Programski jezik AEC

Sažetak: *Danas smo okruženi programibilnim elektroničkim uređajima: računalima, tabletima, mobitelima... Koristimo ih svakodnevno, no znamo da su oni, sami po sebi, manje korisni nego neprogramirljiva elektronika kao što su kalkulatori ili daljinski upravljači. Ono što ih čini korisnijima od neprogramirljive elektronike jesu programi. Za pisanje teksta koristimo programe kao što su Microsoft Office Word, LibreOffice Writer ili WordPad. Za pristup internetu koristimo programe kao što su Internet Explorer, Chrome, Firefox, Edge, Safari ili NetSurf. U stvari, na mnogim programibilnim elektroničkikm uređajima danas, od nekog se programa koji je spremljen na poluvodičkoj memoriji očekuje da pri paljenju uređaja upali ekran, tako da bi bez programa (što se događa ako poluvodička memorija, na kojoj je taj program zapisan, kroz desetljeća ispari do nečitljivosti) programibilni elektronički uređaj bio manje koristan od baterijske lampe, ekran mu se ne bi upalio iako mu je ostatak elektronike ispravan. No, kako se ti programi prave? Većina nas zna da se programi pišu u jezicima posebno dizajniranim da ih relativno lako mogu razumjeti i programirljiva elektronika i ljudi, oni se zovu programski jezici. Mnogi od nas su vjerojatno i napisali već neke jednostavnije programe u nekom od poznatijih programskih jezika. Ali, kako se rade programski jezici? Autor ovog teksta istražio je upravo to radeći pojednostavljeni programski jezik pod nazivom AEC, Arithmetic Expression Compiler.*

Programski jezici, compileri i interpreteri

Program je niz naredbi razumljivih računalu, upute računalu kako će raditi ono što treba. Riječ *program* dolazi od grčkog *pro,* što znači *prije*, i *gramma*, participa od *graphein*, *pisati*, dakle prevela bi se kao *predzapis*, u smislu *ono što je napisano prije no što je računalo počelo nešto raditi*. Programski jezici su jezici dizajnirani da bi bili razumljivi i računalima i ljudima, i u njima se pišu programi.

Ako želimo razumjeti programske jezike, korisno je da poznajemo Moravecov paradoks. On kaže da ono što mi, kao bića s velikim mozgom s dijelovima specijaliziranim za svakakve radnje, percipiramo kao lagano ili teško, ima malu korelaciju s onime što računalo percipira kao lagano ili teško. Iz tog su razloga oni programski jezici koji su ljudima lakši za razumjeti, računalu teži. On nas ne sputava samo u tome da razmišljamo o računalima, nego i u tome da razmišljamo o tome što osjećaju druga živa bića. Recimo, mnogi ljudi vjeruju da životinje s iznimno jednostavnim mozgovima, kao što su kukci, osjećaju bol, pa čak i da biljke osjećaju bol, zato što se nama bol čini kao jednostavan osjećaj, iako je jako nevjerojatno da kukci osjećaju bol, a pogotovo da biljke osjećaju bol, jer ono što se u mozgu mora dogoditi da osjetimo bol zapravo je jako komplicirano. Iluzija da je bol nekakav univerzalan osjećaj toliko je jaka da ljudi nastavljaju vjerovati da ribe osjete bol čak i kad se suoče s činjenicom da ribe kojima je probušena peraja nastavljaju plivati kao da se ništa nije dogodilo. Isto tako, ljudi oklijevaju prije no što pripišu osnovno logičko zaključivanje nekoj životinji, a zapravo je ono što se u našem mozgu događa kad logički razmišljamo veoma jednostavno, to može i računalo raditi.

Drugo, moramo shvatiti da računala zapravo ne govore različitim jezicima. Računala, kao i druga programirljiva elektronika, razumiju samo neki jezik sastavljen od nula i jedinica, različit za svaki tip računala. Na disku računala (vrsti memorije iz koje se podaci ne brišu kad se računalo ugasi, zvan tako jer su prvi diskovi uistinu imali oblik diska), ili na bilo kojem mediju za pohranu podataka, postoje tri vrste datoteka (engleska riječ za datoteku je *file,* dolazi od latinskog *filum* što znači *nit*, ovdje u smislu *nule i jedinice koje su međusobno povezane u lanac*, a hrvatska riječ dolazi od latinskog *datum*, podatak). Jedna vrsta datoteka jesu dokumenti. To su datoteke koje se mogu otvoriti nekim programom i spremiti iz nekog programa (isprva su to većinom bili tekstovi, odakle i naziv *dokument*). Tako postoje DOCX datoteke, koje možemo otvoriti Wordom ili LibreOfficeom. Postoje MP3 datoteke, njih mnogi programi, kao i Windows Media Player, mogu otvoriti, a neki ih, poput Audacityja, mogu i uređivati. Postoje JPG datoteke, i njih isto tako možemo otvoriti raznim programima, kao što je Preview, a neki programi te datoteke mogu i uređivati, kao što su Paint, GIMP ili Photoshop. Druga vrsta datoteke, osim dokumenta, je direktorij (latinski *dirigere* znači,

