|  |  |  |
| --- | --- | --- |
| |  | | --- | | **УНИВЕРЗИТЕТ У НОВОМ САДУ**  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  **НОВИ САД**  **Департман за рачунарство и аутоматику**  **Софтверско инжењерство и информационе технологије**  **Пројектни задатак**  **Студент: Далибор Малић**  **Број индекса: SW-50/2018**  **Предмет: Системска програмска подршка I**  **Тема рада: МАВН преводилац**  **Ментор рада: проф. др Миодраг Ђукић**  **Нови Сад, јун, 2020.** | |  | |

**Садржај**

[1. Увод 1](#_Toc43049215)

[1.1 Дефиниција превођења програма 1](#_Toc43049216)

[1.2 Задатак 2](#_Toc43049217)

[2. Анализа проблема 3](#_Toc43049218)

[3. Концепт рјешења 4](#_Toc43049219)

[3.1 Изглед улазне и излазне датотеке 4](#_Toc43049220)

[3.2 Анализа / разумијевање изворног кода 5](#_Toc43049221)

[3.2.1 Лексичка анализа 5](#_Toc43049222)

[3.2.2 Синтаксна анализа 5](#_Toc43049223)

[3.3 Синтеза / генерисање одредишног кода 6](#_Toc43049224)

[3.3.1 Анализа животног вијека промјенљивих 6](#_Toc43049225)

[3.3.2 Додјела ресурса 6](#_Toc43049226)

[3.3.2.1 Креирање графа/матрице сметњи 6](#_Toc43049227)

[3.3.2.2 Упрошћавање 7](#_Toc43049228)

[3.3.2.3 Фаза избора 7](#_Toc43049229)

[3.3.3 Генерисање одредишног фајла 7](#_Toc43049230)

[4. Програмско рјешење 8](#_Toc43049231)

[4.1 Посао *main* функције 8](#_Toc43049232)

[4.2 Предњи дио компајлера 8](#_Toc43049233)

[4.2.1 Лексички анализатор 8](#_Toc43049234)

[4.2.2 Синтаксни анализатор 9](#_Toc43049235)

[4.3 Задњи дио компајлера 11](#_Toc43049236)

[4.3.1 Алгоритам анализе животног вијека промјенљивих 11](#_Toc43049237)

[4.3.2 Додјела ресурса 11](#_Toc43049238)

[4.3.2.1 Креирање графа/матрице сметњи 11](#_Toc43049239)

[4.3.2.2 Упрошћавање 12](#_Toc43049240)

[4.3.2.3 Фаза избора 12](#_Toc43049241)

[4.3.3 Генерисање одредишног фајла 12](#_Toc43049242)

[5. Верификација 13](#_Toc43049243)

[5.1 Симулација програма са одредишним кодом 13](#_Toc43049244)

[5.2 Успјешно преведени тестни случајеви 14](#_Toc43049245)

[5.3 Неуспјешно преведени тестни случајеви 15](#_Toc43049246)

[5.3.1 Недовољан број регистара 15](#_Toc43049247)

[5.3.2 Лексичке и синтаксне грешке 15](#_Toc43049248)

[5.3.3 Семантичке грешке 16](#_Toc43049249)

[5.3.3.1 Постојање дупликата промјенљивих, функција и лабела 16](#_Toc43049250)

[5.3.3.2 Непостојећа лабела и промјенљива 16](#_Toc43049251)

[5.3.3.3 Погрешно име промјенљивих 17](#_Toc43049252)

**Списак слика**

Слика 1 Компајлер 1

Слика 2 Улазна датотека 4

Слика 3 Излазна датотека 4

Слика 4 Формула за анализу животног вијека 6

Слика 5 Алгоритам анализе животног вијека промјенљивих 11

Слика 6 Симулација програма 13

Слика 7 Успјешно превођење 14

Слика 8 Кодови успјешних превода 14

Слика 9 Недовољан број регистара 15

Слика 10 Лексичке и синтаксне грешке 15

Слика 11 Дупликати имена 16

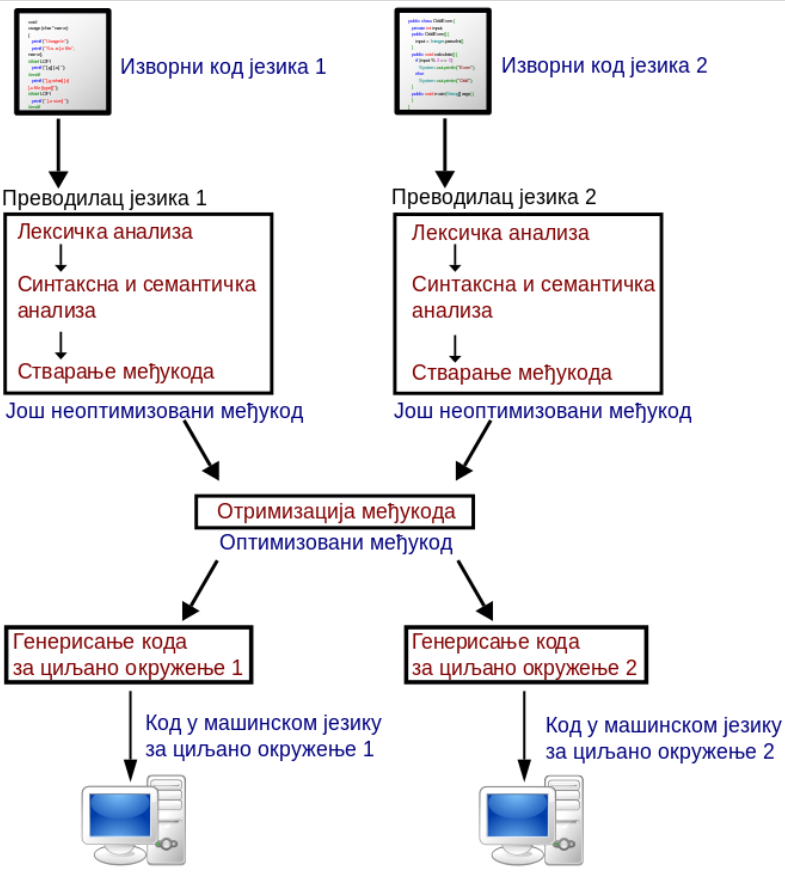
Слика 12 Непостојећи елементи 16

Слика 13 Погрешно име промјенљивих 17

# Увод

## Дефиниција превођења програма

Програмски преводилац (енгл. *Compiler*) је рачунарски програм (или низ програма) који трансформише код једног програмског језика у други програмски језик. Тај процес се обично одвија кроз двије главне етапе: анализу изворног кода (*front end*) и синтезу објектног кода (*back end*).



Слика 1 Компајлер

У етапи анализе (разумијевања кода), изворни програм се преводи у одређену посредну репрезентацију (међукод), која је погодна за даље манипулисање. Ова етапа се може подијелити у три фазе: лексичку, синтаксну и семантичку анализу.

