Универзитет у Београду  
Електротехнички факултет

дипломски рад

**Генератор x86-64 кôда**

**за МикроЈаву**

Проф. др Драган Бојић Лазар М. Цветковић

Београд, MMXX

*мајки Драгани, оцу Миљку,  
дедама Владану и пок. Милораду,  
бабама Љиљани и Смиљани,  
што несебично подржаваше  
мој рад протеклих година*

Садржај

[Увод 5](#_Toc44024122)

[Кратак преглед x86-64 архитектуре 6](#_Toc44024123)

[Генерисање међукôда 7](#_Toc44024124)

[Представљање аритметичких израза 7](#_Toc44024125)

[Троадресни кôд 7](#_Toc44024126)

[Скуп инструкција међујезика 8](#_Toc44024127)

[Појам базичног блока 9](#_Toc44024128)

[Препознавање базичних блокова 9](#_Toc44024129)

[Откривање програмских петљи 9](#_Toc44024130)

[Одређивање информација о живости 10](#_Toc44024131)

[Оптимизација међукода 11](#_Toc44024132)

[Local Value Numbering 11](#_Toc44024133)

[Генерисање асемблерског кôда 12](#_Toc44024134)

[Преамбула 12](#_Toc44024135)

[Рад са регистрима и променљивама 12](#_Toc44024136)

[Табеле за полиморфизам 13](#_Toc44024137)

[Мапирање инструкција међујезика 14](#_Toc44024138)

[Позивање преводиоца 15](#_Toc44024139)

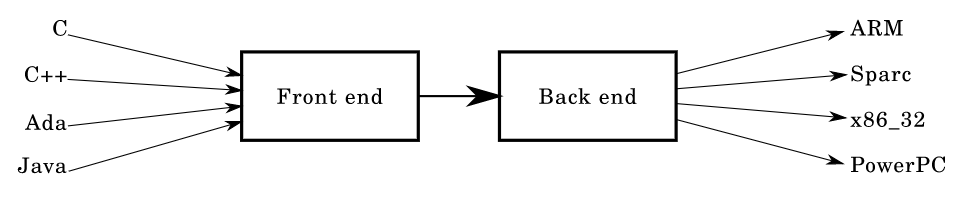
[Закључак 16](#_Toc44024140)

[Додатак А – граматика језика и лексичке структуре 17](#_Toc44024141)

[Литература 18](#_Toc44024142)

# Увод

Основна тема и садржина овог рада јесте приказ концепата и практична реализација генератора кôда за x86-64 инструкцијски сет на основу МикроЈава програмског кôда. МикроЈава је едукативни програмски језик који се користи на Електротехничком факултету Универзитета у Београду на курсу из програмских преводилаца. У питању је упрошћен објектно-оријентисани језик са синтаксом сличној изворној Java-и који подржава готово све језичке конструкте које модерни програмски језици данас имају, конкретно, класе, наслеђивање, полиморфизам, и др. Погодан је као инструмент за учење програмских преводилаца услед своје једноставности, те се на основу овог знања могу правити компликованији преводиоци који су ближи ономе што се данас користи у пракси.

 Данашњи програмски преводиоци коришћени у пракси се најчешће састоје из две грубе целине – front-end преводиоца и back-end преводиоца. Задатак првог јесте да генерише машински независан кôд (енгл. machine independent code) који треба да представља улаз у back-end део који генерише онда кôд за циљну архитектуру. Предност овакве архитектуре јесте што се не само један, већ више језика могу сликати на такав међујезик и тиме back-end је независтан од front-end језика.

*Слика 1 – шематски приказ преводиоца (аутор: Stefano Cherubin - CC BY-SA 3.0)*

У нашем случају, front-end компајлера ће имати задатак да изврши лексичку анализу, парсирање, семантичку анализу и генерисање међукôда (енгл. intermediate representation), док back-end има задатак да генерише x86-64 машински кôд.

Како је већ речено, међукôд се уводи као посебан међујезик који има задатак да апстрахује циљну архитектуру, да уведе независност од платформе. На нивоу међукôда врше компајлерске оптимизације на нивоу појединачних пролаза. Стога је прилично јасно да међујезик мора да буде врло прецизно и унапред осмишљен. Примери данас за њега јесу Common Intermediate Language – међујезик за .NET Framework, као и Java Bytecode.

