УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

ИНТЕРПРЕТЕР АПСТРАКТНОГ СИНТАКСНОГ СТАБЛА ЗА МИКРОЈАВА ПРОГРАМСКИ ЈЕЗИК

Дипломски рад

Ментор:
проф. др. Драган Бојић

Кандидат:
Страхиња Стефановић
2016/0130

Београд, август 2020.

САДРЖАЈ

1. УВОД

Виши програмски језици представљају фундаменталне алате у развоју данашњег софтвера. Из тог разлога је разумевање техника које се користе у имплементацији тих језика од изузетног значаја. Идеја о програмском преводиоцу (енг. *compiler*), програму који преводи изворни програмски кôд написан на неком вишем програмском језику у машински кôд за архитектуру неког конкретног процесора, је добро позната, али постоје и други начини имплеметације модерних програмских језика. Многи програмери доживљавају компајлере као мистериозне црне кутије које производе извршни кôд, али детаљнијим увидом у унутрашњу структуру и начин рада овог процеса, може се постићи његово ефикасније коришћење. [1]

Имплементација програмских језика је област која је темељно проучавана годинама и свакако је једна од најуспешнијих области рачунарства. Данашњи преводиоци су способни да генеришу високо оптимизован кôд из комплексних програма на вишем програмском језику и представљају екстремно сложене софтверске пакете. Разумевање шта и како они раде захтева одређено предзнање како из теорије формалних језика и теорије аутомата, тако и из архитектуре рачунара и алгоритама и структура података. Из тог разлога, писање програмског преводиоца има велики едукациони значај и даје нам сјајан увид у то како ће се програм понашати након што се покрене, где су могући губици у перформансама итд. [2]

Ради смањења комплексности, као изворни језик преводиоца изабран је језик МикроЈава, коришћен у настави на предмету Програмски преводиоци 1 на Електротехничком факултету у Београду. Као што и само име наговештава, МикроЈава је у много аспеката упрошћена верзија познатог програмског језика *Java*, али је, и поред тога, довољно сложен да прикаже све битне концепте израде једног типичног програмског преводиоца.

У овом раду представљен је један могући начин реализације интерпретера за МикроЈава језик који за међурепрезентацију кôда користи апстрактно синтаксно стабло осмишљено у великој мери по узору на оно из онлајн књиге *Crafting Interpreters* [3], а делимично се ослања и на сличну репрезентацију коју предлаже Appel у својој књизи [4].

У другом поглављу се даје мотивација за израду рада. Темељно се обрађују појмови процеса превођења (компајлирања) и процеса интерпретирања и разматрају разлике, предности и мане, између њих.

У трећем поглављу је детаљно објашњена сама имплементација интерпретера за језик МикроЈава уз поделу на потпоглавља која прате стандардне фазе кроз које пролази програм током превођења.

У четвртом поглављу се износи закључак и критички разматрају постигнути резултати на примеру извршавања једног конкретног МикроЈава програма.

2. МОТИВАЦИЈА

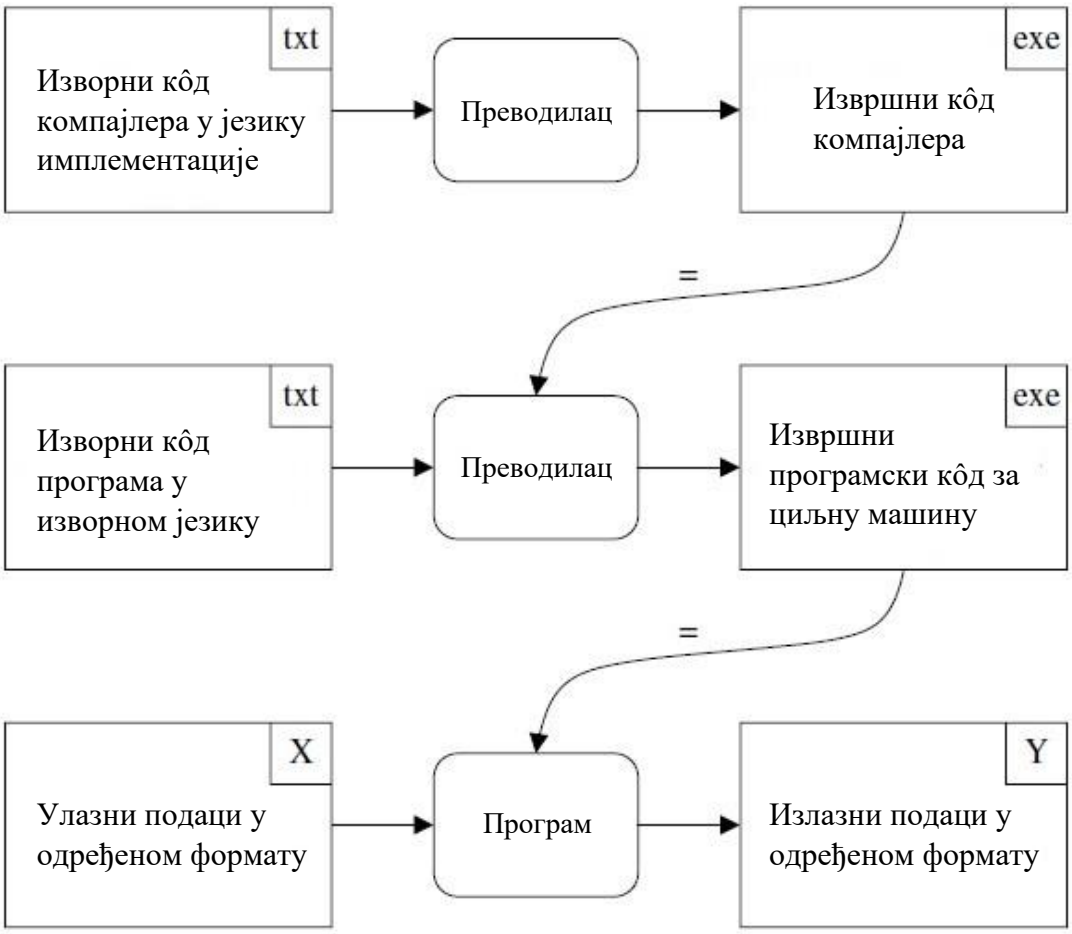
Опште говорећи, **компајлер** је програм који као улаз прихвата програм написан на одређеном језику, а као резултат враћа програм на неком другом језику уз очување функционалности самог програма. Тај процес назива се **превођење**, као да се ради о тексту на неком говорном језику. Готово сви преводиоци преводе са једног улазног, **изворног језика** на само један одређени излазни, **циљни језик**. Нормално је претпоставити да је разлика између ових језика велика, на пример, изворни језик може бити програмски језик *C*, док је циљни у виду машинског кôда неког одређеног процесора. Ипак, то и не мора да буде случај. Програмски језик којим је сâм преводилац написан представља **језик имплементације**. [5]

Поставља се питање, из ког разлога нам је овакав један програм уопште потребан? Заправо, разлог његовог постојања је да хардвер не може да разуме вишу семантику коју програмер користи, већ само машинске инструкције које подржава одређени процесор. Баш то је сврха компајлера, добити формат који хардвер може да разуме и изврши, што је на крају крајева и главни циљ рачунарства.

