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

ПРОЈЕКТНИ ЗАДАТАК

Кандидат: **Давид Вученовић**
Број индекса: **RA 126/2021**

Предмет: **Основи паралелног програмирања и софтверски алати**
Тема рада: **MAVN - ПРЕВОДИЛАЦ**

Ментор рада: **Душан Кењић**

Нови Сад, јун, 2023.

САДРЖАЈ

1. Увод

1.1 Опис проблема и задатак

Потребно је реализовати МАВН (Мипс Асемблер Високог Нивоа) преводилац који преводи код са вишег асемблерског језика на основни МИПС 32битни 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 32битној архитектури (односно симулатору).

Једноставно говорећи, главни циљ је да се из .mavn фајла добије .s фајл.

Слика 1: Сликовити прелаз из улазног .mavn до излазног .s фајла

1.1.1 МАВН

Мавн је алат који преводи програм написан на вишем МИПС 32битном асемблерском језику на основни асемблерски језик. Виши МИПС 32битни асемблер служи лакшем асемблерском програмирању јер уводи концепт **регистарске промјенљиве**. Регистарске промјенљиве омогућавају програмерима да приликом писања користе промјенљиве умјесто правих ресура.

Мавн језик подржава десет МИПС инструкција плус три које су додатно имплементиране.

Инструкција	Основна форма инструкције и њен опис
add	add rdest, rsource1, rsource2 – сабирање
addi	addi rdest, rsource1, immediate – сабирање са константом
b	b label – безусловни скок
bltz	bltz rsource1, label – скок ако је регистар мањи од нуле
la	la rdest, address – учитавање адресе у регистар
li	li rdest, immediate – учитавање константе у регистар
lw	lw rdest, address – читање једне меморијске ријечи
nop	nop – инструкција без операције
sub	sub rdest, rsource1, rsource2 - одузимање
sw	sw rdest, address – упис једне меморијске ријечи
or	or rdest, rsource1, rsource2 – логичко или
sge	sge rdest, rsource1, rsource2 – поставља један ако је први већи од другог
bne	bne rs, rt, label – скок ако вриједности регистара нису исте

Табела 1: Списак подржаних инструкција (посљедње три инструкције су додатне)

Синтакса MABH језика је описана сљедећом 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		E → la rid, mid
	S → E		E → lw rid, num(rid)
			E → li rid, num
			E → sw rid, num(rid)
			E → b id
			E → bltz rid, id
			E → nop
			E → or rid, rid, rid
			E → sge rid, rid, rid
			E → bne rid, rid, id

Табела 2: Синтакса MABH језика

Терминални симболи МАВН језика су:

: ; , ( )

_mem, _reg, _func, num id, rid, mid, eof, add, addi, sub, la, lw, li, sw, b, bltz, nop, *prva, druga, treca*

Декларација функције се врши на следећи начин:

_func funcName

funcName – мора почети словом, у наставку може бити било који низ слова и бројева

Декларација меморијске промјењиве:

_mem varName value

varName – мора почети малим словом m у наставку може бити било који број

Декларација регистарске промјењиве:

_reg varName

varName – мора почети малим словом r у наставку може бити било који број

2. Анализа проблема

Како је проблем којим се бавимо веома сложен, најбоље би било да га раздвојимо у неколико подпроблема односно цјелина. Ако се вратимо на Слику 1, видјећемо отприлике и које су то цјелине. Овим цјелинама претходи учитавање кода из улазног фајла, а након ових целина следи писање новог кода и излазни (.s) фајл. Важно је напоменути да су цјелине међусобно зависне и да **неуспјешност једне цјелине доводи до неусјпешности свих наредних** цјелина. Цјелине су:

1. **Лексичка и синтаксна анализа** – Ово је први корак након што је код учитан. Задатак лексичке анализе је да провјери исправност кода у лексичком смислу односно да провјери да ли су сви симболи који се налазе у улазном фајлу добро дефинисани. Након лексичке, слиједи синтаксна анализа која провјерава граматику, повезује ријечи и сл. Када спојимо све ријечи, ако добијемо синтаксно смислене реченице, ова цјелина је онда успјешно завршена. Како је граматика MABH-а дефинисана може се видјети у Табели 2.

