GNU programerski alati

Aleksandar B. Samardžić
(`asamardzic@matf.bg.ac.rs`)

Matematički fakultet
Univerzitet u Beogradu

Verzija 1.1.0, septembar 2002.

Predgovor

Operativni sistem *UNIX* nastao je na prelazu između 60-tih i 70-tih godina u *AT&T Bell* istraživačkim laboratorijama kao produkt nastojanja grupe programera da za svoje potrebe naprave adekvatno radno okruženje, a pod uticajem operativnog sistema *Multics* na čijem je razvoju većina njih prije toga radila. Od ovakvog inicijalno skromno postavljenog cilja *UNIX* je vremenom izrastao u moćan i veoma raširen operativni sistem i tu svoju poziciju je zadržao do danas.

Postoji više razloga za veliku popularnost koju je ovaj operativni sistem stekao; za početno širenje *UNIX*-a svakako je najzaslužnija njegova implementacija u drugoj iteraciji na *C* programskom jeziku[1], čime je praktično postao prvi operativni sistem implementiran na višem programskom jeziku umjesto na asembleru. Ovo je omogućilo portovanje *UNIX*-a na veći broj arhitektura koje su tada bile u upotrebi, što je predstavljalo znatnu prednost u odnosu na postojeće operativne sisteme koji su bili striktno vezani za arhitekturu za koju su inicijalno bili razvijeni. Takođe, širenju popularnosti *UNIX*-a svakako je bitno doprinijelo to što ga je *AT&T* besplatno dijelio univerzitetima i drugim zainteresovanim stranama od kojih je pak zauzvrat dobijao zakrpe za postojeće programe, kao i sasvim nove programe kojima je onda proširivan bazni operativni sistem - duh slobodne cirkulacije rezultata programerskog rada od početka je bio sastavni dio *UNIX* kulture (sa svim pozitivnim i negativnim posledicama). Ipak, svakako da je najveći razlog uspjeha *UNIX*-a njegov izuzetan inicijalni dizajn; koncepti na kojima je *UNIX* izgrađen su izdržali probu vremena i omogućili mu da čak i sada, 30 godina nakon nastanka, bude tehnički superioran u odnosu na ostale operativne sisteme.

Zajedno sa ekspanzijom *UNIX*-a tokom sedamdesetih i početkom osamdesetih godina došlo je i do njegove fragmentacije, takođe i do negativnih prom-

[1] *C* jezik je upravo namjenski razvijen radi implementacije *UNIX* operativnog sistema

jena sa korisničke tačke gledišta u pogledu povoljnih uslova licenciranja koji su važili u vrijeme pojave ovog operativnog sistema. Nezadovoljstvo takvom situacijom navelo je grupu programera da 1984. pokrenu projekat kreiranja potpuno besplatnih verzija svih programa za koje se obično smatra da sačinjavaju *UNIX* operativni sistem. Većina tih programa zaista biva implementirana na ovaj način; čitav ovaj napor dobija naziv *GNU* projekat[2], pa se onda ti programi označavaju i imenom *GNU* programi.

Po prirodi stvari, prvi skup programa nastao na opisani način su bili programerski alati - oni su bili potrebni da bi se pomoću njih napravili ostali programi; takođe, u to vrijeme se u duhu starijih shvatanja o tome šta obuhvata operativni sistem smatralo da su ovi programi njegov sastavni dio, pa je i sa te strane bilo važno da se isti implementiraju. Većinu ovih alata će se pokazati veoma kvalitetnim, do te mjere da danas većina komercijalnih varijanti *UNIX*-a uključuje ove alate kao standardne. Ti programerski alati biće predmet razmatranja ovog teksta i u narednim poglavljima biće predstavljeni načini korišćenja za svaki od njih ponaosob. Opet po prirodi stvari, ovi alati su bili primarno usmjereni ka pisanju programa na *C* programskom jeziku; mada većina njih sada podržava ravnopravno i druge programske jezike, fokus ovog teksta biće na njihovoj primjeni u *C* programiranju. Inače, za svaki od tih programa postoji i detaljna originalna dokumentacija do koje se može doći `info` naredbom tako što se iza ove naredbe navede ime programa (npr. za *gcc* prevodilac naredbom `info gcc`). Jasno je da ovaj kratki tekst može predstaviti samo uzak podskup elementarnih osobina ovih inače veoma kompleksnih programa, kao i da je pisanje dužeg teksta koji bi udvajao postojeću dokumentaciju besmisleno. Zato *info* stranice treba konsultovati po svim pitanjima koja se tokom korišćenja ovih alata jave, a koja ne budu ovdje razmotrena.

Takođe treba reći i da je učinjen svaki napor da ovaj tekst bude primjenjiv na što širi broj *UNIX* varijacija, ali je ipak sve što u njemu stoji provjereno samo na *Slackware 8.1* distibuciji *Linux*-a i na *FreeBSD*-u *4.6*, tako da u slučaju da nešto od onog što je navedeno u tekstu ne radi na datoj varijaciji *UNIX*-a treba konsultovati za tu varijaciju specifičnu dokumentaciju.

[2]*GNU* je rekurzivna skraćenica od *GNU's Not UNIX* i ne znači ništa

Sadržaj

Poglavlje 1

GNU emacs editor

Prva komponenta koja je programeru potrebna u realizaciji bilo kakvog projekta svakako je editor za unos koda u odgovarajuće fajlove. *GNU emacs* je visoko konfigurabilni programerski editor sa posebnim režimima rada za veliki broj programskih jezika. *emacs* se pokreće naredbom:

```
emacs
```

nakon čega će biti kreiran novi prozor[1] za *emacs* program. *emacs* se, kao i sve ispravno pisane *X Windows* aplikacije, može konfigurisati preko tzv. resursa. Izgled *emacs* prozora po startovanju je prikazan na slici 1.1.

Glavninu prozora zauzima naravno površina namijenjena za unos teksta, pri vrhu prozora nalazi se meni, sa lijeve strane je *scroll-bar*, a pri dnu prozora nalaze se jedno iznad drugog statusna linija i tzv. *minibuffer* koji služi za unos dodatnih parametara komandi *emacs*-a. I pored toga što program sadrži meni i *scroll-bar* kao standardne elemente grafičkog korisničkog interfejsa, te što reaguje na određeni broj komandi mišem, *emacs* je zamišljen tako da se komande prevashodno zadaju sa tastature. Štaviše, radi što bržeg kucanja sve komande se mogu (i trebaju) zadavati bez pomjeranja ruku sa glavnog dijela tastature. Takav rad sa *emacs*-om je najefikasniji i zato treba od početka nastojati da se komande zadaju na ovaj način.

Za zadavanje komandi sa tastature obično se koriste tasteri *Ctrl* i *Alt* u kombinaciji sa nekim slovima. Ovi tasteri se u *emacs* dokumentaciji označavaju

[1] pretpostavka ovog teksta je da se radi u *X Windows* okruženju; naravno, kao i za sve druge programe koji se izvršavaju u posebnom prozoru zgodno je na kraj navedene naredbe za pokretanje dodati & da bi mogao da se nastavi rad u *shell*-u iz koga se *emacs* pokreće

Slika 1.1: Izgled *emacs* prozora.

sa `C` odn. `M` (npr. pritiskanje tastera *f* dok je pritisnut taster *Ctrl* se označava kao `C-f`, a pritiskanje istog tastera dok je pritisnut taster *Alt* kao `M-f`), pa će takva notacija i ovdje biti korišćena.

Prva komanda koju je potrebno znati u *emacs*-u je vjerovatno komanda za napuštanje programa i ona se aktivira sekvencom `C-x C-c` (ovo znači da prvo treba pritisnuti taster *x* dok je taster *Ctrl* uključen, a potom pritisnuti taster *c* opet dok je taster *Ctrl* uključen).

Nakon ove komande, naredni značajan skup komandi koje su neophodne pri editovanju teksta su komande za pomjeranje kroz tekst. Tekuća pozicija je, kao što je to uobičajeno kod unosa teksta, predstavljena kursorom, pa preciznije rečeno ove komande predstavljaju komande za pomjeranje kursora.

U tabeli 1.1 dat je pregled ovih komandi. Treba uočiti da se ovdje komande koje uključuju upotrebu *Ctrl* tastera uglavnom odnose na elementarne jedinice teksta (karaktere, linije), dok se komande koje uključuju upotrebu *Alt* tastera odnose na semantičke jedinice teksta (riječi, rečenice, paragrafi). Pored komandi nabrojanih u tabeli, na većini tastatura alternativno se mogu koristiti standardni tasteri za pomjeranje kursora (tasteri sa strelicama, tasteri *PgUp* i *PgDn* itd.); međutim kombinacije tastera navedene u tabeli omogućavaju da se komande izdaju bez pomjeranja ruku sa glavnog dijela tastature pa ih treba, kako je već sugerisano, što prije usvojiti. Isto važi i za pozicioniranje kursora korišćenjem miša - *scroll-bar* se može koristiti za tu namjenu[2]

Komanda	Efekat komande
`C-f`	pomjeranje kursora za jedan karakter unaprijed
`C-b`	pomjeranje kursora za jedan karakter unazad
`C-p`	pomjeranje kursora na prethodnu liniju
`C-n`	pomjeranje kursora na narednu liniju
`M-f`	pomjeranje kursora za jednu riječ unaprijed
`M-b`	pomjeranje kursora za jednu riječ unazad
`C-a`	pomjeranje kursora na početak linije
`C-e`	pomjeranje kursora na kraj linije
`M-a`	pomjeranje kursora na početak rečenice
`M-e`	pomjeranje kursora na kraj rečenice
`C-v`	pomjeranje kursora za jedan ekran unaprijed
`M-v`	pomjeranje kursora za jedan ekran unazad
`M-<`	pomjeranje kursora na početak teksta
`M->`	pomjeranje kursora na kraj teksta
`C-l`	pomjeranje teksta tako da linija u kojoj se nalazi kursor dođe na sredinu ekrana

Tabela 1.1: Naredbe za pomjeranje kursora u *emacs*-u.

Za većinu komandi u *emacs*-u se može zadati i broj ponavljanja ukoliko je potrebno da se ista komanda izvrši više puta. To se postiže komandom `C-u` nakon čega treba navesti broj ponavljanja i onda normalno komandu koja

[2] *emacs* prozor je implementiran korišćenjem *Athena* seta *widget*-a (kontrola od kojih se sastoji korisnički interfejs), kod kojeg se pomjeranje *thumb*-a na *scroll-bar*-u vrši tako što se klikne srednjim dugmetom miša na *thumb* i onda se povuče mišem držeći srednje dugme pritisnuto, dok se klik lijevim dugmetom miša uvijek prevodi u pomjeranje za jedan ekran unaprijed, desnim dugmetom miša za jedan ekran unazad, a srednjim dugmetom miša na pozicioniranje vrha *thumb*-a na tačku u kojoj se kliknulo

treba da bude izvršena više puta. Tako npr. ako se želi da se kursor pomjeri za 5 linija unaprijed, odgovarajuća sekvenca komandi bi glasila `C-u 5 C-n`. Sa druge strane, zadavanje ili izvršavanje neke komande se može u svakom trenutku prekinuti komandom `C-g`.

Unošenje teksta pomoću *emacs*-a se naravno vrši prosto tako što se ukucaju odgovarajući karakteri. Ukoliko se prilikom unosa teksta dođe do kraja linije na ekranu, dalje uneseni karakeri će biti prikazani u narednoj liniji pri čemu će na kraj prve linije biti stavljena obrnuta kosa crta (\) da označi da na tom mjestu zapravo nije kraj linije. Komanda za zadavanje broja ponavljanja se može koristiti i za unos karaktera - ako se iza ove komande i broja ponavljanja ne zada neka druga komanda već običan karakter, *emacs* će taj karakter unijeti u tekst zadati broj puta.

Za brisanje teksta u *emacs*-u postoji više naredbi. Brisanje karaktera koji se nalazi ispred kursora vrši se pomoću *Del* tastera. Karakter iza kursora se briše komandom `C-d`. Po analogiji, riječ ispred kursora se briše istovremnim pritiskom *Alt* i *Del* tastera, dok se riječ iza kursora briše `M-d` komandom. Slično, tekst od pozicije kursora do kraja linije se briše komandom `C-k`, dok se tekst od pozicije kursora do početka linije briše komandom `M-k`. Komanda `C-k`, ukoliko je linija prazna briše samu liniju. Za brisanje većih djelova teksta, treba pritisnuti istovremeno tastere *Ctrl* i *Space* na početku (odn. kraju) dijela teksta koji se želi obrisati i onda se pozicionirati na kraj (odn. početak) tog dijela teksta i tamo izvršiti komandu `C-w`. Ovako obrisani dio teksta može se u svakom trenutku povratiti naredbom `C-y`. Ako se želi na taj način povratiti neki dio teksta nakon koga je bilo još sličnih brisanja, treba nakon komande `C-y` izdavati komandu `M-y` sve dok se ne pojavi dio teksta koji se želi povratiti. Ukoliko se neki dio teksta ne želi obrisati na ovaj način, već samo sačuvati da bi kasnije bio ubačen na neko mjesto, onda na kraju (odn. početku) tog dijela teksta umjesto komande `C-w` treba zadati komandu `M-w`.

Poništavanje poslednje akcije (tzv. *undo*) u *emacs*-u se vrši komandom `C-x u` ili alternativno komandom `C-_`. Komande koje ne mijenjaju tekst se ne pamte, pa se ne mogu ni poništiti. Sama *undo* komanda takođe se pamti, tako da je za poništavanje njenog efekta potrebno prvo izdati neku komandu koja ne mijenja tekst (da se ne bi nastavilo sa poništavanjem efekta prethodnih komandi), npr. pomjeriti kursor na neki način, a zatim opet izdati *undo* komandu.

Otvaranje fajla u *emacs*-u se vrši komandom `C-x C-f` nakon čega u *minibuffer*-u treba zadati ime fajla. Pritom se može koristiti standardna *shell* ekspanzija

- ukoliko se ukuca početak imena fajla i potom pritisne *Tab* taster ime fajla će biti dopunjeno ukoliko je jedinstveno, a ukoliko nije u donjem dijelu *emacs* prozora biće izlistani svi fajlovi koji imaju takav početak imena (ova lista će sama nestati kada se ukuca puno ime fajla). *emacs* u načelu ne dozvoljava rad sa neimenovanim fajlovima tako da se kreiranje novog fajla vrši tako što se na isti način zada `C-x C-f` komanda i onda specificira ime koje će u ovom slučaju biti dato novom fajlu. Snimanje fajla se vrši komandom `C-x C-s`. Prilikom snimanja *emacs* pravi kopiju stare verzije fajla pod imenom koje je jednako originalom imenu sa karakterom ~ kao sufiksom. Takođe, tokom rada *emacs* automatski povremeno snima tekuću verziju fajla dodajući karakter `#` kao prefiks i sufiks imenu - ovaj fajl može da se iskoristi za oporavak dijela onog što je urađeno u slučaju eventualnog pucanja programa. Ime tekućeg fajla inače uvijek stoji u statusnoj liniji.

emacs naravno podržava rad sa više fajlova. Svakom otvorenom fajlu se dodjeljuje tzv. bafer; takođe u *emacs*-u postoje i neki baferi koji nisu pridruženi fajlovima. Lista tekućih bafera se može dobiti komandom `C-x C-b`. Ova komanda će podijeliti *emacs* prozor na dva potprozora[3]; da bi se oslobodilo drugog prozora treba pozicionirati kursor u onaj prozor u kome se želi ostati i izdati komandu `C-x 1`. Da bi se odjednom sačuvao sadržaj svih bafera koji su povezani sa fajlovima treba izdati komandu `C-x s`, nakon čega će za svaki od takvih bafera koji su editovani biti postavljeno pitanje da li se želi sačuvati njihov sadržaj u odgovarajućem fajlu.

Iz dosadašnjeg izlaganja bi trebalo da se može uočiti način na koji se zadaju komande sa tastature u *emacs*-u - za često korišćene komande koristi se kombinacija tastera *Ctrl* ili *Alt* sa nekim karakterom, dok se za ređe korišćene komande koristi komanda `C-x` koju prati neki karakter ili opet kombinacija tastera *Ctrl* i nekog karaktera. Obzirom da *emacs* sadrži mnogo veći broj komandi nego što ima kombinacija tastera, one najređe korišćene komande zadaju se punim imenom, tako što se nakon komande `M-x` u *minibuffer*-u otkuca ime odgovarajuće komande. Tako se npr. komanda za zamjenu teksta[4] može zadati tako što se prvo izvrši `M-x`, a onda u *minibuffer*-u otkuca ime komande `replace-string`. Prilikom kucanja imena komande može se (kao pri *shell* ekspanziji imena komandi) pritisnuti taster *Tab* i ime komande će biti dopunjeno do one tačke do koje je to ime jedinstveno. Po zadavanju komande treba pritisnuti taster *Enter* da bi komanda bila izvršena (komanda za zamjenu dijela teksta zahtijeva još i dva argumenta: tekst koji će biti zamjenem i tekst koji predstavlja zamjenu koje takođe treba zadati u

[3] ustvari su ti potprozori ono što se u *emacs* terminologiji naziva prozorom

[4] za ovu komandu inače postoji i skraćenica sa tastature

minibuffer-u).

