

**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH
TEHNOLOGIJA**


Sveučilišni studij

SIMULATOR PICOBLAZE RAČUNALA U JAVASCRIPTU

Završni rad

Teo Samaržija

Osijek, 2022.

 FERIT FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK	
Obrazac D1: Obrazac za imenovanje Povjerenstva za završni rad	
Osijek, 29. 8. 2022.	
Odboru za završne i diplomske radove	
Imenovanje Povjerenstva za završni rad	
Ime i prezime Pristupnika:	Teo Samaržija
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	N/A, 2018
OIB studenta:	50107259317
Mentor:	Izv. prof. dr. sc. Ivan Aleksi
Sumentor:	N/A
Sumentor iz tvrtke:	N/A
Predsjednik povjerenstva:	
Član povjerenstva 1:	Izv. prof. dr. sc. Ivan Aleksi
Član povjerenstva 2:	
Naslov završnog rada:	Simulator PicoBlaze računala u JavaScriptu
Znanstvena grana završnog rada:	Arhitektura računala (zn. polje računarstvo)
Zadatak završnog rada:	Napraviti simulator PicoBlaze računala koji se može vrtjeti u internetskom pregledniku.
Prijedlog ocjene pismenog dijela ispita (završnog rada):	
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	
Datum prijedloga ocjene od strane mentora:	
Potvrda mentora o predaji konačne verzije rada:	Datum:

**FERIT**FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK****IZJAVA O ORIGINALNOSTI RADA****Osijek, 29. 8. 2022.****Ime i prezime studenta:**

Teo Samaržija

Studij:

Preddiplomski sveučilišni studij Računarstvo

Mat. br. studenta, godina upisa:

N/A, 2018.

Turnitin podudaranje [%]:

8

Ovom izjavom izjavljujem da je rad pod nazivom: **Simulator PicoBlaze računala u JavaScriptu** izrađen pod vodstvom mentora izv. prof. dr. sc. Ivan Aleksi
moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature ili drugih izvora informacija.

Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Teo Samaržija

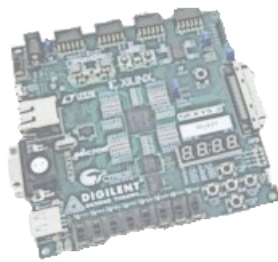
SAŽETAK: Autor teksta bio je frustriran nedostacima postojećih simulatora malog računala PicoBlaze, te je napravio drukčiji simulator PicoBlazea u programskom jeziku JavaScript. Taj se simulator može pokrenuti u modernim internetskim preglednicima¹ (autor misli da je najstariji internetski preglednik u kojem simulator funkcionira kako treba Firefox 52, posljednja verzija Firefoxa koja se može vrtjeti na Windows XP-u, te je to također verzija Firefoxa koja se dobije uz operativni sustav Solaris 11.4, koji je danas najnovija verzija Solarisa). U tekstu slijede detalji o tome kako je autor napravio svoj simulator te koje su prednosti i mane tog simulatora u usporedbi s već postojećim simulatorima. Autor također uspoređuje dijelove simulatora s odgovarajućim dijelovima nekih drugih programa koje je prije napravio (najviše s compilerom koji prevodi njegov programski jezik na WebAssembly). Nisu korišteni nikakvi radni okviri (frameworksi), kôd je pisan uglavnom u VIM-u (za manje izmjene) i Eclipseu (za veće izmjene), za uređivanje slika korišteni su GIMP i Inkscape, za traženje pogrešaka u programu korišteni su alati za programiranje koji se dobiju uz Firefox i internetski servis LGTM. Za formatiranje koda korišteni su Prettier (za HTML i CSS) i ClangFormat (za JavaScript).

Uvod

PicoBlaze (od latinskog *piccus*, rupica na igli, u prenesenom značenju *nešto maleno*, i engleskog *blaze*, plamen. Latinska riječ *piccus* možda je povezana s *picus*, djetlić, jer djetlić svojim kljunom buši drvo. Engleska riječ *blaze* dijeli isti korijen kao i latinski *flamma*. Od latinske riječi *flamma* dolazi, preko francuskog, danas daleko poznatija engleska riječ za plamen, *flame*. Hrvatska riječ *plamen* vjerojatno nije povezana s latinskim *flamma*, jer latinsko *f* dolazi od indoeuropskog *b^h* ili *d^h* i odgovara hrvatskom i engleskom *b* i *d*, nego latinska riječ *flamma* vjerojatno dijeli isti korijen kao hrvatska riječ *bijel*.) je malo računalo koji proizvodi tvrtka Xilinx. Koristi se u ugrađenim sustavima, te kao primjer jednostavnog računala na kolegiju *Arhitektura računala* na FERIT-u. Njegov je procesor dizajniran tako da se u cijelosti može implementirati programibilnim elektroničkim sklopovima (FPGA-ovima, ASIC-ima...), te je pisan u programskom jeziku VHDL (VHDL je kratica od *Very High Speed Integrated Circuits Hardware Description Language*, što znači *jezik za opis strojne opreme (hardware) od integriranih krugova jako velike brzine*. Među studentima elektrotehnike česta je šala, s kojom se i ja slažem, da je VHDL zapravo kratica od *Very Hard Difficult Language* – *vrlo čvrsto težak jezik*²). Takvi procesori, koji su namijenjeni da se sintetiziraju na FPGA-ovima ili ASIC-ima, zovu se mekani procesori (*soft processor*). PicoBlazeov procesor razlikuje se po mnogo čemu od procesora korištenih u stolnim računalima (koje proizvode tvrtke Intel i AMD) i procesora u mobilnim telefonima i tabletima (uglavnom ARM-ovi procesori), i potreban nam je simulator da bismo pokrenuli programe za njega na računalu na kojem programiramo. Dok programiramo za male uređaje, korisno je moći pokrenuti program na računalu na kojem programiramo radi testiranja ili traženja pogreške u programu. Simulator je program koji omogućuje računalu da se pretvara da je nekakvo drukčije računalo. Riječ *simulator* znači *onaj koji kopira ili imitira*, dolazi iz latinskog i prvi put se spominje u Ovidijevim Metamorfozama (11. poglavlje, 634. stih). Dolazi od *simulare* (pretvarati se), od *similis* (sličan). PicoBlaze vjerojatno ne može pokrenuti alate za programiranje (ako i može, oni bi bili jako spori i nepraktični za korištenje – daleko lošiji od alata za programiranje mobitela koji se pokreću na mobitelima), on bez korištenja pomoćnih uređaja može koristiti svega 0.25 KB memorije za spremanje podataka i 9 KB-a memorije za spremanje programa.

¹ <https://flatassembler.github.io/PicoBlaze/PicoBlaze.html>

² A, da sam upisao latinski (o čemu sam i razmišljao prije studija), sada bi mi čitanje Cicerona bilo čvrsto teško nedokučivo.



Slika 1: PicoBlaze računalo (fotografija koju je profesor Ivan Aleksi uključio u svoju prezentaciju)

Za usporedbu, simulator koji je autor napravio velik je 196 KB, a jedna disketa može sadržavati, ovisi kako je formatirana, do 2'000 KB podataka (obično se formatira u FAT12 format, jer je jedini razlog zašto se diskete danas koriste kompatibilnost s prastarim računalima koja jedino to i podržavaju, a FAT12 format dopušta da se koristi oko 1'400 KB). Za simuliranje PicoBlazea najčešće se koriste FIDEX, koji proizvodi tvrtka Fautronix, ili Xilinx ISE, koji, naravno, proizvodi tvrtka Xilinx. Postoje legalne besplatne verzije tih programa koji se mogu skinuti s interneta, i te besplatne verzije podržavaju vjerojatno sve što nekome treba, tako da cijena nije problem.

Eclipse

Eclipse (nazvan po grčkoj riječi za pomrčinu) je besplatan IDE (*integrated developing environment*, program namijenjen da se u njemu piše kod u programskom jeziku i koji olakšava rukovanje raznim alatima za programiranje) primarno namijenjen za pisanje programa u Javi. Prva verzija izdana je 2001. godine. Međutim, danas Eclipse pruža relativno dobru podršku i za JavaScript, te je velik dio koda ovog simulatora napisan u njemu. Prednost nad VIM-om je to što koristi TypeScript Service (TypeScript je programski jezik koji je dizajnirao Microsoft, on se prevodi u JavaScript i Microsoft tvrdi da olakšava automatsko traženje grešaka u programima) kako bi davao smislene sugestije za popunjavanje koda te što za naredbe iz JavaScriptine standardne biblioteke (i one relativno opskurne, koje često zaboraviš kako se koriste) daje kratku dokumentaciju. Nedostatak je što na mome računalu treba dugo da se Eclipse pokrene i što se ponekad zaglavljuje.

VIM

VIM (od *VI Improved*, poboljšani *VI*, jer je *VI*, *Visual Interface*, bio često korišteni tekstualni editor na računalima 1980-ih) je tekstualni editor koji se dobije uz gotovo sve verzije Linuxa i sličnih operativnih sustava (FreeBSD, MacOS...). Prva verzija je napravljena 1991. On je nešto kao Notepad na Windowsu, osim što pruža neke mogućnosti korisne za programiranje, recimo, sintaksno bojanje koda (za to koristi brze algoritme koji su uglavnom točni, ali nekada nisu, što pomalo živcira), nadopunjavanje djelomično napisane naredbe (drukčiji algoritam nego Eclipse, uglavnom daje lošije rezultate), brzi odlazak na određenu liniju (korisno kada compiler javi da je greška u određenoj liniji), prikaz linija koda, i tako dalje. Prednost nad Eclipseom je što se brzo otvara i što se na današnjim računalima nikad ne zaglavljuje, što je osobito važno kad želim napraviti neku izmjenu na brzinu (da vidim kako će internetski preglednik na nju reagirati). Dodatna prednost, iako to meni nije bilo važno, je što se može koristiti dok na računalu nije pokrenuto nikakvo grafičko sučelje, što je važno ukoliko radimo s udaljenog računala na serveru s kojim smo spojeni preko SSH-a (kada možemo koristiti samo tekstualno sučelje slično DOS-u). Naravno, mnoge su značajke VIM-a (uključujući i alatnu traku i izbornu traku, da ne moraš pamtiti kako razne opcije pokrenuti preko tipkovnice) tada nedostupne. Uz Ubuntu Linux dobije se smanjena verzija VIM-a koja ne podržava mnoge korisne stvari (recimo, sintaksno bojanje koda, da se različite vrste riječi u programskom jeziku oboje različitim bojom), a uz Oracle Linux dobije se zastarjela verzija (koja, recimo, krivo oboji višelinijske stringove u JavaScriptu). No, to se može riješiti tako da se s interneta skine izvorni kod novije verzije i da se kompilira.

GIMP

GIMP (GNU Image Manipulation Program) je program za uređivanje rasterske grafike koji se dobije uz većinu verzija Linuxa. Na Oracle Linux mora se zasebno instalirati, no radi uz samo manje probleme (recimo, ruši se ako ga pokušamo postaviti u *Single Window Mode*). GIMP je, tako reći, Linuxov Paint. Ustvari, podržava on i mnoge korisne stvari koje Paint (barem starije verzije, dugo nisam koristio Paint pa ne znam kako novije verzije) ne podržava, kao što su slojevi (layers) i prozirnost (transparency) dijelova slike. Ipak je znatno manje moćan (ali time i lakši za korištenje) od PhotoShopa. Prva verzija GIMP-a izdana je 1996. godine.

Inkscape

Inkscape je Linuxov program za uređivanje vektorske grafike. Prva verzija izdana je 2003. godine. Dok je GIMP primarno namijenjen za uređivanje slika u rasterskim formatima kao što su PNG, BMP i JPG, gdje se za (jako pojednostavljujem) svaki piksel nalazi informacija koje je boje, Inkscape služi za uređivanje slika u vektorskim formatima kao što je SVG, gdje se slika ne opisuje pojedinim pikselima nego oblicima kao što su crte određene debljine i boje, poligoni, pravokutnici, krugovi, elipse, Bezierove krivulje, i tako dalje. Donekle sličnu funkcionalnost ima Microsoft Office Publisher (no on ne podržava SVG format, koji meni treba, a nije ni besplatan) ili Corel Draw (on podržava SVG, ali nije besplatan, a i relativno je težak za korištenje).

Firefox

Firefox je internetski preglednik koji se dobije uz Oracle Linux i jedini je internetski preglednik koji radi prihvatljivo na Oracle Linuxu. Uz Oracle Linux još se dobije i Konqueror 4.14.8 (prastara verzija iz 2008. godine, doba Internet Explorera 7, i nema očitog načina da se instalira nova), koji je praktički beskoristan za današnji internet (rijetko koja se današnja internetska stranica prikazuje ispravno u njemu), i uz njega se ne dobivaju nikakvi alati za programiranje. Danas i najmanje informatički pismeni ljudi znaju što je to Firefox, Firefox i Chrome danas su najpopularniji internetski preglednici. Uz Firefox dobivaju se alati za programiranje: debugger za JavaScript, inspektor CSS-a, alati kojima se može aproksimirati kako će web-stranica izgledati na raznim mobitelima (naravno, kako je većina mobilnih internetskih preglednika bazirana na Chromu, a ne na Firefoxu, rezultati nisu pretjerano točni), inspektori AJAX-a i tako dalje. Alati za programiranje koji se dobiju uz internetske preglednike su primarno post-hoc, oni ne pomažu pri izradi web-aplikacije ili web-stranice, ali pomažu dijagnosticirati problem kad se dogodi. Postoje razni dodaci za IDE-jeve koji su namijenjeni za to da daju iskustvo pisanja programa u JavaScriptu slično kao pisanju programa u drugim popularnim programskim jezicima tako što povezuju Firefox ili Chrome i IDE, no njih je na Oracle Linuxu teško namjestiti.

Prettier

Prettier je formater za kôd, program koji se brine da kôd na programskom jeziku dobro izgleda što se tiče uvlaka, prelaska u nove redove, i slične stvari koje ne mijenjaju značenje koda. Jedan je od rijetkih takvih programa koji može formatirati JavaScript, HTML i CSS kad su miješani u istoj datoteci, kao što je datoteka `PicoBlaze.html`. Po meni, on ne daje baš lijepe rezultate, no takve tvrdnje već spadaju u subjektivni dio programiranja.

ClangFormat

ClangFormat je formater za kôd koji se dobije uz CLANG compiler za C, C++ i Objective-C. Može ga se koristiti s mnogim programskim jezicima, u stvari, uspio sam ga namjestiti da formatira i kôd pisan na mom programskom jeziku, AEC-u (iako ima nekih problema s mojim programskim jezikom, recimo, nema očitog načina da mu se objasni da je `:=` operator pridruživanja u mom programskom jeziku, a ne oznaka za label plus operator `=3`, no tokenizer za moj programski jezik ignorira razmake koje ClangFormat stavlja između `:` i `=`). Po meni, on daje lijepe rezultate za JavaScript. Mana mu je što ne može formatirati HTML i CSS niti JavaScript spremljen u HTML datoteci.

