ELEKTROTEHNIČKI FAKULTET

UNIVERZITETA U BEOGRADU

**Jednoprolazni asembler**

Domaći zadatak iz predmeta sistemsko programiranje

Student: Dejan Golubović 321/2012

Mentor: Saša Stojanović

Beograd, jun 2015

**Opis problema**

Cilj ovo projekta bio je napraviti jednoprolazni asembler. Na ulazu se dobija tekstualna datoteka koja koju treba obraditi i napraviti izalznu datoteku koja se prosleđuje linkeru. U toj izlaznoj datoteci bi trebalo da se nalaze sve informacije potrebne linkeru da izvrši povezivanje. Na izlazu je potrebno generisati izgled tabele simbola, zapise o relokacijama u text i data sekcijama i izgled samih text i data sekcija.

Problem kod jednoprolaznog asemblera je to što se ne može dva puta proći kroz kod i tako razrešiti situacije kada je simbol prvo korišćen a tek kasnije definisan, već se mora rešiti na drugi način. Rešeno je tehnikom zvanom backpatching, gde se za svaki simbol pamte sve pozicije na kojima je referenciran. Nakon prolaska kroz ulaznu datoteku, prođe se kroz sve simbole iz tabele simbola i izvrše se popravke na mestima gde je simbol bio korišćen. Ako je simbol definisan, neće biti problema. Takođe, tu se prave i zapisi o relokacijama.

**Klasa Symbol**

Klasa Symbol predstavlja jedan ulaz u tabeli simbola. Simbol kao polja ima svoje ime (string name), vrednost (int value), redni broj (int id), sekciju kojoj pripada (u kojoj je definisan, ili UND ako je globalan i nedefinisan)(int section), zatim fleg koji označava da li je definisan (bool defined) i identifikaciju da li je lokalni ili globalni (char scopeId). Takođe sadrži i listu svih obraćanja unapred (List<FlinkEntry> \*flink). Jedan ulaz te liste je tipa FlinkEntry koji je opisan u daljem tekstu. Ta lista se popunjava kada se naiđe na korišćenje simbola koji nije definisan, i služi da ne bismo morali da radimo drugi prolaz pri asembliranju.

Od metoda, klasa sadrži metode za dohvatanje i postavljanje novih polja, i konstruktore. Specifičan konstruktor je Symbol(string s) koji nam samo služi da napravimo novi simbol sa određenim imenom. To se koristi samo kod pretrage u tabeli simbola, jer se poređenje dva simbola vrši samo na osnovu imena, preklopljen operator =.

Takođe klasa ima statička polja lastId i lastSection, koja nam služe prilikom generisanja rednih brojeva simbola.

**Klasa FlinkEntry**

Klasa FlinkEntry predstavlja jedan ulaz u listi obraćanja unapred. Od polja sadrži poziciju na kojoj treba izvršiti relokaciju, tip relokacije, zapis instrukcije ili direktive koja je zahtevala relokaciju i sekciju u kojoj je zahtevana relokacija. Od metoda ima one koje služe za dohvatanje i postavljanje opisanih polja.

**Klasa TableOfSymbols**

Tabela simbola je jedna od fundamentalnih klasa ovog projekta. U njoj se nalaze svi simboli koji su se pojavili u ulaznom fajlu. Ova klasa ih čuva, dodaje, menja, pronalazi, i na kraju razrešava relokacije koje su se u njima pojavile. Ona je zapravo stub svega ostalog i sve ostale klase će stalno imati interakciju s njom. Njene metode su stoga uglavnom statičke, a od polja sadrži samo listu simbola i naziv trenutne sekcije.

Sadrži metode za pretragu, umetanje, menjanje simbola u tabeli, koje zapravo obmotavaju metode liste. Zatim, sadži metodu za ispis svih simbola, koja sve simbole upisuje u određeno polje klase WriterOfEverything.

