PROGRAMSKI JEZICI i PREVODIOCI

Tekst predavanja

Pripremio: Samir Ribić

Oktobar, 2019. Godine

1. Osnove

Programski jezici su svakodnevnica života softverskog inženjera. Rijetki su programeri koji cijeli svoj radni vijek provedu koristeći samo jedan programski jezik. Programski jezici se mijenjaju nekad iz razloga mode, nekad zbog različitih potreba u softverskom razvoju. Neki programski jezici su pogodniji za određene domene softvera. U mnogim projektima je potrebno kombinovati i više različitih jezika, na primjer C# i PL/SQL, ili C i asemblerski jezik.

Razlozi učenja programskih jezika

Iako većina programskih jezika ima slične osnovne koncepte (postojanje promjenjivih, potprograma, uslova, petlji), kroz komparativnu analizu programskih jezika povećava se mogućnost adekvatnog izražavanja u razvoju aplikacija. Na primjer, pointeri u jeziku C su praktično rješenje za realizaciju drajvera za periferijske uređaje, jer olakšavaju direktni pristup hardveru, dok se operacije s kompleksnim brojevima moraju realizovati bibliotekom. S druge strane, klasični Fortran ima kompleksne brojeve kao prosti tip, ali direktni pristup hardveru zahtijeva eksterne potprograme, pisane npr. u asemblerskom jeziku. Znajući ovaj podatak, čak i bez potpunog poznavanja programskih jezika, postaje jasno u kojem jeziku bi u izboru između ova dva, bi trebalo pisati sistemske, a u kojem programe za numeričke proračune. Neki jezici imaju i radikalno drugačije koncepte. Funkcionalni i logički programski jezici se orijentišu na pitanje “Šta uraditi”, više nego “Kako uraditi” i prepuštaju proces zaključivanja računaru.

Upoznavanje više programskih jezika kroz komparativnu analizu olakšava kasnije upoznavanje novih. Većina modernih programskih jezika sintaksno podsjeća na C (vitičaste zagrade, for ,if, while, switch), pa je nakon C-a lako preći na C++, Java, PHP, JavaScript ili Perl, dok linijski koncept BASIC-a vodi ka lakšem prihvatanju Fortran-a ili Python-a.

Kroz poređenje s drugim jezicima, postaju jasniji i neki koncepti u već poznatim programskim jezicima. Zašto Java, iako objektno orijentisani jezik zamišljen da bude jezik visokog nivoa ima tako kriptičnu sintaksu? Odgovor je istorijski: zbog uticaja jezika C.

Primjene

Različiti programski jezici su nastajali u cilju pokrivanja različitih oblasti razvoja softvera. Računari su se prvo koristili u vojsci i naučnim institutima za rješavanje **numeričkih problema**. Ovakve aplikacije su zahtijevale uvođenje računa u pokretnom zarezu uz upotrebu nizova i potprograma i prvi jezik koji je omogućio njihov efikasan razvoj je Fortran**. Poslovne aplikacije** se koncentrišu na pravljenje izvještaja,. Kod ovakvih aplikacija ne koristi se pokretni zarez, jer finansijski izvještaji nemaju smisla sa vrijednostima manjim od jednog centa, nego su uvedeni decimalni brojevi s fiksnom tačkom i rad s znakovima. Prvi predstavnik jezika koji zadovoljavaju ove zahtjeve je COBOL, a u novije vrijeme PL/SQL . Oblast **vještačke inteligencije** se uglavnom bavi manipulacijom simbolima a ne brojevima. Rani jezi koji simbole drži kroz ulančane liste je LISP. Za **predstavljanje algoritama** programski jezik treba da stimuliše čitljiv kod, pa su nastali Algol i Pascal. **Sistemsko programiranje** zahtijeva efikasnost zbog stalne upotrebe i tu se pokazao uspješnim C. **Veliki projekti** doveli su do potrebe ponovne upotrebljivosti programa, što je olakšano objektnim jezicima

poput C++ i Java. **Web Softver** se bazira na kolekcija jezika za označavanje (HTML), serversko skriptiranje (PHP), klijentsko skriptiranje (JavaScript). Jezici **opšte namjene** nisu strogo vezani za određenu oblast primjene, nakon manje uspješnih PL/1 i Ada, primjenu su stekli C++, Java, Delphi i drugi.

Kriteriji ocjene jezika

Pri ocjeni programskih jezika, koristiće se sljedeći kriteriji. Čitljivost, Mogućnost pisanja, Pouzdanost: je usaglašenost s svojim specifikacijama. Cijena je ukupna cijena upotrebe programskog jezika

Čitljivost

Čitljivost je lakoća kojom se jezik čita i razumije. Ima puno parametara koji utiču na čitljivost.

Jednostavnost jezika je jedan od parametara koji utiču na čitljivost. Jednostavnost se postiže pravilnim **izborom skupa mogućnosti i konstrukcija.** Ako jezik ima previše mogućnosti (C++) a čitalac i pisac aplikacije poznaju različite podskupove jezika, program će biti težak za čitanje. Slično, ako je mogućnosti premalo, (asemblerski jezik) program će opet biti slabo čitljiv jer se konstrukcije moraju realizovat s velikim brojem naredbi. **Dupliciranа funkcionalnost** negativno utiče na jednostavnost. Pascal, npr, ima samo jedan način za uvećanje varijable za 1 (a:=a+1), dok u jeziku C ih ima čak četiri. **Preopterećenje operatora** takođe negativno utiče na jednostavnost, npr. znak + je logičan simbol za sabiranje brojeva i spajanje stringova, ali prilično loš izbor operatora za množenje kompleksnih brojeva.

Ortogonalnost je sljedeći od parametara koji utiču na čitljivost. Jezici imaju mali skup primitivnih konstrukcija koje se kombinuju na mali broj načina. Jezik je savršeno ortogonalan ako je svaka moguća kombinacija legalna i predvidljiva. Primjeri koji narušavaju ortogonalnost su nemogućnost funkcija u Pascal-u da vrate vrijednost koja nije prosti tip, ili različito ponašanje a=a+2 naredbe u C-u zavisno od toga da li je varijabla cijeli broj ili pointer. Forth je vjerovatno najortogonalniji jezik jer njegovi potprogrami i varijable imaju istu sintaksu, a imena im mogu sadržati bilo koji znak (no ovaj jezik ne spada u čitljive zbog ograničenog broja tipova podataka).

Podržani tipovi podataka su bitan parametar koji doprinosi čitljivosti. Stariji programski jezici imaju samo ugrađene tipove, a kasniji jezici mogu da definišu vlastite. Ako je skup ugrađenih tipova nedovoljan, moraju se koristiti neoptimalni tipovi (npr. C nema logički tip pa se on simulira cijelim brojem, a rani Fortran i BASIC nemaju strukture za čuvanje srodnih podataka u istoj sintaksnoj jedinici).

Sintaksni elementi utiču na čitljivost. Jedan od njih su **forme identifikatora**. Većina jezika ograničava identifikatore na skup slova i cifara, ali preveliko ograničenje otežava čitljivost. Ekstremni primjer su rane verzije BASIC-a koje ograničavaju imena varijabli na jedno slovo. **Specijalne riječi** obično ne mogu da budu varijable. U nekim slučajevima njihovo dodavanje smanjuje jednostavnost, ali povećava čitljivost (npr. jezik Ada umjesto vitičastih zagrada ili begin/end ima posebno end if i end loop**). Forma i značenje** doprinose čitljivosti ako značenje programa direktno slijedi iz sintakse. Ako značenje zavisi od konteksta, to ne doprinosi čitljivosti.

Mogućnost pisanja

Mogućnost pisanja je lakoća kojom se pišu programi. U nekim slučajevima kriteriji koji doprinose čitljivosti, **jednostavnost i ortogonalnost**, doprinose i mogućnosti

pisanja, jer je programer koji piše programski kod često čita i njegove dijelove koji su ranije napisani. Malo konstrukcija, malo primitiva i mali skup pravila za njihovo kombinovanje čine da programer može da piše u novom jeziku nakon znatno manjeg vremena učenja

Podrška apstrakciji je mogućnost definisanja kompleksnih struktura ili operacija na način da se detalji mogu ignorisati. Na primjer u C++ se struktura binarno stablo može realizovati preko proste klase koja ima dva pointera i numeričku vrijednost u svojim poljima, dok u Fortranu 77 za ovu svrhu treba koristiti tri niza. U ovom slučaju C++ je pogodniji.

Izražajnost je predstavljena kroz skup praktičnih načina za opis operacija. APL ima ugrađene operatore za matrične operacije, a Perl operatore za manipulaciju stringovima. Operator ++ u jeziku C, iako smanjuje čitljivost (duplicirana funkcionalnost) povećava mogućnost pisanja, jer se lako kuca.

Pouzdanost

Pouzdanost: je usaglašenost jezika s svojim specifikacijama pod svim uslovima.

Provjera tipova je najvažniji način povećanja pouzdanosti programskog jezika. Funkcija printf u jeziku C ne provjerava tipove parametara, i ako oni nisu usaglašeni sa prvim parametrom koji predstavlja format ispisa može doći do nepredvidivih posljedica. U jeziku Ada se provjeravaju svi tipovi podataka. Pri tome su dimenzije nizova sastavni dio tipa, pa se mnoga prekoračenja indeksa nizova detektuju već u toku kompajliranja, a ostala u toku izvršenja.

Upravljanje izuzecima omogućava hvatanje grešaka pri izvršenju i preduzimanje korektivnih mjera. C++ i Java imaju konstrukcije kojim se mogu detektovati greške pri izvršenju i preusmjeriti program na korisničku rutinu.

Aliasi predstavljaju prisustvo dva ili više različitih metoda referenciranja na istu memorijsku lokaciju. Na primjer u C-u postoji union struktura čiji svi elementi dijele isto područje memorije, dok više pointera može da pokazuje na istu lokaciju. Kako se ovim miješaju tipovi podataka, aliasi smanjuju pouzdanost.

Čitljivost i upisivost obično doprinose pouzdanosti, jer olakšavaju reprezentaciju algoritma. Jezik koji ne podržava "prirodan" način predstavljanja algoritma će zahtijevati "neprirodne" pristupe i time smanjiti pouzdanost

Cijena

Cijena je ukupna cijena upotrebe programskog jezika, mjerena u novcu i vremenu. Na cijenu utiču

- Troškovi obučavanja programera da koriste jezik
- Trošak pisanja programa, koliko je jezik blizak datoj aplikaciji.. Npr, COBOL nije dobar za trigonometrijske proračune
- Brzina kompajliranja programa
- Brzina izvršavanje programa
- Cijena samog kompajlera (neki su i besplatni)
- Pouzdanost, loša pouzdanost vodi većim troškovima
- Troškovi održavanja programa najviše zavise od čitljivosti

Ostali kriteriji ocjenjivanja

Portabilnost je lakoća kojom se programi mogu prebaciti s jedne implementacije na drugu Neki jezici su dobro standardizovani, pa je proces prebacivanja minimalan.

Generalnost je primjenljivost programskih jezika na širok opseg aplikacija koje se mogu u njima razvijati.

Dobra definisanost je kompletnost i preciznost zvanične definicije jezika

Uticaji na dizajn jezika

Dva najvažnija faktora koji su uticali na dizajn programskih jezika su računarska arhitektura i programerske metodologije

Uticaj računarske arhitekture

Dobro poznata računarska arhitektura, Von Neumannova arhitektura je uticala na nastanak imperativnih jezika. Arhitektura je toliko rasprostranjena da su ovi jezici dominantni.

Von Neumanova arhitektura pretpostavlja da su podaci i programi smješteni u memoriji. Memorija odvojena od procesora i podijeljena je u memorijske ćelije.. Podaci i instrukcije se prebacuju iz memorije u procesor a procesor izvršava instrukciju po instrukciju sljedećim algoritmom.

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Arhitektura Von Neuman je vodila ka prvim programskim jezicima visokog nivoa. Instrukcije u ovim jezicima se pišu sekvencijalno i prate kretanje programskog brojača, dok varijable modeliraju memorijske ćelije.

Uticaj programerskih metodologija

U ranim periodima računarstva, 1950-e i rane 1960-e, razvijane su proste aplikacije, uz briga o efikasnosti računara. U kasnim 1960-im,: efikasnost ljudi postaje važnija. To je zahtijevalo veću čitljivost i bolje kontrolne strukture. Metodologija razvoja "Odozgo prema dolje" uticala je na razvoj jezika koji imaju sređene programske strukture i bolju kontrolu tipova.

Kasne 1970-e: dovode do novog pogleda na metodologiju razvoja softvera. Do tada su programski jezici bili procesno orijentisani. Nakon ovoga, oni su sve više orijentisani ka podacima i toku podataka. i raniji programski jezici su imali apstrakciju programa, to jest sakrivanje nepotrebnih detalja implementacije u cilju pojednostavljenja cjeline programa, kroz potprograme. A sada se uvodi apstrakcija podataka, mogućnost definisanja korisničkih tipova podataka i operacija nad njima u istoj sintaksnoj jedinici, prvi put u jeziku Simula 67. To je dovelo do danas najpopularnije paradigme,. objektno-orijentisano programiranje , koje kombinuje apstrakcija podataka, nasljeđivanje i polimorfizam.

Kompromisi u dizajnu jezika

Poboljšanje jednog kriterija u dizajnu jezika ponekad ide na uštrb drugih. Povećanje pouzdanosti ide na uštrb cijene izvršenja. Primjer: Java zahtijeva da se svi pristupi elementima nizova provjere u opsegu što povećava vrijeme izvršenja. Nekada se bira između čitljivost i upisivosti. Primjer: APL pruža mnogo moćnih operatora i velik broj novih simbola što omogućava da se kompleksan račun obavi u kompaktnom programu po cijenu loše čitljivosti Slično, povećanje mogućnosti pisanja može da smanji pouzdanost, primjer je uvođenje pointera u C koji su najčešći uzroci rušenja programa.

Metode implementacije

Programski jezici se implementiraju u osnovi na tri načina : Kompajliranje (programi se prevode u mašinski jezik), čisto interpretiranje (programi se interpretiraju drugim programom koji se zove interpreter) i hibridni Implementacioni sistemi (kompromis između kompajlera i čistih interpretera)

Kompajliranje

Kompajliranje je prevođenje programa u višem programskom jeziku (izvorni jezik) u mašinski jezik . Osobina ovog pristupa je da prevođenje zahtijeva izvjesno vrijeme, ali izvršavanje je dosta brzo, jer su programi prevedeni u mašinski jezik. Proces kompilacije ima više faza

- Leksička analiza: prevodi znakove izvornog programa u leksičke jedinice
- Sintaksna analiza: prevodi leksičke jedinice u stablo parsiranja koje predstavlja sintaksnu strukturu programa
- Semantička analiza: generiše među-kod
- Generisanje koda: generiše se mašinski kod

Nakon kompletiranja dobija se prevedeni modul On se povezuje sa sistemskim modulima da bi se dobio modul za učitavanje u procesu **povezivanja**.

Čisti interpreter

Kod **interpretera** se dijelovi programa napisani na nekom programskom jeziku, nakon sintaksne provjere, odmah izvršavaju. Za razliku od kompajlera, ne mora cijeli izvorni program da bude prvo preveden u drugi programski jezik:

Prednost interpretera se sastoji u tome da se pojedini dijelovi programa mogu mijenjati, a da se cijeli program ne mora ponovo prevoditi. Interpreteri omogućavaju lakšu implementaciju programa, jer se greške u izvršenju se mogu lako i odmah pokazati.

Mana interpretera je u sporije izvršenje (10 do 100 puta sporije od kompajliranih programa). Često zahtijevaju više prostora jer je u RAM memoriji pored korisničkog programa potrebno imati i interpreter (mada na računarima iz 80-ih godina interpreter za BASIC je često bio ugrađen u ROM).

Čisti interpreteri se rijetko koriste za tradicionalne programske jezike višeg nivoa (C, Fortran), ali su se vratili na scenu s Web skriptnim jezicima (npr., JavaScript, PHP)

Hibridni Implementacioni sistemi

Kompromis između kompajlera i čistih interpretera postignut je hibridnim implementacioni sistemima. Kod ovih pristupa se programski jezik visokog nivoa se najprije cijeli prevodi u među-jezik, koga je brže analizirati interpreterom nego izvorni kod u datom jeziku. Nakon ovoga taj među-kod se izvršava na jedan od sljedećih načina.

Bytecode izvršavanje, primijenjeno u jezicima Perl i ranijim verzijama jezika Java, izvršavaju programe za virtualnu mašinu koja ima instrukcije kao i stvarni mikroprocesori, ali čiji je skup instrukcija više prilagođen izvornom jeziku i dizajniran za lakšu analizu. Byte code, pruža portabilnost na bilo koju mašinu koja ima byte code interpreter i run-time system (zajedno se zovu Java Virtual Machine)

Just-in-Time Implementacija u prvoj fazi radi jednako. U početku se prevede program u među-jezik. U trenutku učitavanja, među-jezik. se prevede u potprograme mašinskog koda koga poziva. Verzija u mašinskom kodu se čuva u memoriji za kasnija pozivanja, tako da osim vremenske zadrške na startu daljnje izvršavanje dobijamo performanse izvršenja uporedive s kompajliranjem. Ovi sistemi se koriste za Java programe i .NET jezike.

Preprocesori

Preprocesorski makroi (instrukcije) se koriste da se navede da će se uključiti kod iz druge datoteke. Preprocesor obrađuje program neposredno prije kompilacije za ekspanziju makronaredbi. Poznati primjer je C preprocesor koji ekspanduje #include, #define, i slične naredbe

Programska okruženja

Programska okruženja su skup alata za softverski razvoj. Mogu se sastojati od editora, kompajlera i linkera ali i uključivati širog spektar alata dostupan kroz jedinstveni korisničko interfejs. Primjera za okruženja ima više.

U slučaju UNIX-a, uz sam operativni sistem dolazi i kolekcija razvojnih alata. Iako generalno orijentisan ka komandnoj liniji. sada je korišten kroz GUI (npr., CDE, KDE, ili GNOME) iznad UNIX

Microsoft Visual Studio.NET je veliko, kompleksno vizuelno okruženje. Korišten za gradnju Web aplikacija ali i onih koje nisu vezane za njega .NET jezicima

NetBeans i Eclipse su srodni Visual Studio .NET, namijenjeni za aplikacije u Java

Rezime

Studija programski jezika je važna iz sljedećih razloga :povećanje kapaciteta za upotrebu raznih konstrukcija, omogućuje inteligentnih izbor jezika omogućava lakše učenje novih jezika.

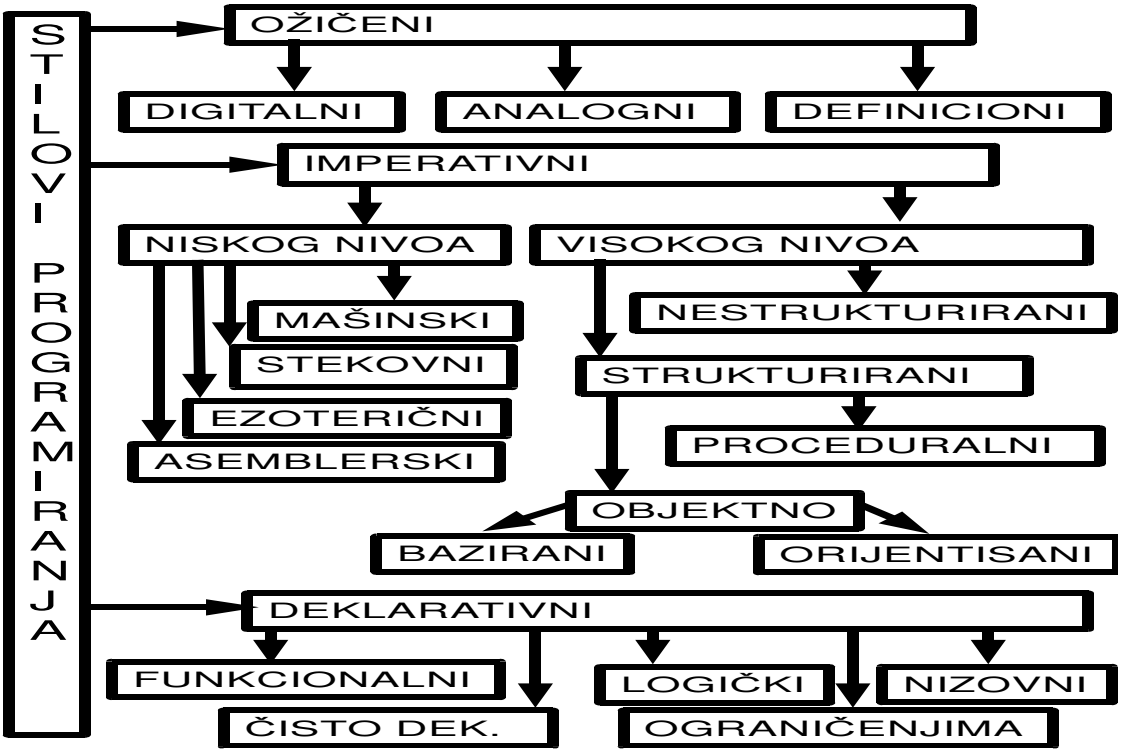
Najvažniji kriteriji za ocjenu programskih jezika uključuju čitljivost, upisivost, pouzdanost, cijenu.

Glavni uticaji na dizajn programskog jezika su bili mašinska arhitektura i metodologije softverskog dizajna.

Glavne metode implementacije programskih jezika su kompajliranje, čisto interpretiranje i hibridna implementacija.

2: Klasifikacija programskih jezika

Sada će se napraviti klasifikacija programskih jezika i reći nešto o glavnim predstavnicima pojedinih programskih jezika ali i načina programiranja koji nisu programski jezici. Na slici je dat shematski jedan od načina podjele programskih jezika i načina programiranja koji se generalno ne mogu smatrati programskim jezikom.



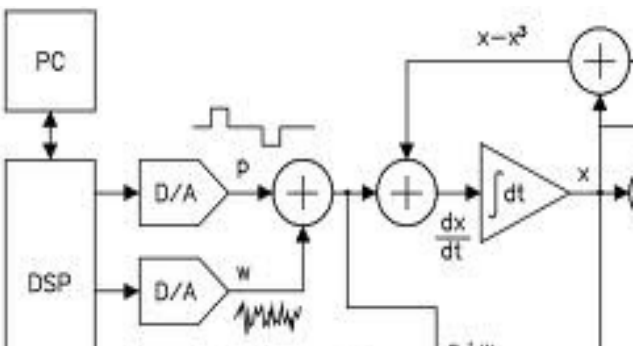
Ožičeno programiranje

Najstariji način programiranja računara je direktnim vezivanjem sklopova u cilju rješavanja konkretnog problema. Takvi računari obično nemaju operativni sistem, a programer često ima ulogu elektroničara. Ovo je najteži i najnefleksibilniji način programiranja, ali u može postići najbolje performanse.

Ožičeno programiranje analognih računara

Analogni računari predstavljaju realni sistem koji treba analizirati ekvivalentnom kontinualnom veličinom.

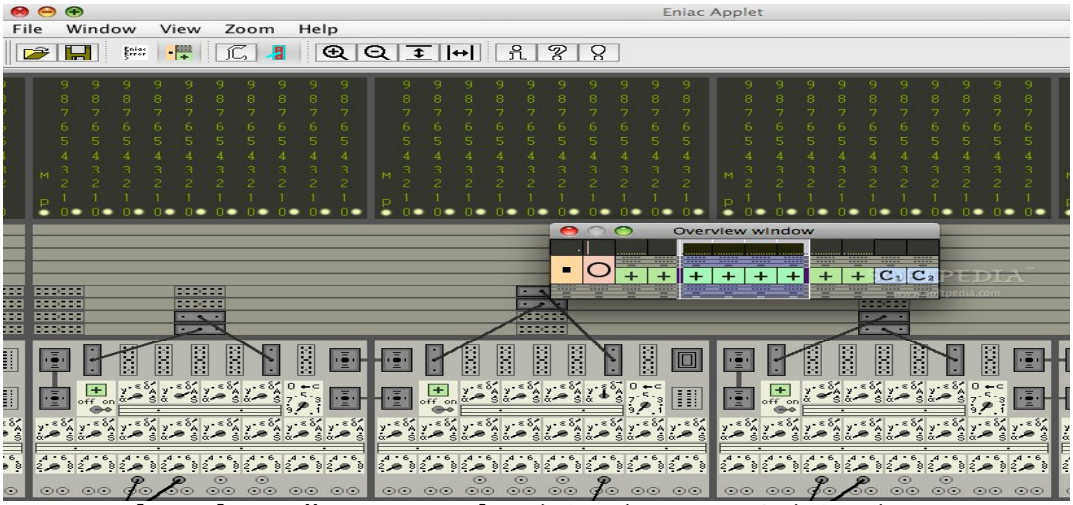
Na primjer, brzina je promjena koordinate u vremenu, odnosno prvi izvod funkcije koordinate.



S druge strane, lako je konstruisati sklop koji na jednim polovima daje napon, a na drugim izvod tog napona pomnožen konstantom. Očitavajući ulazne i izlazne vrijednosti, analogni računari mogu da simuliraju željeni sistem, te su rano korišteni u balistici, aerodimanici i drugim tehničkim disciplinama. Tako izlazni napon od 1,3 V može da predstavlja poziciju od 1300 metara.

Programabilni analogni računar (primjer Heatkit EC-1, 1960) se sastojao od određenog broja pripremljenih sklopova za elementarne operacije sabiranja, množenja konstantom, integriranja i slično, koji su se direktno povezivali žicama. Padom cijena fleksibilnijih digitalnih računara, programabilni analogni računari su se prestali koristiti za opštu namjenu, ali su se zadržali kao zalemljeni sklopovi za rješavanje konkretnih slučajeva, na primjer u samonavodećim raketama.

Ožičeno programiranje digitalnih računara



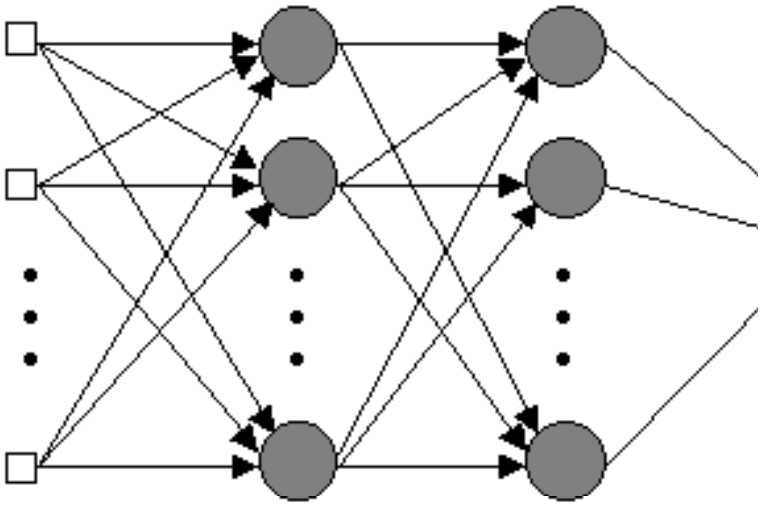
Najraniji digitalni računari Mark 1 (1944) i ENIAC (1946) programirani su direktnim prevezivanjem kablova, kao i u slučaju analognih programabilnih računara, spajajući žice između akumulatora koji su čuvali vrijednosti, tablice funkcija i ulazno-izlaznih sklopova.

Ovaj stil programiranja za aplikativne programe je vrlo brzo napušten, nakon pojave računara s pohranjenim programom.

Ali se direktno programiranje hardverom zadržalo prilikom konstrukcije računara. Na primjer, generisanje slike na ekranu prepušteno je posebnom sklopu, koji vrlo često nije realizovan programabilnim mikroprocesorom, nego jednostavnijim sklopom, ali drastično smanjuje opterećenje glavnog procesora.

Ožičeno programiranje neuralnih mreža

Neuralne mreže su sistemi bazirani na ponašanju sličnom nervnom sistemu živih organizama. Na bazi kombinacije ulaza i očekivanih izlaza, oni uče snalaženje u različitim situacijama. Iako se često simuliraju programima na digitalnim računarima, također se realizuju i analognim ili digitalnim sklopovima.



Jezici za dizajn sklopova

Ožičeno programiranje doživljava svoju modernu inkarnaciju u obliku jezika za opis sklopova. Pojava programabilnih čipova, kao što je FPGA omogućava pravljenje sklopova sa ponašanjem po želji, čime je moguće postići znatno veću brzinu procesiranja u odnosu na ostale načine programiranja.

Postoji više programskih jezika u kojima programiranje podsjeća na današnje programske jezike visokog nivoa. Verilog (1984, T:100+) se široko koristi za opisivanje ponašanje sklopova i njihovu sintezu. LabVIEW (1986, T:51+) definiše ponašanje sklopa koristeći grafički interfejs. LabWindows/CVI (1987, T:51+) kao LabView, ali s C bibliotekama. VHDL (1987, T:51+) opisuje tok podataka kroz sklopove predviđene za sintezu ili simulaciju. Ladder Logic (1993, T:42) omogućava da se grafički definiše aktivnost programabilnog logičkog kontrolera

Tako na primjer realizacija osmobitnog brojača u jeziku Verilog izgleda kao u sljedećem primjeru:

```
module up_counter (out , enable , clk , reset);
output [7:0] out; input enable, clk, reset;
reg [7:0] out;
always @(posedge clk)
if (reset) begin out <= 8'b0 ; end
else if (enable) begin out <= out + 1; end
endmodule
```

Imperativni jezici Niskog nivoa

Pojava računara s eksternim programom, od Zuse Z3 (1941) ukinula je potrebu za hardverskim prevezivanjem u rješavanju programa opšte namjene i zamijenili su se mašinskim jezikom.

Taj princip rada računara zadržao se i danas, iako su vremenom razvijeni programski jezici višeg nivoa koji sakrivaju mašinski jezik od aplikativnog programera i omogućavaju mu lakši razvoj softvera.

Mašinski jezici

Mašinski jezik je serija binarnih brojeva, koji se praktičnije predstavljaju dekadnim, heksadekadnim ili oktalnim brojevima. Svaki model procesora ima svoj mašinski

jezik, a često i programi na računarima s istim procesorom ali različitim ulazno/izlaznim uređajima moraju biti prepravljeni. Tipični mašinski program se sastoji od niza brojeva koji predstavljaju instrukcije koje prenose podatke između procesorskih registara i lokacija u memoriji predstavljenih brojevima. Na ranim računarima su unošeni prekidačima ili bušenom trakom, a u novije vrijeme u okviru višeg programskog jezika.

Postoji veliki broj procesora koji se razlikuju s skupom instrukcija, ovdje će biti spomenut mali broj njih. VAX (1977) arhitektura se odlikuje bogatim skupom instrukcija, ali i komplikovanim hardverom za njeno dekodiranje. ARM (1985) predstavlja arhitekturu redukovanog skupa instrukcija jednake širine riječi, 16 registara, koja je zbog male potrošnje postala raširena u prenosnim uređajima. Intel IA32 (1985) instrukcijski skup predstavlja osnovni radni jezik većine stonih računara. Ovdje je dat primjer programa u mašinskom jeziku za TI-89 programabilni kalkulator, za mjerenje brzine procesora, što se koristi ukoliko želimo programirati maksimalno brze programe na samom kalkulatoru.

```
Exec("203c034fb5e2538066fc4e750000")
```

Asemblerski jezici

Asemblerski jezici (1949, T:21) predstavljaju također široku klasu programskih jezika, čak i širu od klase mašinskih jezika, jer za svaki mašinski jezik postoji jedan ili asemblerskih jezika. Mašinske instrukcije imaju svoju reprezentaciju u čitljivijoj formi preko mnemoničkih skraćenica, a program se najprije prevodi iz asemblerskog u mašinski jezik.

Dati primjer za TI-89 u asemblerskoj reprezentaciji za procesor MC68000 izgleda ovako:

```
TIMER MOVE.L #55555554,D0 ; 203C034FB5E2
LOOP SUBQ.L #1,D0 ; 5380
BNE LOOP ; 66FC
RTS ; 4E75
```

Programe je lakše modifikovati nego u mašinskom jeziku, a naročito žičanom programiranju. Ali u asemblerskom jeziku i relativno jednostavne operacije zahtijevaju mnogo instrukcija, pa se programiranje u asemblerskim jezicima koristi uglavnom u situacijama kada je brzina izvršavanja jako bitna, a kompajleri viših programskih jezika ne pružaju dovoljan nivo optimizacije

Stekovni jezici

Jedan od pristupa u programiranju zasniva se na intenzivnom korištenju strukture steka, koja ima osobinu da se zadnje postavljenom podatku prvo pristupa. Ovakva struktura je prisutna u mnogim mikroprocesorima, a zahtijeva transformaciju aritmetičkih izraza u oblik koji je pogodniji za računarsko parsiranje, nego za čitljivost. Postoji određen broj programskih jezika zasnovanih na ovakvom stilu programiranja.

Forth (1972, T:38) sve podake drži na steku tako da njegove instrukcije nemaju parametara. Zbog kombinovane interaktivnosti, malih hardverskih zahtijeva i brzine rada našao je primjenu u ugrađenim uređajima, pokretačima sistema i robotici. Postscript (1982, T:100+) je često korišten kao kontrolni jezik plotera, laserskih štampača i za formatiranje dokumenata. Factor (2003, T:51+) je dizajniran kao skriptni jezik u igrama, a izvršava se nad Java virtuelnom mašinom. Kao ilustracija

stekovnog stila programiranja slijedi program u jeziku Postscript koji iscrtava veliki znak ">".

```
newpath
100 200 moveto
200 250 lineto
100 300 lineto
2 setlinewidth
stroke
```

Ezoterični jezici

Postoje i jezici u kojima je programiranje jako teško, razvijeni kao dokaz koncepta ili humorističkih razloga. Oni se zovu ezoterični jezici. Jedna grupa ovih jezika ima minimalistički skup instrukcija, često i manji od mašinskih jezika. Turingova mašina (1948) predstavlja apstraktni model uređaja koji manipuliše s trakom podataka koristeći određena pravila tranzicije. Corewar Redcode (1984) je apstraktni mašinski jezik namijenjen za igru borbe između programa s ciljem međusobnog ometanja. Befunge (1993) predstavlja stek orijentisani dvodimenzionalni programski jezik u kome su sve naredbe od jednog znaka a program teče u sva četiri pravca. Brainf*ck (1993) ima samo osam primitivnih naredbi sličnih Turingovoj mašini. Malbolge (1998, T:100+) je namjerno dizajniran da programiranje bude nemoguće teško. Whitespace (2003) koristi samo nevidljive znakove, kao što su razmaci, tabulatori i novi redovi.

Pored minimalističkih ezoteričnih jezika, neki od jezika iz ove skupine imaju namjerno izvrnutu logiku u odnosu na uobičajene više programske jezike. Intercal (1972) je prvi jezik parodije nad uobičajenim programskim jezicima (kao što su naredbe COME FROM ili varijable predstavljene brojevima).

```
0&>:1-:v v *_$.@
^ _$>\:^
```

Faktorijel u jeziku Befunge

Imperativni jezici visokog nivoa

Imperativni jezici visokog nivoa predstavljaju najčešće korištenu grupaciju programskih jezika. Ovi jezici detaljno opisuju kako riješiti određeni problem. Oni se odlikuju upotrebom varijabli za čuvanje stanja, načinom pisanja aritmetičkih izraza koji podsjeća na matematičku notaciju, upotrebom potprograma, naredbama uglavnom na engleskom jeziku i priličnim stepenom neovisnosti od mašine. Najčešće se implementiraju uz pomoć programa kompajlera, koji prevode program u jeziku višeg nivoa u jezik nižeg nivoa ili interpretera koji analizira program i istovremeno ga izvršava.

Jezici nestrukturiranog programiranja

Uvođenje programskih jezika visokog nivoa smanjuje ovisnost od mašine, zahvaljujući naredbama za uslove i petlje, potprogramima i varijablama. Rani programski jezici su uveli stil programiranja koji se zove "špagetno" ili nestruktuirano (mada su u kasnijim verzijama usvojili modernije stilove programiranja), koristeći naredbe bezuslovnih skokova, koje se lako prevode u slične naredbe u mašinskom jeziku. FORTRAN (1957, T:27) predstavlja prvi viši programski jezik opšte namjene i

dalje ima široku primjenu u rješavanju numeričkih problema. Sljedeći primjer u jeziku Fortran II.

```
C POVRSINA TROUGLA
C INPUT - TAPE READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
      READ INPUT TAPE 5, 501, IA, IB, IC
  501 FORMAT (3I5)
      IF (IA) 777, 777, 701
  701 IF (IB) 777, 777, 702
  702 IF (IC) 777, 777, 703
  703 IF (IA+IB-IC) 777, 777, 704
  704 IF (IA+IC-IB) 777, 777, 705
  705 IF (IB+IC-IA) 777, 777, 799
  777 STOP 1
C HERONOVA FORMULA POVRSINE TROUGLA
  799 S = FLOATF (IA + IB + IC) / 2.0
      AREA=SQRTF(S*(S-FLOATF(IA)) * (S - FLOATF(IB)) *
     +     (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
  601 FORMAT (4H A=,I5,5H B=,I5,5H C=,I5,8H AREA=
     + ,F10.2, 13H SQUARE UNITS)
      STOP
      END
```

BASIC (1964, T:7) je jezik bez posebnih deklaracija promjenjivih, korišten kao ugrađeni programski jezik u mikroračunarima..

Ovdje je dat primjer u jeziku BASIC koji računa površinu trougla u kome se uočava upotreba bezuslovnih skokova.

```
10 PRINT "UNESI STRANE" : INPUT A,B,C
20 IF A<=0 OR B<=0 OR C<=0 THEN GOTO 70
30 S = (A + B + C) / 2.0
40 AR = SQR( S * (S - A) * (S - B) * (S - C))
50 PRINT AR
60 GO TO 80
70 PRINT "GRESKA"
80 END
```

COBOL (1959, T:31) se odlikuje deskriptivnom sintaksom i naredbama za rad s datotekama. Dugo se koristio u poslovnom okruženju pa i danas su mnogi programi aktivni. Sljedeći primjer u jeziku COBOL ilustruje unos podataka u sekvencijalnu datoteku.

```
$ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  SeqWrite.
AUTHOR.  Michael Coughlan.
* Primjer rada s sekvencijalnom datotekom
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT StudentFile ASSIGN TO "STUDENTS.DAT"
                ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudentId       PIC 9(7).
   02  StudentName.
       03 Surname      PIC X(8).
       03 Initials     PIC XX.
   02  DateOfBirth.
       03 YOBirth      PIC 9(4).
       03 MOBirth      PIC 9(2).
       03 DOBirth      PIC 9(2).
   02  CourseCode      PIC X(4).
   02  Gender          PIC X.
PROCEDURE DIVISION.
Begin.
    OPEN OUTPUT StudentFile
    DISPLAY "Unesi detalje studenta."
    PERFORM GetStudentDetails
    PERFORM UNTIL StudentDetails = SPACES
      WRITE StudentDetails
      PERFORM GetStudentDetails
    END-PERFORM
    CLOSE StudentFile
    STOP RUN.
GetStudentDetails.
    DISPLAY "Id,Prezime,Inicijali,G,M,D,Kurs,Pol "
    DISPLAY "NNNNNNNSSSSSSSSIIYYYYMMDDCCCCG"
    ACCEPT  StudentDetails.
```

Focal (1969) je sličan BASIC-u namijenjen kao ugrađeni jezik u PDP sistemima. DOS BAT (1981, T:100+) namijenjen za automatizaciju naredbi (skriptiranje) u operativnim sistemima DOS i Windows, upotrebljiv uprkos ograničenim mogućnostima

Proceduralni programski jezici

Rast veličine programa doveo je do potrebe da se programi dijele u sintaksne cjeline kako bi se lakše održavali. To je unaprijedilo preglednost programa. Koriste se uslovi i petlje koji obuhvataju veći broj linija i instrukcija koje mogu obuhvatiti nove uslove i petlje, potprogrami s parametrima, lokalne i globalne promjenjive, prosti i složeniji tipovi podataka i skupovi potprograma – moduli.

Proceduralni programski jezici (ALGOL-oidni)

Jedna grupacija jezika iz ove kategorije ima sintaksu koja se odlikuje mogućnošću da se instrukcija piše u više redova koda, uz obavezan separator između naredbi (slobodni format), te specijalnim ključnim riječima koje označavaju početak i kraj pojedine strukture.

RPG (1959, T:30) se od generatora izvještaja razvio u razvojno okruženje za mainframe sisteme.

ALGOL 60 (1960, T:51+) uveo je složene naredbe, slobodni format i ugniježdene potprograme, pa se koristio za predstavljanje algoritama. Sljedeći primjer u ALGOL 60 rješava kule Hanoja.

```
'begin'
        'comment'
                The Towers Of Hanoi
                Algol-60      ;
        'procedure' movedisk(n, f, t);
        'integer' n;
        'integer' f;
        'integer' t;
        'begin'
                outstring (1, "move ");
                outinteger(1, f);
                outstring (1, " --> ");
                outinteger(1, t);
                outstring (1, "\n");
        'end';
        'procedure' dohanoi(n, f, t, u);
        'integer' n;
        'integer' f;
        'integer' t;
        'integer' u;
        'begin'
                'if' n < 2 'then'
                        movedisk(1, f, t)
                'else'
                        'begin'
                                dohanoi(n - 1, f, u, t);
                                movedisk(1, f, t);
                                dohanoi(n - 1, u, t, f);
                        'end';
        'end';
        dohanoi(3, 1, 3,  2);
'end'
```

PL/I (1964, T:40) dizajniran da uključi osobine Fortrana, COBOL-a i ALGOL-a u jedinstveni jezik, a glavnu primjenu je našao u IBM mainframe računarima. Sljedeći PL/I program čita sa ulaza niz znakova, a zatim prikazuje svaku liniju koja sadrži taj niz znakova-

```
/* Read a string,
and then print every */
find_strings: procedure options (main);
   declare pattern character (100) varying;
   declare line    character (100) varying;
   declare line_no fixed binary;
   on endfile (sysin) stop;
   get edit (pattern) (L);
   line_no = 1;
   do forever;
      get edit (line) (L);
      if index(line, pattern) > 0 then
         put skip list (line_no, line);
      line_no = line_no + 1;
   end;
end find_strings;
```

ALGOL 68 (1968) imao je znatno drugačiju sintaksu od svog prethodnika Algol 60 koja zahtijeva i znakove izvan standardnog ASCII skupa.

Pascal (1970, T:14) je široko korišteni jezik u obrazovanju kao uvod u struktuirano programiranje, a zbog brze kompilacije i izvršenja koda u njemu su razvijane i aplikacije iz stvarnog svijeta. Sljedeći primjer u jeziku Pascal prikazuje kvadrat unesenog broja.

```
program FPProgT15;
var
  x:integer;
procedure XSquare;
begin
  x := 5;
  writeln('The Square Of ', x, ' Is ', x * x);
end;
procedure quit;
begin
  writeln;
  writeln;
  writeln('Press <Enter> To Quit');
  readln;
end;
begin
  XSquare;
  quit;
end.
```

PL/M (1972) je zbog direktne podrške memorijskim lokacijama i interaptima korišten za implementaciju prvog operativnog sistema za mikroračunare, CP/M. SAS (1976, T:32) je specijalizovani sistem za statističku analizu. Icon (1977, T:51+) ima sintaksu sličnu ALGOL-u a namijenjen je analizi teksta. Modula 2 (1978, T:51+) je jezik sličan Pascal-u, sa mogućnostima razdvajanja programa u posebne kompilacijske jedinice. ADA (1980, T:17) zbog stroge provjere tipova predstavlja jedan od preporučenih jezika za aplikacije u kojima je pouzdanost vrlo bitna. Sljedeći primjer u Ada ilustruje upotrebu funkcija kojoj je proslijeđen niz promjenjive veličine.

```
--
-- Function using variable-sized array parameter.
--
with Gnat.Io; use Gnat.Io;
procedure f2 is
   -- An integer array.
   type IntArr is array (Integer range <>) of Integer;
   -- Compute the max of the array.
   function Max (Arr: IntArr) return Integer is
      TheMax: Integer := Arr(Arr'First);
      I: Integer;
   begin
      for i in Arr'First + 1 .. Arr'Last loop
         if Arr(I) > TheMax then
            TheMax := Arr(I);
         end if;
      end loop;
      return TheMax;
   end;
   -- Some random arrays.
   A: IntArr(1..7) := (4, 5, 12, 4, 71, -9, 21);
   B: IntArr(-3..3) := (-4,-21,-3,-18, -7, -29, -10);
begin
   Put(Max(A));
   Put(" ");
   Put(Max(B));
   Put(" ");
   B(-2) := 10;
   Put(Max(B));
   Put(" ");
   Put(Max((2, 4, 21)));
   New_Line;
end f2;
```

PL/SQL (1992, T:16) je jezik za pristup Oracle bazama podataka koji kombinuje SQL sa proceduralnim jezikom sintakse slične jeziku Ada. Primjer u PL/SQL ubacuje 10 redova u tabelu.

```
-- available online in file 'sample1'
DECLARE
  x NUMBER := 100;
BEGIN
  FOR i IN 1..10 LOOP
    IF MOD(i,2) = 0 THEN      -- i is even
      INSERT INTO temp VALUES (i, x, 'i is even');
    ELSE
      INSERT INTO temp VALUES (i, x, 'i is odd');
    END IF;
    x := x + 100;
  END LOOP;
  COMMIT;
END;
```

Proceduralni programski jezici (C-oidni)

Slobodni format je zadržan i u narednoj grupaciji ovih jezika, ali se umjesto ključnih riječi za označavanje blokova koriste posebni znakovi, najčešće vitičaste zagrade. BCPL (1966, T:100+) predstavljao je jezik za pisanje kompajlera za druge programske jezike, zabilježen kao prvi jezik s vitičastim zagradama za blokove, a poznaje samo cjelobrojne podatke. MUMPS (1966, T:51+) korišten prvenstveno u medicinskoj informatici i podržava kompaktnu sintaksu prilagođenu slabim računarskim resursima tog vremena. C (1972, T:2) je jedan od najpopularnijih programskih jezika, prvenstveno namijenjen za sistemsko programiranje. Odlikuje se kompajlerima koji generišu brz kod, bogatim izborom operatora, vitičastim zagradama za blokove i slobodom u tipovima. Sljedeći primjer prikazuje proste brojeve između dva cijela broja.

```
#include <stdio.h>
int checkPrimeNumber(int n);
int main()
{
    int n1, n2, i, flag;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("Prime numberbers between %d and %d are: ", n1, n2);
    for(i=n1+1; i<n2; ++i)
    {
        // i is a prime number, flag will be equal to 1
        flag = checkPrimeNumber(i);
        if(flag == 1)
            printf("%d ",i);
    }
    return 0;
}
// user-defined function to check prime number
int checkPrimeNumber(int n)
{
    int j, flag = 1;
    for(j=2; j <= n/2; ++j)
    {
        if (n%j == 0)
        {
            flag =0;
            break;
        }
    }
    return flag;
}
```

bc (1975, T:51+) je kalkulatorski jezik proizvoljne preciznosti. Awk (1977, T:51+) se koristi za obradu stringova i tekstualnih datoteka, često unutar shell skripti sa jednom linijom, ali i dužih programa. OpenEdge ABL (1984, T:46) je razvojni jezik za poslovne aplikacije, sa naredbama za korisnički interfejs i upite nad podacima. Perl (1987, T:9) ima velike mogućnosti analize tekstova i najpopularniji je jezik za implementaciju dinamičkih web aplikacija koristeći CGI protokol. Sljedeći primjer koji u jeziku Perl puni dva elementa asocijativnog niza i ispisuje ih ilustruje petlje, potprograme i pisanje u slobodnom formatu.

```
Par("200", "Sarajevo");
Par("100", "Banjaluka");
while (($key,$value)=each %array) {
  print("$key,value\n");
};
sub Par {
  my($key, $value) = @_;
  $array{$key} = $value;
}
```

PHP (1995, T:6) se najčešće koristi u serverskim Web aplikacijama kao jezik za generisanje web stranica.

```
<?
$midnight_today = mktime(0,0,0);
print '<select name="date">';
for ($i = 0; $i < 7; $i++) {
    $timestamp = strtotime("+$i day", $midnight_today);
    $display_date = strftime('%A, %B %d, %Y', $timestamp);
    print '<option value="' . $timestamp .'">'.$display_date."</option>\n";
}
print "\n</select>";
?>
```

cg (2003, T:51+) varijanta jezika C za programiranje grafičkih kartica. OpenCL (2008, T:51+) je varijanta jezika C za programiranje grafičkih kartica za procesiranje opšte namjene. Go (2009, T:26) je jezik sa mogućnostima konkurentnog programiranja koji generiše efikasan kod koristeći dinamičke tipove podataka.

Proceduralni programski jezici (FORTRAN-oidni)

Mnogi proceduralni jezici ne zahtijevaju separatore naredbi, ali zato kraj linije obično predstavlja kraj naredbe. COMAL (1973, T:100+) je predstavljao mješavinu BASIC-a i Pascala, korišten kao uvod u struktuirano programiranje za korisnike navikle na BASIC. BourneShell (1977, T:51+) je skriptni jezik za Unix sisteme. Sintaksa mu podsjeća na ALGOL 68. Veliki broj Unix aplikacija je razvijen u ovom jeziku. CSH (1978, T:22) je jezik za automatizaciju komandi u Unix sistemima sa bržim mogućnostima evaluacije izraza u odnosu na Bourne shell. NATURAL (1979, T:51+) je korišten kao razvojni jezik deskriptivne sintakse u ADABAS sistemima za baze podataka. REXX (1979) je korišten kao skriptni jezik za obradu teksta i generisanje izvještaja. X-BASE (1979, T:51+) predstavlja porodicu jezika za rad s datotekama obično u DBF formatu, kao što su DBASE, Clipper i Visual FoxPro. ABC (1981, T:51+) je interaktivni jezik s sličnom namjenom kao BASIC. Occam (1983, T:51+) je prvenstveno korišten kao razvojni jezik na INMOS transpjuterskim sistemima a odlikuje se mogućnostima za paralelno programiranje. Quick BASIC (1985) je jedno od proširenja klasičnog BASIC-a (uz BBC BASIC, Super BASIC i TrueBASIC), prema strukturiranom programiranju. Sljedeći primjer pokazuje višelinijsku IF naredbu u Quick BASIC-u.

```
CLS
INPUT "Unesi broj: ", Number
IF Number < 100 THEN
   PRINT "Vas broj je manji od 100"
ELSE
   PRINT "Vas broj je veci ili jednak 100"
END IF
```

Korn shell (1986, T:51+) je UNIX komandna školjka proširena sa asocijativnim nizovima i aritmetikom u pokretnom zarezu. Clarion (1988, T:100+) u kombinaciji s svojim generatorom koda, predstavlja moćan jezik za aplikacije u oblasti baza podataka. BASH (1989, T:51+) je unapređenje klasičnog Unix shell-a posebno korišten na sistemima otvorenog koda, kao što su Linux i OpenBSD. Fortran 90 (1990) predstavlja proširenje jezika Fortran s modernim programskim strukturama. Z shell (1990, T:51+) je proširenje Bourne Unix komandnog interpretera sa dovršavanjem koda i podrškom varijablama. ct (2008, T:51+) specijalizovani jezik za programiranje višejezgrenih procesora.

Objektno orijentisani programski jezici

Proceduralno programiranje postaje nedovoljno u sistemima u kojima postoji veća koordinacija između potprograma, na primjer kada oni koriste zajedničke globalne varijable. Objektno orijentisano programiranje spaja podatke koji se dijele između potprograma i same potprograme koji ih koriste u sintaksne jedinice koje se zovu klase, a realizacija svake od klasa predstavlja objekat. Ključna osobina ovog stila programiranja je nasljeđivanje klasa koja omogućava da se iskoristi ranije napisani kod bez većih popravki.

Objektno orijentisani programski jezici - poruke

Prva grupacija ovakvih programskih jezika ima sintaksu poruka. Smalltalk (1972, T:48) ima malo ključnih riječi i sve podatke vidi kao objekte koji međusobno komuniciraju porukama. Programi se izvršavaju na posebnoj virtuelnoj mašini i sintaksa im se može mijenjati u toku rada. Sljedeći primjer ispisuje pjesmu 99 bottles of beer.

```
99 to: 1 by: -1 do: [ :i |
        i print. ' bottles of beer on the wall, ' print.
        i print. ' bottles of beer. ' print.
        'take one down, pass it around, ' print.
        (i-1) print. ' bottles of beer on the wall, ' print.
]
```

Beta (1975, T:51+) je čisti objektno orijentisani jezik s konceptom uzoraka koji unificira klase i procedure. Io (2002 , T:51+) eliminiše razliku između instance i klase predstavljajući sve kao objekt, a programe vidi kao stablo podataka.

Objektno orijentisani programski jezici – C-oidni

Najširu primjenu imaju objektno orijentisani jezici, sa sintaksom koja posjeća na programski jezik C. C++ (1983, T:4) je objektno orijentisano proširenje jezika C i koristi se u razvoju svih vrsta sistemskog i aplikativnog softvera. Sljedeći primjer deklariše klasu s jednim konstruktorom i dvije metode.

```
// DateClass.cc
// Demonstrate the definition of a simple class
// and member functions
#include <iostream>
using namespace std;
// Declaration of Date class
class Date {
public:
  Date(int, int, int);
  void set(int, int, int);
  void print();
private:
  int year;
  int month;
  int day;
};
int main()
{
   Date today(1,9,1999);
   cout << "This program was written on ";
   today.print();
   cout << "This program was modified on ";
   today.set(5,10,1999);
   today.print();
   return 0;
}
// Date constructor function definition
Date::Date(int d, int m, int y)
{
  if(d>0 && d<31) day = d;
  if(m>0 && m<13) month = m;
  if(y>0) year =y;
}
// Date member function definitions
void Date::set(int d, int m, int y)
{
  if(d>0 && d<31) day = d;
  if(m>0 && m<13) month = m;
  if(y>0) year =y;
}
void Date::print()
{
  cout << day << "-" << month << "-" << year << endl;
}
```

Objective C (1983, T:5) je također objektno orijentisano proširenje jezika C, ali koristi sistem prosljeđivanja poruka objektima. TCL (1988, T:49) se koristi kao skriptni jezik za brze prototipe, GUI i kao jezik ugrađen u aplikacijama.

Java (1995, T:1) je trenutno vodeći programski jezik, najčešće korišten u web aplikacijama, a izvršava se u virtualnom okruženju. Sljedeći primjer prikazuje klasu koja simulira stek mašinu.

```
import java.io.IOException;
public class AdditionStack {
   static int num;
   static int ans;
   static Stack theStack;
   public static void main(String[] args)
   throws IOException {
      num = 50;
      stackAddition();
      System.out.println("Sum=" + ans);
   }
   public static void stackAddition() {
      theStack = new Stack(10000);
      ans = 0;
      while (num > 0) {
         theStack.push(num);
         --num;
      }
      while (!theStack.isEmpty()) {
         int newN = theStack.pop();
         ans += newN;
      }
   }
}
class Stack {
   private int maxSize;
   private int[] data;
   private int top;
   public Stack(int s) {
      maxSize = s;
      data = new int[maxSize];
      top = -1;
   }
   public void push(int p) {
      data[++top] = p;
   }
   public int pop() {
      return data[top--];
   }
   public int peek() {
      return data[top];
   }
   public boolean isEmpty() {
      return (top == -1);
   }
}
```

J++ (1997) varijanta jezika Java firme Microsoft čije su određene razlike u odnosu na standardni jezik izazvale više kontroverzi. ActionScript (1998, T:35) se koristi kao jezik u klijentskim web aplikacijama u sastavu Flash tehnologije.

C # (2001, T:3) sintaksno je srodan jeziku Java, ali je vezan za .NET infrastrukturu. Primjer u jeziku C# koji na klik mišem na dugme na prozoru mijenja sadržaj unosnog polja izgleda ovako:

```
using System;
using System.Text;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void button1_Click(object sender,
     EventArgs e)
    {
      textBox1.Text = "Mijenjam tekst!";
    }
  }
}
```

D (2001, T:37) je još jedno objektno proširenje jezika C, sa automatskim upravljanjem memorijom ali čiji kompajleri generišu prirodni kod. TOM (2001, T:51+) je dizajniran da omogući neplaniranu ponovnu upotrebljivost koda kroz mehanizam koji omogućavanje dodavanja ne samo metoda nego i klasa i nadklasa. J # (2002, T:100+) zamišljen kao prelazni jezik sa Java na C#. Groovy (2003, T:36) je dosta sličan jeziku Java i koristi se kao skriptni jezik za Java platformu. X10 (2004, T:51+) predviđen je za paralelno programiranje koristeći model koji razdvaja račun u skup mjesta i aktivnosti. C++/CLI (2005, T:51+) ranije poznat kao Managed C++, predstavlja varijantu C++ jezika za .NET/Mono okruženje. Dart (2011, T:51+) je zamišljen kao buduća zamjena za JavaScript.

Objektno orijentisani programski jezici – ALGOL-oidni

Neki proceduralni jezici s sintaksom koja ima ključne riječi za oznaku blokova i separatore naredbi, su također prošireni u objektno orijentisanom pravcu. ABAP (1983, T:28) po sintaksi podsjeća na COBOL i koristi se za u kao razvojni alat za SAP sisteme.

ObjectPascal (1986, T:11) je proširenje Pascala za objektno programiranje, u verziji pod imenom Delphi dosta korišten u aplikacijama korisničkog interfejsa za Windows. Sljedeći primjer ilustruje reakciju na klik na dugme na ekranskom formularu.

```
interface
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello World');
end;
end.
```

Oberon-2 (1988, T:51+) je srodan jeziku MODULA-2, ali je manji uz uključenje objekata. Modula 3 (1988, T:100+) je proširenje jezika Modula 2 objektno orijentisanim mogućnostima.

Objektno orijentisani programski jezici - FORTRAN-oidni

I jezici bez obaveznih separatora između naredbi imaju svoje predstavnike u objektno orijentisanim programskim jezicima.

S (1975, T:51+) je jezik za statističke proračune. Eiffel (1986, T:51+) je jezik s automatskim upravljanjem memorijom i širokom lepezom sistema nasljeđivanja. Lingo (1988, T:51+) za Adobe Director namijenjen je za pravljenje animacija. S plus (1988, T:51+) jedna od komercijalnih implementacija jezika S.

Sather (1990) je razvijen iz Eifell-a ali sada podržava i elemente funkcionalnog programiranja. R (1993, T:19) je trenutno najpopularnija implementacija jezika S. Revolution (1993, T:51+) sada se zove LiveCode, namijenjen za brzo pisanje aplikacija.

Ruby (1995, T:12 dopušta programu da bude svjestan svoje strukture i na taj način se modifikuje.

```
# Box drawing class.
class Box
  # Initialize to given size, and filled with spaces.
  def initialize(w,h)
    @wid = w
    @hgt = h
    @fill = ' '
  end
  # Change the fill.
  def fill(f)
    @fill = f
    return self
  end
  # Rotate 90 degrees.
  def flip
    @wid, @hgt = @hgt, @wid
    return self
  end

  # Generate (print) the box
  def gen
    line('+', @wid - 2, '-')
    (@hgt - 2).times { line('|', @wid - 2, @fill) }
    line('+', @wid - 2, '-')
  end

  # For printing
  def to_s
    fill = @fill
    if fill == ' '
      fill = '(spaces)'
    end
    return "Box " + @wid.to_s + "x" + @hgt.to_s + ", filled: " + fill
  end
private
  # Print one line of the box.
  def line(ends, count, fill)
    print ends;
    count.times { print fill }
    print ends, "\n";
  end
end
# Create some boxes.
b1 = Box.new(10, 4)
b2 = Box.new(5,12).fill('$')
b3 = Box.new(3,3).fill('@')
print "b1 = ", b1, "\nb2 = ", b2, "\nb3 = ", b3, "\n\n"
# Print some boxes.
print "b1:\n";
b1.gen
print "\nb2:\n";
```

```
b2.gen
print "\nb3:\n";
b3.gen
print "\nb2 flipped and filled with #:\n";
b2.fill('#').flip.gen
print "\nb2 = ", b2, "\n"
```

Clarion for Windows (1996, T:100+) dodaje OO mogućnosti ovom jeziku. VB.NET (2001, T:24) je objektno orijentisani BASIC namijenjen za .NET okruženje.

Boo (2003, T:51+) ima sintaksu sličnu Pythonu, namijenjen za izvršavanje u .NET/Mono okruženju. BlitzMax (2004, T:51+) je dijalekt BASIC-a namijenjen razvoju igara i drugih grafičkih aplikacija. Scratch (2006, T:34) ima za ciljnu grupu početnike, programirajuće kretanje sprajtova.

Objektno bazirani programski jezici

Pošto programiranje u objektno orijentisanim programskim jezicima zahtijeva dosta pripremnog koda vezano za deklaraciju klasa, neki programski jezici to izbjegavaju implementirajući samo neke osobine objektnog programiranja, na primjer upotrebu gotovih objekata, ali ne i kreiranje vlastitih klasa koje se mogu nasljeđivati. Time je fleksibilnost programskog jezika smanjena, ali je sam programski jezik lakši za naučiti.

Simula (1967, T:100+) namijenjen za simulaciju sistema sa diskretnim događajima predstavlja prvi programski jezik u kome se javljaju objekti. Max/MSP (1988, T:51+) je vizuelni jezik za razvoj multimedijalnih aplikacija. Visual BASIC (1991, T:7) je postigao popularnost zahvaljujući integrisanom okruženju za dizajniranje korisničkog interfejsa. Lua (1993, T:20) je mali jezik za ugrađene sisteme ali koji podržava više programskih paradigmi.

JavaScript (1995, T:10) je najpopularniji jezik za izvršavanje programa unutar web browser-a u cilju realizacije dinamičkih web stranica. Sljedeći primjer ilustruje promjenu sadržaja HTML stranice koristeći JavaScript program.

```
<!DOCTYPE html>
<html>
<body>
<h1>My Web Page</h1>
<p id="demo">My first paragraph.</p>
<script>
document.getElementById("demo").innerHTML = "Hello";
</script>
</body>
</html>
```

VBScript (1996 , T:51+) se koristi kao skriptni jezik u Windows sistemima i unutar HTML stranica, te unutar ASP stranica na serverskoj strani JScript.NET (2003, T:51+) se kompajlira za .NET okruženje kao jedna od implementacija JavaScript. JavaFX Script (2005, T:51+) također omogućava bogate klijentske aplikacije uz kompajliranje na Java virtualnu mašinu. PowerShell (2006, T:51+) je komandna školjka za Windows operativni sistem, integrisana s .NET okruženjem.

Jezici deklarativnog programiranja

Za razliku od imperativnih jezika koji težište programiranja stavljaju na način rješavanja konkretnog problema, deklarativni programski jezici se koncentrišu na opis samog problema. Stoga se koriste u oblastima kada se očekuje inteligentno ponašanje softvera, prije nego efikasnost računanja. Takve oblasti su mašinsko učenje, dokazivanje teorema, sređivanje aritmetičkih izraza, formatiranje dokumenata itd.

Čisto deklarativni jezici

Za specijalne primjene razvijeni su jezici, koji se često i ne smatraju programskim jezicima jer ne omogućavaju rješavanje svakog izračunljivog problema, ali svoju oblast primjene pokrivaju bolje nego jezici opšte namjene. Ovi jezici su uglavnom podskup nešto šire klase, domenski specifičnih jezika.

Čisto deklarativni jezici SQL

Ovakvi jezici su dominantni u pristupu bazama podataka. SQL (1974) vrlo je popularan jezik za upite nad bazama podataka. Paradox (1985, T:100+) jedan je od ranih relacionih DBMS za mikroračunare. Transact-SQL (1986,T:15) je varijanta SQL s proceduralnim proširenjima korištena u Microsoft i Sybase serverima. Sljedeći primjer u SQL prikazuje sva polja u tabeli imena uz navedeni uslov.

```
SELECT * FROM IMENA WHERE STAROST >20
```

Čisto deklarativni jezici markup

Za formatiranje dokumenata razvijeno je više specijalizovanih jezika. TeX (1978, T:100+) predstavlja jezik za označavanje tekstova u stonom izdavaštvu, posebno kod naučnih tekstova. SGML (1986) predstavlja standard za markiranje dokumenata. HTML (1989) je osnovni jezik na Internetu za predstavljanje dokumenata. Primjer dokumenta u HTML jeziku izgleda ovako:

```
<HTML>
<BODY> Tekst s <B> podebljanjem </B>
 </BODY>
</HTML>
```

CFML (1995, T:51+) , skraćeno od ColdFusion Markup Language, jedan je od prvih specijalizovanih jezika za dinamičke web stranice. XML (1997) se koristi široko za predstavljanje podataka i njihov prijenos na Internetu. XSLT (1997, T:51+) je deklarativni jezik za transformaciju XML dokumenata.

Drugi deklarativni jezici

Uprkos vrlo uskoj primjeni izvjesnu popularnost postigli su i drugi jezici deklarativnog programiranja. Pilot (1962, T:100+) je minimalistički jezik s jednoslovnim naredbama za opis školskih lekcija i testova. Sed (1973, T:100+) se koristi za pretraživanje i izmjene tekstualnih datoteka. . Sed program koji zamjenjuje sve riječi “grad” rječju “selo” u datom tekstu izgleda ovako

```
's/grad/selo/g'
```

Yacc (1974, T:100+) je jezik za opis sintakse drugih programskih jezika i generisanje sintaksnih analizatora. RC (1985) opisuje dijelove izvršnih Windows datoteka koji se učitavaju po potrebi. NXT-G (2006, T:25) je grafički jezik za upravljanje Lego robotima

Jezici logičkog programiranja

Logičkim programiranjem, koje je posebno korišteno u vještačkoj inteligenciji, definiše se skup pravila, potrebnih za zaključivanje. SNOBOL (1962) je rani dinamički tipizirani jezik za analizu teksta. Prolog (1972, T:43) predstavlja programe u formi činjenica i pravila na osnovu kojih izvodi zaključke i upite. U sljedećem primjeru u Prolog-u vidi se način definisanja pravila za nekih tijela sunčevog sistema.

```
kruzi(zemlja, sunce).
kruzi(mars, sunce).
kruzi(mjesec,zemlja).
kruzi(fobos, mars).
planet(P) <= kruzi(P, sunce).
satelit(S) <= kruzi(S, P) and planet(P).
```

Microprolog (1983) za rane mikroračunare koristi dvije modifikovane sintakse, u formi listi i infiksnu. Mercury (1995, T:100+) koristi sintaksu djelimično kompatibilnu s Prolog-om sa striktnim tipovima podataka.

Jezici programiranja ograničenja

Programiranje ograničenjima je stil programiranja u kome se definiše problem uz kriterije koje varijable stanja trebaju zadovoljiti (npr. "x ≤ 5"). Bertrand (1986) je jezik za pravljenja sistema s ograničenjima. Siri (1991) se koristi za reaktivno programiranje, u kome se iz jednog stabilnog stanja prelazi u drugo. Alma-0 (1997) uz nadgradnju jezika Modula 2 ima ključnu riječ FORALL kojom se probavaju razna rješenja. Sljedeći primjer u jeziku Siri rješava sistem linearnih jednačina definišući ih u formi ograničenja.

```
simultaneousEquations x'number y'number z'number:
{ a, b, c: aNumber;
 2*a + 4*b - 3*c = x;
4*a - 5*b + 4*c = y;
-4*a + b - 8*c = z; };
computeEquations: {
result: simultaneousEquations 8 2 -6;
(result.a, result.b, result.c)};
```

Funkcionalni jezici i jezici funkcionalnog nivoa

U funkcionalnim programskim jezicima programi se pišu u fomi definisane funkcije u tipičnom matematičkom obliku. Funkcije se definišu preko drugih funkcija ili sebe same, zavisno od argumenata. Neki od ovih jezika računaju sve izraze koji se javljaju u funkciji (striktna evaluacija), a neki samo kada je izraz korišten (lijena evaluacija).

Funkcionalni jezici listni

Prva grupacija ovih jezika predstavlja programe i podatke u formi listi.

Lisp (1958, T:13) predstavlja sve programe i podatke u obliku ulančane liste predstavljene unutar zagrada i od samih početaka jedan je od vodećih jezika vještačke

inteligencije. LOGO (1967, T:18) je prvi jezik namijenjen učenju programiranja za djecu a takođe smješta dijelove programa u liste. Scheme (1975, T:29) je minimalistička verzija LISP-a dosta korištena za ugrađene sisteme i skriptiranje.

Sljedeći primjer u jeziku Scheme nalazi presjek između dvije liste.

```
(DEFINE (guess lis1 lis2)
(COND ((NULL? lis1) () )
 ((member (CAR lis1) lis2)
  (CONS (CAR lis1) (guess (CDR lis1) lis2)))
(ELSE (guess (CDR lis1) lis2))))
```

ComonLisp (1984, T:51+) uključuje širok izbor tipova podataka i u okviru dodatnih alata podršku za objektno orijentisano programiranje. Dylan (1992, T:51+) sa ciljem da bude dinamički jezik pogodan za razvoj komercijalnog softvera, je u prvim verzijama koristio sintaksu iz Lisp-a, a kasnije je prešao na sintaksu sličnu ALGOL-u.

Racket (1994) ima specijalan makro sistem koji omogućava razvoj posebnih dijalekata jezika. REBOL (1997) je namijenjen za mrežne komunikacije i distribuirano računarstvo. Clojure (2007, T:100+) se prevodi ja Java virtuelnu mašinu i stimuliše višenitno programiranje.

Funkcionalni jezici ML

Čitljiviju sintaksu imaju jezici iz druge grupacije. ISWIM (1966) nije implementiran, ali je uvođenjem ključne riječi WHERE uticao na druge funkcionalne jezike. SASL (1972) je rani funkcionalni jezik bez tipova. ML (1973, T:44) se smatra rodonačelnikom čitave serije funkcionalnih jezika. On uključuje bočne efekte pa stoga uvijek evaluira podizraze.

```
(* ML version of 99 bottles of beer      *)
(*- Using functions parameters and Recursion *)
fun NinetyNineBottlesOfBeer(0) =
  print("\n No more bottles of beer on the wall\n")
| NinetyNineBottlesOfBeer(1) =
   (print("\n 1 bottle of beer on the wall, 1 bottle of beer.");
    print("\n Take one down and pass it around.");
    NinetyNineBottlesOfBeer(0))
| NinetyNineBottlesOfBeer(NumberOfBottles:int) =
   (print("\n "); print(NumberOfBottles);
    print(" bottle of beer on the wall, ");
    print(NumberOfBottles);
    print(" bottle of beer.");
    print("\n Take one down and pass it around.");
    NinetyNineBottlesOfBeer(NumberOfBottles - 1));
```

HOPE (1979) uvodi evaluaciju prema uzorku, pri čemu redoslijed uzoraka nije bitan.

