Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Šmit

# Sistemski programski jeziki

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2016

*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Večina operacijskih sistemov je napisanih v programskem jeziku C. Podobno je s sistemsko programsko opremo, kot so npr. gonilniki, prevajalniki, razhroščevalniki, preverjevalniki diskov ipd. V zadnjem času se je pojavilo nekaj programskih jezikov (D, Go, Nim, Rust itd.), ki naj bi bili primerni tudi za sistemsko programiranje. V okviru diplomske naloge analizirajte trenutno stanje, definirajte kriterije, ki so pomembni za odločanje glede programskega jezika za sistemsko programiranje in primerjajte programske jezike glede na kriterije.

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Šmit sem avtor diplomskega dela z naslovom:

*Sistemski programski jeziki*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 22. februarja 2016 Podpis avtorja:

*Iskreno se zahvaljujem mentorju doc. dr. Juriju Miheliču za vso pomoč pri izdelavi diplomskega dela in čas, ki mi ga je posvetil. Posebna zahvala gre tudi staršem za vso podporo tekom študija.*

# Kazalo

# Seznam uporabljenih kratic in simbolov

| kratica | angleško | slovensko |
|---|---|---|
| **OS** | operating system | operacijski sistem |
| **API** | application programming interface | programski vmesnik |
| **WOCA** | write once, compile anywhere | napiši enkrat, prevedi kjerkoli |
| **GNU** | GNU's Not Unix | GNU Ni Unix |
| **gcc** | GNU C Compiler | prevajalnik jezika C |
| **GCC** | GNU Compiler Collection | zbirka prevajalnikov GNU |
| **SPO** | system sofware | sistemska programska oprema |

# Povzetek

Večina operacijskih sistemov je napisanih v programskem jeziku C. Podobno je s sistemsko programsko opremo, kot so npr. gonilniki, prevajalniki, razhroščevalniki, preverjevalniki diskov ipd. V zadnjem času se je pojavilo nekaj programskih jezikov, ki naj bi bili primerni tudi za sistemsko programiranje. To diplomsko delo predstavi programske jezike D, Go, Nim in Rust. Definirali smo kriterije, ki so pomembni za odločanje, ali je programski jezik primeren za sistemsko programiranje. Predstavimo programske jezike in jih analiziramo glede na definirane kriterije. Na podlagi kriterijev jezike medsebojno primerjamo in ovrednotimo ter jih primerjamo s programskim jezikom C.

**Ključne besede:** sistemsko programiranje, programska oprema, programski jezik.

# Abstract

Most operating systems are written in the C programming language. Similar is with system software, for example, device drivers, compilers, debuggers, disk checkers, etc. Recently some new programming languages emerged, which are supposed to be suitable for system programming. In this thesis we present programming languages D, Go, Nim and Rust. We defined the criteria which are important for deciding whether programming language is suitable for system programming. We examine programming languages and analyze them according to defined criteria. Based on criteria we evaluate programming languages, compare them mutually and with the C programming language.

**Keywords:** system programming, software, programming language.

# Poglavje 1

# Uvod

Večina operacijskih sistemov je napisanih v programskem jeziku C. Podobno je z ostalo sistemsko programsko opremo, kot so npr. gonilniki, prevajalniki, razhroščevalniki, preverjevalniki diskov ipd. Od leta 1972, ko se je prvič pojavil programski jezik C, do danes se je področje programskih jezikov zelo spremenilo. Pojavili so se novi programski jeziki, ki omogočajo višji nivo abstrakcije kot programski jezik C, ki ga nekateri označujejo tudi kot prenosljivi zbirnik (angl. *portable assembler*). Primeri takih višjenivojskih jezikov so Ruby, Python, Java, Javascript, PHP ipd. Programerju se v omenjenih jezikih ni potrebno ukvarjati z nizkonivojskem programiranjem, kot je rezervacija dovoljšnega števila bajtov za določen niz ali pa skrbeti za to, da se pomnilnik sprosti po njegovi uporabi. Zaradi abstrakcije in skrivanja nizkonivojskega programiranja so ti jeziki neprimerni za uporabo v sistemskem programiranju, praviloma pa tudi zmogljivostno slabši kot C. V sistemskem programiranju se področje programskih jezikov tako ni zelo spremenilo in še vedno prevladuje C.

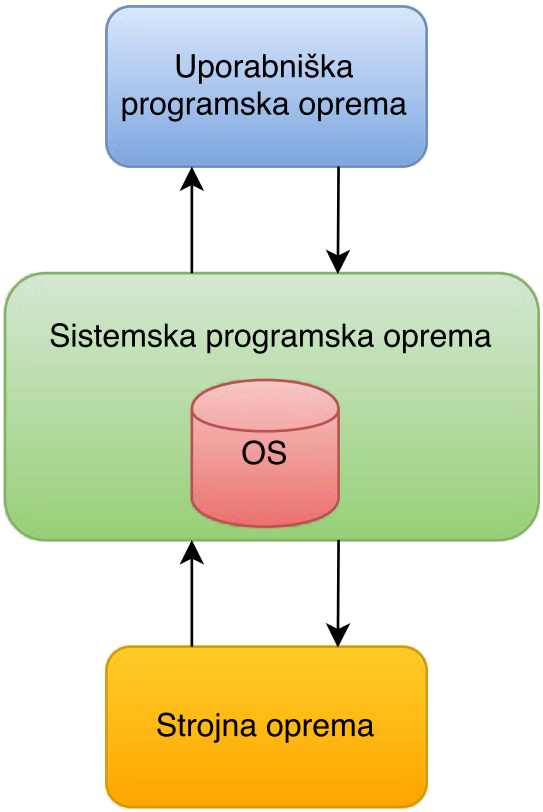
Cilj tega diplomskega dela je ugotoviti, ali obstajajo programski jeziki, ki bi zapolnili to nišo – torej bili podobni ostalim modernim programskim jezikom, obenem pa bi omogočali tudi nizkonivojsko programiranje in dosegali zmogljivosti jezika C. V zadnjih letih se je pojavilo nekaj novih programskih jezikov (D, Go, Nim, Rust ipd.), ki naj bi bili primerni za sistemsko

programiranje. Utopično je pričakovati, da bo kateri od novejših programskih jezikov zamenjal C na področju sistemskega programiranja. A vendar imajo programerji danes lahko izbiro, kateri programski jezik bodo uporabili za razvijanje sistemske programske opreme. V tem diplomskem delu bomo analizirali programske jezike D, Go, Nim in Rust ter poskušali ugotoviti, ali so primerni za sistemsko programiranje.

## 1.1 Sistemska programska oprema

Programsko opremo lahko razdelimo na sistemsko programsko opremo (angl. *system software*) in uporabniško programsko opremo (angl. *application software*). Slednjo predstavljajo uporabniške aplikacije, namenjene končnemu uporabniku, s katerimi uporabnik izvaja določene naloge, kot so npr. pisanje dokumentov, pošiljanje elektronske pošte ali brskanje po spletu.

Sistemska programska oprema predstavlja vmesnik med strojno opremo računalnika in uporabniško programsko opremo, kar prikazuje slika 1.1. Največkrat ta vmesnik predstavlja operacijski sistem, ki je najboljši primer sistemske programske opreme. Za razliko od uporabniške programske opreme SPO leži na nižjem abstraktnem nivoju, kjer tesno sodeluje z jedrom operacijskega sistema in sistemskimi programskimi knjižnicami. Računalnik brez sistemske programske opreme ne more delovati, lahko pa deluje brez uporabniške programske opreme, čeprav v tem primeru resda ni v veliko korist. Medtem ko uporabniška programska oprema nudi storitve končnemu uporabniku, sistemska programska oprema nudi storitve uporabniški programski opremi. Stroge definicije ni in včasih meja med uporabniško in sistemsko programsko opremo ni jasno določena. Dober primer je protivirusni program ali pa spletni brskalnik – v operacijskih sistemih, kot je Chrome OS, spletni brskalnik deluje kot edini grafični vmesnik, kar ga umešča med sistemsko programsko opremo.




Slika 1.1: Sistemska programska oprema kot vmesnik med uporabniško programsko opremo in strojno opremo.

Primeri sistemske programske opreme:

- operacijski sistemi
- sistemske programske knjižnice
- orodja za razvoj programske opreme (npr. prevajalnik, povezovalnik, razhroščevalnik)
- vmesnik z ukazno vrstico (angl. *command-line interface*) (npr. bash, ksh, cmd)
- ukazi v ukazni vrstici (npr. cd, mkdir, pwd, ls, tail)
- prikriti procesi (angl. *daemons*)
- gonilniki (angl. *drivers*)
- strojna programska oprema (angl. *firmware*)
- pomožna programska oprema (angl. *utility software*) (npr. protivirusni program, program za reorganizacijo diska, upravitelj datotek)

Programiranje sistemske programske opreme se imenuje sistemsko programiranje. Za razliko od aplikacijskega programiranja mora imeti programer globlje poznavanje strojne opreme in operacijskega sistema, na katerem dela. Izbrati mora ustrezni programski jezik, ki omogoča neposredno interakcijo s strojno opremo in nizkonivojskimi deli operacijskega sistema. Najpopularnejši tak jezik je že dolgo C.

## 1.2 Programski jezik C

Programski jezik C je nastal med letoma 1969 in 1973 v podjetju AT&T Bell Labs. V tem času je v istem podjetju tekel razvoj prvih različic operacijskega sistema Unix, ki je bil prvotno napisan v zbirnem jeziku. Programiranje v zbirnem jeziku je bilo časovno potratno, koda težko razumljiva in razhroščevanje težko. Programski jezik C je razvil programer Dennis Ritchie, nastal pa je zaradi potrebe po visokonivojskem programskem jeziku, v katerem bi bilo mogoče implementirati jedro sistema Unix in z njim povezane programske opreme. V tistem času sta bila najbolj popularna jezika Fortran, namenjen predvsem matematičnim nalogam, in COBOL, namenjen

poslovnim obdelavam podatkov, manjkal pa je pravi visokonivojski programski jezik, namenjen razvijanju operacijskih sistemov in ostale sistemske programske opreme. C je zapolnil to luknjo in kmalu postal eden izmed najpopularnejših programskih jezikov, kar ostaja še danes. Sodelavci Dennisa so osvojili nov programski jezik in kmalu (l. 1973) je bilo skoraj celotno jedro sistema Unix prepisano v jezik C. Tako je Unix postal eden izmed prvih operacijskih sistemov, kjer je bilo jedro OS napisano v visokonivojskem programskem jeziku.

Programski jezik C je preprost, učinkovit, imperativni programski jezik, ki spada v tretjo generacijo programskih jezikov in v t. i. družino ALGOL programskih jezikov. Predhodnik jezika C je bil B, ki je bil poenostavljena različica programskega jezika BCPL (Basic Combined Programming Language). C nudi programske konstrukte, ki se učinkovito preslikajo k strojnim ukazom in omogoča nizkonivojsko programiranje z neposrednim dostopom do pomnilnika računalnika. Skozi standardizacijo programskega jezika C (prva uradna specifikacija je bila ANSI C) je C postal lažje prenosljiv in tako danes obstaja malo operacijskih sistemov in računalniških arhitektur, za katere ne bi bil na voljo prevajalnik tega jezika. Veliko današnjih programskih jezikov izvira posredno ali neposredno iz C, npr. Java, C++, C#, Objective-C, PHP, Javascript, Python, Perl itd.

## 1.3 Pregled vsebine

Vsebina je zajeta v pet poglavij. V drugem poglavju definiramo kriterije, ki so pomembni za odločanje glede programskega jezika za sistemsko programiranje. Tretje poglavje vsebuje opis in analizo programskih jezikov D, Go, Nim in Rust glede na definirane kriterije. V četrtem poglavju medsebojno primerjamo jezike po kriterijih in jih kritično ovrednotimo. Zadnje poglavje pa sklene delo s sklepnimi ugotovitvami.

# Poglavje 2

# Kriteriji

V tem poglavju predstavimo kriterije, ki so pomembni za odločanje, ali je programski jezik primeren za sistemsko programiranje. Definirali smo naslednje kriterije:

- uporaba sistemskih klicev
- prenosljivost
- hitrost izvajanja
- funkcije prevajalnika
- upravljanje s pomnilnikom
- standardna knjižnica in izvajalno okolje
- povezovanje s programskim jezikom C

Vseskozi bomo vlekli vzporednice s programskim jezikom C kot najpopularnejšim sistemskim programskim jezikom. Obenem se bomo osredotočali predvsem na operacijski sistem GNU/Linux ter njegovim privzetim C prevajalnikom gcc.

## 2.1 Uporaba sistemskih klicev

Sistemski klici predstavljajo vmesnik med uporabniškim prostorom (angl. *user space*) in jedrom operacijskega sistema (angl. *kernel*), ta pa deluje kot vmesnik med programsko in strojno opremo. So mehanizem, s pomočjo katerega procesi od jedra zahtevajo določeno storitev operacijskega sistema (npr. odprtje datoteke, kreiranje novega procesa). Uporabniškim programom ni dovoljeno neposredno izvrševati kode v jedru OS ter tako naslavljati zaščiten del pomnilnika, ki je rezerviran za jedro (angl. *kernel memory*) zaradi varnosti in zanesljivosti takega sistema. Sistemski klici to omogočijo na varen in preverjen način.

