УНИВЕРЗИТЕТ У БЕОГРАДУ

ФИЛОЗОФСКИ ФАКУЛТЕТ

Александар А. Зорић

НОВИ ПРИСТУП СТАТИСТИЧКОЈ АНАЛИЗИ ПОДАТАКА У ПСИХОЛОГИЈИ: ОД РИГИДНИХ СТАТИСТИЧКИХ ПАКЕТА КА ФЛЕКСИБИЛНОМ СИСТЕМУ

докторска дисертација

Београд, 2012

Чланови комисије:

др Лазар Тењовић, доцент

Универзитет у Београду, Филозофски факултет

др Горан Опачић, доцент

Универзитет у Београду, Филозофски факултет

др Срђан Богосављевић, редовни професор

Универзитет у Београду, Економски факултет

Датум одбране: _______________

НОВИ ПРИСТУП СТАТИСТИЧКОЈ АНАЛИЗИ ПОДАТАКА У ПСИХОЛОГИЈИ: ОД РИГИДНИХ СТАТИСТИЧКИХ ПАКЕТА КА ФЛЕКСИБИЛНОМ СИСТЕМУ

Резиме

У раду су укратко приказана четири популарна статистичка пакета, који се донекле разликују у својим филозофијама. Два од њих се заснивају на "отвореном" коду статистичких процедура, коду је доступан јавности. Аутор се залаже за популаризацију ове струје, са основном хитпотезом да писати код користећи програмски језик, који са својим једноставним правилима не дозвољавају недореченост и непотпуну дефинисаност, приморавају аутора кода да употпуности разуме, схвати и научи методу коју покушава да алгоритмизира. У раду је предложен нов императивни матрични језик, који се по својој синтаски ослања на популарну синтаску језика C, а у себи садржи неколико нових конструката примењивих на операције са матрицама. На крају је дат пример програма написаног у коду предложеног језика за једну једноставну статистичку процедуру.

Кључне речи

ЈЕЗИК, СТАТИСТИКА, ПРОГРАМИРАЊЕ

Научна област

Психологија

Ужа научна област

Методологија психолошких истраживања

УДК број

519.21/.24:159.9(043.3)

A NEW APPROACH TO STATISTICAL DATA ANALYSIS IN PSYCHOLOGY: FROM RIGID STATISTICAL PACKAGES TO FLEXIBLE SYSTEMS

Abstract

The paper briefly outlines the four popular statistical packages which are somewhat different in their philosophies. Two of them are based on the "open source" philosophy of statistical procedures; the code of these procedures is publicly available. The author calls for the popularization of this practice, with basic idea that writing a code using a programming language, the language that is based on the simple rules that do not allow ambiguity or incomplete definitions, is forcing the author to fully understand the code and what more to understand and learn the method that he is trying to code. The paper suggests a new imperative language for matrix manipulation, which is based on the popular syntax of C language. The proposed new lanuguage contains several new constructs applicable to operations with matrices. At the end, an example program, for a simple statistical procedure coded in the proposed language, is presented.

Key words

LANGUAGE, STATISTICS, PROGRAMMING

Academic Expertise

Psychology

Major in

Methodology of Psychological Research

UDC number

519.21/.24:159.9(043.3)

Садржај

Увод....................1

Постојећи статистички системи....................11

SPSS....................11

SAS....................14

STATA....................16

R....................18

Циљ рада....................22

Предлог новог језика....................23

Типови података....................23

Матрице....................24

Мапа....................25

Дефиниција матрице, вектора преко операције распона....................26

Предефинисане и специјалне вредности....................27

Променљиве - Операција доделе....................27

Оператор груписања....................28

Елемент матрице, субматрица....................29

Промена вредности елемента....................31

Брисање редова, колона матрице....................32

Транспон матрице....................32

Аритметичке операције....................33

Логичке операције....................39

Релације....................40

Функције....................42

Корисничке функције....................44

Системске функције....................56

Матрична алгебра....................57

Специјалне функције над матрицама....................61

Основне математичке функције....................68

Функције дескриптивне статистике....................70

Функције псеудослучајних бројева....................73

Функције за рад са стринговима....................73

Функције за испис....................76

Функције о основним информацијама матрица....................80

Команде тока....81
Петље....81
Функције за рад са подацима....85
Имплементација језика....93
Пример....101
Испис....105
Закључак....107
Литература....109

Увод

Језик је основа комуникације, али и наравно много више од тога. То је начин на који мислимо и решавамо проблеме. Ово важи како за природне језике, тако и за језике који су намењени комуникацији људи и машина, и то пре свега рачунара, где је ову чињеницу много лакше спознати. Комуникација рачунара и људи, њихових корисника постаје све важнија, како број ових машина у људском окружењу расте. Овај раст за сигурно има позитивно убрзање, тако да се може лако закључити да ће ова комуникација постајати све важнија, ако већ у садашњем тренутку није постала довољно важна и значајна. Језик је наравно основ ове комуникације, и прављење језика преко кога би се рачунару задала жељена трансформација је била основна идеја која је довела до великог броја различитих језика. Рачунари разумеју јако једноставан језик састављен само од нула и јединица који уствари представља операције манипулисања са меморијским јединицама у које је могуће уписати само два стања, са и без електричног напона, што се у логичком представљању своди на нуле и јединице. Овакав језик је наравно беспотребно прекомпликован за људе јер је наравно састављен од речи, слогова које је јако тешко разликовати, зато су први језици били заправо симболичко представљање ових нула и јединица у речи, слова која су људима носили мало више информација и које је стога било лакше запамтити, тј. разумети. Појаваила је се класа језика која се зове системским језицима и са којима је могуће контролисати све ресурсе рачунара. Али због ових базичних функционалности коришћење апстракција вишег реда је постало веома компликовано јер оно захтева коришћење великог броја израза (основних трансформација) како би се постигао жељени резултат. Зато су се почели правити језици вишег реда, који су поседовали одређене структуре преко којих су се лако представљале само одређене специфичне апстракције. И тако су се почели појављивати језици који су били специфични за одређену област, нпр. језици за рад са сликама, грађевинским конструкцијама, великим табелама података,

форматирање текста, итд. Међу овим језицима се издвајају и језици који су били намењени раду статистичара, и то је управо сегмент рачунарских језика који су предмет овог рада.

Програмски језик је престао да буде само начин комуникације са машином већ пре свега формализација апстрактних структура које се могу преставити на основу скупа рачунарских инструкција које овај треба да изврши, а са друге стране и човеку да њиховим коришћењем реши жељени проблем. На други начин речено програмер конципира решење коришћењем апстрактних структура које програмски језик садржи. Структуре програмског језика су градивни блокови чијом комбинацијом програмер решава задати проблем. Овде је нагласак на комбинацији, дакле решење мора бити од структура које су дефинисане у језику. На тај начин језик моделира и начин мишљења програмера (види у Gentleman, 2004). Или како то наводи Чејмберс по коме је „један од критеријума за вредновање доприноса неког језика управо његова подршка корисницима да представе своје идеје кроз поменути језик" (Chambers, 2000) .

Класификација програмских језика има пуно, на основу њихових различитих карактеристика. Једна од основних је рецимо да ли се програми који су у њима писани претходно у целини преводе на машински језик да би се извршили или се преводи и извршава линија по линија. Али сем ових, аутор би рекао превасходно техничких одлика, постоје и одлике које се базирају на коришћеној синтакси језика. На основу синтаксе језике је могуће поделити на императивне, тј. процедуралне и деклеративе, тј. функционалне (види нпр. у Abelson et al., 1996). Императивни језици крећу од инструкције, процедуре коју рачунар треба да изврши, дакле предикта, а затим наводе објекат над којим се извршава трансформација. Они се заправо базирају над трансформацијама вредности у меморији, зато је операција доделе можда и 90% свих операција у коду програма писаним једним оваквим језиком. Деклеративни језици не специфицирају трансформације него резултате који се желе добити, дакле не како већ шта. И они се заправо своде на спецификацију функције којом се добија жељени резултат, стање. У овим језицима операција доделе практично да нема смисла, то је пре

операција једнакости којом се хоће рећи да су два стања иста. Још је једна подела овде битна, а то је да ли је акцената на објекту или на предикту исказа. Ако је објекат на првом месту у исказу онда те језике зовемо објекто орјентисаним. Рецимо да хоћемо да прекопирамо објекат1. У процедурално орјентисаним језицима ова операција би избледала овако[1]

```
copy objekat1
```

на првом месту је инструкција, предикат а затим (евентуално и субијекат те) објекат. У објектно орјентисаној синтаски ово операција би била

```
objekat1.copy()
```

на првом месту је објекат који има одређена својства и одређене методе које може да извршава, и од којих је једна и метода `copy`, тако да се у овој класи језика прво специфицира објекат па предикат. Ово наравно није промена само у синтакси, јер у језику обично не постоји промена „само у", већ то увек носи са собом више или мање других последица. Ова промена има за последицу да су мање важне трансформације, а више податак, објекат, структура и комуникација између њих. Јер да би објекти комуницирали мора постојати одређени начин слања порука од једног до другог, па макар и преко посебног објекат који би био нека врста огласне табле, са порукама од једних до других.

Програмски језици су динамична класа, у којој отрилике сваких десет година долази до појаве нових језика који са собом носе и нове парадигме програмирања. У случају језика који се примењују у статистици, може се погледати приказ животног циклуса једног језика (Leeuw, 2005, Valero-Mora, Udina, 2005) у овом случају језика XLISP-STAT. Још увек није дата јасна анализа зашто неки језици добију на популарности, а неки нестану јако брзо. Али генерално стоји да

1 У раду је латинично писмо кориштено за представљање програмског кода, као и за називе програмских језика тј. статистичких пакета. У случају назива кориштена су велика латинична слова, тако да је назив програмског језика који се изговара са „це" обележено са латиничним словом C, а стастистичког језика који се изговара са „ес" обележено са S

програмски језици, или бар њихова популарност нису дугог века, и да би програмер требао сваке године на научи један нови језик. Јер сваки језик са собом носи и одређени начин решавања проблема (Weisberg, 2005), што је уједно самим тим и одређени начин размишљања. Што доводи уствари и до категоризације који су проблеми решиви, а који нису, напросто све зависи које алгоритамске па и менталне објекте поседујемо, те из које перспективе посматрамо проблем.

Програмски језици у статистици

Једна битна одлика језика, коју наводе корисници специјализованих статистичких језика јесте његово извршавање у интерактивном моду. Дакле могуће је задати групу команди, које ће се извршити све заједно а онда наставити анализу података команду по команду. Дакле после извршења једне инструкције погледају се резултати па се у зависности од њих изда нова команда итд., односно постоји интеракција корисника и система, на основу добијених резултата претходне анализе (Weisberg, 2005, Lang, 2000, Tierney 2005).

Рачунари у статистици имају велику улогу, толико велику да постоји чак област која се зове енг. statistical computing, у преводу статистичко рачунарство, Разне симулације: Монте Карло, bootstraping, визуелизација података, процена модела, и процена његовог поклапања са подацима се углавном обавља кроз процедуре које се понављају више стотина пута, и за које је очигледно потребан рачунар (Gentleman, 2004). Овде се на употребу рачунара гледа као и на употребу математике, као на алат који истраживачу омогућава да разуме добијене податке. Али опет и овде треба подвући да употреба рачунара и квалитативно мења приступ истраживача, јер се неке методе интерактивног процењивања и евалуације података на основу графичких приказа („аналитички графици“) управо заснивају на рачунаревој могућности да у реалном времену изврши жељену анализу (Weisberg, 2005). Учење из податка тј. откривање података би била прва парадигма коју су рачунари дали статистичарима (Chambers, 2000, Mahalanobis, 1965). Али наравно транслација математичког алгоритма у рачунарски није један на један, потребно је узети у обзир физичка ограничења рачунара, како наводе

Алтман и Мек Доналд, погрешни рачунарски алгоритми воде наравно и ка погрешним резултатима (Altman, McDonald 2001). Ово би биле окоснице тих главних знања која чине статистичко рачунарство.

Прављење рачунарског ситема са програмским језиком за употребу у статистици мора узети у обзир шароликост његових корисника. Ведерил и сарадници наводе четири основне врсте корисника статистичких алата: 1) статистичког експерта, 2) примењеног статистичара који најчешће ради у некој грани индустрије 3) статистичара почетника 4) статистичара аматера (Wetherill and all, 1985). Различите су потребе и очекивања ових корисника од статистичког пакета (језика), експерт жели флексибилност и брзину јер он тачно зна шта жели да добије, и жели на лак и једноставан начин да своје идеје представи рачунару. Примењени сатистичар можда не зна колико и експерт, али је можда под још већим временским ограничењем да донесе резултате из прикупљених података, он такође цени брзину и лакоћу коришћења, али жели и да програмски систем буде тај који ће вршити и контролу његовг рада, макар на нивоу да га сигнализира да неки од добијених резултата не морају бити релеванти, што због статистичких разлога, што због примене погрешне методе. Почетници не знају статстику, и не желе ни да је знају, они желе само закључке, код њих систем практивчно треба да одради скоро све самостално и да поседује сва могућа упозорења и контроле о легитимности примене методе на податке које они имају. Статистичари аматери су можда најтежа група јер мало знају, и обично тврде да знају шта желе а то је најчешће погрешно. Због тога систем и његов језик који би могли да изађу у сусрет свим тповима корисника, морали би под број један бити довиљно флексибилни. Поред тога морали би бити и модуларни како би се омогућило свим корисницима да једноставно и лако управљају системима, сваки тип корисника би имао свој модул дизајниран по мери њихових потреба и знања.

Статистички језик би морао да садржи следеће елементе: 1) модул за унос и валидацију података 2) експлорацију података, 3) формулација модела и анализа, 4) модул за даље валидације и провере модела, 5) модул за апликацију модела на другим подацима. У фази дизајна се мора мислити и о имплементацији тачних и

ефикасних нумеричких алгоритама. Програмабилан кориснички интерфејс, као и презентација резултата (табеле и графикони), те портабилност кода на друге оперативне системе, структуре података које је могуће лако записати и прочитати из фајла, те сигурност података су неки аспекти на које се мора мислити при осмишљавању будућег окружења намењеног статистичару. Окружење би тебао да има подршку за модуларне пакете који се могу додавати и самим тим надградити постојећи систем. Задавање команди за различите типове корисника, мора бити прилагођено начину на који ови користе окружење, командни интерактивни мод за напредне кориснике и мени интерфејс са почетнике. Контрола квалитета језика кроз тестне податке са унпред познатим резултатима, је такође нешто о чему је потребно размишљати. Како наводи Мек Кулог (McCullough, 1999), на примеру комерцијалних и већ увелико коришћених система, грешке у софтверу (багови) постоје, само је питање да ли су откривене. И ако су откривене корисници очекују да ће се те грешке и исправити, тј. да је предложени пакет „жив“, да неко води бригу о њему. Комерцијални аспекти, одлике које корисници очекују од комерцијалних система је постојње документације и одржавање програма (нове верзије са исправкама грешки, побољшани алгоритми, ...). Тачна и свеобухватна документација, интерактивна помоћ при раду и дескриптнивни описи грешака, су такође нешто што можда представља главни чинилац на основу кога се корисници определују за или против предложеног решења.

Прилагодљивост различитим типовима корисника је јасно једна од најважнијих обележија које добар статистички језик треба да поседује. Језици за анализу података и статистику морају да покривају читав спектрум: од импривзације и брзог моделирања до имплементације у подате који се са неком фреквенцом прикупљају по истом стандарду (нпр. иста структура података се прикупља сваком месеца), као и специјалне системе за рутинске анализе (Huber, 2000). За посао статистичара се тако може рећи иде од екплорације до продукције, и један систем би требао да може да му помогне у читавом том процесу. Оно што Хубер у свом раду јасно наводи је и постојање два вида комуникације која се често превиђају када је реч о програмским језицима, а која би ипак требало обавезно

узети у обзир. Поред стандардних комуникација човек-машина и машина-машина, потребно је обратити пажњу и на комуникацију машина-човек и човек-човек, значи начин на који рачунар представља податке кориснику, и оно што је такође битно је размена података и резултата између различитих корисника.

Чејмберс, отац програмског језика „C" и касније сарадник у прављењу његове верзије отвореног, јавног кода „R", наводи (Chambers, 2000) да је за статистички језик потребно да омогући:

1. лако задавање једноставих задатака
2. постепено усложњавање модела, а самим тим и прецизирање задатака
3. могућност проширења програмирања и ка нижим нивоима апстракције, тј. прелазак ка системском програмирању
4. имплементирање прорачуна високог квалитета у смислу прецизности али и брзине извршења
5. повезивање наведених захтева у један нови једноставни алат.

Са стране корисника њему је алат потребан да податке организује (у виду табеле података са варијаблама које имају додатна обележија и редовима који представљају вредности мерених објеката на овим обележијима), да податке визуелизира (као начин да корисник прими велику количнину информација које подаци садрже) и да их анализира (овде спадају процедуре нумеричке линеарне алгебре, сортирања и табулације, као и технике оптимизације и генерисање псеудо-случајних бројева које алат треба да омогући). Ово су потребе корисника, а да би језик то могао да подржи потребно је по Чејмберсу да:

1. постоји кориснички интерфејс који се може креирати из самог језика, битно је да постоје методе програмиране у самом језику које ово омогућавају
2. језик мора да омогући прављење нових функција у самом језику, дакле језик би требао да има могућност проширавања
3. све структуре у језику, укључујући и сам језик али и податке, треба да су представљени као објекати који поседују самодескрипцију. (Ово је

парадигма објектно орјентисаног програмирања које је у доба писања чланка била јако популарна.)

4. језик треба да има могућност лаке дефинициеје податка потребних за извршавање одређене функције, ово уствари представља неку врсту експлицитне дефиниције типова података али и више од тог стандардног значења које се придаје у рачунарсту где се под тим мисли да се променљивој придаје тип целобројног тј. реалнег броја тј. текстуалног податка. У статистици је потребно дефинисати и податке као што су номинална, бинарна варијабла, квадратна, регуларна матрица, итд.
5. када је реч о објектима, које би језик требало да поседује, мора бити омогућено наслеђивање њихових метода и одлика
6. постојање једноставаног приступа свим осталим ресурсима система, његовим програмима, библиотекама, ... – постојање универзалног интерфејса. Ово је битна одлика јер је сигурно да један систем неће имати све, зато је важно од почетка размишљати о његовом повезивању са осталим алатима.

Сличног је става је и Ланг (Lang, 2000), по коме језик који ће се користити за статистичке анализе мора да има :

1. спектар алата, који омогућавају поновно коришћење кода
2. могућност прошираваења, надоградње на друге системе тј. постојање интерфејса ка другим програмским окружењима и системима
3. могућност рада са великим базама података, јер како се све информације пребацују у електронски облик, података је све више и више, и ствара се потреба за њиховим анализирањем
4. доступност методологије, тј. да нови поступци буду доступни корисницима за лако коришћење, не само преко лаког корисничког интерфејса већ и преко документације нових процедура и услова за њихово коришћење, ово је поновно подсећање на постојање корисника којима статистика није примарна делатност

5. употреба Интернета који ће систему омогућити интеграцију са другим сиситемима и колаборацију више корисника на истом проблему
6. ефикасност – језици више генерације су обично и спорији, а у случају великих табела података то брзина извршавања постаје једна од битних карактеристика система
7. интеграција са другим библиотекама које су можда писане и у другим језицима, такозваним системским језицимама какви су C, C++, FORTRAN.

Ланг заговара коришћење Java језика за један такав систем који би могао да обједини све ове захтеве, јер заиста Java обезбеђује да систем буде лако портабилан и да буде модуларан, једини проблем је ефикасност, али са новим верзијама Jave она све мање и мање заостаје у брзини за системским језицима који код преводе у извршни бинарни код рачунара.

Зашто нови програмски језик

У раду који објашњава зашто треба писати нове статистичке програме, Стромберг наводи низ најчешћих оправдања зашто не треба писати, показујући да ни једно од њих није уствари оправдано (Stromberg, 2004). Код се може објавити као и чланак, постоји чак и специјализовани часопис који се уствари између осталог бави и тим аспектвом статистике, Journal of Statistical Software. Постоји пуно фондова који се додељују за прављење статистичких програма тако да постоји и материјална основа у овом раду. Писање кода можда није претерано занимљива активност, али видети како корисници користе написани програм је за свакако леп осећај, по Сромбергу, за аутора програма. Чак и незнање програмирања, по њему, не би требала да буде препрака за писање програма, јер као прво то је један вид учења, а као друго на крају се ипак само рачуна да ли код ради или не, тј. да ли даје тачне резултате или не.