KRC (1981) , skraćeno od Kent Recursive Calculator, korišten kao školski jezik funkcionalnog programiranja. CAML (1985, T:100+) je nasljednik jezika ML sa statičkim tipovima, automatskim upravljanjem memorijom i sistemom modula Miranda (1985) je čisto funkcionalni jezik, lijene evaluacije, bez sporednih efekata koji predstavlja programe u formi jednačina proizvoljnog redoslijeda.

Erlang (1986, T:39) je funkcionalni jezik striktne evaluacije, sa širokim mogućnostima višenitnog programiranja i komunikacije putem poruka.

Clean (1987, T:51+) spada u čisto funkcionalne jezike lijene evaluacije. Haskell (1990, T:41) je čisto fukcionalni jezik nastao s ciljem konsolidacije postojećih funkcionalnih jezika i kao baza za buduće.

U sljedećem primjeru iz jezika Haskell vidi se definisanje faktorijela na bazi njegovih parametara, bez njihove analize unutar same funkcije.

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Standard ML (1990, T:47) predstavlja aktuelni dijalekt jezika ML. OCAML (1996, T:100+) dodaje jeziku CAML konstrukcije iz objektno orijentisanog programiranja.

Alice (2000, T:51+) proširuje Standard ML mogućnostima da se izraz vrijednost u odvojenoj niti, a da eventualno pristupanje toj vrijednosti prije vremena vodi do blokade niti. F # (2002, T:33) se koristi u .NET okruženju, uglavnom kao funkcionalni jezik, a uključuje elemente ML i OCAML jezika.

Scala (2003, T:51+) uključuje objektno-orijentisane i funkcionalne elemente i izvršava se u Java izvršnom okrućenju. ATS (2005) je jezik za dokazivanje teorema u kombinaciji s klasičnim programiranjem kroz napredni sistem tipova.

Jezici za obradu nizova

Neki programski jezici u svoje jezgro imaju ugrađene naredbe za manipulaciju nizovima i matricama, čineći operacije poput računanja inverznih matrica trivijalnim.

APL (1964, T:50) je koncizan jezik za manipulisanje nizovima sa specijalnim skupom znakova umjesto ključnih riječi. IDL (1977, T:51+) sintaksno je bliži FORTRAN-u i koristi se za interaktivnu obradu velikih količina podataka. Primjer APL programa koji kreira niz od 5 elemenata, svaki uvećava za 10 te prikazuje sumu cjelog niza:

```
N ←10 34 23 42 32
N + 10
```

Matlab (1978, T:23) omogućava širok spektar manipulacija s matricama i crtanje funkcija, a njegov jezik podržava i imperativno programiranje. Nial (1981) definiše sve podatke kao nizove nizova, uključujući i skalarne. Operatori se mogu pisati u više različitih notacija. A+ (1988) je modifikacija APL koja omogućuje razdvajanje naredbi u više redova. J (1990, T:51+) predstavlja sintezu jezika APL, FP i FL. K (1993) je srodan APL-u ali koristi standardni ASCII skup znakova. Q (2003, T:45) je upitni jezik razvijen iznad K koji omogućava čitljiviji interfejs.

Jezici funkcionalnog nivoa

Jezici funkcionalnog nivoa potpuno zanemaruju operacije nad vrijednostima i koncentrišu se na funkcije. FP (1977) je započeo usku klasu funkcionalnih jezika koja eliminiše varijable. FL (1989) je nasljednik jezika FP sa nekim elementima iz jezika ML.

Jezici više paradigmi

Mnogi jezici ranije spomenuti u tekstu se mogu staviti u više kategorija. Tako na primjer, u C++ je moguće pored dominantog objektno orijentisanog programiranja, programirati i proceduralno, nestrukturirano, pa i funkcionalno. Ipak neki programski jezici toliko ističu svoju pripadnost različitim kategorijama (paradigmama) da ih je teško svrstati u jednu od njih. Mathematica (1988, T:100+) je softverski paket za numeričku i simboličku matematiku a njegov programski jezik podržava više stilova

programiranja. Oz (1991, T:100+) sadrži većinu koncepta glavnih paradigmi programiranja: logičku, funkcionalnu, imperativnu i OO. Python (1991, T:8) se najčešće koristi kao skriptni jezik, a odlikuje se čitljivim kodom i upotrebom uvlačenja blokova umjesto ključnih riječi.

Slijedi primjer programa u jeziku Python koji definiše funkciju što spaja dva stringa:

```
def hello (what):
   text = "Hello, " + what + "!"
   print text
hello ("World")
```

. Go! (2003) namijenjen za agentno bazirano programiranje, podržava logičko, funkcionalno i objektno bazirano programiranje.

Rezime

Tri osnovna načina programiranja: ožičeno, imperativni jezici i deklarativni jezici, se dalje dijele na podkategorije.

Ožičeno programiranje je direktno pravljenje sklopova koji rješavaju dati problem.

Imperativni jezici su najširа kategorija, i dijele se na jezike niskog i visokog nivoa. Podkategorije jezika visokog nivoa su nestruktuirani, proceduralni i objektni jezici.

3. Opis sintakse i semantike

Studija programskih jezika, poput studije prirodnih jezika se dijeli na ispitivanje sintakse i semantike.

Sintaksa jezika je forma ili struktura njegovih izraza, naredbi i programskih jedinica. Semantika jezika predstavlja značenje tih izraza, naredbi i programskih jedinica Na primjer, sintaksa Java while naredbe je

```
while (<boolean_expr>)< statement>
```

Sintaksa i semantika zajedno daju definiciju jezika . Korisnici definicije jezika su implementatori (oni koji programiranju kompajlere ili interpretere tog jezika), programeri aplikativnih programa u tom programskom jeziku, pa i dizajneri drugih jezika (dobijajući ideje iz postojećih jezika).

Elementi jezika

Jezik, bio on prirodni (poput bosanskog) ili vještački (poput C-a) je skup nizova znakova nekog alfabeta. Nizovi simbola jezika se zovu rečenice. Sintaksna pravila jezika definišu koji nizovi znakova iz jezičkog alfabeta predstavljaju jezik.

Formalni opis sintakse programskog jezika, radi jednostavnosti, često ne uključuje opis sintaksnih jedinica na najnižem nivou, Te male jedinice se zovi **leksemi**. Opis leksema se može dati po leksičkoj specifikaciji koja je obično drugačija od sintaksnog opisa jezika. Leksemi programskoj jezika uključuju numeričke literale, operatore i specijalne riječi. O programima se može govoriti kao o nizovima leksema umjesto znakova.

Leksemi se kategorišu po grupama. Na primjer imena varijabli, metoda klasa i drugo u programskom jeziku čine grupu koja se zove identifikatori. Svaka grupa leksema je predstavljena imenom ili tokenom. Stoga je token jezika kategorija njegovih leksema.

Na primjer, identifikator je token koji može imati lekseme, ili instance, kao što su sum i total. U nekim slučajevima, token ima samo jedan mogući leksem Primjer je token koji predstavlja aritmetički operator +.

U sljedećoj Java rečenici,

```
index = 2 * count + 17;
```

Leksemi i tokeni ove rečenice su

Leksem	Token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Prepoznavači i generatori

U principu, jezici se mogu formalno definisati na dva različita načina: prepoznavanjem i generisanjem(iako ni jedno ni drugo ne daje definiciju koja je praktična sama po sebi za ljude koji pokušavaju da nauče ili koristiti programski jezik).

Dio kompajlera koji obavlja sintaksnu analizu je **prepoznavač** za jezik koga kompajler prevodi. U toj ulozi, prepoznavač ne mora testirati sve moguće nizove znakova iz nekog skupa kako bi se utvrdilo da li je svaki pripada tom jeziku.

Generator jezika je program ili uređaj koji se može koristiti za generiranje rečenica jezika. Pošto je broj mogućih rečenica u jeziku gotovo beskonačan, a generatori izabiraju samo neke, generatori imaju ograničenu upotrebljivost za definisanje programskog jezika.

Međutim, ponekad su generatori praktičniji od prepoznavača, jer se oni mogu lakše čitati i razumjeti. Na primjer, analiza dijela kompajlera za sintaksnu provjeru je teži način upoznavanja programskog jezika od unosa probnih rečenica provjere da li je kompajler prihvata.

Formalni metodi opisa sintakse

Formalni metodi generisanja jezika, koji se obično zovu gramatike, koriste se za opis sintakse programskih jezika.

Noam Chomsky sredinom 1950-ih j definisao četiri klase gramatika: opšte, kontekstno ovisne, kontekstno nezavisne i regularne. Dvije vrste od ovih gramatika se koriste za opis sintakse programskih jezik: regularne i kontekstno nezavisne gramatike. Regularnim gramatikama se opisuje oblik tokena, dok se opis cijelog jezika uz minorne izuzetke opisuje kontekstno nezavisnim gramatikama.

Backus – Naurova forma

Jedna vrsta jezičkih generatora namijenjena za opis sintakse prirodnih i kompjuterskih jezika je Backus-Naurova forma (1959) , skraćeno BNF. Prvi ju je predstavio John Backus za opis Algol 58 programskog jezika. Kroz BNF se definiše klasa jezika kontekstno nezavisne gramatike

U BNF uočavaju se neterminalni simboli (pojmovi), terminalni simboli, pravila i startni simbol., Pojmovi se koriste da predstave klasu sintaksnih struktura. Oni se ponašaju kao sintaksne varijable, (poznati i kao neterminalni simboli). Neterminalni simboli se često uokviravaju u oštrougle zagrade. Ili se pišu kosim slovima. Terminalni simboli su leksemi ili tokeni , i pišu se podebljanim slovima ili pod navodnicima. Pravila imaju lijevu stranu (LHS), koja je jedan neterminalni simbol i desnu stranu (RHS) koja je niz terminalnih i/ili neterminalnih sibola

Primjeri BNF pravila:

```
<ident_list> →identifier | identifier, <ident_list>
<if_stmt> →if <logic_expr> then <stmt>
```

Startni simbol je specijalni element iz skupa neterminalnih simbola gramatike . Gramatika se ponaša kao generator rečenica jezika, razvijanjem od startnog simbola.

Pojam, (apstrakcija ili nenterminalni simbol) može imati više desnih strana , pri čemu znak uspravne crte predstavlja izbor između njih.

```
<stmt> →<single_stmt>
| begin <stmt_list> end
```

U osnovnoj BNF, lista od više elemenata se ne označava s trotačkom na njenom kraju, nego se koristi rekurzivna definicija.

```
<ident_list> →ident
| ident, <ident_list>
```

Izvođenje je ponavljanje pravila, počevši od startnog simbola a završavajući rečenicom koja se sastoji samo od terminalnih simbola)

Primjer izvođenja daće se na sljedećoj jednostavnoj gramatici.

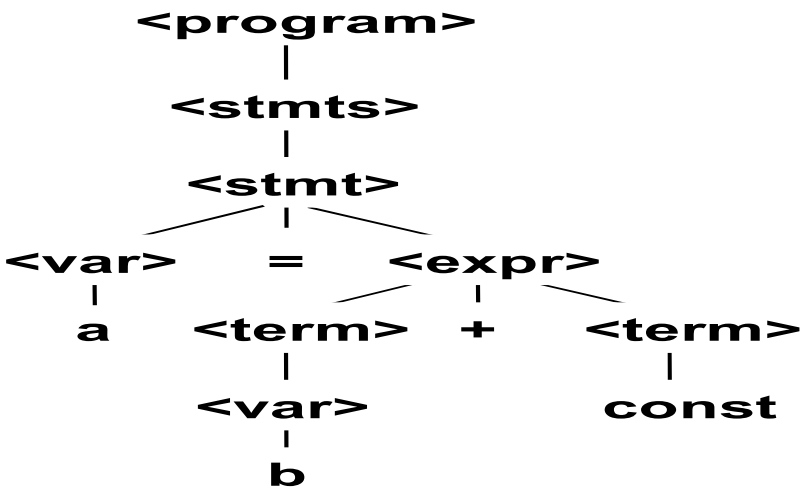
```
<program> →<stmts>
<stmts> →<stmt> | <stmt> ; <stmts>
<stmt> →<var> = <expr>
<var> →a | b | c | d
<expr> →<term> + <term> | <term> - <term>
<term> →<var> | const
```

Izvođenje se obavlja zamjenom neterminalnih simbola jednom od mogućih desnih strana sve dok se ne dobiju samo terminalni simboli. Time je generisana jedna od rečenica u datom jeziku. U primjeru gore navedene gramatike imamo sljedeće izvođenje.

```
<program> => <stmts> => <stmt>
=> <var> = <expr>
=> a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const
```

Ovo izvođenje je stalno raščlanjivalo krajnji lijevi neterminalni simbol u svakoj formi rečenice. Takvo izvođenje se zove krajnje lijevo izvođenje. Ako se raščlanjuje krajnji desni neterminalni simbol, takvi izvođenje se zove krajnje desno izvođenje. Izvođenje može biti da nije ni krajnje lijevo ni krajnje desno.

Izvođenje iz ovog primjera se može vizuelno predstaviti hijerarhijskim stablom parsiranja. Na ovom stablu čvorovi predstavljaju neterminalne simbole koji se dalje



raščlanjuju, dok listovi predstavljaju terminalne simbole.

Višeznačnosti u gramatici

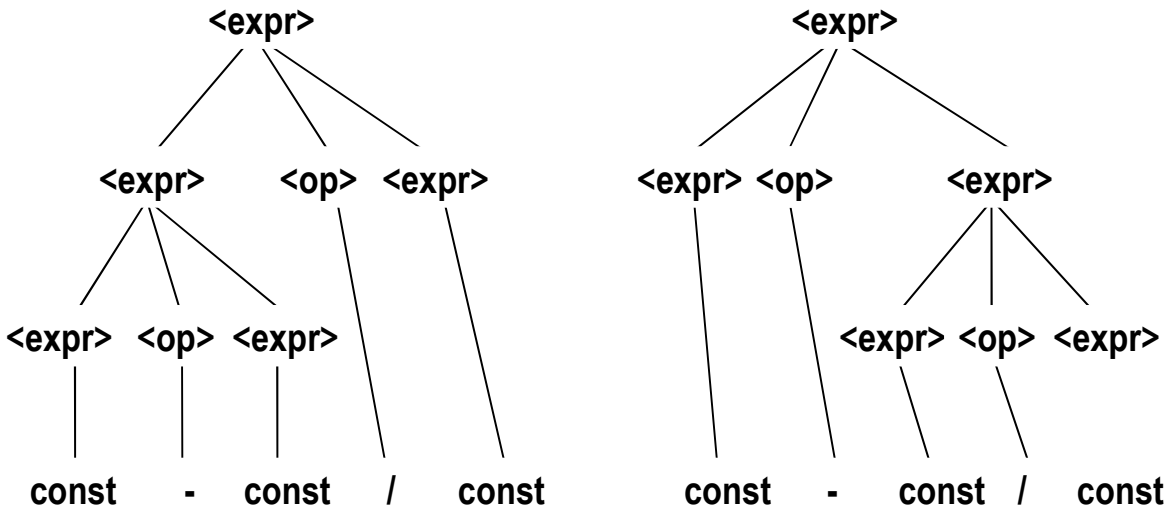
Gramatika je višeznačna ako i samo ako generiše oblike rečenica koii mogu da imaju dva ili više različitih stabala parsiranja

Sljedeća gramatika je višeznačna.

```
<expr> →<expr> <op> <expr> | const
<op> →/ | -
```

U ovakvoj gramatici se izraz const-const/const može generisati kroz dva različita stabla parsiranja.

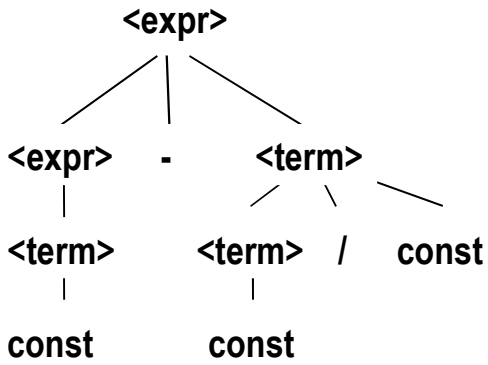
Kako višeznačnost gramatike čini prepoznavače nekonzistentim, potrebno je



konvertovati gramatiku u gramatika s jednoznačnim izrazom. Iako to nije uvijek moguće,u ovom primjeru se to postiže uvođenjem prioriteta operatora. Ako odredimo stablo parsiranja s prioritetom operacija, nećemo imati višeznačnost Gramatika se transformiše u sljedeću.

```
<expr> →<expr> - <term> | <term>
<term> →<term> / const| const
```

Dobija se ovakvo stablo parsiranja.



Asocijativnost operatora

Kada izraz ima više operatora istog prioriteta (npr. a+b+c-d), redoslijed kojim se računaju dijelovi izraza istog prioriteta se zove asocijativnost operatora. Asocijativnost operatora se takođe može odrediti gramatikom

Sljedeće pravilo je višeznačno, u jer npr. u izrazu a+b+c nije jasno da li se prvo računa a, b ili c

```
<var> -> a|b|c
<expr> -> <expr> + <expr> | <var>
```

Pravilo napisano u ovom obliku je jednoznačno, i u izrazu a+b+c se uvijek prvo računa a, pa b pa c.

```
<var> -> a|b|c
<expr> -> <expr> + <var> | <var>
```

Prošireni BNF

U proširenoj verziji BNF, skraćeno EBNF, uvedeni su dodatni elementi u sintaksi pravila kako bi se pojednostavio opis nekih gramatika. Ti elementi su opcioni dijelovi gramatika, alternative između desnih strana, ponavljanja izraza.

Opcioni dijelovi se stavljaju u uglaste zagrade [ ] . U sljedećem primjeru, Neterminalni simbol <proc_call> se može razviti u ident ili ident iza koga slijedi razvijanje <expr_list> unutar zagrada-

```
<proc_call> -> ident [(<expr_list>)]
```

Alternativni dijelovi desnih strana se stavljaju u oble zagrade razdvojeni uspravnim linijama

```
<term> →<term> (+|-) const
```

Ponavljanja (0 ili više puta) se smještaju u vitičaste zagrade { }

```
<ident> →letter {letter|digit}
```

Svaka gramatika koja se može predstaviti u EBNF može se predstaviti i u BNF , samo složenije. Sljedeće dvije gramatike su ekvivalentne, premda je EBNF verzija preglednija.

BNF

```
<expr> →<expr> + <term>
| <expr> - <term>
| <term>
<term> →<term> * <factor>
| <term> / <factor>
| <factor>
```

EBNF

```
<expr> →<term> {(+ | -) <term>}
<term> →<factor> {(* | /) <factor>}
```

Novije varijacije u EBNF , koje se mogu naći u literaturi su

- Alternativno desne strane se smještaju u posebne linije
- Upotreba dvotačke umjesto =>
- Upotreba opt za opcionalne dijelove umjesto uglastih zagrada
- Upotreba oneof za izbore umjesto uspravnih linija unutar oblih zagrada

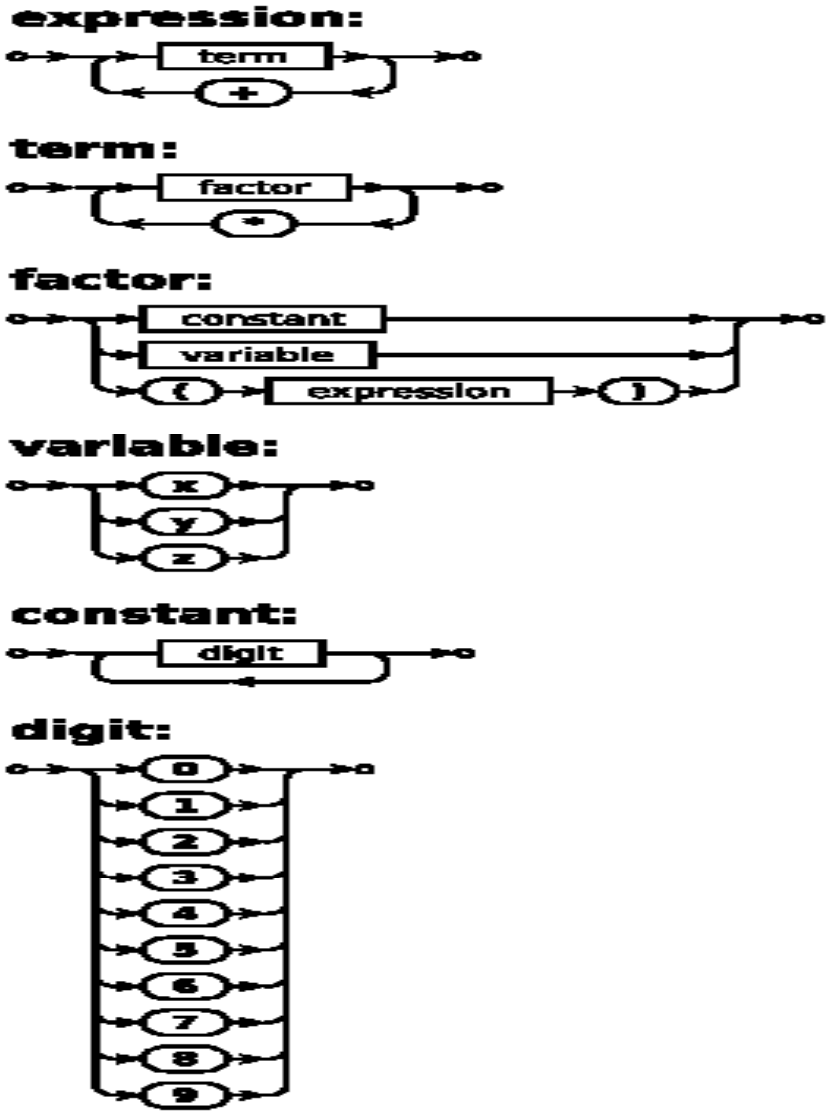
Sintaksni dijagrami

Sintaksni dijagrami su još jedan način predstavljanja sintakse jezika. Manje kompaktni su od BNF, ali su vizuelno razumljiviji. Kod ovih dijagrama se prati sintaksa strelicama između neterminalnih i terminalnih simbola. Neterminalni simboli su predstavljeni pravougaonicima, dok su terminalni simboli predstavljeni ovalnim ili kružnim likovima.

Neka je data sljedeća EBNF gramatika.

```
expression = term | expression, "+" , term;
term       = factor | term, "*" , factor;
factor  = constant | variable | "(" , expression , ")";
variable   = "x" | "y" | "z";
constant   = digit , {digit};
digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Ova gramatika predstavljena kroz sintaksni dijagram izgleda ovako.



Statička Semantika

Sintaksna predstava jezika definiše samo određena formalna pravila, bez ikakvog ulaska u značenje rečenica. Kontekstno neovisne gramatike (CFG), kakve su obično definisane u BNF, ne mogu opisati svu sintaksu programskih jezika Konstrukcije koje predstavljaju problem:

- Gramatika se može predstaviti kao kontekstno neovisna ali bi bila nezgrapna npr. u Java je dopušteno dodijeliti cijeli broj float broju, ali nije dopušteno float dodijeliti cijelom. Definisanje ovog pravila bi znatno povećao broj pravila i terminalnih simbola.
- Gramatika je kontekstno ovisna (npr. Varijable se moraju deklarisati prije upotrebe)

Ovakva pravila se radije definišu statičkim semantičkim pravilima, nego sintaksnim pravilima. Semantička pravila se mogu primjenjivati u trenutku prevođenja i u trenutku izvršavanja programa u navedenom jeziku. Semantički pravila koja se primjenjuju u trenutku prevođenja, a samo indirektno utiču na izvršenje nazivaju se statičkom semantikom.

Atributske gramatike

Atributske gramatike (AGs) predstavljaju dodatke za kontekstno nezavisne gramatike koji uključuju semantiku u čvorove stabla parsiranja. Atributska gramatika se primarno koristi za navođenje statičke semantike. Posebno su bitne u dizajnu kompajlera.

Atributska gramatika je kontekstno nezavisna gramatika G = (S, N, T, P) sa sljedećim dodacima

- Za svaki gramatički simbol x postoji skup A(x) vrijednosti atributa
- Svako pravilo ima skup funkcija koji definiše određene atribute neterminalnih simbola u pravilu
- Svako pravilo ima (moguće prazan) skup predikata za provjeru na konzistentnost attributa

Atributi se dijele u dvije kategorije, sintetizovane i naslijeđene. Sintetizovani atributi prosljeđuju informacije prema vrhu stabla parsiranja. Naslijeđeni atributi prosljeđuju informacije niz stablo parsiranja prema dubljim čvorovima i listovima.

Neka je definisano pravilo oblika

$X_0 \rightarrow X_1 \dots X_n$

Sintetizovani atributi se definišu i računaju funkcijama oblika

$S(X_0) = f(A(X_1), \dots, A(X_n))$

Naslijeđeni atributi se definišu i računaju funkcijama oblika

$I(X_j) = f(A(X_0), \dots, A(X_n))$, za $1 <= j <= n$,

Uz atribute se definišu i predikatske funkcije, koje su Bulov izraz između skupa atributa $f(A(X_0), \dots, A(X_n))$. Jedina dopuštena izvođenja u atributskoj gramatici su ona u kojoj svaki predikat pridružen neterminalnom simbolu ima tačnu vrijednost.

Kategorija sintetizovanih atributa koji se određuju u listovima stabla parsiranja izvan stabla parsiranja se zove vlastiti atributi.

Na primjer, neka je data sljedeća kontekstno nezavisna gramatika.

```
<assign> →<var> := <expr>
<expr> →<var> + <var>
<expr> →<var>
<var> →A | B | C
```

Neka treba definisati pravila slaganja tipova u ovoj gramatici. Varijable mogu biti int ili real tipa. . Kada tipovi operanada u <expr> nisu isti, rezultantni tip je uvijek real. Kada su isti, rezultantni tip odgovara tipu operanada. Definisaćemo sintetizovani atribut actual_type za <var> i <expr> koji je dobijen iz izraza i naslijeđeni atribut expected_type za <expr> koji je određen tipom varijable s lijeve strane izraza dodjeljivanja. Na mjestima gdje se javlja više od jednog primjerka istog neterminalnog simbola u pravilu, koriste se indeksi u uglastim zagradama za izbor jednog od njih. Atributska gramatika koja definiše proširenje gore date gramatike izgleda ovako.

Sintaksno pravilo	Semantičko pravilo	Predikat
<assign> → <var>= <expr>	<expr>.expected_type ← <var>.actual_type	
<expr> → <var>[2]+ <var>[3]	<expr>.actual_type ← if (<var>[2].actual_type =int) and (<var>[3].actual_type =int) then int else real end if	<expr>.actual_type == <expr>.expected_type
<expr> → <var>	<expr>.actual_type ← <var>.actual_type	<expr>.actual_type == <expr>.expected_type
<var> → A | B | C	<var>.actual_type ← lookup(<var>.string)	

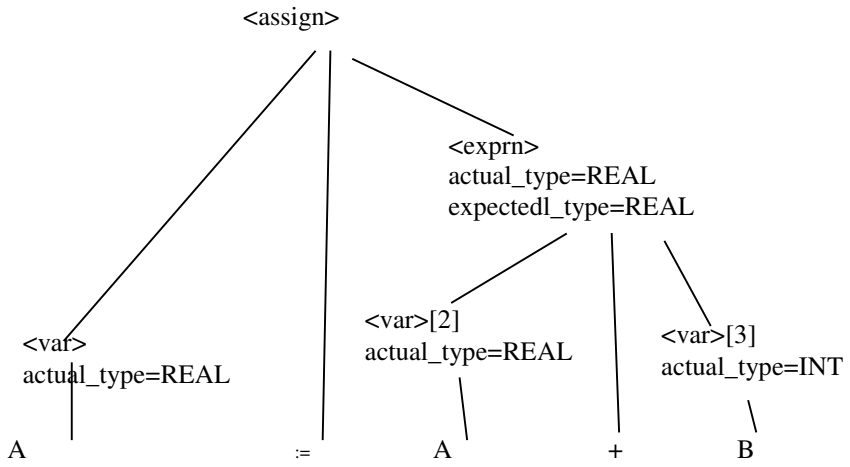
Funkcija lookup gleda tablicu identifikatora i vraća tip varijable.

Vrijednosti atributa se računaju prolaskom kroz stablo parsiranja. Ako su svi atributi naslijeđeni, stablo se treba proći odozgo nadolje. Ako su svi atributi sintetizovani, stablo se treba proći odozdo nagore .U mnogim slučajevima koriste se oba tipa atributa, pa se mogu koristiti kombinacije odozgo nadolje i odozdo nagore.

Neka je varijabla A definisana kao real a varijabla B kao int i neka e dat izraz

A:=A+B

Na sljedećoj slici se vide atributi dodjeljeni čvorovima stabla parsiranja.



Dinamička Semantika

Dok je opis sintakse jezika dosta standardizovan kroz BNF i sintaksne dijagrame, nema jedinstvene prihvaćene notacije ili formalizma za opis semantike. Dodavanje atributa u BNF nije dovoljno za potpuno objašnjavanje šta naredbe nekog programskog jezika rade, pa su nastali različiti načini za predstavljanje ponašanja naredbi tokom njihovog izvršenja, **dinamičke semantike**. Korisne su jer omogućavaju programerima da znaju šta naredbe znače, a autorima kompajlera moraju da tačno znaju šta rade jezičke konstrukcije. Ako je semantika formalno definisana, postaju mogući i dokazi korektnosti i generatori kompajlera. Ovdje će biti spomeniti operaciona, aksiomatska i denotacijska semantika.

Operaciona semantika

U udžbenicima programskih jezika se često funkcionalnost složenijih naredbi objašnjava kroz jednostavnije. Na primjer, u udžbenicima jezika C se naredba for objašnjava sljedećim ekvivalentom.

C Statement	Meaning:

for (expr1; expr2; expr3) {
...
}
	expr1;
	loop: if expr2 == 0 goto out
	...
	expr3;
	goto loop
	out: ...

Na ovom prncipu se bazira i operaciona semantika. Ideja operacione semantike je da opiše značenje naredbe ili programa definišući njen efekt kada bi se izvršavala na mašini. Prvi korak u definisanju operacione semantike je u definisanju među-jezika koji treba da bude što jasniji. Taj među-jezik predstavlja virtualni kompjuter koji treba da izvršava instrukcije korektno i prepozna efekte izvršenja. Kao primjer takvog jezika može se definisati lista naredbi adekvatna za opis semantike prostih kontrolnih naredbi tipičnog programskog jezika.

```
ident  = var
ident  = ident +1
ident  = ident –1
goto  label
if  var relop var goto label
```

U ovim naredbama, relop je jedan od relacionih operatora iz skupa {=, <>, >, <, >=, <=},ident je identifikator, a var je identifikator ili konstanta Ove naredbe su jednostavne i lake za implementaciju.

Operaciona semantika je praktična ako se koristi neformalno (jezička uputstva itd). Vrlo je kompleksna ako se koristi formalno (npr., VDL korišten za opis semantike jezika PL/I.)

Denotacijska semantika

Denotacijska semantika je najrigoroznija metoda opisa sematike. Bazirana na teoriji rekurzivnih funkcija, razvili su je Scott i Strachey (1970). U ovoj semantici se svaki jezički entitet mapira u odgovarajuće matematske objekte, skupove ili funkcije. Proces gradnje denotacijske semantike zahtijeva da se definiše matematski objekt za svaki jezički entitet i da se definišu funkcije koja mapiraju instance jezičkih entiteta u instance koje odgovaraju matematskim objektima.

U ovoj semantici, značenje jezičkih konstrukcija definisano je samo vrijednostima programskih varijabli. Stanje programa je vrijednost svih njegovih varijabli .

$s = \{<i_1, v_1>, <i_2, v_2>, ..., <i_n, v_n>\}$

Ovdje i_1, i_2,, ..., i_n predstavljaju imena varijabli, a v_1, v_2,, ..., v_n predstavljaju njihove vrijednosti.

Neka je VARMAP funkcija koja kada joj je dato ime varijable i stanje, vraća trenutnu vrijednost varijable u navedenom stanju s.

VARMAP(i_j, s) = v_j

Denotacijska semantika se može ilustrovati na sljedećem jednostavnom primjeru koji objašnjava značenje cijelih brojeva. Neka je data sljedeća gramatika.

```
<dec_num> →'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |
<dec_num> ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
```

Denotacijska mapiranja za ovo pravilo su sljedeća

```
Mdec('0') = 0, Mdec ('1') = 1, …, Mdec ('9') = 9
Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>)
Mdec (<dec_num> '1') = 10 * Mdec (<dec_num>) + 1
…
Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9
```

Neka su data dalje sljedeća pravila koja definišu izraze. Pretpostavljamo da su izrazi decimalni brojevi varijable ili binarni brojevi s jednim aritmetičkim operatorom i dva operanda od kojih svaki može biti izraz.

```
<expr>→<dec_num> | <var> | <binary_expr>
<binary_expr>→<left_expr><operator><right_expr>
<left_expr>→<dec_num> | <var>
<right_expr>→<dec_num> | <var>
<operator>→+ | -
<assign>→<var> := <expr>
```

Mapirajuća funkcija za izraz <expr> i stanje s, izgleda ovako.

```
Me(<expr>, s)  =
case <expr> of
  <dec_num> => Mdec(<dec_num>, s)
  <var> =>
    if VARMAP(<var>, s) == undef
    then error
    else VARMAP(<var>, s)
  <binary_expr> =>
    if (Me(<binary_expr>.<left_expr>, s) == undef
    OR Me(<binary_expr>.<right_expr>, s) == undef)
    then error
    else
      if (<binary_expr>.<operator> == '+' then
        Me(<binary_expr>.<left_expr>, s) +
        Me(<binary_expr>.<right_expr>, s)
      else Me(<binary_expr>.<left_expr>, s) *
        Me(<binary_expr>.<right_expr>, s)
```

Notacija s tačkom predstavlja dijete čvor u stablu parsiranja. Na primjer, <binary_expr>.<right_expr> odnosi se na desni dijete čvor neterminalnog simbola <binary_expr>.

Mapirajuća funkcija za naredbu dodjele <assign>i stanje s, koja mapira skupove stanja u skupove stanja U {error} izgleda ovako.

Ma(<assign>, s) =
if Me(<assign>.<expr>, s) == error
then error
else s' = {<i_1,v_1'>,<i_2,v_2'>,...,<i_n,v_n'>},
  where for j = 1, 2, ..., n,
  if i_j == x
  then v_j' = Me(<assign>.<expr>, s)
  else v_j' = VARMAP(i_j, s)

Denotacijskom semantikom se mogu predstavljati i petlje tako što je petlja je konvertovana iz iteracije u rekurziju jer je rekurziju lakše matematski predstaviti nego iteraciju

Denotacijska semantika može se koristiti za dokaz korektnosti programa. Ona pruža rigorozan način razmišljanja o programima. Zbog kompleksnosti, od male je koristi za korisnike jezika, ali je korisna u dizajnu jezika. Korištena je i u sistemima za generisanje kompajlera

Aksiomatska semantika

Aksiomatska semantika je najapstraknija metoda za opis semantike, bazirana na formalnoj logici (predikatski račun). Izvorno je korištena u formalnoj verifikaciji programa. Ne koristi se mašina stanja. Umjesto definisanja značenja programa, ova semantika opisuje šta se može dokazati o programu. Aksiomi ili pravila izvođenja su definisani za svaku vrstu naredbi u jeziku (da se dopusti trasformacija izraza u formalnije logičke izraze). Logički izrazi se zovu tvrdnje.

Ako je tvrdnja data prije naredbe, zove se preduslov i navodi odnose i ograničenja među varijablama koje trebaju ispunjeni u tom trenutku izvršenja. Tvrdnja nakon naredbe je postuslov i definiše nova ograničenja nad varijablama. Najslabiji preduslov je najmanje restriktivan preduslov koji će garantovati postuslov .

Aksiomatska semantika se koristi u dvije primjene: definisanje značenja naredbi i dokazivanje korektnosti programa.

U obije primjene, kada se koristi aksiomatska semantika, svaka naredba se piše u sljedećem obliku

{P} statement {Q}

gdje {P} predstavlja preduslov, a {Q} predstavlja postuslov. Preduslovi i postuslovi najčešće se opisuju logičkim izrazima. Ako se aksiomatska semantika koristi za opis semantike naredbi, ovi uslovi opisuju efekat naredbe. Na primjer, semantika za operator ++ u C jeziku može se opisati na sljedeći način

{variable==x} variable++ {variable==x+1}

Ako se aksiomatska semantika koristi za dokazivanje korektnosti programa, preduslovi možda nisu poznati nego se dobijaju iz postuslova.

Neka je dat sljedeći primjer, pri čemu su varijable a i b cjelobrojne.

a = b + 1 {a > 1}

U datom primjeru moguće je imati više preduslov:a {b > 10}, {b>1} ,{b>100},... i svaki će od njih garantovati da važi a>1, ali je najslabiji preduslov koji zadovoljava postuslov: {b > 0}

Dokazivanje tačnosti programa se obavlja na sljedećem principu. Definiše se postuslov za cijeli program kao željeni rezultat. Iz zadnje naredbe se dobijaju njeni preduslovi, koji postaju postuslovi predzadnje naredbe. Nastavlja se nazad kroz

program do prve naredbe. Ako je preduslov prve naredbe isti kao specifikacija programa, program je ispravan.

Proces dokazivanja korektnosti programa baziran je na nekim aksiomama i pravilima izvođenja.

Pravila izvođenja se pišu u sljedećem obliku.

$$\frac{S1,S2,S3,\ldots Sn}{S}$$

Ova notacija kaže da ako su ispunjena sva pravila Sl, S2,...Sn, tada važi S.

Aksiom za naredbe dodjele kaže da se preduslov može napisati tako da se u postuslovu zamijeni varijabla koja predstavlja lijevu stranu izraza dodjeljivanja zamijeni desnom stranom.

$(x = E)$: $\{Q_{x->E}\}$ x = E {Q}

Primjer: Naći najslabiji preduslov za naredbu s postuslovom

x=2*y-3 {x>25}

Zamijenivši x u uslovu dobijamo

2*y-3>25

y>14

Pravilo posljedica: Ako važi {P}S{Q}, i ako je P posljedica P', a Q posljedica Q' tada je

$$\frac{\{P\}\,S\,\{Q\},P' \Rightarrow P,Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

Npr, pošto je već dokazano {y>14} x=2*y-3 {x>25}, a kako y>30⇒y>14, a x>40⇒y>40, tada važi {y>30} x=2*y-3 {x>40}

Aksiom sekvencijalnog izvršavanja: Ako važe pravila

{Pl} Sl {P2}

{P2} S2 {P3}

Tada važi

$$\frac{\{\mathrm{P1}\}\ \mathrm{S1}\ \{\mathrm{P2}\}\ ,\{\mathrm{P2}\}\ \mathrm{S1}\ \{\mathrm{P3}\}}{\{\mathrm{P1}\}\ \mathrm{S1,S2}\ \{\mathrm{P3}\}}$$

tj. Sl i S2 se korektno izvršavaju jedna za drugom

Na bazi ovih aksioma, izvode se pravila za provjeru jezičkih konstrukcija. Na primjer **pravilo izvođenja za petlju** s provjerom uslova na početku

{P} while B do S end {Q}

se piše u obliku

$$\frac{\{I \wedge B\}\,S\,\{I\}}{\{I\}\,whileBdoS\,\{I \wedge \neg B\}}$$

gdje je i invarijanta petlje. Određivanje invarijante petlje je teško. Ona mora zadovoljiti sljedeće uslove:

- P => i (invarijanta petlje mora u početku biti tačna)
- {I} B {I} (provjera Bulovog izraza ne smije mijenjati vrijednost I)
- {I and B} S {I} (i se ne mijenja izvršenjem tijela petlje)
- (I and (not B)) => Q (Ako je i tačno a B netačno, Q slijedi iz toga)
- Petlja se završava (ovo je teško za dokazati)

Primjer:

while y<>x do y=x+1 end {y=x}

Ako se za preduslov P i invarijantu petlje i uzme uslov y<=x, ova invarijanta petlje ispunjava potrebne uslove.

Razvoj aksioma ili pravila izvođenja za sve naredbe jezika je težak. Aksiomatska semantika je dobar alat za dokaze korektnosti i odličan okvir za zaključivanje o programima, ali ne tako koristan za korisnike programskog jezika i autore kompajlera. Upotrebljivost aksiomatske semantike u opisu značenja programskog jezika ograničena za korisnike programskog jezika i autore kompajlera

Rezime

BNF i kontekstno nezavisne gramatike su ekvivalentni metajezici. Praktični su za opis sintakse programskog jezika. Atributska gramatika za statički opis semantike je deskriptivni formalizam koji može opisati i sintaksu i semantiku jezika. Tri primarna metoda opisa za dinamički opis semantike su operaciona, aksiomatska i denotacijska semantika.

4. Leksička i sintaksna analiza

Sistemi za implementaciju programskog jezika moraju analizirati izvorni kod, bez obzira na specifični pristup kod implementacije (kompajliranje, interpretiranje). Skoro sva sintaksna analiza je bazirana na formalnom opisu sintakse izvornog jezika (BNF).

Sintaksna analiza

Dio sintaksne analize jezičkog procesora gotovo uvijek se sastoji od dva dijela

Dio nižeg nivoa, nazvan **leksički analizator** se bavi malim jezičkim konstrukcijama, kao što su ključne riječi, varijable i brojevi. Matematski on je konačni automat baziran na regularnoj gramatici)

Dio višeg nivoa, nazvan **sintaksni analizator**, ili parser se bavi analizom većih jezičkih konstrukcija, poput uslova, petlji i potprograma. Mmatematski, on je push-down automat baziran na kontekstno slobodnoj gramatici, ili BNF. Prednosti korištenja BNF za opis sintakse su što BNF pruža čist i koncizan opis sintakse jezika. Na bazi BNF se može generisati parser za željeni jezik, bilo ručno programirajući ga ili korištenjem alata za generisanje parsera.

Razlozi razdvajanja leksičke i sintaksne analize u različite faze su

- Jednostavnost: Leksička analiza zahtijeva jednostavnije algoritme nego sintaksna, ali uključivanje leksičkog i sintaksnog dijela u isti proces bi učinilo sintaksnu analizu previše kompleksnom.
- Efikasnost: Pošto se više procesorskog vremena provodi u leksičkoj analizi, može se više koncentrisati na optimizaciju leksičkog analizatora
- Portabilnost: Dijelovi leksičkog analizatora mogu biti neportablini (na primjer, simbol novog reda je različit na Windows, Unix i MacOS sistemima) a da parser i dalje ostane portabilan

Leksička analiza

Leksički analizator uparuje nizove znakova sa uzorcima. On predstavlja pripremu za parser, sintaksni analizator. Leksički analizator pretvara prosti niz znakova na ulazu u podnizove ulaznog programa koji pripadaju zajedno – lekseme. Leksemi pripadaju određenoj leksičkoj kategoriji koja se zove **token**.

Na primjer, sum je leksem, čiji token je IDENT, 1124 je leksem koji token je NUMBER, itd.

Leksički analizator je obično funkcija koju poziva parser kada mu je potreban sljedeći token. Može se praviti na tri osnovna načina.

1. Napisati formalni opis tokena i koristiti softverski alat koji konstruiše tablično upravljani leksički analizator prema ovakvom opisu

2. Dizajnirati dijagram stanja koji opisuje tokene i napisati program koji implementira dijagram stanja

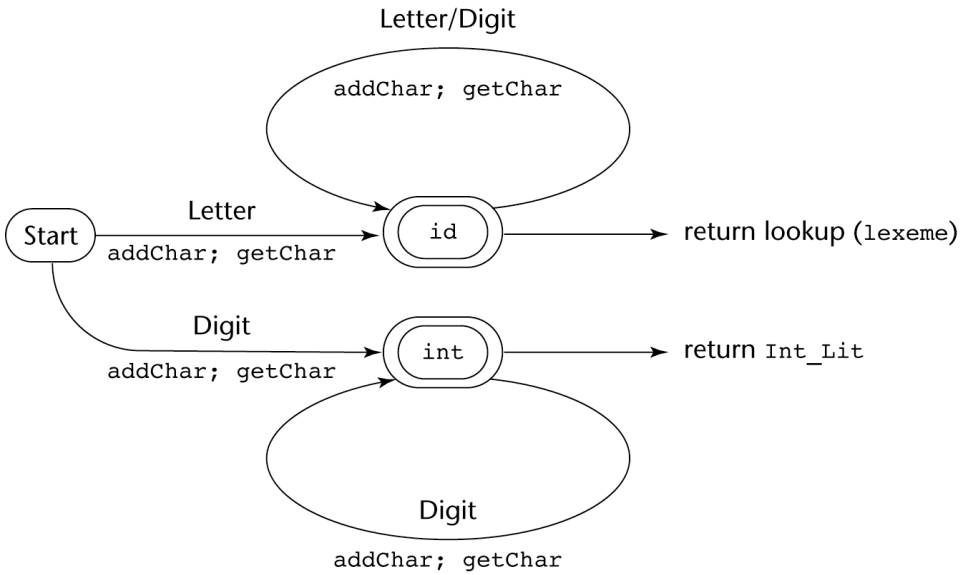
3. Dizajnirati dijagram stanja koji opisuje tokene i ručno napisati program koji tablično implementira dijagram stanja

Dizajn dijagrama stanja

Dijagram stanja je usmjereni graf. Njegovi čvorovi su označeni imenima stanja. Lukovi su označeni znakovima koji ako se pojave na ulazu postižu tranziciju iz stanja u stanje.

Na prvi pogled, morala bi se napraviti tranzicija za svaki mogući simbol na ulazu iz svakog stanja . Takav dijagram bi bio veoma velikim pa su potrebna neka pojednostavljenja. U mnogim slučajevima, prijelazi se mogu kombinovati radi pojednostavljenja dijagrama stanja. Tako na primjer, pri prepoznavanju identifikatora sva mala ili velika slova su ekvivalentna. Tada ih možemo sve grupisati u klasu znak koja može uključiti sva slova. Slično, pri prepoznavanju cijelog broja, sve cifre su ekvivalentne, pa možemo koristiti klasu cifra. Dalje, nakon prepoznavanja niza slova, može se koristiti pregled tabele da se odredi da li je mogući identifikator zapravo rezervisana riječ. Time se rezervisane riječi i identifikatori se mogu zajedno prepoznati (umjesto dijela dijagrama za svaku rezervisanu riječ)

U sljedećem primjeru leksičkog analizatora definisaće se pet potprograma. Potprogram getChar uzima sljedeći znak i smiješta ga u varijablu nextChar, određuje mu klasu i smiješta klasu u varijablu charClass. Potprogram addChar smiješta znak iz nextChar na mjesto gdje se leksem čuva, niz znakova lexeme. Potprogram lookup određuje da li je niz znakova u lexeme rezervisana riječ i vraća kod ključne riječi. Potprogram getNonBlank uzima sljedeći znak, ignorišući prazne znakove. Potprogram lex implementira sljedeći dijagram stanja



Navedeni leksički analizator implementiran je sljedećim programom.

```
/* front.c - a lexical analyzer system for simple
arithmetic expressions */
#include <stdio.h>
#include <ctype.h>
/* Global declarations */
/* Variables */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen();
/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99
/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
/*****************************************************/
/* main driver */
main() {
/* Open the input data file and process its contents */
    if ((in_fp = fopen("front.in", "r")) == NULL)
     printf("ERROR - cannot open front.in \n");
    else {
     getChar();
        do {
       lex();
     } while (nextToken != EOF);
  }
}
/*****************************************************/
/* lookup - a function to lookup operators and parentheses and return the token */
int lookup(char ch) {
   switch (ch) {
   case '(':
     addChar();
```

```
    nextToken = LEFT_PAREN;
       break;
   case ')':
    addChar();
    nextToken = RIGHT_PAREN;
       break;
   case '+':
    addChar();
    nextToken = ADD_OP;
       break;
   case '-':
    addChar();
    nextToken = SUB_OP;
       break;
   case '*':
    addChar();
    nextToken = MULT_OP;
       break;
   case '/':
    addChar();
    nextToken = DIV_OP;
       break;
   default:
    addChar();
    nextToken = EOF;
       break;
 }
return nextToken;
}
/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar() {
   if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
 }
  else
    printf("Error - lexeme is too long \n");
}
/*****************************************************/
/* getChar - a function to get the next character of
input and determine its character class */
void getChar() {
   if ((nextChar = getc(in_fp)) != EOF) {
      if (isalpha(nextChar))
      charClass = LETTER;
       else if (isdigit(nextChar))
      charClass = DIGIT;
       else charClass = UNKNOWN;
   }
    else
```

```
    charClass = EOF;
}
/*****************************************************/
/* getNonBlank - a function to call getChar until it
returns a non-whitespace character */
void getNonBlank() {
  while (isspace(nextChar))
   getChar();
}/
*****************************************************/
/* lex - a simple lexical analyzer for arithmetic
expressions */
int lex() {
 lexLen = 0;
 getNonBlank();
   switch (charClass) {
/* Parse identifiers */
   case LETTER:
    addChar();
    getChar();
       while (charClass == LETTER || charClass == DIGIT) {
     addChar();
     getChar();
    }
    nextToken = IDENT;
       break;
/* Parse integer literals */
  case DIGIT:
   addChar();
   getChar();
     while (charClass == DIGIT) {
    addChar();
    getChar();
   }
   nextToken = INT_LIT;
     break;
/* Parentheses and operators */
 case UNKNOWN:
  lookup(nextChar);
  getChar();
     break;
/* EOF */
 case EOF:
  nextToken = EOF;
  lexeme[0] = 'E';
  lexeme[1] = 'O';
  lexeme[2] = 'F';
  lexeme[3] = 0;
     break;
 } /* End of switch */
 printf("Next token is: %d, Next lexeme is %s\n",
```

```
    nextToken, lexeme);
   return nextToken;
} /*
```

Primjer izvršenja: front.c kada se koristi nad

(sum + 47) / total

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Sintaksna analiza

Parseri kreiraju stablo parsiranja, potrebno za sintaksnu analizu. Parser u toku rada treba da obavi dva zadatka.

Prvi zadatak parsera je kreirati stablo parsiranja. To je je stablo koje predstavlja sintaksnu strukturu niza znakova (simbola) prema nekoj formalnoj gramatici. U nekim slučajevima, to stablo se samo implicitno kreira (ne pamti se), ali se ipak pokupe potrebne informacije.

Drugi zadatak parsera je naći sve sintaksne greške. Greške treba prepoznati u što većem broju. Nakon prepoznate greške, parser se treba oporaviti od greške, nastaviti sa prepoznavanjem. Loše realizovano oporavljanje od greške može dovesti do ogromnog broja pogrešno prepoznatih grešaka ili preskakanja velikog broja stvarnih grešaka.

Kategorije parsera

Dvije su osnovne vrste parsera, odozgo prema dolje (top-down) i odozdo prema gore (bottom-up).

Parseri Odozgo prema dolje prave stablo parsiranja, počevši od korijena stabla. Kada se neterminalni simboli transformišu u terminalne ili nove neterminalne simbole, uvijek se prvo zamjenjuje krajnji lijevi neterminalni simbol desne strane pravila (krajnje lijevo izvođenje).

Parseri odozdo prema gore prave stablo parsiranja počevši od listova i rezultuju redoslijedom koji je obrnut od krajnje desnog izvođenja.

Način definisanja gramatike jezika je vezan za izbor vrste parsera koji će se koristiti. Neke gramatike su povoljnije za parsiranje odozgo prema dolje, a neke za parsiranje odozdo prema gore. Pored ovoga, od načina predstavljanja gramatike zavisi i kompleksnost algoritma parsiranja. Parseri koji rade za svaku jednoznačnu gramatiku su kompleksni i neefikasni. Oni obavljaju posao u vremenu $O(n^3)$, gdje je n dužina ulaza. To je iz razloga što se takvi parseri moraju vraćati i ponovo analizirati rečenicu u mnogo slučajeva. No, određeni podskup jednoznačnih gramatika, može se parsirati u linearnom vremenu (O(n), gdje je n dužina ulaza). Stoga se za potrebe kompajlera gramatike uglavnom transformišu u oblik pogodan za parsiranje u linearnom vremenu.

Parseri odozgo na dolje

Neka su u gramatici terminalni simboli označeni malim slovima, neterminalni velikim slovima a miješani nizovi grčkim slovima. Neka su data sljedeća pravila

```
C→xAα
A →bB
A →cBb
A →a
```

Prema rečenici xAα, parser mora odabrati pravo A pravilo da dobije odgovarajuću rečenicu u krajnje desnom izvođenju koristeći samo prvi token koga je proizveo A . Stoga se simbol C se može transformisati u xbBα, xcBbα ili xaα. Ovdje je uočljiv problem donošenja odluke u parseru. Na parseru je da na bazi ulaznog niza znakova odabere

Parseri koji koriste rekurzivno spuštanje su kodirana implementacija BNF forme gramatike. Odluka koje će se pravilo primijeniti donosi se iz lijevog terminalnog simbola. U prethodnom primjeru sve tri definicije pravila A imaju različit lijevi terminalni simbol, pa je na bazi njega moguće izabrati pravo A pravilo.

Alternativa parserima s rekurzivnim spuštanjem su parseri odozgo nadolje upravljani tablicama, koje sadrže zakodirana stanja automata u koje se prelazi na bazi ulaznih simbola.

Obije vrste parsera odozgo nadolje spadaju u klasu LL. Prvo slovo L znači da se ulazni podaci čitaju s lijeva na desno, a drugo L da se pri primjeni gramatičkih pravila razvija krajnji lijevi neterminalni simbol.

Parsiranje rekurzivnim spuštanjem

Parseri rekurzivnim spuštanjem se realizuju tako što postoji potprogram za svaki neterminalni simbol u gramatici. Taj potprogram dalje analizira rečenice koje se mogu generisati tim neterminalnim simbolom.

Za ovu vrstu parsera, pogodnije je gramatiku pisati u EBNF. EBNF minimizira broj neterminalnih simbola i ima proširenja u formi ugaonih zagrada (opcionalni izraz) i vitičasti h zagrada (ponavljajući izraz) koji se mogu lako realizovati naredbama uslova i petlje.

U sljedećem primjeru, neka je data gramatika za prosti aritmetički izraz:

```
<expr> →<term> {(+ | -) <term>}
<term> →<factor> {(* | /) <factor>}
<factor> →id | int_constant | ( <expr> )
```

Pretpostavimo da imamo leksički analizator lex, koji smiješta kod sljedećeg tokena u varijabli nextToken.

Kada postoji samo jedna desna strana u sintaksnom pravilu, programski kod se generiše na sljedeći način:

Za svaki terminalni simbpl na desnoj strani poredi ga s narednim ulaznim tokenom, ako se slažu nastavi, inače je greška.

Za svaki neterminalni simbol na desnoj strani, pozovi pridruženi potprogram parsiranja

U ovom primjeru gramatike expr ima samo jednu desnu stranu, pa se kod može generisati na sljedeći način.

```
/* Function expr
Parses strings in the language generated by the rule:
<expr> -><term> {(+ | -) <term>}
*/
void expr() {
/* Parse the first term */
  term();
/* As long as the next token is + or -, call
lex to get the next token and parse the
next term */
  while (nextToken == ADD_OP || nextToken == SUB_OP){
    lex();
    term();
  }
}
```

Ova rutina ne prepoznaje greške.

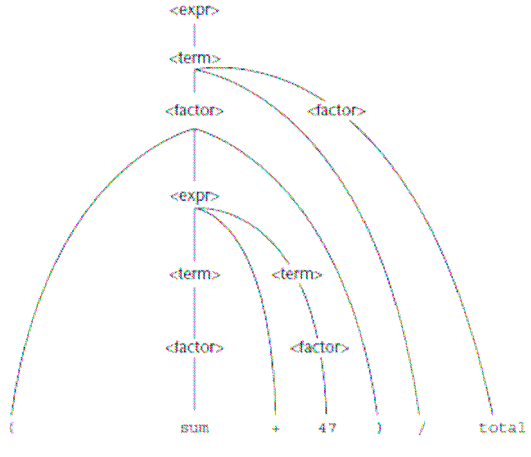
Neterminalni simbol koji ima više od jedne desne strane zahtijeva inicijalni proces da odredi koju desnu stranu parsirati. Odgovarajuća desna strana se odabira na bazi sljedećeg ulaznog tokena. Naredni token se predi s prvim tokenom koji se može generisati svakom desnom stranom dok se ne uoči podudaranje. Ako nema podudaranja, to je sintaksna greška

Nastavak primjera parsera rekurzivnim spuštanjem ovo ilustruje kroz rutinu factor.

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {
  printf("Enter <term>\n");
  /* Parse the first factor */
  factor();
/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
} /* End of function term */
/* Function factor
Parses strings in the language
generated by the rule:
<factor> -> id | (<expr>) */
void factor() {
/* Determine which RHS */
   if (nextToken) == ID_CODE || nextToken == INT_CODE)
/* For the RHS id, just call lex */
     lex();
/* If the RHS is (<expr>) – call lex to pass over the left parenthesis, call expr, and check
for the right parenthesis */
    else if (nextToken == LP_CODE) {
      lex();
      expr();
      if (nextToken == RP_CODE)
        lex();
      else
         error();
    } /* End of else if (nextToken == ... */
    else error(); /* Neither RHS matches */
}
```

Sljedeći primjer prikazuje izvršenje parsiranja nad izrazom

(sum + 47) / total



To rezultuje sljedećim stablom parsiranja

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is total
Enter <expr> Enter <factor>
Enter <term> Next token is: -1 Next lexeme is EOF
Enter <factor> Exit <factor>
Next token is: 11 Next lexeme is sum Exit <term>
Enter <expr> Exit <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
```

Problem lijeve rekurzije:

Za parsiranje parserima odozgo nadolje (LL parserima) gramatika treba biti pripremljena. Najveći problem predstavlja lijeva rekurzija, ilustrovan na sljedećem primjeru.

Neka je dato sljedeće pravilo

$A \rightarrow A+B$

Direktna transformacija ovakvog pravila bi dovela do sintaksne procedure koja poziva samu sebe na svom početku bezuslovno i neograničeno dok se stek ne popuni.

Ako gramatika ima lijevu rekurziju, direktnu ili indirektnu, ona ne može biti osnova parsera odozgo prema dolje. Ipak, gramatika se može izmijeniti da se ukloni lijeva rekurzija sljedećim algoritmom.

1. Za svaki neterminalni simbol, A,

Grupiši A pravila kao

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

gdje ni jedan βs ne počinje s A

2. Zamijeni originalna A-pravila s

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Test razdvojenosti po parovima:

Pored lijeve rekurzije, drugi problem koji limitira parsiranje odozgo nadolje je pitanje da li se može odrediti pravilo izvođenja na bazi pogleda u jedan token.

Def: FIRST(α) = {a | α =>* a β }

(If α =>* ε, ε is in FIRST(α))

gdje =>* znači 0 ili više koraka izvođenja

Za svaki neterminalni simbol, A, u gramatici koja ima više od jedne desne strane, za svaki par pravila, $A \rightarrow \alpha_i$ i $A \rightarrow \alpha_j$, mora važiti

FIRST (α_i) □FIRST (α_j) = □

Primjeri:

A →a | bAB | Bb

B →cB | d

FIRST skupovi za A pravila su {a},{b} i {c,d} koji su međusobno diskjunktni. Stoga, ova gramatika prolazi test razdvojenosti po parovima.

U sljedećem primjeru FIRST skupovi za <variable> pravila (varijabla i element niza) su {identifier} i {identifier} koji očito nisu disjunktni.

```
<variable> →identifier | identifier [<expression>]
```

Razdvajanjem se može riješiti problem. Zamijeni

```
<variable> →identifier | identifier [<expression>]
```

sa

```
<variable> →identifier <new>
<new> →€| [<expression>]
```

ili

```
<variable> →identifier [[<expression>]]
```

(vanjske zagrade su metasimboli EBNF)

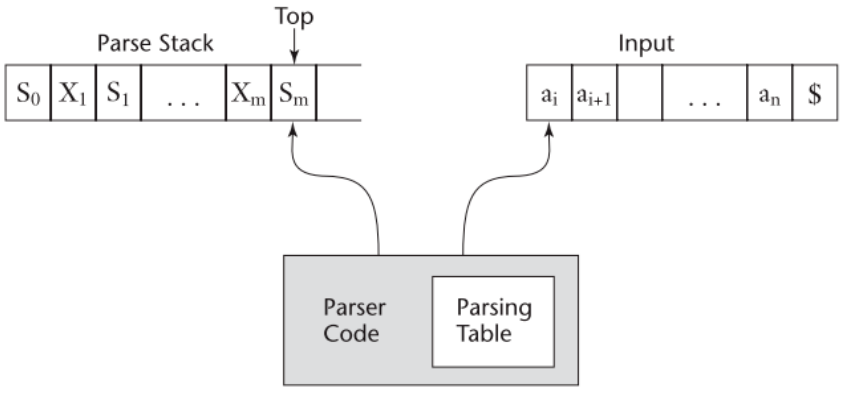
Parseri odozdo prema gore

Najčešći algoritmi parsiranja u odozdo prema gore kategoriji su u LR familiji . Prednosti su im što

- Rade s svakom gramatikom koja opisuje programske jezike.
- Rade nad širom klasom algoritama od drugih algoritama odozfo prema gore, a jednako su efikasni
- Mogu detektovati sintaksnu grešku što prije moguće
- LR klasa gramatika je nadskup klase koja se analizira LL parserima.

Mane algoritma su što je tablicu teško ručno konstruisati, pa program parser nije lako razumljiv. Zato se LR parseri obično konstruišu softverskim alatom.

Algoritam je objavio Donald Knuth. Parser prema gore treba koristiti cijelu historiju parsiranja i gledati stablo do trenutne tačke u cilju pravljenja odluka. Ali kako postoji konačan i relativno mali broj različitih situacija u parsiranju, historija se smješta u



stanje parsera na steku parsiranja.

Struktura LR parsera data na slici, sastoji se od steka parsiranja, ulaza , tabele parsiranja i samog programskog koda. Neka su S – stanja, X – simboli odluke a a ulazni simboli. Na steku parsiranja se nalaze vrijednosti $S_0X_1S_1X_2S_2...X_mS_m$ gdje se Sm nalazi na vrhu steka. Ulazni tekst je u obliku $a_ia_{i+1}...a_n\$$ gdje \$ predstavlja EOF simbol. Prema tome LR konfiguracija čuva stanje za LR parser

$(S_0X_1S_1X_2S_2...X_mS_m, a_ia_{i+1}...a_n\$)$

LR parseri se vode tablicama gdje tablica ima dvije komponente: tablicu ACTION i tablicu GOTO .Tablica ACTION navodi akciju parsera prema njegovom stanju i narednom tokenu. Redovi ove tablece su imena stanja, a kolone su terminalni simboli koji se mogu javiti na ulazu. Tablica GOTO navodi koje stanje staviti na vrh steka parsiranja nakon akcije redukcije. Redovi ove tabele su imena stanja, kolone su neterminalni simboli

Algoritam u tablici ACTION poznaje četiri vrste akcija: Shift, Reduce, Accept i Error, pa se zbog toga zove Shift-Reduce algoritam.

Algoritam radi na sljedeći način.

Neka je data početna konfiguracija: $(S_0, a_1...a_n\$)$

Ponavljaj akcije parsera:

- Ako je ACTION$[S_m, a_i]$ = Shift S, sljedeća konfiguracija je:

$(S_0X_1S_1X_2S_2...X_mS_ma_iS, a_{i+1}...a_n\$)$

Sljedeći ulazni simbol se stavi na stek zajedno s stanjem dati uz parametar Shift

- Ako ACTION$[S_m, a_i]$ = Reduce $A \rightarrow \beta$ i S = GOTO$[S_{m-r}, A]$, gdje r = dužina β; sljedeća konfiguracija je

$(S_0X_1S_1X_2S_2...X_{m-r}S_{m-r}AS, a_ia_{i+1}...a_n\$)$

Za akciju Reduce, držač mora biti uklonjen iz steka.Pošto za svaki gramatički simbol na steku postoji simbol stanja, broj simbola uklonjenih iz steka je duplo veći od broja simbola uz držač. Nakon uklanjanja držača i njegovih povezanih simbola stanja, lijeva strana pravila se stavlja na stek. Na kraju se koristi GOTO tabela , s oznakom reda koja je simbol koji je bio izložen kada se držač i njegovi simboli uklone sa steka, i oznaka kolone koja je neterminalni simbol kao lijeva strana pravila koja se koristi u redukovanju.

- Ako ACTION$[S_m, a_i]$ = Accept, parsiranje je završeno bez grešaka.
- Ako ACTION$[S_m, a_i]$ = Error, parser poziva rutinu za obradu grešaka.

Primjer: Neka je dat skup pravila jezika.

1) E →E + T
2) E →T
3) T →T * F
4) T →F
5) F →(E)
6) F →id

Tablica parsiranja za ovu gramatiku dat je u sljedećoj tabeli.

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Tabela parsiranja se generiše odgovarajućim alatom iz gramatike, npr., yacc

Primjer izvođenja za id + id * id dat je u sljedećoj tabeli.

Stack	Input	Action
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

S obzirom da je na vrhu steka 0, a na ulazu id, iz ACTION[0,id] vidi se da prva akcija je Shift 5. Sa ulaza se izbacuje simbol id, stavi na stek a iza njega se stavi parametar akcije Shift, koji je 5. Pošto je na vrhu steka 5, a sljedeći ulazni simbol +, ACTION[5,+] kaže da je druga akcija Reduce 6. Pravilo broj 6 je F →id, a njegova lijeva strana je F. Desna strana mu ima jedan simbol, pa to znači uklanjanje 2 elementa steka. Sa steka se uklone id i 5. Na vrhu steka je 0. Stoga se gleda sadržaj GOTO[0,F]. Tu je 3. Na stek se stavlja F i 3. Treća akcija je Reduce 4. Pravilo broj 4 je T →F, a njegova lijeva strana je T, desna strana ima 1 simbol. Sa steka se uklone 3 i F.. Na vrhu steka je nula. Stoga se gleda sadržaj GOTO[0,T]. On je 2, pa se na stek stavljaju T i 2. Prije dvanaeste akcije na steku je 0E1+6T9*7F10. Dvanaesta akcija je Reduce 3. Pravilo broj 3 je T →T * F. Lijeva strana mu je T, desna ima 3 simbola. Stoga se sa steka uklanja šest elementa, pa je na vrhu steka 6. Zato se koristi GOTO[6,T]. Tu je 9, pa na stek idu T i 9. Zadnja akcija, u stanju 1 ima simbol kraja na ulazu pa je akcija Accept.

Generatori leksičkih i semantičkih analizatora

Proces pisanja leksičkih i semantičkih analizatora se može automatizovati. Postoje programi koji na bazi BNF ili EBNF sa atributskom gramatikom generišu leksički i sintaksni analizator u nekom od programskih jezika (najčešće C, C++, Java ili Pascal). Sintaksni analizatori su obično LL ili LR tipa.

Lex/flex i yacc/bison

Lex i yacc su najpoznatiji programi ove vrste. Povećanjem popularnosti softvera otvorenog koda uglavnom su zamijenjeni kompatibilnim programima flex i bison, u daljnjem tekstu će se koristiti imena lex i yacc. Ova dva programa rade u paru, lex generiše leksički analizator dok yacc generiše sintaksni analizator kao LR parser.

Lex dokument opisuje lekseme kroz regularne izraze. Dio koda koji se ugrađuje u generisani C program se piše unutar simbola %{ i %} dok se unutar %% simbola navode regularni izrazi koji rezultuju odgovarajućim tokenima. Pored regularnog izraza piše se odgovarajući C kod koji generiše potrebne varijable. Leksem se nalazi u varijabli yytext, ali je za numeričke vrijednosti potrebno generisati i varijablu yylval. Kodovi koji predstavljaju tokene definisani su datoteci y.tab.h koju generiše yacc. Slijedi primjer lex datoteke (primjer.l) za jezik koji poznaje tokene cijelog broja i ključne riječi heat, on, off, target i temperature.

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+       yylval=atoi(yytext); return NUMBER;
heat         return TOKHEAT;
on|off       yylval=!strcmp(yytext,"on"); return STATE;
target       return TOKTARGET;
temperature  return TOKTEMPERATURE;
\n           /* ignore end of line */;
[ \t]+       /* ignore whitespace */;
%%
```

Yacc će dalje generisati sintaksni analizator. Ulazni format za yacc se sastoji od zaglavlja koje se ugrađuje u generisani C kod, liste kodova za tokene i BNF zapisa sintakse jezika. Atributska gramatika je pretstavljena kodom koji se piše u C unutar vitičastih zagrada. Navedeni C kod može pristupati generisanoj yylval varijabli kroz simbole $1, $2, ... koji predstavljaju redni broj odgovarajućeg neterminalnog simbola s desne strane. Yacc datoteka za navedeni jezik (primjer.y) izgleda ovako:

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{
	fprintf(stderr,"error: %s\n",str);
}
int yywrap()
{
	return 1;
}
main()
{
	yyparse();
}
%}
%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE
%%
commands: /* empty */
	| commands command
	;
command:
	heat_switch
	|
	target_set
	;
heat_switch:
	TOKHEAT STATE
	{
		if($2)
			printf("\tHeat turned on\n");
		else
			printf("\tHeat turned off\n");
	}
	;
target_set:
	TOKTARGET TOKTEMPERATURE NUMBER
	{
		printf("\tTemperature set to %d\n",$3);
	}
	;
%%
```

Lex i yacc datoteka se dalje kompajlira koristeći komande

```
lex primjer.l
yacc -d primjer.y
cc lex.yy.c y.tab.c -o comp
```

COCO/R

Coco/R je generator LL parsera koji podržava veći broj odredišnih jezika. Generisani kod je daleko čitljiviji i rad s ovim generatorom parsera je jednostavniji nego u slučaju Lex/Bison kombinacije, ali zahtijeva da se gramatika pretrodno dovede u LL(1) oblik. Kod ovog generatora kompajlera leksički, sintaksni i opcionalno semantički analizator se pripremaju u istoj datoteci. Ugrađeni kod kao atributska gramatika se upisuje unutar simbola (. .). Ključne riječi se navode unutar znakova navodnika. Naslijeđeni atributi se definišu u oštrim zagradama, dok se trenutno obrađivanom leksemu pristupa kroz objekt ili strukturu koji se zove “t”. Sljedeći primjer u Coco/R predviđen za Java verziju generiše interpreter za jezik koji se sastoji od naredbe CALC praćene aritmetičkim izrazom s četiri operacije, prioritetima i zagradama.