Међукод је обично дизајниран да изведе оптимизацију на форми различитој од изворног или одредишног кода. Ова независност има за циљ да омогући да генеричке оптимизације буду подијељене између верзија компајлера подржавајући разне језике и циљне процесоре.

Етапа синтезе, која има за циљ да од међурепрезентације кода креира програм, представља процес који узима у обзир излазни језик, који је најчешће машински језик или асемблерски језик, архитектуру рачунара на којој се врши превођење, окружење под којим се извршава и др. Дијели се на три фазе: генерисање међукода, оптимизација међукода и генерисање извршног кода.

Ефикасност компајлера да креира програм који ће се брже извршавати не зависи нужно од времена које утроши у фази превођења. Другим ријечима, не мора да значи да ће један преводилац компајлирати бржи програм уколико је утрошио мање времена за процес компајлирања од неког другог преводиоца. Чешћа је ситуација да, што је преводилац боље конструисан, то он мора више посла да уради (тј. више времена да потроши) приликом фазе компајлирања, али су зато програми које он производи бржи. Дакле, то не значи да је неопходно да се компајлери пишу у брзим програмским језицима, већ да се пажљиво конструишу. Савремени компајлери се често ипак пишу у брзим програмским језицима, као што су Це и Це++ како би се, поред убрзања извршавања програма добило и на убрзању у фази превођења, односно, убрзању развоја софтвера.

## Задатак

У оквиру пројектног задатка потребно је реализовати МАВН преводилац који преводи програме са вишег асемблерског језика на основни МИПС 32-битни асемблерски језик. Преводилац треба да подржава детекцију лексичких, синтаксних и семантичких грешака као и генерисање одговарајућих извјештаја о евентуалним грешкама. Излаз из преводиоца треба да садржи коректан асемблерски код који је могуће извршавати на МИПС 32-битној архитектури (симулатору).

МАВН (Мипс Асемблер Високог Нивоа) је алат који преводи програм написан на вишем асемблерском језику на основни МИПС 32-битни асемблерски језик. Виши МИПС асемблерски језик служи лакшем асемблерском програмирању јер уводи концепт регистарске промјенљиве. Регистарске промјенљиве омогућавају програмерима да приликом писања инструкција користе промјенљиве умјесто правих ресурса. Ово знатно олакшава програмирање јер програмер не мора да води рачуна о коришћеним регистрима и

њиховом садржају.

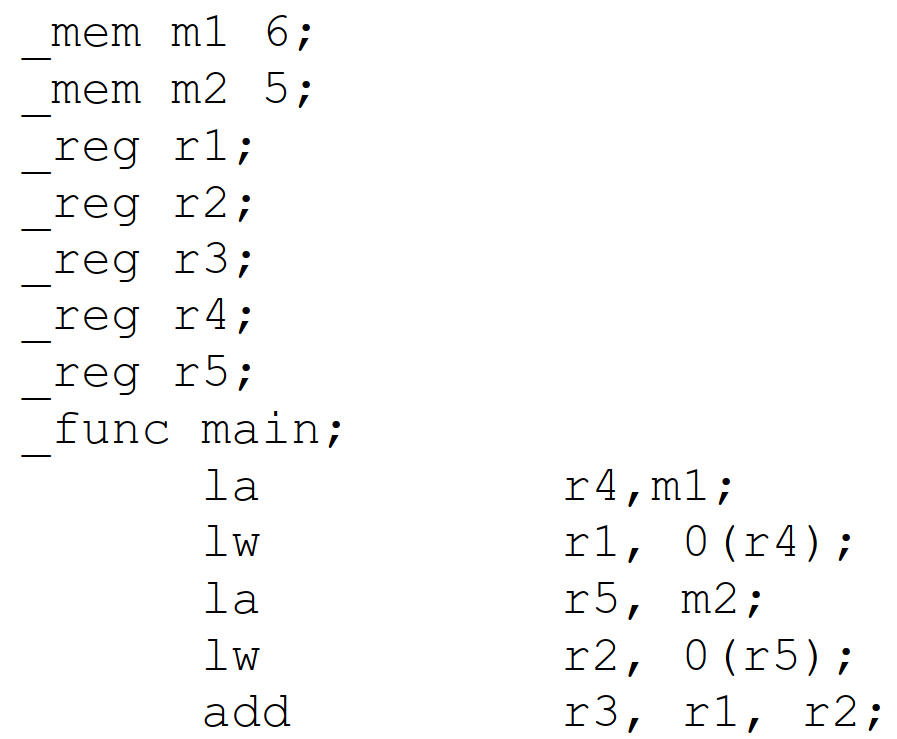
# Анализа проблема

МАВН преводилац има задатак да преведе програме са вишег асемблерског језика на основни МИПС 32-битни асемблерски језик. Прије него што преводилац може да генерише извршни код, предњи дио се мора постарати о томе да ли је изворни код валидан. Валидност изворног кода се огледа у томе да ли прати правила која су постављена вишим програмским језиком. Ова правила се састоје од провјера да ли све „ријечи” изворног кода чине „ријечи” програмског језика, да ли су те „ријечи” структуиране коришћењем правила за формирање „реченица” у програмском језику, као и да ли формиране „реченице” имају валидно значење. Ове проблеме треба ријешити кроз фазе лексичке и синтаксне анализе, у којој се обавља и креирање инструкција. Уколико изворни код задовољава све провјере, онда се на основу њега формира међурепрезентација, која се даље предаје задњем дијелу преводиоца. Задњи дио се реализује кроз фазе анализе животног вијека промјенљивих, креирања графа/матрице сметњи, додјеле ресурса и генерисања одредишног кода. Дати језик подржава 10 инструкција : *add, addi, b, bltz, la, li, lw, nop, sub, sw*, на које ће бити још имплементиране двије аритметичко-логичке: *and, or*, као и једна инструкција приступа меморији *lb*. Преводилац подржава детекцију лексичких, синтаксних и семантичких грешака као и генерисање одговарајућих извјештаја о евентуалним грешкама.