između ostalog, *pokazivati put*). To su posebne datoteke koje programima govore kako su raspoređene datoteke na disku. Programima kao što su Windows Explorer ili Linux Finder njih možemo slijediti da bismo našli datoteku koju tražimo, a možemo ih i uređivati kako bismo kasnije lakše našli neku datoteku. Postoje i treća vrsta datoteka, one se zovu izvršne datoteke. To su programi u obliku u kojemu ih računalo može pokrenuti, te datoteke sadrže nule i jedinice koje razumije procesor računala. Na Windowsima to su EXE datoteke, na Linuxu to su ELF i ELF64 datoteke. U programirljivoj elektronici koju pogone Intelovi i AMD-ovi procesori, kao većina laptopa ili računala, one imaju sličnu strukturu. Za elektroniku koju pogone ARM-ovi procesori, kao što su većina mobitela i tableta, te se datoteke već znatnije razlikuju. Na većini računala nalazi se Notepad, i to u datoteci koja se zove, zajedno s direktorijima koji na nju upućuju, `C:\Windows\system32\NOTEPAD.EXE`. Za DOCX datoteke možemo reći da smo ih otvorili u Wordu ili LibreOfficeu. EXE datoteke, kao što je `NOTEPAD.EXE`, možemo otvoriti, ali nema previše smisla reći da smo ih otvorili u nekom programu. To jest, možemo ih otvoriti u programima kao što je Hexdump, da vidimo te nule i jedinice, no to je rijetko kad korisno (Hexdump je koristan većinom zato što omogućuje čitanje početaka binarnih datoteka da vidimo u kojem su one zapravo formatu). Mogli bismo reći da smo Notepad otvorili Windowsima, u smislu da su Windowsi našli slobodan prostor u radnoj memoriji (dio računala koji pamti podatke koji se trenutno obrađuju i programe koji se trenutno vrte, latinski *memoria* znači *sjećanje* ili *pamćenje*), kopirali Notepad tamo i usmjerili procesor da izvršava nule i jedinice koje se tamo nalaze. Također, na većini računala ne nalazi se program Notepad pisan u programskom jeziku u kojem je pisan. U stvari, protuzakonito je to imati na svom računalu bez dopuštenja Microsofta, a i to krajnjem korisniku vjerojatno ne bi koristilo. Da bismo pokrenuli program napisan u nekom programskom jeziku, a ne u nulama i jedinicama, potrebni su nam posebni programi koji se zovu compileri (latinski *con* znači *zajedno*, a *pilare* znači *zaglaviti nešto negdje*, *compilare* znači *sastaviti*) i interpreteri (latinski *interpretor* znači *prevoditelj*, od *inter* što znači *između* i *pres* što znači *govor*, kao *onaj tko govori između naroda*). Računalo bez njih te programe ne razumije, baš kao ni čovjek koji ne poznaje programski jezik u kojem su oni napisani. Podjela, danas donekle zastarjela, tih programa na compilere i interpretere temelji se na tome da programi koji se zovu compileri spremaju nule i jedinice u izvršne datoteke koje možemo kasnije pokrenuti, dok interpreteri te nule i jedinice izvršavaju čim rečenicu iz programskog jezika prevedu na njih, bez spremanja. Da bi netko mogao pokrenuti program koji smo mi compilirali, ne treba mu compiler. S druge strane, ako netko želi otvoriti program koji se interpretira, treba mu interpreter. Naravno, interpreteri za neke jezike toliko su česti da ih ljudi imaju na računalima, a da to možda i ne znaju. Recimo, sastavni dio svakog današnjeg internetskog preglednika je interpreter za JavaScript. U Internet Exploreru se on zove Chakra, u Edgeu se on zove ChakraCore, u Chromu se on zove V8, u Firefoxu je to SpiderMonkey, u Safariju je to Nitro, u NetSurfu je to Duktape, i tako dalje. Strogo govoreći, jedino je Duktape od njih zapravo interpreter, drugi funkcioniraju tako da stvaraju privremene izvršne datoteke, no, iz perspektive krajnjeg korisnika i drugih programa, ponašaju se kao interpreteri. Zato, kada u Chromeu ili Firefoxu otvorite igru PacMan[1] koju sam napisao na JavaScriptu i istovremeno pritisnete `Ctrl` i `U`, možete vidjeti kako taj program izgleda u programskom jeziku JavaScript, zajedno s komentarima koje sam pisao na hrvatskom jeziku. Naravno, kako Chrome i Firefox nisu primarno alati za programiranje, već programi za surfanje internetom, u njima taj JavaScriptski program neće izgledati lijepo. U stvari, koliko znam, od svih popularnih internetskih preglednika, jedino Safari uopće pokušava obojati različite vrste riječi iz JavaScripta različitom bojom, što se od svakog alata za programiranje očekuje. Zato se na web-stranici nalazi link gdje se ta igrica nalazi na mom GitHub profilu, a GitHubovi serveri u Kaliforniji, između ostalog, besplatno preuređuju datoteke pisane na programskim jezicima tako da i u internetskom pregledniku izgledaju približno kao u alatu za programiranje koji razvija tvrtka Github, on se zove Atom. Također, zato što postoje interpreteri za JavaScript za mnoge programibilne elektroničke uređaje, možete otvoriti taj PacMan i na svom smartphoneu i on će se tamo najvjerojatnije vrtjeti jednako dobro kao i na računalu, iako elektronika u smartphoneu uopće nije kompatibilna s elektronikom u računalu, ARM procesori

1 https://flatassembler.github.io/pacman.html

funkcioniraju na drukčijim principima nego Intelovi i AMD-ovi procesori. Ali, zato, ako imate Windows XP i probate taj PacMan pokrenuti u Internet Exploreru 6, on se neće vrtjeti (jer Internet Explorer 6 ima dosta ograničenu podršku za JavaScript), dok će se, na tom istom računalu, on moći vrtjeti u Firefoxu 52 ili Operi 33. Naravno, nikad se neće vrtjeti tako dobro kao da je compiliran za baš taj elektronički uređaj. Također, Word u sebi sadrži interpreter za VisualBasic.

Asemblerski i strojni jezici

Danas velika većina compilera (jedina iznimka koju znam je *Tiny C Compiler* koji je napisao Fabrice Bellard) zapravo ne prevodi iz programskog jezika izravno u nule i jedinice, nego prevodi iz programskog jezika u jezik koji je, tako reći, između strojnog jezika i programskog jezika, on se zove asemblerski jezik. Zatim oni pozivaju posebne compilere koji znaju prevoditi iz asemblerskog jezika u nule i jedinice, oni se zovu asembleri (francuski *assembler* isto znači *sastaviti*, od latinskog *assimulare*). Poznatiji asembleri su *Microsoft Macro Assembler* (koriste ga Microsoftovi compileri koji se nalaze u sklopu Visual Studia, dostupan je besplatno na internetu, ali protuzakonito ga je koristiti za svoje compilere bez Microsoftova dopuštenja), *Flat Assembler* (njega koristi compiler za AEC, i njega autor ovog teksta smatra najboljim, *Flat Assembler* je ujedno i IDE za asemblerski jezik, to jest, u njemu se može pisati i program i on, između ostalog, oboji različite vrste riječi na asemblerskom jeziku različitom bojom), *Net Assembler* (njega je, tako reći, iz mrtvih oživio Fabrice Bellard kada ga je koristio da napiše programe za brzo pretvaranje datoteka koje sadrže video i zvuk iz jednog formata u drugi, koji su stekli široku primjenu), *GNU Assembler* (dobije se uz Linux, njega koriste poznati besplatni compileri za C i C++ kao što su GCC i CLANG) i *Turbo Assembler* (razvija ga tvrtka Borland za svoje compilere koji nisu besplatni, a ni on sam nije besplatan). Kakav je onda taj jezik koji razumiju asembleri? Pa, da citiram što nam je profesor Željko Hederić rekao na predavanju: *Programeri koji su završili samo srednju školu ne moraju uopće razumjeti kako računalo funkcionira. Za njih se programi sastoje od if-grananja, switch-grananja, for-petlji, while-petlji i tako dalje. Pa te stvari zapravo ne postoje. To je nešto što smo mi izmislili da bismo lakše opisali kako funkcioniraju programi, i kasnije smo ih ugradili u programske jezike, ali to su samo apstrakcije, to su fikcije. U programima koji se vrte na računalu nema for-petlji. Računalo nema pojma što je to for-petlja*. Mislim da je stvarno u bit pogodio, u asemblerskom jeziku i u strojnom jeziku takvih stvari na koje smo navikli još od prvih lekcija iz programiranja uistinu nema. Pa, kako se onda tim jezicima mogu opisati programi? Ako išta znate o programiranju, vjerojatno znate da tipična rečenica u asemblerskom jeziku izgleda približno ovako...

```
add eax,ecx
```