Алтман и Џекман идентификују деветнаест начина на који може да се гледа на статистички програм, у истоименом раду из 2011 (Altman, Jackman, 2011). Они

наводе шест мотива за писање програма: први је за свакако разумевање проблема. Ови аутори цитирају Доналда Кнутха (чувеног аутора „Уметности програмирања“, више томног издања, које још увек није завршено), али уствари сви који предају знају, да се нешто уствари разуме тек када пробамо да некога другог том знању научимо. У програмирању је онај кога треба да научите уствари рачунар, ученик са нултом толеранцијом на недоречености и непрецизност. Програмирање је уствари провера да ли сте разумели проблем, али и да ли сте научили метод за његово решење. Ово је можда и најважнија мотивација за писца ове дисертације. Научити, тј. проверити своје знање кроз програмирање које даје скоро истовремену повратну информацију, али и јасну стуктуру ваших знања.

Програм који је вама послужио, може послужити и другима, а да би се то и десило Алтман и Џекман наводе пар препорука, као што су коришћење проверених алгоритама, постојање документације, обезбеђивање тестних података за тестрање рада ваше методе, прављење програма тако да ови имају одређену дозу отпорности на грешке корисника, итд. Корисници наводе да су користили ваш програм, тако да и добијање заслуга је такође нешто на шта треба рачунати, као и новчана средства, који такође могу да буду значајни мотиватори за ову врсту активности. Али оно што је такође методолошки јако битно је и то да се ваше истраживање може репродуцирати и проверити, ако сте већ осмислили нову методу, програм који сте направили за њу је нешто попут методолошког документа.

Постојећи статистички системи

Наравно већ је побројавање свих система веома амбициозан подухват, а не њихов приказ. Зато се аутор одлучио на преглед најважнијих по критеријуму њиховог утицаја на остале али и броју корисника. Прва два система SAS и SPSS су представници старе парадигме за статистичке програме, које су усвари скуп статистичких процедура. То су затворени системи који скоро да не нуде могућност порширења од стране корисник. Подршка за прављења нових програма у њима је јако слаба, те је њихова надградња од стране корисника јако тешка и компликована. Ови програми представљају стандард у статистичком свету, SAS за напредне кориснике, а SPSS за почетнике. Друга два наведена програма која ће бити приказана су STATA и R. Они представљају окружења која омогућавају кориснику да развија своје програме и да на тај начин проширује систем. И један и други програм имају и посебан језик у коме је могуће развијати нове програме. R је за разлику од осталих и програм отвореног кода, његов код је доступан јавности, и могуће га је бесплатно користити и мењати.

SPSS

Статистички пакет за друштвене науке би био дословни превод назива овог статистичког програма. Направљен је 1968 од стране Нормана Ниа и Надлаи Хула. По неким ауторима, њихов приручник за коришћење из 1970 је једна од најутицајнијих књига у социологији. SPSS се састоји од многобројних статистичких процедура које су развијене у самом програму. Ове процедуре се позивају преко језика четврте генерације, који се устварi састоји од скупа команди. Свака команда поред свог назива, речи која је позива има и скуп додатних параметара. Комада се завршава са тачком „.“. Типичан пример би била комада за фрекфенце вариајбли

```
FREQUENCIES VARIABLES = p1 p2
           /PERCENTILES = 20 40 .
```

Где су `p1` и `p2` имена варијабли за које се извршава анализа. Комада је `FREQUENCIES` а праметри су `VARIABLES` и `PERCENTILES`, са одговарајућим вредностима. Овако специфицирана комада ће приказати табелу фрекфенци за наведене варијабле као и њихов двадесети и четрдесети перцентил.

Кад писања команди, SPSS подржава још стара правила (правила осмишљена педесетих година прошлог века када је улазни медијум за рачунар била папирна картица са рупама на себи), по којима је у једној линији могуће задати само једну команду, и то тако да команда мора да почиње у првој колони. Команда може да се наставља у више редова, као у примеру, али линије које су наставци не смеју да почињу у првој колони. Команда се завршава са тачком.

Резултати анализе ће бити приказани у датотеци (фајлу) са резултатима тј. у прозору и генерално њихово коршћење у другим процедурама није могуће, сем да се резлтати коришћењем команде `OMS` сниме у фајл са подацима који би се касније учитао, тј. представљао радне податке за неку следећу процедуру. У сваком случају, SPSS није дизајниран да се користи програмски, већ пре кориснички тј. као систем коме се команде задају од стране корисника.

Овакав вид коришћења програма је пожељан начин коришћења, ако се подаци анализирају по први пут, или само једном (што је у стварности веома редак случај), тј. када унапред није позната тачна метода која ће се употребити, него анализирање иде корак по корак у зависноти од добијених резултата. У случају када исте, или боље рећи сличне команде треба задавати више од једном SPSS је понудио могућност дефинисања макроа, који је уствари обичан пре-процесор текста (додуше са одређеним бројем функција за стрингове) који пре извршења задатих команди, замени све наведене макро вредности у командама за предатим вредностима у макро позиву. Ова замена се одвија на нивоу стринга, а не на основу вреднсти, тако да обичне аритметичке операције са макро варијаблама нису могуће, нпр. `!x+1`, где би `!x` била макро променљива, је операција која неће

имати никаву последицу, јер сабирање стринга и целобројне вредности није дефинисано у макро језику SPSS-а.

Основна снага SPSS-а није његов језик, нити могућност програмирања и развијања нових процедура у његовом окружењу већ пре свега рад са подацима. SPSS поседује фајл са подацима али и мета-подацима која омогућавају да се дефинишу сва или скоро сва обележија за дато истраживање. Свака варијабла поред имена поседује и ознаку (енг. label) преко које је могуће јасније дефинисати мерену карактеристику, нпр. специфицирати дословно питање из упитника. Могуће је дефинисати и ознаке за сваки одговор, тј. за сваку вредност на варијабли, нпр. дословно навести одговоре на питање из упитника. Поред системског недостајућег податка, вредности вариајбле која није измерена за датог испитаника, тј. није ни требало да буде измерена (у упитницима је то чест случај када одређена питања не желимо да питамо испитанике који су на одређени начин већ одговорили не то питање кроз одговор на неко од претходних питања), SPSS омогућава и дефинисање вредности које се проглашавају за недостајуће. То су вредности кодова за специјалне одговоре који такође не представљају праве мере, нпр. када је испитаник одбио мерење. У том случају прескочити ово питање би значило да га сврставамо у групу оних који нису ни требали да буду питани, а уписивањем било које вредности, без додатне дефиниције, ризикујемо грешку у рачуну. Поред јасне структуре података, SPSS подржава и јако једноставан скуп команди за трансформацију података, као и ефикасан едитор података. Анализом рада статистичара се може лако установити да се највише времена троши у сређивању података (колоквијлани назив би био чишћењу података), а тек незнатан део у анализирању истих. Због свог ефикасног едитор и трансормационих команди SPSS се наметнуо као главно оруђе за рад са подацима. Од верзије 13, SPSS је изменио и начин на који је програм приступао табели података, пре ове верзије целокупниа табела је била учитавана у радну меморију рачунара, што је наравно умногоме отежавало рад са великим базама података, или га употпуности и онемогућавало. Од верзије 13, ова техника је промењена тако да се SPSS сада ослања на рад са тврдим диском као основним носачем

података.

У овом дизајну организације података дошла је до пуног изражаја и SPSS-ова комада `EXECUTE`. Која уствари означава да наведене трансформације над подацима треба извишити над сваким од слогова, редова у бази. На овај начин се јасно одваја манипулација подацима у оквиру једног слога од извршавања истог скупа манипулација на сваком од редова база, које у зависности од броја редова у бази може да буде и знатно.

SPSS поред командног језика садржи и друго окружење које се назива MATRIX и које је практично назависно од остатка система. Ово окружење је намењено матричном рачуну и у њему је могуће испрограмирати статистичке анлизе нпр. каноничка корелациона анлиза се у SPSS-у налази само у облику макроа који користи MATRIX језик. Комуникација MATRIX окружења и остатка SPSS-а је могућа само преко датотеке са подацима. Други проблем је форматирање исписа, које у стандардном окружењу представља богат текстуални испис са форматирањем табела, док је у MATRIX окружењу нема никакве могућности форматирања, већ само једностаног текстуалног исписа.

SAS

Систем за статитичку анализу (енг. Statistical Analysis System) је назив овог програма направљеног 1966. од стране Антони Бара и сарадника. Слично SPSS-у започет је као један скромни студентски пројекат, који је убзо превазишао ову поставку и постао један од најсвеобухватнијих статистичких сиситема. И та свеобухватност уствари представља његов мач са две оштрице, иако постоје све или скоро све статистичке анализе, програм је велик и гломазан, а његово учење самим тим теже и дуже, те је због тога број SAS корисника знатно мањи од броја корисника SPSS-а.

SAS је, као и SPSS, програм којим се управља преко команди које сутвари представљају језик четврте генерације. SAS се назива систем, а не програм, јер пре има одлике оперативног система него једног издвојеног програма. Скоро све што је кориснику потребно, почев од текст едитора, је испрограмирано и налази се у овом систему. Тешко је донети јасан став према овом пакету, са једне стране плени његово богатство, а са друге стране одбија вас његова неразвијена имплементација у прозорском окружењу какав је MS Windows. Може се и претпоставити зашто је то тако, али ипак остаје нејасно, зашто SAS има тако лошу имплементацију у окружење оперативног система.

Свака команда-програм у SAS-у је састављена од најмање два дела. `DATA` дела у коме се специфицирају, сређују подаци на којима ће се радити и `PROC` дела, који специфицира процедуру тј. процедуре које треба извршити (нпр `PROC NLIN` за естимацију параметара нелинеарног регресионог модела).

SAS је подељен у пакете, и његов основни пакет „Base SAS" уствари представља основу за рад са подацима, као и основне процедуре. Оно што се најчешће у статистичком свету подразмева под SAS-ом је уствари његов пакет SAS/STAT који садржи све напредне статистичке процедуре за обраду података. Поред ових основних пакета SAS поседује и SAS/OR пакет намењен операционим истраживањима, SAS/QC, SAS/ETS за економетријске анализе и временске серије, SAS/IML за матрични рачун, SAS/GRAPH пакет за графичко приказивање података.

IML је интерактивни матрични језик који за разлику од SPSS-овог MATRIXA поседује и корисничке функције, функције које корисник (са основним знањем програмирања) може да напише у самом IML језику. Због тога IML представља много озбиљнији алат у рукама статистичара, него што је то SPSS-ов MATRIX. Предвиђен је за програмирање анализа које још увек нису развијене у SAS-у. Како је то наравно скрипт језик, интерпретатор, то су процедуре написане у њему спорије од већ развијених процедура у самом SAS систему (SAS Institute Inc., 2004).

STATA

STATA је назив статистичког програма (по наводима његових аутора има програма се изговара „Стејта") који је направљен 1985. Назив потиче од спајања две речи, статистика и подаци, енг. STAtistics and daTA. За разлику од SAS-а који је од прве верзије рад са подацима засновао на фајлу, и SPSS-у који је на овај начин рада прешао од верзије 13, STATA је и даље остала на техници учитавања целокупних података у меморију рачунара. Због овог дизајна STATA је увела и нека ограничења на величину података и матрица са којима може да ради, иако су наравно ова ограничења реално велика и даље стоји да се јако велике базе података брже анализирају у SAS-у, тј. SPSS-у.

STATA је дизајнирана као језик. Командна линија преко које се издају инструкције програму је преферирани дизајн овог програма, иако у послењим верзијама STATA поседује и меније и остали Winodows-овски кориснички интерфејс. Па ипак главна дистинкција овог програма у односу на остале, је постојање тзв. `ado` датотека, текстуалних датотека са командама СТАТА језика. Ово су у ствари програми писани у STATA језику који извршавају најразличитије статистичке процедуре. STATA долази са великим бројем већ написаних `ado` фајлова, око 80% свих инструкција, процедура је написано у `ado` фајловима, што самим тим значи и да их је могуће мењати, развијати. Ово је омогућило да корисник овог система може да напише нову процедуру и да је подели са осталима, на једноставан начин који је интегрисан са остатком система. Овај приступ орвореног кода (енг. open source) је можда оно што је учинило да STATA постане веома популарна у академским институцијама.

Да би корисници могли да пишу нове процедуре, које се интегришу у систем, било је потребно јасно дефинисати синтаксу команде, која би се после могла парсовати у `ado` датотеци. Формат STATA команде је:

```
[by varlist:] command [varlist] [=exp] [if exp] [in range]
[weight] [using filename] [,options]
```

- `by` представља дефинисање сегмената података над којима треба извршити

жељену команду, команда се извршава над свим субузорцима дефинисаним комбинацијом свих вредности наведених варијабли.

- `varlist` је листа варијабли над којима се извршава команда
- `= exp` је у ствари израз за стварање нове варијабле
- `if` и `in` су изрази за селекцију контигента, субузорка над којим ће се извршити наведена команда
- `using` се користи да би се навела датотека са подацима над којима ће се извршити команда,
- `options` је листа опција.

Овако дефинисана команда се може парсовати уз помоћ команде `syntax`, тако да корисничке процедуре дефинисане у `ado` датотеци могу имитирати основне, системски кодиране комадне, те у исти мах и примати аргументе.

STATA такође поседује и мацро променљиве. Могуће је дефинисати локалне и глобалне мацро промељиве чије је вредности после могуће заменити нпр. у позивима команди. Као и у осталим наведеним пакетима, макро променљиве су део текстуалног пре-процесора који пре него изврши код програма, замени макро промељиве са њиховим вредностима. Али овде треба бити обазрив јер STATA има два начина дефиниције макро променљивих једном када се ова дефинише дословно као предата вредност, а други пут као резултат евалуације макро променљиве.

STATA је добила велику популарност као систем који је могуће проширивати по властитим потребама са програмима из библиотеке програма који су други корисници писали али чији је квалитет контролисан. STATA компанија уврштава корисничке процедуре које су задовољавајућег квалитета у свој стандардни пекет процедура, док се и остале лако могу скинути преко Интернета.

STATA не поседује кориснички дефинисане функције, већ само програме, процедуре које се понашају као стандардне Стејтине команде. Овде се јавља проблем са вредностима које би процедура требало да врати након позива. Из овог

разлога дефинисане су посебне табеле вредности, у које процедура може да упише произвољe скаларе, матрице. Ова табела је доступна све до позива наредне процедуре, када се њихов садржај брише, тј. спрема за нове вредности које нова процедура може да упише. STATA поседује неколико таквих табела тј. листи али две које се најчешће користе су тзв. р и е листе. Стандардна листа је р-листа у коју се уписују вредности са стандардном `return` командом, док програми чији је резултат процена, естимација резултате враћају у е-листи преко команде `ereturn`. Поред ове две постоји и с-листа у коју корисници не могу да уписују вредности већ само да је изчитавају, то је листа системских вредности као што су датум, време, итд. Ово је начин на који STATA разврстава све команде у две групе: р-тип и е-тип у зависноти да ли служи за процену параментара или само за трансформацију података. Нпр. регресија је е-тип команде и враћа коефицијнте регресије у е-листи. На овај начин се кроз изчитавање листи може доћи до резултата неке команде, тај резлтат уписати у неку променљиву и користити га за даље прорачуне. Наравно могуће је на овај начин извући само вредности које је програмер команде ставио у листу, користећи команде `return` тј. `ereturn`.

Одлика која је такође интересантно код STATA-е је да поседује имена колона и редова матрица у самој дефиницији матрице. На овај начин, по ауторима овог пакета, је олакшано исписивање матрице кориснику, али и провера да ли је матрични рачун задат рачунару како треба, бар на основном нивоу. Ако називи редова или колона нису они који су очекивали, лако долазимо до закључка о постојању грешке у програму (STATACorp., 2009, Rabe-Hesketh, 2004).

R

Овај пакет је настао као верзија отвореног кода S језика, који је вероватно добио име од „статистика“, а који је развијен 1975. у AT&T Bell Labs, истој лабораторији где је направљен и C језик. Највећи допринос развоју S језика дао је Џон

Чејмберс. 1993 године на Унуиверзитету Оукланд, на Новом Зеланду, Рос Ихака и Роберт Џентлмен су започели верзију отвореног кода (бесплатну верзију програма која нуди целокупан код апликације на увид јавности) и назвали га „R“. Групи људи који су наставили да унапредјују R убзо се прикључује и оснивач S језика као и многи други. У овом тренутку овај језик се сматра јединицом мере у стастистичком свету.

R спада у групу функцијских језика, који би се у најкраћем могли описати као супротни процедуралним. У овој класи језика акценат је на резултату до кога се долази преко функција, за разлику од процедуралних језика код којих је акценат на инструкцији, тј. промени вредности коју симболичка променљива носи. У чисто функцијским језицима, промељивој када се једном додели вредност, она се не може мењати, док је наравно у процедуралним језицима операција доделе једна од кључних контруката.

По својој синтаски R је сличан популарној класи језика која се заснива на синтакси C језика, док је његова семантика ближа функцијским језицима какав је LISP или APL. Акценат код функцијских језика је генерално на модуларности, тј. прављењу компактних делова кода које је после могуће даље користити и комбиновати са другим модулима и тако „зидати“ даље жељени програм. R такође поседује ову особину, тако да је у оквиру овог језика могуће направити нове функције, обједините их у пакете сличних функција које је после могуће лако делити са осталим корисницима овог језика.

Као функцијски језик, R поседује могућност „рачунања“ са основним елементима језика, а то су функције. Функције такође могу бити аргумени других функција, које ове могу користити па чак и мењати. У основи овог језика је мали број кључних речи и десетак стандардних оператора, али наравно R поседује велики број различитих функција.

Постоји пар основних структура које овај језик користи, вектор тј. матрица, листа и оквир података (енг. data frame). Вектор је стандардно низ елмената истог типа. Листа је скуп именованих објеката, елемената, који могу бити било који од

објеката у језику, и којима се приступа не преко позиције, као што је то случај код вектора, већ преко њиховог имена. Оквир података уствари листа вектора који имају исту дужину, што стандарадно и јесте табела са подацима, скуп вектора исте дужине, било нумеричких било текстуланих који имају јединствено име преко којег им се приступа. Елементи вектора могу бити нумерички, текстуални или логички. Поред ових основних типова података, R поседује и типове података који му омогућавају рачунање над самим језиком. Тако постоји објекат исказа, тј. саме интерне репрезентације рачунског исказа у језику, функција је уствари листа исказа. Поред овога језички објекти су и функцијски позиви и промељиве. Функцисјки позив је такође могуће посматрати као листу и користити њене елементе у наредним рачунима.

R такође у себи саржи имплементацију „лење евалуације“ у којој се вредност израза не израчунава док не постане потребна у неком другом изразу. На овај начин се чува процесорско време, јер се користи само за рачунање онога што је потребно. На сличан начин је имплементирана и операција доделе (која је у R дефинисана преко оператора „<-“ за разлику од већине осталих језика где је то „=“), тако да рецимо у следећем коду

```
a <- c(2, 7, 3)
b <- a
```

симболичкој промељивој `a` додељујемо вредност вектора са три елемента два, седам и три (функција `c()` уствари спаја елементе у низ), док у следећој интрукцији ту исту вредност додељујемо промељивој `b`. У стандардним језицима ово је уствари операција доделе која копира садржај меморијског простра који је додељен промељивој `a` у меморијски просто промељиве `b`. На овај начин смо заузели два пута више меморије, док уствари имамо исте вредности на оба места. R је имплементирао заузимање новог простора тек ако се један од вектора мења. Тако да ће у ово примеру промељива `b` добити свој простор тек након инструкције

```
b[2] <- 10
```

Све су ово одлике које су овом језику дале велике могућности. Али по мишљењу аутора дисертације то је као и увек мач са две оштрице, велике могућности воде ка великој сложености, што опет чини језик тешким за учење. R иако је бесплатан и иако има велику заједницу корисника, ипак одаје утисак да је за његово коришћење потребно напредно знање програмирања. Поред своје комплекосности ту је и пар ствари које отежавају трансфер већ стеченог знања из програмирања у другим језицима. Наравно као прво његова фунционална орјентација која је увек била далеко од доминанте у свету програмских језика, и на коју процедурално орјентисани корисници треба да се навикну. Друго је наравно постојање великог броја функција, које читање кода чине компликованим за почетника, који још увек није упознат са њиховом функцијом. У R ако функција враћа више вредности онда она враћа обејакт типа листе, дакле скуп вредности које се налазе у једном објекту и којима се приступа преко њиховог имена. Наравно ово име је одређено у позиваној функцији и не постоји једноставан начин да се закључи које све вредности у себи носи листа коју функција враћа, тако да се почетник налази у проблему јер још увек не познаје функције, а из имена листе не може да закључи шта су све елементи које она носи.