# Концепт рјешења

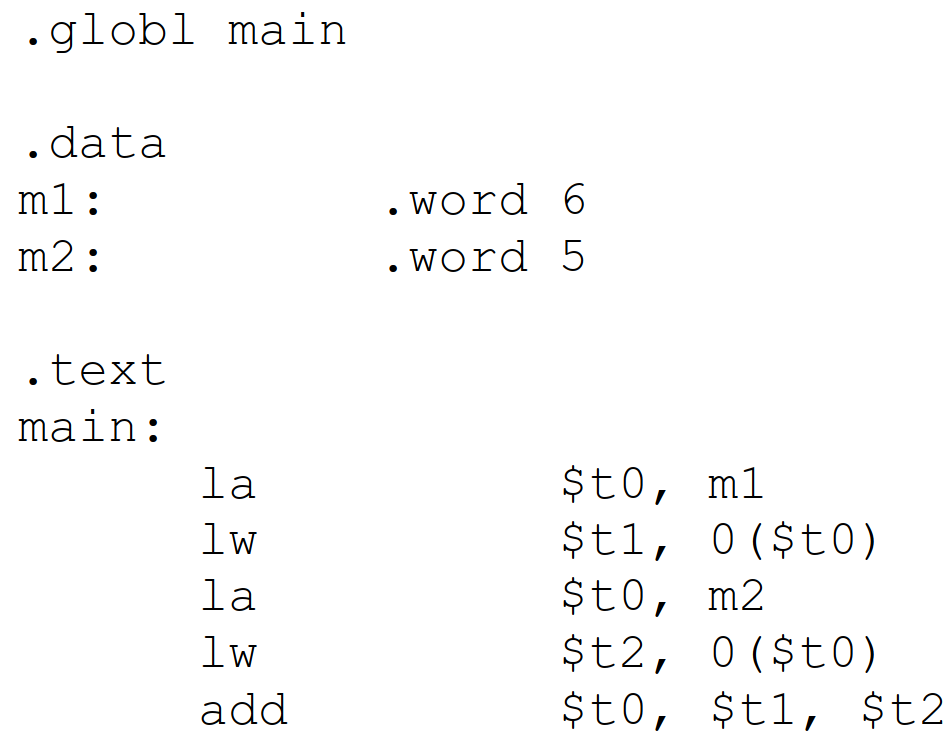
## Изглед улазне и излазне датотеке

Примјер изгледа улазне датотеке / програма на вишем асемблерском језику:



Слика 2 Улазна датотека

Изглед излазне датотеке / програм преведен на основни МИПС 32-битни асемблерски језик:



Слика 3 Излазна датотека

У наставку ће бити приказани концепт рјешења за предњи и задњи дио преводиоца.

## Анализа / разумијевање изворног кода

### Лексичка анализа

Приликом анализирања „ријечи” изворног кода, преводилац користи дефиницију програмског језика и упоређује да ли „ријечи” на које наилази представљају валидне „ријечи” у том програмском језику. Овај процес се назива лексичка анализа.

Лексички анализатор чита карактер по карактер из изворног кода и идентификује цјелине (лексеме), то ради уз помоћ коначног детерминистичког аутомата, преко којег добија свако наредно дозвољено стање за тренутни токен. Након идентификовања лексема, свакој од њих се придружује одговарајућа лексичка категорија, као и јединствена одговарајућа ознака категорије која је придружена лексеми (токен). Свака лексема мора да припада неком токену да би била препозната од стране лексера. Након процесирања изворног кода од стране лексичког анализатора, излаз из ове фазе представља секвенца токена, која представља улаз у наредну фазу превођења. Лексички анализатор не води рачуна о томе да ли је неки идентификатор промјенљива, функција, структура и сл. С обзиром да је ово једина фаза у којој се оперише директно над изворним кодом, задатак лексера јесте и да чува информацију о називу идентификатора који је препознао, вриједности дословне бројевне вриједности коју је препознао и сл.

### Синтаксна анализа

Слично као што синтакса природног језика проучава правила која одређују како се ријечи комбинују у реченице у датом језику, синтакса програмских језика представља низ правила која дефинишу комбинацију токена за које се сматра да дају исправно структуиран фрагмент кода у траженом програмском језику. Дакле, у синтаксној анализи се провјерава да ли су токенизоване лексеме које се добијају из фазе лексичке анализе склопљене пратећи претходно дефинисану граматику програмског језика.

Да се синтакса програмског језика (неки његови дијелови) не би могли дефинисати на више начина, за процес превођења се каже да је синтаксно-вођено. Синтакса МАВН језика описана је граматиком:

*Q → S ; L E → add rid, rid, rid E → bltz rid, id*

*S → \_mem mid num E → addi rid, rid, num E → nop*

*S → \_reg rid E → sub rid, rid, rid*

*S → \_func id E → la rid, mid*

*S → id: E E → lw rid, num(rid)*

*S → E E → li rid, num*

*L → eof E → sw rid, num(rid)*

*L → Q E → b id*

Овa граматика је, због додавања инструкција *and, or* и *lb* проширена са:

*E → lb rid, num(rid) E → and rid, rid, rid E → or rid, rid, rid*

Терминални симболи МАВН језика су:

*: ; , ( )*

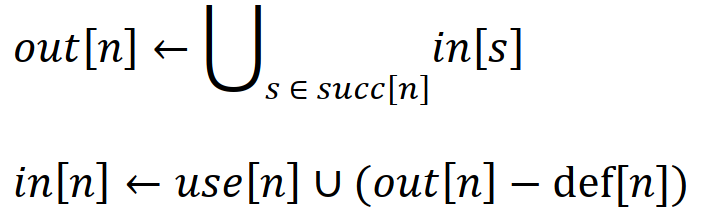
*\_mem, \_reg, \_func, num id, rid, mid, eof, add, addi, sub, and, or, la, lb, lw, li, sw, b, bltz, nop*

У току ове фазе биљеже се све функције, промјенљиве, лабеле и инструкције, користе се бројачи за означавање редног броја инструкција, регистарских промјенљивих, лабела које стоје уз инструкције тј. држе редни број те инструкције. У току записивања инструкција попуњавају се и *dst*, *src*, *def*,и *use* листе варијабли те инструкције. Када се заврши анализа, прави се поредак инструкција, тј. свакој инструкцији се попуњавају листе претходника и сљедбеника.

## Синтеза / генерисање одредишног кода

### Анализа животног вијека промјенљивих

Да би се омогућило да различите промјенљиве које нису истовремено у употреби дијеле исти ресурс, неопходна је информација о животном вијеку промјенљиве. Животни вијек промјенљиве започиње дефиницијом промјенљиве (упис почетне вриједности), а завршава се посљедњом употребом (посљедње очитавање садржаја промјенљиве). У имплементацији се попуњавају листе улазних и излазних промјенљивих сваке инструкције, по формули:



Слика 4 Формула за анализу животног вијека

### Додјела ресурса

#### Креирање графа/матрице сметњи

Проблем који онемогућава да се исти регистар додијели двијема привременим промјенљивима назива се сметња. Најчешћи узрок сметње је преклапање опсега животног вијека промјенљивих. На основу резултата добијених из анализе животног вијека промјенљивих правимо граф сметњи. Колоне и редови матрице су исти и представљају све промјенљиве у програму. На пресјеку одговарајуће колоне и реда је уписан индикатор да ли постоји сметња између промјенљиве у колони и промјенљиве у реду. Сметња постоји између свих промјенљивих које се дефинишу у инструкцији и свих промјенљивих које излазе из инструкције.