Међујезик који ћемо ми користити и над којим ћемо да вршимо генерисање асемблерског кôда биће заснован на троадресном кôду представљеном у виду четворки (енгл. quadruples). Основни принципи и алгоритми јесу детаљно описани у [[ALSU06](#ALSU06)]. Наравно, сви концепти потребни за разумевање биће концизно изложени у каснијим поглављима овог рада.

По генерисању кôда за циљну машину он се води на улаз GNU Assembler алата који то треба да претвори у извршни фајл, а који је могуће извршити на циљној архитектури. Излаз нашег преводиоца за МикроЈаву врши позив одређених функција у оквиру стандардне библиотеке језика C (енгл. stdlib) које врши улазно/излазне операције те је потребно повезивање је потребно претходно повезивање са статичким библиотекама које су такође предмет овог рада, а које ће бити поменуте у делу о генерисању кôда.

Целокупан изворни кôд имплементације преводиоца и овај рад, као и скрипте за покретање преводиоца биће доступан у следећем Git репозиторијуму: https://github.com/cvetkovic/microjava\_x64.

# Кратак преглед x86-64 архитектуре

Архитектура x86-64 је врло једноставна, али моћна 64-битна архитектура, компатибилна уназад са претходним стандардом x86 архитектуре, која је задржала подршку за 16-битне и 32-битне *legacy* програме. Развој ове архитектуре је био вођен потребама у индустрији, као и тражњом за високим перформансама. Прилично је увећан број регистара доступних програмеру, и то шеснаест 64-битних регистара опште намене, као и шеснаест векторских регистара ширине до 512 бита. Инструкцијски сет је потпуно уназад компатибилан са x86 архитектуром, а подржава рад у SIMD режиму кроз SSE екстензију инструкцијског сета. Такође, подржане су операције са покретним зарезом по IEEE-754 [[AMD64\_1](#AMD64_1)].

Процесор који имплементира AMD64 архитектуру често рад

# Генерисање међукôда

Представа изворног кôда у виду међујезика има предност у погледу што је циљна машина идеализована и има бесконачну количину ресурса, док у случају реалне машине скуп ресурса је ограничен.

## Представљање аритметичких израза

****** Ради ефикасне представе аритметичких израза у облику троадресног кôда потребно је најпре извршити њихову конверзију парсирање израза и добијање синтаксног стабла, а онда и конверзију стабла у облик усмереног ацикличног графа (енг. directed acyclic graph - DAG). Разлог за ово јесте да се неки подизрази који су заједнички за аритметички израз који се обрађује не би рачунали више пута (у доњем примеру ), већ само једно, а тај међурезултат сачуван на некој меморијској локацији или регистру. На слици испод је дата представа помоћу синтаксног стабла и усмереног ацикличног графа.

 Атрибутивно-транслациона граматика која омогућава овакво превођење је детаљно описана у [[Boj11](#Boj11)]. Ради ефикасне меморијске представе графа, сви чворови су смештени хеш мапу јер је операција претраге по чвору врло честа. Чворови се чувају као уређене тројке <op, l, r>, где је op – операција, l – леви син и r – десни син (+, \*). У случају унарне операције (uminus), један од синова је *null* референца. Листови графа садрже само референцу на објектни чвор у табели симбола (a, b, c). Алгоритам за конструкцију графа се може пронаћи у [[ALSU06](#ALSU06)].

## Троадресни кôд

Троадресни кôд представља репрезентацију програма у коме се са десне стране знака наредбе доделе вишег програмског језика налази највише једна аритметичка операција. Увођење оваквог ограничења чини троадресни кôд врло погодним за генерисање и оптимизацију кôда за циљну машину.

У овом раду биће имплементиран троадресни кôд у виду четворки. Свака четворка се састоји од кôда операције, два аргумента и резултата. Аргументи су референце на објектне чворове табеле симбола. Такође, могуће су и унарне операције, те се један од аргумената тада изоставља. Поред четворки постоје и друге репрезентације у виду тројки, где је дестинациони операнд имплицитан, и имплицитних тројки. Важно је напоменути да су репрезентације у виду ацикличног усмереног графа описаног у претходном одељку и тројки еквивалентне у смислу аритметичких израза. Поред овога, постоје и индиректне тројке. Највећа предност четворки, иако заузимају више меморије, може да се види приликом оптимизовања кôда и то што је олакшано премештање инструкција унутар и ван базичних блокова.