R користи синтаксу која је по много чему специфична, тако да је и у овом случају трансфер са других језика у најбољем случају минималан, ако не и негативан. Тако рецимо, док је у већини језика, који заиста припадају C фамилији језика, оператор доделе представљен са „=“ у R-у је то „<-“; референцирање елемента листе („хеш“ мапе, структуре) се обично изводи коришћењем оператора престављеног са једним карактером и то тачком, у R-у се за ту операцију користи карактер $. Али зато карактер тачке у R-у нема неко специјално значење тако да се може корисити и у оквиру имена променљиве као било који други валидни карактер.

Циљ рада

Статистика је за сигурно најважнији алат у рукама психолога. Кроз историју развој ове дисциплине је повлачио за собом и развој психолошке науке. Ова веза је толико јака да су психолози дали и свој допринос развоју саме статистике, кроз поступке који су пре свега намењени психолошким подацима и проблемима. Развијеност статистичких модела, али наравно и упознатост психолога са њима, условљава и избор модела који користимо за моделовање психолошких појава. Циљ овог рада је постављање основе за статистички систем кроз предлог новог једноставног програмског језика, који оперише са матрицама, а који ће бити у основи овог система. Систем ће омогућити брз развој, проверу али и једноставну употребу нових статистичких процедура. Наравано не заборавибши ни едукативни карактер које може да има програмирање, систем ће у исто време потенцијално помоћи и у разумевању и усвајању већ постојећих статистичких поступака.

Предлог новог језика

Програмски језик је представљен са синтаксом коју користи, типовима вредности које је могуће користити, функцијама које су саставни део језика, те операторима и релацијама које се изводе над овим вредностима. Прдложени језик има синтаксу засновану на синакси C језика, тј. синтаксу која је слична са већином језика који се тренутно највише у употреби. То је процедурални језик, са само три структуре података: матрица, табела података и листа.

Типови података

Стандардни тип података је нумерички податак, који је интерно увек представљен као double вредност, тј. вредност уписана у 8 бајтова меморије рачунара. Разломљени бројеви се дефинишу са децималном тачком. Пример:

```
1
2.5
-3.755
```

double је стандардан начин приказивања нецелобројних вредности у рачунару, који у себи садржи све непрецизности бројних записа, у овом случају бинарног. Тако да следећи израз у рачунарском свету није тачан,

```
0.4 - 0.3 == 0.1
```

тј. разлика 0,4 и 0,3 није једнака 0,1, овај проблем је добро познат и настаје због заокруживања вредности, тј. немогућности да се децималне вредности прикажу без грешке у заокруживању.

Поред нумеричких података могуће је дефинисати и текстуални податак (стринг). Текстуални податак (низ карактера) се дефинише са ", као у следећем примеру:

```
"Ovo je niz karaktera"
```

Ако у оквиру текста треба навести двоструке наводнике, онда се пре истих мора ставити карактер \:

```
"Petar je rekao \"Ne\""
```

карактер „\” је специјални карактер који означава да следећи карактер такође има специјално значење, тако да: `\"` означава карактер наводника, `\n` представља карактер за нови ред, `\t` је таб карактер.

Матрице

У предложеном језику све вредности се посматрају као матрице. Скаларна вредност се посматра као матрица димензија један са један.

Уколико желимо да дефинишемо матрицу састављену од одређених вредности треба да користимо матрични оператор `[]`, пример:

```
[1, 2; 3 4]
```

уствари представља матрицу

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Зарез, или бланко карактер се користе за одвајање вредности у оквиру једног реда матрице, док се тачка-зарез, тј. нови ред користе за одвање једног од другог реда.

```
[1,  2
 3   4]
```

је у ствари иста дефиниција претходне матрице само можда у неким ситацијама

читљивија.

Елементи матрице су или сви састављени од нумеричких вредности или сви од текстуалних вредности, није могуће мешати ова два типа података у једној матрици.

Мапа

Ово је структура у којој се елементи не реферецирају преко позиције, већ преко кључа, (видети нпр у Knuth, 1973). У вектору имамо први, други, трећи итд. елемент, и њих реферирамо преко њихове позиције. Елементима нумеричког вектора, тј. вектора чији су елементи бројчане вредности, приступамо коришћењем оператора `[]` навдећи редни број елемента,

```
a = [10, 20, 30, 40]
a[3]
```

тако ће испсис овог програма бити 30, јер је то трећи елемент векора `a`. У мапама елементи се не чувају на основу њихове позиције, већ на основу њиховог кључа одоносно имена, који је у предложеном језику увек стринг. Тако да је мапа структура која може да садржи више матрица и или других мапа, кључ преко кога се долази до тих елемената је уствари име саме матрице тј. мапе која је елемент. Тако да је могуће формирати мапу, мапу `a` у примеру, која ће садржати нпр. две матрице, матрице `p` и `q`:

```
a.p = [10, 20, 30, 40]
a.q = 42
```

У статистичком програмском језику R ова се структура назива листама.

Дефиниција матрице, вектора преко операције распона

Вектор је могуће дефинисати као низ бројева са константним инкрементом. Овакав низ се дефинише са операцијом `:`.

Ово је бинарна или тернарна операција. На пример 2:6 представља редни вектор чији су елементи 2, 3, 4, 5 и 6 тим редом.

$$2{:}6 = \begin{bmatrix} 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

Подразумевани инкремент је један, али је могуће променити га наводећи и трећи операнд, 2:6:2, чиме добијамо тернарну операцију, која у овом примеру за резултат даје вектор чији су елементи 2,4 и 6:

$$2{:}6{:}2 = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$$

Операција је дефинисана тако да се нова вредност у низу добија тако што се претходна увећава за инкремент, или се узима права, полазна вредност у првом кораку, све док новонастала вредност није већа од горње границе. Тако на пример 2:6:3, даје само два елемента: 2 и 5. Следећа вредност је 8, али је она већа од горње границе и зато неће бити додата као следећи елемент вектора. Уколико је инкрементална вредност негативна, тада ће се вредности елемената у новонасталом вектору смањивати до онога који је и даље већи од сада доње границе:

$$7{:}2{:}{-2} = \begin{bmatrix} 7 & 5 & 3 \end{bmatrix}$$

Резултат ове операције може бити и празна матрица, нпр 10:2 даје као резултат празну матрицу јер је инкремент који није наведен једнак јединици, а како је почетна вредност већа од крајње, оператор `:` неће направити ни један елемент вектора.

Предефинисане и специјалне вредности

Дефинисане су следеће константе које се често користе у математичким изразима:

ознака	вредност
e	2.718
pi	3.142
true	1
false	0

Дефинисане су још две специјалне вредности, `NaN` и `Inf` . Прва вредност је скраћеница од енглеског „Not a number“ и представља вредност која није рационални број, као што је корен, или логаритам од негативне вредности. По правилу било која операција са овом вредношћу има исту вредност као резултат. Дакле `NaN + 1` је такође `NaN`.

Друга вредност, `Inf` од латинског Infinitum, је ознака за бесконачну вредност, као што је то вредност операције 1/0, наравно `-Inf` је ознака за минус бесконачно.

Још једна специјална ознака `[]` се односи на матрицу која нема ни један елемент, празну матрицу, која има сличан смисао као и празан скуп. Њене димензије су нула са нула, и стандардне операције са њом нису дефинисане.

Променљиве - Операција доделе

Основна операција у сваком процедуралном језику јесте операција „доделе“, давање вредности симболичкој ознаци.

```
a = 1
```

Ова инструкција означава уствари да смо симболу `a` придодали вредност један. Док му не도делимо неку другу вредност, свеједно је да ли стављаму ознаку `1` или `a` . У овом случају је симоличкој промењивој `a` дата једна скаларна вредност.

Предложени језик је процедуралног типа, дакле променљиве могу мењати вредности, тако да нова променљива може имати неку другу вредност касније у програму. Наравно променљивој се може придодати и вредност операције на исти начин:

```
a = 1 + 2
```

Оператор груписања

Овај оператор дефинисан са `[]` и може користити за конкатенацију већ дефинисаних матрица, као и скаларних вредности. Рецимо да имамо матрицу `A` и матрицу `B` које имају подједнак број редова, као:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

у том случају операција `[A,B]` спаја ове две матрице у једну која има четири колоне:

$$[\mathbf{A},\mathbf{B}] = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix}$$

док операција `[A; B]` спаја ове матрице тако да новонастала матрица има четири реда:

$$[\mathbf{A};\mathbf{B}] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Наравно уз помоћ овог оператора се од скалара формира матрица односно вектор:

```
A = [1, 2, 3, 4]
```

У овом примеру смо дефинисали редни вектор са целобројним бројевима од један до четири.

Елемент матрице, субматрица

Нека је дата дефинисана матрица `A`, на следећи начин:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 0 & 7 \\ 3 & 5 & 1 & 8 \\ 3 & 2 & 9 & 4 \end{bmatrix}$$

Референца на елемент који се налази у трећем реду и другој колони се добија, преко операције референцирања, која се такође обележава са `[]`, као:

```
A[3, 2]
```

Ако желимо да референцирамо цео ред матрице, онда можемо да користимо карактер `:` као други аргумент, или да се други елемент оставимо празан. Тако да ако желимо да референцирамо цео трећи ред матрице онда би смо написали `A[3, :]`, или `A[3, ]`, што је наравно:

```
A[3, :] = [3  2  9  4]
```

Референца на колону матрице се постиже на сличан начин стим што сада дво-тачку треба ставити на место првог аргумента, или први аругмент изотавити, употреба двотачке је опциона, али код чини нешто читљивијим:

$$\mathbf{A}[\ ,2] = \begin{bmatrix} 3 \\ 5 \\ 2 \end{bmatrix}$$

Двотачка је ознака за операцију распона, која је бинарна тј. терцијална операција, у овом контексту референцирања елемената матрице може се користит без иједног елемента, што је еквиаленто као да смо написали од првог до послењег реда матрице, у наведеном примеру то би било `1:3`. Наравно могуће је користити и трећи операнд у операцији : тако да добијемо рецимо елементе из сваког другог реда

$$\mathbf{A}(1:3:2,2) = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

оваква анотације подразумева да знамо димензије матрице, тј. у овом примеру број редова. У колико желимо да кажемо сваки други елемент од првог до последњег елемента, онда се уместо послењег може навести једна од специјалних вредност `NaN` или `Inf`:

$$\mathbf{A}(1:\mathrm{NaN}:2,2) = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Када референцирамо елемент вектора, довољно је да користимо један индекс, свеједно да ли се ради о редном или колонском вектору. Наравно и употреба два индекса од којих је један увек једнак један неће бити погрешна.

У предложеном језику матрице се у меморији чувају као низ бројева, слепљених колона једна на другу. Зато је дефинисана и операција референцирања елемента матрице само са једним индексом иако је реч о матрици а не о вектору.

```
A[5]
```

Овај пример, где је `A` матрица дефинисана у претходном примерима са три реда и четири колоне, је уствари други елемент из друге колоне, јер имамо три реда у матрици `A` тако да референца на пети елемет припада другој колони.

Промена вредности елемента

Елементу/елементима матрице се може променити вредност, операцијом доделе:

$$\mathbf{A}([1,3]\ ,\ [2,4]) = \begin{bmatrix} 11 & 12 \\ 21 & \text{NaN} \end{bmatrix}$$

ово уствари значи да елементима у првом и трећем реду из друге и четврте колоне треба променити вредности са вредностима матрице са десне страни једнакости, после ове операције матрица `A` ће имати следеће вредности:

$$\begin{bmatrix} 1 & 3 & 0 & 7 \\ 3 & 11 & 1 & 12 \\ 3 & 21 & 9 & \text{NaN} \end{bmatrix}$$

Пошто су одређена два реда и две колоне тј. треба само променити вредност четири елемента, потребна су четири вредности на десној страни, тако да је дефинисана матрица 2 x 2. Али ове четири вредности могуће је дефинисати и као вектор, опет би био добијен исти резултат, вредности би биле попуњаване колона по колону:

$$\mathbf{A}([1,3]\ ,\ [2,4]) = \begin{bmatrix} 11 & 12 & 21 & \text{NaN} \end{bmatrix}$$

Наравно уколико желиммо да сва четири елемента добију једну исту вредност онда је довољно да на десној страни операције доделе ставимо скалар са жељеном вредношћу, тако

$$\mathbf{A}([1,3]\ ,\ [2,4]) = 100$$

даје уствари матрицу:

$$\begin{bmatrix} 1 & 3 & 0 & 7 \\ 3 & 100 & 1 & 100 \\ 3 & 100 & 9 & 100 \end{bmatrix}$$

Брисање редова, колона матрице

Уколико се реду тј. колони матрице додељује специјална празна матрица `[]` то уствари означава да желимо да обришемо дати ред тј. колону, тако да ако је матрица `A` дефинисана са:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 0 & 7 \\ 3 & 5 & 1 & 8 \\ 3 & 2 & 9 & 4 \end{bmatrix}$$

те потом применимо следећу операцију

```
A[:, 2] = []
```

која уствари представља брисање друге колоне из матрице `A` , онда ће матрица `A` изгледати овако:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 7 \\ 3 & 1 & 8 \\ 3 & 9 & 4 \end{bmatrix}$$

Није дефинисано брисање и редова и колона матрице у исто време, тј. један од аргумената матрице мора бити операција `:` без операнада, односно може бити изостављен.

Транспон матрице

Матрица се транспонује преко оператора `'`. Где ако је матрица `A` дефинисана са

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 7 \\ 3 & 1 & 8 \\ 3 & 9 & 4 \end{bmatrix}$$

њен траспон ће бити

$$\mathbf{A'} = \begin{bmatrix} 1 & 3 & 3 \\ 0 & 1 & 9 \\ 7 & 8 & 4 \end{bmatrix}$$

Аритметичке операције

Све аритметичке операције над скаларима имају стандарде математичке дефиниције, ако су оба операнда у датој операцији рационала, али ако је један од њих специјална вредност `NaN` или `Inf` тада је резултат те операције управо ова вредност, тј.:

```
1 + NaN = NaN

2 * Inf = Inf
```

Сабирање

Једна од стандардних операција је операција сабирања, бинарна операција која је у класичној матричној алгебри дефинисана на домену матрица истих димензија. У предложеном језику, дозвољене су још неке често коришћене ситуације, сабирања матрице и скалара, али и матрице и вектора.

Стандардно матрице морају бити истих димензија да би операција сабирања била могућа

```
A + B = C
```

где је резултујућа матрица `C` уствари збир одговарајућих елемената матрица `A` и `B`

$$c_{ij} = a_{ij} + b_{ij}$$

као у примеру:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 2 \\ 3 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

У предложеном језику дефинисана је и операција сабирања између матрица и вектора, ако вектор има бар једну исту димензију као и матрица, пример:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 20 & 23 & 24 \end{bmatrix}$$

тј. колонски вектор је сабран са сваком колоном матрице, и вектор и матрица имају исти број редова, или

$$\begin{bmatrix} 10 & 20 & 30 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 10 & 23 & 34 \end{bmatrix}$$

где је редни вектор додат на сваки ред матрице.

У предложеном језику дефинисано је и сабирање матрице и скалара, тако што се операција сабирања изврши над сваким елементом матрице са датим скаларом, пример

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} + 10 = \begin{bmatrix} 11 & 12 & 13 \\ 10 & 13 & 14 \end{bmatrix}$$

Оно што треба приметити јесте да ова операција није дефинисана између два вектора која нису истих димензија, тј. није могуће сабрати редни и колонски вектор. Ово није омогућено јер се ово може двоструко протумачити, да ли се као резултат жели добити вектор, сваки елемент једног вектора се сабере са одговарајућим елементом другог вектора, или матрица, где је сваки елемент левог вектора сабран са сваким елементом десног вектора.

Одузимање

Операција одузимања је дифинисана на сличан начин као и сабирање, уствари одузимање се може посматррати као сабирање где је други оперант претходно помножен са `-1`, тј. где је претходо на другом операнду извршена унарна операција `-`.

Инкрементирање

Ово је унарна операција где се сваки елемент нумеричке матрице повећава за вредност један. Овај се оператор означава са `++`. Пише се у постфих формату, дакле иза израза који се хоће повећати, нпр

```
a++ - 10
```

те ако би у промељивој `a` била вредност 3, после извршења овог израза у променљивој би била вредност 4. Али за разлику од C фамилије језика, овај оператор се извршава пре прослеђивања вредности операцији одузимања, наиме стандардно постоје два оператора инкрементирања, један је у префих а други у постфих нотацији. `++a`, би означавало да вредност променљиве `a` треба прво инкрементирати а потом проследити даље, док би `a++` означавао обрнут редослед, прво се проследи тренутна вредност, а потом инкрементира промељива. Овај други начин уствари означава да се прослеђује копија вредсности смештене у промелљиву, што у случају матрица може бити јако меморијски захтевно. Због тог разлога у предложеном језику постоји само инкрементирање па затим прослеђивање вредности, додуше, означава се у постфих нотацији.

Декрементирање

Инверзна операција опсианом инкрементирању, дакле сваки елемент нумеричке промељиве се смањује за један. Обележава се такође у постфих формату, са `--`.

Множење

Множење матрица је у предложеном језику дефинисано на стандардни алгебарски начин. Сваки ред левог операнта се скаларно множи са сваком колоном десног, и сума тих производа представља елемент у резултујућој матрици. Јасно је да множење матрица није дефинисано за било које две матрице, већ само за оне код којих је број колона леве матрице једнак броју редова десне. Пример:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}*\begin{bmatrix}1 & 0 & 1\\2 & 1 & 0\\1 & 1 & 0\end{bmatrix}=\begin{bmatrix}8 & 5 & 1\\20 & 11 & 4\end{bmatrix}$$

тако је први елемент првог реда у резултујућој матици, настао када смо скаларно помножили први ред леве матрице са првом колоном друге

```
1*1 + 2*2 + 3*1 = 8
```

Елементарно множење

У предложеном језику је додата и операција елементарног множења, по аналогији са сабирањем, где се сваки елемент матрице множи са одговарајућим елементом друге матрице

```
A .* B = C
```

где је операција елементарног ножења дефинисана са `.*` а где је резултујућа матрица `C` уствари производ одговарајућих елемената матрица `A` и `B`

$$c_{ij} = a_{ij} * b_{ij}$$

као у примеру:

$$\begin{bmatrix}1 & 2 & 3\\0 & 3 & 4\end{bmatrix}.*\begin{bmatrix}1 & 1 & 2\\3 & 2 & 2\end{bmatrix}=\begin{bmatrix}2 & 2 & 6\\0 & 6 & 8\end{bmatrix}$$

Наравно као и код сабирања и одузимања могуће је елементарно помножити матрицу и вектор, ако су одговарајућих димензија, тако што ће се сваки ред

матрице елементарно помножити са ред вектором, тј. ако је вектор колона, онда ће се свака колона матрице елементарно помножити са овим вектором, пример:

$$[10 \quad 20 \quad 30] .* \begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 40 & 90 \\ 0 & 60 & 120 \end{bmatrix}$$

Јасно могуће је помножити и матрицу и скалар, наравно ова операција даје исти резултат све једно да ли се примењује елементарно или стандардно множење.