2. **Анализа животног вијека** – Када се провјери смисленост самог кода, сачувају све инструкције и промјењиве које код користи, може се прећи у наредну фазу, односно у фазу анализе животног вијека. Анализа животног вијека помаже у квалитетном распоређивању ресурса рачунара. Она ће одредити колико свака променљива „живи“, гдје се користи, докле и сл. Ове информације могу бити корисне приликом додјељивања ресурса (регистара) одређеним променљивима.

3. **Граф сметњи и додјела ресурса** – Пошто имамо информације о „животу“ сваке промењиве, оне се могу искористити како би се направио граф сметњи и подјелили ресурси. Сметња у овом случају се дефинише као немогућност доделе истог ресурса (регистра) двема привременим промењивама. Граф сметњи помаже да те сметње забиљежимо и да их приликом додјеле ресурса узмемо у обзир како би смо квалитетно распоредили ресурсе свим привременим промењивима.

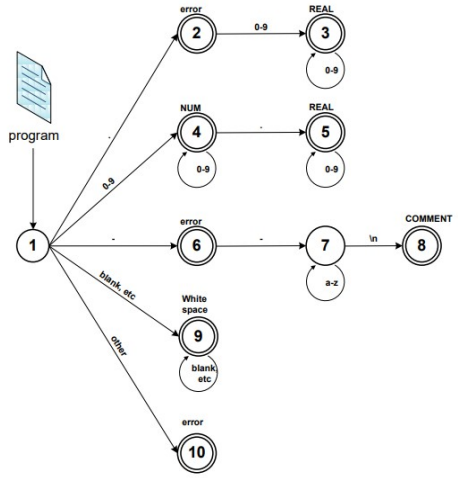
У наставку документације биће укратко објашњен концепт рјешења проблема пратећи исте ове цјелине из анализе проблема.

3. Концепт рјешења

Циљ овог дијела документације је да још детаљније разложи оне три цјелине које су поменуте у Анализи рјешења и да да ширу слику о томе како се овај проблем може рјешити. Почиње се од читања улазног фајла и лексичке анализе те се пролази кроз синтаксну анализу, анализу животног вијека све до графа сметњи и додјеле ресурса.

3.1 Читање улазног (.mavn) фајла и лексичка анализа

Прије него што крене лексичка анализа, неопходно је учитати цијели код у један бафер (нпр. вектор знакова). Када се код учита, узима се знак по знак из бафеа и врши се лексичка анализа. Лексички анализатор се реализује преко коначног детерминистичког аутомата чији је циљ да препозна све симболе у програму.



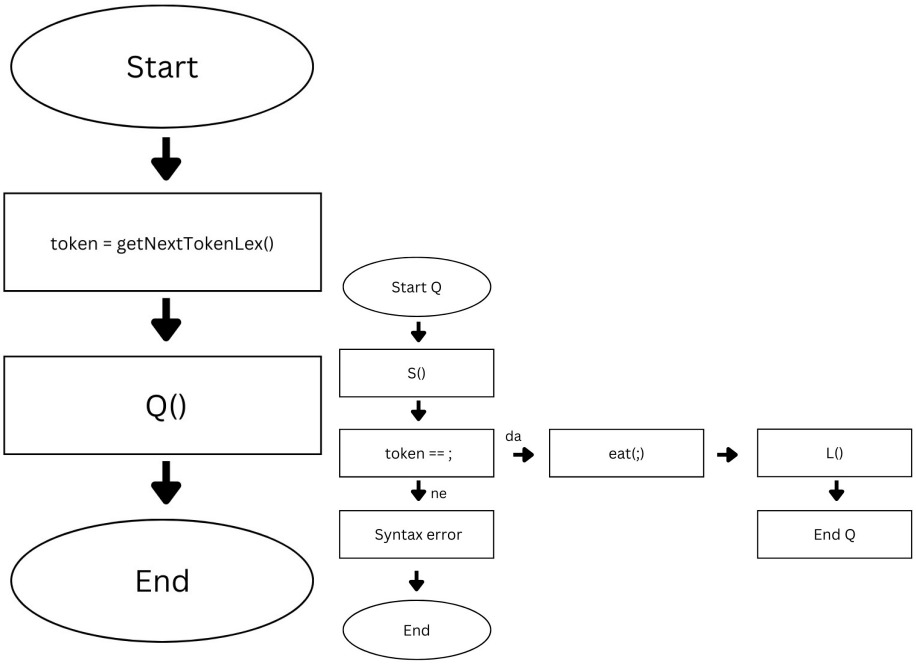
Слика 2: Примјер једног коначног детерминистичког аутомата