Овај дипломски рад је такође показатељ једног интересантног феномена. Да бисмо добили компајлер као извршни програм морамо најпре да позовемо компајлер имплементационог језика у којем је тај циљни компајлер написан (Слика 2.1). Конкретно у овом случају, интерпретер је добијен позивањем *Java* компајлера, а он ће се касније користити за извршавање МикроЈава програма. Свакако могућа је и ситуација када су изворни језик и језик имплементације исти, тада компајлер преводи новију верзију себе и овај процес се назива **bootstrapping**. [5]

На (Слика 2.2) приказана је типична структура једног преводиоца. Део који врши анализу улазног изворног кôда се назива предњи крај (**front-end**), а део који се бави синтезом циљног језика задњи крај (**back-end**). Добар дизајн компајлера би подразумевао да предњи крај нема представу о циљном језику, као и да задњи крај нема представу о каквом изворном језику се ради. Једина ствар која би их повезивала би било познавање семантичке репрезентације.

Овакав дизајн аутоматски нам наговештава и могућност постојања једног другачијег начина рада преводиоца: уколико су нам унапред познати сви улазни подаци, компајлер би могао да изврши акције одређене семантичком репрезентацијом уместо да их само изрази у другачијем облику. Тада се генерисање кôда у задњем крају замењује интерпретирањем, а такав програм се тада назива **интерпретер**. Постоји више разлога због којих би се користио овакав приступ.



Слика 2.1: Превођење и позив компајлера

Слика 2.2: Типична структура једног компајлера

Један од основних разлога је што су интерпретери подразумевано написани на неком вишем програмском језику и самим тим се могу извршавати на скоро свим машинама, док се изгенерисани објектни кôд може извршавати само на циљној архитектури. Другим речима, добијена је већа портабилност. Такође, програмирање интерпретера представља много мањи посао од писања целокупног задњег краја.

Извршавање акција непосредно из семантичке репрезентације омогућава и софистициранију проверу и пријаву грешака. Ово у основи и не мора бити сасвим тачно, али је последица чињенице да се од модерних преводилаца очекује да генеришу што ефикаснији кôд. Из тог разлога, већина генератора кôда одбацује бескорисне информације које не доприносе побољшању перформанси. У том процесу се губе и неке информације које би ипак биле значајне кориснику у дијагностиковању грешака (програмски кôд, број линије и сл.). Још једна предност која не мора нужно да буде на страни интерпретера је сигурност – претпоставка је да је лакше интегрисати злоћудни програм у бинарни кôд него очекивати да ће га интерпретер извршити.

Међутим, највећа предност је лакоћа са којом интерпретер излази на крај са новим програмским кôдом који генерише сâм извршни програм. Док интерпретер може да га посматра као и било који други део програма, преводилац мора да позове себе како би превео тај нови кôд, а потом и да га повеже са остатком извршног програма, уколико је тако нешто уопште могуће. Из претходног се може закључити да уколико програмски језик дозвољава генерисање новог кôда у време извршавања коришћење интерпретера је готово незаобилазно.

Са друге стране, иако на основу до сада наведеног интерпретација као решење делује прилично атрактивно оно са собом носи и одређене практичне проблеме. Први проблем се односи на перформансе. Ако се нека наредба извршава више пута (нпр. унутар петље) она ће бити и анализирана сваки пут пре њеног извршавања. Тако да ће цена интерпретације те наредбе бити много пута већа од извршавања пар машинских инструкција добијених превођењем. Међутим, ова цена се лако може драстично смањити анализирањем кôда само једном и његовим превођењем у неки међуоблик погодан за ефикаснију интерпретацију. И заиста, неки језици су и имплементирани на овај начин без обзира на режијске трошкове које уноси интерпретирање.

Други проблем представља потреба да интерпретер буде присутан у меморији у време извршавања поред самог изворног кôда (или међукôда) што за последицу има знатно већи меморијски отисак од оног који би имао преведени машински кôд. Код малих *embedded* система са веома ограниченом меморијом ово може да буде одлучујући недостатак.

Генерално посматрано, све имплементације програмских језика се на неки начин своде на интерпретирање, само је питање на ком нивоу. Код интерпретације изворног кôда, интерпретер је сложен зато што мора да анализира наредбе на високом нивоу апстракције и онда опонаша њихово извршавање. Код интерпретације међукôда, интерпретер је једноставнији зато што је анализа већ одрађена унапред. Док се код традиционалног приступа превођења генерисањем машинског кôда интерпретирање врши у потпуности од стране циљног хардвера и не постоји софтверска интерпретација, а самим тим не постоје ни додатни режијски трошкови. Детаљнијим сагледавањем ова три нивоа интерпретације јасно се могу уочити предности и мане сваког од њих: [1]

Ниво интерпретације	*Предности*	*Мане*
source-level	иницијална цена превођења не постоји, кашњење извршавања је нула	висока комплексност, лоше перформансе
intermediate code	мања комплексност, нешто боље перформансе	иницијално кашњење услед анализе и формирања међукôда
target code	не постоји софтверско интерпретирање – комплексност је нула, висока ефикасност извршавања	потенцијално велика цена превођења и самим тим и доста приметније кашњење

Табела 2.1: Преглед нивоа интерпретације

Промена изворног кôда у *source-level* интерпретацији не уводи никакве додатне режијске трошкове, док је код стандардног превођења потребно извршити рекомпилацију целог програма или модула. Код међукôда неретко је могуће извршити анализу само измењених делова тако да можемо рећи да је у том погледу *intermediate code* најбоље решење.

Треба напоменути да је недостатак у виду лошије ефикасности коју уводи имплементација интерпретера ретко разлог да би се она искључила у потпуности. *Overhead* у времену и простору који се уноси врло лако може бити апсолутно ирелевантан, посебно у већим рачунарским системима, а корист већа од негативног утицаја на перформансе. [1]

