УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

Документација пројекта

Студент: Невена Прокић
Број индекса: SW6/2019

Предмет: Системска програмска подршка 1
Тема: MAVH преводилац

Нови Сад, јун, 2021.

Садржај

1. Увод

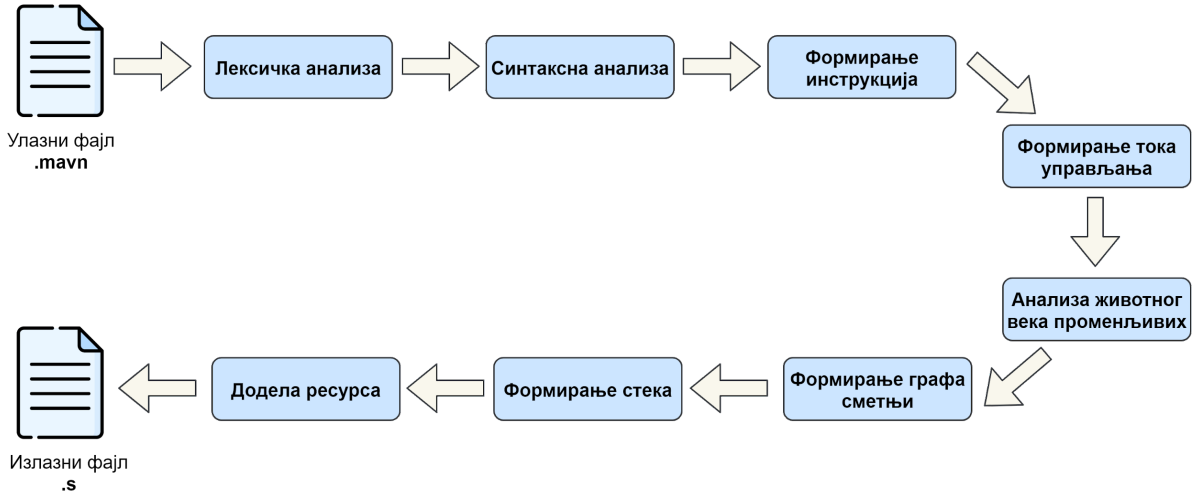
1.1. MABH преводилац

MABH преводилац представља алат који преводи програм написан на вишем MIPS 32bit асемблерском језику на основни асемблерски језик. Коришћење MIPS 32bit асемблерског језика је лакше због концепта регистарских

променљивих. Овај концепт нам омогућава да приликом писања инструкција не морамо да водимо рачуна о коришћеним регистрима и њиховом садржају.

Приликом саме имплементације потребно је имплементирати кораке који ће бити описани у наставку.

1.2. Задатак



Лексичка анализа је представљена преко лексичког анализатора који улазни фајл претвара у листу токена. Токени представљају речи програмског језика.

Синтаксна анализа проверава да ли је листа токена у складу са дефинисаном граматиком.

Формирање инструкција, у овом делу је потребно направити листу варијабла, лабела, функција и инструкција који се налазе у улазном програму на основу токена.

Формирање тока управљања је фаза у којој за сваку инструкцију дефинишемо која је њена претходна и наредна инструкција. Ова фаза представља увод у следећу фазу.

Анализа животног века променљивих представља фазу у којој анализирамо да ли постоји могућност да различите променљиве које нису истовремено у употреби користе исти ресурс.

Формирање графа сметњи је фаза у којој детекцију којим променљивама не могу да се доделе исти регистри због преклапања опсега животног века променљивих.

Формирање стека је фаза у којој се променљиве смештају на стек по одговарајућем алгоритму.

Фаза избора представља фазу у којој се чворови, односно променљиве, боје на основу формираног стека.

Фаза формирања графа сметњи, формирања стека, формирање избора, као и фаза преливања, која овде није обрађена, ће се касније детаљније обрадити и оне заједно представљају *фазу доделе ресурса*.

2. Анализа проблема

Проблеме које можемо уочити јесу саме имплементације делова из задатка самог пројекта. Потребно је имплементирати обавезних 10 MIPS инструкција које подржава MABH језик.

Те инструкције су следеће:

add - (addition) сабирање
addi - (addition immediate) сабирање са константом
b - (unconditional branch) безусловни скок
bltz - (branch on less than zero) скок ако је регистар мањи од нуле
la - (load address) учитавање адресе у регистар
li - (load immediate) учитавање константе у регистар
lw - (load word) учитавање једне меморијске речи
nop - (no operation) инструкција без операције
sub - (subtraction) одузимање
sw - (store word) упис једне меморијске речи

Инструкције које су додате:

and - (AND) логичко и
slt - (set less than) постави 1 ако је вредност првог регистра мања од вредности другог
lui - (load upper immediate) учитавање горњих 16 битова
ori - (OR immediate) логичко или константе и вредности регистра

Додате инструкције је потребно додати приликом лексичке и синтаксне анализе, такође и имплементирати и за њих све исто као и за обавезне инструкције.

Синтакса самог MABH језика се описује граматиком:

Q → S ; L	S → _mem mid num	L → eof	E → add rid, rid, rid
	S → _reg rid	L → Q	E → addi rid, rid, num
	S → _func id		E → sub rid, rid, rid

S → id: E
S → E

E → la rid, mid
E → lw rid, num(rid)
E → li rid, num
E → sw rid, num(rid)
E → b id
E → bltz rid, id
E → nop
E → and rid,rid,rid
E → ori rid,rid,num
E → lui rid,num
E → slt rid,rid,rid

МАВН језик поседује и терминалне симболе и у њих спадају следећи:

