Konstrukcija kompilatora

— Proces kompilacije —

Milena Vujošević Janičić

www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

Pregled

1. Leksička analiza
2. Sintaksička analiza
3. Semantička analiza
4. Generisanje međukoda
5. Optimizacija
6. Generisanje koda

Literatura

- (The Dragon Book) Compilers: Principles, Techniques, and Tools Alfred V. Aho,Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Kompajleri Stanford `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/`
- Konstrukcija kompilatora EPFL `https://lara.epfl.ch/w/cc`
- Skripta Filipa Marića `http://poincare.matf.bg.ac.rs/~filip/kk/materijali/kk.pdf`

Example: **gcc**

- from C to Intel x86

```
#include <stdio.h>
int main(void) {
 int i = 0;
 int j = 0;
 while (i < 10) {
  printf("%d\n", j);
  i = i + 1;
  j = j + 2*i+1;
 }
}
```

gcc test.c –S
⇒ test.s

```
      jmp .L2
.L3:      movl -8(%ebp), %eax
      movl %eax, 4(%esp)
      movl $.LC0, (%esp)
      call printf
      addl $1, -12(%ebp)
      movl -12(%ebp), %eax
      addl %eax, %eax
      addl -8(%ebp), %eax
      addl $1, %eax
      movl %eax, -8(%ebp)
.L2:
      cmpl $9, -12(%ebp)
      jle .L3
```

Konstrukcija kompilatora

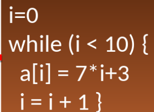
- Izvorni kôd (npr. Scala, Java, C, C++, Python) je dizajniran da ga programeri lako koriste, trebalo bi da odgovara načinu razmišljanja programera kako bi im to pomoglo da budu produktivni, da izbegnu greške, da koriste apstrakcije
- Kôd arhitekture (npr. x86, MIPS, arm, JVM, .NET) je dizajniran tako da može efikasno da se izvršava na hardveru ili virtuelnoj mašini, brz, kompaktan, niskog nivo
- **Kompajleri spajaju ova dva sveta**

Compiler

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write

Compiler



source code
(e.g. Scala, Java,C)
easy to write

Compiler
(scalac, gcc)

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write

Compiler
(scalac, gcc)

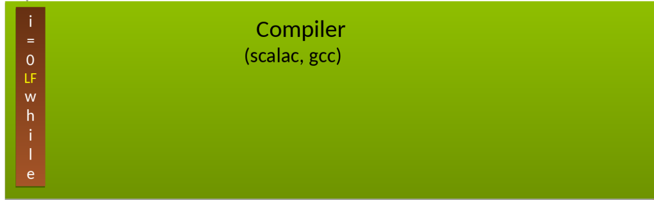
```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, ARM, JVM)
efficient to execute

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write



characters

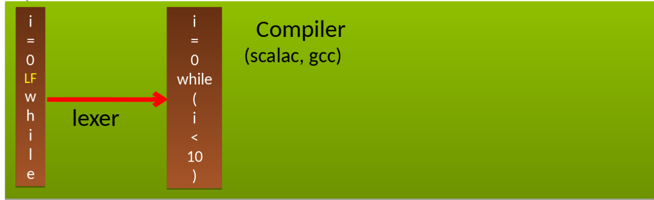
```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, ARM, JVM)
efficient to execute

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write



characters

words

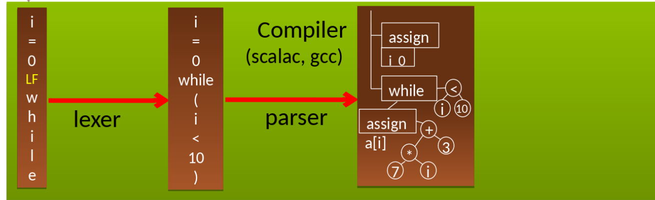
machine code
(e.g. x86, ARM, JVM)
efficient to execute

```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write



characters
words
trees

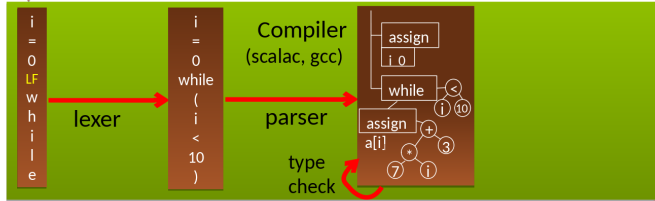
```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, ARM, JVM)
efficient to execute