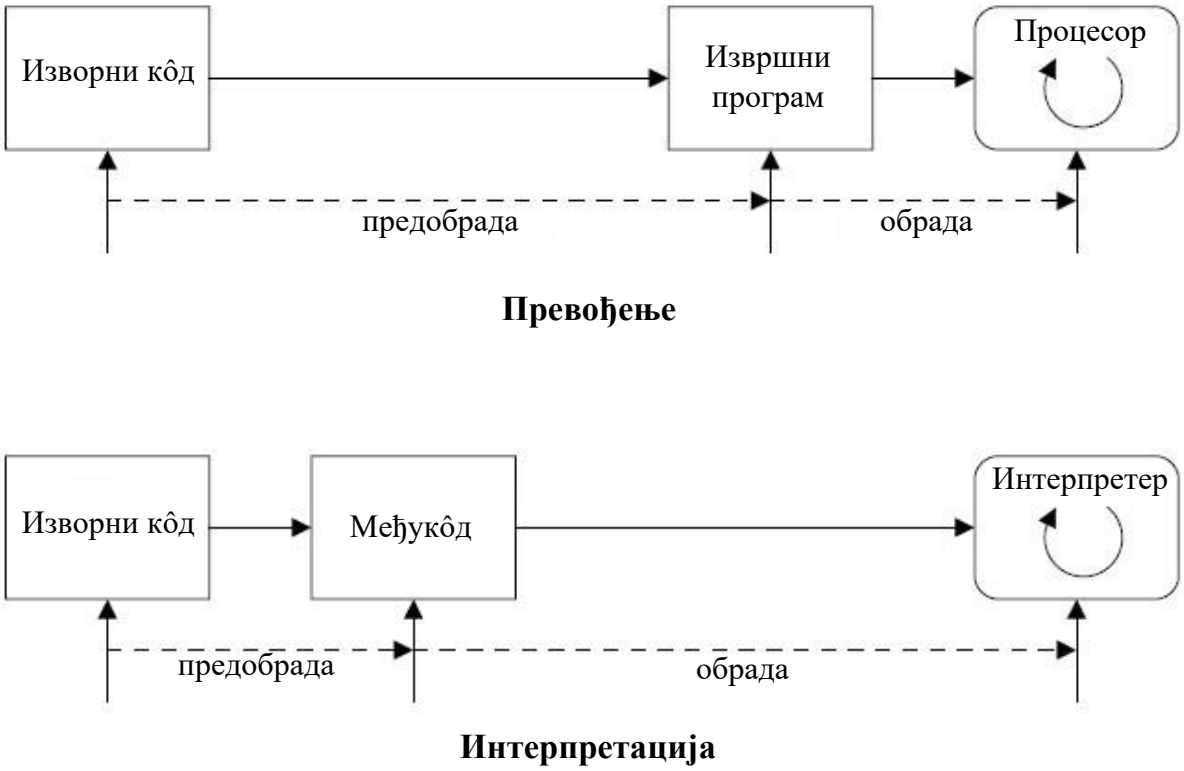
Најзад, не постоји фундаментална разлика између коришћења компајлера и коришћења интерпретера. У оба случаја текст програма се преводи у неки облик међукôда који се потом интерпретира по неком механизму. [5]

Код превођења:

- анализа изворног кôда има значајан удео у обради
- резултат је бинарни кôд ниског нивоа специфичан за циљну архитектуру
- механизам интерпретирања представља процесор
- извршавање програма је релативно брзо

Код интерпретације:

- анализа изворног кôда је минимална до умерена
- међукôд је нека структура података специфична за систем, високог или средњег нивоа
- механизам интерпретирања је програм
- извршавање програма је релативно споро

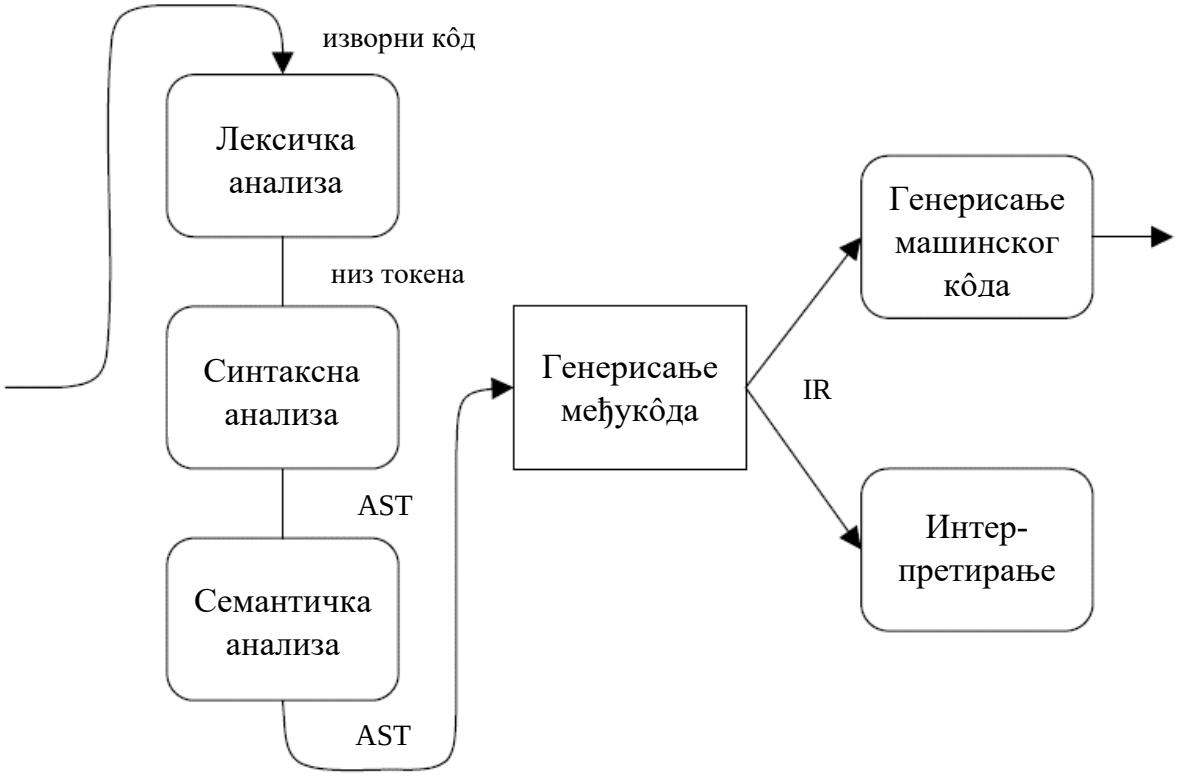


Слика 2.3: Поређење компајлера и интерпретера

3. ИМПЛЕМЕНТАЦИЈА

На (Слика 3.1) приказане су фазе кроз које пролази преводилац приликом обраде изворног програмског кôда. Још једном је приказано у ком се тачно тренутку раздвајају процеси интерпретирања и компајлирања, а у међукораку између неких фаза могуће су, а у пракси и обавезне, и одређене оптимизације кôда како би се добио што ефикаснији програм. Под тим се подразумевају елиминација непотребног (мртвог) кôда, разне оптимизације над петљама, пропагација константи итд. Овај рад се неће бавити техникама оптимизације, али се само напомиње да у савременом развоју компајлера оне имају једну од водећих улога.

У наставку поглавља ће бити описана имплементација сваке од ових фаза на конкретном интерпретеру за МикроЈава програмски језик са детаљнијим осврtом на фазе генерисања међукôда и интерпретирања.