#### Упрошћавање

Фаза упрошћавања добија на улаз граф сметњи и број доступних регистара, а њен резултат је стек. Циљ ове фазе је да прође кроз читав граф, да прво на стек ставља промјенљиве које имају мањи број сметњи од броја доступних регистара, у том процесу све промјенљиве које су у сметњи са овом, додатом промјенљивом, губе ту сметњу и смањује им се број сметњи (ранг), па имају веће шансе да буду наредне убачене. Процес се наставља све док се не испразни граф.

#### Фаза избора

Након што је граф успјешно упрошћен потребно је обојити чворове, тј. додјелити им одговарајуће регистре. Редосљед бојења чворова је одређен стеком на који су се чворови постављали приликом упрошћавања графа. Чворови ће бити бојени обрнутим редосљедом од оног којим су скидани са графа. Придржава се и правило које се користи приликом бојења, а то је да се чвору не додјели боја која је већ додјељена неком од сусједних чворова.

### Генерисање одредишног фајла

Ово је посљедња фаза и она укљујује генерисање излазне датотеке са екстензијом “.s” која садржи преведен и коректан МИПС 32-битни асемблерски језик полазног програма, која поштује сљедеће :

- Уписивање меморијских промјенљивих у секцију за податке - .data водећи рачуна о синтакси асемблерског језика

- Уписивање инструкција у програмску секцију - .text

- Генерисање имена функција као глобалних симбола - .globl и као лабелу на прву њену инструкцију

# Програмско рјешење

## Посао *main* функције

int main();

Унутар главне функција програма се врши одабир улазног фајла кроз функцију *getFilePath*. Ова функција даје мени за одабир, а фајлове чита из предодређеног директоријума уз помоћ библиотеке *dirent.* Главна функција обавља након тога све фазе МАВН преводиоца у току чега враћа резултате извршавања у виду порука или грешки. Након што се све фазе успјешно изврше, слиједи генерисање излазне датотеке путем функције *toMips.* Њен задатак је да испише одредишни код у излазну датотеку, у формату који је раније представљен.

## Предњи дио компајлера

### Лексички анализатор

Лексички анализатор чита карактер по карактер из изворног кода и идентификује цјелине уз помоћ коначног детерминистичког аутомата, преко којег добија свако наредно дозвољено стање за тренутни токен. Након идентификовања лексема, свакој од њих се придружује одговарајућа лексичка категорија, као и јединствена одговарајућа ознака категорије која је придружена лексеми - токен. Задатак лексера јесте и да чува информацију о називу идентификатора који је препознао, вриједности дословне бројевне вриједности коју је препознао и сл.

*enum TokenType*

Подржани токени које лексички анализатор генерише

*map<int, map<char,int>> stateMachine;*

Аутомат прелаза стања, мапа која чува стања као кључ, и матрицу прелаза стања као вриједност

*static const TokenType stateToTokenTable[D\_NUM\_STATE];*

Стања коначног детерминистичког аутомата:

*static const char supportedCharacters[D\_NUM\_OF\_CHARACTERS];*

Карактери које лексички анализатор подржава

*static const int stateMatrix[D\_NUM\_STATE][D\_NUM\_OF\_CHARACTERS];*

Матрица прелаза стања

Све токене лексер чува у листи, коју просљеђује наредној фази превођења, синтаксном анализатору.

### Синтаксни анализатор

*SyntaxAnalysis::SyntaxAnalysis(LexicalAnalysis& lex)*

Синтаксни анализатор добија објекат лексичког анализатора на улаз који служи да се преузме из њега листа добијених токена. Атрибути класе су :

* *variableList*, *labelList*, *instructionList*, *functionList* – листе показивача на објекте класа *Variable*, *Label*, *Instruction* у којe смјештамо све варијабле, лабеле (и функције су лабеле само у посебној листи) добијене у анализатору, које су прије тога прослиђене методама *variableInList*, *labelInList*, *funcNameCorrect*, ради провјера исправности токена, у којима се гледа да ли дати токен постоји већ и да ли је исправно написан,
* *registryNum* – бројач регистарских промјенљивих,
* *instructionNum* – бројач инструкција,
* *currentToken* – тренутни токен који се обрађује,
* *tokenIterator* – итератор за листу токена добијених од лексичког анализатора,
* *errorFound* – индикатор грешке

Методе које припадају класи су:

* добављање листи инструкција, варијабли, функција и лабела (*getInstructions*, *getVariables*, *getFunctions*, *getLabels*),
* придруживање лабеле првој инструкцији која се налази под лабелом (*branchInstructionLabel*),
* тражење промјенљиве у листи (*findVariable*),
* додавање промјенљиве тренутној инструкцији у листе *def* и *use* на основу типа тренутне инструкције као и типа тренутне промјенљиве – *dst/src* (*fillVariableLists*),
* добављање сљедећег токена из листе (*getNextToken*),
* провјера исправности тренутног токена и прелаз на сљедећи (*eat*),
* додавање претходника и сљедбеника свакој инструкцији проласком кроз све инструкције редом; уколико постоји гранање, узима се лабела и проналази инструкција на коју показује (*fillSuccAndPred*),
* обављање синтаксне анализе (*Do*).

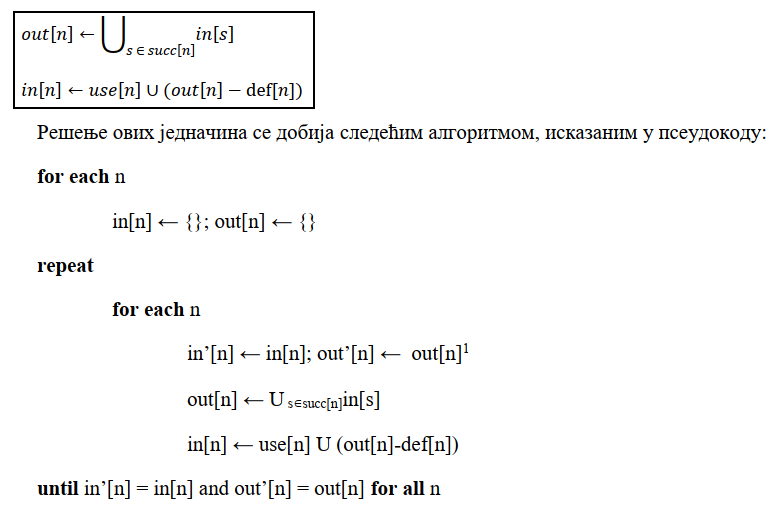
Унутар методе *Do* се провјерава синтакса МАВН језика дата кроз нетерминалне симболе *Q*, *S, L* и *E*, који су представљени као засебне методе. Први корак анализе је *Q*() који у себи (*Q → S ; L)* позива методе *S*(), обрађивање наредног очекиваног токена ; и *L*().