Za pretraživanje u *emacs*-u se koriste komande `C-s` za pretraživanje unaprijed i `C-r` za pretraživanje unazad. Po izdavanju ovih komandi u *minibuffer*-u treba zadati i string koji se traži; pretraživanje će početi još dok se string zadaje. Za ponavljanje pretraživanja treba opet izdati komandu `C-s` odn. `C-r`. Pretraživanje utiče na poziciju kursora, tj. kursor se pri svakom pretraživanju pozicionira na mjesto na kome je pronađen dati string.

Kako je već rečeno, *emacs* prozor može u nekim situacijama biti podijeljen na više potprozora. Da bi se to uradilo dok se edituje neki fajl treba zadati komandu `C-x 2`. Nakon toga se prozor dijeli na dva potprozora i svaki od njih sadrži po jedan bafer koji odgovara datom fajlu. Ovi baferi se mogu nezavisno editovati, mada se najčešće ovo radi da bi se imao stalno na pogledu jedan dio fajla dok se edituje drugi njegov dio. Komandom `C-x o` se prelazi iz jednog u drugi potprozor. Dok se nalazi u nekom potprozoru, može se u njemu otvoriti i neki drugi fajl standardom komandom za otvaranje fajla `C-x C-f`.

Već je pomenuto da statusna linija u *emacs*-u sadrži ime fajla. Ispred imena se nalaze zvjezdice ukoliko je bafer modifikovan - kada se snimi sadržaj bafera u fajl ove zvjezdice nestaju. Na desnom kraju statusne linije nalazi se indikator trenutne pozicije u okviru teksta i to prvo kao broj linije sa prefiksom `L`, a potom kao procenat u odnosu na čitav tekst. Između imena fajla i ovih indikatora nalazi se oznaka *emacs* moda. U *emacs*-u se razlikuju glavni (*major*) i sporedni (*minor*) modovi; u jednom trenutku može biti aktivan samo jedan glavni mod, dok istovremeno može biti aktivno više sporednih modova. Oznaka u statusnoj liniji se odnosi na glavni mod. Glavni mod se određuje prema tipu teksta koji se unosi i *emacs* podržava veliki broj ovih modova koji se odnose na editovanje odgovarajućih vrsta fajlova - tako npr. postoji mod za pisanje *C* programa, mod za pisanje običnog teksta, mod za pisanje TeX dokumenata itd. U svakom od ovih modova *emacs* na odgovarajući način pomaže pri unosu teksta; recimo u *C* modu automatski uvlači nove linije i boji pojedine riječi prema sintaksi jezika, takođe nudi i mnoge druge mogućnosti o kojima će više riječi biti u ostalim poglavljima. Pored toga, *emacs* stavlja na raspolaganje i posebne komande za rad sa datom vrstom teksta - neke od ovih komandi su dostupne i preko posebne stavke u meniju koja se pojavljuje ispred *Help* stavke. Sporedni modovi se odnose na neke šire primjenjive načine obrade teksta koji se mogu uključivati po želji i kombinovati sa glavnim modom i drugim sporednim modovima. Jedan primjer ovakvog moda je tzv. *Auto Fill* mod koji automatski ubacuje novi red nakon određenog broja (70 u podrazumijevanom stanju) karaktera. Ovaj mod se uključuje `M-x auto-fill-mode` komandom. Pored ove komande, ovaj mod obuhvata i komande `C-x f` za zadavanje broja kolone nakon koje će se linije prelamati i `M-q` za prelamanje pasusa kada se isti edituje negdje u sredini.

emacs sadrži i veoma detaljnu dokumentaciju koja je korisniku na raspolaganju opet u vidu funkcija koje se pozivaju na standardni način. Tu je prije svega izuzetan *tutorial* za početnike koji se aktivira komandom `C-h t` - ovo poglavlje nije ništa drugo do prepričan ovaj *tutorial*. Pomoć u vezi neke *emacs* komande može se dobiti na razne načine. Ukoliko se izda komanda `C-h c`, a potom komanda za koju se traži pomoć biće odštampano ime komande. Ukoliko se izda komanda `C-h k`, pa onda komanda za koju se traži pomoć biće otvoren potprozor sa detaljnom dokumentacijom za datu komandu. Ako se pak izda komanda `C-h a` i nakon toga otkuca neki string, *emacs* dokumentacija će biti pretražena i u potprozoru će biti nabrojani svi njeni djelovi u kojima se dati string javlja.

emacs je visoko konfigurabilan program. Pored konfigurisanja standardnih parametara *X Windows* aplikacije koje je pomenuto na početku, svaki segment *emacs*-a koji se odnosi na rad sa tekstom se može konfigurisati. Obzirom da se ispod *emacs*-a zapravo nalazi *Lisp* interpreter, to se ova podešavanja izvode *Lisp* izrazima za dodjeljivanje vrijednosti određenim varijablama odn. za pozivanje funkcija. Sve komande koje se izdaju sa tastature se zapravo na kraju svode na izvršavanje ovakvih izraza. Od posebnog značaja su svakako trajna podešavanja, tj. ona podešavanja za koja se želi da budu na snazi svaki put kada se pokrene *emacs*. Ovakva podešavanja treba kao odgovarajuće *Lisp* izraze upisati u `~/.emacs` fajl. Tako npr. ako se želi postaviti podrazumijevani broj karaktera nakon koga će linija biti prelomljena u ovaj fajl treba staviti liniju:

```
(setq-default fill-column 78)
```

a ukoliko se želi da kada je na snazi *Text* glavni mod da uvijek bude aktivan i *Auto Fill* sporedni mod, u ovaj fajl treba dodati liniju:

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

Ako se npr. hoće da region teksta bude označen pri markiranju sa tastature na isti način kao što to biva kada se region markira mišem, odgovarajuća linija u `~/.emacs` fajlu bi glasila:

```
(setq-default transient-mark-mode t)
```

Za puno konfigurisanje svakako je potrebno upoznati se sa osnovama *Lisp*-a i detaljno proučiti *emacs* dokumentaciju koja se tiče ugrađenih promjenljivih i funkcija.

Na ovom mjestu bi bilo vrlo poželjno da se provježba sa *emacs*-om na primjeru iz narednog poglavlja - najbolje tako što će se problem opisan u prvom pasusu pokušati uraditi samostalno, bez gledanja na priloženo rješenje.

Poglavlje 2

Primjer - rješavanje kvadratne jednačine

Za demonstriranje *GNU* programerskih alata biće korišćen jednostavan primjer programa koji nalazi realne korijene kvadratne jednačine $ax^2 + bx + c$. U ovom poglavlju dat je izvorni kod po fajlovima koji sačinjavaju taj mali projekat. U duhu *UNIX* koncepta pravljenja malih programa koji primaju ulazne podatke sa standardnog ulaza u tekstulanom formatu i štampaju rezultate na standardnom izlazu opet u tekstualnom formatu, program će sa standardnog ulaza čitati koeficijente kvadratne jednačine a, b i c, a na standardnom izlazu ispisati broj realnih korijena te jednačine, kao i same korijene (ukoliko ih ima). Sa druge strane, u duhu proceduralne paradigme na kojoj je zasnovan *C* jezik, izračunavanje korijena jednačine biće izdvojeno u posebnu funkciju i poseban fajl u odnosu na glavni program koji sadrži kod za učitavanje ulaznih podataka odn. ispis rezultata.

Pretpostavićemo da je za ovaj projekat kreiran poseban direktorijum sa imenom `quadric`. Fajl u kome se nalazi funkcija za određivanje korijena kvadratne jednačine zvaće se `quadric.c` i imaće sledeći sadržaj:

```
#include <math.h>
#include "quadric.h"

int
quadric (double a, double b, double c, double *x0, double *x1)
{
  double discr;               /* Diskriminanta jednacine. */
  double mul;                 /* Reciprocna vrijednost imenioca u razlomku
                               * koji odredjuje rjesenja jednacine. */
```

```
  /*
   * Izracunava se diskriminanta jednacine.
   */
  discr = b * b - 4 * a * c;

  /*
   * Ukoliko je diskriminanta veca od 0, jednacina ima dva realna
   * rjesenja.  Izracunavaju se rjesenja jednacine i upisuju u lokacije
   * na koje ukazuju x0 i x1, a iz funkcije se vraca 2 kao broj realnih
   * rjesenja.
   */
  if (discr > 0)
    {
      discr = sqrt (discr);
      mul = 1 / (2 * a);
      *x0 = (-b + discr) * mul;
      *x1 = (-b - discr) * mul;
      return 2;
    }

  /*
   * Ukoliko je diskriminanta jednaka 0, jednacina ima jedno realno
   * rjesenje.  Izracunava se vrijednost tog rjesenja i upisuje u
   * lokaciju na koju ukazuje x0, a iz funkcije se vraca 1 kao broj
   * realnih rjesenja.
   */
  if (discr == 0)
    {
      *x0 = -b / (2 * a);
      return 1;
    }

  /*
   * Ukoliko je diskriminanta manja od 0 jednacina nema realnih
   * rjesenja, pa se iz funkcije vraca 0.
   */
  return 0;
}
```

Interfejs ove funkcije biće dat u `quadric.h` fajlu. Obzirom da onaj ko koristi funkciju treba da vodi računa samo o interfejsu (tj. u principu ne treba da uopšte gleda `quadric.c` fajl), u ovom fajlu je funkcija detaljno dokumentovana. Fajl ima oblik:

```
#ifndef quadric_h
#define quadric_h

/*
 * Funkcija quadric() odredjuje realne korijene kvadratne jednacine
 * a*x^2+b*x+c=0. Argumenti funkcije su:
 *   a, b, c - koeficijenti kvadratne jednacine
 *   x0, x1 - pokazivaci na lokacije u koje ce biti upisana rjesenja jednacine
 * Funkcija vraca broj pronadjenih realnih korijena. Ukoliko jednacina
 * nema realnih korijena, lokacije na koje pokazuju x0 i x1 ostaju
```

```
 * neizmijenjene.  Ukoliko jednacina ima jedan realni korijen, njegova
 * vrijednost biva upisana u lokaciju na koju pokazuje x0. Ukoliko
 * jednacina ima dva realna korijena, njihove vrijednosti bivaju
 * respektivno upisane na lokacije na koje pokazuju x0 i x1.
 */
int quadric (double a, double b, double c, double *x0, double *x1);

#endif
```

Na kraju, glavni program u kome se učitavaju koeficijenti jednačine sa standardnog ulaza, poziva `quadric` funkcija da izračuna realna rješenja jednačine i ispisuju ta rješenja na standardni izlaz biće u fajlu `test.c` (ime potiče od toga što zapravo ovaj dio koda služi za testiranje ispravnosti implementacije funkcije `quadric()`) i imaće oblik:

```
#include <stdio.h>
#include "quadric.h"

/*
 * Program pronalazi realna rjesenja date kvadratne
 * jednacine. Koeficijenti jednacine se ucitavaju sa standardnog ulaza,
 * da bi se potom na standardnom izlazu odstampao broj realnih rjesenja
 * jednacine, a zatim, ako jednacina ima realnih rjesenja, i sama
 * rjesenja. Program nema ugradjenu provjeru gresaka na ulazu.
 */
int
main ()
{
  double a, b, c;               /* Koeficijenti kvadratne jednacine. */
  int count;                    /* Broj realnih rjesenja jednacine. */
  double x0, x1;                /* Realna rjesenja jednacine. */

  /*
   * Ucitavaju se koeficijenti jednacine.
   */
  scanf ("%lf%lf%lf", &a, &b, &c);

  /*
   * Odredjuju se realna rjesenja jednacine.
   */
  count = quadric (a, b, c, &x0, &x1);

  /*
   * Ispisuje se broj realnih rjesenja jednacine.
   */
  printf ("%d\n", count);

  /*
   * Ispisuju se rjesenja jednacine.
   */
  switch (count)
    {
    case 1:
```

```
      printf ("%f\n", x0);
      break;

    case 2:
      printf ("%f\n", x0);
      printf ("%f\n", x1);
      break;
    }

  return 0;
}
```

Mada je za dalji tekst od najvećeg značaja ustvari opisana raspodjela projekta po fajlovima, poželjno je da se u ovom trenutku prouči i sam kod i pritom obrati posebna pažnja na dekompoziciju problema na potprobleme, te na detaljno komentarisanje i davanje deskriptivnih imena promjenljivim u nastojanju da se kod učini što čitljivijim. Sve nabrojano je naravno poželjno pri bilo kakvom programiranju, ali se smatra i obaveznim u *UNIX* zajednici.

Poglavlje 3

GNU indent program za formatiranje koda

U poglavlju o *emacs*-u je pomenuto kako programerski modovi automatski postavljaju pravilno uvlačenje prilikom unosa koda. Na sličan način *emacs* pomaže u vezi još nekih elementarnih formatiranja koda; međutim, ovaj editor ipak ne može sam po sebi obezbijediti potpuno uniformno formatiranje datog koda. Sa druge strane, uniformno formatiranje znatno popravlja čitljivost koda, tako da je obaveza u iole ozbiljnijem projektu obezbijediti da kod bude uniformno formatiran, kao i da to formatiranje bude usaglašeno sa nekim opšteprihvaćenim konvencijama. Naravno, pošto je tako nešto teško osigurati samo nastojanjem programera, pojavili su se alati koji za pojedine programske jezike vrše automatsko reformatiranje fajlova sa kodom da bi se kod u njima doveo u sklad sa željenom konvencijom. Ovakvi alati se označavaju imenom *beautifier*-i.

GNU alat koji je namijenjen za formatiranje *C* koda naziva se *indent*[1]. Upotreba ovog programa je vrlo jednostavna - u komadnoj liniji se iza imena programa navede ime fajla (ili imena fajlova) koje treba obraditi i *indent* će izvršiti reformatiranje, npr:

```
indent hello.c
```

Pritom, reformatirani kod će biti sačuvan u fajlu sa istim imenom, dok će originalni sadžaj fajla biti sačuvan na isti način kao što *emacs* čuva rezervne

[1]treba voditi računa da se na *BSD* varijacijama *UNIX*-a ovim imenom referencira originalni *indent* program, dok se *GNU* verzija komande može pozvati imenom *gindent*

kopije - u fajlu sa sufiksom `~`. Ako se kao argument navede samo jedan fajl, onda je moguće korišćenjem opcije `-o`[2] eksplicitno zadati ime fajla u koji će biti smješten reformatirani kod:

```
indent hello.c -o beautified.c
```

GNU indent ima jako veliki broj opcija (vidjeti odgovarajuće *info* stranice) koje omogućavaju da se podesi skoro svaki zamislivi aspekat formatiranja koda. Ipak, najčešće se koristi nekoliko predefinisanih stilova od kojih svaki podrazumijeva određenu kombinaciju opcija formatiranja. Podrazumijevani stil, koji se primjenjuje kada se ne navede nikakva opcija (kao u gornjim primjerima) je obično *GNU* stil formatiranja[3]. Ovaj stil je primijenjen na formatiranje fajlova iz prethodog poglavlja. Pored *GNU* stila koji se može i eksplicitno zadati opcijom `-gnu`, *indent* podržava i tzv. Kernigan-Riči stil (prema formatiranju koda u knjizi *Programski jezik C* od ovih autora) koji se zadaje opcijom `-kr`, npr.:

```
indent -kr hello.c
```

kao i tzv. Berkli stil koji se zadaje opcijom `-orig`. Ako se pored opcije za stil navede eksplicitno i neka pojedinačna opcija koja se razlikuje od opcije koju za odgovarajući aspekt formatiranja propisuje dati stil, onda pojedinačno zadata opcija ima prioritet i primjenjuje se njome određeno formatiranje.

