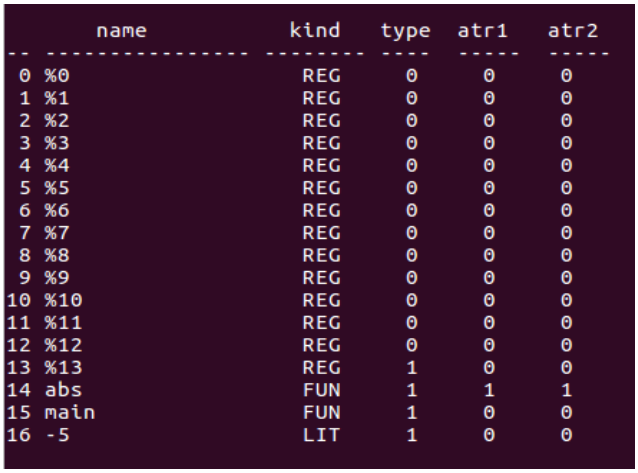
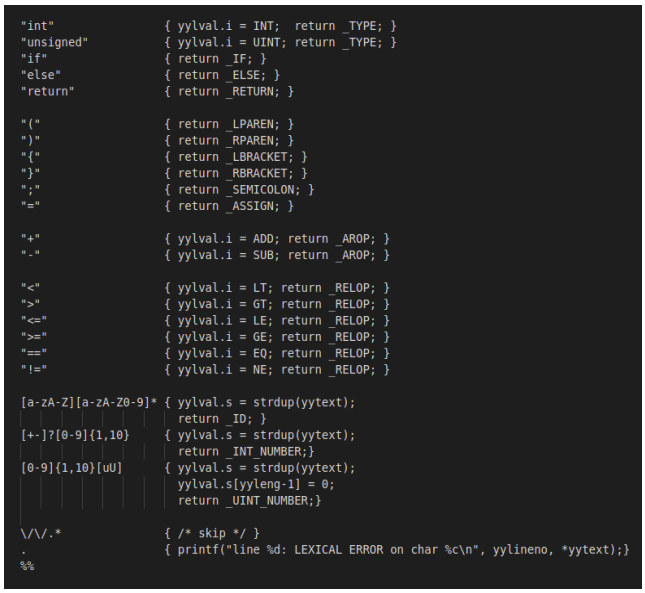
Pisanje regularnih izraza:

- + jedan ili vise
- {3} tacno tri
- {2,5} izmedju 2 i 5
- {3,} 3 ili vise
- * nula ili vise
- ? jedan ili nijedan
- . bilo koji karakter
- \ navodi se kada mislimo na neki karakter identican nekom od specijalnih
- () grupisanje izraza
- | alterantiva jedno ili drugo
- [] navodjenje klase znakova npr [0-9]
- . bilo koji znak sem \n
- ^ negacija, sve sto nije npr. ^T sve sto nije t



	name	kind	type	atr1	atr2
0	%0	REG	0	0	0
1	%1	REG	0	0	0
2	%2	REG	0	0	0
3	%3	REG	0	0	0
4	%4	REG	0	0	0
5	%5	REG	0	0	0
6	%6	REG	0	0	0
7	%7	REG	0	0	0
8	%8	REG	0	0	0
9	%9	REG	0	0	0
10	%10	REG	0	0	0
11	%11	REG	0	0	0
12	%12	REG	0	0	0
13	%13	REG	1	0	0
14	abs	FUN	1	1	1
15	main	FUN	1	0	0
16	-5	LIT	1	0	0



```
"int"               { yylval.i = INT;  return _TYPE; }
"unsigned"          { yylval.i = UINT; return _TYPE; }
"if"                { return _IF; }
"else"              { return _ELSE; }
"return"            { return _RETURN; }

"("                 { return _LPAREN; }
")"                 { return _RPAREN; }
"{"                 { return _LBRACKET; }
"}"                 { return _RBRACKET; }
";"                 { return _SEMICOLON; }
"="                 { return _ASSIGN; }

"+"                 { yylval.i = ADD; return _AROP; }
"-"                 { yylval.i = SUB; return _AROP; }

"<"                 { yylval.i = LT; return _RELOP; }
">"                 { yylval.i = GT; return _RELOP; }
"<="                { yylval.i = LE; return _RELOP; }
">="                { yylval.i = GE; return _RELOP; }
"=="                { yylval.i = EQ; return _RELOP; }
"!="                { yylval.i = NE; return _RELOP; }

[a-zA-Z][a-zA-Z0-9]* { yylval.s = strdup(yytext);
                       return _ID; }
[+-]?[0-9]{1,10}     { yylval.s = strdup(yytext);
                       return _INT_NUMBER;}
[0-9]{1,10}[uU]      { yylval.s = strdup(yytext);
                       yylval.s[yyleng-1] = 0;
                       return _UINT_NUMBER;}

\/\/.*              { /* skip */ }
.                   { printf("line %d: LEXICAL ERROR on char %c\n", yylineno, *yytext);}
%%
```

STRING SIMBOLA	VRSTA SIMBOLA	TIP SIMBOLA	ATRIBUT SIMBOLA	ATRIBUT SIMBOLA
main	FUN	povratni tip funkcije	broj parametara	tip parametra
p	PAR	tip parametra	redni broj parametra	-
x	VAR	tip promenljive	redni broj promenljive	-
100	LIT	tip literala	-	-
%0	REG	tip sadrzaja registra	-	-

1. Sintaksnu gresku predstavlja?
   Pogresan redoslijed simbola.

2. Pretraga tebele u cilju pronalazenja identifikatora se vrsi pomocu funkcije?
   lookup_symbol()

3. Koliko ima globalnih identifikatora u tekstu?
   ```
   int f(int p)
   {
           int m;
           int n;
           return m;
   }
   int main()
   {}
   ```
   Ima ih 2 f i main jer njihov lookup bilo gdje nece nikada biti 0 - globalno vidljivi.

4. Najvise kompajera radi?
   Lokalnu optimizaciju.

5. Koliko ima identifikatora u primjeru?
   ```
   int fun()
   {
           return 0;
   }
   int main()
   {

   }
   ```
   Ima ih 0 jedino sto je lokalno je _TYPE a on nije _ID

6. Koliko ima lokalnih identifikatora u primjeru:

```
int f()
{
        int x;
        int y;
        int z;
        return x+y+z;
}
```

Ima ih 3 i to su x, y, z

7. Koliko stringova odgovara navedenom regulatnom izrazu 0[xX][0-9A-Fa-f]{1,4}?

-0x1234 **(moze)**
-0x1010 **(moze)**
-0x3h45 **(ne moze zbog h)**

8. Koliko puta skener prepozna token identifikatora (_ID) u narednom primjeru?

```
int main()
{
        a=x*5;
}
```

Tri puta: main, a, x;

9. Koji od navedenih pojmova se prepozna u toku parsiranja prikazanog dijela koda?

```
x=y+5;
```

- num_exp **(tacno)**
- exp **(tacno)**
- variable **(netacno - variable je kod deklaracije promjenjive)**
- compund_statement **(netacno)**

10. U datom pravilu meta promjenjiva $3 odgovara vrijednosti pojma/simbola/akcije?

```
a: _A
        {printf("A");}
    b
        {printf("B");}
;
```

b - jer imamo jedan izraz

11. Kakva greska postoji u miniC kodu?

```
int x;
int y;
x=x+y
```

Nedostaje ; a to je sintaksna greska

12. Koji od navedenih pojmova se prepozna u toku parsiranja prikazanog dijela koda?

```
x=f();
```

-exp , argument : Vjerovatno je samo to tacno od ponjudjenog a da je bilo ponudjeno bilo bi tacno i: statement, assignment_statement, function_call;

13. Kakva greska postoji u miniC kodu?
    a = x * y – 3;
    Leksicka jer miniC ne poznaje *

14. Koliko ima globalnih identifikatora?
```
int f()
{
        int x;
        int y;
        return x*y;
}
```
    Jedan i to je f
    Da je pitala lokalnih odgovor bi bio 2: x i y.