Слика 3.1: Фазе програмског превођења

3.1. Лексичка и синтаксна анализа

За лексичку анализу МикроЈава програмског кôда коришћена је библиотека *JFlex* која на основу `lexer.flex` спецификације генерише скенер – лексички анализатор. Добијени скенер представља један детерминистички коначни аутомат који обрађује карактере улазног програмског текста (притом игноришући коментаре и бланко знакове), а као резултат генерише низ токена који се даље прослеђују синтаксном анализатору. У случају неке лексичке грешке у кôду (анализатор не проналази ниједан одговарајући тип токена) скенер ће пријавити непрепознатљив токен, али неће обуставити превођење.

Генерисање синтаксног анализатора, односно LALR парсера, постиже се помоћу *CUP* (Construction of Useful Parsers) библиотеке на основу `parser.cup` граматике МикроЈава језика. Такав парсер од низа токена добијених лексичком анализом формира апстрактно синтаксно стабло. Чворови тог стабла поред основних информација садрже и број линије у кôду на којој се елемент представљен тим чвором налази чиме се омогућава детаљнија пријава грешке кориснику. Све класе чворова овог стабла налазе се у подпакету `interpreter.ast` и њих аутоматски генерише *CUP* алат.

Начин пријављивања грешака је сличан као приликом лексичке анализе. Разлика је у томе што парсер грешке не игнорише већ покушава да се опорави од њих коришћењем алтернативних смена у граматици (смене са `error` симболом). У случају успешног опоравка парсер може да настави даљу анализу кôда и обавести корисника и о евентуалним каснијим грешкама не заустављајући се увек на првој.

Уколико преводилац не пронађе ниједну синтаксну грешку (било да се опоравио од ње или не) прелази у фазу семантичке анализе која сада програм види у облику апстрактног синтаксног стабла и која његовим обиласком и уз употребу додатних структура проверава семантичку исправност програма.

3.2. Семантичка анализа

Претходна фаза синтаксне анализе је уједно и последња фаза у којој се користи неки аутоматизовани начин формирања саставних блокова једног преводиоца, иако технички такви генератори постоје и за наредне фазе компајлирања које ће се разматрати. Као што је већ речено, резултат синтаксне анализе је апстрактно синтаксно стабло које, барем за сада, нема најповољнију структуру за интерпретирање, али за спровођење семантичких провера више него адекватну.

Анализатор обилази стабло од дна ка врху при томе прикупљајући све информације од интереса како би установио семантичку исправност програма. За чување тих информација користи наменске структуре у виду објеката класа из пакета `interpreter.symbols` које се у великој мери базирају на датој имплементацији табеле симбола на курсу Програмски преводиоци 1. Неке од измена су: интуитивнији називи класа, додатне методе у класи табеле симбола коришћене у генерисању међукôда, другачији формат исписа садржаја табеле итд.

3.2.1. Табела симбола

Сваки симбол у МикроЈава програму интерно је представљен објектом класе `Symbol` која садржи следећа поља:

- тип – константа, променљива, метода и друго
- име симбола
- вредност уколико се ради о константи
- цео број који представља редни број параметра код променљиве или укупан број параметара код методе
- референца на објекат класе `Type` о којој ће бити речи у наставку
- хеш табела локалних симбола методе или свих симбола програма

На основу последње ставке може се закључити да је и целокупан програм један симбол са којим се по завршетку семантичке анализе уланчавају сви остали симболи дефинисани у изворном програму, а чије је име дато на почетку самог кôда.

Типови, било прости (уграђени) или изведени (дефинисани од стране самог програмера), енкапсулирани су класом `Type`. Осим ознаке врсте типа (низ, класа, апстрактна класа или неки прости тип), код сложених типова се чува и референца на објекат типа родитеља – код низа то је тип његовог појединачног елемента (спецификација МикроЈава програмског језика не дозвољава низове који садрже

елементе различитих типова), а код класа то је објекат типа натклассе. Класни типови додатно садрже колекцију свих својих поља и метода.

За праћење опсега важења дефинисаних симбола користи се класа `Scope` која у виду уланчане листе памти у ком се опсегу тренутно налази семантички анализатор. Овај механизам прати који симболи су видљиви унутар тела методе, који на нивоу класе, а који симболи су глобални и самим тим има кључну улогу у фази семантичке анализе.

Најзад, ту је и сама табела симбола `SymbolTable`. Она обухвата све до сада поменуте структуре формирајући јединствену колекцију свих симбола дефинисаних у програму. Поседује методе за улазак и излазак из опсега, додавање новог или тражење или брисање већ унетог симбола, као и везивање тренутног опсега за објекат класног типа (крај дефиниције класе) или објекат симбола метода или целокупног програма (крај блока методе или дефиниције програма). Пример садржаја табеле симбола једног једноставног тест програма дат је на (Слика 3.2).

```
program Test
class C
  {
    int i;
    char c;
    bool b;
    {
      int m()
        int x;
        char y;
        bool z;
      {
      }
    }
  }

  C c;
{
  void main()
    int x;
    char y;
    bool z;
  {
  }
}
```

```
Type int: int
Type char: char
Type bool: bool
Constant null: class, null
Constant eol: char, \r\n
Method chr: char
     Variable integer: int
Method ord: int
     Variable character: char
Method len: int
     Variable array: array of no_type
Program Test: no_type
     Type C: class [
          Field i: int
          Field c: char
          Field b: bool
          Method m: int
               Variable this: class C
               Variable x: int
               Variable y: char
               Variable z: bool
     ]
     Variable c: class C
     Method main: no_type
          Variable x: int
          Variable y: char
          Variable z: bool
```

Слика 3.2: Пример садржаја табеле симбола

3.2.2. Семантичке провере

Спецификацијом језика МикроЈава задати су контекстни услови које мора да испуњава сваки програм. Самом имплементацијом табеле симбола обезбеђена су прва два општа контекстна услова. Како се при реферисању симбола увек најпре проверава да ли се он налази у табели почев од тренутног опсега важења ка окружујућим, обезбеђено је да свако име у програму мора бити декларисано пре првог коришћења. Такође, при додавању новог симбола увек се проверава да ли такав већ постоји у тренутном опсегу тако да имплицитно важи и да име не сме бити декларисано више пута унутар истог опсега. Трећи услов се односи на постојање улазне тачке програма, односно глобалне методе `main` без аргумената. Он се једноставно задовољава простом провером да ли се такав симбол налази у табели симбола на крају обиласка синтаксног стабла.