## Скуп инструкција међујезика

Једна од најбитнијих одлука приликом дизајна компајлера јесте избор инструкцијског сета међујезика. Такав језик мора да буде довољно близак циљном језику превођења, а са друге стране не сувише ниског нивоа јер се тиме отежава посао оптимизатору и генератору кôда. Такође, међујезик мора да има способност да подржи све програмске конструкте језике вишег нивоа. У наставку је дат опис минималног скупа инструкција међујезика који омогућава пресликавање свих конструката стандардне MikroJava-е на x86-64 асемблер.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Инструкција | | Први  аргумент | | Други  аргумент | Дестинациони  аргумент | |
| аритметичке инструкције | | ADD  ***сабирање означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| SUB  ***одузимање означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| MUL  ***множење означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| DIV  ***дељење означених бројева*** | | операнд 1 | | операнд 2 | резултат | |
| MOD  ***остатак при дељењу*** | | операнд 1 | | операнд 2 | резултат | |
| NEG  ***негација у другом комплементу*** | | операнд 1 | |  | резултат | |
| рад са меморијом | | LOAD  ***читање из меморије*** | | адреса | |  | одредиште | |
| STORE  ***упис у меморију*** | | вредност за упис | |  | референца на одредиште | |
| PTR | адреса одредишта | |
| MALLOC  ***динамичка алокација***  ***меморије*** | | објектни чвор класе | |  | референца | |
| PTR | адреса | |
| број елемената низа | | ARR | референца | |
| ALOAD  ***читање елемента низа*** | | референца  на низ | | индекс | одредиште | |
| ASTORE  ***упис у елемент низа*** | | вредност за упис | | индекс | референца на низ | |
| GET\_PTR  ***дохватање адресе објекта*** | | објекат типа класе | | циљано поље | одредиште  адресе | |
| рад са потпрограмима | | PARAM  ***прослеђивање***  ***параметра методе*** | | аргумент по вредности | |  |  | |
| CALL/INVOKE\_VIRTUAL  ***позив метода/вирт. метода*** | | име методе | |  |  | |
|  | одредиште | |
| ENTER  ***припрема стека метода*** | | број бајтова за резервацију | |  |  | |
| LEAVE  ***распремање стека методе и***  ***повратак контроле тока на позиваоца тренутне методе*** | |  | |  |  | |
| RET  ***постављање повратне вредности методе*** | | повратна вредност | |  |  | |
|  | **Инструкција** | | **Први**  **аргумент** | | **Други**  **аргумент** | | | **Дестинациони  аргумент** |
| I/O | SCANF  ***читање са stdin*** | |  | | %b, %c, %d | | | резултат |
| PRINTF  ***испис на stdout*** | | %b, %c, %d | | вредност  за испис | | |  |
| контрола тока | JMP  ***безусловни скок*** | |  | |  | | | назив лабеле |
| JL, JLE, JG, JGE, JE, JNE  ***условни скок (<, ≤, >, ≥, ==, !=)*** | | аритметички израз | | аритметички  израз | | | назив лабеле |
| GEN\_LAB  ***генерисање лабеле*** | | назив лабеле | |  | | |  |

## Појам базичног блока

Оптимизатор кôда може у више пролаза међукôда да изврши жељене оптимизације.

У циљу олакшања процеса оптимизације потребно је увести појам базичног блока који се дефинише као секвенца инструкција за које важе следеће чињенице:

* ако се изврши прва инструкција, онда се гарантовано извршавају и све оне након ње у оквиру тог блока,
* контрола тока напушта базични блок само у последњој његовој инструкцији.

Ситуација која може бити неодређена јесте шта се дешава ако се догоди изузетак или прекид негде у блоку. У суштини, ова чињеница је сасвим ирелевантна за дизајн програмских преводилаца јер се кôд дели у базичне блокове само да би се извршила његова оптимизација. Такође, инструкције позивања других процедура не дели базични блок као инструкције скока јер се гарантује повратак из процедуре и наставак секвенцијалног извршења кôда до прве следеће инструкције скока.

