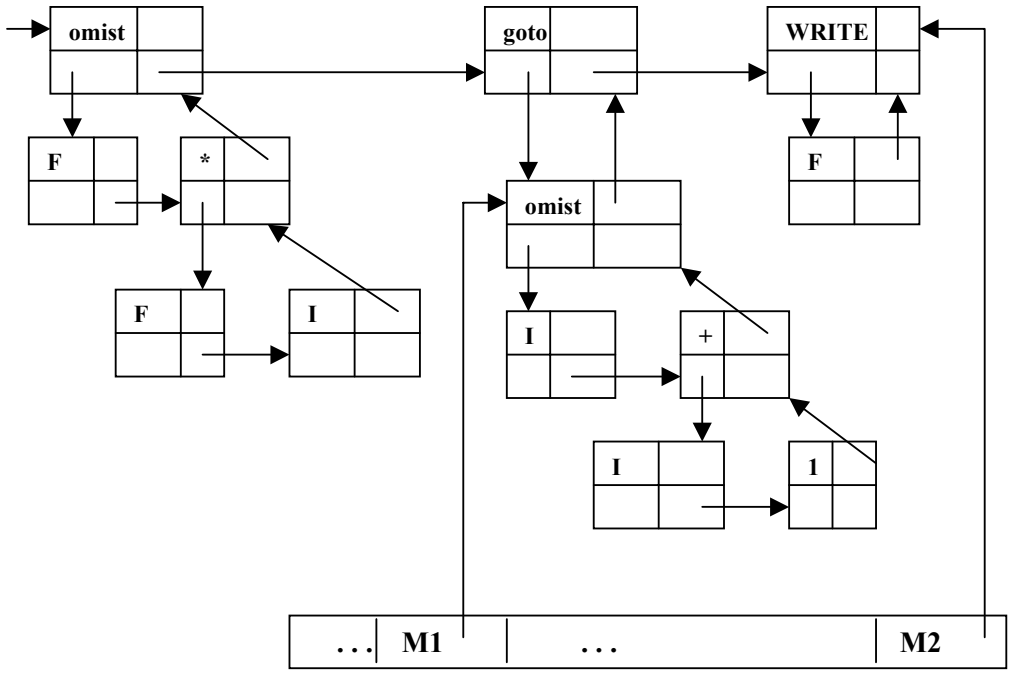
[1] *Trigoli* süntaksi ja analüüsi puu moodustamise juhtimise failid on lisades 8 ja 9**.**

[2] Tegemist pole analüüsi puu moodustamise juhtimise (fail *tri.sem*) veaga: Analüsaator peab tuvastama *märgendi* nii operaatori nime kui ka suunamisoperaatori argumendi rollis, seetõttu on paratamatu, et mitteterminali *label* definitsioonil on semantika.



Joonis 8.7.3.1a**.** Suunamine ja märgendid analüüsi puu fragmendis.

Näeme, et “goto”-tipu alluv-viit osundab märgendiga *M1* tähistatud omistamisoperaatorile, operaatoriga “WRITE” seonduvat “label”-alampuud enam ei ole ning et identifikaatorite tabeli elemendid *M1* ja *M2* on saanud *väärtused* – neiks on märgendatud operaatorite aadressid.



Joonis. 8.7.3.1b. Modifitseeritud andmestruktuurid.

Sellise tulemuse saavutamiseks tuleb programmi analüüsi puust esmalt leida märgendatud operaatorid ja kanda nende aadressid identifikaatorite tabelisse (kui selgub, et märgendil juba on väärtus, väljastatakse veateade ja modifitseeritakse vigade indikaatorit), seejärel

− kui identifikaatorite tabelis on kõikide märgendite väärtused − otsitakse puust kõik suunamisoperaatorid ja kontrollitakse, kas on kasutatud väärtustatud märgendit (kui ei, fikseeritakse viga) ja seejärel − kui eelmistel sammudel vigu ei leitud − teisendatakse analüüsi puu interpreteerimiseks mugavale kujule.

Märkida tuleks vist sedagi, et *puu* pole rangelt võttes kasutatava andmestruktuuri jaoks kohaldatav termin − meie struktuur on *tsükliline graaf*: enne teisendamisi rikkusid puustruktuuri *üles*-viidad, teisendused lisasid aga lisaviidad suunamisoperaatoritega määratud tippudele.

Interpretaator liigub analüüsi puul *lõppjärjekorras*; tänu *üles*-viidale ei kasutata ei rekursiooni ega läbimismagasini. Puu lehtedes olevate andmete jaoks kasutatakse *LIFO*-tüüpi magasini, mille tipust saadakse operand(id) aritmeetika-, võrdlus-, sisestamise, väljastamise ja suunamisoperaatoritele (nonde koodid on puu mitterippuvates tippudes) ja neid täidetakse "üles" liikudes.

8.7.3.2. Kompilaator

Tuletagem meelde: Interpretaator läbib lähteprogrammi analüüsi puud lähtudes tolle programmi loogikast (suunamised, tsüklid jmt.) üldjuhul tsükliliselt, ja lõpetab oma töö interpreteeritava programmi resultaatide fikseerimisega (väljastamise või salvestamisega). Kompilaator seevastu läbib − võimalik, et pärast eeltöötlust − analüüsi puu ja genereerib selle käigus lähteprogrammiga adekvaatse koodi mingis teises (reeglina madalama taseme) keeles. Kuni objektmasinate koodid olid "lihtsad ja läbipaistvad" (nagu *Minsk-32* või *IBM/360* puhul), siis kompilaatorite väljundiks oli reeglina *masinkood*. Tänapäeval (masinkood on "ülekattega" nagu *Intel*il) on kompilaatori objektkoodi rollis tavaliselt *assembler*[1]. Nii on see ka meie *TTS*i *Trigol*-kompilaatori puhul: objektkood on *Intel*i assembler.

Interpretaatorit realiseerides pidime tegelema mõningate (ainult mõningate, sest *Trigol* on triviaalne keel) semantiliste probleemidega − ent Kompilaator võib eirata üsnagi paljusid Interpretaatori probleeme: sisuldasa need jäävad, ent nende lahendamine on mugav jätta assembler-translaatori hooleks (kui *TTS*i Kompilaatori objektkood on assembler, siis kompileeritud assemblertekst antakse üle assembler-translaatorile ning semantikavigade[2] avastamine jääb juba viimase kompetentsi).

Sellest hoolimata on kasulik teha mõningane transleeritava programmi Analüsaatorilt saadud väljund-andmestruktuuride kompileerimiseelne modifitseerimine. Minimaalselt tuleks teha järgmist:

- teha kindlaks[3] vajalike töömuutujate arv − töömuutujad on vajalikud avaldiste vahetulemuste säilitamiseks (meie *Trigol*is omistatakse neile nimed *dTv*i, kus

[1] Pole vahet, on see reaalse või virtuaalse masina oma.
[2] Näiteks, korduvad märgendid, suunamine deklareerimata märgendile jmt.
[3] Algoritm on lihtne: tuleb läbida analüüsi puu ning imiteerida magasini abil kompileerimisaegset tööd, otsides töömuutuja „maksimaalset numbrit" *n*. See *n* ongi vajalike töömuutujate arv.

i=0,1,…,*n* (*n*= vajalike töömuutujate arv − 1)). Assembleri reeglid nõuavad muutujate deklareerimist andmesektsioonis, enne koodisektsiooni algust. Töömuutujad on lokaliseritavad *avaldiste* kosseisus – iga avaldise mõjupiirkonna lõppedes on "ammendatud" kõik töömuutujad. Seega, iga "vaba" töömuutuja on kasutatav järgmises operaatoris;

- luurata puus tingimusoperaatoreid ning teisendada analüüsi puud, hõlbustamaks järgnevat kompileerimist nii, et puu ühekordse kompileerimisaegse läbimisega saaks genereerida kogu objektkoodi.

Kompilaator liigub analüüsi puul sama moodi nagu Interpretaator: lehtedes olevad andmed kogutakse *LIFO*-magasini ning "üles" liikudes genereeritakse käsud, millede operand(id) saadakse magasini tipust.

8.7.3.3. Optimeerimine

Eesti Entsüklopeedia [14, lk.79] annab märksõna *optimaalne* vasteks "(mingist seisukohast) parim, sobivaim, soodsaim". Termin pärineb ladina keelest: *optimum = parim*. Selles võtmes pole mõtet rääkida *optimaalsest koodist* (transleerimise väljundist), kuivõrd pole kriteerumi, mis fikseeriks mingi algoritmiga adekvaatse *optimaalse koodi*. Samas võime me käsitleda *optimeerimist* kui parima lahendi suunas kulgevat *protsessi*, mille resultaat on loodetavasti parem kood, ent ärgem unustagem: me ei saa tõestada, et ta on *optimaalne*.

Ajalooliselt oli optimeerimine varajaste masinate puhul ülioluline, kuivõrd mälu oli vähe, protsessorid olid aeglased ja masina tööaeg oli kallis. Optimeerimiseks tehti reeglina suur arv vahekeelse koodi läbivaatusi. *V. N. Lebedev*i [32, lk. 207] andmetel tegi *Alfa*-translaator (*Algol-60* → masinkood, Novosibirsk 1964) kokku 24 läbivaatust, enamik neist just optimeerimiseks.

Optimeerivad translaatorid töötavad reeglina oluliselt aeglasemad nn. kiiretest, ent annavad tunduvalt lühema ja kiirema objektkoodi. Varasematel aegadel realiseeriti keeled tavaliselt kahes variandis: kiire translaator koos silumisvahenditega ja täitmisaegsete kontrollidega (võimaldamaks silumist), ning optimeeriv variant, mille väljundist olid eemaldatud nii silumisplokk kui ka kontrollid.