[2] na *UNIX*-u se opcije programu obično zadaju tako što se navede crtica i slovo koje predstavlja skraćenicu za opciju; u ovom slučaju u pitanju je opcija sa argumentom (imenom fajla) koji se zadaje iza crtice i slova

[3] *GNU* stil, odnosno one njegovi aspekti koje editor može obezbijediti, primjenjuje i *emacs* za reformatiranje tokom editovanja

Poglavlje 4

GNU gcc C prevodilac

Centralno mjesto u paleti programerskih alata svakako zauzima *C* prevodilac; *GNU* verzija ovog programa jeste *gcc* prevodilac. Ukoliko se *C* program nalazi u jednom fajlu, npr. sa imenom `hello.c`, način pozivanja prevodioca je veoma jednostavan:

```
gcc hello.c
```

gcc će prevesti program i, ukoliko nema sintaksnih grešaka, kreiraće izvršni fajl; u suprotnom će prevodilac ispisati poruke o greškama zajedno sa referencama na linije u kojima se te greške javljaju u izvornom fajlu. *gcc* ima međutim veliki broj opcija kojima se mogu fino podešavati parametri prevođenja; najvažnije od njih biće pobrojane u daljem tekstu. Prva od takvih opcija jeste opcija za zadavanje imena izvršnog programa; ako se ova opcija ne zada, izvršni program će uvijek nositi ime `a.out`. Specificiranje imena izvršnog programa se vrši `-o` opcijom, pa ako se u gornjem slučaju želi da se taj fajl zove `hello`[1], odgovarajuća naredba bi glasila:

```
gcc -o hello hello.c
```

gcc predstavlja ustvari kolekciju prevodioca, trenutno su podržani *C*, *C++*, *Fortran 77* i *Objective C* programski jezici. Koji će prevodilac biti pokrenut određuje se na osnovu ekstenzije fajla, pa tako za fajlove sa ekstenzijom `.c` biva pokrenut *C* prevodilac, za fajlove sa ekstenzijama `.cc`, `.cp`, `.cxx`, `.cpp`, `.c++` i `.C` biva pokrenut *C++* prevodilac itd. Ukoliko ekstenzija fajla

[1] na *UNIX*-u je konvencija da izvršni fajlovi nemaju nikakvu ekstenziju

ne odgovara nekoj od navedenih, tada se prevodilac koji će biti pokrenut može zadati `-x` opcijom koja ima kao argument identifikator jezika za koji će biti pozvan prevodilac; ako je npr. *C* program iz nekog razloga stavljen u fajl sa imenom `hello.code`, onda bi naredba za njegovo prevođenje i davanje imena `hello` izvršnom fajlu glasila:

```
gcc -x c -o hello hello.code
```

Sledeća važna opcija odnosi se na podvrgavanje *C* standardu. *gcc*, kao i svaki drugi prevodilac, podržava neke nestandardne ekstenzije koje se isključuju zadavanjem opcije `-ansi`. Korišćenje nestandardnih ekstenzija je nešto što u načelu treba uvijek izbjegavati; međutim, kod *gcc*-a je u pitanju mali broj inače veoma korisnih ekstenzija, tako da u situacijama kada se zna da će program koji se razvija uvijek biti prevođen ovim prevodiocem ima smisla razmisliti o njihovom korišćenju. Od ovih ekstenzija treba pomenuti: `typeof` operator za određivanje tipa objekta, `complex` tip sa odgovarajućim funkcijama u standardnoj biblioteci za podršku radu sa kompleksnim brojevima, mogućnost deklarisanje polja čija dužina nije konstantna i koja će biti automatski oslobođena po izlasku iz dosega, mogućnost zadavanja inicijalizatora koji nisu konstantni, zadavanje ranga vrijednosti umjesto samo jedne vrijednosti u `case` granama `switch` naredbe i podršku za jednolinijske komentare u *C++* stilu (tj. za `//` komentare). Dobar dio *gcc* ekstenzija (npr. `complex` tip i ne-konstantni inicijalizatori) je ugrađen u poslednji, tzv. *C99* standard *C* jezika.

Kao i svaki drugi komapajler, i *gcc* pored poruka o sintaksnim greškama (u slučaju čijeg postojanja ne biva generisan izvršni fajl) može da generiše i poruke o upozorenjima na konstrukcije koje su sintaksno ispravne, ali liče na semantički neispravne[2]. Šta će sve biti prijavljeno kao upozorenje određuje se opcijama koje imaju prefiks `-W`. Najveći broj ovakvih konstrukcija na koje treba obratiti pažnju biće prijavljivan ako se prosto zada grupna opcija `-Wall`; treba međutim obratiti pažnju i na one konstrukcije koje nisu obuhvaćene ovom opcijom i po potrebi uključiti njima odgovarajuće opcije - jedna takva konstrukcija bila bi korišćenje realnih brojeva na nekoj strani jednakosti, za koju se prijavljivanje upozorenja uključuje opcijom `-Wfloat-equal`.

Svaki iole veći program u jednom trenutku treba debagovati od grešaka. *GNU* alat namijenjen ovom zadatku jeste *gdb* debager i o njemu će više

[2] tipičan primjer je postojanje operatora dodjele u uslovu, gdje bi bilo za očekivati operator jednakosti

riječi biti u jednom od narednih poglavlja; da bi debager međutim mogao da se koristi nad nekim izvršnim fajlom potrebno je da u taj fajl budu upisane odgovarajuće informacije. Ovo se prilikom prevođenja obezbjeđuje `-g` opcijom prevodioca. Najboje je tokom razvoja programa ovu opciju držati stalno uključenu i isključiti je tek pri generisanju finalne, od grešaka (koliko je to moguće) očišćene verzije programa.

Na način sličan `-g` opciji - u smislu da se njihovim uključivanjem u izvršni fajl upisuju dodatne informacije koje služe za neki vid analize izvršnog programa u periodu razvoja programa, funkcionišu i opcije koje predstavljaju podršku za profilaciju izvršnog programa. Više riječi o ovim opcijama biće u poglavlju posvećenom profilaciji.

Jednom kada je debagovanje programa završeno, mogu se uključiti opcije prevodioca namijenje optimizaciji[3]. Slično kao kod opcija koje se odnose na upozorenja, i ovdje se može vrlo precizno definisati kakve se sve optimizacije traže ili se može koristiti neka od grupnih opcija koja u sebi podrazumijeva veći broj elementarnih optimizacija. U ovu drugu grupu spadaju opcije `-O0`, `-O1`, `-O2` i `-O3`, svaka sa rastućim brojem uključenih optimizacija, kao i `-Os` kojom se minimizuje memorija koju će izvršni program da koristi.

Jedan podskup *gcc* opcija se odnosi na preprocesor. Dvije opcije koje su najvažnije u ovoj grupi su opcije za definisanje odnosno ukidanje definicije makroa. Prva opcija se navodi sa prefiksom `-D` koga slijedi ime makroa i opciono znak jednakosti i vrijednost koja se dodjeljuje makrou (u suprotnom se makrou dodjeljuje vrijednost 1). U čitavoj ovoj sekvenci ne smije biti blanko znakova. Druga opcija se navodi na sličan način - iza prefiksa `-U` stavi se ime makroa čija se definicija ukida.

gcc prevodilac ustvari predstavlja omotač za skup programa od kojih svaki obavlja pojedinu fazu prevođenja[4]. Stoga se jedan skup opcija ovog prevodioca odnosi na stepen prevođenja. Opcijom `-E` se specificira da se izvrši samo preprocesiranje, pri čemu se rezultat prikazuje na standardnom izlazu. Opcijom `-S` se zadaje da se prevođenje zaustavi nakon generisanje asemblerskog koda, tj. da se ne generiše mašinski kod. Asemblerski kod pritom biva smješten u fajl sa istim imenom kao fajl sa izvornim kodom, ali sa ekstenzijom `.s`. Sintaksa u kojoj će biti ispisan asemblerski kod je tzv. *AT&T* sintaksa; istu sintaksu koristi *as GNU* asembler, ali treba primijetiti da je u pitanju sintaksa koja se unekoliko razlikuje od sintakse koju uobičajeno

[3] ove opcije podrazumijevaju da kompilacija traje znatno duže nego što je uobičajeno u cilju generisanja efikasnijeg izvršnog koda

[4] ovdje se ubrajaju preprocesor, prevodilac u užem smislu, asembler i linker

koriste drugi asembleri. Najvažnija od opcija koje se odnose na nepotpuno prevođenje svakako je `-c` opcija kojom se zadaje da se generišu samo objektni fajlovi, tj. da se ne izvrši linkovanje. Pritom, objektni fajl dobija isto ime kao fajl sa originalnim kodom, s tim što se ekstenzija mijenja u `.o`. Ova opcija se standardno koristi za svaki projekat koji ima dva ili više fajlova sa izvornim kodom: prvo se svaki od ovih fajlova koristeći ovu opciju prevede u odgovarajući objektni fajl, a zatim se novim pozivom prevodioca izvrši linkovanje tih objektnih fajlova. Na primjeru programa za rješavanje kvadratne jednačine, generisanje objektnih fajlova bi se izvršilo naredbama:

```
gcc -ansi -Wall -c -o quadric.o quadric.c
gcc -ansi -Wall -c -o test.o test.c
```

dok bi se linkovanje izvršilo naredbom:

```
gcc -o test quadric.o test.o
```

Poslednja naredba će prijaviti poruku o grešci koja će biti pojašnjena u narednom pasusu. Ovdje treba uočiti kako kako su pri prevođenju korišćene opcije `-ansi` da se obezbijedi poštovanje *C* jezičkog standarda i `-Wall` da bi bila generisana upozorenja na određene sumnjive konstrukcije u kodu.

Od opcija koje se odnose na linker najvažnija je `-l` opcija kojom se navodi ime biblioteke sa kojom treba linkovati program. Korišćenje ove opcije se može pokazati na prethodnom primjeru; kako je već rečeno, naredba za linkovanje će prijaviti poruku o grešci. Razlog tome je što se u fajlu `quadric.c` poziva `sqrt()` funkcija iz standardne biblioteke, pa se onda mora linkovati `quadric.o` objektni fajl sa odgovarajućim kodom iz standardne biblioteke. Međutim, *gcc* za razliku od većine drugih prevodioca ne linkuje automatski sa čitavom standardnom bibliotekom, već su matematičke rutine iz standardne biblioteke izdvojene u posebnu biblioteku[5] koja se označava imenom `libm`[6]. Zato se mora eksplicitno zadati linkovanje sa tom bibliotekom da bi prevođenje u cjelini bilo uspješno izvršeno:

```
gcc -lm -o test quadric.o test.o
```

[5]ovo je iz razloga efikasnosti, na taj način je dužina izvršnog fajla kraća ako se ne koriste funkcije iz ovog dijela standardne biblioteke

[6]`lib` iz imena biblioteke se nikad na navodi kada se ime biblioteke specificira `-l` opcijom linkera

Poslednji interesantan skup opcija *gcc* prevodioca su opcije koje se odnose na dodatne direktorijume u kojima se traže *header* fajlovi ili biblioteke. *gcc* održava skup standardnih direktorijuma sa ovim fajlovima koje pretražuje tokom prevođenja odn. linkovanja (tu npr. spadaju direktorijumi u kojima se nalaze *header* fajlovi odn. fajlovi sa objektnim kodom standardne biblioteke), a dodatni direktorijumi se mogu zadati opcijama `-I` za *header* fajlove, odnosno `-L` za biblioteke. Nakon ovih opcija odmah se, bez blanko znakova, navodi putanja do direktorijuma koji se dodaje u skup direktorijuma koji se pretražuju za date fajlove.

Kako je već nagoviješteno, neke pomenute opcije *gcc* prevodioca biće detaljnije pojašnjene, a neke koje nisu pomenute biće uvedene u poglavljima posvećenim ostalim alatima za koje su te opcije važne.

Poglavlje 5

GNU make alat za održavanje projekta

U prethodnom poglavlju su navedene naredbe koje je potrebno izvršiti radi prevođenja opisanog programa za rješavanje kvadratne jednačine. Svaki put kada se nešto promijeni u kodu, ove naredbe treba ponovo primijeniti da bi se ažurirao izvršni fajl. Jasno je da uzastopno ponovno kucanje naredbi u *shell*-u nije previše pametna ideja, tako da bi prvi korak ka automatizaciji te procedure bio da se napravi *shell* skript koji bi sadržao te naredbe, pa da se onda rekompajliranje izvrši prostim pozivanjem ovog skripta. Ovo međutim nije dobro rješenje jer bi na taj način svaki put svi fajlovi bili iznova prevedeni, mada to često nije neophodno. Recimo u datom primjeru, ako se promijeni fajl `quadric.c` onda nema potrebe da se regeneriše fajl `test.o` i obrnuto; na ovako malom projektu usporenje ne bi bilo značajno, međutim na iole većem projektu gdje se u najvećem broju slučajeva rekompajliranje pokreće nakon izmjene samo jednog ili par fajlova, gubitak vremena bi bio osjetan. Sa druge strane, vraćajući se na pokretanje naredbi za prevođenje iz komandne linije, čak i ako se zanemari gubitak vremena na kucanju već na posmatranom primjeru se vidi da bi bilo previše komplikovano za programera da nakon svake izmjene određuje koje sve djelove projekta treba prevesti: tako bi ovdje programer morao stalno da vodi računa da kad promijeni fajl `quadric.c` treba da pokrene prvu i treću naredbu, kad promijeni fajl `test.c` da treba da pokrene prvu i drugu naredbu i kad promijeni fajl `quadric.h` da mora da pokrene sve tri naredbe (jer je ovaj fajl uključen i u `quadric.c` i u `test.c` fajlove). Očigledno je potreban neki alat koji će automatski određivati koje djelove projekta treba rekompajlirati; takvi alati se označavaju terminom *make* alati, a *GNU* varijanta ovog alata upravo i nosi ime *make*[1].

Da bi *make* mogao da obavi svoj posao, mora mu se zadati skup pravila na osnovu kojih se vrši ažuriranje fajlova koji zavise od nekih drugih fajlova. Ovaj skup pravila se upisuje u tzv. *makefile* fajl koji najčešće ima ime `Makefile` ili `makefile`, odn. `GNUmakefile` ako sadrži neke konstrukcije specifične za *GNU* verziju *make* alata. Pod pretpostavkom da se u tekućem direktorijumu nalazi *makefile* sa odgovarajućim imenom, pozivanje programa se vrši naredbom:

```
make
```

U suprotnom, ukoliko je potrebno eksplicitno navesti ime *makefile*-a bilo zato što se isti nalazi u nekom drugom direktorijumu ili što nema neko od podrazumijevanih imena, ime *makefile*-a se može zadati `-f` opcijom.

Pravila u *makefile*-u imaju striktan format. U prvoj liniji se navodi fajl (tzv. cilj odn. *target*) koga treba ažurirati kada se neki drugi fajlovi promijene, zatim slijedi dvotačka, a potom lista fajlova (tzv. zavisnosti odn. *dependencies*) od kojih je ovaj prvi zavisan. Potom slijedi proizvoljan broj redova koji sadrže komande za regenerisanje cilja na osnovu zavisnosti; svaki od ovih redova mora obavezno da počinje *Tab* znakom. Tako bi recimo na posmatranom primjeru pravilo za regenerisanje `quadric.o` fajla kada se promijene fajlovi od kojih je ovaj zavisan glasilo:

```
quadric.o: quadric.c quadric.h
	gcc -ansi -Wall -c -o quadric.o quadric.c
```

Obično *makefile* sadrži veći broj pravila; *make* pritom uvijek izvršava samo jedno pravilo i to ili ono koje je prvo u listi ako iza naredbe `make` nije ništa navedeno ili pak ono pravilo čiji je cilj naveden iza `make` naredbe. Ukoliko se u zavisnostima ovog cilja javi neki drugi cilj iz *makefile*-a onda se rekurzivno izvšava i pravilo koje se odnosi na taj drugi cilj i tako redom. Na taj način, u *makefile*-u za posmatrani primjer kao prvo bi trebalo navesti pravilo za generisanje samog izvršnog programa, a potom pravila za generisanje objektnih fajlova, pa bi *makefile* mogao imati oblik:

```
test: quadric.o test.o
	gcc -lm -o test quadric.o test.o
```

[1]slično *indent* programu, na *BSD* varijacijama *UNIX*-a se ovim imenom referencira originalna verzija ovog alata, dok se *GNU make* pokreće komandom *gmake*

```
quadric.o: quadric.c quadric.h
	gcc -ansi -Wall -c -o quadric.o quadric.c

test.o: test.c quadric.h
	gcc -ansi -Wall -c -o test.o test.c
```

Pod pretpostavkom da je *make* program pokrenut bez argumenata, biće analizirano prvo pravilo. Prvi elemenat iz liste zavisnosti je cilj drugog pravila, pa će *make* preći na analizu drugog pravila. Ovdje su zavisnosti obični fajlovi, pa će *make* odrediti (prostim poređenjem datuma zadnje izmjene fajla koji predstavlja cilj i fajlova koji predstavljaju zavisnosti) da li je potrebno ažurirati cilj i eventualno to ažuriranje uraditi pokretanjem komande koja je navedena u drugom pravilu. *make* se nakon toga vraća na analizu prvog pravila, ustanovljava da je i drugi element liste zavisnosti cilj i prelazi na analizu odgovarajućeg (trećeg) pravila. Ovo pravilo se obrađuje na isti načina kao drugo pravilo, ako je potrebno regeneriše se fajl `test.o` i *make* se opet vraća na analizu prvog pravila. Ovaj put sve zavisnosti su već ažurirane tako da se sada mogu porediti vremena njihovih zadnjih izmjena sa vremenom zadnje izmjene cilja prvog pravila i eventualno pokrenuti komanda navedena u prvom pravilu da bi se izvršni fajl ažurirao. Nakon toga *make* završava rad, a svi fajlovi u projektu ostaju u ažuriranom stanju.

