



# 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

## ИСПИТНИ РАД

Кандидат: Никола Совиљ

Број индекса: SW75/2019

Предмет: Системска програмска подршка I

Тема рада: МАВН Преводаилац

Ментор рада: др Миодраг Ђукић

Нови Сад, јун, 2021.

## Sadržaj

1. Uvod	3
1.1 MAVN Prevodilac	3
1.2 Instrukcije	5
2. Analiza problema	6
2.1 Ulazni fajl	6
2.2 Leksička analiza	6
2.3 Sintaksna analiza	6
2.4 Kreiranje instrukcija i promenljivih	6
2.5 Analiza životnog veka promenljivih	7
3. Koncept rešenja	8
3.1 Leksička analiza	8
3.2 Sintaksna analiza	9
3.3 Kreiranje instrukcija i promenljivih	10
3.4 Analiza životnog veka promenljivih	11
4. Programsko rešenje	12
4.1 Leksička analiza	12
4.2 Sintaksna analiza	13
4.3 Kreiranje instrukcija i promenljivih	14
4.4 Analiza životnog veka promenljivih	15
5. Verifikacija	16
5.1 Leksička analiza	16
5.2 Sintaksna analiza	17

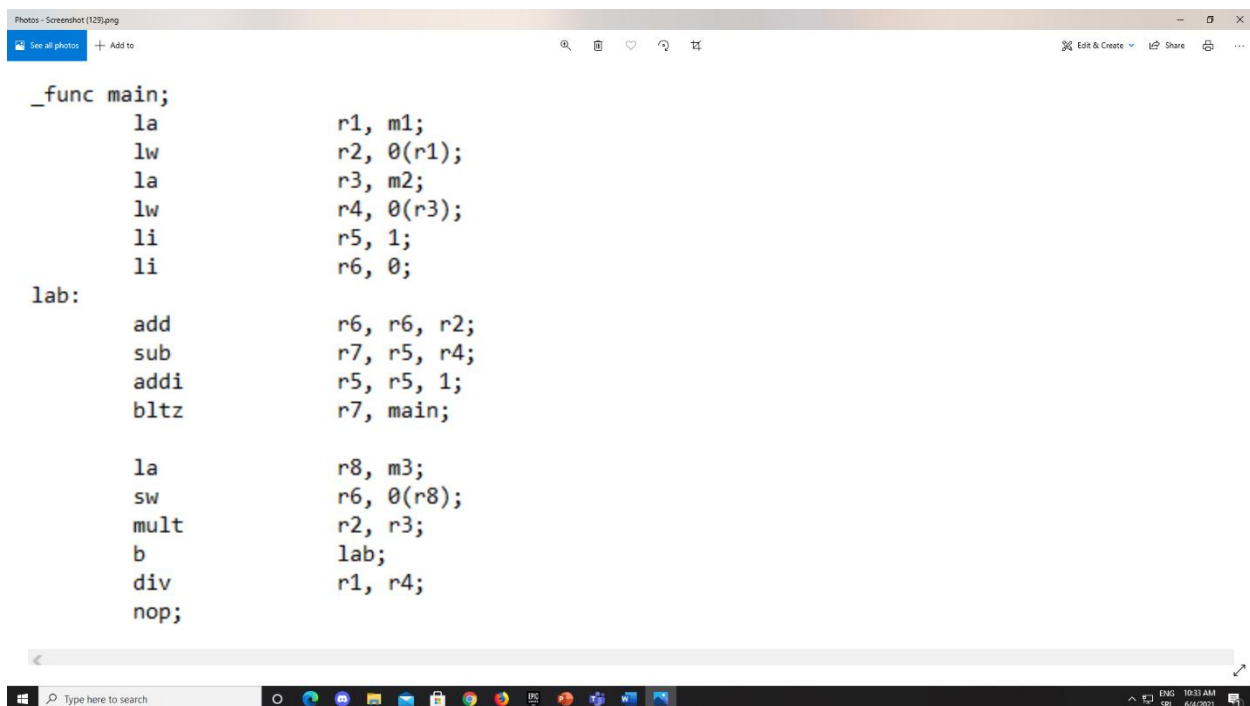
## Spisak slika

Slika 1 Primer MIPS 32bit asemblerskog koda	3
Slika 2 Primer osnovnog asemblerskog koda	2
Slika 3 Automat stanja kod leksičke analize	8
Slika 4 Primer dobrog izvršavanja leksičke analize	16
Slika 5 Primer dobrog izvršavanja sintaksne analize	17

# 1. Uvod

## 1.1 MAVN Prevodilac

MAVN (Mips assembler visokog nivoa) je alat koji prevodi program napisan na višem MIPS 32bit asemblerskom jeziku na osnovni asemblerski jezik. Viši MIPS 32bit asemblerski jezik služi lakšem asemblerskom programiranju jer uvodi concept registarske promenljive. Registarske promenljive omogućavaju programerima lakši rad, jer umesto pravih resurasa (konstanti), mogu da koriste promenljive. Ovo je značajno, jer programer prilikom rada ne mora da vodi računa o zauzeću registara i njihovom sadržaju.



**Slika 1** Primer MIPS 32bit asemblerskog koda

```

section      .text
global      _start                ;must be declared for linker (ld)

_start:                                           ;tell linker entry point

    mov     edx,len                ;message length
    mov     ecx,msg                ;message to write
    mov     ebx,1                  ;file descriptor (stdout)
    mov     eax,4                  ;system call number (sys_write)
    int     0x80                  ;call kernel

