

УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ



ГЕНЕРАТОР x86-64 кôда ЗА МИКРОЈАВУ

ДИПЛОМСКИ РАД

МЕНТОР
ПРОФ. ДР ДРАГАН БОЈИЋ
РЕДОВНИ ПРОФЕСОР

КАНДИДАТ
ЛАЗАР М. ЦВЕТКОВИЋ
127/2016

БЕОГРАД, јул 2020.

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

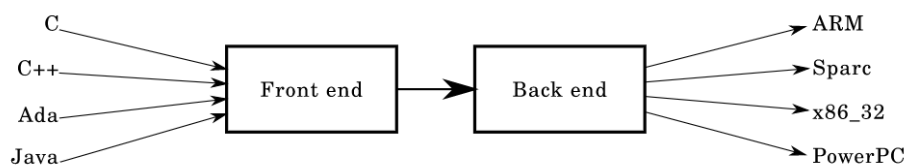
Садржај

1. Увод.....	5
2. Кратак осврт на x86-64 архитектуру.....	6
3. Генерисање међукôда.....	7
3.1 Представљање аритметичких израза.....	7
3.2 Троадресни кôд.....	7
3.3 Скуп инструкција међујезика.....	8
3.4 Појам базичног блока.....	9
3.5 Препознавање базичних блокова.....	9
3.6 Одређивање информација о живости.....	9
4. Генерисање асемблерског кôда.....	11
4.1 Преамбула.....	11
4.2 Рад са регистрима и променљивама.....	11
4.3 Табеле за полиморфизам.....	12
4.4 Мапирање инструкција међујезика.....	13
5. Позивање преводиоца.....	14
6. Закључак.....	15
7. Додатак.....	16
7.1 Пример процедуралног програма.....	16
7.2 Пример објектно оријентисаног програма.....	20
7.3 Граматика језика и лексичке структуре.....	32
8. Литература.....	33
9. Списак коришћених слика.....	33

1. Увод

Основна тема и садржина овог рада јесте приказ концепата и практична реализација генератора кода за x86-64 инструкцијски сет на основу МикроЈава програмског кода. МикроЈава је едукативни програмски језик који се користи на Електротехничком факултету Универзитета у Београду на курсу из програмских преводаца. У питању је упрошћен објектно оријентисани језик са синтаксом сличној изворној Јава-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 преводаца 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 код (енгл. machine independent code) који треба да представља улаз у back-end део који генерише онда код за циљну архитектуру. Предност овакве архитектуре јесте што се не само један, већ више језика може сликати на такав међујезик и тиме је back-end независан од front-end језика.



Слика 1 – шематски приказ преводиоца

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

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

Међујезик који ће бити коришћен и над којим ће се вршити генерисање асемблерског кода биће заснован на троадресном коду представљеном у виду четворки (енгл. quadruples). Основни принципи и алгоритми јесу детаљно описани у [ALSU06]. Наравно, сви концепти потребни за разумевање биће концизно изложени у каснијим поглављима овог рада.

По генерисању кода 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 овог преводиоца за МикроЈаву врши позив одређених функција у оквиру стандардне библиотеке језика C (енгл. stdlib) које врше улазно/излазне операције те је потребно претходно повезивање са статичким библиотекама које су такође предмет овог рада, а које ће бити поменуте у делу о генерисању кода.

Целокупан изворни код имплементације преводиоца и овај дипломски рад, као и скрипте за покретање преводиоца биће јавно доступни у следећем Git репозиторијуму: https://github.com/cvetkovic/microjava_x64.

2. Кратак осврт на x86-64 архитектуру

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 (додато осам 64-битних регистара – r8-r15), као и шеснаест векторских регистара ширине до 512 бита. Инструкцијски сет ради у SIMD режиму кроз SSE екстензију инструкцијског сета. Такође, подржане су операције са покретним зарезом по IEEE-754 [AMD64_1]. Потребно је нагласити да се често мешају x64 и IA64 архитектуре. Друга представља 64-битну Itanium VLIW архитектуру коју су развили заједнички Intel и HP. У рачунарству високих перформанси x86-64 архитектура је последњих десетак година доминантно заступљена [TOP500].

Подржан је 64-битни виртуелни адресни простор, од којих се тренутно користе нижих 48 бита за адресирање, а што је у данашњим потребама апсолутно довољно. Остављен је простор за накнадно повећање коришћења додатних бита. Такође, величина адресе у физичком адресном простору је такође 48-бита. Омогућено је и релативно адресирање податка у односу RIP регистар, што побољшава ефикасност позиционо независног кода (енгл. position independent code), а који овај рад користи приликом генерисања извршног фајла.

Што се тиче асемблера архитектуре, мнемоници су двоадресног формата, где је леви операнд и имплицитни дестинациони операнд. Постоје инструкције које врше проширивање података у регистру његовим знаком на већу ширину (коришћено код инструкција дељења и за индексирање елемента низа). Присутне су и инструкције које врше условно извршавање, налик на оне код ARM архитектуре. Произвођачи процесора често подржавају и додатне инструкције за неке рачунски интензивне операције, попут AES алгорита, те су процесори врло богати инструкцијама доступним програмеру, одн. преводиоцу, а који је често уско грло при експлоатисању оваквих могућности. Модерни процесори чак имају хардверски имплементиран генератор случајних бројева који може да се користи у криптографији.

Постоји више сигурносних режима – привилегија у којима процесор може да се налази у зависности какав код се извршава. Појединачни прстен (енгл. ring) представља поменути ниво привилегије. Често је подржана виртуелизација на нивоу хардвера. Процесорски стек може пуно да деградира перформансе, те се препоручује поравнање на 64-бита, а што је неопходно конвенцијом позивања при коришћењу CALL инструкције и повратка из процедуре инструкцијом RET. Регистар показивача стека указује на последњи додати податак, а стек расте од виших ка нижим меморијским адресама.

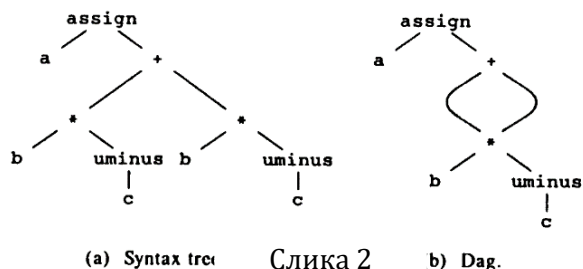
Аритметика са покретним зарезом је подржана и уграђена унутар процесора, и то са осам регистара (FPR0-FPR7). Подржан је рад у 32-битној, 64-битној прецизности, као и у проширеној 80-битној прецизности која је, услед тога што није подржана од већина преводилаца, доступна програмеру само у асемблеру. BCD аритметика је, такође, хардверски подржана.

3. 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.

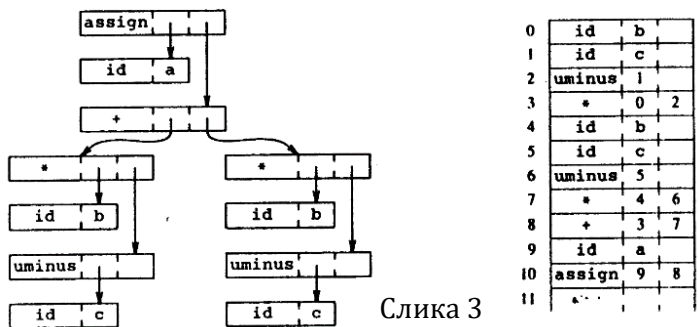
3.1 Представљање аритметичких израза

Ради ефикасне представе аритметичких израза у облику троадресног кôда потребно је најпре извршити њихову конверзију у погодан облик парсирањем израза и добијањем апстрактног синтаксног стабла, а онда и конверзијом стабла у облик усмереног ацикличног графа (енгл. directed acyclic graph - DAG).