LGTM

3 Otvorio sam pitanje na forumu o izradi programskih jezika o tome:
<https://langdev.stackexchange.com/q/1695/330>

LGTM (*Looks Good To Me*) je besplatan internetski servis koji na svojim serverima na zahtijev pokreće razne statičke analize da analiziraju kôd pisan u raznim programskim jezicima pohranjen na GitHubu i sličnim servisima. Statički analizer je program koji pokušava naći greške u drugim programima, a da ih ne pokrene. To je različito od debuggera, programa koji pomaže dijagnosticirati grešku koja se dogodi kad se program pokrene pod njegovim nadzorom. Statički analizeri osobito su korisni za *popustljive* jezike kao što je JavaScript. U JavaScriptu, recimo, ako krivo napišemo ime varijable, compiler neće javiti grešku, nego će se program izvršavati bez upozorenja sve dok ne dođe do mjesta u programu gdje se nalazi takva pogreška (ako smo u strogom načinu rada, *strict mode*), ili čak i dalje (ako smo izvan strogo načina rada, compiler za JavaScript će na tom mjestu deklarirati novu varijablu s imenom jednakim tome krivo napisanom imenu). Isto tako, ako krivo napišemo ime neke funkcije, program će se vrtjeti sve dok ne dođe do te točke, i tek onda javiti pogrešku, tako da nam tako nešto pri brzinskom testiranju lako može promaknuti. U većini drugih programskih jezika takve vrste grešaka ne mogu promaknuti: compiler će javiti o takvim greškama prije no što se program uopće pokrene. Za JavaScript, ako to želimo, moramo koristiti statički analizer.

Mane današnjih simulatora PicoBlazea

Mnogi bi smatrali činjenicu da su danas najčešće korišteni simulatori PicoBlazea zatvorenog koda njihovom velikom manom. PicoBlaze mekani procesor (procesor koji se može u cijelosti implementirati FPGA-ovima) jest otvorenog koda. Međutim, on se može compilirati samo Xilinxovim compilerom za VHDL, koji je zatvorenog koda. Najnapredniji compiler za VHDL koji je otvorenog koda danas je, bez sumnja, GHDL, ali njegova kompatibilnost sa Xilinxovim compilerom je slaba. Zato danas najčešće korišteni simulatori PicoBlazea, koji ciljaju na to da ga simuliraju u VHDL-ovske detalje, sadrže elemente zatvorenog koda. Dakle, to su programi za koje je ilegalno provjeravati da nisu zlonamjerni. Upitno je mogu li nam ekonomski faktori i američki pravni sustav garantirati da Xilinxovi programi nisu Trojanski konji. Često se postavlja pitanje jesu li programi zatvorenog koda odraz prava na privatnost softwareskih firmi, ili su odraz nedostatka transparentnosti? Gdje je crta između privatnosti i nedostatka transparentnosti? Što ljudi pišu u privatnim pismima i privatnim elektroničkim pismima ne tiče se nikoga drugog. Ali programi koji se javno objavljuju ipak se tiču svih nas, zar ne? Isto tako, što se događa u klaonicama ili privatnim zatvorima (koji su česti u Australiji, i smatraju se iznimno nehumanima) nije samo stvar ljudi koji tamo rade, nego svih nas, zar ne? Stoga ne bi trebali postojati zakoni koji sprječavaju da se te stvari provjeravaju.

Po autoru ovog teksta, jedna od glavnih mana današnjih simulatora PicoBlazea je upravo to što oni ciljaju na simuliranje PicoBlazea (i druge mekane procesore) u VHDL-ovske detalje. Da bi to napravili, simulatori moraju biti programi od po stotine ili čak i tisuće MB-a. Ako pokušamo s interneta skinuti i na disk pohraniti na stotine MB-a ili nekoliko GB-a podataka, gotovo sigurno ćemo naletjeti na neke neočekivane probleme (pucanje internetske veze zbog kojeg moramo krenuti ispočetka, greške u datotečnom sustavu nastale naglim gašenjem računala koje ne bivaju očite dok ne pokušamo spremići neku ogromnu datoteku...).

Glavna mana današnjih simulatora PicoBlazea je to što ih se na računalu mora instalirati da bi funkcionirali. Nije ih moguće pokrenuti u internetskom pregledniku ili skinuti ZIP-arhivu i otpakirati je gdje želimo. To je osobit problem na Linuxu iz dva razloga. Prvo, instaliranje programa obično zauzima mjesta na SSD-u (gdje je spremljen Linux i sistemski direktorij gdje se obično spremaju instalirani programi, `/usr/bin`), a ne na HDD-u (na kojem obično ima daleko više slobodnog mjesta, i koji se ne troši kada na njega pišemo ili brišemo s njega). Jedan od načina da se to riješi je postavljanje virtualne mašine čiji se virtualni tvrdi disk nalazi na HDD-u, no to je dugotrajan i kompliciran posao, a i nezgodno je čekati da se pokrene još jedan Linux kad god nam treba neki program koji nam možda često treba. Drugo, programeri koji nemaju iskustva s radom na Linuxu obično pretpostavljaju da postoji samo jedan, nekakav apstraktni, Linux. Simulatori PicoBlazea obično su testirani na Red Hat Linuxu, i programeri vjerojatno pretpostavljaju da, ako tamo radi, radit će na drugim verzijama Linuxa. No to vrijedi samo za najjednostavnije programe pisane u C-u ili Assembleru, čak ne ni za *Hello World* program pisan u C++-u. Istina je da će program koji radi na Red Hat Linuxu vjerojatno raditi bez problema na Oracle Linuxu, CentOS-u i Scientific Linuxu, a možda i na Fedori. No, ako želite pokrenuti program za Red Hat Linux na Ubuntu Linuxu, Debianu ili Mint Linuxu (danas najčešće korištene verzije Linuxa), puno sreće s time. Vrijedi i obratno: programi za Debian rijetko kad se mogu jednostavno pokrenuti na Oracle Linuxu. Iako postoje programi za Linux koji izvrsno rade na mnogim verzijama Linuxa (Firefox, recimo, radi savršeno na Ubuntu Linuxu, a na Oracle Linuxu samo ima problema s prikazom MP4 videa), da bi se to postiglo trebaju programeri koji poznaju Linux u najveće detalje, a takvi su rijetki. Većinom se programi za Linux na mnogim verzijama Linuxa ne daju niti instalirati. I, zapravo, instalacija je nerijetko najveći problem. Pokušaj da se na Oracle Linux instalira Chrome pomoću RPM datoteke skinute s Googlea (namijenjene za Fedoru) dovodi do hrpe poruka o pogrešci, a, ipak, izvršna

datoteka Chromiuma s AppSpota funkcionira uz manje probleme. Programi otvorenog koda, kao što su VIM, mogu funkcionirati na mnogim verzijama Linuxa tako što se oslanjaju na compilere i srodne alate prisutne na Linuxu za instalaciju. No, to za napredne PicoBlaze simulatore, kojima je barem dio koda zatvoren, nije opcija. Nema jednostavnog rješenja za taj problem. Linux je otporniji na računalne viruse od Windowsa dijelom i upravo zato što ne postoji samo jedan Linux, nego mnogo verzija Linuxa koje međusobno nisu posve kompatibilne. Ako bismo napravili da programi za jednu vrstu Linuxa funkcioniraju na svim vrstama Linuxa, olakšali bismo posao nekim programerima, ali još bismo više olakšali posao kriminalcima koji rade računalne viruse. Kompatibilnost među računalima može se koristiti i za dobro i za zlo. Isto vrijedi i za biološke viruse: Ako nema genetske raznolikosti, oni se znatno lakše prenose. Neke su nekoć veoma popularne sorte banana izumrle zbog virusne infekcije, jer su sve jedinke te sorte imale više-manje iste gene za imunitet od virusa. Čim se dogodila mutacija da može jednu zaraziti, mogao je zaraziti sve istim načinom napadanja. Isto tako, šišmiši i ljudi imaju veoma sličan imunološki sustav, i zato su virusi koji mogu napasti i šišmiše i ljude relativno česti (tim više što šišmiši žive noću, kad nema sunca da ubije viruse). Psi i ljudi imaju donekle različit imunosni sustav, i zato su virusi koji mogu napasti i ljude i pse relativno rijetki. Virus koji mogu napasti i ptice i ljude vrlo su rijetki (pod svjetlosnim mikroskopom vidi se razlika između leukocita sisavaca i leukocita ptica), a virusi koji mogu napasti i ljude i gmazove ne postoje. Nekima bi se možda učinilo da su rješenja snažni antivirusni programi, međutim, bilo koji stvaran algoritam za detektiranje računalnih virusa krivo će detektirati neke dobroćudne programe kao viruse, a posljedice mogu biti jednako loše kao i posljedice samih virusa. Kao što ljudski imunosni sustav, postane li presnažan, može uzrokovati auto-imune bolesti i time biti kontraproduktivan, isto se događa s antivirusnim programima. Uostalom, rijetko se koji novi virus može detektirati zastarjelim algoritmima ugrađenima u antivirusne programe, kriminalci koji rade računalne viruse znaju kako ti algoritmi funkcioniraju i kako ih prevariti. Simulator PicoBlazea u JavaScriptu otvorenog je koda, to jest, kôd je dostupan javno na GitHubu (i, budući da je web-aplikacija, ne može učiniti ništa loše računalu ukoliko se pokrene u sigurnom internetskom pregledniku) i pod MIT-jevom je licencom, velik je svega 196KB, i ne zahtijeva nikakve instalacije.

Struktura simulatora za PicoBlaze u JavaScriptu

Simulator PicoBlazea u JavaScriptu nema back-end (kôd koji se vrti na serveru), već se u cijelosti vrti u internetskom pregledniku. Njegov kôd podijeljen je u 7 datoteka, ukupno 3'550 redaka (dakle, balansirao sam između pravila koje smo učili na kolegiju *Razvoj programske podrške po objektivno orijentiranim načelima*, da jedna datoteka treba biti najviše 100 redaka koda, i sugestije korisnika foruma *Philosophical Vegan* zvanog *TelepathyConspiracy*, da sav kôd stavim u jednu datoteku⁴, radio sam nešto između, kao i u compileru za svoj programski jezik):

1. `PicoBlaze.html` – sadrži HTML kôd i CSS kôd te JavaScript vezan za postavljanje izgleda web-aplikacije, sintaksno bojanje asemblerskog koda, postavljanje simulatorske niti, komunikaciju između tokenizera, parsera, pretprocesora i asemblera (u svijetu kompilera to se zove *driver*) te za dohvaćanje primjera asemblerskog koda s autorovoga GitHub profila. Naknadno je dodana mogućnost dodavanja točaka prekida (*breakpoints*), mogućnost skidanja heksadekadske datoteke koju proizvodi assembler (koja se uz pomoć alata za programiranje PicoBlazea veoma lako pretvori u binarni format koji treba PicoBlazeu) te zoran prikaz sedam-segmentnih pokaznika (*seven-segment display*) i prekidača (*switches*, ovdje reagiraju na pritisak miša) pomoću SVG-a generiranog u JavaScriptu. Skidanje heksadekadske datoteke radi se preko standardnih JavaScript klasa `HTMLAnchorElement` (svojstvo `download` i metoda za stvaranje eventa iz JavaScripta `click`), `Uint8Array`, `URL` i `Blob`, bilo je relativno komplicirano za napraviti (Jedan od nedostataka pravljenja web-aplikacija je to što su takve stvari, koje se lagano naprave u drugim okruženjima, na webu radi sigurnosti teške za napraviti.). Točke prekida rade se tako da se brojevi linija koda nalaze u zasebnim HTML elementima `<div>` koje generira JavaScript i ti elementi osim brojeva linija koda sadržavaju i ikone koje predstavljaju točke prekida koje su po defaultu nevidljive. Globalni objekt `machineCode`, osim heksadekadskih stringova koje predstavljaju naredbe u strojnom kodu, čuva i podatke o linijama asemblerskog koda iz kojih dolaze naredbe. Točke prekida dodao sam na prijedlog profesora Tomislava Matića. Datoteka `PicoBlaze.html` ima 1'320 redaka koda. CSS koji se koristi je relativno primitivan (recimo, nema medijskih upita), za pozicioniranje elemenata na

⁴ <https://philosophicalvegan.com/viewtopic.php?p=51387#p51387> *TelepathyConspiracy*, poznat na mnogim drugim internetskim forumima kao *PoliticalVegan*, misli da se pravilo „Jedna datoteka treba sadržavati najviše 100 redaka koda.” odnosi na davno vrijeme dok editori još nisu podržavali *folding*, a da danas sav kôd treba biti u jednoj datoteci i da trebamo koristiti *folding* da se snađemo u toj datoteci.