    mov     eax,1                  ;system call number (sys_exit)
    int     0x80                  ;call kernel

section      .data

msg          db  'Hello, world!',0xa            ;our dear string
len          equ $ - msg                       ;length of our dear string

```

**Slika 2** Primer osnovnog asemblerskog koda

## 1.2 Instrukcije

Sledi lista instrukcija koje su podržane u konkretnom projektu:

**add** – (addition) sabiranje

**addi** – (addition immediate) sabiranje sa konstantom

**b** – (unconditional branch) безусловni skok

**bltz** – (branch on less than zero) skok ako je registar manji od nule

**la** – (load address) učitavanje adrese u registar

**li** – (load immediate) učitavanje konstante u registar

**lw** – (load word) učitavanje jedne memorijske reči

**nop** – (no operation) instrukcija bez operacije

**sub** – (subtraction) oduzimanje

**sw** – (store word) upis jedne memorijske reči

**mult** – (multiplication) – množenje

**div** – (division) – deljenje

**lui** – (load upper immediate) – učitavanje donje polovine konstante u gornju polovinu registra

**jr** (jump register) безусловni skok do instrukcije čija adresa se nalazi u datom registru

## 2. Analiza problema

### 2.1 Ulazni fajl

U ulaznom fajlu se nalazi kod na MIPS 32bit asemblerskom jeziku koga kasnije treba prevesti u osnovni asemblerski jezik. Kada ulazni kod uđe u program, prvi problem koji se javlja je kreiranje liste tokena čiji elementi imaju odgovarajuće nazive instrukcija, promenljivih, konstanti itd. Ovaj problem će razrešiti **leksički analizator**.

### 2.2 Leksička analiza

Kreira listu tokena iz ulaznog fajla. Kada dobijemo odgovarajuću listu tokena, naš program postaje "reaktivan" na greške leksičkog tipa. Da bi postao reaktivan i na greške sintaksnog tipa, mora da prođe **sintaksnu analizu**.

### 2.3 Sintaksna analiza

Sintaksna analiza proverava da li je tekuća lista tokena u skladu sa gramatikom jezika. Gramatika jezika se opisuje pomoću skupa produkcija između terminalnih i neterminalnih simbola. Kada tokeni uspešno pređu i sintaksnu analizu, spremni su za transformaciju u **instrukcije** ili **promenljive**, u zavisnosti od njihovog sadržaja.