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} * 10 = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 3 & 4 \end{bmatrix} .* 10 = \begin{bmatrix} 10 & 20 & 30 \\ 0 & 30 & 40 \end{bmatrix}$$

Дељење

У матричној алгебри дељење у којем је делилац матрица није дефинисано, али је дефинисан инверз матрице који има слично значење. У предложеном језику је дељење дефинисано само ако је делилац скалар, тако да се резултујућа матрица добија тако што се сваки њен елеменат подели са делиоцем, пример:

$$\begin{bmatrix} 6 & 2 & 3 \\ 0 & 4 & 4 \end{bmatrix} / 2 = \begin{bmatrix} 3 & 1 & 1.5 \\ 0 & 2 & 2 \end{bmatrix}$$

Елементарно дељење

Слично елеменатарном множењу, дељење је дефинисано:

- за матрице истих димензија где се сваки елемент бројиоца подели одговарајућим елементом имениоца
- за матрицу и редни вектор ако је број колона у оба операнда исти, сваки елемент сваког реда матрице се подели са сваким елементом вектора, или обрнуто у зависности шта је бројилац а шта именилац
- за матрицу и колонски вектор ако је број редова у оба операнда исти, сваки елемент сваке колоне матрице се подели са сваким елементом вектора, или обрнуто у зависности шта је бројилац а шта именилац

- наравно елементарно дељење је као и обично дељење дефинисано и између матрице, вектора и скалара, на уобичајан начин, над сваким елементом матрице и скаларо се извши ова операција

Експоненцирање

Ехпоненцирање је дефинисано преко оператора `^`. Тако да израз `a^b` уствари значи да вредност `a` дижемо на `b`-ти степен тј. a^b. Ова операција, је за сада, дефинисана само када је основа скалар, тј. није дозвољено степеновање матрице, сем у случају када се матрица диже на -1 степен тј. када се тражи њен инверз. Пример:

$$2\hat{}\begin{bmatrix}1 & 0 & 2\\3 & 4 & 5\end{bmatrix} = \begin{bmatrix}2 & 1 & 4\\8 & 16 & 32\end{bmatrix}$$

Дефинисана је и унарна операција која је уствари дизање на -1 степен тј. инверзна вредност.

```
2^ = 0.5
```

Ова операција је дефинисана, као што је речено, и за матрице (квадратне, несингуларне) и у том случају представља инверз матрице:

$$\begin{bmatrix}1 & 0\\1 & 2\end{bmatrix}\hat{} = \begin{bmatrix}1 & 0\\-0.5 & 0.5\end{bmatrix}$$

Елементарно експоненцирање

У случају да желимо да сваки елемент матрице подигнемо на одређени степен, треба применити елементарно степеновање које је дефинисано са `.^`, при томе наравно важе правила о примени оператора између матрица и вектора код којих је бар једна димензија иста, тј. матрица истих димензија:

$$\begin{bmatrix}1 & 0 & 2\\3 & 4 & 5\end{bmatrix} .\hat{}\ \begin{bmatrix}2 & 1 & 3\end{bmatrix} = \begin{bmatrix}1 & 0 & 8\\9 & 4 & 125\end{bmatrix}$$

Логичке операције

У предложеном језику дефинисане су три основне логичке операције, „и“, „или“ и унарна операција негације.

Све ненулте вредности се посматрају као логичко „тачно“, док је нула исто што и логичко „нетачно“. Специјалана вредност `NaN` се третира као недостајући податак, у терминологији која се коисти приликом обраде података добијених кроз анкетрирање, ово би био системски недостајући податак, тако да резултат логичке операције зависи од другог операнда. А ако и други операнд није дефинисан ода је резултат целе операције `NaN`, тј. недостајући податак. Специјалне вредности `Inf` тј., `-Inf` посматрају се као „тачно“.

Логичке операције „и“ и „или“

Логичко „и“ је представљено са „`&`”, логичко „или“ са „`|`”. Како су ове операције сличне са аритметичким операцијама сабирања и множења, следе и сличне дефиниције домена ових операција. И једна и друга операција су дефинисане између матрица истих димензија или између матрице и вектора ако им је бар једна димензија једнака, и између матрице и скалара.

Резултат логичких операција је матрица чији су елементи састављени од нула и јединица, пример логичке операција између два скалара:

```
3 & 2 = 1
```

Ово је зато што је свака не нулта вредност третирана као вредност „тачно“ а свака нулта вредност као „нетачно“, па је зато дати пример уствари логичка операција између два „тачно“ чији је резултат такође „тачно“ што је једнако један. Наравно важе и правила о примени операција између матрица и вектора код којих је бар једна димензија једнака, пример:

$$\begin{bmatrix} 0 & 2 & 3 \\ 4 & 0 & 6 \end{bmatrix} | \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Логичка операције негације

Негација је представљена ознаком „`!`“. То је унарна операција, чије је резултат нула, тј. нетачно, и један тј. тачно. Било која не нулта вредност се евалуира у логичко „тачно“, док је нула уствари „нетачно“. Пример:

```
!2 = 0
```

Логичке операције и NaN вредност

Специјална `NaN` вредност се користи за кодирање недостајућих података у матрици података. Тако да све аритметичке операције које укључују `NaN` имају као резлтат исту ту вредност. Са логичким операцијама је мало другачије, рецимо да имамо следећи пример

```
NaN | 1 = 1
```

операција логичког или између вредности један и недостајуће вредности је тачно тј. један, јер је за операцију „или“ довољно да један од операнада буде тачан да би резултат био тачан. У случају

```
NaN | 0 = NaN
```

да знамо да је један операнд једнак нетачном, а други је непознат, резултат ће остати непознат.

Релације

Релације поретка и једнакости

Дефинисане су стандарде аритметичке релације:

- веће „>“

- мање „>“
- веће или једнако „>=“,
- мање или једнако „<=“
- једнако „==“, са два знака једнако
- и неједнако „!=“

Резултат сваке релације је логичка вредност „тачно“ која је представљена са вредношћу један тј. логичко „нетачно“ што је представљено са вредношћу нула. За све наведене релације важе правила дефинисаности када за су обе димензије матрица једнаке, тј. између вектора и матрице када је бар једна димензија једнака, тј. између матрице (вектора) и скалара, пример:

$$\begin{bmatrix} 0 & 2 & 3 \\ 4 & 0 & 6 \end{bmatrix} > \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Релације са специјалном вредношћу `NaN` увек имају за резултат вредност `NaN`, па и у случају једнакости. Да бисмо тестирали да ли је нека вредност једнака `NaN` морамо користити специјалну функцију `isnan()`.

Релације над скуповима

Дефинисана је и основна скуповна операција подскупа, за сада само на редовима матрице, која је представљена са оператором `in`. Ова релација проверава да ли су сви елементи наведени у реду левог операнда налазе у реду десног операнда, и као што је то у скуповима, вишеструка појава исте вредности у реду се посматра као једна. Пример:

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 4 & 3 \end{bmatrix} \text{ in } \begin{bmatrix} 1 & 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

за сваки ред леве матрице у операцији је проверен да ли се наведени елементи у њему налазе у реду десне матрице, како лева матрица има три реда, а десна само

један, резултујућа матица је димензија 3 x 1. Када би десна матрица имала два реда, тада би резултујућа матица била димензија 3 x 2, свака колона резултујуће матице би представљала инклузију редова леве матрице у одговарајућем реду десне матрице, пример:

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 4 & 3 \end{bmatrix} \texttt{ in } \begin{bmatrix} 1 & 3 & 2 \\ 1 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Наравно из овога следи и то да ако лева матрица има само један ред, а десна више од једног, резултат је редни вектор, који има колона колико и десна матрица у релацији редова.

Поред `in`, постоји и операције `anyin`, која уствари проверавају не да ли се сви елементи редова леве матрице налазе у реду десне матрице, већ да ли се бар један елемент из реда леве матрице налази у реду десне, тј. дали је пресек два реда не празан скуп. Пример:

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 4 & 3 \end{bmatrix} \texttt{ anyin } \begin{bmatrix} 5 & 6 & 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Функције

Функције представљају основне градивне елементе процедура. То су по правилу логичке целине које процедуру деле на више мањих подпроцедура, те самим тим олакшавају процес програмирања. Функција представља процедуру која има улазне аргументе, који се предају функцији те која може имати и излазне вредности, једну или више њих.

Функције се деле на основу тога да ли су дефинисане унутар језика (системске)

или их је дефинисао сам корисник језика (корисничке). Друга подела је на основу броја аргумената које захтевају (улазних вредности) и броја аргумената које враћају (излазне, резултујуће вредности).

Функције као и променљиве морају имати валидно име, које почиње словним карактером и у себи не садржи карактер размака као ни специјалне карактере који су резервисани за операције, релације, итд. После имена функције у заградама су наведени аргументи који се прослеђују функцији. Ако функција враћа један резултат онда је њега могуће доделити некој променљивој. Пример је системска функција за прављење матрице идетитета:

```
y = eye(10)
```

Ова линија ће позвати функцију `eye` и предати јој вредност десет. Функција `eye` служи за конструкцију матрице индентитета и она ће резултат предати операцији доделе која ће ту вредност, матрицу индетитета реда десет, уписати у променљиву `y`.

Ако функција враћа више од једне вредности као што је то случај нпр. код функције `eig` која рачуна својствене вредности али и својствене векторе неке квадратне матрице, која је у овом примеру означена са `R`. Променљиве у које желимо да упишемо резултујуће матрице ове функције треба навести на левој страни оператора доделе у угластим заградама. Пример: да би смо доделили обе матрице (својствене вредности и векторе) променљивим `L` и `X`, треба написати следећу команду:

```
[L, X] = eig(R)
```

после ове комаде у матрици `L` ће бити дијагонална матрица са својственим вредностима, а у матрици `X` одговарајући својствени вектори.

Све функције враћају барем једну вредност, а то је вредност грешке. Ако се у току извршавања функције наиђе на неку грешку, нпр. тражимо инверз од сингуларне матрице, извршавање функције, а самим тим и програма ће бити прекинуто, сем

ако функцију инверза нисмо позвали тако да се тражимо једна више излазну вредност од стандардног броја вредности које функција враћа, у овом примеру је то један. Нпр. рецимо да имамо следеће линије кода

```
A = [1 2 4
     1 2 4
     4 3 5]
B = inv(A)
B
```

овај позив ће резултирати грешком јер је матрица `A` сингуларна, те ћемо добити следећи испис, и ток програма ће бити прекинут:

```
ERROR : Regular matrix is needed on the line 'B = inv(A)'
```

Ако програм модификујемо тако да од функције `inv` тражимо још једну вредност, коју у наведеном примеру уписујемо у променљиву `err`:

```
A = [1 2 4
     1 2 4
     4 3 5]
[B, err] = inv(A)
err
```

програм ће наставити да функционише, јер ће вредност 85, што је код грешке када је матрица сингуларна, бити смештена у променљиву `err`, док ће променљива `B` остати не промењена, у овом примеру недефинисана. Ако у функцији нема грешке, вредност која ће бити смештена у променљиву `err` је нула.

Корисничке функције

Могућност креирања нових функција и процедура је једна од најважнијих одлика предложеног језика, тј. могућност да корисник напише функцију коју је касније

могуће позивати као и сваку другу системску функцију. На овај начин предложен језик је могуће проширити и прилагодити својим потребама. У оквиру функције код који се извршава се пише коришћењем команди, оператора и функција предложеног језика. Функција се дефинише:

- са кључном речи `function` иза које следе у
- угластим заградама наведене излазне вредности функције, ако их функција има, затим
- знака једнакости,
- валидног имена функције и у
- обичним заградама називи аргумената функције, ако функција захтева улазне аргументе.

Пример:

```
function [y1, y2] = moja_funk(x1, x2, x3)
```

Ово би био пример дефиницје функције која носи име `moja_funk` и која има три аргумента `x1`, `x2` и `x3`, а која враћа две вредности `y1` и `y2`. Име функције мора пратити стандардна правила за именовање променљивих. Име мора почињати словом или карактером „_“ и у себи не сме садржати карактере који су резервисани као обележија операција, нпр. карактере *, +, -.

Функцију је могуће сачувати у датотеци која има име као и функција а који ће имати екстензију „s“. У том случају ако се датотека налази у путањи предложеног језика, по позиву дате функције, језик ће је пронаћи, изпарсовати и извршити функцију. Како на Windows платформи нема разлике између великих и малих слова у имену датотеке, то практично значи да аутоматски нема ни разлике између великих и малих слова у имену функције. Али како парсер чита име функције не из датотеке него из деклерације функције, то значи да се на идентичан начин функција мора и позвати. Из тог разлога је можда најбоље име функције конструисати коришћењем малих слова.

Иза деклерације функције налази се блок комади које представљају функцију и који се налази у витичастим заградама. Пример:

```
function c = hipotenuza(a, b) {
   /* dužinu hipotenuze pravouglog trougla
      a, b su kompatibilne matrice sa dužinama kateta */

   //25.07.2011, v1, Aleksandar

   c = math.sqrt(a^2 + b^2);
}
```

Предложена функција рачуна дужину хипотенузе када су дате дужине две катете правоуглог троугла. Ако би смо после ове дефиниције позвали ову функцију са `hipotenuza(6, 8)` добили би смо вредност десет.

Оно што још треба приметити јесу линије испод деклерације функције. Прва линија почињу са карактерима „/*“, што означава почетак коментара, садржаја који се неће извршавати. Коментар се завршава са „*/“. Коментар се може налазити у једној линији, у том случају је ознака за коментар „//“ и простире се до краја те линије.

Интерпретатор овог кода извршава линију по линију, тј. комаду по команду јер у једној линији кода може бити смештено више команди које су раздвојене карактером „;“. Овај карактер као и карактер за нови ред означава крај команде.

Функција која носи име као и фајл у који је снимљена се назива главном функцијом. Овако дефинисану функцију је могуће позвати тј. извршити из било које друге функције. Поред ове функције у фајлу је могуће дефинисати и друге функције које се у том случају називају локалним. Оне су видљиве само коду главне функције као и другим локалним функцијама. Локална функција може да носи име неке друге глобалне функције, али ће она својој глобалној функцији бити приоритетна, док је наравно друге глобалне функције неће ни видети, тако да ће се њихов позив односити на истоимену глобалну функцију. На овај начин обезбеђено

је немешање функција, тј. могуће је у једном фајлу написати све функције које су потребне за извршење потребног кода.

Датотеке, фајлове са функцијама могуће је разврставати у директоријуме. Ово је начин да се фунције сличног садржаја држе заједно, али и да се додатно држи чист простор имена функција. Ово је начин прављења пакета функција које се унутар себе виде и којима додатни пакети неће сметати, тј. свака функција унутар пакета је јасно дефинисана у односу на остале функција на систему, па макар имале и исто име.

Функцију је могуће позвати тј. извршити на два начина, навођењем имена функције и:

- у заградама, одвојених зарезом вредности који се предају њеним аргументима, што би називали функцијски позвив, и
- без заграда, али тада сви аргументи морају бити именовани и одвојени знаком „/", што називамо макро, или процедуралним позивом.

Функцијиски позив

Као пример узимамо позив већ описане функције која рачуна дужину хипотенузе правоуглог троугла када су дате дужине две катете, а која се зове хипотенуза

```
h = hipotenuza(6, 8)
```

овим позивом функцији хипотенуза предајемо вредности шест и осам и њен резултат додељујемо матрици `h`. Ово је позициони начин предавања вредности аругментима функције, и у њему је могуће неке аргументе изоставити, тако да ако би смо желели да изоставимо други аргумент онда би позив био

```
h = hipotenuza(6)
```

у овом случају други аргумен функције, аргумент `b`, би остао недефинисан (овај позив би довео до грешке јер функција очекује да је `b` дефинисано). Уколико би

желели да изоставимо први аргумент онда би написали

```
h = hipotenuza( , 8)
```

дакле, постојао би један зарез без навођења вредности испред њега, на овај начин аргумент `a` би остао недефинисан. Поред овог позиционог навођена предложени језик подразумева и предавања аругмената на основу имена, у том случају поред навођења имена аргумента навео би се и знак једнакости па тек потом његова вредност, на пример:

```
h = hipotenuza(a=6, b=8)
```

Овакав позив функције омогућава да се лакше изоставе неки аргуенти, тј. чини позив функције читљивијим:

```
h = hipotenuza(b=8)
```

Могуће је комбиновати позициони и именовани тип позива, што се обично и чини, наводјењем првих један или два позициона аргумената, док се остали ређе коришћени аргументи наводе преко имена. Узмимо као пример функцију `mean`, која има следеће аргументе: `x`, `weight`, `trim`, `valid`, `by`. Где је: `x` матрица података за коју треба наћи просечну вреност, `weight` вектор пондера са којим се пондерише свака колона матрице, `trim` је пропорција највећих односно најмањих вредности које треба избацити из анализе, `valid` је аргуемент логичког типа и који одређује да ли треба у анализу узети само валидне податке или и системски недостајуће, и `by` је аргумент текстуалног типа који одређује да ли просек треба наћи за све елементе матрице, или за сваку колону посебно или за сваки ред посебно. Један могући позив функције ако би смо желели да нађемо просек сваке колоне матрице `A`, али само за валидне податке би био:

```
mean(A, , , true)
```

што би био позициони позив који није најлакше прочитати како би се схатило шта су аргументи позива, те је зато боље користити:

```
mean(A, valid = true)
```

подразмева се да се `mean`, просек тражи од матрице `A`, али је сада јасно да је `true` вредност аргумената `valid`, тј. да просек треба наћи само за валидне податке. Овај позив је могуће написати и овако:

```
mean(valid = true, A)
```

чиме би он био мање разумљив, али је ово такође валидан функцијски позив овде дат као пример како предложени језик процесира аругменте позива

Редослед повезивања предатих вредности из позива са аругментима функције је:

- парсер прво идентификује све аргументе који су именовани, а онда
- све преостале, неименовне аргументе у позиву по реду повеже са преосталим аругментима функције, који нису предати на основу имена.

Име аругмената, у позиву, је могуће скратити све док оно јединствено идентификује аругмент функције, тако да је прошли позив било могуће написати и овако:

```
mean(A, v = true)
```

За арументе у функцији је могуће дефинисати стандардне вредности (енг. default values) које ће аргумент узети ако његова вредност није дефинисана тј. наведена у позиву. Тако је стандардна вредност за аргумент `trim` у већ описаној функцији `mean` нула, што заправо значи да се ниједна вредност неће изузети из матрице `x` приликом рачунања просека.

Аргумент може имати поред стандардне вредности и специфицирану и листу дозвољених вредности, тако да рецимо аргумнет `by`, у функцији `mean`, који је текстуалног типа има три дозвољене вредности `columns`, `rows`, `matrix`. Где је вредност `columns`, која је прво наведена самим тим и стандардна вредност овог аргумента (dafault вредност). Тако да ако овај аргумент није наведен у позиву,

функција ће вратити редни вектор са просеком сваке колоне. Могуће је скраћено навести и вредност аргумента који има дефинисану листу могућих вредности, наравно да аругмент функције мора у том случају бити текстуалног формата. Тако да би позив функције `mean` која би вратиал просек за све елементе матрице `A` мога да гласи и

```
mean(A, by="m")
```

уместо

```
mean(A, by="matrix")
```

Како функција `mean` враћа три вредности тј. три вектора/скалара у зависноти од аргумента `by`: први са просеком матрице, други са бројем елемената на основу којих је израчунат тај просек, и трећи са пондерисаним бројем елемената за елементе над којима је израчунат тај просек. Како се излазни аргументи враћају само у позиционом облику, пример позива који памти све три вредности би изгледао овако:

```
[m, n, nw] = mean(A, w, valid=true, by="matrix")
```

Ако би смо желели да само некој променљивој, `nw`, доделимо број пондерисаних валидних вредност које су пондерисане са колонским вектором `w`, морали бисмо да ставимо два зареза испред:

```
[ ,  , nw] = mean(A, w, valid=true, by="matrix")
```

Поред тога, уколико би желели само да запамтимо (у матрице `m` и `err`) само вредности просек и код грешке, као што је наведено код грешке се враћа као додатна вредност поред стандарно дефинсианих, онда би позив изгледао овако:

```
[m,  ,  , err] = mean(A, w, valid=true, by="matrix")
```

Процедурални, мета-командни позив функције

Поред стандардног начина позивања функција могуће је функције позвати и као мета-команду. У овом случају иза имена функције не треба стављати заграде, већ се аргументи једноставно наводе иза имена функције. Знак за крај наводјења вредности једног аргумента је карактер „/“. Док је тачка ознака за крај навођења последње наведеног аргумената и уједно представља крај датог функцијског позива. Карактери за одвајање аргумената и краја позива су узети тако да буду слични са SPSS командим језиком. Пример за позив већ описане функције хипотенуза би био:

```
hipotenuza   a = 8
            /b = 6.
```

Имена аргумената смо могли изоставити из позива, али у том случају би смо добили мање читљив програм:

```
hipotenuza 6 / 8 .
```

Разлика ове врсте позива у односу на стандардни позив, је да он све аругменте који нису број сматра текстуалним вредностима, тако би у следећем примеру

```
moja_funkcija arg1= var1 var2.
```

аргументу функције `moja_funkcija` који се зове `arg1` био предат текстулани вектор са два елемента `["var1","var2"]`, а не вредности варијабли које се зову `var1`, `var2`. Ово је у ствари пречица за позивање процедура над варијаблама у табели података, где су аргументи обично имена варијабли над којима треба урадити анализу, а не колонски вектори података које те варијабле држе.