ово јесте да се неки подизрази који су заједнички на нивоу аритметичког израза који се обрађује не би рачунали више пута (у десном примеру $b \cdot (-c)$), већ само једном. Тај међурезултат се сачува на некој меморијској локацији или у регистру.

Атрибутивно-транслациона граматика која омогућава овакво превођење је детаљно описана у [Вој11]. Ради ефикасне меморијске представе графа, сви чворови су смештени у хеш мапу зато што је операција претраге по чвору врло честа. Чворови се чувају као уређене тројке $\langle op, l, r \rangle$, 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 је детаљно описан у [ALSU06].

3.2 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 аргумента 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.

3.3 Скуп инструкција међујезика

Једна од најбитнијих одлука приликом прављења програмских преводаца јесте пројектовање 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 кода. 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 Микројаве на x86-64 асемблер.

	Инструкција	Први аргумент	Други аргумент	Дестинациони аргумент
аритметичке инструкције	ADD <i>сабирање означених бројева</i>	операнд 1	операнд 2	резултат
	SUB <i>одузимање означених бројева</i>	операнд 1	операнд 2	резултат
	MUL <i>множење означених бројева</i>	операнд 1	операнд 2	резултат
	DIV <i>дељење означених бројева</i>	операнд 1	операнд 2	резултат
	MOD <i>остатак при дељењу</i>	операнд 1	операнд 2	резултат
	NEG <i>негација у другом комплементу</i>	операнд 1		резултат
рад са меморијом	LOAD <i>читање из меморије</i>	адреса		одредиште
	STORE <i>упис у меморију</i>	вредност за упис		референца на одредиште
			PTR	адреса одредишта
	MALLOC <i>динамичка алокација меморије</i>	објектни чвор класе		референца
		број елемената низа	PTR	адреса
	ALOAD <i>читање елемента низа</i>	референца на низ	индекс	одредиште
	ASTORE <i>упис у елемент низа</i>	вредност за упис	индекс	референца на низ
рад са потпрограмима	GET_PTR <i>дохватање адресе објекта</i>	објекат типа класе	циљано поље	одредиште адресе
	PARAM <i>прослеђивање параметра методе</i>	аргумент по вредности		
	CALL/INVOKE_VIRTUAL <i>позив метода/вирт. метода</i>	име методе		одредиште
	ENTER <i>припрема стека метода</i>	број бајтова за резервацију		
	LEAVE <i>распремање стека методе и повратак контроле тока на позиваоца тренутне методе</i>			
	RET <i>постављање повратне вредности методе</i>	повратна вредност		

	Инструкција	Први аргумент	Други аргумент	Дестинациони аргумент
I/O	SCANF <i>читање са stdin</i>		%b, %c, %d	резултат
	PRINTF <i>испис на stdout</i>	%b, %c, %d	вредност за испис	
контрола тока	JMP <i>безусловни скок</i>			назив лабеле
	JL, JLE, JG, JGE, JE, JNE <i>условни скок (<, ≤, >, ≥, ==, !=)</i>	аритметички израз	аритметички израз	назив лабеле
	GEN_LABEL <i>генерисање лабеле</i>	назив лабеле		

Табела 1 – опис инструкција међујезика

3.4 Појам базичног блока

У циљу олакшања процеса оптимизације и генерисања кода 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 код 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 кода до прве следеће инструкције скока.

3.5 Препознавање базичних блокова

Алгоритам поделе кода 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. За конструисање графа контроле тока потребно је додатно прилагодити наведени алгоритам, а што није предмет овога рада.

3.6 Одређивање информација о живости

Једна од корисних поступака приликом откривања базичних блокова унутар неког сегмента програмског кода 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 операнда, 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 B базични блок троадресних наредби, и нека су све непривремене променљиве унутар базичног блока иницијално означене као живе, а привремене као мртве. Крећући се од последње ка првој наредби i : $x = y + z$ у базичном блоку B , урадити:

- (1) доделити наредби i информације о живости из претходне итерације,*
- (2) поставити да променљива x није жива и да нема следеће коришћење,*
- (3) поставити да су променљиве y и z живе, и да је следеће њихово коришћење у i .*

Процедура је иста и за троадресне наредбе које су облика $x = +y$ и $x = y$, а кораци (2) и (3) није могуће заменити јер x може бити y или z . Такође, потребно је напоменути да преводачка оптимизација која ради елиминацију мртвог кода није исто што и одређивање живости.

4. Генерисање асемблерског кода

Ограничења наметнута генератору кода циљне машине су 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, тј. експоненцијалне је сложености. Стога данашњи компајлери примењују разне хеуристике које генеришу добар, али не и гарантовано оптималан програмски код. Овакве технике се примењују пар деценија уназад и врло добро су усавершене по [ALSU06]. Међутим, како ово може да буде врло широка област и готово да нема краја у могућим усавршавањима преводаца, биће представљена најједноставнија варијанта генератора кода на нивоу базичних блокова.

Како је циљни инструкцијски сет x86-64, 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 код се генерише по Intel синтакси, која је релативно читљивију у односу на AT&T синтаксу. Коришћени преводац за генерисани асемблерски код је GNU асемблер.

4.1 Преамбула

Пре преласка на сам генератор кода функција потребно је генерисати одређене асемблерске директиве које ће да 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 МикроЈава 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 ниску (енгл. *string*) за форматирање горе претходно наведених улазно/излазних функција. То је у нашем случају, само рад са целим бројевима и појединачним карактерима. Уколико су у МикроЈава програму дефинисане класе, овде је место где се стављају табеле виртуелних 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 (енгл. *read-only*).

По стандарду МикроЈаве дефинисан је скроман скуп уграђених функција (`ord`, `chr`, `len`), и оне су смештене у секцију *.text.builtin*.

4.2 Рад са регистрима и променљивама

Један од кључних проблема приликом процеса генерисања кода је како најефикасније искористити регистре. То је битно јер регистри имају далеко мање кашњење при приступу у односу на меморију, те 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 код бити генерисан на нивоу базичног блока. Потребно је генерисати одговарајуће асемблерске наредбе на нивоу сваке инструкције

базичног блока, и то дохватање операнда, извршавање инструкције са наведеним операндима и упис резултата на одговарајуће место. Треба имати у виду да x86-64 архитектура дозвољава да у оквиру једне асемблерске наредбе само један операнд буде у меморији.

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` уписати 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 поједностављени алгоритам наведен у [ALSU06].

Алгоритам 3:

Нека је инструкција облика $x = y + z$. Правила за избор регистра за смештање променљиве у су следећа:

- (1) ако у се већ налази у неком регистру, вратити тај регистар;*
- (2) ако у 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 (напомена: 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 избацавање претходне вредности и њено чување у меморију. Такође, за ажурирање табеле дескриптора користи се функција `validate(...)`.

4.3 Табеле за полиморфизам

Полиморфизам представља жељену и неопходну 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 МикроЈава 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] и [Boj11].

4.4 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 операнда, 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 су операције инкрементирања и декрементирања одрађене преко инструкција ADD и SUB, респективно, јер су INC и DEC инструкције асемблера проблематичне зато што не ажурирају апсолутно исти скуп флегова програмске статусне речи као и ADD и SUB, па ако се користе као услов скока може бити потребна додатна CMP инструкција. Због тога се препоручује њихово избегавање [Fog20].