## Препознавање базичних блокова

Алгоритам поделе кôда процедуре на базичне блокове укључује проналажење инструкција вођа (енг. *leaders*) које представљају прву инструкцију у оквиру базичног блока, а онда након тога и проналажење и последње, која је инструкција пред следећег вођу. Базични блокови представљају чворове графа контроле тока (енг. *control flow graph*), док су гране представљене листом следбеника базичног блока. Ово је такође могуће да се одреди, а за то је потребно додатно прилагодити горе наведени алгоритам проналажења базичног блока.

## Откривање програмских петљи

Будући да су петље главни извор паралелизма и представљају највећи простор за оптимизацију кôда корисно је у графу контроле тока бити у могућности препознавања петљи. То је могуће тако што се у графу контроле тока испита постојање циклуса (и неелементарних), а следећа теорема даје потребан и довољан услов да би скуп базичних блокова био препознат као програмска петља.

**Теорема I:**

*Скуп чворова L графа контроле тока је петља ако L садржи чвор e, такав да:*

1. *e није улазни чвор читавог графа контроле тока,*
2. *ниједан чвор из L, осим e нема претходника који није у скупу L,*
3. *сваки чвор из L има путању до e дужине бар један која је цела из L.*

## Одређивање информација о живости

Једна од корисних поступака приликом откривања базичних блокова унутар неког сегмента програмског кôда јесте и прикупљање информација о живости променљивих. Начин на који може да се одреде информације о живости биће детаљно објашњен, али је пре тога потребно увести дефиницију.

**Дефиниција 1:**

*Нека троадресна наредба врши доделу променљивој x. Ако наредба j садржи x као један од својих операнада, и контрола тока може да тече од i ка j тако да између њих нико други не врши доделу над x, тада кажемо да наредба j користи вредност променљиве x додељену у i, а за променљиву x кажемо да је жива у наредби i.*

**Алгоритам 1:**

*Нека је базични блок троадресних наредби, и нека су све непривремене променљиве унутар базичног блока означене као живе, а привремене као мртве. Крећући се од последње ка првој наредби у базичном блоку B, урадити:*

1. *доделити наредби информације о живости из претходне итерације,*
2. *поставити да променљива није жива и да нема слеће коришћење,*
3. *поставити да су променљиве и живе, и да је следеће њихово коришћење у .*

Процедура је иста и за троадресне наредбе које су облика и , а кораци (2) и (3) није могуће заменити јер може бити или . Такође, потребно је напоменути да оптимизација елиминације мртвог кôда није исто што и одређивање живости.

# Оптимизација међукода

Оно чиме ћемо се ми бавити у овом раду нису глобалне оптимизације, тј. оне које се протежу кроз више базичних блокова, већ искључиво локалне оптимизације, тј. оне над једним базичним блоком. Глобалне оптимизације захтевају другачији приступ који је прилично сложенији.

За локалне оптимизације најкорисније је конструисати усмерени ациклични граф, аналогно на начин као што смо то радили код аритметичких израза, а са информацијом у виду да су сада у градивни елементи све инструкције међујезика у оквиру посматраног базичног блока. У ток случају свака променљива изворног кôда представља један чвор, док за сваку наредбу у оквиру базичног блока постоји чвор који је повезан.

Корисно је применити објектно-оријентисане технике на дизајн класа оптимизатора јер је чест случај да се неки од пролаза оптимизатора понавља, те се од свих пролаза прави листа, а сваки имплементациони пролаз имплементира заједнички интерфејс [[LLVM\_IR](#LLVM_IR)].

## Local Value Numbering

Једна од корисних оптимизација у виду смањења извршавања програма јесте елиминација заједничких подизраза (енгл. common subexpression elimination), примена познатих алгебарских идентитета, као и израчунавање константи у време превођења (енгл. constant folding). Све ово може да се одради у једном пролазу алгоритмом на основу усмерених ацикличних графова који следи.