### 2.4 Kreiranje instrukcija i promenljivih

Da bi se znalo da li treba napraviti instrukciju ili promenljivu na osnovu zadatog tokena, mora se unapred znati kog je token tipa. Za to je zadužena klasa **TokenType** o kojoj je biti više reči u nastavku, kao i posebnih funkcija koje su zadužene za samo kreiranje instrukcija i promenljivih.

## **2.5 Analiza životnog veka promenljivih**

Da bi se omogućilo da različite promenljive koje nisu istovremeno u upotrebi dele isti resurs neophodna je informacija o životnom veku promenljive. Tu informaciju generiše analiza životnog veka promenljivih i pokazuje nam koje promenljive mogu biti u potencijalnoj smetnji prilikom dodele resursa.

## 3. Koncept rešenja

### 3.1 Leksička analiza

Za leksičku analizu su podržani karakteri:

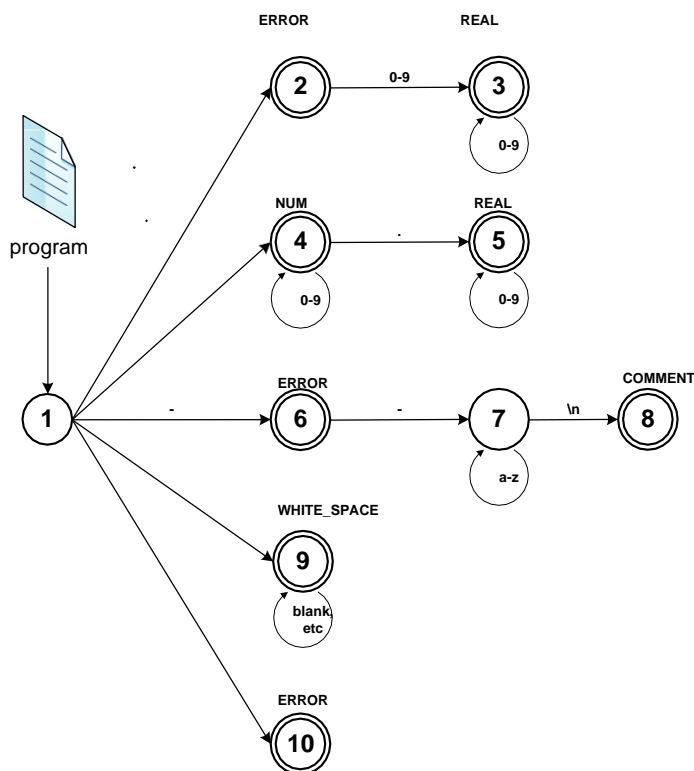
\*cifre (0 – 9)

\*operatori (., -)

\*slova (a – z)

\*specijalni karakteri (razmak, nov red, ...)

Svaki od ovih karaktera predstavljaju stanje sledećeg automata. Kad god se pojavi novi karakter prikazan na linijama automata, analizator menja stanje.



**Slika 3** Automat stanja kod leksičke analize

\*Tipovi tokena: NUM, REAL, COMMENT, WHITE SPACE, ERROR



## 3.2 Sintaksna analiza

Sintaksni analizator prima listu tokena koje su leksički korektne i proverava da li su dati tokeni u skladu sa gramatikom jezika. Dakle, neophodno je prvo kreirati gramatiku jezika koja proverava da li je data lista tokena u korektnom redosedu. Na primer, lista tokena (add , .) će proći leksičku analizu, ali neće proći sintaksnu ukoliko joj to gramatika jezika ne dozvoli. Gramatika jezika se sastoji od skupa produkcija. U produkciji se mogu naći terminalni (izvršivi) i neterminalni simboli. Primer jedne produkcije:

**$E \rightarrow b \text{ id}$**

U ovom slučaju, neterminalan simbol je E, koji mora da se prevede u neki skup terminalnih simbola kako bi se izvršio. b(skok) i id(promenljiva) predstavljaju terminalne simbole. Gramatika jezika u konkretnom projektu (koja pokriva sve navedene instrukcije gore):

<b><math>Q \rightarrow S ; L</math></b>	<b><math>S \rightarrow \_ \text{mem mid num}</math></b>	<b><math>L \rightarrow \text{eof}</math></b>	<b><math>E \rightarrow \text{add rid, rid, rid}</math></b>
	<b><math>S \rightarrow \_ \text{reg rid}</math></b>	<b><math>L \rightarrow Q</math></b>	<b><math>E \rightarrow \text{addi rid, rid, num}</math></b>
	<b><math>S \rightarrow \_ \text{func id}</math></b>		<b><math>E \rightarrow \text{sub rid, rid, rid}</math></b>
	<b><math>S \rightarrow \text{id: E}</math></b>		<b><math>E \rightarrow \text{la rid, mid}</math></b>
	<b><math>S \rightarrow E</math></b>		<b><math>E \rightarrow \text{lw rid, num(rid)}</math></b>
			<b><math>E \rightarrow \text{li rid, num}</math></b>
			<b><math>E \rightarrow \text{sw rid, num(rid)}</math></b>
			<b><math>E \rightarrow b \text{ id}</math></b>
			<b><math>E \rightarrow \text{bltz rid, id}</math></b>
			<b><math>E \rightarrow \text{mult rid, rid}</math></b>
			<b><math>E \rightarrow \text{div rid, rid}</math></b>
			<b><math>E \rightarrow \text{lui rid, num}</math></b>
			<b><math>E \rightarrow \text{jr rid}</math></b>
			<b><math>E \rightarrow \text{nop}</math></b>

### 3.3 Kreiranje instrukcija i promenljivih

Da bi instrukcije bile uspešno kreiranje, moraju se prvobitno kreirati promenljive koje funkcija koristi. Nakon kreiranja promenljivih, sledi popunjavanje lista **use** i **def** koje se nalaze unutar klase **Instruction**, sa promenljivama koje ulaze u sastav instrukcija. Use lista predstavlja promenljive koje određena instrukcija koristi, dok def lista predstavlja listu promenljivih koje određena instrukcija definiše. Uzmimo sledeći primer:

**add r7, r4, r5** – promenljive koje ulaze u use listu su: r4 i r5, a u def listu: r7

Nakon kreiranja use i def lista, zbog kasnijih koraka moramo kreirati listu prethodnika i sledbenika instrukcija. Lista prethodnika je lista u okviru neke instrukcije **instr**, koja predstavlja skup mogućih prethodnih instrukcija koje se izvršavaju pre date instrukcije. Lista sledbenika, analogno, predstavlja skup mogućih sledećih instrukcija koje se izvršavaju nakon izvršavanja instrukcije **instr**. Uzmimo sledeći primer:

**lab:**

**add r7, r4, r5**

**bltz r7, lab**

**mult r2, r3**

Neka instr bude bltz. U njenu listu prethodnika ulazi instrukcija add. S obzirom da je bltz uslovni skok, postoji mogućnost da njen sledbenik bude mult, ali postoji isto tako mogućnost da njen sledbenik bude i instrukcija add, jer se ona nalazi odmah

posle labele lab, koja ulazi u sastav instrukcije bltz. Dakle, obe instrukcije ulaze u listu sledbenika.