Случај са инструкцијама DIV и REM је нешто другачији и оне су сложеније. Наиме, дељење бројева захтева да се тај 64-битни број смести у EDX:EAX регистре, а резултати дељења, односно остатка при дељењу се смештају у EAX и EDX регистре, респективно. Како МикроЈава подржава само 32-битне целобројне величине, потребно је уметнути инструкцију CDQ која врши проширивање EDX знаком броја у EAX. Одговарајућа инструкција за дељење на x86-64 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 heap-у 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 уз коришћење барем два архитектурална регистара. Један је потребан за адресу регистра, и други за индексну променљиву. Пошто МикроЈава 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. $[rbx + ELEMENT_SIZE * rbx]$). Пошто је индекс у МикроЈави 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, она 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 функција *scanf* и *printf* стандардне C библиотеке, респективно, и поступак позивања функција је описан раније. Једина ствар о којој треба водити рачуна јесте да се у EAX регистар смешта нула, што омогућава позивање функција са варијабилним бројем аргумената, а какве су и две горепоменуте. Ово је у нашем случају нула, што се поставља једноставним ексклузивним ИЛИ. Такође, како би се вратио само један карактер приликом позива стандардне *scanf* функције потребно је да формат улаза има додатни бланко карактер.

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.

5. Позивање преводиоца

До сада је концептуално објашњен процес превођења МикроЈава изворног кода 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 код, затим преведе GNU асемблером излазни фајл МикроЈава преводиоца, а онда и покрене извршење програма. Преводилац подржава следеће аргументе:

- (1) -help ⇔ врши испис могућих команди преводиоца
- (2) -input [PATH] ⇔ путања до изворног кода
- (3) -output [PATH] ⇔ путања на којој ће бити уписан преведени код
- (4) -dump_ast ⇔ приказ апстрактног синтаксног стабла изворног кода
- (5) -dump_symbols ⇔ приказ табеле симбола програма
- (6) -dump_ir ⇔ приказ инструкција међујезика
- (7) -dump_asm ⇔ приказ генерисаног асемблерског кода

Додавањем параметра "-invoke_after" приликом позива горенаведене скрипте извршиће се покретање преведеног програма. Преведени програм има исто име као улазни МикроЈава фајл.

6. Закључак

У раду је представљен један једноставни програмски преводилац за језик МикроЈава, а који притом демонстрира готово све базичне функционалности које модерни програмски језици данас поседују, и то низове, класе, наслеђивање, полиморфизам. Језик 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 локалног или глобалног оптимизатора кода, напреднијег генератора међукода, сакупљача ђубрета, итд.

7. Додатак

7.1 Пример процедуралног програма

Дат је МикроЈава изворни код тест програма, представа у међујезику, као и излазни асемблерски код.

```
program test301
```

```

const int nula = 0;
const int jedan = 1;
const int pet = 5;

int niz[];
char nizch[];

{
    void incInput()
    int i;
    {
        read(i);
        print(i);
        i++;
        print(i);
    }

    void main()
    int bodovi;
    {
        bodovi = 0;
        bodovi++;
        bodovi = bodovi + jedan;
        bodovi = bodovi * pet;
        bodovi--;
        print(bodovi);

        niz = new int[3];
        niz[nula] = jedan;
        niz[1] = 2;
        niz[niz[jedan]] =
        niz[niz[0]] * 3;
        bodovi = niz[2]/niz[0];
        print(bodovi);
        print(niz[2]);

        nizch = new char[3];
        nizch[0] = 'a';
        nizch[jedan] = 'b';
        nizch[pet - 3] = 'c';
        print(nizch[1]);
        print(nizch[jedan * 2]);

        bodovi = bodovi + (pet *
        jedan - 1) * bodovi - (3 % 2 + 3 *
        2 - 3);
        print(bodovi);

        incInput();
    }
}

```

GEN_LABEL	incInput				
ENTER	16				
SCANF			%d	i	(A)
PRINTF	%d		i	(A)	
ADD	i	(A)	1	(A)	i
PRINTF	%d		i	(A)	
LEAVE					
GEN_LABEL	main				
ENTER	112				
STORE	0	(A)		bodovi	(A)
ADD	bodovi	(A)	1	(A)	bodovi
ADD	bodovi	(D)	1	(A)	t1
STORE	t1	(D)			bodovi
MUL	bodovi	(D)	5	(A)	t2
STORE	t2	(D)			bodovi
SUB	bodovi	(A)	1	(A)	bodovi
PRINTF	%d		bodovi	(D)	
MALLOC	3		ARR		niz
ASTORE	1	(A)	0	(A)	niz
ASTORE	2	(A)	1	(A)	niz
ALOAD	niz	(A)	1	(A)	t3

ALOAD	niz	(A)	0	(A)	t4	(A)
ALOAD	niz	(D)	t4	(D)	t5	(A)
MUL	t5	(D)	3	(A)	t6	(A)
ASTORE	t6	(D)	t3	(D)	niz	(A)
ALOAD	niz	(A)	2	(A)	t7	(A)
ALOAD	niz	(A)	0	(A)	t8	(A)
DIV	t7	(D)	t8	(D)	t9	(A)
STORE	t9	(D)			bodovi	(A)
PRINTF	%d		bodovi	(A)		
ALOAD	niz	(A)	2	(A)	t10	(A)
PRINTF	%d		t10	(D)		
MALLOC	3		ARR		nizch	(D)
ASTORE	97	(A)	0	(A)	nizch	(D)
ASTORE	98	(A)	1	(A)	nizch	(D)
SUB	5	(A)	3	(A)	t11	(A)
ASTORE	99	(A)	t11	(D)	nizch	(A)
ALOAD	nizch	(A)	1	(A)	t12	(A)
PRINTF	%c		t12	(D)		
MUL	1	(A)	2	(A)	t13	(A)
ALOAD	nizch	(A)	t13	(D)	t14	(A)
PRINTF	%c		t14	(D)		
MUL	5	(A)	1	(A)	t15	(A)
SUB	t15	(D)	1	(A)	t16	(A)
MUL	t16	(D)	bodovi	(A)	t17	(A)
ADD	bodovi	(D)	t17	(D)	t18	(A)
REM	3	(A)	2	(A)	t19	(A)
MUL	3	(A)	2	(A)	t20	(A)
ADD	t19	(D)	t20	(D)	t21	(A)
SUB	t21	(D)	3	(A)	t22	(A)
SUB	t18	(D)	t22	(D)	t23	(A)
STORE	t23	(D)			bodovi	(A)
PRINTF	%d		bodovi	(A)		
CALL	incInput	(A)				
LEAVE						

```
.intel_syntax noprefix
.extern calloc
.extern printf
.extern scanf
.global main
```

```
.section .bss
niz:
    .quad 0x0
nizch:
    .quad 0x0
```

```
.section .rodata
write_character_format:
    .asciz "%c"
write_number_format:
    .asciz "%d"
read_character_format:
    .asciz "%c"
read_number_format:
    .asciz "%d"
```

```
.section .text.builtin
ord_0:
    PUSH rbp
    MOV rbp, rsp
    MOVSX rax, dil
    LEAVE
    RET
```