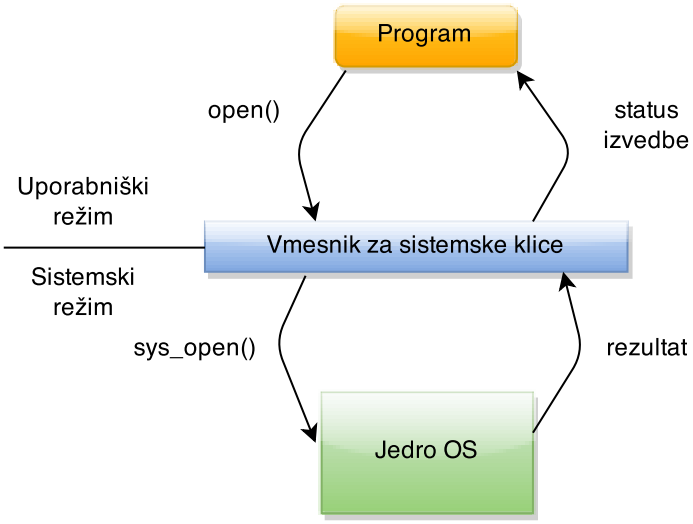
Primer uporabe sistemskega klica open na OS Linux v programskem jeziku C, ki odpre datoteko za branje:

```
int fd = open("file.txt", O_RDONLY);
```

Sistemski klic povzroči spremembo stanja procesorja iz uporabniškega v sistemski režim, ki ima višje privilegije, zato da lahko procesor dostopa do zaščitenega pomnilnika in aparaturne opreme. Kako je menjava režima implementirana, je odvisno od računalniške arhitekture – najbolj pogosta načina sta z uporabo programske prekinitve (zbirni ukaz INT za x86 procesorje) ali pa zbirnih ukazov SYSENTER/SYSEXIT (za procesorje Intel) oz. SYSCALL/SYSRET (za procesorje AMD). Jedro nato izvede zahtevano storitev, programu vrne kodo s statusom napake in spremeni stanje procesorja nazaj v uporabniški režim.

Operacijski sistem omogoča uporabo sistemskih klicev preko aplikacijskega programskega vmesnika (angl. *API - application programming interface*), definiranega v standardni knjižnici programskega jezika, kot je npr. glibc pri programskem jeziku C na operacijskem sistemu Linux (glej sliko 2.1).

S programerskega stališča sistemski klic ni nič drugega kot pa klic določene funkcije. V programskem jeziku C se klic sistemskega klica praviloma izvede iz standardne programske knjižnice jezika C, ta pa s pomočjo t. i. ovojnih




Slika 2.1: Sistemski klici kot posrednik med programom in OS.

funkcij (angl. *wrapper functions*) omogoča uporabo sistemskih klicev uporabniškim in sistemskim programom. To je pomembno predvsem zaradi prenosljivosti programov – obstaja en, enoličen vmesnik za klic sistemskih klicev, čeprav je implementacija teh različna od sistema do sistema. Tak vmesnik definira standard POSIX (*Portable Operating System Interface*), specificiran ravno za namen vzdrževanja združljivosti med različnimi operacijskimi sistemi.

V programskem jeziku C se sistemski klic lahko izvede na dva načina, in sicer s klicem ovojne funkcije, ki je definirana v standardni programski knjižnici C, ali pa s pomočjo rutine `syscall()`, ki se uporablja predvsem v primeru, ko sistemski klic nima svoje ovojne funkcije v programski knjižnici:

```
int ret = syscall(SYS_chmod, "/etc/hosts", 0444);
```

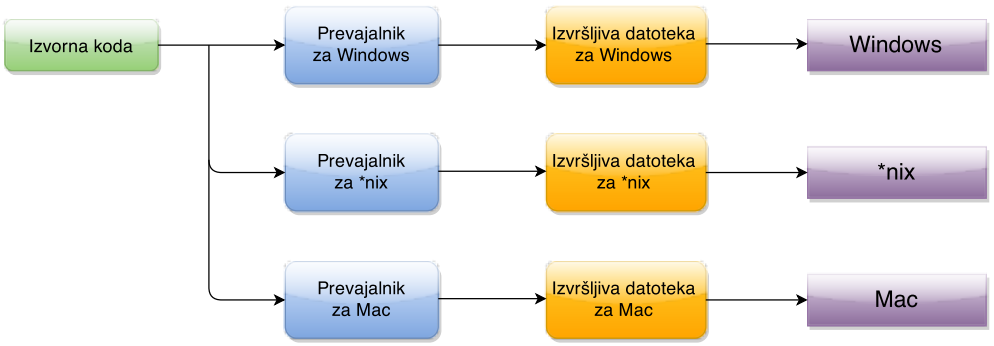
Ker sistemski klici omogočajo dostop do nizkonivojskih sistemskih gradnikov in tako predstavljajo najnižji nivo komunikacije z jedrom OS, je pomembno, da sistemski programski jezik omogoča njihovo uporabo.

## 2.2 Prenosljivost

Pojem prenosljivosti (angl. *portability*) predstavlja uporabo iste programske opreme na različnih računalniških platformah. Prenosljivost lahko razdelimo na dve kategoriji. Eno na nivoju izvorne kode in eno na nivoju prevedene izvršljive kode [2].

Prenosljivost na nivoju izvorne kode v grobem pomeni, da programski jezik sledi filozofiji WOCA (angl. *write once, compile anywhere* – napiši enkrat, prevedi povsod) – glej sliko 2.2. Program naj bi se prevedel na različnih platformah brez dodatnih sprememb izvorne kode. To se doseže z uporabo standardnih programskih knjižnic, ki skrijejo implementacijo za različne sisteme in programerjem ponudijo abstraktno raven funkcij in procedur ter tako prikrijejo spremembe med različnimi platformami. Programski jezik C danes sledi tej filozofiji. A vedno ni bilo tako. Zaradi razlik med implementacijami programskega jezika C na različnih sistemih Unix se je v 80. letih prejšnjega stoletja pojavila potreba po standardizaciji jezika C. Namen je bil, da postane programski jezik neodvisen od zasnove operacijskega sistema in računalniške arhitekture. C programi, napisani samo s pomočjo standardne programske knjižnice C, so prenosljivi na vse računalniške platforme, ki zagotavljajo implementacijo standardne programske knjižnice C. Prvič je bil standardiziran leta 1989 – ta različica se danes pogosto imenuje ANSI C.

Prenosljivost na nivoju izvršljive kode (angl. *write once, run anywhere* – napiši enkrat, zaženi povsod) pa predstavlja neodvisnost programa od sistema, kjer je bil preveden v izvršljivo kodo in sistema, kjer se bo izvajal. Programer lahko prevede program na računalniku z nameščenim operacijskim sistemom Linux in ga nato požene na računalniku z operacijskim sistemom Windows ali OS X. Vendar je to mogoče le na izredno podobnih platformah in v praksi redko uporabno. Obstajajo pa programski jeziki, ki to omogočajo na drug način. Eden izmed takih jezikov je Java, zahvaljujoč njenemu navideznemu stroju JVM (angl. *Java virtual machine*), v katerem teče vmesna koda (angl. *bytecode*). Navidezni stroj tako deluje kot tolmač vmesne kode za potrebe izvajanja na različnih platformah.




Slika 2.2: Napiši enkrat, prevedi povsod.

Za sistemski programski jezik je pomembno, da je prenosljiv na nivoju izvorne kode, saj to močno olajša delo programerjev, ker ne rabijo skrbeti za podrobnosti posamezne platforme, in s tem dobijo zagotovitev, da se bo program uspešno prevedel na različnih platformah. Prenosljivost na nivoju izvršljive kode nismo uvrstili kot kriterij zaradi potrebe po navideznem stroju, kar je za sistemski programski jezik z vidika uporabe pomnilnika in hitrosti neprimerno.

Pod pojmom prenosljivosti lahko razumemo tudi podporo sistemskega programskega jezika za različne operacijske sisteme in računalniške arhitekture. Če je sistemski programski jezik na voljo samo za določen OS ali procesorsko arhitekturo, ga to naredi manj prenosljivega. Dobro je, da ima podporo za čim več operacijskih sistemov in procesorskih arhitektur.

## 2.3 Hitrost izvajanja

Sistemski programi nudijo storitve predvsem uporabniškim aplikacijam, nekateri pa tudi končnim uporabnikom. V kolikor bi bili ti programi počasni in neodzivni, bi s tem bile počasne tudi uporabniške aplikacije, vplivalo pa bi tudi na splošno odzivnost sistema. Prevajalnik sistemskega programskega

jezika mora proizvesti učinkovito kodo glede na platformo, na kateri se bo program izvajal. Izvršljiva datoteka oz. program naj bo hiter v izvajanju in učinkovit v porabi sistemskih virov, predvsem pomnilnika in procesorskega časa. To pride v poštev predvsem v vgrajenih sistemih (angl. *embedded systems*), ko sta ta dva vira pogosto precej omejena.

Eden izmed glavnih razlogov, zakaj programski jezik C danes sodi med najpopularnejše jezike [8], ne samo glede sistemskega programiranja ampak tudi aplikativnega, je njegova hitrost in majhna poraba pomnilnika. K temu prispeva predvsem njegova preprosta zasnova. C programi se prevedejo neposredno v strojno kodo in ne potrebujejo kakršnegakoli tolmača (angl. *interpreter*), JIT prevajalnika (angl. *just-in-time compiler*) ali navideznega stroja pri svojem izvajanju.

## 2.4 Funkcije prevajalnika

Ena izmed dobrodošlih funkcij prevajalnika so t. i. optimizacijske zastavice (angl. *optimization flags*). S pomočjo takšnih zastavic lahko uporabnik optimizira svoj program za točno določen namen. Kjer je hitrost izvajanja ključnega pomena, bi uporabnik uporabil optimizacijske zastavice, s katerimi bi prevajalnik poskusil optimizirati kodo, da bi se program čim hitreje izvedel. V vgrajenih sistemih smo pogosto omejeni s prostorom – takrat bi uporabnik prevedel svoj program tako, da bi ta zavzemal čim manj prostora. Tabela 2.1 prikazuje primer optimizacijskih nivojev pri prevajalniku gcc.

Dobrodošla je tudi možnost uporabe zbirnega jezika (angl. *assembly language*) v sami programski kodi. To je možno na dva načina. S pomočjo t.i. medvrstičnih zbirnih ukazov (angl. *inline assembly*) – to so vrstice kode v zbirnem jeziku, umeščene neposredno v programsko kodo visokonivojskega programskega jezika, ali pa s klicem rutine, napisane v zbirnem jeziku, ki se nahaja v svoji datoteki in nato vključevanjem te datoteke pri prevajanju programa.

| Nivo | Pomen |
|---|---|
| `-O -O1` | Zmanjšanje velikosti programa in hitrosti izvajanja brez velikega vpliva na čas prevajanja. |
| `-O2` | Še večji nivo zmanjšanja velikosti programa in hitrosti izvajanja. |
| `-O3` | Največji nivo zmanjšanja velikosti programa in hitrosti izvajanja. |
| `-O0` | Brez optimizacije, skrajša čas prevajanja in olajša razhroščevanje (privzeto). |
| `-Os` | Optimizacija velikosti programa. |
| `-Ofast` | Optimizacija hitrosti izvajanja. |
| `-Og` | Vklop vseh optimizacij, ki ne vplivajo na težavnost razhroščevanja. |

Tabela 2.1: Prevajalnik gcc pozna sedem optimizacijskih nivojev.

Uporaba zbirnika je priročna predvsem zaradi treh stvari:

- ključne dele kode je mogoče napisati na bolj učinkovit način, kot pa v nekaterih primerih zmore to prevajalnik
- dostop do inštrukcij procesorja
- neposreden klic sistemskih klicev

Prevajalnik gcc omogoča pisanje medvrstičnih zbirnih ukazov v sintaksi zbirnika AT&T z uporabo programskega konstrukta asm oz. __asm__.
Primer kode, ki sešteje vrednosti 20 in 30 ter rezultat shrani v register eax:

```
__asm__ ( "movl $20, %eax;"
          "movl $30, %ebx;"
          "addl %ebx, %eax;"
        );
```

Datoteko z izvorno kodo v zbirnem jeziku (proc.s) pri prevajanju programa lahko vključimo tako:

```
gcc -o program main.c proc.s
```

Čeprav so sistemski klici dosegljivi preko ovojnih funkcij operacijskega sistema (oz. standardne knjižnice) in programerju ni potrebno pisati kode v zbirnem jeziku za dostop do njih, je to potrebno v nekaterih primerih. Če standardne knjižnice C na sistemu ni na voljo, je mogoče sistemski klic izvesti le s pomočjo ukazov v zbirnem jeziku. Taki primeri so sicer redki.

## 2.5 Upravljanje s pomnilnikom

Pomnilnik za spremenljivke se med izvajanjem programa dinamično dodeli na skladu ali kopici. Medtem ko se prostor za spremenljivke na skladu sprosti takoj, ko te niso več v dosegu, je potrebno pomnilnik na kopici sprostiti po njegovi uporabi. Za to se uporabljata dva načina, in sicer avtomatični s t. i. čistilcem pomnilnika (angl. *garbage collector*), ali eksplicitni, za kar mora poskrbeti programer na ustreznem mestu v kodi. Programske jezike lahko glede tega razdelimo na dve kategoriji, in sicer na tiste, ki poznajo avtomatično čiščenje pomnilnika in na tiste, kjer to ni del same specifikacije jezika.

Obe strani imata svoje prednosti in slabosti, čeprav v zadnjem času prevladuje mnenje, da uporaba čistilca pomnilnika prinaša več koristi kot slabosti in je zato vključen v vse pomembnejše programske jezike zadnjih let. Vendar se v nekaterih primerih uporaba ročnega sproščanja pomnilnika obnese bolje, kot pa lahko to opravi čistilec pomnilnika – najbolj očitno je to na sistemih z zelo omejenim pomnilnikom [1]. Optimalno bi bilo, da bi imeli možnost izbire, ali želimo uporabiti čistilec pomnilnika ali ne. Programski jezik C v svoji specifikaciji ne pozna avtomatičnega čiščenja pomnilnika, a je to možno z uporabo zunanjega čistilca pomnilnika, kot je npr. Boehm garbage collector [6], in manjšimi spremembami izvorne kode.