### 3.4 Analiza životnog veka promenljivih

Da bi odredili životni vek promenljivih, moramo već unapred znati liste **use**, **def**, **pred** i **succ**. Ove liste su nam važne kako bi mogli da kreiramo liste **in** i **out**, koje predstavljaju suštinu ove analize. In lista predstavlja listu promenljivih određene instrukcije koje su žive pri ulasu u određenu instrukciju, odnosno one koje su žive (definisane) neposredno pre njenog izvršavanja. Out lista predstavlja listu promenljivih određene instrukcije koje su žive na izlasku iz same instrukcije, odnosno, neposredno nakon izvršavanja iste. Životni vek promenljive počinje njenom definicijom, a završava se poslednjom upotrebom u okviru datog programa/modula. Promenljiva je u nekom segmentu živa sve dok sadrži vrednost koja će biti korišćena kasnije u toku izvršavanja programa. U listu  $out[instr]$  ulaze svi skupovi naslednika instrukcije  $instr$ , kao i elementi liste  $in[instr]$ . U listu  $in[instr]$  ulaze sve promenljive koje se nalaze u  $use[instr]$  listi i sve promenljive koje se nalaze u  $out[instr]$  listi ali bez onih koje su bile definisane u okviru same instrukcije. To pokazuje i sledeći pseudokod:

$$out[n] \leftarrow \bigcup in[s] \mid s \in succ[n]$$
$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

Nakon formiranja in i out listi, imamo direktan uvid u životni vek jedne promenljive. To nam omogućava da uočimo koje promenljive mogu biti u potencijalnoj smetnji prilikom dodele resursa u kasnijoj fazi prevođenja, a koje mogu nesmetano dobijati iste resurse samo zbog činjenice da im se životni vekovi jednostavno ne poklapaju, pa nema rizika da dođe do bilo kakve smetnje. Ovo je bitno jer težimo ka tome da, uz što manje promenljivih, odnosno registara, možemo da skladištimo što više resurasa, kako bi sam program bio efikasniji.

## 4. Programsko rešenje

**int** `main()` – glavna funkcija u programu. Uvezuje sve module programa u skladnu celinu.

**bool** `LexicalAnalysis::readInputFile(string fileName)` – učitava ulaznu datoteku napisanu na MIPS 32bit asemblerskom jeziku visokog nivoa. Vraća true ukoliko je proces uspešan

### 4.1 Leksička analiza

**void** `FiniteStateMachine::initStateMachine()` – inicijalizuje matricu stanja. Stanja predstavljaju različite tipove tokena prilikom leksičke analize

**bool** `LexicalAnalysis::Do()` – vraća true ukoliko je leksički analizator stigao do kraja fajla. Vraća false je tip token T\_ERROR što predstavlja grešku prilikom kreiranja liste tokena

**Token** `LexicalAnalysis::getNextTokenLex()` – prelazi na sledeće stanje u matrici stanja i vraća token

**TokenList&** `LexicalAnalysis::getTokenList()` – vraća listu postojećih tokena

**void** `LexicalAnalysis::printTokens()` – ispis tokena iz postojeće liste tokena.

**Int** `FiniteStateMachine::stateMatrix[NUM_STATES][NUM_OF_CHARACTERS]` – dvodimenzionalni niz koji predstavlja matricu stanja

## 4.2 Sintaksna analiza

**bool SyntaxAnalysis::Do()** – vraća true ako se sintaksna analiza završi uspešno, vraća false ako lista tokena nije u skladu sa gramatikom i njenim produkcijama

**void SyntaxAnalysis::eat(TokenType t)** – izvršava se kada naiđe na analizu terminalnih simbola

\* **TokenType t** – predstavlja tip terminalnog tokena (T\_ID, T\_ADD, T\_LUI, ...)