```
COMPILER Calc
CHARACTERS
  digit = '0' .. '9'.
TOKENS
  number = digit {digit}.
COMMENTSFROM "/*" TO "*/" NESTED
IGNORE '\t' + '\r' + '\n'
PRODUCTIONS
  Calc  (. int x; .)
    = "CALC" Expr <out x> (. System.out.println(x);.) .
  Expr <out int x> (. int y; .)
   = Term <out x>
   { '+' Term <out y> (. x = x + y; .)
   }.
  Term <out int x> (. int y; .)
   =Factor <out x>
    { '*' Factor <out y> (. x = x * y; .)
    }.
  Factor <out int x>
    = number (. x = Integer.parseInt(t.val); .)
    | '(' Expr <out x> ')'.
END Calc.
```

ANTLR

ANTLR je sistem koji generiše sintaksne i leksičke analizatore u Java ili C#. Podržava veoma široku klasu gramatika. U verziji 4 razdvaja sintaksnu od semantičke analize, tako što za ulaz koristi EBNF format bez ugrađenog koda u odredišnom jeziku, dok se semantički obrađivači ugrađuju u izdvojenu Java ili C# klasu.

Rezime

Sintaksna analiza je standardni dio implementacije jezika. Podjeljena je u dva dijela, leksičku analizu (skaniranje) i sintaksnu analizu (parsiranje). Leksički analizator usaglašava uzorke koji izoliraju male dijelove programa. Ručno se obično programira parser rekurzivnim spuštanjem, poznat i kao LL parser. EBNF je pogodna forma gramatike koja će se generisati LL parsiranjem. Problem parsiranja odozdo prema gore je traženje podniza trenutnih rečeničkih oblika. LR familija shift-reduce parsera je načešći pristup za parsiranje odozdo prema gore

5. Imena, vezanja i opsezi

Imperativni jezici su nastali kao posljedica apstrakcije Von Neumannove arhitekture. U ovoj arhitekturi jedan procesor izvršava program u memoriji koji pristupa podacima u toj istoj memoriji. Memorijske ćelije su u jezicima visokog nivoa predstavljene varijablama.

U čisto funkcionalnim jezicima varijable ne postoje. Kod funkcionalnih programskih jezika, izrazi se mogu imenovati. Imenovani izrazi vizuelno izgledaju kao dodjela varijable, ali se njihov sadržaj ne može mijenjati, pa više liče na konstante.

Varijable su okarakterisane atributima, tj. njenim tipom podataka, imenom, opsegom, vremenom života.

Imena

Pri izučavanju imperativnog programskog jezika treba obratiti pažnju na njegove specifičnosti u imenovanju varijabli. U većini jezika imena varijabli se sastoji od slova, podvlačilica i cifara, ali to nije uvijek slučaj. Čak i kada jeste, treba provjeriti da li imena zavise od veličine slova i da li postoje specijalne rezervisane riječi koje ne mogu biti varijable.

Dužina imena predstavlja ograničenje na maksimalan broj znakova koje varijabla može da ima. Ako je ono prekratko, ne mogu se opisati neki složeniji podaci.

Primjeri ograničenja jezika:

- Level 1 BASIC jedno slovo,
- FORTRAN IV: šest slova, FORTRAN 95: maksimum od 31
- C99: bez ograničenja ali samo prvih 63 su značajna; također spoljnja imena su ograničena na 31
- C#, Ada, i Java: bez ograničenja i sva su značajna
- C++: bez ograničenja, ali može implementacija uticati

Specijalni znakovi u imenu (pored slova i cifara) nekada su samo dopušteni, a nekada imaju i posebno značenje. Sljedeći primjeri to ilustruju

- PHP: sva imena varijabli moraju početi znakom dolara
- Perl: sva imena varijabli počinju specijalnim znakom koji određuje tip varijable $ (skalarne varijable), @ (nizovi), %(asocijativni nizovi)
- Ruby: imena varijabli koja počinju s @ su varijable instance; one koje počinju s @@ su varijable klase
- Microsoft BASIC, varijable čije ime se završava znakom $ su alfanumeričke varijable
- Forth dopušta sve ASCII znakove u imenu varijable, ako to ne dolazi u konflikt s numeričkim konstantama i već definisanim riječima i varijablama

Razlikovanje velikih i malih slova (case sensitive) znači da li su varijable Nastavnik, nastavnik i NASTAVNIK tri iste ili tri različite varijable. Radi brže leksičke analize, u C-baziranim jezicima razlikuju se velika i mala slova. U Fortranu i jezicima baziranim na Algol-u velika i mala slova se ne razlikuju. Razlikovanje velikih i malih slova smanjuje čitljivost, jer dvije slične varijable su zapravo različite. Iz tog razloga C programeri ponekad sve varijable pišu malim slovima, a definisane tipove velikim slovima. No u

C++, Java, i C# dešava se da predefinisana imena miješaju velika i mala slova (npr. IndexOutOfBoundsException)

Specijalne riječi su riječi koje imaju posebno značenje u sintaksi programskog jezika. One pomažu u čitljivosti i koriste se za razdvajanje dijelova naredbi. One se dijele na rezervisane riječi i ključne riječi. Iako u oba slučaja nije preporučljivo da varijabla ima ime kao specijalna riječ, razlika je u tome što se rezervisana riječ ne može uopšte koristiti kao korisnički definisano ime, dok ključna riječ u nekom kontekstu je specijalna, a u nekom može biti varijabla. Npr. u Fortranu riječ Real se može koristiti u dva konteksta.

Real VarName (Real je tip podataka pa je Real ključna riječ)

Real = 3.4 (Real je varijabla)

Potencijalni problem sa rezervisanim riječima: Ako ih je previše mogu se desiti konflikti (npr., COBOL ima 300 rezervisanih riječi, među njima vrlo česta imena za varijable LENGTH i COUNT!)

Skupovi ključnih i rezervisanih riječi nisu disjunktni. U jezicima poput C i Pascal sve ključne riječi su ujedno i rezervisane. U PL/1 nijedna ključna riječ nije rezervisana. U JavaScript postoje rezervisane riječi koje se uopšte ne koriste kao ključne riječi.

Varijable

Varijabla programa je apstrakcija računara memorijske ćelije ili skupa ćelija. Prelaskom iz mašinskog jezika na asemblerske jezike se umjesto apsolutne numeričke memorijske adrese uvodi imenovanje podataka, što program čini daleko čitljivijim a samim tim i lakši za pisanje i održavanje. Programeri često misle o imenu varijabile kao imenu za memorijske lokacije, ali postoji mnogo više podataka o varijabli nego samo ime. Varijabla se može okarakterisati kao šestorka atributa: (ime, adresa, vrijednost, tip, vrijeme života i opseg).

Imena varijabli su najčešća imena u programima. Ime je niz znakova koje koristi za identifikaciju nekog entiteta u programu.

Adresa varijable je mašinska adresa memorije sa kojom je u vezi. Ovo povezivanje nije tako jednostavno kao što se može na prvi pogled izgledati. U mnogim jezicima, je moguće da ista varijabla bude povezana s različitim adresama u različitim periodima u programu. Na primjer, ako potprogram ima lokalnu varijabla koja se dodjeljuje u run-time stack kada se zove potprogram se zove, različiti pozivi mogu rezultirati da varijabla ima različite adrese. Adresa varijabla se ponekad naziva L-vrijednost, jer adresa je ono što je potrebno kada je ime varijable pojavljuje se u lijevom dijelu dodjeljivanja.

Moguće je imati više varijabli koje imaju istu adresu. Kada se više od jednog imena varijable može koristiti za pristup istoj memorijskoj lokaciji, varijable se nazivaju aliasi. Aliasi se mogu kreirati u mnogim jezicima putem parametara potprograma, pointera ili unija.

Tip varijable određuje raspon vrijednosti varijable koje ona može imati i skup operacija koje su definirane za vrijednosti tipa. Na primjer, tipa int u Javi određuje opseg vrijednosti od -2147483648 do 2147483647 i aritmetičke operacije za sabiranje, oduzimanje, množenje, dijeljenje, i modul.

Vrijednost varijable je sadržaj memorijske jedne ili više ćelija u vezi sa varijablom. Zgodno je razmišljati o memorije računala u smislu apstraktne ćelija, a ne fizičke

ćelije. Fizičke ćelije, ili pojedinačne adresabilne jedinice, kod većine savremenih računara su bajt-veličine, gdje bajt obično bude osam bitova. Ova veličina je previše mala za većinu programskih varijabli. Apstraktna memorijska ćelija ima veličinu koju zahtijeva varijablu s kojima je povezana. Na primjer, iako vrijednosti floating-point može zauzeti četiri fizička bajta u određenoj implementaciji na određenom jeziku, a vrijednost floating-point se uzima kao da zauzima jedan apstraktni memorijska ćelija. Vrijednost svake jednostavne nonstructured tip se smatra da zauzme jednu apstraktnu ćelije. Od sada, izraz memorijska ćelija znači apstraktno memorijska ćelija. Vrijednost varijable se zove i r-vrijednost.

Vrijeme života varijable je vrijeme tokom koga je ona vezana za odgovarajuću memorijsku ćeliju.

Povezivanje

Povezivanje je pridruživanje između atributa i entiteta ili operacije i simbola. Vrijeme povezivanja je vrijeme kada se ono obavlja. Na primjer ime varijable je poznato u trenutku pisanja programa, ali pripadna adresa će biti poznata tek u trenutku kompajliranja ili izvršavanja. Evo nekih primjera za moguća vremena povezivanja

- Vrijeme definicije jezika - povezivanje simbola operatora sa operacijama, npr određivanje da * predstavlja množenje.
- Vrijeme implementacije jezika-- povezivanje tipa pokretnog zareza s predstavljanjem na datom računaru (npr. big endian ili little endian redanje IEEE 754 bajtova)
- Vrijeme kompajliranja – povezivanje varijable s tipom u C ili Java
- Vrijeme učitavanja – povezivanje C ili C++ static varijable sa memorijskom ćelijom
- Vrijeme izvršenja – povezivanje nestatičke varijable s memorijskom ćelijom

Pri povezivanju tipova sa imenom varijable otvaraju se dva pitanja: kako se tip navodi i kada se dešava povezivanje?

Statičko povezivanje

Povezivanje je **statičko** ako se prvo dešava prije izvršenja i ostaje neizmijenjeno dok program traje. Statičko povezivanje tipova je Tip se može navesti eksplicitnom ili implicitnom deklaracijom. Eksplicitna deklaracija je naredba programa koja se koristi za tipove varijabli. Npr u jeziku C koristi se obavezna deklaracija varijabli navodeći ime tipa iza koga slijedi lista varijabli

```
int a,b,c
```

Implicitna deklaracija koristi podrazumijevani mehanizam za navođenje tipova varijabli (prva pojava varijable u programu). Primjeri implicitnih deklaracija

Fortran: nedeklarisane varijable čije ime počinje slovima I,J,K,L i M su automatski cjelobrojne, dok su ostale realne

Microsoft BASIC: Varijable čija imena završavaju s $ su znakovne varijable, varijable čija imena završavaju s % su cijeli brojevi, nizovi koji nisu deklarisani DIM naredbom imaju 10 elemenata

Perl: Varijable koje počinju znakom $ su skalarne varijable, one koje počinju znakom @ su nizovi.

U C# postoji varijanta između implicitnog i eksplicitnog određivanja tipova: zaključivanja tipa. Sljedeća tri primjera deklarišu i inicijaliziraju varijable na bazi početno dodijeljene vrijednosti.

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

Zaključivanje tipa je i u funkcionalnim jezicima (ML, Miranda i Haskell)

Prednost implicitne deklaracije je upisivost (ne mora se deklarisati varijabla), mana je lošija pouzdanost (ako varijabla nije deklarisana greška u kucanju imena varijable dovodi do programa koji se može prevesti ali pogrešno radi)

Dinamičko povezivanje tipova

Povezivanje je **dinamičko** ako se dešava u trenutku izvršenja ili se može mijenjati u toku izvršenja. Kada je tip varijable je statički vezan, ime varijable se smatra vezanim uz tip, u smislu da su tip i ime varijabla se istovremeno vezani. Međutim, kada je tip varijable je dinamički vezan, ime se može posmatrati kao samo privremeno vezano za tip. Osnovna prednost dinamičkog vezivanja varijabli s tipom je u tome što pruža više fleksibilnosti. Na primjer, program za obradu numeričkih podataka na jeziku koji koristi dinamički tip vezanje se može pisati kao generički program, što znači da da je u stanju da rukuje podacima bilo kojeg numeričkog tipa.

Jezici koji koriste dinamičko vezivanje tipova su Lisp, JavaScript, PHP, XBASE . U sljedećem primjeru u JavaScript prva naredba dodjeljuje tip niza varijabli list, a druga joj dodjeljuje broj u pokretnom zarezu.

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

U Ruby su sve varijable reference na objekt i tako mogu pristupati bilo kom objektu.

Mana dinamičkog povezivanja je visoka cijena zbog potrebne provjere tipova u toku izvršenja i teško uočavanje grešaka u kucanju imena varijable. Stoga u nekim jezicima koji su primarno statičkog povezivanja tipova uvedena je mogućnost deklarisanja varijavli s dinamičkim povezivanjem ako je ono potrebno.

- U C# 2010 uvedena je ključna riječ dynamic koja deklariše varijable kojima se može dodijeliti vrijednost bilo kog tipa

```
dynamic any;
```

- Visual BASIC ima tip podataka variant

```
Dim MyVar As Variant
```

- Isti tip je uvedeni u Delphi

```
Var a : variant;
```

Vezivanje memorije i dužina života

U imperativnim programskim jezicima, jedan od najvažnijih zadataka je dodjeljivanje vrijednosti i račun s varijablama. Memorijska ćelija za koju je varijabla vezana mora na neki način da bude dovedena iz skupa slobodne memorije. Taj proces se zove alokacija. Dealokacija je suprotan proces, vraćanje memorijske ćelije do sada vezane uz varijablu u skup dostupnih memorijskih ćelija.

Vrijeme života varijable je vrijeme tokom koga je ona vezana za odgovarajuću memorijsku ćeliju. Stoga, životno vrijeme varijable počinje kada se ona poveže s odgovarajućom memorijskom ćelijom i završava kada se oslobodi od te ćelije

Kategorije varijabli po vremenu života

Statičke varijable su vezane za memorijske ćelije prije početka izvršenja i ostaju vezane za nju tokom izvršenja. U ovu kategoriju spadaju globalne varijable u Pascal, C, Fortran, ... kao i lokalne varijable u C i C++ koje su označene ključnom rječju **static** varijable i lokalne varijable u Fortran 95 koje su označene naredbom Save. Prednost ovakvog povezivanja je u efikasnosti. Koristi se direktno adresiranje, tako da je pristup memorijskoj lokaciji obično postignut jednom mašinskom instrukcijom. Pored ovoga, ako potprogram treba da pamti ranije stanje varijabli nakon što je napušten pa ponovo pozvan, statičke varijable omogućavaju pamćenje ranije istorije. Mana je u manjoj fleksibilnosti (nema rekurzije)

Stack-dinamičke varijable se vezuju s memorijom se kada se obrađuju njihove naredbe deklarisanja. To se dešava pri izvršenju programa, na primjer prilikom početka izvršavanja potprograma u kome je varijabla deklarisana. Ostali atributi za skalarne varijable se povezuju statički. Primjer ovakvih varijabli su lokalne varijable u C potprogramima i Java metodama. Prednost stek dinamičkih varijabli je što dopuštaju rekurziju, a zauzeti prostor se oslobađa kada nije potreban. Mana je u potrebnom dodatnom kodu za alokaciju i dealokaciju (mada je on relativno brz), što potprogrami ne čuvaju istoriju i što na nekim platformama pristup varijablama je sporiji zbog indirektnog adresiranja.

Eksplicitno heap-dinamičke varijables su neimenovane (apstraktne) memorijske ćelije koje se alociraju i dealociraju eksplicitnim naredbama koje se izvršavaju u toku rada programa, a piše ih programer. Heap je skup memorijskih lokacija čija je organizacija dosta neuređena zbog nepredvidljivosti upotrebe. U C++, operator alokacije, koji se zove new, koristi ime tipa kao svoj operand. Kada se izvrši, kreira se eksplicina heap-dinmamička varijabla čiji je tip naveden kao operand i vrati se adresa. Sljedeći dio koda u C++ alocira heap varijable:

```
int *intnode;      // Kreiraj pointer
intnode = new int; // Kreiraja heap-dynamic varijablu
. . .
delete intnode; // Dealociraj varijablu na koju intnode pokazuje
```

Java sve podatke osim primitivnih skalara vidi kao objekte. Svi Java objekti su eksplicitno heap dinamički, ali se dealociranje ne vrši eksplicitno nego se koristi implicitno skupljanje smeća. C# ima i eksplicitnhe heap dinamičke i stek dinamičke objekte koji su implicitno dealocirani.

Implicitno heap-dinamičke varijable se alokaciraju i dealociraju naredbama dodjele. Takve su sve varijable u APL; svi stringovi i nizovi u Perl, JavaScript, i PHP , varijable u mnogim implementacijama BASIC-a itd. Sljedeći primjer u JavaScript

```
highs = [74, 84, 86, 90, 71];
```

Pez obzira da li je varijabla highs bila korištena u programu i za šta je bila korištena, ona je sada niz od pet numeričkih vrijednosti. Prednost ovakvog načina alokacije je fleksibilnost jer je moguće realizovati generički kod, npr program za sortiranje niza bez obzira na njegove dimenzije i tip podataka sadržaja. Mane su mala efikasnost jer se svi atributi varijable postavljaju pri svakoj dodjeli i gubitak detekcije grešaka pri kompajliranju.

Opseg varijabli

Opseg varijable (scope) je skup naredbi u kojoj je ona vidljiva. Varijabla je **vidljiva** u naredbi ako joj se može pristupati iz te naredbe. Pravila opsega jezika određuju kako se odgovarajuće pojavljivanje imena povezuju sa varijablom, odnosno kod funkcionalnih jezika kako se ime povezuje s izrazom.

Varijabla je **lokalna** u programskoj jedinici ili bloku ako je tu deklarisana. **Nelokalne** varijable jedinice programa su one koje su vidljive ali nisu deklarisane tu. Globalne varijable su specijalni slučaj nelokalnih varijabli.

Statički opseg

ALGOL 60 uveo metodu vezivanja imena nelokalnih varijabli zvanu statički opseg, koja je kopirana od strane mnogih kasnijih imperativnih jezika i još mnogih neimperativnih jezika. Statički opseg je tako nazvan jer opseg varijable može biti statički određen, to jest, prije izvršenja. To omogućuje programerima i kompajlerima određivanje vrste svake varijable u programu jednostavno ispitujući njegov izvorni kod.

Postoje dvije kategorije statičkih opsegom jezika: on u kojima potprogrami mogu biti ugniježdeni, što stvara ugniježden statički opseg, i one u kojima potprogrami ne mogu biti ugniježdeni. U drugoj kategorije, statički opseg također su stvorili potprogrami, ali ugniježdeni opseg su stvorili samo ugniježdene definicijama blokova i klasa.

Ada, Pascal, JavaScript, Common Lisp, Scheme, Fortran 2003+, F # i Python dopuštaju ugniježdene potprograme, ali jezici nastali od C ne.

Statički roditelj potprograma subl, njegovi statički roditelji, i tako dalje do (uključujući i njega) najvećeg obuhvaćajućeg potprograma, nazivaju se statički preci subl. Šta reći o sljedećoj JavaScript funkciji, big, u kojoj su dvije funkcije subl i sub2 ugniježdene:

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

Isti primjer u Pascal-u:

```
program staticki;
var x:integer;
  Procedure sub1;
  var x:integer;
  begin
    x:=7 ;
    sub2;
  end;
  procedure sub2;
  var y:integer;
  begin
    y:=x;
  end;
begin
  x:=3;
  sub1;
end.
```

U gornjem primjeru, u funkciji subl, varijabla x je iz funkcije subl, dok je funkciji sub2, varijabla x je iz funkcije big. Primijetiti da je u subl varijabla x iz big skrivena. Varijable se mogu sakriti iz jedinice tako da imaju "bliže" varijable sa istim imenom

Ada dopušta pristup takvim "skrivenim" varijablama koristeći sintaksu big.x

Blokovi

Mnogi jezici dopuštaju da se novi statički opsezi definišu usred izvršnog koda. Ovaj moćni koncept uveden u ALGOL 60 dopušta da sekcija koda ima svoje lokalne varijable čiji je opseg minimiziran. Takve varijable su obično stek dinamičke, pa im se memorija alocira i dealocira kada se izađe iz sekcije. Takva sekcija se zove blok. Npr. u C baziranim jezicima može se pisati

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

U sljedećoj C nedovršenoj funkciji varijabla count u while petlji je skrivena od ostatka koda u sub funkciji:

```
void sub() {
  int count;
  . . .
  while (. . .) {
   int count;
   count++;
   . . .
  }
  . . .
}
```

Prethodni primjer je ispravan u C i C++ ali ne u Java i C# jer su dizajneri ta dva jezika smatrali da je previše potencijalnih uzroka grešaka da bi trebalo dozvoliti da se isto ime varijable koristi u funkciji i bloku.

U funkcionalnim jezicima postoji konstrukcija slična statičkom opsegu blokova imperativnih jezika. Varijable definisane u tim blokovima se mogu postaviti ali ne i promijeniti naknadno. U Scheme funkcija LET ima sljedeći oblik

```
(LET (
(name1 expression1)
. . .
(namen expressionn))
expression
)
```

Sljedeći primjer računa (a + b) / (c – d)

```
(LET (
(top (+ a b))
(bottom (- c d)))
(/ top bottom)
)
```

U jeziku ML isti primjer izgleda ovako

```
let
  val top = a + b
  val bottom = c - d
in
    top / bottom
end;
```

Redoslijed deklaracija

Pascal i C89 dopuštaju da se varijable deklarišu samo na početku funkcije. C99, C++, Java, i C# dopuštaju da se deklaracije javljaju gdje i naredbe, uz uslov da se varijabla mora deklarisati prije upotrebe.

U C99, C++, i Java, opseg svih lokalnih varijabli je od deklaracije do kraja bloka u kome su deklarisane. U C#, opseg varijable u bloku je cijeli blok bez obzira gdje se nalazi deklaracija u bloku. Pošto se varijabla mora deklarisati prije upotrebe, razlika je minorna. Kako C# i Java ne dopuštaju ugniježden opseg s varijablama istog imena, može se uočiti sljedeći primjer koji je legalan u Java, ali nije u C#.

```
public class test {
public void Blah()
{
   for (int i = 0; i < 10; i++)
   {
      // do something
   }
   int i = 42;
}
}
```

U C++, Java, i C#, varijable se mogu definisati u for naredbama, a njihov opseg je ograničen na for naredbu .

Globalni opseg

C, C++, Pascal, BASIC, PHP, i Python dopuštaju da se deklaracija varijabli pojavi izvan definicija funkcija. C i C++ imaju i deklaracije (samo atributi) i definicije (atributi i

smještaj). Deklaracija izvan definicije funkcije može da odredi i da je ona definisana u drugoj datoteci, kao u sljedećem C primjeru.

```
extern int sum;
```

Ako je u C++, globalna varijabla sakrivena lokalnom s istim imenom, može joj se pristupiti operatorom opsega (::). Na primjer, ako je x globalna, skrivena u funkciji lokalnom varijablom koja se zove x globalnoj se pristupa kao ::x.

U PHP varijable koje su deklarisane u funkciji u funkciji su lokalne za funkciju. Varijable koje su deklarisane izvan funkcije imaju opseg od mjesta deklaracije do kraja programa ali taj opseg prelazi definicije funkcije. Stoga globalne varijable nisu direktno vidljive. Globalnim varijablama se pristupa nizom $GLOBALS ili deklarišući ih kao global .

```
$day = "Monday";
$month = "January";
function calendar() {
  $day = "Tuesday";
  global $month;
  print "local day is $day <br />";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month <br />";
}
calendar();
```

Rezultat je sljedeći:

```
local day is Tuesday
global day is Monday
global month is January
```

U JavaScript pristup globalnim varijablama je kao u PHP, ali nema načina pristupa globalnoj varijabli koja ima isto ime kao lokalna.

U Pythonu se varijable ne deklarišu, nego su implicitno deklarisane kada se nađu u naredbama dodjele. Globalna varijabla se može čitati u funkciji, ali ako se želi dodijeliti unutar funkcije, mora se deklarisati kao global.

```
day = "Monday"
 def tester():
  global day
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day
tester()
```

U gornjem primjeru, ako ne bi bilo naredbe global day, bila bi prijavljena greška UnboundLocalError jer bi bez nje varijabla day bila smatrana lokalnom varijablom.

Ocjena statičkog opsega

Statički opseg radi u mnogim situacijama. Ipak postoje i određeni problemi. Prvi je što u mnogim situacijama pruža više pristupa varijablama nego što treba. Drugo, kako se program razvija, inicijalna struktura se razbija i lokalne varijable postaju globalne, potprogrami također postaju globalni umjesto gniježdeni, pa se dolazi do strukture

koja dosta odudara od projektovane. Alternativa mogu biti kontrola pristupa varijablama i enkapsulacija.

Dinamički opseg

Opseg važenja varijabli u jezicima APL, SNOBOL4, raniji Lisp, neka struktuirana proširenja BASIC-a, je dinamički. To znači da je baziran na sekvenci poziva programskih potprograma, u vremenu, a ne u uokvirenju programskih blokova. Reference na varijable se povezuju s deklaracijama tražeći kroz lanac poziva potprograma koji su doveli izvršenje do ove tačke. Sljedeći primjer u jeziku Beta BASIC ilustruje dinamički opseg. Varijabla A je lokalna za potprogram XX, a globalna za glavni program. Da ovaj jezik koristi statički opseg, varijabla A bi bila i u potprogramu YY globalna, pa bi naredni primjer prikazivao broj 2. Međutim, zbog dinamičkog opsega prosljeđuje se lokalna varijabla iz podprograma XX u YY pa primjer ispisuje vrijednost 8.

```
10 DEF PROC XX
20   LOCAL A
25   LET A=8
30   PROC YY
40 END PROC
50 DEF PROC YY
60   PRINT A
70 END PROC
75 REM GLAVNI PROGRAM
80 LET A=2
90 PROC XX
```

Prednost dinamičkog opsega je u praktičnosti, jer se mnogi parametri potprogramima mogu proslijediti dodjelom vrijednosti varijablama, umjesto navođenjem parametara. Mane su što kad se potprogram izvrši, varijable postaju vidljive svim programima koji ga zovu, nemoguće je statički provjeriti tipove i programi su loše čitljivi.

Opseg i dužina života

Opseg i dužina života varijabli su srodni ali različiti koncepti. U Pascal-u, iako je opseg prostorna karakteristika a dužina života vremenska, strogo su povezani jer lokalne varijable postoje od begin naredbe potprograma u kome su deklarisane do odgovarajuće end naredbe.

Primjer razlike se vidi u static varijabli u C ili C++ funkciji, koje nastavljaju da čuvaju vrijednost i po završetku funkcije u kojoj su deklarisane iako su do narednog poziva te funkcije nepristupačne.

Referencirajuće Okruženje

Referencirajuće okruženje naredbe je skup svih imena vidljivih u naredbi. U jezicima statičkog opsega, to su lokalne varijable plus sve vidljive varijable u okružujućim opsezima

U jezicima dinamičkog opsega referencirajuće okruženje su lokalne varijable uz sve vidljive varijable aktivnih potprograma. Potprogram je aktivan ako je njegovo izvršenja započelo ali još nije završeno.

Koje je referencirajuće okruženje za sljedeći C program?

```
int c, d;
void sub1() {
  int a, b;
  . . .      //1
}  /* end of sub1 */
void sub2() {
  int b, c;
  .. . .     // 2
  sub1();
}  /* end of sub2 */
void main() {
  . . .     // 3
  sub2();
}  /* end of main */
```

Odgovor: Pošto je C jezik statičkog opsega

U tački 1 vide se a , b od subl i globalne c i d

U tački 2 vide se b , c od sub2, d globalno, (c globalni je skriven)

U tački 3 vide se globalne c ,d

Imenovane konstante

Imenovana konstanta je varijabla koja dobija vrijednost samo jednom. Time program dobija na čitljivosti. Npr. dielektrična konstanta u vakuumu se u Pascal-u može definisati kao

```
const eps0=8.854187817e-12;
```

i nakon toga koristiti eps0 u programu, umjesto navođenja ovog broja.

Druga primjena je za parametre programa. Npr. ako na više mjesta imamo deklaraciju nizova na 100 elemenata i petlje koje brojeći do 100 prolaze kroz te nizove, bolje je umjesto 100 pisati imenovanu konstantu. Ako je potrebno povećati ili smanjiti veličinu niza, izmjena se vrši na samo jednom mjestu.

U Pascal-u, konstanti se može dodijeliti samo prosta vrijednost. Modula 2 i Fortran 90 dopuštaju i konstantne izraze. Kod tih jezika vezanje varijabli za imenovane konstante je statičko (manifest konstante).

Ada, C++, i Java dopuštaju izraze bilo koje vrste

```
const int result = 2 * width + 1;
```

C# ima dva tipa imenovanih konstanti, readonly i const. Vrijednosti const se pripreme pri kompajliranju. Vrijednosti readonly imenovanih konstanti se pripreme dinamički pri izvršenju.

Rezime

Zavisnost velikih i malih slova i odnos između imena i specijalnih riječi predstavljaju pitanja dizajna imena u programskim jezivima. Varijable određuje šestorka: ime, adresa, vrijednost, tip, život, opseg. Povezivanje je pridruživanje atributa programskim entitetima. Skalarne varijable se kategorišu kao: statičke, stek dinamičke, eksplicitno heap dinamičke, implicitno heap dinamičke. Opseg je skup naredbi u kojima je varijabla vidljiva i može biti statički i dinamički. Referencirajuće okruženje je skup svih varijabli koje su vidljive u datoj naredbi.

6. Tipovi podataka

Tip podataka definiše kolekciju objekta podataka i skup predefinisanih operacija nad tim objektima. Tipovi podataka su se razvijali od prostih brojeva, preko nizova, slogova, korisničkih tipova do današnjih apstraktnih tipova podataka.

Problem dizajna za sve tipove podataka je koje operacije su definisane i kako se navode?

Primitivni tipovi podataka

Gotovo svi programski jezici imaju skup primitivnih tipova podataka. Primitivni tipovi podataka su oni koji nisu definisani preko drugih tipova podataka. Neki primitivni tipovi podataka su samo odraz hardvera (npr. cijeli brojevi). Drugi zahtijevaju određenu ne-hardversku podršku za njihovu implementaciju.

Cjelobrojni tip

Cjeli brojevi gotovo uvijek imaju odraz u hardveru, jer se operacije nad njima obavljaju instrukcijama ugrađenim u procesor. U nekim slučajevima (Pascal, Fortran) postoji samo jedan cjelobrojni tip, čiji je opseg obično usklađen s dužinom riječi procesora. Može biti i više cjelobrojnih tipova u jeziku. Na primjer, Java poznaje sljedeće cjelobrojne tipove byte, short, int, long, unsigned byte, unsigned short, unsigned int, unsigned long.

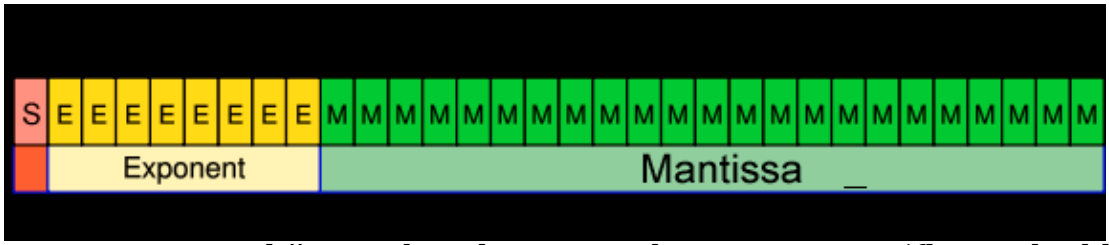
Negativni cijeli brojevi se obično čuvaju u 2komplement formatu ili se izdvoji jedan bit za predznak, a u ostalim bitima čuva apsolutna vrijednost.

Primjer cjelobrojnog tipa koji nije baziran na hardverskim instrukcijama su dugi cijeli brojevi u Python. Oni se pišu s znakom L na kraju

4123894238413904L

Broj u pokretnom zarezu

Zbog konačnog memorijskog prostora ne može se u računaru predstaviti tačno svaki realni broj. Na primjer, broj π ima beskonačno decimalnih cifara, pa se u memoriji može predstaviti samo aproksimativno. Čak i neki realni brojevi koji imaju konačan broj decimalnih cifara (poput 0.1) u binarnom sistemu se ne mogu predstaviti s konačnim brojem cifara.



Mnogi jezici podržavaju bar dva tipa pokretnog zareza (float i double); nekad i više takvih tipova. Ako procesor hardverski podržava instrukcije za operacije u pokretnom zarezu, obično format brojeva u pokretnom zarezu reflektuje format instrukcija. Nejčešći format je IEEE Floating-Point Standard 754 koji predviđa jedan bit za predznak broja, 8 ili 11 bita za eksponent i 23 ili 32 bita za mantisu..

Kompleksni broj

Neki jezici podržavaju kompleksne brojeve, na primjer C99, Fortran, i Python. Svaka vrijednost se sastoji od dva broja u pokretnom zarezu, realnog i imaginarnog. U Fortranu se kompleksne varijable deklarišu ključnom riječju COMPLEX, a pišu notacijom s zagradama i zarezom. Na primjer da se varijabli A dodijeli vrijednost 3+6i, piše se

```
COMPLEX A
A=(3.0,6.0)
```

Python podržava tzv. doslovni oblik: (7 + 3j), gdje je 7 realni dio a 3 imaginarni dio

Decimalni tip

Za poslovne aplikacije tačna reprezentacija brojeva je vrlo bitna, pogotovo u finansijama. Tu nema potrebe za vrijednostima manjim od 0.01, ali se svaka vrijednost mora tačno predstaviti.

COBOL je prvi jezik koji podržava decimalne brojeve, a od novijih jezika C# i F# pružaju decimalni tip. Kod ovog tipa se smiješta fiksni broj decimalnih cifara u binarnom obliku (BCD), tako što svaka decimalna cifra zauzima jedan bajt ili se po dvije decimalne cifre pakuju u jedan bajt.

Ovaj tip omogućava tačno predstavljanje brojeva, ali mana mu je ograničen opseg, veće zauzeće memorije i češća potreba za sotverskim računom.

Boolean

Logički tipovi su najjednostavniji prosti tipovi podataka. Predstavljaju pojedine zastavice i ispunjene uslove u programima. Opseg su im dva elementa "true" i "false" . Postoji u većini modernih programskih jezika osim C89. Mogao bi se implementirati bitom, ali obično se predstavlja bajtom radi jednostavnije operacije. Prednost ovog tipa nad upotrebom cijelih brojeva u istu svrhu je čitljivost.

Character

Znakovni podaci se smještaju kao numerički kodirani. Najčešće kodiranje je ASCII koji kodira znakove u jednom bajtu (ili sedam bita). Znakovni tip char u Algol, Pascal, C , Java, Fortran su ASCII.

Za podršku većem broju jezika u svijetu uvedeno je 16-bitno kodiranje: Unicode (UCS-2). Prvi jezik koji je definisao char u Unicode formatu je Java , a kasnije je dodan u C# i JavaScript koji također podržavaju Unicode.

32-bitni Unicode (UCS-4) je podržao Fortran, počev od 2003 .

Basic i Python umjesto tipa pojedinačnog znaka koriste tip string dužine 1.

Tipovi nizova znakova

Vrijednosti ovog tipa su nizovi znakova, koristi se i termin stringovi. Vrlo su bitni za sve vrste manipulacija s znakovima. Dva pitanja po kome se razlikuju programski jezici su:

- Da li je to primitivni tip ili samo specijalna vrsta niza?
- Da li dužina niza treba biti statička ili dinamička?

Operacije s nizovima znakova

Najčešće operacije s nizovima znakova su dodjela i kopiranje , poređenje u alfabetskom redoslijedu (=, >, itd.), spajanje, pristup podnizovima i poklapanje uzoraka. Dok su operacije nad podacima numeričkog tipa uglavnom slične u različitim jezicima, operacije nad nizovima znakova se mogu dosta razlikovati. Na primjer dodjela stringa stringu postavlja više pitanja šta raditi kada je odredišni niz znakova duži od izvornog.

U standardnom Pascalu niz znakova nije primitivni tip i on je dosta ograničen po pitanju rada s stringovima. Ako u deklaraciji niza znakova stoji klauzula packed, moguća je dodjela i poređenje relacionim operatorima.

```
var aa: packed array [1..10] of character
```

Proširenja Pascal-a Turbo Pascal i Delphi uvode tip String koji je primitivni tip i ima operacije spajanja, funkcije za pristup podnizovima i zaštitu od prekoračenja prilikom dodjele niza nizu.

U C i C++ niz znakova nije primitivni tip. Koristi se niz char znakova i biblioteka funkcija. Većina bibliotečkih funkcija koristi konvenciju da su nizovi znakova zaključeni specijalnim znakom null koji se predstavlja nulom. i nizovi znakova koje generiše kompajler su zaključeni nulom, na primjer:

```
 char str[] = "abc";
```

U ovom primjeru, str je niz char elemenata, ukupno 4 sa kodovima 87, 98, 99 i 0. Problem sa nizovima znakova u C je što neke funkcije nemaju zaštitu od prekoračenja Na primjer, u sljedećem primjeru će biti oštećen sadržaj memorije iza niza dest:

```
char dest [10];
char src="Veliki niz znakova je inicijaliziran";
strcpy(dest, src);
```

C++ u standardnoj biblioteci ima klasu string koja je bolje rješenje od char nizova. Uključuje i odgovarajuće operatore.

```
#include <iostream>
#include <string>
int main()
{
// Stringovi se mogu inicijalizirati C nizom
std::string foo = "fighters";
std::string bar = "stool";
// Operator != poredi stringove na različitost sadrzaja
if (foo != bar) {
        std::cout << "Razliciti esu." << std::endl;
}
//Operator + predstavlja spajanje stringova
    std::cout << (bar + " " + foo) << std::endl;
    return 0;
}
```

U Java stringovi nisu primitivni tip, ali ovaj jezik poznaje dvije vrste klasa iz standardne biblioteke. Klasa String je namijenjena za konstantne stringove, a klasaStringBuffer za nizove znakova koji se mogu mijenjati. Sljedeći primjer kreira string tipa StringBuffer, i na njega nadodaje novi string.

```
public class Test {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Test");
      sb.append(" String Buffer");
      System.out.println(sb);
   }
}
```

Fortran 77 i BASIC podržavaju stringove kao primitivni tip. Operator spajanja stringova u Fortranu je //, a u BASIC-u je +. Sljedeći Fortran primjer deklariše tri stringa različitih dužina i ilustruje spajanje dva stringa.

```
   CHARACTER A*4, B*2, C*8
         A = 'join'
         B = 'ed'
         C = A // B
         PRINT *, C
         END
```

U Microsoft BASIC string varijable u imenu imaju znak $ na kraju, a funkcijom MID$ se može izdvojiti podniz. Nije potrebno rezervisati dužinu. Sljedeći primjer će iz stringa A$ izbaciti blanko znak između dvije riječi

```
10 A$="Moj primjer"
20 B$=MID$(A$,1,3)+MID$(A$,5, 7)
30 PRINT B$
```

Python poznaje stringove kao primitivni tip, sa operacijama gledanja podnizova, spajanjima, pristupima pojedinim znakovima, traženjem i zamjenom. Ipak stringovi u Python-u su nepromjenjivi.

Perl, Perl, JavaScript, Ruby, i PHP poznaju nizove znakova kao primitivni tip i imaju najmoćnije operacije za rad s stringovima. Ova dva jezika svojoj osnovi imaju i operacije slaganja uzoraka regularnim izrazima. Regularni izrazi omogućavaju složena pretraživanja stringova. Na primjer, sljedeći regularni izraz se može koristiti za prepoznavanje identifikatora (oblik slovo pa više slova ili cifara)

```
/[A-Za-z][ A-Za-z\d]+/
```

Regularni izrazu su dodani u C++, Java, Python, C#, and F# kroz biblioteke.

SNOBOL4 podržava stringove kao primitivan tip. Mogućnosti poređenja uzoraka su jače od regularnih izraza, i ekvivalentni su kontekstno neovisnim gramatikama. Sljedeći primjer definiše funkciju WORDPAT koja izdvaja niz slova iz proizvoljnog stringa i dodjeljuje takav niz varijabli WORD.

```
LETTER = 'abcdefghijklmnopqrstuvwxyz'
WORDPAT = BREAK(LETTER) SPAN(LETTER) . WORD
```

U F# stringovi su klase i nadovezuju se + operatorom, a u ML su nepromjenjivi a operator nadovezivanja je ^.

Opcije za dužinu stringa

Ima više pristupa kako odrediti dužinu stringa.

Statički pristup (COBOL, Pascal, Fortran 77, Python, klasa String u Java, klasa String u Ruby) rezerviše u memoriji fiksnu dužinu stringa. Stringovi statičke dužine su uvijek puni. Ako je sadržaj stringa kraći od predviđenog, dopunjava se blankovima.

Kod stringova s **limitiranom dinamičkom dužinom** (C i C++) specijalni znak označava kraj stringa, umjesto čuvanja dužine, tako da string može biti dugačak između nula znakova i maksimalne veličine. Zauzeta memorija za dati string je jednaka njegovoj maksimalnoj veličini

Stringovi s **dinamičkom dužinom** bez maksimuma (SNOBOL4, Perl, JavaScript, PHP, BASIC, C++ STL stringovi, Delphi) niz znakova je promjenjive veličine. Ova opcija zahtijeva dodatni rad zbog dinamičke alokacije, ali pruža maksimalnu fleksibilnost.

Ada podržava sve tri opcije stringova.

Tipovi stringova pružaju veliku pomoć u mogućnosti pisanja. Kao primitivni tip statičke dužine, jeftini su za realizaciju. Stringovi dinamičke dužini su posebno korisni, ali treba zaključiti da li su vrijedni dodatnog programskog rada u implementaciji.

Implementacija nizova znakova

Nizovi znakova su ponekad podržani u hardveru, ali se češće realizuju softverski podrška. Potrebi podaci se čuvaju u deskriptorima. Zavisno od opcija za dužinu stringa, ovi deskriptori se koriste u trenutku kompajliranja i/ili u trenutku izvršavanja.

Statička dužina stringova treba deskriptor samo u vremenu kompajliranja. On se sastoji od imena varijable, dužine i memorijske adrese.

Ograničena dinamička dužina pored deskriptora u trenutku kompajliranja (isti sadržaj kao statička dužina), može ali i ne mora imati deskriptor u toku izvršavanja. U TurboPascal taj deskriptor je smješten neposredno ispred samog teksta niza znakova i sadrži trenutnu dužinu, dok u C on ne postoji jer je znakom nule označen kraj stringa.

Stringovi dinamičke dužine imaju deskriptor u toku izvršenja koji čuva trenutnu dužinu stringa. U nekim jezicima, poput BASIC-a čuva i ime stringa.

Korisnički definisani ordinalni tipovi

Ordinalni tip je onaj čiji se skup mogućih vrijednosti može povezati s skupom pozitivnih cijelih brojeva. Na primjer primitivni ordinalni tipovi u Java su integer, char i boolean.

Enumeracijski tipovi

Enumeracijski (pobrojani) tip je onaj kod koga se u sve moguće vrijednosti navode u definiciji kao imenovane konstante, Interno se oni preslikavaju u cjelobrojni tip, ali ipak postoje tri pitanja

- Da li se pobrojana konstanta može javljati u više od jedne definicije tipa i ako da kako se provjerava tip ove konstante?
- Da li se pobrojani tipovi automatski prevodi u integer?
- Da li se drugi tip može dodijeliti takvoj varijabli?

Sljedeći primjer definicije enumeracijskog tipa je iz C, C++, C# i Java 5.

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

Iako je sintaksa ista, postoje određene razlike. U C enumeracijski tip se automatski konvertuje u cijeli broj između 0 i broja pobrojanih elemenata, (npr. mon dobija vrijednost 0, wed dobija vrijednost 2). U C++ nije dopušteno varijabli tipa enum dodijeliti cijeli broj, ali obrnuto je dopušteno. U C# nema konverzije u cijeli broj uopšte.

Java ne vrši konverziju u int, a elementi enumeracija mogu biti i polja, konstruktori i metode.

U Pascalu se koristi sintaksa deklaracije enumeracije kao u sljedećem primjeru, a varijable enumeracijskog tipa mogu biti indeksi for petlje.

```
type DAYS= (mon, tue, wed, thu, fri, sat, sun);
```

U Ada je sintaksa slična kao u Pascalu, a ista vrijednost se može javljati i u više različitih enumeracija.

```
type DAYS is (mon, tue, wed, thu, fri, sat, sun);
type STARS is (sun, antares, betelquese);
```

U ML, enumeracijski tipovi se definišu kao novi tipovi u deklaraciji tipova podataka.

```
datatype weekdays =  Monday | Tuesday | Wednesday | Thursday | Friday
```

Perl, JavaScript, PHP, Python, Ruby, i Lua nemaju enumeracijski tip.

Enumeracijski tipovi pružaju pomoć u čitljivosti jer konstante ne treba kodirati kao cijele brojeve. Oni u nekim jezicima (npr C#, Ada i Pascal, ali ne C) povećavaju pouzdanost jer kompajler može provjeriti operacije (nema smisla sabirati boje) i opseg enumeracijskih varijabli.

Podintervalni tipovi

Podintervalni tipovi postoje u jezicima Pascal, Modula 2 i Ada. Ovi tipovi predstavljaju uređenu kontinualnu sekvencu ordinalnog tipa.

Sljedeći primjer u Pascal-u ilustruje definiciju podintervalnog tipa. Varijablama tipa index ne može se dodijeliti vrijednost veća od 100 i manja od 1, a varijablama tipa uppercase se mogu dodijeliti samo velika slova.

```
type
  uppercase='A'..'Z';
  index=1..100;
```

U jeziku Ada podintervalni tipovi (subrange) nisu novi tipovi, nego nova imena s dodatnim ograničenjima.

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;
Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

Podintervalni tipovi povećavaju čitljivost jer čitač shvata da varijable čuvaju samo određeni opseg vrijednosti. Povećavaju i pouzdanost jer se dodjela vrijednosti koja je izvan podopsega se smatra greškom kojase može detektovati tokom kompajliranja ili izvršenja.

Implementacija korisnički definisanih ordinalnih tipova

Enumeracijski tipovi se implementiraju kao cijeli brojevi.

Podintervalni tipovi se realizuju kao roditeljski tipovi s tim što kompajler ubacuje ograničenja na varijable podopsega pri svakoj dodjeli vrijednosti toj varijabli.

Skupovni tipovi

Skupovi postoje samo u Pascal i Modula 2. Varijable skupovnog tipa predstavljaju neuređene kolekcije vrijednosti nekog ordinalnog tipa. Operacije nad skupovima modeliraju matematičke operacije: dodjela vrijednosti, pripadnost, unija, razlika, presjek. Ovo je Primjer dodjele, unije i pripadnosti skupova u Pascalu.

```
type numberset = set of 1..40;
var mynumbers, othernumbers, unionnumbers : numberset;
begin
  mynumbers := [2..6];
  othernumbers := [4..10];
  unionnumbers := mynumbers + othernumbers + [14..20];
  if 12 in unionnumbers then writeln ('Belongs');
end
```

Skupovni tipovi se realizuju kao binarni cijeli brojevi kod kojih svaki bit je pridružen odgovarajućem elementu u domeni skupa. Ako je taj bit postavljen na 1 znači da se takav element nalazi u skupu. Operacije unije i presjeka se mogu realizovati kao logičke operacije nad bitima. Skupovi mogu da skrate neke programske konstrukcije, npr. provjera da li je znak samoglasnik se kraće radi sljedećom naredbom nego izrazom s 5 or operatora.

```
If ch in ['a', 'e', 'i', 'o', 'u'] then doit;
```

Tipovi nizova

Niz (array) je spoj homogenih elemenata podataka u kome se individualni element određuje pozicijom relativno u odnosu na prvi element. Prilikom dizajna nizova u programskom jeziku postavljaju se sljedeća pitanja:

- Koji tipovi podataka mogu biti indeksi?
- Da li se i kada se provjerava opseg indeksa?
- Kada se vrši alokacija?
- Koji je najveći broj dimenzija indeksa?
- Da li višedimenzionalni nizovi imaju jednak broj elemenata po dimenziji
- Mogu li se inicijalizirati nizovi objekata?
- Postoje li odsječci nizova?

Indeksiranje nizova

Indeksiranje (indexing, subscripting) je mapiranje indeksa na elemente

array_name (index_value_list) →element

FORTRAN, PL/I, Basic, Ada, COBOL koriste oble zagrade zagrade za predstavljanje indeksa nizu.

```
A=A+B(I)
```

Ada koristi zagrade radi uniformnosti između pristupa nizu i poziva funkcija pošto su oboje mapiranja vrijednosti. Mana oblih zagrada jeu smanjenoj čitljivosti

Većina ostalih jezika (Pascal, C, Modula 2, C++, Java, JavaScript, Perl, PHP, APL) koriste uglaste zagrade za elemente niza. To je dosta čitljivo osim ako su na tastaturi uglaste zagrade zamijenjene drugim simbolima (u YUSCII standardu na nekim terminalima [ ]su zamijenjene slovima š i ć).

```
a=a+b[i];
```

PostScript koristi operator get koji koristi dva parametra na steku.

```
 [1 6 3 0 9] 0 get pstack
1
```

Indeksi u FORTRAN, C, Java mogu biti samo samo cijeli broj. U Pascal, Ada indeksi mogu biti cijeli broj ili enumeracija (uključujući Boolean i char) .

U cilju povećanja pouzdanosti neki jezici provjeravaju da li je indeks u pravilnom opsegu prije pristupa elementu niza, dok drugi radi efikasnosti ne. C, C++, Perl, i Fortran ne obavljaju provjeru indeksa dok Java, Pascal, ML, C#,Ada obavljaju.

Višedimenzionalni nizovi u jednoj grupi jezika (Fortran, BASIC, COBOL, Pascal, C#, Modula 2) se posmatra kao jedinsveni niz. Kod takvih jezika elementu niza se pristupa navodiće njegove koordinate razdvojene zarezima unutar jedinstvenih zagrada.

MyArray[3,7] ili MyArray(3,7)

U drugoj grupi jezika (C, C++, PHP, Java, JavaScript) višedimenzionalni niz se posmatra kao niz nizova, pa se svaka koordinata piše unutar posebnih zagrada.

MyArray[3][7]

Opsezi indeksa i kategorije nizova

Opseg koji može imati indeks niza varira među jezicima. U C, C++, Java, C# najniža vrijednost indeksa je uvijek 0. U Fortran, Basic najniža vrijednost indeksa je 1, neke verzije omogućavaju promjenu. U jezicima Pascal, Modula 2, Ada treba navesti i najnižu i najvišu vrijednost indeksa.

Prema tome kako se pristupa memoriji na bazi indeksa elementa niza, nizove dijelimo u pet kategorija.

Kod **statičkih nizova** opseg indeks se statički poveže i alokacija memorije je statička, tj. poznata je prije izvršenja. Prostor za niz je u području globalnih podataka. Prednost ovakve alokacije nizova je efikasnost, jer se ne troši vrijeme na dinamičku alokaciju. Mana je što je alocirani prostor za niz zauzet tokom cijelog izvršavanja programa.

Fiksno stek-dinamički nizovi koriste statički opseg indeksa (broj elemenata niza je poznat u trenutku kompajliranja), ali se alokacija obavlja u trenutku deklaracije. Prostor za niz je u stek području. Prednost je prostorna efikasnost, što znači da se prostor alociran za niz u jednom potprogramu može koristiti u drugom potprogramu, kada se prvi završi. Mana je potrebno vrijeme alokacije i dealokacije (mada nije veliko)

Kod **Stek-dinamičkih nizova**: opseg indeksa se dinamički dobija i alokacija memorije je takođe dinamička (u toku izvršenja) . Prostor za niz je u stek području. Prednost je fleksibilnost (veličina niza se ne mora znati dok se ne koristi) , dok je mana vrijeme alokacije i ograničenost vremena života niza na vrijeme života potprograma.

Fiksni heap-dinamički nizovi su slični kao fiksni stek dinamički : povezivanje s memorijom je dinamičko ali niz ostaje na fiksnoj poziciji nakon alokacije (tj., povezivanje se desi pri zahtjevu a memorija alocira kroz heap, ne stek). Prednost je u fleksibilnosti, mana u još većem vremenu alokacije i dealokacije u odnosu na stek.

Heap-dinamički nizovi: su najfleksibilniji. i određivanje opsega i alokacija na heap-u su dinamički i mogu se mijenjati u toku rada. Mana je što alokacija i dealokacija zahtijevaju više vremena i mogu se dešavati više puta tokom programa.

Globalno definisani nizovi u Pascal, Fortran, Modula 2, C i C++ i nizovi s static modifikatorom definisani unutar funkcije u C i C++ su statički nizovi. Nizovi u C i C++ unutar funkcije bez static modifikatora su fiksni stek dinamički. C i C++ imaju i fiksne heap-dinamičke nizove (koristeći funkcje malloc, free odnosno operatore new delete). Stek dinamički nizovu su uvedeni u C99.

Ada podržava stek dinamičke nizove kao u sljedećem primjeru.

```
Get(List_Len);
declare
List : array (1..List_Len) of Integer;
begin
...
end;
```

U Java su negenerički nizovi fiksni heap dinamički, jer nakon kreiranja čuvaju opseg i memorijsko područje. Java uključuje i klasu ArrayList za fiksne heap dinamičke nizove.

C# ima generičke heap dinamičke nizove kroz klasu List. Sljedeći primjer prikazuje niz čija se veličina mijenja i elementi dodaju.

```
List<String> stringList = new List<String>();
stringList.Add("Michael");
```

Perl, PHP, JavaScript, Python, i Ruby podržavaju heap dinamičke nizove. Nizovi u Perl i Javascript mogu da rastu funkcijama push i unshift koji dodaju element na početku ili kraju niza,

U JavaScript nizovi mogu da budu rijetki. Na primjer, ako već postoje elementi list[0], list[1] i list[2] i zatim se uvede nareda

list[50]=42;

novonastali niz će imati 4 elementa i dužinu 51. Elementi list[3] do list[49] postaju undefined.

Inicijalizacija nizova

Neki jezici dopuštaju inicijalizaciju vrijednosti elemenata niza u trenutku alokacije prostora. Ova mogućnost jako skraćuje programski kod. Evo nekoliko primjera

Ovo je inicijalizacija niza cjelobrojnih vrijednosti u C, C++, Java, C#

```
int list [] = {4, 5, 7, 83}
```

Nizovi znakova u C i C++ se inicijaliziraju jednom vrijednošću.

```
char name [] = "freddie";
```

Nizovi stringova u C i C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

Java inicijalizacija niza String objekata

```
String[] names = {"Bob", "Jake", "Joe"};
```

Ada ima dva pristupa, navođenjem elemenata i navođenjem preslikavanja.

```
list1: array(1..5) of integer := (1,2,4,5,6);
List2 : array (1..5) of Integer :=
(1 => 17, 3 => 34, others => 0);
```

Python ima vrlo moćne mogućnosti inicijalizacije nizova kroz *List comprehensions* . Sljedeći primjer će kreirati niz čiji su elementi smjesti [0, 9, 36, 81]

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

Inicijalizacija nizova u Fortran 95 izgleda ovako.

```
Integer, Dimension (3) :: List = (/0, 5, 5/)
```

Heterogeni nizovi

Heterogeni niz je onaj čiji elementi ne moraju biti istog tipa. Podržavaju ih Perl, Python, JavaScript, i Ruby

Operacije nad nizovima

Neki jezici imaju definisane aritmetičke operacije nad cijelim nizovima, ne samo nad pojedinim elementima.

- Pascal dopušta dodjelu niza nizu.
- Ada dopušta dodjelu (operatorom :=) i spajanje nizova (operatorom &)
- Nizovi u Python-u se zovu liste (iako su to heap dinamički nizovi). Podržani su spajanje nizova (+), dodjela (=), poređenje reference na nizove (is) i poređenje pojedinačnih elemenata (==)
- Ruby ima poređenje pojedinačnih elemenata (==)
- Fortran pruža element operacije jer su one između pojedinih elemenata. Na primjer, + operator između dva niza rezultuje nizom suma parova elemenata dva niza
- APL ima najmoćnije operacije za rad s vektorima i matricama, kao i unarne operacije, ÷M računa inverznu matricu matrici M, A+B sabira dvije matrice, ΦM izvrće kolone matrice M...

Pravougaoni i neravnomjerni višedimenzionalni nizovi

Pravougaoni niz je višedimenzionalni niz u kome svi redovi imaju isti broj elemenata a sve kolone imaju isti broj elemenata.

Neravnomjerna matrica ima redove s varijabilnim brojem elemenata. Moguće ako je višedimenzionalni niz zapravo niz nizova

Fortran, Ada, Pascal podržavaju pravougaone nizove

Nizovi u Java, JavaScript, PHP, Ruby, Python su neravnomjerni.

C, C++ i C# u deklaracijama poput int a[4][8] realizuju nizove kao pravougaone. Međutim, ako je riječ o nizu pointera čijem se pojedinom elementu pristupa sintaksno na isti način kao pravougaonom (poput int * a[4]), tada nizovi mogu biti neravnomjerni.

Odsječci nizova

Odsječak je podstruktura niza, način pristupa pojedinim dijelovima niza. Na primjer, u višedimenzionalnom nizu, odsječak može biti pojedini red. Korisni su u jezicima koji imaju operacije nad nihzovima

Primjeri odsječaka nizova

Python: Sljedeći primjer će izdvojiti iz niza vector elemente na pozicijama 2,3 i 4. (od elementa 2 do elementa 5 ne uključujući ga)

```
>>> vector=[2,4,5,6,7]
>>> print vector[2:5]
[5, 6, 7]
>>>
```

Fortran 95

```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

Vector (3:6) je niz od četiri elementa

Ruby podržava odsječke slice metodom. Tako na primjer list.slice(2, 2) vraća treći i četvrti element liste list

Nizovi u COBOL se zovu tabele i isključivo su statički. Dimenzija im se deklariše ključnom riječju OCCURS. Međusobno preklapanje elemenata niza omogućava odsječke niza.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-TABLE.
    05 WS-A OCCURS 3 TIMES.
      10 WS-B PIC A(2).
      10 WS-C OCCURS 2 TIMES.
        15 WS-D PIC X(3).
PROCEDURE DIVISION.
  MOVE '12ABCDEF34GHIJKL56MNOPQR' TO WS-TABLE.
  DISPLAY 'WS-TABLE  : ' WS-TABLE.
  DISPLAY 'WS-A(1)   : ' WS-A(1).
  DISPLAY 'WS-C(1,1) : ' WS-C(1,1).
  DISPLAY 'WS-C(1,2) : ' WS-C(1,2).
  DISPLAY 'WS-A(2)   : ' WS-A(2).
  DISPLAY 'WS-C(2,1) : ' WS-C(2,1).
  DISPLAY 'WS-C(2,2) : ' WS-C(2,2).
  DISPLAY 'WS-A(3)   : ' WS-A(3).
  DISPLAY 'WS-C(3,1) : ' WS-C(3,1).
  DISPLAY 'WS-C(3,2) : ' WS-C(3,2).

STOP RUN.
```

Rezultat će biti

```
WS-TABLE  : 12ABCDEF34GHIJKL56MNOPQR
WS-A(1)   : 12ABCDEF
WS-C(1,1) : ABC
WS-C(1,2) : DEF
WS-A(2)   : 34GHIJKL
WS-C(2,1) : GHI
WS-C(2,2) : JKL
WS-A(3)   : 56MNOPQR
WS-C(3,1) : MNO
WS-C(3,2) : PQR
```

Implementacija nizova

Pri realizaciji nizova u programskom jeziku potreban je programski kod i u vrijeme kompajliranja i u vrijeme izvršavanja. Čak i ako je poznata memorijska lokacija početka najjednostavnijeg statičkog niza, za pristup pojedinom elementu njegovu adresu je potrebno izračunati.

Funkcija kojom se računa adresa elementa za jednodimenzionalne nizove izgleda ovako (gdje lower_bound predstavlja donju granicu indeksa elementa niza:

```
address(list[k]) =
address (list[lower_bound])
+ ((k-lower_bound) * element_size)
```

Ako jezik postavlja lower_bound na 0 (primjer C, C++,Java), formula je tada jednostavnija

```
address(list[k]) =
address (list) + k* element_size
```

Za Pristup višedimenzionalnim nizovima koriste se dva uobičajena načina, po redovima (u većini jezika) i po kolonama (u Fortranu). Višedimenzionalni pravougaoni niz se obično linearizuje i za lociranje elementa dvodimenzionalnog niza poredanog poredovima koristi se sljedeća formula.

```
Location (a[i,j]) =
address of a [row_lower_bound,col_lower_bound] +
(((i - row_lower_bound) * n) +
(j - col_lower_bound)) * element_size
```

Za svaku dimenziju niza potrebno je prilikom pristupa jedno sabiranje i jedno množenje.

U trenutku kompajliranja potrebne su sljedeće informacije:

Array
Tip elemenata
Tip indeksa
Koliko ima ukupno indeksa
Donja granica za prvi indeks
Donja granica za prvi indeks
Donja granica za drugi indeks
Donja granica za drugi indeks
..
Adresa niza

Asocijativni nizovi

Asocijativni niz je neuređena kolekcija elemenata podataka koji su indeksirani jednakim brojem vrijednosti koje se zovu ključevi. Sintaksno indeksiranje izgleda slično kao indeksiranje niza, ali u stvari struktura je znatno drugačija. Pored vrijednosti i korisnički ključevi moraju biti smješteni u memoriji. F'izička lokacija elementa asocijativnog niza se računa iz ključa heš funkcijom.

Asocijativni nizovi su ugrađeni tipovi u Perl, PHP, Python, Ruby, i Lua. Uz pomoć klasa postoje i u Java i C#. U Lua, podržani su tabelama.

Asocijativni nizovi su brži za pretragu od običnih nizova, ali su spotiji ako je potrebno obaviti operaciju nad cijelim nizom.

Primjer asocijativnih nizova u Perl

Imena počinju sa %; literali odvojeni zagradama

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed” => 65, …);
```

Indeksiranje vitičastim zagradama i ključem

```
$hi_temps{"Wed"} = 83;
```

Elementi se uklanjaju sa delete

```
delete $hi_temps{"Tue"};
```

Tipovi slogova

Slog je spoj elemenata podataka različitih tipova u kome se pojedini elementi identifikuju imenom. Postoji često potreba u programima za modeliranje je podataka u kod koga pojedini elementi nisu istog tipa ili veličine. Na primjer, informacije o studentima mogu uključivati ime, broj studenta, prosjek ocjena, i tako dalje. Tip podataka za takvu kolekcija može koristiti niz znakova za ime, cijeli broj za broj studenta, broj u pomičnom zarezu za prosjek ocjena, i tako dalje. Slogovi su dizajnirani za ovu vrstu potreba.

Slogovi nisu isto što i heterogeni nizovi. Kod heterogenih nizova, elementi su razbacani na heap-u, dok kod slogova zauzimaju susjedne lokacije.

U dizajnu slogova postavljaju se sljedeća pitanja.

- Koja je sintaksna forma referenci na polja?
- Da li je dopušteno ne-navođenje imena sloga?

Primjeri definicija slogova

COBOL koristi broj nivoa da prikaže hijerarhiju, drugi jezici idu na rekurzivnu definiciju

```
01 EMP-REC.
02 EMP-NAME.
05 FIRST PIC X(20).
05 MID PIC X(10).
05 LAST PIC X(20).
02 HOURLY-RATE PIC 99V99.
```

Slog EMPLOYEE-RECORD sastoji se od sloga EMPLOYEE-NAME i polja HOURLY-RATE. Brojevi 01, 02, i 05 kojima počinju linije deklaracije sloga su brojevi nivoa. Svaka linija koja je praćena linijom s višim brojem nivoa je također slog. Opis PIC definiše format polja. Tako X(20) predstavlja 20 alfanumeričkih znakova a 99V99 predstavlja četiri decimalne cifre s decimalnom tačkom u sredini.

Definicija slogova u Ada ili Pascal ne zahtijevaju brojeve nivoa, nego su definisane navođenjem imena dijelova. Npr u Ada

```
type Emp_Rec_Type is record
  First: String (1..20);
  Mid: String (1..10);
  Last: String (1..20);
  Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

u Pascal

```
type Empt=record
   name:array[1..10] of char;
   salary:real;
end;
var list:Empt;
```

U C, C ++ i C#, slogovi su podržani sa tipom struct. Na primjer

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

U C++, strukture su varijacija klase čiji su svi članovi javni. U C#, struct se također se odnosi na klase, ali su sasvim drugačije.

U C# strukture se alociraju na steku, za razliku od objekata klase, koji se alociraju na heapu.

Java ne poznaje slogove, ali se u istu svrhu mogu koristiti klase i ugniježdene klase.

Reference na slogove i polja

Većina jezika koji poznaju slogove dopuštaju njihovo gniježdenje. Za pristup pojedinačnom polju koriste se dva pristupa

COBOL koristi ključnu riječ OF za predstavljanje elementa polja, gdje je prvi slog koji se navodi najmanji unutrašnji slog koji uokviruje polje.

```
field_name OF record_name_1 OF ... OF record_name_n
```

Ostali jezici koji podržavaju slogove uglavnom koriste notaciju s tačkom između uokviravajućih slogova i pojedinog polja, pri čemu se prvo navodi najširi slog, pa onda redom unutrašnji.

```
record_name_1.record_name_2. .. record_name_n.field_name
```

teacher.personal.name="John"

Puna referenca uključuje imena svih slogova koji uokviruju polje do kojeg se želi doći. Skraćena referenca dopušta izbacivanje imena slogova sve dok je referenca jednoznačna.

Pogledavši primjer deklaracije u COBOL, FIRST, FIRST OF EMP-NAME, i FIRST OF EMP-REC su skraćene reference na polje FIRST OF EMP-NAME OF EMP-REC.

Pascal i Modula 2 imaju with naredbu koja kaže da unutar bloka se podrazumijeva ime sloga.

Operacije nad slogovima

Većina jezika koji poznaju slogove, podržavaju dodjelu sloga slogu, ukoliko su identične strukture ili imena tipa. Dodjela je čista ako su slogovi identični

Ada dopušta poređenje slogova na jednakost.

COBOL omogućava dodjeljivanje sloga slogu i ako nisu identične strukture. MOVE CORRESPONDING kopira polje izvornog sloga u odgovarajuće istoimeno polje odredišnog sloga

Ocjena slogova i poređenje s nizovima

Slogovi se koriste kada je kolekcija podataka heterogena. Pristup elementima nizova je sporiji od pristupa poljima sloga jer su indeksi dinamički a imena polja statička Dinamički indeksi bi se mogli koristiti i sa pristupima slogu, ali ne bi bilo provjere tipa i to bi bilo sporije

Implementacija tipa sloga

U trenutku kompajliranja potrebno je uz ime polja imati navedenu njegovu adresu relativno od početka sloga. Stoga opis sloga izgleda ovako

Slog
Ime polja 1 Tip polja 1 Pozicija polja 1
Ime polja 2 Tip polja 2 Pozicija polja 2
...
Adresa

Tipovi listi

Liste su prvo nastale u prvom funkcionalnom programskom jeziku, LISP. Nedavno su se pojavile i u imperativnim jezicima. Za razliku od nizova, liste su rekurzivno definisane.

Liste u Scheme i Common LISP omeđuju zagradama bez separatora između elemenata

```
(A B C D)
```

Ugniježdene liste imaju isti oblik

```
(A (B C) D)
```

Neke operacije nad listama u LISP i Scheme

(CAR '(A B C)) CAR funkcija vraća prvi element liste koja je parametar

(CDR '(A B C))CDR vraća listu bez 1. elementa

(CONS 'A '(B C)) Spajanje atoma i liste

(LIST 'A 'B '(C D)) Spajanje u listu vraća novu lisrt (A B (C D)).

ML ima liste i operatore nad listama. Liste se navode u uglastim zagradama, a elementi razdvajaju zarezima

```
[5, 7, 9]
```

Spajanje atoma s listom se obavlja operatorom ::. Na primjer

```
3 :: [5, 7, 9]
vraća novu listu: [3, 5, 7, 9].
```

ML ima funkcije koje odgovaraju CAR i CDR, Zovu se hd i tl

```
hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]
```

Tip liste u Python je efektivno niz, pa je objašnjen u tom poglavlju.

Tipovi n-torki

N-torka je tip podataka sličan slogu ali se elementi ne imenuju.

Python uključuje tip neizmjenjive n-torke. Ako se n-torka mora promijeniti, konvertuje se u niz. Kreira se dodjelom literala

```
myTuple = (3, 5.8, 'apple')
MyTuple[1]
```

ML uključuje tip n-torki.