**Алгоритам 2:**

*Нека је B базични блок. Тада за сваку троадресну наредбу у базичном блоку B облика Ti := Li Opi Ri:*

1. *дохватити чворове графа који представљају Li и Ri*
2. *ако су и Ли и Ри константе, тада израчунати Li Opi Ri, и извршити доделу резултата у Ти, а Ти означити као консатнту*
3. *ако је Ли Опи Ри алгебарски идентитет, тада га заменити еквивалентном операцијом или операцијом доделе*
4. *направити чвор од Опи, Ли, Ри*
5. *ако је у хеш табели присутан већ овакав чвор тада заменити операцију и са копирањем у Ти и повезати резултат са Ти, а у случају да није додати нови чвор у хеш табелу и означити га као нову вредност за Ти.*

Оно што је битно напоменути јесте да уколико имамо операцију читања из низа, а онда након тога упис у низ, тада је потребно све чворове који су повезани са објектним чвором тог низа означити као мртве из графа јер може да се деси да је индексна променљива при упису, па онда читању различита, али да садржи исту вредност, те се онда губи исправна семантика израчунавања. Такође, у случају уписа преко показивача у неко поље потребно је обрисати целокупан граф јер у време превођења не може да се одреди вредност адресе на коју ће бити уписано, а то у теорији значи да може да буде упис на било које поље адресног простора. Детаљни опис имплементираног алгоритма се налази у [[KCLT12](#KCLT12)], а представља детаљнији и прилагођени опис сличног алгоритма из [[ALSU06](#ALSU06)].

# Генерисање асемблерског кôда

Ограничења наметнута генератору кôда циљне машине јесу врло стриктна. Циљни програм мора да задржи семантику изворног програма и да буде строго ефикасан и у виду перформанси, и у погледу заузећа ресурса машине. Проблем алокације ресурса представља НП-комплетан проблем, који је експоненцијалне сложености, те стога данашњи компајлери примењују разне хеуристике које генеришу добар, али не и гарантовано оптималан програмски кôд. Овакве технике се примењују пар деценија уназад и врло добро су усавршене [[ALSU06](#ALSU06)]. Међутим, како ово може да буде врло широка област и готово да нема краја у могућим усавршавањима преводилаца, биће представљена најједноставнија варијанта генератора кôда на нивоу базичних блокова.

Како је циљни инструкцијски сет AMD64, постоје две конвенције позивања које су примењене у пракси, и то Microsoft x64 Calling Convention и System V AMD64 ABI. Пошто је циљани оперативни систем овог рада Unix, биће примењена System V AMD64 ABI конвенција позивања. Такође, асемблерски кôд се генерише по Intel синтакси, која је релативно читљивију у односу на AT&T синтаксу. Коришћени преводилац за генерисани асемблерски кôд је GNU асемблер.

## Преамбула

Пре преласка на сам генератор кôда функција потребно је генерисати одређене асемблерске директиве које ће дефинишу функције calloc, scanf и printf као симболе са спољним повезивањем (енгл. extern), јер ће они бити дефинисани у стандардној C библиотеци преводиоца GCC. Ово је неопходно линкеру, јер ће се инструкције међујезика MALLOC, SCANF и PRINTF пресликавати у горе наведене функције, респективно.

Поред дефинисања екстерних функција, потребно је обезбедити простор за глобалне променљиве дефинисане у MicroJava програму, а које су, по правилу, неиницијализоване, те се стога традиционално смештају у *.bss* секцију.

Такође, ради омогућавања рада са стандардним улазом и излазом потребно је дефинисати стринг за форматирање горе претходно наведених улазно/излазних функција. То је у нашем случају, само рад са целим бројевима и појединачним карактерима. Уколико су у MikroJava програму дефинисане класе, овде је место где се стављају виртуелне табеле функција које омогућавају полиморфно понашање. Касније ће бити дато више детаља о њима. Наведено се све пакује у *.rodata* секцију, којој само име говори да је непроменљива.

По стандарду MicroJava-е дефинисан је скроман скуп уграђених функција (ord, chr, len), и оне су смештене у секције *.text.builtin*.