Још једно важно обележије оваквог позива је и то да он не „узима“ излазне вредности, тј. чак и да функција враћа излазне вредности, њих је немогуће предати променљивима. Ово је процедурални или командини начин позива, где нас не итересује вредност коју функција враћа, која се стандардно тада назива процедуром већ само трансформација коју та процедура изводи.

Уколико ипак у оквиру овог начина функцији хоћемо да предамо вредност промењљиве, а не стринг са њеним именом, онда испред ње треба да навести карактер #. Пример

```
a = ["var1","var2"]
moja_funkcija arg1= #a .
```

позив из примера је еквивалентан претходно наведеном. У позивању функција овим начинон, у неким језицима описаним и као макро позив, треба бити опрезан јер он уствари функцији предаје стринг, тј. текстуалне вредности.

Наведена два начина су уствари „програмерски“ и „кориснички“ начин позивања функција. Први начин је можда бржи али је зато кориснику тежи за читање и разумевање.

Пакети функција

Пакети су заправо начин да се функције одвоје једне од других те да додавањем нове групе функција недођe до преклапања имена функција. На овај начин избегавамо ситацију да додајемо нову функцију која има исто име као већ дефинисана функција те да онда дође до грешака у већ тестиранимм програмима, јер смо уствари заменили функцију, која иако се исто зове може да изводи неке сасвим друге трансформације тј. прорачуне.

Пакет је уствари диреторијум (енг. folder) у коме су смештени синтаксне датотеке (fajl-ови са .s екстензијом) у којима се налази код функција. Ако би смо направили директоријум `trig` и у њега снимили датотеку `hipotenuza.s` са већ описаном функцијом која рачуна дужину хипотенузе правоуглог троугла, када су дате дужине две катете, позивање ове функције из других пакета би морало да се одвија преко тачка оператора:

```
trig.hipotenuza(6, 8)
```

На овај начин чак и ако се направи нова функција хипотенуза у неком другом пакету, директоријуму она ће бити јасно одвојена од ове истоимене функције.

Могуће је наравно правити подпакете тј. поддиректоријуме у оквиру пакета, и тада би ако би желели да позовемо одређену функцију морали да користимо тачну путању користећи више тачка оператора како би смо одвојили све директоријуме. Пример:

```
paket.podpaket1.podpaket2.moja_funk()
```

Све функције у оквиру пакета и све функције из свих подпакета тог пакета пронаћи ће функцију без коришћења тачка оператора. Тако да се име пакета може уствари накнадно конструисати. Функције које се налазе у подпакетима виде све функције које су старије, које се налазе изнад њих, тако да гледајући претходни пример све функције из `podpaket1` виде све функције које су дефинисане у оквиру тог подпакета и у оквиру пакета, и оне се не морају позивати са тачка оператором. Али ако нека функција из `podpaket1` хоће да позове функцију која се налази у `podpaket2`, као што је то случај са `moja_funk` онда ће позив морати да користи оператор тачка која ће реферирати не на почетни ниво свих пакета већ на почетак референце уствари овај пекет, пример позива `moja_funk` из неке функције из `podpaket1` би био:

```
.podpaket2.moja_funk()
```

На овај начин смо дошли и до редоследа на основу кога део програма који повезује функцијске позиве са кодом тих функција (енг. linker) одређује о којој се функцији заправо ради, наравно ако није коришћен оператор тачка, јер је у том случају функција је једнозначно одређена:

1. системске функције – ако постоји ситемска функција са тим именом она ће прва бити повезана са позивом, ово заправо значи да је немогуће направити корисничку функцију која ће заменити истоимену сиситемску

2. корисничка функција која је дефинисана у истом фајлу као и функција из које потиче позив

3. корисничка функција која је главна функција фајла који се налази у истом пакету као и функција из које потиче позив

4. корисничка функција која је главна функција фајла који се налази у најближем надпакету пакета у коме је функција из које потиче позив

Предавање променљивих функцијама по референци

Стандардан начин предавања аргумената функцијама јесте по њиховој вредности, тј. матрица која се предаје функцији се ископира у меморији, локалном простору функције, и ова вредност заправо служи као аргумент функције. Свака промена над овом вредношћу је само видљива унутар функције и изводи се над њеном локалном копијом. По завршетку функције матрица која је предата функцији има исту вредност као и пре позива. Пример:

```
function inc(x) {
   x = x+1
   x
}

a = 10;
inc(a)
a
```

Овај програм који дефинише функцију `inc`, функцију која инкрементира предату вредност за један, ће исписати

```
x = 11
a = 10
```

јер се аргумент предаје по вредности тако да је аргумент функције увећан за једну вредност више, али само у оквиру те функције, по изласку из функције вредност променљиве `a` је и даље десет, што се види из последњег реда исписа.

У случају да се аргумент предаје по референци, промене на аргументу функције су трајне. Ово је веома корисно када су предаје меморијски захтеван аргумент нпр.

матрица великих димензија, да би смо спречили њено копирање и дупло заузимање меморијских ресурса, боље је тражити да се аргумент преда по референци.

Ова предаја се врши тако што се у дефиницији функције иза аргумента који се жели узимати по референци стави карактер „`'`” који се иначе на другим местима односи на операцију транспозиције матрице. У овом случају сама матрица се предаје функцији, а локално име треба схватити као њен локални назив. Пример претходне функције би у случају предаје по реферeци био:

```
function inc(x'){
   x = x+1
}

a = 10
inc(a)
a
```

Испис из овог програма је овај пут:

```
x = 11
a = 11
```

Дакле у самој дефиницији функције се дефинише и да ли јој се вредност аргумената предаје по референци или по вредности, позив функције и предаја аргумената је иста у оба случаја. Тако да је на страни програмера, аутора функције да се одлучи за један од ова два начина, а не на кориснику те функције, за њега је позив и даље остаје исти.

Стандардне вредности аргумената и распон валидних вредности

Уколико у функцијском позиву неки аргумент није наведен он ће остати недефинисан, и приликом првог његовог коришћења програм ће пријавити грешку да покушавамо да користимо недефинисану променљиву. Ово се може спречити дефинисањем стандарних (енг. дефаулт) вредности за сваки аругмент у функцији

који ће он добити ако његова вредност није дефинисана у функцисјком позиву

```
function y = stepen(x, stepen=2) {
   y = x^stepen
}

stepen(3)
```

овако дефинисана функција, одређује вредност два као стандардну вредност аргумента `stepen` тако да њен позив `stepen(3)` уствари означава квадрирање вредности три.

Овде треба приметити да се функција и аргумент исто зову, али се разликују по употреби заграда, функцијски позив је обавезно праћен са отвореним заградама, у супротном случају ради се о променљивој.

Уколико би функција степен била дефинисана на следећи начин

```
function y = stepen(x, stepen=[2,3,4]) {
   y = x^stepen
}

stepen(3)
```

где је аргументу степен додељена не једна него више стандардних вредности, то значи да је први елемент у низу стандардна вредност коју ће аргумент добити ако у позиву функције његова вредност није дефинисана, али ако је вредност овог аргумента дефинисана онда она мора бити једна од наведених вредности. Тако да ће позив `stepen(5)` довести до грешке.

Системске функције

Ово су функције које су изворни део предложеног језика, и за њих не постоје „.s“

фајлови. Како су написане у C језику оне су самим тим и нешто брже него корисничке функције. Постоје два пакета међу системским функцијама, `math` и `data` пакет. Први дефинише стандардне аритметичке и тригонометријске функције, док други дешинише функције које раде са табелом података. Ипак треба узети у обзир да већина системских функција не припада ни једном пакету.

Матрична алгебра

lu

Ова функција декомпонује дату квадратну матрицу на две тј. три матрице, тако да се дата квадратна матрица може представити као производ те две матрице, од којих је прва доње троугаона (изнад главне дијагонале су нуле), друга горње троугаона а трећа је пермутациона матрица (матрица која у сваком реду и свакој колони има само једну вредност једнаку један, а све остало су нула вредности).

```
[L,U,P] = lu(A)
```

Ово је класичан проблем LU декомпозиције са парцијалним пивотирањем. Тако да је `L` доњетроугаона матрица са јединицама на главној дијагонали, `U` горњетроугаона матица, а `P` пермутациона матрица. У том случају важи да је

```
L*U = P*A
```

пошто матрица `P` множи матрицу `A` са лева, то уствари значи да пермутујемо колоне ове матрице. Како је матрица `L` доњетроугаона и има јединице на дијагонали њена детерминанта је једнака један, пермутациона матрица има такође детерминанту једнаку један, па остаје да је детерминанта матрице `A` уствари једнака детерминанти матрице `U`, која се, како је ова троугаона матрица, лако рачуна: као производ вредности на главној дијагонали.

У случају да се траже само две матрице као резултат,

```
[M,U] = lu(A)
```

`M` матрица је уствари једнака

```
M = Pt * L
```

тј.

```
M = P' * L
```

исказано у синтакси предложеног језика.

Могуће је позвати ову функцију и тражити као резултат само једну матрицу,

```
Y = lu(A)
```

у том случају у матрици `Y` су дате обадве матрице и `L` и `U`, тј.:

```
Y = L + U - I
```

det

Функција која рачуна детерминанту за дату квадратну матрицу.

inv

Функција за рачунање инверза регуларне квадратне матрице. За разлику од оператора `^` који ће у случају сингуларне матрице избацити грешку, функција `inv`, као и све функције, враћа поред инверза и код грешке који ће у случају сингуларне матрице бити ненулта вредност.

eig

Функција `eig` рачуна својствене вредности и својствене векторе квадратне симетричне матрице. Важна напомена је да ова процедура не проверава симетрију матрице и узима само вредности испод главне дијагонале, претпостављајући да су вредности изнад дијагонале једнаке јер је матрица симетрична.

Од функције је могуће тражити једну или две вредности, ако се тражи само једна вредност, функција враћа вектор са сортираним својственим вредностима у нерастућем поретку, а ако се траже две матрице онда је прва дијагонала матрица са сортираним својственим вредностима на дијагонали, а друга матрица са својственим векторима.

Постоје два облика `eig` функције

```
[L, X] = eig(A)
```

у овом случају решење је уствари решење једначине

```
(A - lI)x = 0
```

где су `l` својствене вредности које ће се наћи на дијагонали матрице `L`, а `x` својствени вектори који ће постати колоне ортогоналне матрице `X` (`X'*X = X*X' = I`).

Други облик ове функције је са два улазна аргумента и тада је то генерализовани проблем својствених вредности тј. случај у коме се својствене вредности не рачунају у `I` простору него у простору који разапињу вектори из `B`.

```
[L, X] = eig(A, B)
```

У том случају се у ствари решава следећи проблем:

```
(A - lB)x = 0
```

тако да су добијени својствени вектори ортогонални у простору матрице Б, тј.

```
X^tBX = I
```

svd

Функција `svd` рачуна сингуларне вредности леве и десне својствене векторе дате матрице. То је декопозија матрице на производ три матрице, од којих су две колонски ортогоналне, а трећа је дијагонална.

$$A = LSD^t$$

Уколико се од функције тражи један аргумент добиће се само вектор са, у нерастућем поретку сортираним, сингуларним вредностима.

Уколико се од функције траже три вредности онда су то:

- матрица левих сингуларних вектора
- дијагонална матрица са сортираним (у нерастућем низу) сингуларним вредностима
- матрица десних сингуларних вектора

```
[L, S, D] = svd(A)
```

тако да је:

```
L*S*D' = A
```

Уколико матрица `A` има више редова него колона, онда је матрица `L` истих димензија као и матрица `A`, Матрица `S` је дијагонална матрица реда к који одговара броју колона матрице `A`, а матрица `D` је ортгонална матрица димензија к са к.

Уколико матрица `A` има више колона него редова, онда је матрица `D` истих димензија као и матрица `A`, ова матрица је колонски ортоганална. Матрица `S` је дијагонална, а матрица `L` је ортоганална, и димензија је к са к, где к представља број редова матрице `A`.

Тако ако је:

```
[L1, S1, D1] = svd(A)

[L2, S2, D2] = svd(A')
```

тада је:

```
S1 = S2

L1 = D2

D1 = L2
```

diag

Ова функција или прави вектор од главне дијагонале дате матрице, ако је матрица аргумент, или ако је аргумент вектор, прави квадратну матрицу са овим вредностима на главној дијагонали.

trace

Траг матрице, или сума вредности на главној дијагонали.

Специјалне функције над матрицама

size

Враћа димензије дате матрице. Аргументи ове функције су `x` и `dimension`. Аргумент `x` је уствари матрица чије димензије тражимо, а `dimension` означава коју димензију желимо: прву, бој редова или другу, број колона дате матрице. Ако овај аргумент није наведен, функција враћа редни вектор са два елемента, бројем редова и бројем колона дате матрице.

nrow

Функција која враћа број редова дате матрице.

ncol

Функција која враћа број колона дате матрице.

make

Прави матрицу датих димензија чији сви елементи једнаки датој вредности. Аргументи ове функције су `rows`, `columns`, `value` и `size`. Први аргумент је број редова нове матрице, други је број колона нове матрице, а трећи вредност сваког од елемената нове матрице, док је четврти аргумент `size` вектор са два елемента који представљају број редова и колона жељене функције. Наравно или се специфицра вектор `size` или скалари `rows` и `coluns`.

$$\texttt{make}(9,2,3) = \texttt{make}(\texttt{size=[2,3], value=9}) = \begin{bmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix}$$

zeros, ones

Ове функције као и `make` праве матрице само где су све вредности једнаке нули тј. јединици. Аргументи ове функције су `rows` и `columns` или `size`. Ово су у ствари само други називи за функцију `make` која прави матрицу од нула тј. од јединица.

```
zeros(2,3) = make(0,2,3)

ones(3,3) = make(1,3,3)
```

eye

Прави матрицу идентитета датих димензија. То је матрица код које су сви елементи на главној дијагонали једнаки један, а све остале ван-дијагоналне

вредности једнаке нули. Аргументи ове функције су `rows`, `columns` или `size`, слично претходним функцијама.

$$\text{eye}(2,3)=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}$$

$$\text{eye}(2)=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$

reshape

Ова функција враћа матрицу датих димензија са елементима дате матрице који се посматрају колонски. Аргументи функције су `x`, `rows`, `columns` и `size`. Где је `x` матрица коју треба трансформисати, `rows` и `columns` број редова тј. колона нове матрице. Ако ова два аргумента нису специфицирана онда треба навести аргумент `size`, који је вектора са два елемента.

Број елемената дате матрице мора одговарати броју елемената резултујуће матрице.

$$A=\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix}$$

$$\text{reshape(A,size=[2,3])}=\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}$$

Уколико се од матрице жели направити вектор настављањем њених колона, сличан се ефекат може постићи и са оператором :.

$$A(:)=\begin{bmatrix}1\\ 2\\ 3\\ 4\\ 5\\ 6\end{bmatrix}$$

find

Враћа „векторски“ индекс елемената матрице различитих од нуле, то је индекс ненултих елемента када матрицу посматрамо као један вектор, добијен када све колоне спојимо једну на другу.

$$A = \begin{bmatrix} 10 & 7 & 8 \\ 2 & 15 & 6 \end{bmatrix}$$

$$\text{find}(A>7) = \begin{bmatrix} 1 \\ 4 \\ 5 \end{bmatrix}$$

any

Ова функција ако је аргумент вектор даје као резултат вредност један ако је бар један од елемената вектора различит од нуле, у супротном функција враћа вредност нула.

Ако је аргумент матрица онда се ова претрага за ненултим вредностима ради над сваком колоном, тако да је резултат редни бинарни вектор који има елемената колико и дата матрица колона.

freq

Функција која враћа фреквенце вредности у датој матрици. Од функције се могу захтевати две вредности, у том случају су то два вектора, први са вредностима и други са њиховим фреквенцама, који су су сортирани у растућем поретку по вредностима.

$$A = \begin{bmatrix} NaN & 3 & 5 \\ 3 & 0 & 4 \end{bmatrix}$$

```
[v,f] = freq(A)
```

у том случају добијају се следећи вектори:

$$v = \begin{bmatrix} 0 \\ 3 \\ 4 \\ 5 \\ NaN \end{bmatrix} \qquad f = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Аргументи ове функције су `x` и `weight`. Први аргумент представља матрицу од чијих се елмената траже фреквенце, док други аргумент представља пондер сваког реда матрице.

sort

Функција служи за сортирање вектора, тј. матрица по одређеном редоследу, и то по одређеној колони, тј. реду. Аргументи су `x`, `order`, `column` и `row`. `x` је матрица која се сортира, `order` је стринг који има вредности `ascending` или `descending`, ако није наведен то значи да је `ascending`. `column` је арумент који означава колону матрице по којој треба сортирати податке, и у том случају се сортирају редови матрице, док је аналогно `row` број реда по коме треба сортирати колоне матрице. Наравно или се специфицира `column` или `row`, ако није ни један наведен онда је то еквивалнтно са `column=1`.

Ова функција узима аргумент по референци, тј. сортирање може да мења саму матрицу, ако се од функције не тражи повратна вредност матрице. Уколико се тражи повратна вредност матрице онда сортирање неће променити саму матрицу.

Ако се од функције траже две излазне вредности онда друга вредност представља индекс редова тј. колона оригиналне матрице поређане као и редови тј. колоне у сортрираној матрици.

$$A=\begin{bmatrix} NaN & 2 \\ 3 & 5 \\ 0 & 2 \\ 2 & 0 \end{bmatrix}$$

```
[ , indx] = sort(A, column=1)
```

flip

Функција `flip` служи за обртање матричних елемената као у огледалу, први ће постати последњи а последњи први. Аргументи су `x` и `what`, где први представља саму матрицу, а други димензију по којој се врши замена елемената. Вредности овог аргуента су `columns` и `rows`, где ако овај аргумент није наведен стандардно се узима `rows`.

Ова функција такође узима референцу дате матрице као аргумент, те ако се од функције не захтева повратна вредност сама матрица ће бити промењена, а ако се тражи повратна вредност онда матрица остаје непромењена.

$$A=\begin{bmatrix} NaN & 2 \\ 3 & 5 \\ 0 & 2 \\ 2 & 0 \end{bmatrix}$$

```
flip(A)
```

после чега матрица А има следећу вредност

$$A=\begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 3 & 5 \\ NaN & 2 \end{bmatrix}$$

insert

Функција која у наведену матрицу убацује нове редове тј. колоне. Аргументи су `x`, `index`, `value` и `what`. Као да смо навели `w[index] = value`, али ова контрукција ће наведене елементе заменити, ако они постоје, док ће ова функција наведене елементе додати испред наведеног индекса. `x` представља матрицу у коју треба додати наведене редове тј. колоне, `index` је индех реда/колоне испред које треба додати нови ред/колону, `value` је матрица са редовима/колонама које се додају, а `what` је аргумент који специфицира шта се додаје редови или колоне. Његове вредности су `rows` или `columns`, што значи да ако није наведен додају се `redovi`.

Наравно број елеманата у аргументу `index` мора одговарати броју редова тј. колона у аргумету `value`, у зависноти шта се додаје. Такође редове тј. колоне из аргуента `value` мора бити могуће додати у наведену матрицу, тј. број редова односно колона мора бити једнак у оба аргумента. Ако је вредност индекса већа од димензија матрице онда се нови елементи додају иза последњег реда/колоне, али ово је могуће урадити и преко оператора конкатенације `[]`.

Пример:

```
insert([10 20;30 40],[1; 2; 2; 1],[1 1;2 2; 3 3; 4 4])
```

овај израз даје као резултат следећу матрицу

$$\begin{bmatrix} 1 & 1 \\ 4 & 4 \\ 10 & 20 \\ 2 & 2 \\ 3 & 3 \\ 30 & 40 \end{bmatrix}$$

Основне математичке функције

Ове функције се налазе у уграђеном пакету `math`. Тако да се оне позивају са префиксом пакета нпр.

```
math.sin(3.14)
```

пошто је ово уграђени пакет директоријум са овим функцијама не постоји. Све ове функције се извршавају над сваким елементом дате матрице, и враћају две вредности, повратну вредност функције и вредност грешке.