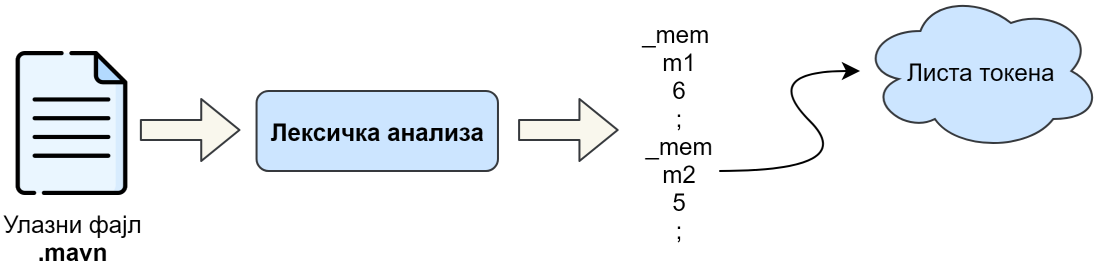
: ; . ( ) _mem, _reg, _func, num id, rid, mid, eof, add, addi, sub, la, lw, li, sw, b, bltz, nop, ori, slt, lui, and

Након провере семантике, имплементација инструкција и осталих алгоритама потребно је генерисати излазни фајл са екстензијом '.s' која преставља програм основног MIPS 32bit асемблерског језика. Сам излазни фајл треба да садржи три секције, а то су:

- секција *.globl* у којој се налазе имена функција које су у улазном програму дефинисане са *_func funcName*
- секција *.data* у којој се налазе меморијске променнљиве које су у улазном програму дефинисане са *_mem varName value*
- секција *.text* у којој се налазе све инструкције, регистри из улазног програма који су дефинисани са *_reg varName* треба да буду замењени са регистарским променљивама (t0, t1, t2, t3).

3. Концепт решења

3.1. Лексички анализатор



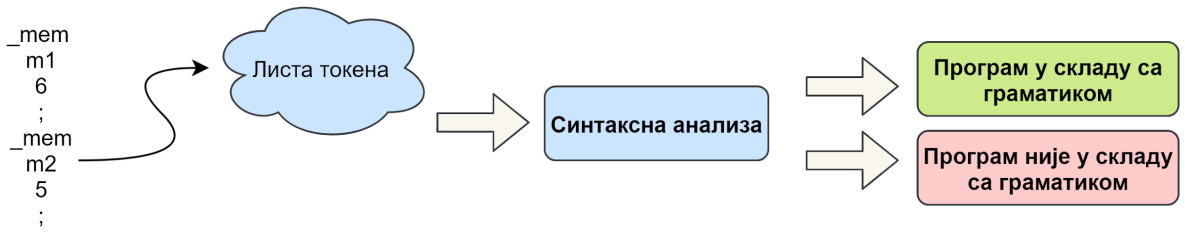
Задатак лексичке анализе јесте да улазни низ карактера претвори у токене. Његова реализација се врши помоћу коначног детерминистичког аутомата. Аутомат има за циљ да препозна све симболе у програму и у томе му помажу два правила:

1. Правило најдуже речи
2. Правило приоритета

Подржава цифре, слова, специјалне карактере(размак, нови ред), операторе (. -).

У нашем случају лексички анализатор је имплементиран са коришћењем правила најдуже речи. Такође овај део се налазио у почетном пројекту већ имплементиран.

3.2. Синтаксни анализатор

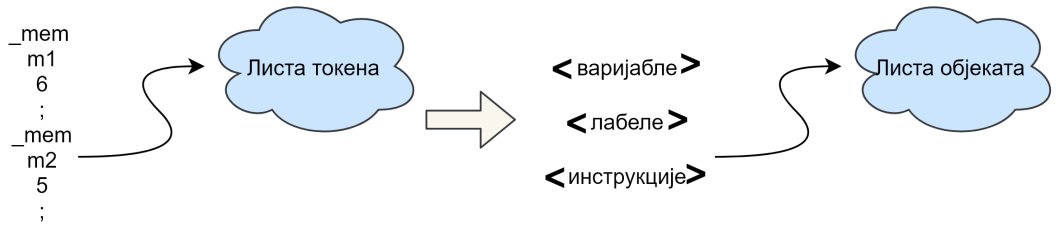


Синтаксни анализатор је имплементиран уз помоћ алгоритма са рекурзивним спуштањем. Позивом рекурзивних функција сваку продукцију претварамо у реченицу. За сваки нетерминални симбол се имплементирана једна функција и по један услов за сваку продукцију.

Текући токен нам представља следећи симбол из листе токена. Позивамо функцију Q на почетку, а затим у наставку од граматике ће бити позиване остале рекурзивне функције за нетерминалне симболе и функција eat за терминалне симболе. Кључну улогу има функција eat која проверава да ли се поклапа очекивани симбол са прослеђеним симболом, уколико ово није случај биће пријављена синтаксна грешка.

Функција *eat* као и синтаксни анализатор је одрађен на вежбама, док се сама граматика разликује.

3.3. Формирање инструкција



У овој фази било је потребно проћи кроз листу токена и направити следеће ствари:

1. Листа варијабли
2. Листа лабела и функција
3. Листа инструкција

Листу варијабли формирају регистарске и меморијске променљиве. Све садрже атрибуте као што су:

- m_type - тип варијабле
- m_name - име варијабле
- m_position - позиција варијабле (почиње од 0)
- m_assignment - регистар варијабле
- m_degree - степен у графу сметњи
- m_interference - листа са којим варијаблама је у сметњи
- m_value - вредност варијабле

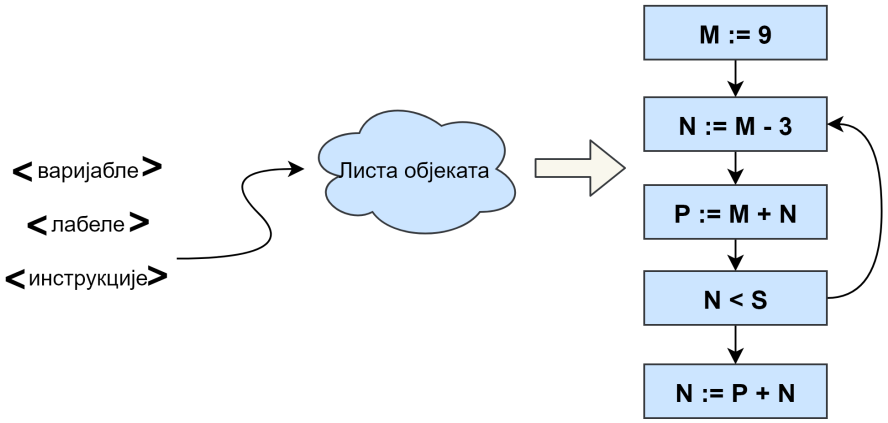
Листу лабела и функција формирају лабеле и функције које су дефинисане у улазном програму. Атрибути које оне садрже су:

- m_type - тип
- m_name - име
- m_position - позиција (позиција инструкције која се налази непосредно испод њих)

Листу инструкција формирају инструкције. Њих формирамо на основу њиховог типа, јер је потребно и попунити скупове које инструкције садрже. Атрибути које оне садрже су:

- m_type - тип инструкције
- m_position - позиција инструкције (почетна позиција је 0)
- m_value - вредност уколико инструкција садржи константу
- m_offset - померај уколико га инструкција садржи
- m_dst - одредишни скуп варијабли
- m_src - изворни скуп варијабли
- m_use - скуп промењљивих које се користе у инструкцији
- m_def - скуп промењљивих које се дефинишу у инструкцији
- m_in - овај скуп се формира по алгоритму у анализи животног века промењљивих
- m_out -овај скуп се формира по алгоритму у анализи животног века промењљивих
- m_succ - скуп инструкција које су наследници текуће инструкције
- m_pred - скуп инструкције које су претходници текуће инстркције
- m_label - лабела уколико је инструкција садржи

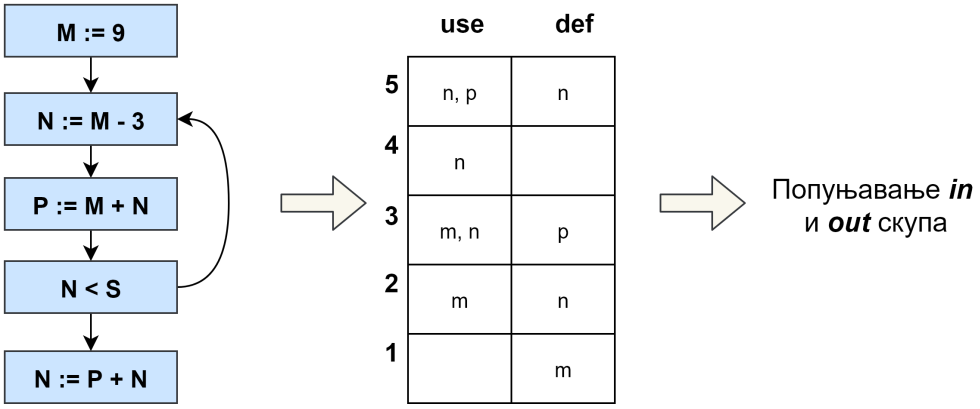
3.4. Формирање тока управљања



Приликом формирања тока управљања посматрамо за сваку инструкцију које су инструкције њени наследници и претходници. Затим свакој промењљивој придружујемо инструкцију наследник/претходник у одговарајућем скупу. Међутим сам ток неће бити серијски, јер постоје инструкције за скокове на лабеле. Такве инструкције су b и bltz. Инструкција b ће у инструкцију на коју скаче додати и себе као претходника, међутим инструкција која се налази у линији испод b неће имати инструкцију b као свог претходника. Док ће инструкција bltz само иснтрукцији на коју скаче додати још једног претходника, тј. себе. Овде када кажемо да инструкција b или bltz скаче на инструкцију

сматрамо да скаче на лабелу, али ће у инструкцији чију позицију има лабела да буде промењен скуп претходника.

3.5. Анализа животног века



У току ове фазе потребно је видети да ли постоји могућност да различите променљиве које нису у исто време у употреби могу делити исти ресурс. Свака променљива има свој животни век који почиње приликом њеног дефинисања, а завршава се последњом употребом.

Уводна фаза је претходно описана фаза, а затим следи алгоритам анализе животног века променљивих. Сам алгоритам можемо представити следећим једначинама:

$$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$$

$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

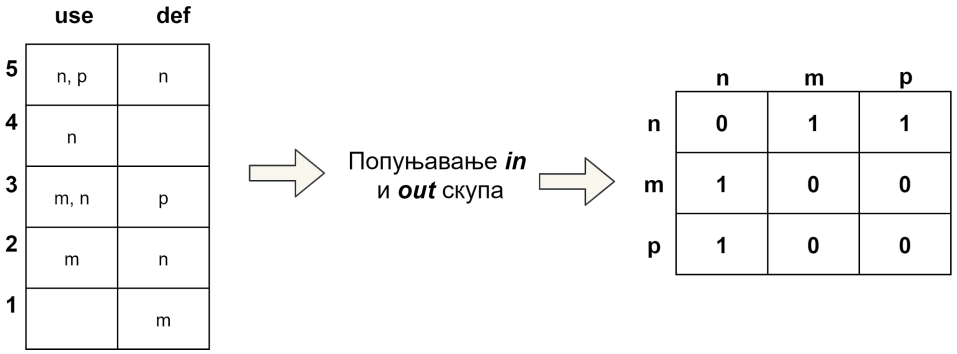
$in[n]$ - скуп променљива које су живе на било којој улазој стрелици чвора
$out[n]$- скуп променљива које су живе на било којој излазној стрелици чвора
s- променљиве из in скупа које припадају $succ$скупу инструкција текуће инструкције

За сваку инструкцију из листе инструкција потребно је попунити in и out скупове. Бољи начин попуњавања је уколико се крене од последње инструкције ка првој и у нашем случају пролазак кроз све инструкције треба поновити два пута због скокова. Проблем настаје уколико се једном прође in и outскупови

инструкције на коју се скаче нису попуњени и тиме сами in и outскупови инструкције скока неће бити комплетно попуњени.