Mada je gornji *makefile* potpuno funkcionalan, on se može učiniti znatno čitljivijim i lakšim za održavanje korišćenjem promjenljivih. Promjenljive se u *makefile*-ovima uvode prosto tako što se navede ime promjenljive, potom znak jednakosti i onda vrijednost koja se dodjeljuje promjenljivoj. Referenciranje promjenljive se vrši tako što se ime promjenljive navede između običnih zagrada, sa znakom `$` kao prefiksom. U datom primjeru bi se mogla uvesti jedna promjenljiva da čuva flegove `-ansi -Wall` koji se prenose *C* prevodiocu. Takođe, može se uvesti jedna promjenljiva da čuva listu biblioteka sa kojima treba linkovati izvršni program, kako bi eventualno dodavanje biblioteka u tu listu bilo pojednostavljeno. Pored toga, obično se i sam prevodilac referencira preko promjenljive zato što razni prevodioci na *UNIX*-u uglavnom rade na isti način, ali imaju drugačija imena pa se onda postiže da se prostom izmjenom vrijednosti ove promjenljive isti *makefile* može koristiti i sa nekim drugim prevodiocem. Sa navedenim izmjenama, *makefile* bi dobio oblik:

```
CC     = gcc
CFLAGS = -ansi -Wall
LDLIBS = -lm

test: quadric.o test.o
```

```
        $(CC) $(LDLIBS) -o test quadric.o test.o

quadric.o: quadric.c quadric.h
        $(CC) $(CFLAGS) -c -o quadric.o quadric.c

test.o: test.c quadric.h
        $(CC) $(CFLAGS) -c -o test.o test.c
```

U toku pisanja komandi za neko pravilo, na raspolaganju su određene specijalne promjenljive; tako promjenljiva `$@` predstavlja cilj pravila, promjenljiva `$^` sadrži listu svih zavisnosti pravila, a promjenljiva `$<` sadrži prvu iz liste zavisnosti. Koristeći ove promjenljive, može se gornji *makefile* kompaktnije zapisati u obliku:

```
CC     = gcc
CFLAGS = -ansi -Wall
LDLIBS = -lm

test: quadric.o test.o
        $(CC) $(LDLIBS) -o $@ $^

quadric.o: quadric.c quadric.h
        $(CC) $(CFLAGS) -c -o $@ $<

test.o: test.c quadric.h
        $(CC) $(CFLAGS) -c -o $@ $<
```

Određivanje liste fajlova od kojih zavisi neki fajl i održavanje te liste je u iole većem projektu veoma naporno; zato je mnogo bolje taj posao prepustiti nekom alatu. Većina modernih prevodilaca podržava kreiranje liste zavisnosti za dati fajl; u *gcc*-u se ta lista generiše opcijama `-M` za listu svih zavisnosti uključujući i *header* fajlove iz standardne biblioteke odnosno `-MM` za listu zavisnosti bez standardne biblioteke. Opcije `-MD` odnosno `-MMD` su ekvivalentne pomenutim, s tim što listu zavisnosti ne ispisuju na standardni izlaz već je smještaju u fajl sa istim imenom kao izlazni fajl[2], ali sa ekstenzijom `.d`. Ovo se može iskoristiti da se u *makefile* automatski uključe ove liste zavisnosti i time izbjegne potreba za njihovim ručnim održavanjem. Da bi se to uradilo potrebno je prilikom generisanja objektnih fajlova zadati opciju `-MMD`[3] *gcc* prevodiocu, pa će liste zavisnosti biti kreirane i smještene u odgovarajuće fajlove sa ekstenzijom `.d`. Potom treba iskoristiti `include` direktivu da bi se ovi fajlovi uključili u *makefile*, a onda se mogu izbaciti *header* fajlovi iz liste zavisnosti za objektne fajlove i *makefile* dobija oblik:

[2] ako takvog ima, ako je pak zadata samo ova opcija onda u fajl sa istim imenom kao ulazni fajl

[3] pretpostavlja se da se fajlovi iz standardne biblioteke na datom sistemu neće mijenjati, te da ih nije potrebno navoditi u listi zavisnosti

```
CC     = gcc
CFLAGS = -ansi -Wall
LDLIBS = -lm

test: quadric.o test.o
        $(CC) $(LDLIBS) -o $@ $^

quadric.o: quadric.c
        $(CC) $(CFLAGS) -c -o $@ -MMD $<

test.o: test.c
        $(CC) $(CFLAGS) -c -o $@ -MMD $<

-include quadric.d test.d
```

Direktiva `include` se uvijek izvršava prije obrade pravila, tako da pravila navedena u fajlovima uključenim u *makefile* ovom direktivom bivaju tretirana kao da su se od početka nalazila u *makefile*-u. Ispred `include` direktive je stavljen znak `-` koji u *makefileu*-u označava da treba nastaviti sa radom čak i ako pri izvršavanju date komande odn. direktive dođe do greške. Prilikom prvog pokretanja *make*-a nad ovim *makefile*-om fajlovi sa ekstenzijom `.d` ne postoje, tako da bi u slučaju da znak `-` nije stavljen ispred `include` direktive odmah bila prijavljena greška i *makefile*-e se uopšte ne bi mogao izvršiti.

Lista zavisnosti koji bude kreirana recimo za fajl `quadric.c` zbog uključene `-MMD` opcije *gcc*-a ima oblik:

```
quadric.o: quadric.c quadric.h
```

Dakle, u pitanju je zapravo još jedno pravilo (sa praznom listom komandi) koje ima `quadric.o` kao cilj. Kada postoji više pravila sa istim ciljem, *make* ih povezuje u jedno pravilo koje u listi zavisnosti ima uniju svih zavisnosti koje se javljaju u listama zavisnosti pojedinačnih pravila. Važno je samo da pritom komande postoje samo u jednom pravilu, kako je ovdje i slučaj. Na taj način se postiže upravo ono što je potrebno - da se ne moraju zavisnosti navoditi ručno već da budu automatski generisane dok će u slučaju da je potrebno izvršiti regenerisanje cilja biti pokrenute odgovarajuće komande kao i ranije. U vezi sa ovim automatskim generisanjem lista zavisnosti potrebno je napraviti još jednu korekciju u gornjem *makefile*-u, ali će ista biti objašnjena nakon što *makefile* bude dalje pojednostavljen.

U poslednjoj verziji *makefile*-a se može primijetiti da su pravila za generisanje objektnih fajlova osim imena fajlova u svemu drugom identična. Ova pravila se mogu objediniti ukoliko se upotrijebi operator `%` uparivanja proizvoljnog broja karaktera u riječima i *makefile* se može napisati na sledeći način:

```
CC     = gcc
CFLAGS = -ansi -Wall
LDLIBS = -lm

test: quadric.o test.o
	$(CC) $(LDLIBS) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ -MMD $<

-include quadric.d test.d
```

Kada se ovdje u prvom pravilu obrađuje neki od objektnih fajlova, on se pronalazi kao cilj drugog pravila (jer se njegovo ime završava na `.o`), a onda se u listi zavisnosti traži fajl sa istim imenom i ekstenzijom `.c` i po potrebi vrši ažuriranje objektnog fajla odgovarajućim komandama.

Može se primijetiti da se slična lista fajlova pojavljuje na dva mjesta u poslednoj verziji *makefile*-a. Obično se u *makefile*-ovima imena fajlova koji sačinjavaju projekat navode samo na jednom mjestu, najčešće u nekoj promjenljivoj na početku fajla), a onda se koriste neke od moćnih komandi za substituciju da se ta imena generišu po potrebi u ostatku *makefile*-a. U ovom slučaju to bi moglo da se uradi običnom komandom za substituciju stringa u sadržaju promjenljive na sledeći način:

```
SOURCES = quadric.c test.c
CC      = gcc
CFLAGS  = -ansi -Wall
LDLIBS  = -lm

test: $(SOURCES:.c=.o)
	$(CC) $(LDLIBS) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ -MMD $<

-include $(SOURCES:.c=.d)
```

Ovom komandom se prosto iza imena promjeljive i dvotačke navodi string koji se mijenja, a potom se stavlja znak jednakosti i navodi string koji predstavlja zamjenu. Svaka pojava prvog stringa u sadržaju promjenljive potom biva zamijenjena drugim stringom i takav izmijenjeni sadržaj predstavlja vrijednost promjenljive u datom slučaju.

Zajedno sa objektnim fajlovima i izvršnim fajlom, kao i fajlovima koji sadrže liste zavisnosti, broj fajlova u direktorijumu projekta počinje da raste. Da ne bi prilikom manipulacije ovim fajlovima greškom došlo do brisanja ili oštećenja fajlova sa kodom, ponekad je zgodno te fajlove izdvojiti u poseban poddirektorijum. Međutim, nakon ovoga bi trebalo dodati ime tog poddirektorijuma ispred svakog pravila u *makefile*-u. Da bi se to izbjeglo, može se iskoristiti specijalna promjenljiva koja se zove `VPATH`. Kada *make* analizira *makefile*, fajlovi koji su navedeni u listi zavisnosti se traže u tekućem direktorijumu ili u direktorijumima navedenim u promjenljivoj `VPATH`, tako da je nakon premještanja fajlova sa kodom u neki poddirektorijum dovoljno postaviti odgovarajuću vrijednost za ovu promjenljivu pa da *makefile* bude obrađen na isti način kao i ranije. Ovakav pristup organizaciji projekta neće biti dalje razrađivan u ovom tekstu, ali ako bi npr. fajlovi sa kodom u posmatranom primjeru bili prebačeni u poddirektorijum `src` tada bi bilo dovoljno u *makefile*-u na kraj liste promjenljivih staviti:

```
VPATH   = src
```

da bi ovi fajlovi bili pronađeni, a onda bi prevođenje teklo na isti način kao i ranije.

Već je pomenuto da se sa gornjim načinom automatskog generisanja zavisnosti može pojaviti jedan problem. Radi se o tome da kada se neki od fajlova sa kodom obriše ili mu se promijeni ime, *make* neće biti u stanju da prevede projekat, jer će taj fajl i dalje da stoji u listi zavisnosti, a neće biti nikakvog pravila da se isti regeneriše. Rješenje ovog problema jeste da se za svako automatski generisano pravilo o zavisnosti doda još jednom pravilo u kome se zavisnosti javljaju kao ciljevi, a lista njihovih zavisnosti (kao i lista komandi za takvo pravilo) je prazna. Kada *make* naiđe na praznu listu zavisnosti u nekom pravilu, smatra se da komande koje sadrži to pravilo uvijek treba da budu primijenjene. Pošto u ovom slučaju nema komandi, smatraće se da je cilj ažuriran i, pošto je pronađen, *make* će biti u stanju da nastavi sa daljom analizom. Generisanje opisanih pravila treba da se vrši zajedno sa automatskim generisanjem liste zavisnosti i najjednostavniji način da se to uradi jeste da se, po generisanju liste zavisnosti u fajlu sa ekstenziojom `.d`, uzme takav fajl i od njega napravi drugi fajl, recimo sa ekstenzijom `.P` koji će sadržati pravilo koje se nalazi u prvom fajlu, kao i još jedno pravilo u kome će lista zavisnosti iz prvog pravila biti cilj, a ostatak pravila će biti prazan. Onda treba taj fajl sa ekstenzijom `.P` uključiti u originalni *makefile* umjesto fajla sa ekstenzijom `.d`. Navedeni postupak se može implementirati tako što će u listi komandi za generisanje objektnih fajlova (kom prilikom će biti generisani i fajlovi sa ekstenzijom `.d`) biti dodata komanda koja koristi program *sed*[4] da automatski edituje fajl sa ekstenzijom `.d` u opisanom smislu

[4] tzv. *stream* editor

i da od njega kreira fajl sa ekstenzijom `.P` koji će onda biti uključen u *makefile*. *Makefile* bi sa ovom komandom i uključivanjem fajlova sa ekstenzijom `.P` umjesto fajlova sa ekstenzijom `.d` imao oblik:

```
SOURCES = quadric.c test.c
CC      = gcc
CFLAGS  = -ansi -Wall
LDLIBS  = -lm

test: $(SOURCES:.c=.o)
	$(CC) $(LDLIBS) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ -MMD $<
	@cp $*.d $*.P; \
		sed -e 's/#.*//' -e 's/^[^:]*: *//' -e 's/ *\\$$//' \
		-e '/^$$/ d' -e 's/$$/ :/' <$*.d >>$*.P; \
		rm -f $*.d

-include $(SOURCES:.c=.P)
```

U dodatoj komandi, drugoj u skupu komandi koje se odnose na ažuriranje objektnih fajlova (treba uočiti da se ova komanda prostire na 4 reda - u *makefile*-ovima je \ oznaka za nastavljanje tekuće linije u narednoj liniji), prvo se fajl sa ekstenzijom `.d` kopira u fajl sa ekstenzijom `.P`, zatim se pokreće *sed* da bi se u fajl sa ekstenzijom `.P` dodalo nedostajuće pravilo i na kraju se briše fajl sa ekstenzijom `.d`. Znak `@` na početku komande označava da komanda ne treba da se odštampa prilikom izvšavanja. U komandi je korišćena i specijalna promjenljiva `$*` koja predstavlja ime fajla u stringu uparenom operatorom `%`. Način na koji se poziva *sed* neće biti ovdje objašnjavan jer je u pitanju veoma kompleksan program; ipak, treba pomenuti da sve ove naizgled čudne sekvence karaktera predstavljaju regularne izraze - kako se regularni izrazi inače javljaju u velikom broju važnih *UNIX* komandi, a njihova sintaksa je unificirana, to bi bilo dobro potruditi se da se ista iz *info* stranica odgovarajućih programa nauči.

Kao što se moglo naslutiti iz metode za ispravku problema sa automatskim generisanjem lista zavisnosti u slučaju brisanja nekog fajla iz te liste, ciljevi u pravilima *makefile*-a ne moraju biti fajlovi; kao što je tamo rečeno, ukoliko cilj nekog pravila nije fajl, onda će se komande koje se odnose na njegovo ažuriranje uvijek izvršiti, pa je to zgodan način da se u *makefile* uključe i sve one komande koje služe za održavanje projekta mimo samog prevođenja. Važno je samo da se takvi ciljevi navedu u listi zavisnosti specijalnog `.PHONY` cilja, čime se izbjegava konflikt u situaciji da se eventualno u datom direktorijumu ipak pojavi fajl sa imenom takvog cilja. Pretpostavimo da u datom *makefile*-u hoćemo da dodamo ciljeve `clean` za brisanje svih fajlova generisanih tokom prevođenja, `beauty` za pokretanje *beautifier*-a i `dist` za kreiranje arhive projekta - tada bi *makefile* imao oblik (uz dodate još neke promjenljive radi bolje čitljivosti i lakše izmjene strukture projekta):

```
PROBLEM = quadric
PROGRAM = test
SOURCES = $(PROBLEM).c $(PROGRAM).c
CC      = gcc
CFLAGS  = -ansi -Wall
LDLIBS  = -lm

$(PROGRAM): $(SOURCES:.c=.o)
        $(CC) $(LDLIBS) -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ -MMD $<
        @cp $*.d $*.P; \
                sed -e 's/#.*//' -e 's/^[^:]*: *//' -e 's/ *\\$$//' \
                -e '/^$$/ d' -e 's/$$/ :/' <$*.d >>$*.P; \
                rm -f $*.d

-include $(SOURCES:.c=.P)


.PHONY: beauty clean dist

beauty:
        -indent *.h *.c
        -rm *~

clean:
        -rm *.o *.P $(PROGRAM)

dist: beauty clean
        -tar -czv -C .. -f ../$(PROBLEM).tar.gz $(PROBLEM)
```

Sada se propuštanje koda kroz *beautifier* može prosto uraditi naredbom:

```
make beauty
```

Na sličan način, jednostavno i bez ponovnog kucanja dugačkih komandi, mogu se obrisati fajlovi generisani prevođenjem, odnosno kreirati arhiva projekta komandama:

```
make clean
```

odnosno:

```
make dist
```

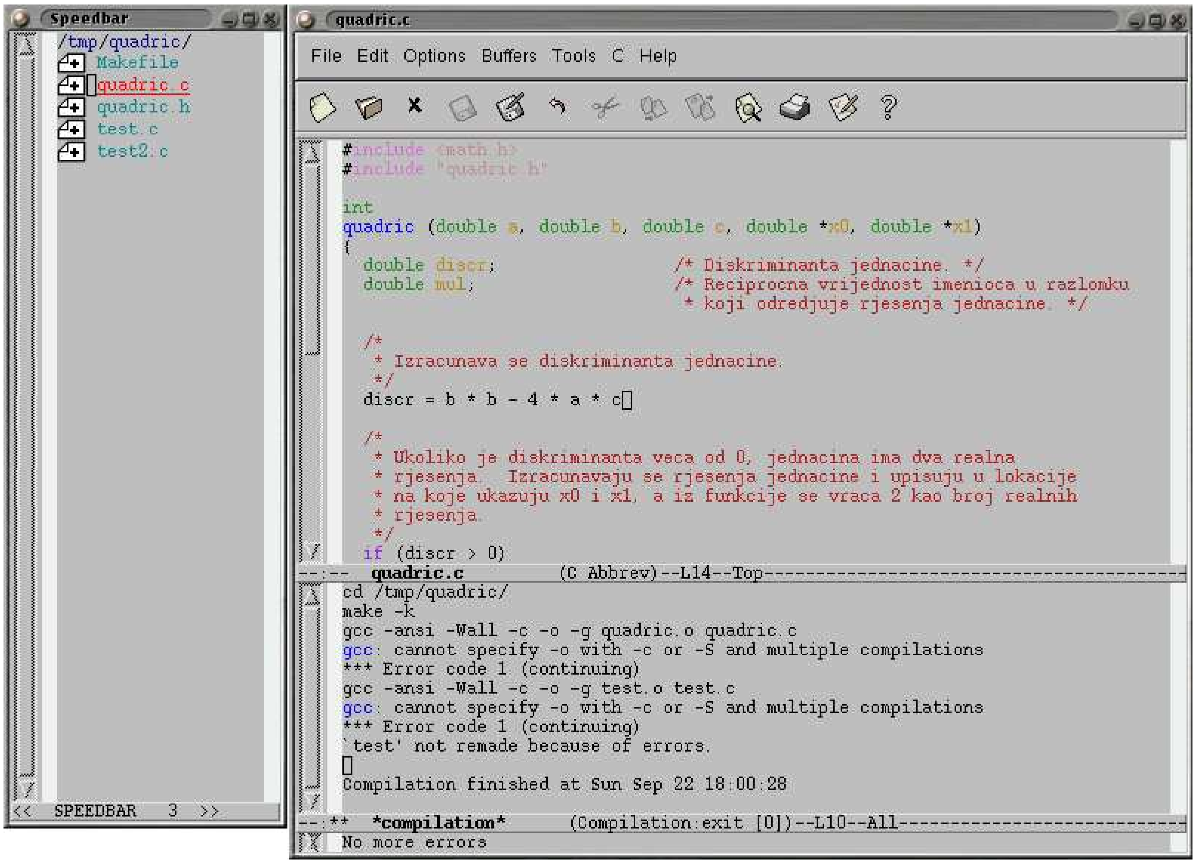
Iz ovog razloga *make* se ne označava samo alatom za automatsko ažuriranje fajlova zavisnih od nekih drugih fajlova, već u punom smislu predstavlja alat za održavanje projekta. U gornjem primjeru treba uočiti korišćenje znaka - ispred komandi koji obezbeđuje da *make* nastavi sa izvršavanjem narednih komandi ukoliko pri izvršavanju neke komande dođe do greške[5]. Takođe treba uočiti kako su kod `dist` cilja u listi zavisnosti navedeni `clean` i `beauty` ciljevi čime se obezbeđuje da prije kreiranja arhive budu obrisani svi fajlovi generisani tokom prevođenja odnosno da kod bude automatski propušten kroz *beautifier*.

