MaMa specifikacija

Vedran Novaković & Marko Doko

Sadržaj

1 uvod

1.1 o dokumentu

Ovaj dokument opisuje **MaMa** softverski paket *krajnjim korisnicima*, kao priručnik i referenca ujedno. Pojedine su sekcije nužne i *razvijateljima*.

1.1.1 distribucija

Distribuiranje ovog dokumenta dozvoljeno je, sa ili bez izmjena i u bilo kojem obliku, pod uvjetom da sve izmjene budu nedvosmisleno naznačene kao takve.

1.1.2 predznanje

Potrebno predznanje za čitanje ovog dokumenta i uporabu njime opisanog softvera uključuje elementarne informacije o računalnim arhitekturama, matematičkoj logici i korištenju programskih prevodioca te interpretera.

1.2 sažetak

MaMa (**Ma**cro **Ma**chine) softverski je paket namijenjen simulaciji RAM strojeva obogaćenih makroima (tzv. makro strojeva). Sačinjavaju ga:

1. kompajler u bytecode (`MaMaC`)
2. bytecode interpreter (`MaMa`)
3. GUI (rudimentarni IDE) (`MaMut`)
4. biblioteka standardnih makroa (`BSO`)
5. prateća dokumentacija i primjeri

MaMa je pisana s namjerom da bude jednostavna, bogata mogućnostima, razumno portabilna te iznimno efikasna implementacija makro stroja.

1.3 uvjeti korištenja

Uvjeti korištenja detaljnije su opisani u dokumentu `LICENSE.txt` iz paketa.

2 RAM stroj

RAM stroj idealizirani je model računala s pohranjenim programom. Svaki je RAM program konačan neprazan niz RAM instrukcija. Instrukcije su numerirane slijedno, počev od 0. RAM stroj posjeduje registar-brojač, koji sadrži adresu instrukcije koja se ima sljedeća izvršiti, i niz od prebrojivo mnogo registara, numeriranih slijedno od 0, koji imaju direktan pristup i sadrže proizvoljno velike prirodne brojeve.

Model RAM stroja realiziran u **MaMa** paketu ima konačno mnogo registara, ostvarenih nepredznačenim 64-bitnim integerima, pa postoji (zanemariva) mogućnost overflowa.

Dozvolimo li da jedan RAM program pozove drugi kao subrutinu u rješavanju nekog većeg problema, tada kažemo da imamo **makro stroj**. **MaMa** je (u svim definicijama konačna) implementacija tog koncepta.

Bitno je napomenuti da makro "ne vidi" RAM program koji ga poziva, tako da sve naredbe koje uključuju eksplicitno mijenjanje registra brojača moraju biti realizirane kao instrukcije stroja, a ne makroi.

2.1 RAM program

U matematičkoj su se teoriji iskristalizirala dva sistema instrukcija RAM programa, za koje će biti pokazano da su jednake u snazi (jedne izrazive pomoću drugih i obratno).

2.1.1 inkrement

Oba sistema imaju zajedničku instrukciju `INC` i koja povećava sadržaj registra i za 1 i prelazi na sljedeću po redu instrukciju.

2.1.2 `SYSTEM 2`

U tom sistemu postoji još samo jedna naredba, `DEC` i k koja smanjuje sadržaj registra i ako je on veći od 0 i postavlja u registar-brojač vrijednost k (skače na labelu k). Ako je vrijednost u registru i jednaka 0, tada se ona ne mijenja i prelazi se na sljedeću po redu instrukciju.

2.1.3 SYSTEM 3

U tom sistemu postoje još dvije naredbe, `DEC` i k te `GOTO` k. `DEC` i k smanjuje sadržaj registra i ako je on veći od 0 i prelazi na sljedeću po redu instrukciju, u suprotnom ne mijenja sadržaj registra i nego u registar-brojač pohranjuje vrijednost k. `GOTO` k bezuvjetno u registar-brojač pohranjuje vrijednost k ne mijenjajući sadržaj niti jednog registra.

2.1.4 napomene

Dogovorno ćemo u sistemu 2 s `DEK` označavati `DEC` iz sistema 3 i obratno. Čitatelju za vježbu ostavljamo da raspiše dekremente iz jednog sistema u drugome i time opravda ovu konvenciju. Također, pokazati da je `GOTO` trivijalno izvediv u sistemu 2, pa time smatramo da i ta naredba u rečenom sistemu postoji kao pokrata.

2.2 rad RAM stroja

2.2.1 inicijalizacija

Prije početka rada RAM stroja, u registre 1 do k spremaju se ulazni podaci RAM programa ($k = 0$ povlači da program nema input).

Smatramo da prije unosa inputa, dakle prilikom "stvaranja" RAM stroja, svi registri, uključivo i registar-brojač, imaju početnu vrijednost 0.

2.2.2 start & stop

RAM program počinje izvršavanje instrukcijom na labeli 0, a staje kada se u registru-brojaču tijekom *automatskog* inkrementiranja nađe labela koja je neposredni sljedbenik najveće labele programa, označimo je s n (takva labela postoji jer je svaki program konačan). Pažnja: *nije* dozvoljen izlaz skokom na nepostojeću labelu programa (što neke teorije dopuštaju, a neke ne).

Iznimka od rečenog su naredbe `GOTO ::` i $\texttt{DE}_{\texttt{C}}^{\texttt{K}}$ *registar* `::` koje smatraju da na labeli n piše naredba `INC %%` a na $n+1$ $\texttt{DEC}_3$ `%% :!` pa skokom na n-tu labelu omogućavaju pravilan prekid rada, koji ne promijeni vrijednost niti jednog registra.

Uočiti da je svaki program konačan, pa tako postoji i najmanji $m \in \mathbb{N}$ takav da se svi registri od uključivo m-tog nadalje u programu ne koriste.