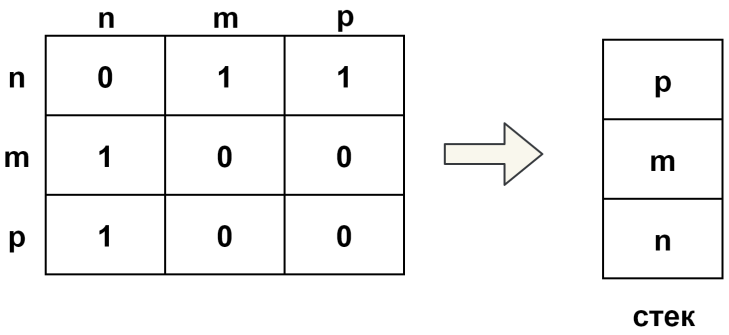
3.6. Додела ресурса

3.6.1. Формирање графа сметњи



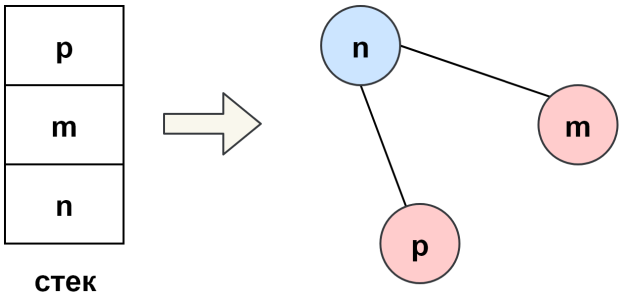
Уколико дође до преклапања животног века двеју променљивих то представља сметњу. Тим променљивама не можемо да доделимо исти регистар. Како бисмо уочили које су све променљиве у сметњи. треба формирати граф сметњи односно матрицу сметњи. Матрица ће бити квадратна матрица чије колоне и регодови представљају променљиве из програма. Уколико су две променљиве у сметњи на пресеку њихових колона и редова биће уписана јединица у супротном ће стајати нула.

3.6.2. Формирање стека



Приликом формирања стека прво је неопходно одредити за сваки чвор. односно промењиву њен степен који представља са колико је чворова у сметњи. Такође треба видети који су то чворови у сметњи са текућим чвором. Затим се попуњава стек са променљивама. Алгоритам за попуњавање стека каже да се ставља први чвор који има степен мањи од К где К представља број регистара. Затим се чвор уклања са графа као и све гране које је он имао. Овај поступак се понавља све док постоје чворови степена К. Након стављања свих чворова на стек са степеном К степен се смањује за један и поступак се понавља. Овај поступак се понавља све док се све промењиве не ставе на стек или док не дођемо у ситуацију преливања. Преливање је следећи проблем који треба решити у оквиру доделе ресурса. Он означава да се граф више не може упростити и самим тим да није могуће за све промењиве које су у сметњи доделити им различите боје. односно регистре. Самим тим дати граф не можемо обојити са К боја. За решавање овог проблема постоје разни алгоритми. али овде то неће бити одрађено. Појава преливања је детектована и пријављена је грешка уколико се она деси.

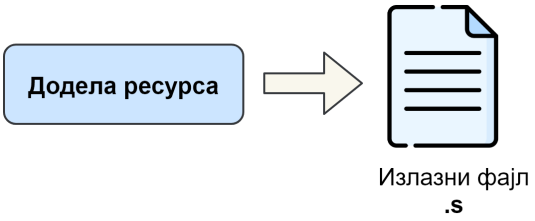
3.6.3. Фаза избора



Након успешног стављања промењивих на стек потребно је да се промењиве скину са стека и обоје. Бојење чворова иде у обрнутом редоследу од оног како су стављани на стек, што значи да ће први обојени чвор бити онај који

је последњи додат на стек. Бојење се врши тако да се чвор не сме обојити истом бојом којом је обојен неки од суседних чворова, односно неки од чворова са којим је он у сметњи.

3.7. Упис у излазни фајл



Приликом уписа у излазни фајл потребно је одвојити секције, инструкције треба да буду у одговарајућем формату. као и саме променљиве које треба приказати преко регистара.

4. Опис решења

4.1. Модул главног програма

У модулу главног програма се налази *main* у коме се позива сваки корак који је описан у делу Концепт решења. Приликом позивања неких од корака извршава се и провера да ли је коректно одрађен тај део програма. Такође су дефинисана имена улазне и излазне датотеке.

4.2. Модул синтаксни анализатор

Прва функција која се извршава у овом модулу је функција *Do*, која након тога се позива функција *Q* и на даље извршавање алгоритма са рекурзивним спуштањем. Остале функције које имплементирају граматику су *S*, *L*, *E*. Остале функције су одрађене на вежбама из овог предмета.

4.3. Модул креирања инструкција

```
Variables* makeVariables(LexicalAnalysis& lex)
```

Функција на основу листе токена формира варијабле. Варијабле могу бити типа *MEM* или *REG*. За сваку варијаблу је потребно да се унесу њени аргументи као што су име, позиција и вредност уколико је има.
Параметри:

- lex - објекат класе *LexicalAnalysis* који садржи листу токена

```
GlobalDatas* makeGlobalDatas(LexicalAnalysis& lex)
```

Функција која формира листу лабела и функција на основу листе токена. Потребно је унети име лабеле или функције.
Параметри:

- lex - објекат класе *LexicalAnalysis* који садржи листу токена

```
Instructions* makeInstruction(LexicalAnalysis& lex,
GlobalDatas* global, Variables* variables)
```

Функција која формира листу инструкција на основу листе токена. Потребно је унети позицију, тип, dst, src скупове, као и неке додатке уколико их инструкција има као што су константа, лабела, offset.
Параметри:

- lex - објекат класе *LexicalAnalysis* који садржи листу токена
- global - листа лабела и функција
- variables - листа варијабли