`sin`	синус дате вредности у радијанима
`cos`	косинус дате вредности у радијанима
`tan`	тангенс дате вредности у радијанима
`asin`	arc sin функција, тј. инверзна синус функција
`acos`	инверзна cos функција
`atan`	инверзна тан функција
`sinh`	хиперболички синус
`cosh`	хиперболички косинус
`tanh`	хиперболички тангенс
`exp`	експоненцирање дате вредности
`log`, `ln`	природни логаритам
`log10`	логаритам са основом десет
`ceil`	заокруживање дате вредности ка већој вредности
`floor`	заокруживање дате вредности ка мањој вредности

`fix` заокруживање ка нули

`round` заокуживање вредности (уз поштовање свих правила заокруживања)

`abs` апсолутна вредност дате вредности

`sqrt` квадратни корен дате вредности

`sgn` предзнак дате вредности, ова функција враћа један ако је вредност позитивна, минус један ако је вредност негативна и нулу ако је вредност нула или `NaN`

Све ове функције, сем функције `sgn`, ако им је предата вредност `NaN`, `Inf` или `-Inf` враћају управо ту вредност, јер је наравно

```
math.sgn(-Inf) = -1
```

Пакет `math` поред наведених функција садржи и две константе, то су

`math.pi` вредност која представља однос обима круга и његовог пречника (приближно 3.14159)

`math.e` Ојлеров број (приближно 2.71828)

Приметите да су ово константе па према томе њихово позивање не захтева употребу заграда као код функција, пример

```
math.sin(math.pi)
```

даје вредност нула.

Функције дескриптивне статистике

`sum`	сума елемената дате матрице
`prod`	производ елемената дате матрице
`min`	минимум
`max`	максимум
`range`	распон
`mean`	просечна вредност
`sd`	стандардна девијација узорка
`var`	варијанса узорка
`skw`	скјунис
`kur`	куртосиз
`descriptive`	функција која враћа све претходне статистике у оквиру једне мапе

Све ове функције представљају основни облик функција које ако им је предат вектор рачунају жељени статистик на свим елементима датог вектора, а ако им је предата матрица онда рачунају статистик за сваку колону.

Све ове функције имају исте аргументе `x`, `weight`, `trim`, `valid` и `by`:

- `x` је матрица за чије елементе тражимо жељени статистик,
- `weight` је вектор са пондерима са којима се пондерише сваки ред матрице
- `trim` представља пропорцију екстремних (најмањих и највећих) вредности које ће бити искључене из рачунања узетих заједно, дакле пропорцију

елемената који ће бити искључени са сваке стране дистрибуције

- `valid` је логички аргумент, ако му је вредност различита од нуле само валидне вредности ће бити укључене у рачун. Дакле све вредности које нису `NaN`, јер је `NaN` резервисана за нестојући податак. Ово треба узети са опрезом јер се у овом случају `Inf` рачуна у валидне податке тако да ће свака операција са њим дати исту ту вредност

- `by` представља текстуални аргумент који означава димензију по којој се врши рачунање статистика. Могуће вредности овог аргумента су `column`, `row` и `matrix`, при чему ако аргумент није наведен његова стандардна вредност је `column`. У случају да је његова вредност `column`, статистк ће бити израчунат за сваку колону, тј. резултат функције је редни вектор са статистицима за сваку колону. А ако је његова вредност `row` онда ће статистик бити израчунат за сваки ред дате матрице, резултат је колонски вектор са статистиком за сваки ред, а ако је `matrix` онда се статистик рачуна за све елементе матрице и у том случају је резултат скалар.

Све ове функције враћају максимално три вредности, и наравно могуће је захтевати и вредност грешке која је у овом случају четврта вредност. Прва вредност коју функција враћа је наравно вредност статистика који функција израчунава, друга вредност је број елемената на основу којих је статистик израчунат и трећа вредност је пондерисани број елемената на оносву којих је статистик израчунат. Ако није коришћен пондер други и трећи аргумент су наравно исти.

Једина разлика у случају функције `descriptive` је да она враћа мапу са свим овим статистицима. Наравно да би се могло приступати елементима мапе, потребно је знати њихове кључеве, који су у овом случају исти као и имена појединачних функције: `min`, `max`, `range`, `sd`, `var`, `kur`, `skw`, `mean`, `n` и `nw`, где су послења два уствари број елемената на основу којих су израчунате ове променљиве.

Ово је једина функција која враћа мапу у предложеном језику. Како је рачунање великог броја дескриптивних статистика у једном пролазу рачунски ефикасно, мапа делује као природије решење за враћање више вредности, иако и даље стоји да је код у том случају не јаснији.

cumsum

Функција `cumsum` ће за дати вектор израчунати кумулативне суме његових елемената, дакле суму тог елемента са свим претходним. Функција има два аргумента `x` и `by`. `x` представља саму матрицу, а аргумент `by` стандардно орјентацију по којој се функција примењује. Вредности су му `column`, што је уједно и стандарда вредност, `row` и `matrix`. Стим да `matrix` прави кумулатив прве колоне, а онда настављајући на следећу итд.

cumprod

Слично функцији кумулативне суме, ове функција рачуна кумулативни производ, дакле производ елемента са производом свих претходних. Аргументи су исти као и у претходној функцији и имају исто значење.

corr

Функција за рачунање интеркорелација тј. кроскорелација између колона датих матрица.

Аргументи ове функције су `x`, `y` и `weight`. Ако је наведен само један аргумент, тј. `x`, функција враћа матрицу интеркорелација колона ове матрице. Ако су наведена два аргумента, и `x` и `y`, онда функција враћа кроскорелације између колона ове две матрице.

Аргумент `weight` представља пондер којим се пондеришу вредности у наведеној или наведеним матрицама. Могуће је навести и празну матрицу као овај аргумент али онда она неће бити узета у обзир.

Функције псеудослучајних бројева

rand

Генерише матрицу случајних бројева. Аргуенти ове функције су `rows`, `columns`, `type` и `size`. Стандардно `rows`, `columns` су бројеви редова тј. колона нове матрице, тј. алтернативно `size` аргумент специфицира то исто кроз, вектор са два елемента. `type` аргумент представља тип дистрибуције из које се потиче случајни број, вредности овог аргумента су `uniform`, што је и стандардна вредност, и означава униформу вредности из интервала 0 до 1, и `normal`, у ком случају случајни број потиче из стандардне нормалне дистрибуције. Уколико није специфициран ниједан аргумент ове функције она враћа скалар из униформне дистрибуције.

srand

Функција која поставља стартну вредност за генератор случајних бројева на дату скаларну вредност, тј. ако функцији није предат ни један аргумент на број секунди од почетка UNIX епохе (1. јануар 1970). Због овага треба бити обазрив јер ако се `srand` функција позива без аргумената у истој секунди псеудослучајна секвенца ће бити иста.

Функције за рад са стринговима

strlen

Враћа дужину стринга у карактерима за сваки елемент матрице стрингова. Ова функција је UTF компатибилна.

strcat

Функција спаја два стринга у један, простом конкатенацијом. Исти ефекат се

добија и коришћењем операције + над стринговима.

strcmp

Врши поређење два стринга. Уколико су исти враћа нулу, а ако је први стринг испред другог (посматрано када би стрингове сортирали по абецеди) онда је резултат минус један, а ако је први иза другог онда је резултат један.

lcase

Враћа матрицу са стринговима претвореним у мала слова. Ова функција је ASCII функција, тј. ради исправно само на ASCII карактерима.

ucase

Као и претходна функција само се стринг конвертује у велика слова.

left

Функција која од сваког елемента дате матрице са стринговима, враћа специфициран број левих карактера. Аргументи функције су `x` матрица из које треба издвојити карактере, и `len` што представља број карактера са леве стране који треба узети. Ако је дати стринг празан, функција враћа празан стринг. Ако је наведени број карактера већи од дужине стринга, или је наведен `NaN`, `Inf`, `-Inf` функција враћа цео стринг. Функција подржава правила бинарних операција између матрица неједнаких димензија, дакле ако једна од матрица није скалар, потребно је да им је бар једна димензија иста. Функција је UTF компатибилна.

right

Функција која се понаша слично као и претходна функција `left`, и за коју важе иста правила, једино што не враћа карактере са почетка стринга него са његовог краја.

implode

Функција која ће све елементе прво наведене (x) матрице стрингова сабрати тј. надовезати један на други. Ако је наведен и други аргумент ове функције, стринговни скалар (by), онда ће он бити коришћен као делимитер између свака два елемента прве матрице.

explode

Функција која супротно претходној, дели први стринг на вектор стрингова користећи као делимитер други наведени елемент. И један и други аргумент морају бити наведени, и и један и други морају бити скалари. Уколико је аргумент by празан стринг дати стринг ће бити издељен на сваки карактер.

str2num

Функција која од дате матрице са нумеричким подацима прави матрицу са стринг подацима тако што сваку вредност конвертује по специфицираном формату који је први аргумент ове функције.

num2str

Функција која од матрице са стринг подацима прави матрицу са нумеричким вредностима. За разлику од претходне функције ова функција има само један аргумент.. Уколико се унети текст не може протумачити као број, функција враћа NaN.

Функције за испис

print

Функција која исписује матрицу као табелу. Матрица коју треба исписати је први аргумент функције који се стандардно зове `x`.

Ову функцију је могуће позвати само наводећи овај један аргумент, матрицу чије вредности треба исписати. Ово може бити било нумеричка било тектуална матрица. Ако је матрица нумерчка и ако је вредност неког елемента `NaN`, онда он неће бити исписан, ћелија овог елемента у табели ће остати празна.

Ако су специфицирани други и трећи аргумент, `rnames` и `cnames`, онда су то матрице са стринговима које представљају називе редова тј. колона дате матрице. Могуће је изоставити неки од ових аргумената. Број редова другог аргумента, `rnames`, мора бити исти као и број редова главне матрице, првог аргумента. Док број колона трећег аргумента, `cnames`, функције, тј. матрице са насловима колона, мора бити исти као и број колона главне матрице, или броју колона главне матрице и матрице са називима редова, у ком случају ће бити насловљена и колона/колоне са називима редова. `rnames` и `cnames` су матрице стрингова. Ако је испис у HTML формату, добиће се табела у којој је могуће неке ћелије спојити. Вертикално спајање ћелија се врши тако што се у ћелију које се жели спојити са ћелијом изнад упише знак „`|`”, док ако желимо да ћелије спојимо хоризонтално у ћелију коју желимо да спојимо са претходном тј. левом ћелијом треба уписати знак „`~`“. Наравно ове карактере је могуће уписати само у матрицу чији су елементи стрингови, било да је то главна `x`, матрица или матрице `cnames` тј. `rnames`.

Четврти аргумент ове функције формат, представља тексутални формат, тј. форматни стринг за сваку колону табеле по којима треба форматирати испис вредности у свакој од колона дате матрице. Уколико је наведено мање формата од броја колона у матрици, последњи наведени формат ће бити коришћен за све

преостале колоне, то значи ако је наведен само један форматни стринг он ће бити коришћен за све вредности у матрици. Овај форматни стринг прати конвенцију уведену још од FORTRAN-а, тако да је испис вредности типа `float`, са три децимална места и осам карактера за целобројну вредност, децималну тачку и број децимала дат са „`%8.3f`“.

Следећи аргументи су `rlines` и `clines`, тим редом, и представљају тип линије која се исписује. Ово су колонски односно редни вектор, са целобројним вредностима који представљају тип линије за сваку хоризонталну односно верткилану линију. Дефинисно је четири типа линија преко четири кода:

- код нула означава да линија неће бити видљива
- код један да је повучена најтања линија
- код два да је повучена дебела линија
- код три да је повучена дупла линија

Ова функција може бити коришћена и за испис линије текста простим навођењем текста који се жели исписати.

heading

Ова функција форматира предати стринг као наслов секције, табеле. Овај стринг је други аргумент функције, док је први аргумент редни број стила којим треба исписати овај наслов. Дефинисано је пет стилова за наслове.

printf

Функција прави стринг по задатом формату од предатих аргумената. Сви аргументи ове функције су скалари. Први аргумент је стринг који представља формат по коме треба форматирати предате аргументе. Формат је стандардни C формат за форматирање излазних вредности:

„`%`“ означава почетак формата за наведени аргумент

„f“ је ознака за разломљени број (број са децималним вредностима)

„d“ је ознака за целобројну вредност

„e“ је ознака за разломљени број који ће бити представљен са експонентом

„E“ као и претходни формат само користи велико слово е за представљање експонента

„s“ је ознака за стринг, тј. уместо овог аргумента биће исписан предати стринг

иза знака „%“ могуће је навести број који означава број карактера који ће бити коришћени за представљање вредности, тако формат „%5d“ означава да ће уколико је број мањи од пет децимала са леве стране бити придодати бланко карактери тако да укупна дужина исписа буде пет. Ако број није могуће приказати у жељеној ширини онда ће без подизања грешке, број бити приказан у онолико карактера колико је потребно. Поред ширине представљања броја могуће је означити и број децимала који ће бити коришћени за представљање броја, тако што ће се после ширине уписати и број децимала одвојен тачком, нпр. формат „%8.2f“ означава да ће број бити заокружен на две децимале и да ће за његов приказ бити коришћено 8 карактера (укључујући и децимални сепаратор). Ако је жељени број децимала мањи од онога који је потребан за потпуно тачно приказивање вредности, дата вредност ће бити заокружена на толико децимала. Могуће је навести само број децимала, док ширина остаје велика по потреби, у том случају за ширину се наводи вредност нула, као у „%f0.3“ означава да ће сви бројеви бити исписани са три децимална места плус број цифара за целебројни део вредности. Наравно број децимала не може бити коришћен за стринг формат.

„\n“ је ознака за нови ред

„\t“ је ознака за карактер табулације, размак (TAB)

„%%“ је ознака за проценат, како је један карактер „%“ ознака за почетак формата једног аргумента, два процента су ознака за сам карактер процента

Уколико од функције није тражена повратна вредност, резултујући стринг ће бити исписан тј. приказан. А уколико је тражена повратна вредност онда се резултујући стринг смешта у одговарајући скалар. У примеру

```
printf("F = %0.3f, df = %d", 455.43522, 433)

s = printf("F = %0.3f, df = %d", 455.43522, 433)
```

прва линија ће исписати:

```
F = 455.435, df = 433
```

док ће друга линија исти тај стринг доделити промењивој s.

Слично функцији print и ова функција може бити коришћена за исписивање текста само наводјењем текста који се жели исписати, али за разлику од функције print која предати стринг сматра једном линијом текста тј. која ће после предатог текста исписати и нови ред, ова функција то неће урадити, већ ако се следећи испис жели сместити у нови ред потребно је навести и карактер за нови ред „\n“. Пример:

```
printf("posle ove linije se prelazi u novi red\n");
```

output

Ова функција регулише да ли ће резултати функција бити исписивани или не, ако јој је аргумент једнак нули, исписивање се обуставља све док се ова функција поново не позове са не нултом вредношћу.

error

Ова функција поред тога што исписује предати стринг зауставља даље извршавање програма.

Функције о основним информацијама матрица

isnum

Функција која тестира да ли је предата матрица нумеричка. Уколико је то тачно функција враћа један у супротном враћа нулу.

isstr

Ова функција враћа јединицу ако је предата матрица матрица стрингова, тј. нулу ако је предата матрица нумеричка.

isnan

Ова функција враћа бинарну матрицу, тако што сваки елемент тестира да ли је једнак врености `NaN`. Ако је једнакост задовољена вредност ће бити један, а ако није онда је вредност тог елемента једнака нула. Ова функција је дефинисана само за нумеричке матрице.

isvalid

Слично претходној функцији ова функција тестира да ли је сваки елемент „коначан“ број. Ако је вредност елемента једнака `NaN`, `Inf` или `-Inf` вредност одговарајућег елемнта у резултујућој матрици је једнака нула, у супротном једнака је један.

Команде тока

Писање функције се своди на програмирање, за које се користе стандардне команде које контролишу ток програма.

if-else питалица

Ово је стандардна команда која зависно од испуњености услова контролише ток програма. После кључне речи `if` се наводи нумерички израз који резултира скаларом. Ако је вредност овог израза различита од нуле услов је задовољен и код који следи ову команду се извршава, било да је то једна команда или блок команди који се налази у витичастим заградама. Ако иза овог блока стоји кључна реч `else`, онда у случају да услов специфициран у команди `if` није задовољен извршава се код у блоку иза `else`. Пример:

```
sign = 1
if a>0
   abs = a
else {
   abs = -a
   sign = -1
}
```

Ово је класичана употреба грањања кода где имамо један кодни блок који се извршава ако је услов задовољен и други ако услов није задовољен.

Петље

Петља је израз за блок кода који може да се извршава више од један пут.

for петља

Ова петља се дефинише са комадом `for`. После ове команде следи име променљиве која ће се увећавати кроз сваки пролазак кроз петљу, и која се зове бројач, а после дифиниције имена знак једнакости и почетна и крајња вредност одвојене карактером „`:`“, и ако је наведен корак, тј. вредност за коју ће се бројач сваки пут увећавати, тј. смањивати ако је корак негативан. Ако овај инкремент није наведен сматра се да је један. Пример:

```
for i=1:10:2 {
   a=2^i
   print([i,a])
}
```

Овај програм ће експоненцирати број два на први, трећи, пети, седми и девети степен, и те вредности исписати заједно са одговарајућом вредношћу степена. Скалар `i`, који је у првом проласку кроз петљу једнак један, у следећем се повећава за два и тако даље док год је мањи или једнак вредности десет, која је наведена као горња граница петље.

У предложеном језику `for` петља се извршава док ког је вредност бројача мања или једнка крајњој вредности ако је инкремент позитиван, тј. док год је вредност бројача већа или једнака крајњој вредности ако је инкремен негативан, тако да у следећем примеру код петље неће бити ни једном извршен:

```
for i=1:0
   g = g + i
```

while петља

`while` петља се извршава, тј. блок наредби који следи услов у овој петљи, док год је услов који следи ову кључну реч испуњен односно док је вредност скалара који следи ову кључну реч различит од нуле. Пример:

```
a=1;
while a<10 {
  b = sqrt(a);
  a++
}
```

Евалуација услова у питалицама

У предложеном језику је имплементирана „лења“ евалуација. Ако у услову `if` тј. `while` питалице имамо оператор „или“, код кога је услов задовољен ако је само један од аргумената тачан, и евалуација првога аргумената установимо да је он тачан, други аргумент овог оператора неће бити евалуиран. Слично важи и за оператор логичког „и“, ако је први аргумент нетачан, други аргумент се не евалуира. На овај начин је могуће писати једноставније, а самим тим и читљивије услове. Пример би био функција која би за специфицирани елемент вектора требало да врати логичку вредност тачно ако је он мањи од десет и нетачно ако није или ако тај елемент ни непостоји тј. задати индекс је изван величине вектора.

```
funcion y = test_el(v, ind) {
  if ind>0 & ind<=max(size(v)) & v[ind]<10
    y = true
  else
    y = false
}
```

Наведени `if` услов има три дела, проверава се да ли је наведени индекс већи од нуле, да ли је мањи или једнак највећој димензији вектора и на крају да ли је тај елемент мањи од десет, ако су сви услови задовољени функција враћа један, а ако нису враћа нулу. Покушај референцирања елемента вектора, матрице који је изван њених димензија, производи грешку и зауставља извршење програма. Али до тога у овом примеру неће доћи јер трећи услов извршава само ако су прва два задовољена, тако да ако је `ind` негативан, први израз у услову ће бити негативан и остала два неће ни бити евалуирана.

break, continue

Ове две команде контролишу ток извршавања у петљи. `break` прекида извршавање блока петље на тој позицији и премешта извршење програма на прву команду након петље, дакле ово је превремени излаза из петље. `continue` слично `break` прекида извршавање блока петље на тој позицији, али не излази из петље, већ поново враћа ток на почетак петље, те започиње нову итерацију. У `for` петљи `continue` ће започети нови пролазак кроз петљу са наравно промењеном вредношћу бројача. Пример:

```
for i=1:10
  if i>3
     break
  else
     print(i)
print(i)
```

Овај програм исписује 1, 2, 3 и 4 где је вредност бројача и од четири условила да се изврши команда бреак и да се напусти петља, тако да 4 исписује прва команда након петље. И `break` и `continue` нису нужне језичке конструкције, могу се врло лако заменити са `if-else` питалицом, али врло често могу да поједноставе код, а самим тим и повећају његову разумљивост.

Функција која испитују постојање, иницијализовање променљивих – def

Ова функција `def`, се користи да би се испитало да ли је променљивој, матрици или варијабли у подацима, додељена вредност. Има само један аргумент, тј. саму матрицу, варијаблу чије се постојање испитује. Тако да `def(m)` враћа један ако је претхоно у коду прменљивој `m` додељена нека вредност односно нула ако ова променљива још није иницијализована, или у коду или кроз функцијски позив. Ово је начин како се проверава да ли је некој промељивој додељена вредност у позиву функције. Пример:

```
function y = funk(a, b) {
  if !def(b)
     y = a.^b
  else
     y = a.^2
}
```

Овако дефинисану функцију је могуће позивати и само са предатим једним аргументом, те ће у том случају функција квадрирати предату вредност. Док ако је аргумент `b` дефинисан функција враћати `b`-ти степен аргумента `a`.