Остале провере се углавном односе на еквивалентност, односно компатибилност типова. Спецификација језика подразумева следеће:

1. Два типа података су **еквивалентна** ако имају исто име или ако су оба низови, а типови њихових елемената су еквивалентни
2. Два типа података су **компатибилна** ако су еквивалентни или ако је један од њих тип референце, а други је типа *null*
3. Тип *src* је **компатибилан при додели** са типом *dst* ако су *src* и *dst* еквивалентни, ако је *dst* тип референце, а *src* типа *null* или ако је *dst* класног типа, а *src* тип класе изведене из њега [6]

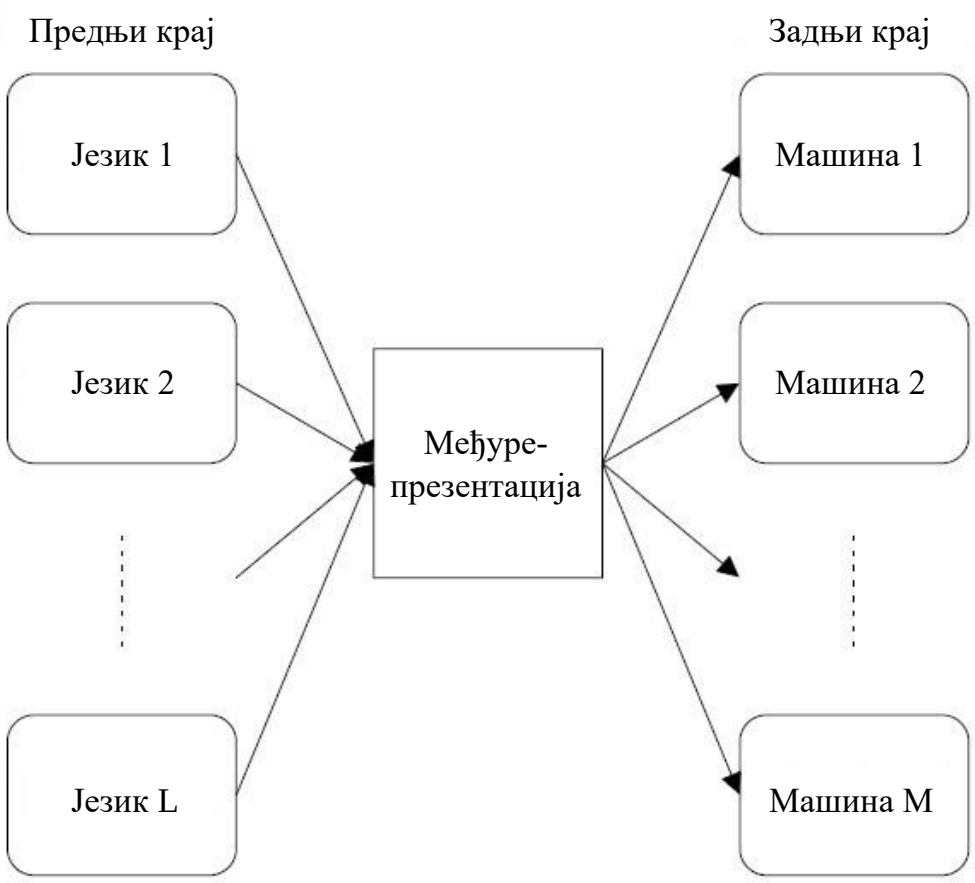
С обзиром да се стабло обилази од дна ка врху, поставља се питање како запамтити типове чворова који претходе оном који се тренутно обилази. Најочигледније, али и најлошије решење је чувати те информације у некој структури као пољу класе `SemanticAnalyzer`. Имајући у виду да програм може да буде произвољне величине то би подразумевало врло нерационално коришћење меморије. Међутим, *CUP* библиотека омогућава да се сваком чвору стабла дода једно поље произвољног типа за чување додатних информација приликом обиласка. Управо у ту сврху су коришћене класе `Symbol` или `Type` у зависности од конкретне потребе. На тај начин се провера типова своди на поређење поља `type` (односно `symbol.type`) два чвора.

Поред провере компатибилности типова постоје и одређени контекстни услови на нивоу наредби. *Break* и *continue* наредбе су дозвољене само унутар петље (решава се праћењем нивоа угнежденост петљи), *return* наредба само унутар тела методе и слично.

3.3. Генерисање међукôда

Уколико улазни програм испуњава све наведене контекстне услове добијамо резултат семантичке анализе у виду апстрактног синтаксног стабла, овога пута анотираног додатним корисним информацијама, и табелу симбола са свим декларисаним именима у програму. Ово стабло, иако валидан приказ програма, није најпогодније за интерпретацију зато што садржи велики број сувишних међучворова који су последица дате спецификације граматике језика (Слика 3.4). Из тог разлога се уводи ова додатна фаза како би се добила једноставнија репрезентација без губитка информација или утицаја на исправност програма.

Штавише, фаза генерисања међукôда је саставни део данашњих преводиоца који прате већ поменуту структуру са предњим и задњим крајем и, осим једноставности, главни добитак је незанемарљиво смањење програмерског посла потребног за превођење L програмских језика на M различитих машина (Слика 3.3).



Слика 3.3: Стварање преводиоца за L језика и M машина

У овом конкретном примеру користи се међурепрезентација високог нивоа у виду измењеног апстрактног синтаксног стабла, али могуће су, и врло честе у пракси, и репрезентације ниског нивоа, као нпр. троадресни кôд.

```
int getI()
{
  return this.i;
}
```

```
MethodDecl(
  SpecialReturnType(
    TypeName(
     int
    ) [TypeName]
  ) [SpecialReturnType]
  MethodName(
   getI
  ) [MethodName]
  NoFormalParameters(
  ) [NoFormalParameters]
  NoLocalVarDeclarations(
  ) [NoLocalVarDeclarations]
  MethodEntry(
  ) [MethodEntry]
  Statements(
    NoStatements(
    ) [NoStatements]
    ReturnResultStatement(
      TermExpr(
        FactorTerm(
          DesignatorFactor(
            DesignatorChaining(
              DesignatorName(
               this
              ) [DesignatorName]
             i
            ) [DesignatorChaining]
          ) [DesignatorFactor]
        ) [FactorTerm]
      ) [TermExpr]
    ) [ReturnResultStatement]
  ) [Statements]
) [MethodDecl]
```

Слика 3.4: Приказ дела синтаксног стабла декларације једне методе

Такве репрезентације су погоднije за проблеме зависне од саме машине као што су алокација регистара и избор инструкција. Занимљиво је и да се неретко као међукôд користи програмски језик *C* јер је флексибилан, преводи се у ефикасан машински кôд и његови компајлери су одавно широко распрострањени (првобитни *C++* компајлер је користио *C* преводилац као задњи крај). [2]

3.3.1. Структура стабла међурепрезентације

Генерално гледано, граматика сваког вишег програмског језика се може поделити у две кључне групе: **изрази** и **наредбе**. Изрази резултирају неком вредношћу (променљиве, литерали, позиви метода...), док наредбе, иако не враћају никакву вредност, производе тзв. **бочне ефекте** (декларација променљиве, класе или методе, дефиниција константе, контролне структуре...). Водећи се овом поделом и угледом на имплементацију Roberta Nystroma [3], структура измењеног стабла је осмишљена тако да су сви чворови конкретизација једне од корених апстрактних натклаcа – `Expression` и `Statement`.