Programski jezik C omogoča nizkonivojski dostop do pomnilnika. S tem je mišljeno neposredno branje oz. spreminjanje vrednosti pomnilniških besed na določenem pomnilniškem naslovu. Za to se uporabljajo posebni programski tipi, imenovani kazalci (angl. *pointers*). Vrednost spremenljivke, ki je

definirana kot kazalec, je pomnilniški naslov, na katerem se nahaja neka vrednost. Z deferenciranjem takega kazalca lahko neposredno spreminjamo to vrednost.

Primer zapisa vrednosti 0xF000FF0F na pomnilniški naslov 0x20000000:

```
volatile unsigned int *addr = (unsigned int *) 0x20000000;
*addr = (unsigned int) 0xF000FF0F;
```

To je uporabno predvsem, ko hočemo dostopati do specifičnih registrov strojne opreme oz. pri dostopu do pomnilniško preslikanih vhodno/izhodno naprav (angl. *memory-mapped devices*). Pri programiranju gonilnikov ali mikrokontrolerjev (angl. *microcontrollers*) pride to velikokrat v poštev.

## 2.6 Standardna knjižnica in izvajalno okolje

Standardna knjižnica (angl. *standard library*) je urejena zbirka podobnih oziroma istovrstnih elementov konfiguracije programske opreme, ki pomaga pri razvoju, delovanju in vzdrževanju [3]. Je skupek raznovrstnih funkcij, ki omogočajo različne operacije, od ovojnih funkcij namenjenih sistemskim klicem (npr. odpiranje datoteke), do funkcij za delo s nizi, ki ne uporabljajo nobenega sistemskega klica (npr. sestavljanje nizov).

Standardna knjižnica sistemskega programskega jezika naj vsebuje le nujne programske konstrukte, ki jih programer potrebuje za implementacijo večjega dela programske opreme. Nepotrebno je v standardno knjižnico vključevati elemente, ki niso nujno potrebni, npr. funkcije za delo z dokumenti XML. Take funkcije naj bodo v dodatnih knjižnicah, ki se jih lahko nato vključi po potrebi. To pripomore k čim manjši velikosti standardne knjižnice, kar pomeni tudi manjšo velikost končnih programov, ki se ob prevajanju povezujejo s standardno knjižnico [5]. To je še posebej pomembno na sistemih z manjšo velikostjo pomnilnika, kot so vgrajeni sistemi.

Standardna knjižnica C (angl. *C standard library* ali *ISO C library*), specificirana v ANSI C in POSIX standardih, ima več implementacij, npr. GNU C Library oz. glibc, BSD Libc in Microsoft C Run-time library. Ob-

staja tudi veliko knjižnic, namenjenih za uporabo v vgrajenih sistemih, med bolj poznanimi so npr. dietlibc, uClibc in musl. Kot primerjavo povejmo, da je statična knjižnica dietlibc velika samo 120KB, kar je več kot 15x manj od statične knjižnice glibc, ki je velika približno 2MB. Preprost program, ki izpiše "Pozdravljen svet", pa je z uporabo statičnega povezovanja z dietlibc 100x manjši v primerjavi z glibc [4].

Na velikost izvršljivih datotek vpliva tudi izvajalno okolje oz. izvajalni sistem (angl. *runtime*). V programskem jeziku C izvajalno okolje crt0 vsebuje nabor zagonskih rutin, ki se med prevajanjem vključijo v generirano izvršljivo datoteko. Pred zagonom programa opravi potrebno inicializacijo procesa, ki je potrebna, preden se pokliče vstopna točka programa, t. i. metoda main. Funkcije, ki jih opravlja izvajalni sistem, variirajo od programskega jezika do programskega jezika. Medtem ko crt0 skrbi predvsem za klic vstopne točke programa, je lahko v drugih programskih jezikih izvajalno okolje bolj bogato in vključuje ostale minimalne rutine, potrebne za izvajanje programov, npr. čistilec pomnilnika ali pa navidezni stroj. Za sistemski programski jezik je dobro, da sta standardna knjižnica in izvajalno okolje čim manjša, saj to vpliva na velikost izvršljivih datotek.

## 2.7 Povezovanje s programskim jezikom C

Največ sistemske programske opreme je napisane v programskem jeziku C. Izumljanje tople vode je v razvoju programske opreme pogosto nezaželena praksa in ponovna uporaba kode (angl. *code reuse*) je pomemben programerski koncept. Cilj je prihranek časa in sredstev – tudi če je potrebno za uporabo kode plačati licenco, se to v večini primerov izplača, saj licenca tipično stane le 1-20 % stroškov, ki bi jih imeli, če bi hoteli komponento sami razviti [7]. Tu sicer treba prišteti še čas, ki ga porabimo za iskanje, učenje in integracijo zunanje komponente v naš produkt. Zaželeno je, da sistemski programski jezik omogoča povezovanje s programskim jezikom C ravno za potrebe ponovne uporabe že napisane sistemske programske opreme.

# Poglavje 3

# Analiza jezikov po kriterijih

V zadnjem času so se pojavili nekateri novi programski jeziki, ki naj bi bili primerni za sistemsko programiranje. V tem poglavju predstavimo programske jezike D, Go, Nim in Rust, in jih analiziramo glede na kriterije, definirane v prejšnjem poglavju. Jezike primerjamo s programskim jezikom C. Kjer je to smiselno, z izseki programske kode v posameznem jeziku pokažemo uporabo programskih konstruktov, ki so pomembni za sistemsko programiranje.

## 3.1 D

Programski jezik D je objektno orientiran, statično tipiziran (angl. *statically typed*), imperativni programski jezik, namenjen sistemskemu in aplikativnemu programiranju. Omogoča tudi funkcijsko in metaprogramiranje. Kot avtorja jezika sta podpisana Walter Bright iz podjetja Digital Mars in Andrei Alexandrescu, ki deluje kot glavni razvijalec. Prva različica je bila izdana leta 2001, medtem ko je različica 1.0 luč dneva ugledala l. 2007. D se oglašuje kot "tisto, kar je C++ želel biti" [10]. Ima preprosto sintakso, podobno programskemu jeziku C, lastnosti jezika pa vleče tudi s sodobnejšimi programskimi jeziki kot so Java, Python, Ruby in C#. Zaradi preproste sintakse in podobnosti z jezikom C je krivulja učenja nizka, koda pa hitro razumljiva. Obljublja moč in hitrost, primerljivo z jezikoma C/C++ ter primerljivo produktivnost

z dinamičnimi programskimi jeziki, kot sta Ruby in Python. Je visokonivojski programski jezik z dostopom do nizkonivojskih programskih konstruktov in operacijskega sistema. Programi, napisani v programskem jeziku D, se prevedejo neposredno v strojno kodo. Trenutna stabilna različica je 2.070.0.

Vzorec programske kode v D:

```
import std.stdio;
void main() {
    writeln("Hello from D!");
}
```

### 3.1.1 Uporaba sistemskih klicev

Programski jezik D je odvisen od standardne programske knjižnice C, katero za svoje delovanje potrebuje nameščeno na sistemu. Vmesnik sistemskih klicev je definiran v paketu core.sys v knjižnici DRuntime, ki skrbi za nizkonivojske funkcije in je del standardne knjižnice Phobos. Podobno kot programski jezik C tudi D pozna opcijsko prevajanje (angl. *conditional compilation*) – vključevanje le določene izvorne kode glede na vnaprej določene parametre. Ob prevajanju se tako vključi le modul za ciljni operacijski sistem.

Primer OS neodvisnega programa, ki pokliče sistemski klic, kateri izpiše ID trenutnega procesa:

```
version(Posix) {
    import core.sys.posix.unistd;
    writefln("pid: %d", getpid());
} else version(windows) {
    import core.sys.windows.windows;
    writefln("pid: %d", GetCurrentProcessId());
}
```

Ker se D zanaša na standardno knjižnico C, ima tako dostop do praktično vseh njenih ovojnih funkcij za sistemske klice pod pogojem, da so tudi ti definirani v knjižnici DRuntime. Ta vsebuje vmesnike do večine najpogosteje uporabljanih sistemskih klicev, vendar ne vseh - knjižnica je še nepopolna, a podpora raste z vsako novo različico jezika D. V kolikor želimo uporabljati

sistemski klic, za katerega DRuntime ne ponuja svoje ovojne funkcije, je potrebno v program vključiti deklaracijo ovojne funkcije sistemskega klica iz standardne knjižnice C in njenih specifičnih podatkovnih tipov.

Primer deklaracije ovojne funkcije sistemskega klica ptrace na OS Linux, kateri nima podpore v DRuntime:

```
import core.sys.posix.sys.types;
enum __ptrace_request { /* enum declaration */ }
extern(C) long ptrace(__ptrace_request request, pid_t pid,
    void *addr, void *data);
```

### 3.1.2 Prenosljivost

Za programski jezik D trenutno obstajajo trije prevajalniki, če ne štejemo poleg še vseh eksperimentalnih prevajalnikov. Uraden prevajalnik in referenčna implementacija jezika D je DMD (*Digital Mars D compiler*). Obstajata še GDC, ki uporablja zaledni del (angl. *backend*) zbirke prevajalnikov GCC (*GNU Compiler Collection*) in LDC, ki uporablja zaledni del prevajalniškega ogrodja LLVM. Vsem je skupno to, da prevedejo programe neposredno v strojno kodo.

Tabela 3.1 prikazuje, na katerih OS so na voljo prevajalniki. GDC in LDC lahko generirata izvršljivo kodo za vse platforme, katere podpirata zbirki prevajalnikov GCC in LLVM, vendar mora obstajati podprtost za nove platforme tudi v standardni knjižnici oz. izvajalnem okolju [11]. Te podpore pa ni veliko za ostale operacijske sisteme, ki niso omenjeni v tabeli 3.1. LDC podpira še eksperimentalno iOS in Android ter GDC Android. Vsi trije podpirajo procesorske arhitekture amd64 in i386, GDC in LDC pa ponujata podporo še za arhitekturo arm [12].

| DMD | GDC | LDC |
|---|---|---|
| Linux | Linux | Linux |
| Windows | Windows | Windows |
| OS X | | OS X |
| FreeBSD | | FreeBSD |
| | | OpenSolaris |

Tabela 3.1: Operacijski sistemi, za katere so na voljo prevajalniki jezika D.

### 3.1.3 Hitrost izvajanja

Programski jezik D obljublja hitrost primerljivo s C/C++. Kar pripomore k temu je dejstvo, da se izvorna koda prevede neposredno v strojno kodo. Optimizacija kode in hitrost izvajanja sta odvisni od prevajalnika – GDC in LDC proizvedeta bistveno hitrejše programe kot uradni DMD, kar je posledica dejstva, da uporabljata visoko optimizirana zaledna dela prevajalnikov GCC oz. LLVM [12].

Veliko kritik glede počasnosti je bil deležen čistilec pomnilnika, ki je v celoti implementiran v programskem jeziku D in je del knjižnice DRuntime [13, 14, 15]. Kritike niso naletele na gluha ušesa, tako da se je glavni razvijalec Andrei Alexandrescu odločil preprogramirati obstoječ čistilec pomnilnika z namenom doseganja boljših zmogljivosti [16]. Delo je v teku, do takrat pa je rešitev v kritičnih delih kode lahko izklop čistilca pomnilnika, kar dosežemo z ukazom:

```
GC.disable();
```

Če izklopimo čistilec pomnilnika, mora za sprostitev pomnilnika poskrbeti programer, tako kot je to v navadi v programskem jeziku C, uporabljamo pa lahko le podmnožico standardne knjižnice, ki ne zahteva uporabe avtomatičnega čistilca. Sčasoma bo tudi standardna knjižnica v celoti delovala brez uporabe čistilca pomnilnika [17].

### 3.1.4 Funkcije prevajalnika

Uradni prevajalnik DMD omogoča malo svobode pri optimizaciji pri prevajanju – za največji poudarek na hitrosti programov se ob prevajanju priporoča skupna uporaba zastavic -O -release -inline -boundscheck=off. Nekaj več opcij imamo z uporabo GDC ali LDC, ki omogočata iste optimizacijske nivoje kot njuna zaledna dela GCC oz. LLVM.

V programskem jeziku D je možno pisati medvrstične ukaze v zbirnem jeziku, kar velja za vse tri prevajalnike, razlika je le v sintaksi zbirnega jezika. DMD podpira sintakso Intel, GDC AT&T, medtem ko LDC podpira obe. Podobno kot pri programskem jeziku C je uporaba zbirnika možna z uporabo ukaza asm.

Primer programa, ki izpiše 'Hello world' s pomočjo sistemskega klica write in uporabo medvrstičnih zbirnih ukazov v sintaksi Intel za arhitekturo amd64:

```
void main() {
        string str = "Hello world";
        auto sptr = str.ptr;
        auto len = str.length;

        asm {
                mov RSI, sptr;
                mov RDX, len;
                mov RDI, 1; // stdout
                mov RAX, 1; // SYS_WRITE
                syscall;
        }
}
```

### 3.1.5 Upravljanje s pomnilnikom

Programski jezik D ima podporo za avtomatično čiščenje pomnilnika, obenem pa omogoča tudi eksplicitno upravljanje s pomnilnikom. Podobno kot jezik C pozna funkciji malloc in free, s katerima je mogoče ročno upravljanje s pomnilnikom:

```
import core.stdc.stdlib;
int *num = cast(int*)malloc((int).sizeof * 50);
// do stuff
free(num);
```

Čistilec pomnilnika tako pridobljenega kosa pomnilnika ne sprosti avtomatično, kar je dobrodošlo, saj tako lahko pišemo programe, ki delujejo v celoti brez čistilca pomnilnika. Da pa programski jezik D omogoča uporabo funkcije malloc tudi z vklopljenim čistilcem pomnilnika, obstaja še njen duplikat, definiran v drugem paketu:

```
import core.memory;
int *num = cast(int*)GC.malloc((int).sizeof * 50);
```

V tem primeru čistilec pomnilnika poskrbi za sprostitev pomnilnika in ni potrebna uporaba funkcije free.