```
val myTuple = (3, 5.8, 'apple');
#1(myTuple);
```

Nova N-torka se može definisati type deklaracijom

```
type intReal = int * real;
```

Tipovi unija

Unija (varijantni slog) je tip čijim varijablama je dopušteno da smjeste različite vrijednosti tipova u različita vremena izvršenja. Unije se koriste iz dva razloga:

Prvi razlog je ušteda prostora kada se na isto mjesto trebaju čuvati podaci različitog tipa. Na primjer, neka se tabela varijabli u sastoji od niza struktura čiji svaki element sadrži ime i vrijednost koja je cijeli broj boolean ili broj u pokretnom zarezu. Kako

pojedini element može biti samo jednog od ovih tipova, te vrijednosti mogu dijeliti istu lokaciju.

Drugi razlog je pristup internoj strukturi podataka. Na primjer, ako je potrebno pročitati mantisu i eksponent broja u pokretnom zarezu, to se može učiniti ako broj u pokretnom zarezu dijeli isti prostor s nizom osmobitnih cijelih brojeva.

Unije postavljaju sljedeća pitanja:

- Da li je potrebna provjera tipova pri dodjeli vrijednosti (ona se obavlja dinamički, pri izvršenju)
- Da li su unije varijanta slogova ili poseban tip?

Diskriminisane i slobodne unije

Fortran, C, i C++ pružaju konstrukcije unija u kojim nema jezičke podrške za provjeru tipova. Ovakva unija je slobodna unija . Kod sljedeće C unije, varijabli x se dodjeljuje besmislena float vrijednost

```
union flexType {
  int intEl;
  float floatEl;
};
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl;
```

Provjera tipova unija zahtijeva da svaka unija uključi indikator tipa koji se zove diskriminanta. Podržavaju ih Ada i Pascal. Sljedeći primjer u Ada ilustruje diskriminisane unije.

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
  Filled: Boolean;
  Color: Colors;
  case Form is
    when Circle => Diameter: Float;
    when Triangle =>
      Leftside, Rightside: Integer;
      Angle: Float;
      when Rectangle => Side1, Side2: Integer;
   end case;
end record;
```

Ocjena unija

Slobodne unije su nesigurne jer ne podržavaju provjeru tipova. Radi sigurnosti, Java i C# ne podržavaju unije. Ada diskriminisane unije su bezbjedne.

Pointeri i reference

Pointerska varijabla ima opseg vrijednosti koji se sastoji od memorijskih adresa i specijalne vrijednosti , nil . Vrijednost nil indicira da se pointer trenutno ne može koristiti da pokazuje na memorijsku ćeliju.

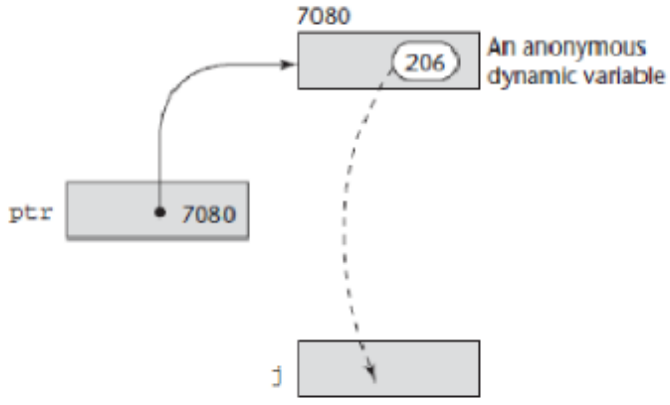
Pointeri su uvedeni s dva cilja na umu. Prvo, oni pružaju mogućnost indirektnog adresiranja, poznatog u asemblerskom jeziku. Drugo, oni omogućavaju upravljanje dinamičkom memorijom. Pointer se može koristiti za pristup lokaciji u području gdje je pristup dinamički kreiran (koristeći heap). Varijable u heap području često nemaju pridružene identifikatore i može im se pristupiti samo preko pointera. Varijable bez imena se zovu anonimne varijable. Problemi dizajna pointera

- Koji je opseg i dužina života pointerske varijable?
- Koja je dužina života heap dinamičke varijable?
- Da li su pointeri ograničeni na tipove vrijednosti na koju pokazujut?
- Da li se pointeri koriste za dinamičko upravljanje memorijom, indirektno adresiranje ili oboje?
- Treba li jezik podržavati tipove pointera, tipove referenci ili oboje?

Pointerske operacije

Dvije fundamentalne pointerske operacije su dodjela i dereferenciranje

Dodjela se koristi za postavljanje vrijednosti pointerske varijable na korisnu adresu Ako se pointerske varijable koriste samo za dinamičku memoriju, mehanizam alokacije bilo operatorom bilo potprogramom inicijalizira pointersku varijablu. Ako se koriste za indirektno adresiranje varijabli koje nisu heap dinamičke, mora postojati ugrađeni operator ili potprogram za dobavljanje adrese varijable koja se dodjeljuje pointerskoj varijabli.



Dereferenciranje povlači vrijednost na lokaciji predstavljenoj pointerskom vrijednošću. Dereferenciranje može biti eksplicitno ili implicitno

C++ koristi eksplicitne operacije preko *

```
j = *ptr
```

postavlja j na vrijednost koju pokazuje ptr

Primjer dodjele pointera

Operacija dodjele

```
j = *ptr
```

Problemi sa pointerima

Viseći pointeri se dešavaju ako pointer pokazuje na na heap-dinamičku varijablu koja je bila dealocirana. Opasni su jer ako se to područje ponovo alocira oni pokazuju na besmislene podatke. Mogu se desiti npr. u sljedećem slučaju.

1. Alocira se prostor na heap i dodijeli varijabli pl da pokazuje na njega
2. Varijabli p2 se dodijeli pl
3. Dealocira se prostor na heap alociran s pl, pl postaje nil ali p2 pokazuje na sada nevažeče područje

Izgubljena heap-dinamička varijabla je alocirana heap-dinamička varijabla koja više nije dostupna korisničkom programu (zove se i smeće). Dešavaju se npr u sljedećem slučaju.

1. Pointer pl se postavi da pokazuje na novokreiranu heap-dinamičku varijablu
2. Pointer pl se kasnije postavi da pokazuje na drugu novokreiranu heap-dinamičku varijablu

Proces gubitka heap-dinamičkih varijabli se zove oticanje memorije.

Pointeri u Ada

U Ada se pointeri zovu access tipovi. Ne koristi se poseban operator za dereferenciranje.

```
type person is
        record
                name :string(1..4);
                age    :0.. 150;
        end record;
fred   :person
type person_ptr is access person;
someone      :person_ptr;
...
someone:= new person;
fred.name   := "fred";
someone.name:= "jim ";
```

Viseći pointeri su djelomično izbjegnuti jer se dinamički objektui mogu automatsi dealocirati na kraju opsega pointerskog tipa

Problem izgubljene heap-dinamičke varijable nije uklonjen u Ada (moguće s UNCHECKED_DEALLOCATION)

Pointeri u C i C++

Pointeri u C i C++ su vrlo fleksibilni ali se moraju oprezno koristiti. Pointeri mogu pokazivati na bilo koju varijablu, bez obzira kada ili gdje je alocirana. Koriste se za dinamičko smještanje memorije i adresiranje . Moguća je i pointerska aritmetika: uvećavanje ili umanjenj pointera za cjelobrojnu vrijednost i oduzimanje dva pointera. Dereferenciranje je eksplicitno i koristi se simbol *. Postoji operator adrese &

Domenski tip ne mora biti fiksan (void *) .void * može pokazivati na bilo koji tip i biti provjeravan (ne može se dereferencirati)

Primjer pointera u C i C++

```
float stuff[100];
float *p;
p = stuff;
*(p+5) je ekvivalentno s stuff[5] i p[5]
*(p+i) je ekvivalentno s stuff[i] i p[i]
```

Usporedba pointera u Pascal C i Ada data je na sljedećoj tabeli

	Pascal	C	Ada
Pristup polju strukture na koju pokazuje pointer	aˆ.fieldname	*a.fieldname a->fieldname	a.fieldname
Kopiranje pointera	b := a;	b = a;	b := a;
Alociranje	new(a);	a=malloc (sizeof(float));	a:=new float;
Kopiranje vrijednosti	bˆ := aˆ;	*b = *a;	b.all := a.all

Tipovi referenci

Reference su slične pointerima, sa jednom razlikom. Pointeri sadrže proizvoljnu memorijsku adresu, a reference isključivo pokazuju na objekat u memoriji. Stoga referenca nema pointersku aritmetiku. C++ uključuje specijalni pointerski tip referencni tip koji se koristi primarno za formalne parametre . Sljedeći primjer ilustruje reference.

```
int result = 0;
int &ref_result = result;
...
ref_result = 100; //mijenja i result
```

Prednosti prosljeđivanja po referenci i vrijednosti

Java svim instancama klasa pristupa preko referenci. Na primjer u narednom primjeru varijabla strl je referenca na tip objekta String, kojoj se dodjeljuje objekat.

```
String str1;
. . .
str1 = "This is a Java literal string";
```

C# uključuje reference iz Java i pointere iz C++, ali upotreba pointera nije preporučljiva.

Ocjena pointera

Pointeri su kao goto-- proširuju opseg ćelija kojima se može pristupiti preko varijable, što je korisno i opasno. Pointeri ili reference su potrebni za pristup dinamičkim strukturama podataka –ne može se dizajnirati moderan jezik bez njih. Reference u Java i C# pružaju mogućnosti pointera bez opasnosti koje pružaju viseći pointeri.

Implementacija pointera

Pointeri sami po sebi su jednostavna struktura. Na većini računarskih arhitektura oni su varijabla koja sadrži linearnu memorijsku lokaciju., koja se često mapira u cjelobrojnu vrijednost Neke arhitekture su drugačije, npr Intel mikroprocesori koriste

segment i ofset (DOS, Windows 3), pa se kod njih pointerska varijabla sastoji iz dva dijela.

Problem visećih pointera se može izbjeći složenijom organizacijom pointera po cijenu . troška u vremenu i prostoru.. Jedna takva je **tombstone**: dodatna ćelija heap-a koja je pointer na heap-dinamičku varijablu. Stvarna pointerska varijabla pokazuje samo na tombstone. Kada se heap dinamička varijabla dealocija, tombstone ostaje ali je nil. Još jedan pristup su **brave i ključevi**: Pointerske vrijednosti su parovi (ključ, adresa). Heap-dinamičke varijable su varijabla plus ćelija za cjelobrojnu vrijednost brave. Kada je heap-dinamička varijabla alocirana, kreira se vrijednost brave i smješta u ćeliju brave i ćeliju ključa pointera. Svako dereferenciranje provjerava jesu li jednaki i ključ i brava.

Heap upravljanje

Unutar izvršnog mehasnizma programa koji dopušta dinamičko upravljanje memorijom nalazi se heasp sistem. Vrlo kompleksan izvršni proces čuva podatke o tome koje su ćelije jedne ili više veličina slobodne ili zauzete.Na primjer ispred područja rezervisanog za dodjelu programima se nalazi ulančana struktura koje pokazuju na slobodne i zauzete blokove.

Najjednosavniji heap mehanizam čuva adresu zadnje slobodne memorijske lokacije, i kada se alocira prostor ta adresa se dodijeli pointeru za pristup heap dunamičkoj varijabli, a zadnja slobodna memorijska lokacija uveća za zauzeti prostor. Ako se oslobodi neka heapdinamička varijabla, njena veličina i pozicija se stave u listu slobodnih blokova, koja se pretražuje pri novim alociranjima.

Pošto više pointera može pokazivato na istu heapdinamičku varijablu, postavlja se pitanje kada smijemo da oslobodimo njenu memoriiju? **Brojači referenci** čuvaju brojač u svakoj ćeliji u kome se nalazi koliko pointera pokazuju na ovu ćeliju Svako dodjeljivanje pointera na heapdinamičku varijablu uvećava ovaj brojač. Prednost pristupa je što je proces odluke o oslobađanju ćelije dosta brz- Mane su u prostoru (ako je veličina brojača uporediva s veličnom podatka na koji pokazuje) , sporijoj operaciji dodjele pointera i problemima s cirkularnim listama (svi elementi takve liste imaju brojač referenci najmanje 1)

Provjera tipova

Provjera tipova je aktivnost kojom se osigurava da su operandi operatora kompatibilnih tipova, Kompatibilni tip je onaj koji je legalan za operator ili dopušten pod pravilima jezika da se implicitno konvertuje kompajlerski generisanim kodom u legalan tip. Ova konverzija se zove prinuda (coercion).

Ako operatori nisu kompatibilnog tipa dešava se greška tipa. Programski jezik je strogo tipiziran ako se greške tipova odmah prepoznaju. Prednost strogih tipova: dopušta prepoznavanje varijabli koje rezultuju u greškama tipova

Ako je povezivanje tipa statičko, gotovo sva provjera tipova može biti statička. Ako su povezivanja tipova dinamički i provjera se mora dinamički raditi

Stroga provjera tipova

Jezik je strogog tipa ako se svaka greška tipova odmah detektuje. Ako varijable mogu imati vrijednosti različitih tipova, ta detekcija se mora vršiti i u toku izvršenja programa ako se dodjeljuje vrijednost koja nije u listi predviđenih tipova.

FORTRAN 95 nije: strogo tipiziran. parametri, Tip parametara potprograma se ne provjerava, a postoji i EQUIVALENCE deklaracija kojom varijable dijele memorijski prostor.

C i C++ nisu strogo tipizirani- Provjera tipova parametara funkcija se može izbjeći, a unije se ne provjeravaju na tipove

ML i F#, Pascal, Ada, Java i C# se smatraju strogo tipiziranim jezicima. Ipak, i u nekim od njih postoje mogućnosti da se zaobiđe. Typecasting u Java i C#, i u Ada funkcija Unchecked_Conversion

Forth nema nikakvu provjeru tipova kao i asemblerski jezik

Ekvivalentnost tipova

Ekvivalentnost imena tipova znači da dvije varijable imaju ekvivalentne tipove ako su u istoj deklaraciji ili u deklaraciji s istim imenom tipa. Ovo je lako implementirati ali previše restriktivno. Prvo, svi tipovi moraju imati ime. Time, podposezi cijelog broja nisu ekvivalentni cijelom broju . U sljedećem primjeru iz Ada, varijabla count se ne može dodijeliti varijabli index.

```
type Indextype = 1..100;
count = Integer;
index = Indextype;
```

Formalni parametri potprograma moraju biti istog tipa kao odgovarajući aktuelni (Pascal)

Ekvivalentnost strukture dvije varijable znači da su ekvivalentne ako im tipovi imaju istu strukturu . Ovo je fleksibilnije i teže za realizovati. Na primjer, sljedeća dva tipa su ekvivalentna.

```
type Celsius = Float;
Fahrenheit = Float;
```

Ako to ne želimo koriste se izvedeni tipovi koji su bazirani na postojećem ali nisu ekvivalentni.

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

Strukturalna ekvivalencija postavlja razna pitanja. Npr: Da li su dva tipa sloga ekvivalentna ako imaju istu strukturu ali imaju različita imena polja? Da li su dva niza ekvivalentna ako su isti osim različitih indeksa? (npr. var a: array [1..10] of integer i var b: array [0..9] of integer) Da li su sva enumeracijska tipa ekvivalentna ako im se komponente drugačije nazivaju a imaju isti broj komponenti? Strukturalnom ekvivalencijom se ne može razlikovati tip različite strukture (npr. Različite jedinice za brzinu a obije float)

U C struct, enum i union imaju ekvivalenciju imenom, a nizovi strukturom . Treba reči da typedef ne uvodi novo ime nego samo daje dodatno ime postojećem. U sljedećem primjeru book, anybook i book2 su međusobno ekvivalentnog tipa, a book3 nije. Kada se ukloni komentar, kompajliranje neće proči.

```
s #include <stdio.h>
#include <string.h>

typedef struct Books {
   char title[50];
   char author[50];
} Book;
struct knjiga {
   char title[50];
   char author[50];
} Book3;

int main( ) {
   Book book,anybook;
   struct Books book2;
   struct knjiga book3;
   strcpy( book.title, "1984");
   strcpy( book.author, "George Orwell");
   book2=book;
 //  book3=book;
   return 0;
}
```

Izuzetak je situacija u C kada su strukture definisane u različitim datotekama. Tada se koristi strukturna ekvivalencija. Ovog izuzetka nema u C++.

Pascal je previše striktan s ekvivalencijom imena, jer su dimenzije niza sastavni dio tipa, pa se potprogramu ne mogu proslijediti nizovi različitih veličina.

Ada koristi ekvivalenciju imena u svim tipovima osim kod neograničenih nizova kada se koristi strukturna ekvivalencija.

Fortran i COBOL nemaju imenovane korisničke tipove, pa kod njih nema ekvivalencije imena.

U objektnim jezicima, kompatibilnost tipova je vezana za hijerarhiju objekata. Obično se objektima naslijeđene klase može dodijeliti objekat roditeljske klase.

Rezime

Tipovi podataka jezika su njegov veliki dio i određuju stil i upotrebu jezika. Primitivni tipovi podataka većine jezika uključuju numeričke, znakovne i Bulove tipove Korisnički definisane enumeracije i tipovi pod-opsega povećavaju čitljivost i pouzdanost programa .Nizovi i slogovi su uključeni u većinu jezika. Pointeri se koriste za fleksibilno adresiranje i kontrolu dinamičkog upravljanja memorijom. Tipovi su ekvivalentni ako se varijable instance mogu dodijeliti jedan drugom.

7. Izrazi i naredbe dodjele

Izrazi su osnovni način računa u programskom jeziku. Da se razumije način računa izraza treba biti familijaran s redoslijedom računa operatora i operanada. Suština imperativnih jezika je dominantna uloga naredbe dodjele Ove naredbe računaju izraze i imaju bočni efekat upisa vrijednosti varijabli.

Aritmetički izrazi

Račun aritmetike je bio jedna od motivacija za razvoj prvih programskih jezika. Aritmetički izrazi sastoje se od operatora, operanada, zagradi i poziva funkcija. Operatori se dijele na unarne, binarne i ternarne. Unarni operator ima jedan operand, binarni operator ima dva operanda, ternarni operator ima tri operanda. Pitanja dizajna aritmetičkih izraza su:

- Koja su pravila prioriteta operatora?
- Koja su pravila asocijativnosti operatora?
- Koji je redoslijed računa operanada?
- Koji su bočni efekti računa operanada
- Postoji li preopterećenje operatora?
- Da li je dopušteno miješanje tipova u izrazima?

Načini pisanja izraza

Većina programskih jezika koristi infiksnu formu pisanja aritmetičkog izraza. Kod ove forme, izrazi se sastoje od operanada između kojih se nalaze operatori.

```
a+5*b-(c+2)
```

Prefiksna forma je primijenjena u Lispu i njegovim derivatima kao što je Scheme. Ovakav oblik pojednostavljuje sintaksnu analizu i unificira operatore i pozive funkcija.

```
(- (+ a (* 5 b)) (+ c 2))
```

Postfiksna forma je primijenjena u jezicima Forth i Postscript . Izrazi se računaju preko internog steka, sa operatorima koji obavljaju operacije nad vrijednostima na njegovom vrhu. Primjer iz jezika Forth.

```
c @ 2 + b @ 5 * a @ + -
```

Jezička forma je primijenjena u COBOL-u i asemblerskim jezicima. Umjesto zapisa izraza u matematički uobičajenom obliku, kalkulacije se provode naredbama.

```
MULTIPLY B BY 5 GIVING M.
ADD A TO M.
ADD 2 TO M
SUBTRACT C FROM M.
```

Pravila prioriteta i asocijativnosti operatora

Pravila prioriteta operatora

Pravila prioriteta operatora za račun izraza definišu redoslijed u kome se "susjedni" operatori različitih nivoa prioriteta računaju. Prioriteti operatora su nastali prije programskih jezika, jer je uobičajeno u matematici da se množenje obavlja prije sabiranja. Tipični nivoi prioriteta su

1. zagrade
2. unarni operatori
3. ** (stepenovanje ako ih jezik podržava)
4. *, /
5. +, -

Jezici bazirani na C imaju veliki broj operatora, i time i njihovih prioriteta. Sljedeća tabela pokazuje prioritete operatora u C

Ptioritet	Operator	Značenje	Asocijativno
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Pravila asocijativnosti operatora u računu izraza definišu redoslijed kojim se računaju operatori istog nivoa prioriteta. Na primjer u izrazu a+b-c, gdje su sabiranje i oduzimanje jednakog prioriteta, postavlja se pitanje da li se prvo izračuna a, pa njemu

sabere b, pa od njega oduzme c ili se prvo izračuna c, oduzme od b pa sabere s a. U prvom slučaju govorimo o lijevoj asocijativnosti, a u drugom o desnoj asocijativnosti.

Tipično u programskim jezicima četiri osnovne operacije su lijevo asocijativne.

Stepenovanje (**) u Fortran-u je desno asocijativno. Stepenovanje (^) u BASIC-u je lijevo asocijativno. Stepenovanje u Ada (**) nije asocijativno, tj moraju se koristiti zagrade (A ** B) ** C.

Unarni operatori se računaju s desna na lijevo (npr neki operatori u FORTRAN,C), ili s lijeva na desno (Ruby, Ada, Pascal)

U APL su svi operatori su jednakog prioriteta i asocijativni s desna na lijevo.

U Ruby svi aritmetički, relacioni i operatori dodjeljivanja, indeksiranja nizova, šiftovanja i operacija nad bitima su implementirani kao metode. Rezultat je da se ti operatori mogu preklopiti aplikacionim programima.

U Lisp-u i Forth-u se operatori realizuju potprogramima koji se moraju eksplicitno pozvati

Pravila prioriteta i asocijativnosti se mogu prevazići zagradama

Uslovni izrazi

C-bazirani jezici (npr., C, C++, Java) poznaju ternarni operator uslovnog izraza oblika

expression_1 ? expression_2 : expression_3

Ako je expression_1 imao tačnu vrijednost, rezultat će biti expression_2 inače će rezultat biti expression_3.

Primjer:

```
average = (count == 0)? 0 : sum / count ;
```

Evaluira se kao da je bilo

```
if (count == 0)
average = 0
else
average = sum /count ;
```

Redoslijed računa operanada

Redoslijed kada se primjenjuju operatori nad operandima određen je prioritetom i asocijativnosti operatora. No postavlja se i pitanje način i vrijeme računa operanada.

Što se tiče načina, varijable se računaju tako što se dohvati vrijednost iz memorije. Konstante se nekad računaju kao dohvatanje iz memorije, nekad unutar instrukcije u mašinskom jeziku. Izrazi u zagradama se moraju prvo izračunati kako bi se koristili kao operandi. Najinteresantniji slučaj je kada je operand poziv funkcije.

Bočni efekti

Redoslijed računa operanada je nebitan, ako se ne javljaju bočni efekti koji mogu da utiču na rezultat izraza. To su situacije kada jedan od operanada mijenja neki od drugih.

Najčešći primjer su funkcionalni bočni efekti, situacije kada funkcija mijenja dvosmijerni parametar ili nelokalnu varijablu. U sljedećem primjeru, ako funkcija fun mijenja vrijednost varijable a, nije svejedno da li će se prvo računati lijevi ili desni sabirak.

```
b = a + fun(&a);
```

Dva su moguća rješenja za problem funkcionalnih bočnog efekata. Jedno je napisati definiciju jezika koja **zabranjuje** funkcionalne bočne efekte. To znači da su zabranjeni dvosmjerni parametri i nelokalne reference u funkcijama. Rješenje radi, ali je nefleksibilno.

Drugo rješenje je da se u definiciji jezika odredi koji je redoslijed računa operanada. Na primjer, u Java se operandi računaju s lijeva na desno. Mana ovog rješenja je što se ograničava optimizacija generisanog koda.

Preopterećeni operatori

Upotreba operatora za više od jedne namjene se zove preopterećenje operatora. Dosta je uobičajeno da se operator + koristi za sabiranje int i float brojeva, a u Java i BASIC-u i za spajanje stringova.

U nekim slučajevima oni su potencijalni problem. Npr., u C i C++ operator & je bitovsko AND ili adresa varijable. To smanjuje čitljivost programa, i ponekad predstavlja gubitak mogućnosti detekcije greške, npr ako bi se u narednoj narednom izrazu ispustila varijabla a, dobili bi pogrešan rezultat, koji ponekad neće biti detektovan,

a & b;

Neki jezici koji podržavaju apstraktne tipove podataka, na primjer C++, C#, i F#, omogućavaju korisničko preopterećenje operatora. Na primjer, mogu se definisati operacije sabiranja i množenja matrica. Operator se onda realizuje pišući potprogram koji obavlja novu operaciju. Na primjer

A * B + C * D

se može pisati umjesto

MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))

Dati primjer pomaže čitljivosti. Međutim, korisnički operatori je mogu i smanjiti, na primjer, ako se definiše da * obavlja oduzimanje matrica. Čitljivost može trpiti, čak i kada operatori imaju smisla. Na primjer množenje matrice skalarom i množenje matrice matricom se implementiraju u odvojenim potprogramima koje je nekad teško naći.

Konverzija tipova

Sužavajuća konverzija je kada se konvertuje objekat u tip koji ne može uključiti sve vrijednosti izvornog tipa, na primjer float u int. Kada je izvorna vrijednost veća od maksimalne dopuštene odredišne vrijednosti, prave se greške od nekoliko redova veličine.

Proširujuća konverzija je kada se konvertuje objekat u tip koji može uključiti makar aproksimacije svih vrijednosti izvornog tipa, na primjer int u float. i ovdje su moguće greške konverzije, npr, ako se dodijeli cijeli broj 4023223221 float varijabli koja zauzima 4 bajta, tri najniže cifre rezultata će biti pogrešne.

Miješani režim

Izrazi miješanog režima imaju operande različitog tipa, na primjer

```
float a,b;
int c;
a=b+c;
```

Ovdje je uočljiva implicitna konverzija tipova. Ovaj pristup povećava upisivost i čitljivost, ali smanjuje pouzdanost, jer se smanjuje mogućnost otkrivanja grešaka tipova u kompajleru. U većini jezika, svi numerički tipovi koji se konvertuju u izrazima, koriste konverzije proširenja.

U Ada, izuzev kod stepenovanja nema implicitne konverzije u izrazima. Sljedeći primjer će izazvati grešku.

```
A : Integer;
B, C, D : Float;
...
C := B * A;
```

Eksplicitna konverzija tipova

U mnogim jezicima postoji mogućnost konverzije tipova u izrazima navodeći ime tipa u koji se konvertuje. Ove konverzije mogu biti proširujuće i sužujuće. U nekim slučajevima, prijavljue se upozorenje ako će sužujuća konverzija dovesti do značajnog gubitka opsega. Ove konverzije u C-baziranim jezicima se zovu casting.

Primjeri

C: (int)angle

Ada: Float (Sum)

Ada, ML i C# simaju sintaksu sličnu pozivu funkcije. Sintaksa u C je drugačija jer postoji viš tipova čija imena imaju dvije riječi (unsigned char).

Greške u izrazima

Pri izvršavanju aritmetičkih izraza mogu se desiti greške. Uzroci grešaka su greške u runtime **konverziji tipovs**, **prirodno ograničenje aritmetike**, npr dijeljenje s nulom ili logaritam negativnog broja i ograničenje kompjuterske aritmetike, npr prekoračenje (rezultat predstavlja više od najveće dopuštene vrijednosti) i podkoračenje (rezultat je ispod najmanje dopuštene vrijednosti). Neki izvršni sistemi prekidaju izvršenje programa u slučaju ovakvih grešaka, neki ih ignorišu, a neki imaju razrađen sistem obrade izuzetaka.

Relacioni i logički Izrazi

U okviru naredbi uslova koriste se relacioni izrazi koji se povezuju logičkim izrazima.

Relacioni Izrazi

Relacioni izrazi koriste relacione operatore i operande raznih tipova u cilju njihovog poređenja. Oni imaju dva operanda i operator koji ih poredi. Poređenja su na jednakost, nejednakost, a za tipove podataka gdje to ima smisla i poređenje na redoslijed. (<,>,<=,>=)

Simboli za relacione operatore se mogu ponekad razlikovati. Na primjer, operator različitosti može biti

!= (C), /= (Ada), ~= (Lua), .NE. (Fortran), <> (Pascal, BASIC, ML, F#), # (Modula 2)

Operatori za jednakost su obično =, == ili .EQ.

JavaScript i PHP su jezici s povezivanjem tipova s varijablama u trenutku izvršenja. imaju dva dodatna relaciona operatora, === i !== koji predstavljaju poređenje na jednakost vrijednosti i tipova nakon konverzije. Na primjer "7" == 7 je true u JavaScript, jer pri poređenju operanada relacionim operator, string se konvertuje u broj. Ipak, "7" === 7 je false, jer se ne vrši konverzija tipova. Ruby koristi == za jednakost s konverzijom ili eql? za jednakost bez konverzije.

Logički izrazi

Logički izrazi koriste logičke operatore i kao rezultat imaju logičku vrijednost. Operandi su logički i rezultat je logički Operacije su obično logičke AND, OR, NOT, ponegdje i XOR. Neke verzije BASICa imaju i IMP i EQU.

I ovi operatori se različito zovu u raznim programskim jezicima.

FORTRAN77	FORTRAN90	C	Ada
.AND.	and	&&	And
.OR.	or	||	Or
.NOT.	not	!	Not

C prije verzije 99 nema logički tip.. Koristise int tip sa 0 za netačno i ne-nula za tačno. Neobična karakteristika C izraza: a < b < c je ispravno ali daje neočekivan rezultat:.Lijevi operator se računa čineći 0 ili 1. Rezultat računa se poredi s trećim operandom (tj., c)

Kratkospojno izračunavanje

Ako je moguće računati izraz u kome se rezultat dobije bez računa svih operanada govorimo o kratkospojnom izračunavanju.

Primjer:

```
(13*a) * (b/13–1)
```

Ako je a=0, ne mora se računati (b/13-1) , jer je rezultat 0 bez obzira na b. Kratkospojno računanje se rijetko primjenjuje u aritmetičkiom izrazima, ali je uobičajeno u logičkim izrazima. Npr u sljedećem Pascal primjeru ne treba računati dio izraza b>5 ako je a bio manji od 5 jer će rezultat svakako biti netačan.

```
if (a>5) and (b>4) then writeln('Yes');
```

Problem s kratkospojnim računom nastaje ako se program zasniva na njemu, a kompajler ga nije implementirao. U sljedećem primjeru kada je index=length, LIST [index] će izazvati problem (ako LIST ima length -1 elemenata)

```
index = 1;
while (index <= length) && (LIST[index] != value)
index++;
```

C, C++, i Java: koriste kratkospojno računanje za obične logičke operatore (&& i ||), ali pružaju i bitovske logičke operatore koji nisu kratko spojeni (& |) .

Ruby, Perl, ML, F#, i Python koriste kratkospojne operatore. U BASIC-u operatori obično nisu kratkospojni.

Ada: programer može birati irmeđu nekratkospojnih (**and** i **or**) i (kratkospojnih je **and then** i **or else**)

Kratkospojeno izračunavanje predstavlja problem ako ima bočnih efekata u izrazima npr. (a > b) || (b++ / 3)

Naredbe dodjele

Opšta sintaksa naredbi kojima se dodjeljuje vrijednost varijabli je sljedeća.

```
<target_var> <assign_operator> <expression>
```

Npr.

```
a:=5+c;
```

Operator dodjele se razlikuje u programskim jezicima.

=	FORTRAN, BASIC,C-bazirani jezici
:=	ALGOL, Pascal, Ada
<-	Smalltalk
→	CASIO kalkulatori,
←	APL
TO	COBOL

U BASICu i Fortranu operator = se koristi i kao relacioni operator jednakosti. To otežava čitljivost i komplikuje kompajler. Zato C bazirani jezici koriste == kao relacioni operator.

U Ada, Pascal Fortran, BASIC dodjela je pojedinačna naredba. C bazirani jezici dopuštaju dodjelu unutar izraza.

Uslovna odredišta

Perl dopušta uslovna odredišta u dodjeljivanju.

```
($flag ? $total : $subtotal) = 0
```

Ekvivalentno s

```
if ($flag){
$total = 0
} else {
$subtotal = 0
}
```

Složeni operatori

Skraćeni metod za česte dodjele uveden ju ALGOL; prihvatio je C

Primjer

```
a = a + b
```

Može se kraće pisAti.

```
a += b
```

Unarni operatori dodjele u C-baziranim jezicima kombinuju inkrementiranje i dekrementiranje s dodjelom Primjeri unarnih operatora dodjele:

sum = ++count (count uvećan pa dodijeljen u sum)

sum = count++ (count dodijeljen pa uvećan u sum)

count++ (count uvećan)

-count++ (count uvećan pa negiran)

Dodjela kao izraz

U C baziranim jezicima, naredbe dodjele imaju rezultat koji se može koristiti kao operand unutar izraza. U sljedećem primjeru ch = getchar() se uzme, rezultat (dodijeljen ch) koristi kao uslov za while naredbu

```
while ((ch = getchar())!= EOF){…}
```

Dodjela listom

Perl i Ruby podržavaju dodjelu listom. Sljedeći primjer dodjeljuje istovremeno tri varijable.

```
($first, $second, $third) = (20, 30, 40);
```

Miješana dodjela

Naredbe dodjele mogu biti u miješanom režimu. Ako odredišna varijabla nije istog tipa kao izraz koji joj se dodjeljuje, postavlja se pitanje kakva konverzija će se obaviti.

U Fortran, C, i C++, svaka vrijednost numeričkog tipa se može dodijeliti varijabli numeričkog tipa .

U Pascal integer se može dodijeliti real, ne i obrnuto.

U Java i C#, dopušteno je samo prisilno dodjeljivanje koje proširuje opseg

U Ada, nije dopuštena prisilna dodjela.

Dodjela u funkcionalnim jezicima

Varijable u funkcionalnim jezicima su samo imena vrijednosti koja se nikad ne mijenjaju. Tako u ML

val cost = quantity * price;

kreira uvijek novu varijablu cost koja nema veze s prethodnim.

Rezime

Aritmetički izrazi sastoje se od operatora, operanada, zagradi i poziva funkcija . Operatori se računaju onim redoslijedom koji definiše njihov prioritet i asocijativnost . Preopterećenje operatora predstavlja višestruku upotrebu istog operatora nad različitim tipovima. Izrazi mogu biti miješanog tipa. Pored proste dodjele, moguće su i dodjele složenim operatorima.

8. Kontrolne strukture i naredbe

Kontrola toka govori o redoslijedu izvršavanja pojedinih dijelova programa, unutar izraza, između naredbi programa i između jedinica programa. Unutar izraza ona je definisana kroz prioritet i asocijativnost operatora, te kratkospojno izračunavanje. O kontroli toka između jedinica programa biće riječi u poglavljima o podprogramima i konkurentnosti. Sada će se nešto reći o kontroli toka između naredbi programa.

Kontrolne naredbe FORTRAN-u su bazirane direktno na hardveru IBM 704, (bezuslovni skokovi). Rastom složenosti programa pokazale su se neadekvatnim. Mnogo istraživanja i argumentacija je vršeno tokom 1960-ih oko problema čitljivosti i skalabilnosti. Važan rezultat istraživanja: Dokazano da se svi algoritmi predstavljeni dijagramom toka mogu kodirati dvosmijernim izborom i petljom s kontrolom uslova na početku.

Kontrolna struktura predstavlja naredbu kontrole i skup naredbi čije izvršenje ona kontroliše. Na primjer, to je naredba while u C uključujući test da li je petlja završena i sve naredbe unutar petlje. Neki jezici dopuštaju da kontrolna struktura ima više ulaza. To smanjuje čitljivost programa uz malo povećanje fleksibilnosti

Naredbe izbora

Naredba izbora pruža način izbora između dvije ili više grane izvršenja. Kada program u svom izvršenju dođe do ovakve naredbe, zavisno od izračunatog uslova nastavlja dalje nekom granom izvršenja. Ove naredbe se mogu podijeliti u dvije osnovne kategorije: izbor između dva puta i izbor između više puteva .

Izbor između dvije opcije

Opšti oblik izbora između dvije opcije izgleda ovako:

```
if control_expression
then clause
else clause
```

Iako je kod većine imperativnih jezika prihvaćen ovakav oblik naredbe uslova, postoje sljedeći problemi dizajna:

- Koja je forma i tip kontrolnog izraza?
- Kako se navode then i else klauzule?
- Kako će se navesti značenje gniježdenih izbora?

Kontrolni izraz predstavlja izraz logičkog (Boolean) tipa, ali u C89, C99, BASIC, Python, i C++, kontrolni izraz može biti aritmetički. Ako se ne stavlja then ili neki drugi sintaksni marker za then klauzulu, kontrolni izraz se stavlja u zagrade .U jezicima poput Ada, Java, Ruby, i C#, kontrolni izraz mora biti Boolean

Oblik klauzula: U mnogim današnjim jezicima (C, Pascal, C++, Java, C#) then i else klauzule mogu biti jedna naredba ili složena naredba sastavljena od više naredbi smještenih unutar vitičastih zagrada ili begin/end sekvence.

```
if (a==2)
   caller(2) ;
else {
   m=80;
   prepare(a);
 }
```

ili u Pascal-u

```
if a=2 then
   caller(2)
else begin
   m:=80;
   prepare(a);
end;
```

U Perl, sve klauzule se moraju razdvojiti vitičastim zagradama (moraju biti složene)

U Fortran 95, Ada, Modula 2, moderni BASIC i Ruby, klauzule su sekvence naredbi koje se završavaju sa jednom ili dvije rezervisane riječi . Ruby:

```
if count == 0 then
   result = 0
else
   result = 1
end
```

Python koristi uvlačenje da označi tijelo if naredbe

```
if x > y :
   x = y
   print "case 1"
```

Jezici koji dopuštaju varijantu naredbe if bez else klauzule, a da se klauzule sastoje od proste ili složene naredbe imaju problem **dvoznačnog else** zbog dvoznačnosti gramatike. Sljedeći Java primjer to ilustruje..

```
if (sum == 0)
if (count == 0)
result = 0;
else result = 1;
```

Koji if je vezan uz else, tj. da li je prethodni primjer ekvivalentatn s

```
if (sum == 0)
  {
     if (count == 0)
        result = 0;
        else result = 1;
  }
```

ili

```
if (sum == 0)
  if (count == 0) {
    result = 0;
    else result = 1;
```

U većini jezika kod kojih se javlja ovaj problem uvedeno je statičko semantičko pravilo: else odgovara najbližem if .

Perl zahtijeva da se sve then i else klauzule predstave složenim naredbama , Stoga kod njega ne postoji problem dvoznačnogg else.

```
if ($sum == 0) {
  if ($count == 0) {
      result = 0;
   }
   else  {
      $result = 1;
   }
```

Ovaj problem se ne javlja ni u jeziku Ruby zbog obavezne ključne riječi end na kraju if naredbe.

```
if sum == 0 then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Problema nema ni u Pythonu koji pripadnost pojedinoj klauzuli određuje uvlačenjem.

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```

U funkcionalnim jezicima selektor nije naredba, nego funkcija. U sljedećem primjeru u Lisp, varijabli a će se dodijeliti vrijednost drugog argumenta funkcije IF ako se prvi evaluirao tačno, a drugog ako se prvi evaluirao netačno.

```
(setq a (IF (<= n 1) 15 20))
```

Primjer u F#

```
let y =
    if x > 0 then x
    else 2 * x;;
```

Naredbe trostrukog izbora

U starijim verzijama Fortran-a nudi mogućnost grananja u tri pravca, zavisno od vrijednosti kontrolnog izraza. Aritmetička if naredba će izazvati skok na jednu od tri navedene labele zavisno od toga da li je vrijednost kontrolnog izraza negativna, nula ili pozitivna.

```
 IF (A+3) 10,100,200
```

Naredbe višestrukog izbora

Naredbe višestrukog izbora dopuštaju izbor izvršenja jedne ili više naredbi ili grupa naredbi, na bazi vrijednosti aritmetičkog izraza. Kako se mogu realizovati na više načina nego naredbe izbora između dvije opcije, javlja se i više problema dizajna:

- Koji je oblik i vrsta kontrolnog izraza?
- Kako se odabira izbirljivi segment?

- Da li je tok kroz strukturu ograničen na samo jedan izabirljivi segment?
- Kako se navode vrijednosti za svaki slučaj?
- Šta se radi s izrazom čija vrijednost nije predstavljena?

C, C++, i Java za višestruki izbor koriste switch naredbu, čiji je opšti oblik dat u sljedećem primjeru.

```
switch (expression) {
case const_expr_1: stmt_1;
…
case const_expr_n: stmt_n;
[default: stmt_n+1]
}
```

U C switch naredbi kontrolni izraz je samo cjelobrojni tip. Segmenti za izbor mogu biti sekvence naredbi, blokovi ili složene naredbe Kada je expression jednak nekoj od konstanti, program se nastavlja od case naredbe uz koju je navedena odgovarajuća konstanta. Za neprepoznate vrijednosti koristi se default klauzula (ako nema default, cijela naredba ne radi ništa). Nema implicitnog grananja na kraju izabirljivih segmenata, što pravi problem s pouzdanošću. U sljedećem primjeru je zaboravljena naredba break prije case 2: pa se parni brojevi broje i kada je parametar neparan.

```
switch (index) {
case 1:
case 3: odd += 1;
   sumodd += index;
case 2:
case 4: even += 1;
      sumeven += index;
default: printf("Error in switch, index = %d\n", index);
}
```

Ispravan primjer bi zahtijevao naredbu break na kraju svake case sekcije koji će izazvati skok na kraj case naredbe.

```
switch (index) {
case 1:
case 3: odd += 1;
   sumodd += index;
   break;
case 2:
case 4: even += 1;
      sumeven += index;
      break;
default: printf("Error in switch, index = %d\n", index);
}
```

C# se razlikuje od C po tome što ima statičko semantičko pravilo koje zabranjuje implicitno izvršenje više od jednog segmenta. Svaki izabirljivi segment mora završiti bezuslovnim grananjem (goto ili break). Takođe u C# kontrolni izraz i case konstante mogu biti stringovi

PHP ima istu sintaksu naredbe switch kao C, uz dodatnu mogućnost da kontrolni izraz i case konstante mogu biti string, cijeli broj ili broj u pokretnom zarezu.

Za razliku od jezika sličnih C, u Pascal i Ada naredba case koja služi za višestruki izbor ima semantičko pravilo da se po završetku sekvence , kontrola se prosljeđuje prvoj

naredbi nakon case naredbe. To je pouzdanije od switch naredbe u C. Sintaksa ove naredbe u Ada je sljedeća.

```
case expression is
when choice list => stmt_sequence;
…
when choice list => stmt_sequence;
when others => stmt_sequence;]
end case;
```

U Ada izrazi mogu biti bilo kog ordinalnog tipa, segmenti se mogu sastojati od jedne naredbe ili biti složeni, samo jedan segment se može izvršiti po izvršenju ili konstrukciji i nisu dopuštene ne-predstavljene vrijednosti. Lista izbora je lista konstanti koja može uključiti pod-opsege i Boolean OR operator.

Ruby ima dva oblika case naredbi koje se mogu nalaziti i kao naredbe i kao aritmetički izraz

1. Jedan oblik sa when uslovima, koji je kao višestruki if

```
leap = case
  when year % 400 == 0 then true
  when year % 100 == 0 then false
  else year % 4 == 0
end
```

2.Drugi koristi case vrijednost i when vrijednosti

```
case in_val
   when -1 then neg_count++
   when 0 then zero_count++
   when 1 then pos_count++
   else puts "Error – in_val is out of range"
end
```

Sračunati bezuslovni skok

Stariji programski jezici pružaju fukcionalnost višestrukog izbora kroz sračunati i dodijeljeni bezuslovni skok. Sračunati bezuslovni skok sadrži listu labela na koje se skače zavisno od aritmetičkog izraza.

Ovaj primjer u Fortran IV će skočiti na labelu 12 ako je INDEX =1, na 24 ako je INDEX=2 i na labelu 36 ako je INDEX=3

```
 GO TO (12,24,36), INDEX
```

Slična naredba primjer postoji i u Microsoft BASIC:

ili

```
ON A GOTO 10,20,300
```

Višestruki izbor pomoću if

Višestruki izbori se mogu pojaviti kao direktno proširenje izbora između dvije mogućnosti, koristeći else-if klauzule npr u Python:

```
if count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

Python primjer napisan kao case naredba u Ruby

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```

Ovakav način izbora javlja se u Scheme verziji Lisp-a, kroz funkciju COND.

```
(COND
(predicate1 expression1)
(predicate2 expression2)
...
(predicaten expressionn)
[(ELSE expressionn+1)]
)
```

Na primjer

```
COND
((> x y) "x is greater than y")
((< x y) "y is greater than x")
(ELSE "x and y are equal")
)
```

Iterativne naredbe

Ponovljeno izvršenje naredbe ili složene naredbe postiže se iteracijom ili rekurzijom. Iterativna naredba se zove i petlja. Osnovna pitanja dizajna za iteracijske kontrolne naredbe su:

1. Kako se kontroliše iteracija (brojanjem ili logičkim uslovom)?

2. Gdje su kontrolni mehanizmi u petlji koji određuju da li će se nastaviti ponavljanje (na početku, kraju ili u sredini)?

Brojačke petlje

Brojačka iterativna naredba ima varijablu petlje i značenje koje specificiraju početnu, krajnju i vrijednost koraka. Varijabla petlje se uvećava pri svakom prolasku kroz petlju za vrijednost koraka i provjerava da li je dostigla potrebnu vrijednost. Pitanja dizajna brojačkih petlji su:

- Koji su tip i opseg varijable petlje?
- Da li je legalno za varijablu petlje ili parametre petlje da se mijenjaju u tijelu petlje i kako to utiče na kontrolu petlje?
- Trebaju li se parametri petlje računati samo jednom ili pri svakoj iteraciji?

Fortran je prvi jezik koji je uveo brojačke petlje. Opšti oblik je sljedeći:

```
DO label var = start, finish [, stepsize]
```

npr, sljedeći primjer će dodijeliti vrijednosti varijablama a(1), b(1), a(3), b(3), a(5) i b(5).

```
   do 20,i=1,5,2
   a(i)=5
20 b(i)=8
```

U slučaju Fortran-a, odgovor na pitanja dizajna je sljedeći:

1. Varijabla petlje mora biti INTEGER. Korak može biti bilo koja vrijednost osim nule .Parametri mogu biti izrazi

2. Varijabla petlje se ne može mijenjati u petlji, ali parametri mogu, pošto se računaju samo jednom, to ne utiče na kontrolu petlje

3. Parametri petlje se računaju samo jednom

Od FORTRAN 95 uveden je drugi oblik koji izbjegava numeričku labelu.:

```
[name:] Do variable = initial, terminal [,stepsize]
…
End Do [name]
```

Ada takođe pruža brojačku petlju.

```
for var in [reverse] discrete_range loop ...
end loop
```

Petlja for u Ada dizajnirana je s sljedećim elementima:

- Tip varijable petlje mora biti diskretnog opsega (Diskretni opseg je podopseg cijelog ili enumeracijski tip).
- Varijabla petlje ne postoji izvan petlje.
- Varijabla petlje se ne može mijenjati u petlji, ali diskretni opseg može, to ne utiče na kontrolu petlje.
- Diskretni opseg se računa samo jednom.
- Ne može se granati u tijelo petlje .

Primjer

```
Count : Float := 1.35;
for Count in 1..10 loop
Sum := Sum + Count;
end loop;
```

U BASIC se koriste ključne riječi FOR i NEXT za zaokruživanje petlje. Naredba FOR može imati korak naveden naredbom STEP.

Petlja for u Pascal ima samo korake 1 i -1 navedene kroz klauzule to i downto.

Petlja s tri izraza

U C baziranim jezicima funkcionalnost brojačke petlje se postiže naredbom for, koja je u suštini logički kontrolisana petlja, ali se koristi najčešće u primjenama brojačke petlje. Naredba for ima sljedeći opšti oblik:

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

Izračuna se najprije prvi izraz, i dok je drugi izraz zadovoljen, izvršavaju se naredbe u tijelu petlje. Na kraju svake iteracije izvršava se treći izraz.

Zbog fleksibilnosti izraza u C, izrazi mogu biti cijele naredbe dodjele, pa čak i sekvence naredbi dodjele razdvojene zarezima. Vrijednost višenaredbenog izraza je vrijednost zadnje naredbe u izrazu. Ako je drugi izraz otsutan, to je beskonačna petlja. Odgovor na pitanja dizajna:

- Nema eksplicitne varijable petlje
- Sve se može promijeniti u petlji
- Prvi izraz se računa jednom, a druga dva pri svakoj iteraciji

C++ se razlikuje od C po tome što kontrolni izraz **može** biti bool tipa, a početni izraz može uključiti definicije varijabli (opseg je od definicije do kraja tijela petlje)

U Java i C# što kontrolni izraz **mora** biti Boolean

Petlja fiksnog broja

Podvarijanta brojačke petlje u kojoj se ne koristi brojačka varijabla nego se definiše samo koliko puta se treba petlja ponoviti javlja se u jeziku LOGO.

REPEAT 20 [FORWARD 400 LEFT 45]

Logički kontrolisana petlja

U mnogim slučajevima, skup naredbi mora biti više puta izvršen, ali ponavljajuća kontrola se temelji na logičkom izrazu, a ne suprotno. Za ove situacije, logički kontrolisane petlje su pogodna. Zapravo, logički kontrolisane petlje su generalnije od brojačkih petlji, jer se svaka brojačka petlja može izgraditi od logičke petlje, ali ne i obrnuto. Logički kontrolisane petlje imaju manje pitanja dizajna:

- Da li uslov kontrole petlje testirati na početku ili na kraju petlje (Predtest ili posttest)?
- Treba li logički kontrolisana petlja biti specijalni oblik brojačke petlje ili posebna naredba?

C i C++ imaju predtest i posttest oblike u kojima kontrolni izrazi mogu biti aritmetički:

```
while (ctrl_expr)
loop body
```

i

```
 do
loop body
while (ctrl_expr)
```

Npr. u c# primjer za obije petlje.

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}
value = Int32.Parse(Console.ReadLine());
do {
  value /= 10;
  digits ++;
} while (value > 0);
```

Java je kao C i C++, osim što kontrolni izraz mora biti bool tipa i u tijelo se mora ući samo na početku -- Java nema goto

Ada ima predtest verziju, ali ne posttest , posttest se postiže sa iskokom u sredini.

```
While_Loop :
   while X <= 5 loop
      X := Calculate_Something;
   end loop While_Loop;
```

FORTRAN prije verzije 90 nema logičku petlju, u Fortran 90 je uvedena predtest petlja.

```
  do while (counter > 0)
    factorial = factorial * counter
    counter = counter - 1
  end do
```

Pascal za predtest koristi while ..do petlju a za posttest repeat ... until. Petlja repeat until se ponavlja ako uslov nije ispunjen.

Perl i Ruby imaju dvije predtest logičke petlje, while i until i dvije posttest petlje. Primjer sve četiri petlje u Perl:

```
while(condition)
{
   statement(s);
}
until(condition)
{
   statement(s);
}
do
{
   statement(s);
}while( condition );
do
{
   statement(s);
}until( condition );
```

Sve četiri petlje u Ruby

```
while conditional do
   code
end
until conditional do
   code
end
begin
   code
end until conditional
begin
   code
end while conditional
```

Korisnički mehanizmi kontrole petlji

Nekad je praktično da programeri odaberu lokaciju za kontrolu petlje (ne samo na vrhu ili dnu petlje). Stoga u nekim jezicima postoji način za iskakanje iz petlje. Sintaksa za izlaz je Jednostavna ako je u pitanju jedna petlja (npr., break). Problemi dizajna za naredbe za iskakanje iz petlje

- Treba li uslov biti dio naredbe iskakanja?
- Treba li se kontrola prenijeti na više od jedne petlje?

C , C++, Python, Ruby, i C# imaju bezuslovne izlaze iz petlji za koje se ne navode labele(break)

Java i Perl imaju bezuslovne izlaze s labelama (break u Java, last u Perl)

C, C++, i Python imaju bezlabelsku kontrolnu naredbu, continue, koja preskače ostatak treutne iteracije ali ne izlazi iz petlje

Java i Perl imaju labelirane verzije continue

Primjer labele u break (Java)

```
search:
  for (i = 0; i < arrayOfInts.length; i++) {
    for (j = 0; j < arrayOfInts[i].length; j++) {
      if (arrayOfInts[i][j] == searchfor) {
        foundIt = true;
        break search;
      }
    }
  }
```

U PHP break i continue imaju dodatni opcionalni numerički argument koji određuje koliko nivoa petlji se iskače ili vraća na početak.

U Ada uslov je sastavni dio naredbe iskakanja iz petlje.

```
Exit_Loop :
   loop
      X := Calculate_Something;
      exit Exit_Loop when X > 5;
      Do_Something (X);
   end loop  Exit_Loop;
```

Iteracija bazirana na strukturi podataka

Mana brojačke petlje je što omogućuje da kontrolna varijabla mijenja vrijednosti samo kroz ekvidistantne sekvencijalne vrijednosti opsega cijelih brojeva. Stoga neki jezici omogućavaju da se kroz određenu strukturu podataka definiše skup vrijednosti koje će kontrolna varijabla imati. To je posebno značajno u objektno orijentisanim jezicima koji omogućavaju kolekcije podataka apstraktnog tipa. Broj elemenata u strukturi podataka kontroliše iteraciju petlje. Kontrolni mehanizam je poziv iteratorske funkcije koja vraća naredni element u nekom izabranom redoslijedu, ako ga ima, inače se petlja prekida.

Na primjer, u C baziranim jezicima, for se može koristiti za korisnički definisan iterator:

```
for (p=root; p==NULL; traverse(p)){
}
```

PHP za prolazak kroz nizove u petljama posjeduje iteratorske funkcije current (pokazuje na jedan element niza),next (pomijera current na naredni element niza) i reset (pomijera current na prvi element niza)

```
reset $list;
print ("First number: " + current($list) + "<br />");
while ($current_value = next($list))
  print ("Next number: " + $current_value + "<br \>");
```

Naredba foreach, je jednostavniji način za prolazak kroz sve elemente niza. Za svaku iteraciju petlje, jedan od elemenata niza navedenog kao parametar se dodjeljuje varijabli

```
 <?php
$colors = array("red", "green", "blue", "yellow");
foreach ($colors as $value) {
    echo "$value <br>";
}
?>
```

U Java za iteriranje for naredbom kroz svaku kolekciju koja realizuje Iterator interface može se koristiti metoda next koja pomijera pokazivač u kolekciji. Od Java 5.0 uvedena je nova sintaksa naredbe for koja se popularno zove foreach, za prolazak kroz nizove i druge klase koje implementiraju Iterable interfejs npr., ArrayList

```
for (String myElement : myList) { … }
```

C# foreach naredba kreće kroz elemente niza i drugih kolekcija:

```
Strings[] = strList = {"Bob", "Carol", "Ted"};
foreach (Strings name in strList)
  Console.WriteLine ("Name: {0}", name);
```

- Notacija {0} indicira poziciju stringa koji se prikazuje

U Ada se može iterirati kroz sve elemente podintervalnog tipa.

```
subtype MyRange is Integer range 0..99;
MyArray: array (MyRange) of Integer;
for Index in MyRange loop
…
end loop;
```

Lua ima dva oblika iterativnih naredbi, jedna kao Fortran Do, i opštiji oblik:

```
for variable_1 [, variable_2] in iterator(table) do
…
end
```

Najčešće korišteni iteratori su pairs i ipairs

U Python for petlja je isključivo za iteriranje kroz strukture podataka, ali izborom odgovarajuće strukture postiže se brojačka petlja.

```
for loop_variable in object:
- loop body
[else:
- else clause]
```

Na primjer

```
for count in range(1, 11):
    print(count)
```

Object je opseg koji je lista vrijednostiu u zagradama ([2, 4, 6]), ili poziv funkcije range (range(2,7), što vraća 2, 3, 4, 5, 6. Varijabla petlje uzima vrijednosti u datom opsegu, po jednu za svaku iteraciju. Klauzula else, koja je opcionalna, se izvrši ako se petlja normalno završava.

Ruby poznaje blokove, sekvence koda omeđene ključnim riječima do...end ili vitičastim zagradama. Oni se mogu proslijediti kao parametar iteratorskim metodama times (poziva blok onoliko puta kolika je numerička vrijednost objekta) i each(poziva blok za svaki element) i upto (ponavlja blok brojanjem). Sljedeći primjer će ispisati 4 puta "Hey", zatim sve elemente niza list i na kraju brojeve od 1 do 5.

```
 4.times {puts "Hey!"}
 list = [2, 4, 6, 8]
 list.each {|value| puts value}
1.upto(5) {|x| print x, " "}
```

Bezuslovna grananja

Bezuslovni skok prebacuje izvršenje programa na navedeno mjesto u programu. Komanda je principijelno jednostavna, ali komplikuje generalnu strukturu programa. Najčešće se zove goto. Njena primjena i uključenje u jezike predstavljala je jednu od najžešćih debata u 1960im i 1970im . Jezici koji poznaju goto naredbu (Fortran, C, C++, C#, Pascal, COBOL, BASIC) uvode odgovarajući način deklaracije labele, dopuštenih mjesta za skokove, koja može biti predstavljena brojem ili tekstom. U ranijim verzijama BASIC-a, svaka linija programa je trebala imati labelu (linijski broj). Neke verzije BASIC-a imaju mogućnost skoka na bilo koju liniju izračunatu izrazom, što čini kod posebno nepredvidljivim.

```
GOTO A*3
```

Neki jezici ne podržavaju goto naredbu (npr., Java, Modula 2, Python, Ruby) ali tada obično dodaju naredbe za iskok iz petlje, što je najčešća korisna primjena naredbe goto.

Čuvane naredbe

Konceptualni jezik GCL koga je dizajnirao Dijkstra namijenjen je za podršku novim programerskim metodologijama za verifikaciju (korektnost) tokom razvoja. Bio je osnova za dva lingvistička mehanizma za konkurentno programiranje(u CSP i Ada), a implementirane u verifikacijskom jeziku Promela. Osnovna ideja je: ako redoslijed računa nije važan, program ga ne treba navoditi

Čuvane naredbe izbora

Oblik čuvanih naredbi izbora je sljedeći. Svaka naredba u liniji izbora ima Bulov izraz, čuvar.

```
if <Boolean exp> -> <statement>
[] <Boolean exp> -> <statement>
...
[] <Boolean exp> -> <statement>
fi
```

Značenje ovakve naredbe je da kada se dođe do if konstrukcije, računaj sve Bulove izraze. Ako je više od jednog tačno, odaberi neki nedeterministički. Ako nijedan nije tačan, to je izvršna greška.

Čuvana petlja

Forma

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
...
[] <Boolean> -> <statement>
od
```

Značenje je da za svaku iteraciju treba izračunati sve Boolean izraze. Ako je više od jednog izraza tačno, odaberi neki nedeterministički pa počni ponovo. Ako nije ni jedan, izađi iz petlje

Ovako definisane kontrolne strukture pojednostavljuju verifikaciju programa, jer u if naredbama programer je natjeran da treba razmotriti sve slučajeve. Na primjer:

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + I
fi
```

Ako je If i = 0 and j > i, program se ponaša nedeterministički. Ako je i=j a oba su različita od 0, program prijavljuje grešku

Zaključak

Postoji više naredbi za kontrolu toka. Svaki program se može napisati koristeći naredbu izbora i petlje sa testom na početku. Dodavanje novih naredbi je kompromis između veličine jezika i upisivosti. Funkcionalni i logički programski jezici imaju dosta drugačije kontrolne strukture

9. Potprogrami

Potprogrami omogućavaju da se dio programa koji se nalazi na jednom mjestu koristi i na drugim mjestima u programu. Podprogrami, osim korutina, imaju sljedeće tri osobine.

- Svaki potprogram ima jednu ulaznu tačku
- Pozivajući program je zaustavljen tokom izvršenja pozvanog potprograma
- Kontrola se uvijek vraća pozivaocu kada se završi pozvani potprogram

Parametri

Definicije i deklaracije potprograma

U većini programskih jezika definicija potprograma se ne izvršava, nego samo opisuje opisuje interfejs kao i akcije apstrakcije potprograma.

Python i Lisp su izuzeci. U ovim jezicima funkcije se definišu naredbom definicije funkcija koje se prethodno izvršavaju. Na primjer u Python sljedeći dio koda će definisati jednu od dvije različite definicije iste funkcije, zavisno od postavljenog uslova:

```
if . . .
  def fun(. . .):
    . . .
else
  def fun(. . .):
    . . .):
```

Poziv potprograma je zahtjev za njegovim izvršenjem . U nekim jezicima postoji posebna ključna riječ za poziv potprograma. Kada se potprogram pozove, a prije nego je završio, kaže se da je **aktivan**. Zaglavlje potprograma je prvi dio definicije i uključuje ime, vrstu potprograma i formalne parametre

Parametarski profil (tj signatura) potprograma čine broj, redoslijed i tipovi njegovih parametara. Protokol potprograma je parametarski profil potprograma, plus povratni tip funkcije.

Dok definicija potprograma sadrži zaglavlje i tijelo, izvršni kod potprograma, deklaracije potprograma pružaju protokol, ali ne tijelo potprograma . Deklaracije funkcija u C i C++ se često zovu prototipovi. Deklaracije su neophodne u programskim jezicima koji ne dopuštaju reference unaprijed.

Slaganje formalnih i stvarnih parametara

Unutar deklaracije potprograma navedeni su formalni parametri. Formalni parametar je pomoćna varijabla listana u zaglavlju potprograma i korištena u njemu Stvarni parametar predstavlja vrijednost ili adresu koja se koristi tokom naredbe poziva potprograma. Kako se povezuju stvarni i formalni parametri.

Poziciono vezivanje, postoji u većini programskih jezika. Povezivanje stvarnih parametara na formalne je po poziciji: prvi stvarni parametar se poveže na prvi formalni parametar i tako dalje. Ovo je bezbjedno i efikasno ako je lista parametara relativno kratka.

Kod **vezivanja po ključnim riječima** ime formalnog parametra na koji se veže stvarni parametar je navedeno s stvarnim parametrom. Prednost je što se parametri mogu javljati u bilo kom redoslijedu, izbjegavajući greške neusaglašenosti parametara. Mana je što korisnik mora znati i navoditi imena formalnih parametara. Npr. u Python, potprogram se može pozvati i na sljedeći način.

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

Ada, Fortran 95+, Visual BASIC i Python omogućavaju miješanje pozicionog i vezivanja po ključnim riječima.

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

Podrazumijevane vrijednosti formalnih parametara

U nekim jezicima (C++, Python, Ruby, Ada, PHP), formalni parametri mogu imati podrazumijevane vrijednosti (ako nisu stvarni parametri proslijeđeni). U C++, podrazumijevani parametri se moraju pojaviti zadnji, jer su parametri vezani za poziciju. U sljedećem primjeru

```
float compute_pay(float income, float tax_rate,
  int exemptions = 1)
```

poziv kao

```
pay = compute_pay(20000.0, 0.15);
```

Promjenjiv broj parametara

U većini jezika koji nemaju podrazumijevane parametre (Fortran, Pascal) broj stvarnih parametara u pozivu mora odgovarati broju formalnih parametara u deklaraciji ili definiciji. Ali u C, C++, Perl, JavaScript, i Lua to nije obavezno. Na programeru je da tada osigura usaglašenost parametara. Ova osobina jezika nije bezbjedna, ali može biti praktična.

U C se funkcije koje imaju promjenjiv broj prametara (printf) deklarišu koristeći tri tačke

```
void print(char * format,...);
```

U ovako deklarisanoj funkciji, proslijeđenim argumentima se pristupa kroz niz tipa va_list. On se najprije otvori sa va_start, makro instrukcijom, zatim redom jedan po jedan argument preuzima sa va_arg i na kraju zatvori očitavanje parametara sa va_end.

C# metode mogu primiti promjenjiv broj parametara ako su istog tipa—odgovarajući formalni parametar je niz params. Npr. neka je unuitar klase Myclass definisana metoda

```
public void DisplayList(params int[] list) {
   foreach (int next in list) {
      Console.WriteLine("Next value {0}", next);
   }
}
```

i deklarisan objekt

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

Tada se ova metoda može pozvati na dva sljedeća načina

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

U Ruby, prvih nekoliko parametara koji su prosti tipovi navedeni u aktualnim parametrima, dodjeljuju se po poziciji formalnim parametrima. Svi parametri koji predstavljaju hash mapiranje dodjeljuju se jednom parametru koji predstavlja anonimni hash. Posljednji parametar može biti niz označen zvjezdicomi njegovi elementi se dodjeljuju preostalim formalnim parametrima osim zadnjeg a ostatak parametara se dodijeli zadnjem formalnom parametru kao niz. U ovom primjeri unutar tester funkcije p1 će biti 'first', p2= {mon => 72, tue => 68, wed => 59}, p3=2 a p4=[4,6,8]

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  . . .
end
. . .
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

U Python, stvarni parametar je lista vrijednosti a odgovarajući formalni je ime s zvjezdicom .

```
def manyArgs(*arg):
  print "I received", len(arg), "arguments:", arg
>>> manyArgs(1)
I received 1 arguments: (1,)
>>> manyArgs(1, 2,3)
I received 3 arguments: (1, 2, 3)
```

U Lua, varijabilni broj parametara je predstavljen formalnim parametrom s tri tačke, pristupa im se for naredbom ili višestrukom dodjelom od tri tačke Ovdje je ipairs iterator kroz niz proslijeđenih parametara.

```
function multiply (. . .)
  local product = 1
  for i, next in ipairs{. . .} do
    product = product * next
  end
  return product
end
```

Procedure i funkcije

Postoje dvije kategorije potprograma. Procedure su skup naredbi koje obavljaju određeni račun. Efekat njihovog poziva je kao da je dodana nova naredba. U jezicima koji nemaju odvojene sintaksne mogućnosti za procedure i funkcije (većina jezika nastalih od C) procedure se definišu kao funkcije koje ne vraćaju vrijednosti. U jezicima koji ih podržavaju (Ada, Pascal, Fortran) zovu se procedure ili subroutine.

Funkcije strukturalno liče na procedure, ali semantički odgovaraju matematičkim funkcijama. Funkcije vraćaju vrijednosti, a procedure ne. Od njih se ne očekuju bočni

efekti, ne trebaju da mijenjaju nikakve druge podatke, osim što izračunaju i vrate vrijednost. U praksi ipak, mnoge funkcije imaju bočne efekte

Problemi dizajna potprograma

U dizajnu potprograma postavljaju se sljedeća pitanja

- Da li su lokalne varijable statičke ili dinamičke?
- Mogu li se definicije potprograma javljati u drugim definicijama potprograma?
- Koji metodi prosljeđivanja parametara su dati?
- Da li se provjeravaju tipovi parametara?
- Ako se potprogrami prosljeđuju kao parametri i mogu se gnijezditi, koje je referencirajuće okruženje proslijeđenog potprograma?
- Mogu li se potprogrami preopteretiti?
- Mogu li potprogrami biti generički?

Lokalno okruženje referenciranja

Potprogram može imati svoje lokalne varijable i sadržati druge potprograme. Opseg važenja ovih varijabli i potprograma je unutar opsega važenja potprograma.

Lokalne varijable mogu biti stek dinamičke ili statičke. Prednost stek dimaničkih lokalnih varijabli je podrška rekurziji i što se smještaj za lokalne varijable dijeli između potprograma. Mane su što je potrebno vrijeme za alokaciju i dealokacija i vrijeme inicijalizacije, što se takvim memorijskim lokacijama mora indirektno pristupati i što se ne može se pamtiti istorija. Ako su lokalne varijable statičke , tada su prednosti i mane suprotne u odnosu na stek dinamičke lokalne varijable.

U većini modernih programskih jezika lokalne varijable su stek dinamičke. U C i C++ mogu se opcionalno deklarisati da budu statičke, ključnom rječju static. Metode u C++, Java, i C# mogu imati samo stek dinamičke varijable. U Fortran-u ako funkcija nije deklarisana da podržava rekurziju, njegove lokalne varijable su obično statičke.

Jezici koji ne zahtijevaju obavezno deklarisanje varijabli se mogu međusobno razlikovati kada je potrebno deklarisati varijable. Lua i Fortran varijable smatraju globalnim ako nisu deklarisane, specifično unutar potprograma. Suprotno ovome, Python i JavaScript varijable zahtijevaju da se globalne varijable deklarišu unutar potprograma ako se koriste u njemu.

Gniježdeni potprogrami su uvedeni s Algolom i dugo su podržani jedino u Pascal, Algol, Ada i Modula 2, dok jezici nastali od C i Fortran nisu dopuštali deklaraciju potprograma unutar potprograma. U novije vrijeme JavaScript, Python, Ruby, i Lua dopuštaju i gniježdenje potprograma.

Modeli prosljeđivanja parametara

Parametri se mogu proslijediti potprogramu na više načina, zavisno od toga da li se koriste za prosljeđivanje vrijednosti potprogramu ili čitanju iz njega, te načina kako se kopiraju u potprogram.

Semantički model prosljeđivanja parametara

Formalni parametri se organizuju po jednom od tri različita semantička modela.

1) Ulazni režim: Mogu primati podatke od odgovarajućeg stvarnog parametra

2) Izlazni režim: Mogu prosljediti podatke stvarnom parametru

3) Ulazno-izlazni režim: Mogu primati i prosljeđivati podatke.

Uzeće se za primjer tri potprograma koji kao parametar koriste jedan niz cijelih brojeva. Prvi potprogram računa sumu elemenata niza i takvom potprogramu parametar treba proslijediti u ulaznom režimu. Drugi potprogram uvećava sve njegove elemente za 1, i njemu se prosljeđuje u ulazno/izlaznom režimu. Treći potprogram kreira novi niz, koji nema istorije, pa se takav parametar prosljeđuje u izlaznom režimu.

Konceptualni modeli prijenosa parametara

Parametri se mogu proslijediti na dva osnovna načina: fizičko premještanje vrijednosti i premještanje puta pristupa. U prvom slučaju se sama vrijednost kopira u područje predviđeno za parametre. U drugom slučaju se obično prosljeđuje pointer na datu vrijednost.

Implementacijski modeli prijenosa parametara

Različite kombinacije semantičkih i konceptualnih modela prijenosa parametara rezultuju različitim implementacijskim modelima: po vrijednosti, po rezultatu, vrijednosti i rezultatu, po referenci, po imenu.

Prosljeđivanje po vrijednosti (Ulazni)

Kod ovog metoda, vrijednost stvarnog parametra se koristi za inicijalizaciju odgovarajućeg formalnog parametra. Normalno se implementira kopiranjem. Može se implementirati i prenosom puta pristupa ali nije preporučljivo (nije lako omogućiti zaštitu upisa)

Mane (ako je fizičko pomijeranje): potrebna dodatna memorija (dva puta se smiješta) a stvarno premiještanje može biti zahtjevno (za velike parametre)

Mane (ako je metoda po stazi pristupa): mora se zaštititi od upisa u pozvanom potprogramu, a pristup košta više jer se koristi indirektno adresiranje.

Prosljeđivanje po rezultatu (Izlazni)

Kada se parametar prosljeđuje po rezultatu, nema vrijednosti koja se šalje potprogramu. Odgovarajući formalni parametar radi kao lokalna varijabla. Vrijednost se proslijedi u pozivaočev stvarni parametar, kada se kontrola vrati pozivaocu, fizičkim premještanjem. Ovaj pristup zahtijeva dodatnu memoriju i operaciju kopiranja.

Potencijalni problem se dešava ako se ista varijabla stvarnog ulaznog parametara prosljeđuje različitim parametrima potprograma, npr sub(pl, pl); nejasno je koji formalni parametar se kopira u pl. U sljedećem C# primjeru nejasno je da li će se varijabli a dodijeliti vrijednost 17 ili 35.