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

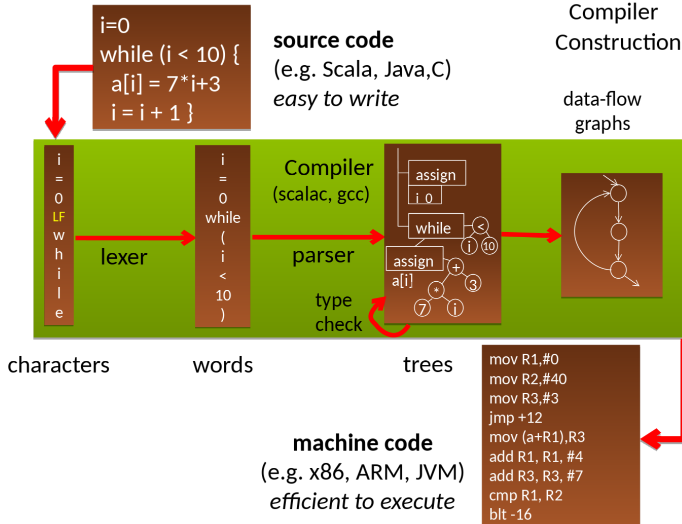
source code
(e.g. Scala, Java,C)
easy to write

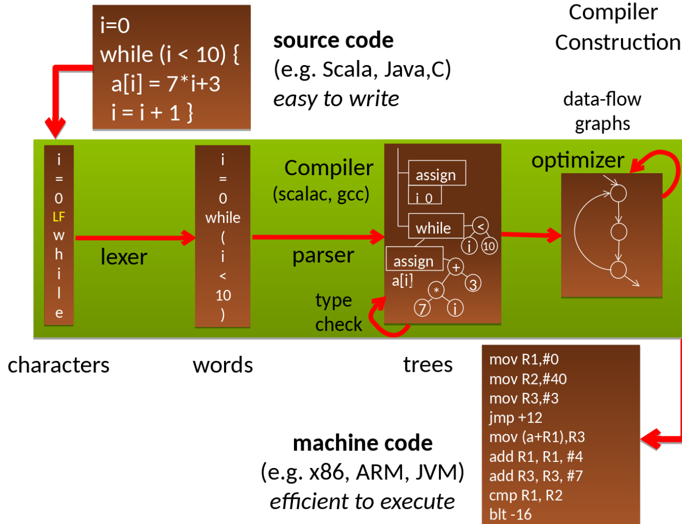


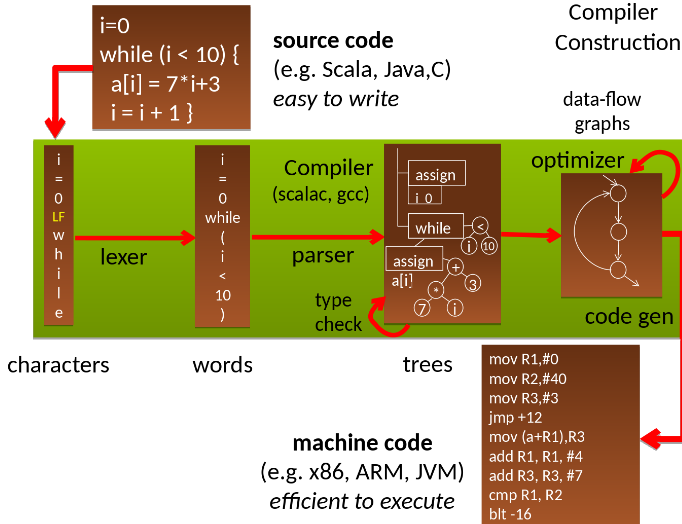
characters words trees

```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, ARM, JVM)
efficient to execute


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
Construction
data-flow
graphs
Compiler
(scalac, gcc)
lexer
parser
type
check
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
Construction
data-flow
graphs
Compiler
(scalac, gcc)
optimizer
lexer
parser
type
check
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
Construction
data-flow
graphs
Compiler
(scalac, gcc)
optimizer
lexer
parser
type
check
code gen
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Pretprocesiranje i linkovanje

- Osnovne faze prevođenja, osim kompilacije, obuhvataju i pretprocesiranje i linkovanje
- Faza pretprocesiranja je pripremna faza kompilacije
- Faza linkovanja je neophodna faza kako bi se na osnovu proizvoda kompilacije napravio izvršivi program

Pretprocesiranje

- Kompilator ne obrađuje tekst programa koji je napisao programer, već tekst programa koji je nastao pretprocesiranjem
- Jedan od najvažnijih zadataka pretprocesora je da omogući da se izvorni kôd pogodno organizuje u više ulaznih datoteka.
- Pretprocesor izvorni kôd iz različitih datoteka objedinjava u tzv. *jedinice prevođenja* i prosleđuje ih kompilatoru.
- Na primer, u `.c` datoteke koje sadrže izvorni kôd uključuju se *datoteke zaglavlja*
- Rezultat rada pretprocesora, može se dobiti korišćenjem GCC prevodioca navođenjem opcije `-E` (na primer, `gcc -E program.c`). (Sa opcijom `-P` biće uklonjeni markeri linija i proizveden c fajl koji može da se direktno prevodi dalje.)

Pretprocesor

- Suštinski, pretprocesor vrši samo jednostavne operacije nad tekstualnim sadržajem programa i ne koristi nikakvo znanje o samom programskom jeziku. Pretprocesor analizira samo *pretprocesorske direktive*
- Primer pretprocesorskih direktiva su direktive u programskom jeziku C/C++ `#include` (za uključivanje sadržaja neke druge datoteke) i `#define` koja zamenjuje neki tekst, *makro*, drugim tekstom.

Primer — pretprocesiranje