Почетно стање је означено бројем 1. Стања у којима се завршава претрага, називају се финална стања и на слици су представљени двоструким круговима. Критеријуми за прелазак из једног стања у друго се налазе на стрелицама које се налазе између два стања.

3.2 Синтаксна анализа

Након што се препознају сви симболи (односно ријечи), треба да се провјери да ли су те ријечи смислено поређане у неку реченицу, односно треба да се провјери да ли постоји продукција за учитане ријечи (симболе). Примјер скупа продукција је већ наведен у Табели 2 (друга страна).

Да би се сазнало да ли је нека реченица граматички исправна, одређује се њено поријекло – обавља се извођење. Извођење се обавља тако што се креће од почетног симбола, након чега се у итерацијама умјесто нетерминалних симбола смјењују њихове продукције.



Слика 3: Сликовити примјер алгоритма за синтаксну анализу (лијево је синтаксни анализатор а десно је Q функција)

Осим што провјерава исправност кода, синтаксна анализа истовремено, купећи токене прави низ променљивих које се користе у програму (у овом случају то су регистарске промењиве) и низ инструкција које се налазе у коду. Ово помаже наредним дјеловима преводиоца у анализи животног вијека, формирању графа сметњи као и у додјели ресурса.

3.3 Анализа животног вијека

Да би се омогућило да различите промјењиве које нису истовремено у употреби дијеле исти ресурс, неопходна је информација о животном вијеку промјењиве. Животни

вијек промјењиве започиње дефиницијом промјењиве, а завршава се посљедњом употребом. Промјењива је у неком сегменту „жива“ ако садржи вриједност која ће бити коришћена у току извршавања програма.

Да би се уопште бавили анализом животног вијека промјењиве, најприје треба да се за сваки чвор (инструкцију) одреде инструкције које претходе том чвору (скуп pred[n]) и инструкције које иду послије те инстукције (инструкције насљеднице – succ[n]). Даље дефинишемо скуп промјењивих def[n] које чвор n дефинише, тј. задаје им вриједност (исто што и destination), као и скуп промјењивих use[n] које чвор n користи, тј. очитава (исто што и source).

3.3.1 Алгоритам анализе животног вијека

За сваку инструкцију остало је још само да се одреди скуп out[n] и скуп in[n]. Једначине за use и in:

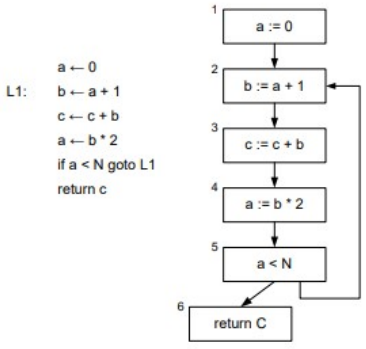
$out[n] \leftarrow U_{s \in succ[n]} in[s]$

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

Рјешење ових једначина се добија сљедећим алгоритмом, исказаном у псеудокоду:

for each n
  in [n] ← {}; out[n] ← {}
repeat
  for each n
    in’[n] ← in[n]; out’[n] ← out[n];
    out[n] ←$U_{s \in succ[n]}$ in[s]
    in[n] ← use[n] U (out[n] – def[n])
until in’[n] = in[n] and out’[n] = out[n] **for all** n

Лисе in’ и out’ су копије садржаја листа in и out на почетку сваке итерације.