i da se ona prevodi kao *Zbroji brojeve koji se nalaze u registrima eax (*extended accumulator register, *produženi akumulatorski registar) i ecx (*extended counter register, *produženi brojački registar) i spremi ih u eax*. Registri su, naime, dijelovi procesora u koje procesor može spremiti male količine podataka, kao što su brojevi ili nekoliko slova, kako ih ne bi morao stalno dohvaćati iz memorije. Manje je poznato da, kada procesor tu naredbu izvrši, on ne nepravi samo to. U procesoru se, osim registara, koji se sastoje od nekoliko baitova podataka (bait, *byte,* je skupina od 8 bitova, to jest, 8 nula ili jedinica), nalaze i takozvane zastavice (*flags*), koje sadrže po jedan bit podataka, to jest, jednu nulu ili jedinicu. Riječ *bit* na engleskom znači *mali dio*, u smislu *najmanja moguća količina informacije*. Riječ *byte* skovana je zato što bi se na mnogim engleskim dijalektima (iako ne i na standardnom, to jest, na londonskom dijalektu) *byte* izgovaralo *bît*, to jest, isto kao i *bit* samo s dugim naglaskom (*bit* se izgovara *bȉt*). Jedna od zastavica koja se nalazi u svakoj procesorskoj jezgri je *carry flag* (zastavica koja nosi, u smislu, zastavica koja nosi prvu jedinicu rezultata ukoliko ona nije stala u registar), *CF*, koja se postavlja u jedinicu ukoliko je prva znamenka u oba binarna broja koja su se zbrajala bila jedinica. Jedna od tih zastavica je i *zero flag* (zastavica nule), koja se postavlja u jedinicu ukoliko je rezultat jednak nuli. I u asemblerskom jeziku postoje naredbe kao što su...

```
jz neka_memorijska_adresa
```

i ta naredba znači *Ako je zero flag postavljen u jedinicu, nastavi program izvršavati od memorijske adrese* `neka_memorijska_adresa`*, inače nastavi od iduće naredbe koja slijedi nakon ove u memoriji.* Naredba...

```
jnz neka_memorijska_adresa
```

...znači *Ako zero flag* ***nije*** *postavljen u jedinicu, nastavi program izvršavati od memorijske adrese* `neka_memorijska_adresa`*, inače nastavi od iduće naredbe koja slijedi nakon ove u memoriji.*

I takvim se naredbama mogu raditi grananja i petlje. Sve koje se mogu napraviti u višim programskim jezicima, mogu se napraviti i na asemblerskom i na strojnom jeziku. Ako se to čini nevjerojatnim, moguće je dokazati i mnogo radikalnije tvrdnje, recimo, da bi Turingov stroj (hipotetski stroj od trake s poljima u kojima se nalaze simboli iz neke abecede, čitača trake i registra koji pamti jedan takav simbol, koji razumije jedino naredbe oblika *Ako na traci čitaš simbol A, a u registru se nalazi B, napiši na traku C, a u registar D, i pomakni čitač trake ulijevo.*) mogao napraviti sve što mogu i današnja računala, samo što bi bio beskrajno sporiji. U stvari, isto vrijedi i za binarni Turingov stroj, Turingov stroj sa samo dva simbola u abecedi.

Većina današnjih compilera, kada dovrši s prevođenjem programa na asemblerski jezik, sami pozovu asembler. Compiler za AEC to ne radi, jer je to komplicirano za isprogramirati, a i cilj je da korisnici razumiju kako on radi, da se vidi da on prevodi na asemblerski jezik, a da dalje nije njegov posao.

Dijelovi compilera

Compiler za AEC pisan je u JavaScriptu i mnogi njegovi dijelovi mogu se pokrenuti u bilo kojem donekle modernom internetskom pregledniku, uključujući i Internet Explorer 6, koji je dostupan na autorovom blogu[2]. Dok se vrti u internetskom pregledniku, može prevoditi samo aritmetičke izraze iz AEC-a na asemblerski. Dok je moj blog bio na serverima (*server* ili *poslužitelj* je veliko računalo na kojem se nalaze web-stranice, mailovi i slično, i koji poslužuje druga, obično manja, računala, a ne izravno ljude, naziv dolazi od latinskog *servus*, sluga) tvrtke 000webhost na Cipru, bilo je moguće iz internetskog preglednika skinuti gotov projekt koji se može otvoriti u FlatAssembleru i kao takav prevesti na strojni jezik, naime, da program na serveru, koji sam napisao, od asemblerskog koda (*code* je francuska riječ za *tekst*, na engleskom i hrvatskom danas se koristi u značenju *tekst na programskom jeziku*) koji mu pošalje compiler napravi cijeli projekt FlatAssemblerskog IDE-a (*integrated developing environment*, ugrađena okolina za razvoj, program namijenjen tome da se u njemu pišu programi na nekom programskom jeziku, koji boja razne vrste riječi u programskom jeziku različitom bojom i koji upravlja compilerom, asemblerom, linkerom i drugim programima koji su potrebni za prevođenje programa na strojni jezik, da se programer o njima ne mora brinuti). No, 000webhost me zabranio jer je njegov moderator smatrao da je nešto što sam napisao na svom blogu govor mržnje. Ja sam onda svoj blog prebacio na GitHub, koji je poznat po tome da dopušta da se na njemu objavljuju nepodobne stvari. Međutim, GitHub ne dopušta da se programi koje smo napisali vrte na njihovim serverima, tako da to skidanje već gotovog FlatAssemblerskog projekta više ne funkcionira. Programski kod koji je to omogućavao, pisan u PHP-u, dostupan je na mom GitHub profilu[3], da ga netko tko želi imati AEC na svojoj web-stranici (da objasni osnove teorije o compilerima) ne mora ponovno pisati.