Tänapäeval on optimeerimine säilitanud oma tähtsuse reaalajas töötavate programmide ja suuremahuliste teadusarvutuste puhul (sh. paralleelprotsessingut ja/või *GRID*-tehnoloogiat kasutades).

8.7.3.3.1. Trigol

*Trigol*i optimeeriv kompilaator kasutab teadlikult ainult koodi tasemel optimeerimist ja üht kõrgema taseme võtet; põhjuseks on anda võimalus kursuse "Automaadid, keeled ja translaatorid" kuulajatele kompilaatori täiendamiseks.

Madalama taseme lihtsaim optimeerimisvõte on liigsete infovahetuskäskude vältimine: püütakse eemaldada ülearused *register*→*mälu* ja *mälu*→*register*-käsud. Näiteks, *P6.asm* sisaldab lõiku

```
M1:   mov   ax,I
      add   ax,1
      mov   dTv0,ax
      mov   ax,dTv0
      mov   I,ax
      mov   ax,I
      cmp   ax,n
      jg    M2
      mov   ax,F
      mov   dx,I
      mul   dx
      mov   dTv0,ax
      mov   ax,dTv0
      mov   F,ax
```

"Normaalsem" kood on järgmine (see on saadud *Trigl*i optimeeriva kompilaatoriga):

```
M1:   mov   ax,I
      add   ax,1
      mov   I,ax
      mov   ax,I
      cmp   ax,n
      jg    M2
      mov   ax,F
      mov   dx,I
      mul   dx
      mov   F,ax
```

"Kõrgema taseme" (e. semantikataseme) optimeerimise vahendiks on *Trigol*is **konstant-avaldiste** avastamine ja nende väärtuste arvutamine juba kompileerimise ja mitte lahendamise ajal. Konstant(alam)avaldise all mõeldakse aritmeetilist või loogilist (alam)-avaldist, kus binaarse operatsiooni mõlemad operandid on konstandid (ja mitte muutujad).

Näiteks, programmi *P4.tri* kompileerimisresultaat on esitatud allpool. Mõtlemiskoht lugejale: ons see minimaalne kood või saaks veel midagi kokku hoida?

```
; # IF 1 = 1 THEN F := 7 * ( 3 + ( 2 * 5 ) ) ; F := 100  #
;
; Program P4.asm
      .MODEL      small
      .STACK      100h
      .DATA
F     DW    0
dTv0  DW    0
      .CODE
ProgramStart:
      mov   ax,@data
      mov   ds,ax
MExi1:      mov   ax,91
      mov   F,ax
MExi2:      mov   ax,100
      mov   F,ax
      mov   ah,4ch
      int   21h
      END   ProgramStart
```

Usutavasti vajab ülaltoodud tekst kommentaari: töömuutujat *dTv0* ja töömärgendeid *MExi1* ja *MExi2* ei kasutata. Põhjus on lihtne: *Trigol*i optimeeriv kompilaator "näeb" töö ajal ainult magasini kahte (või üht) tipmist elementi; ta ei tegele "globaalse" optimeerimisega, vaid ainult lokaalsega.

8.7.3.3.2. Muud tuntumad võimalused

Selles jaotises on toetutud *David Gries*i [16, lk. 420 jj.], *Philip Lewise* jt. [35, lk. 571 jj.], *Alfred V Aho* ja *Jeffrey D Ullman*i [1, lk. 327 jj.] raamatutele ning N. Liidu selle valdkonna üldtunnustatud liidri, *Igor Pottossin*i 1979. aasta loengusarjale TÜ AK ja Eesti Teaduste Akadeemia Küberneetika Instituudi suvekoolis (Elbis). Tuntumad võimalused on järgmised:

Lineaarsed lõigud. Intuitiivselt, lineaarne lõik on operaatorite jada, mis ei sisalda märgendeid ega suunamist. Näiteks:

```
…a:=2; b:=3; c:=a+b; a:=b:=44; . . .
```

Ilmselt saame selle lõigu asendada tekstiga

```
c:=5; a:=b:=44; …
```

Ja kui *Trigol*-kompilaator töötleks lineaarseid lõike, siis *P4.tri* sisaldaks ainult üht operaatorit:

```
F:=100;
```

Aritmeetilised avaldised. Lihtsaim võte on ajaliselt "kallite" tehete asendamine lihtsamate ja kiirematega. Näiteks,

`y:=a*2;` võib asendada avaldisega `y:=a+a;` ning
`y:=(a+b)↑2;` (kus ↑ tähistab astendamist) avaldistega `temp:=a+b; y:=temp*temp;`

Kui operandid on täisarvud, siis „kahega korrutamise" asendamine nihutamistehtega on oluliselt "odavam", isegi liitmisega võrreldes: `y:=a*2;` ja kiirema `y:=a+a;` asemel on veel kiirem käsk `y<<=1;` Veel suurema ajavõidu saame asendades „kahe astmega" jagamise nihutamisega paremale ja paarisarvulisuse kontrollis arvu kahega jagamise jäägi testimise asemel viimase biti väärtuse testimisega (kui see on 0, siis on tegemist paarisarvuga).

Lewis jt.[35, lk. 574] toovad näite, kus antud avaldise transleerimiseks saab kasutada ainult registreid 1 ja 2. Avaldiseks on

```
A*B+(C+D)*(E+F)
```

*IBM*i-sarnases assembleris on selle kodeerimiseks vaja 9 käsku:

```
L       1,A
M       1,B
L       2,C
A       2,D
ST      1,TEMP
L       1,E
ADD     1,F
MR      1,2
A       1,TEMP
```

Tehete järjekorra muutmisega saame kokku hoida ühe direktiivi:

```
L       1,C
A       1,D
L       2,E
A       2,F
MR      1,2
L       2,A
M       2,B
AR      1,2
```

Seega, kompileeriti avaldis `(C+D)*(E+F)+(A*B).` Siiski, tehete järjekorra muutmine pole alati ohutu. Näiteks, `X–1+1` ei anna ületäitumist ja `X+1–1` annab, kui `X` väärtus on *int*-tüüpi muutuja maksimaalselt võimalik väärtus. *Igor Pottossin* rääkis, et nende *Alfa*-translaatori (*Algol-60* → masinkood, Novosibirsk 1964) võimas optimeerimisplokk tuli füüsikute tungival nõudmisel muuta väljalülitatavaks: kõrgeltkvalifitseeritud kasutajad mängisid keeruliste valemite programmeerimisel tihti vea piiril (kas vahetulemus annab liiga suur või väike olles ületäitumise või ei)[1] ning programmeerisid seetõttu valemeid kaugeltki mitte optimaalsel (kiiruse ja käskude arvu suhtes) kujul.

Levinud (ja ohutu) võte on avaldistest ühiste alamavaldiste eraldamine, näiteks [35, lk. 575] avaldis

```
(A+B)*C+D/(A+B)
```

tuleks kompileerida kui

```
TEMP:=A+B; TEMP*C+D/TEMP;
```

Tsüklite optimeerimine võib evida märgatavat effekti. Peamised võtted on:

- tsükli puhastamine: invariantse koodi viimine tsükli ette. Näiteks, programmilõik
  `FOR I:= 1 STEP 1 until N DO` A[I]:=C [I]+E*F;
  on mõttekas asendada koodiga
  `TEMP:=E*F; FOR I:= 1 STEP 1 until N DO` A[I]:=C [I]+TEMP;
- tsüklite liitmine, näiteks:
  `FOR I:= 1 STEP 1 UNTIL N DO` A[I]:=0;
  `FOR I:= 1 STEP 1 UNTIL N DO` B[I]:=0; asemel on parem täita operaator
  `FOR I:= 1 STEP 1 UNTIL N DO BEGIN` A[I]:=0; B[I]=0; END

[1] Tuletagem meelde tolleaegsete arvutite arhitektuurist tigitud piiranguid. Füüsikud kasutasid terminit „pessimiseerimine" „optimiseerimise" asemel − eks ole, optimism ja pessimism.

- tsüklite lahutamine − kui meil on kasutada kaks protsessorit, siis eelmise näite resultaat on otstarbekas jaotada: `A` nullimine esimese ja `B` nullimine teise protsessori tööks;
- tsüklioperaatori täitmiskordade vähendamine, näiteks tsükli
  `FOR I:= 1 STEP 1 UNTIL N DO` A[I]:=0;
  asemel on kiirem variant (kui N on paarisarv)
  `FOR I:= 1 STEP 2 UNTIL N DO BEGIN` A[I]:=0; A[I+1]:=0; END
- "kallite" tehete asendamine "odavamatega": – laename näite [35, lk. 578].
  `FOR I:= 1 STEP 1 UNTIL N DO` A[I]:=I*5;
  asemel on "odavam" kirjutada
  `J:=5; FOR I:= 1 STEP 1 UNTIL N DO BEGIN A[I]:=J; J:=J+5; END`
- indeksmuutujate asendamine lihtmuutujatega, näiteks (eriti, kui masinal pole indeksregistreid), siis programm

```
FOR I:=1 STEP 1 UNTIL N DO
      FOR J:=I+1 STEP 1 UNTIL N–1 DO BEGIN
            IF A[I,J]>A[I,J+1] THEN BEGIN
                  TEMP:=A[I,J];
                  A[I,J]:=A[I,J+1],
                  A[I,J+1]:=TEMP;
                  END
            END
```

töötab kiiremini, kui ta kodeerida nii:

```
FOR I:=1 STEP 1 UNTIL N DO
      FOR J:=I+1 STEP 1 UNTIL N–1 DO BEGIN
            TEMP:=A[I,J];
            IF TEMP>A[I,J+1] THEN BEGIN
                  A[I,J]:=A[I,J+1];
                  A[I,J+1]:=TEMP;
                  END
            END
      END
```

Kasutamata objektide elimineerimine. Sellisteks on deklareeritud, ent avaldistes kasutamata konstandid, muutujad, mida kas ei väärtustata üldse või väärtustatakse, kuid ei kasutata; märgendid, millele ei suunata ning programmilõigud, kuhu ei anta kunagi juhtimist. Mõistagi, neil juhtudel tuleb alati väljastada hoiatusteated: tegu võib olla vajalike asjadega, mis on "kasutud" programmeerija juhuslike vigade tõttu.