15. Koliko puta skener prepozna token identifikatora (_ID) u primjeru?
```
int f(int p)
{
        return p+p;
}
```
    Cetiri puta f, p, p, p

16. Ulazni miniC kod sadrzi koji niz tokena?
```
int f()
{
        a = 0:
}
```
    _TYPE _ID _LPAREN _RPAREN _LBRACKET _ID _ASSIGN _INT_NUMBER _RBRACKET

17. Koji od navedenih pojmova se propozna u toku parsiranja dijela koda?
```
int main()
{
        return a;
}
```
    - exp **(tacno)**
    - return_statement **(tacno)**
    - function_call **(netacno)**

18. U datom pravilu meta promjenjiva $4 odgovara vrijednosti pojma/polja/simbola/akcije?
```
sentence: _CAPITAL_WORD
                {printf("_CAPITAL_WORD"); words_counter++;}
          words
                {printf("words");}
          _DOT
                {printf("_DOT");}
        ;
```
    Odgovor: {printf("words");}

19. Vrijednost polja kind u tabeli simbola za simbol s je?

```
int f()
{
        int s;
        int t;
        return x;
}
```

VAR

20. Koji regularan izraz odgovara sledecem zahtjevu:
Simbol se sastoji od znaka "_" iza kojeg slijede jedno ili dva mala slova a iza nijedno ili vise velikih slova ili cifara.
Odgovor: _[a-z][a-z]?[A-Z]*[0-9]* ili _[a-z]{1,2}[A-Z0-9]* i slicno.

21. Koji od navedenih pojmova se prepozna u toku parsiranja prikazanog dijela koda?
x=x+a;
-literal **(netacno)**
-exp (**tacno)**
-statement **(tacno)**
-variable **(netacno)**

22. Koji niz tokena ce se prepoznati u miniC datoteci? a=f();
_ID _ASSIGN _ID _LPAREN _RPAREN _SEMICOLON (Ne zaboravi ;)

23. Formalna gramatika se sastoji od?
- simbola **(tacno)**
- startnog pojma **(tacno)**
- pojmova **(tacno)**
- pravila **(tacno)**
- tokena **(netacno)**

24. Koji niz tokena ce biti prepoznat u ulaznoj miniC datoteci
a = b + (c – d);
_ID _ASSIGN _ID _AROP LPAREN _ID _AROP _ID _RPAREN _SEMICOLON

25. Bootstraping je proces u kome kompajler:
sam sebe kompajlira

26. Upisati vrijednost polja attr u tabeli simbola za simbol b

```
int f()
{ int a; int b; return a+b+5; }
```

Attr kod promjenjivih je redni broj promjenjive pa je onda 2;

27. Koja vrijednost je zapisanja u polju attr u tabeli simbola za simbol f?

```
int f(unsigned p)
{
        int n;
        unsigned m;
        n=6;
}
```

Bude zapisana vrijednost 1 jer attr sazdrzi broj parametara funkcije
Da je pitala za m odgovor bi bio 2 jer je druga promjenjiva po redu

28. Kakva greska postoji u miniC kodu?

```
int f(int p) { }
int main() { return f(); }
```

semanticka jer poziv funkcije f mora da ima parametar tipa int.

29. Za lokalne identifikatore u miniC programu vazi da:
- postoje samo za vrijeme izvrsenja funkcije
- se oslobadjaju na kraju izvrsenja funkcije

30. Koji od regulatnih izraza definise string cija je minimalna duzina 2 znaka
[a-z][0-9]?[a-zA-Z]

31. Kakva greska postoji u miniC kodu?

```
a=x + y ? 3;
```

leksicka jer ? nije definisan

32. Koji parseri su pogodni za automatsko genrisanje?
LR parseri

33. Koju semanticku provjeru treba izvrsiti prilikom prepoznavanja pojma function_call?
Uporediti broj parametara i argumenata, uporediti tipove parametara i argumenata

34. Koja vrsta greske postoji u miniC kodu?

```
int f2(int num) { return num*num; }
```

leksicka jer ne prepoznaje *

35. Da li je ovaj kod semanticki ispravan?

```
int f(int num) { int n; n=10; return num + n + n; }
int main() { int n; n=5; return f(n-3); }
```

jeste tacan je.

36. Da li je ovaj kod semanticki ispravan?

```
int f(int num){return num;}
int main() { int n; n=8; f=n+5; }
```

ne sa lijeve strane = je funkcija

37. Koja vrsta greske postoji u dijelu miniC koda?
int main() { int n; unsinged m; if(n<m)return n+m; }
semanticka jer su n i m razlicitog tipa.

38. Koja vrsta gresek postoji u dijelu miniC koda?
int main() { int n; unsigned m; if(n<7) return m; }
semanticka jer je povratna vrijednost pogresnog tipa.

39. Regularnom izrazu [0-9]+\.[0-9]*
- 1 **(netacno)**
- 1. **(tacno)**
- 12. **(tacno)**
- 123.123 **(tacno)**
- .5 **(netacno)**

40. Semantika programskog jezika se opisuje?
neformalno

41. Sintaksa programskog jezika se opisuje?
BNF notacijom (nisam sig.)

42. Koje registre u tabeli imamo?
%0 do %12 **radni registri**
%13 **registar za povr. vrijednost funkcije**
%14 **pokazivac frejma**
%15 **pokazivac vrha stack-a**

43. Skener-flex globalne promjenjive:
char* yytext - **omogucava pristup stringu poslednjeg prepoznatog simbola**
int yyleng - **sadrzi duzinu stringa poslednje prepoznatog simbola int**
yylineno - **sadrzi broj trenutne linije int yylval - sadrzi vrijednost prepoznatog simbola**

44. Kom dijelu miniC koda odgovara izgenerisani asemblerski kod?
PP: Lokalne promjenjive su definisane sa a pa b

subs -8(%14), -4(%14), %0	reg0=(b-a)
adds -4(%14), -4(%14), %1	reg1=(a+a)
subs %0, %1, %0	reg0 = (b-a)-(a+a)
mov %0, 04(%14)	a= reg0

Rjesenje: a= (b-a) - (a+a);

45. 44. Kom dijelu miniC koda odgovara izgenerisani asemblerski kod?
PP: Lokalne promjenjive su definisane sa a pa b

subs $2, -8(%14), %0	reg0=2-b
mov %0, -4(%14)	a=reg0 tj a=2-b

Rjesenje: a=2-b;

46. 45. Koliko ukupno registara ima hipotetski asembler
    Odgovor: 16

47. 44. Kom dijelu miniC koda odgovara izgenerisani asemblerski kod? PP:
    Lokalne promjenjive su definisane sa a pa b
    adds -4(%14), $3, %0 reg0=a+3;
    mov %0, -4(14) a=reg- tj a=a+3
    Rjesenje a=a+3;

48. Koliko zauzimaju registri a koliko memorijske lokacije?
    Registri i mem. lokacije zauzimaju po 4 bajta.

49. Kako su definisane labele?
    Labele pocinju malim slovom iza cega mogu biti slova, cifre, i donja crta. Iza labele se stavlja dvotacka. Sistemske labele pocinju znakom @

50. Operandi mogu biti: neposredni, registarski, direktni, indirektni i indeksni.
    Dijele se na:
    ulazne (moze biti bilo koji od navedenih)
    izlazne (svi osim neposrednih)

    neposredni (cijeli oznaceni ili neoz. brojevi $12 $-2)
    registarski (odg. oznaci registra a vrijednost vrijednosti reg.)
    indirektni (%0)
    direktni (odgovara labeli)
    indeksni (4(%14), -8(%14))

51. Naredba poredjenja:
    CMPX ulazni operand, ulazni operand

52. Naredba skoka
    JMPx ulazni operand

53. Naredba
    JEQ(jedna od JEQ) ulazni operand

54. Naredbe rukovanja stekom:
    PUSH ulazni operand
    POP izlazni operand

55. Naredba poziva potprograma:
    CALL ulazni operand

56. Naredba povratka iz podprograma:
    RET (nema operanada)

57. Naredba sabiranja
ADDx ulazni operand, ulazni operand, izlazni operand

58. Nadredba oduzimanja
SUBBx ulazni operand, ulazni operand, ilazni operand

59. Naredba mnozenja
MULx ulazni operand, ulazni operand, izlazni operand

60. Naredba dijeljenja
DIVx ulazni operand, ulazni operand, izlazni operand

61. Naredba prebacivanja
MOV ulazni operand, ilazni operand

62. Naredba konverzije cijelog broja u razlomljeni
TOF ulazni, izlazni

63. Naredba konverzije razlomljenog broja u cijeli
TOI ulazni, izlazni

64. Direktiva zauzimanja memorijskih lokacija
WORD broj

65. Radni registar se zauzima kod smijestanja rezultata svakog aritmetickog izraza sa dva operanda i jednim operatorom (num exp)
Oslobadja se cim se preuzme njegova vrijednost