**Token SyntaxAnalysis::getNextToken()** – dobavlja sledeći token iz liste tokena za sintaksnu analizu

Primeri produkcija unutar gramatike jezika:

```
void SyntaxAnalysis::l()
{
    if (currentToken.getType() == T_END_OF_FILE)
    {
        eat(T_END_OF_FILE);
    }
    else
    {
        q();
    }
}
```

```
void SyntaxAnalysis::q()
{
    s();
    eat(T_SEMI_COL);
    l();
}
```

## 4.3 Kreiranje instrukcija i promenljivih

**void** createVariables(**LexicalAnalysis**& lex) – kreiranje promenljivih

\* **LexicalAnalysis**& lex – instanca leksičkog analizatora preko kog se dobavlja lista tokena za kreiranje promenljivih

**void** createUseDef(**LexicalAnalysis**& lex) – kreiranje lista use i def, pomoću ugrađenog iteratora se dobavlja vrednost svih promenljivih koje se nalaze u sasavu instrukcije

\* **LexicalAnalysis**& lex – instanca leksičkog analizatora preko kog se dobavlja lista tokena za kreiranje promenljivih

**void** createSuccPred(**LexicalAnalysis**& lex) – kreiranje lista prethodnika i sledbenika

\* **LexicalAnalysis**& lex – instanca leksičkog analizatora preko kog se dobavlja lista tokena za kreiranje promenljivih

**static Instruction\*** makeInstruction(**unsigned int** pPos, **InstructionType** tType, **Variable\*** dDst, **Variable\*** sSrc1, **Variable\*** sSrc2) – kreira instrukcije na osnovu sledećih podataka:

\* **unsigned int** pPos – pozicija instrukcije

\* **InstructionType** tType – tip instrukcije

\* **Variable\*** dDst – definisana promenljiva

\* **Variable\*** sSrc1 – promenljiva1 koja se koristi

\* **Variable\*** sSrc2 – promenljiva2 koja se koristi

**typedef map<Instruction\*, Variable\*>::iterator it3** – mapa koja mapira labele na prve instrukcije koje se izvršavaju posle njene deklaracije (koristi se zbog skokova)

## 4.4 Analiza životnog veka promenljivih

**void** `livenessAnalysis(Instructions instructions)` – izvršavanje analize životnog veka promenljivih

\* **Instructions** `instructions` – lista instrukcija na osnovu koje se izvršava analiza

Najpre treba isprazniti in i out liste od date instr instrukcije, zatim napraviti kopije in i out lista sve dok one ne budu iste sa originalom. Dok se to ne ostvari neophodno je izvršiti sledeći algoritam:

$$out[n] \leftarrow \bigcup in[s] \mid s \in succ[n]$$
$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

**bool** `variableExists(Variable* variable, Variables variables)` – proverava da li se data promenljiva nalazi u datoj listi promenljivih

\* **Variable\*** `variable` – data promenljiva

\* **Variables** `variables` – data lista promenljivih

\*vraća true ukoliko se promenljiva nalazi unutar liste

\*vraća false ukoliko se ne nalazi

Ova funkcija se koristi prilikom unosa promenljivih u in listu, da bi se proverilo da li se neka promenljiva nalazi u def listi date instrukcije. Tamo se ne smeju sadržati promenljive koje su u okviru date instrukcije definisane