Keerulised seigad seonduvad (plokkstruktuuriga keelte) plokkide, eraldi transleeritavate alamprogrammide, rekursiooni ja paralleelprotsessinguga (kui loetleda neist enamtuntuid). *Eero Vainikko* [58, lk. 12] kirjutab: "Kuigi on olemas teatud reeglid, mida optimaalsel programmeerimisel arvestada, jääb suur osa programmikoodi optimeerimistööst kompilaatori kanda. Ilmne reegel on: mida keerulisem keel, seda raskem on kompilaatoril teha õigeid optimeerimisotsuseid. Näiteks keeles *Fortran77* kirjutatud programmi on tunduvalt lihtsam optimeerida kui keeles *C++*, põhjuseks *Fortran77* staatiline mäluhaldus.

Fortran95 täiendab *Fortran77* standardit moodsate keeleliste vahenditega, arvestades seejuures, et optimeeritavus säiliks niipalju kui võimalik. Objekt-orienteeritud kontseptsioonist rakendatakse vaid teatud lihtsam osa, mis ei kahjusta programmikoodi optimeeritavust arvutuskiiruse mõttes."

KIRJANDUS

[1] Aho Alfred V.,. Ullman Jeffrey D, The Theorie of Parsing, Translation and Compiling, vol. 2: Compiling, Prentice-Hall Inc, Englewood Cliffs, N. J., 1973 (А. Ахо, Дж. Ульман, Теория синтактического анализа и компиляции, том 2, Компиляция, «Мир», М 1978).

[2] АЛГОЛ 68. Методы реализации, Издательство Ленинградского университета, Ленинград, 1976.

[3] Angermeyer John, Jaeger Kevin, Bapna Kumar Raj, Barkakati Nabajyoti, Dhesikan Rajakopalan, Dixon Walter, Dumke Andrew, Fleig John, Goldman Michael, The Waite Group's MS-DOS® Developer's Guide, Second Edition, Howard W. Sams&Company, Indianapolis 1989.

[4] Bauer Friedrich L., Gnatz Rupert, Hill Ursula, Informatik. Aufgaben und Lösungen, Erster Teil, Springer-Verlag Berlin, Heidelberg, New York, 1975.

[5] Бежанова М. М., Поттосин И., В., Математическое обеспечение ЭВМ: операционные средства, Новосибирск 1986.

[6] Белокурская И. А., Кушнерев Н. Т., Неменман М. Е., Диспечер ЭВМ «Минск-32», «Статистика», М 1973

[7] Богданов В.В., Ермаков Е.А., Маклаков А.В., Программирование на языке АЛМО, Москва, изд-во <<Статистика>>, 1976.

[8] Браун П., Макропроцессоры и мобильность программного обеспечения, «Мир», Москва, 1977.

[9] CODASYL Systems Committee, Feature Analysis of Generalized Data Base Management Systems, May 1971.

[10] Coffron James W., Programming the 8086/8088, SYBEX Inc., Berkeley, 1983.

[11] Computer Dictionary, Microsoft Press®, Redmond, 1991.

[12] Донован Дж., Системеое программирование, «Мир», Москва 1975.

[13] Джермейн К., Программирование на IBM/360, «Мир», М. 1973.

[14] Eesti Entsüklopeedia 7, Eesti Entsüklopeediakirjastus, Tallinn 1994.

[15] Экхауз Р., Моррис Л., Мини-ЭВМ: организаци и программирование, М 1983.

[16] Грис Д., Конструирование компиляторов для цифровых вычислительных машин, «Мир», Москва 1975.

[17] Griswold R. E., Poage J. F., Polonsky I. P., The SNOBOL4 Programming Language, second edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1971.

[18] Грисуолд Р., Поудж Дж., Полонски И., Язык программирования СНОБОЛ-4, «МИР», М 1980.

[19] Henno Jaak, Arvuti ja keeled I, Formaalsed keeled, grammatikad ja translaatorid, Tallinna Tehnikaülikool, Informaatikainstituut, TTÜ Kirjastus, Tallinn 2006.

[20] Henno Jaak, Võhandu Leo, Huvitav informaatika I, Eesti NSV Haridusministeerium, ENSV Vabariiklik Õpetajate Täiendusinstituut, Tallinn 1987.

[21] Higman B., Programmiersprachen. Eine vergleichende Studie, Leipzig, 1971.

[22] Isotamm A., Masinkood ja assembler, Tartu Riiklik Ülikool, 1988.

[23] Isotamm A., Makrokeel, Tartu Riiklik Ülikool, 1988.

[24 Jürgenson Rein, Programmeerimine kolmanda põlvkonna arvutieile, „Valgus", Tallinn 1977.

[25] Kaasik Ü., Kull I., Programmeerimine ALGOLis, TRÜ, Tartu 1979

[26] Kaasik Ü., Ääremaa K., Jaeger A., Kalberg M., Ehasalu J., Ermann S., Elektronarvuti „MINSK-32", Õppevahend, Tartu Riiklik Ülikool, Arvutuskeskus, Tartu 1970.

[27] Камынин С.С., Любимский Э.З. Алгоритмический машинно-ориентированный язык – АЛМО. В сб.: <<Алгоритмы и алгоритмические языки>>. ВЦ АН СССР, 1967, вып. 1.

[28] Kelder T., Kaasik Ü., Programmeerimiskeel C, Programme kõigile, Tartu Ülikool, Arvutuskeskus, Tartu 1989.

[29] Kernighan Brian W, Ritchie Dennis M., The C Programming Language, Second Edition, AT&T Bell Laboratories, Murray Hill, New Jersey, *sine data.*

[30] Knaggs Peter, Forth: An underview, http://dec.bournemouth.ac.uk/forth/forth.html (18.03.06)

[31] Lafore Robert, The Waite Group's Turbo C® Programming for the PC, Revised Edition, Howard W.Sams & Company, 1989.

[32] Лебедев В. Н., Введение в системы программирования, «Статистика», М. 1975.

[33] Lebherz Eric, Hypernews Computer Language List v. 1.4, http://www.hypernews.org/HyperNews/get/computing/lang-list.html (7. 09.2005)

[34] Lévenéz Éric, Languages june 26, 2006, © Éric Lévenéz 1999 – 2006, http://www.levenez.com/ lang/ (21.07.06)

[35] Lewis Philip M., Rosenkrantz Daniel J., Stearns Richard E., Compiler Design Theory, Addison-Wesley Publishing Company, 1976 (Ф. Люис, Д. Розенканц, Р. Стринз, Теоретические основы проектирования компиляторов, «МИР», М 1979.)

[36] McCarthy John, History of Lisp, 1979 http://www-formal.stanford.edu/jmc/history/lisp/ (18.12.06)

[37] Microsoft. MacroAssembler 5.1, 1987, published in the U.S. and Canada.

[38] Майерс Г., Надежность программного обеспечения, «Мир», Москва 1980.

[39] Marmelstein Robert E., Programming games in C, M&T Books, New York, 1994.

[40] Неслуховский К. С., Пособие по программированию для ЭЦВМ «Минск-32», «Советское Радио», М.. 1975.

[41] Operatsioonisüsteem DOS ja ASSEMBLER-keele kasutamine andmetöötluse automatiseerimisel, Tallinn 1977.

[42] Polak Sergey, In Search of the Ideal Programming Language http://members.aol.com/SergeyP/paper.html (18.03.06.)

[43] Пособие по программированию на ЭВМ «Раздан-3», методические указания, Каунас 1969.

[44] Пратт Т, Языки программирования: разработка и реализация, «Мир», М. 1979.

[45] Programme kõigile VIII, koostanud Ü. Kaasik ja H. Niilisk, Programmeerimine FORTRANIS, Tartu Riiklik Ülikool, Arvutuskeskus, Tartu 1974.

[46] Процессор ЭВМ ЕС-1020, «Статистика», М . 1975.

[47] Сборник научных программ на ФОРТРАНЕ, Вып. 1 и 2, Москва, «Статистика», 1974.

[48] Семенов Ю. А., Программирование на языке ФОРТ, М., «Радио и связь», 1991.

[49] Система математического обеспечения ЕС ЭВМ, «Статистика», М . 1974.

[50] Скотт Р., Сондак Н., ПЛ/1 для программистов, «Статистика», Москва . 1977.

[51] Стэбли Д, Логическое программирование в системе /360, «Мир», М. 1974.

[52] Шилд Герберт, Программирование на BORLAND C++ для профессионалов, 1997.

[53] Tamme Tõnu, Tammet Tanel, Prank Rein, Loogika. Mõtlemisest tõestamiseni, Tartu Ülikooli Kirjastus, Tartu 2002.

[54] Толковый словарь по вычислительной технике, Microsoft Press®, Русская Редакция, 1995.

[55] Tombak M., Soo V., Pöial J., A Forth-Oriented Compiler Compiler and its applications. *Forth Dimensions*, Vol.16 No.5 Oackland, USA 1995, pp. 21-22.

[56] Turbo Assembler®. User's Guide, Version 1.0, Printed in the U.S.A., 1989.

[57] Uffenbeck John, Microcomputers and microprocessors: the 8080, 8085 and Z-80: prog-ramming, interfacing and troubleshooting, 2nd ed., Prentice-Hall, New Jersey, 1991.

