Programski jezici

Osvrt na različite programske jezike promotrimo najprije iz povijesne perspektive. Najstarija računala zahtijevala su da programer prevede sve algoritme na strojni jezik. Taj je pristup dao još veću važnost izučavanju algoritama, jer je efikasniji algoritam smanjivao broj linija programskog koda i broj prepisivanja heksadecimalnog koda sa kojima se opisivala naredba, a time i vjerojatnost pogreške.

Prvi korak prema pojednostavljivanju ovog zadatka bilo je izostavljanje brojeva kojima se predstavljao op-code i operandi strojnih instrukcija. S tim u vezi, postalo je jako popularno pridruživati različite mnemonike različitim op-kodovima i koristiti ih umjeto heksadecimalnog prikaza. Tako bi primjerice, umjesto korištenja op-koda za punjenje registra (ili pohranjivanje) sa vrijednošću iz memorije, programer bi pisao LD ili ST (skraćenice od LOAD i STORE). U slučaju operanda, postojala su pravila koja su dozvoljavala korištenje opisnih imena za lokacije u memoriji koja su se koristila umjesto memorijskih adresa u instrukcijama. Pri tom, registrima su se pridjeljivala imena R0, R1.

Na taj je način, korištenjem mnemonika, naziva za memorijske lokacije i registre, znatno se povećala čitljivost programa. S tim u vezi, usporedimo dva programa, jedan napisan u strojnom jeziku, a drugi «preveden» na jezik mnemonika:

156C	LD R5, Price
166D	LD R6, Troškovi
5056	ADDI R0, R5, R6
3065	ST R0, UkupnaCijena
C000	Hlt

Dakle, lokaciji 6C pridružili smo ime Cijena, lokaciji 6D TroškoviNabave, lakaciji 6E ime UkupnaCijena. Sasvim je jasno, da forma na desnoj strani tablice predstavlja puno bolji način za razumijevanje rutine od strojnog jezika. Kada su se ove tehnike pojavile, programeri su ih koristili kao način da program napišu na papir, a na računalo su ga prenosili u strojnom jeziku. Nije trebalo dugo, pa da se i sam postupak prevođenja prepozna kao procedura koju može izvoditi samo računalo. Tako su se razvili programi nazvani asembleri koji su prevodili program napisan mnemoničkim kodom u strojni jezik (assembler jer su se odgovarajući mnemonici i oznake slagali sa op-kodom odnosno operandom). Mnemonički sustav razrađen za predstavljanje programa prepoznat je kao jezik i nazvan asemblerskim jezikom.

S vremenom, strojni jezik smatran je prvom generacijom strojnih jezika, a assembler II generacijom. Iako je asembler imao niz prednosti u odnosu na strojni jezik, ipak mu je nedostajalo cjelovitije okruženje. Primitive su bile iste kao i kod strojnog jezika, razlika je postojala samo u sintaksi, pa je program napisan u asembleru ovisio o procesoru kojem je bio namijenjen i nije se baš jednostavno prenosio na računalo sa drugim tipom procesora jer je trebalo voditi računa o novom obliku instrukcijskog skupa i drugačijim registrima. Drugi nedostatak očitovao se u načinu koji je programer bio prisiljen usvojiti ukoliko je želio nešto isprogramirati. Programer je bio prisiljen razmišljati o malim, inkrementalnim koracima strojnog jezika. Situacija je usporediva sa primjerom vezanim za problem opisa gradnje kuće ako bismo morali razmišljati o položaju svake cigle, daske i slično. Činjenica je da u nekom trenutku treba elemente kuće predstaviti i na takav način, ali je o funkcionalnosti projekta lakše i razumljivije govoriti koristimo li veće jedinice kao što su sobe, kuhinja, vrata, prozori i sl.

Ukratko, elementarne primitive od kojih je proizvod sastavljen, ne moraju biti one koje

koristimo kada ga projektiramo. Projektirati je lakše koriste li se primitive na većem nivou na kojem svaka primitiva označava koncept. Slijedom ove filozofije, počeli su se razvijati novi programski jezici prilagođeniji razvoju softwarea od asemblerskih jezika. Tako su nastali jezici treće generacije koji su se od prethodne razlikovali u primitivama koje su bile na višem nivou i neovisne o procesoru. Tipični primjeri su FORTRAN (FORmula TRANslator) za matematičke i znanstvene proračune i COBOL (Common Business Oriented Language) kojeg je razvila US Navy za poslovne aplikacije.

Zajednička karakteristika programskih jezika 3. generacije bio je skup primitiva pomoću kojih možemo razvijati program. Svaka takva primitiva trebala se moći predstaviti sa skupom nižih koje možemo predstaviti strojnim jezikom. Identificirani skup primitiva program prevodioc je prebacivao u strojni jezik. Takvi su programi ličili na asemblere druge generacije, samo što su jednu primitivu pretvarali u čitav niz strojnih instrukcija.