66. Kao pokazivac steka registara se koristi promjenjiva free_reg_num koja sadrzi broj prvog slobodnog radnog registra. Ako je free_reg_num >12 = fatalna greska

67. Labele u generisanom kodu moraju biti jedisntvene. Svaka labela se zavrsava brojem. Promjenjiva labnum sadrzi broj aktuelne labele

68. Za slog
```
struct {
            int x;
            int y;
    }z;
z.y=z.x;    generisi kod
```

MOV $z, %0 // Adresu od u reg0
MOV 0(%0), 4(%0) // Prvi operand je prvo smjesten na stack pa onda drugi (stack raste ka dole)

69. Za niz: int n[10];
n[0]=n[1];

MOV $n, %0 // Adresu niza (nultog el u)
MOV 4(%0), 0(%0)

70. Ko mijenja bite status registra?
Naredba poredjena CMP u skladu sa razlikom prvog i drugog ulaznog operanda

71. Prethodni pokazivac frejma se stavlja?
Nakon povratne adrese

72. Sta se sve nalazi na stack frame-u od visih ka nizim adresama
argumenti, povratna adresa, prethodni pokazivac frejma lokalne promjenjive, pokazivac steka

73. Kom dijelu miniC koda odgovara izgenerisani asemblerski kod?

Subs -8(%14), -4(%14), %0	(reg0=b-a)
adds -4(%14), -4(%14), %1	(reg1=a+a)
subs %0, %1, %0	(reg0=reg0-reg1; reg0=(b-a)-(a+a))
mov %0, -4(%14)	(a=rego0)

Rezultat: a=(b-a)-(a+a);

74. Odrediti koliko ima baznih tipova u dijelu asemblerskog

```
@switch0:                 //BB1  (labela pocetak BB, JMP kraj BB)
JMP test0                 // test0 koje nije vidljivo je rezultat nekog skoka i BB je

@case0_0 :                //BB2 (pocinje labelom zavrsava skokom)
MOV $1, state
JMP @exit0                // exit0 koje nije vidljivo …

@case0_1:                 // BB3
MOV $2, state
JMP @exit0

@default:                 //BB4
MOV$0, state //BB4
JMP ...                   //Svaka instrukcija nakon skoka je bazni blok a posto imamo ...
                          //znaci da imamo dalje jos instrukcija
```

Ukupno: 7

Algoritam: Prva instrukcija je pocetak bloka Svaka instrukcija koja je cilj nekog skoka je pocetak bloka, svaka instrukcija nakon skoka je pocetak bloka

75. Koja optimizacija moze da se primjeni na naredbi?
t=1+2;
Slaganje konstanti

76. Kom iskazu odgovara prikazani Java asemblerski program
    iload b
    iload c
    iadd
    istore a

    Odgovor:a=b+c;

77. Prilikom generisanja koda najcesce koriscene promjenjive se smjestaju?
    U registre

78. Kod dvoprolaznog kompajlera prvi prolaz obuhvata?
    Prednji modul kompajlera (front end)

79. Kod dvoprolaznog kompajlera drugi prolaz obuhvata?
    Zadnji modul kompajlera (back end) i to faze generisanja i optimizacije koda.

80. Optimizacija koja se vrsi na jednom baznom bloku?
    Lokalna

81. Optimizacija petlji ima za cilj da
    Smanji broj naredbi u tijelu petlje

82. Oznaci tacan odgovor:
    Kompajler programskog jezika c za Intel procesore se sastoji od prednjeg modula pm1 i zadnjeg modula zm1.
    Kompajler programskog jezika Java za SPARC procesor se sastoji od prednejg modula pm2 i zadnjeg modula zm2.
    Kompajler programskog jezika c++ za MIPS procesore se sastoji od prednjeg modula pm3 i zadnjeg modula zm3.
    Ako svi kompajleri koriste isti medjujezik tada pm1 i zm3 odgovaraju kompajleru.

    Tacno: Programskog jezika c za MIPS procesor

83. Dat je izraz u postfiksnoj notaciji. Tom izrazu odgovara sledeci izraz u infiksnoj notacija?
    2 a b + unarni_minus * c d -/

    Postupak: Gledamo sa lijeva dok ne nadjemo prvu operaciju kad na nju naidjemo onda je izvrsimo sa prethodna dva (ulijevo), izbrisemo iz niza i ponavljamo postupak

    2 (a+b) unarni_minus * cd-/
    2-(a+b)*cd-/
    2*(-(a+b))cd-/
    2*(-(a+b)) (c-d)/
    (2*(-(a+b)))/(c-d)

    Odgovor: (2*(-(a+b)))/(c-d);

84. Odrediti u kojoj liniji se nalazi suvisna naredba u asemblerskoj sekvenci?

```
@while0:    CMPU a,b
            JEQ @false1
@true0:     CMPU a,b            // Vec je poredjeno flag je podesen
            JLEU @false1
@true1:     SUBU a,b,a
            JMP @exit1
@false1:    SUBU b,a,b
@exit1:     ADDU a,b,c
```

U liniji 3

85. Sta se u toku optimizacije medjukoda radi sa suvisnim(nedostupnim) naredbama?
Izbacuju se.

86. Odrediti na kojoj liniji se zavrsava dugi bazni blok u dijelu asemblerskog koda?

```
MOV $0, suma            // BB1
MOV $0, I               // BB1
@lab3: CMPS i, $5       // BB2
JGTS @exit3             // BB2
ADDS suma,i,%0          // BB3
MOV %0, suma            // BB3
```

U liniji 4

87. Odrediti u kojoj liniji se nalazi nedostupna naredba u asemblerskoj sekvenci?

```
@lab0: CMPS c,d
       JE @true0
       JNE @exit0
       SUBS %4, $1, %3      // Ova
@true0: MOC v,d
```

88. Kom dijelu miniC koda odgovara izgenerisani asemblerski kod?

```
SUBS $2, -8(%14), %0        reg0=2-b
MOV %0, -4(%14)             a=reg0;         a=2-b
```

Odgovor: a=2-b;

89. Dat je izraz u infiksnoj notaciji. Tom izrazu odgovara sledeci izraz u postfiksnoj notaciji?
(a+b)/(-5*(-3+c));
(a+b)(-5*(-3+c))/
ab+ (-5*(-3+c))/
ab+(5 unarni_minus * (-3+c))/
ab+(5 unarni_minus * (3 unarni_minus c+))/
ab+ 5 unarni_minus 3 unarni_minus c + * /

90. Datom izrazu u infiksnoj notaciji (a-b)/(-(a+d)-a)*c odgovara:
(a-b)/(-(8+d)-a) c *
(a-b)(-(8+d)-a)/c *
(ab-)(-(8+d)-a)/c *
(ab-)(-(8d+)-a)/c *
(ab-)(8d+ unarni_minus -a)/c *
ab-8d+unarni_minus a -/c *

91. Odrediti u kojoj liniji je moguca algebarska transformacija u asemblerskoj sekvenci
CALL pp
MOV %3, b
MOV c,a
MULS $2, %2, %3 // **LINIJA 4**
CMPS a,b
JGRS @lab1

Redukcija se moze vrsiti u smislu:

- Neke instrukcije se mogu uprostiti (ovo se zove *redukcija snage*, engl. *strength reduction*):

```
x := x * 0          x := 0
y := y ^ 2          y := y * y
y := 2 * x          y := x + x
x := x * 8          x := x << 3
x := x * 15         t := x << 4; x := t - x;
```

92. Interpreter sadrzi?
Leksicku, sintaksnu, generisanje medjukoda.

93. U hipotetskom asemblerskom jeziku kao pokazivac frejma se koristi registar?
%14

94. Koja optimizacija je primjenjena u sledecoj transformaciji koda?
a=2; b=4+3; b=7;
Slaganje konstanti jer je a+a =4 a ovde je u prekompajliranju to sracunato i odma zapisano.