```
#Include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     printf("9");
  return 0;
}
```

Primer — pretprocesiranje

```
#Include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     printf("9");
  return 0;
}
```

```
primer.c:1:2: error: invalid preprocessing
                     directive #Include
```

Povezivanje

- Povezivanje je proces kreiranja jedinstvene izvršne datoteke od jednog ili više objektnih modula koji su nastali ili kompilacijom izvornog kôda programa ili su objektni moduli koji sadrže mašinski kôd i podatke standardne ili neke nestandardne biblioteke
- Pored *statičkog povezivanja*, koje se vrši nakon kompilacije, postoji i *dinamičko povezivanje*, koje se vrši tokom izvršavanja programa (zapravo na njegovom početku).
- Opcijama kompajlera, kompilaciju i povezivanje je moguće razdvojiti

Primer — povezivanje

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     print("9");
  return 0;
}
```

Primer — povezivanje

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a == 9)
     print("9");
  return 0;
}
```

```
primer.c:(.text+0x20): undefined reference to ‘print’
collect2: ld returned 1 exit status
```

Pregled

1. Leksička analiza
   - Leksika
   - Leksička analiza
   - Regularni izrazi i konačni automati
   - Izazovi modernih leksera
2. Sintaksička analiza
3. Semantička analiza
4. Generisanje međukoda

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write

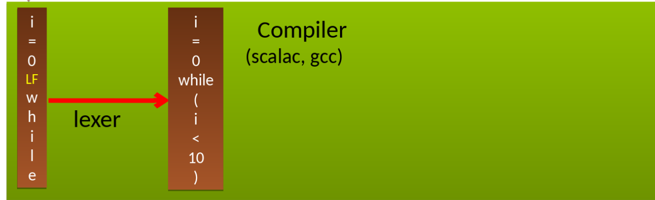
Compiler
(scalac, gcc)

i
=
0
LF
w
h
i
l
e

characters

machine code
(e.g. x86, ARM, JVM)
efficient to execute

```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
(scalac, gcc)
lexer
characters
words
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Leksika

- Programski jezici moraju da budu precizni
- Leksika je podoblast sintakse koja se bavi opisivanjem osnovnih gradivnih elemenata jezika.
- Programi se računaru zadaju nizom karaktera. Na primer, kod:

```
if (v < 120)
  s = s0 + v*t;
```

  je, zapravo, samo niz karaktera: `if_(v < 120)\n\ts = s0 + v*t;\n`.
- U okviru leksike, definišu se reči i njihove kategorije.
- U programskom jeziku, reči se nazivaju **lekseme**, a kategorije **tokeni**.

Leksika

- Dakle, pojedinačni karakteri se grupišu u nedeljive celine **lekseme** koje predstavljaju osnovne leksičke elemente, koji bi bili analogni rečima govornog jezika i pridružuju im se **tokeni** koji opisuju leksičke kategorije kojima te lekseme pripadaju.
- U prirodnom jeziku, kategorije su imenice, glagoli, pridevi
- U programskom jeziku, tokeni mogu da budu identifikatori, ključne reči, operatori...
- Na primer, za token `identifikator`, primeri leksema su `v`, `s`, `s0`, `t`

Primer

```
if (v < 120)
  s = s0 + v*t;
```

```
leksema   token
------------------------
if        kljucna rec
(         zagrada
v         identifikator
<         operator
120       celobrojni literal
)         zagrada
s         identifikator
=         operator
s0        identifikator
+         operator
v         identifikator
*         operator
t         identifikator
;         interpunkcija
```

Leksika

- Neki tokeni sadrže samo jednu reč, a neki mogu sadržati puno različitih reči
- Programski jezik C sadrži više od 100 različitih tokena: 44 različite ključne reči, identifikatore, celobrojne vrednosti, realne vrednosti, karakterske konstante, stringovske literale, dve vrste komentara, 54 operatora...
- Drugi moderni programski jezici imaj sličan nivo kompleksnosti tokena

Leksička analiza

- **Leksička analiza je proces izdvajanja leksema i tokena**, osnovnih jezičkih elemenata, iz niza ulaznih karaktera.
- Leksičku analizu vrše moduli kompilatora koji se nazivaju **leksički analizatori** (lekseri, skeneri). Oni obično prosleđuju spisak izdvojenih leksema (i tokena kojima pripadaju) drugom modulu (sintaksičkom analizatoru) koji nastavlja analizu teksta programa.
- Leksički analizator najčešće radi na zahtev sintaksičkog analizatora tako što se sintaksički analizator obraća leksičkom analizatoru kada god mu zatreba naredni token.