***Metoda backpatch()***

Na kraju, najkompleksnija metoda ove klase je metoda backpatch(). Ona služi za razrešavanje relokacija i prepravku odgovarajućih lokacija u text i data sekcijama. Algoritam rada ove metode je sledeći. Pravimo petlju i prolazimo kroz sve simbole liste. Za svaki simbol na početku proverimo da li je definisan ili globalan. Ako nije ni definisan ni globalan, vraćamo grešku da smo naišli na nedefinisan simbol. U suprotnom, idemo dalje sa obradom.

Za svaki simbol, prolazimo kroz njegovu listu obraćanja unapred, ako je ima. Za početak, pravimo zapis o relokaciji. On će biti napravljen samo u slučaju da imamo apsolutni tip relokacije, ili relativni gde simbol nije definisan u istoj sekciji u kojoj je korišćen. Ako smo prošli taj uslov, idemo dalje. Ako je simbol globalni, referisaćemo sam simbol u zapisu o relokaciji. Ako je lokalni, referisaćemo sekciju u kojoj je definisan. Proveravamo sekciju u kojoj je koriščen simbol. Ako je text sekcija, dodajemo zapis u rel.text sekciji izlaznog fajla, ako je data, dodajemo u rel.data sekciji. Zatim gledamo koji je tip relokacije, i u zavisnosti od njega vršimo popravku koda u text i data sekcijama na izlazu.

Tip 1 je za instrukciju jmp. Tu imamo relativni pomeraj. Ako je simbol lokalni upisujemo (vrednost\_simbola - 4), ako je globalni, upisujemo (-4). I izvršimo popravku u text sekciji na mestu određenom poljem counterText iz FlinkEntry objekta na kojem smo trenutno.

Tip 2 je za instrukcije uslovnog skoka. Tu imamo relativni pomeraj. Ako je simbol lokalni upisujemo (vrednost\_simbola - 4), ako je globalni, upisujemo (-4). I izvršimo popravku u text sekciji na mestu određenom poljem counterText iz FlinkEntry objekta na kojem smo trenutno.

Tip 3 je za direktivu .long. Tu imamo apsolutni skok. Ako je simbol lokalni upisujemo vrednost\_simbola, ako je globalni, upisujemo 0. Zatim proverimo da li je simbol referisan u data ili text sekciji i u skladu s tim vršimo popravke na izlazu.

Tip 4 je za instrukciju ldch. Tu imamo apsolutni skok. Ako je simbol lokalni upisujemo vrednost\_simbola, ako je globalni, upisujemo 0. I izvršimo popravku u text sekciji na mestu određenom poljem counterText iz FlinkEntry objekta na kojem smo trenutno.

Tip 5 je za instrukciju ldch. Tu imamo apsolutni skok. Ako je simbol lokalni upisujemo vrednost\_simbola, ako je globalni, upisujemo 0. I izvršimo popravku u text sekciji na mestu određenom poljem counterText iz FlinkEntry objekta na kojem smo trenutno.

Nakon svega opisanog, metoda završava sa radom, i imamo razrešene sve relokacije.

**Klasa Instruction**

Klasa instrukcija je svojevrsna pomoćna klasa za generisanje izlaza. Napravljena je da bi se delegirala odgovornost klase Line. Njen glavni cilj je da kodira zadatu instrukciju. Od polja, ima statički niz stringova u kojem se nalaze imena svih instrukcija. Takođe imamo i 8 stringova koji služe da sačuvamo podatke o instrukciji za upis. Ako je instrukcija dupla (ldc), upisaćemo svih 8, ako nije (uglavnom nije), upisujemo 4 bajta.

Prilikom kodiranja instrukcija koristi se njihov redni broj u tom nizu. Stoga, imamo medote za pronalazak zadata instrukcije i vraćanje njenog rednog broja, zatim za kodiranje instrukcije i upis u polja, i na kraju metoda za upis instrukcije u WriterOfEverything.

Prilikom instrukcija uslovnog skoka, jmp, ldch i ldcl, ova klasa kreira dodaje i zapise u listi obraćanja unapred, ako je potrebno, ako su instrukcije sa labelom. To se radi u metodi readInstruction(). Ako se labela nalazi u tabeli simbola, dodajemo novi FlinkEntry za taj simbol. U suprotnom kreiramo novi simbol i dodajemo FlinkEntry.