Ова функција проверава и постојање варијабле у таблици података, тако код `def($pol)` проверава да ли у подацима постоји варијабла `pol`.

Функције за рад са подацима

Како је предложени језик намењен прављењу статистичких анализа над подацима, дефинисана је и структура која држи податке. Преузета је структура која се користи у SPSS-у јер од тренутно популарнијих статистичких пакета има највише мета информација, тј. информација које описују податке. Тако у предложеној структури поред података и назива варијабли ту су и:

- описи, лабеле варијабле тј. текстови који у краћем тексту описује варијаблу
- описи, лабеле појединих вредности тј. кодова варијабле
- дефиниција варијабле која пондерише податке
- дефиниције мултиплих варијабли, тј. варијабли које су направљене од више варијабли и које се посматрају као једна нпр. када је испитаник могао да наведе више одговора на неко питање
- дефиниције вредности које су проглашене за недеостајуће вредности

Подаци се налазе у специјалној матрици која се референцира са `$`. Ово није права

матрица већ пре колекција колонских вектора тј. варијабли. У правој матрици није могуће мешати нумеричке и стринговне податке, тј. матрица је или нумеричка или стринговна. Али је у `$` логичкој матрици могуће мешати типове података, тако да ако имамо неке варијабле које су нумеричке а неке које су стринг, следећа команда ствара грешку:

```
a = $;
```

У овој матрици испитаници се налазе у редовима, а варијабле у колонама, тако је `$[2,3]` у ствари податак за другог испитаника на трећој варијабли. Варијаблама можемо такође приступати и преко имена. Тако

```
$[3, "VAR1"]
```

означава вредност из трећег реда за варијаблу `VAR1`. Поред тога варијабла се може реферирати и преко префикса `$`, тако да следећи код је еквивалентан претходном, те такође претставља референцу на трећи елемент варијабле `VAR1`

```
$var1[3]
```

Уколико желимо да референцирамо комплетну варијаблу тј. вредности за све редове, све мерене објекте, онда можемо да користимо оператор :, тако да

```
$[:, "VAR1"]
```

представља вектор резултата свих објеката на варијабли `VAR1`. Ово је могуће краће записати реферирањем само једне варијабле преко оператора `$`, тако

```
$var1
```

уствари означава вектор варијабле која се зове `VAR1`, тј. представља такође све вредности измерених објеката на датој варијабли. Код референцирања варијабли преко имена нема разлике између малих и великих слова, ово је зато што је усвојен SPSS-ов модел података који не прави разлику између величине слова која се користе у имену варијабле.

Из ове псеудо матрице могуће је на исти начина извлачити вредности али јој и придавати као што је то случају код осталих матрица. Тако

```
$[3, "VAR1"] = 10
$var1[4] = 20
```

Прва линија означава да се објекту из трећег реда на варијабли `var1` додељује вредност десет, а друга линија да се објекту у четвртом реду додељује вредност двадесет.

Дефинисан је и пакет функција које су уграђене у језик које започињу са префиксом `data`. Ово је слично већ описисаном пакету `math`.

open

Ова функција служи за учитавање података из сачуваног фајла у SPSS формату (ови фајлови стандардно носе екстензију „sav“) у радну меморију рачунара. То значи да се сви подаци налазе у меморији, те им се јако брзо приступа, али наравно величина базе података на којој је могуће радити је ограничена величином радне меморије. Могуће је имати у једном тренутку само једне податке учитане у меморију. Други позив ове функције уствари ће прво ослободити податке учитане у првом позиву, а затим учитати податке специфициране у другом позиву ове функције. Захтевани аргумент ове функције је име датотеке у коме се налазе подаци које треба учитати. Функција има два аргумента `file`, који означава путању до датотеке са подацима, и `type` који представља тип података, за сада је подржан само један тип `sav` тј. `spss`, и то је уједно и стандардна вредност овог аргумента ако он није наведен. Једина повратна вредонст ове функције је код грешке. Пример:

```
data.open("c:/folder/podaci.sav")
```

close

За разлику од претходне ова функција брише, претходно учитане податке, из радне меморије. Ова функција нема аргумент.

save

Ова функција ће сачувати претходно учитане податке у датотеку. Функција се може позвати без аргумената и онда ће се подаци сачувати преко исте датотеке из које су и учитани, а ако је наведено име датотеке у позиву ове функције онда ће подаци бити у њој сачувани. Поред аргумента `file` и ова функција има и аргумент `type` који означава тип датотеке у коме треба сачувати податке.

```
data.save("radni_fajl.sav")
```

addVar

Функција која додаје нову варијаблу у структуру података. Има два стандардна аргумента `name` и `type`. Први означава име нове варијабе коју треба додати, а други њен тип. Аргумент имена мора поштовати сва правила о именовању варијабли, дакле неможе почињати са бројем и неможе у себи садржавати специјалне карактере какви су размак и карактери који су резервисани за аритметичке операције.

Аргумент `type` је стринг који одређује ког типа је нова варијабла, `numeric`, `string`, `multiple` и `multiple binary`. Ако није наведен, стандардно се узима прво наведени тип, тј. нумерички. Ако је наведена опција мултипле, која представља да је то уствари скуп варијабли који представљају одговоре на питање на које је било могуће дати више одговора, онда је потребно навести и трећи аргумент `variables`, који уствари специфицира имена варијабли које ће заједно представљати нову варијаблу, у којој сваки код представља један одговор који је испитанк могао да наведе.

Уколико нова „мултипла бинарна“ варијабла представља одговоре на више бинарних варијабли, онда је потребно навести и четврти аргумент, `values`, у коме

се наводе вредности које представљају одговоре који се рачунају као „да“ одговори. Ако заиста имамо бинарне варијабле онда је то вредност један.

Ако је специфицирана опција „`multiple`“ или „`multiple binary`“ онда је потребно навести само једно име за нову варијаблу и оно треба да има префих „`$`“.

Уколико је наведено „`multiple`“ и специфициран аргумент `values`, ново настала варијабла ће бити „`multiple binary`“ типа. А ако је наведено „`multiple binary`“, а аргумент `values` са вредностима није наведен, сматра се да је само вредност један одговор „да“.

Уколико је наведно да нова варијабла треба да буде типа „`numeric`“ или „`string`“, наведено име мора бити јединствено, тј. варијабла са тим именом не сме већ бити дефинисана. Уколико се прави нова „`multiple`“ или „`multiple binary`“ варијабла, како је то уствари само мета варијабла, конструкција од више обичних варијабли, ако је већ дефинисана под предложеним именом биће замењена са ново предложеном.

addCase

Функција која додаје нове (празне) редове у структуру података. Први аргумент, `number`, ове функције је број нових редова које треба додати, а други, `where`, ако је наведен, означава редни број реда испред кога треба додати нове редове. Ако други аргумент није наведен, нови редови се додају на крај структуре податаака.

delCase

Функција која брише наведене редове из структуре података. Редови се наводе преко њиховог редног броја, у аргументу `index`.

delVar

Функција која брише наведене варијабле из структуре података. Варијабле се наводе преко њиховог редног броја или имена.

getVarName

Функција која враћа име варијабле ако је она наведена преко њеног редног броја у структури података, тј. враћа имена варијабли варијабла које је сачињавају ако је наведена „мултипла“ варијабла. Функција има један аргумент са именом или позицијом варијабле који се зове `variable`.

setVarName

Функција која служи да се већ постојећој варијабли промени име. Функција захтева два стринг аргумента, истих димензија, матрицу са старим и новим именима, `variable` и `name`.

getVarLabel

Funkcija koja za navedene varijable vraća njihove labele. Argument je `variable`.

setVarLabel

Функција која служи да се дефинисаној варијабли додели нова лабела. Захтева два аргумента, са истим бројм елемената. Први аргумент, `variable`, означава варијаблу, а други, `label` њену нову лабелу.

getValLabel

Ова функција служи да се прочита лабела која је повезана са одређеном вредношћу на наведеној варијабли. Ако је наведен само један аргумент, `variable`, онда је то скалар који специфицира варијаблу било преко имена било преко редног броја, функција у том случају враћа две вредности вектор са вредностима и вектор стрингова са лабелама које су придружене са овим вредностима. Уколико функција има два аргумента онда је то име варијабле и вредност, `value`, а функција враћа њену лабелу.

setValLabel

Ова функција захтева три аргумента, `variable`, `value` и `label`, и служи за дефинисање лабела за вредности за дате варијабле. Први аргумент специфицира варијабле (било преко имена, било преко њеног редног броја), други аргумент је вектор вредности, а трећи је вектор стрингова који представљају лабеле наведених вредности. Ако је наведени стринг празан стринг тј. „“, онда се то уствари своди на брисање дефиниције вредност – лабела.

weight

Функција која пондерише податке са наведеном варијаблом. Једини аргумнет је `by`. Уколико аргумент није наведен функција брише претходно наведену дефиницију пондера, тј. подаци неће бити пондерисани.

filter

Функција која селектује само део редова, податак за анализу, на овнову вредности у наведеној варијабли. Једини аргумнет је `by`. Уколико аргумент није наведен функција брише претходно наведену дефиницију филтера, тј. подаци ће бити посматрани у целокупном обиму.

Константе у пакету дата

У овом пакету дефинисане је пет константи, којима се наравно не могу директно мењати вредности, сем одговарајућим функцијама из овог пакета.

ncase

Ово је вредност која означава број редова у табели података.

nvar

Ово је број варијабли, тј. колона у табели података. То је број само „`numeric`“ и „`string`“ варијабли, „`multiple`“ варијабле нису урачунате.

nmul

Ово је број дефинисаних мултиплих варијабли.

weightName

Име пондера ако су подаци пондерисани, тј. празан стринг у супротном случају.

filterName

Име филтера, тј. варијабле која служи као филтер, ако су подаци филтрирани, тј. празан стринг у супротном случају.

Имплементација језика

Програмски језик, је у ствари нотација рачунања која је разумљива и људима, корисницима, али и рачунарима. Коришћењем правила датог програмског језика пише се програм који представља низ смислених инструкција. Програм у том облику није погодан и за извршавање на рачунару, већ се скуп инструкција који су описани у програму морају превести у форму коју рачунар може да изврши. Генерално, програмски језик се састоји од "речи", којима се задају инструкције. Речи и њихов однос су стриктно дефинисани граматиком програмског језика, која у ствари омогућује то разумевање обадвеју страна, корисника и рачунара. Процес превођења написаног програма у форму коју рачунар може да изврши назива се компајлирање (енг. compile), ако је циљ да се добије посебна извршна датотека, датотека писана само машинским језиком, тј. интерпретирање ако је циљ да се код програма само изврши. Интерпретирање је различито од компајлирања јер се програмски код извршава инструкцију по инструкцију, једна инструкција се изкомпајлира, изврши па следећа итд. Код компајлирања програм се преведе у целини, и тек после тога извршава. Превођења написаног програма у програмском језику у форму коју рачунар може да изврши, започиње посматрањем написаног програма као низа карактера (слова, бројева и знакова) и дељењем тог низа на речи, симболе (енг. токен), овај процес извршава део који се назива лексички анализатор. Производ овог процеса више није низ карактера него низ речи. После тога се проверава синтаксичка структура кроз синтаксни анализатор чији је резултат „синтаксно дрво". Овај процес се обично назива парсовање програма и у ствари представља можда најважнији део у превођењу програма у машински код.

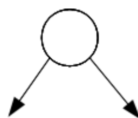
Ако језик подржава експлицитно дефинисање различитих структура, рецимо стрингова и бројева онда је могуће кроз семантички анализатор установити да ли постоје семантичке грешке, нпр. множење између два стринга, операција која стандардно није дефинисана ни у једном програмском језику. Уколико нема оваквих дефиниција, тада је ове грешке могуће идентификовати тек у фази

извршавања програма, што наравно успорава његово извшавање, јер се пре сваке извршена операције морају проверити њени аргументи.

Парсовање програма, тј. контрукција програмских језика је много лакша ако се језик заснива на формалној граматици, тако да се већина програмских језика заснива на контекстуално-слободној граматици (Hopecroft et al., 2001), јер је њихова конструкција тј. конструкција њихових парсера практично технички посао.

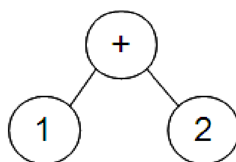
Суштина парсовања је уствари конструкција „синтаксног дрвета", уређена структуре, са јасно означеним почетком, која повезује операторе и њихове аргументе, и која се графички може представити преко дрвета (Aho 1986., Прешић 2005.). Конструкција дрвета није нужна, али га у ствари сваки парсер може направити.

Дрво је састављено од листова, где би лист био једноставна структура која садржи „главу", име, идентификатор и показивач на друга два листа, као што је приказано на следећем цртежу.



Слика 1. Градивни елемент синтаксног дрвета, са главом и два показивача

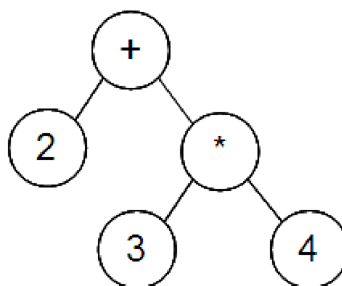
Тако би уствари операцију сабирања два броја, `1+2`, могли да преставимо следећим дрветом:



Слика 2. Синтаксно дрво којим је представљено сабирања два броја

где су изостављени показивачи са дрвета чија је вредност 1, тј. 2, јер они заправо представљају вредности, а не операције са аргументима, тј. њихови показивачи не показују ни на један други лист.

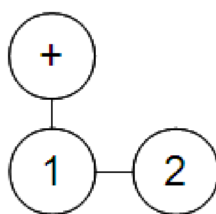
На сличан начин се може представити и дрво за операције различитог приоритета, нпр. израз `2 + 3 * 4`.



Слика 3. Синтаксно дрво за две операција (сабирања и множења: 2+3*4) различитог приоритета

На приказаном дрвету се јасно види редослед операција, сабира се вредност два са резултатом операција множења између вредности три и четири. Овакво дрво другим речима омогућава и једноставну евалуацију израза, кроз рекурзивно позивање једне те исте функције која на основу идентификатора и датих аргумената враћа вредност операције.

Предложена структура, са главом и са левим и десним показивачем (ЛД дрво), се у пракси замењује са еквивалентном структуром са главом, десним и доњим показивачем (ДД дрво), која је разумљивија када се желе представити не бинарне операције тј. функцијски позиви који могу имати арбитрарни број аргумената. Тако би већ наведена операција 1 + 2, била престављена са



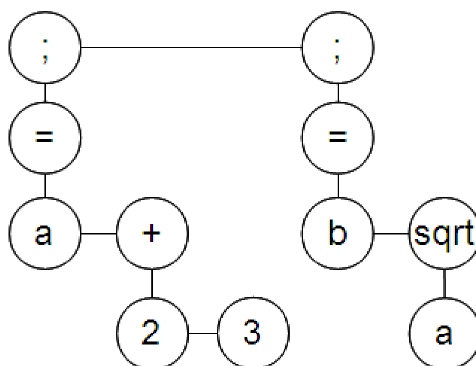
Слика 4. ДД дрво

што би се уствари свело да операција „+“ има два аргумента „1“ и „2“. Јасно је да уместо знака „+“ може да стоји било која бинарна операција, тј. било која функција која има два аргумента.

Ако би сада имали један једноставан програм од две линије који би симболичким језиком могли да преставимо са

```
a = 2 + 3
b = sqrt(a)
```

које би уствари означавале да у сиболичку променљиву `a` упсиујемо вредност операције сабирања између два и три, тј. где у другој линији у симболичку проенљиву `b` уписујемо вредност квадратног корена од вредности коју има симболичка променљива `a`, где смо унарну операцију квадратног корена обележили као функцију, која се зове `sqrt` и која прима један аргумент. Дрво овог програма би се могло графички представити овако

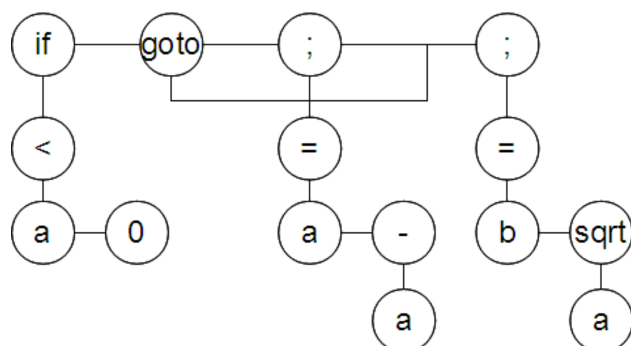


Слика 5. ДД дрво програма од две програмске линије са доделама вредности промељивима

где би дрво са ознаком „`;`“ уствари означавало једну линију програма, дрво са „`=`“ операцију доделе која има два аргумента: леви аргумент у који смештамо вредност десног аргумента. На сличан начин се може представити и стандардна `if-else` структура. Пример програма:

```
if a<0
  a = -a
b = sqrt(a)
```

који би у графичком представљању изгледао овако



Слика 6. ДД дрво са грањањем коришћењем `if` структуре

`if` питалица је уствари кључна реч `if` и иза ње бројчани, скалрни израз који се евалуира. Ако је он тачан, тј. ако је вредност израза различита од нуле, онда се извршава друга линија кода, а ако је нетачан, тј. једнак нули, онда се извршава наредба скока „`goto`“ која ток програма премешта на прву следећу линију после `if` питалице. На овај начин направљено је дрво које уствари представља линије кода програма, где се после извршења једне линије прелази на следећу, а свака линија се извршава рекурзивно.

Код прављења компајлера, следећа фаза би представљала прављење група појединачних једноставних инструкција за сваку линију кода, тј. линеаризацију направљеног дрвета, јер се у тој форми много лакше ради оптимизација кода, тј. замена више инструкција са њима еквивалнентим мањим бројем, које се због тога и извршавају брже.

Парсер је уствари алгоритам који од низа речи (токена) прави синтаксно стабло, које се после може рекурзивно извршавати. Лексички анализатор је задужен да од низа карактера који представља програм направи низ речи које ће служити персеру као улаз.

Лексички анализатор је уствари једноставан алгоритам који учитава карактер по карактер из текстуалног улаза који представља код програма и који одређује да ли учитани карактер представља комплетну реч или не. Стандардно размак представља завршетак једне речи, ако се пре њега такође није налазио размак. Генерално лексички анализатор мора бити у стању да гледа и у напред неколико карактера, јер рецимо карактер „<“ може представљати целу реч која представља релацију мање од, али ако се после њега налази карактер „=“ онда то представља реч „<=“ која уствари означава релацију мање или једнако. У сваком случају резултат је низ речи које треба склопити у структуру синтаксног дрвета.

При прављењу дрвета битно је обратити пажњу на приоритет операција, тј. на њихов редослед у извршавању, тако се израз `1+2*3`, евалуира као `1+(2*3)`, а не као `(1+2)*3` иако заграде нису наведене, што значи да парсер прво од речи „2“, „*“ и „3“ треба да направи дрво, а да затим од тог дрвета и речи „1“ и „+“ напави финално дрво, које представља израз `1+2*3`.

Поред овог правила битно је и правило „прве десне заграде“, које уствари значи да се низ речи од отворене заграде до прве затворене заграде прво трансформише у дрво те да се затим наставља парсовање остатка низа речи.

Симболичка табела

Сваки програм користи симоличке промењиве у које смешта вредности тренутних операција како би оптимизовао број операција које треба извршити, али и како би написани код био читљивији и лакши за одржавање. Тако следећи програм

```
a = 1 + 2 + 3
b = 4 + a
c = 4 * a
```

користи променљиву `a` у коју у првој линији смешта вредност збира три броја, док у другој и трећој линији позива ову вредност у изразима који формирају вредности које се смештају у прменљиве `b` и `c`.

Из тог разлога се формира именик свих променљивих које се користе и за њих се одређује, алоцира посебнио место у меморији рачунара пре почетка извршења функције којој припадају.