RAM program ne mora nikad stati. Dozvoljene su ∞ petlje (i rekurzivni pozivi makroa, ali to je specifičnost paketa **MaMa**, jer neke teorije to ne dopuštaju). Ako RAM stroj, jednom pokrenut, ikada stane, iz registra 0 iščitamo rezultat izračunavanja.

2.2.3 taktovi stroja

Rad RAM stroja odvija se u taktovima. U svakom se taktu izvrši jedna i samo jedna instrukcija stroja (ako je riječ o pozivu makroa, izvrši se točno taj makro poziv, ali u stvari čitav pozvani makro program).

Stroj započinje radom na nultoj instrukciji RAM programa, pročitavši u registru-brojaču početnu vrijednost 0. Tijekom rada stroj pročita vrijednost registra-brojača, recimo r, i skoči na r-tu liniju programa. Prije izvršavanja novodohvaćene instrukcije vrijednost registra-brojača poveća se za 1. Neke instrukcije mogu svojim izvršavanjem promijeniti vrijednost registra-brojača, kako je prije opisano, ali je ne mogu izravno saznati.

3 MaMa

U ovoj ćemo sekciji opisati apstraktni **MaMa** sustav. Svaki sustav koji odgovara neformalnim specifikacijama iz ove sekcije, s eventualnim kompatibilnim nadogradnjama, jest **MaMa** sustav, kojega je naš softverski paket tek jedna moguća implementacija.

Svaki se **MaMa** sustav sastoji od bytecode kompajlera izvornog **MaMa** programa, te od interpretera tog bytecodea. Ovdje ćemo se baviti strukturom izvornog programa i bytecodea. U kasnijim ćemo odjeljcima reći nešto više o samim softverskim komponentama.

3.1 MaMa program verzija 0

Dajemo precizne, ali neformalne sintaktičke (i dijelom semantičke) definicije elemenata **MaMa** programa.

3.1.1 neformalna sintaksa programa

MaMa program neprazna je tekstualna ASCII datoteka (podrška za Unicode nije u planu) duljine najviše 2 GB, s obaveznom ekstenzijom `.MMs`[1] i elementima niže opisanima:

- Program je neprazan niz direktiva i instrukcija.
- Sve su naredbe i direktive *case-insensitive*.
- Argumenti instrukcija i direktiva odvojeni su od instrukcije (direktive) i od drugih argumenata barem jednom bjelinom, ako nije određeno drugačije.
- Svaka instrukcija ili direktiva mora početi i završiti unutar iste linije.
- Komentari počinju od prvog pojavljivanja znaka # u liniji i traju do kraja te linije. Sadržaj komentara pri kompilaciji se ignorira.
- Linija je ili labelirana ili nelabelirana. Nelabelirana linija koja sadrži instrukciju ili direktivu počinje s 0 ili više bjelina, te potom instrukcijom (direktivom), uz opcionalne bjeline i/ili komentar na kraju. Nelabelirane linije koje sadrže samo 0 ili više bjelina i/ili komentar broje se, ali ignoriraju prilikom kompilacije.

[1] **MaMa** source

- Linija može biti labelirana ako i samo ako sadrži instrukciju. Labelirana linija počinje opcionalnim bjelinama, zatim slijede labela, barem jedna bjelina i potom neka instrukcija, uz opcionalne bjeline i/ili komentar na kraju.

3.1.2 neformalna sintaksa direktiva

Direktive su uputstva kompajleru. Zasad postoje dvije direktive. Svaka direktiva mora doći prije bilo koje instrukcije i smije se pojaviti najviše jednom u tijelu programa. Relativan poredak direktiva je nebitan.

- `SYSTEM` 2 odnosno `SYSTEM` 3 implicira semantiku naredbi kako je to za rečene sisteme gore opisano. Ova direktiva je *obavezna*!
- `VERSION` i, $0 \leq i \leq 15$, zahtijeva generiranje bytecodea verzije i. Ako nije prisutna, podrazumijeva se zadnja podržana verzija bytecodea.

3.1.3 neformalna sintaksa instrukcija

Postoji šest instrukcija **MaMa** programa:

- `INC` *registar*
- `DEC` *registar labela*
- `DEK` *registar labela*
- `SET` *registar registar* $\vee$ *konstanta* [2]
- `GOTO` *labela*
- `CALL` *ime_MaMa_programa* $R_0\ R_1 \cdots R_k$

U svakoj instrukciji pod varijablom *registar* smatramo jedno od:

- `%i` $i \in \mathbb{N}$, $0 \leq i < m < 2^{28} - 1$ indeks RAM registra kao cjelobrojna konstanta jezika C u bazi 8, 10 ili 16 *bez sufiksa*.
- `%%` indeks m-tog (pomoćnog) registra, kako je već objašnjeno.

[2] `SET` je redundantna instrukcija, vidi odjeljak 3.1.6 !

Pod pojmom *konstanta* smatramo 64-bitne nepredznačene cjelobrojne konstante jezika C u bazi 8, 10 ili 16 *bez sufiksa*.

U programima nije moguće adresirati u instrukcijama za skok izravno, brojem programske linije! Taj je princip podložan greškama i vrlo nespretan zbog neizvršivih linija. Za adresiranje služi *labela* nekog od ova tri tipa:

- `:`*`label`* *`label`* je neprazan niz grafičkih karaktera[3] i predstavlja simboličko ime linije na čijem se početku pojavljuje. Ne mogu postojati dvije različite linije s istom labelom!
- `::` n-ta linija (prva iza zadnje).
- `:!` trenutna linija na kojoj se ova labela javlja. Idealno za ∞ i one-liner petlje `GOTO :!` ili $\texttt{DE}_{\texttt{C}}^{\texttt{K}}$ *registar* `:!` oblika.