* Метода *S*(), које означава да се тренутно потенцијално обрађује исказ, се провјеравају тренутни токени, који у том тренутку могу бити индикатор меморијске или регистарске промјенљиве, функције, лабела, или ако није ниједно од претходног, ради се о изразу који се онда обрађује кроз методу *E*(). Уколико се ради о исказу, креира се промјенљива/лабела која се провјерава на валидност и потом додаје у одговарајућу листу.
* Метода *L*() има два могућа исхода, ако је тренутни токен ознака за крај датотеке тада се провјерава постојање *main* функције, да ли су све искоришћене лабеле дефинисане и позива се метода *fillSuccAndPred* којом се свакој инструкцији додају њени сљедбеници и претходници, након свега слиједи завршетак анализе. Уколико се није дошло до краја позива се поново метода *Q*() (рекурзија).
* Метода *E*() провјерава тренутни токен, и уколико се не ради о некој од дефинисаних инструкција, баца се грешка. У супротном, открива се о којој инструкцији је ријеч, креира се та инструкција, пуне се њене листе *dst, src, def* и *use*, поставља позиција инструкције и додаје у листу инструкција. Уколико унутар инструкције постоје бројевне вриједности, оне се чувају у атрибуту инструкције *m\_number*, ако је инструкција гранања, показивач на лабелу се чува у атрибуту инструкције *m\_label.*

## Задњи дио компајлера

### Алгоритам анализе животног вијека промјенљивих

Скуп *out[n]* је унија *„in“* скупова свих насљедника чвора *n*, односно унија свих промјенљивих *s* које припадају скупу *„in“* сваког насљедника чвора *n*. Скуп *in[n]* чине све промјенљиве из скупа *use[n]* плус промјенљиве које јесу у *out[n]*, али нису у *def[n]*. Одатле слиједе једначине тока података за анализу животног вијека промјенљивих.



Слика 5 Алгоритам анализе животног вијека промјенљивих

Функција *livenessAnalysis* извршава овај алгоритам пролазећи кроз све инструкције и мјења листе улазних и излазних промјенљивих. Она ће изменити све инструкције све док посљедња и претпосљедња итерација измјена не буду дале исте резултате, тј. док не буду имале исте скупове улазних и излазних промјенљивих.

### Додјела ресурса

#### Креирање графа/матрице сметњи

Граф сметњи је представљен класом *InterferenceGraph* која посједује атрибуте: показивач на листу промјенљивих и матрицу сметњи. Матрица сметњи се гради на основу датих листа инструкција и промјенљивих, преко функције *buildInterferenceGraph*. Унутар методе се издвајају све регистарске промјенљиве преко функције *onlyRegistryVariables*, а након тога се пролази кроз све инструкције, и за сваку инструкцију се праве сметње између промјенљивих које се дефинишу и које излазе из инструкције. Сметње се чувају у матрицу, и постављају у оба правца. Нпр. ако је сметња између 2. и 3. промјенљиве, она се биљежи у матрици на позицијама [2][3] и [3][2].

#### Упрошћавање

Ову фазу обавља функција *doSimplification* којој просљеђујемо граф сметње и број доступних регистара. Пролази се кроз све регистарске промјенљиве и узима ред из матрице са индексом промјенљиве. Пролазећи кроз тај ред, бројимо број сметњи које има са неким другим промјенљивима и чувамо у мапу (ранг чвора). Након тога, пролазимо кроз мапу и прво убацујемо на финални стек све промјенљиве које имају мање сметњи од броја регистара, а свима које су у сметњи са ово промјенљивом смањујемо број сметњи за 1. Када прођемо кроз све варијабле ранга мањег од *\_\_REG\_NUMBER\_\_* значи да су нам остале само које су прво имале већи ранг али како смо избацивали ове са мањим бројем, тако су ове губиле сметње и кроз наредне кораке се могу и оне убацити. Ипак ако остане нека која има већи ранг од дозвоњеног, потребно ју је прелити у меморију.

#### Фаза избора

Алокација регистара се врши у функцији *doResourceAllocation*, којој просљеђујемо стек добијен фазом упрошћавања и граф сметњи. Показивач на промјенљиву која је тренутно у обради – *currentVariable*, претходну промјенљиву – *previousVariable,* листа промјенљивих којима је додјељен регистар – *save*. Алокација регистара се обавља све док има промјенљивих на стеку. Прво се скида елемент са врха стека, уколико се ради о првом елементу, додаје се у листу оних којима је додјељен регистар. Ако није прва, провјерава се која боја је доступна овој промјенљивој, тј. пролази се кроз све промјенљиве које имају сметњу са овом промјенљивом, и налази се боја коју ниједна промјенљива у сметњи нема. Овај процес се одвија до испражњавања стека, али се може десити и да се прије тога испразни листа доступних регистара за неку промјенљиву, и у том случају се избацује грешка, јер неке промјенљиве неће добити регистар.