Leksička analiza

- Tokenima mogu biti pridruženi i neki dodatni **atributi**.
  - Svakom tokenu `identifikator` može biti pridružen pokazivač na mesto u specijalnoj tabeli simbola koja sadrži informacije o pojedinačnim identifikatorima (npr. ime identifikatora, njegov tip, lokaciju u kodu na kome je deklarisan i slično).
  - Svakom tokenu `brojevna_konstanta` može biti pridružena njena numerička vrednost i slično.
  - ...

Regularni izrazi i konačni automati

- Token, kao skup svih mogućih leksema, opisuje se formalno pogodnim obrascima koji mogu da uključuju cifre, slova, specijalne simbole i slično.
- Ti obrasci za opisivanje tokena su obično **regularni izrazi**, a mehanizam za izdvajanje leksema iz ulaznog teksta zasniva se na **konačnim automatima**.
- Generisanje je bitno programerima, a prepoznavanje kompajlerima

Regularni izrazi

- Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su:
  - **karakterske klase** - navode se između [i] i označavaju jedan od navedenih karaktera. Na primer, klasa [0-9] označava cifru.
  - **alternacija** - navodi se sa | i označava alternativne mogućnosti. Na primer, a|b označava slovo a ili slovo b.
  - **opciono jedinstveno pojavljivanje** - navodi se sa ?. Npr. a? označava da slovo a može, a ne mora da se javi.
  - **opciono višestruko ponavljanje** - navodi se sa * i označava da se nešto javlja nula ili više puta. Npr. a* označava niz od nula ili više slova a.
  - **pozitivno ponavljanje** - navodi se sa + i označava da se nešto javlja jedan ili više puta.

  Npr. [0-9]+ označava neprazan niz cifara.

Regularni izrazi — primer

- Razmotrimo identifikatore u programskom jeziku C. Govornim jezikom, identifikatore je moguće opisati kao "neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom". Ovo znači da je prvi karakter identifikatora ili slovo ili podvlaka, za čim sledi nula ili više slova, cifara ili podvlaka.
- Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:
  `([a-zA-Z]|_)([a-zA-Z]|_|[0-9])*`
  Ovaj izraz je moguće zapisati još jednostavnije kao:
  `[a-zA-Z_][a-zA-Z_0-9]*`

Lex

- Lex je program koji generiše leksere na programskom jeziku C, a na osnovu zadatog opisa tokena u obliku regularnih izraza. Na primer, jedan jednostavan opis leksičkog analizatora koji prepoznaje identifikatore, realne konstante, operatore i zagrade može biti sledeći:

```
%%
[a-zA-Z_][a-zA-Z_0-9]*          return IDENTIFIKATOR;
[+-]?([0-9]*[.])?[0-9]+         return REALNA_KONSTANTA;
[+-]?0[1-7][0-7]*               return OKTALNA_KONSTANTA;
[-+*/()]                        return *yytext;
[ \t\n]+                        /* beline se preskaču */
%%
```

Primer — kompilacija — leksička analiza

```
#include<stdio.h>
int main() {
  int a = 09;
  if (a == 9)
     printf("9");
  return 0;
}
```

Primer — kompilacija — leksička analiza

```
#include<stdio.h>
int main() {
  int a = 09;
  if (a == 9)
     printf("9");
  return 0;
}
```

```
primer.c:4:9: error: invalid digit "9" in octal constant
```

Ključne reči i identifikatori

- Razlikujemo ključne reči i identifikatore: identifikator ne može biti neka od ključnih reči, npr ne možemo da napravimo promenljivu koja bi se zvala `if` ili `while`
- Postoje ključne reči koje zavise od konteksta, engl. *contextual keywords* koje su ključne reči na određenim specifičnim mestima programa, ali mogu biti identifikatori na drugim mestima.
- Na primer, u C# -u reč `yield` može da se pojavi ispred `break` ili `return`, na mestima na kojima identifikator ne može da se pojavi. Na tim mestima, ona se interpretira kao ključna reč, ali može da se koristi na drugim mestima i kao identifikator.

Ključne reči i identifikatori

- C# 4.0 ima 26 takvih konteksno zavisnih ključnih reči, C++11 ih takođe ima dosta.
- Većina je uvedena revizijom postojećeg jezika sa ciljem da se definiše novi standard: sa velikim brojem korisnika i napisanog koda, vrlo je verovatno da se neka reč već koristi kao identifikator u nekom postojećem programu.