Alternativu kompajlerima predstavljaju interpreteri kao još jedna karakteristika jezika III generacije. Ti su programi slični prevoditeljima, samo što ujedno i izvode instrukcije nakon što ih prevedu, a ne pohranjuju ih za neku buduću upotrebu. Znači, kao rezultat rada interpretera nemamo neku vrstu kopije programa u strojnom jeziku u memoriji nego postupak prevođenja nastupa uvijek kada pokrećemo program.

Neovisnost o stroju

Razvojem treće generacije jezika, neovisnost o vrsti računala na kojem se izvodi program uvelike je postignut. Kako se izjave u jezicima treće generacije ne odnose na atribute ni jednog procesora, mogu se lako prevoditi za bilo koje računalo. Teoretski, program napisan jezikom treće generacije može se koristiti na bilo kojem stroju ako za njega imamo kompajler.

Stvarnost je međutim pokazala da stvari i nisu baš tako jednostavne. Nakon što se kompajler osmisli, određene karakteristike stroja na koji se primjenjuje ponekad se odražavaju kao uvjeti na jezik kojeg prevodimo. Primjerice, različiti načini na koji procesori izvode U/I operacije uvjetovali su da isti jezik ima različite karakteristikeili dijalekte za različite strojeve. Zbog toga je potrebno makar malo promijeniti program premiještamo li ga sa stroja na stroj. Problem se može usporediti sa nedostatkom podudarnosti koja postoji u smislu riječi različitih jezika-za nešto što se na jednom jeziku kaže sa jednom ili dvije riječi, na drugom treba izraziti čitavom rečenicom.

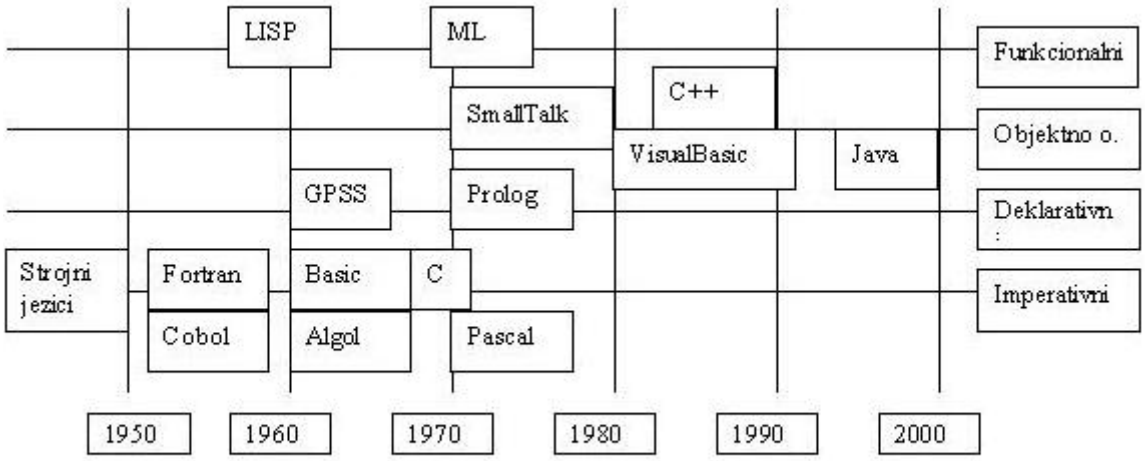
Dakle, jezici III generacije nisu ispunili uvjet prenosivosti i neovisnosti o platformi, što i nije bilo toliko tragično jer:
- programi su se uvijek mogli prenositi uz minimalnu izmjenu
- ideja o prenosivosti razvila se do te mjere da je inicirala dosezanje dalekosežnijih ciljeva-tipičan primjer je ideja o razvijanju programskog okruženja koje bi omogućilo komunikaciju u puno slobodnijoj formi od one ograničene sintaksom i semantikom programskih jezika. Kao rezultat nastao je čitav spektar programskih jezika koji ruši klasičnu podjelu na generacije programskih jezika.

Programske paradigme (pristupi)

Generacijski pristup programskim jezicima temelji se na linearnoj skali na kojoj je položaj jezika određen stupnjem kojim je korisnik jezika oslobođen zahtijeva iz svijeta računala.

Prema takvoj podjeli, na krajnje lijevom dijelu skale bili bi jezici pomoću kojih se problemi rješavaju u okruženju u kojem se ljudi prilagođavaju karakteristikama stroja, a što se više primičemo desnom dijelu skalu, nailazimo na jezike kod kojih stroj udovoljava ljudskim karakteristikama. U stvarnosti, razvijanje programskih jezika nije teklo na ovakav način, nego duž različitih skala koje karakteriziraju različiti pristup procesu programiranja (karakteriziraju paradigmu). Stoga bi se sam povijesni razvoj bolje predstavio koristimo li više usporednih linija, od kojih svaka karakterizira odgovarajući pristup.

Tako imamo:



razvojne linije sa različitim programskim jezicima koje im pripadaju.