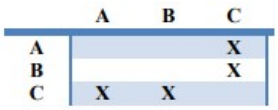


	succ	use	def	I		II		III	
				out	in	out	in	out	in
6		c			c		c		c
5	2,6	a		c	ac	ac	ac	ac	ac
4	5	b	a	ac	bc	ac	bc	ac	bc
3	4	bc	c	bc	bc	bc	bc	bc	bc
2	3	a	b	bc	ac	bc	ac	bc	ac
1	2		a	ac	c	ac	c	ac	c

Слика 4: Примјер графа тока управљања (лијево) и приказ итерација алгоритма животног вијека променљивих a, b и c из примјера

3.4 Граф сметњи

Проблем који онемогућава да се исти регистар додјели двјема привременим промјенљивим назива се сметња. Најчешћи узрок сметње је преклапање опсега животног вијека променљивих. Граф сметњи се често представља матрицом сметњи. Колоне и редови матрице су исти и представљају све промјенљиве у програму. На пресјеку одговарајуће колоне и реда биће уписан индикатор да ли постоји сметња између промјенљиве у колони и промјенљиве у реду.



Слика 5: матрица сметњи за примјер са слике 4

Да би се из резултата анализе животног вијека променљивих направио граф сметњи, користе се два правила:

1. За сваку дефиницију променљиве A, у чвору који није MOVE, додај у граф сметњи нове сметње између променљиве A и сваке променљиве Bi, која живи на излазу чвора: (A, B1) ... (A, Bn)
2. За MOVE инструкцију A=C, додај графу сметњи нове сметње између промјенљиве A и сваке промјенљиве Bi живе на излазу чвора, која није једнака C:(A, B1) ... (A, Bn); гдје је Bi различито од C.

Друго правило овде не користимо (немамо MOVE инструкцију).

3.5 Додјела ресурса

Фазе које претходе овој подразумјевају да се на циљаној платформи налази неограничен број ресурса. Ова фаза своди неограничен број ресурса на ограничен. У овом случају, ми на располагању имамо четири регистра, што значи да све регистарске променљиве треба да сведемо на четири.

Проблем додјеле ресурса се своди на проблем бојења графа. Код додјеле ресурса циљ је обојити граф са бројем боја мањим од количине слободних ресурса на циљаној платформи.

Алгоритам за додјелу ресурса се састоји од четири подфазе:

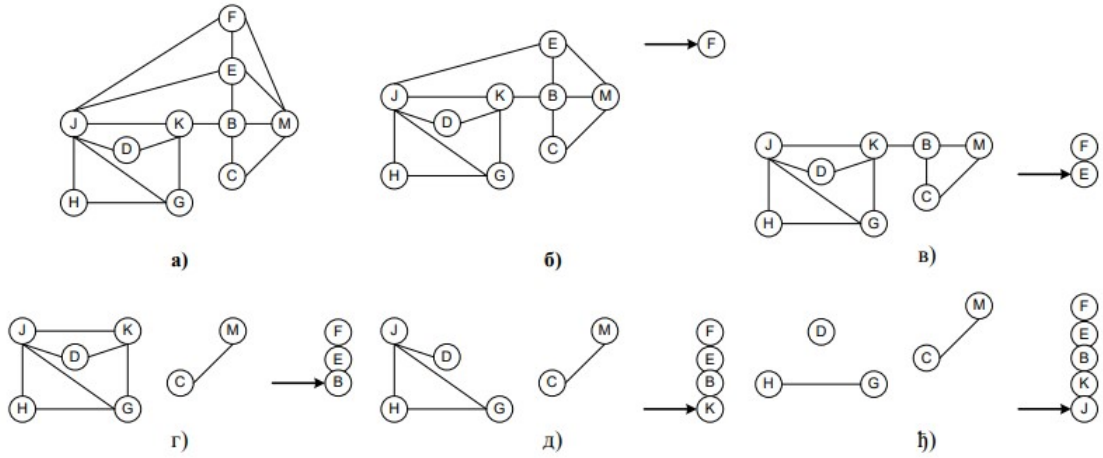
1. Фаза формирања (build)
2. Фаза урпошћавања (simplify)
3. Фаза преливања (spill)
4. Фаза избора (select)

3.5.1 Фаза формирања (build)

У овој фази се формира граф сметњи. Начин на који се прави граф сметњи је описан на прошлој страни (3.4 Граф сметњи).