3.1.4 tips & tricks

Iako se možda čini nepotpunim rješenjem nemogućnost imenovanja registara suvislim mnemonicima, to je i dalje moguće učiniti! Dovoljan je neki (npr. C) pretprocesor i nešto u duhu sljedeće sekvence na početku predloška vašeg budućeg **MaMa** programa:

```
#define ZBROJ    %7
#define REZULTAT %0
#define TEMP     %%
```

Propustite li vaš predložak kroz pretprocesor, dobit ćete na izlazu sva pojavljivanja simboličkih imena zamijenjena odgovarajućim registrima, odnosno kompajleru prihvatljiv program.

Također, primijetite da ne postoji naredba `HALT` ili neka sličnog imena što bi zaustavila rad stroja. Mnoge teorije takvu naredbu ne dozvoljavaju, a ona je ekvivalentna s `GOTO ::` pa je nije potrebno uvoditi.

3.1.5 `CALL` semantika

`CALL` instrukcija izvršava proizvoljni **MaMa** program u makro kontekstu. Kako je riječ o netrivijalnoj operaciji, moli se čitatelj da prouči okvirni dizajn čitavog **MaMa** sustava u sekciji 4 prije nastavka ovog dijela.

[3] po ISO C `isgraph()` klasifikaciji

"`CALL` *`ime_MaMa_programa`* $R_0\ R_1 \cdots R_k$" stvara novu instancu interpretera i poziva je u makro kontekstu uz sljedeće vezanje registara (pod terminom *program* smatramo pozivajući, a pod *makro* pozvani program):

makro	**program**	**ograničenja**
%0	R_0	R_0 je *registar* i obavezan parametar
%1	R_1	$R_1 \cdots R_k$ je niz od 0 ili više članova i $\forall 1 \leq j \leq k$
$\cdots_m$	$\cdots_p$	R_j je *registar* ili nenegativna cjelobrojna konstanta
%k	R_k	jezika C u bazi 8, 10 ili 16, strogo manja od 2^{64}

Pojam *vezanje registara* označava pojavu da pozvani makro ima `čitaj` i `piši` pristup (odgovarajućim pozivom proslijeđenim) registrima pozivajućeg programa, kako je tablicom navedeno, preko indeksa nekog svog registra. Pozivatelj i makro dijele one i samo one registre koji su eksplicitno ovakvim pozivom vezani. Primijetiti da je vezanje registara inherentno rekurzivno, tj. moguće je proizvoljno mnogo nivoa indirekcije. Također, važno je znati da je tijekom izvršavanja proizvoljnog, ali danog makro programa privremeni registar (%%) uvijek lokalan za taj program.

Preostala mogućnost je da se neki registri pozvanog makroa pune proslijeđenim konstantnim vrijednostima. To je ujedno i jedina mogućnost ukoliko korisnik stvara prvu instancu interpretera i zahtijeva od nje izvođenje nekog **MaMa** programa. O tome više u opisu interpretera.

ime_MaMa_programa ne mora biti prisutno. Ako nije navedeno, podrazumijeva se rekurzivni poziv tekućeg programa. Ako jest, podvrgava se identičnom procesu kakav je opisan za istoimeni interpreterov argument.

Opisani model makro poziva posve je analogan pozivu procedure (tj. subrutine) u nekom programskom jeziku koji ima prijenos argumenata po referenci (tzv. "pass by reference"), kao što su npr. BASIC ili FORTRAN, s time da smo se dogovorili da ćemo za svaku subrutinu uvijek imati jedan (i to prvi) argument namijenjen samo za povratnu vrijednost.

Sve što makro program radi vezanim registrima, u biti čini korespondentnim registrima pozivajućeg programa (odnosno rekurzivno unatrag po stablu poziva), pa se savjetuje *izniman oprez* ako se isti registar pozivatelja veže na više registara makroa. Iako je takav kôd nečitak i podložan greškama, njegova je semantika jednoznačno određena jer je RAM stroj sekvencijalan.

Od svakog se *pristojnog* **MaMa** programa ipak zahtijeva da na kraju izvršavanja nije promijenjena početna vrijednost niti jednog vezanog ne-izlaznog (ne-%0) registra, osim ako to nije njegova standardna semantika.

Programi koji taj zahtjev ne poštuju ne mogu biti dijelom biblioteke standardnih makroa (`BSO`) i trebali bi detaljno obavijestiti korisnika o:

- broju formalnih parametara
- dozvoljenim vrijednostima stvarnih parametara
- kada se i kako kojem ulaznom vezanom registru može nepovratno promijeniti početna vrijednost

3.1.6 primjeri MaMa programa

Zavirite u poddirektorij `lib` za real-life primjere **MaMa** programa. Isti se nalaze u datotekama s ekstenzijom `.MMs` kao dio `BSO` biblioteke ili kao dodaci.

Npr. moguće je napisati program `SET`, koji će služiti kao makro što postavlja arbitrarni registar na arbitrarnu konstantu ili vrijednost nekog drugog registra, iako je kod nas minimalizam ustupio mjesto efikasnosti ugrađene komande. Pogledajte na rečeno mjesto za dokaz da dodavanjem te komande u jezik nismo nimalo povećali njegovu prijašnju snagu.

3.2 MaMa bytecode verzija 0

Bytecode je platformski neovisna binarna reprezentacija **MaMa** programa, u fajlu s obaveznom ekstenzijom `.MMb`[4] i duljine najviše `2 GB`, koju izvršava **MaMa** interpreter. Bytecode koristi *nativni* byte-ordering računala na kojemu je kreiran, što je naznačeno u deskriptoru i to interpreter mora uvažiti pri dohvatu cjelobrojnih podataka.

U binarnom prikazu cjelobrojnih riječi koristit ćemo konvenciju da su slijeva nadesno bitovi navedeni od najviše do najmanje značajnog.