chr_0:

```
PUSH rbp
MOV rbp, rsp
MOV al, dil
LEAVE
RET
```

```
.section .text
incInput_1:
```

```
PUSH rbp
MOV rbp, rsp
SUB rsp, 16
pushq rbx
pushq r12
pushq r13
pushq r14
pushq r15
sub rsp, 24
```

```
LEA rdi, [rip +
read_number_format]
LEA rsi, DWORD PTR [RBP - 4]
XOR eax, eax
CALL scanf
MOV eax, DWORD PTR [RBP - 4]
MOV esi, eax
LEA rdi, [rip +
write_number_format]
XOR eax, eax
```

```

CALL printf
MOV ebx, DWORD PTR [RBP - 4]
MOV ecx, 1
ADD ebx, ecx
MOV DWORD PTR [RBP - 4], ebx
MOV esi, ebx
LEA rdi, [rip +
write_number_format]
XOR eax, eax
CALL printf

add rsp, 24
popq r15
popq r14
popq r13
popq r12
popq rbx
LEAVE
RET
main:
PUSH rbp
MOV rbp, rsp
SUB rsp, 112
pushq rbx
pushq r12
pushq r13
pushq r14
pushq r15
sub rsp, 24

MOV eax, 0
MOV ebx, 1
ADD eax, ebx
MOV ecx, 1
MOV DWORD PTR [RBP - 4], eax
ADD eax, ecx
MOV DWORD PTR [RBP - 92], eax
MOV edx, 5
MOV DWORD PTR [RBP - 4], eax
IMUL eax, edx
MOV DWORD PTR [RBP - 44], eax
MOV edi, 1
SUB eax, edi
MOV DWORD PTR [RBP - 4], eax
MOV esi, eax
LEA rdi, [rip +
write_number_format]
XOR eax, eax
CALL printf
MOV rdi, 3
MOV rsi, 4
XOR eax, eax
CALL calloc
MOV niz, rax

MOV rsi, niz
MOV r8, 0
MOV r9d, 1
MOV DWORD PTR [rsi + 4 * r8], r9d
MOV r10, 1
MOV r11d, 2
MOV DWORD PTR [rsi + 4 * r10],
r11d

```

```

MOV r12, 1
MOV r13d, DWORD PTR [rsi + 4 *
r12]
MOV r14, 0
MOV r15d, DWORD PTR [rsi + 4 *
r14]
MOV eax, DWORD PTR [rsi + 4 * r15]
MOV ebx, 3
MOV DWORD PTR [RBP - 61], eax
IMUL eax, ebx
MOV DWORD PTR [rsi + 4 * r13], eax
MOV rcx, 2
MOV edx, DWORD PTR [rsi + 4 * rcx]
MOV rdi, 0
MOV DWORD PTR [RBP - 20], eax
MOV eax, DWORD PTR [rsi + 4 * rdi]
MOV DWORD PTR [RBP - 48], edx
MOV DWORD PTR [RBP - 8], eax
MOV eax, DWORD PTR [RBP - 48]
CDQ
MOV ebx, DWORD PTR [RBP - 8]
IDIV ebx
MOV ecx, eax
MOV DWORD PTR [RBP - 68], ecx
MOV DWORD PTR [RBP - 28], r15d
MOV DWORD PTR [RBP - 32], r13d
MOV DWORD PTR [RBP - 4], ecx
MOV esi, ecx
LEA rdi, [rip +
write_number_format]
XOR eax, eax
CALL printf
MOV rax, niz
MOV rbx, 2
MOV ecx, DWORD PTR [rax + 4 * rbx]
MOV DWORD PTR [RBP - 88], ecx
MOV esi, ecx
LEA rdi, [rip +
write_number_format]
XOR eax, eax
CALL printf
MOV rdi, 3
MOV rsi, 1
XOR eax, eax
CALL calloc
MOV nizch, rax

MOV rdx, nizch
MOV rdi, 0
MOV sil, 97
MOV BYTE PTR [rdx + 1 * rdi], sil
MOV r8, 1
MOV r9b, 98
MOV BYTE PTR [rdx + 1 * r8], r9b
MOV r10d, 5
MOV r11d, 3
SUB r10d, r11d
MOV r12b, 99
MOV BYTE PTR [rdx + 1 * r10], r12b
MOV r13, 1
MOV r14b, BYTE PTR [rdx + 1 * r13]
MOV BYTE PTR [RBP - 77], r14b
MOV DWORD PTR [RBP - 57], r10d
MOV sil, r14b
LEA rdi, [rip +
write_character_format]

```

```

XOR eax, eax
CALL printf
MOV r15d, 1
MOV eax, 2
IMUL r15d, eax
MOV rbx, nizch
MOV cl, BYTE PTR [rbx + 1 * r15]
MOV BYTE PTR [RBP - 49], cl
MOV DWORD PTR [RBP - 96], r15d
MOV sil, cl
LEA rdi, [rip +
write_character_format]
XOR eax, eax
CALL printf
MOV edx, 5
MOV edi, 1
IMUL edx, edi
MOV esi, 1
MOV DWORD PTR [RBP - 40], edx
SUB edx, esi
MOV r8d, DWORD PTR [RBP - 4]
MOV DWORD PTR [RBP - 76], edx
IMUL edx, r8d
ADD r8d, edx
MOV DWORD PTR [RBP - 84], edx
MOV eax, 3
CDQ
MOV r9d, 2
IDIV r9d
MOV r10d, edx
MOV r11d, 3

```

```

MOV r12d, 2
IMUL r11d, r12d
MOV DWORD PTR [RBP - 12], r10d
ADD r10d, r11d
MOV r13d, 3
MOV DWORD PTR [RBP - 24], r10d
SUB r10d, r13d
MOV DWORD PTR [RBP - 72], r8d
SUB r8d, r10d
MOV DWORD PTR [RBP - 36], r8d
MOV DWORD PTR [RBP - 16], r11d
MOV DWORD PTR [RBP - 4], r8d
MOV DWORD PTR [RBP - 53], r10d
MOV esi, r8d
LEA rdi, [rip +
write_number_format]
XOR eax, eax
CALL printf
CALL incInput_1

add rsp, 24
popq r15
popq r14
popq r13
popq r12
popq rbx
LEAVE
RET

```

7.2 Пример објектно оријентисаног програма

Дат је МикроЈава изворни код тест програма, представа у међујезику, као и излазни асемблерски кôд.

```
program MJProgram
    const int size = 10;

    class Point
    {
        int x, y;
        {
            void setX(int x) { this.x
= x; }
            void setY(int y) { this.y
= y; }

            void toString()
            {
                print('(');
                print(x); print(',');
print(' '); print(y);
                print(')');
            }
        }

        abstract class Shape
        {
            int r;
            Point point;
            {
                void place(int x, int y,
int r)
                {
                    point = new Point;
                    point.setX(x);
                    point.setY(y);
                    this.r = r;
                }

                abstract int O();
                abstract int P();
                abstract void toString();
            }

            class Circle extends Shape
            {
                {
                    int O() { return 2 * r *
3; }
                    int P() { return r * r *
3; }

                    void toString()
                    {
                        print('C');
print('i'); print('r');
print('c'); print('l');

                        print('e');
                        print('(');
                        print(r); print(',');
print(' '); point.toString();
                        print(')');
                    }
                }

                class Square extends Shape
                {
                    {
                        int O() { return 4 * r; }
                        int P() { return r * r; }

                        void toString()
                        {
                            print('S');
print('q'); print('u');
print('a'); print('r');
print('e');

                            print('(');
                            print(r); print(',');
print(' '); point.toString();
                            print(')');
                        }
                    }

                    Shape shapes[];
                    int index, O, P;
                    {
                        void main()
                        {
                            int i;
                            int x, y, z;
                            char choice;

                            {
                                shapes = new Shape[size];

                                read(choice);
                                for (; choice != '.' &&
index < size;)
                                {
                                    if (choice == 'c' ||
choice == 'C')
                                    {
                                        read(x); read(y);
                                        read(z);
                                        shapes[index] = new
Circle;
                                        shapes[index].place(x,
y, z);
                                    }
                                    else if (choice == 's' ||
choice == 'S')

```

```

        {
            read(x); read(y);
        }
read(z);
        shapes[index] = new
Square;
        shapes[index].place(x,
y, z);
    }
    else break;

    O = O +
shapes[index].O();
    P = P +
shapes[index].P();

    index++;
    read(choice);
}

for (i = 0; i < index; i++)
{
    shapes[i].toString();
    print(chr(10));
}

if (index > 0)
{
    O = O / index;
    P = P / index;
}

print('O'); print(' ');
print('='); print(' '); print(O);
print(chr(10));
print('P'); print(' ');
print('='); print(' '); print(P);
}
}