3.5.2 Фаза упрошћавања (simplify)

Граф се боји примјеном сљедеће хеуристике: Претпоставимо да чвор M графа G има мање од K сусједа. Нека је G' граф који се добија уклањањем чвора M, G' = G – {M}. Очигледно је да ако је могуће обојити G' онда је могуће обојити и G, пошто кад се чвор M дода обојеном графу G', његови сусједи могу бити обојени са највише K-1 боја, тако да увијек постоји слободна боја за чвор M. Ово води ка рекурзвином алгоритму бојења графа, у ком се чворови чији је ранг мањи од K поступно уклањају из графа сметњи и гурају на стек. Свако упрошћавање доводи до снижавања ранга чвора сусједних чвору који се уклања, чиме се ствара могућност за даље урпошћавање.



Слика 5: Примјер фазе упрошћавања

3.5.3 Фаза преливања (spill)

До ове фазе долази уколико нема довољно регистара, односно ако се граф сметњи не може више упростити, а да још увијек постоје чворови у њему. Иначе, када нема више доступних регистара, требало би неке промењљиве премјестити у меморију. То се овде не ради, него се само пријављује грешка ако нема довољно регистара.

3.5.4 Фаза избора (select)

Након што је граф успјешно упрошћен потребно је обојити чворове. Редослед бојења чворова је одређен стеком на који су се чворови постављали приликом упрошћавања графа. Дакле, чворови ће бити бојени обрнутим редосљедом од оног којим су скидани са

графа. Једино правило које се користи приликом бојења је да се чвору не додјели боја која је већ додељена неком од суседних чворова.

У овом одјељку је теоретски пређен концепт рјешења. Детаљно је (колико је то могуће за кратку документацију) пређен концепт рјешења, објашњени су алгоритми који се примјењују за сваку фазу. У наставку слиједе описи модула који се користе за имплементацију овог концепта рјешења.

4. Опис рјешења

4.1 Модул Constants

У овом модулу се налазе константе које су коришћене у току писања програма.

```
const int IDLE_STATE = 0;
const int START_STATE = 1;
const int INVALID_STATE = -2;
```

горње три константе представљају стања коначног аутомата

`const int NUM_STATES = 50;` *број могућих стања*

`const int NUM_OF_CHARACTERS = 47;` *број карактера подржаних од стране аутомата са коначним бројем стања*

`const int __INTERFERENCE__ = 1;` *ознака да је дошло до сметње између двије промјењиве у графу сметњи*

`const int __EMPTY__ = 0;` *ознака да између двије промјењиве нема сметњи у графу сметњи*

`const int __REG_NUMBER__ = 4;` *број доступних регистара у процесору*

```
const int LEFT_ALIGN = 20;
const int RIGHT_ALIGN = 25;
```

посљедње двије константе служе за испис

4.2 Модул Types

У овом модулу су одређена сва стања као и прелазу између стања.

`enum TokenType` *садржи типове токена*

`enum InstructionType` *садржи типове инструкција*

`enum Regs` *употребљиви регистри*

4.3 Модул Token

Модул односно класа која описује један токен.