3.2.1 deskriptor

Deskriptor bytecodea početni je dio bytecodea, duljine 64 bita, i formiran je kao $4 + 4$ okteta (oktet = 8 bitova) kako slijedi:

```
0 ESDi iiii  iiii iiii  iiii iiii  iiii iiii

4 VVVV rrrr  rrrr rrrr  rrrr rrrr  rrrr rrrr
```

[4] **MaMa** bytecode

pri čemu je riječ od prvih 4 okteta spremljena u little-endian byte orderingu, a značenje pojedinih polja bitova je ovakvo:

- `E` 1 akko je nativni big-endian byte-ordering, 0 akko je to little-endian
- `S` 1 akko je `SYSTEM 3`, 0 akko je `SYSTEM 2`
- `D` 1 akko su pristutne debugging informacije u vidu oznaka linija izvornog kôda, 0 akko je bytecode namijenjen za neinteraktivno izvršavanje
- `i` 29 `i`-bitova je broj instrukcija bytecodea
- `V` $(\texttt{0000})_2 \leq \texttt{VVVV} \leq (\texttt{1111})_2$ verzija bytecodea
- `r` 28 `r`-bitova broj je registara, ne uključujući `%%`

Svi se ostali podaci mogu dobiti od sistema (ukupna veličina bytecode fajla) ili izračunati na temelju ovih. Razlog za fiksiranjem byte orderinga prve 32-bitne riječi je taj što interpreter mora odmah ustanoviti byte-ordering bytecodea, a to može akko se zna točna lokacija na kojoj taj podatak piše, što je najlakše postići ako se zahtijeva proizvoljni, ali točno određeni byte ordering (little-endian odabran je kao prevladavajući na PC platformama).

Ovisno o vrijednosti `D` flaga, iza deskriptora slijedi instrukcijski blok (`D` $= 0$) ili niz od I debugging zapisa (`D` $= 1$), gdje je I broj instrukcija bytecodea.

3.2.2 debugging zapisi

Svaki debugging zapis je sastavljen od $4 + 4 = 8$ okteta. Prva 4 okteta čine file pointer na korespodentnu instrukciju, a druga 4 okteta su broj linije, počev od 1, izvornog fajla koja je tu instrukciju producirala. Oba dijela imaju vodeći bit 0.

3.2.3 instrukcijski blok

Instrukcijski blok je niz svih instrukcija bytecodea. Instrukcijski blok slijedi iza niza debugging zapisa (ako ih ima), odnosno iza deskriptora.

Svaka instrukcija ima zaglavlje (*header*) instrukcije [`4 okteta`]. Vodeća 4 bita instrukcijskog headera odlučuju točno jednu od instrukcija ovako:

`GOTO:` `1??? ???? ???? ???? ???? ???? ???? ????`
31 `?`-bitova je adresa bezuvjetnog skoka (file pointer na instrukcijski zapis).

`CALL`: `010d dddd dddd dddd dddd dddd pppp pppp`
Iza headera slijedi `CALL` blok, čije je ustrojstvo, kao i značenje pojedinih polja bitova u headeru opisano niže u zasebnom odjeljku.

`SET`: `011C ???? ???? ???? ???? ???? ???? ????`
Bit `C` je 1 ako iza headera slijedi 64-bitna konstanta, inače je 0 te iza headera slijedi 32-bitni kôd registra (na donjih 28 bitova, ostalo su nule). Preostalih 28 bitova headera kôd su registra kojeg treba setirati.

$\texttt{DE}_{\texttt{C}}^{\texttt{K}}$: $\texttt{001}_{\texttt{0}}^{\texttt{1}}$ `???? ???? ???? ???? ???? ???? ????`
Preostalih 28 bitova adresa su registra na kojemu $\texttt{DE}_{\texttt{C}}^{\texttt{K}}$ djeluje. Neposredno iza ovog headera slijedi riječ od 4 okteta kao file pointer na instrukcijski header cilja uvjetnog skoka. Vodeći bit te riječi ima jednaku vrijednost kao `S` bit deskriptora (semantika instrukcije potpuno je sadržana u njenom kôdu).

`INC`: `0001 ???? ???? ???? ???? ???? ???? ????`
Preostalih 28 bitova čine adresu registra na kojemu `INC` djeluje.

`rezervirano`: `0000 ???? ???? ???? ???? ???? ???? ????`
Ovo je rezervirani instrukcijski header. Uzrokuje run-time error pri interpretiranju. Namijenjen je budućim proširenjima **MaMa** jezika.

3.2.4 `CALL` detalji

Značenje pojedinih polja bitova u headeru `CALL` insrukcije je sljedeće:

- `P` duljina imena programa, može biti 0 (8 bitova)
- `D` ukupna duljina `CALL` bloka nakon imena programa (21 bit)

`CALL` blok slijedi neposredno iza headera pripadne instrukcije i sastoji se od:

1. opcionalnog imena programa (makroa), bez terminatora
2. jednog ili više zapisâ, od kojih je svaki ili 32-bitni kôd registra ili 64-bitna konstanta. Svakom zapisu prethodi jedan oktet, koji je 0 ako iza njega slijedi kôd registra, odnosno 1 ako slijedi konstanta

Razlog: zbog različitih duljina binarnih prikaza dvaju tipova argumenata (kôd registra ili konstanta) nužno je naznačiti unaprijed što se mora iščitati.

3.2.5 napomene

Prilikom kompilacije sve se labele prevode na apsolutne file pointere. Labelama `::` i `:!` pridružuju se redom pointeri $(\texttt{7F FF FF FF})_{16}$ i $(\texttt{00 00 00 00})_{16}$, koji pokazuju na zadnji mogući, odnosno nulti byte bytecodea. Kako na tim adresama ne mogu biti instrukcijski headeri (jer je adresa prevelika za 4-oktetni header ili pokazuje na deskriptor) to je interpreteru jasno da se ne radi o validnim pointerima, već oznakama za posljednju, odnosno tekuću liniju programskog kôda.

Štoviše, nije potrebno niti emitirati n-tu i $(n{+}1)$-vu instrukciju kad je njihova jedina svrha omogućiti legalan izlaz iz programa ne mijenjajući vrijednost niti jednog registra. Interpreter je slobodan pri "skoku" na file pointer koji kodira labelu `::` jednostavno prekinuti izvođenje **MaMa** programa.