## Рад са регистрима и променљивама

Један од кључних проблема приликом процеса генерисања кôда је како најефикасније искористити регистре. То је битно јер регистри имају далеко мање кашњење при приступу у односу на меморију, јер се могу користити за чување привремених међурезултата, података који се често користе, за бројач итерација петље и др. Проблем настаје јер потражња за регистрима далеко надмашује њихов број, што имплицира честу замену (енгл. swapping).

Раније је наведено да ће асемблерски кôд бити генерисан на нивоу базичног блока. Потребно је генерисати одговарајуће асемблерске наредбе на нивоу сваке инструкције базичног блока, и то дохватање операнада, извршавање инструкције са наведеним операндима и упис резултата на одговарајуће место. Треба имати у виду да циљна процесорска архитектура не дозвољава да једна инструкција приступа више од једне меморијске локације.

У циљу вођења кохерентне слике меморије уводе се структуре података под називом дескриптори. Разликујемо две врсте, и то:

1. дескриптор регистра – односи се на регистар, те чува информацију о томе која променљива се тренутно налази у њему;
2. дескриптор адресе – односи се на конкретну меморијску локацију, те чува информацију која се променљива налази на тој адреси.

Такође, потребно је водити и рачуна о списку променљивих које су у dirty стању у регистрима. На крају сваког базичног блока потребно је све променљиве које су dirty да се упишу у меморију, а ради једноставности усвојено је да једна променљива може да буде у највише једном регистру.

Напоменуто је да је избор регистра за чување променљиве једна од критичних операција. Ову функцију ћемо називати у даљем тексту getRegister(…), а следи алгоритам поједностављени алгоритам наведен у [[ALSU06](#ALSU06)].

***Алгоритам 3:***

*Нека је инструкција облика . Правила за избор регистра за смештање променљиве су следећа:*

1. *ако се већ налази у неком регистру, вратити тај регистар;*
2. *ако се не налази ни у једном регистру и постоји слободан регистар, тада вратити такав слободан регистар;*
3. *ако оба горња случаја нису испуњена онда Round-Robin алгоритмом вратити неки регистар. Овде може да настане проблем ако се више пута позива ова функција заредом, па је потребно обезбедити да се не врати сваки пут исти регистар.*

Притом, функција getRegister() се у овој имплементацији не бави било каквим уписом у регистар који враћа, већ је за то задужена функција fetchOperand(…) која има улогу да у дати регистар смести жељену вредност уз евентуално избацивање претходне вредности и њено чување у меморију. Такође, за ажурирање табеле дескриптора користи се функција validate(…).

## Табеле за полиморфизам

Полиморфизам представља жељену особину модерних објектно-оријентисаних програмских језика, а може представљати компликован проблем у случају вишеструког наслеђивања. Проблем је како у време извршавања одредити коју методу позвати.

Како MicroJava омогућава само једноструко наслеђивање, а проблем се решава једноставно генерисањем табеле виртуелних функција, и то тако што се за сваку методу која је дефинисана, наслеђена или преклопљена на нивоу појединачне класе дефинише табела показивача на те функције у асемблеру. Табела се налази у .rodata секцији, а ово значи да се у време превођења мора одредити структура табеле, као и помераји појединачних функција унутар табеле. Начини генерисања кода биће описани у имплементацији инструкције INVOKE\_VIRTUAL. Процедура и додатни концепти генерисања и коришћења табела су описани у [[App02](#App02)] и [[Boj11](#Boj11)].

## Мапирање инструкција међујезика

Аритметичке инструкције ADD, SUB и MUL међујезика испољавају заједничке особине и мапирају се у асемблер на исти начин и то, бирање регистра за смештање операнада, њихово дохватање, издавање одговарајуће асемблерске инструкције ADD, SUB и IMUL респективно, те ажурирање дескриптора дестинационог регистра. Инструкција NEG се може одрадити врло једноставно, комбинацијом до сада наведених знања. Овде је битно напоменути да операције инкрементирања и декрементирања су одрађене преко инструкција ADD и SUB, респективно, јер су INC и DEC инструкције асемблера проблематичне зато што не генеришу исти скуп флегова програмске статусне речи као и ADD и SUB, па ако се користе за услов скока може бити потребна додатна CMP инструкција. Због тога се препоручује њихово избегавање [[Fog20](#Fog20)].