Radi rekompajliranja projekta nije neophodno napuštati editor - *emacs* ima u `Tools` meniju opciju `Compile...` kojom se poziva *make*. Prije pozivanja, komandna linija kojom se poziva *make* biće ispisana u *minibuffer*-u. U podrazumijevanom stanju, *make* se pokreće sa opcijom `-k` koja označava da se sa prevođenjem nastavi koliko god je moguće dalje, tj. da se recimo prevođenje ne prekida čim se naiđe na sintaksnu grešku u nekom fajlu, već da se nastavi sa drugim fajlovima iz liste. Tekući direktorijum u kome *emacs* pokreće *make* je onaj direktorijum u kome se nalazi trenutno otvoreni fajl, a opcijom `-C` može sa specificirati da *make* promijeni svoj tekući direktorijum. Ukoliko je ova ili bilo koja druga *make* opcija potrebna, dovoljno je navesti samo prilikom prvog pokretanja *makefile*-a, za ubuduće će *emacs* zapamtiti opciju i sam će je dodavati u komandnu liniju.

Nakon pokretanja rekompajliranja, u donjoj polovini *emacs* prozora biva otvoren potprozor u kome se ispisuju rezultati prevođenja. Ukoliko se prilikom prevođenja jave greške, u *emacs*-u se uzastopnom primjenom naredbe `` C-x ` `` može pozicionirati na fajlove i linije u kojima je greška prijavljena. Ukoliko se ovoj izuzetno korisnoj funkciji *emacs*-a doda mogućnost prikaza stabla direktorijuma projekta koja se aktivira opcijom `Display Speedbar` opcijom iz `Tools` menija *emacs*-a, zatim mogućnost direktnog pristupa raspoloživoj programerskoj dokumentaciji komadnom `M-x man`, te mogućnosti interaktivnog rada sa debagerom i drugim programerskim alatima bez napuštanja *emacs*-a koje će biti pojašnjene u odgovarajućim poglavljima, jasno je zašto *UNIX* programerima nisu potrebna tzv. integrisana razvojna okruženja - *emacs* ima sve što takva okruženja nude i još mnogo više od toga. Tipičan primjer radne sesije u *emacs*-u tokom ispravljanja sintaksnih grešaka

[5]striktno posmatrajući, ovaj znak nije neophodan ispred poslednje komande u skupu komandi, ali nije ga zgoreg napisati svuda radi eventualnog kasnijeg dodavanja komandi u listu

prikazan je na slici 5.1.



Slika 5.1: Primjer sesije u *emacs*-u. Gornji potprozor sadrži fajl sa kodom, u donjem potprozoru su rezultati prevođenja. Pošto ima sintaksnih grešaka, odgovarajućom komandom pozicioniran je kursor u fajlu sa kodom na jednu na jednu od njih (u pitanju je zapravo ispušten terminator na kraju prethodne naredbe). Opis greške je istovremeno automatski postavljen na vrh potprozora sa rezultatima prevođenja. Sa lijeve strane prozora nalazi se *speedbar* sa hijerarhijom direktorijuma projekta.

Na kraju ovog poglavlja trebalo bi da se stekao uvid o velikom mogućnostima *GNU make*-a po svim aspektima održavanja projekta. Treba uočiti i kako je dobijen vrlo generički *makefile* koji se može uz minimalne izmjene koristiti i za druge projekte slične strukture, čak nezavisno od programskog jezika na kome se projekt implementira. Pored primjene u programiranju, *make* program se može koristiti i svuda gdje postoji potreba da se neki fajlovi automatski ažuriraju na osnovu nekih drugih fajlova. Tako je npr. *make* korišćen i prilikom pisanja ovog teksta - originalni tekst je napisan na LaTeX-u iz koga su onda generisane verzije u raznim drugim formatima, pri čemu su to generisanje verzija kao i drugi aspekti održavanja teksta automatizovani putem odgovarajućeg *makefile*-a.

Poglavlje 6

GNU gdb debager

Kako je već pomenuto, da bi debager mogao da se koristi obavezno je prilikom prevođenja uključiti `-g` opciju *gcc* prevodioca. Ovo je najbolje uraditi tako što će se na kraj liste opcija sadržane u promjenljivoj `CFLAGS` u *makefile*-u dodati i ova opcija. Treba napomenuti i da se alternativno, ukoliko će za debagovanje biti korišćen *GNU gdb* debager, može koristiti i `-ggdb` opcija - u tom slučaju će u izvršni fajl biti upisane i neke dodatne informacije koje omogućavaju da se iskoriste dodatne opcije *gdb*-a.

Nakon što je obezbijeđeno da je projekat preveden sa nekom od navedenih opcija uključenom, debager se pokreće tako što se iza imena programa navede ime izvršnog fajla koji će biti debagovan, uz eventualne argumente komandne linije. Koristeći pokazni primjer i pod pretpostavkom da se nalazimo u korenom direktorijumu projekta, naredba za pokretanje debagera bi glasila:

```
gdb test
```

Nakon ovoga, aktivira se *gdb shell* koji je veoma sličan običnom *shell*-u, samo što naravno ovaj prepoznaje samo komande koje se odnose na debagovanje. Program koji se debaguje se iznova pokreće naredbom `run`, ili kraće `r` - za sve komande *gdb*-a dovoljno je navesti samo onoliko karaktera sa početka imena koliko ih razlikuje od drugih komandi, što za većinu komandi znači da je prvo slovo imena dovoljno[1]. Dalje se sve odvija kao pri normalnom pokretanju programa - obzirom da pokazni primjer očekuje da se na početku

[1] *gdb shell*, baš kao i pravi *shell*, podržava automatsko kompletiranje imena komande pritiskom na *Tab* taster, što takođe treba obavezno koristiti u cilju izbjegavanja kucanja punih imena komandi

unesu koeficijenti kvadratne jednačine čija se realna rješenja traže, program će na odgovarajućem mjestu zastati očekujući unos, a potom će biti izračunati korijeni i ispisani rezultati, nakong čega će *gdb* odštampati poruku o normalnom završetku programa i prikazati prompt očekujući dalje naredbe.

Naravno, svrha sesije sa debagerom nije da se izvrši program kao što bi bio izvršen i bez debagera, već da se program izvršava u koracima, prateći šta se tokom tog izvršavanja tačno dešava. Postoje brojne naredbe *gdb*-a kojima se može upravljati ovakvim izvršavanjem programa i koje će u ostatku poglavlja biti opisane. Međutim, prije nego što se pređe na razmatranje tih naredbi treba reći da se iz *gdb*-a izlazi naredbom `quit`, dok se pomoć može u svakom trenutku dobiti naredbom `help`. Takođe treba pomenuti i da se koristeći komandu `shell` može bez izlaska iz *gdb shell*-a izvršiti bilo koja *UNIX* naredba, a koristeći komandu `make` može se pokrenuti *make* bez i da se kuca `shell make`.

Da bi program mogao da se izvršava korak po korak, potrebno je prije njegovog pokretanja postaviti neku prekidnu tačku (*breakpoint*). Prekidne tačke se postavljaju naredbom `breakpoint` pri čemu se pozicija prekidne tačke može zadati na više načina. Najjednostavnije je za specificiranje prekidne tačke navesti ime funkcije na čijem početku se hoće da se *gdb* zaustavi. Tako bi postavljanje prekidne tačke na početak posmatranog programa bilo izvedeno naredbom `breakpoint main`. Nakon ovoga *gdb* ispisuje poruku da je postavio prekidnu tačku, navodeći u toj poruci redni broj prekidne tačke. Sve dalje operacije sa tom prekidnom tačkom zahtijevaju ovaj broj kao argument. Alternativno, prekidna tačka se često zadaje i tako što se navedu razdvojeni dvotačkom ime fajla i broj linije u koje se prekidna tačka postavlja. Ako se prekidna tačka postavlja u tekući fajl, onda je dovoljno navesti samo broj linije; obzirom da je po pokretanju *gdb*-a tekući fajl onaj koji sadrži `main()` funkciju, prethodno pomenuta prekidna tačka se mogla postaviti i naredbom `breakpoint 17`. Često se za postavljanje prekidne tačke kao argument koriste `+` ili `-` praćeni nekim brojem u kom slučaju se prekidna tačka postavlja onoliko linija koda iza odn. ispred tekuće pozicije u izvršavanju programa koliko iznosi dati broj.

Kada se nakon postavljanja prekidne tačke pokrene program naredbom `run`, izvršavanje će biti zaustavljeno na početku `main()` funkcije. Treba naglasiti da se izvršavanje zaustavlja uvijek ispred linije u kojoj je postavljena prekidna tačka, tj. instrukcija koja se nalazi u toj liniji ne biva izvršena. Izvršavanje može biti nastavljeno na više načina. Jedna mogućnost je da se postavi prekidna tačka recimo na sledećoj liniji koda naredbom `breakpoint 20`, pa da se onda nastavi program naredbom `continue`. Naredba `continue` služi za nastavljanje izvršavanja programa do prve prekidne tačke na koju se naiđe u toku tog izvršavanja. U ovom slučaju će program očekivati da se unesu ulaznu podaci da bi se potom opet zaustavio zbog prekidne tačke u liniji 20. Jedna digresija na ovom mjestu - već nakon nekoliko pokretanja izvršnog programa ili sesija sa debagerom postaće očigledno da je neprestano unošenje ulaznih podataka naporno. Bolje rješenje je da se ulazni podaci upišu u neki fajl, a da se potom prilikom pokretanja programa (bilo normalno, bilo preko debagera) preusmjerava standardni ulaz na ovaj fajl. U tu svrhu se može u direktorijumu projekta kreirati fajl recimo sa imenom `data.in` sa ulaznim podacima. Npr. za jednačinu $x^2 - 3x + 2$ taj fajl bi imao oblik:

```
1 -3 2
```

i onda bi, nakon pokretanja debagera, program trebalo pokretati naredbom `run <data.in`.

Prekidna tačka se briše naredbom `delete`, pri čemu se kao argument navodi redni broj prekidne tačke. Ako se ne navede agument, onda se brišu sve prekidne tačke. Na isti način funkcionišu naredbe `disable` odn. `enable`, kojima se privremeno isključuju odn. opet uključuju pojedine prekidne tačke.

Umjesto postavljanja *breakpoint*-a na narednu instrukciju da bi se prešlo na nju, može se za kretanje kroz kod korak po korak iskoristiti naredba `next`. Kada se koristi bez argumenata, ova naredba specifira izvršavanje tekuće instrukcije programa; kada se pak navede argument, koji mora biti broj, onda se izvršava onoliko sledećih naredbi programa koliko iznosi dati broj. Ukoliko se kreće kroz pokazni primjer naredbom `next`, primijetiće se da ova naredba tretira poziv funkcije `quadric()` kao jednu instrukciju, tj. ne ulazi se u ovu funkciju. Ukoliko se želi kretati korak po korak kroz program, ali uz ulaženje u funkcije, onda treba koristiti naredbu `step`. Ova naredba se koristi na potpuno isti način kao i `next` naredba. Ukoliko se želi u jednom koraku završiti tekuća funkcija i zaustaviti se nakon povratka iz nje, onda treba upotrebiti naredbu `finish`.

Svaki put kada se izvrši neka od naredbi za kretanje kroz program, *gdb* štampa tekuću instrukciju. Međutim, radi lakšeg orijentisanja u kodu, može se koristiti i `list` nareba. Ako se zada bez argumenata, ova naredba štampa 10 narednih linija koda počev od tekuće instrukcije, sa jednim argumentom štampa 10 linija koje okružuju liniju koja je data tim argumentom i sa dva argumenta štampa sve linije između linija datih argumentima.

Unekoliko slična prethodnoj naredbi je naredba `info`, pa će i ona biti pojašnjena na ovom mjestu. Ova naredba služi za prikaz informacija o stanju debagera i stanju izvršavanja programa. Ova naredba je obavezno praćena argumentom koji određuje koja grupa informacija će biti prikazana. Broj grupa informacija koje se na ovaj način mogu prikazati je jako veliki. Tako se npr. naredbom `info breakpoints` prikazuje lista svih postavljanih prekidnih tačaka, naredbom `info source` informacije o tekućem fajlu, naredbom `info registers` sadržaj registara glavnog procesora, naredbom `info float` sadržaj registara numeričkog koprocesora itd.

Izvršavanje programa korak po korak samo po sebi nije dovoljno za debagovanje programa. Druga neophodna komponenta za uspješno obavljanje ovog zadatka je mogućnost očitavanja vrijednosti promjenljivih i izraza tokom izvršavanja programa. *gdb* naravno pruža ovakvu mogućnost - naredbom `print` se može u trenutku kada je izvršavanje programa prekinuto odštampati vrijednosti proizvoljne promjenljive ili izraza koji su navedeni kao argument ove naredbe. Neka je recimo prekidna tačka postavljena na liniju 14 fajla `quadric.c` (ova prekidna tačka bi bila specificirana naredbom `breakpoint quadric.c:14`). Tada se mogu ispisati vrijednosti argumenata funkcije `quadric()` naredbama `print a`, `print b` i `print c`, zatim recimo vrijednost promjenljive `discr` naredbom `print discr`[2], ili recimo vrijednosti izraza koji ulaze u izračunavanje diskriminante naredbama `print b*b` i `print -4*a*c`. Treba uočiti da je sintaksa izraza uobičajena C-ovska sintaksa.

Za prikaz sadržaja memorijskih lokacija služi naredba `x` (od *examine*). Ova naredba može kao argument da ima adresu memorijske lokacije koja će biti prikazana, a takođe postoje opcije za zadavanje broja memorijskih lokacija koje će biti izlistane, te formata u kome će njihov sadržaj biti interpretiran. Za prikaz registara procesora odn, koprocesora služe već pominjane naredbe `info registers` odn. `info float`.

Veoma često je potrebno da se sadržaj neke promjenljive ili izraza prikaže svaki put kada dođe do zaustavljanja programa. Ovo se može postići naredbom `display` koja ima istu sintaksu kao i naredba `print` osim što se nakon izdavanja ove naredbe dati promjenljiva odn. izraz ne prikažu samo jednom već svaki put kada dođe do zaustavljanja programa. Naredba `display` je slična naredbi `breakpoint` u smislu da se pri svakom pozivanju ove naredbe dodjeljuje redni broj promjenljivoj odn. izrazu koji se specifiraju za prikazivanje, te da se njihov prikaz može ukinuti `undisplay` naredbom (ekvivalent `delete` naredbe za prekidne tačke) odn. privremeno isključiti i opet uključiti nared-

[2] dobra prilika da se podsjeti kako neinicijalizovane promjenljive na C-u imaju potpuno proizvoljne vrijednosti

bama `disable display` i `enable display` (ekvivalenti naredbi `disable` i `enable` za prekidne tačke), pri čemu se za svaku od ovih naredbi kao argument navodi redni broj koji su promjenljiva odn. izraz dobili prilikom poziva `display` naredbe. Postoji i naredba `info display` koja ispisuje informacije o svim promjenljivim odn. izrazima specificiranim za prikazivanje.