```
void Fixer(out int x, out int y) {
x = 17;
y = 35;
}
...
f.Fixer(out a, out a);
```

Prosljeđivanje po vrijednosti i rezultatu (UlaznoIzlazni)

Ovo je kombinacija prosljeđivanja po vrijednosti i po rezultatu. Parametar se kopira u potprogram pri pozivu i po izlasku iz potprograma kopira se u pozivaoca. Primijenjeno u nekim verzijama Fortran-a i Ada. Ponekad se zove prosljeđivanje po kopiji. Pristup kombinuje prednosti i mane prosljeđivanja po rezultatu i prosljeđivanja po vrijednosti

Prosljeđivanje po referenci (UlaznoIzlazni)

Kod ovog načina prosljeđivanja parametara, prosljeđuje se staza pristupa, što je obično memorijska adresa prosljeđene varijable, elementa niza ili druge memorijske lokacije gdje je podatak. Zove se i prosljeđivanje dijeljenjem (u Pascal je pogrešno nazvan poziv po imenu). Prednost pristupa je što je proces prosljeđivanja efikasan jer nema kopiranja i duple memorije. Mane su što je sporiji pristup formalnim parametrima (u odnosu na prosljeđivanje po vrijednosti), što su mogući neželjeni bočni efekti (ako je normalno očekivan samo jednosmijerni prijenos) i neželjeni aliasi kao u sljedećoj C++ funkciji.

```
void fun(int &first, int &second)
```

Ako se proslijedi ista varijabla dva puta kao u sljedećem pozivu, tada first i second postaju aliasi.

```
fun(total, total)
```

Prosljeđivanje po imenu (UlaznoIzlazni)

Ovaj način prosljeđivanja je tekstualna zamjena. Formalni parametri se povežu za metodu pristupa u trenutku poziva, ali je stvarno povezivanje u trenutku reference ili dodjele. Sljedeći primjer u Algol 60

```
begin integer i;
  integer procedure sum(i, j);
    integer i, j;
     comment parameters passed by name;
     begin
       integer sm;
       sm := 0;
       for i := 1 step 1 until 100 do sm := sm + j;
       sum := sm
      end;
      print(sum(i, i*10 ))
  end
```

Za razliku od ostalih načina prosljeđivanja izraz i*10 se računa u svakoj iteraciji petlje. Pristup se pokazao komplikovanim za implementaciju, pa se ne koristi u modernim jezicima za standardne potprograme, ali je sličan pristup korišten u makroima i generičkim potprogramima.

Implemenacija metoda prosljeđivanja parametara

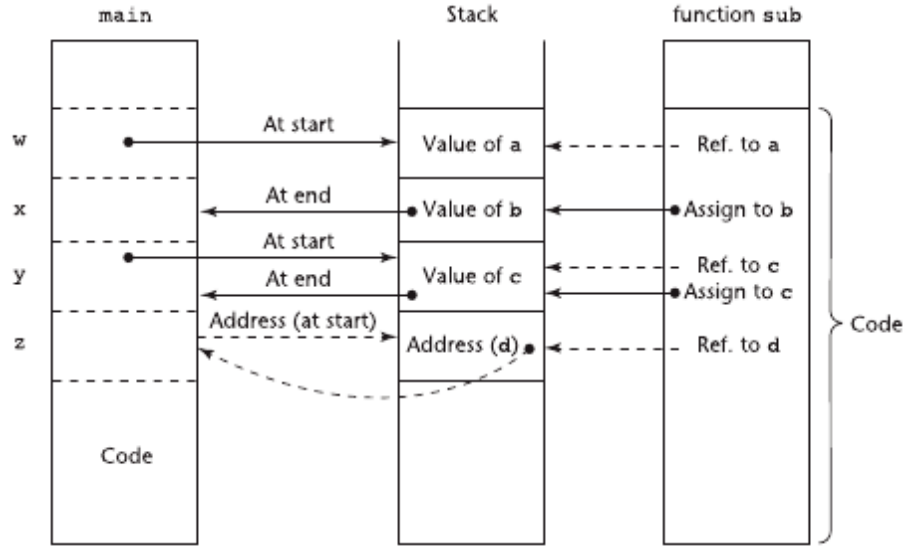
U većini jezika prosljeđivanje parametara ide kroz izvršni stek.

Sljedeća slika ilustruje prosljeđivanje četiri parametra potprogramu

sub (int a, int b, int c, int d)

sa aktuelnim parametrima sub (w,x,y,z)

(gdje se w prosljeđuje po vrijednosti, x po rezultatu, y po vrijednosti i rezultatu, z po referenci)



Prosljeđivanje po referenci je najjednostavnije za implementaciju, bez obzira na tip parametra, samo se adresa smjesti na stek. Mala ali fatalna greška s prosljeđivanjem po referenci ili po rezultatu vrijednosti: formalni parametar koji odgovara konstanti može slučajno biti promijenjen

Metode prosljeđivanja parametara za vodeće jezike

C koristi isključivo prosljeđivanje po vrijednosti. prosljeđivanje po referenci se postiže pointerima kao parametrima

C++ pored ovoga ima specijalni pointerski tip koji se zove tip reference za prosljeđivanje po referenci

Java sve parametre prostog tipa prosljeđuje po vrijednosti, ali pošto se objektima uvijek pristupa po referenci, efekat je da se ovim parametrima pristupa po referenci.

Ada ima tri semantička režima za prenos parametara: in, out, in out; in je podrazumijevani režim. Formalni parametri deklarisani kao out se mogu dodijeliti ali ne referencirati; oni deklarisani kao in se mogu referencirati ali ne dodijeliti; in out parametri se mogu referencirati i dodijeliti

Fortran 95 - Parametri se mogu deklarisati kao in, out, ili inout

Pascal – Po vrijednosti, ako je deklarisan s var onda po referenci

C# - Podrazumijevano: prosljeđivanje po vrijednosti. Prosljeđivanje po referenci je navedeno formalnim parametrom i stvarnim parametrom s ref

PHP: vrlo slično kao C# , s tim što se može u pozivu u parametru navesti znak & ispred parametra.

Perl: svi stvarni parametri su smješteni u predefinisani niz koji se zove @_

Python i Ruby koriste prosljeđivanje dodjelom (sve vrijednosti podataka su objekti)

COBOL potprograme definisane u PROCEDURE DIVISION, npr sa

PROCEDURE DIVISION USING GROSS,DEARNESS.

i pozvane s

CALL "MYSUB" USING G1,D-A.

realizuje kao poziv po vrijednosti ili poziv po referenci.

Parametri s provjerom tipova

Važno za pouzdanost je provjeriti da li su proslijeđeni parametri istog tipa kao formalni parametri potprograma.

FORTRAN 77 i originalni C nisu provjeravali usaglašenost tipova formalnih i stvarnih parametara, dok je u jezicima Pascal, FORTRAN 90, Java, i Ada to uvijek potrebno.

ANSI C pruža izbor korisniku. Ako se funkcije definišu na stari K&R način

```
double sin(x)
double x;
{ ... }
```

tada se tipovi parametara ne provjeravaju. Ako se definišu sa prototipima (osim u slučaju deklaracije s tri tačke) tada se provjerava prosljeđivanje parametara.

```
double sin(double x);
```

Relativno novi jezici Perl, JavaScript, i PHP ne zahtijevaju provjeru tipova.

U Python i Ruby, varijable nemaju tipove (objekti imaju), pa provjera tipova parametara nije moguća.

Problemi dizajna kod prosljeđivanja parametara

Dva važna pitanja dizajna kod prosljeđivanja parametara

- Da li je efikasnost prioritetna
- Da li je prosljeđivanje parametara Jednosmijerni ili dvosmijerni proces

Ova dva pitanja su u konfliktu. Dobro programiranje preporučuje ograničen pristup varijablama, što znači jednosmijerni pristup kada je god moguće, pa je prosljeđivanje po vrijednosti preporučeno. Ali prosljeđivanje po referenci je mnogo efikasnije za prosljeđivanje većih struktura kao što su nizovi, jer se kopira na stek samo jedna memorijska riječ.

Parameteri koji su imena potprograma

Nekad je zgodno proslijediti imena potprograma kao parametre. Tada je u potprogramu moguće indirektno pozvati drugi potprogram. Pitanja dizajna koja se javljaju:

- Da li se tipovi parametara provjeravaju?
- Koje je korektno referencno okruženje za potprogram poslan kao parametar?

Provjera tipova parametara

C i C++: funkcije se ne mogu proslijediti kao parametri nego kao pointeri na funkcije i njihovi tipovi mogu uključiti tipove parametara pa se parametri mogu provjeravati na tipove

FORTRAN 95 provjera tipova

Ada ne dopušta potprogramske parametre, alternativa je Ada generička mogućnost

Java ne dopušta imena metoda kao parametre

Pascal poznaje proceduralne tipove, čime procedure i funkcije mogu biti parametri procedura i funkcija. No, nisu se morali provjeravati tipovi parametara proslijeđenih parametara.

Referentno okruženje za parametre koji su funkcije

Ako jezik dopušta gniježdene potprograme (Pascal, Ada, JavaScript), postavlja se i pitanje koje će biti referentno okruženje proslijeđenog potprograma. Na izboru su tri pristupa.

- Pliko vezivanje Okruženje naredbe koja poziva - Za jezike dinamičkog opsega
- Duboko vezivanje: Okruženje definicije proslijeđenog potprograma - Za jezike statičkog opsega
- Ad hoc vezivanje: Okruženje pozivaoca koji je proslijedio potprogram

U sljedećem primjeru u sintaksi JavaScript (koji normalno koristi duboko vezivanje), kada bi se promijenila definicija jezika vide se razlike između ova tri pristupa.

```
function sub1() {
  var x;
  function sub2() {
   alert(x); //Dialog box with the value of x
  };
  function sub3() {
   var x;
    x = 3;
    sub4(sub2);
   };
  function sub4(subx) {
   var x;
   x = 4;
   subx();
  };
   x = 1;
   sub3();
};
```

Kada se sub4(sub2):

- Za plitko vezivanje, referencno okruženje je od sub4, pa je x iz sub4, a izlaz je 4.
- Za duboko vezivanje, referencno okruženje je od subl, pa je izlaz l.
- Za ad hoc vezivanje, veza je na lokalnu x u sub3, pa je izlaz 3.

Indirektni poziv potprograma

Još jedan način, pored prosljeđivanja funkcija i procedura kao parametara drugim funkcijama i procedurama, za izvršavanje potprograma koji se unaprijed ne znaju je indirektni poziv potprograma. Poziv potprograma se vrši kroz pointer ili referencu na potprogram postavljenu prije poziva potprograma.

U C i C++ ovu ulogu imaju pointeri na funkcije

```
int myfun2 (int, int); // Deklaracija funkcije
int (*pfun2)(int, int); //pointer na funkcije
pfun2 = myfun2; // dodjela funkcije pointeru
(*pfun2)(first, second); // poziv uz dereferenciranje
pfun2(first, second); // Poziv na drugi način
```

U C#, pointeri na metode se zovu delegati

```
public delegate int Change(int x);
```

Ovo se može instancirati metodom koja koristi i vraća int

```
static int fun1(int x);
```

Delegate Change se instancira:

```
Change chgfun1 = new Change(fun1);
```

ili kraće:

```
Change chgfun1 = fun1;
```

Poziv funl kroz delegate chgfunl:

```
chgfun1(12);
```

Objekti delegatske klase mogu imati više metoda, koji se dodaju += operatorom:

```
Change chgfun1 += fun2;
```

U Python, JavaScript i Ruby, kao i većini funkcionalnih jezika funkcija se može dodijeliti varijabli kao i svaki podatak što također omogućava indirektni poziv.

Preopterećeni i generički potprogrami

Preopterećeni potprogram

Preopterećeni potprogram ima isto ime kao drugi potprogram u istom referencirajućem okruženju. Na primjer, moguće je imati tri funkcije invert, od kojih jedna kao parametar ima dva cijela broja, druga float, a treća matricu i koje računaju recipročnu vrijednost razlomka, realnog broja ili inverznu matricu. Iako sve tri imaju isto ime, moguće ih je razlikovati jer svaka verzija preopterećenog potprograma ima jedinstven protokol

C++, Java, C#, i Ada uključuju preopterećene potprograme

U Ada, povratni tip preopterećene funkcije se može koristiti za smanjenje višeznačnosti poziva (tako dvije preopterećene funkcije mogu imati iste parametre)

Ada, Java, C++, i C# dopuštaju korisnicima da pišu više verzija potprograma s istim imenom

Generički potprogrami

Generički ili polimorfni potprogram koristi parametre raznih tipova pri pokretanju. Preopterećeni potprogrami su specijalni slučaj polimorfnih programa pružaju **ad hoc polimorfizam**. Različiti preopterererećeni potprogrami ne moraju se slično ponašati.

Objektno orijentisani jezici, podržavaju **polimorfizam podtipa**, koji znači da metode pisane za tip T, se mogu koristiti i u svim tipovima izvedenim od tipa T.

Potprogram koji uzima generički parametar korišten kao izraz tipa koji opisuje tip parametara potprograma pruža **parametarski polimorfizam.** To je jeftina zamjena za dinamičko povezivanje jer se dešava u trenutku kompajliranja.

Generički potprogrami u C++ se zovu predlošci (template). Parametar naredbe template može biti class identifikator ili typename identifikator, što je ekvivalentno. Identifikator predstavlja ime tipa. Sljedeći primjeri parametarskog polimorfizma u C++

```
template <class Type>
Type max(Type first, Type second) {
  return first > second ? first : second;
}
```

Sljedeći predložak će imati instance za svaki tip za koji je operator > definisan. Na primjer, ako je tip oba parametra bio int, tada nastaje funkcija:

```
int max (int first, int second) {
   return first > second? first : second;
}
```

Verzije generičkog potprograma se kreiraju implicitno kada je potprogram imenovan u pozivu ili se njegova adresa dobije & operatorom. Tako na primjer ako su postojale varijable i pozivi funkcija kao u sljedećem listingu,

```
int a, b, c;
char d, e, f;
...
c = max(a, b);
f = max(d, e);
```

U gornjem primjeru nastaju dvije kopije potprograma max, jedan za poređenje dvije int i jedan za poređenje dvije char varijable.

Od Java 5.0 na raspolaganju su generičke metode. Deklarišu se na sljedeći način

```
generic_class<T>
```

Za razliku od predložakau u C++ tip T u Java generičkim klasama može biti samo klasa, ne i primitivni tip. Sljedeći primjer definiše metodu koja kao parametar ima niz tipa T.

```
public static <T> T doIt(T[] list) {
...
}
```

Metoda se poziva za niz stringova na sljedeći način.

```
doIt<String>(myList);
```

C# 2005 - Podržava generičke metode slične onim u Java 5.0 uz jednu razliku: stvarni tipski parametri u pozivu se mogu izostaviti ako kompajler može obraditi nenavedeni tip

```
class MyClass {
  public static T DoIt<T>(T p1) {
   ...
  }
}
int myInt = MyClass.DoIt(17); // Calls DoIt<int>
string myStr = MyClass.DoIt('apples');
// Calls DoIt<string>
```

Ada

Verzije generičkih potprograma su kreirane kompajlerom kada su instancirane naredbom deklaracije. Generičkim potprogramima prethodi generic klauzula koja lista generičke varijable koje mogu biti tipovi ili drugi potprogrami.

Problemi dizajna kod funkcija

Kod dizajna funkcija postavljaju se tri pitanja:

- Da li su dopušteni bočni efekti?
- Koji tipovi povratnih vrijednosti su dopušteni?
- Koliko vrijednosti funkcija vraća

Bočni efekti kod funkcija su štetni. Sljedeći primjer u C to ilustruje.

```
#include <stdio.h>
#include <string.h>
char buff[100];
char * upcase(char * a) {
  char *p;
  strcpy(buff,a);
  p=buff;
  while (*p) {
    if (*p>='a' && *p<='z')
      *p -= 32;
    p++;
  }
  return buff;
}
void main() {
  printf("%s %s",upcase("prvi"),upcase("drugi"));
}
```

U prethodnom primjeru bilo bi očekivano da funkcija vrati proslijeđeni niz znakova uz pretvorena mala slova u velika. Međutim, zbog bočnog efekta, promjene globalne varijable, naredba printf će dva puta prikazati PRVI.

Većina programskih jezika ne ograničava bočne efekte. Neki jezici ograničavaju bočne efekte time što su parametri uvijek u ulaznom režimu da se izbjegnu (Ada), opštim dizajnom jezika (Haskell, ML) ili limitiranjem funkcija na samo proste aritmetičke izraze (DEF FN naredba u Microsoft BASIC)

Većina imperativnih jezika ograničava tip podatka koji funkcija može da vrati.

- C dopušta bilo koji tip osim nizova i funkcija.
- C++ je kao C ali dopušta i korisničke tipove.
- Pascal dopušta da se može iz funkcije vratiti samo skalarni tip.
- Ada potprogrami mogu vratiti bilo koji tip (ali Ada potprogrami nisu tipovi pa oni ne mogu biti vraćeni).
- Java i C# metode mogu vratiti bilo koji tip (ali pošto metode nisu tipovi one ne mogu biti vraćene).
- Python i Ruby smatraju metode objektima prve klase, pa se mogu vratiti kao bilo kova druga klasa.

Priroda funkcija je da vraćaju jednu i samo jednu vrijednost, pa su tako i definisane u većini jezika. Izuzetak su Lua i Ruby gdje funkcije mogu vratiti više vrijednosti.

U Ruby ako iza return naredbe stoji lista vrijednosti razdvojena zarezima, funkcija vraća niz čiji su elementi tu navedeni.

Lua dopušta funkcijama da vrate više vrijednosti

```
return 3, sum, index
```

U pozivu se navodi

```
a, b, c = fun()
```

Korisnički definisani preopterećeni operatori

Ada, C++, Python, i Ruby moguće je preopteretiti i aritmetičke operatore da se njihov račun preusmjeri na funkciju. Na primjer, može se definisati da operator * množi kompleksne brojeve.

U C++ se pri realizaciji klase mogu definisati i operatori koristeći ključnu riječ operator.

```
Complex operator +(Complex &second) {
return Complex(real + second.real, imag + second.imag);
}
```

U Python operatori se mogu predefinisati definišući funkcije specijalnih naziva.

```
def __add__ (self, second):
  return Complex(self.real + second.real, self.imag +second.imag)
```

Ada primjer

```
function "*" (A,B: in Vec_Type): return Integer is
  Sum: Integer := 0;
begin
  for Index in A'range loop
    Sum := Sum + A(Index) * B(Index)
  end loop
  return sum;
end "*";
```

...

```
c = a * b; -- a, b, and c are of type Vec_Type
```

Klosure

Pascal i JavaScript omogućavaju definisanje potprograma unutar potprograma. Kada je unutrašnji potprogram aktivan, on ima dostup do svojih lokalnih varijabli, globalnih varijabli ali i lokalnih varijabli potprograma koji ga uokviruje. Šta ako programski jezik dopušta poziv unutrašnjih potprograma i sa drugih mjesta u programu, a ne samo iz uokviravajućeg potprograma? Tada moraju biti kreirane i varijable uokviravajućeg potprograma čak i ako on nije u tom trenutku aktivan. Slična situacija se dešava i ako jezik dopušta dodjele funkcije (ne vrijednosti nego baš funkcije) varijabli.

Klosura je potprogram s njegovim referentnim varijablama. Podržane su u skriptnim, funkcionalnim jezicima i C#. U sljedećem primjer u JavaScript klosura je anonimna funkcija definisana unutar makeAdder funkcije.

```
JavaScript primjer:
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("Add 10 to 20: " + add10(20) +"<br />");
document.write("Add 5 to 20: " + add5(20) +"<br />");
```

Isti primjer u C# postiže se ugniježdenim anonimnim delegatom.

```
static Func<int, int> makeAdder(int x) {
return delegate(int y) { return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

Potprogrami s više ulaznih tačaka

Nije dobra praksa da potprogram ima više ulaznih tačaka, ali u starijim programskim jezicima postoje primjeri i za to. Potprogrami u BASIC-u mogu imati više ulaznih tačaka ako se koriste linijski brojevi.

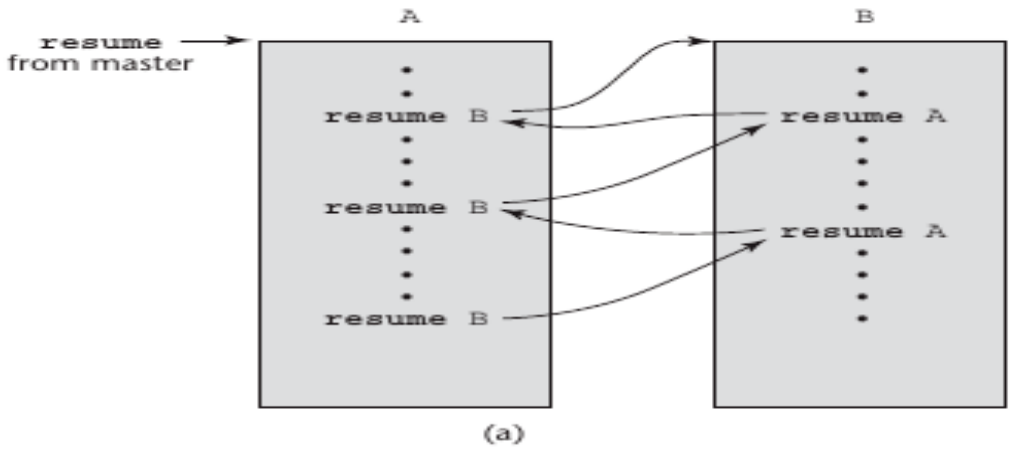
```
10 GOSUB 100
20 GOSUB 110
30 STOP
100 PRINT “ULAZ 1”
110 PRINT “ULAZ 2”
120 RETURN
```

Starije verzije FORTRAN-a ključna riječ ENTRY za ulazak na drugom mjestu od predviđenog.

```
program main
call sub1()
call sub1entry()
end program main
subroutine sub1()
write(*,*) 'subroutine call executes this part'
entry sub1entry()
write(*,*) 'both calls execute this part'
end subroutine sub1
```

Korutine

Korutina je je također potprogram s više ulaza. One se razlikuju od običnih potprograma koji rade na principu da pozivalac čeka da se cijeli potprogram završi. Kod korutina pozivalac i pozvana korutina moraju biti na jednakom nivou. Poziv korutine se zove resumiranje. Prvo resumiranje korutine ide na njen početak. Povratak iz korutine ide na mjesto iza poziva, ali kasniji ulazi u korutinu nastavljaju njeno izvršenje od mjesta gdje se zadnji put vratila u pozivaoca. nakon zadnje naredbe u korutini.



(a)

Korutine pružaju kvazi-konkurentno izvršenje programskih jedinica izvršenje se prepliće ali ne preklapa.

Korutine su podržane jedino u Lua.

```
co = coroutine.create(function ()
       for i=1,10 do
         print("co", i)
         coroutine.yield()
       end
     end)
coroutine.resume(co)    --> co   1
print(coroutine.status(co))   --> suspended
coroutine.resume(co)    --> co   2
coroutine.resume(co)    --> co   3
```

Implementacija potprograma

Poziv potprograma treba da obavi nekoliko operacija. Te operacije su:

- Prosljeđivanje parametara
- Stek dinamička alokacija lokalnih varijabli
- Snimanje izvršnog statusa pozivajućeg programa
- Prebacivanje kontrole i sređivanje za povratak
- Ako je gniježđenje potprograma podržano, mora se srediti pristup nelokalnim varijablama

Povratak iz potprograma treba da obavi sljedeće operacije.

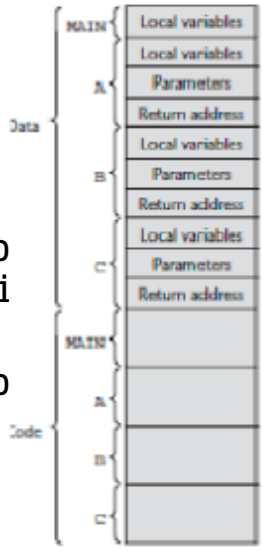
- Parametri ulaznog i izlaznog režima moraju imati vraćene vrijednosti
- Dealokacija stek dinamičkih lokalnih varijabli
- Vraćanje izvršnog statusa
- Povratak kontrole pozivaocu

Složenost ovih operacija zavisi od mogućnosti koje programski jezik pruža.

Implementacija “prostih” potprograma

Prosti potprogrami se javljaju u ranijim verzijama Fortrana i BASIC-u. Kod ovih potprograma nema rekurzivnih poziva, pa lokalne varijable ne treba čuvati na steku. Sve lokalne varijable su statičke. Stoga je je u pozivu potrebno uraditi sljedeće:

- Snimi izvršni status pozivaoca
- Proslijedi parametre
- Proslijedi povratnu adresu pozvanom potprogramu
- Prebaci kontrolu na pozvanog



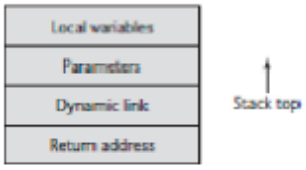
U povratku iz potprograma, potrebno je uraditi sljedeće:

- Ako je prosljeđivanje po rezultatu vrijednosti korišteno ili parametri u izlaznom režimu, premjesti trenutne vrijednosti ovih parametara u odgovarajuće stvarne parametre
- Ako je funkcija, premjesti vrijednost funkcije na mjesto gdje ga pozivalac može uzeti
- Vrati izvršni status pozivaoca
- Vrati kontrolu pozivaocu

U memoriji dakle treba čuvati statusne informacije, parametre, povratnu adresa, povratne vrijednosti za funkcije. Skup podataka koji se čuva za pojedini poziv potprograma zove se **aktivacijski slog**. Za proste potprograme on je odvojen od izvršnog koda i sastoji se od parametara, lokalnih varijabli i povratne adrese, ali njegova lokacija u memoriji može biti unaprijed poznata i fiksna za svaki potprogram.

Implementacija potprograma s lokalnim stek-dinamičkim varijablama

Ako jezik podržava rekurziju (mogućnost više istovremenih aktivacija potprograma) potreban je kompleksniji aktivacijski slog, kao i kod poziva. Aktivacijski slog se nalazi u izvršnom steku. Kompajler mora generisati kod za implicitnu alokaciju i dealokaciju lokalnih varijabli



Tipični aktivacijski slog za jezik sa stek dinamičkim lokalnim varijablama pored parametara, povratne adrese i lokalnih varijabli uključuje i dinamički link. Dinamički link pokazuje na vrh instance aktivacijskog sloga pozivaoca. Instanca aktivacijskog sloga se dinamički kreira pri pozivu potprograma.

Potprogram koji se trenutno izvršava ima postavljen pokazivač okruženja na njega i lokalnim varijablama i parametrima se pristupa relativno pod pozicije pokazivača okruženja. Relativna pozicija variajble se zove local_offset. Za lokalnu varijablu local_offset se može odrediti u trenutku kompajliranja

Neka je dat sljedeći primjer u jeziku C u kome funkcija main poziva funkciju fun1, funkcija fun1 poziva funkciju fun2, funkcija fun2 poziva funkciju fun3:

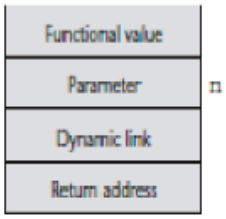
```
void fun1(float r) {
int s, t;
fun2(s);
}
void fun2(int x) {
int y;
fun3(y);
}
void fun3(int q) {
}
void main() {
float p;
fun1(p);
}
```

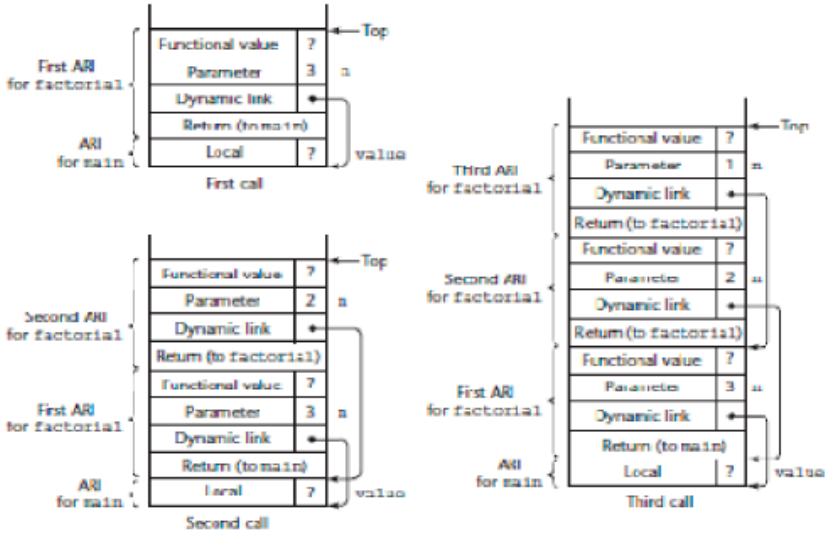
Aktivacijski slogovi na steku u pojedinim trenucima poziva izgledaju kao na sljedećoj slici.

Skup dinamičkih linkova na steku u datom trenutku se zove dinamički lanac, ili lanac poziva.

Implementacija gniježdeni potprograma

Neki ne-C-bazirani jezici statičkog opsega(npr., Fortran 95, Pascal, Ada, Python, JavaScript, Ruby, i Lua) koriste stek dinamičke lokalne varijable i dopuštaju gniježdenje potprograma. Ugniježdeni potprogram ima pristup lokalnim varijablama statičkog roditelja, ali te varijable nisu lokalne za sam ugnježdeni potprogram. Kod ovakvih jezika, aktivacijski slog se mora dodatno proširiti, jer sve varijable kojima se može nelokalno pristupiti nisu u tekućoj nego se nalaze u svojoj instanci aktivacijskog sloga na steku.





U aktivacijski slog pored dinamičkog linka (koji pokazuje na aktivacijski slog pozivaoca), dodaje se i statički link koji pokazuje na aktivacijski slog statičkog roditelja.

Proces traženja ne-lokalne reference tada se sastoji od dva dijela

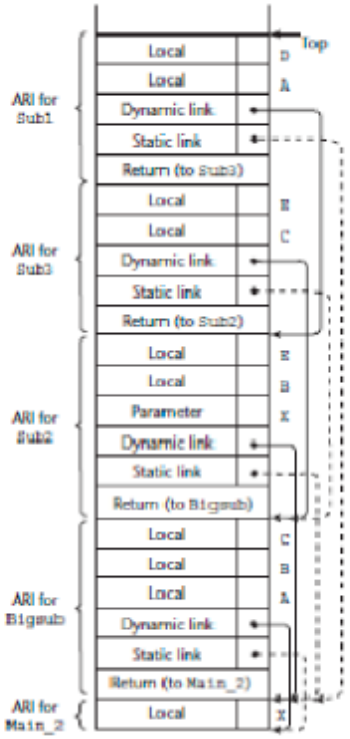
- Nađi instancu aktivacijskog sloga koja se odnosi na datog statičkog roditelja
- Odredi poziciju untar te instance aktivacijskog sloga

Pozicija unutar instance aktivacijskog sloga je već poznata u trenutku kompajliranja. Traženje korektne instance aktivacijskog sloga je nešto duži zadatak. Pravila statičke semantike garantuju da sve nelokalne varijable kojima možemo pristupiti su alocirane u nekoj instanci aktivacijskog sloga koja je na steku kada se vrši referenca. Odgovarajuća instanca se nalazi kroz statički lanac. Statički lanac je lanac statičkih linkova koji povezuje određene instance aktivacijskih slogova. Statički link u instanci aktivacijskog sloga za potprogram A pokazuje na jednu od instanci aktivacijskog sloga statičkog roditelja od A. Statički lanac od instance aktivacijskog sloga povezuje ga s svim statičkim precima

Primjer Ada Program u kome Main_2 poziva Bigsub, Bigsub poziva Sub2, Sub2 poziva Sub3, Sub3 poziva Subl

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
     begin -- of Sub1
       A := B + C; <----------------------1
     end; -- of Sub1
     procedure Sub2(X : Integer) is
       B, E : Integer;
       procedure Sub3 is
         C, E : Integer;
         begin -- of Sub3
            Sub1;
            E := B + A: <--------------------2
         end; -- of Sub3
       begin -- of Sub2
          Sub3;
          A := D + E; <----------------------3
       end; -- of Sub2 }
  begin -- of Bigsub
      Sub2(7);
  end; -- of Bigsub
begin
  Bigsub;
end; of Main_2 }
```

Sadržaj steka u tački 1 dat je na sljedećoj slici



Pri pozivu potprograma statički link mora pokazivati na najnoviju instancu aktivacijskog sloga statičkog roditelja, što se mora uraditi kroz pretragu dinamičkog lanca.

Glavna mana statičkih lanaca je što je nelokalna referenca je spora ako je velika dubina gniježdenja, pošto se za pristup varijabli mora prolaziti kroz ulančanu listu.

Implementacija blokova

Blokovi su korisnički opseg varijabli koji pretstavljaju manju jedinicu od potprograma

Primjer u C

```
{int temp;
  temp = list [upper];
  list [upper] = list [lower];
  list [lower] = temp
}
```

Život varijable temp u gornjem primjeru počinje kada kontrola uđe u blok. Prednost lokalne varijable poput temp je što ne može interferirati s drugim varijablama s istim imenom

Za implementaciju blokova postoje dva metoda.

U prvom pristupu, blokovi se tretiraju kao potprogrami bez parametara pozvani s iste lokacije. Svaki blok ima aktivacijski slog, instanca se kreira pri svakom izvršenju bloka

U drugom pristupu, pošto se maksimalna količina memorije potrebna za blok može statički odrediti, količina memorije se može alocirati nakon lokalnih varijabli u aktivacijskom slogu potprograma u kome se nalazi blok.

Rezime

Definicija potprograma predstavlja akcije predstavljene potprogramom. Potprogrami mogu biti funkcije i procedure. Lokalne varijable u potprogramima mogu biti stek-dinamičke ili statičke.Tri su modela prosljeđivanja parametara: Ulazni, Izlazni, and UlaznoIzlazni. Neki jezici dopuštaju preopterećenje operatora. Potprogrami mogu biti generički. Korutina je specijalan potprogram s više ulaza.

Semantika povezivanja potprograma zahtijeva više akcija implementacijom. Prosti potprogrami zahtijevaju relativno jednostavne akcije. Stek dinamički jezici su kompleksniji. Potprogrami s stek dinamičkim lokalon im varijablama i gniježdenim potprogramima imaju dvije komponente: stvarni program i aktivacijski slog. Instance aktivacijskog sloga sadrže formalne parametre i lokalne varijable. Statički lanci su primarni metod pristupa nelokalnim varijablama u jezicima statičkog opsega s gniježdenim potprogramima. Pristup nelokalnim varijablama u jezicima dinamičkog opsega može se realizovati dinamičkim lancom ili kroz centralnu tabelu varijabli

10. Apstraktni tipovi podataka i koncepti enkapsulacije

Apstrakcija je način predstavljanja entiteta koji uključuje samo najkarakterističnije atribute. Ona se koristi u cilju pojednostavljenja globalne slike u svijetu. Na primjer, automobil može imati hiljade atributa (boja sjedišta, izgled ručice za otvaranje prozora, natpis na mjenjaču, sadržaj rezervnog alata ...) ali za većinu primjena za predstavu o autu su dovoljna samo tri najkarakterističnija: marka, boja i registarski broj.

Koncept apstrakcije je fundamentalan u programiranju i računarskim naukama, jer omogućava podjelu posla u razvoju softvera. Apstrakcija je moguća nad procesima i podacima.

Gotovo svi programski jezici podržavaju apstrakciju procesa preko potprograma. Potprogrami omogućavaju realizaciju određenog posla, bez potrebe da pozivalac zna mnogo detalja o načinu realizacije. Na primjer sljedeći poziv funkcije

sort(niz);

odrađuje uređivanje niza bez potrebe da pozivalac brine o detaljima algoritma za sortiranje.

Gotovo svi programski jezici dizajnirani nakon 1980 podržavaju i apstrakciju podataka

Uvod u apstrakciju podataka

Apstraktni tipovi podataka su korisnički definisani tipovi podataka koji zadovoljavaju sljedeća dva uslova:

- Predstavljanje i operacija nad objektima su definisani u jednoj sintaksnoj jedinici
- Interno predstavljanje objekata datog tipa je skriveno od programskih jedinica koje koriste ove objekte, pa su jedine operacije moguće one koje su date u definiciji tipa

Prednosti prvog uslova su u organizaciji i izmjenjivosti programa (sve vezano za strukturu podataka je zajedno) i odvojeno kompajliranje

Prednosti drugog uslova su u većoj pouzdanosti, čitljivosti a nekada i performansama. Sakrivanjem predstavljanja podataka, korisnik ne može direktno pristupiti objektima tipa ili zavisiti od reprezentacije, što omogućuje njenu promjenu bez promjene korisničkog koda (pouzdanost). Korisnik apstraktnog tipa podataka brine o manjem broju varijabli koje su potrebne za pomoć u realizaciji algoritma (čitljivost). Konačno, moguće je mijenjati algoritam implementacije apstraktnog tipa (npr. promjenu pretraživanja tablice iz sekvencijalnog u binarno pretraživanje) bez uticaja na ostatak programa (performanse)

Na primjer, apstrakni tip steka treba da ima sljedeće operacije:

push (za smiještanje podatka na vrh steka)

pop (za uklanjanje podatka s vrha steka)

isempty (za provjeru da li je stek prazan).

Dijelu programa koji koristi ovaj tip je nebitno da li je stek realizovan kao ulančana lista, niz, dvostruko spregnuta lista ili uz pomoć instrukcija za rad sa stekom u mašinskom jeziku niti bi trebao da direktno pristupa elementima te liste ili niza.

Pitanja dizajna

Apstraktni tipovi podataka postavljaju sljedeća pitanja dizajna.

- Kojeg je oblika sadržilac interfejsa prema tipu? Da li se apstraktni tipovi podataka definišu baš kao takvi (većina objektno orijentisanih jezika) ili kroz generalnije jedinice skupova potprograma i varijabli (poput paketa u Ada)
- Mogu li se apstraktni tipovi parametrizirati?
- Kakva kontrola pristupa je data?
- Da li je specifikacija tipa fizički odvojena od implementacije?

Apstrakcija podataka u Ada

Enkapsulacijska konstrukcija se zove paket. Paket se sastoji od dva dijela, koji se takođe zovu paketi:

- Specifikacijski paket (interface)
- Tijelo paketa (implementacija imenovanih entiteta)

Skrivanje informacija je postignuto tako što specifikacijski paket ima dva dijela, javni i privatni. Ime apstraktnog tipa se javlja u javnom dijelu specifikacijskog paketa. Ovaj dio može uključiti i predstavljanje neskrivenih tipova. Predstavljanje apstraktnog tipa se nalazi u dijelu specifikacije koji se zove privatni dio. U ovom dijelu se nalaze **privatni tipovi** podataka. Oni imaju ugrađene operacije za dodjelu i poređenja na jednakost i nejednakost. Pored njih postoje i **limitirani privatni** tipovi koji nemaju ugrađene operacije.

Razlog zašto se u specifikacijskom paketu navode i javni i privatni dio je što kompajler mora da zna veličinu objekata koje generiše prije kompajliranja programa koji ih koriste, pa mora poznavati oba dijela. S druge strane, korisnici moraju vidjeti ime tipa, ali ne reprezentaciju (pa ne mogu vidjeti privatni dio)

Sljedeći primjer u jeziku Ada pokazuje kako izgleda specifikacijski paket za apstrakni objekt steka.

```
package Stack_Pack is
  type stack_type is limited private;
  max_size: constant := 100;
  function empty(stk: in stack_type) return Boolean;
  procedure push(stk: in out stack_type; elem:in
    Integer);
  procedure pop(stk: in out stack_type);
  function top(stk: in stack_type) return Integer;
private -- hidden from clients
  type list_type is array (1..max_size) of Integer;
  type stack_type is record
    list: list_type;
    topsub: Integer range 0..max_size) := 0;
  end record;
end Stack_Pack
```

Implementacijski paket izgleda npr. ovako.

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk: in Stack_Type) return Boolean is
  begin
   return Stk.Topsub = 0;
  end Empty;
  procedure Push(Stk : in out Stack_Type;
   Element : in Integer) is
  begin
    if Stk.Topsub >= Max_Size then
      Put_Line("ERROR - Stack overflow");
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
  end Push;
  procedure Pop(Stk : in out Stack_Type) is
  begin
   if Empty(Stk)
     then Put_Line("ERROR - Stack underflow");
   else Stk.Topsub := Stk.Topsub - 1;
   end if;
  end Pop;
  function Top(Stk : in Stack_Type) return Integer is
  begin
   if Empty(Stk)
     then Put_Line("ERROR - Stack is empty");
   else return Stk.List(Stk.Topsub);
   end if;
  end Top;
end Stack_Pack;
```

Klijent koji koristi specifikacijski i implementacijski paket izgleda ovako:

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type;
   -- Creates a Stack_Type object
  begin
    Push(Stack, 42);
    Push(Stack, 17);
    Topone := Top(Stack);
    Pop(Stack);
    ...
   end Use_Stacks;
```

Apstrakcija podataka u Turbo Pascal i Modula 2

Sličan koncept apstrakcije podataka primijenjen u jeziku Ada, može se naći u Modula 2 (moduli) i Turbo Pascal-u (unit-i).

Moduli u Modula 2 se, kao i u Ada razdvajaju u dvije datoteke, definicioni modul u kome se deklarišu operacije vidljive klijentima i implementacioni, u kome se piše programski kod koji izvršava te operacije, kao i one operacije koje su skrivene od klijenata,

Unit u Turbo Pascal i kasnije Delphi, se piše u jednoj datoteci izvornog koda koja se sastoji od sekcija interface (deklaracija tipova i potprograma vidljivih klijentima), implementation (skriveni dio implementacije), initialization (opcionalni dio koji se izvršava pri startu programa) i finalization (opcionalni dio koji se izvršava pri završetku programa)

Apstrakcija podataka u C++

C++ je proširenje jezika C klasama i objektima. Ovo proširenje je bazirano na C struct tipu i Simula 67 klasama. U C++ klasa je jedinica enkapsulacije. C++ klase su tipovi, za razliku od Ada paketa koji definišu više različitih tipova. Programska jedinica koja pristupa Ada paketima može pristupiti njenim javnim entitetima direktno preko imena, dok se elementima C++ klase može pristupati samo kroz instance klase.

Podaci u C++ klasama se zovu podatkovni članovi, dok se operacije nad njima zovu članske funkcije. Podatkovni članovi i članske funkcije se javljaju u dvije verzije: zajedničke za klasu i vezane za instancu. Svi primjerci (instance) klase dijele jednu kopiju funkcija članice, ali svaki primjerak klase ima vlastitu kopiju podatkovnih članova.

Instance mogu biti statičke, stek dinamičke, ili heap dinamičke. Statičkim i stek dinamičkim instancama pristupa se direktno kroz vrijednost varijabli, dok se heap dinamičkim pristupa preko pointera. Heap dinamičke instance se kreiraju new operatorom a uništavaju delete operatorom.

Skrivanje informacija je postignuto kroz sekcije unutar deklaracije klase. Ključna riječ **private** određuje skrivene entitete, koji se mogu koristiti samo unutar instance klase. Entiteti koji su dostupni klijentima opisani su uz ključnu riječ **public.** Entiteti kojima se može pristupati samo u naslijeđenim klasama su definisani unutar sekcije označene ključnom riječju **protected**.

Dvije posebne vrste članskih funkcija su konstruktori i destruktori. **Konstruktori** su funkcije koje inicijaliziraju podatke instance. Imaju isto ime kao ime klase, ali ih može biti više kopija s različitim vrstama parametara. Implicitno su pozvani kada se kreira instanca, a mogu se i eksplicitno pozvati. Mogu alocirati memoriju ako je dio objekta heap-dinamički. **Destruktori** su funkcije koje se pozivaju nakon uništenja instance, obično za oslobađanje prostora na heap. Implicitno su pozvani na kraju života objekta, a mogu se i eksplicitno pozvati. Ime destruktora je ime klase ispred koga je tilda (~).

C++ dopušta da se tijelo članskih funkcija navodi u samoj njenoj definiciji. Primjer definicije takve klase u jeziku C++, koja sadrži tri privatna člana , tri javne članske funkcije, konstruktor i destruktor izgleda ovako.

```
#include <iostream.h>
class Stack {
  private:
    int *stackPtr;
    int maxLen;
    int topSub;
public:
  Stack() {
    stackPtr = new int [100];
    maxLen = 99;
    topSub = -1;
  }
~Stack() {delete [] stackPtr;}; //destructor
void push(int number) {
  if (topSub == maxLen)
   cerr << "Error in push--stack is full\n";
   else stackPtr[++topSub] = number;
}
void pop() {
  if (empty())
  cerr << "Error in pop--stack is empty\n";
   else topSub--;
}
int top() {
  if (empty())
    cerr << "Error in top--stack is empty\n";
  else
    return (stackPtr[topSub]);
}
int empty() {return (topSub == -1);}
```

Primjer programa koji koristi ovako definisan apstraktni tip.

```
void main() {
  int topOne;
  Stack stk; //** Create an instance of the Stack class
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
...
}
```

U prethodnom primjeru sakrivanje informacija je prisutno pri kompajliranju korisnika Stack klase, ali ne i na nivou analize izvornog koda. C++ dopušta i da se klase pišu iz dva dijela,l zaglavlja i tijela, koji se tada čuvaju u odvojenim datotekama. Tako je zaglavlje Stack klase dato u sljedećem kodu.

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private:
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
  Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

Implementacija klase je data u odvojenoj datoteci, djelomično datoj u sljedećem listingu

```
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;};
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Apstrakcija podataka u Objective C

Objective C je takođe proširenje jezika C klasama i objektima, ali je sintaksa klasa i objekata bazirana na SmallTalk-u.

Interfejs klase opisan je unutar direktive interface. Direktive se piši sa znakom @ ispred.

```
@interface class-name: parent-class {
instance variable declarations
}
method prototypes
@end
```

Implementacija klase je u sadržiocu implementation sljedeće sintakse.

```
@implementation class-name
method definitions
@end
```

Prototipi metoda imaju sljedeći oblik.

```
(+ | -)(returntype) method-name [: (formalparams)];
```

Znak plus predstavlja metodu klase, a znak minus metodu instance. Ugaone zagrade označavaju da su formalni parametri opcioni. Parametri se pišu na drugačiji način od onog kako se to radi u C.

U ovom primjeru

```
-(void) meth1: (int) x;
```

metod se zove methl: i ima jedan parametar x. U sljedećem primjeru metoda se zove meth2:second: .

```
-(int) meth2: (int) x second: (float) y;
```

Definicije metoda se pišu kao prototipi, osim što umjesto dvotačke koriste vitičaste zagrade.

Poziv metoda bez parametara se obavlja unutar uglastih zagrada

```
[object-name method-name];
```

Ako metoda ima jedan parametar uz ime se navodi dvotačka s parametrom. Ako metoda ima više parametara a jedan dio imena, dvotačke se navode između pojedinih parametara.

```
[myAdder add1: 7: 5: 3];
```

Konstruktori u Objective-C se zovu inicijalizatori i pružaju inicijalne vrijednosti. Objekti u Objective-C se kreiraju pozivom metode alloc.

Skrivanje informacija u Objective-C postignuto je pomoću direktiva @private i @public koje navode nivoe pristupa instancnih varijabli u definiciji klase. Nema načina za ograničenje pristupa metodi.

Metoda koja čita varijablu u Objective-C, ima isto ime kao ime varijable, dok metoda koja upisuje varijablu se zove setXXXX gdje je XXXX ime varijable. Varijable koje automatski dobijaju ove metode se navode u property direktivi.

```
@property int sum;
```

Sljedeći primjer ilustruje opis interfejsa i implementacije u Objective C

```
// stack.m
#import <Foundation/Foundation.h>
// Interface section
@interface Stack: NSObject {
  int stackArray [100];
  int stackPtr;
  int maxLen;
  int topSub;
}
-(void) push: (int) number;
-(void) pop;
-(int) top;
-(int) empty;
@end
// Implementation section
@implementation Stack
-(Stack *) initWith {
  maxLen = 100;
  topSub = -1;
  stackPtr = stackArray;
  return self;
}
-(void) push: (int) number {
if (topSub == maxLen)
  NSLog(@"Error in push--stack is full");
else
  stackPtr[++topSub] = number;
}
-(void) pop {
if (topSub == -1)
  NSLog(@"Error in pop--stack is empty");
else
  topSub--;
}
-(int) top {
if (topSub >= 0)
  return stackPtr[topSub]);
else
  NSLog(@"Error in top--stack is empty");
}
-(int) empty {
return topSub == -1);
}
int main (int argc, char *argv[]) {
int temp;
NSAutoreleasePool *pool =
  [[NSAutoreleasePool alloc]
 init];
Stack *myStack = [[Stack alloc]initWith];
[myStack push: 5];
[myStack push: 3];
```

```
temp = [myStack top];
NSLog(@"Top element is:%i", temp);
[myStack pop];
temp = [myStack top];
NSLog(@"Top element is:%i", temp);
temp = [myStack top];
[myStack pop];
[myStack release];
[pool drain];
return 0;
}
@end
```

Objective C je preuzeo sintaksu iz dva dosta različita jezika, C i Smalltalk. Nedostatak mu je što metode nemaju ograničenje pristupa.

Apstrakcija podataka u Java

Apstraktni tipovi u Java su slični kao u C++, osim sljedećih elemenata:

- Svi korisnički definisani tipovi su klase
- Svi objekti se alociraju kroz heap i pristupa im se kroz referencne varijable
- Tijelo metode je unutar zaglavlja, pa su apstraktni tipovi deklarisani i definisani u jednoj sintaksnoj jedinici.
- Individualni entiteti u klasama imaju kontrolne modifikatore pristupa (private ili public), umjesto klauzula koje razdvajaju sekcije.
- Java ima drugi mehanizam opsega, pakete, umjesto friends. Svi entiteti u svim klasama paketa koji nemaju modifikatore kontrole pristupa su vidljivi u paketu

Primjer u jeziku Java

```
class StackClass {
  private int [] stackRef;
  private int maxLen,topIndex;
  public StackClass() { // A constructor
  stackRef = new int [100];
  maxLen = 99;
  topIndex = -1;
}
public void push(int number) {
  if (topIndex == maxLen)
    System.out.println("Error in push—stack is full");
   else stackRef[++topIndex] = number;
}
public void pop() {
  if (empty())
    System.out.println("Error in pop—stack is empty");
  else --topIndex;
}
public int top() {
  if (empty()) {
   System.out.println("Error in top—stack is empty");
   return 9999;
  }
  else
  return (stackRef[topIndex]);
}
public boolean empty() {return (topIndex == -1);}
}
```

Apstrakcija podataka u C#

C# je sličan C++ i Java po načinu definicije klasa, ali uključuje i neke nove konstrukcije. Kao u Java, sve instance klase su heap dinamičke.

Jezik podržava konstruktore i destruktore. Podrazumijevani konstruktori su raspoloživi za sve klase. Automatsko skupljanje smeće se koristi za većinu heap objekata pa se destruktori rijetko koriste.

Najveća razlika u odnosu na C++ je značenje klasa definisanih sa klauzulom struct. U C++ struct su klase čiji su svi članovi public, dok u C# struct predstavljaju malo zahtjevne klase koje ne podržavaju nasljeđivanje.

Za pristupanje vrijednostima polja C# pruža property kao mogućnost implementacije getter i setters bez potrebe poziva tih metoda. Sljedeći primjer ilustruje C# Property DegreeDays. On se ponaša kao public član klase, ali dodjela i čitanje automatski pozivaju get i set metode.

```
public class Weather {
  public int DegreeDays { //** DegreeDays is a property
  get {return degreeDays;}
  set {
    if(value < 0 || value > 30)
      Console.WriteLine(
         "Value is out of range: {0}", value);
      else degreeDays = value;}
    }
private int degreeDays;
...
}
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Apstraktni tipovi podataka u Ruby

I u jeziku Ruby enkapsulacijska konstrukcija je klasa. Klasa se navodi unutar ključnih riječi class i end.

Varijable unutar klase se dijele na lokalne varijable metoda, varijable instance i varijable klase. Lokalne varijable imaju uobičajena imena identifikatora, koja počinju slovom. Imena varijable instance počinju znakom @, dok imena varijable klase počinju s dva znaka @@.

Metode instance imaju sintaksu Ruby funkcija (def ... end). Metode klase se razlikuju od metoda instance po tome što im ime počinje imenom klase razdvojeno tačkom.

Konstruktori se zovu initialize i može biti jedan po klasi koji se implicitno poziva kada se objekat kreira s new. Ako treba više konstruktora oni moraju imati različita imena i eksplicitno pozvati new.

Klase u Ruby su dinamičke. Novi članovi se dodaju u toku izvršavanja programa. Na primjer, neka je definisana klasa

```
class myClass
  def meth1
    ...
  end
end
```

Novu metodu je moguće dodati izvršavanjem nove definicije klase.

```
class myClass
  def meth2
   ...
  end
end
```

Metode je moguće i brisati, pozivom metode remove_method, koja kao parametar ima ime metode koja se briše.

Sakrivanje informacija u Ruby klasama se može postići sekcijama sa klauzulom private i public u definiciji klase ili navođenjem liste metoda iza tih klauzula.

```
class MyClass
def meth1
...
end
...
private
def meth7
...
end
...
end # of class MyClass
```

ili

```
class MyClass
def meth1
...
end
...
def meth7
...
end
private :meth7, . . .
end # of class MyClass
```

Svi članovi klase su privatni, pa im se može pristupati samo kroz setter i getter metode. U Ruby, podaci instance pristupačni kroz pristupne metode se zovu atributi.

```
def sum
@sum
end
def sum=(new_sum)
@sum = new_sum
end
```

Primjer Stack klase u Ruby:

```
# Stack.rb
class StackClass
# Constructor
  def initialize
    @stackRef = Array.new(100)
    @maxLen = 100
    @topIndex = -1
  end
# push method
  def push(number)
    if @topIndex == @maxLen
       puts "Error in push - stack is full"
     else
        @topIndex = @topIndex + 1
      @stackRef[@topIndex] = number
  end
end
# pop method
   def pop
      if empty
        puts "Error in pop - stack is empty"
     else
       @topIndex = @topIndex - 1
     end
   end
# top method
def top
   if empty
       puts "Error in top - stack is empty"
   else
      @stackRef[@topIndex]
   end
end
# empty method
def empty
   @topIndex == -1
end
end # of Stack class
# Test code for StackClass
myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop
# The following pop should produce an
# error message - stack is empty
myStack.pop
```

Parametrizirani apstraktni tipovi podataka

Nekada je potrebno parametrizirati apstraktne tipove, Parameterizirani ADT dopuštaju dizajn ADT koji mogu smjestiti elemente bilo kog tipa, na primjer u koga se može smjestiti po potrebi stek stringova, cijelih brojeva ili objekata. Problem postoji samo kod jezika s statičkim povezivanjem tipova, jezici kod kojih se tip dinamički određuje (Ruby, Python, JavaScript) to postižu automatski.

Jezici s statičkim povezivanjem tipova C++, Ada, Java 5.0, i C# 2005 pružaju podršku za parametrizirane ADTs

Parameterizirani ADT u Ada

Primjer steka dat ranije je ograničen na 100 elemenata i cjelobrojne elemente. Za rješavanje ovakvih ograničenja, mogu se koristiti Ada generički paketi. Primjer steka je

Tip stek postaje fleksibilniji s generičkim tipom elementa i veličinom steka

```
generic
Max_Size : Positive;-- Generic parameter
type Element_Type is private; -- Generic parameter
package Generic_Stack is
 -- The visible entities, or public interface
 type Stack_Type is limited private;
 function Empty(Stk : in Stack_Type) return Boolean;
 procedure Push(Stk : in out Stack_Type;
   Element : in Element_Type);
 procedure Pop(Stk : in out Stack_Type);
 function Top(Stk : in Stack_Type) return Element_Type;
 -- The hidden part
 private
  type List_Type is array (1..Max_Size)of Element_Type;
 type Stack_Type is
 record
   List : List_Type;
   Topsub : Integer range 0..Max_Size := 0;
 end record;
end Generic_Stack;
```

Parameterizirani ADT u C++

C++ pruža parametrizirane ADT kroz konstruktore s parametrima i generičke klase kroz predloške. U primjeru sa stekom, Konstruktori s parametrima rješavaju problem fiksne veličine.

```
class Stack {
…
Stack (int size) {
  stk_ptr = new int [size];
  max_len = size - 1;
  top = -1;
};
…
}
Stack stk(100);
```

Predlošcima se može realizovati tip stack sa proizvoljnim tipom podataka koji se smješta u njega.

```
 <class Type> class Stack {
   private: Type *stackPtr;
      const int maxLen;
      int topPtr;
   public: Stack() {
       stackPtr = new Type[100];
       maxLen = 99;
       topPtr = -1;
   }
 …
}
```

Parameterizirane klase u Java 5.0

Java 5.0 poznaje parametrizirane apstraktne tipove u kojima generički parametri moraju biti klase. Najčešći generički tipovi su kolekcije poput LinkedList i ArrayList. Oni mogu smjestiti bilo koji objekt (ali ne i primitivne tipove). Za očitavanje objekata potrebno je koristiti prisiljavanje tipova (type casting).

```
//* Create an ArrayList object
ArrayList myArray = new ArrayList();
//* Create an element
myArray.add(0, new Integer(47));
//* Get first object
Integer myInt = (Integer)myArray.get(0);
```

Od verzije Java 5, eliminisana potreba za prisiljavanje tipova uvođenjem generičkih klasa. Generičke klase imaju parametre u kosim zagradama.

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

Generičke klase se kreiraju sljedećom sintaksom.

```
public class MyClass<T> {
…
}
```

Parameterizirane klase u C# 2005

C# 2005 ima sličnu sintaksu generičkih klasa kao u Java 5.0

```
class Test<T>
{
    T _value;
    public Test(T t)
    {
        // The field has the same type as the parameter.
        this._value = t;
    }
    public void Write()
    {
        Console.WriteLine(this._value);
    }
}
class Program
{
    static void Main()
    {
        // Generic type Test with an int type parameter.
        Test<int> test1 = new Test<int>(5);
        // Call the Write method.
        test1.Write();
        // Generic type Test with a string type parameter.
        Test<string> test2 = new Test<string>("cat");
        test2.Write();
    }
}
```

Konstrukcije enkapsulacije

Veliki programi imaju dvije specijalne potrebe:

- Drugi način organizacije od proste podjele u potprograme i apstraktne tipove podataka
- Način parcijalne kompilacije da se prevode samo dijelovi programa (kompilacijske jedinice koje su manje od cijelog programa)

Očigledno rješenje: grupisanje logički srodnih potprograma u jedinicu koja se posebno kompajlira (kompilacijske jedinice)

Takve kolekcije su enkapsulacija. Enkapsulacije se smještaju u biblioteke sa ciljem da budu naknadno upotrebljive.

Gniježdeni potprogrami

Jedan od načina enkapsulacije je organizacija programa gniježdenjem definicija potprograma u logički veće potprograme koji ih koriste. Gniježdeni potprogrami su podržani u Ada, Pascal, Fortran 95, Python, i Ruby, ali i u tim jezicima nisu primarna enkaspulacijska konstrukcija.

Enkapsulacija u C

C nema apstraktne tipove podataka, ali postoji mogućnost da se datoteke koje sadrže jedan ili više potprograma nezavisno kompajliraju i na taj način formira biblioteka. Interfejs ove biblioteke se smiješta header datoteku (.h) koja sadrži imena tipova i

funkcija koji su implementirani. Korisnički program onda može pristupiti korištenjem #include predprocesorske specifikacija, koja u program uključuje zaglavlje, prostim kopiranjem izvornog koda. Djelomično sakrivanje predstavljanja podataka je moguće kroz publikovanje pointera na strukture. Izvorni kod implementacije nije potrebno imati za njenu upotrebu, jer se povezivanje obavlja linkerom.

Problem sa enkapsulacijom u C je što linker ne provjerava usaglašenost tipova između zaglavlja i vezane implementacije. Ako je npr. promijenjen broj parametara funkcije ili neki od njihovih tipova, a korisnik koristi novo zaglavlje, uz staru biblioteku dobiveni izvršni program će vjerovatno krahirati.

Enkapsulacija u C++

C++ pruža dvije enkapsulacijske konstrukcije. Jedna je preuzeta iz C: mogu se definisati datoteke s zaglavljem i kodom C. Druga konstrukcija su klase. Kada se one koriste za enkapsulaciju, klasa se deklariše u jednoj datoteci, kao interfejs (prototip), dok su definicije članova u posebnoj datoteci.

U C++ se dešava nedostatak generalnije konstrukcije enkapsulacije za operacije koje djeluju nad objektima iz različitih klasa ali koje prirodno ne pripadaju ni jednoj od njih. Na primjer ako postoji klasa vektora i klasa matrica, a potrebno je definisati funkciju za množenje vektora matricom, ta funkcija bi morala pristupati internim članovima obije klase, a ne bi se mogla definisati ni u jednoj od njih.

Rješenje ovog problema predstavljaju friend funkcije. One pružaju pristupu privatnim elementima klase tako što se funkcija deklarisana izvan klase ključnom riječju friend uključi u klasu,

```
class Matrix; //** A class declaration
class Vector {
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
class Matrix { //** The class definition
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};//** The function that uses both Matrix and Vector objects
Vector multiply(const Matrix& m1, const Vector& v1) {
  ...
}
```

Ada paketi

Ada specifikacijski paketi mogu uključiti bilo koji broj deklaracija podataka ili potprograma, različitih tipova. Ada paketi se mogu separatno kompajlirati. Takođe, specifikacija paketa i tijelo se mogu separatno kompajlirati.

C# Asemblije

Jezici namijenjeni za .NET platformu imaju enkapsulacijsku jedinicu veću od klase, koje se zovu asemblije (assembly). To je kolekcija datoteka koja može postati jedinstvena DLL ili EXE.Svaka datoteka sadrži modul koji se može separatno kompajlirati. DLL u .NET jezicima je kolekcija klasa i metoda koje se individualno linkuju s izvršnim programom u trenutku izvršavanja programa.

Friend funkcije nisu potrebne, jer C# ima modifikator pristupa internal; član klase markiran s internal je vidljiv svim klasama u asembliji kojim pripada.

Java arhive

Kompajlirane Java klase se mogu smjestiti u zajedničku datoteku, arhivu. Kolekcija datoteka koja će postati jedinstvena jar datoteka

Delphi unit

ObjectPascal u Delphi verziji koristi konstrukciju unit, koja u svom interfejsnom dijelu može deklarisati više klasa i korisničkih tipova.

Enkapsulacija imenovanja

Veliki programi definišu puno globalnih imena, koja se moraju logički grupisati. Pri tome se dešava da različiti programerski timovi kreiraju ista imena funkcija, klasa ili globalnih varijabli, što je posebno čest slučaj kada se koriste biblioteke.

Enkapsulacija imena se koristi za kreiranje novog opsega imenovanja. Ove enkapsulacije su logičke, one ne moraju biti kontinualne, nego se različite kolekcije mogu smjestiti u iste imenske prostore bez obzira gdje se nalaze.

C++ Imenski prostori

U C++ moguće je smjestiti svaku biblioteku u svoj imenski prostor. To se postiže sljedećom konstrukcijom

```
namespace myStackSpace {
// naše funkcije i klase
}
```

Pojedinim imenima unutar imenskog prostora može se pristupati kvalifikacijskim operatorom ::.

```
myStackSpace::topSub
```

Imenski prostor se ne mora navoditi ako je ime proglašeno dostupnim direktivom using. U prvoj varijanti, sa using se proglašava dostupnim jedan objekt.

```
using myStackSpace::topSub;
```

Druga varijanta omogućava dostupnost svih imena iz cijelog imenskog prostora.

```
using namespace myStackSpace;
```

C# takođe koristi imenske prostore.

Java Paketi

Paketi mogu sadržati više od jedne definicije klase. Klase u paketu su parcijalni prijatelji, jer mogu pristupati public i protected elementima drugih klasa.

Klase koje pripadaju nekom paketu se deklarišu koristeći direktivu package, npr

```
package stkpkg;
```

Puno ime objekta unutar paketa dostupno je koristeći navodeći ime paketa, npr:

```
stkpkg.myStack.topSub
```

Klijenti paketa mogu koristiti puno ime ili koristiti import deklaraciju, koja skraćuje ovu konstrukciju. Nakon

```
import stkpkg.myStack;
```

klasa myStack je dostupna navođenjem samo njenog imena, bez potrebe za navođenjem imena paketa.

Sve klase u paketu postaju dostupne bez navođenja imena paketa upotrebom džoker znakova u import deklaraciji.

```
import stkpkg.*;
```

Ada Paketi

Ada paketi se mogu koristiti i za enkapsulaciju imena. Paketi mogu biti definisani i u hijerarhiji sličnoj hijerarhiji datoteka.

Vidljivost članova paketa iz programske jedinice koja ih koristi može biti dobijena with klauzulom. Npr. sljedeća klauzula se mora navesti ako se želi pristupati paketu Ada.Text_IO

```
with Ada.Text_IO;
```

Elementima ovog paketa se pristupa uz navođenje imena paketa.

```
Ada.Text_IO.Put
```

Ako se želi ne navoditi ime paketa, koristi se klauzula use. Nakon ove klauzule dovoljno je navoditi Put.

```
use Ada.Text_IO;
```

Ruby moduli

Za enkapsulaciju imena, Ruby podržava klase i module. Modul tipično enkapsulira kolekcije konstanti i metoda. Moduli se, za razliku od klasa, ne mogu instancirati ili podklasirati i ne mogu definisati varijable.

Metode definisane u modulu moraju uključiti ime modula, kao što se vidi iz ovog primjera deklaracije modula.

```
module MyStuff
  PI = 3.14159265
  def MyStuff.mymethod1(p1)
    ..
   end
  def MyStuff.mymethod2(p2)
  ...
  end
end
```

Pristup sadržaju modula se postiže require metodom.