[58]Vainikko Eero, Fortran95 ja MPI, Tartu Ülikool, Arvutiteaduse instituut, Tartu 2004.

[59] Venners Bill, Bytecode Basics, First Published in JavaWorld, September 1996 (http: // www.artima.com/underthehood/bytecode.html 10.10.06).

[60] van Wijngaarden A., Mailloux B. J., Peck J. E. L., Koster C H. A., Sintzoff M., Lindsey C. H., Meertens L. G. L. T., Fisker R. G., Revised Report on the Algorithmic Language ALGOL 68, Springer Verlag, Berlin, Heidelberg, New York, 1975/ А. ван Вейнгаарлен и др., Пересмотренное сообщение об АЛГОЛЕ 68, «МИР», Москва, 1979.

[61] Wilson L.B., Clark L.G., Comporative Programming Languages, Addison-Wesley Publishers Limited, 1988.

[62] Wirth Niklaus, Algorithms & Data Structures. Prentice-Hall, 1986.

[63] Wirth N., Kolmkümmend aastat programmeerimiskeeli ja translaatoreid, tõlkinud Jaan *Penjam*, A&A, nr.1, 1993, lk. 13-24.

[64] http://www.uv.tietgen.dk/staff/mlha/PC/Prog/asm/int/21/index.htm#4C (5.02.06)

[65] http://en.wikipedia.org/wiki/BIOS_Interrupt_Calls (6.02.06)

[66] http://www.cs.utexas.edu/users/EWD/ (12.02.06)

[67] http://en.wikipedia.org/wiki/ALGOL_58 (4.03.06)

[68] [borg] http://caesum.com/download.php (16.03.06)

[69] http://en.wikipedia.org/wiki/Forth_programming_language (18.03. 06)

[70] http://www.forth.org.ru/ (18. 03. 06)

[71] http://www.forth.ru/ (18.03.06)

[72] http://www.phact.org/e/forth.htm (18.03.06)

[73] http://www.colorforth.com/bio.html . (18.03.06)

[74] http://www.bashedu.ru/konkurs/tarhov/images/pc_031.jpg (26.03.06)

[75] http://www.bashedu.ru/konkurs/tarhov/images/rc_005.jpg . (26.03.06)

[76] http://en.wikipedia.org/wiki/FORTH#Structure_of_the_language (24.04.06)

[77] http://en.wikipedia.org/wiki/Ken_Thompson (5.05.06)

[78] http://en.wikipedia.org/wiki/C_programming_language (5.05.06)

[79] http://www.psych.usyd.edu.au/pdp-11/core.html (7.05.06)

[80] http://en.wikipedia.org/wiki/PDP-11#PDP-11_instructions (11.03.06)

[81] http://www.computermuseum.li/Testpage/Burroughs-5500-DP-System.htm (26.03.06)

[82] http://www.bashedu.ru/konkurs/tarhov/english/minsk-32.htm (26.03.06)

[83] http://www.bashedu.ru/konkurs/tarhov/russian/razdan-3.htm (26.03.06)

[84] http://www.village.org/pdp11/faq.pages/WhatPDP.html (27.03.06)

[85] http://en.wikipedia.org/wiki/C_language#Early_developments (3.05.06)

[86] http://en.wikipedia.org/wiki/PDP-7 (5.05.06)

[87] http://en.wikipedia.org/wiki/RPG_programming_language (23.05.06)

[88] http://www.sp.cmc.msu.ru/staff/sva.html (23.05.06)

[89] http://wiki.zzz.ee/index.php/Deduktsioon (23.05.06)

[90] http://www.crews.org/curriculum/ex/compsci/articles/generations.htm (26.05.06)

[91] http://www.webopedia.com/DidYouKnow/Hardware_Software/2002/ FiveGenerations.asp (26.05.06)

[92] http://washington.uwc.edu/about/faculty/johnson_m/mech4gen/allgens.html (26.05.06)

[93] http://www.cs.umass.edu/~weems/CmpSci635A/Lecture1/L1.12.html (26.05.06)

[94] http://www.webopedia.com/TERM/A/artificial_intelligence.html (26.05.06)

[95] http://en.wikipedia.org/wiki/UNCOL (29.05.06)

[96] http://homepage.ntlworld.com/michael.harley/uncol.html (29.05.06)

[97] http://www.everything2.com/index.pl?node_id=763383 (29.05.06)

[98] http://hopl.murdoch.edu.au/famtree.prx?exp=143 (29.05.06)

[99] http://64.233.161.104/search?q=cache:1KCCfmrB4J:www.cs.aau.dk/~bt/SPOF06/SPOF06-9-3.ppt+UNCOL&hl=et&ct=clnk&cd=88 (29.05.06)

[100] http://64.233.161.104/search?q=cache:HX_baiG5jNAJ:www.cs.drexel.edu/static/reports/DU-CS-06-02.pdf+UNCOL&hl=et&ct=clnk&cd=94 (29.05.06)

[101] http://en.wikipedia.org/wiki/Noam_Chomsky (27.06.06)

[102] http://www.csis.ul.ie/cobol/Course/COBOLIntro.htm (06.07.06)

[103] http://pascal.comsci.us/syntax/statement/assignment.html (06.07.06)

[104] http://turnbull.dcs.st-and.ac.uk/history/Biographies/Lukasiewicz.html (12.07.06)

[105] http://en.wikipedia.org/wiki/Jan_Lukasiewicz (12.07.06)

[106] http://www.rbtl.ru/konkurs/tarhov/russian/es-1060.htm (21.07.06)

[107] http://www.physics.emory.edu/~weeks/software/mandel.c (08.05.06).

[108] http://et.wikipedia.org/wiki/Paradigma (12.05.06)

[109] http://en.wikipedia.org/wiki/Programming_paradigm (12.05.06)

[110] http://en.wikipedia.org/wiki/Programming_paradigm (15.02.2004)

[111] http://www.agnesscott.edu/lriddle/women/love.htm (5.05.06)

[112] http://schools.keldysh.ru/project-1689/history.htm (9.10.06)

[113] http://en.wikipedia.org/wiki/SNOBOL (10.10.06)

[114] http://en.wikipedia.org/wiki/Compiler_compiler (21.08.06)

[115] http://en.wikipedia.org/wiki/List_of_compiler-compilers (21.08.06)

[116] http://en.wikipedia.org/wiki/Java_bytecode (10.10.06)

[117] http://java.sun.com/docs/books/vmspec/2nd-edition/html/Overview.doc.html (16.10.06)

[118] http://www.javaworld.com/javaworld/jw-09-96/bcsupport/source/ ConversionDiversion.java (16.10.06)

[119] http://www.cat.nyu.edu/~meyer/jvmref/ref-Java.html (18.10.06)

[120] http://java.sun.com/docs/books/vmspec/2nd-edition/html/Mnemonics.doc.html (17.10.06)

[121] http://en.wikipedia.org/wiki/Common_Intermediate_Language (24.10.06)

[122] http://en.csharp-online.net/HelloWorld_in_MSIL (24.10.06)

[123] http://www2.latech.edu/~acm/helloworld/csharp.htm (25.10.06)

[124] http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx (24.10.06)

[125] http://dotnet.di.unipi.it/EcmaSpec/PartitionIII/cont4.html#_Toc526908965 (26.10.06

[126] http://mono-project.com/Main_Page (10.110.06)

[127] http://mono-project.com/FAQ:General (10.11.06)

[128] http://mono-project.com/Supported_Platforms (10.11.06)

[129] http://www.chomsky.info/ (pilt. 27.06.06)

[130] http://en.wikipedia.org/wiki/ Lisp_programming_language (18.12.06)

[131] http://en.wikipedia.org/wiki/Lambda_calculus (18.12.06)

[132] http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq-doc-7.html (18.12.06)

[133] [http://www.webopedia.com/TERM/B/bytecode.html (10.10.06)

[134] http://en.wikipedia.org/wiki/Java_virtual_machine (13.10.06)

[135] http://en.wikipedia.org/wiki/.NET_Framework#Common_Language__Infrastructure_.28CLI.29 (10.10.06))

[136] http://en.wikipedia.org/wiki/Java_virtual_machine (13.10.06)

[137] http://reviews.zdnet.co.uk/hardware/components/0,1000001694,39189912,00.htm (28.06.06)

[138] http://en.wikipedia.org/wiki/IA-32 (31.01.07)

LISAD

1. Tekstifail Teek.asm

Teek.asm

```
; The Waite Group's MS-DOS Developer's Guide, Second Edition,
; John Angermeyer jt,Howard W. Sams & Company, 1989, pp 724-725 [3]
;
; bin2dec
; INPUT: AX - number to be displayed
;      CH - minimum number of digits to be displayed
;      DX = 0, if number is unsigned, 1, if signed
; OUTPUT None
;
;
      .MODEL      small
      .STACK      100h
      .DATA
      .CODE
      PUBLIC bin2dec
      PUBLIC readint
;
bin2dec     PROC  NEAR
      push  ax          alamprogramm peab säilitama ülemise taseme
      push  bx          programmi registrite seisud — magasinis.
      push  cx          Nende taastamist vt. alamprogrammi koodi
      push  dx          lõpus.
      mov   cl,0
      mov   bx,10
      cmp   dx,0
      je    more_dec
      or    ax,ax
      jnl   more_dec
      neg   ax
;     @DisChr     '-'
      push  ax
      push  dx
      mov   dl,'-'
      mov   ah,02h
      int   21h
      pop   dx
      pop   ax
more_dec:
      xor   dx,dx
      div   bx
      push  dx
      inc   cl
      or    ax,ax
      jnz   more_dec
; Main Digit Print Loop - Reverse Order
      sub   ch,cl
      jle   morechr
      xor   dx,dx
```

```
morezero:
      push  dx
      inc   cl
      dec   ch
      jnz   morezero
morechr:
      pop   dx
      add   dl,30h
; DisChr
      push  ax
      push  dx
      mov   ah,02h
      int   21h
      pop   dx
      pop   ax
      dec   cl
      jnz   morechr
      pop   dx
      pop   cx
      pop   bx
      pop   ax
      ret
bin2dec     ENDP
;
; vt. ka
; Turbo Assembler. Users Guide, Version 1.0, Borland International, 1989
; p 55 [56]; Angermayer et al, [3],p.560.
;
; readint
; INPUT: none
; OUTPUT AX - sisestatud ja teisendatud arv
;
;
readint     PROC  NEAR
      push  bx
      push  cx    ;save
      push  dx    ;registers; res=ax
      xor   ax,ax   ;arv
ring: push  ax    ;save
      mov   ah,1  ;DOS keyboard input
      int   21h   ;get the next symbol
      mov   cl,al ;new digit in cl
      sub   cl,30h      ;symbol->10nd-nr
      cmp   al,13 ;Enter?
      jz    oki   ;valmis
      pop   ax    ;senine arv
      mov   dx,10
      mul   dx    ;senine arv * 10
      xor   ch,ch ;prepare for 16-bit add
      add   ax,cx ;new digit is added in
      jmp   ring
oki:  mov   dl,10 ;linefeed
      mov   ah,2  ;DOS display output
      int   21h
      pop   ax
      pop   dx
      pop   cx
```

```
      pop   bx
      ret
readint ENDP
      END
```