4.4. Модул креирања тока управљања

```
void makeFlowControl(Instructions* instructions)
```

Функција која попуњава скупове *succ* и *pred* сваке функције. *Succ* скуп представља инструкције које су наследници текуће инструкције. *Pred* скуп представља инструкције које су претходници текуће инструкције.
Параметри:

- instructions - листа инструкција

```
void fillSetsIfBranchInstructions
(Instructions::iterator& iter, Instructions* instructions)
```

Функција која додатно попуњава *pred* и *succ* скупове. Ова функција се активира уколико имамо скокове као што су *b* и *bltz*. Тада је потребно да у *pred*

скуп инструкцијама на коју се скаче дода текућа инструкција (инструкција скока).
Параметри:

- iter - итератор који показује на текућу инструкцију
- instructions - листа инструкција

4.5. Модул анализирања животног века променљивих

```
void livenessAnalysis(Instructions& instructions)
```

Функција која попуњава *in* и *out* скупове свих инструкција. Попуњавање ових скупова се врши на основу алгоритма који је претходно био описан у Концепту решења.
Параметри:

- instructions - листа инструкција

```
bool variableExists(Variable* v, Variables variables)
```

Функција проверава да ли варијабла већ постоји у листи.
Параметри:

- v - варијабла
- variables - листа варијабли у којој је потребно проверити постојање

4.6. Модул креирања графа сметњи

```
void makeListOfVariable(Instructions& instructions)
```

Функција која формира листу варијабли које долазе у обзир за формирање матрице сметњи.
Параметри:

- instructions - листа инструкција

```
void initialisationMatrix()
```

Иницијализација матрице сметњи. Сви елементи се постављају на *__EMPTY__* односно нулу, док касније могу бити измењени.

```
void makeMatrix(Instructions& instructions, int num)
```

Ова функција мења елементе матрице из *__EMPTY__* у *__INTERFERENCE__* уколико између одговарајућих променљивих постоји сметња. Помоћне функције су `writeInterferenceOut` и `writeInterferenceIn`, које проверавају *in* и *out* скупове варијабли и на основу њих проверавају постојање сметњи.
Параметри:

- instructions - листа инструкција
- num - број променљивих које су типа , оне не долазе у обзир приликом посматрања сметњи

```
InterferenceGraph& buildInterferenceGraph
(Instructions& instructions, int num)
```

Функција која формира матрицу сметњи. Позива све потребне функције да би се матрица формирала и попунила.
Параметри:

- instructions - листа инструкција
- num - број променљивих које су типа , оне не долазе у обзир приликом посматрања сметњи

```
bool check(Variable* var, Variables listOfVar)
```

Функција која проверава да ли та варијабла већ постоји у листи.
Параметри:

- var - варијабла
- listOfVar - листа на основу које се проверава постојање

4.7. Модул креирања стека

```
bool notOnStack(string v, vector<string> s)
```

Функција која проверава да ли је варијабла већ постављена на стек или није.
Параметри:

- v - име варијабле
- s - листа имена варијабли које су на стеку

```
std::stack<Variable*>* doSimplification
(InterferenceGraph& ig, Variables& variables, int num)
```

Функција која поставља варијабле на стек. Потребно је да на основу степена варијабли и броју расположивих регистара одреди редослед по коме ће се варијабле постављати на стек и поставити их на стек.
Параметри:

- ig - граф сметњи
- variables - листа варијабли
- num -број варијабли које су типа *MEM*

```
    void reduceDegreeForVariables(Variables::iterator&
var, Variables& variables)
```

Функција која се активира након постављања неке варијабле на стек. Она треба да смањи степен варијабле која је постављена на стек на нулу, док за варијабле које су биле са њом у сметњи треба степен смањити за један.
Параметри:

- var - варијабла која је постављена на стек
- variables - листа свих варијабли

```
    void setDegreeForVariables(Variables& variables,
InterferenceGraph& ig, int number)
```

Функција која поставља свакој варијабли степен на основу броја са колико је променљивих у сметњи. Такође је потребно попунити листу варијабли са којима је у сметњи текућа варијабла.
Параметри:

- ig - граф сметњи
- variables - листа варијабли
- num -број варијабли које су типа *MEM*

4.8. Модул доделе ресурса

```
    bool doResourceAllocation(SimplificationStack*
simplificationStack, InterferenceGraph* ig, int num)
```

Функција која скида варијабле са стека и додељује им регистре. Исти регистар се не може доделити варијаблама које су у сметњи. Потребно је да све варијабле са стека добију одговарајући регистар.

Параметри:

- simplificationStack - стек на коме се налазе варијабле
- ig - граф сметњи
- num -број варијабли које су типа *MEM*

```
vector<Regs> makeListOfChooseRegs(InterferenceGraph* ig,
Variable* var, int num)
```

Функција која прави листу регистара који су додељени варијаблама које су са текућом варијаблом у сметњи. Уз помоћ ове листе можемо закључити који регистри се не могу доделити текућој варијабли.
Параметри:

- ig - граф сметњи
- var - текућа варијабла која је скинута са стека
- num -број варијабли које су типа *MEM*

4.9. Модул уписа у фајл

Класа `class WriteFile` садржи следеће методе за упис добијених резултата у излазну датотеку са екстензијом ".s" која садржи преведен и коректан MIPS 32bit асемблерски језик полазног програма.

```
void writeGlobal()
```
Функција уписује *.global* део у излазну датотеку.

```
void writeData()
```
Функција уписује *.data* део у излазну датотеку. Све варијабле типа *MEM* и њихове вредности.

```
void writeText()
```
Функција уписује *.text* део у излазну датотеку. У овом делу су исписане инструкције и сви њихови делови као и лабеле које се налазе у програму.