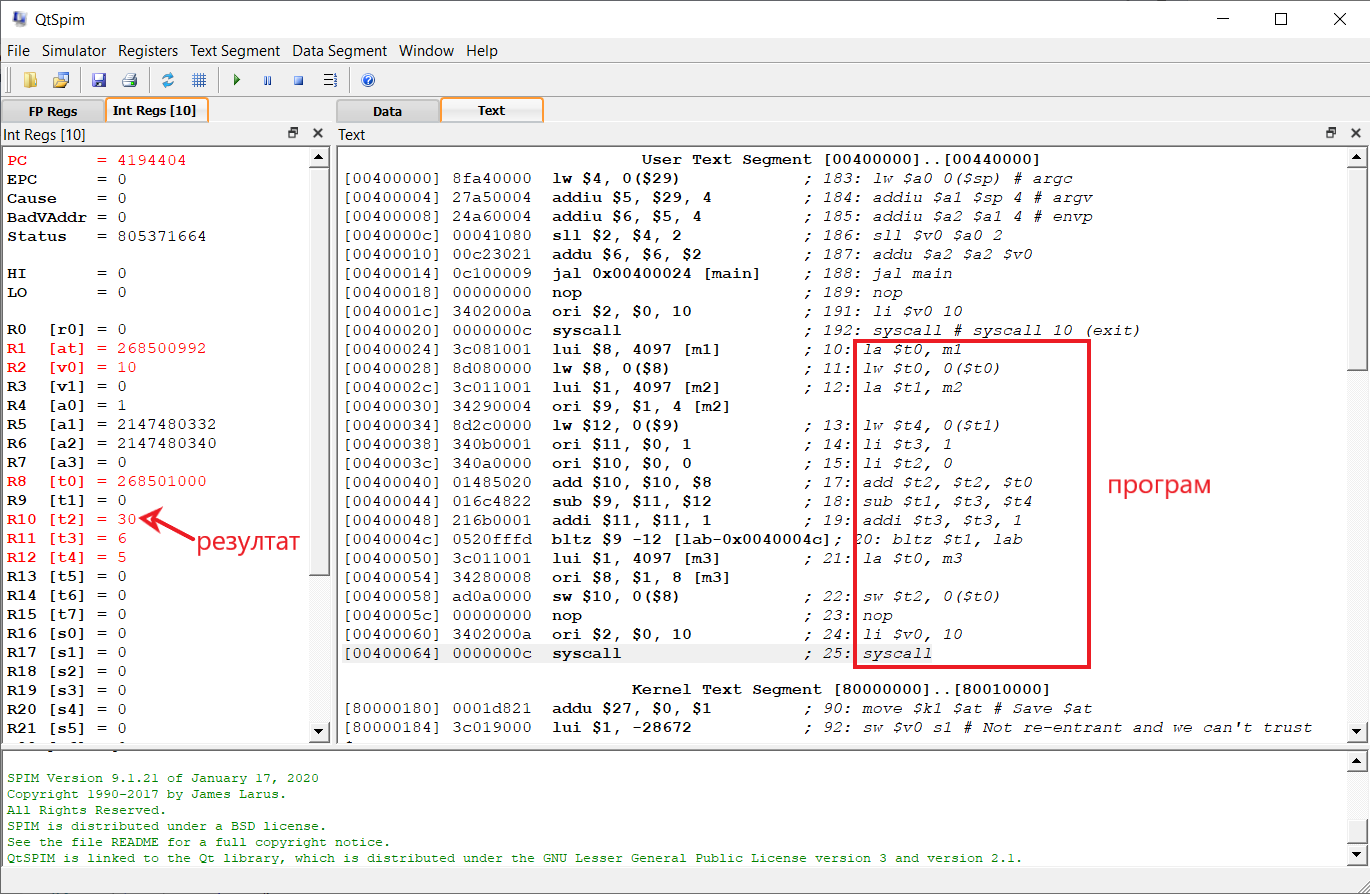
### Генерисање одредишног фајла

Задњи корак је генерисање излазог фајла, који мора бити извршив МИПС фајл. Овај посао обавља функција *toMips* којој се просљеђује референца објекта класе *SyntaxAnalysis* и име фајла који смо изабрали за улазни. У првом дијелу фајла, у секцији *.globl* записујемо све функције, у секцији *.data* записујемо све меморијске промјенљиве, и у секцији *.text* записујемо све инструкције, лабеле, лабеле функција. На крају је додата и наредба изласка из МИПС програма, да би се исти извршио у симулатору.

# Верификација

## Симулација програма са одредишним кодом

Исправност рада програма је провјерена уз помоћ *QtSpim* симулатора. Примјер извршавања симулације програма *multiply.mavn,* који је преведен у *multiply\_result.s* је дат на слици.

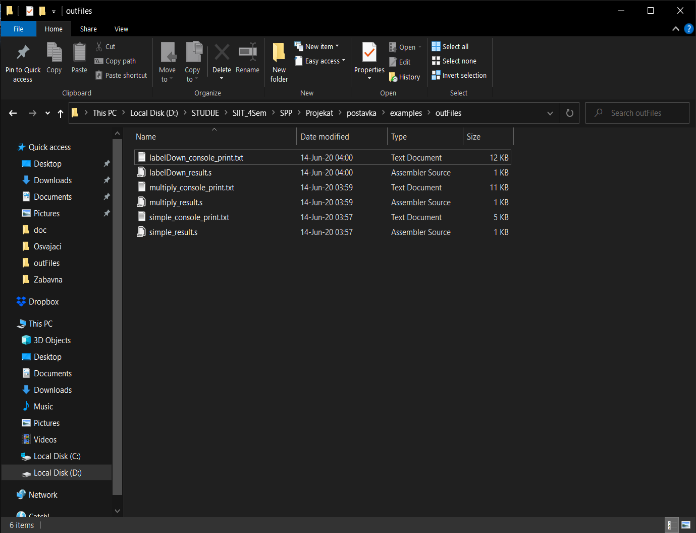
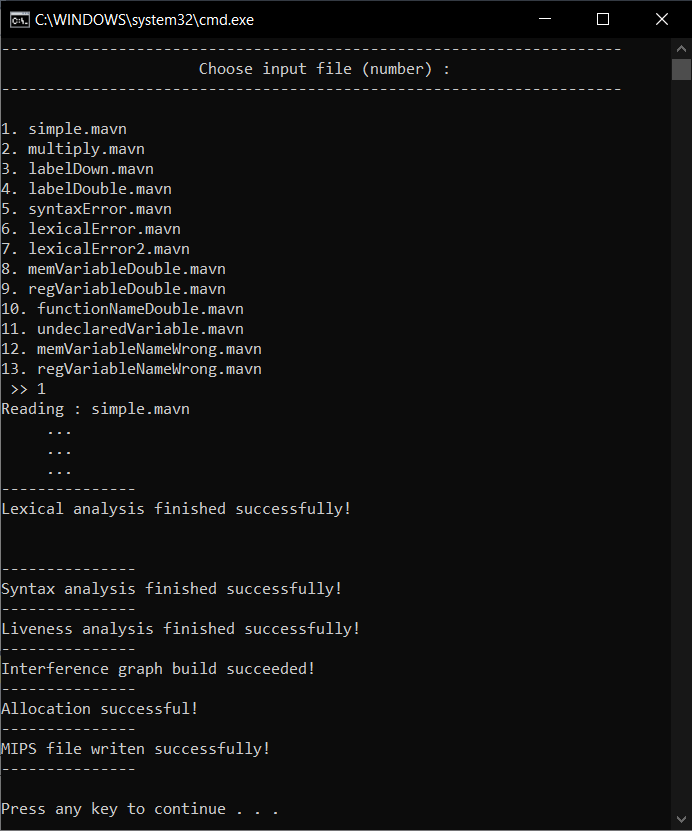


Слика 6 Симулација програма

У наставку су приказана извршавања тестних случајева, у којима се провјерава рад преводиоца за подржане инструкције, бацање изузетака у случају функција, лабела и варијабли са именом које већ постоји, лексичке и синтаксне грешке, као и случај када се користе инструкције гранања на лабелу која до тада није дефинисана.