```

GEN_LABEL	setX				
ENTER	32				
GET_PTR	this	(A)	x	(A)	t1
STORE	x	(A)	PTR		t1
LEAVE					(D)

GEN_LABEL	setY				
ENTER	32				
GET_PTR	this	(A)	y	(A)	t2
STORE	y	(A)	PTR		t2
LEAVE					(D)

GEN_LABEL	toString				
ENTER	48				
PRINTF	%c		40	(A)	
GET_PTR	this	(A)	x	(A)	t3
LOAD	t3	(D)			t4
PRINTF	%d		t4	(D)	
PRINTF	%c		44	(A)	
PRINTF	%c		32	(A)	
GET_PTR	this	(A)	y	(A)	t5
LOAD	t5	(D)			t6
PRINTF	%d		t6	(D)	
PRINTF	%c		41	(A)	
LEAVE					

GEN_LABEL	place				
ENTER	80				
GET_PTR	this	(A)	point	(A)	t7
MALLOC	Point	(A)	PTR		t7
GET_PTR	this	(A)	point	(A)	t8
LOAD	t8	(D)			t9
PARAM	t9	(D)			
PARAM	x	(A)			
INVOKE_VIRTUAL	setX	(A)			
GET_PTR	this	(A)	point	(A)	t10
LOAD	t10	(D)			t11
PARAM	t11	(D)			
PARAM	y	(A)			
INVOKE_VIRTUAL	setY	(A)			
GET_PTR	this	(A)	r	(A)	t12

STORE	r	(A)	PTR	t12	(D)
LEAVE					

GEN_LABEL	0				
ENTER	32				
GET_PTR	this	(A)	r	(A)	t13 (A)
LOAD	t13	(D)			t14 (A)
MUL	2	(A)	t14	(D)	t15 (A)
MUL	t15	(D)	3	(A)	t16 (A)
RETURN	t16	(D)			
JMP					CGL_0
GEN_LABEL	CGL_0				
LEAVE					

GEN_LABEL	P				
ENTER	48				
GET_PTR	this	(A)	r	(A)	t17 (A)
LOAD	t17	(D)			t18 (A)
GET_PTR	this	(A)	r	(A)	t19 (A)
LOAD	t19	(D)			t20 (A)
MUL	t18	(D)	t20	(D)	t21 (A)
MUL	t21	(D)	3	(A)	t22 (A)
RETURN	t22	(D)			
JMP					CGL_1
GEN_LABEL	CGL_1				
LEAVE					

GEN_LABEL	toString				
ENTER	48				
PRINTF	%c		67	(A)	
PRINTF	%c		105	(A)	
PRINTF	%c		114	(A)	
PRINTF	%c		99	(A)	
PRINTF	%c		108	(A)	
PRINTF	%c		101	(A)	
PRINTF	%c		40	(A)	
GET_PTR	this	(A)	r	(A)	t23 (A)
LOAD	t23	(D)			t24 (A)
PRINTF	%d		t24	(D)	
PRINTF	%c		44	(A)	
PRINTF	%c		32	(A)	
GET_PTR	this	(A)	point	(A)	t25 (A)
LOAD	t25	(D)			t26 (A)
PARAM	t26	(D)			
INVOKE_VIRTUAL	toString	(A)			
PRINTF	%c		41	(A)	
LEAVE					

GEN_LABEL	0				
ENTER	32				
GET_PTR	this	(A)	r	(A)	t27 (A)
LOAD	t27	(D)			t28 (A)
MUL	4	(A)	t28	(D)	t29 (A)
RETURN	t29	(D)			
JMP					CGL_2
GEN_LABEL	CGL_2				
LEAVE					

GEN_LABEL	P				
ENTER	48				
GET_PTR	this	(A)	r	(A)	t30 (A)

LOAD	t30	(D)		t31	(A)
GET_PTR	this	(A)	r	t32	(A)
LOAD	t32	(D)		t33	(A)
MUL	t31	(D)	t33	t34	(A)
RETURN	t34	(D)			
JMP				CGL_3	
GEN_LABEL	CGL_3				
LEAVE					

GEN_LABEL	toString				
ENTER	48				
PRINTF	%c		83	(A)	
PRINTF	%c		113	(A)	
PRINTF	%c		117	(A)	
PRINTF	%c		97	(A)	
PRINTF	%c		114	(A)	
PRINTF	%c		101	(A)	
PRINTF	%c		40	(A)	
GET_PTR	this	(A)	r	t35	(A)
LOAD	t35	(D)		t36	(A)
PRINTF	%d		t36	(D)	
PRINTF	%c		44	(A)	
PRINTF	%c		32	(A)	
GET_PTR	this	(A)	point	t37	(A)
LOAD	t37	(D)		t38	(A)
PARAM	t38	(D)			
INVOKE_VIRTUAL	toString	(A)			
PRINTF	%c		41	(A)	
LEAVE					

GEN_LABEL	main				
ENTER	112				
MALLOC	10		ARR		
SCANF			%c	shapes	(A)
GEN_LABEL	CGL_4			choice	(A)
JE	choice	(A)	46	(A)	CGL_5
JGE	index	(N/A)	10	(N/A)	CGL_5
JNE	choice	(N/A)	99	(N/A)	L0
JMP					L1
GEN_LABEL	L0				
JNE	choice	(A)	67	(A)	L2
GEN_LABEL	L1				
SCANF			%d	x	(A)
SCANF			%d	y	(A)
SCANF			%d	z	(A)
MALLOC	Circle	(A)		t39	(A)
ASTORE	t39	(D)	index	shapes	(A)
ALOAD	shapes	(A)	index	t40	(A)
PARAM	t40	(D)			
PARAM	x	(A)			
PARAM	y	(A)			
PARAM	z	(A)			
INVOKE_VIRTUAL	place	(A)			
JMP					L7
GEN_LABEL	L2				
JNE	choice	(A)	115	(A)	L3
JMP					L4
GEN_LABEL	L3				
JNE	choice	(A)	83	(A)	L5
GEN_LABEL	L4				
SCANF			%d	x	(A)
SCANF			%d	y	(A)
SCANF			%d	z	(A)
MALLOC	Square	(A)		t41	(A)