95. U hipotetskom asembleru registar %15 se koristi kao?
Pokazivac vrha steka.

96. Pozeljne osobine kompajlera su:
Dobra dijagnostiga gresaka i dobra optimizacija.

97. Koje faze kompajliranja pripadaju zadnjem modulu kompajlera (back end)?
Generisanje koda, optimizacija koda. (Pazi da ne zaokruzis medjukoda)

98. Koje faze kompajliranja pripadaju prednjem modulu kompajlera (back end)?
Leksicka, sintaksna, semanticka analiza

99. Kom dijelu miniC odgovara izgenerisani asemblerski kod?
Adds -4(%14), $3, %3
MOV %0, -4(%14)
a=a+3;

100. Koji niz tokena ce se prepoznati?

```
int f()
{
        a= 0;
}
```

_TYPE _ID _LPAREN _RPAREN _LBRACKET _ID _ASSIGN _INT_NUMBER _SEMICOLON _RBRACKET

101. Koliko ukupno ima globalnih identifikatora u sledecem primjeru?

```
int zero(){}
int abs()
{
        int x;
        if(x>0)
                return x;
        else
                return -x;
}
int main(){}
```

Ima ih 3: zero, abs i main

102. Oznaci tacan odgovor. Odrediti ispravan FOLLOW skup pojma T
E -> T X
T - > (E) | int y
X -> + E | epsilon
Y - > *T | epsilon

Odgovor:{ +, $ }} Nez kako se rad

103. U kojoj liniji je moguca algebarska transformacija u asemblerskoj sekvenci?

```
CALL pp
MOV %3, b
MOV c, a
MULS $2, %2, %3     // Ovde: Jer je moglo ADDS %2, %2, %3
CMPS a, b
JGTS @lab1
```

104. Odrediti tacan odgovor. Odrediti ispravan FIRST skup pojma X

E -> T X
T -> (E) | int y
X - > + E | epsilon
Y - > *T | epsilon
Odgovor:(+, epsilon) (Nez kako se ovo radi)

105. Koji od navedenih stringova odgovara regularnom izrazu (a+b?) | (cd*e), posmatrati cio string.

ce, aaa, abce ne zadovoljava

106. Koji regularni izraz odgovara sledecem zahtjevu: Simbol se sastoji od donje crte iza koje slijedi jedno ili dva mala slova a iza njih nijedno ili vise velikih slova ili zifara

_[a-z][a-z]?((A-Z)|(0-9))*

107. Za ulaz:

```
int f()
{
        int b;
        return b + 2 + f();
}
```

Ispisati vrijednost polja attribute u tabeli simbola za f.

Odgovor 2: Nije mi jasno zasto..

108. Sintaksnu gresku pretstavlja?

Pogresan redoslijed simbola.

109. Kakva je greska u miniC kodu?

```
a= x* y  - 3.
```

Leksicka

110. Koliko puta skener prepozna token _ID u sledecem kodu?

```
int f()
{
        int s;
        s=0;
}
```

Tri puta: f, s, s

111. Koja vrijednost je upisana u polje ptype u tabeli simbola za simbol f?

```
int f (unsigned p)
{
        int n;
        unsigned m;
        n=6;
}
```

Konstanta UINT jer je to tip parametra funkcije.

Prevodjenje je prepoznavanje iskaza (recenica) izvornog jezika i sastoji se od prepoznavanja rijeci(simbola) i recenice(pojma).

Prevodjenje = Analiza + Sinteza + Optimizacija
Analiza - prepoznavanje iskaza izvornog programa
Sinteza - generisanje iskaza ciljnog programa
Optimizacija - poboljsavanje programa

ANALIZA:
(SKENER) Leksicka analiza - prepoznavanje rijeci(simbola), otkrivanje pogresnih simbola
(PARSER) Sintaksna analiza - prepoznavanje iskaza (recenica), otkrivanje formalno pogresnih iskaza
Semanticka - prepoznavanje znacenja iskaza, otkrivanje sematicki pogresnih iskaza

SINTEZA(Generator koda): prethodno prepoznati iskazi(recenice) izvornog prog. jezika se prevode u recenise ciljnog jezika (asemblerski npr.)

Skener

Prepoznaje leksicke jedinice (simbole) iz ulaznog teksta preuzimajuci znak po znak koji pokusava da prepozna ignorisuci znakove koji razdvajaju sibole (" ", \t, \n) te reaguje na nedozvoljene znakove.

String simbola je leksema, klasa simbola (skup stringova) je token. Skener dijeli ulazni tekst na lekseme, identifikuje token svake lekseme tako sto preuzme opis simbola i na osnovunjega prepoznaje i kalsifikuje simbole. Dijagram prelaza je usmjeren graf u ciji sastav ulaze cvorovi (stanja), usmjerene spojnice (moguci prelasci), labele i znakovi (koji obrazuju labele).

Generator skenera: Simboli se opisuju pomocu regularnih izraza. Generator tabele prelaza odredjuje ponasanje skenera (FLEX).

skener.l je flex specifikacija koju FLEX pretvara u lex.yy.c fajl koji definise funkciju yylex() i predstavlja skener u obliku C programa kojeg sam c kompajler revodi u izvrsni oblik skenera.

Unutar .l fajla izmedju %% %% se pisu pravila.

Flex generise skener u obliku C funkcije yylex() koja se pravi na osnovu regularnih izraza i njima pridruzenih akcija. Flex zahtjeva definiciju funkcije yywrap() koja opisuje ponasanje skenera kada naidje na EOF znak.

yylex() ponavlja prepoznavanje simbola i izvrsava zadate akcije sve dok se u okviru akcija ne izvrsi return iskaz.

Globalne promjenjive unutar skenera su:
- yytext Promjenjiva za pristup stringu poslednjeg prepoznatog simbola

- yyleng Duzina stringa poslednjeg prepoznatog simbola
- yylineno Sadrzi broj trenutne linije
- yyval Prenos vrijednosti prepoznatog simbola

Parser

Provjerava da li je ulazni niz simbola (tokena) u skladu sa gramatikom, kao ulaz od skenera prima niz tokena a kao izlaz daje stablo parsiranja programa.

Da bi razlikovao ispravne od neispravnih nizova tokena potrebno mu je:
- Jezik sa opis ispravnih nizova tokena
- Metoda za ralikovanje ispravnih od neispravnih nizova tokena

BNF gramatika se sastoji od pravila koja odredjuju dozvoljene nacine redjanja pojmova i simbola. Jedan od pojmova je i polazni pojam koji je zapravo korjen stabla.

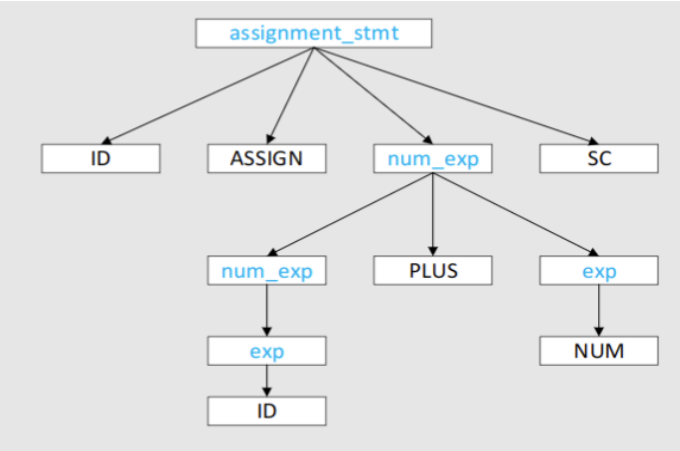
Simboli su na nivou leksicke analize dok su pojmovi na nivou sintaksne analize.

EBNF(ExtendedBNF) na desnoj strani pravila moze da sardzi:

- [...] Sadrzaj zagrada se moze pojaviti jednom ili nijednom (0 ili 1)
- {...} Sadrzaj zagrade se moze pojaviti nijednim jednom ili vise puta (0 ili vise)
- (...) Grupisanje
- | Alterantive

Pravila se koriste u parseru tako sto se lijeva strana mijenaj desnom. Simboli se ne mogu dalje zamijeniti jer ne postoje pravila za simbole.

Izvodjenje je niz promjenjivih pravila gdje se pocinje od pocetnog pojma, primjenjuje se zamjena pravila, jedan po jedan pojam cime se formira stablo ciji su listovi simboli (terminali) a cvorovi pojmovi (neterminali). Jedno stablo parsiranja moze iamti vise izvodjenja.



Leftmost derivation (najljevlje izvodjenje, izvodjenje s lijeva) je izvodjenje u kojem se uvijek primjenjuje pravilo na prvi pojam sa lijeve strane.