U izrazima koji se zadaju u naredbama `print` i `display` mogu figurisati samo globalne promjenljive u programu ili lokalne promjenljive u onoj funkciji u kojoj je izvršavanje programa zaustavljeno, tj. samo one promjenljive koje su trenutno u dosegu. Ukoliko se npr. u trenutku kad se sa izvršavanjem programa prešlo u funkciju `quadric()` pokuša očitati vrijednost promjenjive `count` iz funkcije `main()` naredbom `print count` biće prijavljena greška. Naredbom `frame` mogu se dobiti podaci o tekućoj funkciji, dok se naredbom `backtrace` dobija hijerarhija svih poziva funkcija koja je dovela izvršavanje programa do date tačke. Pritom, svakom pozivu funkcije je dodijeljen redni broj, pri čemu poziv tekuće funkcije ima broj 0, poziv funkcije iz koje je pozvana tekuća funkcija broj 1 i tako redom. Da bi se moglo pristupiti nekoj promjenljivoj iz funkcije koja nije tekuća, treba se prvo privremeno pozicionirati na odgovarajući nivo hijerarhije poziva, što se radi opet naredbom `frame`, s tim što se ovdje navodi argument i to broj koji je dodijeljen toj funkciji u hijerarhiji. Dakle, za očitavanje vrijendosti promjenljive `count` iz `main()` funkcije bi trebalo prvo izvršiti naredbu `frame 1`, a potom naredbu `print count`. Alternativno, kretanje kroz hijerahiju poziva može se vršiti naredbama `up` odn. `down` kojima se pomjera za po jedan nivo naviše odn. naniže u hijerarhiji. Tako se u posmatranom primjeru ispisivanje vrijednosti promjenljive `count` moglo postići i sekvencom naredbi `up`, `print count`. Treba primijetiti da se kretanjem kroz stek ne remeti tok izvršavanja programa, tj. da kad se izda neka naredba za nastavak izvršavanja programa *gdb* se automatski vraća u najniži nivo hijerarhije i izvršavanje programa se nastavlja od tačke u kojoj se bilo stalo. U vezi sa hijerarhijom poziva treba još pomenuti i da se varijacijama naredbe `info` mogu dobiti neke korisne informacije - tako se npr. naredbom `info args` u svakom trenutku može dobiti spisak vrijednosti argumenata tekućeg poziva (neopterećen ostalim informacijama koje se inače dobijaju uz njega naredbom `frame`), a naredbom `info locals` se može dobiti spisak vrijednosti svih lokalnih promjenljivih u tekućem pozivu.

Koristeći *gdb* moguće je i izmijeniti način izvršavanja programa. Naredbom `set variable` koju slijede ime promjenljive, znak jednakosti i nova vrijednost vrši se izmjena vrijednosti promjenljive. Naredbom `jump` koju slijedi specifikacija linije koda skače se sa tekuće tačke u izvršavanju programa na zadatu liniju. Naredbom `return` vrši se trenutni povratak iz tekuće funkcije (ako je funkcija deklarisana da vraća neku vrijednost, onda se obavezno iza ove naredbe mora zadati neka povratna vrijednost).

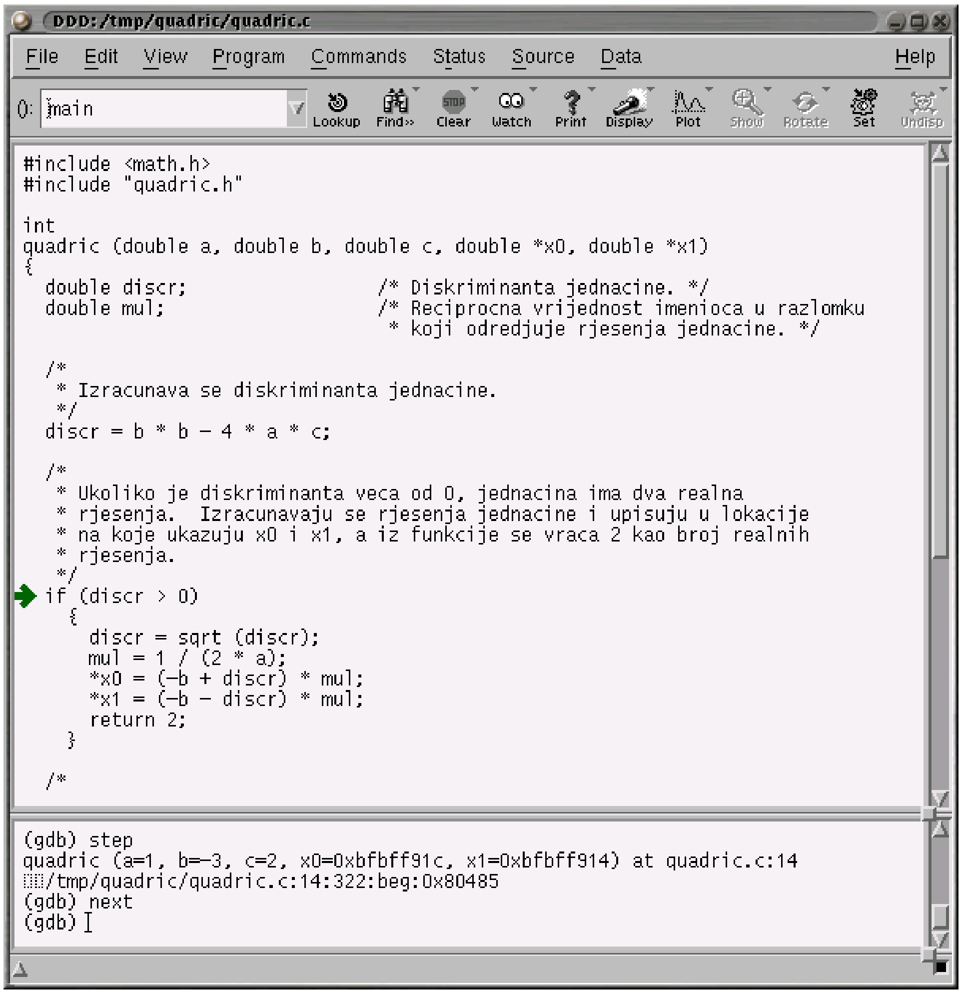
Na kraju, ukoliko se želi u nekom trenutku tokom debagovanja pogledati listing mašinskih instrukcija u koje je preveden posmatrani program, to se može uraditi naredbom `disassemble`. Sintaksa asemblera je već pomenuta *AT&T* sintaksa.

Mada je u *gdb*-u uloženo dosta napora da se korišćenje debagera učini jednostavnijim, ipak stalno kucanje komandi može biti zamorno. Za programere koji su navikli na debagovanje u nekom grafičkom okruženju rešenje je *GNU DDD* (*Data Display Debugger*). Radi se praktično o grafičkom omotaču oko *gdb*-a, gdje se komande *gdb*-a mogu izdavati preko menija i *toolbar*-a, a prekidne tačke postavljati ili vrijednosti promjenljivih i izraza očitavati pomoću miša. Primjer sesije sa *DDD*-om je prikazan na slici 6.1. Treba uočiti kako se u donjem dijelu prozora vidi *gdb*-ov *prompt* u kome se može sa *gdb*-om raditi na uobičajeni način, kucanjem komandi.

Međutim, *DDD* spada u onu grupu *GNU* programerskih alata koji se obično ne isporučuju uz *UNIX* distribucije, već se moraju naknadno nabavljati i instalirati. Zbog te činjenice, kao i zato što je prirodno da se debagovanje vrši u okruženju u kome se obavljaju ostali programerski zadaci, najbolje je za ovu svrhu koristiti *emacs*, koji u odgovarajućem režimu rada podržava sličan način debagovanja kao i *DDD*. Naime, u `Tools` meniju *emacs*-a postoji opcija `Debugger...` kojom se pokreće *gdb*, pri čemu interakcija sa njime ide kroz poseban *emacs* bafer. Kao i kod pokretanja *make*-a iz *emacs*-a, u *minibuffer*-u je nakon pokretanja ove komande moguće zadati parametre komandne linije za pokretanje debagera (obavezno je navesti bar ime programa), nakon čega se otvara prozor za interakciju sa njime.

U trenutku kada je pokrenut debager, meni za fajlove koji sadrže kod dobija podmeni `Gud`[3] sa komandama debagera. Stavkama iz ovog menija, naravno i odgovarajućim kombinacijama tastera, moguće je postaviti prekidnu tačku na liniju koda u kojoj se nalazi kursor, upravljati tokom izvršavanja programa, odštampati vrijednost promjenljive ili izraza nad kojom se nalazi kursor i tome slično. Tokom debagovanja, *emacs* prozor je podijeljen na dva potprozora (jedan za debager, drugi za fajl sa kodom), kao na slici 6.2. Pritom, komande koje se izdaju u potprozoru sa debagerom počinju sekvencom `C-c` (npr. komanda `up` se poziva sa `C-c <`), dok komande koje se izdaju u potprozoru koji sadrži kod počinju sekvencom `C-c C-a` (npr. komanda `up`

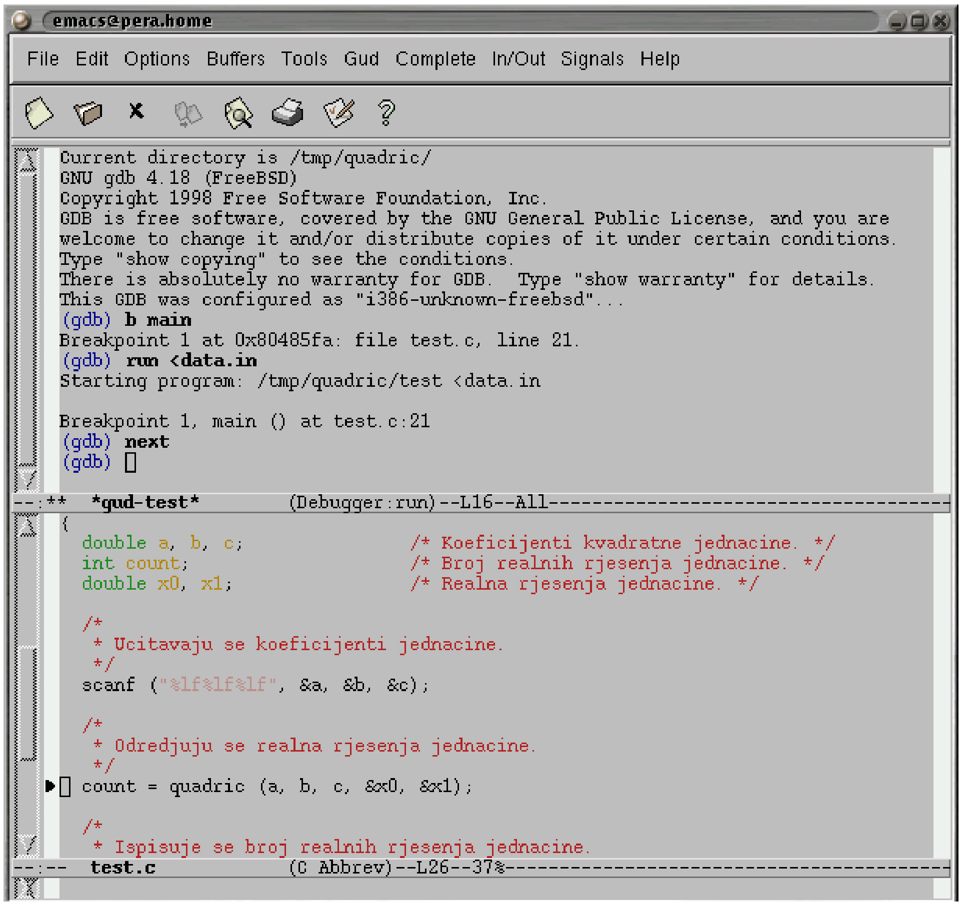
[3] *grand unified debugger*



Slika 6.1: Primjer sesije sa *DDD*-om.

se ovdje poziva sa `C-c C-a <`). Izuzetak je komanda za postavljanje prekidne tačke, koja se u prozoru sa kodom zadaje sa `C-x` koga slijedi pritisak na *Space* taster.

Pretjerano korišćenje debagera je loša navika; umjesto toga, mnogo bolje je greške otkrivati pažljivim pregledom samog koda. Ipak, u situacijama kada je upotreba debagera neizbježna, *gdb* je, bilo da se koristi iz komandne linije ili iz nekog od pomenutih okruženja, moćan alat za obavljanje tog zadatka. Osim onih pomenutih kroz ovo poglavlje, *gdb*, kao i drugi ovdje razmatrani alati, ima još mnogo drugih korisnih opcija o kojima se može (i treba) više



Slika 6.2: Primjer sesije sa *gdb*-om u *emacs*-u.

pročitati u originalnoj dokumentaciji.

Poglavlje 7

GNU gprof program za analizu izvršavanja programa

Korak u razvoju programa koji slijedi debagovanje jeste optimizacija. Već je rečeno da na polju lokalne optimizacije *gcc* prevodilac može dosta toga da uradi. Što se globalne optimizacije tiče, tu opet važi savjet iz poslednjeg pasusa poglavlja o *gdb*-u: najbolji rezultati se mogu postići pažljivim pregledom samog koda i eventualnim unapređenjima na planu korišćenih algoritama. Ipak, postoje i alati koji mogu da analiziraju izvršavanje programa i izračunaju neke pokazatelje u pogledu njegove efikasnosti; takvi alati se označavaju imenom *profiler*-i i *GNU* varijanta ovog programa jeste *gprof*.

Da bi neki program mogao biti analiziran pomoću *gprof*-a, potrebno je i prilikom prevođenja i prilikom linkovanja u listu opcija prevodioca dodati i `-pg`. Ovu opciju je potrebno uključiti kako prilikom generisanja objektnih fajlova tako i prilikom generisanja izvršnog fajla (tj. linkovanja). Nakon ovoga treba normalno izvršiti program; izvršavanje programa će teći sporije nego inače jer će prilikom izvršavanja biti prikupljane i informacije potrebne za analizu koje će biti upisane u fajl pod imenom `gmon.out` u direktorijumu iz koga je pokrenut program. Prilikom kasnije analize treba voditi računa na koji način je izvršavan program - djelovi programa koji nisu aktivirani neće biti zastupljeni u izvještaju profilacije, naravno ne zato što su posebno efikasno napisani već zato što se kroz njih nije prolazilo.

Obzirom da je pokazni primjer izuzetno jednostavan, za potrebe demonstriranja rada *profiler*-a test program će biti nešto izmijenjen u smislu da neće rješavati jednu jednačinu sa datim koeficijentima već će umjesto toga generisati veliki broj (u ovom slučaju 1000000) pseudo-slučajno odabranih trojki koeficijenata, rješavati jednačinu za svaku od tih trojki i onda uvrštavati rješenja u jednačinu u cilju provjere njihove ispravnosti. Fajl `test.c` će za ovako postavljen zadatak imati oblik:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "quadric.h"

/*
 * Program testira funkciju za odredjivanje realnih korijena kvadratne
 * jednacine. Koeficijenti jednacine se generisu kao slucajni brojevi iz
 * datog intervala, a onda se poziva funkcija za odredjivanje korijena
 * da bi se potom korijeni uvrstili u jednacinu i ispitivalo se koliko
 * tako dobijena vrijednost odstupa od 0. Ni za jednu kombinaciju
 * koeficijenata ne smije ova vrijednost preci zadatu tacnost; u
 * suprotnom se program prekida i smatra se da metoda nije dovoljno
 * precizna.
 */
int
main ()
{
  const int n = 1000000;        /* Broj testova. */
  const double lo = -10, hi = 10;       /* Opseg iz koga ce uzimati vrijednosti
                                         * koeficijenti jednacine. */
  const double epsilon = 0.000001;      /* Tacnost na koju se vrsi testiranje. */
  double a, b, c;               /* Koeficijenti kvadratne jednacine. */
  int count;                    /* Broj realnih rjesenja jednacine. */
  double x0, x1;                /* Realna rjesenja jednacine. */
  int i;                        /* Brojac u petljama. */

  /*
   * Zadati broj puta...
   */
  for (i = 0; i < n; i++)
    {
      /*
       * ...generisu se slucajne vrijednosti koeficijenata iz datog
       * opsega,...
       */
      a = lo + (hi - lo) * rand () / RAND_MAX;
      b = lo + (hi - lo) * rand () / RAND_MAX;
      c = lo + (hi - lo) * rand () / RAND_MAX;

      /*
       * ...odredjuju se realna rjesenja jednacine...
       */
      count = quadric (a, b, c, &x0, &x1);

      /*
       * ...i ispituje se tacnost rjesenja.
       */
      switch (count)
        {
```

```
        case 1:
          assert (fabs (a * x0 * x0 + b * x0 + c) < epsilon);
          break;

        case 2:
          assert (fabs (a * x0 * x0 + b * x0 + c) < epsilon);
          assert (fabs (a * x1 * x1 + b * x1 + c) < epsilon);
          break;
        }
    }

  return 0;
}
```

Kada se prevede i izvši ovaj program, *profiler* se pokreće naredbom:

```
gprof -b test
```

tj. iza imena programa navode se opcije i ime izvršnog fajla. Pritom je važno da se *profiler* pokrene iz istog direktorijuma iz koga je pokretan i izvršni program odn. iz onog direktorijuma u kome se nalazi fajl `gmon.out`. Opciju `-b` treba uvijek koristiti jer će u suprotnom *profiler* pored rezultata analize uvijek ispisivati i objašnjenje o načinu interpretiranja rezultata.

Postoje dva osnovna skupa rezultata koje ispisuje *profiler* i to tzv. *flat profile*, te graf poziva. Za dati primjer ispis *profiler*-a bi bio oblika

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 72.12      1.50     1.50                             main
 27.88      2.08     0.58  1000000   580.00   580.00  quadric

                        Call graph


granularity: each sample hit covers 4 byte(s) for 0.48% of 2.08 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    1.50    0.58                 main [1]
                0.58    0.00 1000000/1000000     quadric [2]
-----------------------------------------------
                0.58    0.00 1000000/1000000     main [1]
[2]     27.9    0.58    0.00 1000000         quadric [2]
-----------------------------------------------

Index by function name

   [1] main                    [2] quadric
```

Razmotrimo prvo *flat profile*. U poslednjoj koloni dato je ime funkcije na koju se podaci iz odgovarajućeg reda odnose. U prvoj koloni se nalazi procenat ukupnog vremena tokom izvršavanja programa koje je provedeno u posmatranoj funkciji. U drugoj koloni je dato zbirno vrijeme, odn. vrijeme koje je tokom izvršavanja programa provedeno u datoj funkciji i funkcijama koje su iznad nje na listi, dok je u trećoj koloni dato vrijeme provedeno u samoj funkciji. U četvrtoj koloni nalazi se broj pozivanja te funkcije u programu. U petoj koloni dato je prosječno vrijeme izvršavanja funkcije po jednom pozivu, u pikosekudnama. Ovo vrijeme se odnosi na kod same funkcije, ne i na pozive nekih drugih funkcija iz nje, dok je ukupno vrijeme po pozivu provedeno u samoj funkciji i funkcijama koje su iz nje pozivane dato u šestoj, pretposlednjoj koloni.