Prilikom kompilacije računaju se translacijske tablice za registre. To znači da se smanjuju memorijski zahtjevi (i sprječavaju zlonamjerni useri da unesreće sustav nečim tipa "`CALL rekurzija %100000000`") tako da se:

1 zapamte se 0 i svi korišteni indeksi registara

2 neka su i, j zapamćeni indeksi, k proizvoljan, tada vrijedi:
$\big[(i < k < j \Rightarrow k \text{ nije zapamćen}) \ \wedge \ (j - i > 1)\big] \implies j = i + 1$

3 ponavlja se postupak pod **2**

Ovo je samo kriptičan zapis onoga što se trivijalno postiže sortiranim asocijativnim poljem u kojemu se nalazi 0 i pri nailasku na njih ubacuju ostali registri kao ključ, te se potom slijedno, počev od 0, asociraju stare vrijednosti s novom, svaki put za 1 većom od prethodne.

Zbog toga je neobično važno sve ulazne registre držati kontinuiranim (npr. od 1 do k, kako i definicija RAM stroja nalaže), i svaki spomenuti barem jednom u tijelu programa!

Pomoćni registar `%%` u instrukcijske se headere i `CALL` blokove zapisuje kao binarni prikaz broja $(\texttt{0F FF FF FF})_{16} = 2^{28} - 1$. Interpreter koristi informaciju iz deskriptora o stvarnom broju registara kako bi tu vrijednost tijekom izvođenja pridijelio ovim dvama oznakama.

Cjelokupna struktura debugging bytecodea prikaziva je kao:

1. deskriptor [8 okteta]

2. debugging zapisi [8-align]
3. instrukcijski blok

Primijetimo da je u debugging bytecodeu lako naći pripadnu liniju izvornog kôda tekuće instrukcije ili instrukciju koja pripada zadanoj liniji izvornog kôda (ili prvoj izvršivoj liniji nakon nje), npr. binarnim pretraživanjem.

Debugging zapisi omogućuju i baratanje bytecodeom po principu slučajnog pristupa instrukcijama, za što ne zahtijevaju nužno dodatnu radnu memoriju.

Cjelokupna struktura normalnog bytecodea prikaziva je kao:

1. deskriptor [8 okteta]
2. instrukcijski blok

3.3 kompatibilnost...

Poželjno je da sav softver bude što je više moguće backward (unatrag) kompatibilan, dok god to ne komplicira ili ne unazađuje suviše njegov design.

3.3.1 ...jezika

Sintaksa i semantika **MaMa** programskog jezika u verziji n bit će podržana u svim **MaMa** implementacijama $(n+1)$-ve verzije jezika.

To znači da ćete u paketu koji implementira npr. 4-tu verziju jezika moći koristiti, izravno ili putem nekih komandnih opcija, jezik u verziji 3, ali isti nećete nužno moći koristiti u paketu baziranom na jeziku verzije 5.

Od ovog se pravila može odstupiti ukoliko su specifikacijski zahtjevi toliko različiti između sukcesivnih verzija da bi poštivanje ovog pravila iziskivalo distribuciju dvaju nezavisnih aplikacija, za staru i novu verziju. Ako se od pravila odstupa, nužno je i staru verziju paketa ostaviti dostupnom.

3.3.2 ...bytecodea

Rečeno vrijedi i za bytecode, tj. bytecode verzije m moći će se izvršavati na **MaMa** implementaciji čija specifikacija opisuje bytecode verzije $m+1$.

Generiranje bytecodea neposredno prethodne verzije poželjna je opcija, ali nije nužna. Program može zahtijevati, direktivom `VERSION`, generiranje bytecodea neke od prethodnih verzija, ali kompajler to ne mora prihvatiti. U tom slučaju kompajler je dužan prekinuti kompilaciju.

4 manual

U ovoj se sekciji bavimo našim **MaMa** sustavom iz "ptičje perspektive", tj. interaktivnim aspektima korisniku neposredno vidljivih elemenata designa.

Ovaj dio dokumentacije služi kao priručnik korisniku za instaliranje i redovitu uporabu **MaMa** softverskog paketa. Podrazumijeva se da je korisnik upoznat sa sadržajem svih prethodnih poglavlja.

4.1 quick start

. . . ipak nije za nestrpljive! Da biste uspješno koristili **MaMa** paket, savjetuje se dobro proučiti sljedeća poglavlja!

4.1.1 sistemski zahtjevi

MaMa ne traži mnogo. Za čitanje dokumentacije trebat će vam PDF ili DVI preglednik. Dodatni zahtjevi napomenuti su u instalacijskoj dokumentaciji.

4.1.2 pribavljanje

MaMa je dostupna na Webu `https://github.com/venovako/MaMa`
Svaka **MaMa** distribucijska arhiva imenovana je `MaMa`$LBVD.xxx$, gdje su:

- L je verzija **MaMa** jezika, počev od 0
- B je verzija **MaMa** bytecodea, počev od 0
- V je, za dane L i B, release verzija
- D je oznaka sadržaja paketa i ciljane platforme
- X xxx je ekstenzija, ovisna o D

L, B, V i D svaki su jedan alfanumerički znak.

Varijante pod D objašnjene su na Webu i podložne promjenama. Pažljivo odaberite distribuciju (installer, source. . .) i obavezno provjerite checksum!