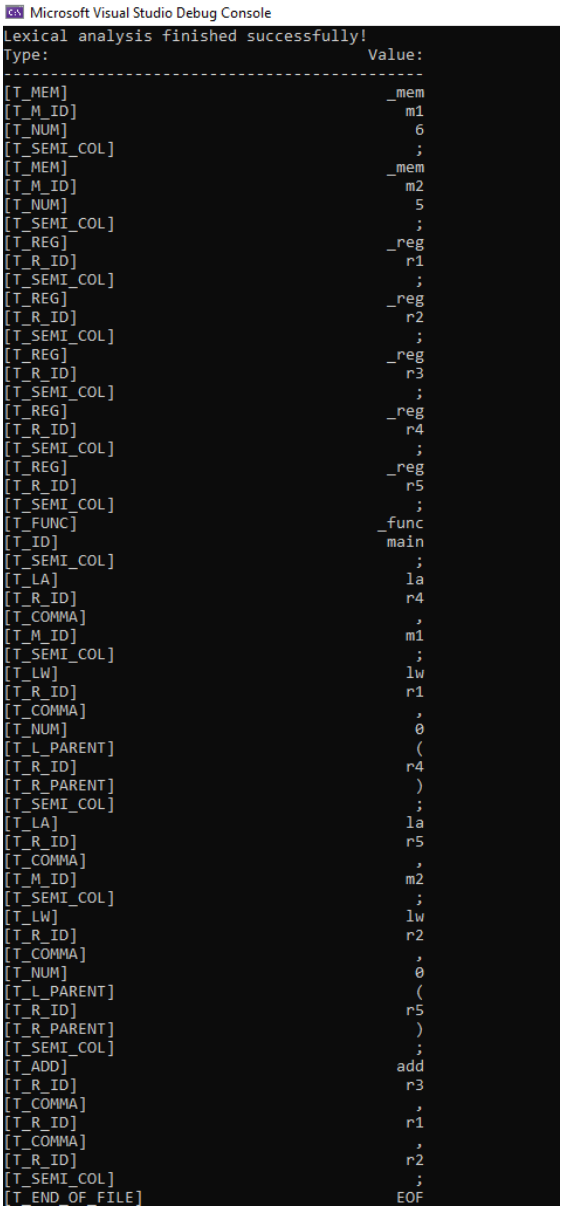
5. Верификација

Приликом верификације самог програма проверени су сви резултати. Имплементација самог пројектног задатка је извршена поступно и коришћењем вежби које су рађене на овом предмету било је омогућено да се сваки корак провери. Такође, неке кораке је било могуће проверити цртањем на папиру и упоређивањем са добијеним резултатима. Добијени резултати су се исписивали у конзоли или посматрали у debugger-у. Провером актуелних вредности променљивих, атрибута, број елемената итд. које је debugger приказивао, је могло да се уочи да ли је добијени резултат одговарајући и тачан.

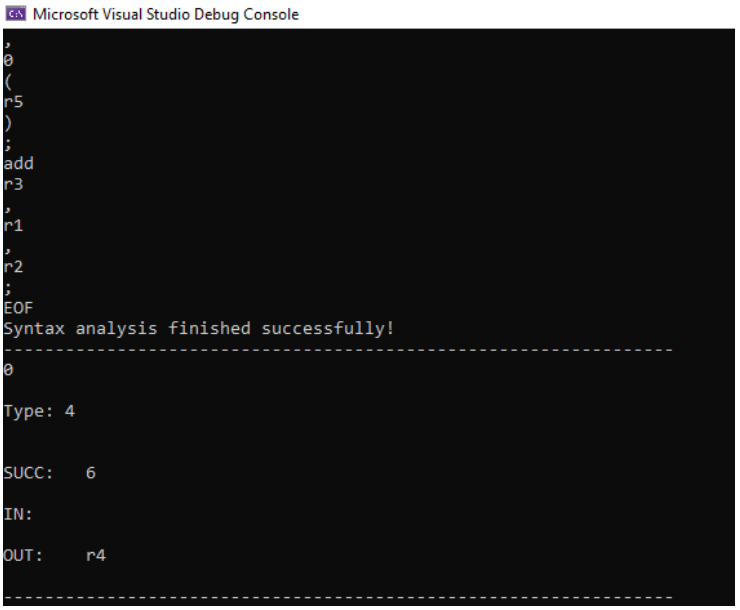
Неке од функција као што су *do* у *SyntaxAnalysis*, *doResourceAllocation*, имају проверу да ли је извршење било коректно. Такође уколико се јави *NULL* вредност за стек након формирања стека тј. извршења функције *doSimplification*, долази до преливања и програм се зарвшава.

**SyntaxAnalysis* класа је дата на једном термину вежби као и верификација. **Такође и делови као што су *LivenessAnalysis*, *InterferenceGraph*, *ResourceAllocation* су били делови вежби које су рађене на овом предмету и имплементиране током самих вежби.

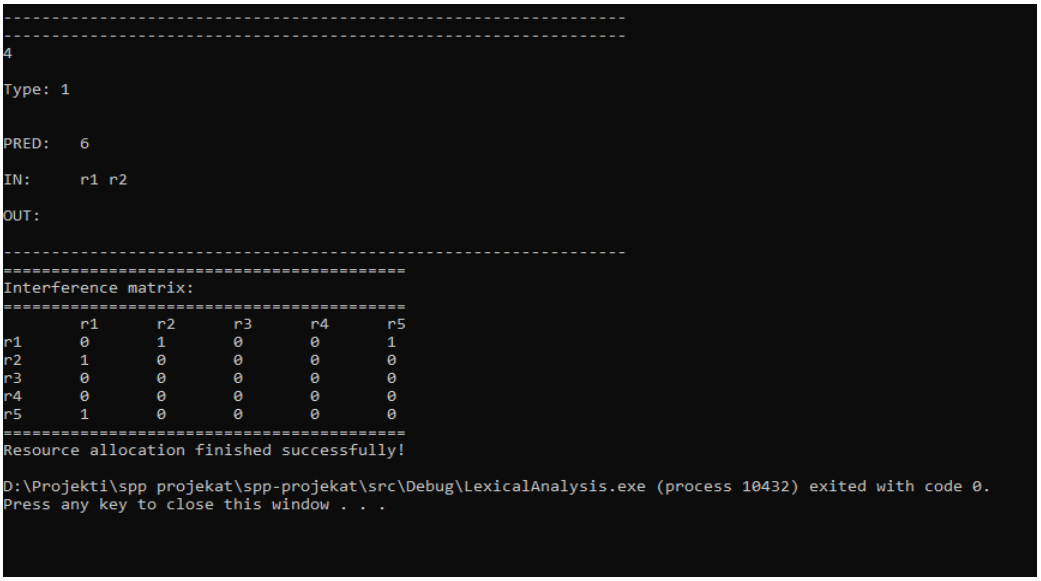
У наставку следе слике конзоле и исписа приликом коректних и некоретких извршавања.