Pokazni primjer je opet veoma jednostavan i pravolinijske strukture, tako da su gornji podaci sasvim očekivani. Ipak, čak i na tako jednostavnom primjeru interesantno je uporediti odnos vremena utrošenog na generisanje koeficijenata i ispitivanje ispravnosti rješenja sa vremenom samog rješavanja jednačine - možda pomalo neočekivano, ispostavlja se da rješavanje jednačine traje kraće. Kod komplikovanijih programa, *flat profile* omogućava da se vrlo lako uoče djelovi programa u kojima se potroši najviše vremena i koje prema tome ima najviše smisla optimizovati.

Graf poziva predstavlja slične informacije kao i *flat profile*, samo interpretirane na nešto drugačiji način. Kod ovog grafa su, razdvojeni linijama sa crticama, dati podaci za svaku funkciju ponaosob o funkcijama iz kojih je ona pozvana i o funkcijama koje ona poziva. Svakoj funkciji je dodijeljen redni broj (ispisan u uglastim zagradama) i u prvoj koloni je označena funkcija na koju se dati blok podataka odnosi. U drugoj koloni za tu funkciju je dat procenat vremena ukupnog izvršavanja programa koji je proveden u datoj funkciji i funkcijama koje su iz nje pozvate. U trećoj koloni je dat iznos vremena proveden u posmatranoj funkciji, ne računajući vrijeme u funkcijama koje su iz nje pozvate, koje je dato u četvrtoj koloni. U narednoj koloni je dat broj pozivanja te funkcije, dok je u poslednjoj koloni dato ime funkcije sa indeksom iz liste. Za funkcije iz kojih je pozivana funkcija posmatrana u datom bloku grafa poziva, vrijeme u trećoj koloni se odnosi na vrijeme provedeno u pozivima posmatrane funkcije, a u četvrtoj koloni na vrijeme provedeno u funkcijama pozivanim iz posmatrane funkcije. U pretposlednjoj koloni nalaze se dva broja, od kojih se prvi odnosi na broj poziva posmatrane funkcije iz funkcije iz koje je ona pozivana, dok se drugi odnosi na ukupan broj nerekurzivnih poziva posmatrane funkcije. Tako se npr. u drugom bloku gornjeg grafa vidi da je iz funkcije `main()` funkcija `quadric()` pozivana

1000000 puta dok je istovremeno ukupan broj nerekurzivnih poziva funkcije `quadric` takođe jednak 1000000. Za funkcije koje su pozivane iz funkcije posmatrane u datom bloku grafa poziva treća kolona predstavlja vrijeme provedeno u toj funkciji, ne računajući funkcije koje su iz nje pozivane i za koje je vrijeme provedeno u njima dato u četvrtoj koloni. U petoj koloni su opet data dva broja od kojih prvi predstavlja broj poziva te funkcije iz funkcije posmatrane u datom bloku, dok je drugi ukupan broj nerekurzivnih poziva te funkcije. U prvom bloku gornjeg grafa opet se iz ove kolone vidi da je funkcija `quadric()` pozivana 1000000 puta iz funkcije `main()` dok je ukupan broj nerekurzivnih poziva te funkcije takođe jednak 1000000. U slučajevima uzajamno rekurzivnih ili samorekurzivnih poziva u svakom bloku se izdvaja po jedan red za svaki ciklus ovakvog tipa.

Ukoliko se prilikom pozivanja *profiler*-a navede opcija `-l`, onda su jedinice prikazivanja pojedinačne linije koda umjesto funkcija. Da bi ova opcija mogla da se koristi, potrebno je prilikom prevođenja pored `-pg` opcije zadati i `-g` ili `-ggdb` opciju. Opcija `-l` *profiler*-a je korisna uglavnom za *flat profile*, kod grafa poziva je korišćenje funkcija kao jedinica prikaza znatno informativnije. Sa druge strane, ukoliko se pri prevođenju zada i opcija `-a`, istom prilikom će u fajl pod imenom `bb.out` biti upisan broj prolaza kroz svaki blok koda, što takođe može biti veoma korisna informacija za optimizaciju programa. Ovaj fajl nije međutim u previše čitljivom formatu, pa je bolje konvertovati ga u format koji *gprof* može da obradi, što se radi pomoću skripta `bbconv.pl`. Ovaj skript se nažalost obično ne nalazi u distribucijama *UNIX*-a, već je potrebno izvaditi ga iz originalne verzije *GNU binutils* paketa, koji u sebi obuhvata i *gprof*. Kada se instalira ovaj skript (dovoljno ga je staviti negdje u *path*), tada prvo treba konvertovati format fajla `bb.out` i smjestiti rezultat recimo u fajl `bb.gmon.out` naredbom:

```
bbconv.pl <bb.out >bb.gmon.out
```

a onda pokrenuti *gprof* naredbom:

```
gprof -l -A -x -s test gmon.out bb.gmon.out
```

Efekat opcije `-l` je, kao što je već rečeno, da se profilacija sprovodi nad pojedinačnim linijama koda. Opcija `-A` označava da kao rezultat analize treba da budu ispisani fajlovi sa kodom sa naznačenim brojem prolazaka kroz pojedine segmente koda. Opcija `-x` označava da su segmenti koda koji će biti označeni pojedinačne linije, a ne čitavi blokovi koda. Na kraju, opcija

`-s` označava da kao ulaz u *gprof* treba umjesto samo `gmon.out` fajla korisiti više fajlova, čija će imena biti navedena iza imena izvršnog programa koji se analizira (ovi fajlovi će biti povezani u fajl sa imenom `gmon.sum` i taj fajl će biti korišćen kao ulaz u *gprof*. Jedan segment izlaza ovakve naredbe na pokaznom primjeru za fajl `quadric.c` ima oblik (sa komentarima ručno izbačenim radi bolje preglednosti):

```
1000000 ->  if (discr > 0)
               {
 628162 ->       discr = sqrt (discr);
 628162 ->       mul = 1 / (2 * a);
 628162 ->       *x0 = (-b + discr) * mul;
 628162 ->       *x1 = (-b - discr) * mul;
 628162 ->       return 2;
               }

 371838 ->   if (discr == 0)
               {
  ##### ->       *x0 = -b / (2 * a);
  ##### ->       return 1;
               }
```

Iz gornjeg listinga se vidi da je prva `if` naredba izvršena 1000000 puta, dok su naredbe u tijelu ove `if` konstrukcije izvršene 628162 puta. Druga `if` naredba je pak izvršena 371838 puta; naredbe u tijelu te konstrukcije nisu izvršene nijednom, što znači da nije bilo nijedne kombinacije koeficijenata koja bi dala jedno realno rješenje.

Sve informacije koje generiše *profiler*, a koje se tiču vremena, prikupljaju se tako što se nakon kratkih regularnih intervala program zaustavlja i gleda se gdje se u tom trenutku unutar njega nalazi (dužina intervala je uvijek odštampana na početku *flat profile* izvještaja). Na taj način ove informacije treba uzimati sa određenom rezervom, jer su podložne statističkim devijacijama. Da bi se obezbijedilo da one budu pouzdanije, poželjno je da se program izvršava što duže. Sa druge strane, informacije koje se odnose na broj poziva funkcije ili broj prolaza kroz neki blok koda se dobijaju brojanjem, tako da su u pitanju potpuno tačni brojevi.

Poglavlje 8

GNU ar arhiver i kreiranje biblioteka

Funkcija `quadric()` iz pokaznog primjera je napisana tako da rješava precizno određen problem nezavisno od konteksta u kome se taj problem javlja. Zato se može očekivati da ova funkcija bude korisna i za neke druge programe, a ne samo za konkretan program koji je u pokaznom primjeru uz nju napisan. Izdvajanjem ove funkcije u poseban fajl znatno je olakšano njeno korišćenje i u nekim drugim projektima, ali ipak još uvijek treba svaki put uključivati i prevesti fajl `quadric.c` u svakom projektu u kome će ova funkcija da se koristi. Bolje rješenje bi bilo prekompajlirati ovu funkciju, zajedno sa eventualno još nekim funkcijama slične namjene, i onda je u takvom obliku uključivati u projekte u kojima je potrebna. Skup prekompajliranog koda ovog oblika naziva se bibliotekom. U zavisnosti od toga kako se taj kod povezuje sa projektom u kome će funkcije sadržane u biblioteci biti korišćene razlikuju se statičke i dinamičke biblioteke; u ovom tekstu biće razmotrene samo statičke biblioteke koje nisu ništa drugo do skup objektnih fajlova objedinjenih u jedan fajl (uz još neke dodatne informacije upisane u taj fajl, a koje se odnose na sadržaj objektnih fajlova, preciznije na listu funkcija koje su implementirane u tim fajlovima). Alat koji se koristi za kreiranje biblioteka na *UNIX*-u jeste običan arhiver *ar*. Biblioteka se kreira tako što se iza imena arhivera navedu odgovarajuće komandne opcije, potom ime biblioteke, a onda imena objektnih fajlova koji će ući u sastav biblioteke. Po konvenciji, ime fajla koji sadrži biblioteku na *UNIX*-u počinje prefiksom `lib`, a ekstenzija za fajlove sa statičkim bibliotekama je `.a`, pa bi naredba za kreiranje biblioteke od fajla `quadric.obj` glasila:

```
ar -crusv libquadric.a quadric.o
```

ar spada u grupu programa kojima se opcije ne navode iza crtice. Opcija `c` označava da arhivu treba kreirati ako ne postoji. Opcija `r` označava da fajlove koji su nabrojani treba umentuti u arhivu uz zamjenu prethodne verzije svakog fajla ako takva postoji u arhivi. Opcija `u` označava da u arhivu treba umetati samo one fajlove koji su noviji od verzije fajla koja već eventualno postoji u arhivi. Opcija `s` označava da su fajlovi koji se dodaju u arhivu objektni fajlovi, te da treba u arhivu dodati i informacije o sadržaju tih fajlova. Opcija `v` označava da tokom arhiviranja program treba da prikazuje detaljne informacije o tome šta radi.

Jednom kada je biblioteka napravljena, da bi se koristila iz nekog projekta dovoljno je uključiti odgovarajući *header* fajl u fajlove iz kojih se pozivaju funkcije biblioteke, kao i linkovati projekat sa bibliotekom. U posmatranom primjeru, fajl `quadric.h` je već uključen u fajl `test.c`, tako da jedino što treba uraditi da bi se rekompajlirao projekat, pri čemu je ovaj put funkcija `quadric` izdvojena u bibliotku, jeste da se navede biblioteka prilikom linkovanja:

```
gcc -L. -lquadric -lm -o test test.c
```

Treba uočiti da je ovdje prevođenje fajla `test.c` i linkovanje urađeno u jednom koraku. Opcijom `-L` dodat je tekući direktorijum (a to treba da bude koreni direktorijum projekta) u listu direktorijuma koji se pretražuju za biblioteke, a opcijom `-l` navedeno je ime biblioteke (ime biblioteke se uvijek navodi bez prefiksa `lib` i ekstenzije). Ove opcije su pomenute u poglavlju o *gcc* prevodiocu.

Na kraju, evo nove (i finalne) verzije *makefile*-a koja obuhvata i kreiranje biblioteke:

```
LIBRARY = quadric
PROGRAM = test
SOURCES = quadric.c
CC      = gcc
CFLAGS  = -ansi -Wall
LDFLAGS = -L.
LDLIBS  = -lm -l$(LIBRARY)

all: lib$(LIBRARY).a $(PROGRAM)

$(PROGRAM): $(PROGRAM).c lib$(LIBRARY).a
	$(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $@ $^
```

```
lib$(LIBRARY).a: $(SOURCES:.c=.o)
        ar crusv $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ -MMD $<
        @cp $*.d $*.P; \
                sed -e 's/#.*//' -e 's/^[^:]*: *//' -e 's/ *\\$$//' \
                -e '/^$$/ d' -e 's/$$/ :/' <$*.d >>$*.P; \
                rm -f $*.d

-include $(SOURCES:.c=.P)


.PHONY: beauty clean dist

beauty:
        -indent *.h *.c
        -rm -f *~

clean:
        -rm -f *.a *.o *.out *.P *.sum $(PROGRAM)

dist: beauty clean
        -tar -czv -C .. -f ../$(PROBLEM).tar.gz $(PROBLEM)
```

Treba uočiti da bi ovaj *makefile* vjerovatno u realnim uslovima bio razbijen na dva - jedan za biblioteku i jedan za test program. Izmjene u odnosu na prethodnu verziju *makefile*-a bi trebalo da su razumljive; ovdje će biti samo pomenuto da je, obzirom na prethodno poglavlje, pod `clean` ciljem dodata naredba za brisanje fajlova eventualno generisanih *gprof*-om ili *ar*-om. Za uključivanje dodatnih fajlova u biblioteku dovoljno je staviti njihova imena na kraj liste imena u promjenjivoj `SOURCES` u *makefile*-u.

Poglavlje 9

GNU CVS alat za kontrolu verzija

Tokom razvoja iole većeg softverskog projekta dolazi do situacija kada je potrebno održavati paralelno dvije ili više verzija projekta ili kada je potrebno vratiti se na prethodnu verziju projekta ili dijela projekta. Ručno održavanje verzija je moguće, ali veoma naporno, pa su zato napravljeni alati koji automatizuju taj postupak. *GNU* alat koji se koristi za ovu namjenu jeste *CVS* (*Concurrent Versions System*).

Osnovna paradigma u *CVS* sistemu jeste skladište, odn. lokacija na koju se smještaju fajlovi koji sačinjavaju projekat, pri čemu su svakom faju pridružene i dodatne informacije koje omogućavaju da se restauriše bilo koja njegova ranija verzija. *CVS* je veoma kompleksan program koji omogućava da više programera radi na istom projektu, pa čak i na istom fajlu pri čemu je spajanje izmjena maksimalno automatizovano; takođe, *CVS* omogućava da se skladište nalazi i na udaljenoj[1] lokaciji i da mu se pristupa putem *TCP/IP* veze. Međutim, podešavanje *CVS*-a za takav način rada je razmjerno komplikovan zadatak[2], tako da će na ovom mjestu biti primarno razmatrana situacija kada jedan programer koristi *CVS* na svom računaru za održavanje verzija projekta.

Prije nego što se počne sa korišćenjem *CVS*-a, mora se inicijalizovati skladište. Ovo je potrebno uraditi samo jednom, jedno skladište može da sadrži veći broj projekata. Inicijalizacija skladišta se obavlja naredbom oblika:

[1] bilo gdje na Internetu

[2] posebno ukoliko se nastoji da se pristup skladištu ostvari sigurnom konekcijom

```
cvs -d ~/cvsroot init
```

Ovdje je `cvs` naravno ime programa. Opcijom `-d` se navodi lokacija skladišta. Prilikom inicijalizacije, taj direktorijum će biti kreiran ukoliko ne postoji. Po konvenciji, koreni direktorijum skladišta se zove `cvsroot` i on se mora postaviti tamo gdje korisnik ima pravo upisa, u ovom slučaju u njegovom *home* (~) direktorijumu. Na kraju komandne linije data je *CVS* komanda - u ovom slučaju komanda `init` za inicijalizaciju skladišta.