ekranu uglavnom se koristi JavaScript. Iskreno, ne da mi se učiti napredni CSS kad izgleda da mogu i bez toga. U CSS-u se koriste varijable, no dodan je i *fallback* za internetske preglednike koji ne podržavaju CSS-ove varijable (njih ne podržava, koliko je meni poznato, niti jedan internetski preglednik koji se može vrtjeti na Windows XP, zastarjelom, ali na zastarjelim računalima u obrazovnim ustanovama još uvijek popularnom operativnom sustavu). Velik dio HTML-a (sve tablice) također se generira u JavaScriptu, uglavnom pomoću JavaScriptinih višelinijjskih stringova i JavaScriptine naredbe `innerHTML`. Razni internetski preglednici rade probleme ako se tako pokušaju generirati SVG elementi (u JavaScriptu se generiraju SVG elementi koji predstavljaju sedam-segmentne pokaznike, prekidače i LED-ice), pa sam za svaki slučaj to radio JavaScriptinim naredbama `createElementNS`, `setAttribute` i `appendChild`. U datoteci `PicoBlaze.html` još se nalazi i kod za simulaciju UART-a, protokol pomoću kojega PicoBlaze može komunicirati s terminalima i simulatorima terminala, ali on je po defaultu isključen. Terminali su, naime, uređaji s tipkovnicom i ekranom pomoću kojih možemo komunicirati s programima na udaljenim računalima koji imaju CLI sučelje, to jest, sučelje slično tipičnim DOS-ovim programima. Naknadno sam datoteku `PicoBlaze.html` razdjelio u datoteke `PicoBlaze.html` (koja sada sadrži samo HTML te samo onaj JavaScript koji je potreban da se sakrije moja e-mail adresa od spambotova), `styles.css` (koja sadrži CSS), `headerScript.js` (koja sadrži deklaracije potrebne drugim datotekama te skriptu za ponovno postavljanje layouta ako prozor promijeni veličinu) i `footerScript.js` (koja sadrži funkcije za počinjanje i završavanje simulacije te JavaScript koji stvara SVG-ove). A 22. kolovoza 2023. godine izdvojio sam nizove s mnemonikama i pretprocesorskim direktivama iz datoteke `headerScript.js` u zasebnu datoteku koju sam nazvao `list_of_directives.js`, da bih si olakšao pravljenje automatskih testova parsera. Pri pisanju datoteke `PicoBlaze.html` naletio sam na, koliko se sjećam, tri ozbiljnija problema. Prvi je bio to što sam u JavaScriptu pokušao napraviti da moj program sintaksno oboji (*syntax highlight*, u biti, da različite vrste riječi u programskom jeziku budu obojane različitom bojom) asemblerski program kako ga korisnik ukucava. Pokušao sam mnogo stvari, ali nije išlo. Na kraju sam odlučio da ne pokušavam bojati program dok ga korisnik ukucava, nego da dodam gumb „*Highlight Assembly*” koji korisnik stisne kad želi da mu se sintaksno oboji program, što je relativno lagano za isprogramirati. Otvorio sam pitanje o tom problemu na forumu o izradi programskih jezika⁵ (jer pretpostavljam da su mnogi koji su izradili programski jezik napravili i editor za njega u JavaScriptu, pa su možda naletjeli na isti problem i uspjeli ga riješiti, pa će mi moći pomoći), pa su se moderatori žalili da mi je pitanje *off-topic*, i na forumu o programiranju općenito⁶, na kojem skoro tjedan dana nisam dobio nikakav odgovor. Odgovor koji sam dobio na forumu o programiranju općenito predlaže mi da iskoristim to što je asemblerski kod u monospace fontu, pa da dodam još jedan transparentan `<pre>` element ispred tog `<pre>` elementa u koji se ukucava asemblerski kod, te da u tom transparentnom `<pre>` elementu prikazujem highlightirani asemblerski kod. Ne sviđa mi se baš ta ideja. Ako to napravim, sigurno ću slomiti kompatibilnost s WebPositiveom (jer on ignorira CSS naredbe da prikazuje asemblerski kod u monospace fontu), ako ne i s Firefoxom 52 (u kojem monospace fontovi, kako se meni čini, nisu zapravo monospace). Druga dva problema na koja sam naletio pišući `PicoBlaze.html` bila su da su različiti internetski preglednici različito interpretirali CSS koji sam napisao. Prvi od njih bilo je da su se *tooltipsi* koji se pojave kada korisnik prijeđe mišem preko gumba sa slikom (*play*, *pause*, *stop*, *single step* i *fast forward*) dobro prikazivali u Firefoxu, ali su u Chromeu bili pomaknuti znatno u lijevo. Netko mi je na internetskom forumu predložio da to jednostavno mogu riješiti tako da u CSS pravila za gumbe dodam pravilo „`position: relative`”⁷. Ne razumijem kako to rješenje funkcionira, ali izgleda da funkcionira. A drugi problem bio je da WebPositive, internetski preglednik koji se dobije uz operativni sustav Haiku, blisko srodan Safariju (i WebPositive i Safari baziraju se na WebKitu), nije stavljao onu svijetlosivu pozadinu na gumbe, pa su slova na gumbima (jer su onda to bila crna slova na ljubičastoj pozadini) u njemu bila nečitka. No, kad sam u CSS dodao „`background: #cccccc;`”, radilo je kako treba. WebPositive još uvijek renderira neke stvari u mom PicoBlaze Simulatoru pogrešno, recimo, on asemblerske programe ne renderira u monospace fontu, i nema očitog načina da mu se zada da to napravi. Isto tako, WebPositive krivo renderira SVG gradijente u ikonama *play*, *pause*, *fast forward* i *single step*. I to nije samo u mom simulatoru PicoBlazea, nego i

5 <https://languagedesign.stackexchange.com/q/1681/330>

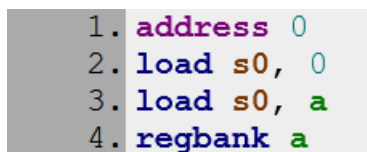
6 <https://stackoverflow.com/q/76566400/8902065>

7 <https://stackoverflow.com/a/64995638/8902065>

drugdje na mojoj web-stranici, recimo, u WebPositiveu su strelice na dnu mog SVG PacMana⁸ nevidljive dok se na njih ne prijeđe mišem. No, te stvari ne spriječavaju da se moj PicoBlaze Simulator koristi u WebPositiveu. Drugi internetski preglednik za operativni sustav Haiku, Otter (bliski srodnik Operi), bolje renderira web-stranice nego što to radi WebPositive, ali Otter ne podržava dovoljno JavaScripta da se u njemu pokrene moj PicoBlaze Simulator (iako u Otteru, na primjer, moj SVG PacMan radi bez problema). Nisam isprobao svoj PicoBlaze Simulator u Safariju i Operi, no ne očekujem tamo velike probleme. U svakom slučaju, mislim da svatko tko želi koristiti moj PicoBlaze Simulator lako može na svoje računalo instalirati neki internetski preglednik u kojem se on može pokrenuti. U svim preglednicima na mobilnom operacijskom sustavu Androidu u kojima sam isprobao svoj PicoBlaze Simulator, osim u Dolphinu (Dolphin se bazira na WebKitu, kao i Safari i WebPositive), *layout* se pokida kad se prijeđe u *landscape* (pejzažni) način rada (kad mobitel okrenemo horizontalno umjesto vertikalno). Mislim da razumijem zašto, ali ne znam kako to popraviti. Pretpostavljam da bi netko tko zna napredni CSS to mogao popraviti. Otvorio sam pitanje o tom problemu na internetskom forumu⁹, ali nisam dobio nikakav smisleni odgovor. S obzirom na to da se taj problem ne događa u Dolphinu, a Dolphin je bliski srodnik Safariju (oboje se baziraju na WebKitu), pretpostavljam da se taj problem ne događa ni u Safariju na iPhoneu, iako nisam isprobao. U Firefoxu 52 ako se pokrene na Windowsima XP (ali, začudo, ne i ako se ta ista verzija Firefoxa pokrene na Solarisu) `<div>` element s brojevima linija i `<pre>` element u koji se ukucava asemblerski kôd imaju različite prorede, tako da linije koda i oznake brojeva linija nisu *alignirane*. Nisam to uspio riješiti, i, ustvari, nemam osjećaj ni da bih to trebao riješiti. Mislim da to nije moj bug, niti da je bug u Firefoxu, nego da je to bug u operativnom sustavu Windows XP. Slično tako, u nekim starijim verzijama Microsoft Edgea, baziranim na EdgeHTML-u (moderne verzije Microsoft Edgea baziraju se na Blinku, kao i Chrome), `<div>` element s brojevima linija ne *scrollira* se zajedno s onim `<pre>` elementom u koji se ukucava asemblerski program. Nisam ni to uspio riješiti. Tako da se u Firefoxu 52 pokrenutom na Windows XP-u i u nekim starijim verzijama Microsoft Edgea, u kojima moj simulator PicoBlazea inače radi, ne mogu koristiti točke prekida (*breakpoints*). U Firefoxu 52 na Windows XP još se i mogu tako da, recimo, asemblerski program koji želimo debugirati kopiramo u Notepad i u Notepadu u statusnoj traci vidimo koji je redni broj linije u kojoj mislimo da je problem, pa onda taj broj kliknemo u Firefoxu 52, ali takav *work-around*, koliko vidim, ne postoji u Microsoft Edgeu. I još sam naletio na problem da sam, kad sam birao boje za 7-segmentne pokaznike, pazio da izgleda slično kao na pravom PicoBlazeu. I na ekranu na laptopu uistinu je tako izgledalo. Međutim, kad sam to isprobao na svom mobitelu (na kojem je, izgleda, zasićenje boja znatno slabije), tamo su brojevi na 7-segmentnim pokaznicima bili nečitki, nijansa crvene koje sam odabrao za upaljene segmentne jedva da se razlikovala od nijanse tamnosive koja je prikazivala ugašene segmente. To sam riješio tako da sam nijansu crvene znatno posvijetlio. Nije to bilo prvi put da sam se tako opekao igrajući se s bojama, što izgleda lijepo na jednom ekranu često je na drugom ekranu nečitko. Kad sam nakon tri godine pisao primjer „*Regbanks-Flags Test*“, naletio sam na još jedan problem sa sintaksnim bojanjem asemblerskog koda. Naime, on asemblerski kod:

```
address 0
load s0, 0
load s0, a
regbank a
```

oboji ovako:



Slika 2: Neispravno sintaksnno bojanje koda

Što je netočno. Token `a` u naredbi „`load s0, a`“ heksadekadski je konstanta (`a` je heksadekadski 10), i treba biti obojena isto kao token `0` u naredbi „`load s0, 0`“, a ne kao u „`regbank a`“, gdje je `a` naziv regbanke. S obzirom na to kako je moj sintakсни bojač koda strukturiran, nema jednostavnog rješenja za taj problem. U IDE-u za x86 assembler FlatAssembler takve se

⁸ <https://flatassembler.github.io/pacman.html>

⁹ <https://atheistforums.org/thread-61911-post-2009508.html#pid2009508>

diskrepancije ne mogu dogoditi, jer on za sintaksno bojanje koda koristi isti algoritam koji koristi interno za parsiranje koda. Ali ja tako ne mogu raditi ni za svoj programski jezik ni za PicoBlaze assembler, jer za oba ta jezika tokenizer briše komentare, a ne mogu si dopustiti da pri sintaksnom bojanju koda obrišem komentare. Za sada postoji jednostavan *work-around*: Kada su tokeni `a`, `b` i `c` heksadekadske konstante, možemo umjesto njih pisati `0a`, `0b` i `0c`, i bit će obojane točno. Korisnik Discorda *ahnfelt*, autor programskog jezika Firefly, kaže mi da je on naletio na sličan problem pri sintaksnom bojanju svog programskog jezika. Predlaže mi da u sintaksnom highlighteru deklariram globalnu varijablu `lastHighlightedMnemonic`, da u nju spremam mnemonike prije nego što ih highlightiram, te da tokene `a` i `b` highlightiram CSS klasom `flag` samo ako je `lastHighlightedMnemonic` jednak `regbank`, a `c` samo ako je `lastHighlightedMnemonic` jednak `jump`, `return` ili `call`. No, primijetite da sintaksno bojanje koda ni tada ne bi bilo točno. U naredbi „`jump c`”, token `c` je heksadekadska konstanta (naredba znači „*Skoči na 13. naredbu u EPROM-u.*”, jer je `c` heksadekadski broj 12, a naredbe se broje od nule), a u naredbi „`jump c, abort`” token `c` je zastavica *carry* (naredba znači: „*Ako je zastavica carry postavljena u jedinicu, skoči na label koji se zove abort.*”). Da bismo znali je li `c` u PicoBlazeovom asemblerskom kodu heksadekadska konstanta ili zastavica, ne moramo samo znati prethodni token, nego i idući. Takve stvari ne smetaju parseru, ali su noćna mora za sintaksno bojanje koda. Kad sam, nekoliko tjedana nakon što sam napisao „*Regbanks-Flags Test*”, pisao primjer „*Preprocessor Test*”, uočio sam i problem da sintaksni highlighter nakon znakova veće (`>`) i manje (`<`) umeće točka-zarez (`;`), čineći programe koji koriste te operatore sintaksno netočnima. Nisam točnije dijagnosticirao problem, samo sam o tome dodao upozorenje na početak programa „*Preprocessor Test*”. Naknadno sam još i dodao da *syntax highlighter* odbija *highlightirati* programe koji sadrže znakove `<`, `>` i `&`, s porukom o pogrešci: „*Sorry about that, but syntax highlighting of the programs containing '<', '>' and '&' is not supported yet.*”, te sam tražio pomoć s time na nekoliko internetskih foruma^{10 11 12}, ali tu pomoć za sada nisam dobio.

Dok sam pisao potprogram `PicoBlaze.html`, bilo je i nekih problema za koje sam bio siguran da ću naletjeti na njih, a zapravo nisam naletio. Recimo, bio sam uvjeren da će TOR Browser odbijati *fetchati* PicoBlazeove asemblerske programe s mog GitHub profila (s domene `raw.githubusercontent.com`) iz JavaScripta koji se nalazi na mojoj web-stranici (na domeni `flatassembler.github.io`), da je to jedan od načina na koji TOR Browser štiti od *cross-site scripting* napada. Međutim, kada sam to *fetchanje* primjera asemblerskih programa isprobao u TOR Browseru, radilo je. Otvorio sam *thread* na forumu o tome¹³. U svakom slučaju, TOR Browser ne pruža zaštitu koju sam mislio da pruža. Mislim da je sigurnost koju pružaju internetski preglednici dosta nekonzistentna. Izgleda, recimo, da je dohvaćanje JSON-a s primjerima s nečijeg GitHub profila (istina, internetski preglednik ne može znati da je to moj GitHub profil i da ja imam kontrolu nad tim JSON-om) toliko rizično da će svaki moderni internetski preglednik odbiti to napraviti (morao sam zbog tog taj JSON kopirati sa svog GitHub profila na svoju web-stranicu). Ali dohvaćati PicoBlaze asemblerske programe s nečijeg GitHub profila i nesanitizirane ih postavljati kao `innerHTML` toliko je sigurno da će to dopustiti čak i internetski preglednici koji su napravljeni da budu iznimno sigurni, kao što je TOR Browser. Zar nije dohvaćati nešto što ćeš nesanitizirano postaviti kao `innerHTML` iz nečega što se percipira kao nepouzdan izvor zapravo još opasnije nego dohvaćati JSON iz tog izvora? Ja razumijem zabraniti dohvaćati JavaScript koji nešto naređuje internetskom pregledniku iz nepouzdanih izvora, ali JSON nije skripta koja nešto naređuje. Ne vidim kako bi netko mogao napraviti *cross-site scripting* napad izmjenjivajući nečiji JSON, ali vidim kako bi to netko mogao napraviti izmjenjivajući nešto što se postavlja kao `innerHTML` (recimo, da u to umetne „``”). Valjda ta odluka potječe iz vremena kada se JSON parsirao naredbom `eval`, pa si uistinu mogao napraviti *cross-site scripting* napad tako

10 https://www.reddit.com/r/PicoBlaze/comments/159chq6/problems_with_syntax_highlighting_picoblaze/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

11 <https://www.forum.hr/showpost.php?p=99274899>

12 <https://philosophicalvegan.com/viewtopic.php?p=51800#p51800>

13 https://www.reddit.com/r/TOR/comments/jv3dln/tor_browser_appears_to_allow_crosssite_scripting/?utm_source=share&utm_medium=web2x&context=3

da umetneš neispravan JSON (kakav naredba `JSON.parse`, koja se danas koristi za parsiranje JSON-a, ne bi prihvatila, ali `eval` bi). Otvorio sam pitanje na forumu o tome¹⁴.