4.1.3 kompilacija

Ovo vas ne zanima ako ste skinuli installer. Imate li potrebu sami kompilirati **MaMa** paket, budite sigurni da posjedujete funkcionalne:

- ISO C++ kompajler, standardne biblioteke i prikladni linker
- recentnu LaTeX distribuciju, želite li modificirati dokumentaciju
- build sistem stvar je implementacije i podložan je promjenama; za detalje proučite odgovarajuću dodatnu dokumentaciju u paketu
- za kompilaciju GUI komponenti proučite instalacijsku dokumentaciju

Otpakirajte source distribuciju, pozicionirajte tekući direktorij vašeg shella na početni distribucijski direktorij, te proučite `INSTALL.txt` dokument. U njemu će biti detaljno naznačeno koje fajlove i kako morate modificirati, te koji build sistem upotrijebiti da biste kompajlirali i instalirali paket.

4.1.4 instalacija

Čitav **MaMa** paket sastoji se od { `bin`, `doc`, `etc`, `lib`, `src` } poddirektorija smještenih u direktorij po izboru. Defaultno je to `C:\MaMa` na Windowsima i `/opt/MaMa` na *nixu, ako korisnik nije izmijenio platformske postavke prilikom kompilacije. Taj se direktorij zove `MAMA_HOME` u daljnjem tekstu. Kako bi programi iz paketa znali gdje su instalirani, korisnik (na njemu dostupan način) postavi environment varijablu `MAMA_HOME` na glavni instalacijski direktorij. Ne učini li to, smatra se da ta varijabla ima rečenu default vrijednost.

Path separatori i delimiteri za danu platformu podešavaju se, kako je već opisano, prilikom kompilacije. Najčešće su jedno od:

	separator	delimiter
Windows	\	;
*nix	/	:

Sadržaj pojedinih direktorija (uz nužni mutatis mutandis za Windowse):

- `bin` sačinjavaju izvršni programi `MaMa` i `MaMaC`, te GUI komponente
- `doc` sadrži barem ovaj dokument u DVI i PDF formatu
- `etc` tvore razni primjeri i ostalo

- `lib` sadrži `BSO` (biblioteku standardnih makroa) i dodatne primjere u bytecode i izvornom obliku
- `src` izvorni kôdovi izvršnih programa

Smatramo da je postavljena i environment varijabla `MAMA_PATH`, ili ako nije, da sadrži kao defaultnu vrijednost `%MAMA_HOME%\lib` na Windows, odnosno `$MAMA_HOME/lib` na *nix sustavu.

4.1.5 korištenje

Detaljniji opis je u poglavljima o interpreteru, kompajleru i GUI IDEu.

4.1.6 troubleshooting

Use the source, Luke! Ako baš zapne, zatražite...

4.1.7 support

Pitanja i bug-reporte molimo slati na `mailto:venovako@gmail.com`

4.2 `MaMaC` – MaMa kompajler

`MaMaC` je kompajler koji iz tekstualnog **MaMa** programa producira ekvivalentan **MaMa** bytecode. Poziva se, uz opcionalne flagove, ovako:
`MaMaC [-v] [-g]` *datoteka_izvornog_kôda* gdje je:

- `-v` *verbose* – ispisuje korake i implementacijski definirarnu statistiku
- `-g` *debugging* – stvara debugging bytecode
- *default* – ispisuje samo poruke o sintaktičkim i drugim greškama

Kompajler je dužan prekinuti kompilaciju na prvoj sintaktičkoj greški, te obavijestiti korisnika o njoj i eventualnim mogućnostima oporavka od nje.

Kompajler smije promijeniti stanje filesistema ako i samo ako je kompilacija protekla bez grešaka i vanjskih terminirajućih signala.

4.2.1 pretraga datoteka

datoteka_izvornog_kôda apsolutni je ili relativni path do fajla koji sadrži izvorni program. Ako datoteka danog imena postoji, kompajlira se, a dobiveni se bytecode sprema u fajl imenovan tako da se uzme ime datoteke izvornog kôda i promijeni ekstenzija u `.MMb`. *Ako nije nužno drugačije postupiti, uvijek postavite ekstenziju* `.MMs` *datoteci izvornog kôda*!

Iz rečenog slijedi da se bytecode i izvorni kôd defaultno uvijek nalaze u istom direktoriju. Radi ispravnog ponašanja paketa u cjelini, to se *vrlo preporuča*!

4.2.2 ispis

Retke i stupce (kolone) ispisa imenujemo slijedno od 0. Podrazumijevamo da ekran ima barem 80 kolona. Dajemo format poruke o sintaksnoj greški:

```
01234567890123456789012345678901234567890123456789012345678901234567890123456789
SYNTAX ERROR @ LINE <**line**> : <neobavezni kratki opis greške>
<**line**> = broj linije u izvornom kôdu, počev od 1 i poravnat desno u polju od 10 kolona
```

Statistika u *verbose* režimu je implementacijski zavisna. U pravilu sadrži broj sintaksnih elemenata (linija, pojedninih tipova instrukcija), veličinu i byte ordering bytecodea, broj registara i veličinu pojedinih bytecode sekcija, vrijeme početka i završetka, odnosno trajanje kompilacije...

4.3 `MaMa` – MaMa interpreter

`MaMa` je interpreter **MaMa** bytecodea. Poziva se sa sljedećim parametrima:
`MaMa [-v|-s]` *ime_MaMa_programa registri*

- `-v` *verbose* – ispisuje svaki korak izračunavanja
- `-s` *step-by-step* – kao pod `-v`, ali čeka korisnikovu reakciju nakon svakog koraka (idealno za debugging i edukacijske svrhe)
- *default* – ispisuje rezultat iz registra 0
- *ime_MaMa_programa* – bez ekstenzije, istovjetno `CALL` argumentu
- *registri* – argumenti za vezanje ili punjenje registara (v. kasnije)

Postoji i *quiet* režim rada, u kojem se vraća samo status nakon (ne)uspješnog izvršavanja. Uključuje se automatski i jedino prilikom makro poziva.

4.3.1 izvršavanje: korisnički i makro kontekst

Važno: Ovdje se opisuju rekurzivni pozivi interpretera kao pozivi izvršnog `MaMa` programa. Implementacija je slobodna interno realizirati rekurzivne pozive bilo kako, dok god je očuvano značenje korisničkog i makro konteksta.