Na sličan način se izdaju i druge *CVS* komande. Prvo uvijek ide `cvs` kao ime programa (koji u stvari u sebi sadrži više komandi), potom tzv. globalne opcije (tj. opcije koje su iste za sve komande), onda ime komande i na kraju eventualno lokalne opcije (tj. opcije koje su specifične za datu komandu). Opcija `-d` kojom se zadaje lokacija skladišta je globalna opcija i ona se obično navodi svaki put, pošto je uz svaku *CVS* komandu potrebno specifirati tu lokaciju. Alternativno, može se postaviti *environment* promjenljiva `CVSROOT` da ukazuje na lokaciju skladišta i onda se ista ne mora navoditi svaki put pri pokretanju `cvs` programa.

Prvi korak pri upotrebi *CVS*-a za kontrolu verzija nekog projekta jeste unošenje projekta u skladište, što se obavlja naredbom `import`. Kada se kreira osnovna struktura direktorijuma projekta, recimo pokaznog primjera, tada se unošenje projekta u skladište može izvršiti naredbom (pod pretpostavkom da smo pozicionirani u koreni direktorijum projekta):

```
cvs -d ~/cvsroot import -m "Unos projekta u skladiste" quadric MatF start
```

U gornjoj komandi, naredbu `import` slijedi lokalna opcija `-m` kojom se zadaje poruka koja ostaje sačuvana kao podsjetnik šta je prilikom datog slanja u skladište promijenjeno[3], potom slijedi ime direktorijuma u skladištu u kome će biti čuvan projekat (ovaj direktorijum biva kreiran ispod korenog direktorijuma skladišta), zatim nešto što treba da ukaže na organizaciju koja je vlasnik projekta i na kraju obavezno treba da stoji `start`.

Nakon što se unese projekat u skladište, može se obrisati direktorijum koji je za njega kreiran. Naime, projekat je sada smješten u skladištu i dalji rad sa njime treba da se počne kreiranjem radnog direktorijuma projektu na osnovu one verzije projekta koja je unesena u skladište.

[3]ako se ova poruka ne zada u komandnoj liniji, *CVS* će po izdavanju komande otvoriti editor očekujući da se poruka unese na taj način

Dovlačenje radne verzije projekta iz skladišta vrši se naredbom `checkout`. Pod pretpostavkom da smo pozicionirani u direktorijumu u kome želimo da napravimo radnu kopiju, ova operacija bi se izvršila naredbom:

```
cvs -d ~/cvsroot checkout quadric
```

Ovdje je `checkout` naredba za kreiranje radne kopije, dok je `quadric` ime direktorijuma u skladištu koji sadrži projekat na kome želi da se radi. Nakon ovoga u tekućem direktorijumu biće kreiran direktorijum `quadric` sa radnom kopijom projekta u koji se može ući i normalno raditi sa projektom. Treba primijetiti da će ovaj direktorijum, kao i svi njegovi poddirektorijumi, sadržati pored originalnih fajlova i poddirektorijuma i poddirektorijum pod imenom `CVS` - u pitanju su poddirektorijumi u koje *CVS* smješta svoje administrativne fajlove i koji se ne smiju mijenjati ili brisati.

Po kreiranju radne kopije se, kako je rečeno u prethodnom pasusu, normalno radi sa njome. Nakon što su unesene određene izmjene, može se ažurirati verzija projekta u skladištu. Pretpostavimo da je promijenjen fajl `quadric.c`, kao i *makefile*, u tom slučaju bi slanje izmjena bilo izvršeno naredbom (pod pretpostavkom da se nalazimo u korenom direktorijumu radne kopije projekta):

```
cvs -d ~/cvsroot commit -m "Razne izmjene" quadric.c Makefile
```

Ovdje je `commit` ime naredbe za slanje izmjena, opcijom `-m` se opet navodi propratna poruka (koja naravno treba da je mnogo informativnija nego kako gore izgleda), a potom imena fajlova koji se ažuriraju. Alternativno, komanda je mogla da se zada i na sledeći način:

```
cvs -d ~/cvsroot commit -m "Razne izmjene" .
```

Naime, `commit` komanda, kao i većina drugih *CVS* komandi, je rekurzivna, pa kada se izda nad nekim direktorijumom onda će ona biti izvršena nad tim direktorijumom, svim njegovim fajlovima i rekurzivno nad svim njegovim poddirektorijumima. U gornjem primjeru, *CVS* bi tokom tog rekurzivnog izvršavanja `commit` naredbe odredio da su samo fajlovi `quadric.c` i `Makefile` izmijenjeni, pa bi samo njih i ažurirao.

Kada se želi da se ažurira radna verzija projekta na osnovu onog što se nalazi u skladištu, primjenjuje se `update` komanda. Ovu komandu ima smisla koristiti samo kada na projektu radi više programera, jer ako na njemu radi samo jedan programer, onda ne postoji mogućnost da u skladištu ima fajlova koji su ažurniji od onih u radnoj kopiji. Ažuriranje radne kopije bi u datom primjeru bilo izvršeno naredbom:

```
cvs -d ~/cvsroot update .
```

Ukoliko na projektu radi više programera, onda rad treba striktno da se odvija tako što će se svaki put kada hoće da se nastave aktivnosti na projektu ažurirati radna kopija na osnovu sadržaja skladišta naredbom `update`, da bi se potom radilo na ovoj radnoj kopiji, pa onda jednom kada se završi sesija sa projektom sadržaj skladišta se ažurira na osnovu radne kopije naredbom `commit`[4]. Redovno korišćenje `update` i `commit` naredbi je veoma važno kada na projektu radi više programera, na taj način se ažurno prati šta drugi rade i rano otkrivaju eventualne simultane izmjene na istom fajlu.

U podrazumijevanom režimu rada, *CVS* dozvoljava da više programera pravi izmjene na istom fajlu. *CVS* može da radi i u ekskluzivnom režimu, sa zaključavanjem fajlova, ali za tim najčešće nema potrebe. Kada se po primjeni naredbe `update` pokaže da je fajl na kome se trenutno radi neko u međuvremenu već izmijenio i poslao u skladište, *CVS* će pokušati automatski da poveže te izmjene sa izmjenama koje su napravljene na radnoj kopiji. Ukoliko ova operacija uspješno prođe, programer neće ni primijetiti da je bilo simultanog editovanja; u suprotnom, *CVS* opet spaja dvije verzije fajla naznačavajući jasno gdje postoje konflikti i ostavljajući programeru da ih sam riješi, nakon čega treba `commit` naredbom da pošalje finalnu verziju u skladište. Ovaj mehanizam praktično u većini situacija savršeno funkcioniše, tako da rijetko ima potrebe za prelaskom na režim rada sa zaključavanjem fajlova.

Da bi se naznačilo da neće više da se radi na projektu koji je preuzet iz skladišta, koristi se naredba `release`. Na pokaznom primjeru, ova operacija se može izvesti naredbom (nakon što se prvo pozicioniramo u direktorijum koji je iznad korenog direktorijuma radne kopije):

```
cvs -d ~/cvsroot release -d quadric
```

Ovdje je `release` naravno ime komande, lokalna opcija `-d` (treba uočiti da nosi istu oznaku kao i globalna opcija za specificiranje lokacije skladišta,

[4]obično se ovo radi na dnevnoj bazi, tj. na početku radnog dana se ažurira radna kopija na osnovu sadržaja skladišta, da bi se na njegovom kraju opet ažuriralo skladište na osnovu radne kopije

ali da nema konflikta jer ih razlikuje pozicija u komandnoj liniji) označava da treba obrisati radnu kopiju, dok se na kraju komadne linije navodi ime direktorijuma koji sadrži radnu kopiju projekta. Po izdavanju ove komande, *CVS* provjerava da li su svi fajlovi iz radne kopije ažurirani u skladištu i ispisuje odgovarajuću poruku tražeći da se potvrdi ili poništi namjeravana akcija. Ako dobije potvrdu, *CVS* u skladištu označava da na datom projektu više niko ne radi i briše radnu kopiju.

Svaku verziju nekog fajla koja se šalje u skladište *CVS* označava brojem, počev od 1.1 pa nadalje. Ovi brojevi se označavaju kao brojevi revizija i nemaju veze sa verzijama projekta (u zavisnosti od toga koliko često se radi na pojedinim fajlovima iz projekta biće različiti i njihovi brojevi revizija). Postavljanje oznake verzije na pojedine fajlove ili češće na čitav projekat označava se u *CVS* terminologiji *tag*-ovanjem. Naredba kojom bi se dodijelio *tag* recimo imena `quadric-1_0_0` čitavom projektu bi glasila:

```
cvs -d ~/cvsroot tag quadric-1_0_0 .
```

Tag se dodjeljuje direktno tekućoj verziji projekta u skladištu, a ne onome što se nalazi u radnom direktorijumu, pa o tome treba voditi računa ako između radnog direktorijuma i skladišta postoje razlike.

Komanda `checkout` ima lokalnu opciju `-r` koja omogućava da se navede ime verzije projekta na osnovu koje će biti kreirana radna kopija. Ukoliko ova opcija nije navedena, komanda naravno kreira radnu kopiju na osnovu poslednjih revizija fajlova koji sačinjavaju projekat.

Porukama koje su prilikom slanja u skladište pridružene pojedinim revizijama fajla može se pristupiti `log` naredbom. Tako bi poruke koje su pridružene fajlu `quadric.c` (uz još neke dodatne informacije) bile izlistane naredbom:

```
cvs -d ~/cvsroot log quadric.c
```

Tokom rada na projektu svakako će doći do situacije da neke fajlove ili direktorijume treba dodati u projekat ili obrisati iz njega. Ovo se radi naredbama `add` odn. `remove`. Da bi fajl ili direktorijum bio dodat u skladište, isti treba da postoji u radnoj kopiji; da bi bio obrisan iz skladišta isti treba da bude obrisan iz radne kopije. Ovim naredbama se samo naznačava *CVS*-u da treba da unese odgovarajuće izmjene u skladište, stvarne operacije će biti obavljene pri narednom izdavanju `commit` naredbe. Za razliku od većine drugih *CVS* naredbi, naredbe `add` i `remove` nisu rekurzivne, što znači da treba da se izdaju za svaki fajl ponaosob. Ako se npr. odluči da se u biblioteku doda rješavanje linearne jednačine i isto implementira u fajlovima `linear.h`, odn. `linear.c`, tada bi se unošenje ovih fajlova u skladište sprovelo naredbama (pod pretpostavkom da se nalazimo u korenom direktorijumu radne kopije):

```
cvs -d ~/cvsroot add linear.h linear.c
cvs -d ~/cvsroot commit -m "Dodato rjesavanje linearne jednacine" .
```

Tekuća verzija fajla iz radne kopije se može u svakom trenutku uporediti sa verzijom koja se nalazi u skladištu naredbom `diff`. Ako hoćemo recimo da uporedimo tekuću verziju *makefile*-a sa onom iz skladišta, odgovarajuća komanda bi glasila:

```
cvs -d ~/cvsroot diff Makefile
```

Nakon ovoga će u formatu *UNIX diff* programa za poređenje fajlova biti ispisane razlike.

Jednom kada se *tag*-ovima označe verzije projekta, *CVS* pruža mogućnost razdvanjanja razvoja projekta na dvije ili više grana (pri čemu je svaka od njih zasnovana na drugoj verziji), kao i kasnijeg spajanja tih grana. Pretpostavimo da je nakon kreiranja stabilne verzije 1.0.0 projekta iz pokaznog primjera započet rad na dodavanju rješavanja linearnih jednačina u biblioteku i da je u jednom trenutku stablu projekta dodijeljena verzija 1.1.0. Pretpostavimo dalje da je otkrivena greška u `quadric` funkciji i da se želi ista odmah ispraviti i načiniti raspoloživom korisnicima. Recimo da je ispravka trivijalna; međutim, pošto se intenzivno radi na novim opcijama u biblioteci, glavna verzija 1.1.0 projekta nije stabilna i najbolje bi bilo nekako napraviti ispravku na verziji 1.0.0 i tu ispravljenu verziju napraviti raspoloživom korisnicima, a onda ispravku uključiti i u glavnu verziju. *CVS* omogućava da se ovo elegantno riješi pomoću grananja. Prvo je potrebno kreirati radnu kopiju na osnovu verzije 1.0.0 naredbom:

```
cvs -d ~/cvsroot checkout -r quadric-1_0_0 quadric
```

Nakon toga treba ući u radni direktorijum i kreirati grananje, što se izvodi `tag` naredbom sa `-b` opcijom:

```
cvs -d ~/cvsroot tag -b quadric-1_0_1
```

Dalje treba obrisati radnu kopiju verzije 1.0.0 i kreirati radnu kopiju na osnovu verzije 1.0.1 (prethodno se naravno treba pozicionirati u direktorijum iznad direktorijuma sa radnom kopijom):

```
cvs -d ~/cvsroot release -d quadric
cvs -d ~/cvsroot checkout -r quadric-1_0_1 quadric
```

Nakon ovoga se normalno radi sa radnom kopijom - sve izmjene biće unošene u granu 1.0.1 i neće ničim uticati na glavnu granu 1.1.0 razvoja projekta. Kada se želi da se ispravke napravljene u verziji 1.0.1 uključe u glavnu granu projekta (tj. da se spoje razdvojene grane projekta), to se jednostavno može uraditi `-j` (od *join*) opcijom `update` naredbe. Pod pretpostavkom da se nalazimo u korenom direktorijumu radne kopije kreirane na osnovu glavne 1.1.0 verzije projekta, povezivanje bi se izvršilo naredbom:

```
cvs -d ~/cvsroot update -j quadric-1_0_1 .
```

Ukoliko se prilikom povezivanja javi neki konflikt, isti će biti razrešavan na način koji je već opisan ranije.

emacs ima ugrađenu podršku za rad sa određenim brojem *CVS* komandi. U `Tools` meniju *emacs*-a postoji `Version Control` podmeni koji sadrži odgovarajuće stavke. Kada je fajl koji se edituje u *emacs*-u dio radne kopije, u statusnoj liniji je ispisana oznaka sistema za kontrolu verzija[5], u ovom slučaju *CVS*-a, kao i broj revizije fajla. Između ove dvije oznake stoji `-` ukoliko verzija fajla na disku nije modifikovana u odnosu na verziju u skladištu, odnosno `:` ukoliko jeste modifikovana. Osnovna komanda za rad sa *CVS*-om u *emacs*-u jeste `C-x C-q` koja odgovara `commit` naredbi *CVS*-a. Po izdavanju ove naredbe, verzija tekućeg fajla u skladištu se ažurira prema sadržaju fajla u radnoj kopiji. Pritom, prvo se otvara potprozor u kome se unosi propratna poruka i tek pošto se u ovom prozoru izda komanda `C-c C-c` fajl biva odaslat u skladište. U potprozoru za unos propratne poruke može se koristiti i komanda `M-n` kojom se unosi ista propratna poruka koja je korišćena za prethodni fajl. Ukoliko je eventualno fajl koji se komandom `C-x C-q` namjerava poslati u skladište tamo već izmijenjen onda će *emacs* umjesto `commit` *CVS* naredbe izvršiti `update` naredbu i na taj način uključiti izmjene iz skladišta u tekući fajl.

Nazivi komandi koji se odnose na kontrolu verzija se unekoliko razlikuju od naziva komandi *CVS*-a. Tako se npr. komanda za dodavanje fajla odn. direktorijuma u skladište u *emacs*-u označava kao `Register` komanda, dok je naziv za istu komandu u *CVS*-u `add`. To je zato što je odgovarajući podsistem *emacs*-a generički, tj. kako je već pomenuto podržava pored *CVS*-a i druge sisteme za kontrolu verzija. Ipak, većina komandi *CVS*-a je prisutna,

[5] *emacs* podržava i druge sisteme za kontrolu verzija pored *CVS*-a

a takođe postoji poseban režim rada sa direktorijumima, koji se aktivira komandom `C-x v d` i u kome je moguće *CVS* operacije obaviti nad više fajlova istovremeno. Naravno, sve komande su detaljno opisane u *emacs* dokumentaciji, pa istu treba konsultovati u slučaju nejasnoća. Pored ovog osnovnog režima za rad za sistemom za kontrolu verzija koji se uvijek isporučuje uz *emacs*, postoji i dodatak koji je bolje prilagođen radu sa *CVS*-om i koji se naziva *pcl-cvs*, pa se isti po potrebi može instalirati i koristiti na sličan način kao osnovni režim.

Ovim poglavljem se završava pregled *GNU* programerskih alata, na osnovu koga bi trebalo da se stekla predstava o tome koliko je *UNIX*, opremljen ovim alatima, moćno okruženje za programiranje[6]. Štaviše, svi ovi alati se kontinualno unapređuju i, što je najvažnije od svega, pravili su ih i održavaju ih programeri za programere, što obezbeđuje da ovi alati jesu i da će biti među najproduktivnijim postojećim sistemima za obavljanje poslova vezanih za programiranje.

[6]treba imati na umu da jedan broj *GNU* alata nije predstavljen u ovom pregledu: npr. *flex* i *bison* alati za generisanje parsera ili *autoconf* i *automake* alati za automatsko konfigurisanje projekta, odnosno kreiranje *makefile*-ova