2. Fail P6.exe

P6.exe

```
; Borg Disassembler v2.27 : C:\TTS\p6.exe
;
;             Created by Borg Disassembler
;                   written by Cronos

1000:0000
;------------------------------------------------------------------------
1000:0000 ;Segment : 1000h     Offset : 00h     Size : 100h
1000:0000 ;16-bit Code
1000:0000
;------------------------------------------------------------------------
1000:0000 start:
1000:0000 b81010              mov    ax, 1010h  P6: 1010h=@data
1000:0003 8ed8                mov    ds, ax
1000:0005 b409                mov    ah, 09h
1000:0007 bb0100              mov    bx, 01h
1000:000a b91100              mov    cx, 11h
1000:000d ba0c00              mov    dx, 0ch  mov dx,OFFSET Sisse
1000:0010 cd21                int    21h
1000:0012 b409                mov    ah, 09h
1000:0014 bb0100              mov    bx, 01h
1000:0017 b90200              mov    cx, 02h
1000:001a ba2a00              mov    dx, 2ah  mov dx,OFFSET n_S
1000:001d cd21                int    21h
1000:001f e8b700              call   loc_000000d9  call readint
1000:0022 a30400              mov    word ptr[04h], ax   mov n,ax
1000:0025 b80100              mov    ax, 01h
1000:0028 a30600              mov    word ptr[06h], ax   mov F,ax
1000:002b b80000              mov    ax, 00h
1000:002e a30800              mov    word ptr[08h], ax   mov I,ax
1000:0031 a10800              mov    ax, word ptr[08h]   M1: mov ax,I
1000:0034 050100              add    ax, 01h
1000:0037 a30a00              mov    word ptr[0ah], ax   mov dTv0,ax
1000:003a a10a00              mov    ax, word ptr[0ah]   mov ax,dTv0
1000:003d a30800              mov    word ptr[08h], ax   mov I,ax
1000:0040 a10800              mov    ax, word ptr[08h]   mov ax,I
1000:0043 3b060400            cmp    ax, [04h]           cmp ax,n
1000:0047 7f14                jg     loc_0000005d        jg M2
1000:0049 a10600              mov    ax, word ptr[06h]   mov ax,F
1000:004c 8b160800            mov    dx, [08h]           mov dx,I
1000:0050 f7e2                mul    dx
1000:0052 a30a00              mov    word ptr[0ah], ax   mov dTv0,ax
1000:0055 a10a00              mov    ax, word ptr[0ah]   mov ax,dTv0
1000:0058 a30600              mov    word ptr[06h], ax   mov F,ax
1000:005b ebd4                jmp    31h                 jmp M1
1000:005d ;                                   XREFS First: 1000:0047
Number : 1
```

```
1000:005d loc_0000005d:                          M2:
1000:005d b409                  mov     ah, 09h
1000:005f bb0100                mov     bx, 01h
1000:0062 b90800                mov     cx, 08h
1000:0065 ba2000                mov     dx, 20h    mov dx,OFFSET Trykk
1000:0068 cd21                  int     21h
1000:006a b409                  mov     ah, 09h
1000:006c bb0100                mov     bx, 01h
1000:006f b90200                mov     cx, 02h
1000:0072 ba2d00                mov     dx, 2dh    mov dx,OFFSET F_S
1000:0075 cd21                  int     21h
1000:0077 a10600                mov     ax, word ptr[06h]  mov ax,F
1000:007a ba0000                mov     dx, 00h
1000:007d 3d0000                cmp     ax, 00h
1000:0080 7f03                  jg      loc_00000085    jg s1h2o3w
1000:0082 ba0100                mov     dx, 01h
1000:0085 ;                                        XREFS First: 1000:0080
Number : 1                                    s1h2o3w:
1000:0085 loc_00000085:
1000:0085 b501                  mov     ch, 01h
1000:0087 e80400                call    loc_0000008e   call bin2dec
1000:008a b44c                  mov     ah, 4ch
1000:008c cd21                  int     21h
1000:008e ;                                        XREFS First: 1000:0087
Number : 1
1000:008e loc_0000008e:                       bin2dec  kood teek.asm-ist
1000:008e 50                    push    ax
1000:008f 53                    push    bx
1000:0090 51                    push    cx
1000:0091 52                    push    dx
1000:0092 b100                  mov     cl, 00h
1000:0094 bb0a00                mov     bx, 0ah
1000:0097 83fa00                cmp     dx, 00h
1000:009a 7410                  jz      loc_000000ac   je more_dec
1000:009c 0bc0                  or      ax, ax
1000:009e 7d0c                  jge     loc_000000ac   jnl more_dec
1000:00a0 f7d8                  neg     ax
1000:00a2 50                    push    ax
1000:00a3 52                    push    dx
1000:00a4 b22d                  mov     dl, 2dh    mov dl,'-'
1000:00a6 b402                  mov     ah, 02h
1000:00a8 cd21                  int     21h
1000:00aa 5a                    pop     dx
1000:00ab 58                    pop     ax
1000:00ac ;                                        XREFS First: 1000:009a
Number : 2
1000:00ac loc_000000ac:                       more_dec:
1000:00ac 33d2                  xor     dx, dx
1000:00ae f7f3                  div     bx
1000:00b0 52                    push    dx
1000:00b1 fec1                  inc     cl
1000:00b3 0bc0                  or      ax, ax
1000:00b5 75f5                  jnz     loc_000000ac   jnz more_dec
1000:00b7 2ae9                  sub     ch, cl
1000:00b9 7e09                  jle     loc_000000c4   jle morechr
1000:00bb 33d2                  xor     dx, dx
1000:00bd 52                    push    dx          morezero: push dx
```

```
1000:00be fec1                  inc     cl
1000:00c0 fecd                  dec     ch
1000:00c2 75f9                  jnz     0bdh        jnz morezero
1000:00c4 ;                                         XREFS First: 1000:00b9
Number : 1
1000:00c4 loc_000000c4:                       morechr:
1000:00c4 5a                    pop     dx
1000:00c5 80c230                add     dl, 30h
1000:00c8 50                    push    ax
1000:00c9 52                    push    dx
1000:00ca b402                  mov     ah, 02h
1000:00cc cd21                  int     21h
1000:00ce 5a                    pop     dx
1000:00cf 58                    pop     ax
1000:00d0 fec9                  dec     cl
1000:00d2 75f0                  jnz     loc_000000c4  jnz morechr
1000:00d4 5a                    pop     dx
1000:00d5 59                    pop     cx
1000:00d6 5b                    pop     bx
1000:00d7 58                    pop     ax
1000:00d8 c3                    ret
1000:00d9 ;                                         XREFS First: 1000:001f
Number : 1
1000:00d9 loc_000000d9:                       readint  kood teek.asm-ist
1000:00d9 53                    push    bx
1000:00da 51                    push    cx
1000:00db 52                    push    dx
1000:00dc 33c0                  xor     ax, ax
1000:00de 50                    push    ax            ring: push ax
1000:00df b401                  mov     ah, 01h
1000:00e1 cd21                  int     21h
1000:00e3 8ac8                  mov     cl, al
1000:00e5 80e930                sub     cl, 30h
1000:00e8 3c0d                  cmp     al, 0dh
1000:00ea 740c                  jz      loc_000000f8   jz oki
1000:00ec 58                    pop     ax
1000:00ed ba0a00                mov     dx, 0ah
1000:00f0 f7e2                  mul     dx
1000:00f2 32ed                  xor     ch, ch
1000:00f4 03c1                  add     ax, cx
1000:00f6 ebe6                  jmp     0deh           jmp ring
1000:00f8 ;                                         XREFS First: 1000:00ea
Number : 1
1000:00f8 loc_000000f8:                             oki:
1000:00f8 b20a                  mov     dl, 0ah
1000:00fa b402                  mov     ah, 02h
1000:00fc cd21                  int     21h
1000:00fe 58                    pop     ax
1000:00ff 5a                    pop     dx
1010:0000
;---------------------------------------------------------------------------
1010:0000 ;Segment : 1010h     Offset : 00h     Size : 30h
1010:0000 ;16-bit Code
1010:0000
;---------------------------------------------------------------------------
1010:0000 59                    db      59h        ;'Y' tegelikult: pop cx
1010:0001 5b                    db      5bh        ;'['             pop bx
```

```
1010:0002 c3                db      0c3h
1010:0003 00                db      00h
1010:0004 00                db      00h         n
1010:0005 00                db      00h
1010:0006 00                db      00h         F
1010:0007 00                db      00h
1010:0008 00                db      00h         I
1010:0009 00                db      00h
1010:000a 00                db      00h         dTv0
1010:000b 00                db      00h
1010:000c 49                db      49h          ;'I'   Sisse
1010:000d 6e                db      6eh          ;'n'
1010:000e 70                db      70h          ;'p'
1010:000f 75                db      75h          ;'u'
1010:0010 74                db      74h          ;'t'
1010:0011 20                db      20h          ;' '
1010:0012 74                db      74h          ;'t'
1010:0013 68                db      68h          ;'h'
1010:0014 65                db      65h          ;'e'
1010:0015 20                db      20h          ;' '
1010:0016 76                db      76h          ;'v'
1010:0017 61                db      61h          ;'a'
1010:0018 72                db      72h          ;'r'
1010:0019 69                db      69h          ;'i'
1010:001a 61                db      61h          ;'a'
1010:001b 62                db      62h          ;'b'
1010:001c 6c                db      6ch          ;'l'
1010:001d 65                db      65h          ;'e'
1010:001e 20                db      20h          ;' '
1010:001f 24                db      24h          ;'$'
1010:0020 56                db      56h          ;'V'   Trükk
1010:0021 61                db      61h          ;'a'
1010:0022 72                db      72h          ;'r'
1010:0023 69                db      69h          ;'i'
1010:0024 61                db      61h          ;'a'
1010:0025 62                db      62h          ;'b'
1010:0026 6c                db      6ch          ;'l'
1010:0027 65                db      65h          ;'e'
1010:0028 20                db      20h          ;' '
1010:0029 24                db      24h          ;'$'
1010:002a 6e                db      6eh          ;'n'    n_S
1010:002b 3d                db      3dh          ;'='
1010:002c 24                db      24h          ;'$'
1010:002d 46                db      46h          ;'F'    F_S
1010:002e 3d                db      3dh          ;'='
1010:002f 24                db      24h          ;'$'
```