2. `TreeNode.js` – sadrži JavaScript klasu pod nazivom `TreeNode`, koja sadrži metode vezane za evaluaciju parsiranih aritmetičkih izraza, metodu za ispis LISP-ovih izraza radi debugiranja parsera, te metode za pretragu struktura koje radi pretprocesor. Ta datoteka ima 100 redaka koda.
3. `assembler.js` – radi semantičku provjeru asemblerskog koda (recimo, je li prvi argument naredbe `load` uistinu registar) pomoću strukture koje radi parser te spaja strukturu koju radi parser i strukture koje radi pretprocesor u strojni kod u heksadekadskom obliku. Također radi neke sintaksne provjere koje parser ne radi, recimo, nalazi li se između dva argumenta naredbe `load` zarez. Ima 1'050 redaka koda. Pretvoriti strojni kod u heksadekadskom obliku u binarni oblik (kakav razumije PicoBlaze) nije lagano u JavaScriptu, jer najmanja jedinica memorije koja se u JavaScriptu može adresirati jest byte, 8 bitova, a svaka naredba u strojnom kodu PicoBlazea je 18 bitova, što nije cijeli broj byteova. Ali je zato strojni kod u heksadekadskom obliku iznimno lagano pretvoriti u binarni kod kakav razumije PicoBlaze pomoću Xilinxovih alata. Za razliku od ostalih potprograma, `assembler.js` i `simulator.js` svoje rezultate ne vraćaju kao povratnu vrijednost funkcije, nego ih pišu u globalne varijable. Potprogram `assembler.js` piše svoje rezultate u globalni objekt `machineCode`, odakle ih potprogram `simulator.js` čita. Riječ *assembler* znači *sastavljač*, dolazi, preko francuskog, od latinskog *assimulare* (sastaviti), od *ad* (na) i *simul* (zajedno).
4. `parser.js` – Parser (od latinskog *pars, dio*) je dio kompilera (u ovom slučaju, kompilera za asemblerski jezik) koji drugim dijelovima kompilera kaže koja je riječ u programskom jeziku gramatički povezana s kojom drugom riječi. To radi tako što radi strukturu zvanu AST, *abstract syntax tree*, apstraktno sintaksno stablo. Kao objašnjenje zašto je to potrebno, uzmite u obzir sljedeću rečenicu iz Cezarovog *De Bello Gallico* (koja je bila na županijskom natjecanju iz latinskog jezika 2016. godine, 6. svitak, 24. poglavlje): *Ea, quae fertilissima totius Germaniae sunt, loca Graecis aliquibus nota fama esse loquuntur*. S kojom je riječi povezana riječ *nota* (poznata)? Ako znate samo malo latinskog, vjerojatno biste pomislili da je riječ *nota* gramatički povezana s *fama* (slava, glasina), da je to pleonazam i da riječ *nota* treba zanemariti. No, to je krivo. Riječ *nota* povezana je s riječju *loca* (mjesto, lokacije). Riječ *fama* je u ablativu jednine (ablativ je latinski padež koji odgovara hrvatskom lokativu, instrumentalu te genitivu u značenju *iz koga* ili *iz čega*), a igrom slučaja na latinskom jeziku akuzativ množine u drugoj deklinaciji u srednjem rodu i ablativ jednine prve deklinacije imaju isti nastavak (to jest, isti su u pisanom latinskom ako ne označavamo naglaske, inače je *a* u nastavku u *fama* dugo, a u *nota* kratko). Rečenica znači: *Kažu (loquuntur) da su (esse) ona (ea) mjesta, koja (quae) su (sunt) najplodnija (fertilissima) u cijeloj (totius) Germaniji, nekim (aliquibus) Grcima (Graecis) glasinom poznata*. Pridjev *fertilis* (*plodan*) ima nepravilan superlativ *fertilissimus* umjesto *fertilimus*, ne znam ima li još takvih pridjeva u latinskom jeziku¹⁵. Imenica *locus* (mjesto) obično je muškog roda i množina joj je *loci*, ali, kad označava neku površinu (kao u ovom slučaju), onda je srednjeg roda i množina joj je *loca*. Glagol *biti* je jednom u prezentu (*sunt*), a jednom u infinitivu (*esse*), zbog jednog nevažnog detalja iz latinske sintakse koji se zove *akuzativ s infinitivom*. Glagol *loquuntur* je takozvani deponentni glagol, morfološki je pasivan, a semantički aktivan (zato je nastavak *-untur*, a ne *-unt*). Pridjev *totus* (cijeli) ima nepravilni genitiv jednine na *-ius* (*totius*), kao još nekolicina latinskih pridjeva. Dobar su posao napravili oni koji su sastavljali taj test da nađu rečenicu s toliko mnogo nepravilnih riječi, i koju, kako kaže moj profesor, nitko nije točno preveo! Takve rečenice, u kojima nije na prvi pogled očito koja je riječ gramatički povezana s kojom drugom riječi, postoje i u programskim jezicima, i u takvim bi rečenicima parser spojio riječi *nota* i *loca* da budu djeca jednog čvora sintaksnog stabla. Ta mi je rečenica ostala u sjećanju jer mi je profesor pričao da je ispravljao test neke učenice koja je tu rečenicu krivo prevela nešto kao *Najplodnija Njemica...*, no to bi se već teško dalo objasniti kao rezultat pogrešnog parsiranja. Ta su se najplodnija mjesta u Germaniji nalazila oko nekakve šume. Kasnije je u tom tekstu bilo *Ea (tamo = u toj šumi; ea s kratkim a je zamjenica ona, a ea s dugim a je prilog tamo) nascuntur (rađaju se) alces (sjeverni jeleni)...*, a ona je to navodno prevela s *Ona rađa sjeverne jelene...*, a to opet nije stvar krivog parsiranja kad po morfologiji vidimo da je *nascuntur* pasiv i vidimo da je u množini (*rađa* bi bilo *nascit*, a ne *nascuntur*). Parser za asemblerski jezik bio je mnogo lakši za napisati nego parser za AEC, moj programski jezik. Parser za moj programski jezik¹⁶ dugačak je 950 redaka, dok datoteka `parser.js` sadrži 125 redaka. Zapravo, jedino što je nužno

14 <https://security.stackexchange.com/q/270915/249645>

15 Otvorio sam pitanje o tome na forumu: <https://latin.stackexchange.com/q/21069/8533>

parsirati u assembleru za PicoBlaze jesu aritmetički izrazi. Algoritam napisan u datoteci `parser.js` ide ovako:

1. Pronađi parove otvorenih i zatvorenih zagrada u nizu koji ti je dao tokenizer. Parovi otvorenih i zatvorenih zagrada nalaze se, naime, u aritmetičkim izrazima te kao oznaka da je ono što se nalazi u registru pokazivač. Kada nađeš neki par zagrada, prebaci ono između zagrada u novi niz, obriši to iz originalnog niza, i pokreni rekurziju s novim nizom kao argumentom. Ako zagrade nisu dobro zatvorene, javi poruku o pogrešci. U parseru za svoj programski jezik zadao sam da se i zagrade obrišu iz sintaksnog stabla. U asemblerskom jeziku za PicoBlaze to ne bi imalo smisla, budući da zagrade imaju značenje da je u registru pokazivač, pa bih si time samo zakomplicirao assembler. Potprogram u `parser.js` zato za svaki par zagrada umeće čvor s tekstom `()`, i njegova su djeca (polje `children` iz klase `TreeNode`) niz koji vrati rekurzija.
2. Prolazi kroz niz koji ti je dao tokenizer i za svaku riječ provjeri nalazi li se na popisu mnemonika (tako se tradicionalno zovu glagoli u asemblerskom jeziku¹⁷) ili pretprocesorskih direktiva. Ti se popisi nalaze u datoteci `PicoBlaze.html`. Ako se riječ na koju si upravo naišao nalazi jednom od tih popisa, a nije da je riječ jednaka `enable` ili `disable` i da je dužina niza jednaka jedinici (jer `enable` i `disable`, osim što mogu biti glagoli, mogu biti i, recimo to tako, prilozi glagola `return`), premjesti sve između tog glagola i znaka za novi red (isključivo) u novi niz, pokreni rekurziju i proglasi ono što rekurzija vrati djecom čvora u kojem je taj glagol. To funkcionira zato što svaka rečenica u asemblerskom jeziku počinje s glagolom te, osim u aritmetičkim izrazima, ne postoji lingvistička rekurzija, to jest, u asemblerskom jeziku ne postoje složene rečenice. Kao zanimljivost, neki lingvisti (ustvari, danas možda samo Daniel Everett, a većina se lingvista slaže da Daniel Everettov opis pirahanske gramatike proturječi Chomskyjevoj univerzalnoj gramatici i zbog toga misle da je netočan) tvrde da je pirahanski jezik, slabo dokumentirani jezik iz Brazila, takav (da u njemu ne postoje složene rečenice).
3. Parsiraj aritmetičke izraze. Prvo se baktaj s unarnim operatorima, njih se detektira kao tokeni `+` (plus) i `-` (minus) ispred kojih se ne nalazi ništa (prvi token u nizu), ili se ispred njih nalaze tokeni `,` (zarez), `(` (otvorena zagrada) ili token koji sadrži znak za novi red. Zatim konstruiraj lambda-funkciju `parseBinaryOperators` koja prima niz operatora te prolazi originalni niz u potrazi za njima, i, kada ih nađe, njihove susjedne tokene proglašava njihovom djecom i briše iz niza. Ta se lambda-funkcija prvo poziva za niz samo s operatorom potenciranja, `^`, jer on ima najveći prioritet, zatim se poziva za operatore množenja i dijeljenja, `*` i `/`, te konačno za zbrajanje i oduzimanje, `+` i `-`. Parser za moj programski jezik mora paziti na razliku između lijevo-asocijativnih operatora i desno-asocijativnih operatora (kao što su ternarni uvjetni operator `?:`, operator pridruživanja `=` i skraćena pridruživanja `+=`, `-=`, `*=` i `/=`). No, budući da su svi aritmetički operatori lijevo-asocijativni, to u parseru za asemblerski jezik nije potrebno.

Na kraju bismo trebali dobiti asemblerski kod u obliku LISP-ovog S-izraza, koje JavaScript u datoteci `PicoBlaze.html` za potrebe traženja grešaka u assembleru ispisuje na preglednikovu konzolu JavaScriptinom naredbom `console.log`. Profesor Ivan Aleksi predlagao mi je da ne ugradim podršku za aritmetičke izraze u svoj assembler i da ni ne parsiram asemblerski kod nego da ga pretvorim u dvodimenzionalno polje stringova, gdje svaki redak iz asemblerskog koda predstavlja jedno jednodimenzionalno polje u tom dvodimenzionalnom polju, da prvi string u tom

16 <https://raw.githubusercontent.com/FlatAssembler/AECforWebAssembly/master/parser.cpp>

17 Navodno se tako zovu jer ih je lakše zapamtiti nego nizove nula i jedinica u strojnom jeziku, od starogrčkog *μνημονικός* (*mnemonikos*) što znači *pamćenje*. Morate se naviknuti da je informatika prepuna besmislenih imena. Uzmite u obzir i da je ime *digitalna elektronika* poprilično besmisleno, ono se prevodi kao *prst-jantar*. Nekome bi u antici, tko zna latinski i grčki, to ime vjerojatno bilo smiješno. Imam jednu anegdotu iz svog studentskog života o tome: Došla je knjižničarka na početku odmora u predavaonicu informirati profesora Gorana Martinovića da će biti nekakav simpozij gdje bi trebali doći profesori koji se bave humanističkim predmetima, a da misli da se to tiče profesora Martinovića jer on predaje kolegij zvan *Dizajn programske podrške*. A profesor Martinović joj odgovori: *Ah, kolegice, pa nije Vam to nikakav dizajn, to se samo tako zove*. Knjižničarka ga je, naravno, onda začuđeno pogledala. Ili, kako je moderator Latin Language StackExchangea zvan *cmw* komentirao na naziv „*objektivno orijentirano programiranje*”: „*Nije ga smišljao neki filolog, to je očito.*” (https://latin.stackexchange.com/questions/20811/how-would-you-say-object-oriented-programming-in-latin#comment42987_20814). Ne slažete se s njime? Probajte nekome tko nije programer objasniti etimologiju tog naziva, mislim da će vam postati jasno o čemu priča.

jednodimenzionalnom polju bude potencijalni naziv labela ili prazan string, da drugi string bude glagol, i tako dalje. Ja mislim da je raditi na taj način još kompliciranije.

Tri godine kasnije, na početak potprograma `parser.js` dodao sam dio za parsiranje `if`-grananja, `if-else`-grananja i `while`-petlji, algoritmom u biti identičnim onim što ga koristim u compileru za svoj programski jezik, uz jedinu bitnu razliku da u `parser.js` ne podržavam `elseif` naredbu. Zapravo, isprva sam mu zadao da izvrši dvije `for`-petlje, jednu koja parsira `if`-grananja, a drugu, poslije nje, koja parsira `while`-petlje. Također sam mu bio zadao da prije no što parsira `if`-grananje ili `while`-petlju, provjeri da ga nije roditeljska rekurzija već isparsirala, tako da provjeri da li je polje `children` u čvoru s tekstom „*while*” ili „*if*” prazno, te da parsira samo ako je prazno. Kasnije sam to izmijenio da se parsiranje i `if` i `while` naredbi vrši u jednoj `for`-petlji, te sam izbacio provjere jesu li te naredbe već parsirane (što mislim da se ne može dogoditi ako se sve to radi u jednoj `for`-petlji). Time sam malo pojednostavio kod, a funkcionalnost bi trebala ostati ista. Tako da `parser.js` sada ima 247 redaka.

Za C++ (ne znam kako je za JavaScript) postoje mnogi frameworksi koji su namijenjeni tome da olakšaju pisanje parsera, među njima su najpoznatiji YACC i BISON. Ja ih za pisanje parsera za svoj programski jezik nisam koristio, jer mi se nije dalo učiti ih. Zbog toga me je kritizirao korisnik Reddita *Fofeu*¹⁸, koji tvrdi da je doktor informatike koji specijalizira teoriju compilera. On kaže da, osim što frameworksi za pisanje parsera skraćuju kôd parsera 5 ili 10 puta, također odbijaju parsirati nekonzistentne gramatike. A, po njemu, trivijalno je nenamjerno smisliti nekonzistentnu gramatiku za programski jezik. I kaže da mu se ne da proučiti gramatiku koju sam smislio da vidi je li konzistentna, jer je vjerojatnost da je amater poput mene uspio smisliti konzistentnu gramatiku koja je toliko komplicirana (950 redaka u čistom C++-u) zanemariva.

Da sam PicoBlaze asemblerski kôd išao parsirati pomoću alata kao što je BISON (recimo da postoji BISON kompatibilan s JavaScriptom), kako bih mu mogao objasniti onaj problem s `enable` i `disable`, da su te riječi nekada glagoli, a nekada prilozi, i da ih treba različito parsirati ovisno o tome? Otvorio sam to pitanje na forumu o izradi programskih jezika¹⁹.

5. `preprocessor.js` – Prima strukturu koju pravi parser i određuje adrese labela (labelsu su oznake na koje je moguće skočiti naredbom `goto`, ili, kako se u asemblerskom jeziku zove ta naredba, `jump`). To je znatno lakše napraviti za PicoBlaze nego za Intelove i AMD-ove (x86) procesore, jer su za PicoBlaze sve naredbe u strojnom jeziku jednake dužine (18 bitova), pa možemo svaki puta kada nađemo na mnemoniku u AST-u povećati trenutnu adresu za jedan. Za x86, taj algoritam ne bi bio točan, jer, recimo, asemblerska naredba `int 0x3` (u doba DOS-a služila za pozivanje debuggera) ima 8 bitova (u heksadekadskom formatu je `cc`), dok `int 0x20` (u doba prvih verzija DOS-a služila za zatvaranje programa) ima 16 bitova (u heksadekadskom formatu je `20cd`), a `mov rax, [3*rbx+7]` ima 40 bitova (heksadekadski `8b485b440007`). Potprogram `preprocessor.js` također izvršava i prema rezultate direktiva `constant` i `namereg` (za preimenovanje registara u smislene nazive) u `Map` (klasa čiji objekti sadržavaju parove ključ-vrijednost, dostupna u standardnoj biblioteci JavaScripta od vremena Internet Explorera 11). Primijetite da se pretprocesor u kontekstu asemblera jako razlikuje od pretprocesora u kontekstu compilera. U compilerima, pretprocesor se pokreće još prije tokenizera. U assemblerima, pretprocesor se pokreće nakon parsera, ali prije jezgre asemblera. U mnogim je assemblerima pretprocesor Turing-potpun (Turing-complete), što je rijedak slučaj u višim programskim jezicima (koliko znam, to još vrijedi jedino za PERL). Zapravo, to se u višim programskim jezicima smatra lošim jer programske jezike ne treba moći parsirati samo compiler za taj programski jezik, nego i drugi alati za programiranje. Jedna od čestih kritika PERL-a je upravo *Only PERL can parse PERL*.. Potprogram `preprocessor.js` ima 140 redaka.