ASTORE	t41	(D)	index	(A)	shapes	(A)
ALOAD	shapes	(A)	index	(A)	t42	(A)
PARAM	t42	(D)				
PARAM	x	(A)				
PARAM	y	(A)				
PARAM	z	(A)				
INVOKE_VIRTUAL	place	(A)				
JMP					L6	
GEN_LABEL	L5					
JMP					CGL_5	
GEN_LABEL	L6					
GEN_LABEL	L7					
ALOAD	shapes	(A)	index	(A)	t43	(A)
PARAM	t43	(D)				
INVOKE_VIRTUAL	0	(A)			t44	(A)
ADD	0	(D)	t44	(D)	t45	(A)
STORE	t45	(D)			0	(A)
ALOAD	shapes	(A)	index	(A)	t46	(A)
PARAM	t46	(D)				
INVOKE_VIRTUAL	P	(A)			t47	(A)
ADD	P	(D)	t47	(D)	t48	(A)
STORE	t48	(D)			P	(A)
ADD	index	(A)	1	(A)	index	(A)
SCANF			%c		choice	(A)
JMP					CGL_4	
GEN_LABEL	CGL_5					
STORE	0	(A)			i	(A)
GEN_LABEL	CGL_6					
JGE	i	(A)	index	(A)	CGL_7	
ALOAD	shapes	(N/A)	i	(N/A)	t49	(N/A)
PARAM	t49	(D)				
INVOKE_VIRTUAL	toString	(A)				
PARAM	10	(A)				
CALL	chr	(A)			t50	(A)
PRINTF	%c		t50	(D)		
GEN_LABEL	L8					
ADD	i	(A)	1	(A)	i	(A)
JMP					CGL_6	
GEN_LABEL	CGL_7					
JLE	index	(A)	0	(A)	L9	
DIV	0	(N/A)	index	(N/A)	t51	(N/A)
STORE	t51	(D)			0	(A)
DIV	P	(D)	index	(A)	t52	(A)
STORE	t52	(D)			P	(A)
GEN_LABEL	L9					
PRINTF	%c		79	(A)		
PRINTF	%c		32	(A)		
PRINTF	%c		61	(A)		
PRINTF	%c		32	(A)		
PRINTF	%d		0	(A)		
PARAM	10	(A)				
CALL	chr	(A)			t53	(A)
PRINTF	%c		t53	(D)		
PRINTF	%c		80	(A)		
PRINTF	%c		32	(A)		
PRINTF	%c		61	(A)		
PRINTF	%c		32	(A)		
PRINTF	%d		P	(A)		
LEAVE						

```
.intel_syntax noprefix
.extern calloc
.extern printf
.extern scanf
.global main
```

```
.section .bss
shapes:
    .quad 0x0
P:
    .long 0x0
index:
```



```

        .long 0x0
O:      .long 0x0

.section .rodata
write_character_format:
        .asciz "%c"
write_number_format:
        .asciz "%d"
read_character_format:
        .asciz "%c"
read_number_format:
        .asciz "%d"

_vft_Point:
        .quad setX_1
        .quad setY_1
        .quad toString_1

_vft_Circle:
        .quad place_1
        .quad O_1
        .quad P_1
        .quad toString_2

_vft_Square:
        .quad place_1
        .quad O_2
        .quad P_2
        .quad toString_3

.section .text.builtin
ord_0:
        PUSH rbp
        MOV rbp, rsp
        MOVSX rax, dil
        LEAVE
        RET

chr_0:
        PUSH rbp
        MOV rbp, rsp
        MOV al, dil
        LEAVE
        RET

.section .text
setX_1:
        PUSH rbp
        MOV rbp, rsp
        SUB rsp, 32
        pushq rbx
        pushq r12
        pushq r13
        pushq r14
        pushq r15
        sub rsp, 24

        MOV QWORD PTR [RBP - 8], rdi
        MOV DWORD PTR [RBP - 20], esi

        MOV rax, QWORD PTR [RBP - 8]
        ADD rax, 8
        MOV ebx, DWORD PTR [RBP - 20]
        MOV QWORD PTR [rax], rbx

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET

toString_1:
        PUSH rbp
        MOV rbp, rsp
        SUB rsp, 48
        pushq rbx
        pushq r12
        pushq r13
        pushq r14
        pushq r15
        sub rsp, 24

        MOV QWORD PTR [RBP - 12], rdi

        MOV al, 40
        MOV sil, al
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV rbx, QWORD PTR [RBP - 12]
        ADD rbx, 8
        MOV rbx, QWORD PTR [rbx]
        MOV DWORD PTR [RBP - 16], ebx
        MOV esi, ebx
        LEA rdi, [rip +
write_number_format]
        XOR eax, eax
        CALL printf
        MOV cl, 44

```

```

        MOV sil, cl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV dl, 32
        MOV sil, dl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV rdi, QWORD PTR [RBP - 12]
        ADD rdi, 12
        MOV rdi, QWORD PTR [rdi]
        MOV DWORD PTR [RBP - 4], edi
        MOV esi, edi
        LEA rdi, [rip +
write_number_format]
        XOR eax, eax
        CALL printf
        MOV sil, 41
        MOV sil, sil
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET
place_1:
        PUSH rbp
        MOV rbp, rsp
        SUB rsp, 80
        pushq rbx
        pushq r12
        pushq r13
        pushq r14
        pushq r15
        sub rsp, 24

        MOV QWORD PTR [RBP - 12], rdi
        MOV DWORD PTR [RBP - 4], esi
        MOV DWORD PTR [RBP - 60], edx
        MOV DWORD PTR [RBP - 76], ecx

        MOV rax, QWORD PTR [RBP - 12]
        ADD rax, 16
        MOV QWORD PTR [RBP - 24], rax
        MOV rdi, 16
        MOV rsi, 1
        XOR eax, eax
        CALL calloc
        MOV QWORD PTR [rax], OFFSET
_vft_Point
        MOV rbx, QWORD PTR [RBP - 24]
        MOV [rbx], rax

        MOV rcx, QWORD PTR [RBP - 12]
        ADD rcx, 16
        MOV rcx, QWORD PTR [rcx]

        MOV rdi, rcx
        MOV esi, DWORD PTR [RBP - 4]
        MOV QWORD PTR [RBP - 56], rdi
        MOV rax, QWORD PTR [RBP - 56]
        MOV rax, [rax]
        CALL [rax + 8 * 0]

        MOV rdx, QWORD PTR [RBP - 12]
        ADD rdx, 16
        MOV rdx, QWORD PTR [rdx]
        MOV rdi, rdx
        MOV esi, DWORD PTR [RBP - 60]
        MOV QWORD PTR [RBP - 40], rdi
        MOV rax, QWORD PTR [RBP - 40]
        MOV rax, [rax]
        CALL [rax + 8 * 1]

        MOV rdi, QWORD PTR [RBP - 12]
        ADD rdi, 8
        MOV esi, DWORD PTR [RBP - 76]
        MOV QWORD PTR [rdi], rsi

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET
O_1:
        PUSH rbp
        MOV rbp, rsp
        SUB rsp, 32
        pushq rbx
        pushq r12
        pushq r13
        pushq r14
        pushq r15
        sub rsp, 24

        MOV QWORD PTR [RBP - 16], rdi

        MOV rax, QWORD PTR [RBP - 16]
        ADD rax, 8
        MOV rax, QWORD PTR [rax]
        MOV ebx, 2
        IMUL ebx, eax
        MOV ecx, 3
        MOV DWORD PTR [RBP - 8], ebx
        IMUL ebx, ecx
        MOV eax, ebx
        JMP CGL_0
CGL_0:

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET
P_1:
        PUSH rbp

```

```

MOV rbp, rsp
SUB rsp, 48
pushq rbx
pushq r12
pushq r13
pushq r14
pushq r15
sub rsp, 24

MOV QWORD PTR [RBP - 28], rdi

MOV rax, QWORD PTR [RBP - 28]
ADD rax, 8
MOV rax, QWORD PTR [rax]
MOV rbx, QWORD PTR [RBP - 28]
ADD rbx, 8
MOV rbx, QWORD PTR [rbx]
MOV DWORD PTR [RBP - 20], eax
IMUL eax, ebx
MOV ecx, 3
MOV DWORD PTR [RBP - 4], eax
IMUL eax, ecx
MOV DWORD PTR [RBP - 44], ebx
MOV ebx, eax
MOV eax, ebx
JMP CGL_1
CGL_1:

    add rsp, 24
    popq r15
    popq r14
    popq r13
    popq r12
    popq rbx
    LEAVE
    RET
toString_2:
    PUSH rbp
    MOV rbp, rsp
    SUB rsp, 48
    pushq rbx
    pushq r12
    pushq r13
    pushq r14
    pushq r15
    sub rsp, 24

    MOV QWORD PTR [RBP - 16], rdi

    MOV al, 67
    MOV sil, al
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV bl, 105
    MOV sil, bl
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV cl, 114
    MOV sil, cl
    LEA rdi, [rip +
write_character_format]

    XOR eax, eax
    CALL printf
    MOV dl, 99
    MOV sil, dl
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV dil, 108
    MOV sil, dil
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV sil, 101
    MOV sil, sil
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r8b, 40
    MOV sil, r8b
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r9, QWORD PTR [RBP - 16]
    ADD r9, 8
    MOV r9, QWORD PTR [r9]
    MOV DWORD PTR [RBP - 20], r9d
    MOV esi, r9d
    LEA rdi, [rip +
write_number_format]
    XOR eax, eax
    CALL printf
    MOV r10b, 44
    MOV sil, r10b
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r11b, 32
    MOV sil, r11b
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r12, QWORD PTR [RBP - 16]
    ADD r12, 16
    MOV r12, QWORD PTR [r12]
    MOV rdi, r12
    MOV QWORD PTR [RBP - 28], rdi
    MOV r13, QWORD PTR [RBP - 28]
    MOV r13, [r13]
    CALL [r13 + 8 * 2]

    MOV r14b, 41
    MOV sil, r14b
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf

    add rsp, 24
    popq r15
    popq r14
    popq r13