**Klasa List**

Klasa List obezbeđuje funkcionalnosti liste. Moguće je dodavanje elemenata na početak, kraj ili nakon određenog elementa, kao i brisanje i pretraživanje liste. Klasa je realizovana kao generička, pa se koristi na dva mesta u projektu, u svakom simbolu je lista FlinkEntry tipa, i u TableOfSymbols lista simbola.

**Klasa Line**

Klasa Line nam je klasa koja možda radi i najveći posao u našem projektu. Njen cilj je da za dobijeni ulazni string generiše odgovarajuće zapise u klasi WriterOfEverything.

Od polja ima LC koji nam služi da znamo na kojo smo poziciju unutar trenutne sekcije. Zatim string line što je zapravo linija koju treba obraditi, pa fleg isEnded, koji označava da li smo došli do kraju ulaznog fajla.

***Metoda performLine(string s)***

Ova klasa koristi metodu performLine da prihvati ulazni string izgeneriše izlaz. Ovoj metodi se prosleđuje cela linija pa je potrebno pročitati prvu reč te linije i odrediti da li se radi u instrukciji, sekciji ili direktivi, i kojoj. Za svaku situaciju postoje posebne metode i one se pozivaju u ovoj metodi.

***Metoda performData()***

Ako prva reč iz linije počinje sa .data, znači da imamo data sekciju. Proverimo da li je simbol sa tim imenom već u tabeli simbola, ako jeste, vraćamo grešku zbog duple definicije simbola. Ako nije, dodajemo nov simbol u tabelu simbola, i u klasi tabela simbola menjamo currentSection na ima te sekcije. Na analogni način rade metode performText() i performBss().

***Metoda performSkip()***

Obrađuje .skip direktivu. Uvećava LC za zadati broj i upisuje u odgovarajuću sekciju odgovarajući broj nula.

***Metoda performLong()***

Obrađuje .long direktivu. Pročitamo operanda i proverimo da li je broj.

Ako jeste broj, upišemo taj broj u odgovarajuću sekciju.

Ako nije broj, tražimo simbol sa tim imenom u tabeli simbola. Ako ga pronađemo, dodajemo mu novi FlinkEntry. Ako ga nismo pronašli, dodajemo ga u tabelu simbola i dodajemo mu FlinkEntry. Onda upišemo vrednost tog simbola u WriterOfEverything, svakako će doći do relokacije, sa tipom 3.

Metode performWord() i performChar() su predviđene da se u njima ne mogu naći labele, pa kod njih izvršavamo samo prvi deo prethodno opisanog algoritma.

***Metoda performLabel()***

Pročitamo naziv labele. Ako je samo labela u liniji, idemo dalje. Ako nije, čitamo ostatak linije i obrađujemo direktivu, instrukciju, ili sekciju. Dve labele u liniji su greška. Na kraju, proveravamo da li se labela nalazi u tabeli simbola. Ako se nalazi i definisan je, to je greška, dupla definicija simbola. Ako se nalazi i nije definisan, definišemo ga. Ako se ne nalazi, dodajemo labelu (simbol), u tabelu simbola.

***Metoda performAlign()***

Ako LC nije deljiv sa operandom, povećavamo ga da bude deljiv.

***Metoda performGlobal()***

Čitamo labele iza direktive .global. Za svaku radimo sledeće.

Ako je u tabeli simbola i definisan je, postavimo mu scopeId na global. Označimo ga da je globalan. Ako je u tabeli simbola i nije definisan, dodajemo mu FlinkEntry, jedan zapis u listi obraćanja unapred. Takođe proverimo da li postoji UND sekcija, i ako postoji, taj simbol pripada njoj, ako ne postoji, kreiramo je, dodajemo je kao simbol u tabeli simbola i označimo da simbol pripada njoj.