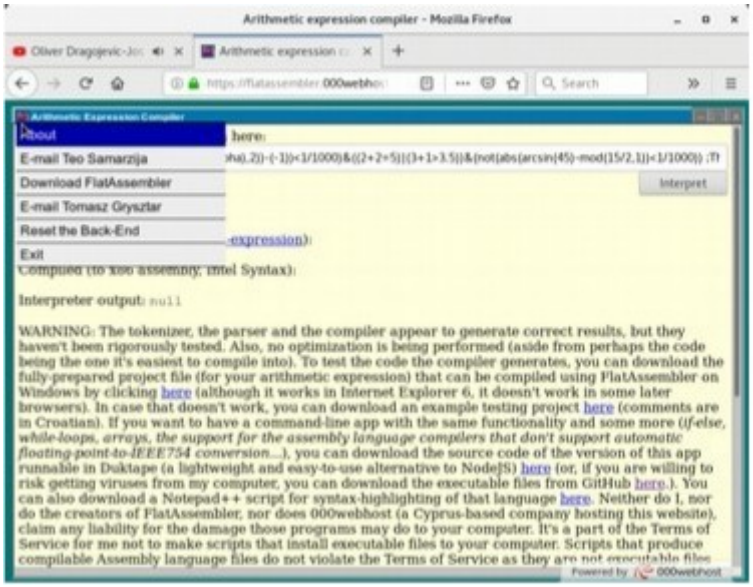
Compiler u širem smislu sastoji se od: drivera, pretprocesora, thin-layersa, tokenizera, leksičkog analizera, parsera, sintaksnog highlightera, compilera u užem smislu, emitera, asemblera, linkera i statičkog analizera. Kad se AEC pokrene u internetskom pregledniku, on, osim asemblerskog koda, ispisuje i međurezultate koje daju tokenizer i parser.

Driver (engleski *driver* znači *vozač* ili *upravljač*) je dio compilera koji pokreće druge dijelove compilera i omogućuje komunikaciju među njima. On se obično prvi otvara i zadnji zatvara, te on pruža sučelje korisniku ili IDE-u. Grafičko sučelje koje vidimo kad otvorimo AEC u internetskom pregledniku moglo bi se smatrati driverom. Riječ *driver* ima i jedno drugo, znatno različito, značenje u informatici, kao program koji omogućuje računalu da komunicira s nekim uređajem.

2 https://flatassembler.github.io/compiler.html

3 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/tree/master/back-end

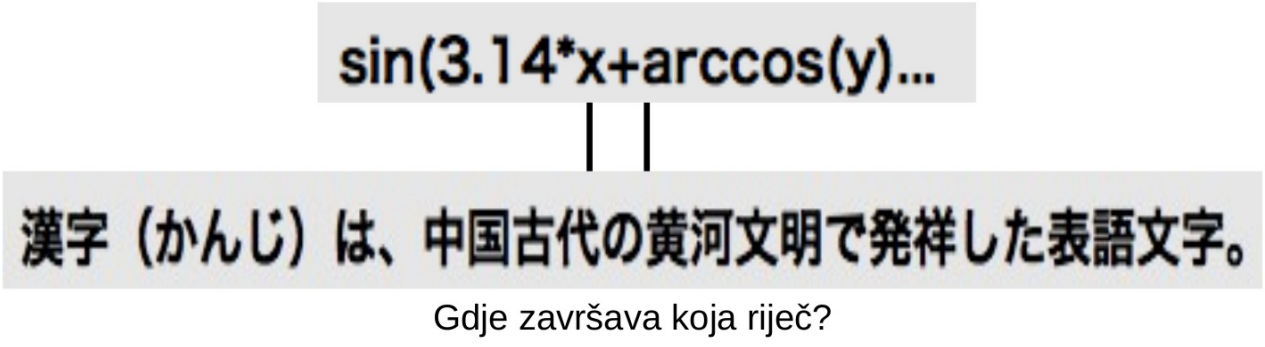
Kada kažemo da smo instalirali printer, zapravo mislimo da smo instalirali driver za printer. Osim što je riječ o računalnim programima, veze između ta dva pojma više nema.



Slika 1: Grafičko sučelje koje AEC stvara kad se pokrene u internetskom pregledniku (ovdje u Firefoxu) primjer je drivera.

Pretprocesori i thin-layersi (*thin* znači *tanak*, *lay* znači ležati, *layer* znači *ono što leži* ili *sloj*) potrebni su za neke jezike, ali ne sve. Oni obrađuju datoteku prije no što je predaju compileru. Pretprocesori općenito služe da više datoteka, koje je u alatima za programiranje lakše uređivati, spoje u jednu datoteku, kako je compileru lakše obraditi. Thin-layersi pretvaraju iz visokog jezika u malo niži, recimo, Objective-C se prije compiliranja pretvara u C. Pretprocesori se u svojoj moći razlikuju od jezika do jezika. C-ov pretprocesor omogućuje vrlo malo toga, dok su PERL-ov ili FlatAssemblerov, recimo, Turing-potpuni (Turing-complete), mogu sve što može i taj jezik ako se interpretira.

Tokenizer (engleski *token* znači *oznaka*) dio je compilera koji razdvaja riječi iz programskog jezika, i taj dio ostalim dijelovima compilera označava gdje završava koja riječ. Naime, netokenizirani tekst na nekom programskom jeziku računalu izgleda otprilike kao što nama izgleda japansko pismo (naravno, ako nismo proučavali japanski jezik). Programski jezici, kao i japansko pismo, imaju mehanizme za razdvajanje riječi, ali oni nisu trivijalni. Riječi iz programskog jezika, kada ih tokenizer izdvoji iz rečenica, zovu se tokeni.