Поља:

`TokenType tokenType;` *тип токена*

`std::string value;` *вриједност токена*

Методе:

`TokenType getType();` *враћа тип токена*
`void setType(TokenType t);` *поставља тип токена*
`std::string getValue();` *враћа вриједност токена*

`void setValue(std::string s);` *поставља вриједност токена*
`void makeToken(int begin, int end, std::vector<char>& program, int lastFiniteState);` *чита бафер програма од позиције begin до позиције end и тај дио смјешта у value. У tokenType се смјешта тип токена који се добије помоћу lastFiniteState*
`void makeErrorToken(int pos, std::vector<char>& program);` *ова метода прави токен грешке на основу бафера и позиције бафера*
`void makeEofToken();` *прави токен за крај датотеке*
`void printTokenInfo();` *исписује вриједност и тип токена*
`void printTokenValue();` *исписује вриједност токена*
`std::string tokenTypeToString(TokenType t);` *на основу прослеђеног типа токена исписује тип токена али у виду стринга*

4.4 Модул Exceptions

Овај модул служи све врсте exception-а који могу да се „баце“ уколико дође до неког проблема. Сваки exception је дефинисан као класа насљедник класе runtime_error. Примјер једног exception-а и њеног позива дати су у наставку.

Дефиниција:

```
class LexError : public std::runtime_error
{
public:
        LexError() : runtime_error("\nException! Lexical analysis failed!\n") {};
};
```

Употреба:

`throw LexError();` *служи као пријава лексичке грешке*

4.5 Модул Writter

Овај модул садржи све функције које служе за испис информација на терминал приликом процеса превођења и уз то, уколико је превођење успјешно, садржи функцију за испис кода у .s фајл.

`void printInFile(std::string fileName, Instructions& instructions, Variables& variables);` *на основу свега урађеног, ова функција исписује код у .s фајл*
`void printInstructions(Instructions& instructions);` *исписује инструкције на терминал*

`void printVariables(Variables& variables);` *исписује промјењиве на терминал*
`void printHorizontalLine();` *исписује хоризонталну линију на терминал*
`void printInterferenceGraph(InterferenceGraph& interferenceGraph);` *исписује граф сметњи на терминал*

4.6 Модул FiniteStateMachine

Овај модул служи да дефинише аутомат са коначним бројем стања, правила прелаза и како се долази до одређених стања (токена).

Поља:

`StateMachine stateMachine;` *садржи FSM стања и матрицу прелаза у наредно стање као унутрашњу мапу (репрезентација аутомата)*
StateMachine је заправо:
```
typedef std::map<int, std::map<char,int>> StateMachine;
```
`static const TokenType stateToTokenTable[NUM_STATES];` *мапирање стања у токене*

`static const char supportedCharacters[NUM_OF_CHARACTERS];` *низ подржаних карактера*
`static const int stateMatrix[NUM_STATES][NUM_OF_CHARACTERS];` *матрица прелаза стања, редови – тренутно стање, колоне – наредно стање*

Методе:

`int getNextState(int currentState, char transitionLetter);` *враћа број следећег стања на основу параметара, први параметар је тренутно стање а други је слово прелаза*

`void initStateMachine();` *иницијализација аутомата*

`static TokenType getTokenType(int stateNumber);` *на основу броја стања преузима се токен*

4.7 Модул IR

У овом модулу се налазе класа *Variable* и класа *Instruction*. Прва класа описује једну промјењиву а друга инструкцију.

4.7.1 Класа Variable

Поља:

`enum VariableType` *енумерација типа промјењиве*

- *MEM_VAR – промјењива је меморијска локација*
- *REG_VAR – промјењива је регистар*
- *NO_TYPE – ни једно ни друго, константа*

`VariableType m_type;` *тип промјењиве*

`std::string m_name;` *назив промјењиве*
`int m_position;` *позиција промјењиве*
`int m_value;` *вриједност промјењиве*

`Regs m_assignment;` *регистар који јој је додјељен*

У овом модулу се налазе конструктори који се позивају у зависности од типа промењљиве. Свако поље поседује методе за добављање вредности и промену вредности.