Ako nije u tabeli simbola, proverimo da li postoji UND sekcija, i ako postoji, taj simbol pripada njoj, ako ne postoji, kreiramo je, dodajemo je kao simbol u tabeli simbola i označimo da simbol pripada njoj.

Analogno za performExtern().

**Klasa WriterOfEverything**

Klasa WriterOfEverything služi nam da sačuva sve podatke koje treba upisati u izlazni fajl. Ona je zapravo posrednik između svih ostalih klasa u projektu i samog izlaznog fajla. Sve klase upisuje u ovu klasu. Tek nakon što se uradi sav posao u projektu, ova klasa upisuje svoja polja u izlazni fajl.

Od polja, ima statička polja:

1) string text - zapis text sekcije, heksadecmalne vrednosti

2) string data - zapis data sekcije, heksadecmalne vrednosti

3) string relText - zapisi o relokacijama u text sekcijama

4) string relData - zapisi o relokacijama u data sekcijama

5) string tableOfSymbols - tabela simbola

6) string error - zapis o grešci

7) int counterText - lokacija na kojoj se trenutno nalazimo u text sekciji

8) int counterData - lokacija na kojoj se trenutno nalazimo u data sekciji

I statička polja errorCode i errorLine, koja nam služe za obradu greške.

Od metoda ima one za menjanje, dodavanje, brisanje iz odgovarajućih sekcija. Kao i za inkrementiranje odgovarajućih brojača. Ima metodu za postavljanje greške. I za proveru da li ima greške. Na kraju, ima metodu za upis koja vraća string koji sadrži sve ostale stringove spojene.

**Glavni program**

U glavnom programu, koji nije preterano opširan, rade se jednostavne operacije. Za početak se pročita fajl čije je ime uneto u komandnoj liniji. Onda se na osnovu tog imena kreira izlazni fajl sa dodatkom \_o. Nakon što otvorimo fajl, citamo ga u petlji liniju po liniju i tu procitanu liniju (string), saljemo metodi performLine(string s) klase Line. Tu se izvršava obrada linije, u sekciji Klasa Line opisan je način rada metode performLine. Ako dođe do greške, prekidamo čitanje fajla i petlju. Takođe u petlji inkrementiramo globalni brojač lineNumber, da bismo znali u kojoj liniji je došlo do greške (ako do nje dođe). Nakon čitanja i obrade svih linija, pozivamo metod backpatch() klase TableOfSymbols. Ta metoda nam razrešava zapise o relokacijama. Zatim, ako nije bilo greške, upisujemo tabelu simbola u izlazni fajl. Onda upišemo i ostalo (ako ima greške, grešku, ako nema upisujemo text i data sekciju i zapise o relokacijama). Na kraju, zatvaramo ulazni i izlazni fajl.

**Testovi**

***Test1.txt***

1) Sve tipove relokacija. Instrukcije jmp, je, ldc koje generišu relokacije, kao i direktivu .long

2) Kako se instrukcija ldc kodira kao ldch i ldcl

3) Direktivu .skip

4) Rad sa tri različite sekcije.

5) Pokazujemo kako se vrše relokacije i u text i u data sekciji

6) Labelu u praznoj liniji, i labelu sa direktivom nakon nje

7) Globalne simbole, tj. direktivu .global na proizvoljnom mestu u kodu

8) Relativni skok unutar iste sekcije. Ne treba da se pravi zapis o relokaciji

9) Instrukciju add

***Text2.txt***

1) Sekcije istog tipa sa različitim imenima

2) Direktive .extern, .align, .word i .char

3) Instrukciju jmp sa uglastim zagradama, skok na labelu, kao i instrukciju sub

4) Rad bez .end direktive, programa radi, .end je opciono

5) Globalni simbol koji nije definisan, relokacija sa njim, sekcija UND

***Test3.txt***

U ovom testu testiramo instrukcije mov, int, shr, jmp bez labele, sa i bez uglastih zagrada

Takođe, uočavamo kako se nakon .end direktive ne čita ništa više iz fajla

***Test4.txt***

U ovom testu testiramo grešku zbog nedefinisanog simbola.