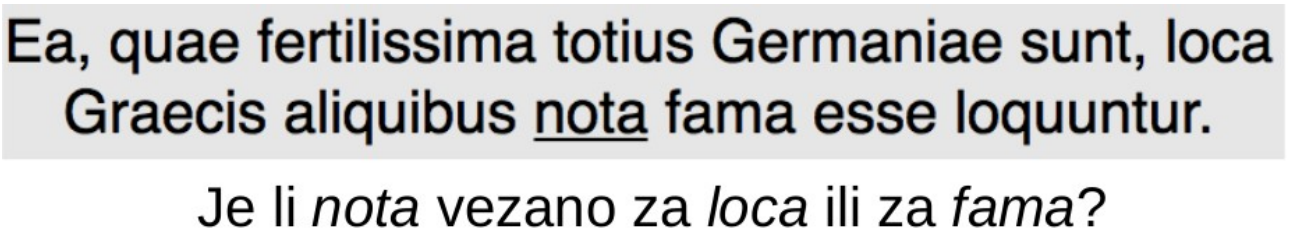
Slika 2: Ilustracija kojom sam u svom prvom seminaru, kod profesora Gorana Martinovića, objašnjavao kako računalu izgleda netokenizirani program

Leksički analizer (*lexis* na grčkom znači *riječ, ana* znači kroz, a *lyein* znači *olabaviti, analizirati* znači *rastaviti*) je dio compilera koji određuje vrste riječi, jer bez toga sintaktička se analiza ne može izvršiti. Programski jezici su obično takvi da se iz samog oblika riječi može odrediti koja je

vrsta, bez poznavanja značenja ijedne riječi u rečenici i bez poznavanja sintaktičkih struktura (kao na esperantu: ako riječ na esperantu završava na -o, -oj, -on ili -ojn, znači da je imenica, i imenice jedino tako i mogu završavati).

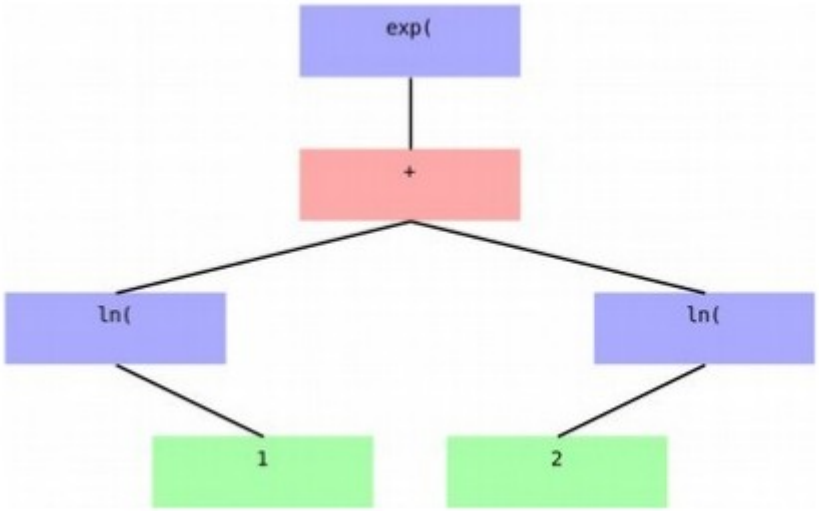
Slika 3: Kako računalu izgleda program koji nije leksički analiziran

Parser (latinski *pars* znači *dio*) je dio compilera koji ostalim dijelovima compilera označava koja je riječ u programskom jeziku sintaksno povezana s kojom. To se radi pravljenjem sintaksnih stabala u memoriji (zvanih AST). Većina compilera, još od doba prvih viših programskih jezika, za parsiranje aritmetičkih i logičkih izraza koristi Dijkstrin Shunting-Yard algoritam (koji se sastoji od jednog prioritetnog reda i jednog stoga; *shunting yard* je engleski naziv za dijelove tračnica gdje vlak skreće pod oštrim kutom, jer je Edsgera Dijkstru to što se događa u memoriji računala dok se on vrti podsjećalo na takvo skretanje vlaka). Njegova prednost je to što se vrti u linearnom vremenu, samo jednom prođe kroz svaki token. Compiler za AEC ga ne upotrebljava, nego je autor osmislio svoj algoritam koji se lako iskaže u JavaScriptu. On se vrti u kvadratnom vremenu ako nema zagrada, te u kubnom vremenu ako ima zagrada (naime, on za svaku izmjenu niza s tokenima poziva JavaScriptinu naredbu `splice`, koja se vrti u linearnom vremenu). Iako je to u teoriji mnogo lošije od Shunting Yarda, u praksi se na današnjim računalima čak i najduži izrazi koji se u praksi nalaze u programima compiliraju u zanemarivo kratkom vremenu.



Slika 4: Kako računalu izgleda program koji nije parsiran

U starijim je internetskim preglednicima jedini način prikazivanja AST-a pretvaranje AST-a u LISP--ove S-izraze, koji početnika u programiranju vjerojatno još više zbunjuju nego aritmetički izrazi kojima su dodane zagrade. Ako se AEC pokrene u modernom internetskom pregledniku (bilo koji koji podržava manipulaciju SVG-a iz JavaScripta, to uključuje i Internet Explorer 11), parser može grafički prikazivati AST-ove aritmetičkih izraza.



Slika 5: Sintaksno stablo (AST) koje je compiler nacrtao za izraz `exp(ln(1)+ln(2))`

Sintaksni highlighter je program koji boja različite vrste riječi u programskim jezicima različitim bojama. Smatra se da je tako tekst na programskom jeziku lakši za čitanje. On ne mora biti vezan za compiler, no danas najčešće jest. Ja sam napisao i jedan koji nije vezan, to jest, skriptu za program zvan Notepad++ koja boja program pisan na AEC-u. Kao zanimljivost, Microsoft Edge ima tu mogućnost za engleski jezik, jer to navodno ubrzava čitanje onima koji slabije znaju engleski.

```
finit
mov [result],2f
fld dword [result]
mov [result],2f
fld dword [result]
faddp st1,st0
mov [result],5f
fld dword [result]
fcomip st1
fstp dword [result]
jna l26167
fld1
jmp l2661
l26167:
fldz
l2661:
fstp dword [result]
```

Sintaksno obojeni assemblerski tekst. Plavo su mnemonike (glagoli), crveno varijable, zeleno brojevi, ružičasto pridjevi, tirkizno registri, a žuto oznake za preskakanje (*labels*).