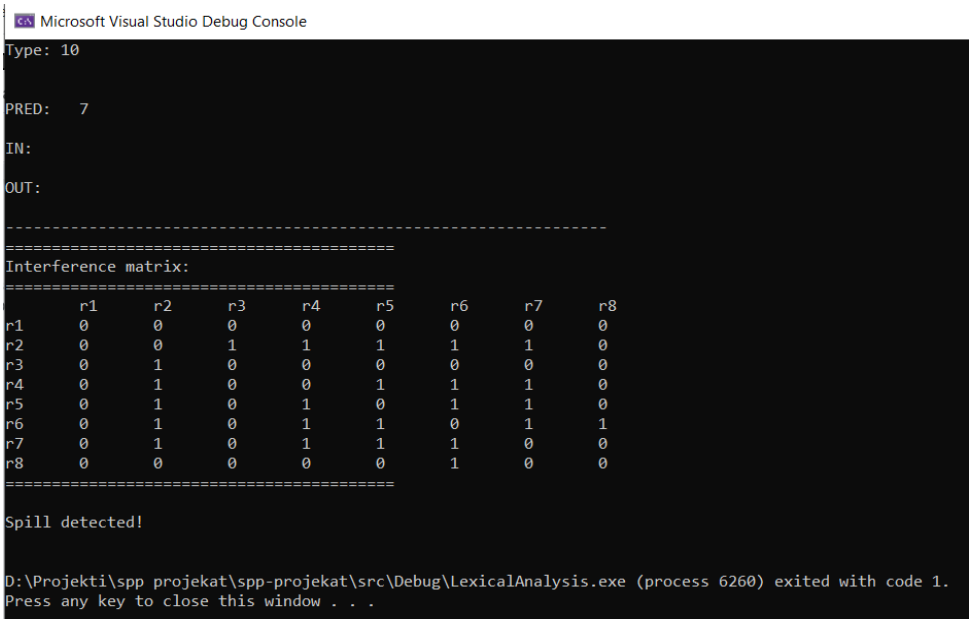
Пример успешне лексичке анализе



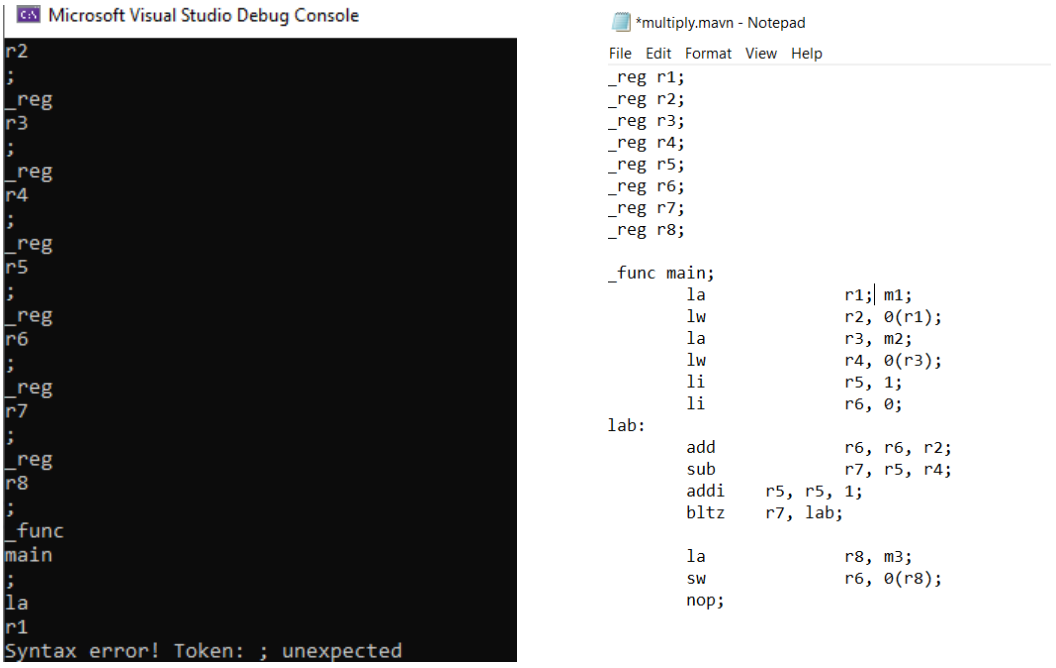
Пример успешне синтаксне анализе и испис инструкције