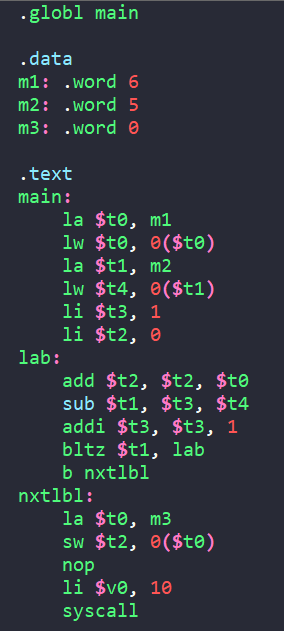
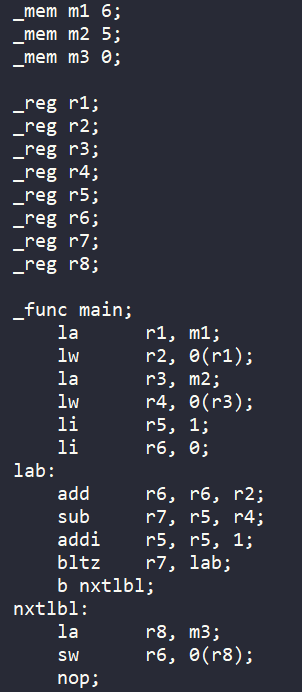
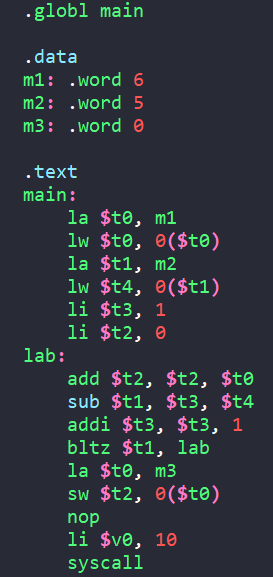
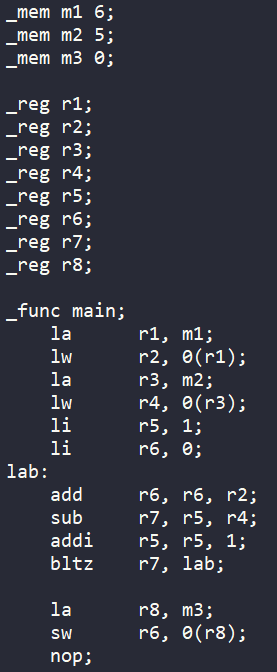
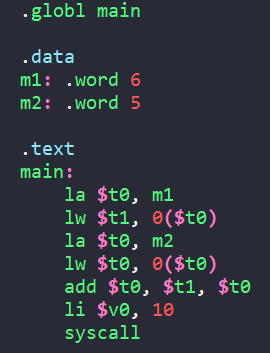
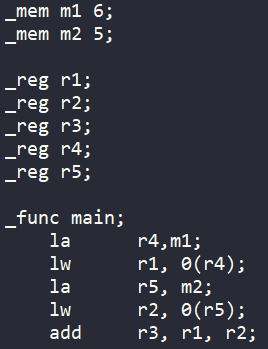
## Успјешно преведени тестни случајеви

Успјешно преведени програми дају испис на конзоли приказан на слици 7. Сваки има МИПС излазни фајл као и фајл који садржи исписе у току свих фаза превођења.



Слика 7 Успјешно превођење

Овдје су приказани улазни и излазни фајлови, тј. изворни и одредишни код успјешно преведених програма.

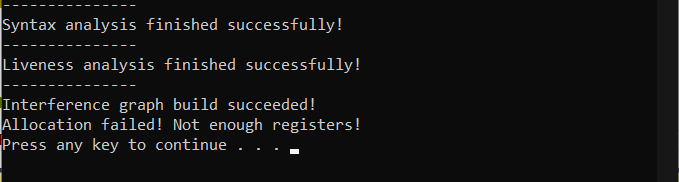


Слика 8 Кодови успјешних превода

## Неуспјешно преведени тестни случајеви

### Недовољан број регистара

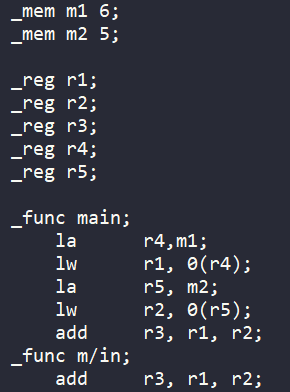
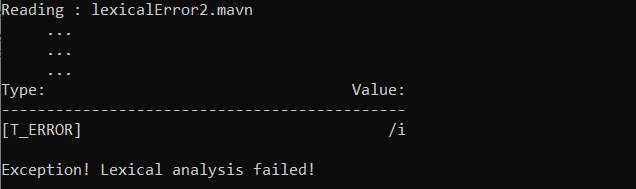
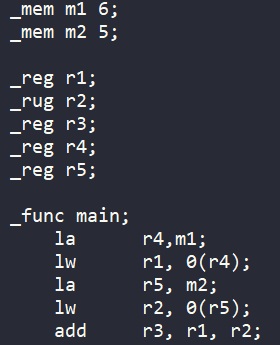
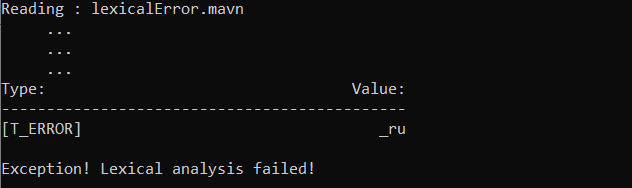
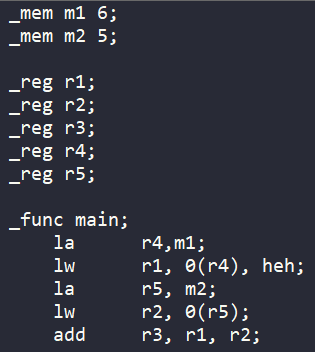
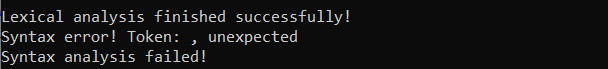
У случају да број регистара који додјељујемо програму није довољан да би свака његова регистарска промјенљива добила регистар, тада је потребно преливање у меморију, у случају овог преводиоца ће само бити исписана грешка, као на слици 9.



Слика 9 Недовољан број регистара

### Лексичке и синтаксне грешке

Лексичке грешке се откривају усљед читања токена из улазног фајла који се не поклапа са стањима који постоје у аутомату коначних стања. Синтаксне грешке се октривају у току синтаксне анализе, на примјер ако нека инстукција има вишак параметара, као на слици 10.