Ključne reči i identifikatori

- Uvođenjem kontekstualno zavisnih ključnih reči, smanjuje se rizik da se postojeći program neće kompilirati sa novom verzijom standarda.
- S druge strane, ovo komplikuje i dodaje nove koncepte u rad leksera

Pregled

1. Leksička analiza

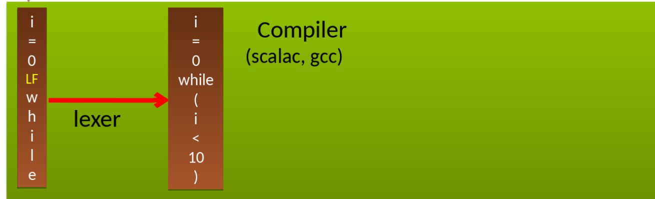
2. Sintaksička analiza
   - Apstraktno sintaksno stablo
   - Gramatike
   - Primeri
   - Potisni automati
   - BNF, EBNF, sintaksički dijagrami

3. Semantička analiza

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write



characters

words

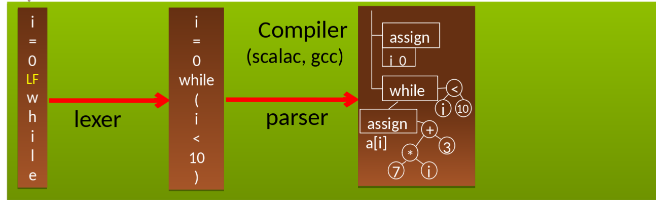
```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, ARM, JVM)
efficient to execute

Compiler Construction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write



characters words trees

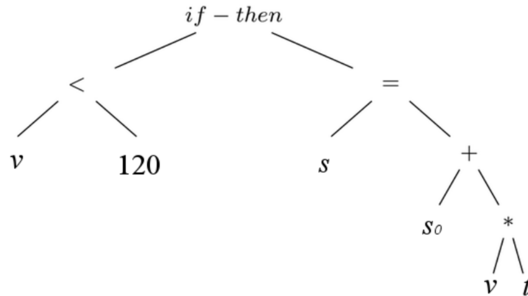
machine code
(e.g. x86, ARM, JVM)
efficient to execute

```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

Sintaksička analiza

- Sintaksa definiše **formalne relacije između elemenata jezika**, time pružajući strukturne opise ispravnih niski jezika.
- Sintaksa prirodnih jezika definiše načine na koji pojedinačne reči mogu da kreiraju ispravne rečenice jezika. Slično je i sa programskim jezicima, gde se umesto ispravnih rečenica **razmatraju ispravni programi**.
- Sintaksa se bavi samo **formom i strukturom jezika** bez bilo kakvih razmatranja u vezi sa njihovim značenjem.
- Sintaksička struktura rečenica ili programa se može predstaviti **u obliku stabla** koje predstavlja međusobni odnos pojedinačnih tokena.

Primer



Sintaksička analiza

- Sintaksom programa obično se bavi deo programskog prevodioca koji se naziva **sintaksički analizator ili parser**.
- Rezultat njegovog rada se isporučuje daljim fazama u obliku sintaksičkog stabla (**stablo apstraktne sintakse** i **apstraktno sintaksičko stablo**).

Sintaksička analiza

- Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (npr. skup svih identifikatora, skup brojevnih literala, i slično).
- Međutim nije moguće konstruisati regularne izraze kojim bi se opisale konstrukcije koje se javljaju u programskim jezicima. Na primer, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetički izrazi.
- Zbog toga su nam potrebne gramatike.

Gramatike

- **Sintaksa jezika se obično opisuje gramatikama.**
- U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći govorni jezik kao metajezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi.
- Za opis sintaksičkih konstrukcija programskih jezika koriste se uglavnom **kontekstno-slobodne gramatike** (engl. *context free grammars*).

Kontekstno slobodne gramatike vs regularni izrazi

- Kontekstno-slobodne gramatike su izražajniji formalizam od regularnih izraza.
- Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno-slobodnim gramatikama, tako da je kontekstno-slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (ali regularni izrazi obično daju koncizniji opis).