3. Djgpp päisfail BIOS.H

```
int bioscom(int cmd, char data, int port);
int biosdisk(int cmd, int drive, int head, int track, int   sector,  int
nsects, void *buffer);
int biosequip(void);
int bioskey(int cmd);
int biosmemory(void);
int biosprint(int cmd, int byte, int port);
long biostime(int cmd, long newtime);

djgpp REGISTER.H

typedef struct {
        unsigned ax, bx, cx, dx, si, di, bp, f;
        } REGISTERS;
```

4. Djgpp päisfail DOS.H

```
#ifndef _DOS_H_
#define _DOS_H_

#include <pc.h>

union REGS {
  struct {
    unsigned long ax;
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long cflag;
    unsigned long flags;
  } x;
  struct {
    unsigned char al;
    unsigned char ah;
    unsigned short upper_ax;
    unsigned char bl;
    unsigned char bh;
    unsigned short upper_bx;
    unsigned char cl;
    unsigned char ch;
    unsigned short upper_cx;
    unsigned char dl;
    unsigned char dh;
    unsigned short upper_dx;
  } h;
};

struct SREGS {
  unsigned short cs;
  unsigned short ds;
  unsigned short es;
  unsigned short fs;
  unsigned short gs;
  unsigned short ss;
};

struct ftime {
  unsigned ft_tsec:5;   /* 0-29, double to get real seconds */
```

```
  unsigned ft_min:6;    /* 0-59 */
  unsigned ft_hour:5;   /* 0-23 */
  unsigned ft_day:5;    /* 1-31 */
  unsigned ft_month:4;  /* 1-12 */
  unsigned ft_year:7;   /* since 1980 */
};

struct date {
  short da_year;
  char  da_day;
  char  da_mon;
};

struct time {
  unsigned char ti_min;
  unsigned char ti_hour;
  unsigned char ti_hund;
  unsigned char ti_sec;
};

struct dfree {
  unsigned df_avail;
  unsigned df_total;
  unsigned df_bsec;
  unsigned df_sclus;
};

#ifdef __cplusplus
extern “C” {
#endif

int bdos(int func, unsigned dx, unsigned al);
int bdosptr(int func, void *dx, unsigned al);
int int86(int ivec, union REGS *in, union REGS *out);
int int86x(int ivec, union REGS *in, union REGS *out, struct SREGS
*seg);
int intdos(union REGS *in, union REGS *out);
int intdosx(union REGS *in, union REGS *out, struct SREGS *seg);

int enable(void);
int disable(void);

int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);

int getcbrk(void);
int setcbrk(int new_value);

void getdate(struct date *);
void gettime(struct time *);
void setdate(struct date *);
void settime(struct time *);

void getdfree(unsigned char drive, struct dfree *ptr);

void delay(unsigned msec);
int _get_default_drive(void);
void _fixpath(const char *, char *);

#ifdef __cplusplus
}
#endif

#endif
```

5. Ühisvälja ja alamprogrammide kirjelduste fragment

```
/* header for robot. February 17, 1996. Waltham, Massachusetts.  */


#define F0 '\xf0'
#define NF '\x0f'


 extern int fid;
 extern char Mstack[70][512];
 extern int IJstack[70][2];
 extern char Vstack[70][16];
 extern int mask[4];
 extern int savemask[4];
 extern int piir[4];
 extern char M[514]; /* each byte is a row: 0..3 flags, 4..7 values  */
 extern char flags[4];
 extern char values[4];
 extern char cleaner[4];
 extern char F[514];      /*  "function"  */
 extern int f1[257];
 extern int f1_index;  /* the indexes where Fi=minor */
 extern char fname[40];
 extern char minor;       /* '0' or '1' */
 extern int ols[4][256];   /* overlapping scheme */
 . . . . .
extern  char TEE[40];     /* drive & path */
extern  char CmdLine[60]; /* dirarray tegemiseks */
extern  char **viidad;    /* array of pointers to directory lines */
extern  char *Ptr;        /* first pointer to directory member */
extern char **R_viidad;
extern char *R_Ptr;
extern int rarv;

int set_w(void);
void kets(void);
void fillF(void);
int scout(void);
. . . .
void gen_formula(void);
```

6. Ühisväli (fragment)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <process.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <dos.h>
#include <conio.h>
#include <alloc.h>
#include “jupp.h”
#include “pcwproto.h”
#include <menu.h>

 int fid=0xffff;
 char Mstack[70][512];
 int IJstack[70][2];
 char Vstack[70][16];
 int mask[4]={0x1b0,0x0d8,0x036,0x01b};/*it gives the overlapping s. */
 int savemask[4];
 int piir[4];
 char M[514];      /* each byte is a row: 0..3 flags, 4..7 values  */
 char flags[4] = { 0x80,0x40,0x20,0x10 };
 char values[4] = { 0x08,0x04,0x02,0x01 };
 char cleaner[4] = { 0xf7,0xfb,0xfd,0xfe };
 char F[514];      /*  “function”  */
 int f1[257];
 int f1_index;  /* the indexes where Fi=minor */
 char fname[40];
 char minor;       /* ‘0’ or ‘1’ */
 int ols[4][256];   /* overlapping scheme */
. . .
 char TEE[40];     /* drive & path */
 char CmdLine[60]; /* dirarray tegemiseks */
 char **viidad;   /* array of pointers to directory lines */
 char *Ptr;        /* first pointer to directory member */
```

7. *Minsk-32* assembleri transleerimisprotokoll

```
                                  010 010          ЗАГЛ  ПЕЧАТЬ ЧИСЛА
**        .70 00 0 0000 0 1000
        ОБЛАСТЬ  1                010 020          БАЗ     0
**        .70 00 0 0030 0 1000
0 0000-0 0002                     010 030 ВХ       РЗВ     3
                                  010 040          РИП     16
0 0003  -17 00 0 0404 0 0026      010 041          СУ      4;+16
0 0004  -20 01 1 0400 0 0027      010 042          ПАИ     :1;+0
0 0005  -10 00 0 0030 1 0000      010 043          П       +8;СЧР
0 0006  -10 00 0 0031 1 0001      010 080          П       `       `;ППЕЧ
0 0007  -17 00 0 0000 0 0032      010 090          ГРУП    +3
0 0010  -10 00 1 0001 1 0002      010 091          П       ППЕЧ;ППЕЧ+1
0 0011    71 00 0 0033 2 0000     010 110 ЦИКЛ     ЛУ      +740000000000В;ЧИСЛ
0 0012    63 00 0 0034 0 0000     010 120          ЛСДР    +102В
0 0013   -02 01 0 0003 1 0002     010 130          ЗС      :1;3;ППЕЧ+1
0 0014    60 00 0 0035  2 0000    010140           ЛСДЗ    +4;ЧИСЛ
0 0015   -20 01 0 0000 0 0025     010 150          МАС     :1:КИ1
0 0016   -22 00 0 0011 1 0000     010 160          ИС      ЦИКЛ;СЧР
0 0017   -60 00 0 1000 1 2401     010 170          ЗАКР    ПЧ
0 0020   -67 00 0 1140 1 2401     010 171          ЫЖ      СТ;1;ПЧ
**         00 00 0 0020 0 2000
0 0021    40 00 0 0050 0 0001     010 190          КОСЛ5 0;ППЕЧ;4
0 0022  -60 00 0 0400 1 2401      010 200          ОСВ     ПЧ
0 0023  -17 00 0 1004 0 0026      020 010          ВУ      4;+16
0 0024  -21 00 0 0000 0 0000      020 020          ВЫХ     ВХ;0
0 0025   00 00 0 0010 0 0000      020 030 КИ1      КИ      1;0
0 0026   00 00 0 0000 0 0020      *                КЧ      +16
0 0027   00 00 0 0000 0 0000      *                КЧ      +0
0 0030   00 00 0 0000 0 0010      *                КЧ      +8
0 0031   17 01 3 2170 3 3074      *                КТ      `       `
0 0032   00 00 0 0000 0 0003      *                КЧ      +3
0 0033   74 00 0 0000 0 0000      *                КЧ      +740000000000В
0 0034   00 00 0 0000 0 0102      *                КЧ      +102В
0 0025   00 00 0 0000 0 0004      *                КЧ      +4
**         70 00 0 0000 0 2000
        ОБЛАСТЬ  2                020 050          БАЗ     1;РАБ
**        .70 00 0 0010 0 2000
1 0000-1 0000                     020 060 СЧР      РЗВ     1
**         70 00 0 0060 0 2000
1 0001-1 0005                     020 070 ППЕЧ     РЗВ     5
**         70 00 0 0000 0 3000
        ОБЛАСТЬ  3                020 090          БАЗ     2;ОБЩ
2 0000   00 00 1 0432 0 2547      020 100 ЧИСЛ     КЧ      1234567Д
```

8. Trigoli süntaks (tekstifail tri.grm)