Случај са инструкцијама DIV и REM је нешто другачији и оне су сложеније. Наиме, дељење бројева захтева да се тај број 64-битни број смести у EDX:EAX регистре, а резултати дељења, односно остатка при дељењу се смештају у EAX и EDX регистре, респективно. Како MicroJava подржава само 32-битне целобројне величине, потребно је уметнути инструкцију CDQ која врши проширивање EDX знаком броја у EAX. Одговарајућа инструкција за дељење на AMD64 асемблеру је IDIV.

Инструкције типа LOAD јесу релативно једноставне и неће бити детаљно описиване, док инструкције типа STORE по спецификацији међујезика подржавају упис директно и преко показивача. У случају уписа преко показивача имаћемо регистарско индиректно адресирање, док је ситуација са директним уписивањем тривијална.

Алокација меморије на хипу инструкцијом MALLOC захтева посебну пажњу. И то што захтева позивање функције calloc стандардне C библиотеке. Овом инструкцијом може да се алоцира простор за низ или за инстанцу класе. Случај низа је једноставан јер је у инструкцији међујезика наведена величина низа и то се једноставно прослеђује calloc функцији. У случају алокације меморије за објекат класе, тада је потребно обезбедити простор величине збира свих поља унутар класе увећан за осам, што представља додатно поље 64-битног показивача на виртуелну табелу функција те класе. На основу типа класе потребно је на адресу враћену функцијом calloc уписати адресу одговарајуће табеле виртуелних функција.

Приступ елементима низа инструкцијом ALOAD се одвија на једноставан начин више регистара и то. Један је потребан за адресу регистра, и други за индексну променљиву. Пошто MicroJava подржава низове различитих елементарних типова, могућа је самим тим и променљива величина тих елементарних типова, те се индексна променљива множи са величином појединачног елемента (нпр. [rbx + 4 \* rbx]). Пошто је индекс често 32-битна величина, потребно га је проширити на 64-битну величину, те се индексна променљива учитава у регистар инструкцијом MOVSXD. Процедура је аналогна за ASTORE инструкцију међујезика.

Инструкција међујезика GET\_PTR се у нашем случају своди на једноставно сабирање првог аргумента инструкције са померајем жељеног поља.

Најсложенији је поступак позивања других процедура (функција) у оквиру програма. Прослеђивање параметара се врши инструкцијом међујезика PARAM која се своди на смештање првих шест аргумената функције у одговарајуће регистре у складу са System V AMD64 ABI, и то у RDI, RSI, RDX, RCX, R8, R9 редом, а онда и стављањем осталих параметара на стек у обрнутом редоследу. Након тога може да се изда инструкција CALL. Процедура за позив функције унутар класе јесте индиректни скок и то тако што се на основу адресе имплицитног показивача this одреди адреса виртуелне табеле функција из које се дохвати адреса жељене функције а онда на њу скочи инструкцијом CALL. По повратку из позване функције, сви стековски параметри се скидају, а ако позвана функција враћа повратну вредност, онда се налази у RAX.

Поступак уласка и изласка из метода помоћу инструкција ENTER и LEAVE је релативно једноставан. При уласку се вршу стављање RBP на стек, његово ажурирање да показује на показивач стека а онда и резервисање простора за локалне променљиве на стеку. Потребно је да величина поменутог простора буде дељива са 16, по конвенцији. На изласку из методе потребно је сачувати dirty глобалне променљиве у меморији и издати инструкције LEAVE и RET. Такође ако се мења неки од регистара чија вредност мора да буде сачувана приликом позива процедура, онда је њих потребно сачувати некако (нпр. на стеку). Повратна вредност методе се прослеђује RET инструкцијом међујезика, и онда је тривијална, као и инструкција међујезика GEN\_LABEL.

Рад са улазом/излазом се своди на позивање фунцкија scanf и printf стандардне C библиотеке, респективно, и поступак позивања фунцкија је описан раније. Једина ствар о којој треба водити рачуна јесте да се у EAX регистар смешта нула, што омогућава позивање фунцкија са варијабилним бројем аргумената, а какве су и две горепоменуте. Ово је у нашем случају нула, што се поставља једноставним ексклузивним ИЛИ.