Rightmost derivation (najdesnje izvodjenje, izvodjenje sa desna) je izvodjenje u kojem se uvijek primjenjuje pravilo na prvi pojam sa desne strane.

Postoje izvodjenja koja ne prate pravila ni jedne ni druge vrste.

Gramatika je dvosmislena onda kada za jedan ulazni string imamo vise stabala sto je LOSE. Ako svaki string ima jedinstveno stablo parsiranja tada ima jedinstveno i najljevlje i najdesnje izvodjenje.

Kod BISON-a:

- $$ Oznacava vrijednost pojma sa lijeve strane pravila
- $i -||- desne strane pravila

Na steku $i oznacava lokacije ispod vrha steka, $$ nakon njihove redukcije oznacava vrh steka.

Parser pokusava da prepozna desnu stranu pravila i da je redukuje u lijevu stranu sve dok ne dodje do polaznog pojma (SIALZNO PARSIRANJE).

BISON:
Unutar parser.y se nalazi BISON specifikacija na osnovu koje BISON formira LR parser u obliku C programa (parser.tab.c) kojeg C kompajelr prevodi u izvrsni oblik LR parsera.

Bison generise LR parser u obliku C funkcije yyparse() koja ukazuje na gresku ako vrati vrijednost razlicitu od nule.

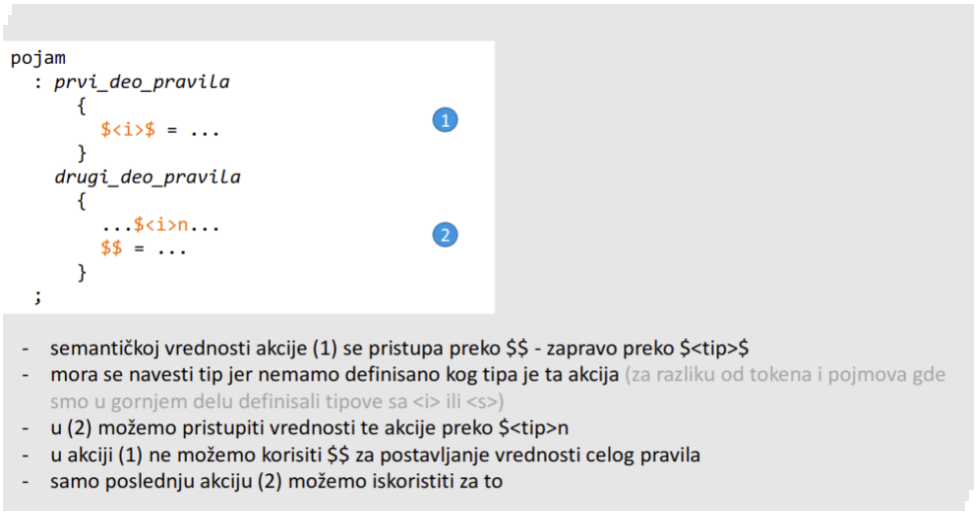
Pravila u BNF notaciji:
- Podrazumjeva se da svaki pojam i simbol iz pravila posjeduju vrijednost gdje vrijednost simbola odredjuje skener a vrijednost pojma parser.

A - > B {$$=5;} C {$$=$2;} => A=5 (Samo ne {} na prvo mjesto desne strane pravila)
A - > B {$$=5;} C {$$=1;} => A=1

Skener parseru daje token i njegovu vrijednost.

Napomena za semantiku: Vrijednost pojma exp je indeks u tabeli simbola gdje se nalazi rezultat izraza, vrijednost pojma literal je indeks u tabeli simbola na kome se on nalazi, povratan vrijednost funkcije se nalazi u registru %13 i ujedno vrijednost pojma function_call je registar %13.

Pojam-pravilo-akcija:



Dvosmislena gramatika

Gramatika je dvosmislena kada za jedan ulazni stringpostoji vise stabala parsiranja ili vise od jednog izvodjenja. Problem dvosmislenosti gramatike je nudjenje dvije ili vise ravnopravnih mogucnosti izvodjenja cime se kompajleru ostavlja da se odluci za jednu od vise interpretacija programa. Rjesenje je deklaracija lijeve asocijativnosti %left +,*

Generatoru LR parsera moraju biti saopstene nedvosmislene gramatike da bi on u svaki element tabele akcija i relaza mogao da smjesti oznaku samo jedne akcije dok se u suprotnom moze javiti vise akcija kao kandidata za isti element tabele tj. stvara se konflikt.

shift-reduce konflikti se javljaju kada posmatrajuci trenutni token moze da se izvrsi i jedna i druga akcija.

reduce-reduce konflikti se javljaju kada postoji vise previla koja se mogu primjeniti na ulazni niz. (predstavljaju ozbiljan problem i bilje je prouciti i izmijeniti gramatiku)

Bison rjesava konflikte na dva nacina, kod shift-reduce konflikta prednost daje shift akciji a kod reduce-reduce konflikta daje prednost reduce naredbi.
Deklaracije prioriteta se mogu definisati samo jednom za jedan token i to se radi upotrebom %prec modifiaktora kojim se deklarise prioritet nekog pravila.

Postoje sledece deklaracije (navode se umjesto %token):
- %left - lijevo asocijativan
- %right - desno asociajtivan
- %nonassoc - nije asocijativan
- %precedence - definise samo prioritet ne i asocijativnost

Postoji i modifikator (navodi se kod pravila %prec) - navodi se na kraju pravila gdje pravilo preuzima prioritet navedenog tokena za potrebe promjene prioriteta u nekom kontekstu.

Token je par kojeg cine vrsta tokena i vrijednost tokena a lekseme su instance tokena koje su prepoznate tokom parsiranja.

Regularni izrazi su vrsta formalnih jezika. Za svaki regularni izraz postoji konacan automat.

Tipovi gramatika:
- regularne gramatike (regualtni jezici)
- kontekstno slobodne gramatike (context free)
- kontekstno osjetljive gramtiek (context sensitive)
- gramatike bez restrikcija

Svaka gramatika se sastoji od konacnog skupa pojmova, konacnog skupa simbola, konacnog skupa pravila i pocetnog pojma.

Iduci odozgo ga dole , godnja je podskup donje i sve tako.

Dva osnovna tipa parsera:

- Silazno parsiranje (top-down) - pokusavamo da od pocetnog pojma izvedemo ulazni niz simbola prateci pravila produkcije (lijevu stranu zamjenjujemo desnom) - Stablo parsiranja se kreira od korjena prema listovima

- Ulazno parsiranje (bottom up) - pokusavamo da od ulaznog niza simbola dobijemo pocetni pojam (desna strana pravila se zamjenjuje lijevom) - Stablo parsiranja se kreira od listova prema korjenu

Bison je LALR(1) parser - bottom up, left to right, rightmost derivation, 1 lookahead token.

Obrada gresaka i generisanje koda

Poruku o sintaksnoj gresci nam "izbacuje" bison kada je detektovana greska u gramatici (prepoznati tokeni nisu u valjanom redoslijedu), stanje masine je "greska" i nastupa zaustavljanje.

Poruka o sintaksnoj gresci dolazi od nas te nema greske u gramatici pa je stanje masine "nije greska" i nastavlja se.

Proces oporavka od greske je proces u kome se tezi nastavku parsiranja ukoliko se naidje na gresku ukoliko je to moguce, uz davanje odgovarajucih informacija o greskama.

Token **error** je token koji je uvijek definisan te ga bison generise kada god se pojavi greska, ako imamo pravilo koje ga prepoznaje imamo i mogucnost oporavka i parsiranje moze da se nastavi.

Blokovi se u miniC realizuju tako sto se svakom broju dodjeli redni broj, u tabeli simbola se za VAR pamti i redni broj bloka gdje variajbla postoji dokle postoji i blok. U kontekstu generisanja koda realizuju se preko posebnih frejmova za blokove (slicno kao funkcije). Promjenjive su na steku dok traje blok.

Strukture se realizuju preko posebnih tabela simbola za svaku strukturu gdje se u TS pamti relativna pozivija polja zbog pristupa prilikom generisanja koda (slicno lokalnim promjenjivim). Na stack-u je potrebno zauzeti odgovarajuci broj lokacija.

Kod nizova se u TS pamti broj elemenata a na stack-u se zauzme odgovarajuci broj loakcija.