Compiler u užem smislu zamjenjuje riječi iz AST-a njihovim prijevodima na strojni jezik, pri tome za gotovo sve naredbe koristi kratice (iz asemblerskog jezika) kako bi olakšao traženje grešaka. Program se kasnije može asemblirati i pokrenuti. S obzirom na to da Intelovi i AMD-ovi procesori sadrže stog kojim se jednostavno upravlja, FPU stog, AST-ovi se mogu zamjenjivati prijevodima na

asemblerski jezik redoslijedom koji određuje jednostavan DFS algoritam, tako da se prvo prevedu čvorovi koji su najniže u stablu.
Emiter (*emittere* na latinskom znači *izbaciti* ili *poslati iz*) je dio nekih compilera, on služi da se compiliranje može izvoditi na različitim (virtualnim ili stvarnim) računalima. Compiler za AEC ima emiter koji omogućava da se jedan dio vrti u internetskom pregledniku, a drugi na serveru koji podržava PHP. CLANG, besplatan compiler za C, C++ i Objektivni C, poznat ponajprije po tome što ga koristi XCODE (IDE za pravljenje programa za Mac, iPhone i iPad), ne vrti se izravno na procesoru, već se jedan dio vrti u virtualnoj mašini zvanoj LLVM (low-level virtual machine), a s obzirom na to da on koristi GNU Assembler da bi stvarao strojni kod (koji se vrti izravno na CPU-u), za komunikaciju s njime potreban mu je emiter.
Često je pitanje čemu danas služi asembler, kad postoje viši programski jezici. Pa, kao prvo, compilerima je lakše ispisati asemblerski kod, koji je tekst, nego raditi s binarnim datotekama. Kao drugo, na taj je način lakše naći greške u programu koje je uzrokovala nesavršenost u compileru. U asemblerski se jezik čovjek ipak mora manje udubiti da bi ga razumio nego u nule i jedinice. Treće, što ako stvarno, iz nekog razloga, moramo znati koje će točno nule i jedinice biti u nekoj datoteci? Je li manja vjerojatnost da ćemo pogriješiti ako ih sve ručno pišemo, ili ako koristimo kratice iz asemblerskog jezika?

Assemblerski jezik je, sa svim svojim kraticama, prema strojnom jeziku isto što je i ovo prema engleskom jeziku.

Linker (engleski *link* znači *karika u lancu*) je rječnik manje poznatih riječi iz programskog jezika, kako compiler u užem smislu ne bi morao sve znati. U mnogim programskim jezicima moguće je tvoriti nove riječi te ih spremati u librariese i koristiti u više programa. Linker je program koji zna čitati te librariese. Ustvari, pod riječi linker razumijemo dva površno slična, ali u dubini dosta različita pojma, dinamički linker i statički linker. Svaki današnji operativni sustav programima koji se na njemu vrte pruža linker koji omogućuje da se pozivaju potprogrami iz DLL-ova (dinamičkih linkerskih biblioteka). To se zove *dinamičko linkiranje*. Programi koje sam do sada pisao na AEC-u oslanjaju se na dinamičko linkiranje da bi pozivali funkcije iz standardne biblioteke programskog jezika C (koje autor ovog teksta relativno dobro poznaje i olakšavaju mu posao). Linkeri koji se dobivaju uz compilere za C i slične jezike obično još omogućavaju i *statičko linkiranje*, da se potprogrami iz DLL-ova ugrade u izvršni program, pa da on onda ne ovisi o DLL-ovima da bi se mogao vrtjeti.
Statički analizer dio je compilera koji pokušava otkriti logičke greške u programu, a da ga ne pokreće. To jest, on otkriva dijelove programskog koda koji su na programskom jeziku gramatički ispravni, ali im semantika vjerojatno nije ono što je programer uistinu mislio reći. To je različito od debuggera, debuggeri su programi koji nisu nužno vezani za compiler i koji pokušavaju otkriti te stvari tako što vrte program, i općenito su uspješniji. Najpoznatiji debuggeri su GDB i LLDB.

O compileru za AEC
Compiler za AEC pisan je u JavaScriptu i C-u, povezanih preko Duktape radnog okvira, ima oko 2000 redaka koda. Duktape (engleski za *ljepljiva traka,* u smislu *radni okvir koji spaja dva jako različita programska jezika*) je interpreter za JavaScript pisan u cijelosti u C-u, po standardu C99 (koji prihvaća svaki današnji compiler za C), dostupan pod MIT licencom. Njega koristi i NetSurf internetski preglednik, na sličan način kao što Chrome koristi V8 ili kao što Firefox koristi SpiderMonkey. Neke je stvari znatno lakše napraviti u C-u nego u JavaScriptu, i zato nisam cijeli compiler napisao u JavaScriptu. C sam odabrao zato što smo ga učili na fakultetu, pa će drugim

studentima biti blizak, a i onda sigurno neću naletjeti na probleme s kompatibilnosti, jer compilera za C99 ima za gotovo svaki programibilni elektronički uređaj. Mogao sam za te stvari koristiti i neki drugi jezik, recimo, napravio sam to što sam u C-u radio i u Javi, koristeći radni okvir Rhino. To je također dostupno na mom Github profilu[4], no nisam se potrudio da to bude kompatibilno s uređajima koji nemaju najnoviju verziju Java Runtime Environmenta (tako da ne možemo biti sigurni da će raditi na računalu koje nema najnovije alate za programiranje na Javi). Naime, ono što u JavaScriptu nije lako napraviti, a trebalo mi je za moj compiler, jest pretvaranje decimalnih brojeva u IEEE 754 heksadekadski zapis[5] (kakav se nalazi u strojnom jeziku, FlatAssembler automatski pretvara decimalne brojeve u njega, drugi asembleri to ne rade, a cilj compilera je, naravno, biti kompatibilan sa što većim brojem asemblera) i rad s datotekama. Izvorni kod potprograma pisanog u C-u dostupan je na mom blogu[6], a izvršne datoteke compilera i primjernih programa na AEC-u dostupne su u ZIP-arhivi na mom Github profilu[7] (nisam ih bio stavio na svoj blog jer 000webhost, na čijim je serverima prije bio moj blog, izričito ne dopušta da se izvršne datoteke postavljaju na njihove servere).

Osim aritmetičkih izraza, AEC, kada je pokrenut u okruženju koji mu omogućava pristup datotekama i pretvaranje decimalnih brojeva u heksadekadski zapis, tada on podržava i `If`-grananja (*if* je engleski veznik sa značenjem *ako*), naredbe `Else` (*else* je prilog sa značenjem *inače* ili *ako ne*) i `ElseIf` (*inače ako*), `While`-petlje (*while* je veznik koji znači *dok*) i jednodimenzionalna polja (nizovi podataka iste vrste u radnoj memoriji). Naravno, kao nizak programski jezik, podržava umetanje asemblerskog koda: između `AsmStart` i `AsmEnd`. Umetanje asemblerskog koda je potrebno napraviti barem na početku AEC-ovskog programa, jer compiler za AEC očekuje da programer komunicira s asemblerskim compilerom o tome kako će biti oblikovana izvršna datoteka (i hoće li je uopće biti, da se neće umjesto izvršne datoteke stvoriti DLL ili nešto slično). I to je temeljna razlika u tome kako je dizajniran C, a kako AEC: C cilja na to da se umetnuti asemblerski kod ne mora nikad koristiti, i da se compiler brine o tome kakva će se datoteka stvoriti. Donekle pristupačan uvod u AEC, na hrvatskom jeziku, imate u obliku prezentacije na mom GitHub profilu[8].