Kontekstno slobodne gramatike

- **Kontekstno-slobodne gramatike su određene skupom pravila.** Svako pravilo ima levu i desnu stranu.
- Sa leve strane pravila nalaze se tzv. **pomoćni simboli (neterminali)**, dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. **završni simboli (terminali)**.
- Svakom pomoćnom simbolu pridružena je neka sintaksička kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se **početnim simbolom** (ili aksiomom).
- **Niska je opisana gramatikom ako ju je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.**

Gramatike

- Kontekstno-slobodne gramatike čine samo jednu specijalnu vrstu formalnih gramatika.
- U kontekstno-slobodnim gramatikama sa leve strane pravila nalazi se uvek tačno jedan neterminalni simbol, a u opštem slučaju sa leve strane pravila može se nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

Kontekstno slobodne gramatike

- Pokazano je da je jezik identifikatora programskog jezika C regularan i da ga je moguće opisati regularnim izrazom. Sa druge strane, isti ovaj jezik je moguće opisati i formalnom gramatikom:

$$
\begin{array}{rcl}
I & \rightarrow & XZ \\
X & \rightarrow & S \mid P \\
Z & \rightarrow & YZ \mid \varepsilon \\
Y & \rightarrow & S \mid P \mid C \\
S & \rightarrow & \mathtt{a} \mid \ldots \mid \mathtt{z} \mid \mathtt{A} \mid \ldots \mid \mathtt{Z} \\
P & \rightarrow & _ \\
C & \rightarrow & \mathtt{0} \mid \ldots \mid \mathtt{9}
\end{array}
$$

- Neterminal S odgovara slovima, P podvlaci, C ciframa, X slovu ili podvlaci, Y slovu, podvlaci ili cifri, a Z nizu simbola koji se mogu izvesti iz Y tj. nizu slova, podvlaka ili cifara. ϵ označava praznu reč.

Primer 1

$$\begin{array}{lcl} I & \rightarrow & XZ \\ X & \rightarrow & S \mid P \\ Z & \rightarrow & YZ \mid \varepsilon \\ Y & \rightarrow & S \mid P \mid C \\ S & \rightarrow & \mathtt{a} \mid \ldots \mid \mathtt{z} \mid \mathtt{A} \mid \ldots \mid \mathtt{Z} \\ P & \rightarrow & _ \\ C & \rightarrow & 0 \mid \ldots \mid 9 \end{array}$$

Identifikator `x_1` je moguće izvesti na sledeći način:

$$I \Rightarrow XZ \Rightarrow SZ \Rightarrow xZ \Rightarrow xYZ \Rightarrow xPZ \Rightarrow$$

$$x_Z \Rightarrow x_YZ \Rightarrow x_CZ \Rightarrow x_1Z \Rightarrow x_1$$

Primer 2

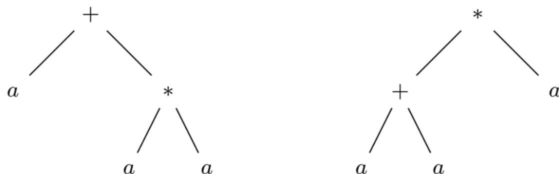
$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & a.
\end{array}
$$

Izraz $a + a * a$ se može izvesti na dva načina:

(1) $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$

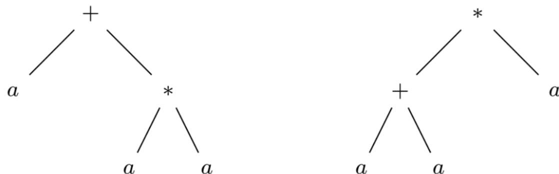
(2) $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$

Primer 2



Dva drveta koja odgovaraju različitim izvođenjima (različitom prioritetu operatora). Prvo izvođenje odgovara levom stablu, a drugo desnom stablu.

Primer 2



Problem sa prethdnom gramatikom je to što se u njoj ne oslikava prioritet (niti asocijativnost) operatora i stoga se takve gramatike ne mogu direktno koristiti u sintaksičkoj analizi (pre njihovog korišćenja potrebno je na neki način precizirati prioritet i asocijativnost operatora).

Primer 3

$$
\begin{array}{lcl}
E & \rightarrow & E+T \\
 & | & T \\
T & \rightarrow & T*F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & a
\end{array}
$$

Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

Primetimo da je ova gramatika u određenom smislu preciznija od prethodne gramatike, s obzirom da je njome jednoznačno određen prioritet i asocijativnost operatora.
Izraz $a + a * a$ se sada može izvesti samo na jedan način:
$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$

Kontekstno-slobodne gramatike

- Kontekstno-slobodne gramatike se obično formiraju nad tokenima iz leksičke analize.
- Tako bi pravila za F obuhvatala promenljive (npr. opisane tokenom id) i brojevne konstante (npr. opisane tokenom num).
- Za ulaznu nisku `s0 + v * t` leksički analizator bi prepoznao tokene `id + id * id` i to bi bilo prepoznato kao ispravan arimtetički izraz na osnovu gore navedene gramatike.

Kontekstno-slobodne gramatike i potisni automati

- Na osnovu gramatike jezika formira se **potisni automat** na osnovu kojeg se jednostavno implementira program koji vrši sintaksičku analizu.
- Formiranje automata od gramatike se obično vrši automatski, uz korišćenje tzv. generatora parsera (engl. *parser generator*), poput sistema Yacc, Bison ili Antlr.