CMPx ima dva ulazna operanda i postavlja bite status reg.
Naredba bezuslovnog skoka smijesta vrijednost ulaznog operanda u programski brojac
Uslovni skok ima ulazni operand a ispunjenost uslova zavisi od bita status registra.
CALL stavlja trenutnu vrijdnost programskog brojaca na vrh stack-a
RET preuzima vrijednost vrha stack-a i smijesta u prog brojac
Izuzetak u ADDx i SUBx je kada rezultat ne moze stati u izlazni operand
WORD je zauzimanje uzastopnog broja mem lokacija
Zauzimanje registra se radi ADD/SUBB oslobadjanje sa MOV
Funkcija cuva lokalne promjenjive i parametre na stack frame-u a brojac lok. promjenjivih je VAR_NUM.
Ako funkcija ne sadrzi return povratna vrijednost je ono sta se zadesi u registru %13

Medjukod i generisanje koda

Faze zavisne od izvornog jezika (analiza) pripadaju prednjem modulu kompajlera - front end
Faze zavisne od ciljnog jezika (sinteza) pripadaju zadnjem modulu kompajlera - back end

Prakticnost podjele na dva modula se ogleda u tome sto se prevodjenje jednog izvornog jezika na vise moze realizovati preko jednog prednjeg i vise zadnjih modula a prevodjenje sa vise izvornih na jedan te isti ciljni jezik preko vise prednjih i jednog zadnejg modula. Ovo za posljedicu ima brze prilagodjavanje promjenama koje nastaju u polaznim i ciljnim arhitekturama.

Medjureprezentacija (Intermediate representation -IR) se realizuje u razlicitim oblicima (kod ili neka struktura) te vecina prevodialca ima neku IR.

Medjujezik (Intermediate language - IL) predstavlja ciljni jezik za prednji modul a izvorni jezik za zadnji modul cime se dobija manji nivo apstrakcije u odnosu na polazni a veci u odnosu na ciljni. Neki prevodioci mogu koristiti sekvencu razlicitih IR. Neki oblici IR su: postfiksna notacija, apstraktno stablo AST, troadresni kod TAC ..

AST je reprezentacija izvornog koda u obliku stabla koriscena u sistemima za analizu i tranformaciju programa.

Postfiksna notacija (postupak objasnjen u pitanjima gore), operator je uvijek iza svojih operanada, ne zahtjeva zagrade, prilagodjena je stack masinama/stack arhitekturama. Moze se dobiti postorder obilaskom sintaksnog stabla.

TAC je low level IL koji se lako prevodi u konkretan asemblerski jezik. Koristi najvise 3 operanda/adrese.

Primjer:

operator source => result
operator op1, op2 => result

result := op1 operator op2
result := op1

Ocigledno je da se neki izrazi moraju predstaviti sa vise instrukcija gdje se medjurezultati smjestaju u privremene promjenjive.

Postoje razlicite implementacije/zapisi koje mogu biti u numerickom obliku (odredjene vrijednosti kodiraju i operacije i operande).

Generator koda preuzima ispravan medjukod i tabelu simbola te proizvodi izrsni (apsolutni) masinski kod ili objektni (relokatibilni) masinski kod ili asemblerski kod. Podrazumjeva se da je proizvedeni kod ispravan i da efikasno koristi resurse racunara. generisanje koda se zasniva na odredjivanju nacina implementacije svake naredbe medjukoda pomocu naredbi koda.

Registarsko adresiranje je najbrzi i najkraci nacin adresiranja. Ako postojeci broj registara nije dovolja potrebno je prelivanje (spilling) gdje se neke vrijednosti moraju cuvati u memoriji.

Postoje dva pristupa:
- Register to Register Model gdje pokusavamo sve da stavimo u registar, biramo vrijednosti koje ce se cuvati u memoriji
- Memory to Memory Model gdje se za sve promjenjive zauzima memorija a vrijednosti po potrebi prebacujemo u registre\

Cilj kompajlera je optimizovanje procesa alokacije u skladu sa arhitetkurom , programom, ogranicenjima.

Register Interference Graph (RIG) je pristup rjesavanju problema alokacije (R-R model) koji modeluje promjenjive kao cvorove a grane postoje izmedju promjenjivih koje se istovremeno koriste sto znaci da dvije promjenjive mogu dijeliti isti registar ako nisu povezane (ne koriste se u istom trenutku). Cvorovi se boje i postoji onoliko boja koliko postoji registara gdje susjedni cvorovi moraju imati razlicite boje.

Optimizacije

Optimizacije predstavljaju izvrsavanje razlicitih transformacija u cilju poboljsanja iskoriscenja resursau cilju:

- ubrzavanja izvrsavanja
- smanjenja memorisjkih zahtjeva
- smanjenja velicine koda
- smanjenja pristupa disku
- smanjenja potrosnje energije ...

Vrste optimizacije:

- Optimizacije medjukoda (prevodioci)
- Optimizacije zavrsnog koda
- Optimizacije izvrsnog koda i izvrsavanja (programeri) - rucno ili vezano uz algoritme prevodjenja

Optimizacija medjukoda

Optimizacija medjukoda je masinski nezavisna optimizacija koja podrazumjeva tasnformacije medjukoda koje znacajno mogu poboljsati koriscenje resursa sto moze dovesti do razlicitih poboljsanja. Optimizacije se sastoje iz dvije komponente: analize i transformacije koda (izbaci naredbu, zamjeni, promjeni argumente ...).

Bazni blok (basic block BB) je najduza sekvenca instrukcija sa jednom tackom ulaza i jednom tackom izlaza. Tok kontrole/izvrsavanja (control-flow) ulazi u BB preko prve instrukcije a izlazi bez zaustavljanja ili grananja osim u poslednjoj instrukciji.

- Nema ulaska u BB (osim na pocetku BB)
- Nema izlaska iz BB (osim na kraju BB)
- BB ne sadrzi labele (osim eventualno u prvoj instrukciji)
- BB ne sadrzi skokove, grananje ili zaustavljanje osim eventualno u poslednjoj instrukciji
- Tok izvrsavanja tece od prve do poslednje isntrukcije bez zaustavljanja
- Postoji jedan pocetni BB
- Cvorovi sa return su zavrsni

Algoritam: Prva instrukcija je pocetak bloka Svaka instrukcija koja je cilj nekog skoka je pocetak bloka, svaka instrukcija nakon skoka je pocetak bloka

Dijagram toka (control-flow graph, CFG) je usmjeren graf kod koga su cvorovi bazni blokovi, grana postoji ako se izvrsavanje programa moze nastaviti od poslednje instrukcije ulaznog BB do prve instrukcije izlaznog BB i ovaj dijagram predstavlja moguce redoslijede tj. tokove izvrsavanja.

Nivoi nad kojima se primjenjuju oprimizacije su:

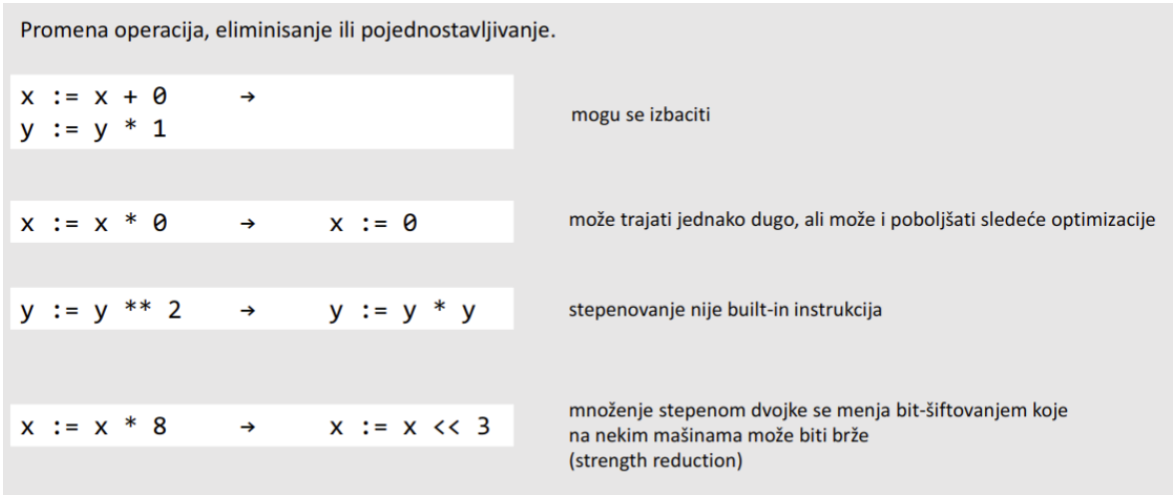
- lokalne (unutar jednog baznog bloka):
  najjednostavnije, trose malo resursa, algoritmi jednostavni, male promjene ali kad ih se primjeni na vise mijesta vidljiv je napredak