Primjere programa u AEC-u imate u već spomenutoj mapi koju sam sažeo programom ZIP. Program `bottles.aec` ispisuje humorističnu ponavljajuću englesku pjesmicu 99 Bottles Of Beer, izvršna datoteka je `bottles.exe`. Program `euclid.aec` traži od korisnika da unese dva prirodna broja, a on će Euklidovim algoritmom (izražen `While`-petljom i `If-Else`-grananjem) naći njihov najveći zajednički djelitelj, izvršna datoteka je `euclid.exe`. Program `fibonacci.aec` računa brojeve u Fibonaccijevom nizu rekurzivnom formulom tako da ih sprema u jednodimenzionalno polje i čita iz jednodimenzionalnog polja, izvršna datoteka je, naravno, `fibonacci.exe`. Program `months.aec`, s izvršnom datotekom `months.exe`, koristi višestruko (`If-ElseIf-Else`) i ugniježđeno (`If-If`) grananje da bi odredio koliko mjesec, čiji je redni broj unio korisnik, ima dana u gregorijanskom kalendaru. Program `permutations.aec` iliti `permutations.exe` najkompliciraniji je od tih programa, on koristi stogove (struktura podataka u memoriji računala gdje podatke možemo dodavati i oduzimati s početka, ali ih ne možemo stavljati na sredinu ili uzimati iz sredine, kao što, kad imamo stog sijena, sijeno možemo uzimati i dodavati samo na vrh i s vrha tog stoga, inače će se stog srušiti) da

4 https://github.com/FlatAssembler/SimpleCalculator/raw/master/SimpleCalculator.zip

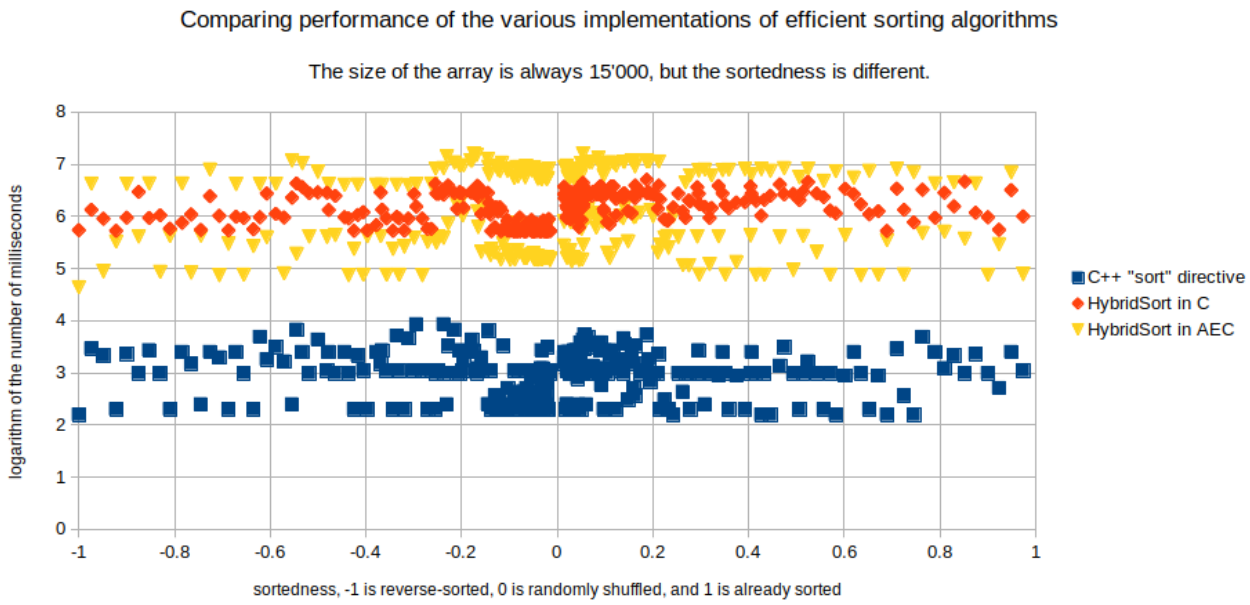
5 Namjerno sam upotrijebio rjeđe korištenu riječ *heksadekadski* umjesto *heksadecimalni*, jer smatram da riječ *heksadecimalni* nije ispravna. Naime, to bi trebalo značiti *koji se odnosi na bazu 16*. No, *heksa* je grčka riječ za broj *šest*, a *deci* je latinski korijen sa značenjem *deset*. Reći *heksadecimalni* je kao da kažemo *sixnaest* ili *šestteen*. Treba reći *heksadekadski* jer je *deka* grčka riječ za broj *deset*, ili reći *sedecimalni*, budući da je *sedecim* latinska riječ za broj 16. Donald Knuth je u *Art of Computer Programming* napisao da bi najsmislenije bilo *heksadekadski* preimenovati u *senidenarni*, to jest, upotrijebiti kao korijen latinski dijelni broj, a ne kardinalni, budući da je heksadekadski sustav zapravo dobio ime po tome što se **sastoji** od 16 znamenki.