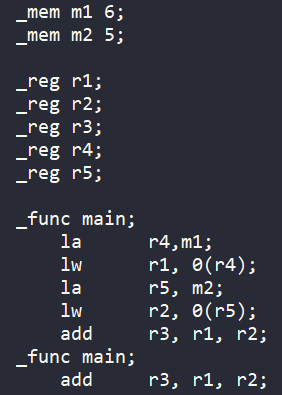
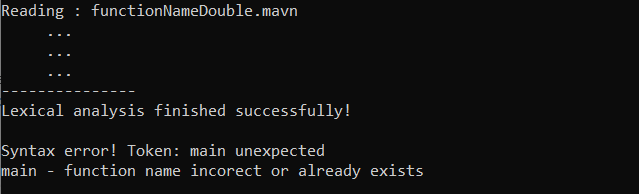
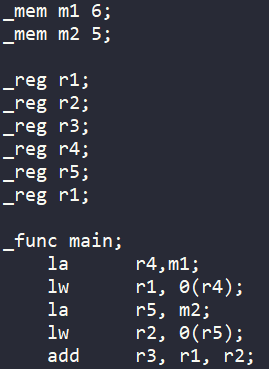
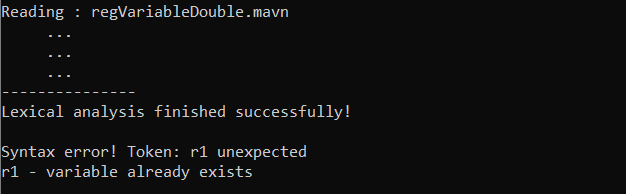
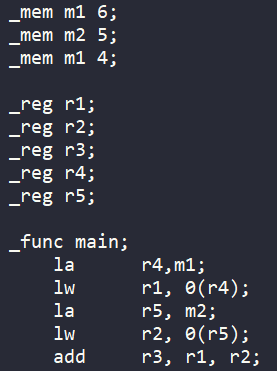
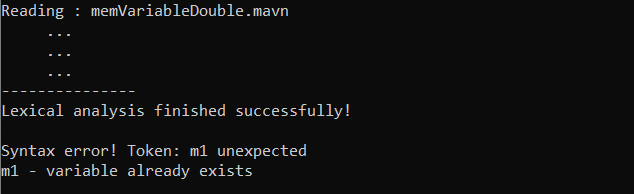
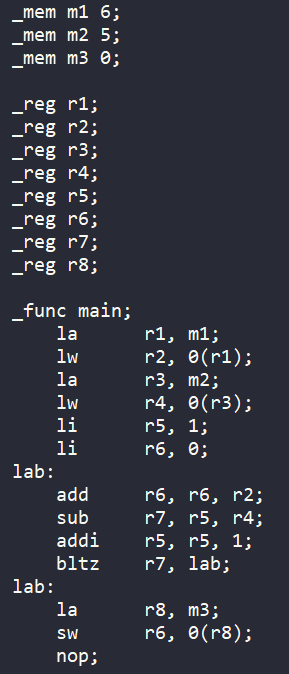
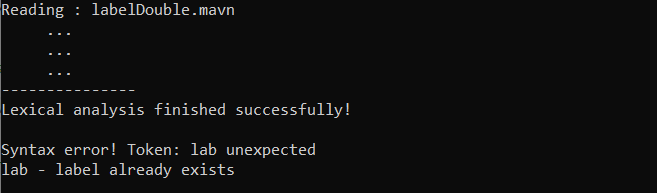


Слика 10 Лексичке и синтаксне грешке

### Семантичке грешке

#### Постојање дупликата промјенљивих, функција и лабела

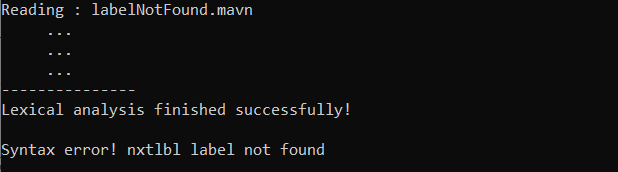
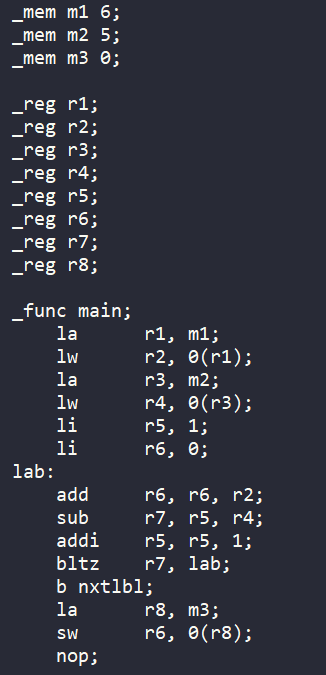
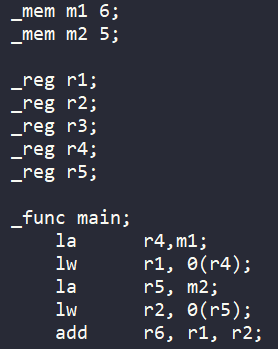
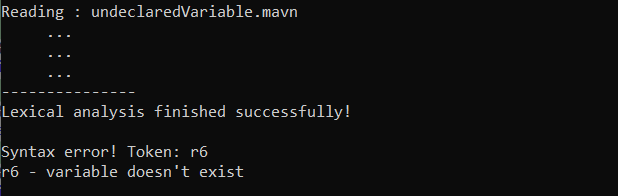
Уколико се покуша декларисати промјенљива која је већ декларисана, баца се изузетак са поруком о постојању промјенљиве. Лабеле и функције такође не смију имати дупликате, тј. не смију постојати друге функције и лабеле са истим именом.



Слика 11 Дупликати имена

#### Непостојећа лабела и промјенљива

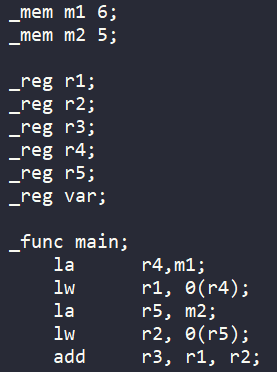
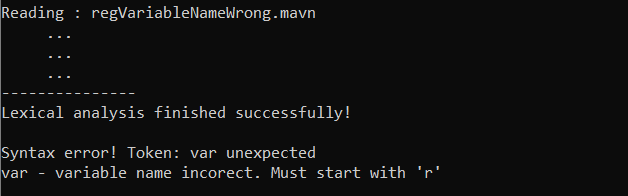
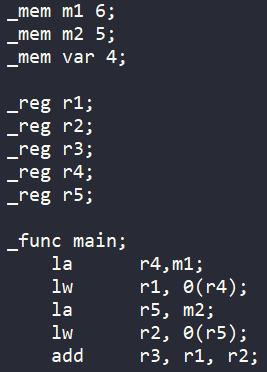
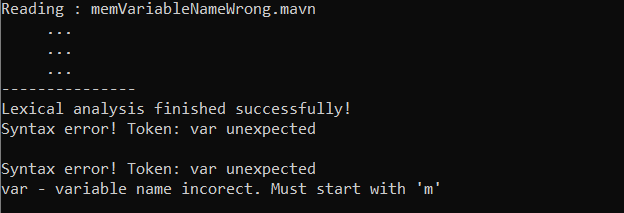
Инструкцијама гранања је потребна лабела на коју ће да се смјесте, али уколико та лабела није дефинисана нигдје, што се открива по завршетку извршавању синтаксне алализе, баца се изузетак о непостојећој лабели. Слична је ситуација и са промјенљивом, која мора да буде декларисана да би се користила у инстукцијама, уколико то није случај баца се изузетак. Примјери оваквих ситуација и изузетака су дати на слици 12.



Слика 12 Непостојећи елементи

#### Погрешно име промјенљивих

Приликом декларације, свака промјенљива мора имати исправан формат имена, тј. меморијске промјенљиве морају да почињу са *‘m’*,а регистарске са*‘r’*.



Слика 13 Погрешно име промјенљивих