```

```

    popq r12
    popq rbx
    LEAVE
    RET
O_2:
    PUSH rbp
    MOV rbp, rsp
    SUB rsp, 32
    pushq rbx
    pushq r12
    pushq r13
    pushq r14
    pushq r15
    sub rsp, 24

    MOV QWORD PTR [RBP - 12], rdi

    MOV rax, QWORD PTR [RBP - 12]
    ADD rax, 8
    MOV rax, QWORD PTR [rax]
    MOV ebx, 4
    IMUL ebx, eax
    MOV eax, ebx
    JMP CGL_2
CGL_2:

    add rsp, 24
    popq r15
    popq r14
    popq r13
    popq r12
    popq rbx
    LEAVE
    RET
P_2:
    PUSH rbp
    MOV rbp, rsp
    SUB rsp, 48
    pushq rbx
    pushq r12
    pushq r13
    pushq r14
    pushq r15
    sub rsp, 24

    MOV QWORD PTR [RBP - 16], rdi

    MOV rax, QWORD PTR [RBP - 16]
    ADD rax, 8
    MOV rax, QWORD PTR [rax]
    MOV rbx, QWORD PTR [RBP - 16]
    ADD rbx, 8
    MOV rbx, QWORD PTR [rbx]
    MOV DWORD PTR [RBP - 32], eax
    IMUL eax, ebx
    MOV DWORD PTR [RBP - 28], ebx
    MOV ebx, eax
    MOV eax, ebx
    JMP CGL_3
CGL_3:

    add rsp, 24
    popq r15
    popq r14

    popq r13
    popq r12
    popq rbx
    LEAVE
    RET
toString_3:
    PUSH rbp
    MOV rbp, rsp
    SUB rsp, 48
    pushq rbx
    pushq r12
    pushq r13
    pushq r14
    pushq r15
    sub rsp, 24

    MOV QWORD PTR [RBP - 24], rdi

    MOV al, 83
    MOV sil, al
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV bl, 113
    MOV sil, bl
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV cl, 117
    MOV sil, cl
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV dl, 97
    MOV sil, dl
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV dil, 114
    MOV sil, dil
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r8b, 40
    MOV sil, r8b
    LEA rdi, [rip +
write_character_format]
    XOR eax, eax
    CALL printf
    MOV r9, QWORD PTR [RBP - 24]
    ADD r9, 8
    MOV r9, QWORD PTR [r9]
    MOV DWORD PTR [RBP - 28], r9d
    MOV esi, r9d
    LEA rdi, [rip +
write_number_format]

```

```

        XOR eax, eax
        CALL printf
        MOV r10b, 44
        MOV sil, r10b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r11b, 32
        MOV sil, r11b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r12, QWORD PTR [RBP - 24]
        ADD r12, 16
        MOV r12, QWORD PTR [r12]
        MOV rdi, r12
        MOV QWORD PTR [RBP - 40], rdi
        MOV r13, QWORD PTR [RBP - 40]
        MOV r13, [r13]
        CALL [r13 + 8 * 2]

        MOV r14b, 41
        MOV sil, r14b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET
main:
        PUSH rbp
        MOV rbp, rsp
        SUB rsp, 112
        pushq rbx
        pushq r12
        pushq r13
        pushq r14
        pushq r15
        sub rsp, 24

        MOV rdi, 10
        MOV rsi, 8
        XOR eax, eax
        CALL calloc
        MOV shapes, rax

        LEA rdi, [rip +
read_character_format]
        LEA rsi, BYTE PTR [RBP - 17]
        XOR eax, eax
        CALL scanf
CGL_4:
        MOV al, BYTE PTR [RBP - 17]
        MOV bl, 46
        CMP al, bl
        JE CGL_5

        MOV eax, index
        MOV ebx, 10
        CMP eax, ebx
        JGE CGL_5
        MOV al, BYTE PTR [RBP - 17]
        MOV bl, 99
        CMP al, bl
        JNE L0
        JMP L1
L0:
        MOV al, BYTE PTR [RBP - 17]
        MOV bl, 67
        CMP al, bl
        JNE L2
L1:
        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 8]
        XOR eax, eax
        CALL scanf
        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 12]
        XOR eax, eax
        CALL scanf
        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 16]
        XOR eax, eax
        CALL scanf
        MOV rdi, 20
        MOV rsi, 1
        XOR eax, eax
        CALL calloc
        MOV QWORD PTR [rax], OFFSET
_vft_Circle
        MOV QWORD PTR [RBP - 30], rax

        MOV rax, shapes
        MOVSXD rbx, index
        MOV rcx, QWORD PTR [RBP - 30]
        MOV QWORD PTR [rax + 8 * rbx],
rcx
        MOV rdx, QWORD PTR [rax + 8 *
rbx]
        MOV rdi, rdx
        MOV esi, DWORD PTR [RBP - 8]
        MOV QWORD PTR [RBP - 40], rdi
        MOV edx, DWORD PTR [RBP - 12]
        MOV QWORD PTR [RBP - 30], rcx
        MOV ecx, DWORD PTR [RBP - 16]
        MOV rax, QWORD PTR [RBP - 40]
        MOV rax, [rax]
        CALL [rax + 8 * 0]

        JMP L7
L2:
        MOV al, BYTE PTR [RBP - 17]
        MOV bl, 115
        CMP al, bl
        JNE L3
        JMP L4
L3:
        MOV al, BYTE PTR [RBP - 17]
        MOV bl, 83
        CMP al, bl
        JNE L5
L4:

```

```

        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 8]
        XOR eax, eax
        CALL scanf
        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 12]
        XOR eax, eax
        CALL scanf
        LEA rdi, [rip +
read_number_format]
        LEA rsi, DWORD PTR [RBP - 16]
        XOR eax, eax
        CALL scanf
        MOV rdi, 20
        MOV rsi, 1
        XOR eax, eax
        CALL calloc
        MOV QWORD PTR [rax], OFFSET
_vft_Square
        MOV QWORD PTR [RBP - 96], rax

        MOV rax, shapes
        MOVSXD rbx, index
        MOV rcx, QWORD PTR [RBP - 96]
        MOV QWORD PTR [rax + 8 * rbx],
rcx
        MOV rdx, QWORD PTR [rax + 8 *
rbx]
        MOV rdi, rdx
        MOV esi, DWORD PTR [RBP - 8]
        MOV QWORD PTR [RBP - 108], rdi
        MOV edx, DWORD PTR [RBP - 12]
        MOV QWORD PTR [RBP - 96], rcx
        MOV ecx, DWORD PTR [RBP - 16]
        MOV rax, QWORD PTR [RBP - 108]
        MOV rax, [rax]
        CALL [rax + 8 * 0]

        JMP L6
L5:
        JMP CGL_5
L6:
L7:
        MOV rax, shapes
        MOVSXD rbx, index
        MOV rcx, QWORD PTR [rax + 8 *
rbx]
        MOV rdi, rcx
        MOV QWORD PTR [RBP - 72], rdi
        MOV rax, QWORD PTR [RBP - 72]
        MOV rax, [rax]
        CALL [rax + 8 * 1]
        MOV DWORD PTR [RBP - 21], eax

        MOV edx, 0
        MOV edi, DWORD PTR [RBP - 21]
        ADD edx, edi
        MOV DWORD PTR [RBP - 56], edx
        MOV rsi, shapes
        MOVSXD r8, index
        MOV r9, QWORD PTR [rsi + 8 *
r8]
        MOV DWORD PTR [RBP - 21], edi
        MOV rdi, r9
        MOV QWORD PTR [RBP - 64], rdi
        MOV O, edx

        MOV r10, QWORD PTR [RBP - 64]
        MOV r10, [r10]
        CALL [r10 + 8 * 2]
        MOV DWORD PTR [RBP - 100], eax

        MOV r11d, P
        MOV r12d, DWORD PTR [RBP - 100]
        ADD r11d, r12d
        MOV DWORD PTR [RBP - 76], r11d
        MOV r13d, index
        MOV r14d, 1
        ADD r13d, r14d
        MOV P, r11d
        MOV index, r13d
        LEA rdi, [rip +
read_character_format]
        LEA rsi, BYTE PTR [RBP - 17]
        XOR eax, eax
        CALL scanf
        JMP CGL_4
CGL_5:
        MOV eax, 0
        MOV DWORD PTR [RBP - 4], eax
CGL_6:
        MOV eax, DWORD PTR [RBP - 4]
        MOV ebx, index
        CMP eax, ebx
        JGE CGL_7
        MOV rax, shapes
        MOVSXD rbx, DWORD PTR [RBP - 4]
        MOV rcx, QWORD PTR [rax + 8 *
rbx]
        MOV rdi, rcx
        MOV QWORD PTR [RBP - 88], rdi
        MOV rax, QWORD PTR [RBP - 88]
        MOV rax, [rax]
        CALL [rax + 8 * 3]

        MOV edi, 10
        CALL chr_0
        MOV BYTE PTR [RBP - 41], al

        MOV dl, BYTE PTR [RBP - 41]
        MOV sil, dl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
L8:
        MOV edi, DWORD PTR [RBP - 4]
        MOV esi, 1
        ADD edi, esi
        MOV DWORD PTR [RBP - 4], edi
        JMP CGL_6
CGL_7:
        MOV eax, index
        MOV ebx, 0
        CMP eax, ebx
        JLE L9
        MOV eax, 0
        CDQ
        MOV ebx, index
        IDIV ebx
        MOV ecx, eax
        MOV DWORD PTR [RBP - 52], ecx
        MOV O, ecx
        MOV eax, P
        CDQ

```

```

        IDIV ebx
        MOV ecx, eax
        MOV DWORD PTR [RBP - 45], ecx
        MOV P, ecx
L9:
        MOV al, 79
        MOV sil, al
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV bl, 32
        MOV sil, bl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV cl, 61
        MOV sil, cl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV dl, 32
        MOV sil, dl
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV edi, 0
        MOV esi, edi
        LEA rdi, [rip +
write_number_format]
        XOR eax, eax
        CALL printf
        MOV edi, 10
        CALL chr_0
        MOV BYTE PTR [RBP - 22], al

        MOV sil, BYTE PTR [RBP - 22]
        MOV sil, sil
        LEA rdi, [rip +
write_character_format]

        XOR eax, eax
        CALL printf
        MOV r8b, 80
        MOV sil, r8b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r9b, 32
        MOV sil, r9b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r10b, 61
        MOV sil, r10b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r11b, 32
        MOV sil, r11b
        LEA rdi, [rip +
write_character_format]
        XOR eax, eax
        CALL printf
        MOV r12d, P
        MOV esi, r12d
        LEA rdi, [rip +
write_number_format]
        XOR eax, eax
        CALL printf

        add rsp, 24
        popq r15
        popq r14
        popq r13
        popq r12
        popq rbx
        LEAVE
        RET

```

7.3 Граматика језика и лексичке структуре

Program	= "program" ident {ConstDecl VarDecl AbstractClassDecl ClassDecl } "{" {MethodDecl } "}"
ConstDecl	= "const" Type ident "=" (numConst charConst boolConst) {, ident "=" (numConst charConst boolConst) } ";"
VarDecl	= Type ident "[" "[" "]" "{" ident "[" "[" "]" } ";"
ClassDecl	= "class" ident ["extends" Type] "{" {VarDecl} [{" {MethodDecl} "}"] "}"
AbstractClassDecl	= "abstract" "class" ident ["extends" Type] "{" {VarDecl} [{" {MethodDecl} AbstractMethodDecl} "}"] "}"
MethodDecl	= (Type "void") ident "(" {FormPars} ")" {VarDecl} "{" {Statement} "}"
AbstractMethodDecl	= "abstract" (Type "void") ident "(" {FormPars} ")" ";"
FormPars	= Type ident "[" "]" "{" ident "[" "]" }
Type	= ident
Statement	= DesignatorStatement ";" "if" "(" Condition ")" Statement ["else" Statement] "for" "(" {DesignatorStatement } ";" {Condition} ";" {DesignatorStatement } ")" Statement "break" ";" "continue" ";" "return" {Expr} ";" "read" "(" Designator ")" ";" "print" "(" Expr {, numConst} ")" ";" "{" {Statement} "}"
DesignatorStatement	= Designator (Assignop Expr "(" {ActPars} ")" "++" "--")
ActPars	= Expr {, Expr}
Condition	= CondTerm {" " CondTerm}
CondTerm	= CondFact {"&&" CondFact}
CondFact	= Expr {Relop Expr}
Expr	= ["-"] Term {Addop Term}
Term	= Factor {Mulop Factor}
Factor	= Designator [{" {ActPars} "}] numConst charConst boolConst "new" Type [{" Expr }]" "(" Expr ")"
Designator	= ident ["." ident [{" Expr }"]]
Assignop	= "="
Relop	= "==" "!=" ">" ">=" "<" "<="
Addop	= "+" "-"
Mulop	= "*" "/" "%"

Ključne reči:	program, break, class, abstract, else, const, if, new, print, read, return, void, for, extends, continue
Vrste tokena :	ident = letter {letter digit "_"}. numConst = digit {digit}. charConst = "" printableChar "" boolConst = ("true" "false").
Operatori:	+, -, *, /, %, ==, !=, >, >=, <, <=, &&, , =, ++, --, ::, zarez, ., (,), [,], {, }
Komentari:	// do kraja linije

8. 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 1”*, 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. Torczon, *Engineering a Compiler*, Morgan Kaufmann, Burlington, 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.
- [TOP500] The TOP500 Project
<https://www.top500.org/lists/top500/2020/06/>
- [ZSR] Z. S. Rakić, *Materijali za predavanja i vežbe iz predmeta „Programski prevodioci”*, Univerzitet u Novom Sadu, Novi Sad.

9. Списак коришћених слика

- Слика 1 S. Cherubin, *Compiler design*, 2016, CC BY-SA 3.0
https://upload.wikimedia.org/wikipedia/commons/thumb/c/cc/Compiler_design.svg/1024px-Compiler_design.svg.png
- Слика 2, 3 A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 2006.