Poziv nove instance interpretera iz već postojeće moguć je jedino u makro, a iz korisničkog okruženja u korisničkom kontekstu. Dakle, kontekst se može odrediti i ustanovljuje se prije parsiranja komandne linije.

Potom se ispituju neobavezne opcije, zatim se pronalazi program, o čemu više u sljedećem odjeljku, a potom se preostali argumenti komandne linije vežu na ili pune registre nove instance **MaMa** stroja kako slijedi:

- Ako je riječ o korisničkom kontekstu, tada ili nema više argumenata, ili su svi preostali argument nenegativne cjelobrojne konstante u bazi 8, 10 ili 16, i njima se redom popunjavaju input registri ($\%1 \cdots \%k$) stroja.
- U makro kontekstu mora preostati bar još jedan argument. Prvi među njima mora biti *registar*, a ostali, ako ih ima, ili su konstante ili registri. Podrazumijeva se *quiet* režim rada nove instance, te se vrši vezanje registara, kako je to opisano semantikom `CALL` instrukcije u 3.1.5.

U oba slučaja neodgovarajući argumenti impliciraju neuspješan završetak rada nove, i rekurzivno svih pozivajućih instanci interpretera, ako postoje.

4.3.2 pronalaženje programa

Prvom obaveznom argumentu komandne linije (*ime_MaMa_programa*) dodaje se ekstenzija `.MMb` i pokušava otvoriti tako imenovani fajl. Ako se to ne uspije, taj se fajl traži redom u direktorijima navedenim u environment varijabli `MAMA_PATH`. Staje se na prvom uspješnom otvaranju fajla, koji se izvršava. Ne nađe li se bytecode niti u jednom direktoriju, interpreter neuspješno završava rad.

Iz rečenog slijedi da se interpreter ne brine o tome je li bytecode sinkroniziran s izvornim kôdom – to je isključivo korisnikova dužnost!

Default `MAMA_PATH` varijable ukazuje na to da je razumno u njoj imati `BSO`.

4.3.3 ispis

Svi ispisi vrše se na `stdout` (osim grešaka, koje se ispisuju na `stderr`). Kod `-v` ili `-s` opcije interpreter očekuje debugging bytecode (u protivnom neuspješno prekida rad) i u direktoriju bytecodea datoteku izvornog kôda (ako je nema, naredbe se ispisuju generički).

Ako za bytecode postoji rečeni izvorni kôd, tada se kao sljedeća instrukcija ispisuje stvarna linija sljedeće instrukcije. Ako to nije slučaj, onda se sljedeća instrukcija konvertira u niz znakova, pri čemu se eventualni file pointeri iz tijela instrukcije zamijenjuju labelama konstruiranima kao `:` koju slijedi redni broj instrukcije u bytecodeu. Sve su instrukcije također labelirane svojim rednim brojem. Specijalne oznake za `::` i `:!` labele poprimaju upravo tu tekstualnu reprezentaciju.

Uz instrukcije, moraju se ispisati i njihovi učinci na registarsku traku u takvom formatu da je moguće jednoznačno rekonstruirati vrijednost u svakom korištenom registru.

U defaultnom režimu rezultat se ispisuje u točno jednoj liniji i kao dekadska vrijednost nultog registra, poravnata desno u polju od prvih 20 kolona.

Retke i stupce (kolone) ispisa imenujemo slijedno od 0. Podrazumijevamo da ekran ima barem 80 kolona. Format ispisa u *verbose* režimu, kao i u *step* režimu, do na interakciju s korisnikom, koju opisujemo naknadno, je:

```
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
# MaMa started @ ‘ctime() output‘
<prazna linija>
PROGRAM ime_MaMa_programa
ARITY k # k je broj argumenata ove instance danog MaMa_programa
<prazna linija>
% <broj registara, ne računajući %%>
: <broj instrukcija>
<prazna linija>
SYSTEM s # s = 2 ili s = 3
VERSION v # 0 <= v <= 15
<prazna linija>
<početno stanje registara>
# ponavljaj 1 ili više puta...
<prazna linija>
<instrukcija koja se ima izvršiti>
# opcionalna interakcija u step modu
<stanje registara nakon prethodne instrukcije>
# ...kraj ponavljanja
<prazna linija>
<završno stanje registara>
RESULT <rezultat>
```

```
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
# komentar, također služi za interaktivne upite korisniku i njegove odgovore
```

```
01234567890123456789012345678901234567890123456789012345678901234567890123456789
%<reg___i> <20 col deci 64-bit>
<reg___i> = i u dekadskom prikazu, ili %, ako se radi o privremenom registru
<20 col deci 64-bit> = 20 kolona rezerviranih za prikaz nepredznačenog 64-bitnog
cijelog broja u bazi 10 kao vrijednosti u pripadnom registru
```

Oznake registara i pripadne vrijednosti u ispisu stanja poravnate su desno u poljima od 10, odnosno 20 kolona, respektivno. U ispisu stanja sudjeluju *samo oni* registri koji su spomenuti u prethodno izvršenoj instrukciji. Početno i završno stanje registara prikazuju stanja svih registara.

Linije izvornog kôda prenose se doslovno kad je to moguće, do na uklanjanje eventualnih završnih bjelina. Generički ispisane instrukcije labelirane su (na rečeni način), kapitaliziranog imena, a cjelobrojni argumenti i reference registara ispisuju se u bazi 10. Prije ispisa instrukcijske linije, generički labelirane ili izvorne, u polju od prvih 10 kolona, poravnato desno, ispisuje se pripadni broj linije izvornog kôda. Svi ti elementi odvojeni su po jednom bjelinom.

Rezultat je dekadska vrijednost nultog registra, ispisana uz desno poravnanje u polju od 20 kolona.

4.3.4 interakcija

Interpreter u `-s` modu interagira s korisnikom, ili direktno, vlastitim sučeljem, ili nekom GUI ljuskom koja poštuje niže opisani protokol.