Tri godine kasnije, dodao sam FlatAssemblerovu naredbu `display` u pretprocesor svog asemblera za PicoBlaze. Ta naredba za vrijeme asembliranja ispisuje stringove ili konstante na terminal, da možemo provjeriti jesu li konstante dobro izračunate. Također sam dodao `if`-grananja, `if-else`-grananja i `while`-petlje, slične kao što postoje u FlatAssembleru. Glavna razlika između tih stvari u FlatAssembleru i u mom assembleru za PicoBlaze je u što se u `if`-grananjima i `while`-petljama u mom PicoBlaze assembleru ne mogu nalaziti mnemonike ili naredba `address`, jer ne znam kako

18 https://www.reddit.com/r/ProgrammingLanguages/comments/iehbmj/comment/g2kevm/?utm_source=share&utm_medium=web2x&context=3

19 <https://languagedesign.stackexchange.com/q/1679/330>

bih, da to dopustim, izračunavao adrese labela. U FlatAssembleru, recimo, mogu postojati programi kao što je ovaj (ne pazim na sintaksu, ali nadam se da će uspjeti shvatiti o čemu se radi):

```
address 0
firstLabel:
if firstLabel=secondLabel
    load s0,s0
endif
secondLabel:
```

Na tako nešto, FlatAssembler upada u beskonačnu petlju, jer čim izračunaš adresu labelu `secondLabel`, ona se promijeni. Odlučio sam taj problem riješiti tako da se u `if`-grananjima i `while`-petljama mogu nalaziti samo pretprocesorske naredbe, a ne i mnemonike. Istina je da se neke funkcije `if`-grananja i `while`-petlja time gube, ali neke ostaju, a lagano je za implementirati. FlatAssemblerove makro-naredbe bile bi još korisnije nego `if`-grananja i `while`-petlje, ali nemam pojma kako ih implementirati.

Nedugo nakon što sam implementirao `display` i `if` i `while`, uočio sam jedan veoma neobičan bug u pretprocesoru. Naime, pretprocesorska naredba „`display a`” (za prelazak u novi red, jer je ASCII kod znaka za novi red 10, a 10 je u heksadekadskom `a`) ne funkcionira ako je u vrijeme asembliranja UART isključen. Nisam uspio točnije dijagnosticirati problem, i nemam pojma kako pretprocesor uopće može znati je li UART uključen ili isključen. Otvorio sam GitHub issue o tom problemu²⁰.

Pretprocesor, prije nego što krene procesirati program, definira dvije konstante: `PicoBlaze_Simulator_in_JS` i `PicoBlaze_Simulator_for_Android`. Ukoliko pretprocesor misli da je pokrenut u internetskom pregledniku, `PicoBlaze_Simulator_in_JS` postavlja u 1, a `PicoBlaze_Simulator_for_Android` u 0. Ukoliko pretprocesor detektira da je pokrenut u mom programu `PicoBlaze_Simulator_for_Android` (više o njemu na kraju ovog teksta), tako što je deklariran globalni objekt `PicoBlaze`, koji u `PicoBlaze_Simulator_for_Android` služi kao sučelje pomoću kojeg komuniciraju potprogrami pisani u JavaScriptu s potprogramima pisanima u Javi (potprogrami pisani u Javi vide ga pod nazivom `WebAppInterface`), to je *vice versa*. Gotovo svi asembleri danas imaju pretprocesorske naredbe koje omogućuju asembliranom programu da detektira kojom se verzijom asemblera asemblira, uglavnom i naprednije nego ove što sam ugradio u svoj simulator PicoBlazea. Compiler za moj programski jezik nema pretprocesor. Jedino u njemu što bi donekle podsjećalo na pretprocesor je to što driver, ako su prve dvije riječi u nizu koji vrati tokenizer „`#target WASI`” ili „`#target browser`”, postavlja da globalna varijabla `compilation_target` bude jednaka toj drugoj riječi, i onda izbriše te prve dvije riječi iz niza (prije nego što ga preda parseru). Budući da tokenizer briše komentare, prije rečenice „`#target WASI`” ili „`#target browser`” mogu postojati komentari, ali ne mogu postojati nikakve deklaracije. Naime, postoje dvije veoma različite okoline u kojima se WebAssembly (koji ispisuje moj compiler) može pokretati, to su internetski preglednik (*browser*) i WASI (*WebAssembly System Interface*, što je pokušaj da se napravi standard pomoću kojeg je moguće izrađivati command-line programe koji će se vrtjeti na više operacijskih sustava koristeći WebAssembly). Oni, između ostalog, zahtijevaju da se globalne varijable deklariraju na različit način. A svi programi koje ispisuje moj compiler koriste globalnu varijablu zvanu `$stack_pointer`. To je, dakle, bio *quick-and-dirty fix* da se moj compiler može koristiti ako ciljamo WASI.

6. `simulator.js` – Taj se potprogram pokreće u zasebnoj dretvi kad pritisnemo tipku *play* ili *fast forward*, a u istoj dretvi ako pritisnemo tipku *single step*. On čita strojni kod u heksadekadskom obliku koji je u globalni objekt `machineCode` (deklariran u `PicoBlaze.html`) upisao potprogram `assembler.js`, te simulira PicoBlaze pišući i čitajući iz memorije (globalni objekt `memory` tipa `Uint8Array` deklariran u `PicoBlaze.html`), registara (niz, zvan `registers`, od dva globalna objekta tipa `Uint8Array` deklarirana u `PicoBlaze.html`, te globalna varijabla `PC`, *program counter*, koja pokazuje na naredbu u memoriji koja se trenutno izvršava) i zastavica (globalnih nizova `flagC`, `flagZ` te globalnih varijabli `flagIE` i `regbank`), pišući u izlaze (globalni objekt `output`), čitajući, koristeći DOM, ulaze iz one tablice s 256 HTML-ovih `input-a`, te upravljajući stogom `callStack`. Naknadno je dodano da za svaku naredbu

20 https://github.com/FlatAssembler/PicoBlaze_Simulator_in_JS/issues/8

provjerava je li na njoj breakpoint (to je moguće tako što globalni objekt `machineCode`, osim heksadekadskih stringova koji predstavljaju naredbe u strojnom kodu, sadrži i redni broj linije asemblerskog koda odakle naredbe dolaze, pa potprogram `simulator.js` može provjeriti nalazi li se redni broj linije asemblerskog koda odakle dolazi trenutna naredba strojnog koda u globalnom nizu breakpoints). Potprogram `simulator.js` ima 690 redaka. Razmišljao sam isprva o tome da jezgru simulatora napišem u svom programskom jeziku, a ne u JavaScriptu (preko svog kompilera koji cilja WebAssembly, standardizirani JavaScript bytecode), no odlučio sam da to ipak ne radim tako. Naime, compiler za moj programski jezik kompatibilan je samo s najnovijim internetskim preglednicima, ne može ciljati niti Microsoft Edge, a, *rebus sic stantibus*, danas tako nešto nije opcija ako želimo da nam web-aplikacija bude popularna. Uostalom, sigurno bih naletjeo na neke probleme vezane za komunikaciju potprograma pisanih u JavaScriptu i potprograma pisanih u mom programskom jeziku, tako da je upitno bih li se manje namučio pišući taj simulator u svom programskom jeziku.

Pri pisanju `simulator.js` naletio sam na problem da je on daleko sporiji od drugih simulatora PicoBlazea, o čemu sam pisao pred kraj ovog teksta. Također sam naletio na problem da ne znam kako se zastavice (flags) na PicoBlazeu ponašaju kada se promijeni *regbank*, da li zadržavaju svoj sadržaj ili postoje zasebne zastavice za svaki *regbank* (kao na računalu Z80, koji je po mnogo čemu sličan PicoBlazeu, ali o njemu ima daleko više informacija na internetu). Odlučio sam u `simulator.js` uprogramirati da za svaki *regbank* postoje zasebne zastavice (kao na Z80), a ispod tablice sa zastavicima napisati napomenu „*I have good reasons to think the emulation of flags is unrealistic, especially when it comes to REGBANKs.*”. Također sam naletio na problem da sam i u `assembler.js` i u `simulator.js` bio uprogramirao krivi *opcode* (Ono što se u asemblerskom jeziku zove mnemonika, u strojnom jeziku zove se *opcode*. Riječ *opcode* dolazi od *operation code*.) za mnemoniku *star* (*store argument*, stavi podatak u registar u drugoj *regbanki*, naime, zamišljeno je da glavni program koristi jednu *regbanku*, a funkcije koriste drugu *regbanku*, i da se argumenti koje šaljemo funkcijama spremaju u drugu *regbanku*), ali to sam brzo dijagnosticirao i ispravio kad sam primjer *Binary to Decimal* pokrenuo na pravom PicoBlazeu (usporedio sam heksadekadsku datoteku koju ispisuje moj assembler i heksadekadsku datoteku koju ispisuje Xilinxov assembler).

Korisnik GitHuba *agustiza* koristio je moj simulator na sveučilištu u Argentini te je uočio da sam ja pogrešno implementirao *bit-shifting* operaciju *sra*, te mi je 21. kolovoza 2023. napravio *pull request* s ispravkom tog buga²¹, koji sam prihvatio.

7. `tokenizer.js` – Tokenizer je dio kompilera koji kaže drugim dijelovima kompilera gdje završava koja riječ, a gdje počinje druga, u programskom jeziku. Riječi u programskom jeziku zovu se tokeni (engleski *token* izvan svijeta informatike znači *oznaka*). U programskim jezicima riječi ne moraju nužno biti odvojene razmacima, nego većina programskih jezika koristi malo kompleksnije mehanizme odvajanja riječi. Velika većina programskih jezika koristi mehanizam odvajanja riječi koji podsjeća na japansko pismo. Naime, japansko pismo ima tri skupa znakova: *hiragana*, *katakana* i *kandži*. *Kandži* su kineski znakovi i koriste se za pisanje korijenja riječi (i izvorno japanskih riječi i ranih posuđenica iz kineskog). *Katakana* je slogovno pismo koje se na japanskom jeziku koristi za pisanje korijenja stranih riječi ili onomatopeja. Drugim riječima, čim naučite katakanu, možete, čim vidite neki tekst na japanskom jeziku, prepoznati koje su riječi posuđenice i po njima odrediti o čemu je tekst. To nije slučaj na, recimo, korejskom jeziku. U Južnoj Koreji se i gramatički afiksi i strane riječi pišu hangulom, a u Sjevernoj Koreji se čak i korijeni riječi pišu hangulom (jer je u Sjevernoj Koreji u sklopu lingvističkog purizma zabranjeno korejski jezik pisati kineskim znakovima). Makar hangul bio lakši za naučiti ljudima naviknutim na alfabete nego što je katakana, da bismo u tekstu na korejskom jeziku prepoznali posuđenice, moramo dekodirati čitav tekst, a ne samo posebno označene riječi (kao što su na japanskom jeziku posuđenice posebno označene tako što su pisane katakanom). *Hiragana* je slogovno pismo koji se koristi za pisanje gramatičkih afiksa (prefiksa i sufiksa) na japanskom jeziku. Većina gramatičkih afikasa u japanskom su sufiksi, tako da, kada dođemo do nekog hiraganskog znaka, znamo da je to nastavak prethodne riječi, a ne nova riječ. S druge strane, ako naiđemo na kineski znak ili katakanski znak nakon niza hiraganskih znakova, to je najčešće nova riječ. Sličan postupak za odvajanje riječi postoji u većini programskih jezika. Tokenizer, osim toga, briše komentare. Potprogram `tokenizer.js` ima 95 linija. Tokenizer za moj programski jezik²², za usporedbu, ima 260 linija. Jedna od glavnih razlika između algoritma koji

21 https://github.com/FlatAssembler/PicoBlaze_Simulator_in_JS/pull/10

22 <https://github.com/FlatAssembler/AECforWebAssembly/raw/master/tokenizer.cpp>

sam koristio za svoj programski jezik i algoritma koji sam koristio za PicoBlaze je ta što u PicoBlazeu nisam pokušavao zapisivati broj stupca (*column number*), jer su svi reci u assemblerskom jeziku kratki, pa podaci o stupcima gdje počinje koji token neće previše pomoći u traženju pogreške. Također, tokenizer za moj programski jezik ne smatra znak za novi red tokenom, dok tokenizer za PicoBlazeov assemblerski jezik smatra. Tokenizer za moj programski jezik mora se brinuti o escape-sequence znakovima u stringovima (`\"`...), dok tokenizer za PicoBlazeov assembler ne mora. U compileru za moj programski jezik tokenizer zamjenjuje tokene poput `'a'` odgovarajućim ASCII vrijednostima (što možda i nije bila dobra ideja, jer poruke o pogreškama tipa *unexpected token* zbog toga mogu biti nejasne), u assembleru za PicoBlaze to radi potprogram `TreeNode.js`.

Kada sam pisao primjer PicoBlaze assemblerskog programa zvan „*Regbanks-Flags Test*”, naletio sam na idući problem. Programi kao što je ovaj:

```
address 0
load s9, " "
```

Prva naredba znači „*Počni pisati strojni kod od adrese 0.*” (i svaki PicoBlaze assemblerski program mora počinjati takvom naredbom), a druga naredba znači „*U registar s9 učitaj ASCII vrijednost razmaka.*”. Kad sam svom PicoBlaze simulatoru zadao da to asembliira, dobio sam ovakvu poruku o pogrešci: „*Line #2: The AST node "load" should have exactly three child nodes (a comma is also a child node).*”. To sam na brzinu riješio tako što sam umjesto druge naredbe napisao:

```
load s9, 20 ;Space
```

Naime, 20 je heksadekadski ASCII kod od razmaka. Pretpostavio sam odmah da se programi kao što je onaj koji sam napisao gore ne daju asembliirati zbog nekog buga u tokenizeru, ali nisam točnije dijagnosticirao problem. Tek sam sutradan uspio točnije dijagnosticirati i riješiti problem. Trebalo mi je malo vremena da se prisjetim kako program koji sam napisao tri godine ranije funkcionira.