```
`programm'  ->    `programm12'#
`programm12'      ->    #`operaatorid'
`operaatorid'     ->    `operaator'
                  ->`operaatorid13';`operaatorid'
`operaatorid13'   ->    `operaator'
`operaator' ->    `label':`operaator'
            ->    `omistamine'
            ->    `iflause'
            ->    `suunamine'
            ->    `lugemine'
            ->    `kirjutamine'
`label'           ->    #i#
`omistamine'      ->    `muutuja':=`omistamine1'
                  ->`muutuja':=`loogilav'
`omistamine1'     ->    `aritmav'
`muutuja'   ->    #i#
`iflause'   ->    `tingimus'`operaator'
`suunamine' ->    GOTO`label'
`aritmav'   ->    `yksliige'
            ->    `aritmav'+`aritmav2'
            ->    `aritmav'-`aritmav3'
`aritmav2'  ->    `yksliige'
`aritmav3'  ->    `yksliige'
`yksliige'  ->    `tegur'
            ->    `yksliige'*`yksliige4'
            ->    `yksliige'/`tegur'
`yksliige4' ->    `tegur'
`tegur'           ->    #i#
            ->    #c#
            ->    (`tegur5'
`tegur5'    ->    `aritmav')
`loogilav'  ->    `aritmav'<`loogilav6'
            ->    `aritmav'>`loogilav7'
            ->    `aritmav'<=`loogilav8'
            ->    `aritmav'>=`loogilav9'
            ->    `aritmav'/=`loogilav10'
            ->    `aritmav'=`loogilav11'
`loogilav6' ->    `aritmav'
`loogilav7' ->    `aritmav'
`loogilav8' ->    `aritmav'
`loogilav9' ->    `aritmav'
`loogilav10'      ->    `aritmav'
`loogilav11'      ->    `aritmav'
`tingimus'  ->    IF`loogilav'THEN
`lugemine'  ->    READ #i#
`kirjutamine'     ->    WRITE #i#
```

9. Trigoli semantikafail (tekstifail tri.sem)

```
4=1    $ #i#
11=2   $ #c#
p13=10 $ omistamine->muutuja:=omistamine1
p26=11 $ yksliige->yksliige/tegur
p25=12 $ yksliige->yksliige*yksliige4
p21=13 $ aritmav->aritmav-aritmav3
p20=14 $ aritmav->aritmav+aritmav2
p12=15 $ label->#i#
p18=16 $ suunamine->GOTO label
```