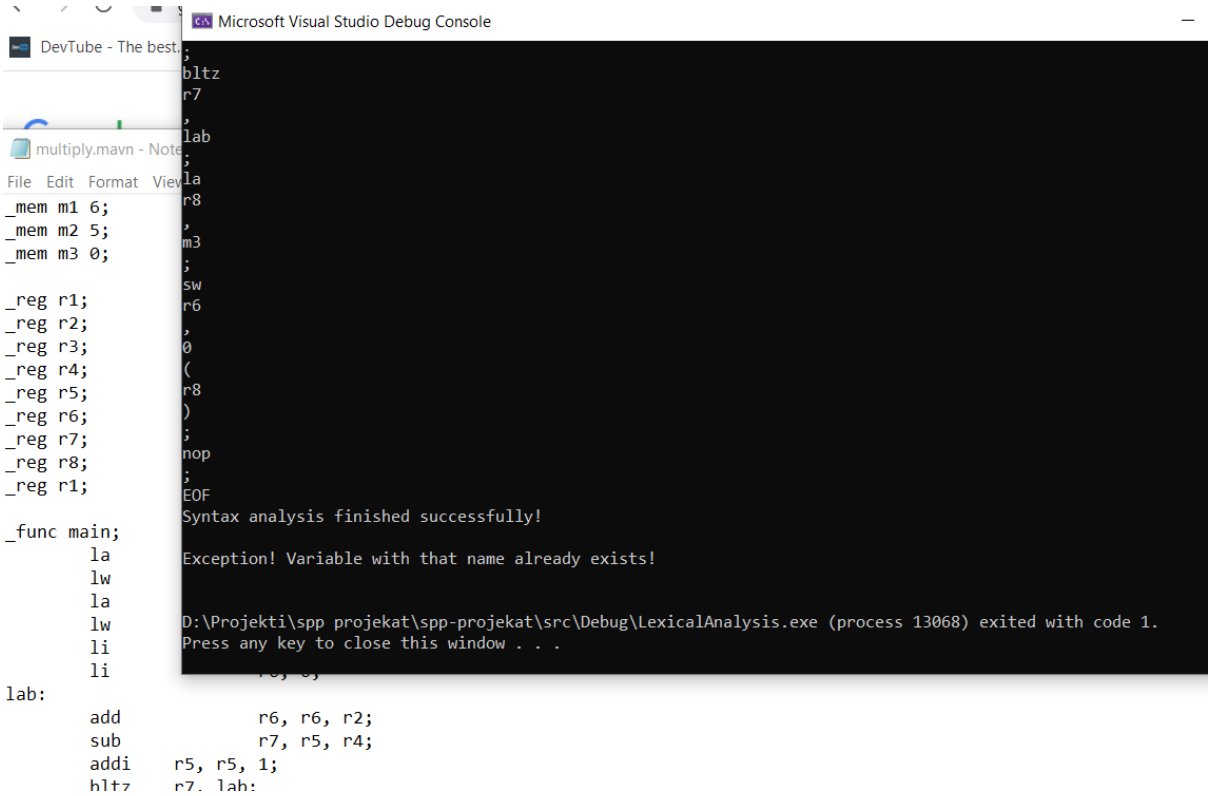
Пример успешног извршавања целог програма. Испис матрице сметњи и успешна додела ресурса



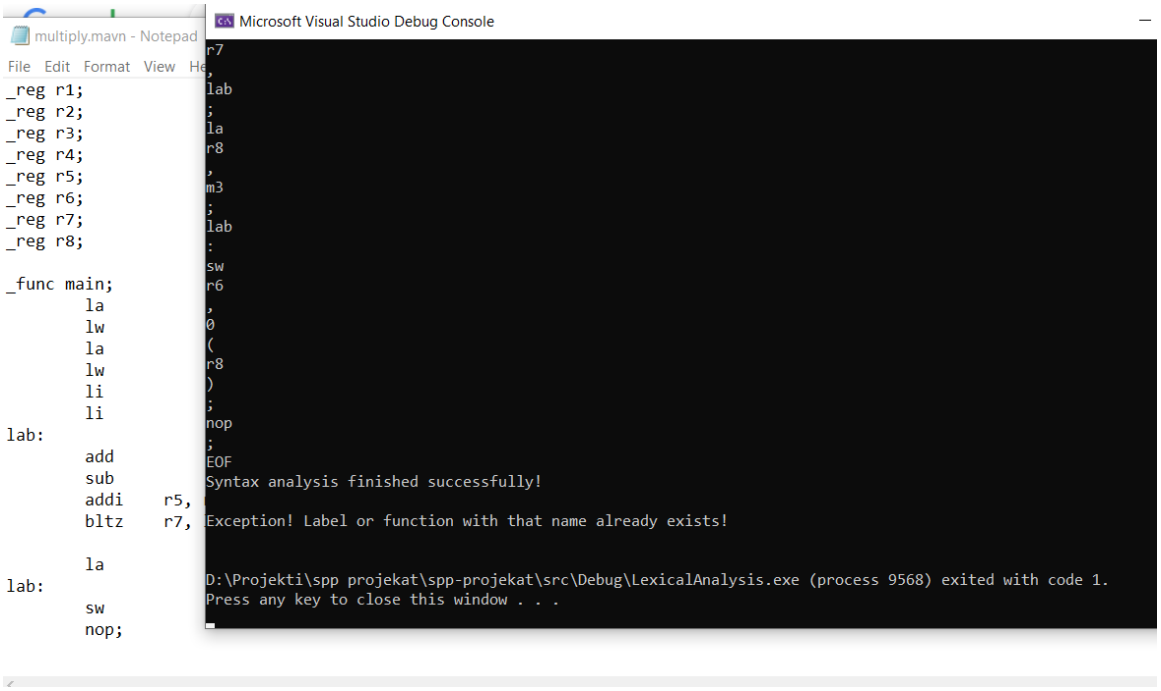
Пример појаве преливања



Пример грешке приликом синтаксне анализе. Неочеквани токен “ ; ” док треба да стоји “ , ”



Имена варијабли морају бити јединствена. Појава варијабле са истим именом



Имена лабели и функција морају бити јединствена. Појава лабеле са истим именом

6. Закључак

Након имплементације целог задатка, програм је извршаван на примерима који су дати уз пројекат. Решења су проверавана уз помоћ цртања на папиру, debugger-а и *QtSpim* симулатора. У конзоли се врши испис кључних корака током имплементације, као и поруке о успешности односно неуспешности изршења програма до краја.

Сам пројекат је модуларан и одвојен у целине, односно кораке по којима се врши превођење са вишег асемблерског језика на основни MIPS 32bit асемблерски језик.