Додата метода: `void isVariableCorrect();` *провјерава да ли је промјењљива добро именована (нпр. регистарска променљива мора почети са r док меморијска променљива мора почети са m, ако није, враћа Exeception*

4.7.2 Класа Instruction

Поља:

`int m_position;` *позиција инструкције*

`InstructionType m_type;` *тип инструкције*
`Label* m_label;` *ово је поље за инструкције које користе лабелу*
`Variables m_dst;` *Destination регистри инструкције*
`Variables m_src;` *Source регистри инструкције*
`Variables m_use;` *промјењљиве које инструкција користи*
`Variables m_def;` *промјењљиве које инструкција дефинише*
`Variables m_in;` *промјењљиве које улазе у инструкцију*
`Variables m_out;` *промјењљиве које излазе из инструкције*
`std::list<Instruction*> m_succ;` *инструкције које слиједе*
`std::list<Instruction*> m_pred;` *инструкције које претходе*

Методе:
Свако поље има getter и setter и за сваку листу постоји метода add која додаје објекат у листу. Такође постоји предефинисан оператор исписа за инструкцију

4.8 Модул Label

Овај модул (класа) описује једну лабелу.

Поља:

`enum labelType` *енумерација за тип лабеле, тип лабеле може бити ID или FUNCTION_ID*
`labelType m_type;` *тип лабеле*
`std::string m_name;` *назив лабеле*

Методе:

`void checkLabel();` *метода која провјерава исправност лабеле. Лабела је исправна ако почиње словом, у супротном Exception*
`std::string getName() const;` *враћа име лабеле*

4.9 Модул LexicalAnalysis

Овај модул је задужен за лексичку анализу кода.

Поља:

`std::ifstream inputFile;` *назив улазног фајла*

`std::vector<char> programBuffer;` *бафер програма*
`unsigned int programBufferPosition;` *тренутна позиција бафера програма*
`FiniteStateMachine fsm;` *машина коначних стања*
`TokenList tokenList;` *листа токена*
`Token errorToken;`

Методе:

`void initialize();` *иницијализација*
`bool Do();` *ова метода покреће лексичку анализу*
`bool readInputFile(std::string fileName);` *учитава улазни фајл*
`Token getNextTokenLex();` *добавља наредни токен*
`TokenList& getTokenList();` *враћа листу токена*
`void printTokens();` *исписује токене на екран*
`void printLexError();` *исписује лексичку грешку уколико је пронађена*

`void printMessageHeader();` *исписује заглавље за табелу токена и вриједности*

4.10 Модул SyntaxAnalysis

Овај модул извршава синтаксну анализу кода.

Поља:

`LexicalAnalysis& m_lexicalAnalysis;` *референца лексичке анализе*
`TokenList::iterator m_tokenIterator;` *итератор на листу токена коју смо добили од лексичке анализе*
`Token m_currentToken;` *тренутни токен за синтаксну анализу*
`Instructions* m_instructions;` *листа свих инструкција*
`Variables* m_memVars;` *листа меморијских промјенљивих*
`Variables* m_regVars;` *листа регистарских промјенљивих*
`Labels m_labels;` *листа лабела*

`bool m_syntaxError;` *индикатор синтаксне грешке*

`int m_instructionCounter;` *бројач инструкција*
`int m_variableCounter;` *бројач промјенљивих*

Методе:

У модулу су дефинисане методе за нетерминалне симболе Q, S, L и E.

`bool doSyntaxAnalysis();` *метода која започиње процес синтаксне анализе*
`void printSyntaxError(Token& token);` *метода за испис токена који је довео до синтаксне грешке*
`void eat(TokenType t);` *„једе“ токен који му је прослеђен, баца exception уколико је прослеђен токен који се не очекује*
`Token getNextToken();` *враћа следећи токен*
`bool variableExists(Variable* var);` *провјерава да ли прослеђена промјенљива већ постоји у програму, ако постоји, баца exception*
`void addVariable(Variable* var);` *додаје промјенљиву*
`bool labelExists(Label* label);` *провјерава да ли прослеђена лабела већ постоји у листи лабела*
`void addLabel(Label* label);` *додаје лабелу у листу лабела*
`Variable* getVariable(std::string name);` *тражи промјенљиву са прослеђеним именом, уколико је не може наћи - exception*
`Label* getLabel(std::string name);` *тражи лабелу са прослеђеним именом*

4.11 Модул LivenessAnalysis

Овај модул је задужен за анализу животног вијека промјењивих.