```
require 'myStuffMod'
...
MyStuff.mymethod1(x)
...
```

Rezime

Koncept apstraktnih tipova podataka i upotreba u programu je značajan korak u razvoju jezika. Dvije glavne osobine ADT su pakovanje podataka s njihovim operacijama i skrivanje informacija

Ada pruža pakete koji simuliraju ADT. C++ apstrakcija podataka se postiže klasama.Java apstrakcija podataka je slična C++

Ada, C++, Java 5.0, i C# 2005 podržavaju parameterizirane ADT. C++, C#, Java, Ada, i Ruby pružaju enkapsulacije imenovanja

11. Podrška objektno-orijentisanom programiranju

Uvod

Objektno-orijentisano programiranje (OOP) je izrazito popularna paradigma u programskim jezicima, jer je praktična za veće softverske projekte u kojima je moguće koristiti isti kod za različite namjene. Programski jezici koji podržavaju objektno orijentisano programiranje se razlikuju po svom nastanku i načinu kako je ono dodano u taj jezik.

- Određeni jezici su proceduralno orijentisani, ali koriste se i objekti kao praktičnija zamjena za neke biblioteke (Clipper, PHP)
- Jedna grupa jezika podržava proceduralno i podatkovno orijentisano programiranje (npr., Ada 95 i C++)
- Neki podržavaju funkcionalno programiranje (npr., CLOS, F#)
- Noviji jezici ne podržavaju druge paradigme OOP, ali koriste klasične imperativne strukture (npr., Java and C#)
- Neki su čisti OOP jezici (npr., Smalltalk i Ruby)

Jezik koji podržava Objektno-orijentisano programiranje treba da omogući realizaciju sljedeća tri koncepta:

- Apstraktni tipovi podataka
- Nasljeđivanje
- Polimorfizam

Pošto su apstraktni tipovi podataka opisani u prethodnom poglavlju, sada će biti riječi o nasljeđivanju i polimorfizmu.

Nasljeđivanje

Ponovna upotrebljivost koda, mogućnost da se isti programski kod koristi u raznim projektima povećava produktivnost. Apstraktni tipovi podataka su korisni u različitim projektima ali sami po sebi su teški za ponovnu upotrebu, jer ih treba bar malo mijenjati u novom projektu. Često osoba koja vrši modifikacije nije originalni autor programa, pa je prinuđena utrošiti vrijeme u analizi. Dalje, svi ADT su međusobno nezavisni i na istom nivou apstrakcije, pa se ne slažu dobro sa modelom podataka koje žele opisati.

Nasljeđivanje rješava oba gornja problema, ponovna upotreba ADT nakon manjih promjena i definisanje hijerarhija između podataka. Nasljeđivanje omogućava kreiranje novih apstraktnih tipova podataka od postojećih, kroz dijeljenje zajedničkih dijelova podataka i operacija nad njima.

Objektno-Orijentisani koncepti

Objektno orijentisano programiranje je karakteristično po svojoj terminologiji.

Apstraktni tipovi podataka se zovu **klase**. Instance klasa, tj. varijable koje su tipa neke klase, se zovu **objekti.** Klasa koja nasljeđuje elemente neke druge klase je **izvedena**

klasa ili podklasa. Klasa od koje druga klasa nasljeđuje njene elemente je **roditeljska klasa** ili nadklasa.

Na primjer, u razvoju programa za crtanje, može se definisati klasa Pisaljka, koja može da kao jedno od svojih svojstava ima boju. Specijalizovana pisaljka može biti Linija, koja nasljeđuje klasu Pisaljka, sa svojstvom boje, ali i dodanim svojstvima debljine linije, početne i krajnje tačke. Drugačija pisaljka je sprej, koji ima boju, pa takođe nasljeđuje pisaljku, ali ima i podatak o vremenu pritiska, te tački na kojoj je apliciran.

Potprogrami koji definišu operacije nad objektima se zovu **metode.** Pozivi metoda se zovu **poruke.** Poruke imaju dva dijela: ime metode i odredišni objekt. Slanje poruke se razlikuje od poziva potprograma po tome što potprogram prima podatke kao parametre ili im pristupa preko nelokalnih varijabli. Poruka se šalje objektu da pozove metodu koja uglavnom obrađuje podatke koji su unutar samog objekta. Cijela kolekcija metoda nad objektom se zove **protokol poruka** ili interfejs poruka.

U najjednostavnijem slučaju klasa nasljeđuje sve entitete roditelja, ali može dodavati nove metode i varijable. Pored nasljeđivanja metoda onakve kakve su u mnogim objektno orijentisanim jezicima klasa može modifikovati naslijeđeni metoda, zadržavajući možda njegovo ime. Kaže se da novi metoda **preklapa** naslijeđeni metoda, a da je metod roditelja **preklopljen** .

Nasljeđivanje se može kombinovati s kontrolom pristupa enkapsuliranim entitetima, pa su moguće situacije u kojim klasa može sakriti entitete od podklasa, situacije u kojim klasa može sakriti entitete od klijenata ili situacije u kojim klasa može sakriti entitete od klijenata, a da ih podklase vide.

Klase i objekti imaju po dvije vrste varijabli i metoda. **Varijable klase** su jedinstvene po klasi i svi objekti klase je dijele. **Varijable instance** su jedne po objektu. Na primjer, oznaka boje linije je varijabla instance, jer na ekranu ima više linija, dok je varijabla koja čuva ukupan broj linija varijabla klase. **Metode klase** prihvataju i obrađuju poruke namijenjene za klasu, dok **metode instance** prihvataju poruke za objekte.

Nasljeđivanje se dijeli na **prosto** i **višestruko.** Prosto nasljeđivanje dopušta da je klasa direktni nasljednik samo jedne klase, dok kod višestrukog klasa je direktni potomak dvije ili više klasa.

Mana nasljeđivanja kao mehanizma za ponovnu upotrebu je što kreira međuzavisnosti između klasa što komplikuje njihovo održavanje.

Dinamičko povezivanje

Pored apstraktnih tipova i nasljeđivanja, objektno orijentisani jezici imaju i karakteristiku polimorfizma postignutu kroz dinamičko povezivanje poruka s definicijom metoda. Na primjer, klasa A može definisati metodu draw, a klasa B koja nasljeđuje klasu A može definisati svoju metodu draw koja sa ponaša nešto drugačije. Ako korisnik klase A ima varijablu koja predstavlja referencu na objekte klase A, ona takođe pokazuje i na objekte klase B. Na izvršnom sistemu je da odredi koja verzija metode draw će se pozvati.

Kada hijerarhija klasa uključuje klase koje preklapaju metode i takve se metode zovu kroz polimorfne varijable, povezivanje na korektni metod će biti dinamičko, i u jezicima statičkog tipa.

Dinamičko povezivanje dopušta softverskim sistemima da se proširuju tokom razvoja i održavanja. Npr. katalog knjiga biblioteke treba da prikaže podatke o knjizi, njenom autoru, nazivu, žanru i ISBN broju, slika, broj strana, format papira. Sistem se može lako proširiti na katalog časopisa, koji dijele neke podatke s knjigama (slika, broj

strana, format), ali imaju neke izmjene u ispisu (nemaju ISBN broja i autora, ali imaju datum izlaska). Stoga se mogu iskoristiti postojeće metode GetPicture, GetPages i GetFormat, za unos zajedničkih podataka, a napraviti nove verzije metoda EnterData i PrintPublication za specifičnost koju imaju časopisi.

Ako je hijerarhija klasa isplanirana dovoljno duboko, da za neke klase metoda nema smisla, ali je predviđena da se kasnije može definisati, takva metoda se zove **apstraktna metoda**. Apstraktna metoda ne uključuje definiciju nego samo protokol. Ako bi imali generalnu klasu publikacija, metode GetPicture, GetPages i GetFormat bi u njoj bile apstraktne (jer su moguće i publikacije u formi zvučnih zapisa za slijepe), ali bi bile definisane u klasama Book i Magazine.

Apstraktna klasa je ona koja uključuje bar jedan apstraktni metod i ne može se praviti njena instanca

Pitanja dizajna OOP jezika

Objektno orijentisani jezici postavljaju sljedeća pitanja dizajna, koja će biti diskutovana u narednim tačkama:

- Da li su svi tipovi podataka objekti
- Da li su podklase podtipovi?
- Kakva je provjera tipova i polimorfizam
- Postoje li prosto i višestruko nasljeđivanje
- Na koji način ide alokacija i dealokacija objekata
- Da li je povezivanje dinamičko ili statičko
- Da li je dopušteno gniježdenja klasa
- Kako se vrši inicijalizacija objekata

Čisto objektno-orijentisani jezici (Smalltalk, Ruby, Eifell) definišu **sve tipove podataka kao objekte**. Nema razlike između predefinisanih i korisničkih tipova. Prednost pristupa je u eleganciji i čistoći dizajna, a mana je što su operacije nad prostim objektima (npr. cijeli broj) sporiji nego što bi bili da su ovi tipovi nezavisni od hijerarhije objekata. Kod jezika koji **dodaju objekte** na kompletan sistem tipova (C++, Objective C, Object Pascal), operacije nad prostim tipovima su brze, ali je konfuzan sistem tipova (dvije vrste entiteta) i kompleksan jezik. Treći pristup je dodati sistem tipova iz imperativnih jezika za primitivne tipove, ali **sve ostalo da budu objekti** (Java, C#) što pruža brze operacije nad prostim objektima i mali sistem tipova ali ostaje konfuzija zbog dva sistema tipova.

Da li postoji odnos uspostavljen između roditeljskog objekta i objekta podklase, tako da se svaka varijabla tipa podklase je ujedno i varijabla tipa roditeljske klase? Izvedena klasa je podtip ako ima “je” odnos s roditeljskom klasom. U većini jezika podklasa može samo dodavati varijable i metode i preklapati naslijeđene metode na kompatibilan način (metoda u naslijeđenoj klasi mora imati isti broj parametara kao u roditeljskoj klasi). Ruby dopušta brisanje metoda u nasljeđenoj klasi, i u tom slučaju nije potpuni “je” odnos između roditeljske i naslijeđene klase.

Polimorfizam može zahtijevati dinamičku provjeru parametara metoda i povratne vrijednosti (Ruby, PHP). Dinamička provjera tipova je spora i odlaže detekciju grešaka. Ako su preklopljene metode ograničene na iste tipove parametara i povratni tip, provjere se mogu raditi statički

Višestruko nasljeđivanje dopušta da klasa nasljeđuje podatke i metode iz dvije ili više klasa koje nisu u istom stablu klasa. To omogućava prirodnije modeliranje nekih podataka. Npr. klasa KinderJaje može biti istovremeno nasljednik klase Čokolada i klase Igračka. Ipak postoje mane višestrukog nasljeđivanja. Višestruko nasljeđivanje povećava jezičku i implementacijsku kompleksnost zbog sukoba imena. Na primjer ako su obije klase Čokolada i Igračka definisale metodu PostaviBoju, nejasno je od koje klase se preuzima ova metoda u klasi KinderJaje. Drugi nedostatak je potencijalna neefikasnost – dinamičko povezivanje košra više za višestruko nasljeđivanje, ali ne mnogo (za C++ dodatno sabiranje i pristup nizu).

Alokacija i dealokacija postavlja dva pitanja dizajna. Prvo pitanje je odakle se objekti alociraju? Dva tipična mjesta su heap dinamički i stek dinamički objekti. Ako su svi objekti heap-dinamički, reference se mogu ujediniti kroz pointere ili referencne varijable Ako su objekti stek dinamički postoji problem s odsjecanjem pri dodjeli podtipa roditelju. U sljedećem primjeru u C++, kopiranje varijable bl u varijablu al će odsjeći dodatne elemente koje je uvela klasa B.

```
class A { int x; }
class B : A { int y ;}
void test() {
  A a1;
  B b1;
  a1=b1 ;
;
```

Drugo pitanje vezano za alokaciju i dealokaciju je da li je dealokacija eksplicitna ili implicitna? Ako je implicitna, mora postojati način prepoznavanja nekorištenih objekata, a ako je eksplicitna javlja se rizik visećih pointera.

Da li povezivanje poruka na metode treba biti dinamičko? Ako nije, gube se prednost dinamičkog povezivanja. Ako jeste, to je manje efikasno od statičkog povezivanja. U nekim jezicima, korisnik može da specificira koje metode će se povezivati dinamički a koje statički.

Ako je nova klasa potrebna samo jednoj klasi nema razloga da je definišemo tako da je druge vide. Može li se nova klasa gnijezditi unutar klase koja je koristi? U nekim slučajevima, nova klasa se gnijezdi unutar potprograma umjesto unutar druge klase

Pitanje kako dobijaju elementi klase početne vrijednosti pri kreiranju, postavlja nova pitanja. Da li se inicijalizacija implicitno obavlja ili je potrebna eksplicitna inicijalizacija? Pri inicijalizaciji podklase, da li se automatski poziva inicijalizacija roditeljske klase ili to mora programer navesti.

Provjera tipova i polimorfizam

Povezivanje poruka s metodama je dinamičko. Kada se pošalje poruka objektu, proces traži metodu. Ako nije nađena traži se nadklasa i tako redom do klase system koja nema nadklasa. Jedina provjera tipova u Smalltalk je dinamička i jedina greška tipova se događa kada se pošalje poruka objektu koji nema odgovarajuće metode.

Nasljeđivanje

Smalltalk podklase nasljeđuju sve varijable instance, metode instance i metode klase njegovih nadklasa. Sve podklase su podtipovi (ne mogu se sakriti metode u roditeljskim klasama). Svo nasljeđivanje je implementacijsko nasljeđivanje.

Preklopljenim metodama se pristupa pseudovarijablom super.. Ne postoji višestruko nasljeđivanje

Ocjena za Smalltalk

Sintaksa jezika je jednostavna i regularna. Dobar primjer snage koju pruža mali jezik (ali velika biblioteka). Spor je u poređenju s konvencionalnim kompajliranim imperativnim jezicima. Problem pouzdanosti: Dinamičko povezivanje dopušta da greške tipova budu neprepoznate do izvršenja. Značaj ovog jezika: uveo je grafički korisnički interfejs i uticao na napredak OOP

Podrška OOP u C++

C++ je razvijen iz C i SIMULA 67. On je među najkorištenijim OOP jezicima. Podržava klasičnu imperativnu paradigmu (kroz varijable i funkcije) kao i OOP. Objekti mogu biti statički, stek dinamički i heap dinamički. Ako su heap dinamički potrebna je eksplicitna dealokacija objekata koristeći delete operator.

Definicije klasa mogu uključivati konstruktore i destruktore koji se implicitno pozivaju prilikom kreiranja ili uništavanja objekata.

Nasljeđivanje

Klasa ne mora biti podklasa druge klase, može biti korijen stabla hijerarhije klase. Razrađena je kontrola pristupa entitetima klase.

Kontrole pristupa za članove su

- Private (vidljivo samo u klase i prijateljima)
- Public (vidljivo u podklasama i klijentima)
- Protected (vidljivo u podklasama ali ne i klijentima)

Pored ovoga, proces podklasiranja se može deklarisat s kontrolom pristupa (private ili public), koje definišu potencijalne promjene u pristupu preko podklasa

- Private izvođenje – naslijeđeni public i protected članovi su privatni u podklasama
- Public izvođenje public i protected članovi su public i protected u podklasama

Primjer nasljeđivanja u C++

```
class base_class {
private:
  int a;
  float x;
protected:
  int b;
  float y;
public:
  int c;
  float z;
};
class subclass_1 : public base_class { … };
//  b i y su protected a
// c i z su public
class subclass_2 : private base_class { … };
// U ovom slučaju, b, y, c, i z su private,
// I nijedna izvedena klasa nema pristupa privatnim
// elementima base_class
```

Private izvođenje čini da podklase nisu podtipovi. Član koji nije dostupan u podklasi (zbog private izvođenja) može se deklarisati da postane vidljiv koristeći operator razlučenja opsega (::), npr.,

```
class subclass_3 : private base_class {
base_class :: c;
…
}
```

Kada je potrebno private izvođenje? Klasa pruža članove koji moraju biti vidljivi, pa su definisani kao javni članovi. Izvedena klasa dodaje nove članove ali ne želi da njeni klijenti vide članove roditeljske klase čak i ako su oni bili javni u definiciji roditeljske klase. Npr. klasa Krug je specijalni slučaj klase Elippse, ali se želi spriječiti da korisnik pozove metodu SetAxes(int x, int y) nad objektima tipa Krug, jer bi krug postao nešto što više nije krug.

U C++ podržano je višestruko nasljeđivanje.

```
class Thread { . . . };
class Drawing { . . . };
class DrawThread : public Thread, public Drawing { . . . };
```

Ako postoje dva naslijeđena člana s istim imenom može im se prići operatorom razlučivanja opsega :: .

Dinamičko povezivanje

U C++ dinamičko povezivanje je opcionalno. Metod se može definisati kao virtual, što znači da se može pozivati polimorfnim varijablama i dinamički povezati s porukama. Neka su definisane sljedeće klase:

```
class Shape {
public:
virtual void draw() = 0;
...
};
class Circle : public Shape {
public:
void draw() { . . . }
...
};
class Rectangle : public Shape {
public:
void draw() { . . . }
void fill() { . . . }
...
};
class Square : public Rectangle {
public:
void draw() { . . . }
...
};
```

Sljedeća upotreba ovih klasa ilustruje poziv i sa dinamički povezanim i sa statički povezanim metodama.

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq; // ptr_shape pokazuje na  Square objekt
ptr_shape->draw();//Dinamički vezan draw u Square klasi
rect->fill();//Statički vezan fill u Rectangle klasi
```

Virtualna metoda čija je definicija postavljena na nulu (draw u Shape) je čisto virtualna klasa. Klasa koja ima barem jednu virtualnu funkciju se zove apstraktna klasa i od ovih klasa se ne može praviti instanca.

Ocjena

C++ ima razvijenu kontrolu pristupa (za razliku od Smalltalk). C++ omogućava višestruko nasljeđivanje. U C++, programer mora odlučiti u trenutku dizajna koje metode će statički a koje dinamički vezati. C++ je manje fleksibilan u pravljenju generičkih medoda od Smalltalk čija je provjera tipova dinamička, ali zbog interpretiranja i dinamičkog povezivanja, Smalltalk je često deset puta sporiji od C++.

Podrška OOP u ObjectiveC

Nastao u približno isto vrijeme kada u C++, ObjectiveC također ima primitivne tipove i objekte. No, dok sintaksa objekata u C++ je bazirana na sintaksi C (poziv metoda je modeliran slično pozivu funkcija) u ObjectiveC rad s objektima liči na slanje poruka u Smalltalku.

Nasljeđivanje

ObjectiveC podržava samo prosto nasljeđivanje, svaka klasa mora imati roditeljsku,(osim najviše NSObject) koja se navodi u interface direktivi

```
#import "Stack.h"
@interface Stack (StackExtend)
-(int) secondFromTop;
-(void) full;
@end
```

Kao u Smalltalk svako ime metode se može pozvati u svakom objektu. Poziv metode se vrši u srednjim zagradama.

Umjesto višestrukog nasljeđivanja postoji mogućnost proširivanja klasa sa još dvije konstrukcije; kategorije i protokoli

Kategorije (zovu se i mixini) predstavljaju dodatne interfejse klasa. Njima se dodaju nove metode postojećoj klasi. Sljedeći primjer predstavlja kategoriju StackExtend klase Stack.

```
#import "Stack.h"
@interface Stack (StackExtend)
-(int) secondFromTop;
-(void) full;
@end
...
@implementation Stack (StackExtend)
```

Protokoli su slični apstraktnim klasama. Oni predstavljaju listu deklaracija metoda koje se trebaju implementirati

```
@protocol MatrixOps
-(Matrix *) add: (Matrix *) mat;
-(Matrix *) subtract: (Matrix *) mat;
@optional
-(Matrix *) multiply: (Matrix *) mat;
@end
```

Klasa koja koristi metode definisane u protokolu se deklariše navođenjem njegovog imena unutar < i > znakova.

```
@interface MyMatrixClass: NSObject <MatrixOps>
```

Dinamičko povezivanje

Dinamičko povezivanje je znatno drugačije implementirano u odnosu na većinu objektno orijentisanih jezika. Ono se postiže varijablom tipa id. Kada je varijabla ovog tipa, ona može pristupiti bilo kojem objektu. Takvoj varijabli se može dodijeliti objekt i pozvati njegova metoda ako postoji.

```
Circle *myCircle = [[Circle alloc] init];
Square *mySquare = [[Square alloc] init];
[myCircle setCircumference: 5];
[mySquare setSide: 5];
// Create the id variable
id shapeRef;
//Set the id to reference the circle and draw it
Support for Object-Oriented Programming
shapteRef = myCircle;
[shapeRef draw];
// Set the id to reference the square
shapeRef = mySquare;
[shapeRef draw];
```

Ocjena

OOP u ObjectiveC je adekvatan, ali ima nedostataka. Implemetacija polimorfizma kroz id varijable je nepraktična jer je dopušten pristup svim objektima, ne samo onim iz legalne hijerarhije. Kategorije i protokoli pružaju mogućnosti slične višestrukom nasljeđivanju

Podrška OOP u Java

U ovom jeziku svi podaci su objekti osim primitivnih tipova. No i svi primitivni tipovi imaju omotačke klase koje smještaju jedan podatak. Svi objekti su heap-dinamički, pristupa im se kroz referencne varijable, i većina se alocira s new. Metoda finalize se implicitno poziva kada skupljač smeća treba da oslobodi prostor koji je objekt zauzeo, ali njen trenutak izvršavanja nije predvidiv.

Nasljeđivanje

Jedna od razlika nasljeđivanja u Java u odnosu na C++ je što Java poznaje final metode. Ako je metoda definisana s final, ona ne može biti preklopljena u naslijeđenoj klasi.

Podklase su podtipovi, stoga, za razliku od C++ private derivacije nisu podržane.

Podržano je samo prosto nasljeđivanje ali postoji apstraktna kategorija klasa koja pruža neke prednosti višestrukog nasljeđivanja (interface). Interfejs može uključiti samo deklaracije metoda i imenovane konstante, tj.,

```
public interface Comparable <T> {
   public int comparedTo (T b);
}
```

Dinamičko povezivanje

U Java, sve poruke se dinamički vežu za metode. Statičko povezivanje se koristi ako je metoda final, static ili private tj ne može se preklopiti pa dinamičko povezivanje nema svrhe)

Ugniježdene klase

Java podržava tri načina za ugniježdene klase: unutrašnje klase, statičke ugniježdene klase i lokalne ugniježdene klase. Sve su skrivene od svih klasa u svojim paketima osim one klase unutar koje su definisane.

Nestatičke klase koje se direktno gnijezde se zovu unutrašnje klase. Unutrašnja klasa može pristupiti članovima u kojoj je definisana.

Statička ugniježdena klasa ne može pristupiti članovima klase unutar koje je definisana.

Lokalna ugniježdena klasa je definisana unutar metoda neke klase. Njeni članovi su vidljivi unutar te metode i nemaju specifikatora pristupa. Ovakva klasa može pristupiti članovima klase u kojoj je definisana.

Ocjena

Ciljevi podrške za OOP u Java su slični kao C++. Nema podrške za proceduralno programiranje. Sve klase moraju imati roditelje. Dinamičko povezivanje se koristi kao "normalan" način za povezivanje definicije metode i poziva. Umjesto višestrukog nasljeđivanja koriste se interfejsi.

Podrška OOP u C#

Podrška za OOP u C# je slična kao Java. C# uključuje i class i struct konstrukcije. Klase se slično definišu kao u Java. struct su manje moćne stek dinamičke konstrukcije koje nemaju nasljeđivanje.

Nasljeđivanje

Sve C# klase se izvode iz korijenske klase, Object. Sintaksa je slična C++ za definisanje klasa

```
public class NewClass : ParentClass { . . . }
```

Metoda naslijeđena od roditeljske klase se može zamijeniti u izvedenoj klasi oznakom definicije navođenjem oznake new. Verzija u roditeljskoj klasi se može zvati prefiksom base:

```
base.Draw()
```

Dinamičko povezivanje

Da se omogući dinamičko povezivanje metoda, koristi se pristup sličan u C++. Metoda u baznoj klasi se označi kao virtual, a metode u izvedenoj klasi se označe kao override

Apstraktne metode se označavaju ključnom rječju abstract i moraju se implementirati u svim podklasama.

abstract public void Draw();

Ako klasa uključuje apstraktne metode, tada se i cijela klasa mora označiti sa abstract.

Gniježdene klase

C# klasa koja se direktno definiše u nekoj klasi ponaša se kao Java statička ugniježdena klasa. C# podržava ugniježdene koje se ponašaju kao nestatičke klase u Java

Ocjena

C# je najnoviji c-bazirani OO jezik. Razlike između C# i Java podrške za OOP su relativno minorne, uz poboljšanje dodavanjem struct.

Podrška OOP u ObjectPascal/Delphi

Postoje dvije vrste deklaracija klasa: object (sa eksplicitnim dereferenciranjem pointera na objekat kao u C++) i class (sa implicitnim kao u Java). Klase deklarisane kao object mogu biti statičke, stek dinamičke i heap dinamičke, dok su klase deklarisane kao class samo heap dinamičke. Još jedna ADT jedinica je unit sa interfejsnim i implementacijskim dijelom koja je slična Ada paketima.

Nasljeđivanje

Klase ne moraju imati roditelja. Podržane su apstraktne klase. U Class podržane su kategorije kontrola pristupa poznate iz C++: public, private, protected i nova kategorija published koja datog člana eksportuje u grafički interfejs.

Nema višestrukog nasljeđivanja, ali su podržani interfejsi

Ako dvije metode imaju isto ime, potrebna ključna riječ overload

Dinamičko povezivanje

Podrazumijevano povezivanje metoda je statičko. Ključnom riječju virtual ili dynamic u roditeljskoj odnosno override u naslijeđenoj klasi postiže se dinamičko povezivanje.

Podrška OOP u ObjectPascal/Delphi

```
// Full Unit code.
// ---------------------------------------------------------
// You must store this code in a unit called Unit1 with a form
// called Form1 that has an OnCreate event called FormCreate.
unit Unit1;
interface
uses
Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs;
type
// Class with overloaded methods
  TRectangle = class
  private
    coords: array[0..3] of Integer;
  public
    constructor Create(left, top, right, bottom : Integer);
    Overload;
  published
// Only one version of an overloaded method can be published
    constructor Create(topLeft, bottomRight : TPoint); Overload;
end;
implementation
// Constructor version 1 : takes integers as the coordinate values
constructor TRectangle.Create(left, top, right, bottom: Integer);
begin
// Save the passed values
  coords[0] := left;
  coords[1] := top;
  coords[2] := right;
  coords[3] := bottom;
end;
// Constructor version 2 : takes points as the coordinate values
constructor TRectangle.Create(topLeft, bottomRight: TPoint);
begin
// Save the passed values
  coords[0] := topLeft.X;
  coords[1] := topLeft.Y;
  coords[2] := bottomRight.X;
  coords[3] := bottomRight.Y;
end;
```

Ocjena:

ObjectPascal je relativno siguran jezik dosta čitljivog koda. Mogućnosti OOP su približne C++. Delphi je striktno vezan za GUI

Podrška OOP u Ruby

Kao i u Smalltalk u Ruby sve je objekt i sav račun se obavlja kroz prosljeđivanje poruka. I aritmetički operatori predstavljaju pozive metoda. Npr a+5 je efektivno a.+ 5 .

Definicije klasa u Ruby se razlikuju od C++ i Java po tome što su izvršive, dopuštajući sekundarnim definicijama da dodaju članove postojećim definicijama . Definicije metoda su također izvršive što omogućava programeru da bira između različitih verzija metoda prostim ubacivanjem u if...then konstrukcije.

Sve varijable su reference na objekte bez definisanih tipova. Imena varijabli instance počinju znakom @.

Objekt se kreira navodeći ključnu riječ new, koja poziva konstruktor. Konstruktor se obično zove initialize.

Kontrola pristupa je različita za podatke i metode. Podaci instance su privatni i ne mogu se mijenjati. Metode mogu biti javne, privatne ili zaštićene (protected). Pravo pristupa metodu se provjerava pri izvršenju

Ako je potrebno pristupati varijabli instance moraju se definisati metode pristupa (getter i setter)

```
class MyClass
# A constructor
def initialize
  @one = 1
  @two = 2
end
# A getter for @one
def one
  @one
end
# A setter for @one
def one=(my_one)
  @one = my_one
end
end
# of class MyClass
```

Nasljeđivanje

Kontrola pristupa nasljeđenom metodu može biti drugačija nego u roditeljskoj klasi

```
class MySubClass < BaseClass
```

Konstruktor nasljeđene klase može pozvati konstuktora iz roditeljske klase navodeći riječ super.

Zbog potpuno dinamičke mogućnosti dodavanja i uklanjanja metoda iz klase, podklase ne moraju biti podtipovi

Dinamičko povezivanje

Sve varijable su bez tipova i polimorfne. Stoga je povezivanje metoda s porukama isključivo dinamičko.

Ocjena

Ruby je čisto OO jezik i stoga je podrška OOP apsolutno adekvatna. Ipak, ne podržava apstraktne klase, ne podržava u potpunosti višestruko nasljeđivanje a kontrola pristupa slabija nego kod drugih jezika koji podržavaju OOP. Također, kao interpreterski jezik razmjerno je sporiji od kompajliranih poput C++

Implementacija OO konstrukcija

Pri implementaciji OO jezika treba obratiti pažnju na dva bitna elementa:

- Strukture memorije za varijable instance
- Dinamičko povezivanje poruka metodama

Čuvanje podataka instance

Zapisi instance klase (CIRs) čuvaju stanje objekta. Oni mogu biti poznati statički (u trenutku kompajliranja). Sastoje se od svih varijabli instance koje su dostupne. Ako klasa ima roditelja varijable instance podklase se dodaju na roditeljski CIR, u memoriji odmah iza njih. Pošto je CIR statičan, pristup varijablama instance se radi kao da je u pitanju pristup slogovima.

Dinamičko povezivanje metoda klase

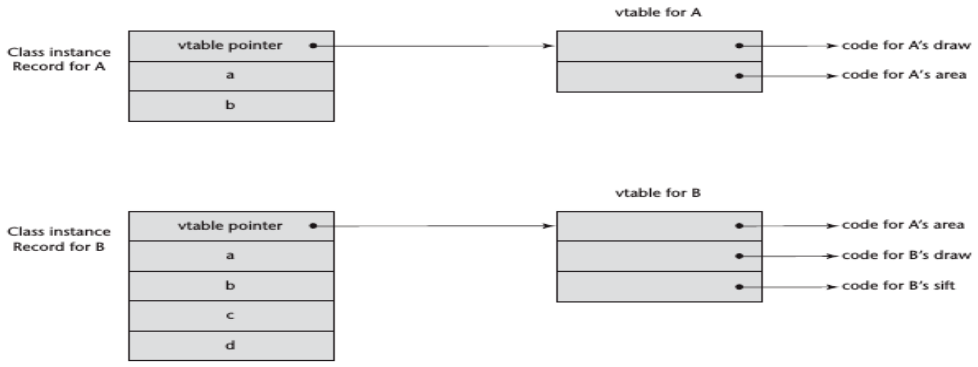
Metode u klasi koje su statički povezane ne trebaju biti ugrađene u CIR. One se implementiraju kao obični potprogrami, koji imaju jedan dodatni parametar u pozivu: pointer na objekt.

Metode koje će se dinamički povezati moraju imati elemente u CIR No, kako je lista metoda zajednička za sve objekte klase, potrebno ju je u memoriji držati na samo jednom mjestu. Unutar instanci klase u CIR treba postaviti pointer na listu dinamički povezanih metoda. Memorijska struktura u kojoj se ova lista čuva se ponekad zove tabela virtualnih metoda (vtable). Pozivi metoda se mogu predstaviti kao pozicije od početka vtable.

Neka je dat sljedeći Java primjer,

```
public class A {
public int a, b;
public void draw() { . . . }
public int area() { . . . }
}
public class B extends A {
public int c, d;
public void draw() { . . . }
public void sift() { . . . }
}
```

Zapisi instanci klasa izgledaju kao na sljedećoj slici.



Kako metoda area nije predefinisana u klasi B, njen pokazivač na metodu i dalje pokazuje na metodu iz klase A.

Rezime

- OO programiranje uključuje tri fundamentalna koncepta ADT, Nasljeđivanje, Dinamičko povezivanje.

- Problemi dizajna suz ekskluzivnost objekata, podklase i podtipovi, provjera tipova i polimorfizam, prosto i višestruko nasljeđivanje, Dinamičko povezivanje, eksplicitna i implicitna dealokacija objekata, i gniježdene klase.
- Smalltalk je čisti OOL
- C++ ima dva odvojena sistema tipova (hibrid)
- Java nije hibridni jezik kao C++; ona podržava samo OO programiranje
- C# je baziran na C++ i Java
- Ruby je novi čisti OOP jezik; pruža neke nove ideje u podršci OOP
- JavaScript nije OOP jezik ali pruža interesantne varijacije
- Implementacija OOP zahtijeva nove strukture podatakaa

12. Podrška asinhronim događajima

Moderni softver sve više se piše u okruženjima koja nisu prosto sekvencijalno izvršavanje programa, karakteristično za imperativne programske jezike. Sve češća je potreba da se pojedini dijelovi programa izvršavaju istovremeno (konkurentno) ili da se u programu dešavaju događaji inicirani od strane korisnika u bilo kom trenutku. Programi treba da znaju i reagovati na greške koje se dešavaju pri izvršavanju.

Konkurentnost

Konkurentno izvršavanje se dešava na četiri nivoa: mašinske instrukcija (dvije ili više instrukcija izvršava se istovremeno), naredbe u višem programskom jeziku (dvije ili više naredbe istovremeno), nivo jedinice (dva ili više potprograma se istovremeno) i nivo programa (dva ili više program se istovremeno). Programski jezici se ne bave konkurentnošću na nivou instrukcije i programa.

Konkurentnost može biti fizička, logička i kvazi. Fizička konkurentnost se javlja kada više nezavisnih procesora izvršavaju istovremeno različite programe ili dijelove programa. Logička konkurentnost se dešava kada se dijeli vrijeme jednog procesora između različitih programa ili njihovih niti. Kvazikonkurentnost se dešava kod korutina.

Konkurentnost na nivou potprograma

Task je programska jedinica koja se izvršava istovremeno s drugim programskim jedinicama. Taskovi se razlikuju od običnih potprograma što:

- Mogu se implicitno započeti
- Kada programska jedinica započne izvršenje taska, ona nije nužno suspendovana.
- Kada se završi izvršenje taska, kontrola se ne mora vratiti pozivaocu

Dvije glavne kategorije taskova razlikuju se po adresnom prostoru u kome se izvršavaju. Teški taskovi (procesi) izvršavaju se u vlasitom adresnom prostoru. Laki taskovi (niti) svi rade u istom adresnom prostoru.

Sinhronizacija taskova

Task je nevezan ako ne komunicira niti utiče na izvršenje drugih taskova u programu ni na koji način. Često je potrebno, međutim, uvesti sinhronizaciju, mehanizam koji kontroliše redoslijed taskova. Sinhronizacija se postiže pomoću dijeljenih nelokalnih varijabli, parametara i prosljeđivanja poruka. Dvije vrste sinhronizacije su kooperativna sinhronizacija i kompetitivna sinhronizacija.

Kooperativna sinhronizacija je potrebna u situacijama kada Task A mora čekati da task B završi aktivnost prije nego task A može nastaviti izvršenje, npr., problem proizvođača i potrošača

Kompetitivna sinhronizacija je potrebna kada dva ili više taska moraju koristiti neki resurs koji se ne može istovremeno koristiti npr. dijeljeni brojač. Sinhronizacija u ovom slučaju se postiže uzajamnim isključivanjem

Kontrola izvršenja task-ova se vrši programom koji se zove raspoređivač koji mapira izvršenje taska na dostupne procesore

Najčešće metode za sihronizaciju su semafori, monitori i poruke

Semafori

Semafor je struktura podataka koja se sastoji od varijable i red za spremanje zadataka deskriptora. Semafor se može koristiti za implementaciju stražara na kodu koji pristupa zajedničke strukture podataka. Semafori imaju samo dvije operacije, čekanje i ispuštanje (izvorno nazvan P (wait) i V (signal) po Dijkstra. Semafori se mogu koristiti za konkurenciju i saradnju

Wait operacija radi po sljedećem algoritmu

```
wait(aSemaphore):
if aSemaphore.value > 0 then
   aSemaphore.value= aSemaphore.value-1
else
   put the caller in aSemaphore's queue
   attempt to transfer control to a ready task
end
```

Signal operacija

```
signal(aSemaphore):
if aSemaphore's queue is empty then
   aSemaphore.value= aSemaphore.value +1
else
   put the calling task in the task ready queue
   transfer control to a task from aSemaphore's queue
end
```

Semafor su jednostavnog principa i često su raspoloživi kao funkcija biblioteka operativnog sistema. Mana im je što je lako pogriješiti u njihovoj upotrebi u sinhronizaciji.

Monitori

Monitori postoje u Ada, Java, C#. To je apstrakni tip podataka za dijeljene podatke koji uokviri dijeljene podatke u ograničeno područje.

Monitori automatski rješavaju kompetitivnu sinhronizaciju. Dijeljeni podaci su u monitoru (umjesto u klijentskim jedinicama). Sav pristup ovim podacima je rezidentan u monitoru. Monitorska implementacija garantuje sinhronizovan pristup dopuštajući jedan pristup u trenutku. Poziv monitorskih procedura se stavi u red ako je monitor zauzet u trenutku poziva

Kooperacijska sinhronizacija je i dalje programerski zadatak.

Prosljeđivanje poruka

Prosljeđivanje poruka je model za konkurentnost. On može modelirati semafore i monitore, i može se koristiti za obije vrste sinhronizacije. Komunikacija taskova je kao posjeta ljekaru—ili on čeka vas ili vi njega, ali kada ste obojica spremni sastanete se. Slično poruke se mogu slati samo u trenutku kada ih primalac očekuje, i to jedna po jedna. Kada se pošiljaočeva poruka prihvati od primaoca, konkretni prijenos se zove rendezvous.

Za podršku konkurentnosti kroz prosljeđivanje poruka, jezik treba mehanizam da se dopusti da task kaže kada može primati poruke i način za zapamtiti ko čeka na primljene poruke i pošten način za izbor poruka

Podrška konkurentnosti u Ada

U Ada 83 uključen je model konkurentnosti baziran na prosljeđivanju poruka. Taskovi su jedinica konkurentnosti. Ada taskovi imaju specifikaciju i tijelo kao paketi. Specifikacija taska definiše interfejs:

```
task Task_Example is
  entry Entry_1 (Item : in Integer);
end Task_Example;
```

U tijelu taska opisuje se akcija koja se dešava kada bude rendezvous. Task koji pošalje poruku se suspenduje čekajući da se poruka prihvati. U narednom primjeru, task se izvršava do vrha accept klauzule i čeka poruku.

```
task body Task_Example is
   begin
   loop
   accept Entry_1 (Item: in Float) do
   ...
   end Entry_1;
   end loop;
 end Task_Example;
```

Poruka se šalje pozivom potprograma navedenog uz klauzulu entry u specifikaciji taska. Tokom izvršenja accept klauzule, pošiljalac je suspendovan i prelazi na izvršavanje tijela obrađivača poruke navedenom uz accept klauzuli.

```
Task_Example.Entry_1(2.8)
```

Parametri accept klauzule mogu prenijeti informacije u bilo kom ili oba smjera. Svaka accept klauzula ima pridružen red u kome se smještaju poruke na čekanju.

Task koji ima accept klauzule bez drugog koda se zove server task (kao u primjeru gore).

Task bez accept klauzula se zove actor task. Actor task može slati poruke drugim taskovima. Pošiljalac mora znati entry ime primaoca, ali primalac ne mora znati o pošiljaocu.

Task može imati i više entry tačaka. Tada specifikacijski task ima entry klauzulu za svaku, dok tijelo taska ima accept klauzulu za svaku entry klauzulu smještenu u select klauzulu koja je u petlji. Sljedeći primjer pokazuje task s više ulaza

```
Task Body Teller is
   loop
     select
       accept Drive_Up(formal params) do
       …
       end Drive_Up;
       …
     or
       accept Walk_Up(formal params) do
       …
       end Walk_Up;
       …
     end select;
   end loop;
 end Teller;
```

Semantika Taskova s više accept klauzula je sljedeća:

- Ako je tačno jedan entry red neprazan uzmi poruku iz njega
- Ako je više od jednog entry redova neprazno odaberi jedan nedeterministički s koga se uzima poruka
- Ako su svi prazni, čekaj

Ova konstrucija se zove selektivno čekanje

Praćene accept klauzule predstavljaju accept klauzula čuvarima. Klauzula accept može da ima čuvara, koji se navodi u klauzuli when ispred nje.

```
when not Full(Buffer) =>
  accept Deposit (New_Value) do
     …
   end
```

Klauzula accept sa when klauzulom je otvorena ili zatvorena. Klauzula čiji čuvar je true se zove otvorena. Klauzula čiji čuvar je false se zove zatvorena. Klauzula bez čuvara je uvijek otvorena

Semantika select s praćenim accept klauzulama je sljedeća:

- select prvo provjeri čuvare na svim klauzulama
- Ako je tačno jedan otvoren, čeka se njegov red čekanja za poruke
- Ako ih je više od jednog, nedeterministički odaberi red čekanja iz koga se uzimaju poruke
- Ako su sve zatvorene, a nema else klauzule to je greška
- Klauzula select može uključiti klauzulu else za izbjegavanje greške
- Ponavlja se petlja

Primjer taska s praćenim accept klauzulama sa benzinskom stanicom koja je nepristupačna ako je garaža puna ili ako nema goriva (specifikacijski dio)

```
task Gas_Station_Attendant is
  entry Service_Island (Car : Car_Type);
  entry Garage (Car : Car_Type);
end Gas_Station_Attendant;
```

Implementacijski dio:

```
Task body Gas_Station_Attendant is
  begin
    loop
      select
        when Gas_Available =>
        accept Service_Island (Car : Car_Type) do
          Fill_With_Gas (Car);
        end Service_Island;
      or
        when Garage_Available =>
        accept Garage (Car : Car_Type) do
          Fix (Car);
        end Garage;
      else
      Sleep;
      end select;
    end loop;
  end Gas_Station_Attendant;
```

Izvršenje taska je završeno ako je kontrola došla do kraja. Ako task nije kreirao zavisne taskove i završio je, on je prekinut. Ako je task kreirao zavisne taskove i završio je, neće biti prekinut dok zavisni taskovi ne budu prekinuti

Task se može završiti i terminate klauzulom. Ona se navodi bez parametara i odabira kada nema otvorenih accept klauzula. Kada je terminate odabrana u tasku, task se završava samo kada je njegov master i svi taskovi koji su vezani za master završili ili čekaju na terminate klauzulu. Potprogram se ne napušta dok svi njegovi zavisni taskovi nisu završili.

Prioritet svakog taska se može postaviti s

```
 pragma priority
 pragma Priority (expression);
```

Prioritet taska se uzima u obzir kada je task u redu čekanja

Konkurentnost u Ada 95

Ada 95 uključuje Ada 83 osobine za konkurentnost uz dodatak dvije nove, zaštićene objekte i asinhronu komunikaciju.

Zaštićeni objekti su efikasniji način implementacije dijeljenih podataka koji dopušta pristup dijeljenim podacima bez rendezvous. Zaštićeni objekt je apstraktni tip podatka, ali je bliži monitoru nego tasku. Pristup zaštićenom objektu je kroz poruke ili zaštićene potprograme. Zaštićena procedura pruža uzajamno isključiv pristup čitanja/pisanja zaštićenim objektima. Zaštićena funkcija pruža konkurentni samo za čitanje pristup zaštićenim objektima

```
protected Buffer is
 entry Deposit(Item : in Integer);
 entry Fetch(Item : out Integer);
 private
     Bufsize : constant Integer := 100;
     Buf : array (1..Bufsize) of Integer;
     Filled : Integer range 0..Bufsize := 0;
     Next_In,
     Next_Out : Integer range 1..Bufsize := 1;
end Buffer;
protected body Buffer is
 entry Deposit(Item : in Integer)
  when Filled < Bufsize is
  begin
     Buf(Next_In) := Item;
     Next_In := (Next_In mod Bufsize) + 1;
     Filled := Filled + 1;
end Deposit;
entry Fetch(Item : out Integer) when Filled > 0 is
begin
   Item := Buf(Next_Out);
   Next_Out := (Next_Out mod Bufsize) + 1;
   Filled := Filled - 1;
end Fetch;
end Buffer;
```

Ocjena Ada

Model prosljeđivanja poruka za realizaciju konkurentnosti je moćan i generalan. Zaštićeni objekti su bolji način za sinhronihzovane dijeljene podatke. Ako nisu distribuirani procesori izbor između monitora i taskova s prosljeđivanjem poruka je stvar ukusa. Za distribuirane sisteme, prosljeđivanje poruka je bolji model za konkurentnost

Java niti

Konkurentna jedinice u Javi su metode po imenu run, čiji kod može biti u istovremenom izvršenju s drugim takvim metodama (drugih objekata) i sa glavnom metodom. Proces u kome se metoda run izvršava se zove nit. Java niti su lagani taskovi, što znači da su svi oni rade u istom adresnom prostoru. Ovo je drugačije od Ada zadataka, koji su u teškoj kategoriji taskova (oni rade u svom adresnom prostoru). Važna razlika je u tome da niti zahtijevaju daleko manje resursa od Ada taskova.

Postoje dva načina za definiranje klase s metodom run. Jedan od njih je da se definiše potklasa od predefinirane klase Thread i preklopi svoju metodu run.

U Adi, zadaci mogu biti akteri ili serveri i zadaci komuniciraju jedni sa drugima kroz accept klauzule. Java run metode su svi akteri i ne postoji mehanizam za njih da komuniciraju jedni s drugima, osim kroz metodu join i kroz zajedničke podatke.

Klasa Thread nije prirodni roditelj bilo koje druge klase. Ona daje neke usluge za svoje podklase, ali se ne odnosi na njihove računske sposobnosti. Thread je jedina klasa na raspolaganju za stvaranje konkurentnih programa u Java. Klasa Thread uključuje pet

konstruktora i zbirku metoda i konstanti. Metoda run, koja opisuje izvršnje niti, uvijek je preklopljena podklasom Thread. Metoda start klase Thread počinje svoju nit kao istovremena jedinica pozivom svoje run metode. Neobično u toj kontroli je što se poziv vraća odmah pozivatelju, koji onda nastavlja svoje izvršenje, paralelno sa novo započetom metodom run. Tipični izgled klase koja je izvedena iz Thread izgleda ovako.

```
Class myThread extends Thread
 public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Klasa Thread ima više metoda koje kontrolišu izvršenje niti

- yield je zahtjev iz izvršne niti da se procesor dobrovoljno prepusti
- sleep metoda se može koristiti od pozivaoca metode da blokira nit
- join metoda se koristi da se metoda natjera da odloži izvršenje dok se run metoda druge niti ne završi

Podrazumijevani prioritet niti je kao prioritet one koja ga je kreirala. Ako main kreira nit onda je nit prioriteta NORM_PRIORITY. Postoje i MAX_PRIORITY te MIN_PRIORITY. Prioritet niti se može mijenjati metodom setPriority

Kompetitivna sinhronizacija uz Java niti postiže se specijalnim modifikatorom synchronized. Metod koji uključije synchronized modifikator zabranjuje drugim metodama izvršavanje na objektu za vrijeme svog izvršenja.

```
…
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

Gornja dva metoda su sinhronizovana što sprečava njihovu međusobnu interferenciju. Ako samo dio metode mora da se obavlja bez interferencije, može se sinhroninizovati kroz synchronized statement

```
synchronized (expression)
  statement
```

Kooperacijska sinhronizacija u Java se postiže metodama wait, notify, i notifyAll. Sve one su definisane u Object, korijenskoj klasi u Java, pa ih svi nasljeđuju. Metoda wait mora se pozvati u petlju. Metoda notify se poziva da se kaže jednoj čekajućuj niti da se događaj na koji ona čeka dogodio. Metoda notifyAll budi sve niti na listi čekanja

Ocjena Java niti

Java podrška konkurentnosti je jednostavna ali radi svoj posao. Nije moćna kao Ada taskovi

C# niti

Niti u C# podsjećaju na Java ali s određenim razlikama. Umjesto jedne metode run(Java), svaki metod se može izvršavati u svojoj niti. Nit u C# se kreira kreiranjem Thread objekta. Kreiranje niti ne pokreće konkurentno izvršenje. Izvršemje se mora zahtijevati pozivom Start metode.

```
public void MyRun1() { . . . }
. . .
Thread myThread = new Thread(new ThreadStart(MyRun1));
...
myThread.Start();
```

Nit se može zaustaviti da čeka da se završi druga nit s Join. Npr. sljedeći poziv će čekati dok se nit B ne završi.

```
B.Join();
```

Nit može suspendovati samu sebe pozivom metode Sleep čiji je parametar broj milisekundi koliko će biti neaktivna ili prekinuti izvršenje pozivom metode Abort

Postoje tri načina sinhronizacije C# niti.

- Klasa Interlocked je korištena kada jedine operacije koje treba sinhronizovati su inkrementiranje ili dekrementiranje cijelog broja
- Naredba lock korištena je da se označi kritična sekcija koda u niti i taj dio će biti u uzajamnom isključenju.

lock (object) {... }

- Monitor klasa se koristi kada je potrebna veća kontrola nad sinhronizacijom.

C#'s ocjena konkurentnosti

C# pruža napredak u odnosu na Java, svaki metod može imati svoje niti. Završetak niti je čišći nego u Java. Sinhronizacija je više sofisticirana

Uvod u obradu izuzetaka

Izvršne greške (npr dijeljenje s nulom ili pristup zaključanoj datoteci) su jedna od najneugodnijih osobina izvršnih okruženja. U jeziku bez obrade izuzetaka kada se izuzetak dogodi, kontrola prelazi operativnom sistemu, gdje se ispiše poruka i program završi izvršenje.

U jeziku s obradom izuzetaka programima je dopušteno da se uhvate neki izuzeci, dajući mogućnost popravke i nastavka. Mnogi jezici dopuštaju da se hvataju ulazno/izlazne greške (uključujući EOF).

Izuzetak je svaku neobični događaj, pogreška ili ne, koji se može prepoznati hardverom ili softverom, a koji može zahtijevati posebnu obradu. Specijalna obrada koja se može zahtijevati nakon što se detektuje izuzetak se zove obrada izuzetaka. Jedinica koda za obradu izuzetaka se zove obrađivač izuzetaka.

Prednosti ugrađene obrade izuzetaka

Kod za detekciju grešaka je dosadan i opterećuje listing, kao u sljedećem primjeru koji ručno provjerava da li su indeksi niza uneseni u pravilnom opsegu.

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
sum += mat[row][col];
else
System.out.println("Index range error on mat, row = " +
row + " col = " + col);
```

Obrada izuzetaka pojednostavljuje kod i ohrabruje programere da prati razne moguće greške

I programi sa neobičnim situacijama koji nisu greške se mogu strukturno pojednostaviti

Prosta obrada izuzetaka

U jezicima bez sofisticirane obrade izuzetaka, za detekciju grešaka koriste se, između ostalog, sljedeće metode.

U jezicima poput C uobičajeno je da povratna vrijednost funkcije ili pointer na neki od parametara predstavljaju status izvršenja potprograma.

Fortran i COBOL imaju mogućnost da se proslijedi labela kao parametar svim potprogramima. U slučaju greške nastavlja se od proslijeđene labele. Sljedeći primjer u Fortran-u u slučaju greške čitanja idi na liniju 100:

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

BASIC poznaje naredbu ON ERROR GOTO koja centralizovano obrađuje sve greške, navodi joj se kao parametar broj linije od koje se program nastavlja u slučaju greške.

Pitanja dizajna obrade izuzetaka

Prilikom dizajna specijalnih sintaksnih konstrukcija za obradu izuzetaka postavljaju se sljedeća pitanja

- Kako i gdje se obrađivači izuzetaka navode i koji im je opseg?
- Kako je događaj izuzetka vezan uz obrađivač izuzetaka?
- Može li se informacija o izuzetku proslijediti obrađivaču?
- Gdje se izvršenje eventualno nastavlja,nakon što obrađivač izuzetaka završi izvršenje? (nastavak ili vraćanje)
- Da li postoji forma finalizacije?
- Kako se navode korisnički definisani izuzeci ?
- Trebaju li postojati obrađivači izuzetaka za programe koji nemaju svoje?
- Mogu li se ugrađeni izuzeci eksplicitno aktiviratii?
- Da li se hardverske greške smatraju kao izuzeci koji se mogu obraditi?
- Postoje li ugrađeni izuzeci?
- Kako se izuzeci mogu eventualno deaktivirati?

Obrada izuzetaka u Ada

Okvir za obrađivač izuzetaka u Ada je tijelo potprograma, tijelo paketa, task ili blok. Pošto su obrađivači izuzetaka lokalni za kod u kome se izuzetak može podići, oni nemaju parametara.

Obrađivači se smjeste na kraj bloka ili jedinice u kojoj se događaju

Oblik obrađivača je sljedeći:

```
begin
....
exception
 when Izuzetak{|Izuzetak} => Naredbe
    ...
  [when others =>  Naredbe]
...
end
```

Ako blok ili jedinica u kojoj je izuzetak pokrenut nema obrađivač za taj izuzetak, izuzetak se prosljeđuje na drugo mjesto za obradu:

- U slučaju procedure izuzetak se proslijedi pozivaocu
- Za blokove, proslijedi opsegu u kome se pojavljuju
- Tijelo paketa – proslijedi deklaracijskom dijelu jedinice koja je deklarisala paket (ako je bibliotečna jedinica, program se prekida)
- Task – nema prosljeđivanja. Ako ima obrađivač, izvrši, ako ne označi ga završenim

Blok ili jedinica koja generiše izuzetak ali ga ne obradi se uvijek prekida (kao i blok ili jedinica na koju se prosljeđuje a ne obrađuje ga)

Korisnički definisani izuzeci se deklarišu na sljedeći način:

```
exception_name_list : exception;
```

Aktiviranje izuzetaka obavlja se naredbom raise.:

```
raise [exception_name]
```

(Ime izuzetka nije obavezno ako je u obrađivaču, u tom slučaju prosljeđuje se isti izuzetak)

Uslovi izuzetka se mogu onemogućiti sa:

```
pragma SUPPRESS(exception_list)
```

Nekoliko predefinisanih izuzetaka su;

- CONSTRAINT_ERROR – ograničenja indeksa, opsega itd.
- NUMERIC_ERROR – numerička operacija ne može vratiti ispravnu vrijednost (prekoračenje, dijeljenje s nulom itd)
- PROGRAM_ERROR – poziv potprograma čije tijelo nije obrađeno
- STORAGE_ERROR - sistem nema dovoljno heapa
- TASKING_ERROR – greška vezana za taskove

Ocjena

Ada dizajn za obradu izuzetaka je bio napredan 1980, Do dodavanja obrade izuzetaka u C++, Ada uz PL/I je bila jedini jezik koji je to podržavao. Dodavanje OOP u Ada nije pratilo unapređenje obrade izuzetaka.

Obrada izuzetaka u C++

C++ je dodao obradu izuzetaka 1990. Dizajn ovog dijela jezika baziran je na jezicima CLU, Ada, i ML.

Opšti oblik C++ obrađivača izuzetaka je

```
try {
-- dio koda koji može podići izuzetak
}
catch (formal parameter) {
-- obrađivač
}
...
catch (formal parameter) {
-- obrađivač
}
```

Kada se desi izuzetak unutar dijela koda označenog s try ključnom riječju, on će proslijediti parametar i zavisno od parametra će se izvršiti jedan od catch obrađivača.

Obrađivač catch je sintaksno napisan kao preopterećena funkcija. Za razne vrste izuzetaka, obrađivači se pišu s različitim protokolima parametara.

Formalni parametar obrađivača se može koristiti za prenos informacije obrađivaču. Najčešće se deklariše kao varijabla jednog tipa. Formalni parametar ne mora imati varijablu, može biti i ime tipa (npr. float) , samo da se razlikuje od drugih obrađivača. Pored ovoga, on može biti trotačka, kada se obrađuju neobrađeni izuzeci

Izuzeci se mogu podići naredbom

```
throw [expression];
```

Naredba throw bez operanda se može javiti samo u obrađivaču. Ako se desi, samo se proslijedi izuzetak, koji se obradi na drugom mjestu. Tip izraza čini obrađivač jednoznačnim

Neobrađeni izuzetak se proslijedi pozivaocu funkcije gdje se ponovo aktivira. Ako ni pozivalac nema obrađivača prosljeđivanje pozivaodima ide sve do main funkcije. Ako ni ona nema obrađivača, poziva se podrazumijevani obrađivač. Podrazumijevani obrađivač, se zove unexpected, prosto prekida program. Obrađivač unexpected se može redefinisati korisnikom

Nakon završetka obrađivača, kontrola prelazi na prvu naredbu iza zadnjeg obrađivača u sekvenci obrađivača čiji je element

Za razliku od Ada, nema predefinisanih izuzetaka. Svi izuzeci su korisnički definisani.

Pri deklaraciji funkcije mogu se navesti izuzeci koji se mogu aktivirati. Ako funkcija aktivira druge izuzetke throw naredbom, program se prekida.

Ocjena

Nepraktična osobina C++ je što izuzeci nisu imenovani. Nema predefinisanih hardverskih i sistemskih izuzetaka. Vezivanje izuzetaka za obrađivače kroz tip parametara ne popravlja čitljivost.

Obrada izuzetaka u Java i C#

U Java i C# obrada izuzetaka je bazirana na C++, ali više usklađena s OOP filozofijom Svi izuzeci su objekti klasa koje nasljeđuju Throwable klasu

Java biblioteka uključuje dvije klase Throwable :

- Klasa Error, Emituje ih Java interpreter za događaje kao heap overflow. Nikada ih korisnički programi ne obrađuju

- Klasa Exception je roditeljska klasa za korisnički definisane izuzetke. Postoje dvije predefinisane podklase, IOexception i Runtimeexception (npr., ArrayIndexOutOfBoundsexception i NullPointerexception

Obrađivači izuzetaka su kao u C++, osim što svaki catch zahtijeva imenovani parametar i svi parametri moraju biti nasljednici Throwable.

Sintaksa try klauzule je kao u C++. Izuzeci emituju s throw, kao u C++, ali često throw uključuje new operator da kreira objekat kao u:throw new MyException();

Povezivanje izuzetka s obrađivačem je jednostavnije u Java nego u C++

Izuzetak se veže za prvi obrađivač s parametrom iste klase kao emitovani objekt ili njegovim nasljednikom. Izuzetak se može obraditi ili dalje emitovati uključujući throw u obrađivač, a obrađivač može emitovati i drugi izuzetak)

Ako nema obrađivača u try konstrukciji, potraga se nastavlja u najbližoj uokviravajućoj try konstrukciji. Ako nema obrađivača u metodi izuzetak se prosljeđuje pozivaocu netode. Ako nema obrađivača sve do main, program se prekida

Da se osigura da su svi izuzeci uhvaćeni, obrađivač se može uključiti u svaku try konstrukciju koja hvata sve izuzetke. Taj obrađivač mora imati formalni parametar tipa Exception i biti zadnji u try konstrukciji.

Java throws klauzula u deklaraciji metoda je sintaksno slična ali semantički drugačija od throws klauzule u C++. Ona znači da metoda može pokrenuti izuzetak ali ga ne obrađuje. Metod ne može deklarisati više izuzetaka u throws klauzuli od metoda koga preklapa.

Izuzeci klase Error i RunTimeException i svi njeni nasljednici se zovu neprovjereni izuzeci. Svi drugi izuzeci se zovu provjereni izuzeci. Provjereni izuzeci koji se aktiviraju u metodi trebaju biti Listani u throws klauzuli ili obrađeni u samoj metodi.

Klauzula finally može se pojaviti na kraju try konstrukcije. Namjenaje da se navede kod koji će se izvršiti bez obzira šta se dešava u try konstrukciji. Njen oblik je

```
finally {
...
}
```

Primjer

try sa finally se može koristiti izvan obrade izuzetaka

```
try {
  for (index = 0; index < 100; index++) {
    ...
    if (...) {
      return;
    } //** end of if
} //** end of try clause
finally {
  ...
} //** end of try construct
```

Assert

Zbog kompromisa između performansi i pouzdanosti, u program se mogu uključiti provjere vrijednosti izraza. Ove provjere se opcionalno uključuju u kod izborom kompajlerske opcije.

Naredbe assert su naredbe u programu koje deklarišu logički izraz i prema trenutnom izračunatom izrazu

- Ako je izraz true ništa se ne događa
- Ako je izraz false emituje se AssertionError izuzetak

Imaju dva oblika

–assert condition;

–assert condition: expression;

Može se onemogućiti tokom izvršenja bez modifikacije ili rekompajliranja programa.

C#

C# ima sličnu sistem obrade izuzetaka kao Java, osim što nema throws klauzulu.

Ocjena

Tipovi izuzetaka imaju više smisla nego u C++. Klauzula throws je bolja nego u C++.Klauzula finally je često korisna. Java emutuje razne izuzetke koji se mogu obrađivati korisničkim programima

Uvod u obradu događaja

Rukovanje događajima je slično rukovanju izuzecima. U oba slučaja, obrađivači se implicitno pozovu na neku pojavu nešto, bio to izuzetak ili događaj.

Dok izuzeci mogu biti stvoreni bilo eksplicitno od strane korisnika koda ili implicitno hardverom ili izvršnim okruženjem, događaje stvaraju spoljne aktivnosti, kao što je korisnička interakcija kroz grafičko korisničko sučelje (GUI).

Većina događaja je uzrokovana korisničkom interakcijom kroz grafičke objekte ili komponente, često nazivaju widgeti, grafičke kontrole. Najčešći widgeti su dugmad, klizne trake, velika unosna polja. Implementacija reakcije na korisničke interakcije sa GUI komponenti je najčešći oblik rukovanja događajima.

Jedan događaj je obavijest da je došlo nešto posebno, kao što je klik miša na grafičku kontrolu. Strogo govoreći, događaj je objekat koji se implicitno stvara u vrijeme izvršavanja.

Obrađivač događaja je segment koda koji se izvršava kao odgovor na pojavu događaja. Obrađivači omogućuju program da donese odgovor na akcije korisnika.

Iako se programiranje upravljano događajima dugo koristilo prije pojave GUI pojavio, ono je postalo naširoko korištena programska metodologija kao odgovor na popularnost ovih interfejsa.

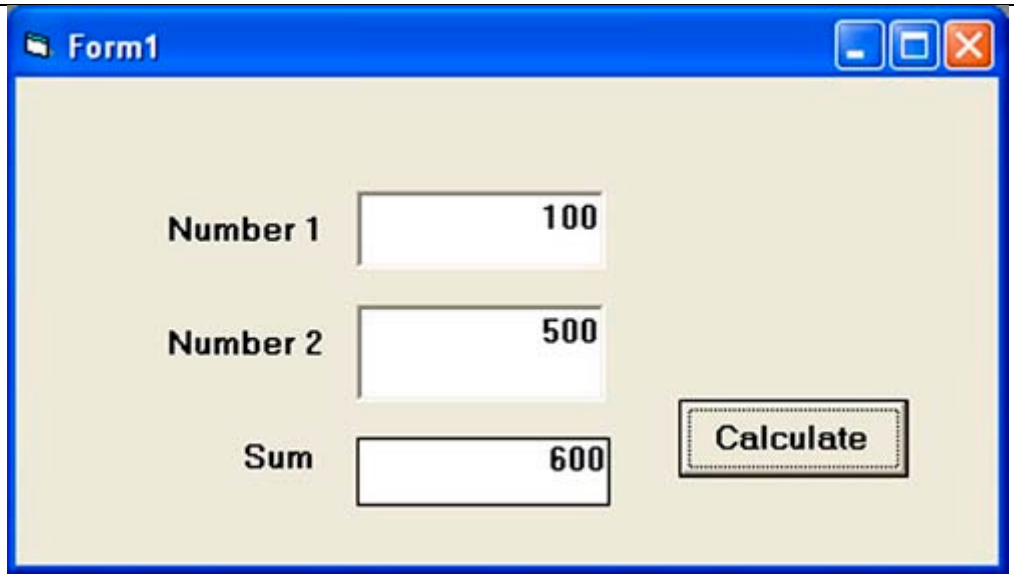
Kao primjer, razmotrite GUI predstavljen korisnicima Web pregledača. Mnogi Web dokumenti koje prikazuje preglednik su sada dinamična. Takav dokument može predstaviti obrazac narudžbenice za korisnika, koji bira robu klikom tipke. Potrebne interne proračune u vezi sa ovim klikovima na dugmad obavljaju obrađivači događaja koji reagiraju na klik događaja. Ova vrsta programiranja event-driven se često radi u jezicima na strani klijenta programski jezik, kao što su JavaScript.

Događaji u Visual BASIC-u

Visual BASIC je popularizovao programiranje upravljano događajima. U verzijama 1-6 upravljanje događajima nije bilo centralizovano, nego je svaka grafička kontrola

postavljena na prozor aplikacije imala svoj predefinisani skup događaja. Nakon dizajniranja izgleda prozora specijalnim grafičkim editorom povezuju se odgovori na događaje. Na primjer, uz dizajnirani formular na slici i dugme Command 1 može se povezati sljedeći obrađivač događaja.

```
Private Sub Command1_Click()
'To add the values in text box 1 and text box 2
Sum = Val(Text1.Text) + Val(Text2.Text)
'To display the answer on label 1
Label1.Caption = Sum
End Sub
```



Nakon prve verzije Visual BASIC.net sistem događaja je promijenjen, kao i sam jezik, jer je uveden puni sistem nasljeđivanja klasa. Obrađivačima se šalju dva parametra ByVal sender As Object i ByVal e As StartEventArgs.

Java model događaja

Java GUI aplikacije najčešće se pišu koristeći klase iz paketa swing ili awt. Korisničke interakcije s GUI komponentama kreiraju događaje koji se hvataju slušačima događaja (listener). Generator događaja to javi slušaču događaja šaljući poruku.

Klasa koja implementira slušača mora implementirati interfejs za slušača. Interfejs navodi metode koje se pozivaju na neke od registrovane događaje. Sljedeći primjer ilustruje upotrebu grafičkih komponenti i osluškivača definisanih u interfejsima WindowListener i ActionListener.

```
import java.awt.*;
import java.awt.event.*;
public class AL extends Frame implements WindowListener,ActionListener {
 TextField text = new TextField(20);
 Button b;
 private int numClicks = 0;
 public static void main(String[] args) {
   AL myWindow = new AL("My first window");
   myWindow.setSize(350,100);
   myWindow.setVisible(true);
 }
public AL(String title) {
     super(title);
     setLayout(new FlowLayout());
     addWindowListener(this);
     b = new Button("Click me");
     add(b);
     add(text);
     b.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    numClicks++;
    text.setText(" Clicked " + numClicks + " times");
  }
  public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
  }
  public void windowOpened(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public void windowClosed(WindowEvent e) {}
}
```

Obrada događaja u C#

Svi obrađivači događaja u C# imaju isti protokol. Povratni tip je void, a parametri su tipa object i EventArgs. Nijedan od ovih parametara se ne mora koristiti za jednostavne situacije. Npr, obrađivač za registrovanje promjene na CheckBox može izgledati ovako.

```
private void rb_CheckedChanged (object o, EventArgs e){
if (plain.Checked) . . .
. . .
}
```

GUI komponente već imaju registrovan određeni skup događaja. Da se registruje novi događaj, mora se kreirati novi EventHandler objekt. Njegovom konstruktoru se šalje ime metode obrađivača. Novi objekt se doda predefinisanom delegatu za događaj komponentnog objekta (koristeći += operator).