Како смо за формирање првобитног синтаксног стабла користили библиотеку која аутоматизује цео процес, били смо поштеђени не само осмишљавања саме структуре, већ и репетитивног кодирања класа свих типова чворова које се готово и не разликују. Овога пута немамо ту олакшицу и то може да представља проблем. Поготово када је број тих класа огроман. У ту сврху смо се послужили згодним триком, под називом **метапрограмирање**. [3] То је програмерска техника код које рачунарски програми имају могућност да третирају друге програме као податке које обрађују. Односно, програм је дизајниран тако да чита, генерише, анализира или трансформише друге програме, па чак и да модификује самог себе док се извршава.

```
defineIRClass(outputDirectory, "Expression", Arrays.asList(
  "Binary   : Expression left,
              Binary.Operation operation,
              Expression right",
  "Call     : Expression callee, List<Expression> arguments",
  "Group    : Expression expression",
  "Index    : Expression array, Expression index",
  "Literal  : Object value",
  "Logical  : Expression left,
              Logical.Operation operation,
              Expression right",
  "New      : String type, Expression size",
  "Property : Expression object, String name",
  "Unary    : Unary.Operation operation, Expression right",
  "Variable : String name"
));
```

Слика 3.5: Позив методе за генерисање класе `Expression`

У овом случају, циљ нам је да изгенеришемо *Java* класе новог стабла уз само неколико линија кôда и згодно форматирање. То је одрађено класом `GenerateIR` пакета `utility` која као улазни аргумент програма прима жељени одредишни директоријум у који ће сместити све изгенерисане `.java` датотеке. Позивом помоћне приватне методе `defineIRClass` (Слика 3.5 и Слика 3.6) генерише се одговарајућа апстрактна класа унутар које се налази интерфејс `Visitor` и све конкретне потклассе са свим својим пољима као што је наведено листом стрингова.

```
defineIRClass(outputDirectory, "Statement", Arrays.asList(
  "Assignment  : Expression destination, Expression value",
  "Block       : List<Statement> statements",
  "Call        : Expression.Call expression",
  "Class       : String name,
                 String superClass,
                 List<Class.Field> fields,
                 List<Statement.Method> methods",
  "Constant    : String name, Object value",
  "Control     : Control.Type type",
  "Declaration : Declaration.Type type, String name",
  "Decrement   : Expression number",
  "For         : Statement initializer,
                 Expression condition,
                 Statement increment,
                 Statement body",
  "If          : Expression condition,
                 Statement thenBranch,
                 Statement elseBranch",
  "Increment   : Expression number",
  "Method      : boolean isVoid,
                 String name,
                 List<String> parameters,
                 List<Statement> body",
  "Print       : Expression expression, Integer width",
  "Program     : List<Statement> statements",
  "Read        : Declaration.Type type,
                 Expression destination",
  "Return      : Expression value"
));
```

Слика 3.6: Позив методе за генерисање класе `Statement`

На овим исечцима се уједно могу видети и сви типови чворова (са својим пољима) које ће ново стабло међукôда поседовати. Очигледно је да се њихов број драстично смањио у односу на иницијално синтаксно стабло што ће, као што је већ напоменуто, омогућити доста лакше праволинијско интерпретирање.

3.3.2. Пројекција еквивалентних чворова

Генерисање међукôда се, слично фази семантичке анализе, врши обиласком апстрактног синтаксног стабла од дна ка врху. За праћење контекста користе се два стека: стек израза `expressionStack` и стек наредби `statementStack`. Они су потребни због природе обиласка стабла који је *postorder* – чвор родитељ се обилази тек након обиласка све његове деце.

У (Табела 3.1) су приказане промене стања стека израза на ограниченом примеру са бинарним логичким оператором. Сличан принцип се примењује и у сложенијим ситуацијама. Окружујуће витичасте заграде означавају да се унутрашњи елемент налази унутар објекта из претходног корака и да се тако прослеђује конструктору сложенијег израза, док стрелица са десне стране указује на врха стека.

	expressionStack
var & arr[5]	Expression.Variable(**var**) ←
var & **arr**[5]	Expression.Variable(**arr**) ← Expression.Variable(**var**)
var & arr[**5**]	Expression.Literal(**5**) ← Expression.Variable(**arr**) Expression.Variable(**var**)
var & **arr[5]**	Expression.Index(**{arr}, {5}**) ← Expression.Variable(**var**)
var & arr[5]	Expression.Logical(**{var}, AND, {{arr}, {5}}**)

Табела 3.1: Упрошћени приказ стања стека израза

Код формирања чворова наредби је нешто сложенији поступак. Као што је речено, све декларације променљивих и дефиниције класа и метода се класификују као наредбе. Оне се морају груписати на одговарајући начин и повезати са окружујућим ентитетом – поља и методе са класом, а глобалне променљиве и дефиниције класа и глобалних метода са програмом. Из тог разлога је стек наредби у ствари стек листе наредби. На пример, при наиласку на дефиницију методе додаје се нова празна листа на стек. Све наредбе које се налазе у телу функције, укључујући и декларације локалних променљивих, се додају у ту листу. Када се дође до краја дефиниције листа се скида са врха стека и додаје објекту `Statement.Method` који се опет додаје као и претходне наредбе у листу на врху стека наредби. Поступак се

наравно понавља док се не обиђу сви чворови у стаблу када се коначно листа на дну стека везује за симбол самог програма.

С обзиром да су по завршетку фазе семантичке анализе уланчани сви опсези за одговарајуће симболе потребно је пронаћи другачији начин за симулирање промене опсега важења. Из тог разлога је за навигацију кроз већ попуњену табелу симбола додат још један помоћни стек `scopeStack`. На (Слика 3.7) је приказана метода `enterScope` за улазак у опсег важења на основу имена програма или методе ако је у питању симбол, односно имена класе уколико се ради о типу. Разлика у односу на `openScope` методу коришћену у семантичкој анализи је што се уопште не алоцира простор већ се само користе претходно попуњене колекције симбола.

```
public void enterScope(String name)
{
  Symbol symbol = scopeStack.peek().get(name);

  if (symbol.getKind() == Symbol.TYPE)
  {
    scopeStack.push(symbol.getType().getMembers());
  }
  else
  {
    scopeStack.push(symbol.getLocals());
  }
}
```

Слика 3.7: Промена опсега важења када је табела симбола попуњена

Такође, промена опсега је увек испраћена модификацијом стека наредби што се може приметити на примеру дефиниције класе (Слика 3.8).

```
symbolTable.enterScope(classSymbol.getName());
statementStack.push(new LinkedList<>());
List<Statement> statements = statementStack.pop();
List<Statement.Method> methods = new LinkedList<>();

for (Statement statement : statements)
{
  methods.add((Statement.Method) statement);
}

symbolTable.exitScope();

statementStack.peek().add(new Statement.Class(line, name,
superClass, fields, methods));
```

Слика 3.8: Комплементарне операције

Овим другим обиласком стабла коначно је и изгенерисан међукôд. У меморији он се чува као објекат класе `Statement.Program` који садржи листу свих наредби које интерпретер треба да изврши. Осим тога, међукôд ће бити сачуван у датотеци за накнадно интерпретирање, а можда и за евентуалну оптимизацију и генерисање машинског кôда уколико се тако нешто омогући каснијим развојем преводиоца. Позивом главног програма корисник може да изабере да ли жели одмах да интерпретира своје решење или не, као и да проследи име датотеке у коју жели да смести међукôд (уколико то не учини преводилац ће искористити назив изворног кôда уз промену екстензије у `.ir`).