Методе:

Садржи четири фил методе (fillPred, fillSucc, fillUse, fillDef) које се покрећу прије саме анализе животног вијека како би се попуниле листе pred, succ, use и fill за сваку инструкцију.

`Instruction* getLabelInstruction(Instructions* instructions, Label* label);` *добавља инструкцију која садржи само лабелу*

`bool variableExists(Variables& variables, Variable& variable);` *провјерава да ли промјењива variable постоји у листи variables*

`void livenessAnalysis(Instructions* instructions);` *покреће анализу животног вијека*

4.12 Модул InterferenceGraph

Модул InterferenceGraph прави граф сметњи.

Поља:

`Instructions* m_instructions;` *показивач на листу инструкција*
`Variables* m_variables;` *показивач на листу промјењивих*
`int** m_values;` *матрица која представља граф сметњи*
`int m_size;` *димензија матрице*

Методе:

`void doInterferenceGraph();` *метода која прави граф сметњи, динамички алоцира колико меморије треба за граф сметњи и попуњава је вриједностима __EMPTY__ или __INTERFERENCE__*

`void freeInterferenceGraph();` *метода која ослобађа динамички алоцирану меморију за граф сметњи*

4.13 Модул Simplification

Задатак овог модула је да направи стек промјењивих које ћемо користити за додјелу ресурса.

Методе:

`Variable* findVariableByPosition(Variables* variables, int position);` *метода која враћа промјењиву на основу позиције*

`bool isSimplificationPossible(std::map<Variable*, int> nodes, int numberOfRegisters);` *гледа да ли је уопште могуће упростити граф даље*

`std::map<Variable*, int> makeNewNodeMap(InterferenceGraph* ig,std::vector<std::vector<int>>& graph);` *враћа мапу чији је пар вриједности промјењива и ранг промјењиве (чвора) у графу*

`std::map<Variable*, int>::iterator findVariable(std::map<Variable*, int>& nodes);` *ова функција враћа итератор на промјењиву из мапе која треба следећа да се стави на стек*

`bool isSimplificationOver(std::vector<std::vector<int>>& graph, int size);` *функција која провјерава да ли је упрошћавање готово*

```
void removeNode(std::vector<std::vector<int>>& graph, int size, int position);
```
брише чвор са графа
```
std::stack<Variable*>*doSimplification(InterferenceGraph* ig,int numberOfRegisters);
```
метода која покреће упрошћавање

4.14 Модул ResourceAllocation

Модул који обавља додјелу ресурса.

```
bool doResourceAllocation(std::stack<Variable*>* simplificationStack,
InterferenceGraph* ig);
```
метода која обавља додјелу ресурса, враћа true уколико је успјешно завршена а false у супротном

4.15 main.cpp

Функција main редом прави објекте сваког модула који има класу и редом позива модуле како би заједно превели код из .mavn у .s фајл.

5. Верификација и тестирање

Програм је тестиран на основу датих примера у фолдеру examples (simple.mavn и multiply.mavn). Сваки појединачни модул има своју провјеру да ли је посао добро одрађен, у супротном прекида се извршавање и пријављује се грешка и исписује одговарајућа порука. Програм је отпоран на све врсте лексичких и синтаксних грешака, као и грешака семантичке природе (недефинисани коришћени подаци, нпр променљиве, лабеле и слично). Решење је приказано у датотеци simple.s, тј. multiply.s.

6. Закључак

Реализован је пример МАВН преводиоца који преводи програме са вишег асемблерског језика на основни МИПС 32битни асемблерски језик. Преводилац детектује лексичке и синтаксне грешке и генерише одговарајуће извештаје о грешкама. Излаз из преводиоца садржи коректан асемблерски код који је могуће извршавати на МИПС 32битној архитектури (симулатору). Све ово је верификовано на тестним случајевима и задатим провјерама.