```
plain.CheckedChanged +=
new EventHandler(rb_CheckedChanged);
```

Obrada događaja u Delphi

VCL komponente uglavnom imaju pripremljene događaje OnClick, OnDataChange itd koji su definisani kao tip procedure of object. Prilikom pravljenja komponente, vidljivi novi obrađivači se smještaju u published odjel. Za komponente koje su nasljednik klase TwinControl mogu se definisati i metode s ključnom riječju message koje definišu reakciju na Windows sistemske poruke

Obrada događaja u proceduralnim jezicima

U jezicima kao što je C, aplikacije zasnovane na događajima zahtijevaju obrađivače pišu se u formi glavne petlje koja dohvata i obrađuje događaje kao u sljedećem predlošku

```
while (! Quitevent() {
GetNextEvent(&eventcode);
switch(eventcode) {
case EVENT1;
...
case EVENT2;
....
… }
}
```

Rezime

Konkurentno izvršenje može biti na nivou instrukcije, naredbe, potprograma

Fizička konkurentnost: više procesora izvršava konkurentne jedinice

Logička konkurentnost: konkurentne jedinice su na jednom procesoru

Dvije mogućnosti za konkurentnost na nivou potprograma: Kompetitivna sinhronizacija i Kooperacijska sinhronizacija

Mehanizmi: semafori, monitori, rendezvous, niti

Ada puža široke mogućnosti obrade izuzetaka uz ugrađene izuzetke.

C++ ne uključuje predefinisane izuzetke, a izuzeci su vezani za obrađivače povezivanjem tipa izraza u throw naredbi za formalni parametar catch funkcije

Java izuzeci su slični C++ izuzecima osim što Java izuzetak mora biti nasljednik Throwable klase. Dodatno Java uključuje finally klauzulu

Događaj je notifikacija da se nešto desilo što zahtijeva obradu obrađivačem događaja

13. Funkcionalni Programski Jezici

Dizajn imperativnih jezika, o kome je bilo riječi u prethodnim poglavljima je baziran na von Neumann arhitekturi. Za većinu programera, imperativni jezici su zadovoljavajući i napredovali su kroz duži period. Primarno pitanje imperativnih jezika je efikasnost

Dizajn funkcionalnih jezika je baziran na matematičkim funkcijama. Solidna teoretska baza bliža korisnicima ali nevezana za arhitekturu na kojim programi rade

Matematičke funkcije

Matematička funkcija je mapiranje članova jednog skupa (domena) u drugi (kodomena). Važna karakteristika matematičkih funkcija je da redoslijed njihovog računa nije definisan iteracijama nego rekurzijom i uslovnim izrazima. Druga važna karakteristika je da matematičke funkcije za isti argument uvijek vraća istu vrijednost, jer funkcije nemaju bočne efekte.

Funkcije se obično pišu na bazi imena funkcije praćenim listom parametara i izrazom koje one mapiraju, na primjer

cube (x) ≡ x * x * x

Ovdje su domen i kodomen realni brojevi. Parametar x predstavlja bilo koji član skupa domena koji ima fiksnu vrijednost tokom računa.

Lambda izrazi

Imenovanje funkcije nije neophodno, Na primjer, prilikom rješavanja linearne jednačine često se sa strane navode privremene funkcije koje se primjenjuju na oba izraza koji se javljaju u jednačini.

$$\frac{x-3}{2} = \frac{x-4}{2} \mid \times 2$$

Širu teoriju neimenovanih funkcija razradio je Alonzo Church kroz pojam lambda izraza. Lambda izrazi opisuju neimenovane funkcije . Lambda izrazi se primjenjuju navođenjem parametra nakon izraza. Na primjer

(λ(x) x * x * x)(2)

se računa kao 8

Funkcionalne forme

Funkcija višeg reda ili funkcionalna forma je izraz koji uzima funkcije kao parametre i rezultuje funkcijom kao svojim rezultatom

Kompozicija je funkcionalna forma koja uzima dvije funkcije kao paramere i pokreće funkciju čija je vrijednost prvi stvarni parametar primijenjen na primjenu drugog

Forma: h ≡ f º g

što znači h (x) ≡ f (g (x))

Za

f (x) ≡ x + 2

g (x) ≡ 3 * x,

h ≡ f º g

Funkcija h računa (3 * x)+ 2

Primjena na sve je funkcionalna forma koja uzima jednu funkciju kao parametar i vraća listu vrijednosti primijenjenu na svaki element liste parametara. Ova funkcionalna forma piše se simbolom α

Za h (x) $\equiv$ x * x

α (h, (2, 3, 4)) daje (4, 9, 16)

Osnove funkcionalnih programskih jezika

Dizajn funkcionalnih jezika je da liči na matematičke funkcije što više. Osnovni proces računa je fundamentalno drugačiji u funkcionalnim programskim jezicima u odnosu na imperativne jezike. U imperativnom jeziku obave se operacije, a rezultat se smjesti u varijablama za kasniju upotrebu. Upravljanje varijablama je glavno pitanje i izvor kompleksnosti za imperativno programiranje. U FPL, varijable nisu neophodne kao što je slučaj u matematici . Bez varijabli, iteracija nije moguća, pa se ponavljanje operacija oslanja na rekurziju. To je i glavna mana funkcionalnih jezika, s obzirom da rekurzija puni prostor na steku i manje je efikasna od iteracije. Ipak, ima jedan specijalni slučaj rekurzije koji se zove repna rekurzija. Ako je posljednja aktivnost funkcije rekurzivni poziv, tada se ne mora alocirati novi stek okvir i rekurzivne funkcija koje se mogu automatski konvertovati u iteraciju

Kako u čisto funkcionalnim jezicima nema varijabli, ne čuva se stanje programa, u FPL, evaluacija funkcije uvijek pruža isti rezultat za iste parametre. Ta osobina se zove referencijalna transparentnost.

Lisp

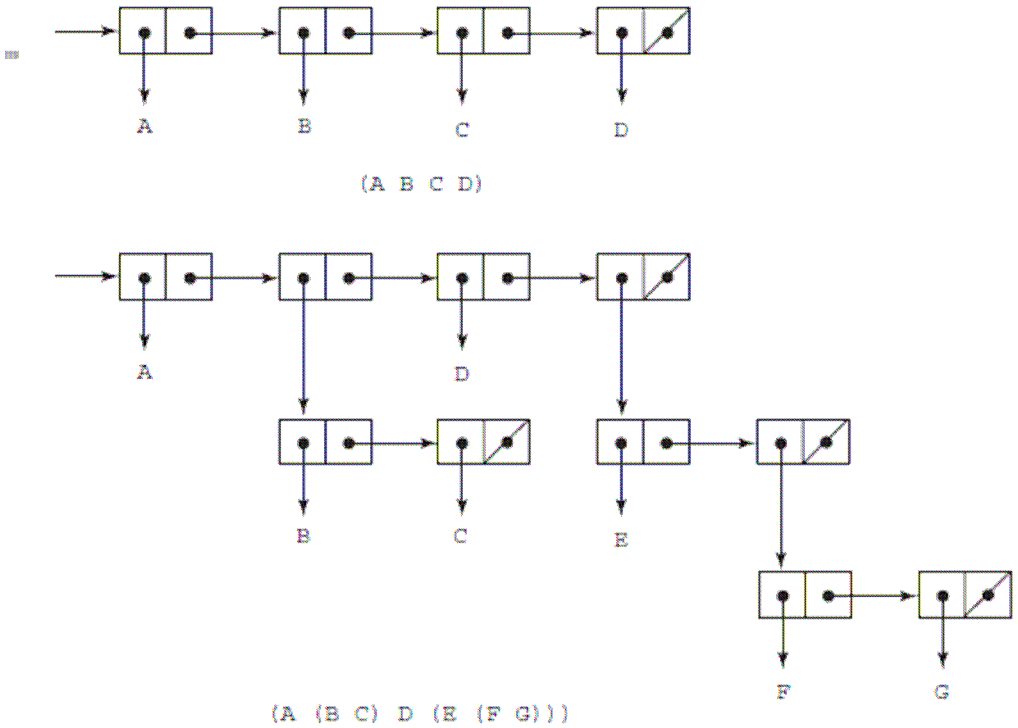
LISP je najstariji i najšire korišten funkcionalni programski jezik. Od ovog jezika je nastalo više dijalekata jezika. Iako je samo najstarija verzija bila čisto funkcionalna (kasnije su dodani imperativni elementi) nasljednici ovog jezika predstavljaju dobru ilustraciju funkcionalnih jezika.

Tipovi podataka i strukture

U prvoj verziji Lispa, tipovi podataka su samo atomi i liste. Atomi su simboli poput identifikatora ili numeričke vrijednosti. Liste se pišu u zagradama i mogu se sastojati od atoma i novih listi.

(A B (C D) E)

U memoriji svaki član liste je predstavljen sa dva polja. Prvo polje je pointer na atom ili drugu listu, a drugo je pointer na naredni element liste, pointer na atom ili nil vrijednost. Ovakva forma omogućava predstavljanje složenih listi kao na slici.



Po uzoru na univerzalnu Turingovu mašinu, pojavila se ideja univerzalne Lisp funkcije. U tu svrhu, nađen je način da liste predstavljaju ne samo zapis podataka nego i način zapisa definicije funkcija. Primjena funkcija i podaci imaju isti oblik. Npr. Ako je lista (A B C) interpretirana kao podatak, ona je prosta lista od tri atoma A, B, i C. Ako je interpretirana kao primjena funkcije, to znači da funkcija A je primijenjena nad parametrima, B i C. Prvi LISP interpreter se pojavio kao demonstracija univerzalnosti notacije

Lambda notacija se koristi za definisanje funkcija i primjenu nedefinisanih funkcija.

(function-name (LAMBDA (x y z) expression)

Scheme

1970-ih nastaje Scheme, dijalekt LISP-a, dizajniran kao čistiji, moderniji i jednostavniji od tadašnjih dijalekata LISP. Originalni Lisp je koristio dinamički opseg, dok Scheme koristi statički opseg vaijabli. Funkcije u Scheme su entiteti prve klase. To znači da mogu biti vrijednosti izraza i elementi listi, mogu se dodijeliti varijablama i proslijediti kao parametri drugim funkcijama.

Scheme interpreter radi kao petlja koja čita izraz koga unese korisnik, izračuna ga funkcijom EVAL i prikaže rezultat.

Pri računu izraza, prvo se izračunaju parametri bez posebnog redoslijeda. Vrijednost parametara se zamijeni u tijelo funkcije, a zatim se izračuna tijelo funkcije. Vrijednost zadnjeg izraza u tijelu je vrijednost funkcije.

Primitivne funkcije u Scheme

Aritmetičke funkcije su prefiksnog tipa. Argumenti im mogu biti cijeli, realni ili kompleksni brojevi. Mogu imati fiksni ili promjenjiv broj argumenata. Funkcije su date u sljedećoj tabeli.

(number? x)
(complex? x)
(real? x)
(rational? x)
(integer? x)
(exact? z)
(inexact? z)
(exact-integer? x
(exact-nonnegative-integer? x
(exact-rational? x)
(= z1 z2 z3 ...)
(< x1 x2 x3 ...)
(> x1 x2 x3 ...)
(<= x1 x2 x3 ...)
(>= x1 x2 x3 ...)
(max x1 x2 ...)
(min x1 x2 ...)
(+ z1 ...)
*(* z1 ...)*
(- z1 ...)
(/ z1 ...)
(gcd n1 ...)
(lcm n1 ...)
(zero? z)
(positive? x)
(negative? x)
(odd? x)
(even? x)
(abs x)
(quotient n1 n2)
(remainder n1 n2)
(modulo n1 n2)
(integer-floor n1 n2)
(integer-ceiling n1 n2)
(integer-truncate n1 n2)
(integer-round n1 n2)
(integer-divide n1 n2)
(integer-divide-quotient qr)
(integer-divide-remainder qr)
(floor x)
(ceiling x)
(truncate x)
(round x)
(floor->exact x)
(ceiling->exact x)
(truncate->exact x)
(round->exact x)
(rationalize x y)
(rationalize->exact x y)
(simplest-rational x y)
(simplest-exact-rational x y)
(exp z)
(log z)
(sin z)
(cos z)
(tan z)
(asin z)
(acos z)
(atan z)
(atan y x)
(sqrt z)
(expt z1 z2)
(make-rectangular x1 x2)
(make-polar x3 x4)
(real-part z)
(imag-part z)
(magnitude z)
(angle z)
(conjugate z)
(exact->inexact z)
(inexact->exact z)

npr.,

```
(+ 18 3 10.8)
31.8
(= 2 (+ 1 1))
#t
(sqrt 2)
1.4142135623730951

(sqrt (- 2))
+1.4142135623730951i
```

Rezultati funkcija poređenja (=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?, NEGATIVE? ...) su #T za tačan rezultat i #F za netačan rezultat (ponekad () se koristi za false)

Definicija funkcije

Lambda izrazi predstavljaju neimenovane funkcije koje se mogu koristiti u izrazima. Ovi izrazi se definišu funkcijom, LAMBDA. npr., (LAMBDA (x) (* x x)). U ovom primjeru, x je povezna varijabla koja se postavlja na vrijednost na koju se lambda izraz primjenjuje i ne mijenja se dok se mogu primijeniti. Sljedeći primjer će definisati lambda izraz koji kvadrira broj i izračunati kvadrat broja 7.

```
((LAMBDA (x) (* x x)) 7)
```

Funkcija DEFINE se koristi za kreiranje imenovanih konstanti i funkcija. Ova funkcija ima dva oblika. Prvi oblik se koristi za definisanje imenovanih konstanti.

```
(DEFINE pi 3.141593)
(DEFINE two_pi (* 2 pi))
```

Koristeći lambda izraze na ovaj način se mogu definisati i funkcije.

```
(define square (lambda (num) (* num num)))
(square 5)
```

Drugi oblik je namijenjen za definisanje funkcija i ima nešto jednostavniju sintaksu.

```
(DEFINE (square x) (* x x))
(square 5)
```

Funkcija DEFINE je drugačija od ostalih funkcija jer se njen prvi parametar se nikad ne računa. Drugi parametar se računa i veže uz prvi.

Ulazno-Izlazne funkcije

Postoje funkcije za ulazno/izlazne operacije. One odstupaju od čisto funkcionalnog modela programiranja, jer mijenjaju stanje programa. Podatak se čita funkcijom (read) koja vraća unesenu vrijednost, a prikazuje na ekranu funkcijom (DISPLAY expression). Funkcijom (NEWLINE) se prelazi u novi red. Sljedeći primjer će prikazati uneseni broj uvećan za 3.

```
(display (+ 3 (read)))
```

Kontrola toka:

Izbor se obavlja funkcijom IF koja ima tri parametra.

```
(IF predicate then_exp else_exp)
```

npr.,

```
(IF (<> count 0)
(/ sum count)
0)
```

Za višestruki izbor predviđena je funkcija COND. Generalna forma ove funkcije je.

```
(COND
(uslov1 vrijednost1 )
(uslov2 vrijednost2 )
...
(ELSE uslov vrijednost))
```

Funkcija COND vrati vrijednost zadnjeg izraza u prvom paru čiji predikat je true. Primjer za COND :

```
(DEFINE (leap? year)
(COND
((ZERO? (MODULO year 400)) #T)
((ZERO? (MODULO year 100)) #F)
(ELSE (ZERO? (MODULO year 4)))
))
```

Funkcije listi:

Funkcija QUOTE ima jedan parametar, i vraća ga bez računanja. Iako naviknutim na imperativne jezike takva funkcija izgleda čudno, QUOTE je potreban jer Scheme interpreter, zvan EVAL, uvijek računa parametre na pozive funkcija prije primjene funkcije. QUOTE se koristi da se izbjegne račun parametara kada nije potreban. To je tipično slučaj ako parametar funkcije treba biti lista s vrijednostima, a ne vrijednost funkcije koja se zove kao prvi element liste. QUOTE se može skratiti apostrof prefiks operatorom

'(A B) je ekvivalentno s (QUOTE (A B))

Liste se kreiraju funkcijama CONS i LIST .

CONS uzima dva parametra, prvi je atom ili lista, drugi je lista, i pravi listu čiji je prvi element prvi parametar, a drugi parametar je ostatak liste.

npr., (CONS 'A '(B C)) vraća (A B C)

LIST može imati proizvoljan broj parametara i vraća listu s parametrima kao elementima. Npr, (list 1 2 34) vrača (1 2 34).

Funkcija CAR koristi listu kao parametar i vraća prvi element te liste.

npr., (CAR '(A B C)) vraća A

(CAR '((A B) C D)) vraća (A B)

Funkcija CDR koristi listni parametar; vraća listu nakon uklanjanja prvog parametra

npr., (CDR '(A B C)) vraća (B C)

(CDR '((A B) C D)) vraća (C D)

Uvedeno je nekoliko funkcija kao skraćenica za pristup drugim dijelovima liste, sekvenca od nekoliko slova A i D između C i R u imenu. Tako na primjer CAADAR je funkcija koja predstavlja

(CAR (CAR (CDR (CAR X))))

Funkcije za testiranje vrijednosti

Predikatska funkcija: EQ? uzima dva simbolička parametra i vraća #T ako su oba parametra atomi i jednaki, inače vraća #F

npr., (EQ? 'A 'A) vraća #T

(EQ? 'A 'B) vraća #F

Ako se EQ? poziva s listnim parametrima, rezultat nije pouzdan. Također EQ? ne radi za numeričke atome.

Funkcija LIST? Ima jedan parametar; vraća #T ako je parametar lista; inače #F.

Funkcija NULL? ima jedan parametar; vraća #T ako je parametar prazna lista; inače #F. Funkcija NULL? vraća #T ako je parametar ().

Sljedeći primjer definiše funkciju member koja uzima atom i prostu listu; vraća #T ako je atom u listi; #F inače

```
DEFINE (member atm lis)
(COND
((NULL? lis) #F)
((EQ? atm (CAR lis)) #T)
((ELSE (member atm (CDR lis)))
))
```

Ovaj primjer definiše funkciju equalsimp koja uzima dvije proste liste kao parametre, vraća #T ako su jednake; #F inače

```
(DEFINE (equalsimp lis1 lis2)
(COND
((NULL? lis1) (NULL? lis2))
((NULL? lis2) #F)
((EQ? (CAR lis1) (CAR lis2))
(equalsimp(CDR lis1)(CDR lis2)))
(ELSE #F)
))
```

Naredni primjer definiše equal koja uzima dvije generalne liste kao parametre, vraća #T ako su jednake; #F inače

```
(DEFINE (equal lis1 lis2)
(COND
((NOT (LIST? lis1))(EQ? lis1 lis2))
((NOT (LIST? lis2)) #F)
((NULL? lis1) (NULL? lis2))
((NULL? lis2) #F)
((equal (CAR lis1) (CAR lis2))
(equal (CDR lis1) (CDR lis2)))
(ELSE #F)
))
```

Naredni primjer definiše funkciju append koja uzima dvije liste kao paramertre, i vraća prvu parametarsku listu s elementima druge parametarske liste na kraju

```
(DEFINE (append lis1 lis2)
(COND
((NULL? lis1) lis2)
(ELSE (CONS (CAR lis1)
(append (CDR lis1) lis2)))
))
```

Kreiranje statičkog opsega varijable

Funkcija LET kreira varijable čije vrijeme važenja je unutar te naredbe. One se mogu koristiti unutar te funkcije.

Opšti oblik:

```
(LET (
(name_1 expression_1)
(name_2 expression_2)
...
(name_n expression_n))
body
)
```

Ova funkcija izračuna sve izraze, poveže vrijednosti s imenima i izračuna tijelo funkcije. Primjer :

```
(DEFINE (quadratic_roots a b c)
(LET (
(root_part_over_2a
(/ (SQRT (- (* b b) (* 4 a c)))(* 2 a)))
(minus_b_over_2a (/ (- 0 b) (* 2 a)))
(DISPLAY (+ minus_b_over_2a root_part_over_2a))
(NEWLINE)
(DISPLAY (- minus_b_over_2a root_part_over_2a))
))
```

Rep rekurzija u Scheme

Funkcija je rep-rekurzivna ako je rekurzivni poziv zadnja operacija u funkciji. Pisanjem funkcija u ovakvom obliku, može se izbjeći glavni nedostatak funkcionalnih jezika: orijentisanost na rekurziju, koja zahtijeva veliki memorijski prostor i vrijeme za

prosljeđivanje parametara. Rep-rekurzivna funkcija se može konvertovati u iteraciju kompajlerom. Štaviše, Scheme jezička definicija zahtijeva da sistemi konvertuju rep-rekurzivne funkcije u iteraciju.

Kao ilustraciju, pogledati sljedeću definiciju faktorijela.

```
 (DEFINE (factorial n)
(IF (= n 0)
1
(* n (factorial (- n 1)))
))
```

Ova funkcija nije repno rekurzivna, jer je zadnja funkcija koja se poziva množenje.

No, modifikacija u sljedeći oblik dovodi do repno rekurzivne funkcije.

```
(DEFINE (facthelper n factpartial)
(IF (= n 0)
factpartial
facthelper((- n 1) (* n factpartial)))
))
(DEFINE (factorial n)
(facthelper n 1))
```

Funkcionalne forme u Scheme

Funkcionalna forma kompozicija je već korištena u ranijim primjerima.

(CDR (CDR '(A B C))) vraća (C)

Moguće je kompoziciju i posebno definisati i primijeniti nad operandima.

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

Tada će

```
((compose CAR CDR) '((a b) c d))
```

vratiti c.

Funkcionalna forma Primijeni na sve postiže se funkcijom map. Ona je definisana ovako.

```
(DEFINE (map fun a_list)
(COND
((NULL? a_list) '())
(ELSE (CONS (fun (CAR a_list))
(map fun (CDR a_list))))
))
```

Sljedeći primjer mapira lambda funkciju koja računa kub na

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

(27 64 8 216).

Ovaj primjer vraća listu čiji su članovi kvadratni korijeni brojeva između 1 i 3.

```
(map sqrt '(1 2 3))
```

COMMON LISP

Common Lisp je kombinacija raznih dijalekata LISP u ranim 1980s . To je velik i kompleksan jezik, suprotno od Scheme. Podržava slogove, nizove, kompleksne

brojeve, stringove, ulaz/izlaz, Pakete s kontrolom pristupa i iterativne naredbe. Jezik se može interpretirati i kompajlirati.

Funkcije u Common Lispu se definišu funkcijom defun

```
(DEFUN factorial (x)
(IF (<= n 1)
1
(* n factorial (– n 1)))
))
```

Podrazumijevan je statički opseg, ali je moguć i dinamički opseg deklaracijom varijable da bude 'special'.

QUOTE operator s naopakim apostrofom omogućava da se računaju neki dijelovi liste koji imaju zarez kao prefiks. Na primjer

`(a (* 3 4) c) vraća (a (* 3 4) c), dok `(a ,(* 3 4) c) vraća (a 12 c)

ML

ML je funkcionalni jezik statičkog opsega sa sintaksom bližom Pascal-u nego LISP-u. Za razliku od Lisp i Scheme, ML može imati deklaracije tipova funkcionalnih parametara i povratnih vrijednost, ali se to malo koristi. jer u ML postoji zaključivanje tipova da odredi tipove nedeklarisanih promjenjivih. Strogo je tipiziran (Scheme je u suštini bez tipova) i nema prisilu tipova. Uključuje upravljanje izuzecima i module za apstraktne tipove podataka, liste i operacije nad njima

Varijable treba shvatiti kao imena za vrijednosti, jer jednom postavljene ne mogu se mijenjati.

Oblik deklaracije funkcije u ML je

```
fun name (parameters) = expression;
```

npr.,

```
fun cube (x : int) = x * x * x;
```

Povratni tip funkcije se može navesti

```
fun cube (x) : int = x * x * x;
```

Ako tip nije naveden, podrazumijeva se int (za numeričke vrijednosti), Korisničle preopterećene funkcije nisu dopuštene, Ako želimo cube funkciju za realne parametre, treba nam drugo ime

Uslovi u ML se navode sljedećom sintaksom:

```
if expression then then_expression else else_expression
```

prvi izraz mora imati Boolean vrijednost

```
fun fact(n : int): int =
  if n <= 1 then 1
  else n * fact(n – 1);
```

Usaglašavanje uzoraka se koristi da funkcija radi s različitim oblicima parametara.

```
fun fact(0) = 1 |
fact(n : int) : int = n * fact(n – 1)
```

Liste se navode u uglastim zagradama [3, 5, 7]. Prazna lista je [] . Funkciji CONS iz Lisp-a odgovara binarni infiksni operator, ::.

```
 4 :: [3, 5, 7],
```

što postaje [4, 3, 5, 7]

Funkciji CAR odgovara unarni operator hd, a CDR je unarni operator tl.

```
fun length([]) = 0
| length(h :: t) = 1 + length(t);
fun append([], lis2) = lis2
| append(h :: t, lis2) = h :: append(t, lis2);
```

Naredba val veže ime za vrijednost (slično DEFINE u Scheme).

```
val distance = time * speed;
```

Ona se dosta koristi u let izrazima kao kratica za neke dijelove izraza.

```
let
  val radius = 2.7
  val pi = 3.14159
in
  pi * radius * radius
end;
```

Lambda izrazi se pišu ključnom rječju fn. Mogu se koristiti npr. u filter funkciji, koja kao prvi argument ima funkciju, a drugi listu čiji se svaki element testira funkcijom navedenom kao prvi argument. Filter funkcija vraća listu elemenata koji zadovoljavaju dati test.

```
filter(fn(x) => x < 100, [25, 1, 50, 711, 100, 150, 27,161, 3]);
```

Funkcionalna forma "Primjena na sve " postignuta je map funkcijom . Sljedeći primjer vraća listu čiji su elementi kubovi vrijednosti navedenih u listi.

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

Funkcionalna forma "kompozicija" postignuta je operatorom o. Sljedeći primjer gradi funkciju h koja je kompozicija funkcija f i g.

```
val h = g o f;
```

Haskell

Haskell je sličan ML po sintaksi, i mogućnostima kao što su statički opseg, strogi tipovi, prepoznavanje tipova, usaglašavanje uzoraka.

Razlikuje se od ML i drugih funkcionalnih jezika po tome što je čisto funkcionalan (nema varijabli, naredbi dodjele, i bočnih efekata)

Neke sintaksne razlike od ML se vide što nema ključne riječi za definisanje funkcija, i svi uzorci parametara imaju isti oblik.

```
fact 0 = 1
fact n = n * fact (n – 1)
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

U definiciji funkcije mogu se dodati čuvari, koji preciznije definišu uslove pod kojima su parametri važeći.

```
fact n
  | n == 0 = 1
  | n > 0 = n * fact(n – 1)
sub n
  | n < 10 = 0
  | n > 100 = 2
  | otherwise = 1
square x = x * x
```

Liste u Haskell

Elementi listi pišu se u uglastim zagradama, npr.,

```
directions = ["north", "south", "east", "west"]
```

Dužina liste se određuje operatorom: # ,npr.,

```
#directions
```

vraća 4

Aritmetičke sekvence za kreiranje listi dobijaju se koristeći .. operator, npr.,

```
[2, 4..10]
```

vraća [2, 4, 6, 8, 10]

Spajanje listi obavlja se s ++ npr.,

```
[1, 3] ++ [5, 7]
```

rezultuje u [1, 3, 5, 7]

CONS, se postiže preko operatora dvotačke (kao u Prolog) ,npr.,

```
1:[3, 5, 7]
```

rezultuje s [1, 3, 5, 7]

Razumijevanje listi koristi notaciju sličnu matematičkoj notaciji definicije skupa {x | uslov}. Ovaj primjer vraća listu kvadrata prvih 20 pozitivnih cijelih brojeva:

```
[n * n | n <- [1..20]]
```

Sljedeći primjer vraća listu faktora datog parametra.

```
factors n = [i | i <- [1..n `div` 2],n `mod` i == 0]
```

Evo primjera algoritma Quicksort u Haskellu. On je prepis definicije algoritma: Skup elemenata liste koji su manji ili jednaki vrijednosti glave liste se sortiraju i spoje s glavom liste, a skup elemenata liste koji su manji ili jednaki vrijednosti glave liste se sortiraju i spoje iza glave liste,

```
sort [] = []
sort (a:x) = sort [b | b <-x; b <= a]
          ++ [a] ++
          sort [b | b <- x; b > a]
```

Lijena evaluacija

Jezik je striktan ako zahtijeva da se svi parametri funkcije potpuno računaju. Jezik je nestriktan ako nema taj zahtjev. Na primjer, ako se prvi parametar funkcije ne koristi u funkciji, on se neće računati. Nestriktni jezici pružaju neke mogućnosti kao što su beskonačne liste. Jedan od načina računa parametara u nestriktnim jezicima je lijena

evaluacija. To znači da se računaju samo vrijednosti koje su potrebne i u trenutku kada postanu potrebne.

Mogu se definisati beskonačni skupovi, npr skup pozitivnih brojeva i kvadrata pozitivnih brojeva.

```
positives = [0..]
squares = [n * n | n <- [0..]]
```

Sljedeći primjer bi proslijedio listu squares funkciji member da odredi da li je 16 kvadrat broja, ponavljaće dok ne nađe.

member squares 16

```
member [] b = False
member(a:x)b =
   (a == b)||member x b
```

Ako pozovemo sa member squares 16 to će raditi samo kada je parametar za squares perfektan kvadrat, inače vrti zauvijek. Sljedeća definicija ispravno radi:

```
member2 (m:x) n
| m < n = member2 x n
| m == n = True
| otherwise = False
```

F#

F# je .NET funkcionalni programski jezik s razvojnim okruženjem, baziran na OCaml, nasljedniku ML i Haskell. Premda je funkcionalni ima i imperativne i OOP elemente. F# programi mogu sarađivati s ostalim .NET programima. Također je podržan od strane Visual Studio. F# uključuje različite tipove podataka. Među njima su torke, poput onih Python i funkcionalnog jezika ML i Haskell, liste, diskriminisane unije, slogovi, promjenjivi i nepromjenjivi nizovi. Liste u F # 's liste su slične onima od ML, osim što su elementi su odvojeni tačka/zarezom a hd i tl se moraju pozvati kao metode List.

F# podržava vrijednosti sekvence, koji su tipovi iz .NET namespace System.Collections.Generic.IEnumerable. U F#, sekvence se skraćeno pišu kao

```
seq <tip>,
```

gdje <tip> tip može biti generički.

Na primjer, seq <int> je sekvenca cijelih vrijednosti. Sekvence se mogu kreirati generatorima i mogu se ponoviti. Najjednostavnije sekvence se generiraju s rasponom izraze, kao u sljedećem primjeru:

```
let x = seq {1..4};;
```

U primjere F #, pretpostavljamo da je interaktivni prevodilac se koristi, što zahtijeva dva tačka-zareza na kraju svake izjave. ovaj izraz generira dalje [1; 2; 3; 4]. (elementi liste i sekvence su odvojeni tačka-zarezom).

Stvaranje sekvence je lijeno.; Na primjer, sljedeći primjer generiše ogromnu sekvencu, ali za prikaz se generišu sao prva četiri.

```
let y = seq {0..100000000};;
y;;
val it: seq<int> = seq[0; 1; 2; 3;. . .]
```

Prva linija definiše y; druga zahtijeva prikaz y, treća je izlaz F# interaktivnog interpretera. Podrazumijevani korak za sekvence je 1, ali se može promijeniti stavljanjem u sredinu opsega dodatnog parametra koji predstavlja korak.

```
seq {1..2..7};;
```

Ovo generiše seq [1; 3; 5; 7].

Vrijednosti sekvence se mogu iterirati for-in konstrukcijom, kao u primjeru

```
let seq1 = seq {0..3..11};;
for value in seq1 do printfn "value = %d" value;;
```

Što daje

```
value=0
value=3
value=6
value=9
```

Uvlačenje se koristi za prikaz definicije funkcije

```
let f =
   let pi = 3.14159
   let twoPi = 2.0 * pi
   twoPi;;
```

F#, kao ML, ne prisiljava numeričke vrijednosti, npr. Da je navedeno 2 umjesto 2.0 bilo bi greška

Ako je funkcija rekurzivna mora se koristiti ključna riječ rec

```
let rec factorial x =
if x <= 1 then 1
else n * factorial(n – 1)
```

Lambda izrazi se pušu koristeći ključnu riječ fun.

```
(fun a b –> a / b)
```

Primjene za funkcionalne jezike

APL se koristi za programe koji se urade i odbace.

LISP za vještačku inteligenciju, predstavljanje znanja, mašinsko učenje, obradu prirodnog jezika, simboličku matematiku, model govora i vizije

Scheme je korišten kao uvodni jezik na nekim univerzitetima

ML se koristi za dokazivanje nekih teorema

Haskell kao potpuno funkcionalni jezik je idealan za skalabilne programe u paralelnim okruženjima

F# se koristi kao funkcionalni dodatak u .NET okruženjima.

Poređenje funkcionalnih i imperativnih jezika

Imperativni jezici imaju sljedeće karakteristike

- Efikasni u izvršenju
- Kompleksna semantika
- Kompleksna sintaksa

- Konkurentnost je se mora posebno programirati

Funkcionalni jezici:

- Prosta semantika
- Proste sintaksa
- Neefikasno izvršenje
- Programi mogu automatski biti konkurentni

Uticaj funkcionalnih na imperativne jezike

U novije vrijeme elementi funkcionalnih jezika se uključuju u vodeće imperativne jezike. Ovo su primjeri:

Neimenovana funkcija u JavaScript

```
function (formal-parameters) {
body
}
```

Lambda izraz u Python

```
map(lambda x: x ** 3, [2, 4, 6, 8])
```

Lambda izraz u C#

```
i => (i % 2) == 0
```

Lambda blok u Ruby

```
times = lambda {|a, b| a * b}
x = times.(3, 4)
```

Rezime

Funkcionalni programski jezici koriste primjenu funkcija, uslovne izraze, rekurziju i funkcionalnu formu umjesto imperativnih vrijednosti kao što su varijable i dodjele .

LISP je počeo kao čisto funkcionalni jezik a zatim uključio imperativne mogućnosti .

Scheme je jednostavni dijalekt LISPa sa statičkim opsegom

COMMON LISP je veliki LISP-bazirani jezik.

ML je statičkog opsega strogo tipizirani funkcionalni jezik koji uključuje prepoztnavanje tipova, rukovanje izuzecima, strukture podataka i apstraktne tipove.

Haskell je funkcionalni jezik lijene evaluacije koji podržava beskonačne liste i prepoznavanje skupova .

F# je Microsoft .NET funkcionalni jezik .

Čisto funkcionalni jezici imaju prednosti nad imperativnim jezicima, ali njihova niža efikasnost na postojećim arhitekturama je spriječila široku upotrebu .

14. Logički Programski Jezici

Uvod

Jezici za logičko programiranje su vrsta deklarativnih programskih jezika. Programi u logičkim jezicima se izražavaju u formi simboličke logike. Izvršni sistemi ovih jezika koriste proces logičkog zaključivanja da dođu do rezultata .

Logički programski jezici su deklarativni a ne proceduralni. Samo se rezultati navode (a ne i postupci kako se dobijaju)

Propozicija je logički iskaz koji može i ne mora biti tačan. Sastoji se od objekata i odnosa između njih

Simbolička logika se može koristiti za osnovne potrebe formalne logike:

- Izražavanje propozicija
- Izražavanje odnosa između propozicija
- Opis kako se nove propozicije mogu dobiti iz drugih

Određeni oblici simboličke logike korišteni za logičko programiranje zovu se **predikatski račun**

Predstavljanje objekata

Objekti u propozicijama se predstavljaju **prostim termovima**: Prosti termovi su konstante ili varijable. **Konstanta** je simbol koji predstavlja objekt. **Varijabla** je simbol koji predstavlja različite objekte u različitom trenutku. Varijabla u logičkom jeziku razlikuje se od varijabli u imperativnim jezicima.

Atomske propozicije sastoje se od **složenih termova.** Složeni term je jedan element matematičke relacije, napisan kao matematička funkcija. Složeni term sastoji od dva dijela

- Funktor: funkcijski simbol koji imenuje relaciju
- Uređena lista parametara (n-torka)

Primjeri:

```
student(jon)
like(seth, OSX)
like(nick, windows)
like(jim, linux)
```

Propozicije se mogu navesti u dva oblika:

- Činjenica je propozicija koja se smatra tačnom
- Upit je propozicija čija se tačnost treba odrediti

Složena propozicija se sastoji od jedne ili više atomskih propozicija. Propozicije se povezuju operatorom. Operatori su navedeni na sljedećoj slici.

Ime	Simbol	Primjer	Značenje
negacija	¬	¬ a	not a
konjunkcija	∩	a ∩ b	a and b
disjunkcija	∪	a ∪ b	a or b
ekvivalencija	≡	a ≡ b	a is equivalent to b
implikacija	⊃	a ⊃ b	a implies b
	⊂	a ⊂ b	b implies a

Varijable se mogu pojavljivati u propozicijama, ali samo kada su uvedene kvantifikatorima. Kvantifikatori su navedeni na sljedećoj slici.

Ime	Primjer	Značenje
univerzalni	∀X.P	Za svako X, P je tačno
egzistencijalni	∃X.P	Postoji X takav da je P tačno

Primjer upotrebe operatora i kvantifikatora.

$$\forall X.(\text{woman}(X) \supset \text{human}(X))$$
$$\exists X.(\text{mother}(\text{mary}, X) \cap \text{male}(X))$$

Prva rečenica je zapis: “Sve žene su ljudi.” Druga rečenica: je zapis “Postoji muškarac čija je Mary majka”.Previše je načina da se kaže ista stvar, što komplikuje izvođenje zaključaka u predikatskoj logici. Zato se koristi standardna forma koja se zove klauzalna forma.

Klauzalna forma je zapis propozicije u sljedećem obliku:

Bl ∪ B2 ∪... ∪ Bn [] Al ∩ A2 ∩ ... ∩ Am

Ako su svi A-ovi tačni, onda je bar jedan B tačan

Preduslov je desna strana ovog izraza.

Posljedica je lijeva strana

Primjer:

father (louis, al) ∪ father (louis, violet) ⊂
father (al, bob) ∩ mother (violet, bob) ∩ grandfather (louis, bob)

Predikatski račun i Dokazivanje teorema

Propozicijama se otkrivaju nove teoreme koje se izvode iz poznatih aksioma i teorema.

Razrješenje (resolution) je princip zaključivanja koji omogućava da se zaključene propozicije računaju iz datih propozicija

Sljedeći primjer predstavlja jednostavnu vrstu razrješenja.

$$ima(sejo,telefon) \subset prikljucio(sejo,telefon)$$
$$koristi(sejo,telefon) \subset ima(sejo,telefon)$$

Iz ova dva pravila slijedi

$$koristi(sejo,telefon) \subset priljucio(sejo,telefon)$$

Ovo zaključivanje je postignuto tako što su lijeve strane propozicija spojene operacijom OR praveći lijevu stranu nove propozicije. Desne strane propozicija se spajaju operacijom AND u desnu stranu nove propozicije. Nakon ovoga se izbacuju članovi koji se javljaju na obije strane.

Ako propozicije sadrže varijable zaključivanje je složenije i uključuje proces **unifikacije** i **instancijacije**.

Unifikacija je traženje vrijednosti varijabli u propozicijama koje omogućuju odgovarajućem procesu zaključivanja da uspije.

Instancijacija je dodjela privremenih vrijednosti varijablama da unifikacija uspije.

Nakon instancijacije varijable vrijednošću, ako slaganje ne uspije, moramo se vratiti (backtrack) i probati instancirati s drugom vrijednošću.

Jedan od jednostavnijih načina dokazivanja je dokaz kontradikcijom. Kod ovog načina dokaza kao cilj se postavi negacija teorema koji se tvrdi propozicijim. Teorem se tada dokaže nalaženjem nekonzistentnosti.

Radi lakšeg razrješenja, uvodi se još jednostavniji oblik pisanja propozicije od klauzalne forme. To su **Hornove klauzule**. Hornova klauzula može imati samo dva oblika:

- Oblik s zaglavljem ima jednu atomska propozicija na lijevoj strani
- Oblik bez zaglavlja ima praznu lijevu stranu

Većina propozicija se može pisati u Hornovom obliku

Prolog

Logički programski jezici karakterišu se deklarativnom semantikom. Postoji prost način da se odredi značenje svake tvrdnje i ne zavisi od toga kako se naredba koristi. Njihova semantika je jednostavnija od semantike imperativnih jezika, kod kojih semantika zavisi od položaja naredbe u programu, tipova podataka, stanja varijabli itd. Logičko programiranje se koristi u sistemima za upravljanje bazama podataka, ekspertnim sistemima, obradi prirodnog jezika i sličnim primjenama.

Programiranje u logičkim jezicima je neproceduralno. Programi ne navode kako se rezultat računa nego oblik rezultata.

Prolog je razvijen na univerzitetima University of Aix-Marseille i University of Edinburgh, u cilju istraživanja obrade prirodnog jezika i automatskog dokaza teorema.

Postoje tri osnovne varijante Prologa: Marseille sintaksa, Edinburgh sintaksa i Micro Prolog. Ovdje će biti opisana najpopularnija Edinburgh sintaksa,

Termovi

Term u Prologu je konstanta, varijabla, ili struktura. Konstanta može biti atom ili cijeli broj. Atom je simbolička vrijednost u Prologu. Atom se sastoji od

- niza slova, cifara ili podvlačilica koje počinju malim slovom
- ili niza ASCII znakova uokvirenih apostrofima

Varijable su nizovi slova, znakova i podvlačilica koje počinju velikim slovom. Prilikom instancijacije dešava se povezivanje varijable za vrijednost. Ovo povezivanje traje dok ne zadovolji kompletan cilj

Struktura: predstavlja atomsku propoziciju predikatskog računa i piše se u sljedećem obliku.

funktor(list parametara)

Lista parametara se može sastojati od atoma, varijabli i drugih struktura.

Iskazi činjenica

Prolog ima dvije osnovne rečeničke forme koje odgovaraju Hornovim klauzuama sa i bez zaglavlja. Hornove klauzule bez zaglavlja koriste se za hipoteze, na primjer:

```
female(shelley).
male(bill).
father(bill, jake).
```

Iskazi pravila

Hornove klauzule s zaglavljem se koriste za opis pravila. Njihov oblik je

```
posljedica :- preduslov .
```

Desna strana predstavlja preduslov (if dio). Preduslov može biti term ili konjunkcija. Konjunkcija je više termova povezanih implicitnim AND operacijama, koje se predstavlju zarezom.

Lijeva strana je posljedica (then dio) i ora biti jedan term

Primjer pravila

```
ancestor(mary,shelley):- mother(mary,shelley).
```

U pravilima se mogu se koristiti varijable (univerzalni objekti) da se generalizuje značenje:

```
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
sibling(X,Y):- mother(M,X), mother(M,Y),
father(F,X), father(F,Y).
```

Iskazi ciljeva

Za dokaz teoreme, teorema se navodi u formi propozicije za koju želimo da sistem nađe ili opovrgne dokaz – ciljna tvrdnja. Ima isti format kao Hornova propozicija bez zaglavlja. U sljedećem primjeru

```
man(fred)
```

sistem će odgovoriti sa Yes ili no.

Konjunktivne propozicije i propozicije s varijablama takođe mogu biti ciljevi

```
father(X,mike)
```

U datom primjeru će sistem vratiti skup svih vrijednosti varijabli X za koje je uslov ispunjen.

Zaključivanje u Prologu

Sada će se nešto reći na koji način Prolog dolazi do zaključaka. Upiti u Prologu se zovu ciljevi. Ako je upit složena propozicija, svaka od činjenica je podcilj

Da se dokaže tačnost cilja, mora se naći lanac zaključivanja. Neka je npr. dat cilj Q. Cilj Q je tačan ako je naveden kao propozicija ili se može izvesti iz činjenice da važi A i niza propozicija poput:

```
B :- A
C :- B
P :- C
Q :- P
```

Neka su data sljedeća pravila

father(bob).

man(X) :- father(X).

Na upit

father(bob)

odgovor se nalazi direktno iz propozicije. No na upit

man(bob)

varijabla X će se zamijeniti sa vrijednošću bob i iz pravila man(bob) :- father(bob) krenuti kroz lanac pravila da se nađe pravilo father(bob), koje postoji i upit vraća Yes.

Postoje dva osnovna pristupa u zaključivanju.

Zaključivanje odozdo prema gore kreće s lančanjem unaprijed. Ono počinje s pravilima i činjenicama i traži sekvencu prema cilju. Ovakvo zaključivanje je dobro ako je velik skup moguće tačnih odgovora

Zaključivanje odozgo prema dolje obavlja lančanje unazad. Ono počinje od cilja i traži sekvencu koja ide prema činjenicama u bazi. Dobro ako je mali skup mogućih tačnih odgovora. Prolog koristi lančanje unazad.

Kada cilj ima više podciljeva, između kojih treba da se bira može se ići također dvije strategije.

Pretraga u dubinu traži kompletan dokaz za prvi podcilj prije rada na sljedećim podciljevima. Pretraga u širinu istražuje sve podciljeve paralelno. Prolog koristi pretragu u dubinu jer ona zahtijeva manje resursa

Sa ciljem s više podciljeva, ako ne uspije, vraća se na prethodni podcilj s alternativnim rješenjem: **backtracking**. To znači da treba početi pretragu gdje je prethodna stala

Može uzeti mnogo vremena i prostora jer može tražiti sve moguće dokaze za sve moguće ciljeve.

Prosta aritmetika

Prolog nije jezik s jakim matematičkim mogućnostima, no ipak ima cjelobrojne varijable i aritmetiku. U ranijim verzijama Prologa aritmetičke operacije su bile funktori ali danas postoji **is** operator i aritmetički izrazi s uobičajenim infksnim zapisom. Operator is: uzima aritmetički izraz s desne strane i varijablu s lijeve

```
A is B / 17 + C
```

U ovom primjeru, ako su varijable B i C instancirane, tada A dobija vrijednost.,

Ovo nije isto što i naredba dodjeljivanja. Na primjer ovakva konstrukcija nije legalna ni ako Sum nije instanciran (referenca na desnu stranu je nedefinisana) ni ako jeste instanciran (jer lijevi operand ne može imati vrijednost ako je već ima).

Sum is Sum + Number.

Sljedeći primjer računa koliko je koji auto prešao kilometara utrke uzimajući u obzir srednju brzinu i dužinu staze.

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :- speed(X,Speed),
time(X,Time),
Y is Speed * Time.
```

Kada se da upit

distance(chevy, Chevy_Distance).

Kreira se variabla Chevy_Distance *i* prikaže njena vrijednost 2205.

Praćenje

Komanda trace prikazuje detalje o procesu zaključivanja. Kada je uključena, daju se informacije u četiri situacije

- Poziv , kada je započeto računanje cilja
- Izlaz, kada je cilj zadovoljen
- Ponovni pokušaj, kada se desi backtrack
- Neuspjeh, kada cilj ne uspije

Primjer upotrebe naredbe trace za prethodni upit. Prolog pravi interne varijable koje se zovu _broj.

```
trace.
distance(chevy,Chevy_Distance).
(1) 1 Call: distance(chevy, _0)?
(2) 2 Call: speed(chevy, _5)?
(2) 2 Exit: speed(chevy, 105)
(3) 2 Call: time(chevy, _6)?
(3) 2 Exit: time(chevy, 21)
(4) 2 Call: _0 is 105*21?
(4) 2 Exit: 2205 is 105*21
(1) 1 Exit: distance(chevy, 2205)
Chevy_Distance = 2205
```

Strukture listi

Lista je sekvenca proizvoljnog broja elemenata. Elementi mogu biti atomi, atomske propozicije ili drugi termovi uključujući druge liste. Liste se pišu u uglastim zagradama, elementi su razdvojeni zarezima,

```
[apple, prune, grape, kumquat]
```

Prazna lista se označava ovako

```
[]
```

Notacija za prvi element i ostatak liste je razdvajanje uspravnom crtom

```
[X | Y]
```

Kao primjer, definisaćemo relacije koje spajaju dvije liste

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
append (List_1, List_2, List_3).
```

Nakon poziva

append(X, Y, [a, b, c]).

dobijamo sve kombinacije X i Y koje spojene daju listu [a,b,c]

X = [] Y = [a, b, c]

X = [a] Y = [b, c]

X = [a, b] Y = [c];

X = [a, b, c] Y = []

Drugi primjer je predikat za izvrtanje elemenata liste

```
reverse([], []).
reverse([Head | Tail], List) :-
reverse (Tail, Result),
append (Result, [Head], List).
```

MicroProlog

Za računare ograničene memorije razvijen je MicroProlog, jezik sintakse koja više liči na Lisp, Sintaksa listi je u oblim zagradama (kao u Lisp-u), a i predikati se pišu sa sintaksom liste. Na primjer:

((father john smith))

Ovo je bila bezuslovna činjenica. Imena varijabli počinju velikim slovom X ili Y. Uslovne rečenice se piši tako da prvi član složene liste predstavlja posljedicu, a ostali uslove.

```
((likes X Y) (knows X Y) (likes Y X ))
```

Simple sintaksa je infiksna i učitava se kao separatni program. U ovoj sintaksi se prethodna rečenica opisuje kao:

```
X likes Y if X knows Y and Y likes Y
```

Rezime

Simbolička logika je osnova za logičko programiranje

Logički programi su neproceduralni

Prolog izjave su činjenice, pravila i ciljevi

Zaključivanje je primarna aktivnost Prolog interpretera

Premda ima dosta nedostataka s trenutnim stanjem logičkog programiranja ono se koristi u mnogim područjima

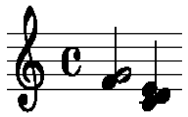
15. Domensko specifični jezici

Jedna od definicija programskog jezika je da je to skup pravila za upravljanje algoritmom. (Horowitz). No, pored opisa algoritama, postoji još niz drugih načina komunikacije između čovjeka i računara koji su nazvani jezicima.

Na primjer, skraćenice; HTML (hypertext markup language) i UML (Universal markup language) uključuju riječ jezik u sebi iako nisu programski jezici.

HTML kaže browseru šta da prikaže ali ne kroz algoritam, nego kroz opis izgleda. Browser ima slobodu da interpretira format i prikaže izgled. HTML ne opisuje algoritam, nego se interpretira različitim algoritmima u algoritmu za prikaz strane, validatorima, i serverima pretraživačima

Postoje i jezici namijenjeni ljudima koji uopšte nemaju veze s programiranjem. LilyPond je namijenjen za lakše pisanje muzičkih zapisa namijenjenih za štampu.



```
<< g2 \\ { f4 <<c4 d4 e4>> } >>
```

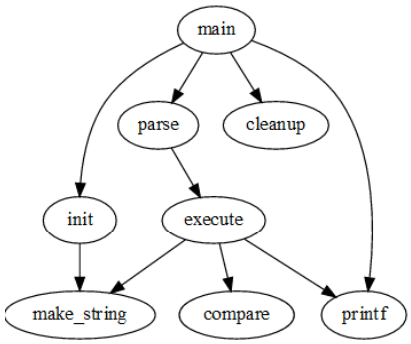
Načini realizacije ovakvih jezika su raznovrsni. Lylipond je realizovan koristeći C++, Scheme i LaTeX (koji je sam drugi domenski specifični jeziku)

Nije neuobičajeno vidjeti komande nekih programskih jezika koje su same mali jezici. PHP i JavaScript uključuju funkcije čiji su parametri regularni izrazi, SQL se ugrađuje u parametre objekata u C++ i Delphi, a čak i GWBASIC imao je složene naredbe poput:

```
play "c4.d8"
draw "u10r3d10l3"
```

Nekada se prave specijalizovani jezici kao alternativa gafičkom dizajnu. dot (www.graphviz.org) je namijenjen za crtanje usmjerenih grafova.

```
digraph G {
 main -> parse -> execute;
 main -> init;
 main -> cleanup;
 execute -> make_string;
 execute -> printf
 init -> make_string;
 main -> printf;
 execute -> compare;
}
```



Logo je prvi programski jezik namijenjen djeci. Kreirao ga je Seymour Papert 1968. Papert je prvi koji je vidio kako kompjuteri mogu mijenjati učenje Logo je razvijen iz Lisp-a, ali sa manje zagrada i fokusom na grafici. Ovim jezikom se upravlja malim robotom s pisaljkom ili strelicom na ekranu. Robotom kornjačom se upravlja komandama koje omogućavaju kretanje za određen broj prostornih jedinica (piksela ili milimetara) i zakretanje za određen broj ugaonih stepeni:

```
forward d, backward d, turnright a, turnleft a
pendown, penup, turnup a, turndown a,
spinright a, spinleft a
```

Kontrolne strukture su uvedene radi crtanja složenijih likova s ciljem da se tako upozna pojam petlji i uslova.:

```
repeat n cmds, ifelse c cmds cmds,
to procname params cmds, procname
```

Razni primjeri malih jezika

Primjeri

- Program "units"

You have: (1e-14 lightyears + 100 feet) / s
You want: furlongs per half fortnight
Answer: 376067.02 (sličan calc...)

- Regularni izrazi

b(c|de)*f – slaže li se sa bdedecf?

- Makefiles: Komande pod određenim uslovima

Određuje redoslijed njihovog izvršenja (sa paralelizacijom)

- Lex, yacc, coco/r: prave format drugog jezika!

Prevodi se u kod za tokeniziranje i parsiranje

- Awk: obradi svaku liniju struktuirane datoteke

$2==$3 { sum += $0; print $0, sum }

- Protokoli: Neke vrste jezika se ugrađuju I u komunikacijske protokole.

Mnogi su tekst bazirani
HTTP, FTP , SMTP

```
zahtjev k cs.jhu.edu:     GET /holy/grail HTTP/1.0
cs.jhu.edu odgovara:      404 Not Found
```

Komunikacija s sendmail demonom

```
220 blaze.cs.jhu.edu ESMTP Sendmail 8.12.9/8.12.9; Tue, 31 Jan 2006 11:06:02 -
0500 (EST)
helo emu.cs.jhu.edu
250 blaze.cs.jhu.edu Hello emu.cs.jhu.edu [128.220.13.179], pleased to meet you
expn cs325-staff
250-2.1.5 Jason Eisner <jason@...>
250 2.1.5 Jason Smith <jrs026@...>
quit
221 2.0.0 blaze.cs.jhu.edu closing connection
Connection closed by foreign host.
```

- PERL (za skriptiranje I upravljanje tekstom)
- VHDL (za opis hardvera)
- TeX i LaTex (za izgled dokumenta)
- JQuery (dodatne naredbe u web klijentima)
- HTML/SGML (markiranje dokumenta)
- Postscript (grafika na niskom nivou)
- Open GL (3D grafika)
- SQL (baze podataka)
- Tcl/Tk (GUI skripte)

- Macromedia Director (multimedija)
- Prolog (logika)
- Mathematica/Maple (simbolički račun)
- AutoLisp/AutoCAD (CAD)
- Emacs Lisp (editovanje)
- Excel Macro Language (poslovne kalkuacije)

Definicije malih jezika ili Domensko specifičnih jezika

"Programski jezik krojen za određenu primjenu. Nije opšte namjene, nego hvata semantiku domene, ni manje ni više"

"Apstrakcija domene, jezik koga možete objasniti planiranom korisniku za manje od jednog dana."

"Čista notacija razmišljanja o problemu u domeni i komunikacija s drugim ljudima i automatskim rješavačima."

Interni DSL su domensko specifični jezici koji su sastavni dio ili dodatak drugih programskih jezika ili softverskih paketa (JQuery, PLSQL, LaTex, AutoLisp, Excel makro jezik)

Eksterni DSL su samostalni programi (kompajleri ili interpreteri) koji implementiraju domenski specifični jezik (awk, Perl, Prolog,VHDL)

Aplikacijske domene

Primjene za domenske jezike obuhvataju:

- Opis hardvera
- raspored komponenti
- Uzorci teksta
- Grafika i animacija
- Kompjuterska muzika
- Distribuirani i paralelni računari
- Baze podataka
- Logika
- Sigurnost
- Rasporedi
- Modeli
- Simulacije
- Grafički interfejsi
- Leksička i sintaksna analiza
- Symboličko računarstvo
- Attributske gramatike
- CAD/CAM
- Robotika

Zašto jezici za korisnike

Većina programera nisu pravi programeri nego su nastavnici, mašinski inžinjeri, sekretarice, knjigovođe, menadžeri, dizajneri svjetla ...

Omjer takvih programera u odnosu na "profesionalne" je 20:1 sa tendencijom rasta.

Načini implementacija DSL

Domenski specifični jezici se realizuju na tri osnovna načina

Ugradnja u postojeće jezike

Kod ovog pristupa iskoristi se postojeći programski jezik opšte namjene i dodaju mu se nove naredbe prilagođene određenoj primjeni. Jezik u koji se domenski jezik dodaje se zove **domaćinski jezik**.

Neki domaćinski jezici su dizajnirani za proširenje: Lua, Tcl/Tk, Za složenije izmjene domaćinskih jezika, uobičajeno je pravljenje pretprocesora, koji konvertuju dijelove izvornog koda u miješanom jeziku u čisti kod u domaćinskom jeziku.

Glavna prednost ovog pristupa je što jezik automatski dobija osnovne strukture kao što su petlje, lokalne varijable. Ponaša se kao domaćinski jezik s dodatnim naredbama, operatorima. Druga prednost je laka implementacija. Često nije potrebno se baviti leksičkom i sintaksnom analizom, nego koristiti makro-instrukcije ili generičke tipove. Mana je što se ovako mogu realizovati samo jezici namijenjeni programerima.

Primjer Pro C je dodatak SQL naredbi u C program

```
 for ( ; ; )
{
 if ( askn("Enter employee dept  : ,&deptno) < 0 )
      break;
    EXEC SQL WHENEVER NOT FOUND GOTO nodept;
    EXEC SQL SELECT DNAME
            INTO :dname
            FROM DEPT
            WHERE DEPTNO = :deptno;
    dname.arr[dname.len] = '\0';
```

Interpreter

Interpreteri prevode i izvršavaju program liniju po liniju. Oni Proizvode izlaz prije nego se vidi cijeli program pa su korisni su ako je program dugačak ili ako korisnik želi da vidi rezultate izvršenja prije pisanja novog dijela. Pogodni su za sve vrste korisnika, osim ako domenski jezik zahtijeva dobre performanse

Primjeri domenskih jezika koji se realizuju interpreterski:

- Komandni jezici poput Unix shell, scripting languages,
- Excel
- Upitni jezici: SQL, Prolog, ...
- Client-server protokoli: HTTP, Dynagraph , ...

Kompajler

Prevodi cijeli program u jezik nižeg nivoa. Prevođenje se obavi samo jednom, a izvršava više puta. Kod domenskih jezika kompajleri ne prevode samo u mašinski kod, nego i u jezike visokog nivoa poput C++ ili Java

Primjeri:

Coco/R u C++, koji se kompajlira s g++

16. Semantička analiza i međureprezentacije

Nakon predstavljanja semantike različitih vrsta programskih jezika, sada ćemo se pozabaviti analizom semantike u kompajlerima. Semantička analiza se obavlja nakon sintaksne, iz apstraktnog stabla sintakse.

Semantičke provjere

Sintaksna analiza nije dovoljna da kompajler prepozna da li rečenica pripada jeziku. Nakon provjere formalne sintakse treba naći preostale greške koje bi učinile program neispravnim, kao što su nedefinisane varijable i tipovi podataka i greške tipova koje se mogu statički prepoznati. Semantička analiza treba saznati i druge korisne informacije za kasnije faze, kao što su tipovi izraza i raspored podataka u programu (koji podatak se može saznati tek ako se ima neki drugi)

Sljedeća funkcija u jeziku C je sintaksno ispravna, ali pokazuje nekoliko grešaka koje se mogu otkriti u toku semantičke analize: poput dva puta deklarisane varijable k, pristupa varijabli q koja nije deklaruisana, pristup elementu strukture preko varijable koja nije deklarisana, greške o broju i tipu parametara u pozivu funkcije, skok na nepostojeću labelu, break koji nije u petlji. Ima i grešaka koje bi mogle, ali se zbog specifikacija jezika ne otkrivaju kao što je pristup elementu niza f koji je izvan opsega.

```
void foo(int a, char * s);
int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
 j = i + k;
  printf("%s,%s.\n",p,q);
  goto label23;
}
```

Dok se u jezicima koji se kompajliraju sintaksne provjere u potpunosti obavljaju pri kompajliranju, a u čisto interpretiranim pri izvršenju, semantičke provjere se mogu razdijeliti između ta dva vremenska perioda.

Statičke provjere obavlja kompajler prilikom prevođenja programa u međureprezentaciju, ali u nekim slučajevima i kasnije, iako prije izvršenja. Pristupanje nedeklarisanoj varijabla se može detektovati prilikom kompajliranja. Poziv nepostojećeg eksternog potprograma se saznaje u fazi linkovanja.

Dinamičke provjere rade se pri izvršenju programa. Primjer predstavlja provjera da li je prekoračen indeks niza izvan opsega deklaracije(u jezicima u kojima je to bitno), da li otvorena datoteka postoji, da li je korisnik unio ispravnu numeričku vrijednost na mjestu gdje se ona očekuje.

Po cilju provjere, može se klasifikovati na sljedeći način.

Provjera jedinstvenosti kontroliše da li su imena u datom opsegu jedinstvena. Provjera se obavlja u jezicima koji zahtijevaju deklaracije varijabli. Na primjer u Pascal-u

```
var i:integer; k,i: real;
```

je nelegalno jer varijabla i nije jedinstvena.

Provjere kontrole toka prate da li su naredbe kontrole toka pravilne strukture. Iako ovaj zadatak više liči na sintaksnu analizu, postoje i kontrole toka koje se provjeravaju u semantičkoj. Npr, u jeziku C, naredba *break* je sintaksno samostalna naredba, ali ima smisla samo unutar *while*, *do*, *for* i *switch* naredbi.

Provjera tipova predstavlja provjeru kompatibilnosti operatora i operanada u izrazima i pozivima potprograma. O ovim provjerama je već dosta bilo riječi.

Logičke provjere su provjere grešaka koje se javljaju u programu koji je sintaksno i semantički ispravan ali ne radi ispravno. Ove provjere se uglavnom obavljaju dinamički i to od strane čovjeka, ali neke se mogu detektovati i statički, uz pisanje upozorenja (warning). Primjer su čitanje varijabli kojima nije dodijeljena vrijednost ili upotreba operatora dodjele na mjestu gdje normalno treba biti operator poređenja,

```
if (a=5) {call(1); }
```

Primjeri grešaka koje se detektuju u semantičkim provjerama su:

Undeclared identifier

Multiply declared identifier

Index out of bounds

Wrong number or types of args to call

Incompatible types for operation

Break statement outside switch/loop

Goto with no label

Realizacija semantičke analize

Generatori parsera obično imaju mogućnost ugradnje vlastitog koda koji se izvršava na onim mjestima kada je prepoznat odgovarajući sintaksni element. Taj kod se zove semantička akcija, pa je logično da se semantička analiza može raditi kao dio semantičkih akcija. Tako na primjer u generatoru YACC mogu se dodati uz sintaksu, semantičke akcije poput sljedećih:

```
expr : expr PLUS expr {
          if ($1.type == $3.type &&
             ($1.type == IntType ||
               $1.type == RealType)) $$.type = $1.type
          else error("+ applied on wrong type!");
          GenerateAdd($1, $3, $$);
        }
```

No, ovo rješenje je prihvatljivo samo za jednostavnije kompajlere. Kako broj tipova raste i elemenata izraza ugrađeni kod u semantičkim akcijama postaje sve glomazniji i težak je za čitati i održavati. Dodatno otežanje je što kompajler mora analizirati program u redoslijedu parsiranja, što čini provjere poput postojanja labele na koju *goto* naredba skače teškim kada je u pitanju skok unaprijed.

Ako sintaksna analiza završava generisanjem apstraktnog stabla sintakse, semantičke provjere se mogu obavljati i odvojeno od sintaksnih provjera. Nakon što je AST generisan, može se korigovati u AST' koji sadrži dodatne informacije o tipovima podataka koji se nalaze u listovima ovog stabla.

Stoga bi pseudokod kompajlera izgledao ovako:

```
void Compile() {
  AST tree = Parser(program);
  if (TypeCheck(tree))
    IR ir =
      GenIntermedCode(tree);
    EmitCode(ir);
  }
}
```

Semantički analizator (funkcija TypeCheck(tree) u gornjem pseudokodu) radi u dvije faze prolazeći kroz AST koga je kreirao parser. Algoritam izgleda ovako

```
Prolaz 1: Za svaki opseg u programu
 Obradi deklaracije. Za svaku deklaraciju
   dodaj nove elemente u Tabelu simbola i
   prijavi varijable koje su više puta deklarisane
 Obradi naredbe. U naredbama
    nađi nedeklarisane varijable i prijavi greške
   ažuriraj apstraktno stablo sintakse:
      Za sve čvorove koji predstavljaju identifikatore postavi
      vezu na tabelu simbola.
Prolaz 2: Obradi ponovo sve naredbe u programu
      koristi informacije iz tabele simbola za određivanje tipa
       izraza i grešaka tipa
```

Tabela simbola

U području statičkog opsega važenja identifikatora dati identifikator treba čuvati u tabeli simbola. No sam identifikator nije jedinstvena predstava podatka ili potprograma, jer u većini jezika isto ime se može koristiti više puta:

- Ako mu se deklaracije javljaju u različitim opsezima. Ovo je slučaj sa većinom jezika koji dopuštaju lokalne varijable

- Ako predstavljaju različite vrste imena. Neki jezici dopuštaju isto ime u različitim objektima u istom opsegu. Tako u Java: isto ime može ime biti

klasa, polje klase, metoda klase i lokalna varijabla metode a da se nalaze u istom opsegu važenja varijabli

```
class Test {
        int Test;
        void Test( ) { double Test; }
 }
```

- Ako jezik dopušta preopterećenje potprograma, na primjer Java i C++ (ali ne Pascal ili C) mogu imati isto ime za više metoda/potprograma ako su broj i tipovi parametara jedinstveni

```
int add(int a, int b);
float add(float a, float b);
```

Tabela simbola stoga mora da se dinamički mijenja u toku semantičke analize. Kako se mijenja opseg važenja varijabli i drugih simbola prilikom prolaska kroz apstraktno stablo sintakse tako treba da se mijenja i pristupačnost elementu tabele simbola.

Tabela simbola prati imena koja su deklarisana u programu. Svaki njen element treba pored imena da sadrži atribute ili vezu na njih. Atributi koje sadrži element tabele simbola su:

- vrsta imena (varijabla, imenovani tip, klasa, metoda, funkcija...)
- tip (int, float, ...)
- nivo gniježdenja (dubina statičkog opsega)
- relativna memorijska lokacija (gdje će se nalaziti pri izvršenju)

Funkcije koje omogućavaju povezivanje imena sa atributom tipa su:

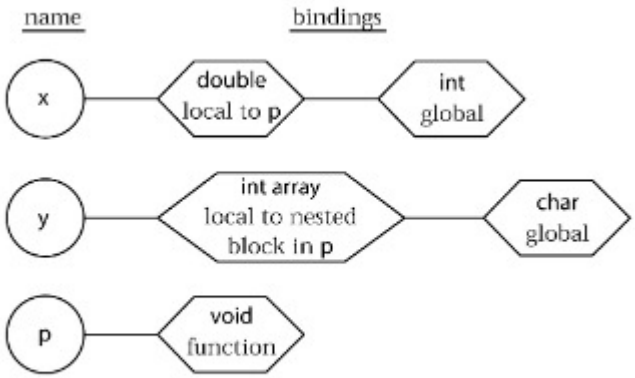
- Type Lookup(String id) , vraća ime tipa za navedeni identifikator
- Void Add(String id, Type binding), dodaje vezu tipa sa imenom u tabeli simbola

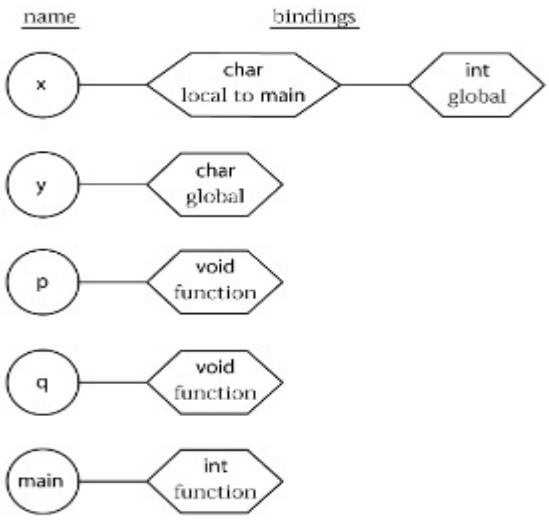
Jedan od načina organizacije tabele simbola je da prilikom prevođenja elementi tabele sadrže imena koja se povežu sa ulančanim listama atributa, sortiranim po dubini. Kako se deklariše novi simbol, ako već ne postoji to ime u tabeli on se ubacuje u tabelu i njegovi atributi pridruže u ulančanoj listi. Ako simbol već postoji tada se samo atributi ubacuju u pripadnu ulančanu listu ukoliko nije u pitanju dvostruko deklarisanje simbola u istom opsegu.

Neka je dat primjer programa

```
int x;
char y;
void p(void)
{ double x;
…
{ int y[10]; /*  AAA */
…
}
…
}
void q(void)
{ int y;
…
}
main()
{ char x;  /*  BBB */
…
}
```

U trenutku kada je kompajler u fazi semantičke analize došao do mjesta označenog sa AAA, tabela simbola i njene veze sa atributima izgleda kao na sljedećoj slici





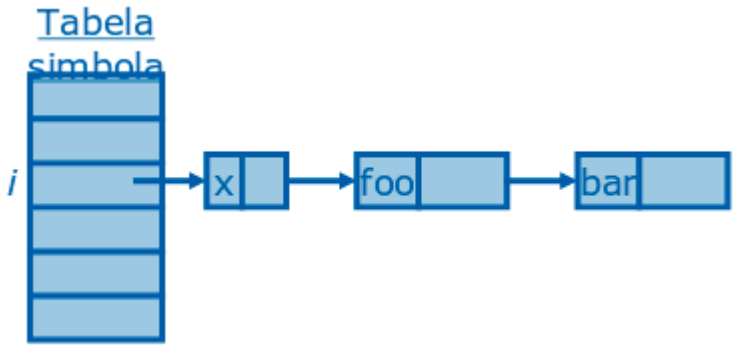
Kada je došao do mjesta označenog sa BBB, izgleda kao na sljedećoj slici.

Uz tabelu simbola i ulančanu listu atributa mogu se dodati dodatne strukture radi povećanja efikasnosti njihovog kreiranja i pretraživanja.

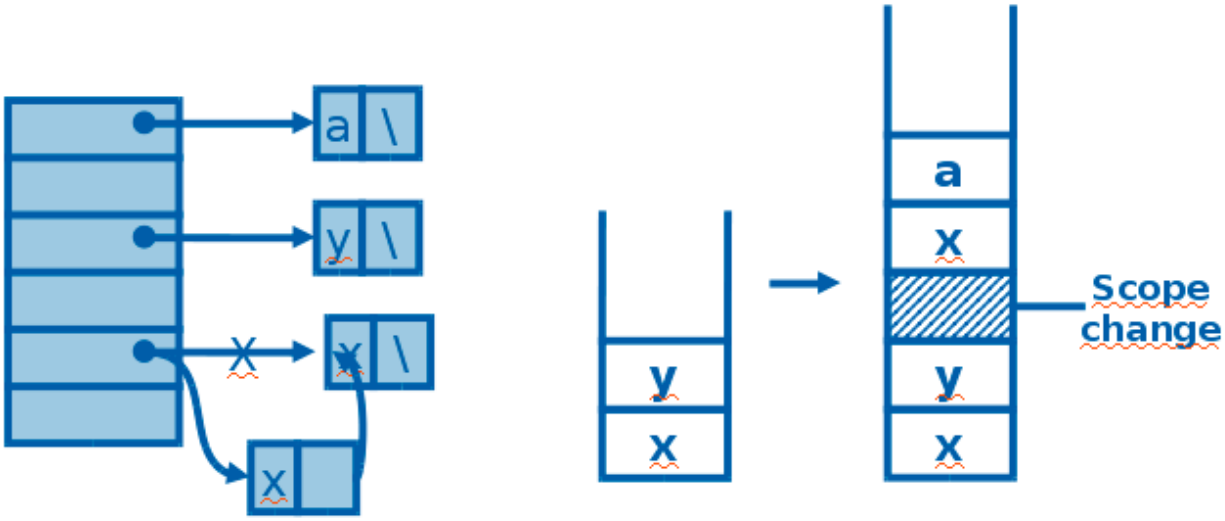
Pretraživanje tabele simbola se obavlja brže ako se koriste **hash funkcije**. One računaju iz imena simbola njegovu poziciju u tabeli i time izbjegavaju sekvencijalnu pretragu tablice.

```
pozicija=Hash(simbol)
```

Pošto više simbola može imati jednaku hash vrijednost (npr. ako je hash funkcija zbir ASCII kodova imena, tada varijable *barka* i *kraba* imaju istu hash vrijednost), tabela simbola se može organizovati i kao na sljedećoj slici sa nizom pokazivača na ulančane liste imena koji imaju istu hash vrijednost, kao na sljedećoj slici.

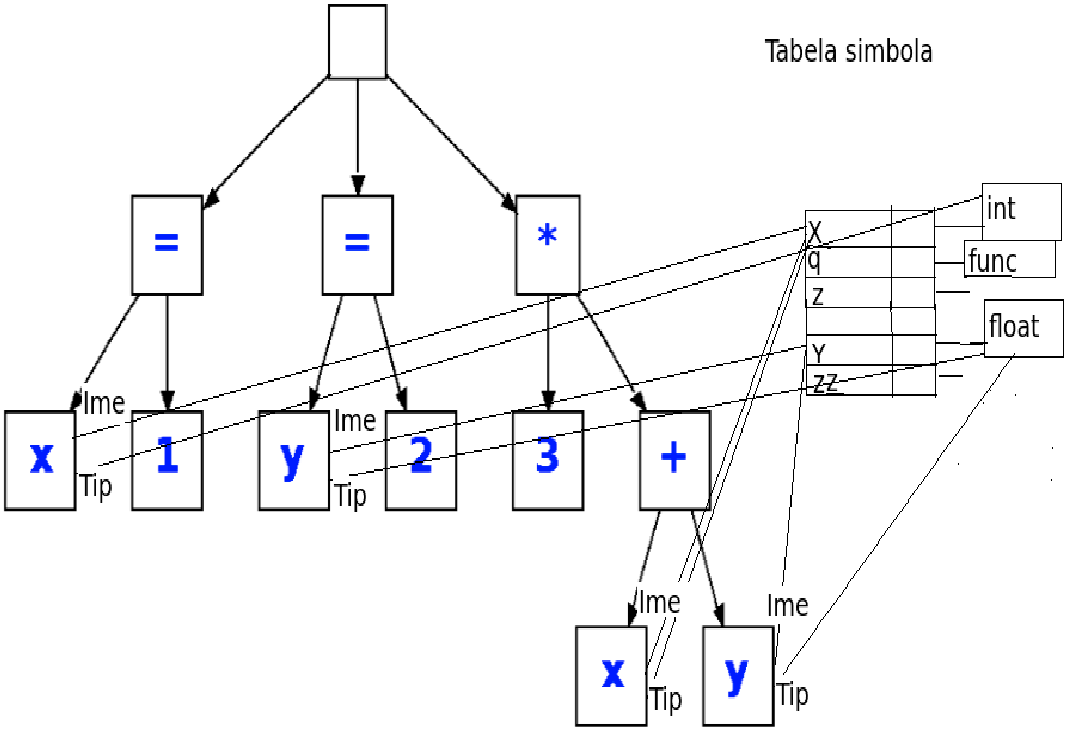


Pri izlasku iz opsega važenja varijable (kraj potprograma ili u nekim jezicima bloka) treba izbaciti iz listi koje predstavljaju atribute sve elemente prikopčane na nju. Ubrzanje ovog postupka može se postići dodavanjem još jedne strukture podataka: stack za praćenje"nivoa gniježdenja" pri prolazu kroz stablo. Svaka novodeklarisana varijabla (i drugi objekt koji se kreira u tabeli simbola) se pored unosa u tabelu simbola čuva i na posebnom steku. Pri početku svakog novog bloka postavi se marker na poziciji steka. Po završetku opsega, treba izbrisati iz ulančanih listi samo one



atribute varijabli koje se na steku nalaze na poziciji između njegovog vrha i markera.