Инструкције скока међујезика се деле на безусловне и условне. Обе карактерише то што је обавезно чување локалних и глобалних променљивих пре скока. У случају условног скока додаје се CMP асемблерска инструкција која генерише одговарајући услов скока.

# Позивање преводиоца

До сада је концептуално објашњен процес превођења MicroJava изворног кôда у асемблер циљне архитектуре. Процес покретања преводиоца за жељени кориснички програм је дат у наставку.

Преводилац се покреће позивом скрипте “mjavac.sh”. Она представља омотач позива извршном окружењу JVM да преведе достављени изворни кôд, затим преведе GNU асемблером излазни фајл MicroJava преводиоца, а онда и покрене извршење програма. Преводилац подржава следеће аргументе:

1. -help ⬄ врши испис могућих команди преводиоца
2. -input [PATH] ⬄ путања до изворног кôда
3. -output [PATH] ⬄ путања на којој ће бити уписан преведени кôд
4. -dump\_ast ⬄ приказ апстрактног синтаксног стабла изворног кôда
5. -dump\_symbols ⬄ приказ табеле симбола програма
6. -dump\_ir ⬄ приказ инструкција међујезика
7. -optimize\_ir ⬄ укључивање оптимизација
8. -dump\_asm ⬄ приказ генерисаног асемблерског кôда

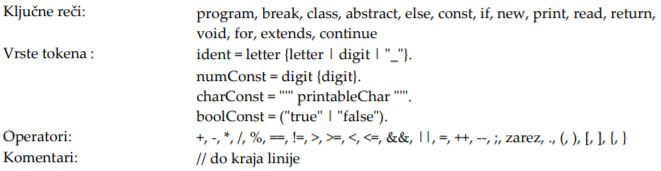
Покретањем скрипте под називом “mjavacr.sh” врши се све исто што и претходна скрипта уз покретање генерисаног програма.

# Закључак

У раду је представљен један једноставни програмски преводилац за језик MicroJava, а који притом демонстрира готово све базичне функционалности које модерни програмски језици данас поседују, и то низове, класе, наслеђивање, полиморфизам. Језик MicroJava пружа довољно конструката да се могу писањем преводиоца за њега стећи и добро научити основе програмских преводилаца, а што представља неопходно знања за даље изучавање и конструкцију напредних преводилаца.

Програмски преводиоци су широка, готово непресушна област за нове идеје, те тако овај рад има безброј места на којима је могуће додати неку нову функционалност и унапредити ефикасност преводиоца. Следи скроман списак предложеног простора за даљи рад: унапређење униформности међујезика, јер су неке инструкције у њему редундантне (конкретно, ALOAD и ASTORE), имплементација напреднијег генератора међукôда, сакупљача ђубрета, као и многе друге.

# A screenshot of text Description automatically generatedДодатак А – граматика језика и лексичке структуре

****

# Литература

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 2006.

[AMD64\_1] Advanced Micro Devices, *AMD64 Architecture Programmer’s Manual Volume 1: Application Programming*, Revision 3.22, December 2017

[App02] A. W. Appel, *Modern Compiler Implementation in Java,* Cambridge University Press, second edition, 2004.

[Avr10] N. Avramović, D. Bojić, *Jednostavan generator kôda za Mikrojavu*. Elektrotehnički fakultet, Beograd, 2010.

[Boj11] D. Bojić, *Materijali za predavanja i vežbe iz predmeta ,,Programski prevodioci 1ˮ*,Elektrotehnički fakultet, Beograd, 2011.

[Fog20] A. Fog, *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers,* Technical University of Denmark, 2020.

[GNUas] GNU Project, *Using as.*

[KCLT12] K. Cooper, L. Torczoh, *Engineering a Compiler*, Morgan Kaufmann, Burlingon, MA, second edition, 2012.

[LLVM\_IR] The LLVM Foundation, *LLVM Compiler Infrastructure Language Reference.*

[MIT6.172] C. Leiserson, J. Shun, *6.172 Performance Engineering of Software Systems*. Fall 2018. Massachusetts Institute of Technology: MIT OpenCourseWare.

[ZSR] Z. S. Rakić, *Materijali za predavanja i vežbe iz predmeta ,,Programski prevodiociˮ*,Univerzitet u Novom Sadu, Novi Sad.