- globalne (intraproceduralne) optimizacije (unutar jednog dijagrama toka - tijelo funkcije):
  kod odredjenjih optimizacija kao sto su propagacija kopije ili zajednicki podizraz se mogu pojaviti odredjeni problemi (kontekst putanja toka)

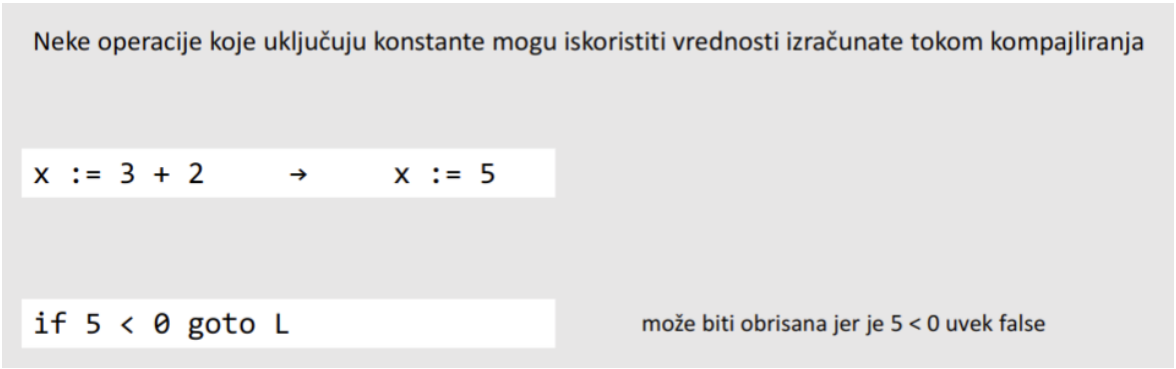
- medjuproceduralne (interproceduralne) primjenjuju se nad cijelim programom i najkomlikovanije su:
  svaki poziv se zamjeni cijelom funkcijom uz odgovarajuce izmjene ili dodatke za argumente i povratne vrijednosti (problem je to sto funkcije mogu biti nepoznate: pokazivac na funkciju, rekurzija). Drugi pristup su Grafovi poziva (str 39 prez. 10)

U lokalne spadaju:

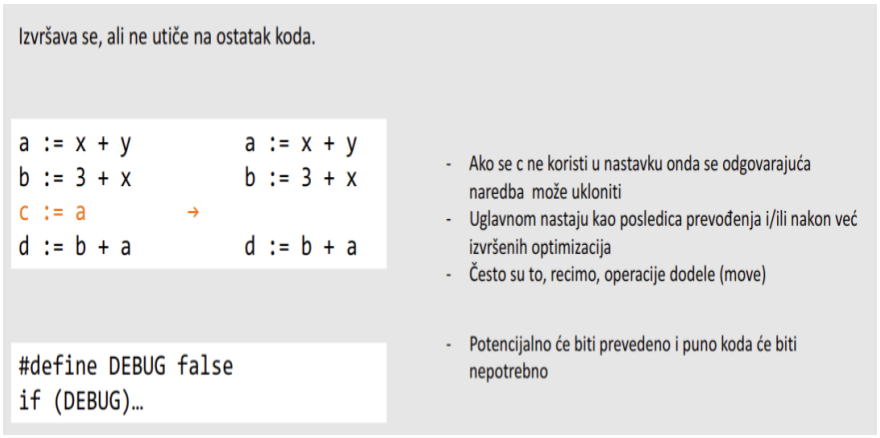
- Algebarske transformacije:



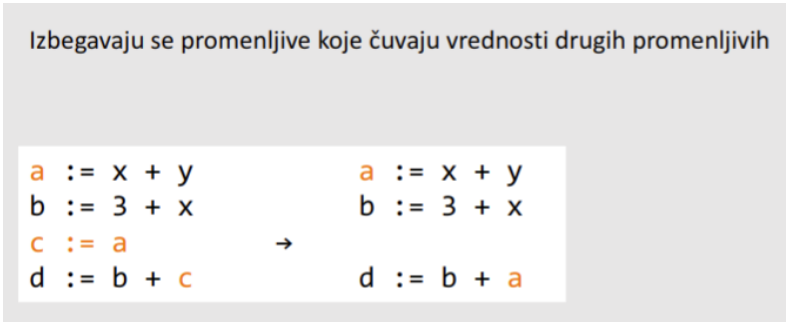
- Slaganje konstanti (Constant folding):



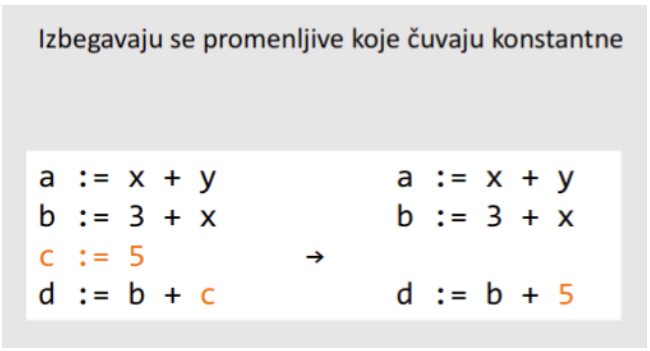
- Mrtav kod (dead code elimination):



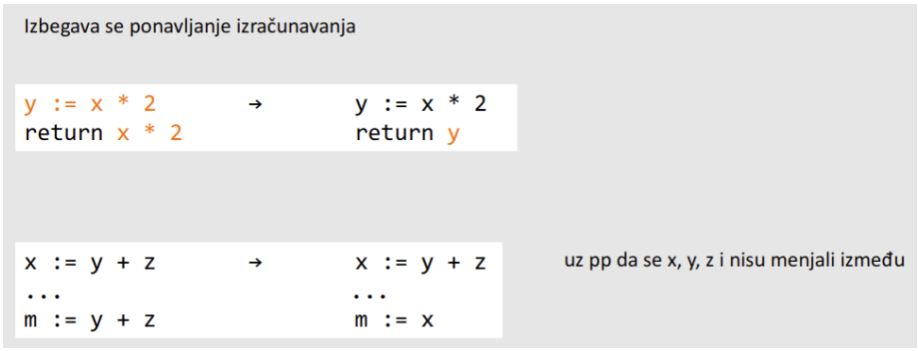
- Propagacija kopije (copy propagation - korisna za naredne optimizacije):



- Propagacija konstante (constant propagation):



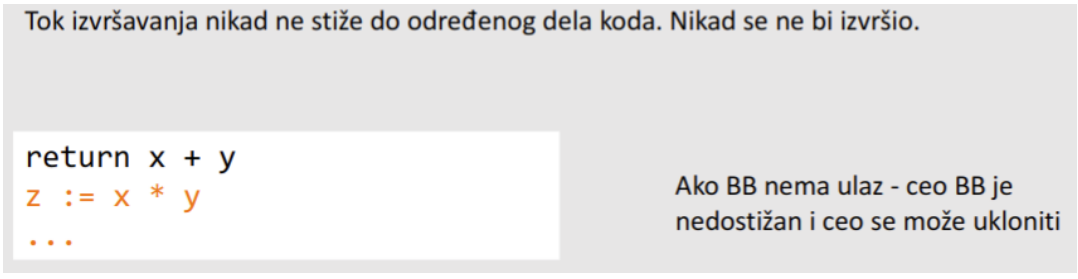
- Zajednicki podizraz:



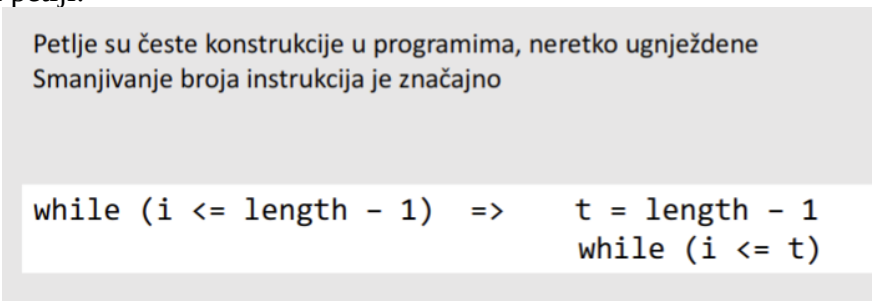
Dodatni primjeri prezentacija 10 strana 26.