Naredno ubrzanje se može postići razdvajanjem tabela simbola u posebne tabele za različite vrste simbola. Najčešće korisnički definisani tipovi imaju svoje vlastite tabele simbola.



Dekorisano apstraktno stablo sintakse

Nakon obrade deklaracija u apstraktnom stablu sintakse, slijedi obrada naredbi u tom stablu. U čvorovima apstraktnog stabla sintakse se nalaze dodatni atributi koji su pokazivači na tabelu simbola i atributa koji su pridruženi simbolima (identifikatorima) u tabeli. Tako na primjer, svaki čvor koji predstavlja varijablu ima vezu na njeno ime u tabeli simbola i njen tip u listi atributa, kao na slici. Apstraktno stablo sintakse kome su dodane sve informacije o identifikatorima i tipovima zove se dekorisano apstraktno stablo sintakse. Kompajler sada ima potrebne informacije za drugi dio semantičke analize, provjeru tipova.

Provjera tipova

Prilikom novog prolaska kroz dekorisano apstraktno stablo sintakse, sada se u svakom njegovom listu imaju informacije o njegovom tipu podataka. Te informacije treba proslijediti ostalim čvorovima ili provjeriti da li se smiju koristiti date kombinacije tipova podataka.

Semantika opisana atributskom gramatikom za pojedine jezičke pojmove se sada prevodi u odgovarajući programski kod. Kada su u pitanju aritmetički izrazi za svaku produkciju koja je prepoznata u apstraktnom stablu sintakse možemo imati akcije kao u tabeli.

Produkcija	Akcija dodjele i provjere tipa
E→literal	E.type = char
E→num	E.type = integer
E→id	E.type = id.type
E→E1 % E2	E.type = if E1.type=integer and E2.type = integer then integer else typeerror
...	...

Naredbe nemaju tipove, pa se njima dodjeljuje specijalni tip void. Ali i u pojedinim dijelovima naredbi ima posla za provjeru tipova. Na primjer, izraz koji se test odnosi u naredbama if i while u mnogim jezicima mora biti boolean tipa. Više provjera ima u naredbama dodjeljivanja. Provjera mora da vidi može li se lijevoj strani dodijeliti vrijednost, npr. Lijeva strana ne može biti konstanta ili tipa nekompatibilnog desnoj strani.

Primjer akcija koje se obavljaju pri provjeri tipova dat je u sljedećoj tabeli.

S→id = E	S.type : if id.type = E.type then void else typeerror
S→if E then S1	S.type = if E. type = booleanthen void else typeerror
S→while E do S1	S.type = if Eiype = boolean then void else typeerror
S→S1; S2	S.type =if Sl.type= void and S2.type = void then void else typeerror

Ako jezik podržava strukturalnu ekvivalenciju tipa, tada se provjera tipova obavlja rekurzivnom funkcijom koja treba da prepozna različite slučajeve.

```
function sequiv (Type : s, t) : boolean
begin
if s and t are the same basic type
return (TRUE);
else if (s = array(sl, s2)) and (t : array(t1, t2))
return (sequiv(sl, t1) and sequiv(s2, t2))
else if (s = pointer(s1)) and (t = pointer(t1))
return (sequiv(sl, t1))
else
return (FALSE);
end if
end { sequin}
```

U ovoj fazi se dešavaju i konverzije i prilagođavanje tipova. Na primjer, konstanta 15 je u toku leksičke analize prepoznata kao cijeli broj. Ispostavi se, međutim da se ova konstanta sabira sa vrijednošću varijable koja je tipa float. To je u većini jezika legalno, ali je u tom slučaju potrebno informisati generator koda da konstantu 15 konvertuje u konstantu u pokretnom zarezu. Tada se u apstraktno stablo sintakse ubacuju i novi čvorovi koji predstavljaju akciju konverzije tipova.

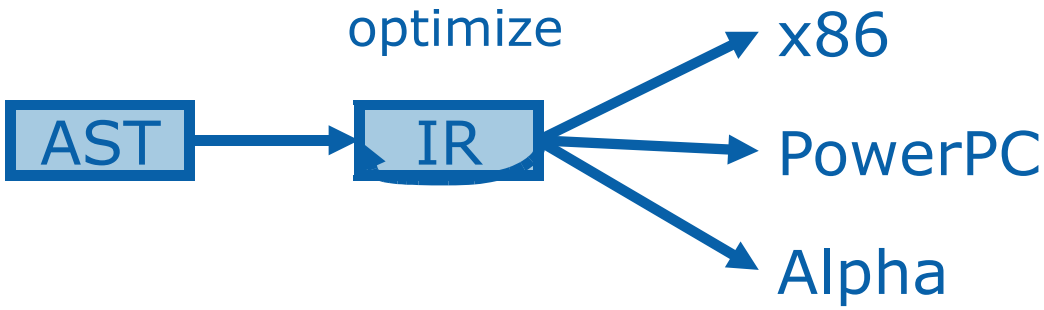
Međureprezentacije

Po završetku semantičke analize dobijeno je dekorisano apstraktno stablo sintakse. Ovo stablo predstavlja strukturalno znatno drugačiji način zapisa programa u odnosu na izvorni kod, ali i dalje čuva većinu informacija (osim komentara, makroinstrukcija, redundantih zagrada i slično) kao i izvorni kod.

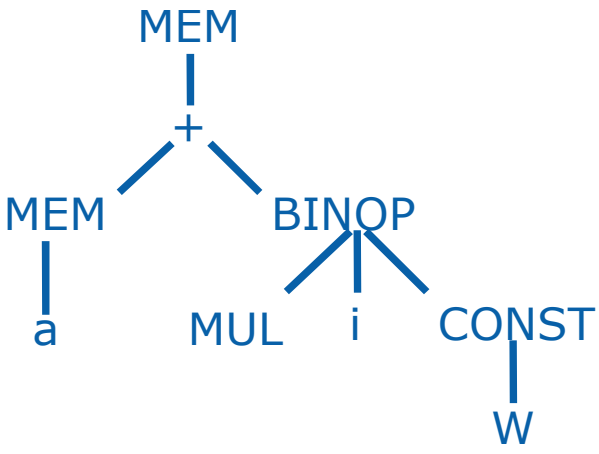
Dobijeno stablo je samo jedna od međureprezentacija koje se mogu javiti u procesu konverzije između izvornog i mašinskog koda. Višeprolazni kompajleri prevode iz jedne međureprezentacije u drugu, tokom svog rada.

Međureprezentacije su potrebne ako se želi napraviti dobar optimizirajući kompajler, a pogotovo ako je njegov cilj generisanje koda za različite platforme. Za razliku od leksičke, sintaksne i semantičke analize, transformacije međureprezentacija nisu obavezne u kompajlerima ali povećavaju njihov kvalitet, a ponekad ih i pojednostavljuju.

Na primjer, ako je cilj kompajlera da podržava 4 izvorna jezika (npr. C, C++, Ada, Fortran) i 5 odredišnih platformi (x86, PowerPC, Alpha, ARM, JVM), ovdje se dolazi do



20 kombinacija izvornog i odredišnog jezika za koje je potrebno napisati kompajler, sa svim fazama koje kompajleri imaju. S druge strane, ako se razvije jedna neutralna međureprezentacija tada treba realizovati samo 4 leksičke, sintaksne i semantičke analize, 1 optimizaciju na nivou međureoprezentacija i 5 generisanja koda.



Međureprezentacija je svaki način predstavljanja programa između apstraktnog stabla sintakse i asemblerskog ili mašinskog koda. Pred dizajnera kompajlera se postavlja pitanje koje međureprezentacije će postojati, na koji način ih realizovati, koje su strukture i koliko su bliske mašinskom jeziku ili jeziku visokog nivoa.

Za izbor MR može se ići na korištenje postojeće MR. Najpopularnije su Java JVM i Microsoft MSIL za reprezentacije niskog nivoa, a može se iskoristiti i neka od interno

korištenih međureprezentacija iz ranijeg projekta. Korištenje postojeće MR pruža uštede zbog višestruke upotrebe, ali ona mora biti izražajna i odgovarajuća za kompajlerske operacije. Nekada to i nije slučaj. Npr. Međureprezentacija visokog nivoa sa fiksnim skupom tokena koji postoje je nepogodna za upotrebu u kompajlerima za drugačije jezike, dok je neke međureprezentacije niskog nivoa možda neoptimalno konvertovati i mašinski kod.

Ako se odluči da se dizajnira vlastita MR, treba odrediti koliko će biti bliska mašinskom kodu, koliko izražajna i koje strukture.

Ukoliko je brzina izvršavanja prevedenog programa važnija od brzine kompajliranja, treba uvesti više međureprezentacija, kombinovati ih i prevoditi iz jedne u drugu kroz različite faze.

Zavisno od orijentacije prema izvornom ili odredišnom jeziku, međureprezentacije dijelimo u tri nivoa.

MR visokog nivoa se koriste u ranijoj fazi procesa dobijanja odredišnog koda i konvertuju se kasnije u niži nivo. Primjer takve međureprezentacije je dekorisano apstraktno stablo sintakse . Ove međureprezentacije čuvaju konstrukcije jezika visokog nivoa kao što su tok programa (uslovi, petlje), varijable, metode. Sa njima su moguće optimizacije visokog nivoa koristeći osobine izvornog jezika. Primjer predstavlja zamjena konstantnih izraza, npr. A=2+3; da se transformiše u ekvivalentni izraz A=5.

MR srednjeg nivoa, predstavljaju osobine izvornog jezika na jezičko nezavisan način. One su posrednik između AST i asemblerskog jezika. Karakterišu se upotrebom nestruktuiranih naredbi skokova, registara, i memorijskih lokacija pa liče na programe u mašinskom jeziku, ali instrukcije ne moraju biti slične instrukcijama odredišnog procesora niti broj registara treba biti ograničen. Jedan primjer ovakvih međureprezentacija je troadresna mašina, čije instrukcije se pišu u obliku četvorke:

a = b OP c

Popularne su i stek mašine (kao što je Java bytecode)

MR niskog nivoa predstavljaju asemblerski kod kome se eventualno dodaju pseudo instrukcije. Na primjer, procesor Intel i386 nema instrukciju CMOV za uslovnu dodjelu vrijednosti registru a Pentium II je ima. U međureprezentaciji se može dodati i ovakva instrukcija, a pri generisanju finalnog koda se navede odredišni procesor i time odredi da li će se ova operacija obaviti posebnom instrukcijom ili simulirati drugim instrukcijama. Ove reprezentacije su mašinski ovisne i prevođenje iz njih u asemblerski kod je trivijalno jer one jesu asemblerski kod uz male dodatke. Optimizacije koje ove reprezentacije dopuštaju su niskog nivoa poput određivanja redoslijeda instrukcija i podešavanje memorijskog smještaja.

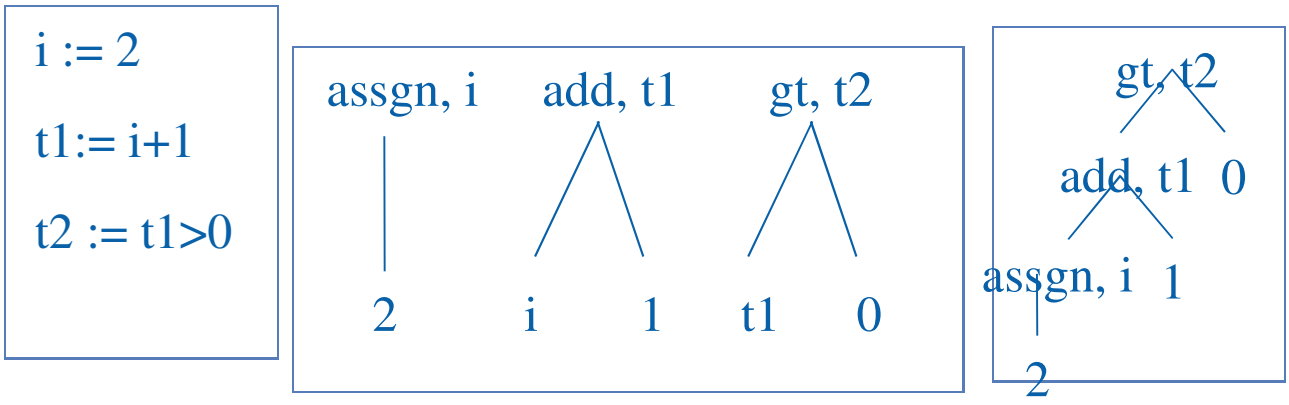
Prema strukturi međureprezentacije se dijele na grafičke, linearne i hibridne.

Grafičke međureprezentacije čuvaju predstavu programa u razgranatim strukturama, kao što su stabla i grafove. Ove strukture nisu lake za preuređivanje, ali bolje predstavljaju osobine izvornog koda i omogućavaju prepoznavanje većih programskih blokova. Mogu se koristiti za predstavljanje sintakse, toka programa ili međuzavisnosti podataka.

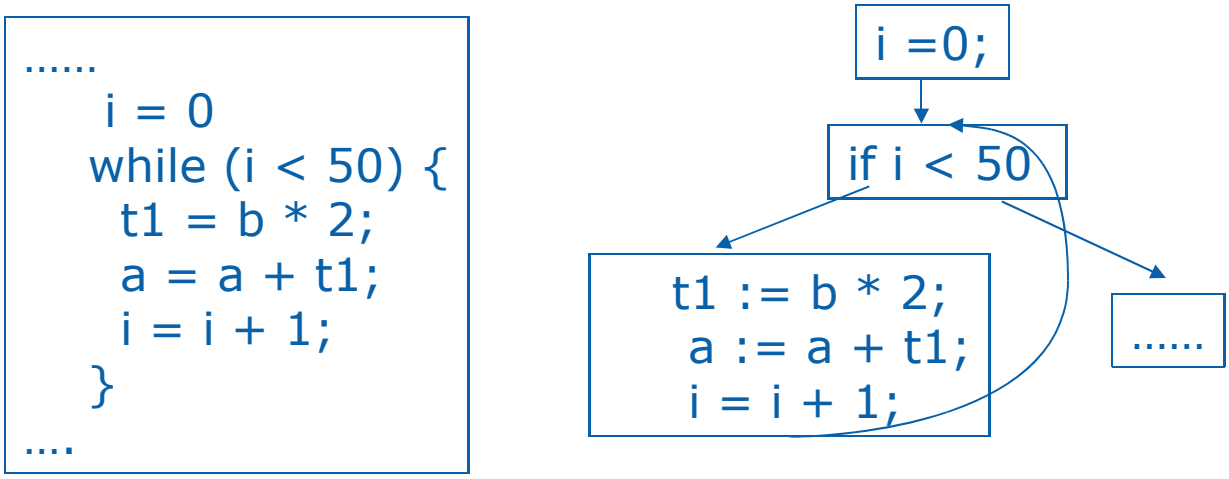
1. Prvi primjer je apstraktno stablo sintakse. Apstraktno stablo sintakse je međureprezentacija visokog nivoa, direktno dobijena iz stabla parsiranja izbacivanjem redundantnih čvorova. Ova reprezentacija čuva sintaksnu strukturu i

održava informacije izvornog koda. Primjene su u source-to-source prevođenju, semantičkoj analizi kod kompajlera i interpretera i editorima koji kontrolišu sintaksu.

2. Grafovi osnovnih blokova predstavljaju operacije koje ne uključuju programska grananja. Najčešće predstavljaju međuzavisnost operatora u aritmetičkim izrazima i sastoje se od čvorova koji imaju do dva djeteta, dok je u čvoru predstavljena operacija. Sljedeći primjer prikazuje tri jednostavne naredbe kako se predstavljaju ovakvim grafovima slikom u sredini, a zatim na desnoj strani kako se ova tri grafa povezuju u jedan u cilju bolje optimizacije.



3. Grafovi kontole toka predstavljaju programska grananja tako što se svaki čvor se odnosi osnovni blok, dio osnovnog bloka, ili pojedinačnu naredbu. Svaka linija u ovom grafu predstavlja mjesto bezuslovnog ili uslovnog skoka iz jednog dijela programa u drugi. Na sljedećoj slici se vidi predstavljanje programa razlaganjem u graf kontrole toka. Čvorovi ovog grafa su osnovni blokovi kao u sljedećem primjeru, ali mogu biti i pojedinačne naredbe.



Grafovi kontrole toka pribavljaju korisne informacije koje služe za prepoznavanje petlji, uklanjanje redundantno računanje, dodjelu registara, raspoređivanje instrukcija itd.

Osnovni blok je niz susjednih naredbi u koji tok izvršenja uđe na početku i izađe na kraju bez zaustavljanja i grananja osim na njegovom kraju. Program se siječe u osnovne blokove po sljedećem pravilu:

Počeci blokova su:

- Prva naredba programa
- Odredište uslova
- Naredbe iza grananja

Osnovni blok se sastoji od početka i svih naredbi do sljedećeg početka bloka ne uključujući njega.

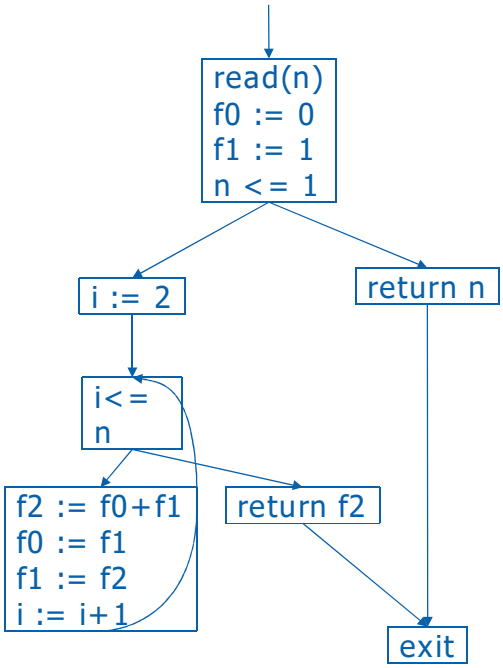
Pogledajte sljedeći primjer u višem programskom jeziku funkcije

```
unsigned int fibonacci (unsigned int n) {
        unsigned int f0, f1, f2;
        f0 = 0;
        f1 = 1;
        if (n <= 1)
                return n;
        for (int i=2; i<=n; i++) {
                f2 = f0+f1;
                f0 = f1;
                f1 = f2;
        }
        return f2;
}
```

Transformisan u međukod sa troadresnom mašinom izgleda ovako

```
read(n)
        f0 := 0
        f1 := 1
        if n<=1 goto L0
        i := 2
L2:     if i<=n goto L1
        return f2
L1:     f2 := f0+f1
        f0 := f1
        f1 := f2
        i := i+1
        go to L2
L0:     return n
```

Razbijen na graf sa osnovnim blokovima izgleda ovako:



4. Graf zavisnosti predstavlja prikaz odnosa između naredbi koji ograničavaju redoslijed izvršenja. Na primjer naredbe a:=2 i b;=8 se mogu izvršiti istovremeno ili obrnutim redoslijedom, ali naredbe a:=2 i b:=a+1 ne mogu. Kod ovih grafova se predstave međuzavisnosti podataka između naredbi. Čvorovi ovog grafa predstavljaju naredbe a linije predstavljaju zavisnosti. Ovi grafovi se grade za specifične optimizacije i odbacuju kada su optimizacije obavljene.

Primjer: Neka su date sljedeće instrukcije međureprezentacije.

```
s1          a := b + c
s2       if a>10 goto L1
s3          d := b * e
s4          e := d + 1
s5  L1:     d := e / 2
```

Uočava se pet vrsta zavisnosti između naredbi na sljedećem grafu:

a) kontrolna zavisnost: Naredbe s3 i s4 se izvrše samo za a<=10

b) Zavisnost toka ili read-after-write zavisnost: s2 koristi vrijednost definisanu u sl

c) antizavisnost ili write-after-read zavisnost: s4 definiše vrijednost korištenu u s3

d) output zavisnost ili write-after-write zavisnost: s5 definiše vrijednost definisanu u s3

e) input zavisnost ili read-after-read koja ne ograničava redoslijed izvršenja ali se koristi u optimizacijama.: s5 koristi vrijednost korištenu i u s3

Linearne reprezentacije izgledaju kao pseudokod ili program u asemblerskom jeziku,. One su lakše za preuređivanje, ali teže za prepoznavanje globalnog toka programa. Ovo su međureprezentacije nižeg nivoa MR prije finalnog generisanja koda. Njih karakterišu linearne sekvence instrukcija i liče na asemblerski kod za apstraktnu mašinu sa eksplicitnim uslovnim i bezuslovnim skokovima. Primjeri su

- Instrukcijski skup odredišne mašine
- Stack mašina
- Dvoadresna mašina
- Troadresna mašina

Stek mašina je zasnovana na operand stack-u na koji se smještaju parametri naredbi. Naredbe uzimaju podatke za steka, brišu ih i rezultat ponovo stavljaju na stek. Ovaj kod je lako generisati i kompaktnog je oblika ali je težak za preuređivanje. Primjer reprezentacije izraza x-2*y u ovom obliku

```
Push 2
Push y
Multiply
Push x
subtract
```

Dvoadresna mašina za binarne operacije koristi odredište kao jedan od izvora,. Primjer reprezentacije izraza x-2*y u ovom obliku je:

```
t1 =2
t2= y
t2 *= t1
t4 = x
t4 -= t1
```

Troadresna mašina u svim operacijama ima nula do dva izvorišta i jedno odredište.

```
t1 := 2
t2 := y
t3 := t1*t2
t4 := x
t5 := t4-t3
```

Ova reprezentacija se simbolička. Praktičnije je troadresnu reprezentaciju čuvati u formi tabele koja ima četiri kolone: operacioni kod, prvi i drugi operand, te rezultat kao u sljedećem primjeru

	op	arg1	arg2	result
(0)	Uminus	c		t1
(1)	Mult	b	t1	t2
(2)	Uminus	c		t3
(3)	Mult	b	t3	t4
(4)	Plus	t2	t4	t5
(5)	Assign	t5		a

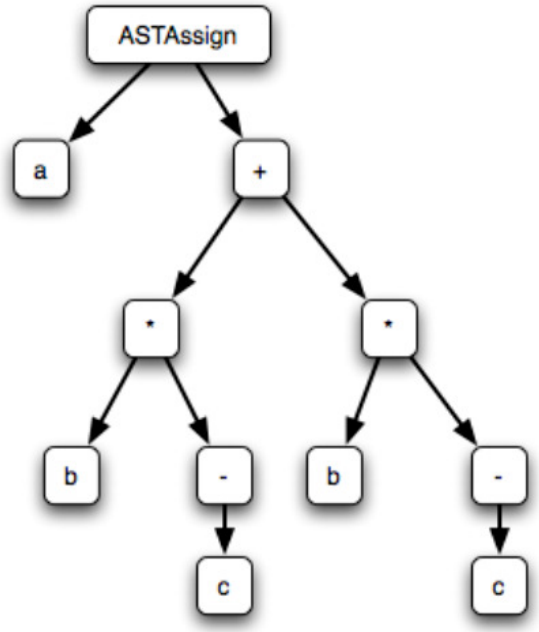
Hibridne reprezentacije su kombinacija grafičke i linearne MR. Jedan od primjera je da se program podijeli na osnovne blokove, dijelove koji se izvršavaju bez grananja unutar njih. Svaki osnovni blok se sastoji od linearne reprezentacije programskog koda, a oni su međusobno povezani vezama poput grafova za predstavljanje toka programa.

Konverzija međureprezentacija

Neka su odabrane međureprezentacije različitih nivoa. Sada treba odrediti detalje njihove implementacije i način transformacije iz jedne u drugu. Neka je odabrano

apstraktno stablo sintakse kao međureprezentacija visokog nivoa, dobijena nakon sintaksne analize.

```
a=b*(-c)+ b*(-c)
```



Ekvivalentna reprezentacija srednjeg nivoa, u obliku troadresne mašine može da izgleda ovako.

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

U ovoj međureprezentaciji troadresnim kodom uočava se više privremenih vrijednosti koje se zovu t0, tl, t2, ... One mogu čuvati vrijednosti bilo kog tipa, a tip se dobija iz drugih MR. Kako se vidi iz ovog primjera, varijable koje se javljaju u međureprezentaciji nisu samo one koje daje programer u ulaznom programu, nego ih pravi i kompajler za međurezultate. Jedna od odluka koja se treba donijeti pri realizaciji ove međureprezentacije je da li za svaku generisanu troadresnu naredbu biti uvođena nova pomoćna varijabla, ili će se one ponovno koristiti. Na primjer u izrazu x – 2 * y, imamo dva različita troadresna koda:

U prvoj varijanti različite vrijednosti koriste odvojena imena:

```
t1 := 2
t2 := y
t3 := t1*t2
t4 := x
t5 := t4-t3
```

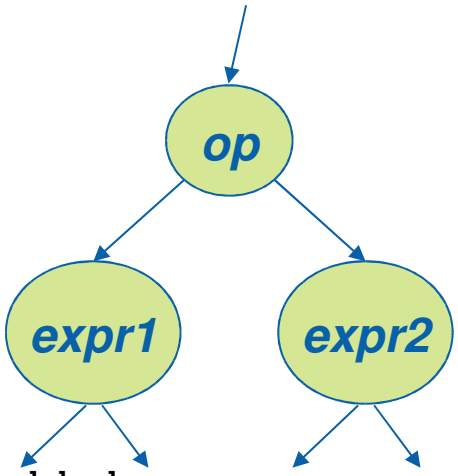
U drugoj varijanti, ako različite vrijednosti koriste ista imena ponovno korišten je privremenih varijabli štedi prostor ali kvari analizu i optimizacije. Tako rezultat tl*t2 više nije dostupan ako se tl ponovo koristi.

```
t1 := 2
t2 := y
t1 := t1*t2
t2 := x
t1 := t2-t1
```

Za reprezentacije nižeg nivoa se ove privremene varijable dalje transformišu u odgovarajuće registre ili memorijske lokacije na steku, heapu ili statički alociranom prostoru. Kompajler može odrediti da neke privremene kao i varijable koje su postojale u izvornom kodu budu u registrima.

Algoritam koji transformiše MR visokog nivoa u MR niskog nivoa zasniva se na rekurzivnom pozivu funkcije koja za svaki čvor AST stabla proizvodi code za tu vrstu čvora. Nazovimo tu funkciju generate. Generate funkcija vraća privremenu varijablu (ili registar) koji čuva rezultat

Za aritmetičke operacije dio AST stabla na slici



Realizuje se sljedećim pseudokodom:

```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Izuzetak predstavljaju operatori poput && i ||. U C i Java, oni su kratkospojni, pa se mora se ubaciti i kontrola toka.

Konverzija operatora || sa uključenim kratkospojnim računom izgleda ovako:

```
If expr1 is true, don't eval expr2
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Konverzija operatora && sa uključenim kratkospojnim računom izgleda ovako:

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( ifnot_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Operator pristupa nizovima exprl [expr2] može se realizovati sljedećim kodom.

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

Sekvenca naredbi se realizuje uzastopnim pozivom generate.

```
statement1;
statement2;
. . .
statementN;
generate(statement1)
generate(statement2)
. . .
generate(statementN)
```

Naredba uslova if

```
if (expr)
  Statement;
```

Rezultuje sljedećim kodomss

```
E = new_label()
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( E: )
```

Za petlje treba emitovati labelu za početak petlje i generisati uslov i tijelo petlje

```
while (expr)
  statement;
E = new_label()
T = new_label()
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```

Funkcijski pozivi

```
x = f(expr1,  expr2,  . . .);
```

Rezultuju generisanjem koda za smiještanje parametara na stek i pozivom potprograma.

```
a = generate(f)
foreach expr-i
  ti = generate(expri)
  emit( push ti )
emit( call_jump a )
emit( x = get_result )
```

Dodjela vrijednosti varijabli zahtijeva drugačiji tretman lijeve i desne strane u iskazima dodjeljivanja. Desna strana je vrijednost (r-value) a lijeva strana je lokacija (l-value). Zato se može uvesti posebna funkcija lgenerate koja vraća registar s adresom. Tada se može za

```
expr1 = expr2;
```

generisati

```
r = generate(expr2)
l = lgenerate(expr1)
emit( store *l = r )
return r
```

Za razne vrste čvorova dobija se opšti oblik funkcije generate

```
Reg generate(ASTNode node)
{
  Reg r;
  switch (node.getKind()) {
  case BIN: t1 = generate(node.getLeft());
            t2 = generate(node.getRight());
            r = new_temp();
            emit( r = t1 op t2 );
            break;
  case NUM: r = new_temp();
            emit( r = node.getValue() );
            break;
  case ID:  r = new_temp();
            o = symtab.getOffset(node.getID());
            emit( r = load sp + o );
            Break;
  Case IF: E = new_label()
           t = generate(expr)
           emit( ifnot_goto t, E )
           generate(statement)
           emit( E: )
  …
  }
  return r
}
```

17. Optimizacija i generisanje koda

Svrha optimizacije je dobijanje što kraćeg, bržeg ili energetski efikasnijeg mašinskog ili pseudokoda. Faze kompajlera u kojima se optimizacija obavlja su za vrijeme transformacije međureprezentacija između semantičke analize i generisanja koda, tokom generisanja koda i nakon generisanja koda.

Analiza i Transformacija

Prije optimizacije, potrebno je analizirati programski kod, kako bi se odradile optimizacije koje zahtijevaju globalno razumijevanje toka programa. Primjer takvih optimizacija je izbacivanje naredbi koje se nikada neće izvršiti, npr. tijelo naredbe if (0) { ...} u C++, premještanje dijelova koda na mjesta na kojima se brže izvršavaju, zamjena petlji prostim kopiranjem itd.

Da bi saznali gdje su petlje, koji blokovi su dostupni iz drugih blokova, ima li uslova koji su stalno neispunjeni u grananjima i petljama, obavlja se **analiza toka kontrole**. Međureprezentacije hibridnog tipa (osnovni blokovi i povezani graf između njih na mjestima gdje izvršenje prelazi sa jednog na drugi) su pogodne za analizu toka kontrole. Pored toga, može se pratiti i da li je neki podatak nepotreban, pa ako jeste, instrukcije koje ga generišu takođe se mogu izbaciti. Praćenje trenutka kada je varijabli dodijeljena vrijednost i trenutka kada je ta vrijednost čitana je osnova **analize toka podataka**.

Ove dvije analize osiguravaju bezbjednu optimizaciju, da uklanjanje nekog koda ne dovede do prestanka ispravnog izvršavanja programa. Informacije koje za ovo trebaju nisu očigledne i direktno navedene u kodu, nego je potrebna sistematična analiza

Primjer: uklanjanje neaktivnog koda u reprezentaciji četvorkama. Neka je dat osnovni blok sa sljedećim instrukcijama;

```
x = y + 1;
y = 2 * z;
x = y + z;
z = 1;
z = x;
```

Prva i četvrta instrukcija se mogu ukloniti jer se dodijeljena vrijednost varijablama x i z nije čitala do nove dodjele. U ovom slučaju informacije o tome šta se može izbaciti su dobijene analizom toka podataka.

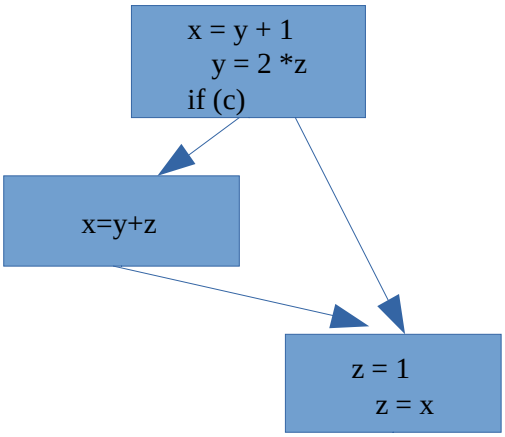
Situacija se komplikuje ako se mora uključiti i analiza toka kontrole. U sljedećem primjeru se prva naredba ne smije uklanjati, jer ako je c=false, podatak x iz prve linije može biti korišten.

```
x = y + 1;
y = 2 * z;
if (c) x = y + z;
z = 1;
z = x;
```

Još teže je uočiti međuzavisnosti toka kontrole ako je međureprezentacija nižeg nivoa, jer imamo labele i skokove bez eksplicitnih uslova i petlji. Ovaj primjer bi zapravo u reprezentaciji četvorkama izgledao ovako.

```
label1:
   x = y + 1
   y = 2 * z
   jumpifnot c label3
   x = y + z
label3:
   z = 1
   z = x
```

Stoga je ključ za uspješnu globalnu optimizaciju takva međureprezentacija koja sadrži **graf toka kontrole**, (CFG), koji predstavlja grafičku predstavu programa i uključuje tok računa i kontrole. Čvorovi ovog grafa su osnovni blokovi, koji su sekvence negranajućih naredbi, a linije predstavljaju tok kontrole na mjestima gdje se obavljaju skokovi na labelu. Svaki blok ima više ulaznih/izlaznih linija.



Problem međuzavisnosti instrukcija sada se razbija na dva problema:

Kako se informacija o definisanju i korištenju podataka propagira između instrukcija?

Kako informacija ide između blokova prethodnika i sljedbenika?

Uvešćemo najprije pojam aktivnih varijabli. To su varijable čija će vrijednost biti potrebna nakon instrukcije i osnovnih blokova. Skup aktivnih varijabli prije instrukcije I naziva se in[I] , a skup aktivnih varijabli nakon instrukcije I naziva se out[I].

Pitanje 1: za svaku instrukciju I, koja je veza između in[I] i out[I]?

```
in[I] = {y,z}
x = y + z
out[I] = {x,z}
in[I] = {y,z,t}
x = y + z
out[I] = {x,t}
in[I] = {x,t}
x = x + 1
out[I] = {x,t}
```

U prvom primjeru prije instrukcije x=y+z aktivne su varijable y i z, a nakon nje aktivne su x i z. Kako se aktivnost određuje? Gledanjem unazad. Znajući varijable aktivne nakon I, mogu se zaključiti varijable aktivne prije I.

- Sve varijable koje I koristi su aktivne prije I. Zovu se ***uses*** I
- Sve varijable aktivne nakon I su također aktivne prije I, osim ako I piše u njih. Zovu se ***defs****od* I

Sljedeća skupovna jednačina daje odnos između in i out varijabli.

in[I] = (out[I] – def[I]) ∪ use[I]

$$out[B] = \bigcup_{B' \in succ(B)} in[b']$$

Primjer: Neka je dat osnovni blok:

```
aktivne1:
    x = y+1
aktivne2:
   y = 2*z
Aktivne3:
   if (d)
aktivne4:
```

Za svaku izvršnu tačku ovog osnovnog bloka, definisaće se skupovi aktivnih varijabli.

aktivnel = in[B] = in[I1]

aktivne2 = out[I1] = in[I2]

aktivne3 = out[I2] = in[I3]

aktivne4 = out[I3] = out[B]

Odnos između skupova aktivnih varijabli definiše se sljedećim skupovnim jednačinama.

aktivnel = (aktivne2 – {x}) ∪ {y}

aktivne2 = (aktivne3 – {y}) ∪ {z}

aktivne3 = (aktivne4 – {}) ∪ {d}

U ovom osnovnom skupu tok informacija je *unazad,*što znači da su potrebni out[] skupovi za račun in[] skupova. Informacije se šalju prema gore.

Pitanje 2: za svaki osnovni blok B, sa sljedbenicima B_1, ..., B_n, koji je odnos između out[B] iin[B_1] ... in[B_n]

Primjeri:

Pravilo: Varijabla je aktivna na kraju bloka B ako je aktivna na početku *svih* sljedbenika. Ovo uzima u obzir sva moguća izvršenja, mada je konzervativno neki putevi se možda neće desiti

Zajedno se dobija sistem jednačina toka podataka.

$$in[I] = (out[I] - def[I]) \cup use[I]$$
$$out[B] = \bigcup_{B' \in succ(B)} in[B']$$

Jednačine rješavamo algoritmom

```
Postavi sve skupove aktivnih varijabli in[...] i out[...] na  prazne skupove.
Ponavljaj sljedeće dodjele sve dok se vrijednoti in[...] i out[...]  mijenjaju
   Za svaku instrukciju I izračunaj
      in[I] = ( out[I] – def[I] ) ∪ use[I]
   Za svaki osnovni blok
      out[B] = ∪ in[B']
```

Nakon dobijanja stabilnog stanja saznaje se koje varijable nisu aktivne nakon svake instrukcije, pa se mogu primijeniti optimizacije eliminacijom mrtvog koda.

Optimizacije

Optimizacije predstavljaju transformacija koda u cilju dobijanja boljeg. Kriterij koji kaže šta je to bolji kod, zove se metrika. Metrike mogu biti:

- Performance: vrijeme, instrukcija, ciklusi
- Prostor: Smanji upotrebu memorije
- Veličina koda
- Energija

Optimizacije možemo podijeliti u tri kategorije.

Tradicionalne optimizacije transformišu program za smanjenje obima posla. One se koriste nad međureprezentacijama višeg i srednjeg nivoa. Primjeri: propagacija i pakovanje konstanti,algebarska pojednostavljenja, eliminacija neaktivnog koda itd. Ove optimiozacije obično poboljšavaju sve metrike.

Opcionalne transformacije ne poboljšavaju kod same po sebi jer djeluju na poboljšanje samo nekih metrika, npr. rezultuju bržim ali dužim kodom. Primjeri su inlining, odmotavanje petlji.

Dodjela resursa je mapiranje programa na specifična hardverska svojstva. Ovo su optimizacije vrlo niskog nivoa. Primjeri ovakve optimizacije su dodjela registara, raspoređivanje instrukcija, paralelizam data streaming i prefetching

Tehnike optimizacije u međureprezentacijama visokog nivoa

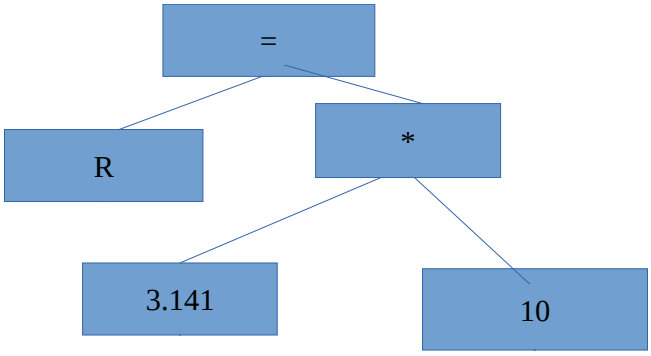
Kada je međukod u formatu u kome se ima globalna slika programa (dekorisano apstraktno stablo sintakse) mogu se izvesti određene optimizacije

Pakovanje konstanti

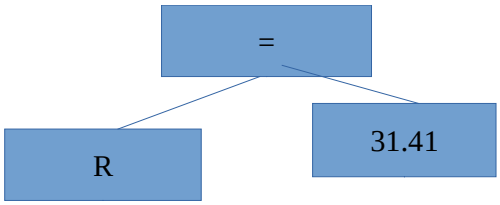
U mnogim programima su operandi izrazi čiji dijelovi mogu biti izračunati i u toku kompajliranja. Na primjer

R = 3.141 * 10;

Dio ASN u koji je preveden iztaz izgleda ovako



U analizi stabla je očigledno pojednostavljenje



Algebarska pojednostavljenja

Za specijalne slučajeve operanda mogu se primijeniti algebarska pojednostavljenja:

a * 1 ≡ a

a/1 ≡ a

a * 0 ≡ 0

a - a ≡ 0

b || false ≡ b

Ponavljanjem ovih optimizacija se skraćuju kompleksni izrazi. Pri implementaciji ove optimizacije obratiti pažnju na komutativnost i asocijativnost operatora.

Eliminacija nedostupnog koda

U ovoj vrsti optimizacije izbacuje se kod koji se nikad neće izvršiti U sljedećem primjeru, naredba if postaje nepotrebna i može se kompletna izbaciti.

```
#define DEBUG 0
. . .
if (DEBUG)
  print("Current value = ", v);
```

Ova optimizacija se obično obavlja na visokom nivou, jer je lako uočiti u apstraktnom stablu sintakse dijelove koji odgovaraju if (false) ili while (false) naredbama. Puno je teže realizovati na srednjem ili niskom nivou kada se te konstrukcije pretvore u uslovne i bezuslovne skokove.

Izbacivanje invarijantog koda iz petlje

Ako se račun neće mijenjati između iteracija petlje, izbaci ga izvan petlje. Na primjer:

```
for (i=0;i<N;i++)
  A[i] = A[i] + x*x;
```

Može se zamijeniti sa

```
t1 = x*x;
for (i=0;i<N;i++)
  A[i] = A[i] + t1;
```

Ova optimizacija nije trivijalna. Treba odrediti kada je izraz invarijantan. Pri analizi dijela apstraktnog stabla sintakse koje predstavlja petlju uočiti one dijelove stabla koji ne koriste varijablu brojačke petlje, ne koriste ni jednu varijablu čija se vrijednost mijenja u toku petlje, ne primaju korisničke podatke i ne pozivaju po0tprograme sa bočnim efektima.

Smanjenje snage

Ova optimizacija je slična algebarskom pojednostavljenju. Mogu se zamijeniti spore operacije (mult, div) bržima (add, sub, bit shift)

x * 2 ≡ x+x

x * 2^c ≡ x << c

x/2^c ≡ y >> c

Mogu se prepoznati i neke karakteristične operacije u petljama

```
for (i=0;i<N;i++) {
  v = 4*i;
  A[v] = y
}
```

i zamijeniti množenje sabiranjem

```
v = 0;
for (i=0;i<N;i++) {
  A[v] = y ;
  v = v + 4;
}
```

Inlining

U nekim slučajevima se zamijeni poziv funkcije njenim tijelom. Npr.

```
void a() {x=8; }
void b() { a(); }
```

se zamijeni sa

```
void a() {x=8; }
void b() { x=8; }
```

Treba biti oprezan s ovom optimizacijom. Prvo ne smiju se rekurzivne funkcije ugrađivati. Dalje, ova optimizacija (mada dovodi do ubrzanja) uglavnom povećava veličinu koda. Kompajleri koriste heuristiku da odrede kada će se ovo raditi, što se svodi na problem ranca.

Inlinig je koristan u objektno orijentisanim jezicima koji imaju puno kratkih metoda, tipično jednolinijskih za inicijalizaciju varijable instance. Kod ovih metoda je naredba dodjele često kraća od naredbi poziva potprograma, pripreme parametara i čišćenja steka.

Eliminacija podizraza

Ako potprogram računa isti izraz više puta, može se zamijeniti njegova vrijednost varijablom. Na primjer

```
a = b + c;
c = b + c;
d = b + c;
```

Može se transformisati u

```
t = b + c
a = t;
c = t;
d = b + c;
```

Gornji primjer pokazuje da treba paziti u ovoj optimizaciji: podizraz se može zamijeniti novom varijablom samo dok njegovi elementi ne dobiju novu vrijednost

Propagacija konstanti

Ako je vrijednost varijable poznata kao konstanta u trenutku kompajliranja, može se zamijeniti varijablu konstantom

```
n = 10;
c = 2;
s = s + i*c;
n = 10;
c = 2;
s = s + i*2;
```

Varijabla se može zamijeniti konstantom samo ako se može dokazati da je varijabla konstantne vrijednosti.

Pojednostavljenje toka kontrole

Ako znamo rezultat uslova, eliminiši nekorištenu granu. U sljedećem primjeru else dio izraza i test uslova se može obrisati.

```
if (10 > 5) {
  ...
} else {
  ...
}
```

Kombinacijom drugih optimizacija, kao što su propagacija konstanti, pakovanje konstanti otkrivaju se ovakve, očito besmislene pojave u kodu mogu desiti.

Tehnike optimizacije u međureprezentacijama srednjeg nivoa

Pretpostaviće se da je troadresna mašina sa zapisom uređenih četvorki (operacija, odredište, izvor1, izvor2) korištena kao međureprezentacija srednjeg nivoa. U ovoj fazi mogu se ponoviti neke od tehnika optimizacije visokog nivoa i koristiti neke nove.

Pakovanje konstantii

Razradom izraza i njihovim konvertovanjem u reprezentaciju srednjeg nivoa, javljaju se nove mogućnosti pakovanja konstanti. Na primjer, prevođenje indeksa niza cijelih brojeva:

x = A[2];

Dovodi do sljedeće međureprezentacije troadresnim kodom:

```
t1 = 2*4
t2 = A + t1
x = *t2
```

Očigledno, se prva naredba može zamijeniti sa

t1=8

Eliminacija podizraza

Indeksi nizova i polja struktura stvaraju nove izraze koji se ponavljaju i mogu zamijeniti jednostavnijim, ali sada na nivou međureprezentacije srednjeg nivoa. Na primjer sljedeće dvije linije izvornog koda

```
x = A[i];
y = B[i]
```

rezultuju međukodom:

```
t1 = 4*i;
t2 = A + t1;
x = *t2;
t3 = 4*i;
t4 = B + t3;
y = *t3;
```

u kome se sekvenca 4*i ponavlja 2 puta, pa se naredba t3 = 4*i može zamijeniti sa t3=t1.

Eliminacija neaktivnog koda

Ako se rezultat neke naredbe nikad ne koristi, tu naredbu možemo izbaciti. U sljedećem primjeru

```
x = y + 1;
y = 1;
x = 2 * z;
```

Vidimo da se vrijednost varijable x koja je definisana u prvoj liniji nije korištena do treće linije kada je postavljena na novu. Stoga ekvalentni kod posataje:

```
y = 1;
x = 2 * z;
```

Za postizanje ove optimizacije treba generisati graf međuzavisnosti varijabli. Varijabla je neaktivna ako nije korištena nakon dodjele i ako se ukloni kod koji dodjeli vrijednost neaktivnoj varijabli, ovo tranzitivno otvara nove detekcije neaktivnog koda.

Propagacija kopije

Nakon dodjele x = y, zamijeni upotrebe x sa y, jer će to omogućiti kasniju efikasniju upotrebu registara

x = y;

z = y

Se mijenja u

x=y

z=x

Ove zamjene se mogu koristiti samo do naredne dodjele varijabli x ili y.

Tehnike optimizacije u međureprezentacijama niskog nivoa i generisanju koda

Međureprezentacija srednjeg nivoa razlikuje se od mašinskog i asemblerskog jezika da bi se omogućilo generisanje koda za različite pozadine. Apstrakcija srednjeg nivoa je pojednostavila mnoge optimizacije koje su već opisane, ali su one koje su moguće na niskom nivou su još brojnije.

Razlike između međureprezentacije na srednjem nivou i one koje su vezane za instrukcijski skup arhitekture u tome što međureprezentacije pružaju prost, uniforman skup operacija, dok ISA pružaju mnogo specijalizovanih instrukcija.

Izbor instrukcija: Često jedna instrukcija radi više operacija u međureprezentaciji, ali i obrnuto. Kako odabrati instrukcije?

Prosto rješenje je mapirati svaku MR operaciju u instrukciju (ili više njih) uz odgovarajuće memorijske operacije, na primjeri, instrukcija srednjeg nivoa

x = y + z;

se mapira u npr. sljedeće instrukcije niskog nivoa za i386

```
mov EAX,[y]
mov EBX, [z]
add EAX,EBX
mov [x],EAX
```

Problem sa ovim pristupom je neefikasna upotreba instrukcijskog skupa. Instrukcijski skup pruža više načina da se uradi ista stvar, i prethodni primjer se mogao realizovati i kraće i brže.

Sljedeći primjer prikazuje kako se međureprezentacija srednjeg nivoa za naredbu a[i+1]=b[j] transformiše u i386 instrukcije gledajući osnovne adresne režime

t1 = j*4
t2 = b+t1
	mov eax,[j]
	imul eax,4
	add eax,b

t3 = *t2	mov ecx,[eax]

t4 = i+1
t5 = t4*4
t6 = a+t5
	Mov ebx,[i]
	Inc ebx
	Imul ebx,4
	add ebx,a

*t6 = t3	Mov [ebx],ecx

Ali ako se koristi indeksno adresiranje sa SIB bajtom, može se dobiti brži i kraći kod.

t1 = j*4 t2 = b+t1	mov eax,[j] Mov ecx,[4*eax+b]
t3 = *t2	mov ecx,[eax]

t4 = i+1
t5 = t4*4
t6 = a+t5
	mov ebx,[i]
	Inc ebx
	Mov ebx,[4*ebx+a]

*t6 = t3	Mov [ebx],ecx

Arhitekturalne razlike među procesorima jako utiču na izbor instrukcija. RISC procesori (PowerPC, MIPS) u svim aritmetičkim operacijama zahtijevaju da se podaci eksplicitno stavljaju u registre

```
ld      8(r0), r1
add     $12, r1
```

CISC procesori (x86) imaju kompleksne instrukcije (npr., MMX and SSE). Kod njih aritmetičke operacije mogu pristupiti memoriji, pa se prethodni efekat postiže samo jednom memorijskom operacijom po instrukciji.

```
add     eax,[esp+8]
```

Izbor instrukcija dodatno usložnjavaju adresni režimi, jer neki procesori imaju bogat izbor režima adresiranja. Na primjer, na x86 ima instrukciju

add [EAX+2*EBX-8],EBX

U kojoj se adresa odredišta računa kao linearna kombinacija vrijednosti registara.

Za česte slučajeve generisanja koda za specifičnu arhitekturu uvode se idiomi. ***Idiom*** je jedna instrukcija koja predstavlja uzorak ili uobičajenu sekvencu instrukcija za određenu arhitekturu. Nekada su idiomi neočekivani. Na primjer, postavljanje registra EAX na vrijednost 0 je preporučljivije na i386 uraditi naredbom XOR EAX,EAX nego očiglednijom instrukcijom MOV EAX,0 jer je prva varijanta kraća i brža. U drugoj varijanti neposredni operandi se kodiraju u instrukciju, praveći je većom i skupljom za dohvatanje i izvršenje.

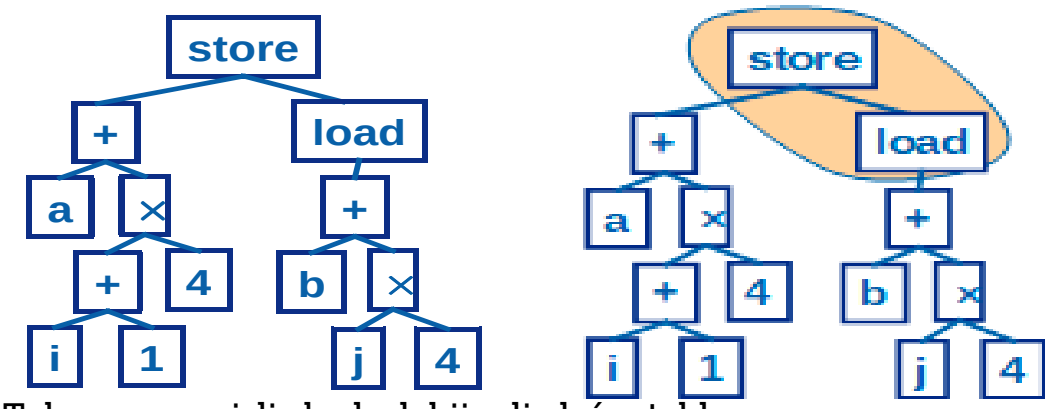
Ali lista idioma je konačne veličine, i na njoj se ne može zasnovati optimalni izbor instrukcija. Pošto se međureprezentacije srednjeg nivoa, formata troadresne mašine mogu dosta razlikovati od odredišnog mašinskog jezika, treba nam generalnije mapiranje iz međureprezentacije srednjeg nivoa u mašinski jezik.

Iz linearnog formata međureprezentacije se ne mogu saznati optimalne instrukcije koje treba generisati ukoliko one zavise od nesusjednih instrukcija međureprezentacije. Na primjer iz ove međureprezentacije srednjeg nivoa za naredbu a[i+1]=b[j]

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

se ne vidi mogućnost generisanja instrukcije move.l (A0), (A1) koja kopira vrijednost sa memorijske lokacije na drugu za Motorola 68000 procesor, jer se čitanje prve lokacije obavlja u trećoj instrukciji a upis u drugu u sedmoj.

Odgovarajući uzorci instrukcija se mogu naći dodatnom reprezentacijom srednjeg nivoa, koja je dobijena nakon završene optimizacije na srednjem nivou. Reprezentacija troadresnim instrukcijama transformiše se ponovo u reprezentaciju

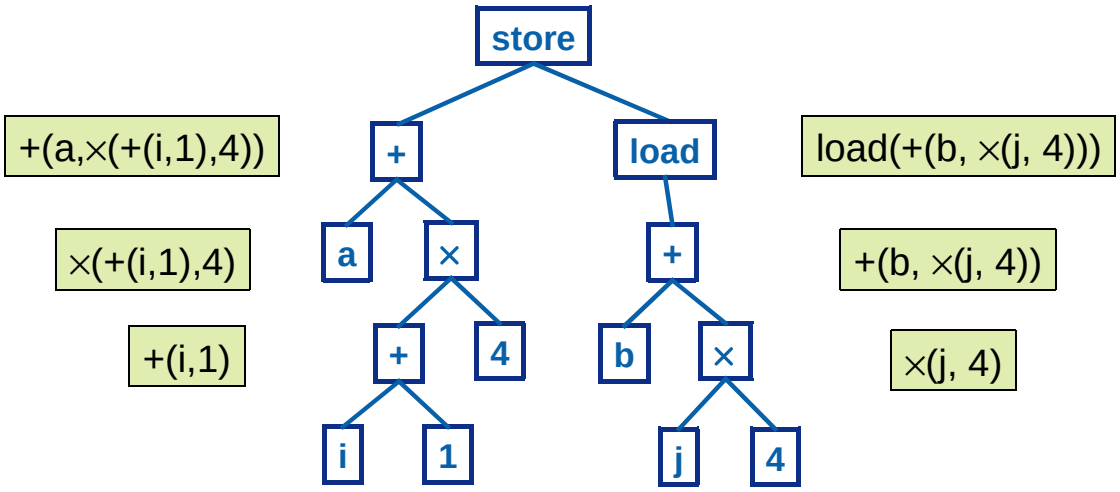


stabla. Tako se za ovaj dio koda dobija sljedeće stablo:

Iz ovakve reprezentacije sada se direktno vidi da su čvorovi store i load zapravo povezani i da se mogu predstaviti jednom cjelinom koja se zove pločica. Pločica odgovara mašinskoj instrukciji.

Stablo možemo posmatrati i kao izraz u prefiksnoj formi.

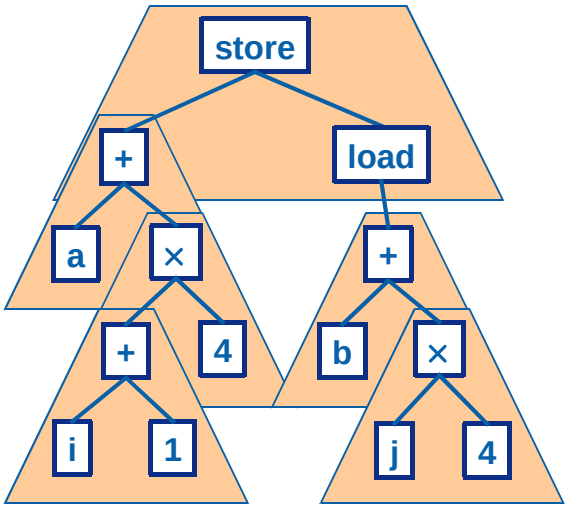
store(+(a,×(+(i,1),4)),load(+(b, ×(j, 4))))



U tablici se mogu držati uzorci dijelova ovog stabla. Ona sadrži Pravilo, Uzorak za slaganje i zamjenu, cijenu, predložak za generisanje koda a može uključiti akcije, npr. Generisati ime registra

#	Pattern, replacement	Cost	Template
1	$+(reg_1,reg_2) \rightarrow reg_2$	1	`add r1, r`
2	$\times(reg_1,reg_2) \rightarrow reg_2$	10	`mul r1, r`
3	$+(num,reg_1) \rightarrow reg_2$	1	`addi num, r1`
4	$\times(num,reg_1) \rightarrow reg_2$	10	`muli num, r1`
5	$store(reg_1, load(reg_2)) \rightarrow done$	5	`movem (r2),(r1)`

Rekurzivnim procesom iz ovog stabla i prepoznatih pločica u njemu dolazimo do nove reprezentacije niskog nivoa.



Assembly

```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem (rb), (ra)
```

Dodjela registara

Rad sa podacima u registrima je znatno brži od rada s podacima u memoriji, pa je poželjno podatke što je više moguće čuvati u registrima. U sljedećem primjeru sabiranja na x86, ako se varijable čuvaju u registrima imamo kraći i brži kod nego kada su u memoriji

mov eax,[1000] add eax,[2000]	mov eax,esi add eax,edi

Na ARM procesorima, koji imaju više raspoloživih slobodnih registara, a zahtijevaju da se većina operacija obavlja između registara, uštede su i veće.

Nažalost, dodjela registara nije lagan problem. Malo je dostupnih registara, na mnogim arhitekturama broj raspoloživih registara je jednocifren. Uz to neki registri se koriste za posebnu namjenu, što još dodatno ograničava spisak raspoloživih registara. Primjer: registri za cijele brojeve, registri za brojeve u pokretnom zarezu, registri za duple riječi, akumulator, registri grananja, stack pointer, konstantni registri (R0 na MIPS Risc je konstanta), string registri ESI i EDI na x86, flag registar...

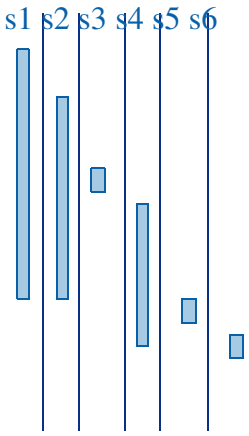
Dodjela registara je optimizacija sa najvećim efektom na performance i zato joj se treba posvetiti pažnja bez obzira na spomenute poteškoće.

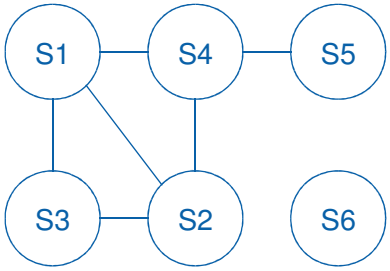
Pravilna odluka koje podatke treba držati u memoriji a koje u registrima dobija se algoritmom koji za svaki podatak gleda graf međuzavisnosti podataka i zaključi koliko vremena je podatak aktivan. Algoritam izgleda ovako:

- Dodijeli područja aktivnosti za svaku vrijednost
- Odredi preklapajuće opsege (interferencija)
- Izračunaj korist od čuvanja vrijednosti u toku područja aktivnosti u registru (cijena)
- Dodijeli fizičke registre (dodjela)
- Generiši kod

Područja aktivnosti se dobijaju grafom međuzavisnosti podataka. Podatak je aktivan od trenutka kada mu je dodijeljena vrijednost do trenutka kada se zadnji put koristi u tom dijelu koda, ili ponovo dodjeljuje obično osnovom bloku. Na sljedećoj slici uz šest naredbi dodjele različitim varijablama vidi se područje njihove aktivnosti u srednjem dijelu slike. Na primjer, varijabla s4 je prvi put korištena u četvrtoj liniji a zadnji put u

```
s1 ← 2
s2 ← 4
s3 ← s1 + s2
s4 ← s1 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

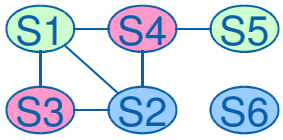




šestoj.

Preklapajući opsezi za svaki podatak se izdvoje u graf poput onog na desnoj strani slike. Npr. Podatak s3 se istovremeno koristi sa S1 i S2, dok S6 se ne preklapa ni sa jednim.

Sada se problem izbora registara svodi na problem bojenja grafova. Svaki čvor dobije boju što znači da svakom području aktivnosti se dodjeljuje registar. Ako dva čvora imaju vezu između njih ne mogu imati istu boju , što znači da ako dva područja aktivnosti interferiraju ne mogu koristiti isti registar



Prethodni graf se mogao obojiti u tri boje, što znači da se mogu koristiti tri registra. Isti registar može privremeno čuvati podatke sl i s5, drugi može čuvati s3 i s4 a treći s2 i s6. Prije dodjele registra drugom podatku, podatak se snima u memoriju. Prethodni primjer, sada izgleda ovako sa dodjelom registara.

```
r1 = 2
r2 = 4
r3 = r1 + r2
s3 = r3
r3 = r1 + 1
s1 =  r1
r1 = r1 * r2
s2 =  r2
r2 = r3 * 2
```

Raspoređivanje instrukcija

Moderni mikroprocesori nastoje da što više aktivnosti obavljaju paralelno. Primjer predstavljaju protočne strukture, višejezgreni procesori, procesori sa veoma širokom instrukcijskom riječi itd. Protočna struktura omogućava da se svaka instrukcija završava jedan mašinski ciklus iza prethodne, pod uslovom da su podaci koji su potrebni toj instrukciji raspoloživi na vrijeme. Ako to nije slučaj, izvršavanje će trajati duže.

Sljedeći MIPS RISC primjer se izvršava 10 ciklusa jer registar $1 u trenutku izvršenja druge instrukcije još nije spreman, a isto se događa i sa registrom $3 na početku četvrte instrukcije.

lw $1,0($0)

add $2,$2,$1

lw $3,4($0)

add $4,$4,$3

Raspoređivanje ili promjena redoslijeda instrukcija može se vršiti radi poboljšanja performanci i garantovanje tačnosti. Promjena redoslijeda prethodnog primjera u naredni će smanjiti vrijeme izvršenja na 8 ciklusa.

lw $1,0($0)

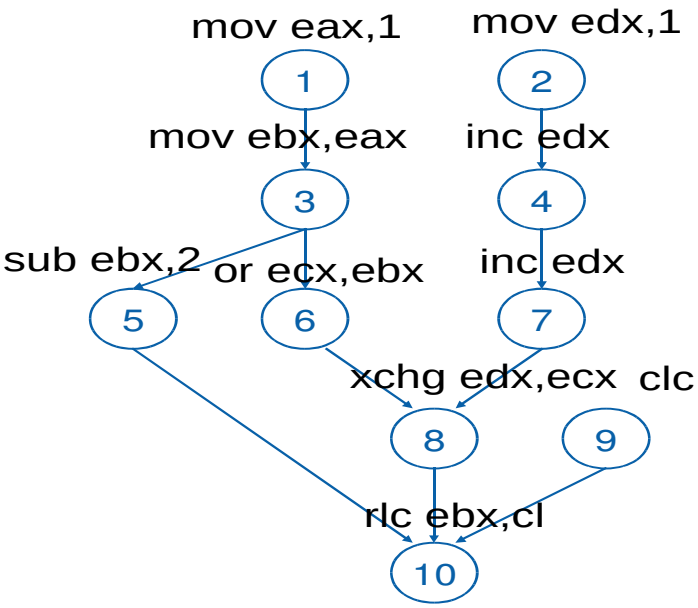
lw $3,4($0)

add $2,$2,$1

add $4,$4,$3

Treba reći da su ove optimizacije veoma mašinski specifične. Nije rijetka situacija da novija verzija procesora, potpuno jednakog instru kcijskog skupa kao prethodna ima drugačija pravila kako redoslijed instrukcija utiče na vrijeme izvršavanja. Stoga se ove optimizacije trebaju raditi u kasnoj fazi generisanja koda.

Za implementiranje algoritma raspoređivanja instrukcija koristi se tablica koja uzima u obzir očekivana kašnjenja svake od instrukcija. Iz podataka koje instrukcije obrađuju kreira se graf zavisnosti instrukcija



Gledajući vrijeme kašnjenja, i zavisnosti, odredi se prioritet. Na primjer, nakon prve instrukcije mov eax,1 kandidati za drugu su mov edx,1 ili mov ebx,eax. Ako je pravilo

procesora da upotrebaistog registra u dvije susjedne instrukcije izaziva zastoj, onda se bira mov edx,1.

Pozicioniranje osnovnih blokova

Većina optimizacijskih tehnika koje su spomenute se obavljaju unutar jednog osnovnog bloka. Ali na performanse utiče i redoslijed osnovnih blokova u memoriji. Kada je program velik, on će imati više grešaka stranice i promašaja u u tablici stranica u procesoru (TLB). Loše pozicioniranje osnovnih blokova povećava greške keša, a nasumično pozicioniranje povećava šansu za ove efekte, jer skokovi između osnovnih blokova koji su u različitim stranicama može povećati broj grešaka stranice.

Problem se smanjuje pozicioniranjem osnovnih blokova na takve lokacije da oni blokovi između kojih se često grana budu na bliskim memorijskim lokacijama.Ova optimizacija analizira staze izvršenja i preuredi kod da one budu u kontinualnoj memoriji. U nekim slučajevima se bezuslovni skokovi iz jednog osnovnog bloka u drugi zamjenjuju nadovezanim osnovnim blokovima.

Ako ima uslovnih skokova, tj. grananja s nejednakom vjerovatnoćom izvršenja, osnovni blokovi se poredaju tako da se učini češćim slučaj nastavljanja na sljedeći blok, dok se rjeđi slučajevi dosežu naredbom uslovnog skoka. Podaci o tome koja instrukcija je češća dobivaju se empirijski: naredba if ima dva skoka, jedan za preskakanje kada uslov nije ispunjen i jedan za preskakanje else sekcije, kada je izvršen kod sa ispunjenim uslovom. Prema vrsti relacionog operatora može se pretpostaviti koja sekcija se češće izvršava (npr za operator > češća je then sekcija, a za operator jednakosti češća je else sekcija) i preurediti program da se češći osnonvni blok završi bez naredbe skoka.

Prednosti ove optimizacije su

- Duže sekvence bez grananja
- Više operacija u kešu
- Gušći tok instrukcija
- Manje hazarda u cjevovodu
- Prebacivanjem rjeđeg koda manje grešaka stranice

Tehnike optimizacije nakon generisanja koda

Nakon kompajliranja, ima još ima nedostataka koje prethodne tehnike optimizacije nisu otkrile. Savršeni kod se neće nikada postići jer su raspoređivanje i alokacija NP-Complete problemi, i ma kako se unapređivala optimizacija, optimizator možda nije implementirao svaku transformaciju. Može se uložiti dodatni rad na raspoređivanju i dodjeli registara i implementirati nove optimizacije *ili* Optimizirati kod nakon kompajliranja

Peephole optimizacija

Osnovna ideja ove vrste optimizacije je otkriti lokalna poboljšanja gledajući dio koda. Ovdje se ne gleda međuzavisnost cijelog asemblerskog programa , nego se koristi mala okolina koda. Za svaku instrukciju se posmatra se okolina od npr 5 instrukcija ispred i iza nje. U njoj se uoče idiomi koji se mogu zamijeniti. Npr: ako se uoči

```
MOV EBX,1000
MOV [EBX],EAX
ADD EBX,4
```

Ova sekvenca se zamjenjuje sa

```
MOV [1000],EAX
MOV EBX,1004
```

Peephole optimizacija ima dva osnovna pristupa. Klasična peephole optimizacija (McKeeman) ima ograničen skup ručno kodiranih uzoraka. Obavlja se detaljna pretraga uzoraka u malim prozorima, sa mali skupom uzoraka i bržim izvršenjem. Generisanje koda je lokalno

Moderna Peephole Optimizacija (Davidson, Fraser) namijenjena je za kompleksnije ISA i veći skup uzoraka. Pogodnija je i za situacije kada se nove verzije procesora proširuju novim instrukcijama koje ubrzavaju rad a nisu podržane starom verzijom kompajlera, kao i kada neke optimizacije srednjeg nivoa nisu podržane kompajlerom. Optimizator se sastoji iz tri dijela: Ekspander konvertuje asemblerski listing u MR niskog nivoa. Simplifier jednom prođe kroz međureprezentaciju I obavi dodatna pakovanja konstanti, algebarska pojednostavljenja I beskorisne efekte Matcher usaglasi uzorke, zamijeni kod i generiše asemblerski program.

Optimizacija u vrijeme linkovanja

Linker može poboljšpati performanse programa pravilnim smiještanjem potprograma. Ako se uočava da potprogram A zove B, linker može staviti A i B na susjedne lokacije, čime se pokušava da oni budu u istoj memorijskoj strani. Ista strana smanjuje radni skup u operativnom sistemu (i greške stranice) a susjedne lokacije smanjuju konflikte memorijskog keša. No, mnogi potprogrami mogu zvati B ili A, i za donijeti odluku kako poredati potprograme u cilju poboljšanja performansi potreban je graf poziva. Graf poziva se može napraviti statički ili dinamički.

U statičkoj verziji, ako se na više mjesta u potprogramu A poziva potprogram B, pravi se graf sa linijama između A i B, uz oznake linija sa brojem koliko se puta B poziva iz A. Jasno, ovaj pristup poziv potprograma unutar petlje računa kao jedan poziv.

Dinamičko pravljenje grafa poziva potprograma se radi profiliranjem. U ovom pristupu linker ubaci dodatni kod pri svakom pozivu potprograma koji uvećava brojač. Brojači su u statički inicijaliziranoj memoriji i pri probnoj upotrebi programa evidentiraju koliko je koji potprogram pozivan. Kasnije linkovanje koristi te informacije, optimalno poreda potprograme i izbaci brojače.

Optimizacija u toku izvršenja

Sasvim drugi pristup je prebacivanje optimizacija niskog nivoa u vrijeme izvršenja. U jezicima poput Java i C# se isporuči bytecodes (tj. MR) umjesto mašinskog. Razlika pristupa je što se binarni kod izvrši na mašini a bytecode se izvrši na *virtualnoj* mašini. No, u Just in time kompajlerima, prilikom učitavanja programa, bytecode se konvertuje u mašinski kod koji je prilagođen aktivnom procesoru. Zahtjevi za memorijom su veći i vrijeme pokretanja sporije, ali su moguće optimizacije niskog nivoa koje ne bi bile moguće kod binarnog koda generisanog za srodnu ali različitu mašinu. Npr. Procesor Pentium I nema naredbu CMOV (uslovna dodjela), Pentium III je ima, ali je neće koristiti ako je program kompajliran za stariji procesor. Pri izvršenju, ako je program u byte kodu, Just in time kompajler može prepoznati pravi procesor i generisati adekvatan kod.