Osim tih datoteka, u GitHub repozitoriju nalaze se slike napravljene u GIMP-u (Linuxov Paint) i Inkscapeu (Linuxov Publisher) te primjeri programa za PicoBlaze koje je moguće dohvatiti klikajući na *examplese*. Slike koje predstavljaju *play*, *pause*, *stop*, *fast forward* i *single step* su u SVG formatu i uređivane u Inkscapeu. Ikona za *Assembler Test* u GIF je formatu i uređivana u GIMP-u. Ikone za primjer *Hexadecimal Counter* te ikona koja predstavlja točku prekida (*breakpoint*) isto su uređivane u GIMP-u, ali spremljene su u PNG formatu, jer ih PNG format bolje sažima nego GIF format. Pozadina je fotografija PicoBlaze računala koju je profesor Ivan Aleksi uključio u svoju prezentaciju, posvijetljena u GIMP-u i spremljena kao GIF.

22. kolovoza 2023. godine korisnik GitHuba *agustiza* napravio je *pull request* u kojem se nalaze upute JEST-u (JavaScriptski framework koji služi za automatska testiranja) kako da automatski testira tokenizer²³, koji sam ja prihvatio. Taj *agustiza* mi je napisao da misli da bi automatske testove bilo mnogo lakše raditi da su moji potprogrami JavaScriptski moduli, a ne da se uključuju sa `<script src="...">`, a ja sam mu napisao da bih onda slomio kompatibilnost s Firefoxom 52 (jer je prva verzija Firefoxa koja podržava module Firefox 60), a da mi je važno da moj simulator PicoBlazea funkcionira u njemu. Kasnije smo dodali automatsko testiranje parsera, evaluacije aritmetičkih izraza od strane potprograma `TreeNode.js`, te malo automatskog testiranja `simulator.js`-a. Međutim, *test coverage* još je uvijek jako nizak, i, ako ćete doprinositi ovom projektu, nemojte se oslanjati isključivo na te automatske testove, već radite i ručno testiranje.

U compileru za moj programski jezik još postoji, kao i u compilerima za većinu programskih jezika, semantički analizer²⁴. To je dio kompilera koji „lovi” gramatički netočne rečenice koje prolaze kroz parser, ali koje bi srušile jezgru kompilera da dođu do nje. Takvih izraza, koji se na prvi pogled čine gramatički ispravnima, a zapravo nisu, ima i u ljudskim jezicima, i zovu se gramatičke iluzije. Najpoznatiji primjer takvog izraza jesu komparativne iluzije, rečenice tipa „*More people have been to Russia than I have.*”, „*Više je ljudi bilo u Rusiji nego što sam ja bio.*”. Na prvi se pogled ta rečenica doima ispravna, no, ako pokušate odrediti neko preciznije značenje, shvatit ćete da nešto s tom rečenicom sintaksno ne valja. Da bi poredbeni rečenica imala smisla, ako je subjekt glavne surečenice u množini, a predikat joj je glagol s nultom valencijom (kao glagol biti u egzistencijalnom značenju), poredbeni surečenica ne može imati isti taj predikat, ali u jednini. Parser u našem mozgu, očito, kao ni parseri u većini kompilera, nije napravljen da lovi sve sintaksne greške. Primijetite da je ovo posve druga vrsta besmislice nego Chomskyjeva rečenica „*Bezbojne zelene ideje spavaju bijesno.*”, Chomskyjeva rečenica je gramatički ispravna, samo što riječi imaju kontradiktorna značenja. Isto je to drugi fenomen nego poznata (obnovljena u raspravama o vezi

23 https://github.com/FlatAssembler/PicoBlaze_Simulator_in_JS/pull/11

24 <https://github.com/FlatAssembler/AECforWebAssembly/raw/master/semanticAnalyzer.cpp>

programskih i ljudskih jezika) Walter Burleyeva rečenica o kojoj je raspravljao 1328. godine „*Omne homo habens asinum videt illum.*”, nju je teško gramatički analizirati, ali, po njemu, svatko tko govori latinski složit će se da je to gramatički ispravna rečenica i da znači „*Ne posjeduješ magarca ako ga ne vidiš.*”. Govornici engleskog jezika ne slažu se što znači „*More people have been to Russia than I have.*”, ustvari, velika većina ljudi, kada razmisli, slaže se da ona ne znači ništa. Ne znam mogu li rečenice kao što je „*Omne homo habens asinum videt illum.*” postojati u programskim jezicima²⁵, ali rečenice kao što su „*More people have been to Russia than I have.*” sigurno mogu. Ipak, mislim da semantički analizer nije potreban za asemblerski jezik, jer mislim da je na asemblerskom jeziku nemoguće konstruirati rečenicu kao što je „*More people have been to Russia than I have.*”, iako su takve rečenice moguće u višim programskim jezicima.

Ako radite neki viši programski jezik (a ne assembler), nikad ne možete biti sigurni da će se vaš compiler ponašati smisleno u svim mogućim situacijama. Dati ću vam ideju o tome jednom nedavnom anegdotom iz svog života. Pitao sam na internetskom forumu na koje sve probleme možeš naletjeti kad implementiraš ternarni uvjetni operator `?:`²⁶. Meni se čini da mnogi programski jezici s tim operatorom imaju problema. Recimo, PHP ga neispravno parsira, on ga parsira kao lijevo-asocijativni operator, a treba ga se parsirati kao desno-asocijativni operator (što onemogućuje da se pomoću operatora `?:` u PHP-u skraćeno pišu *else-if* izrazi)²⁷. Moj compiler koji prevodi moj programski jezik na x86 asemblerski jezik (koji sam napisao u 4. razredu gimnazije, nije pretjerano kvalitetan projekt, pisan je u JavaScriptu i može se pokrenuti u internetskom pregledniku²⁸) krivo prevodi taj operator na asemblerski jezik. Naime, on prevodi drugi i treći operand prije nego što prevede prvi operand, a to, kako me je upozorio *rici*²⁹, može dovesti do greške kao što je dijeljenje s nulom. Naime, netko bi se mogao pokušati zaštititi od dijeljenja s nulom tako što napiše „`d = 0 ? 0 : n / d`” (ako je varijabla `d` jednaka nuli; `=` u mom programskom jeziku znači jednakost, a ne pridruživanje kao što znači u C-u), a to neće funkcionirati u dijalektu mog programskog jezika koji se prevodi na x86 jer se „`n / d`” (treći operand) izračunava prije nego što se izračunava „`d = 0`” (prvi operand). Pa mi je *kaya3* odgovorio da je jedna od stvari na koju moramo paziti kad implementiramo `?:` da se naš compiler ponaša smisleno ako netko zabunom pokuša kao drugi i treći operand staviti strukture različitog tipa (Za moj programski jezik, smisleno bi bilo da semantički analizer to ne propusti do jezgre compilera.). Hm, ne sjećam se da sam na to mislio kad sam pisao svoj compiler koji prevodi moj programski jezik na WebAssembly (Za onaj moj compiler koji ga prevodi na x86 to nije bitno, jer on ne podržava strukture.). Idem isprobati kako će moj compiler reagirati na to³⁰. On je ispisivao ovakvu poruku o pogrešci: `Line 10, Column 29, Internal compiler error: Some part of the compiler attempted to compile an array with size less than 1, which doesn't make sense. Throwing an exception!`

A u programu koji sam mu zadao da compilira nigdje nema *arraysova*. Stvarno apsurdna poruka o pogrešci, zar ne? Spadala bi u kategoriju *not even wrong*, ne možeš smisliti kako bi neka greška u compileru dovela do te poruke. Trebalo mi je nekoliko sati da shvatim što se zapravo u mom compileru događa. Što i nije iznenađujuće s obzirom na to da sam taj compiler pisao tri godine ranije, kad sam bio druga godina. Vjerojatno bi mi trebalo manje vremena (a ne nekoliko sati) da otkrijem što se u mom compileru događa da znam koristiti debugger. U biti, taj kôd kojim sam testirao kako će moj compiler reagirati na to da netko stavi strukture različitog tipa kao drugi i treći operand operatora `?:` *triggerao* je dva buga u mom compileru, jedan u semantičkom analizeru, a jedan u jezgri compilera. Onaj u jezgri compilera *triggerao* se samo ako su strukture prazne. I tko zna koliko takvih situacija, u kojima se moj compiler neće ponašati smisleno, još ima, kojih nisam svjestan. Pa, ipak, viši programski jezici razlog su zašto računala mogu sve ono što mogu danas. Da postoje samo strojni jezici i asemblerski jezici, internet gotovo sigurno ne bi postojao.

U nekim programskim jezicima još mogu postojati i rečenice kao što je „*Time flies like an arrow.*” (Je li *flies* ovdje glagol ili imenica? Je li *like* ovdje glagol ili veznik? O tome ovisi kako ćemo parsirati tu rečenicu.). Za takve programske jezike compileri između tokenizera i parsera imaju leksički analizer, koji parseru naznačuje koja je riječ koje vrste. Takve su rečenice moguće u C-u i C++-u, i to je poznato kao *typedef problem*. On je pojava da, izvan konteksta, `prvi*drugi`; (točka-zarez je dio izraza) može značiti i

25 Korisnik StackExchangea *kaya3* misli da se rečenice kao što je „*Omne homo habens asinum videt illum.*” mogu napraviti u Rustu i u drugim programskim jezicima koji podržavaju *pattern-matching*:

<https://languagedesign.stackexchange.com/a/1463/330>

26 <https://languagedesign.stackexchange.com/q/1469/330>

27 https://wiki.php.net/rfc/ternary_associativity

28 <https://flatassembler.github.io/compiler.html>

29 <https://stackoverflow.com/a/62104607/8902065>

30 Program kojim sam to isprobao dostupan je na mom GitHub profilu:

<https://github.com/FlatAssembler/AECforWebAssembly/issues/19#issue-1751055396>

„Deklariraj pokazivač koji se zove 'drugi' koji će pokazivati na instancu klase ili strukture koja se zove 'prvi'.“, ali može značiti i „Uzmi vrijednosti varijabli koje se zovu 'prvi' i 'drugi', pomnoži ih, stavi rezultat na sistemski stog i onda ga zanemari.“. Drugo značenje je, naravno, besmisleno (zanemarimo li da se operator množenja `*` u C++-u može preopteretiti u nešto sa side-effectsima), ali je gramatički moguće. Rečenice tipa „*Time flies like an arrow*.“ nisu moguće niti na PicoBlazeovom asemblerskom jeziku (osim što one riječi `enable` i `disable` mogu biti i glagoli i prilozi, ali to se trivijalno riješi u parseru) niti u mom programskom jeziku (barem se nadam da je tako), zato ni u PicoBlaze Simulator ni u compiler za svoj programski jezik nisam ugradio leksički analizer.

Primjeri programa za PicoBlaze

Web-aplikacija nudi osam primjera programa za PicoBlaze. Oni se nalaze, kao i njihov popis u JSON formatu, na autorovom GitHub profilu, te ih potprogram `PicoBlaze.html` dohvaća koristeći JavaScriptinu naredbu `fetch`. Prvi se primjer zove *Fibonacci Sequence*. On ispisuje Fibonaccijeve brojeve koji stanu u jedan byte (koliko su veliki PicoBlazeovi registri, svih 32), dakle, manje od 256. One koje se mogu prikazati u BCD (*binary coded decimal*, tako se u 8 bitova mogu prikazati brojevi manji od 100) formatu tako i ispisuje, a, one koji se ne mogu, ispisuje u heksadekadskom formatu. Također koristi bitovne operacije da bi izbrojao koliko ima jedinica u binarnom zapisu svakog od tih brojeva. Kad završi, javlja da je završio naredbom `return`, što ne bi prošlo na pravom PicoBlazeu. Da bi ispisao broj u novi red, on ispisuje na port sa sljedećom adresom, što na pravom PicoBlazeu isto ne bi prošlo. Ima 116 redaka, uključujući brojne komentare.

15	00	07	00
16	00	13	00
17	00	03	00
18	00	08	00
19	00	21	00
1a	00	03	00
1b	00	09	00
1c	00	34	00
1d	00	02	00
1e	00	10	00
1f	00	55	00
20	00	05	00
21	00	11	00
22	00	89	00
23	00	04	00

Slika 3: Dio izlaza programa "Fibonacci Sequence". 7. broj u Fibonaccijevom nizu jest broj 13, i on u svom binarnom zapisu (1101) ima 3 jedinice. 8. broj u Fibonaccijevom nizu jest 21, i on u svom binarnom zapisu (10101) isto ima 3 jedinice. Deveti broj u Fibonaccijevom nizu je 34, i on u svom binarnom zapisu (100010) ima dvije jedinice. 10. broj u Fibonaccijevom nizu jest broj 55, koji u svom binarnom zapisu (110111) ima 5 jedinica. A 11. broj u Fibonaccijevom nizu jest broj 89, koji u svom binarnom zapisu (1011001) ima 4 jedinice.

Program *Gray Code* koristi bitovne operacije da bi binarne brojeve pretvarao u i iz Grayevog koda. To je način kodiranja brojeva koji se često koristi u digitalnoj elektronici, ima svojstvo da se susjedni brojevi razlikuju samo u jednoj binarnoj znamenki. Recimo, brojanje od 0 do 10 u binarnom sustavu ide (promijenjene znamenke u susjednim brojevima su boldirane, a lijeve nule, koje nije potrebno pisati, u zagradama su): (000)0, (000)**1**, (00)**10**, (00)**11**, (0)**100**, (0)**101**, (0)**110**, (0)**111**, **1000**, **1001**, **1010**. U Grayevom kodu ti brojevi su: (000)0, (000)**1**, (00)**11**, (00)**10**, (0)**110**, (0)**111**, (0)**101**, (0)**100**, **1100**, **1101**, **1111**. To svojstvo je korisno jer ne moramo računati na to da su računala koja izmjenjuju poruke potpuno sinkronizirana, da jedno računalo počne čitati broj sa žice tek kad ga je drugo računalo već dovršilo mijenjati. To je osobito bilo važno za računala iz 1940-ih, na kojima implementiranje protokola sinkronizacije (Manchestersko kodiranje...) nije bilo jednostavno ili čak ni moguće. Program *Gray Code* koristi algoritam opisan na engleskoj Wikipediji, te se vrti u beskonačnoj petlji čitajući s ulaza. Ima 25 redaka. Program *Assembler Test* je besmislen program koji koristi sve naredbe koje podržava assembler za PicoBlaze, da bude lakše testirati assembler. Ako se pokrene u simulatoru, vrti se u beskonačnoj petlji. Ima 83 reda.

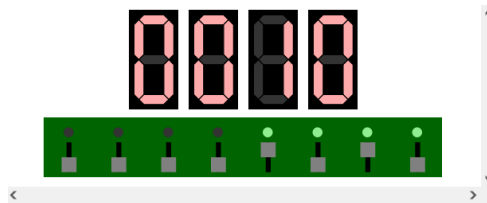
```

1. ;Examples of the assembly syntax:
2. address 0
3. ;Data directives...
4. beginningOfDataDirectives: ;Label
5. load s0, s1
6. load s1, 14'd / 2
7. star s2, s3
8. star s3, 1 + 2 * 3
9. namereg sf, stack_pointer ;Preprocessor
10. store s4, (stack_pointer)
11. store s5, FF
12. fetch s6, (sa)
13. fetch s7, A
14. input s8, (sb)
15. input s9, (1 + 2) * 3
16. output sa, (sc)
17. output sa, -1 + 15'd / 2
18. constant eight, 15'd/2 ;Rounding
19. outputk eight, a
20. jump endOfDataDirectives
21. regbank a
22. <

```

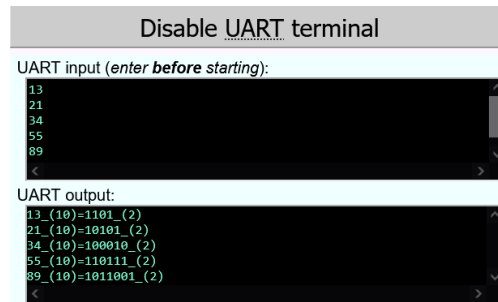
Slika 1: Program "Assembler Test" besmislen je program koji se dobije uz moj simulator PicoBlazea, njemu je jedini cilj ispoljiti greške u asembleru ukoliko one postoje.