6 https://flatassembler.github.io/Duktape.zip

7 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/ArithmeticExpressionCompiler.zip?raw=true

8 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/seminar/Prezentacija.odp?raw=true

bi ispisao sve permutacije znamenki unesenog broja. Program `rose.aec` odnosno `rose.exe` koristi matematičke funkcije kako bi nacrtao lik zvan polarna ruža.
Najkompliciraniji program koji sam napisao u svom programskom jeziku je moj pokušaj da napravim program koji će što brže poredati velik niz brojeva po veličini. Profesor Alfonzo Baumgartner, koji me je ohrabrio da ovo napišem, i Stjepan Poljak napisali su 2005. detaljan opis tog problema i objavili su ga u Osječkom matematičkom listu. Ukratko, postoji QuickSort algoritam, koji je 1959. otkrio Tony Hoare, i on se vrti u linearitmičnom vremenu (dakle, relativno brzo) ukoliko je niz nasumce ispremiješan, ali se vrti u kvadratnom vremenu (što je za velike nizove neprihvatljivo sporo) ukoliko se dogodi da je niz već približno poredan po veličini. Također postoji i MergeSort algoritam, njega je 1945. otkrio John von Neumann (i to je najranije otkriveni efikasan algoritam razvrstavanja po veličini ili po abecedi), i on se uvijek vrti u linearitmičnom vremenu, i treba mu jednako vrijeme da poreda niz brojeva bio on nasumično ispremiješan ili već poredan, ali je na današnjim računalima nekoliko puta sporiji nego što je QuickSort u najboljem slučaju. Autor ovog teksta došao je na ideju da bi se mogao napraviti hibridni algoritam, koji bi se, na temelju toga koliki je postotak niza već poredan, ponašao kao MergeSort ako se čini da je to optimalno ili kao QuickSort ako se čini da je to optimalno. No, mjerenja su pokazala da to najvjerojatnije nije dobro rješenje, barem ne onako kako je autor to isprogramirao:



Kako se vidi na dijagramu, to je višestruko sporije od C++-ove `sort` naredbe (koja radi to isto), i, ne samo to, nego izgleda da taj odabir hoćemo li koristiti MergeSort (sa strana) ili QuickSort (na sredini) uopće ne ubrzava razvrstavanje (nije očito da je QuickSort u najboljem slučaju brži od MergeSorta), kao da odlučivanje da će koristiti QuickSort računalo usporava više nego što ga korištenje QuickSorta kad je on povoljan ubrzava. O tome je autor napisao dugačak seminar i objavio ga je na svom Github profilu[9]. Izvorni kod tog AEC-ovskog programa, s komentarima na hrvatskom jeziku, također je dostupan na autorovom GitHub profilu[10], kao i izvršna datoteka za Windows[11].

Primjer za kraj

Ovaj tekst vjerojatno ne bi bio potpun da ne stavim neki kraći program iz tog svog jezika u njega. Jedan brzi primjer mogao bi biti program koji bi izračunao 20. broj u Fibonaccijevom nizu. Leonardo iz Pise, zvan Fibonacci, bio je matematičar koji je živio krajem 12. i početkom 13. stoljeća. Kao i mnogi tadašnji matematičari, bavio se mnogim prirodnim znanostima. Jedno od

9 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/seminar/seminar.pdf
10 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/raw/master/HybridSort/hsort.aec
11 https://github.com/FlatAssembler/ArithmeticExpressionCompiler/blob/master/HybridSort/hsort.exe?raw=true

pitanja koje je sebi postavio bilo je koliko brzo bi se zečevi razmnožavali ako ima dovoljno hrane za svakoga od njih. Napravio je neke eksperimente. Otkrio je da doista postoji pravilo. Naime, broj zečeva u nekoj generaciji jednak je zbroju broja zečeva u prethodne dvije generacije. Na primjer, ako u sadašnjoj generaciji postoje tri zeca, a u prethodnoj su generaciji bila dva zeca, u sljedećoj će generaciji biti 5 zečeva. Od tada, niz brojeva u kojima je svaki broj jednak zbroju prethodna dva broja naziva se Fibonaccijevim nizom. Nulti broj u tom nizu definiran je kao nula, a prvi broj jedan. Dakle, taj slijed ide ovako: 0,1,1,2,3,5,8,13,21 ... Ima više načina da se napravi program koji će ih računati, najjednostavniji je, naravno, onaj koji pamti te brojeve u jednodimenzionalnom polju.

```
1. AsmStart
2. ;Ovdje umetnuti asemblerski kod specifican za operativni
   sustav gdje ce se program vrtjeti.
3. AsmEnd
4. n:=20
5. i:=0
6. While i<n | i=n ;Vertikalna crta je logicki operator ILI,
   dakle, ovo znaci "Ponavljaj dok je 'i' manje od 'n' ili je
   'i' jednako 'n'.".
7.   If i=0
8.       fib(i):=0
9.   ElseIf i=1
10.      fib(i):=1
11.  Else
12.      fib(i):=fib(i-1)+fib(i-2)
13.  EndIf
14.  i:=i+1
15. EndWhile
16. fib(n) ;Stavi zadnji Fibonaccijev broj koji si izracunao u
   asemblersku varijablu "result".
17. AsmStart
18. ;Ispisati rezultat, deklarirati asemblerske varijable,
   pozvati dinamicki linker, itd...
19. AsmEnd
```

Kao što se vidi iz tog programskog koda, komentari (dijelovi teksta na programskom jeziku koje compiler preskače, najčešće radi dodavanja pojašnjenja na prirodnom jeziku), se u AEC-u pišu između znaka `;` (točka-zarez) i kraja retka, kao i na FlatAssemblerskom dijalektu asemblerskog jezika. Kao što se vidi u 13. i 15. retku, AEC sadržava ključne riječi `EndIf` i `EndWhile`. To sam odlučio prije svega zato što je prvi programski jezik koji sam naučio bio SmallBasic, koji to također ima. Danas većina programskih jezika, uključujući C, Javu i JavaScript, umjesto njih koristi vitičaste zagrade (znakove `{` i `}`). Uostalom, smatram da je lakše pratiti tekst na programskom jeziku ako je jasno završava li u nekom retku grananje ili završava li tamo petlja. Naime, ako imamo naredbe `EndIf` i `EndWhile`, znamo da, kada vidimo `EndWhile`, tamo završava petlja, a da, kada vidimo `EndIf`, tamo završava grananje. U većini današnjih programskih jezika, za oboje koristimo znak `}`. Također, korištenje naredbi `EndIf` i `EndWhile` onemogućava greške u programima poznate pod nazivom *dangling else*. Prednost korištenja vitičastih zagrada je, naravno, to što ekvivalentni programi sadrže manje znakova. U većini programskih jezika postoji, osim while-petlje, i for-petlja (*for* znači *za* ili *za svaki*), koja se upotrebljava u slučaju da je broj ponavljanja unaprijed poznat. Nju je teže implementirati u compiler no što je while-petlju, i zato nje nema u AEC-u. Primjere kako se na asemblerskom jeziku deklarira da želimo stvoriti izvršnu datoteku operacijskog sustava Windows, te kako se ispisuju rezultati, deklariraju asemblerske varijable i uvoze simboli iz DLL-ova, imate u svim onim AEC-ovskim programima na mom GitHub profilu.