Imperativna ili proceduralna paradigma predstavlja tradicionalni pristup procesu programiranja. Na imperativnoj paradigmi temelje se algoritmi napisani pseudokodom i oni napisani strojnim jezikom. Kao što samo ime govori, imperativna paradigma definira postupak programiranja kao niz sekvenci naredbi koje na odgovarajući način manipuliraju sa podacima da bi dobili rezultat. Procesu programiranja pristupamo pronalaskom algoritma kojeg zatim izražavamo nizom naredbi.

U suprotnosti sa imperativnom paradigmom, koje traže od programera da smisli algoritam, je deklarativna paradigma koja od programera traži da se problem opiše u formi koja je prilagođena nekom općem algoritmu (primjer iz prologa sa predavanja). U takvom okruženju programer je taj koji daje precizne izjave vezane za problem, a ne onaj koji otkriva algoritam. Glavna poteškoća u razvijanju programa koji se temelje na deklarativnoj paradigmi je postojanje i otkrivanje takvog općeg algoritma. Zbog toga su prvi deklarativni jezici ličili na strogo namjenske, konstruirani za specifične aplikacije. Primjerice, deklarativni pristup bio je jako dugo korišten za simulaciju sustava (ekonomskog, fizičkog, političkog...) sa ciljem testiranja hipoteza. Kod takvih postavki, u osnovi je algoritam koji predstavlja proces u kojem se simulira odmicanje vremena uzastopnim preračunavanjem vrijednosti parametara (trgovinski deficit, rast domaće proizvodnje) na temelju prijašnjih vrijednosti. Primjena deklarativnog jezika znači da je netko jednom primijenio algoritam za neku ponavljujuću proceduru. Programer ima jedino za zadatak opisati odnose među parametrima koji se simuliraju.

Funkcionalna paradigma gleda na proces razvoja programa kao na spajanje unaprijed

definiranih "crnih kutija" od kojih svaka prihvaća određene ulaze i daje odgovarajuće izlaze. Matematičari te kutije promatraju kao funkcije što je i razlog imenu za ovaj pristup. Primitive funkcionalnih programskih jezika sadrže elementarne funkcije od kojih programer mora konstruirati zahtijevnije funkcije. Zato programer koji koristi takav programski jezik gleda na razvoj software kao na pronalaženje načina na koji se spajaju elementarne, primitivne funkcije da bi se proizveo sustav koji proračunava željeni rezultat. Kao primjer možemo uzeti funkciju koja računa srednju vrijednost: konstruirana je od funkcije koja računa sumu svih ulaznih vrijednosti i funkcije koja broji koliko tih ulaznih vrijednosti imamo, pa pišemo

(Divide (SumaBrojeva)(BrojemBrojeva)).

Primjer 2:

Pretpostavimo da imamo funkciju Sort koja nam sortira listu i drugu funkciju-First, koja pronalazi prvi član sa liste. U tom će nam slučaju funkcija

(First(Srt(List))

vratiti najmanju vrijednost sa liste.
Zagovornici funkcionalne paradigme naglašavaju da kreiranje složenog softwarea od predefiniranih primitivnih funkcija vodi prema dobro organiziranom sustavu čija je struktura prirodno modulirana terminima funkcija. U trenutku kada se razvije nova funkcija, ona može postati primitiva nekoj složenijoj. (Originalna verzija LISPa imala je samo nekoliko primitiva, a današnji lisp ih ima stotine). Funkcionalna paradigma stvara okruženje u kojem se hijerarhija strukture lako nazire.

Kod objektno orijentirane paradigme, jedinice sa podacima su aktivni objekti, a ne pasivne strukture kao kod imperativne paradigme. Da bi pojasnili promotrimo neku od listi sa imenima. Kod tradicionalne, imperativne paradigme, lista je zapravo skup podataka. Bilo koji program koji poseže za listom, mora imati algoritam za izvođenje neke akcije. Stoga, lista je pasivna u smislu da sa njom barata neki program izvana. Sama po sebi nema načina za reorganizacijom. Kod objektno orijentiranog pristupa, liste su objekti sastavljeni od pasivnih lista i procedura koje omogućavaju da se liste organiziraju (ubaci novi ulaz, poslože...). Program koji poseže za listom ne mora imati ni jednu od navedenih rutina. Za razliku od operativne rutine koja će složiti listu, program koji pristupa listi složenoj u duhu OOP-a, zatražit će od liste da se sortira. Drugi je primjer GUI. Ikone su objekti. Svakoj je pridružen skup istih procedura koje opisuju kako objekti reagiraju na vanjske događaje-što rade kada jedan put kliknemo, kako se ponašaju kada držimo pritisnut botun na mišu i slično. Svaki objekt egzistira kao odvojena, zasebna jedinica. Jednom kada su osobine ovog entiteta definirane, iste se mogu ponovno koristiti kad god nam zatrebaju. Kod OOP sve komunikacije izvode se na uniforman način što je direktno primjenljivo za upotrebu na mreži.