BNF, EBNF, sintaksički dijagrami

- Za opis sintakse koriste se i određene varijacije kontekstno-slobodnih gramatika.
- Najpoznatije od njih su BNF (Bakus-Naurova forma), EBNF (proširena Bakus-Naurova forma) i sintaksički dijagrami.
- BNF je pogodna notacija za zapis kontekstno-slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksički dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse.

BNF (Bakus Naurova forma) primer

Uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksičkih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objektnom jeziku.

Jezik celih brojeva u dekadnom brojnom sistemu:

```
<ceo broj> ::= <neoznacen ceo broj>
             | <znak broja> <neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra>
             | <neoznacen ceo broj><cifra>
<cifra> ::= 0|1|2|3|4|5|6|7|8|9
<znak broja> ::= +|-
```

EBNF (proširena Bakus-Naurova forma)

- Vitičaste zagrade {...} okružuju elemente izraza koji se mogu ponavljati proizvoljan broj (nula ili više) puta. Alternativno, moguće je korišćenje i sufiksa *.
- Uglaste zagrade [...] okružuju opcione elemente u izrazima, tj. elemente koji se mogu pojaviti nula ili jedan put. Alternativno, moguće je korišćenje i sufiksa ?.
- Sufiks + označava elemente izraza koji se mogu pojaviti jednom ili više puta.
- Obične male zagrade se koriste za grupisanje.
- Ponekad se usvaja konvencija da se terminali od jednog karaktera okružuju navodnicima " kako bi se razlikovali od metasimbola EBNF.

EBNF

Identifikatori:

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> }
<slovo ili _> ::= "a"|...|"z"|"A"|...|"Z"|"_"
<cifra> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Ceo broj:

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>}
<cifra> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

EBNF

Naredba grananja if sa opcionim pojavljivanjem else grane može se opisati sa:

```
<if_naredba> ::= if "(" <bulovski_izraz> ")"
                   <niz_naredbi>
               [ else
                   <niz_naredbi> ]
<niz_naredbi> ::= "{" <naredba> ";" { <naredba> ";" } "}"
```

Primer — kompilacija — sintaksna analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if a == 9
     printf("9");
  return 0;
}
```

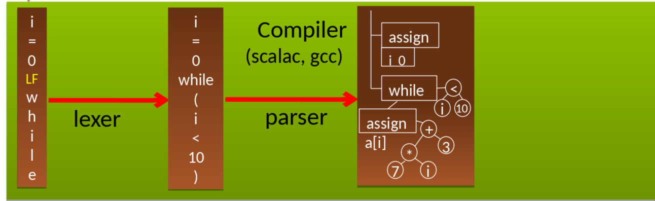
Primer — kompilacija — sintaksna analiza

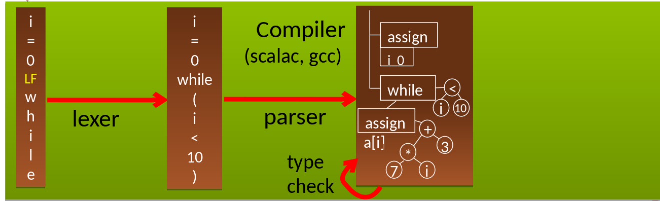
```
#include<stdio.h>
int main() {
  int a = 9;
  if a == 9
     printf("9");
  return 0;
}
```

```
primer.c:5:6: error: expected ‘(’ before ‘a’
```

Pregled

1. Leksička analiza
2. Sintaksička analiza
3. Semantička analiza
   - Formalna i neformalna semantika
   - Upozorenja i različiti tipovi semantičke analize
   - Provera tipova
4. Generisanje međukoda


Compiler
Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
(scalac, gcc)
lexer
parser
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Compiler
(scalac, gcc)
parser
type check
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Semantika

- Semantika pridružuje značenje ispravnim konstrukcijama na nekom programskom jeziku.
- Semantika programskog jezika određuje značenje jezika.
- Obično je značajno teže definisati nego sintaksu.

Semantika

- Semantika može da se opiše formalno i neformalno, često se zadaje samo neformalno.
- Uloga neformalne semantike je da programer može da razume kako se program izvršava pre njegovog pokretanja.
- Na primer, semantika naredbe `if(a<b) a++;` neformalno se opisuje sa „ukoliko je vrednost promenljive a manja od vrednosti promenljive b, onda uvećaj vrednost promenljive a za jedan”

Formalna semantika

- Formalno zadavanje semantike je značajno teže od zadavanja leksike i sintakse i za to postoje različiti formalizmi koji su pogodni za različite potrebe
- Operaciona semantika, denotaciona semantika, aksiomatska semantika
- Formalne semantike koriste se za izgradnju alata koji se koriste za naprednu semantičku analizu softvera
- Ovi alati se mogu koristiti kao dopuna semantičkoj analizi koju sprovode kompajleri

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Upotreba nedefinisanog simbola, npr korišćenje nedeklarisane promenljive