3.4. Интерпретација

Интерпретирање међукôда је реализовано класом `Interpreter` која имплементира интерфејсе:

1. `Expression.Visitor<Object>`
   тип `Object` због повратне вредности којом резултира израз након његовог израчунавања (не зна се унапред тачно којег је типа та вредност)
2. `Statement.Visitor<Void>`
   тип `Void` као омотач за случајеве када се не очекује повратна вредност, али је потребан конкретан тип – извршавање наредби нема повратну вредност

За обилазак стабла међурепрезентације користи се метода `evaluate` за изразе, односно метода `execute` за наредбе. С обзиром да је програм потпуно семантички исправан (ово не значи да не може доћи до неке *runtime* грешке), процес интерпретирања је прилично праволинијски и интуитиван.

3.4.1. Структуре коришћене у време извршавања

Кључну улогу у фази интерпретирања имају класе из пакета `interpreter.runtime`. Оне омогућавају инстанцирање референтних типова, позивање метода, манипулацију контроле тока, али и одржавају интерно стање интерпретера. Управо за чување стања је намењена класа `Environment`. Она је суштински пандан класи `Scope`, али за разлику од ње поседује само три методе – дефинисање, дохватање и доделу вредности имену из окружења. Промена опсега значи и промену окружења, односно, механизам функционисања је исти.

Класни типови су представљени класом `RuntimeClass` која памти своје име, натклasu уколико постоји, поља и методе. Њихове инстанце `RuntimeInstance` чувају своје копије поља и референцу на класу којој припадају како би могле да

приступе методама – поља су локална за инстанцу (МикроЈава не подржава статичка поља) док се методе чувају на нивоу класе (као да су статичке). Питање које се може поставити је како се онда приступа пољима унутар методе ако она није везана за објекат. Одговор на њега је скривени параметар *this* који показује на објекат над којим је позвана метода. На (Слика 3.9) се може видети начин дохватања чланова класе. У случају да се ради о методи потребно је прво везати (*bind*) инстанцу са параметром *this* (подебљани део кôда).

```
public Object get(String member)
{
  if (fields.containsKey(member)) return fields.get(member);

  RuntimeMethod method = runtimeClass.getMethod(member);
  if (method != null) method.bind(this);

  return method;
}
```

Слика 3.9: Метода `get` класе `RuntimeInstance`

На овај начин би се омогућио приступ пољима само експлицитно преко параметра *this*. Због тога се у оквиру самог окружења врши провера уколико неко име није нађено преко свог имена, а постоји параметар *this* дефинисан у опсегу – суштински, као да је у кôду вештачки додато `this.` (Слика 3.10).

```
public Object get(String name)
{
  ...

  if (values.containsKey(Interpreter.THIS))
  {
    RuntimeInstance instance = (RuntimeInstance)
                               values.get(Interpreter.THIS);
    Object object = instance.get(name);
    if (object != null) return object;
  }

  ...
}
```

Слика 3.10: Метода `get` класе `Environment`

Алоцирање низова је омогућено класом `RuntimeArray` која је у ствари омотачка класа за низ елемената типа `Object`. Поседује методе за приступ и промену индивидуалног елемента, као и за дохватање дужине низа.

Механизам позива функција и промене контекста извршавања остварен је интерфејсом `RuntimeCallable` и класом која га имплементира `RuntimeMethod`. Приликом позива методе формира се нови опсег важења који се уланчава са тренутним окружењем интерпретера. Потом се сваки прослеђени аргумент мапира на одговарајући параметар методе да би се на крају извршило само тело функције (Слика 3.11). Након тога окружење методе се одбацује, а интерпретер наставља извршавање од места позива.

```
@Override
public Object call(Interpreter interpreter, List<Object> arguments)
{
  Environment environment =
          new Environment(interpreter.getEnvironment());

  if (thisInstance != null)
  {
    environment.define(Interpreter.THIS, thisInstance);
  }

  for (int i = 0; i < arguments.size(); i++)
  {
    environment.define(method.parameters.get(i), arguments.get(i));
  }

  try
  {
    interpreter.execute(method.body, environment);
  }
  catch (Return aReturn)
  {
    return aReturn.value;
  }

  if (!method.isVoid)
  {
    throw new InterpretingException(method.line,
    "Missing return statement in method '" + method.name + "'");
  }

  return null;
}
```

Слика 3.11: Интерпретација позива методе

У кôду је још једном назначен тренутак када се објекат везује са методом класе.

Такође, први пут наилазимо и на неку грешку која се проверава у време извршавања. Уколико је метода декларисана тако да враћа неку вредност, а то се у неком сценарију извршавања не догоди, интерпретер ће зауставити извршавање и пријавити грешку. Друга и последња могућа *runtime* грешка је *NullPointerException* до које може доћи ако се приступа неалоцираном низу или класној инстанци.

Треба напоменути да је са оваквом спецификацијом језика МикроЈава непотребна употреба интерфејса `RuntimeCallable`, али је он уведен да би се омогућила лакша интеграција неких додатних функционалности као што су конструктори.

Због природе интерпретације обиласком стабла коју карактерише нелинеарно извршавање инструкција јавља се проблем у неким ситуацијама приликом скокова када је врло тешко одредити дестинацију скока. Типични примери тога су ранији повратак из функције или искакање из петље. У [3] предложено је ефикасно решење употребом механизма обраде изузетака у *Java* програмском језику. Наиме, ако унапред знамо где можемо да ухватимо неки изузетак онда тај *try-catch* блок можемо користити као лабелу у асемблерском кôду. На (Слика 3.11) може се видети дато решење на примеру повратка из функције. Уколико се у телу методе наиђе на `return` наредбу, генерише се `Return` изузетак који извршавање интерпретера помера тачно на прву наредну инструкцију након позива методе. Слични принцип се користи и за `break` и `continue` наредбе (Слика 3.12).

```
if (statement.initializer != null)
{
  execute(statement.initializer);
}

while (isTrue(evaluate(statement.condition)))
{
  try
  {
    execute(statement.body);
  }
  catch (Break aBreak) { break; }
  catch (Continue aContinue)  { }

  if (statement.increment != null)
  {
    execute(statement.increment);
  }
}
```