U globalne spadaju:

- Nedostupan kod:



- Optimizacija u petlji:



Optimizacije ciljnog koda: Mnoge navedene optimizacije se mogu primjenitii nad asemblerskim jezikom (jer smo se mi odlucili da nam asemblerski jezik bude ciljni jezik, da smo prevodili iz miniC u npr. Javu onda bi nam Java bila ciljni jezik).

Posmatraju se kratke sekvence uzastopnih naredbi (posmatramo ih kroz zamisljeni prorez koji pomjeramo preko naredbi duz koda) i zamjenjuju kracim brzim sekvencama (trazimo na osnovu nekih unapred zadatih slucajeva tipa suvisne naredbe i skokovi, nedostupne naredbe, algebarske trasnformacije i slicno). Primjeri su vec vidljivi u pitanjima iznad.

Neki od najcescih pojava: ista poredjenja (nepotrebno jer je falg ostao nepromjenjen), punjenje registra istim vrijednostima iako u medjuvremenu nije mijenjan, uzastopno pisanje i citanje iz istog registra i iste memorijske lokacije (MOV %0,a pa zatim MOV a, %0), naredbe koje slede iza naredbe skoka a nisu cilj neke druge naredbe skoka (nedostupno nikad se nece izvrsiti treba izbaciti), naredba skoka ciji je cilj naredba bezuslovnog skoka (skok na neku labelu koja odma skace na drugu - skociti odma na tu drugu).

Napomena: Nismo vise u IR sa baznim blokovima i CFG nego se nalazimo u ciljnom kodu.

Optimizacije izvornog koda: sa aspekta programera obicno nakon prevodjenja uz staticke analize (analize koda) i dinamicke analize (analiza izvrsavanja: brzina, koriscenje memorije i drugih resursa)

Analiza izvrsnog koda (iz aspekta programera) se ogleda u zamjeni algoritma brzim algoritmom (primjer sortiranja), zamjena jedne biblioteke drugom, promjena arhitekture koja izvrsava program, neke rucne "mini optimizacije " (koje naredbe se kad brze izvrsavaju...)

Analiza izvrsavanja (iz aspekta programera): prikupljanje podataka prilikom izvrsavanja, analiza podataka i mjerenje performansi, odredjivanje dijelova koji se mogu ili ih je moguce optimizovati.

Profajleri su alati koji obavljaju analizu performansi.

Runtime

Izvrsna okruzenja (runtime environment) predstavljaju podrsku izvrsavanja programa napisanih na visim programskim jezicima. Kompajler je duzan da ispravno **prevede sve apstrakcije** definisane jezikom (npr kalse i sl.), da **omgouci podrsku** za sve te apstakcije na ciljnoj masini.

Izvrsno okruzenje podrazumjeva:
- Organizaciju memorije (kako predstaviti podatke, kako dijeliti memoriju)
- Upravljanje zivotnim vijekom promjenjivih, kako pristuapti razlicitim vrstama podataka
- Upravljanje pozivima funkcija (kako pozvati, akako prenijeti argumente, kako se vratiti iz funk.)
- Upravljanje ostalim apstrakcijama (izuzeci, garbage collection itd.)

Izvrsno okruzenje predstavlja "masinu"; racuanrski model, interfejs za izvrsavanje te moze biti konkretna ili virtuelna masina.

Prevodilac koristi logicki adresni prostor (virtuelne masine) koji OS mapira taj prostor na ciljnu masinu (na fizicke adrese).

Za svaki poziv funkcije se odredjuje blok sa podacima koji odgovaraju i definisu taj poziv funkcije (Aktivacioni blok - activation record AR). Funkcije se izvrsavaju odredjenim redom pri zavrsetku jedne funkcije kontrola se vraca na funkciju koja ju je pozvala.

Sekvenca pozivanja je niz aktivnosti koje se izvrsavaju u cilju formiranja aktivacionog bloka.

Funkcija pozivalac racuna vrijednosti za argumente, zauzima i postavlja mjesto za podatke koji predstavljaju argumente, postavlja povratnu adresu, prenosi kontrolu na funkciju koja se poziva.

Funkcija koja je pozvana postavlja kontrolu veze , obezbjedjuje mjesto za lokalne podatke ...

Sekvenca povratka je niz aktivnosti koje izvrsavaju funkcije pozivalac i pozvana u cilju oslobadjanja aktivacionog bloka.

Pozvana funkcija postavlja povratnu vrijednost (mjesto u AR ili registar), oslobadja prostor za lokalne podatke te prebaci kontrolu na funkciju koja ju je pozvala.

Funkcija pozivalac oslobadja mjesto zauzeto za podatke koji predstavljaju argumente.

Glabalne promjenjive: ako nema unutrasnjih funkcija svaka promjenjiva kojan ije lokalna za jednu nije lokalna ni za sve ostale. Smijestaju se uglavnom u static dijelu gdje su adree fiksne i poznate tokom prevodjenja. Lokalne prom. se mogu zvati isto kao globalne - vidjljivost imena neke druge globalne promjenjive je u svim funkcijama (cije dekalracije su nakon nje) osim u slucaju da u funkciji postoji lokalna sa istim imenom.

Ugnjezdene funkcije:
Funkcije definisane u okviru neke druge funkcije koje imaju pristup podacima te spoljasnje, okruzujuće funkcije. Slicno blokovima te postoji vise nivoa ugnjezdavanja.
Razlika u odnosu na blokove (koji se izvrsavaju poznatim redom) mi ne znamo kako ce izgledati aktivacije funkcija (stek, ARs, sta ako je neka rekurzivna).

Kako pronaci (run-time) koji tacno AR sadrzi podatke za promenljivu koja nam treba u ugnjezdenoj funkciji –ne znamo raspored na steku.

Pristupna veza: dodavanje adrese najnovije aktivacione funkcije (spoljasnje gledano sa aspekta ugnjezdenosti) u svaki AR funkcije. Tako se stvara lanac aktivacija gdje se duz tog lanca mogu pronaci sve dostupne promjenjive.

Memory manager vodi racuna o tome koliko je memorije zauzeto/slobodno, zauzima trazene dijelove memorije, trazi od OS prosirenje po potrebi, oslobadja odredjene dijelove memorije. Sve ovo za cilj ima minimizaciju zauzete memorije i optimizaciju upotrebe (brze izvrsavanje).

Fragmentacija: pocinje se od velike kolicine slobodne momorije (uzastopnih lokacija), memorija se zauzima vrijednostima razlicite velicine, neki djelovi se oslobadjaju i nastaju prazna mjesta (rupe). Ponavljanjem se moze doci do znacajne fragmentacije sto dovodi do toga da prividno nema

memorije (u komadu) iako je zapravo ima ali nije dobro iskoriscena.

Kontrolni mehanizmi se ogledaju u spajanju sukcesivnih slobodnih prostora, optimalno trazenje prostora, restart.

Memorija se moze osloboditi rucno gdje se oslobadja sve sto vise nije potrebno (ako se izostavi dolazi do memory leak sto ne mora ali mzoe biti veliki problem), te se onome sto je oslobodjeno ne smije pristupati.

Garbage collection predstavlja autoamtsko oslobadjanje memorije koja se vise ne koristi (problematicno sa aspekta sporog ukupnog izvrsavanja, velikih pauza, fragmentacije te namjerne ili nenamjerne lijenosti programera da po zavrsetku upotrebe oslobodi resurse ocekujuci da to uradi GC ali sa zakasnjenjem).

Glavni zadatak je kako odrediti dostupen podatke?
Postoje dva pristupa:

- **reference counting**(broje se reference na objekat kada broj padne na 0 objekat postaje nedostupan i postaje kandidat za GC) . U svakom trenutku se zna status jednog objekta sto je brze.

- **trace-based GC**(u odredjenim momentima se odredjuje skup dostupnih objekata, krene se od pocetnog i prate se sve reference. Kada se formira skup dostupnih objekata svi ostali postaju kandiati za brisanje). U jednom trenutku se saznaje status svih objekata sto je prilika za manju fragmentaciju.