Naknadno su dodani primjeri *Binary to Decimal* i *Hexadecimal Counter*. *Binary to Decimal* postavljen je kao prvi primjer s lijeva. On čita 8-bitni binarni broj unesen pomoću onih SVG-ovskih prekidača, pretvara ga u decimalni broj i rezultat prikazuje na sedam-segmentnim pokaznicima, te se vrti u beskonačnoj petlji. Pretvaranje binarnog broja u decimalni radi se algoritmom za pretvaranje binarnog broja u BCD, uz izmjenu da pazi nalazi se broj između 0 i 99, između 100 i 199 ili je više od 200 (8-bitni binarni brojevi mogu biti najviše 255). Uz to on zapošljava i LED-ice tako što na njima prikazuje broj pretvoren u Grayev kod. *Binary to Decimal* ima 54 retka. Isprobao sam ga na pravom PicoBlazeu i radio je. On koristi regbankse i zastavice, ali ne oslanja se na neku pretpostavku o tome kako se zastavice ponašaju kad se promijeni regbank, tako da to nije dokaz da je emulacija zastavica u mom simulatoru realistična što se tiče regbanksa.



Slika 4: Primjer izvršavanja programa "Binary to Decimal". Preko prekidača mu je unesen binarni broj (0000)1010 (prekidač gore, naravno, označava jedinicu, a prekidač dolje nulu), a on ga je pretvorio u dekadski broj 10 te je to ispisao na 7-segmentnom displeju. Također ga je pretvorio u Grayev kod (0000)1111, te je to ispisao pomoću LED-ica (primijetite da su zadnje 4 LED-ice upaljene, dok su prve četiri ugašene).

Hexadecimal Counter besmislen je program koji pali i gasi LED-ice te prikazuje heksadecimalne brojeve na 7-segmentnim pokaznicima. Nakon svih tih primjera dodan je i primjer *Decimal to Binary*, koji pretvara dekadске brojeve u binarne, a dekadski brojevi se u njega upisuju koristeći UART, a isto tako se iz UART-a čitaju binarni brojevi koji su rezultati. *Decimal to Binary* najkompliciraniji je PicoBlaze program koji sam ikada napravio, ima 283 retka. *Decimal to Binary* je sada drugi po redu s lijeva u popisu primjera u simulatoru (da ne prestrašim početnike, koji će najvjerojatnije prvo kliknuti prvi primjer s lijeva).



Slika 5: Primjer izvršavanja programa "Decimal to Binary". On čita dekadске brojeve iz UART terminala, pretvara ih u binarne brojeve, te rezultate ispisuje na UART terminal.

Program *Decimal to Binary*, na žalost, za sada nisam uspio pokrenuti na pravom PicoBlazeu. Kad sam pokušao, nije radio, a nisam imao vremena dijagnosticirati problem, čak ni to je li problem u mom programu ili je li problem u tome kako sam postavio PicoBlaze. Tako da trebate taj program kao primjer korištenja UART-a na PicoBlazeu uzeti *cum grano salis*. Dodao sam upozorenje o tome na početak tog programa i otvorio pitanje o tome na nekoliko internetskih foruma, ali do sada nisam naletio na nekog stručnjaka za PicoBlaze na nekom internetskom forumu (a nije pomoglo to što moderatori nisu baš bili oduševljeni mojim pitanjem, s programom od skoro 300 redaka u kojem ne znam ni otprilike gdje bi se nalazila greška). Kao što sam napisao dok sam opisivao *simulator.js*, imam dobar razlog za vjerovati da je emulacija regbanksova u mom simulatoru PicoBlazea nerealistična, ali program *Decimal to Binary* ne koristi regbankse, tako da nije u tome problem. Korisnik StackExchangea *Sep Roland*³¹ ima neke negativne komentare na moj primjer programa *Decimal to Binary* koji nisu vezani za to radi li taj program na pravom PicoBlazeu. Jedan od njegovih prigovora je da *Decimal to Binary* ima mnogo nepotrebnih provjera ispravnosti. Kako on kaže, „Your ‘basic sanity checks’ are more of ‘insanity’ checks.”, „Tvoje provjere ispravnosti više su provjere ludosti.”. On tvrdi da sve te provjere ispravnosti ne samo da usporavaju program i čine ga teško čitljivim ljudima, nego da možda i povećavaju rizik od pogreške (Što ako taj kôd koji provjerava je li se dogodila greška zapravo pogrešan, i prekine izvođenje programa iako se zapravo greška koju on traži nije dogodila?). Slično mi je i netko na Redditu rekao za moj compiler koji prevodi AEC na WebAssembly: „Ti kažeš da taj tvoj compiler ima 5'500 redaka, ali oko 50% toga jesu iste ili slične provjere i poruke o pogrešci nakon kojih slijedi naredba `exit(1)`, a u ostatku je hrpa bacanja iznimki. Mrzio bih raditi u timu s nekim tko piše takve programe.”. Ja imam dojam da u svoje programe ugrađujem premalo provjera ispravnosti, a ne previše. Recimo, u parseru za moj programski jezik bila je greška da se pri parsiranju desno-asocijativnih operatora radi čitanje izvan rubova vektora. Po svim pravilima zdrave logike, mislio sam, u tom bi slučaju C++-ova klasa `vector` bacala iznimku. Međutim, nije, nego je program radio ispravno na Linuxu i Windowsima, a na FreeBSD-u je izazivao *segmentation fault*³². Da sam dodao još koju naoko-besmislenu provjeru u parser za moj programski jezik, to mi se ne bi dogodilo. I mislim da nije moguće da se pogreška zbog koje *Decimal to Binary* ne radi na pravom PicoBlazeu nalazi u tim *sanity checkingsima*, jer *sanity checkingsi* u *Decimal to Binary*, ukoliko detektiraju pogrešku, pozivaju proceduru abort koja ispisuje „ERROR!” prije nego što upadne u beskonačnu petlju. Da je greška u nekom od tih *sanity checkingsa*, ispisao bi se „ERROR!”, ne bi bilo da PicoBlaze ne reagira ni na što na UART terminalu (kao što je bilo kad sam pokušao pokrenuti *Decimal to Binary* na pravom PicoBlazeu).

Oko tri godine poslije toga, napisao sam primjer *Regbanks-Flags Test*. To je program koji provjerava kako se zastavice na PicoBlazeu uistinu ponašaju kad se promijeni *regbank*. Ima 211 redaka, a jezgra mu je ovo:

```
regbank a
load s0, 0
sub s0, 0
regbank b
load s0, 1
sub s0, 0
regbank a
jump z , success
```

31 <https://codereview.stackexchange.com/questions/253951/converting-decimal-to-binary-in-assembly/253978#253978>

32 <https://github.com/FlatAssembler/AECforWebAssembly/issues/3>

jump nz, failure

Naredba „regbank a” znači „Postavi regbanku A kao trenutnu regbanku.” (regbank je ovdje mnemonika, glagol, a ne imenica). Naredba „load s0, 0” znači „Učitaj u registar s0, u trenutnoj regbanki, broj 0.”. U dijalektima asemblera koji ciljaju na ugrađene sustave (*embedded systems*) za učitavanje vrijednosti u registar obično se koristi mnemonika *load* (engleska riječ za *učitati*), a u asemblerskim dijalektima koji ciljaju PC-e i mobitele obično se koristi mnemonika *mov* (od latinskog *movere*, *premjestiti*). Naredba „sub s0, 0” znači „Oduzmi vrijednost 0 od registra s0.”, mnemonika *sub* dolazi od latinske riječi za *oduzeti*, *subtrahere*, od *sub* (ispod) i *trahere* (vući). Ako se nakon izvršenja te naredbe u registru nalazi vrijednost nula (kao u ovom slučaju), zastavica *z* u trenutnoj regbanki postavlja se u jedinicu. Naredba „regbank b”, naravno, znači „Postavi regbanku B kao trenutnu regbanku.”. Zatim se u registar *s0* učitava vrijednost 1, pa, kad se ponovno izvrši naredba „sub s0, 0”, zastavica *z* postavlja se u nulu (a ne u jedinicu, kao zadnji put). Zatim se vraćamo u regbanku A i ispitujemo je li zastavica *z* postavljena u jedinicu ili u nulu.

Na koji će label skočiti (*jump*) nakon što se izvrši taj dio programa? Ako sam ja u pravu, i u PicoBlazeu postoje zasebne zastavice (*flags*) za svaki regbank (kao na Z80), skočit će na label *success*. Ako je profesor Ivan Aleksi u pravu, i u oba rebanksa na PicoBlazeu koriste se iste zastavice, skočit će na label *failure*. U mom simulatoru, naravno, skače na *success*. Program *Regbanks-Flags Test* ispisuje svoje rezultate na UART, a, u slučaju da to ne upali, označava ih i LED-icama. Kada idući put budem imao pristup PicoBlazeu, isprobat ću taj program na njemu. Većina ljudi koji su mi se javili na internetskim forumima misli da je profesor Ivan Aleksi u pravu, ali nitko nije dao uvjerljiv dokaz (recimo, snimku zaslona UART terminala PicoBlazea kad se na njemu pokrene taj moj program *Regbanks-Flags Test*). Program *Regbanks-Flags Test* dodan je na kraj, kao 7. primjer.

Nekoliko tjedana nakon što sam napisao *Regbanks-Flags Test*, znatno sam unaprijedio pretprocesor svog asemblera za PicoBlaze, pa sam napisao primjer *Preprocessor Test*. Program *Preprocessor Test* za vrijeme asembliranja koristeći *if-else*-grananja i *while*-petlji izračunava brojeve u Fibonaccijevom nizu manje od 100 i ispisuje ih na terminal koristeći novododanu naredbu *display* (koju sam preuzeo iz FlatAssemblera). Program *Preprocessor Test* ne sadrži mnemonike i ne prevodi se ni u kakav strojni kod. On je dodan na kraj, kao 8. primjer.

Nakon tih 8 primjera, slijedi link na ZIP arhivu s programima pisanim na dijalektu mog programskog jezika koji cilja x86, većina s mnogo umetnutog asemblerskog koda, i link na upute kako na Linuxu isprobati analogni sat pisan tim dijalektom mog programskog jezika. To sam dodao jer mislim da bi danas svaki programer trebao znati osnove x86 asemblerskog koda (assemblerski kod računala na kojem radi i za kojeg pravi programe), da znati PicoBlaze asemblerski kôd nije zamjena za to. Analogni sat pisan onim dijalektom mog programskog jezika koji cilja WebAssembly može se pokrenuti u modernom internetskom pregledniku³³, ali pokrenuti onaj drugi je znatno kompliciranije, i zato sam linkao na upute kako to napraviti na Linuxu (a upute kako to napraviti na Windowsima nalaze se u ZIP arhivi). Compiler za moj programski jezik koji cilja na x86 ispisuje asemblerski kôd kompatibilan s i486, ali se onaj analogni sat zbog naredbi koje koristim u umetnutom asemblerskom kodu (*movzx...*) ne može pokrenuti na i486, nego samo na i586 i novijima (dakle, recimo, može u DosBoxu, danas najčešće korištenom simulatoru DOS-a). Kad sam pisao program *roseForDOS.aec*, pazio sam da u umetnuti assembler ne ubacim ni jednu instrukciju nekompatibilnu s i486, tako da bi se *roseForDOS.aec* iz te ZIP arhive trebao moći pokrenuti na i486 (iako nisam isprobao). A, u načelu, svaki se program za stare x86 procesore može pokrenuti na današnjim računalima. DOS se može pokrenuti na današnjim računalima, i neki programi funkcioniraju bez problema, a neki samo s manjim problemima. Recimo, sve što sam isprobao u QBASIC-u u DOS-u na današnjem računalu radilo je. Igrica *Prince of Persia*, ako se pokrene u DOS-u na današnjem računalu, može se pokrenuti i odigrati prvi level, ali se zaglavi pri prijelazu iz prvog levala na drugi (u DosBoxu, koji simulira i stari hardware, može se cijela odigrati). Međutim, Windows 3.11 ne mogu se pokrenuti na današnjem računalu. Probao sam ih instalirati, i pred kraj instalacije izbacilo me je s porukom da se procesor ne može prebaciti u 32-bitni način rada. Ne znam kako je to moguće, ali iz iskustva znam da je tako. Na internetu na više mjesta piše da se Windows 95 ne može pokrenuti na današnjem računalu jer u njemu postoji bug koji mu onemogućava da se pokrene na računalu s više od 480 MB RAM-a, ali da se Windows 98 može. Windows XP znatno se manje oslanja na BIOS nego Windows 98 i radi daleko više pretpostavki o tome kako hardware funkcionira, i zato se Windows XP ne može pokrenuti na današnjem računalu (probao sam).

33 <https://flatassembler.github.io/analogClock.html>

I, nakon ta dva linka, u flexboxu s primjerima, slijede link na GitHub i Reddit gdje me se može kontaktirati u vezi s tim PicoBlaze Simulatorom (recimo, ako imamo novi primjer koji želimo dodati). Naime, na Redditu sam otvorio subreddit o PicoBlazeu.

Tako da u flexboxu s primjerima ima ukupno 10 divova: osam primjera PicoBlaze programa i dva diva s linkovima. U slučaju da netko taj moj PicoBlaze simulator pokuša pokrenuti u internetskom pregledniku u kojem se JavaScript ne uspije izvršiti, u tom flexboxu nalazi se poruka o tome i link na web-stranicu TOR Browsera. TOR Browser, naime, koristi SpiderMonkey compiler za JavaScript, koji je poznat po tome da najbolje optimizira kod, znatno bolje nego V8 (koji koristi Chrome), a važno mi je da se JavaScript u mom PicoBlaze simulatoru dobro optimizira, jer se pri simulaciji mnogo puta izvršava isti JavaScript.