D pozna tudi kazalce, ki jih definira podobno kot jezik C. Z operatorjem * in & deferenciramo oz. referenciramo spremenljivke, omogoča pa tudi aritmetiko s kazalci:

```
int[] array = [0, 0, 0];
int *ptr = &array[0]; // pointer to array[0]
*ptr = 1;
*(ptr+2) = 3; // array is now [1, 0, 3]
```

### 3.1.6 Standardna knjižnica in izvajalno okolje

Standardna programska knjižnica jezika D se imenuje Phobos. Vendar Phobos vedno ni bil prva izbira kot programska knjižnica za delo z D-jem. Do prihoda različice 2 jezika D (t. i. D2) je za popularnost tekmoval (in izgubljal boj) s programsko knjižnico Tango, razvito s strani odprtokodne skupnosti programerjev. Največji problem različice 1 jezika D je bilo dejstvo, da knjižnici med seboj nista bili združljivi. To je bila pereča tema v D skupnosti predvsem zaradi ponovne uporabe že napisane kode oz. vključevanja programskih paketov v naše programe – če je bila taka koda napisana s

pomočjo druge knjižnice, kot smo jo uporabljali mi, je tako nismo mogli ponovno uporabiti.

S prihodom aktualne različice D2 se je del kode, kriv za nezdružljivost, prestrukturiral v svoj del imenovan DRuntime, ki ga nato uporabljata Phobos in Tango. DRuntime je izvajalno okolje jezika D in predstavlja plast abstrakcije nad prevajalnikom. Različni prevajalniki imajo svojo implementacijo knjižnice DRuntime. Zaradi zahtevnega reprogramiranja knjižnice Tango za prehod (angl. *port*) na D2, se je delo na le-tej zelo upočasnilo, s prihodom D2 pa je Phobos napredoval do te mere, da ni več potrebe po Tangu. Tako je Phobos danes praktično edina izbira kot standardna knjižnica jezika D2.

Primer nezdružljivosti izvorne kode med knjižnicama (levo Phobos, desno Tango):

```
import std.stdio;

void main() {
    write("Phobos");
}
```

```
import tango.io.Console;

void main() {
    Cout("Tango") ();
}
```

### 3.1.7 Povezovanje s programskim jezikom C

Programski jezik D je bil zasnovan z namenom čim lažjega povezovanja s programskim jezikom C. Tako D pozna C podatkovne tipe, kot so size_t, struct in union, ki so analogni tistim v C-ju. Funkcije iz standardne programske knjižnice C je možno poklicati neposredno brez potreb po ovojnih funkcijah in pretvarjanja argumentov.

Primer klica C funkcije strcmp:

```
extern(C) int strcmp(const char * str1, const char * str2);
int main() {
    return strcmp("foo", "bar");
}
```

Na podoben način lahko dostopamo do globalnih spremenljivk, ki so deklarirane v izvorni kodi C programa:

```
extern (C) extern __gshared int c_var;
```

Tudi v kolikor hočemo povezovati z ostalimi knjižnicami, ki niso del standardne knjižnice C, je to možno z minimalno napora. Potrebno je napisati vmesno datoteko (angl. *interface file*), ki je v bistvu samo pretvorba zaglavne datoteke (angl. *header file*), napisane v programskem jeziku C. Pri prevajanju programa je nato potrebno vključiti želeno knjižnico C in vmesno datoteko. Pred tem pa je vredno preveriti, ali vmesna datoteka že obstaja na spletni strani projekta Deimos [18], kjer skupnost programerjev jezika D objavlja vmesnike do popularnejših knjižnic, napisanih v C/C++.

Tudi v obratni smeri (klic D programa/funkcije iz programskega jezika C) je stvar enostavna. V programskem jeziku D definiramo funkcijo s ključno besedo extern (C):

```
extern (C) int sum(int a, int b) {
    return a + b;
}
```

Nato jo lahko v izvorni kodi jezika C pokličemo kot vsako drugo zunanjo funkcijo:

```
extern int sum(int a, int b);
int main() {
    return sum(5, 3);
}
```

## 3.2 Go

Programski jezik Go (ali golang) je bil ustvarjen l. 2007 (prvič izdan l. 2009) v tehnološkem gigantu Google. Ustvarjalci jezika so računalničarji Robert Griesemer, Rob Pike in Ken Thompson, ki so dobro poznani računalniški javnosti. Med drugim Thompson velja za očeta operacijskega sistema Unix, Pike je sodeloval pri razvoju operacijskih sistemov Plan 9 in Inferno, skupaj pa sta zasnovala način kodiranja znakov UTF-8. Eden izmed glavnih razlogov za nastanek novega jezika je bil C++ in njegova naraščajoča kom-

pleksnost. Plan je bil razviti preprostejši sistemski programski jezik, ki bi bil podoben sodobnim, dinamičnim jezikom, ob tem pa bi ohranil hitrost, primerljivo s prevedenimi jeziki, kot sta C in C++. Veliko truda so vložili v razvoj učinkovitega čistilca pomnilnika, ki ga veliko sistemskih jezikov nima, in v čim manjši čas prevajanja programov [19]. Go je imperativni, statično tipiziran programski jezik, ki ponuja veliko podporo sočasnosti (angl. *concurrency*) in programiranju strežniških aplikacij. Tako ne čudi dejstvo, da je Go uporabljen v implementaciji Googlovega strežnika za prenašanje datotek dl.google.com. Sintaksa je ohlapno izpeljana iz programskega jezika C, vsebuje pa tudi elemente Pascala in modernih dinamičnih jezikov. Trenutna stabilna različica je 1.5.3.

Vzorec programske kode v Go:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello from Go!")
}
```

### 3.2.1 Uporaba sistemskih klicev

Vmesnik za sistemske klice je bil prvotno definiran v programskem paketu syscall v standardni programski knjižnici jezika Go. Zaradi različnih težav [20] so ustvarjalci jezika Go prenehali vzdrževati paket syscall, katerega bo počasi zamenjal novi paket golang.org/x/sys. Ta je trenutno v eksperimentalni fazi in še ni vključen v standardno programsko knjižnico, vendar se bo to zgodilo s prihodnjimi različicami jezika Go. Ker ustvarjalci priporočajo uporabo novega paketa [21], bomo tega uporabili tudi v naših primerih. Novi paket namestimo z ukazom go get -u -v -x golang.org/x/sys/unix.

V programskem jeziku Go vsak sistemski klic poleg ostalih vrednosti (Go omogoča funkcije, ki vračajo več vrednosti) vedno vrne tudi spremenljivko tipa error. Če je ta spremenljivka enaka nil, se je sistemski klic uspešno izvedel, v kolikor ni, pa vsebuje opis napake. To spominja na programski jezik

C, kjer ovojne funkcije sistemskih klicev prav tako nastavljajo spremenljivko errno v primeru napak.

Primer sistemskega klica open na operacijskih sistemih tipa Unix, ki vrne datotečni deskriptor (angl. *file descriptor*) in opis morebitne napake:

```
import "golang.org/x/sys/unix"
// somewhere in main...
fd, err := unix.Open("a.txt", unix.O_RDONLY, unix.S_IRUSR)
if err != nil {
    panic(err)
}
```

Programski jezik Go ima implementiran svoj nabor sistemskih klicev in za klic teh ne potrebuje pomoči standardne knjižnice C – torej ne uporablja ovojnih funkcij standardne knjižnice C, kot je glibc, ampak v ozadju uporablja zbirni jezik za izvedbo sistemskih klicev. To pa ne velja za alternativni prevajalnik gccgo, ki uporablja zaledni del GCC – ta sistemske klice izvrši preko standardne knjižnice jezika C [22]. Če določenega sistemska klica ni v standardni knjižnici, ga lahko izvršimo preko funkcij RawSyscall oz. RawSyscall6, katerim podamo številko sistemskega klica in potrebne argumente.

### 3.2.2 Prenosljivost

Trenutna različica privzetega prevajalnika gc podpira operacijske sisteme Linux, Windows, OS X, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, Plan 9 in Solaris. Prevajalnik zna generirati kodo za procesorske arhitekture amd64, i386, arm, arm64 in ppc64 [25]. Alternativa prevajalniku gc je gccgo, ki je čelni del (angl. *frontend*) zbirke prevajalnikov GCC in je od GCC različice 4.6 tudi vključen v uradno izdajo te zbirke prevajalnikov. S pomočjo gccgo je Go na voljo za široko paleto operacijskih sistemov in procesorskih arhitektur, ki jih podpira GCC. Rezultat prevajanja obeh prevajalnikov je izvršljiva datoteka, programi, napisani v programskem jeziku Go, pa so prenosljivi na nivoju izvorne kode.

Pomembna razlika med njima je, da privzeti prevajalnik gc prevaja statično,

medtem ko gccgo dinamično. To naredi programe, prevedene s privzetim prevajalnikom, lažje prenosljive, saj niso odvisni od različice standardne knjižnice jezika C – kar rešuje problem distribucije dinamično prevedenih programov na ostale operacijske sisteme, ki imajo inštalirano starejšo različico standardne knjižnice C od tiste, s katero je bil program povezan med prevajanjem.

### 3.2.3 Hitrost izvajanja

Eden izmed ciljev načrtovalcev jezika Go so zmogljivosti, ki bi se lahko primerjale s programskim jezikom C. V začetnih fazah Go ni bil hiter - primerjal se je lahko z interpretiranimi jeziki, kot sta Java in Python, bil pa je daleč od zmogljivosti prevedenih jezikov, kot je C, kar je bila predvsem posledica prevajalnika gc. Prevajalnik so v času močno izboljšali, še vedno pa je prostor za izboljšave, predvsem v določenih algoritmih standardne knjižnice in avtomatičnega čistilca pomnilnika. Prevajalnik jezika Go ima vključeno obsežno in zapleteno orodje za profiliranje programov, katerega uporabo ustvarjalci priporočajo v primeru performančnih težav. Hitrost je lahko odvisna tudi od izbire prevajalnika. Programi, prevedeni z gccgo, so lahko v aplikacijah, ki intenzivno izrabljajo procesorski čas, v nekaterih primerih lahko do 30 % hitrejši [23].

### 3.2.4 Funkcije prevajalnika

Različica 1.5 programskega jezika Go je prinesla velike arhitekturne spremembe, ena izmed njih je tudi ta, da je privzeti prevajalnik gc sedaj v celoti implementiran v Go (z malo pomoči zbirnega jezika) [24]. Gc ne podpira optimizacijskih zastavic pri prevajanju. Ustvarjalci jezika to opravičujejo s tem, da je optimizacija vključena že kot privzeto in ponujajo le izklop te. Go programi so torej že v osnovi hitri kolikor se da, če še vedno niso dovolj, pa bo treba krivca iskati v sami izvorni kodi. Z uporabo prevajalnika gccgo pa imamo na voljo vse optimizacijske zastavice, ki jih ponuja zbirka

prevajalnikov GCC.

Go pozna svoj dialekt zbirnega jezika, ki temelji na zbirnem jeziku za operacijski sistem Plan 9. Privzeti prevajalnik gc omogoča povezovanje z datotekami v zbirnem jeziku, vendar pa uporaba medvrstičnih zbirnih ukazov v programski kodi Go ni možna. To velja za oba prevajalnika.

### 3.2.5 Upravljanje s pomnilnikom

Programski jezik Go za upravljanje s pomnilnikom skrbi avtomatično, in sicer s čistilcem pomnilnika. Z nastavitvijo okoljske spremenljivke GOGC je sicer čistilec možno izklopiti, vendar je to namenjeno predvsem razhroščevanju programov. Go namreč ne ponuja eksplicitnih ukazov za dodeljevanje in sproščanje pomnilnika, tako kot to pozna C z ukazi, kot sta malloc in free.

Go pozna kazalce, vendar v omejenem obsegu, saj za razliko od jezika C ne omogoča aritmetike z njimi. Kot razlog za to ustvarjalci navajajo varnost, saj je tako nemogoče referencirati neveljaven pomnilniški naslov, poenostavi pa tudi implementacijo čistilca pomnilnika [19]. Sintaksa za delo s kazalci je ista kot v C z operatorjema * in &, pozna pa tudi vgrajeno funkcijo new, ki vrne kazalec do spremenljivke in rezervira dovolj pomnilnika za spremenljivko tipa T, ki ga podamo kot argument funkciji.

Prikaz uporabe kazalcev s funkcijo new:

```
func change_value(ptr *int) {
    *ptr = 99
}
func main() {
    ptr := new(int)
    fmt.Println(*ptr) // prints 0 (default value for int)
    change_value(ptr)
    fmt.Println(*ptr) // prints 99
}
```

Kot vidimo, Go ne omogoča veliko opcij pri nizkonivojskem upravljanju s pomnilnikom.

### 3.2.6 Standardna knjižnica in izvajalno okolje

Standardna knjižnica jezika Go se pogosto omenja kot knjižnica, ki je na voljo vključno z baterijami (angl. *batteries-included library*). S tem se misli na knjižnico, ki je bogata s funkcionalnostmi in omogoča takojšnji začetek dela brez potreb po namestitvi dodatnih modulov. Eden izmed jezikov, za katerega pravijo, da ima tudi standardno knjižnico z vključenimi baterijami, je Python. Standardna knjižnica Go tako vsebuje veliko zbirko programskih paketov, ki zagotavljajo širok nabor funkcionalnosti, od paketov za programiranje spletnih strežnikov, do paketov za povezovanje z različnimi podatkovnimi bazami. Nenazadnje je možno preprosti spletni strežnik, ki odgovori na zahtevo HTTP, napisati v praktično 10 vrsticah kode. Čeprav so taki paketi dobrodošli, pa vendar niso nujno potrebni za sistemsko programiranje in pripomorejo k velikosti standardne knjižnice. K le-tej pripomore tudi neodvisnost privzetega prevajalnika gc od jezika C, kar pomeni, da je izvajalno okolje, vključno z vmesnikom sistemskih klicev, v celoti implementirano v Go.

### 3.2.7 Povezovanje s programskim jezikom C

Go omogoča povezovanje s programskim jezikom C s posebnim orodjem `cgo`, ki je del standardnega prevajalnika gc. Vključi se ga tako, da se v programski kodi Go programa uvozi poseben psevdo paket C. V komentarju pred uvozom psevdo paketa, t. i. preambuli (angl. *preamble*), se vključi potrebne zaglavne datoteke jezika C, ob tem pa lahko tudi poljubno C kodo, do katere želimo dostopati iz programske kode Go:

```
// #include <string.h>
// #include <time.h>
// time_t mytime;
// char* get_time() { mytime = time(NULL); return (ctime(&
    mytime)); }
import "C"
```

Do funkcij in podatkovnih tipov jezika C se nato dostopa s predpono C. Potrebna je pretvorba podatkovnih tipov iz C v Go in obratno, za kar standardna knjižnica zagotavlja ustrezne funkcije:

```
rs = int(C.strcmp(C.CString("foo"), C.CString("bar")))
fmt.Println("c_time: ", C.GoString(C.get_time()))
```

V obratni smeri je stvar malo bolj zapletena, saj ni podpore za klic Go funkcij iz programskega jezika C, v kolikor ni vstopna točka programa, t.i. metoda main, napisana v Go. Edina možnost v tem primeru je start Go programa, ki pokliče funkcijo v C, ki nato pokliče funkcijo v Go. Če želimo funkcijo uporabiti v programski kodi jezika C, je potrebno pred njo dodati //export func-name, v C programski kodi pa vključiti zaglavno datoteko _cgo_export.h, katero samodejno ustvari prevajalnik gc.

## 3.3 Nim

Programski jezik Nim (prej poznan kot Nimrod) je statično tipiziran, imperativni sistemski programski jezik, ki podpira proceduralno, objektno usmerjeno, metaprogramiranje in funkcijsko programiranje. Prvič se je pojavil l. 2008. Avtor jezika je Andreas Rumpf, ki s pomočjo odprtokodne skupnosti vodi razvoj še danes. Nim poskuša programerju dati visokonivojske programske konstrukte, poznane iz dinamičnih jezikov, in uporabo modernejših programskih paradigem, kot je metaprogramiranje, ob tem pa ne želi žrtvovati učinkovitosti in prenosljivosti programskega jezika C, s katerim je tesno povezan. Namen novega jezika je bilo poiskati čim boljše razmerje med zmogljivostjo in produktivnostjo ter biti varnejši kot C. Sintaksa je preprosta in spominja na Python in Pascal, podobnosti pa vleče tudi iz programskih jezikov Modula 3, Delphi, Ada in C [26]. Trenutna različica jezika je 0.13.0. Ker Nim še ni dosegel stabilne različice 1.0, se sintaksa in semantika jezika do takrat še lahko spremenita. Primeri programske kode v tem delu so preverjeni z omenjeno različico jezika, različica 1.0 pa naj bi bila pripravljena v letu 2016.

Vzorec programske kode v Nim:

```
echo("Hello from Nim!\n")
```

### 3.3.1 Uporaba sistemskih klicev

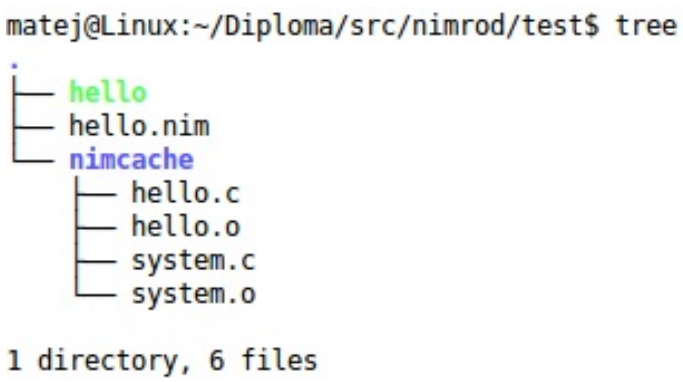
Programski jezik Nim sodi med jezike, ki za generiranje izvršljivih datotek potrebujejo C prevajalnik in standardno programsko knjižnico jezika C. Sistemske klice izvaja preko ovojnih funkcij knjižnice C – tako ima dostop do vseh, ki so definirani v C knjižnici. Deklaracija sistemskih klicev za sisteme POSIX se nahaja v standardni knjižnici v datoteki lib/posix/posix.nim, npr. sistemski klic mkdir je definiran kot:

```
proc mkdir*(a1: cstring, a2: Mode): cint {.importc, header:
    "<sys/stat.h>".}
```

Če določen sistemski klic manjka v standardni knjižnici Nim, ga lahko dodamo tako, da ga deklariramo kot zunanjo funkcijo programskega jezika C, podobno kot v zgornjem primeru. Če pa sistemskega klica ni na voljo niti v knjižnici C, vmesnik za sistemske klice v jeziku C pozna rutino syscall, preko katere lahko izvršimo take klice. Te funkcije ni v standardni knjižnici jezika Nim, zato jo moramo dodati mi, kar je trivialno – funkcijo deklariramo kot vsako drugo, ki jo hočemo poklicati v naši programski kodi. Primer programa, kjer deklariramo omenjeno rutino in preko nje izvedemo sistemski klic getcpu (specifičen za OS Linux), za katerega glibc ne pozna ovojne funkcije:

```
proc c_syscall(a1: cint, a2: ptr cint, a3: ptr cint, a4:
   cint): cint {.varargs, importc: "syscall", header: "<
   unistd.h>".}

var cpu, node : cint = 0
var ret = c_syscall(309, addr cpu, addr node, 0)
echo("CPU: ", cpu, " Node: ", node)
```




Slika 3.1: Datoteke, ki nastanejo pri prevajanju datoteke hello.nim.

### 3.3.2 Prenosljivost

Programski jezik Nim pozna en prevajalnik, imenovan preprosto nim. Na voljo je za operacijske sisteme Windows (XP ali kasnejši), Linux, OS X (10.4 ali kasnejši) ter procesorske arhitekture amd64, i386, ppc64 in arm [27]. Prevajalnik prevede program v dveh korakih. V prvem koraku programsko kodo v jeziku Nim pretvori v programsko kodo jezika C. Rezultat so datoteke v jeziku C, ki jih odloži v poddirektorij nimcache, kar prikazuje slika 3.1. Ustvarjena C koda ni neodvisna od platforme – generirana C koda za operacijski sistem Linux se ne prevede na Windows in obratno. Je pa zato programski jezik prenosljiv na nivoju programske kode Nim. V drugem koraku datoteke z izvorno kodo jezika C prevede v izvršljive datoteke. Za to uporabi C prevajalnik.

Prevajalnik nim zna proizvesti C kodo še za nekatere ostale operacijske sisteme, npr. Solaris, FreeBSD. NetBSD, OpenBSD ter procesorske arhitekture sparc in mips [28]. Generirano C kodo je nato potrebno prevesti s C prevajalnikom na operacijskem sistemu in procesorski arhitekturi, za katera je bila generirana C koda. Nim poleg pretvarjanja kode v programski jezik C omogoča isto še za programske jezike C++, Objective C in JavaScript.

### 3.3.3 Hitrost izvajanja

Programski jezik Nim je glede hitrosti primerljiv s programskim jezikom C. To je pričakovano, saj prevajalnik pretvori izvorno kodo v programsko kodo jezika C, nato pa z uporabo visoko optimiziranih prevajalnikov, kot sta gcc ali clang, dobimo podobno učinkovite programe, kot bi jih v jeziku C. Edini presežek (angl. *overhead*), ki ga ima Nim napram C, je čistilec pomnilnika, ki pa ga je v skrajnem primeru možno tudi onemogočiti.

### 3.3.4 Funkcije prevajalnika

Prevajalnik nim privzeto prevaja programe v načinu za razhroščevanje (angl. *debug mode*). Za prevajanje programov, primernih za objavo (angl. *release mode*), se pri prevajanju uporabi zastavico -d:release. Razlika med obema je tudi v uporabi različnega optimizacijskega nivoja, ki ga uporabi C prevajalnik pri prevajanju generiranih datotek s programsko kodo C v izvršljivo datoteko. Za prevajalnik gcc se v prvem primeru ne uporabi optimizacijskih zastavic, medtem ko se v drugem uporabi optimizacijski nivo -O3, kar pomeni najvišji nivo optimizacije. Katere optimizacijske zastavice so bile podane C prevajalniku, je možno videti v generiranih datotekah z izvorno kodo C, kjer je v komentarju na začetku zapisan ukaz za C prevajalnik.

Ker prevajalnik pretvori izvorno kodo v jezik C, Nim podpira tudi uporabo medvrstičnih zbirnih ukazov z ukazom asm. Edina razlika napram uporabi asm v jeziku C je uporaba trojnih narekovajev za začetek in konec uporabe konstrukta asm in to, da je potrebno Nim spremenljivke vključiti med znaki ‘. Katero sintakso zbirnega jezika je treba uporabiti, pa je odvisno od izbire C prevajalnika. Primer funkcije, ki sešteje dve števili z uporabo medvrstičnih zbirnih ukazov v sintaksi zbirnika AD&T in vrne rezultat tipa int:

```
proc sum(a, b: int): int {.compilerProc, inline.} =
    asm """
        addl %%ecx, %%eax
        :"=a"(‘result‘)
```

```
        :"a"(`a`), "c"(`b`)
    """

echo(sum(3,5))
```

### 3.3.5 Upravljanje s pomnilnikom

Programski jezik Nim glede upravljanja s pomnilnikom ponuja dobrodošlo izbiro, saj omogoča tako avtomatično čiščenje pomnilnika s čistilcem pomnilnika, kot tudi eksplicitno upravljanje. Medtem ko je čistilec pomnilnika privzeto vključen, je mogoče ročno upravljanje s pomnilnikom doseči z družino ukazov alloc in dealloc, ki so analogni ukazom malloc in free iz programskega jezika C. Eksplicitno dodeljen pomnilnik ne smemo pozabiti sprostiti po uporabi, saj čistilec pomnilnika ne poskrbi za njega in lahko pride do uhajanja pomnilnika (angl. *memory leak*). Uporaba omenjenih ukazov je namenjena predvsem povezovanju s programskim jezikom C in ne splošni uporabi, kar je vidno tudi iz standardne knjižnice, kjer je takih primerov malo. Pri klicih funkcij standardne knjižnice C pa je to priročno – primer klica funkcije fread, ki pričakuje kot argument kazalec do izravnalnika (angl. *buffer*):

```
const size = 4000 # 4000 bytes
var buf = alloc(size)
var result = fread(buf, 1, size, file)
# ...
dealloc(buf)
```

Nim pozna kazalce, vendar ne omogoča aritmetike z njimi v stilu C-ja – kot razlog navajajo veliko število napak pri takšnih operacijah. Se pa da s pomočjo metaprogramiranja z uporabo predlogov (angl. *templates*) in prekrivanja operatorjev (angl. *operator overloading*) aritmetiko s kazalci do neke mere simulirati [30]. Nim kazalce obravnava s pomočjo operatorjev addr in [], ki so analogni operatorjem & in * v C-ju:

```
# nim
var x = 2
var y = addr(x)
y[] = 7
```

```
# C
int x = 2;
int *y = &x;
*y = 7;
```

### 3.3.6 Standardna knjižnica in izvajalno okolje

Standardna knjižnica programskega jezika Nim je razdeljena na tri dele, ki jih ustvarjalci imenujejo čiste knjižnice (angl. *pure libraries*), nečiste knjižnice (angl. *impure libraries*) in ovojnice (angl. *wrappers*). Čiste knjižnice so neodvisne od zunanjih programskih knjižnic (datoteke .dll na Windows oz. lib*.so na Linux), medtem ko nečiste niso. Ovojnice predstavljajo vmesnik do knjižnic v programskem jeziku C, npr. vmesnik mysql za dostop do programskega vmesnika MySQL C.

Največji del standardne knjižnice predstavljajo čiste knjižnice. Čeprav med njimi najdemo tudi komponente, ki niso nujne za sistemsko programiranje, kot so funkcije za delo z dokumenti XML in strežniki HTTP, pa je standardna knjižnica še vedno dovolj kompaktna. K manjši velikosti pripomore tudi dejstvo, da se standardna knjižnica Nim velikokrat navezuje na standardno knjižnico C in ne izumlja tople vode. Tako npr. funkcija formatBiggestFloat iz modula za delo z nizi v ozadju uporablja funkcijo sprintf iz standardne knjižnice jezika C, ki je sama ne implementira. To posledično prinese manjšo velikost izvršljivih datotek.

### 3.3.7 Povezovanje s programskim jezikom C

Ker je programski jezik Nim tesno povezan z jezikom C je medsebojno povezovanje enostavno. Za klic standardnih funkcij programskega jezika C je potrebno deklarirati funkcijo s pragmo .importc, ki pove prevajalniku, da uvažamo C funkcijo, ob tem pa povedati, v kateri zaglavni datoteki se nahaja. Podobno velja za ostale nestandardne funkcije, ki jih z direktivo –passL: vključimo pri prevajanju ali pa že v izvorni kodi Nim s pragmo .compile po-

vemo, naj se C datoteka prevede skupaj z našim programom:

```
{.compile: "calc.c".}
proc c_multiply(x, y: cint): cint {.importc.}
proc printf(str: cstring) {.header: "<stdio.h>", importc: "
    printf", varargs.}

printf("%s \n", "printf from C") #std C func
echo("3*4= ", c_multiply(3,4))   #custom C func
```

Če želimo funkcijo, napisano v Nim, uporabiti v programskem jeziku C, uporabimo pragmo .exportc:

```
proc nim_strcat*(s1, s2: cstring): cstring {.exportc.} =
$s1 & $s2
```

Nim pozna orodje c2nim, ki omogoča samodejno pretvarjanje programske kode C (ANSI C) v programsko kodo Nim. Orodje ni del standardne knjižnice, vendar je njegova uporaba dokumentirana v standardnem priročniku. Namenjeno je predvsem translaciji posameznih delov C kode, rezultat pa je smiselno preveriti ročno, saj orodje ne obljublja delovanja povsem brez napak [29].

## 3.4 Rust

Programski jezik Rust je programski jezik, namenjen predvsem sistemskemu programiranju. Jezik je nastal iz osebnega projekta programerja Graydona Hoareja, ki je bil takrat zaposlen v podjetju Mozilla. Mozilla je l. 2010 začela financirati razvoj jezika Rust in ga podpira še danes, k njegovemu razvoju pa velik del prispevajo tudi člani odprtokodne skupnosti. Prva različica je bila izdana l. 2010, maja 2015 pa je bila izdana prva stabilna različica 1.0. Je preveden programski jezik, ki podpira funkcijsko, proceduralno, objektno usmerjeno in sočasno programiranje. Razvili so ga s fokusom na tri cilje: varnost, hitrost in sočasnost [31]. Osredotoča se na nizkonivojsko programiranje in je primeren za razvijanje operacijskih sistemov – Redox je OS, napisan v jeziku Rust [36]. Sintaksa jezika je podobna C/C++, vendar je semantično

zelo različen od omenjenih. Rust ni najbolj primeren jezik za začetnike v programiranju, saj je znan po tem, da ima strmo krivuljo učenja. Trenutni največji projekt, v katerem se uporablja Rust, je Servo – eksperimentalni, izvajalni sistem spletnega brskalnika (angl. *web browser engine*) podjetja Mozilla. Trenutna različica jezika je 1.6.0.

Vzorec programske kode v Rust:

```
fn main() {
    println!("Hello from Rust!");
}
```

### 3.4.1 Uporaba sistemskih klicev

Programski jezik Rust sistemske klice izvede preko standardne knjižnice jezika C. Na operacijskem sistemu Linux je to knjižnica glibc, eksperimentalno pa podpira še knjižnico musl. Klici ovojnih funkcij za sistemske klice so definirani v paketu libc. V Rust terminologiji se programskim paketom, ki vključujejo podobne funkcionalnosti, reče zaboj (angl. *crate*).

Rust je glede varnosti zelo strog, zato vsak klic nevarne kode (v tem primeru C) zahteva uporabo ključne besede unsafe, s katero povemo prevajalniku, da sprosti svoje omejitve glede varnosti izvajanja kode – v nasprotnem primeru se program ne prevede uspešno. Primer uporabe sistemskega klica chdir, kjer se želimo premakniti mapo višje:

```
let parent_dir = CString::new("..").unwrap();
unsafe {
    libc::chdir(parent_dir.as_ptr());
}
```

Če za sistemski klic ne obstaja ovojna funkcija v glibc ali pa manjka deklaracija le-te v zaboju libc, klic lahko izvršimo preko funkcije syscall, ki je na voljo v glibc. V primeru spodaj deklariramo funkcijo syscall in preko nje izvršimo sistemski klic gettid (številka sistemskega klica na 64-bitnem Linux je 186), ki nima svoje ovojne funkcije v glibc:

```
extern {
    pub fn syscall(num: c_long, ...) -> c_long;
}
// somewhere in main...
unsafe {
    let ttid = syscall(186);
    println!("Thread ID: {}", ttid);
}
```

Ker Rust sistemske klice izvaja preko standardne knjižnice C, lahko tako dostopa do vseh ovojnih funkcij za sistemske klice. V primeru, da standardne knjižnice C ni, pa lahko sistemske klice izvrši neposredno z uporabo medvrstičnih zbirnih ukazov.

### 3.4.2 Prenosljivost

Rust kot zaledni del svojega prevajalnika rustc uporablja prevajalniško ogrodje LLVM. Podpora za računalniške platforme je razdeljena v tri sklope. V prvem sklopu so operacijski sistemi in procesorske arhitekture, na katerih se zagotavlja pravilno delovanje prevajalnika in standardne knjižnice preko avtomatičnih testov ob vsaki spremembi jezika. Sem spadajo platforme Windows (7 ali kasnejši), Linux (2.6.18 ali kasnejši) in OS X (10.7 Lion ali kasnejši) na procesorskih arhitekturah amd64 in i386.

V sklopu 2 so platforme, za katere se zagotavlja uspešno prevajanje, a se avtomatični testi ne izvajajo, zato ni zagotovljeno pravilno delovanje. Sem spadata operacijska sistema ARM Linux in MIPS Linux, ki delujeta na procesorskih arhitekturah arm oz. mips.

V sklopu 3 pa so platforme, ki jih Rust sicer podpira, vendar ni zagotovljena zanesljivost delovanja. Podpora sloni predvsem na prispevkih odprtokodne skupnosti in se zna z vsako novo različico jezika spremeniti. Pogosto se zahteva tudi nekaj dodatnega dela za uspešno prevajanje na teh platformah. Tu npr. spadajo sistemi *BSD, iOS, Android in Solaris ter procesorska arhitektura ppc64 [32].

### 3.4.3 Hitrost izvajanja

Eden izmed glavnih ciljev načrtovalcev jezika Rust je bil hitrost izvajanja. Rust kot zaledni del svojega prevajalnika uporablja prevajalniško ogrodje LLVM, ki vsebuje visoko optimiziran prevajalnik in generira hitre in učinkovite izvršljive datoteke. K hitrosti pripomore tudi odsotnost avtomatičnega čistilca pomnilnika. Ker se večino napak, povezanih s pomnilnikom, ugotovi že med prevajanjem ali pa za to poskrbi prevajalnik, kar je opisano v podrazdelku 3.4.5, to posledično prinese manjše in hitrejše izvajalno okolje.

### 3.4.4 Funkcije prevajalnika

Optimizacijski koraki pri prevajanju so privzeto izklopljeni, kar pripomore k hitrejšemu prevajanju programov in posledično hitrejšemu razvoju. Vendar je bila ta odločitev deležna nekaj kritik znotraj Rust skupnosti, saj so se novi programerji, ki so prvič preizkušali Rust, pritoževali nad počasnim izvajanjem programov [33]. Vklop optimizacije ima v programskem jeziku Rust lahko velik vpliv na hitrost izvajanja. Rust podpira 4 nivoje optimizacije, kjer 0 pomeni izklopljeno optimizacijo in 3 najbolj agresivno. Prevajalniku se nivo optimizacije poda z zastavico `-C -opt-level=`.

Rust kot pravi sistemski jezik pozna uporabo medvrstičnih ukazov, kar podpira z uporabo makra `asm!`. A je zato treba uporabiti t. i. nočno gradnjo (angl. *nightly build*) prevajalnika rustc, saj v stabilni različici to še ni možno – sintaksa makra asm se lahko še spremeni, stabilna različica pa mora vedno zagotavljati kompatibilnost za nazaj. Makro omogoča tudi izbiro sintakse zbirnega jezika (Intel ali AT&T, ki je privzeta). Spodnji primer prikazuje operacijo *res=x-y* z uporabo medvrstičnih zbirnih ukazov v sintaksi AT&T:

```
unsafe {
    asm!("sub $2, $1"
         : "=r"(res)
         : "r"(x), "r"(y)
    );
}
```

### 3.4.5 Upravljanje s pomnilnikom

Rust je že od vsega začetka bil načrtovan kot sistemski jezik brez avtomatičnega čistilca pomnilnika. Kot glavni razlog navajajo dejstvo, da imajo jeziki z avtomatičnim čistilcem pomnilnika večje in kompleksnejše izvajalno okolje, Rust pa je načrtovan s ciljem, da deluje čim bližje strojni opremi in omogoča delovanje brez izvajalnega okolja in standardne knjižnice.

Sistem za upravljanje s pomnilnikom je v Rust zelo kompleksen [34], zato ga bomo opisali le površno. Rust spoštuje in uveljavlja programski idiom RAII (angl. *Resource acquisition in initialization*). Obstajata dva tipa kazalcev. Prvi tip so reference (angl. *references*), ki se imenujejo tudi pametni kazalci. Pametni zato, ker so zaščiteni pred pogostimi napakami pri delu s pomnilnikom, kot sta uhajanje pomnilnika in nestabilni kazalci (angl. *dangling pointer*). Četudi Rust nima avtomatičnega čistilca pomnilnika, pa varnost pomnilnika doseže že pri prevajanju. V kolikor prevajalnik ugotovi tako napako, reagira na dva načina, odvisno od napake – ali sporoči napako in se program ne prevede uspešno ali pa avtomatično generira ukaze, ki sprostijo pomnilnik. To naredi tako, da vstavi ukaze v zbirniku ali specifične inštrukcije ogrodja LLVM na mestih, kjer gre spremenljivka, ki kaže na prostor na kopici, izven dosega. Tako pri uporabi pametnih kazalcev ni potrebno eksplicitno sprostiti pomnilnika. Z vidika uporabnika je to podobno, kot to počne avtomatičen čistilec pomnilnika, le da se to preverja že pri prevajanju in ne med izvajanjem programa.

Primer funkcije, ki dodeli prostor na kopici za spremenljivko tipa int (Rust ji poleg tega še priredi vrednost 5), v programskih jezikih Rust (levo) in C (desno). V primeru jezika Rust prevajalnik avtomatično sprosti pomnilnik, medtem ko v C pride do uhajanja pomnilnika:

```
fn test() {
// allocate memory on heap
    let x = Box::new(5);
} // memory is freed
```

```
void test() {
    int *x = (int *)malloc(
        sizeof(int));
} // memory leak
```

Drugi tip kazalcev so surovi kazalci (angl. *raw pointers*) in so analogni tistim, ki jih poznamo iz programskega jezika C. Deferenciranje in referenciranje surovih kazalcev je isto kot v C z uporabo * in &, aritmetika s kazalci pa je možna z uporabo besede offset. Deferenciranje surovih kazalcev zahteva uporabo bloka unsafe, saj je v tem primeru programer odgovoren, da kazalec kaže na veljaven pomnilniški naslov.

### 3.4.6 Standardna knjižnica in izvajalno okolje

Za razliko od nekaterih drugih programskih jezikov ubereta standardna knjižnica in izvajalno okolje jezika Rust minimalistični pristop z namenom biti čim manjša. Vsebujeta le nujno potrebne programske konstrukte, primitivne tipe in knjižnice (npr. V/I knjižnica) za delo z jezikom. Tako v standardni knjižnici zaman iščemo podporo za regularne izraze, delo z dokumenti XML, naključno generiranje števil ali celo podporo za povezovanje s programskim jezikom C (zaboj libc). Vse te knjižnice so na voljo kot eksterni zaboji, ki jih po potrebi namestimo in vključimo v naše programe. Za čim lažje upravljanje z zaboji in njihovimi medsebojnimi odvisnostmi Rust nudi posebno orodje Cargo. Nekateri pomembnejši zaboji se bodo v prihodnosti vključili v standardno knjižnico, ko bodo dovolj stabilni in stestirani, to velja npr. za zaboj libc, drugi pa bodo vedno na voljo samo kot izbirni [35]. Zaboj std predstavlja standardno knjižnico in je avtomatično vključen v vse programe.

### 3.4.7 Povezovanje s programskim jezikom C

Zaboj libc predstavlja vmesnik za povezovanje s programskim jezikom C. Preko njega lahko dostopamo do C podatkovnih tipov (npr. size_t), C konstant (npr. ENOENT) in funkcij, deklariranih v zaglavnih datotekah standardne programske knjižnice C (npr. strlen). Tudi dostop do zunanjih oz. nestandardnih knjižnic jezika C je preprost. Funkcijo je potrebno definirati kot eksterno in zunanjo knjižnico, ki je lahko statična ali dinamična, vključiti pri prevajanju. Spodnji primer kaže primer klica dveh C funkcij: strlen, ki

je del standardne knjižnice jezika C in funkcije sum, katere implementacija v programski kodi C se nahaja v statični knjižnici, ki jo vključimo ob prevajanju programa:

```
extern crate libc;
use libc::*;
use std::ffi::*;

extern {
    fn sum(a:c_int, b:c_int) -> c_int; // sum.c -> libsum.a
}

fn main() {
    let s = CString::new("fooBar").unwrap();
    let len = unsafe { libc::strlen(s.as_ptr()) };
    let res = unsafe { sum(7,4) };
}
```

Ključna beseda extern omogoča povezovanje jezikov v obratni smeri – klic funkcije v jeziku Rust iz programske kode C. Primer deklaracije take funkcije, atribut no_mangle je potreben, da prevajalnik izklopi izmaličenje imen funkcij (angl. *name mangling*), kar bi zmotilo povezovalnik:

```
#[no_mangle]
pub extern fn get_string() -> *const u8 {
    "Rust string\0".as_ptr()
}
```

# Poglavje 4

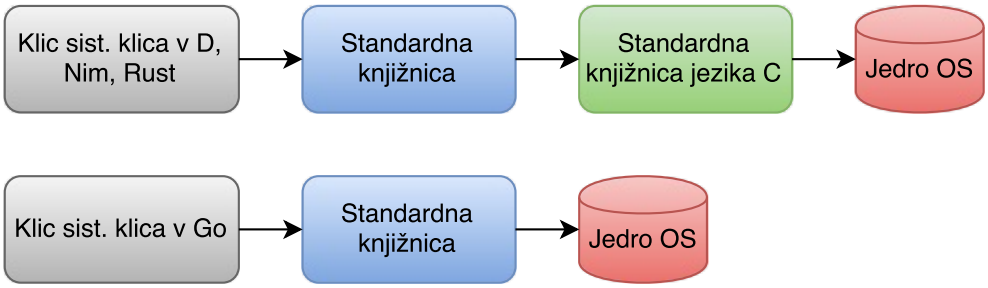
# Medsebojna primerjava po kriterijih

To poglavje vsebuje medsebojno primerjavo primerjanih jezikov glede na kriterije. Ugotavljamo, kje se med jeziki D, Go, Nim in Rust pojavljajo razlike, kje določen programski jezik nudi več od ostalih, in jezike ovrednotimo po posameznih kriterijih. V podrazdelkih 4.3 in 4.6 je smiselna kvantitativna obravnava, zato naredimo krajši eksperiment ter primerjamo rezultate.

## 4.1 Uporaba sistemskih klicev

Programski jeziki D, Nim in Rust izvajajo sistemske klice preko standardne knjižnice jezika C oz. njenih ovojnih funkcij. Posebnost je Go, ki ima implementiran svoj nabor sistemskih klicev in za njihovo izvedbo ne potrebuje standardne knjižnice jezika C. Razliko med koraki izvedbe prikazuje slika 4.1. Vsi štirje jeziki v svojih standardnih knjižnicah ponujajo ovojne funkcije večine sistemskih klicev popularnejših operacijskih sistemov, lahko pa se zgodi, da kakšen manjka, ali pa manjka ovojna funkcija v standardni knjižnici jezika C. V tem primeru sistemski klic izvedemo preko C rutine syscall oz. v primeru jezika Go funkcije RawSyscall.

Opomnimo, da zgoraj navedena dejstva glede jezika Go veljajo za privzeti




Slika 4.1: Koraki izvedbe sistemskih klicev v jezikih D, Nim, Rust in Go.

prevajalnik gc – prevajalnik gccgo izvaja sistemske klice preko standardne knjižnice C. Lahko ugotovimo, da so v primeru uporabe sistemskih klicev jeziki enakovredni med sabo in tudi z jezikom C.

## 4.2 Prenosljivost

Tabeli 4.1 in 4.2 prikazujeta podporo primerjanih programskih jezikov za nekatere bolj znane operacijske sisteme in procesorske arhitekture. Vsi trije poznajo prirojen prevajalnik, ki deluje na najpopularnejših operacijskih sistemih Linux, Windows in OS X. Prav tako vsi podpirajo procesorski arhitekturi i386 in amd64, ki sta danes del večine osebnih računalnikov ter arhitekturo arm, ki je pogosta predvsem v vgrajenih sistemih in mobilnih napravah. Razlike se pojavljajo pri manj znanih OS in procesorskih arhitekturah. Tam imata največ podpore Go in Nim, še posebej Go podpira nekatere bolj eksotične OS, kot so NetBSD, DragonFly BSD in Plan 9.

Primerjani programski jeziki so prenosljivi na nivoju izvorne kode in standardne knjižnice jezikov omogočajo pisanje programske kode neodvisno od platforme, na kateri bo tekel program. Iz vidika kriterija prenosljivosti lahko zaključimo, da velikih razlik med jeziki ni – čeprav Go in Nim ponujata polno podporo za več operacijskih sistemov in procesorskih arhitektur, pa D in Rust ponujata vsaj delno podporo za mnoge izmed njih.

| | D | Go | Nim | Rust |
|---|---|---|---|---|
| Linux | ✓✓ | ✓✓ | ✓✓ | ✓✓ |
| Windows | ✓✓ | ✓✓ | ✓✓ | ✓✓ |
| OS X | ✓✓ | ✓✓ | ✓✓ | ✓✓ |
| FreeBSD | ✓✓ | ✓✓ | ✓ | † |
| OpenBSD | | ✓ | ✓ | † |
| Solaris | ✓ | ✓ | ✓ | † |
| iOS | † | ✓ | ✓ | † |
| Android | ✓ | ✓ | ✓ | † |

Tabela 4.1: Podprtost operacijskih sistemov.

| | | |
|---|---|---|
| Legenda: | ✓✓ | za OS obstaja prirojen prevajalnik (angl. *native compiler*) |
| | ✓ | prevajalnik zna generirati kodo za ta OS (angl. *cross-compilation*) |
| | † | eksperimentalna/delna podpora |

| | D | Go | Nim | Rust |
|---|---|---|---|---|
| i386 | ✓ | ✓ | ✓ | ✓ |
| amd64 | ✓ | ✓ | ✓ | ✓ |
| arm | ✓ | ✓ | ✓ | ✓ |
| arm64 | † | ✓ | ✓ | † |
| ppc64 | † | ✓ | ✓ | † |
| sparc | | ✓ | ✓ | |
| mips | † | ✓ | ✓ | † |

Tabela 4.2: Podprtost procesorskih arhitektur.

| | | |
|---|---|---|
| Legenda: | ✓ | polna podpora |
| | † | eksperimentalna/delna podpora |

## 4.3 Hitrost izvajanja

Primerjava hitrosti računalniških programov je občutljivo opravilo, kar še posebej za velja mikroprimerjalne teste (angl. *microbenchmarks*), kjer se merijo izseki programske kode in ne aplikacija v resničnem svetu [37]. Splošna hitrost programskega jezika je odvisna od več faktorjev. Velja dejstvo, da so programski jeziki, katerih prevajalniki prevedejo izvorno kodo v izvršljivo kodo, praviloma hitrejši od interpretiranih programskih jezikov. Jeziki D, Go, Nim in Rust prevajajo neposredno v izvršljive datoteke. Hitrost je velikokrat odvisna tudi od prevajalnika in njegove sposobnosti generiranja učinkovitih programov. Prevajalniki, ki kot zaledni del uporabljajo enega že izmed obstoječih prevajalnikov, kot sta GCC ali LLVM, ki veljata za visoko optimizirana prevajalnika, so pogosto bolj uspešni v smislu generiranja hitre in učinkovite kode. Vsi štirje primerjani jeziki ponujajo vsaj en prevajalnik, ki kot zaledni del uporablja ali GCC ali LLVM, kar je razvidno iz tabele 4.6. Na spletu najdemo nekaj primerjalnih testov hitrosti, kjer so primerjani programski jeziki D, Go, Nim in Rust [38, 39, 40, 41]. Kratek povzetek omenjenih testov bi bil, da so programski jeziki D, Nim in Rust med seboj dokaj enakovredni in primerljivi s programskim jezikom C. Malo zaostaja le Go.

Hitrost primerjanih programskih jezikov smo preverili tudi sami in implementirali štiri programe:

- cat: Program, ki bere datoteko vrstico po vrstico in rezultat izpisuje na standardni izhod.
- copy: Program, ki bere datoteko v blokih po 4096 bajtov in le-te zapisuje v drugo datoteko.
- prime: Program, ki izračuna n-to praštevilo.
- arr: Program, ki dodeluje in sprošča bloke pomnilnika naključnih velikosti.

Prvi in drugi program sta poenostavljeni različici dveh sistemskih programov cat in copy na operacijskih sistemih tipa Unix. Tretji program testira hitrost gnezdenih zank, medtem ko četrti hitrost dodeljevanja in sproščanja

| Jezik | Prevajalnik | Različica |
|---|---|---|
| C | clang | 3.6.0 |
| C | gcc | 4.9.2 |
| D | DMD | 2.07.0 |
| D | GDC | 4.9.2 (2.066.0) |
| D | LDC | 0.17.0 (2.068.2) |
| Go | gc | 1.5.3 |
| Go | gccgo | 5.0.1 |
| Nim | nim | 0.13.0 |
| Rust | rustc | 1.6.0 |

Tabela 4.3: Različice prevajalnikov, uporabljene za primerjalni test.

pomnilnika. Primerjalni testi so bili opravljeni na OS Linux (Lubuntu 15.04), ki mu je bilo preko programa za virtualizacijo Oracle Virtual Box dodeljeno 2GB pomnilnika in dva jedra procesorja Intel i7-4600M. Uporabljene različice prevajalnikov so navedene v tabeli 4.3. Rezultati so v tabelah 4.4 in 4.5. Vnosi v tabeli pomenijo relativen čas glede na programski jezik C, torej C=1.

Tekom primerjalnih testov smo opazili velike razlike med različicami prevajalnikov. Nekaj decimalk višja različica prevajalnika je v nekaterih primerih prinesla tudi petkratno pohitritev (program arr in razlika med različicami prevajalnika DMD 2.0.66 in 2.0.70). To je razumljivo, saj gre za mlade programske jezike, ki se konstantno spreminjajo in razvijajo. Iz istega razloga verjetno pri omenjenem programu tako zelo odstopa prevajalnik GDC, saj je trenutna različica tega prevajalnika še vedno na različici jezika D 2.0.66.

V četrtem testu se pokaže moč jezika Rust in njegovega kompleksnega upravljanja s pomnilnikom ter je tako edini primer v naših testih, kjer primerjan programski jezik preseže hitrost jezika C. Rust in C imata v omenjenem testu prednost, saj ne vsebujeta čistilca pomnilnika. Zanimiva je primerjava odstotka pomnilnika, ki so si ga rezervirali jeziki med izvajanjem programa.

| Jezik | Prevajalnik | Rel. čas |
|---|---|---|
| C | clang | 1 (18s) |
| C | gcc | 1.01 |
| D | LDC | 1.03 |
| Nim | nim + clang | 1.17 |
| Rust | rustc | 1.17 |
| D | DMD | 1.18 |
| Nim | nim + gcc | 1.19 |
| Go | gc | 1.20 |
| Go | gccgo | 1.32 |
| D | GDC | 1.46 |

| Jezik | Prevajalnik | Rel. čas |
|---|---|---|
| C | clang | 1 (44s) |
| Nim | nim + clang | 1 |
| C | gcc | 1.01 |
| Rust | rustc | 1.01 |
| D | DMD | 1.02 |
| D | GDC | 1.02 |
| Nim | nim + gcc | 1.02 |
| D | LDC | 1.03 |
| Go | gc | 1.07 |
| Go | gccgo | 1.07 |

Tabela 4.4: Program cat (levo) in copy (desno).

| Jezik | Prevajalnik | Rel. čas |
|---|---|---|
| C | gcc | 1 (40s) |
| C | clang | 1 |
| D | GDC | 1.01 |
| D | LDC | 1.01 |
| Nim | nim + clang | 1.01 |
| Nim | nim + gcc | 1.01 |
| Rust | rustc | 1.11 |
| Go | gc | 1.12 |
| Go | gccgo | 1.12 |
| D | DMD | 1.23 |

| Jezik | Prevajalnik | Rel.čas | %RAM |
|---|---|---|---|
| Rust | rustc | 0.86 | 18.3 |
| C | clang | 1 (16s) | 16.3 |
| Go | gccgo | 1 | 38 |
| C | gcc | 1.01 | 16.3 |
| D | LDC | 1.07 | 35.9 |
| D | DMD | 1.08 | 36 |
| Nim | nim + gcc | 1.29 | 19.4 |
| Nim | nim + clang | 1.30 | 19.6 |
| Go | gc | 1.38 | 35.1 |
| D | GDC | 4.81 | 18.8 |

Tabela 4.5: Program prime (levo) in arr (desno).

Čeprav je gccgo enako hiter kot C, pa zato med izvajanjem porabi dvakrat več pomnilnika.

Na podlagi naših testov nedvoumnih zaključkov ne moremo dati. Razlike so majhne in se velikokrat razlikujejo bolj med različnimi prevajalniki znotraj enega programskega jezika, kot pa med samimi jeziki. Kar pa lahko ugotovimo na podlagi naših testov, je, da so programski jeziki D, Go, Nim in Rust hitri ter v večini primerov primerljivi z zmogljivostmi jezika C. Mogoče malo zaostaja le programski jezik Go.

## 4.4 Funkcije prevajalnika

Tabela 4.6 prikazuje uradne prevajalnike primerjanih sistemskih programskih jezikov in njihove zaledne dele. Opazimo, da imajo vsi jeziki vsaj en prevajalnik, ki kot zaledni del uporablja zbirki prevajalnikov GCC ali LLVM, ki veljata za visoko optimizirana prevajalnika in nudita veliko funkcij in različnih optimizacij pri prevajanju. Največ svobode nam ponujata D in Nim, ki ponujata oba, ob tem da ima D še lastni prevajalnik.

| Jezik | Lastni zaledni del | Zaledni del GCC | Zaledni del LLVM |
|---|---|---|---|
| D | DMD | GDC | LDC |
| Go | gc | gccgo | |
| Nim | | nim (preko gcc) | nim (preko clang) |
| Rust | | | rustc |

Tabela 4.6: Prevajalniki primerjanih jezikov.

Jeziki D, Nim in Rust omogočajo uporabo medvrstičnih zbirnih ukazov, medtem ko jih Go ne. Vsi trije programski jeziki, ki podpirajo medvrstične zbirne ukaze, omogočajo tudi izbiro sintakse zbirnega jezika na procesorski arhitekturi x86 (Intel ali AT&T). Pri D je to odvisno od izbire prevajalnika, pri jeziku Nim je to odvisno od izbire C prevajalnika, Rust pa omogoča

poljubno izbiro sintakse, kjer v sami programski kodi prevajalniku povemo, katero sintakso smo uporabili.

## 4.5 Upravljanje s pomnilnikom

Tabela 4.7 razdeli jezike glede na avtomatično in eksplicitno upravljanje s pomnilnikom. Jeziki D, Go in Nim imajo čistilec pomnilnika, ki pa ga je možno v D in Nim tudi izklopiti in upravljati s pomnilnikom samo ročno, kar je dobrodošla možnost, saj imamo tako izbiro. C programerji bodo najbolj domači v jeziku D, ki ima isto sintakso kot C za delo s kazalci. Rust nima čistilca pomnilnika in podobno kot C ponuja le ročno upravljanje z njim. Poleg tega napram ostalim primerjanim jezikom ponuja močnejši in kompleksnejši sistem za delo s pomnilnikom, kar je eden izmed glavnih prednosti in ciljev jezika Rust – doseči varnost in integriteto pomnilnika brez čistilca pomnilnika. Najmanj možnosti pri nizkonivojskem upravljanju s pomnilnikom nudi Go, kjer ni možno neposredno naslavljati pomnilnika.

| Jezik | avt. upr. s pom. | ekspl. upr. s pom. |
|---|---|---|
| D | ✓(opc.) | ✓ |
| Go | ✓ | |
| Nim | ✓(opc.) | ✓ |
| Rust | | ✓ |

Tabela 4.7: Možnosti upravljanja s pomnilnikom.