## 5. Verifikacija

### 5.1 Leksička analiza

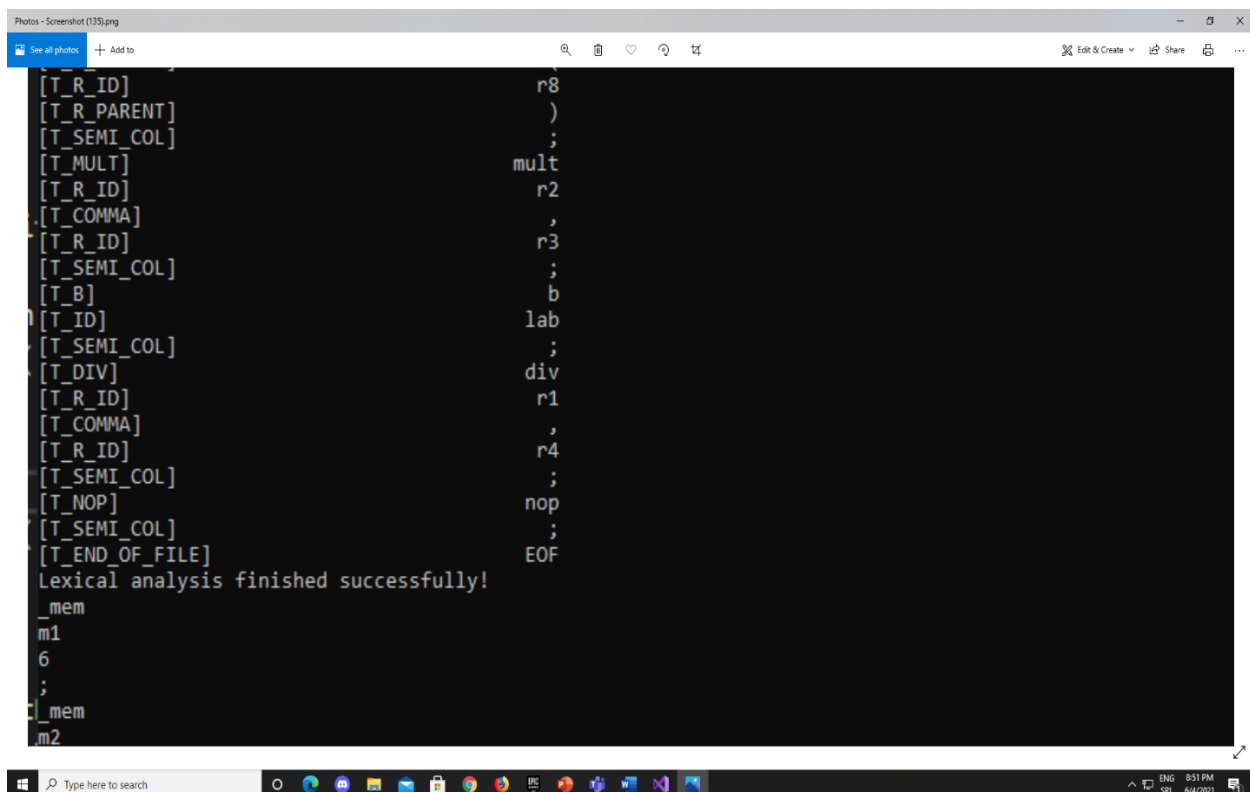
`throw runtime_error("\nException: Infinite state detected! There is something very wrong with the code !\n")` – greška prilikom korišćenja matrice stanja

`void LexicalAnalysis::printLexError()`

`throw runtime_error("\nException! Lexical analysis failed!\n");` – Greška na nivou leksičke analize

`void LexicalAnalysis::printTokens()`

`cout << "Token list is empty!" << endl;` - može se desiti da je lista prazna



```
[T_R_ID] r8
[T_R_PARENT] )
[T_SEMI_COL] ;
[T_MULT] mult
[T_R_ID] r2
[T_COMMA] ,
[T_R_ID] r3
[T_SEMI_COL] ;
[T_B] b
[T_ID] lab
[T_SEMI_COL] ;
[T_DIV] div
[T_R_ID] r1
[T_COMMA] ,
[T_R_ID] r4
[T_SEMI_COL] ;
[T_NOP] nop
[T_SEMI_COL] ;
[T_END_OF_FILE] EOF
Lexical analysis finished successfully!
_mem
m1
6
;
_mem
m2
```