Слика 3.12: Интерпретација `for` петље

3.4.2. Извршавање наредби и евалуација израза

Предност интерпретирања је пре свега то што се операције које желите да извршите само своде на већ одрађену имплементацију у вишем програмском језику у којем је писан интерпретер. Најочигледнији пример је условна контрола тока (Слика 3.13). На неком нижем нивоу апстракције било би потребно уградити инструкције условних и безусловних скокова, поставити одговарајуће лабеле итд. Међутим, овде се само МикроЈава кôд транслира у еквивалентни *Java* кôд.

```
@Override
public Void visit(Statement.If statement)
{
  if (isTrue(evaluate(statement.condition)))
  {
    execute(statement.thenBranch);
  }
  else if (statement.elseBranch != null)
  {
    execute(statement.elseBranch);
  }
  return null;
}
```

Слика 3.13: Интерпретација `if` наредбе

Исти случај је и са аритметичким, логичким и релационим операторима – своде се на одговарајуће *Java* операторе. Чињеница да су интерпретеру познати улазни подаци може се искористити да се делови апстрактног синтаксног стабла који су небитни за само извршавање одстране из разматрања. Осим у *if-else* структури, такав случај је и код логичких оператора (Слика 3.14). Уколико је израз са леве стране `&&` оператора (логичко И) нетачан, израз са десне стране се не мора ни израчунавати (слично и за логичко ИЛИ) што може значајно редуковати путању обиласка стабла.

```
@Override
public Object visit(Expression.Logical expression)
{
  Object left = evaluate(expression.left);

  Object result;

  switch (expression.operation)
  {
  case AND:
    if (!isTrue(left)) result = left;
    else result = evaluate(expression.right);
    break;
  case OR:
    if (isTrue(left)) result = left;
    else result = evaluate(expression.right);
    break;
  default:
    throw new InterpretingException(expression.line,
    "Unrecognized logical operation.");
  }

  return result;
}
```

Слика 3.14: Интерпретација логичких операција

3.4.3. Интерпретер као независна апликација

Интерпретер је могуће покренути и као самосталну апликацију прослеђивањем путање до датотеке са већ изгнерисаним међукôдом (мора имати `.ir` екстензију). На тај начин је потпуно испоштована структура преводиоца са почетка рада (Слика 2.2). Предњи крај чине лексички, синтаксни и семантички анализатори са генератором међукôда, док је задњи крај сâм интерпретер. Тако је теоријски омогућена интерпретација програма написаног на било ком језику уколико се он трансформише у исти међукôд. Такође, задњи крај се може заменити оптимизатором и генератором машинског кôда уколико је циљ добити оптималније извршавање (тада ће преводилац бити истоветан оном са Слика 2.2).

4. ЗАКЉУЧАК

На самом крају, треба упоредити перформансе интерпретирања и превођења на једном конкретном примеру. Рачунање Фибоначијевог броја простом рекурзивном формулом је згодан пример зато што захтева велики број позива функције и може да представља тежак задатак чак и при најефикаснијем извршавању. Да би се најбоље приказао контраст у перформансама за пример компајлера изабран је преводилац за програмски језик *C* (конкретно GCC v7.5.0) који је познат по генерисању оптималног кôда. Програми чије се извршавање упоређује дати су на (Слика 4.1).

fibonacci.mj

```
program Fibonacci
{
  int fibonacci(int n)
  {
    if (n < 2) return n;
    return fibonacci(n - 1) +
           fibonacci(n - 2);
  }

  void main()
  {
    print(fibonacci(40));
    print(eol);
  }
}
```

fibonacci.c

```
#include <stdio.h>

typedef unsigned long long ullong;

ullong fibonacci(ullong n)
{
  if (n < 2) return n;
  return fibonacci(n - 1) +
         fibonacci(n - 2);
}

int main(int argc,
         const char *argv[])
{
     printf("\n%lld\n",
            fibonacci(40));

     return 0;
}
```

Слика 4.1: Рачунање Фибоначијевог броја

Времена која се упоређују су искључиво времена извршавања програма – не укључују генерисање међукôда нити његово читање из датотеке:

- C преводилац – *0.041s*
- МикроЈава интерпретер - *70.747s*

Иако је исход донекле могао да се наслути, тешко је било претпоставити да је извршавање преведеног кôда чак и до неколико редова величине ефикасније од интерпретације. Показује се да су режијски трошкови које уноси обилазак стабла приликом честих позива метода превелики, а због константног приступа оперативној меморији. Са друге стране, у машинском кôду се цео механизам своди на инструкције скока уз проблем са великим бројем активационих записа на стеку. Међутим, тај проблем је код интерпретера још израженији јер постоји још један додатни ниво индирекције у виду *Java* виртуелне машине.

Треба напоменути да изабрани пример са крајње неефикасним рачунањем Фибоначијевих бројева ипак представља екстремни случај који се у просечном, добро написаном програму највероватније неће десити. Генерално, интерпретер ће имати сасвим прихватљиво време извршавања иако не уводи никакве оптимизације у кôд и захтева много мање времена и труда за коректну имплементацију.

На основу свега приказаног читалац се могао уверити у предности и мане интерпретирања поменуте у другом поглављу (Табела 2.1). Интерпретирани програмски језици су широко распрострањени у модерном софтверском инжењерству без тенденције да се то промени. Најутицајнији примери су наравно *JavaScript*, језик интернет претраживача, и *Python*, језик чија популарност непрестано расте због све масовније употребе машинског учења у индустрији.

5. ЛИТЕРАТУРА

[1] D. Watson, A Practical Approach to Compiler Construction, Cham: Springer, 2017.

[2] A. V. Aho, M. S. Lam, R. Sethi и J. D. Ullman, Compilers, Principles, Techniques, & Tools, Boston: Addison-Wesley, 2007.

[3] R. Nystrom, „Crafting Interpreters,“ [На мрежи]. Available: https://craftinginterpreters.com. [Последњи приступ Април 2020].

[4] A. W. Appel и J. Palsberg, Modern Compiler Implementation in Java, Cambridge: Cambridge University Press, 2004.

[5] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs и K. Langendoen, Modern Compiler Design, New York: Springer, 2012.

[6] Д. Бојић, „Материјали са предавања и вежби курса Програмски преводиоци 1,“ Електротехнички факултет, Универзитет у Београду, [На мрежи]. Available: http://ir4pp1.etf.rs. [Последњи приступ Јул 2020].

[7] K. D. Cooper и L. Torczon, Engineering a Compiler, Burlington: Morgan Kaufmann Publishers, 2012.

[8] C. N. Fischer, R. K. Cytron и R. J. LeBlanc, Jr., Crafting a Compiler, Boston: Addison-Wesley, 2010.

[9] S. Marr и S. Ducasse, „Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches,“ у *Proceedings of ACM International Conference on Object Programming Systems Languages & Applications (OOPSLA '15)*, Pittsburgh, 2015.

[10] D. A. Watt и D. F. Brown, Programming Language Processors in Java: Compilers & Interpreters, Essex: Pearson Education Limited, 2000.

Сви графички прикази у раду су, уз превод текста, преузети из књиге *Modern Compiler Design* [5].