## 4.6 Standardna knjižnica in izvajalno okolje

Tabela 4.8 prikazuje primerjavo velikosti standardnih knjižnic v obliki statičnih knjižnic na operacijskem sistemu Linux in procesorski arhitekturi amd64. Najbogatejša s funkcionalnostmi je knjižnica jezika Go, ki je t. i. knjižnica z vključenimi baterijami (angl. *batteries-included library*). Bogato knjižnico

ima tudi jezik D, medtem ko Nim in Rust strmita h kompaktnosti in minimalističnemu pristopu.

| Jezik | lib*.a knjižnice |
|---|---|
| C | 3 (glibc) |
| D | 28 (LDC), 52 (GDC), 67 (DMD) |
| Go | 61 (gc), 62 (gccgo) |
| Nim | 6 |
| Rust | 8 |

Tabela 4.8: Velikost standardnih knjižnic v MB.

| Jezik | Prevajalnik | Minimum | Hello world |
|---|---|---|---|
| C | clang | 4.3 | 4.4 |
| C | gcc | 6.2 | 6.2 |
| Nim | nim + clang | 27 | 27 |
| Nim | nim + gcc | 55 | 55 |
| D | LDC | 170 | 223 |
| Rust | rustc | 292 | 300 |
| D | DMD | 371 | 547 |
| D | GDC | 346 | 1600 |
| Go | gccgo | 232 | 1800 |
| Go | gc | 713* | 1600* |

Tabela 4.9: Velikosti programov void main() {} in "Hello world"v KB.

Velikost izvršljivih datotek je pomembna v sistemih, kjer je prostor kritičnega pomena, npr. v vgrajenih sistemih. Tabela 4.9 prikazuje primerjavo velikosti dveh programov. Prvi primer programa je najmanjši možen program, napisan v določenem jeziku, ekvivalent programu void main() {} v

jeziku C. Drugi primer je dobro znani program, ki izpiše "Hello world". Programi so bili dinamično prevedeni v načinu za objavo (angl. *release mode*) oz. z najvišjim nivojem optimizacije (-O3), na njih pa smo s pomočjo orodja strip zaradi lažje medsebojne primerjave odstranili nepotrebne simbole, kot so npr. simboli za razhroščevanje. Tabela poda grobo razmerje velikosti izvajalnega okolja med primerjanimi programskimi jeziki. Daleč najboljši v tem segmentu je jezik Nim, ki zaradi svojega majhnega izvajalnega okolja in uporabe prevajalnika jezika C proizvede precej manjše izvršljive datoteke napram ostalim. Rust in D sta si dokaj enakovredna, pri čemer je pri D velikost zelo odvisna od izbire prevajalnika. Prevajalnika jezika Go proizvedeta največje izvršljive datoteke. Opomba: Privzeti prevajalnik gc pozna le statično prevajanje in je zato v tabeli napisan samo kot informacija, saj je nesmiselno primerjati dinamično in statično povezane programe.

## 4.7 Povezovanje s programskim jezikom C

Vsi štirje jeziki omogočajo povezovanje s programskim jezikom C. Le jezik Go ima manjšo omejitev. Namreč, če želimo poklicati Go funkcijo iz programske kode jezika C, je to možno le v primeru, da je vstopna točka (metoda main) v jeziku Go in ne C. Standardne knjižnice primerjanih jezikov ponujajo C podatkovne tipe in funkcije za pretvarjanje podatkovnih tipov, če je to potrebno.

# Poglavje 5

# Sklepne ugotovitve

Programski jeziki D, Go, Nim in Rust so nastali zaradi potrebe po boljšem C/C++. Nezmožnost varnega upravljanja pomnilnika v jeziku C in naraščajoča kompleksnost jezika C++ so bili glavni razlogi za nastanek novih jezikov. Ob tem hočejo v svet sistemskega programiranja, kjer že dolgo kraljuje C, pripeljati moderne programske konstrukte, poznane iz dinamičnih in sodobnih programskih jezikov.

V četrtem poglavju smo glede na definirane kriterije v drugem poglavju jezike medsebojno primerjali. Primerjani programski jeziki omogočajo neposredno uporabo sistemskih klicev, povezovanje s programskim jezikom C, primerjajo se lahko s hitrostjo jezika C in omogočajo podporo za večino popularnejših operacijskih sistemov in procesorskih arhitektur. Razlike se pojavljajo v možnostih uporabe medvrstičnih zbirnih ukazov, velikosti standardne knjižnice in izvajalnega okolja ter posledično velikosti izvršljivih datotek, in predvsem v upravljanju s pomnilnikom.

Programski jezik Go zaradi obveznega čistilca pomnilnika, nezmožnosti pisanja medvrstičnih zbirnih ukazov, majhnih možnosti nizkonivojskega upravljanja s pomnilnikom ter večjih izvršljivih datotek kot ostali, ni primeren za sistemsko programiranje. Čeprav je Go na začetku bil mišljen kot sistemski programski jezik, pa je v zadnjem času njegov cilj predvsem uporaba v spletnih storitvah, strežnikih in ostalih omrežnih aplikacijah. Zaradi bogate

standardne knjižnice in velikega števila funkcionalnosti je izmed primerjanih programskih jezikov najbolj primeren za aplikativno programiranje.

Jezika D in Nim omogočata omenjene stvari, ki jih Go ne. Ponujata opcijski čistilec pomnilnika, vendar neuporaba le-tega prinese tudi slabosti. Neuporaba čistilca pomnilnika zahteva drugačen slog kodiranja, saj lahko uporabimo le podmnožico jezika oz. standardne knjižnice, ki ne zahteva uporabe čistilca pomnilnika (npr. ne smemo uporabiti operatorja `new` oz. ne smemo uporabiti delov standardne knjižnice, kjer se uporablja omenjeni operator). To omeji funkcionalnosti programskega jezika, a obenem omogoča eksplicitno upravljanje s pomnilnikom. Razvijalci programskega jezika D trenutno delajo na tem, da bi standardna knjižnica delovala brez uporabe čistilca pomnilnika. V primeru, da v jezikih D ali Nim izklopimo čistilec pomnilnika, pa vseeno uporabljamo operator `new`, pride do uhajanja pomnilnika. To pa je nekaj, kar se v jeziku Rust ne more zgoditi.

Rust je bil razvit izključno za namen sistemskega programiranja. Je tudi edini programski jezik, ki doseže varnost pomnilnika brez uporabe čistilca pomnilnika, kar je svojevrsten dosežek. Primeren je za pisanje operacijskih sistemov in uporabo v vgrajenih sistemih. Žal pa ga njegova kompleksnost naredi manj privlačnega kot jezika D ali Nim. To smo ugotovili tudi tekom pisanja diplomskega dela, saj smo za pisanje vzorcev programske kode in primerjalnih testov v jeziku Rust porabili 2-3x več časa kot pri ostalih jezikih.

Sistemsko programiranje je širok pojem. V kolikor lahko preživimo s čistilcem pomnilnika ali pa s pomnilnikom upravljamo ročno in se zadovoljimo s podmnožico funkcionalnosti, ki jo nudita oba jezika, bosta D in Nim dovolj dobra. D ponuja bogatejšo standardno knjižnico z več funkcionalnostmi, Nim na drugi strani proizvede manjše izvršljive datoteke. Izbira med njima bo tudi v osebnih preferencah programerja. C programerji bodo najbolj domači v jeziku D, medtem ko bodo programerji, navajeni skriptnih jezikov, kot je npr. Python, dali prednost jeziku Nim. V ostalih primerih bo prva izbira Rust, ki pa zahteva veliko vloženega truda in časa. A zato tudi ponuja največ.

# Literatura


[1] Matthew Hertz, Emery D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management", Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA, October 16-20, 2005.

[2] Mooney, J.D., "Bringing Portability to the Software Process", Technical Report TR 97-1, Dept. of Statistics and Computer Science, West Virginia Univ., Morgantown, WV, 1997.

[3] Slovensko društvo Informatika, II. izdaja [Online]. Dosegljivo: http://www.islovar.org/izpisclanka.asp?id=8196 [Dostopano 3. 3. 2015].

[4] Comparison of C/POSIX standard library implementations for Linux [Online]. Dosegljivo: http://www.etalabs.net/compare_libcs.html. [Dostopano 4. 3. 2015].

[5] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Philippe Gerum, "Building embedded linux systems, 2nd edition", O'Reilly & Associates, Inc., Sebastopol, CA, 2008.

[6] A garbage collector for C and C++. Dosegljivo: http://www.hboehm.info/gc/ [Dostopano 3. 3. 2016].

[7] Steve McConnell, "Rapid Development: Taming Wild Software Schedules", Microsoft Press, WA, 1996.

[8] TIOBE Index for February 2016 [Online]. Dosegljivo: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html [Dostopano 8. 2. 2015].

[9] Andrei Alexandrescu, "The D Programming Language", Addison-Wesley Professional, 2010.

[10] The D Programming Language [Online]. Dosegljivo: http://wiki.dlang.org [Dostopano 3. 5. 2015].

[11] Adam D. Ruppe, "D Cookbook". Packt Publishing Ltd, 2014.

[12] Compilers [Online]. Dosegljivo: http://wiki.dlang.org/Compilers [Dostopano 12. 12. 2015].

[13] Real World Comparison, GC vs. Manual Memory Management [Online]. Dosegljivo: http://3d.benjamin-thaut.de/?p=20 [Dostopano 23. 5. 2015].

[14] A look at the D programming language [Online]. Dosegljivo: http://fgda.pl/post/8/a-look-at-the-d-programming-language [Dostopano 23. 5. 2015].

[15] D's Garbage Collector Problem [Online]. Dosegljivo: http://pointersgonewild.com/2014/09/09/ds-garbage-collector-problem/ [Dostopano 23. 5. 2015].

[16] Reddit - D's Garbage Collector Problem [Online]. Dosegljivo: https://www.reddit.com/r/programming/comments/2g03af /ds_garbage_collector_problem/ckent8c [Dostopano 23. 5. 2015].

[17] Language Reference - Garbage Collection [Online]. Dosegljivo: https://dlang.org/spec/garbage.html [Dostopano 23. 5. 2015].

[18] D-Programming-Deimos [Online]. Dosegljivo: https://github.com/D-Programming-Deimos/ [Dostopano 9. 7. 2015].

[19] The Go Programming Language - FAQ [Online]. Dosegljivo: https://golang.org/doc/faq [Dostopano 15. 9. 2015].

[20] The syscall package [Online]. Dosegljivo: https://golang.org/s/go1.4-syscall [Dostopano 15. 9. 2015].

[21] Package syscall [Online]. Dosegljivo: https://golang.org/pkg/syscall/ [Dostopano 15. 9. 2015].

[22] Ian Lance Taylor, "The Go frontend for GCC", Google, 2010.

[23] The Go Blog - Gccgo in GCC 4.7.1 [Online]. Dosegljivo: https://blog.golang.org/gccgo-in-gcc-471 [Dostopano 28. 10. 2015].

[24] Go 1.5 Release Notes [Online]. Dosegljivo: https://golang.org/doc/go1.5 [Dostopano 14. 10. 2015].

[25] Installing Go from source [Online]. Dosegljivo: https://golang.org/doc/install/source [Dostopano 14. 10. 2015].

[26] Nim Programming Language - General FAQ [Online]. Dosegljivo: http://nim-lang.org/question.html [Dostopano 3. 11. 2015].

[27] Nim Compiler [Online]. Dosegljivo: https://github.com/nim-lang/Nim [Dostopano 3. 11. 2015].

[28] Module system [Online]. Dosegljivo: http://nim-lang.org/docs/system.html [Dostopano 7. 2. 2016].

[29] c2nim User's manual [Online]. Dosegljivo: http://nim-lang.org/0.11.0/c2nim.html [Dostopano 3. 11. 2015].

[30] Passing c-array to nim [Online]. Dosegljivo: http://forum.nim-lang.org/t/1188 [Dostopano 23. 12. 2015].

[31] The Rust Programming Language - The Book [Online]. Dosegljivo: https://doc.rust-lang.org/book/ [Dostopano 11. 12. 2015].

[32] Platform support [Online]. Dosegljivo: https://doc.rust-lang.org/book/getting-started.html#platform-support [Dostopano 11. 12. 2015].

[33] Make rustc and cargo produce optimized binaries by default [Online]. Dosegljivo: https://github.com/rust-lang/rfcs/pull/967 [Dostopano 14. 12. 2015].

[34] Ownership [Online]. Dosegljivo: https://doc.rust-lang.org/book/ownership.html [Dostopano 18. 12. 2015].

[35] RFC 1242 is rust-lang crates [Online]. Dosegljivo: https://github.com/rust-lang/rfcs/blob/master/text/1242-rust-lang-crates.md [Dostopano 14. 12. 2015].

[36] Redox - Your Next(Gen) OS [Online]. Dosegljivo: http://www.redox-os.org/ [Dostopano 14. 12. 2015].

[37] Gil, J. Y., Lenz, K., Shimron, Y., "A Microbenchmark Case Study and Lessons Learned", ACM, 2011.

[38] Benchmarks of the longest path problem in various languages [Online]. Dosegljivo: https://github.com/logicchains/LPATHBench [Dostopano 3. 2. 2016].

[39] KMeans benchmark [Online]. Dosegljivo: https://github.com/andreaferretti/kmeans [Dostopano 3. 2. 2016].

[40] Some benchmarks of different languages [Online]. Dosegljivo: https://github.com/kostya/benchmarks [Dostopano 3. 2. 2016].

[41] Perlin noise benchmark [Online]. Dosegljivo: https://github.com/nsf/pnoiseh [Dostopano 3. 2. 2016].