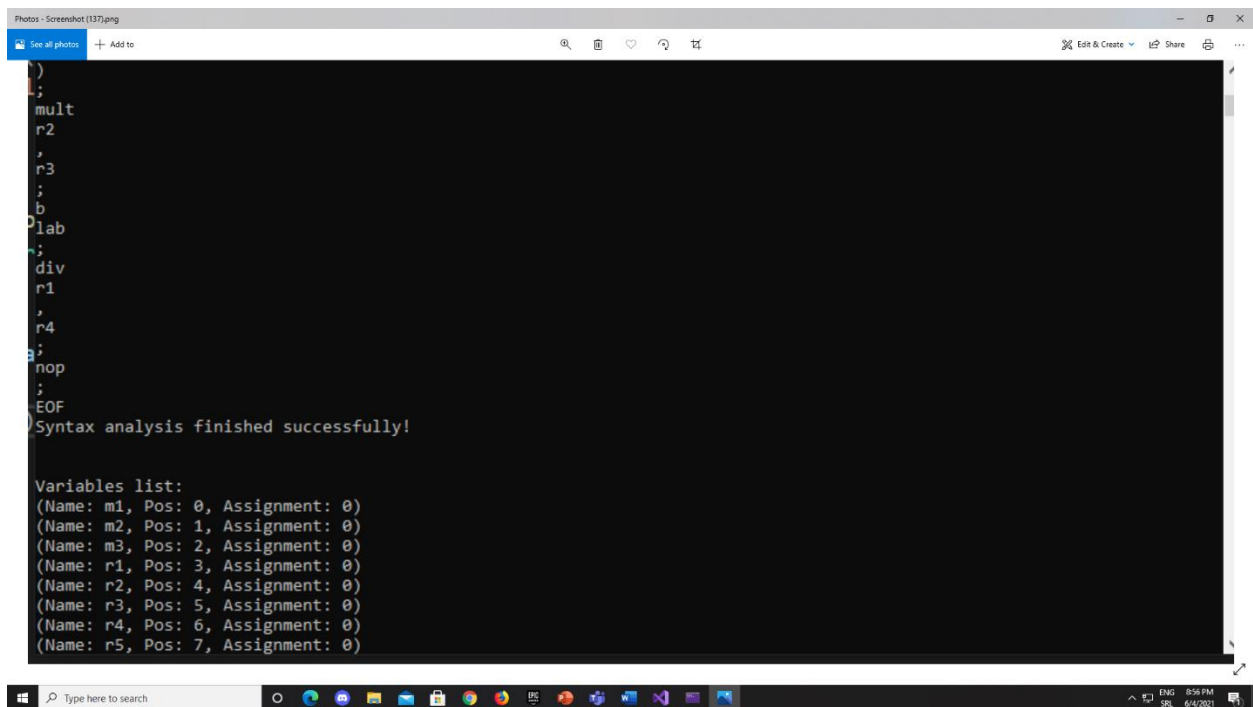
**Slika 4** Primer dobrog izvršavanja leksičke analize



## 5.2 Sintaksna analiza

```
void SyntaxAnalysis::printSyntaxError(Token& token)
{
    cout << "Syntax error! Token: " << token.getValue() << "
unexpected" << endl;
}

throw runtime_error("\nException! Syntax analysis failed!\n");
```



The screenshot shows a Windows Photos application window titled "Photos - Screenshot (137).png". The main content is a dark-themed terminal window. The terminal output is as follows:

```
)
;
mult
r2
,
r3
;
b
lab
;
div
r1
,
r4
;
nop
;
EOF
)
Syntax analysis finished successfully!

Variables list:
(Name: m1, Pos: 0, Assignment: 0)
(Name: m2, Pos: 1, Assignment: 0)
(Name: m3, Pos: 2, Assignment: 0)
(Name: r1, Pos: 3, Assignment: 0)
(Name: r2, Pos: 4, Assignment: 0)
(Name: r3, Pos: 5, Assignment: 0)
(Name: r4, Pos: 6, Assignment: 0)
(Name: r5, Pos: 7, Assignment: 0)
```

The Windows taskbar is visible at the bottom, showing the search bar and various application icons. The system tray on the right indicates the language is ENG, the date is 6/4/2021, and the time is 8:56 PM.

**Slika 5** Primer dobrog izvršavanja sintaksne analize