Nakon jednog ciklusa, kako je naznačeno u gornjem odjeljku, ispiše se komentirani upit:

```
01234567890123456789012345678901234567890123456789012345678901234567890123456789
Step [[+/-]i] | setPc [[+/-/.]a] | %[...] [value] ?
```

Iza upitnika slijedi jedna praznina i korisnikov odgovor kao jedna od naredbi (punim imenom ili slovom kapitalizirane kratice ili samo `<Enter>` za `step`):

`S step [[+/-]i]`

- defaultni je korak 1, ako ga se ne postavi drugačije
- bez argumenta smatra se "`step` *defaultni_korak*"
- `i` > 0 ... izvrši bez zaustavljanja sljedećih `i` ciklusa, tj. jednokratno postavi korak na `i`, izvrši `i` ciklusa, resetira korak na trenutni default, te se zaustavi uz navedeni upit
- `i` $= 0$ ... izvrši bez zaustavljanja sve naredne cikluse (*continue*)
- `i` < 0 ... postavi defaultni korak na $|\,i\,|$, izvrši toliko ciklusa bez zaustavljanja i stane uz navedeni upit

`P setpc [[+/-/.]a]`

- bez argumenta prekida rad interpretera
- `+a` ... inkrementira registar-brojač (programsko brojilo) za `a`
- `-a` ... dekrementira registar-brojač (programsko brojilo) za `a`
- ako je `a` nepredznačen validni redni broj instrukcije kao cjelobrojna konstanta jezika C u bazi 8, 10 ili 16, registar-brojač (a.k.a. programsko brojilo – program counter) postavlja se na `a`
- `.a` ... skače na instrukciju kojoj pripada prva izvršiva linija izvornog kôda rednog broja većeg ili jednakog `a`

`% %[...] [value]`

- bez argumenata ponavlja ispis vrijednosti svih registara
- `...` validna adresa registra, kao nenegativna cjelobrojna konstanta jezika C u bazi 8, 10 ili 16, ili `%`
- bez zadnjeg argumenta ispisuje vrijednost referenciranog registra
- `value` validna vrijednost registra kao nenegativna cjelobrojna C konstanta u bazi 8, 10 ili 16
- sa zadnjim argumentom prisutnim postavlja vrijednost referenciranog registra na `value`

Pojašnjenje uz sve naredbe `%`-tipa (koje barataju s registrima): njihovo uspješno izvršavanje rezultira ponovnim interaktivnim upitom (možete odrediti da se i kako nastavi izvršavanje programa ili nastaviti rad s registrima).

4.4 `MaMut` – MaMa GUI

`MaMut` je aplikacija koja služi kao rudimentarni IDE.

4.4.1 značajke

`MaMut` posjeduje najosnovnije funkcionalnosti IDEa:

- uređivanje source (izvornog) kôda
- bytekompajliranje izvornog kôda
- selektiranje sintaksnih grešaka u izvornom kôdu

- podešavanje okoline (environment varijabli, tekućeg direktorija...) i prosljeđivanje argumenata za izvršavanje **MaMa** programa
- osnovne mogućnosti debuggiranja, kao što su:
  - konzolni prozor za izravnu komunikaciju s interpreterom
  - grafički prikaz i izmjena vrijednosti sviju korištenih registara
  - grafičko upravljanje izvršavanjem u *step* režimu
  - selektiranje sljedeće instrukcije izvornog kôda u *step* režimu

4.4.2 detalji

`MaMut` pokreće kompajler, odnosno interpreter, kao subprocese s preusmjerenim ulazom i izlazom na od roditelja kontrolirane streamove.

Preporučena rezolucija ekranskog prikaza je 800×600 i veća.

4.5 `BSO` – biblioteka standardnih makroa

`BSO` sačinjavaju najkorisniji **MaMa** programi za probleme elementarne aritmetike i opće namijene. `BSO` jamči da je svaki njen program:

- evaluator neke totalne (rekurzivne) funkcije
- optimalne vremenske složenosti
- stabilan u smislu da vrijednosti input registara po završetku programa nisu promijenjene, osim ako je to standardna semantika

Zasad su u planu mnoge funkcije, koje uglavnom odgovaraju skupu razumnih CISC asemblerskih instrukcija. Minimalna implementacija `BSO`a *mora* imati sve sljedeće makroe implementirane s ovdje zadanom semantikom!

- `MIN` ... $\%0 := \min\{\,\%1, \%2\,\}$
- `MAX` ... $\%0 := \max\{\,\%1, \%2\,\}$
- `ADD` ... $\%0 := \%1 + \%2$
- `SUB` ... $\%0 := \max\{\,\%1 - \%2, 0\,\}$
- `DST` ... $\%0 := |\,\%1 - \%2\,|$

- `MUL` ... $\%0 := \%1 \cdot \%2$
- `POW` ... $\%0 = \%1^{\%2}$
- `LT` ... $\%0 := \%1 < \%2$
- `LE` ... $\%0 := \%1 \leq \%2$
- `EQ` ... $\%0 := \%1 = \%2$

Pišete li vlastite makroe i distribuirate li ih trećoj strani, budite sigurni da je kod implementacije dodatnih makroa nedvosmisleno definirano je li riječ o evaluatoru totalne ili parcijalne funkcije, te, kod parcijalne funkcije, što se zbiva u slučaju nedozvoljenog ulaza.

Također, zapamtite da parcijalno rekurzivne funkcije, koje nisu rekurizvne, ne mogu imati svoj evaluator u `BSO` kolekciji.

`NE`, `GE`, `GT` makroi za respektivne operatore $\neq$, $\geq$, $>$ ne postoje jer su negacija odgovarajućih postojećih operatora.

5 TODO

- Specificirati GUI detaljnije.

Primjedbe i komentare uputite na `mailto:venovako@gmail.com`

8. lipnja 2018.