Mane simuliranja PicoBlazea u JavaScriptu

Naravno, taj simulator PicoBlazea što sam ga napravio u JavaScriptu ima i svoje nedostatke naspram *fully-featured* simulatora. Prvo, on ne pokušava interpretirati VHDL, tako da ne može simulirati PicoBlazeove s modificiranim VHDL kodom (osim ako ne promijenimo JavaScript, ali opet nam to neće pomoći da nađemo greške u VHDL kodu). Drugo, grafičko sučelje mu je mnogo manje efektivno nego sučelje koje pružaju najčešće korišteni simulatori PicoBlazea. Dobro je poznato da je u web-aplikacijama teško napraviti dobro korisničko sučelje. Profesor Ivan Aleksi mi je predlagao da probam koristiti neki JavaScript radni okvir (framework) za pravljenje korisničkih sučelja, kao što je ReactJS, no za njih treba vremena da se nauče, a i pitanje je koliko uistinu pomažu. Korisnik foruma `atheistforums.org` zvan *bennyboy* predložio mi je da pregledam primjere dobrog web-dizajna na web-stranici CSS frameworka Bootstrap, da me možda to inspirira da napravim dobar dizajn za svoj PicoBlaze Simulator koji je lagan za implementirati na webu³⁴. Treća je mana što je nemoguće napraviti realistični tajming. Jedna od prednosti PicoBlazea za korištenje u ugrađenim sustavima, gdje trebaju mala računala, jest upravo to što je lagano odrediti koliko će se dugo neki komad koda izvršavati ako znamo na koliko MHz-a radi (PicoBlaze može raditi na frekvenciji do oko 130 MHz), svaka instrukcija traje točno dva takta. U JavaScriptu je to nemoguće simulirati, jer JavaScript, na primjer, ima sakupljanje smeća koje se pokreće (što se JavaScriptskog programa tiče) nedeterministički. Uz to, na danas prosječnom računalu taj moj simulator može izvršiti oko 20 instrukcija po sekundi (u fast-forwardu u Firefoxu, pregledniku koji najbrže izvršava JavaScript), daleko manje od 75'000'000 operacija u sekundi koji bi bili potrebni za realistični tajming. Jedan čovjek na internetskom forumu tvrdi da ga vjerojatno najviše usporava to što se simulacijska dretva prekine kad se izvrši jedna instrukcija, a ponovno postavlja kad se treba izvršiti nova, a na postavljanje i uništavanje JavaScriptine dretve troše se mnogi računalni resursi³⁵. Dakle, da bismo postigli realističan tajming, morali bismo posve preurediti simulator, a vjerojatno i napisati ga u drugom programskom jeziku (a puno sreće da ga onda pokrenete u internetskom pregledniku). Pokušao sam ga ubrzati tako što sam manipulacije stringovima zamijenio bitovnim operacijama gdje je to jednostavno za napraviti, ali to nije urodilo plodom. Također, simulacija UART-a poprilično je nerealistična, a nije očito kako bismo u programskom jeziku JavaScript napravili dobar simulator terminala. Iako moj simulator nije dostatan za neke potrebe, smatram da je dostatan za potrebe studenata i da će ih spasiti *hasslea* instalacije naprednijih simulatora.

Korisnik foruma `atheistforums.org` zvan *bennyboy* ima dvije ideje kako ubrzati moj program. On misli da potprogram `assembler.js` znatno usporava to što se za svaki čvor u apstraktnom sintaksnom stablu mnogo puta provjerava je li riječ o registru, te da bih to trebao provjeriti samo jednom i rezultat te provjere pridružiti *boolean* varijabli³⁶. On također misli da bih u `simulator.js` trebao više koristiti *switch-case*, a manje *if-else*, jer se *switch-case* kompilira u optimiziraniji asemblerski kod (barem *bennyboy* i *HappySkeptic* tako misle; Meni to, kao nekome tko je napravio compiler za svoj programski jezik, nema previše smisla)³⁷. Bilo bi zanimljivo dati si truda i provjeriti te teze. *HappySkeptic* je pokušao analizirati performanse mog PicoBlaze simulatora dok se vrti program *Decimal to Binary* pomoću Chromeovih alata za programiranje i došao je do zaključka da on provodi više od 99% svog vremena crtajući SVG dijagrame³⁸ (koji su, naravno, za program *Decimal to Binary* beskorisni, jer program *Decimal to Binary* koristi UART za komunikaciju s korisnikom, a ne prekidače, LED-ice ili 7-segmentne pokaznike, koji se prikazuju SVG-

34 <https://atheistforums.org/thread-61911-post-2113649.html#pid2113649>

35 https://www.reddit.com/r/asm/comments/jyfrxy/how_to_implement_breakpoints_in_a_simulator/gd5ysu7/?utm_source=reddit&utm_medium=web2x&context=3

36 <https://atheistforums.org/thread-61911-post-2112572.html#pid2112572>

37 <https://atheistforums.org/thread-61911-post-2112817.html#pid2112817>

38 <https://atheistforums.org/thread-61911-post-2113176.html#pid2113176>

ovima). U tom bi se slučaju moj simulator PicoBlazea, barem za programe kao što je *Decimal to Binary*, mogao znatno ubrzati tako da omogućimo korisniku da *disable* SVG-ove, kao što se sada može *disable*ati UART. No, meni se to ne čini da je dobro objašnjenje. Da je tako, moj bi se program vrtio neprihvatljivo sporo u WebPositiveu, koji veoma sporo crta SVG-ove (zato je moj SVG PacMan neigriv u WebPositiveu). A u WebPositiveu moj se simulator PicoBlazea vrti otprilike jednako brzo kao što se vrti u Chromeu.

Budući da je moj simulator PicoBlazea neprihvatljivo spor ako se pokrene na mobilnom telefonu, razmišljao sam o tome da pokušam napraviti Android aplikaciju gdje će simulator biti pisan u Javi (dakle, da ponovno napišem `simulator.js`, ali ne u JavaScriptu, nego u Javi). Java isto nije idealan jezik za pisanje simulatora (sakupljanje smeća...), ali neki uspješni simulatori ipak jesu pisani u Javi (jDosbox...), jer je Java ipak brža nego JavaScript (statičko tipiranje...). Tako da sam pokušao napraviti native Android aplikaciju koja će omogućiti mobitelu da se pretvara da je PicoBlaze, dijelom i zato da se okušam kakav bih bio razvijatelj aplikacija za mobitele. Međutim, uz mnogo truda sve što sam uspio napraviti je da assembler pisan u JavaScriptu (`assembler.js`, `parser.js`, `preprocessor.js`, `tokenizer.js` i `TreeNode.js`) i program pisan u Javi komuniciraju (budući da je JavaScript koji koristi moj assembler za PicoBlaze dosta napredan, nisam mogao koristiti Rhino ili Duktape, nego sam baš morao komunicirati s onim JavaScript engineom ugrađenim u Android, koji ima komplicirano sučelje) te da program pisan u Javi dohvća i parsira onaj JSON popis primjera i primjere koristeći frameworkse Retrofit i GSON, pa sam odustao od te ideje. Ako netko želi nastaviti taj moj rad, dostupan je na mom GitHub profilu³⁹. Mnogi na internetskim forumima kažu da, kad se s razvoja desktop ili mobilnih aplikacija prebaciš na razvoj web-aplikacija, kao da si se vratio dva desetljeća u prošlost. Meni se ne čini da je tako. Napraviti korisničko sučelje od gumbova i labela uistinu je lakše kad ciljaš mobitel nego kad ciljaš web: kad ciljaš mobitel, to možeš napraviti bez kodiranja. Popuniti tablicu podacima (kao što moj PicoBlaze Simulator puni tablicu podacima iz globalnog objekta `machineCode`, strojnim kodom u heksadekadskom obliku i linijama asemblerkog koda odakle dolaze naredbe) već je podjednako teško na mobitelu i na webu. A dohvatiti nešto s interneta ili parsirati JSON daleko je lakše na webu nego na mobitelu.

Zahvale

Posebno zahvaljujem profesoru Ivanu Aleksiju što me potakao da ovo napravim i što je sakupio informacije na internetu potrebne za to. Simulator PicoBlazea koji se lako pokrene na raznim računalima osobito je bio potreban u vrijeme pandemije, kad je postojala mogućnost da se laboratorijske vježbe moraju raditi od kuće (na svu sreću, nije do toga došlo, ali to se lako moglo dogoditi).

Zahvaljujem i programerima koji su napravili internetski servis LGTM, statički analizer za JavaScript koji me je upozorio na neke greške koje sam napravio. JavaScript je relativno loš programski jezik i takvi su alati korisni.

I zahvaljujem GitHubu što hosta moju web-stranicu (uključujući i PicoBlaze Simulator), i još me nije zabranio zbog govora mržnje. Prije je moju web-stranicu hostala ciparska tvrtka 000webhost, pa su me zabranili zbog govora mržnje. Za slučaj da me i GitHub zabrani, ovaj PicoBlaze Simulator hosta se i na SourceForgeu⁴⁰, ali na SourceForgeu ne postam ništa kontroverzno zbog čega bi me imali razloga zabraniti. Ako me GitHub zabrani, onih šest primjera neće biti dostupno ni na SourceForgeu, jer ih i PicoBlaze Simulator koji se vrti na SourceForgeu dohvća s mog GitHub profila. Ali mislim da će to biti lagano srediti u tom slučaju.

SourceForge mi, za razliku od GitHuba, dopušta da na njihovim serverima vrtim PHP. Razmišljam o tome da napravim nekakav back-end tako da korisnici mogu dijeliti vlastite PicoBlaze programe i komentirati na njih. To bi se onda moglo vrtjeti samo na SourceForgeu, ne bi moglo na GitHubu. I za to bih morao naučiti znatno više PHP-a nego što sada znam.

39 https://github.com/FlatAssembler/PicoBlaze_Simulator_for_Android

40 <https://picoblaze-simulator.sourceforge.io/>

Abstract: *The author of the text was frustrated by the shortcomings of existing simulators of the small computer called PicoBlaze, so he created a different PicoBlaze simulator in the JavaScript programming language. This simulator can be run in modern Internet browsers⁴¹ (the author thinks that the oldest Internet browser in which the simulator works properly is Firefox 52, the last version of Firefox that can be run on Windows XP, and it is also the version of Firefox that comes with Solaris 11.4, which is the latest version of Solaris today). The text explains the details of how the author made his simulator and what the advantages and disadvantages are of that simulator compared to the existing simulators. The author is also comparing the parts of the simulator with similar parts of other programs he has made earlier (mostly a compiler that compiles his programming language to WebAssembly). No frameworks were used, the code was written mainly in VIM (for minor changes) and Eclipse (for major changes), GIMP and Inkscape were used for image editing, programming tools that come with Firefox were used for debugging the program, and the LGTM internet service was also used to search for errors. Prettier (for HTML and CSS) and ClangFormat (for JavaScript) were used to format the code.*

41 <https://flatassembler.github.io/PicoBlaze/PicoBlaze.html>

Životopis

Ja sam Teo Samaržija, rođen sam 1999. u Osijeku. Osnovnu školu sam završio u Donjem Miholjcu (Osnovna škola August Harambašić), a u srednju školu išao sam u Opću gimnaziju u Donjem Miholjcu (1. razred, dio 2. razreda, kraj 3. razreda i 4. razred), te u Opću gimnaziju u Našicama (kraj 2. razreda i početak 3. razreda).

Često sam sudjelovao na informatičkim natjecanjima. U 6. razredu osnovne osvojio sam 3. mjesto na županijskom natjecanju⁴², u 7. razredu 4. mjesto na državnom natjecanju⁴³, u 8. razredu 6. mjesto na državnom natjecanju⁴⁴ (sve je to organizirao Infokup), u srednjoj školi sam se, kad sam bio 3. razred, natjecao na HONI-ju, te sam osvojio 15. mjesto⁴⁵. 2019. godine sam, kao student prve godine, sudjelovao na STEM Games natjecanju iz programiranja, na kojem je moja ekipa osvojila 7. mjesto⁴⁶. Osim na natjecanjima iz informatike, sudjelovao sam, kad sam bio 2. razred srednje škole, i na AZOO-vom državnom natjecanju iz latinskog jezika, gdje sam osvojio 7. mjesto⁴⁷.

Kad sam bio gimnazijalac, izdavačka kuća Panon izdala mi je knjigu pod naslovom *Jezici za gimnazijalce*⁴⁸. Knjiga *Jezici za gimnazijalce* ima dva dijela, prvi je o povijesnoj lingvistici, a drugi o vezi programskih i prirodnih jezika. Naravno, sada i o jednom i o drugom znam mnogo više nego što sam znao kad sam pisao tu knjigu. Kad sam bio student prve godine, sudjelovao sam na konferenciji *Kopački Rit, Jučer, Danas, Sutra* s prezentacijom *Toponimija Baranje u svjetlu novih promišljanja*⁴⁹. 2022. godine napisao sam tekst *Etimologija Karašica*, o ideji da je to k-r što se ponavlja u nazivima rijeka u Hrvatskoj (Krka, Korana, Kravarščica, Krbavica, Krapina, dvije Karašice) bila ilirska riječ za *teći* (da je ilirska riječ za *teći* bila **karr~kurr*, recimo, da ime *Karašica* dolazi od nepotvrđenog ilirskog imena koje se može rekonstruirati ili kao **Kurrurrissia* ili kao **Kurirrissia*, a ime *Krapina* od **Karpona* ili, manje vjerojatno, od nečeg kao **Kurrippuppona*, i tako dalje), on je objavljen u Valpovačkom godišnjaku (što je spomenuto u Glasu Slavonije⁵⁰), na mojoj web-stranici⁵¹ te će, ako sve bude u redu, biti objavljen u časopisu za nacionalne manjine Regionalnim studijama u Sopronu. U tom tekstu, pomoću mjerenja kolizijske entropije hrvatskog jezika i Monte Carlo simulacija rođendanskog paradoksa, dokazujem da je vjerojatnost da se taj k-r uzorak dogodi slučajno negdje između 1/300 i 1/17. I to je, koliko znam, jedini do sada objavljeni članak koji primjenjuje teoriju informacija i teoriju vjerojatnosti na nazive mjesta u Hrvatskoj.

Govorim hrvatski (izvorni govornik), engleski (vjerojatno C1 razina), latinski (vjerojatno B2 razina⁵²) i malo njemačkog (imam B1 Deutsche Sprachdiplom).

Upisao sam 2018. godine FERIT u Osijeku, smjer Računarstvo, i trenutno sam student treće godine.

42 <https://informatika.azoo.hr/natjecanje/dogadjaj/175/rezultati>

43 <https://informatika.azoo.hr/natjecanje/dogadjaj/235/rezultati>

44 <https://informatika.azoo.hr/natjecanje/dogadjaj/288/rezultati>

45 http://hsin.hr/honi/arhiva/2016_2017/rezultati.php?show=KU_PS3

46 <https://stemgames.hr/wp-content/uploads/2019/05/SG-Rezultati-2019-znanje-T.pdf>

47 https://www.azoo.hr/images/rezultati_klas_jezici_lat_gimnazije2016.pdf

48 <http://www.glas-slavonije.hr/365330/3/Gimnazijalac-Teo-napisao-i-objavio-knjigu-o-jeziku>

49 <https://bib.irb.hr/datoteka/957836.Kopacki.pdf>

50 <http://www.glas-slavonije.hr/vijest.aspx?id=498689>

51 <https://flatassembler.github.io/Karasica.doc>

52 Tako procjenjuje korisnik Reddita CaiusMaximusRetardus na temelju mog YouTube videa na latinskom jeziku o životu nakon smrti: https://www.reddit.com/r/latin/comments/130dbyq/comment/ji632f2/?utm_source=reddit&utm_medium=web2x&context=3

Leo Samorzijs