Овде је важно споменути и разлику између глобалних и локалних променљивих. Свака функција која се извршава има своју посебну симболичку табелу, и променљиве које се налазе у њој називају се локалне променљиве, тј. њихова вредност је видљива само коду у оквиру те функције. За друге функције та вредност је невидљива, или уопште немају промељиву са тим именом или истоимена промељива може имати било коју другу вредност. Глобалне променљиве су оне променљиве које су видљиве у оквиру свих функција и којима све функције могу приступати. Наравно дефинисање глобалне променљиве захтева специјалну команду која ће дефинисати да се променљива не тражи у локалној симболичкој табели већ у специјално одређеној табели коју зовемо глобална јер су у њој глобалне променљиве.

Функцијски позиви у оквиру програма преусмеравају ток извршења програма на код позиване функције, и у том случају се ствара нова симболичка табела у коју се уписују вредности које су прослеђене функцији. Замислимо да имамо дефинисану функцију `dabra` на следећи начин:

```
function y = dabra(x) {
  y = log(x) + x
  x = 33
}
```

коју би позвали са позивом из следећег кода

```
a = 10
dabra(a)
print a
```

У треунутку евалуације овог позива, биће створена нова меморијска табела за функцију `dabra` и у њу ће, на месту резервисаном за променљиву `x`, бити уписана вредност десет јер је то вредност промељиве `x` у тренутку позива функције `dabra`. Треба приметити да у овој табели неће бити променљиве `a`, тј. резервисаног места за њу, јер је промељива `a` дефинисана изван функције. У оквиру резервисаног меморијског простора за овај функцијски позив налазиће се и простор за вредност променљиве `x` из треће линије кода те функције. На крају функције овај меморијски простор, резервисан за овај функцијски позив, биће обрисан и извршење ће бити настављено, иза позива, дакле у трећем реду другог кода, а у оквиру меморијског простора који је био резервисан за овај код. На овај начин промена вредности променљиве `x` ни на који начин није утицала на остатак програма, тј. на вредност променљиве `a`, jer se te promene odvijaju u dva odvojena memorijska prostora. Ова врста позива функције се назива предавање аргумената функцији по вредности.

Други начин позива би био позивање на основу референце, која би уствари значила да се функцији не предаје копија вредности већ сама променљива, тј. њен простор у меморији, у описаном примеру простор који заузима променљива `a`, која ће се у оквиру функције дабра звати x. Самим тим ће промене на променљивој `x` у оквиру функције `dabra` остати трајне и након завршета те функције, тј. промениће и вредност промељиве `a`.

Пример

Најчешћи пример који се користи за представљање матричног језика је програм за примену мултипле линеарне регресије, ово због тога што је овај статистички модел јако познат али и због његове једноставности. Сам модел се може представити, ако не само у једној, онда у неколико матричних једначина. Пратећи ову традицију, овде ће бити наведен програм за примену мултипле регресије у предложеном језику, који је поједностављен и који пре свега служи као приказ предложеног језика.

```
01  function [b, beta, se, rsq,  rsq_adj, n, m] = linreg(x, y, variables,
dependent) {
02      if def(x) {
03          if !def(y) error("'y' argument is not deffined")
04          //deleting missing data
05          suma = sum(x, by="r")+y
06          indx = find(!isvalid(suma))
07          x[indx, ] = []
08          y[indx] = []
09      }
10      else {
11          variables = ucase(variables)
12          dependent = ucase(dependent)
13          [x, y, indx] = data.get(variables, dependent)
14      }
15      if nrow(x) < ncol(x)
16          error("There should be more rows then columns in indepedent x,
variables")
17      if nrow(x) != nrow(y)
18          error("Indepedent x and dependent y have different number of rows")
19      if ncol(y) != 1
20          error("Depedent variable 'y', should have just one column")
21
22      [n, m] = size(x);
23      x = [ones(nrow(x), 1), x]
```

```
24      xvar = var(x)'
25      yvar = var(y)
26      xxi = (x'*x)^
27      b = xxi * x'*y
28      beta = b.*math.sqrt(xvar / yvar)
29      beta = beta[2:NaN]
30      rsq = corr(y,x*b)^2
31      rsq_adj = 1 - (n-1)/(n-m-1)*(1-rsq)
32      se = math.sqrt(diag(xxi * var(y)*(1-rsq_adj)))
33      sig = 2* (1 - prob.tcdf(math.abs(b./se), make(size=size(b), value=n)))
34      stderr = math.sqrt(yvar*(1-rsq)*(n-1)/(n-m-1))
35
36      heading(1, "Regression")
37      if def(dependent) {
38          rlab = data.getVarLabel(dependent)
39          if rlab=="" rlab = dependent
40      }
41      heading(2, "on: "+rlab)
42
43      heading(2, "Model Summary")
44      clab = ["R","R Square","Adjusted R Square","Std. Error of the
Estimate"]
45      print([math.sqrt(rsq), rsq, rsq_adj, stderr], cname=clab,
format="%.3f")
46
47
48      heading(2, "ANOVA")
49      atbl = make(3,5,NaN)
50      totss = yvar * (n-1)
51      atbl[ ,1] =[totss*rsq; totss*(1-rsq);totss]
52      atbl[ ,2] = [m;n-m-1;n-1]
53      atbl[ ,3] = atbl[ ,1] ./ atbl[ ,2]
54      atbl[3,3] = NaN
55      atbl[1,4] = atbl[1,3]/atbl[2,3]
56      atbl[1,5] = 1 - prob.fcdf(atbl[1,4], m, n-m-1)
57
58      clab = ["Sum of Squares","df","Mean Square","F","Sig."]
59      rlab = ["Regression";"Residual";"Total"]
```

```
60      print(atbl, rlab, clab, format=["%.3f","%.0f","%.3f"])
61
62      heading(2, "Coefficients")
63      if def(variables)
64         rlab = data.getVarLabel(variables')
65      else
66         rlab = num2str("x%d",(1:m-1)')
67      rlab = ["Constant";rlab]
68
69      ctop = ["Unstandardized Coefficients","~","Beta","t", "Sig."]
70      cbot = make(value="|", size=size(ctop))
71      cbot[1:2]=["B","Std.Err."]
72      clab = [ctop;cbot]
73      clear ctop, cbot;
74
75      print([b,se,[NaN;beta],b./se, sig],rlab,clab, format="%.3f")
76   }
```

Програм се састоји од седамдесет и шест линија, у коме највећи део линија проверава улазне податке или форматира испис. Сам програм за рачунање модела мултипле регресије је написан у двадесет линија од 22. до 34. и од 50. до 56., што ако се изузму линије које форматирају код, излази да око једне трећине кода представља уствари рачунање модела.

Функција се зове `linreg` и може да прима четири аргумента, или аргументе `x` и `y`, који представљају матрицу независних и вектор зависних вредности, и у ком случају се регресија колонских вредности из `y` изводи на основу колонских вредности из `x`; или се као аргументи функције наводе имена независних варијабли у аргументу `variables` и име зависне варијабле у аргументу `dependent`. Линије од 3. до 9. се извршавају ако је наведен, дефинисан аргумент `x`, и у том случају проверавамо да ли је наведен и аргумент `y`, те ако није пријављујемо грешку (линија 3.). У петој линији сабирамо све колонске вредности из `x` и `y`, те затим у шестој линији тражимо све „невалидне“ вредности, тј. све вредности које су `NaN` или `Inf` тј. `-Inf`. Добијене индексе тих елемената у промењљивој `indx`, користимо како би смо обрисали редове у матрици `x`, тј. елементе у вектору `y`.

```
indx = find(!isvalid(suma))
x[indx, ] = []
y[indx] = []
```

Приметити да је брисање редова у матрици `x` извршено не навођењем колонског индекса, наведени су само индекси редова у променљивој `indx`, и запета.

У линији 13. ове две матрице смештамо податке из варијабли које су специфициране у друга два аргумента (`variables` и `dependent`).

15. до 20. линија представља проверу величина улазних матрица, број редова у матрици `x` несме бити мањи од броја колона, а мора бити једнак броју редова у матрици `y`, која при томе мора имати само једну колону.

Број испитаника, `n` и број независних варијабли `m` се рачунају у линији 22. После тога се додаје вектор једницина колонама у `x`, и рачунају њихове варијансе као и варијаса података из `y`. 26. линија рачуна инверз кроспродукта матрице `x` и смешта је у матрицу `xxi`, јер ће нам овај инверз бити потребан и код рачунања регресионих коефицијената, линија 27, и код рачунања њихових стандардних грешки, линија 32.

Стандардизовани (бета) регресиони коефицијенти се добијају у линији 28, али како су ове вредности један од излазних параметара, вектор бета се смањује за један елемент, тј. за први коефицијент који одговара константи у регресионом моделу (колону једница у независне колоне смо додали на прву позицију, линија 23), јер у стандардизовано облику овај аргумент нема пуно смисла, (у линији 75, овај ћемо елемент вратити назад као вредност `NaN`, ради форматирања исписа ових вредности у табели коефицијената).

Коефицијент детерминације и његова кориговани вредност се рачунају у линијама 30 и 31. У линији 32 се рачунају стандардене грешке регресионих коефицијената, а у 33. њихове значајности, користећи функцију `tcdf`, која се налази у директоријуму `prob` (а која овоом приликом није описана).

Од 36. линије креће испис података, прво је наслов анализе, а затим назив зависне

варијабле. Уколико зависна варијабла има дефинисану лабелу користи се она, а ако не онда њено име. После тога иде табела са њеним поднасловом, у којој су дати коефицијент мултипле корелације, коефицијент детерминације, његова коригована вредност и стандардна грешка процене. Од 48. до 56. линије се конструише табела анализе варијансе, са сумама квадрата, степенима слободе, просечним квадратима, F количником и његовом значајношћу. Ова табела се исписује у 60 линији, а у 58 и 59 се конструишу наслови колона и редова ове табеле.

Од 62 линије се исписује табела коефицијената и њихових значајности. Ова табела има две колоне „B“ и „Std.Err.“, које деле заједнички наднаслов, „Unstandardized Coefficients“. Да би се ово постигло наслов табела је уствари матрица стрингова са два реда, у првом реду су наслови колоноа, стим да наслов прве колоне треба да се простире и над другом. Зато је у другу колону уписана вредност „~“, која означава да ту ћелију треба спојити са претходном тј. оном која се налази лево од ње. У следећем реду ове матрице, налазе се поднаслови прве две колоне, односно текст „|” за све остале елементе, и она означава да дату ћелију треба спојити са претходном нагоре, тј. ћелијом изнад ње.

Испис

Показни програм за мултиплу регресију је извршен над подацима који су кориштени за пример овог статистичког модела студентима психологије (Тењовић, 2001). Описана функција `linreg` је позвана следећим кодом:

```
data.open("IZBISTRE.SAV")
linreg variables = ekstra e neurot/
       dep = msukup .
```

Добијен је следећи испис који се састоји од три табеле:

Regression

on: MISSISSIPPI SKALA-UKUPAN SKOR

Model Summary

R	R Square	Adjusted R Square	Std. Error of the Estimate
.648	.420	.405	14.805

ANOVA

	Sum of Squares	df	Mean Square	F	Sig.
Regression	17800.830	3	5933.610	27.072	.000
Residual	24547.886	112	219.178		
Total	42348.716	115			

Coefficients

	Unstandardized Coefficients		Beta	t	Sig.
	B	Std.Err.			
Constant	32.288	19.492		1.656	.100
EXTRAVERSION	-.150	.082	-.142	-1.800	.074
RELATIVNA UKUPNA IZLOŽENOST STRESORIMA	45.163	9.190	.354	4.915	.000
NEUROTICISM	.493	.079	.492	6.242	.000

Слика 7. Добијени испис из програма за мултиплу регресију

Добијене вредности одговарају вредностима наведеним у литератури који су добијени коришћењем програма SPSS. Ово наравно није потврда нумеричке тачности програма, већ само демонстрација да се користан програм може направити у осамдесетак линија кода, и то кода који је више мање транскипција матричних формула описаних у статистичкој литератури.

Закључак

Језик је, бар је то дуго психолозима јасно, мисаони алат. „Моћи решити“ је уствари исто као и „моћи представити“. Језик је мисаони алат који структурира знање. Програмски језик је све то само у још већој мери, јер је и сам више структуиран од природног језика, тако да је „моћи испрограмирати“ исто што и решити проблем користећи конструкте који постоје у програмском језику.

Програмирање, као структурирање постављеног проблема тј. као алгоритмизација његовог решења је заправо и разумевање проблема али и учење његовог решења.

Ово су можда најважнији разлози зашто по мишљењу аутора ове тезе психолози треба да се баве конструкцијом формалних језика. Предложени језик, са свим својим недостатцима и манама, је уствари тек демонстрација да је уз помоћ јако једноставних и основних знања програмирања могуће направити програмски језик, који може бити и основ за читав систем намењен некој области, у овом случају статистичкој анализи података.

У раду су представљена и четири статистичка система из реда најпопуларнијих. SPSS и SAS се заснивају на принципу кода који није доступан јавности. То су затворене програмске целине намењене појединим статистичким процедурама. Основна идеја иза ових система је примена већ постојећих метода на нове податке кроз добијање исписа резултата ових метода и њихово касније тумачење. Надограња и развој нових процедура је од секундарне важности у овој парадигми, и у суштини је ограничено само на конструктуре ових система. Друга два описана статистичка система R и STATA су заснована на идеји да корисник може и треба да развија своје статистичке процедуре, али и да исте дели са другима. У оба система постоји стандардизован начин прављења нових процедура преко програмског језика који је основни део система и који ово омогућава, али и протоколи који омогућавају да се новонаписане процедуре деле између корисника. Доступност кода омогућава и учење статистичке методе која је кодирана али у исти мах и развој нових метода, јер прављење новог се неретко заснива на унапређењу већ постојећег или на његовој надоградњи. Оно што је битно

напоменути, а што се одности на оба ова система је да поседују језике који имају врло архаичну синтаксу, те је трансфер са других програмских језика пре негативан, ако га уопште и има.

Друго важно питање по коме треба упоредити већ постојећа решења је и постајање корисничког интерфејса, комуникационог канала корисиника и система. Једино SPSS, од описаних система, има интерфејс који је заснован на Windows окружењу и који је вредан помена. Сви остали су ту компоненту ставили у други план, те на тај начин одредили своје примарне кориснике као напредне кориснике којима је програмирање познато. По мишљењу аутора ово је погрешна орјентација, јер корисник статистичких процедура не мора нужно бити и вичан програмирању, поготову је то случај у психологоји, где величина података тј. узорака није условљавала и напредно коришћење рачунара од стране истраживача. У психологији су подаци са којима се уобичајно ради релативно мали у осносу на могућности рачунара. У другим наукама које користе статистику то није тако, типичан пример би био демографија где су стандарадно велики узорци (па све до нивоа пописа). На овај начин је постојање корисничког интерфејса, али таквог којег је могуће дизајнирати у систему, уствари један од предуслова за популарност система. И од свих описаних SPSS је далеко најпопуларнији, аутор би рекао управо из овог разлога.

Отворененост кода, и његова модуларност, на основу које је могуће даље слагати нове модуле је једна од најважнијих одлика предложеног система. Језик који је довољно једноставан, да буде разумљив и статистичару почетнику, али и довољно свеобухватан да се са њим могу решити реални проблеми је функција коју је аутор покушавао да екстремизира у овом раду кроз предлог језика који би требао да буде у основи новог статистичког система. Питање да ли је успео у овоме, се наравно ни не поставља, јер је у програмирању, бар до сада одговор увек био одричан. Конкретан језик је увек био тек једна итерација у решавању овог проблема. Оно што је битно су идеје, концепти које је неки језик створио и за собом оставио, а тај се суд може донети само из историјске перспективе.

Литература

Abelson, H., Sussman, G. & Sussman, J. (1996). Structure and Interpretation of Computer Programs. Cambridge: MIT Press.

Aho, A. V. (2007). Compilers, principles, techniques, & tools. (2nd ed. ed.). Addison-Wesley.

Altman, M., & McDonald, M. P. (2001). Choosing Reliable Statistical Software. Political Science and Politics, 34 (3), 681-687.

Chambers, J. (2000). Users, Programmers, and Statistical Software. Journal of Computational and Graphical Statistics, 9 (3), 404-422.

Everitt, B. S., & Rabe-Hesketh, S. (2004). Handbook of statistical analyses using STATA, third edition. Chapman And Hall/CRC.

Gentleman, R. (2004). Some Perspectives on Statistical Computing. The Canadian Journal of Statistics, 32 (3), 209-226.

Hopcroft, J. (2001). Introduction to automata theory, languages, and computation. Boston: Addison-Wesley.

Huber, P. (2000). Languages for Statistics and Data Analysis. Journal of Computational and Graphical Statistics, 9 (3), 600-620.

Knuth, D. (1973). The art of computer programming Vol. 3, Sorting and Searching. Reading, Mass: Addison-Wesley Pub. Co.

Lang, D. T. (2000). The Omegahat Environment: New Possibilities for Statistical Computing. Journal of Computational and Graphical Statistics, 9 (3), 423-451 .

Leeuw, J. (2005). On Abandoning XLISP-STAT. Journal of Statistical Software, 13 (7), 1-5.

Mahalanobis, P. C. (1965). Statistics as a Key Technology. The American Statistician, 19 (2), 43-46.

McCullough, B. D. (1998). Assessing the Reliability of Statistical Software: Part I. The American Statistician, 52 (4), 358-366.

Prešić, S. B. (2005). Algoritmika, 1 (niska algoritama značajnih i onih težih). Beograd: Kolorton.

SAS Institute Inc. (2004). SAS/IML 9.1 user's guide. Cary, N.C: SAS Pub.

STATACorp. (2009). STATA user's guide: release 11. College Station, Tex: STATACorp LP.

Stromberg, A. (2004). Why Write Statistical Software? The Case of Robust Statistical Methods. Journal of Statistical Software, 10 (5), 1-8.

Tenjović, L. (2001). Statistika u prihologiji: priručnik. Beograd: Centar za primenjenu psihologiju.

Tierney, L. (2005). Some Notes on the Past and Future of Lisp-Stat. Journal of Statistical Software, 13 (9), 1-15.

Valero-Mora, P., & Udina, F. (2005). The Health of Lisp-Stat, Journal of Statistical Software, 13 (10), 1-5.

Weisberg S. (2005). Lost Opportunities: Why We Need a Variety of Statistical Languages, Journal of Statistical Software, 13 (1), 1-12.

Wetherill, G. B., & Curram, J. B. (1985). The Design and Evaluation of Statistical Software for Microcomputers. The Statistician, 34 (4), 391-427.

Yeo, G.K. (1984). A Note of Caution on Using Statistical Software. The Statistician, 33 (2), 181-184.

Биографија

Рођен 1973. године у Београду. Матурирао 1992. у Математичкој гимназији. Студирао Машински факултет у Београду од 1992 до 1993, када се уписао на одељење за психологију на Филозофском факултету у Београду. Због исказаног успеха у дотадашњем школовању, од октобра 1997. био је стипендиста Републичког фонда за развој научног и уметничког подмлатка, а од марта 1998. до дипломирања стипендиста Српске академије наука и уметности.

Дипломирао на Одељењу за психологији Филозофског факултета у Београду 1998. године са просечном оценом 9.19. Тема дипломског рада била је "Факторска анализа бинарних варијабли у стандардизованом имаж простору”. 1999 године примљен је на одељење као асистент – приправник на предмету Статистика у психологији. Исте те године уписује и постдипломске студије на смеру за Квантитативне методе. Магистрирао 2006 са темом „Проверавање структуралних хипотеза конфирмативним методом канoничке дискриминативне анализе“.

Прилог 1.

Изјава о ауторству

Потписани-а Александар Зорић

број уписа ______________________________

Изјављујем

да је докторска дисертација под насловом

„Нови приступ статистичкој анализи података у психологији: од ригидних статистичких пакета ка флексибилном систему“

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 17. јула 2012

____

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора: Александар Зорић

Број уписа ____

Студијски програм: Психологија

Наслов рада: Нови приступ статистичкој анализи података у психологији: од ригидних статистичких пакета ка флексибилном систему

Ментор ____

Потписани Александар Зорић

изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу Дигиталног репозиторијума Универзитета у Београду.

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 17. јула 2012

____

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

„Нови приступ статистичкој анализи података у психологији: од ригидних статистичких пакета ка флексибилном систему“

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Цреативе Цоммонс) за коју сам се одлучио/ла.

1. Ауторство
2. Ауторство - некомерцијално
3. Ауторство – некомерцијално – без прераде
4. Ауторство – некомерцијално – делити под истим условима
5. Ауторство – без прераде
6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 17. јула 2012

____