```
p32=3  $ `aritmav'<`loogilav6'
p33=4  $ `aritmav'>`loogilav7'
p34=5  $ `aritmav'<=`loogilav8'
p35=6  $ `aritmav'>=`loogilav9'
p36=7  $ `aritmav'/=`loogilav10'
p37=8  $ `aritmav'=`loogilav11'
p44=18 $ tingimus->IF loogilav THEN
p45=20 $ lugemine->READ #i#
p46=21 $ kirjutamine->WRITE #i#
```

10. Baitkood

Alljärgnev tabel on kompileeritud mitmest allikast ([59], [116], [120], [118] ja [119] – viimane on, muide, parim – konkreetne, täpne ja näidetega).

Tabelis kasutame alljärgnevaid tähistusi, lisaks vajavad mõned käsud või nende grupid kommentaare:

- konstantide vektor on c[], nurksulgudes on indeksid;
- (lokaalsete) muutujate vektor on m[], nurksulgudes on indeksid;
- kui tabelis on [i], siis see tähendab, et indeks tuleb esitada ilmutatud kujul. Muidu määrab indeksi üheselt käsukood ise;
- konstantide ja muutujate tüübid on määratud nendega opereerivate käskude mnemokoodide prefiksitega (*i – int*, *f – float*, *a – address* jne);
- mõnede käskude sufiks *_w* osutab 2-baidisele indeksile (maksimaalväärtusega $ffff_{16}$); sama pika indeksruumi kasutamiseks tuleb üldjuhul kodeerida käsk *wide*;
- *push* tähistab magasini kirjutamist (väärtused vektoritest või vahetud operandid);
- A tähistab *vektorit* (käsus tuleb näidata ilmutatud kujul indeksit), *A tähistab viita vektorile;
- magasinioperatsioonide semantikat kirjeldame „*FORTH*i notatsioonis" kujul „magasini seis enne operatsiooni --- magasini seis pärast operatsiooni". Sedatüüpi operaatorid on *pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2* ja *swap*;
- sama notatsiooni kasutame ka aritmeetika- ja loogikatehete kirjeldamisel, kusjuures *a* ja *b* on operandid, mille tüüp on määratud käsukoodiga. Täisarv-tüüpi arvude jagamise (mnemokood *idiv* või *ldiv*) resultaadiks on jagatise täisosa;
- *b>>a* tähistab *b* väärtuse nihutamist *a* kahendkohta paremale (*b<<a* – vasakule), nihutajana toimivad *a* viis kahendkohta;
- käsk *iinc* evib kaht operandi, *a* ja *b*, *a* on lokaalse muutuja indeks ning selle väärtust modifitseeritakse *b* väärtusega. Näiteks, *iinc 1 10* puhul *m[1]:=m[1]+10* ja *iinc 1 –1* puhul *m[1]:=m[1]–1* (eks ole, tuttavad *increment*- ja *decrement*-režiimid *PDP*-arhitektuurist ja *C*-keelest);
- teisendusprogrammide puhul tähistame magasinielemente nende tüüpide prefiksitega;
- võrdlustehete operandid on *a* ja *b* (võivad hõivata ka kaks magasinielementi, sel juhul on kirjutatud *a1 a2* ja *b1 b2*) ning võrdlustehte resultaat kirjutatakse magasini operandide (2 või 4 elemendi) asemele. Resultaadiks on *tõeväärtus*;

- kui võrdlustehte mnemokoodil pole *sufiksit*, siis *resultaat t* määratakse järgmiselt: kui $a<b$, siis −1, kui $a=b$, siis 0 ja kui $a>b$, siis 1. Sufiks võib olla kas *l* (*less than*) või *g* (*greater than*), sel juhul on *t* kas 1 või 0
- tingimusliku suunamise käsu assemblerformaat on <kood><märgend>. Baitkoodi interpretaatori jaoks on magasini tipus enne käsu täitmist *tõeväärtus* ja sellest lähtudes kas antakse juhtimine märgendile või ei. Näiteks käsk *ifeq* suunab, kui magasini tipus on 0 (seega, tegelik suunamine toimub nulli järgi) Käsu *FORTH*-stiilis spetsifikatsioon on (t ---). Tabeli semantika-veerus on näidatud tingimus, mille täidetuse korral suunatakse märgendile;
- teine tingimusliku suunamise käskude komplekt ühendab võrdlemis- ja suunamiskäsu; formaat on siingi <kood><märgend>, ent magasinispetsifikatsioon on (a b ---). Koodi prefiks on kas *if_i* või *if_a*, *i* puhul võrreldakse magasinis olevaid *a* ja *b* kui *int*-tüüpi arve ning *a* puhul on *a* ja *b* viidad (aadressid), võrreldakse aadresse (ja mitte neil olevaid väärtusi);
- tingimusteta suunamise formaat on ka <kood><märgend>, ent selle käsuga antakse juhtimine alati üle märgendatud operaatorile;
- alamprogrammi pöördumine *jsr, jsr_w* on samuti formaadiga <kood><märgend>, kus märgendiga on varustatud alamprogramm (*sr=subroutine*). Spetsifikatsioon on järgmine: (--- naasmisaadress). Naasmisaadressi kasutab väljakutsutav alamprogramm (see on *jsr*-käsu aadress+3 ja *jsr_w* puhul käsu aadress +5, *w=wide*);
- *ret*-käsuga väljastatakse muutujate tabeli *m* indeks *i*: alamprogramm kirjutas sinna oma resultaadi (so, *resultaat* $\rightarrow$ *m[i]*);
- suur grupp käske on küllaltki spetsiifilised (ja keerukad) – näiteks *tableswitch, lookupswitch, getrstatic, putstatic* jpt. Meie raamatu maht (ja suunitlus) ei võimalda neid lähemalt tutvustada, alljärgnevas tabelis on nende „semantika" veerus piirdutud vaid osutamisega nende käskude kõige üldisemale otstarbele. Detailset infot soovitame hankida [119];

dec	**hex**	**mnemo**	**semantika**	dec	**hex**	**mnemo**	**semantika**
0	00	nop	goto next	1	01	aconst_null	push Ø-viit
2	02	iconst_m1	push int –1	3	03	iconst_0	push c[0]
4	04	iconst_1	push c[1]	5	05	iconst_2	push c[2]
6	06	iconst_3	push c[3]	7	07	iconst_4	push c[4]
8	08	iconst_5	push c[5]	9	09	lconst_0	push c[0]
10	0a	lconst_1	push c[1]	11	0b	fconst_0	push c[0]
12	0c	fconst_1	push c[1]	13	0d	fconst_2	push c[2]
14	0e	dconst_0	push c[0]	15	0f	dconst_1	push c[1]
16	10	bipush	push bait	17	11	sipush	push 2 baiti
18	12	ldc	push 32b-konst.	19	13	ldc_w	2-baidine index
20	14	ldc2_w	push 64b-konst	21	15	iload	push m[i]
22	16	lload	push m[i]	23	17	fload	push m[i]
24	18	dload	push m[i]	25	19	aload	push viit (m[i])
26	1a	iload_0	push m[0]	27	1b	iload_1	push m[1]
28	1c	iload_2	push m[2]	29	1d	iload_3	push m[3]

30	1e	lload_0	push m[0]	31	1f	lload_1	push m[1]
32	20	lload_2	push m[2]	33	21	lload_3	push m[3]
34	22	fload_0	push m[0]	35	23	fload_1	push m[1]
36	24	fload_2	push m[2]	37	25	fload_3	push m[3]
38	26	dload_0	push m[0]	39	27	dload_1	push m[1]
40	28	dload_2	push m[2]	41	29	dload_3	push m[3]
42	2a	aload_0	push viit m[0]	43	2b	aload_1	push viit m[1]
44	2c	aload_2	push viit m[2]	45	2d	aload_3	push viit m[3]
46	2e	iaload	push int A[]	47	2f	laload	push long A[]
48	30	faload	push float A[]	49	31	daload	push double A[]
50	32	aaload	push *A	51	33	baload	push boolean A[]
52	34	caload	push char A[]	53	35	saload	push short A[]
54	36	istore	pop m[i]	55	37	lstore	pop m[i]
56	38	fstore	pop m[i]	57	39	dstore	pop m[i]
58	3a	astore	pop viii m[i]	59	3b	istore_0	pop m[0]
60	3c	istore_1	pop m[1]	61	3d	istore_2	pop m[2]
62	3e	istore_3	pop m[3]	63	3f	lstore_0	pop m[0]
64	40	lstore_1	pop m[0]	65	41	lstore_2	pop m[2]
66	42	lstore_3	pop m[3]	67	43	fstore_0	pop m[0]
68	44	fstore_1	pop m[1]	69	45	fstore_2	pop m[2]
70	46	fstore_3	pop m[3]	71	47	dstore_0	pop m[0]
72	48	dstore_1	pop m[1]	73	49	dstore_2	pop m[2]
74	4a	dstore_3	pop m[3]	75	4b	astore_0	pop viit m[0]
76	4c	astore_1	pop viit m[1]	77	4d	astore_2	pop viit m[2]
78	4e	astore_3	pop viit m[3]	79	4f	iastore	pop int A[]
80	50	lastore	pop long A[]	81	51	fastore	pop float A[]
82	52	dastore	pop double A[]	83	53	aastore	pop *A
84	54	bastore	pop boolean A[]	85	55	castore	pop char A[]
86	56	sastore	pop short A[]	87	57	pop	a ---
88	58	pop2	a b ---	89	59	dup	a --- a a
90	5a	dup_x1	a b --- b a b	91	5b	dup_x2	a b c --- c a b c
92	5c	dup2	a b --- a b a b	93	5d	dup2_x1	a b c --- b c a b c
94	5e	dup2_x2	a b c d --- c d a b c d	95	5f	swap	a b --- b a
96	60	iadd	a b --- a+b	97	61	ladd	a b --- a+b
98	62	fadd	a b --- a+b	99	63	dadd	a b --- a+b
100	64	isub	a b --- a-b	101	65	lsub	a b --- a-b
102	66	fsub	a b --- a-b	103	67	dsub	a b --- a-b
104	68	imul	a b --- a×b	105	69	lmul	a b --- a×b
106	6a	fmul	a b --- a×b	107	6b	dmul	a b --- a×b
108	6c	idiv	a b --- a/b	109	6d	ldiv	a b --- a/b
110	6e	fdiv	a b --- a/b	111	6f	ddiv	a b --- a/b
112	70	irem	a b --- jääk(a/b)	113	71	lrem	a b --- jääk(a/b)

114	72	frem	a b --- jääk(a/b)	115	73	drem	a b --- jääk(a/b)
116	74	ineg	a --- (–1)×a	117	75	lneg	a --- (–1)×a
118	76	fneg	a --- (–1)×a	119	77	dneg	a --- (–1)×a
120	78	ishl	a b --- a ×2^b	121	79	lshl	a b --- a ×2^b
122	7a	ishr	a b --- a /2^b	123	7b	lshr	a b --- a /2^b
124	7c	iushr	a b --- b>>a	125	7d	lushr	a b --- b>>a
126	7e	iand	a b --- a&b	127	7f	land	a b --- a&b
128	80	ior	a b --- a|b	129	81	lor	a b --- a|b
130	82	ixor	a b --- a€b	131	83	lxor	a b --- a€b
132	84	iinc	kommentaarid	133	85	i2l	i --- l1 l2
134	86	i2f	i --- f	135	87	i2d	i --- d
136	88	l2i	l --- i	137	89	l2f	l --- f
138	8a	l2d	l --- d	139	8b	f2i	f --- i
140	8c	f2l	f --- l	141	8d	f2d	f --- d
142	8e	d2i	d --- i	143	8f	d2l	d --- l
144	90	d2f	d --- f	145	91	i2b	i --- b
146	92	i2c	i --- c	147	93	i2s	i --- s[hort]
148	94	lcmp	a1a2 b1b2 --- t	149	95	fcmpl	a b --- a<b
150	96	fcmpg	a b --- a>b	151	97	dcmpl	a b --- a<b
152	98	dcmpg	a b --- a>b	153	99	ifeq	mine, kui =
154	9a	ifne	mine, kui ≠	155	9b	iflt	mine, kui <
156	9c	ifge	mine, kui ≥	157	9d	ifgt	mine, kui >
158	9e	ifle	mine, kui ≤	159	9f	if_icmpeq	mine, kui a=b
160	a0	if_icmpne	mine, kui a≠b	161	a1	if_icmplt	mine, kui a<b
162	a2	if_icmpge	mine, kui a≥b	163	a3	if_icmpgt	mine, kui a>b
164	a4	if_icmple	mine, kui a≤b	165	a5	if_acmpeq	mine, kui a=b
166	a6	if_acmpne	mine, kui a≠b	167	a7	goto	mine
168	a8	jsr	mine alamprogr	169	a9	ret	tagasta indeks
170	aa	tableswitch	*switch*-operaat.	171	ab	lookupswitch	*switch*-oper.
172	ac	ireturn	tagasta *int*	173	ad	lreturn	tagasta *long*
174	ae	freturn	tagasta *float*	175	af	dreturn	tagasta *double*
176	b0	areturn	tagasta *viit*	177	b1	return	välju *void*
178	b2	getstatic	anna *klass.väli*	179	b3	putstatic	lisa väli klassile
180	b4	getfield	anna objekti väli	181	b5	putfield	lisa väli klassile
182	b6	invokevirtual	aktiveeri virt. meetod	183	b7	invokespecial	aktiveeri objekti meetod
184	b8	invokestatic	aktiveeri staat. meetod	185	b9	invokeinterface	aktiveeri liides
186	ba		ei kasutata	187	bb	new	loo klass
188	bc	newarray	loo vektor	189	bd	anewarray	loo viidavektor
190	be	arraylength	annab vektori pikkuse	191	bf	athrow	veateade
192	c0	checkcast	tüübikontroll	193	c1	instanceof	tüübikontroll

194	c2	monitorenter	sünkroniseerim.	195	c3	monitorexit	sünkronis.
196	c4	wide	2-baidine index	197	c5	multianewarray	loo massiiv
198	c6	ifnull	mine, kui viit=Ø	199	c7	ifnonull	mine, kui viit ≠Ø
200	c8	goto_w	mine 2-baidise nihkega	201	c9	jsr_w	mine alam-programmi (2-baidine nihe)
202	ca	breakpoint	reserv. silujale	254	fe	impdep1	reserveeritud
255	ff	impdep2	reserveeritud				

11. Netkood

Esitame siin fragmendi *.NET*-baitkoodi ja *MSIL* (*CIL*)-direktiivide transleerimistabelist. Juhuslikult on valitud näiteks käsugrupp "vii magasini (*push*) massiivi element", kus-juures magasinis peab olema nii massiivi algusaadress kui ka indeksi väärtus, ent noid seiku alljärgnev tabel ei kajasta. Kui vektori aadress on A, siis $A[i]$ tähistab vektori elementi (meie tabelis: elemendi aadressi) ning koodide eristamine johtub operandi (massiivi elemendi) tüübist (*int, unsigned int* ja *float*) ning pikkusatribuudist (*int8=bait*, *int16*=2 *baiti, int32*= 4 *baiti* või *int64*=8 *baiti*); magasini kirjutatakse nad kõik kui 32-bitised arvud (8baidised arvud võtavad enda alla kaks magasinielementi). Baitkood on antud 16-ndarvuna.

Kood$_{16}$	**Direktiiv**	**Semantika**
90	ldelem.i1	push `int8`
92	ldelem.i2	push `int16`
94	ldelem.i4	push `int32`
96	ldelem.i8	push `int64`
91	ldelem.u1	push `uint8`
93	ldelem.u2	push `uint16`
95	ldelem.u4	push `uint32`
96	ldelem.u8	sama, mis *ldelem.i8*
98	ldelem.r4	push `float32`
99	ldelem.r8	push `float64`

INDEKS