```
int main() {
  int a = 5, b = 7;
  c = a + b;
}
```

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Upotreba nedefinisanog simbola, npr korišćenje nedeklarisane promenljive

```
int main() {
  int a = 5, b = 7;
  c = a + b;
}
```

```
semantika.c: In function ‘main’:
semantika.c:3:3: error: ‘c’ undeclared (first use in this function)
    3 |   c = a + b;
      |   ^
```

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Upotreba nedefinisanog simbola, npr korišćenje nedefinisane funkcije ili nedefinisane klase

```
int main() {
  int a = f(7);
}
```

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Upotreba nedefinisanog simbola, npr korišćenje nedefinisane funkcije ili nedefinisane klase

```
int main() {
  int a = f(7);
}
```

```
semantika.c: In function ‘main’:
semantika.c:2:11: warning: implicit declaration of function ‘f’ [-Wimplicit-function-declaratio
    2 |   int a = f(7);
      |           ^
/usr/bin/ld: /tmp/ccXR19yq.o: in function ‘main’:
semantika.c:(.text+0x17): undefined reference to ‘f’
collect2: error: ld returned 1 exit status
```

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Simboli definisani više puta u istom dosegu, npr dva puta deklarisana promenljiva, klasa ili metod u klasi sa istim imenom (za metod i sa istim potpisom)

```
int main() {
  int a = 5, b = 7;
  int a = 0;
}
```

Semantičke greške

Semantičke greške se otkriju nakon leksičke i sintaksne analize. Primeri:

- Simboli definisani više puta u istom dosegu, npr dva puta deklarisana promenljiva, klasa ili metod u klasi sa istim imenom (za metod i sa istim potpisom)

```
int main() {
  int a = 5, b = 7;
  int a = 0;
}
```

```
semantika.c: In function ‘main’:
semantika.c:3:7: error: redefinition of ‘a’
    3 |   int a = 0;
      |       ^
semantika.c:2:7: note: previous definition of ‘a’ was here
    2 |   int a = 5, b = 7;
      |       ^
```

Semantičke greške

- Greške u tipovima
  - Indeksiranje skalara

    ```
    int a;
    a[5] = 42;
    ```

  - Indeksiranje indeksom koji nije celobrojnog tipa

    ```
    float i;
    ...
    a[i] = 42;
    ```

  - U nekim jezicima `if` naredba zahteva `boolean` vrednost

    ```
    if (42) ....
    ```

Primer — greška

```
int main() {
  int a;
  a[5] = 45;
  return a;
}
```

Primer — greška

```
int main() {
  int a;
  a[5] = 45;
  return a;
}
```

```
semantika.c: In function ‘main’:
semantika.c:4:4: error: subscripted value is neither array nor pointer nor vector
    4 |   a[5] = 45;
      |    ^
```

Primer — greška

```
int main() {
  int a[5];
  float i = 2.0;
  a[i] = 3;
}
```

Primer — greška

```
int main() {
  int a[5];
  float i = 2.0;
  a[i] = 3;
}
```

```
semantika.c: In function ‘main’:
hello.c:4:4: error: array subscript is not an integer
    4 |   a[i] = 3;
      |    ^
```

Primer — greška

Nepodržana automatska konverzija između dva različita tipa

```
int main() {
  double a;
  char* c = "0";
  a = c;
}
```

Primer — greška

Nepodržana automatska konverzija između dva različita tipa

```
int main() {
  double a;
  char* c = "0";
  a = c;
}
```

```
semantika.c: In function ‘main’:
hello.c:4:7: error: incompatible types when assigning to type ‘double’ from type ‘char *’
    4 |   a = c;
      |       ^
```

Primer — upozorenje

Neželjena automatska konverzija između dva različita tipa

```
int main() {
  int a;
  char* c = "0";
  a = c;
}
```

Primer — upozorenje

Neželjena automatska konverzija između dva različita tipa

```
int main() {
  int a;
  char* c = "0";
  a = c;
}
```

```
semantika.c: In function ‘main’:
hello.c:4:5: warning: assignment to ‘int’ from ‘char *’ makes integer from pointer without a cast [-W
    4 |   a = c;
      |     ^
```

Semantičke greške

- Greške u pozivima funkcija, prosleđivanju parametara
  - U pozivu funkcije f(...), identifikator f mora da je deklarisan kao funkcija
  - U pozivu funkcije, broj argumenata treba da se poklapa sa brojem argumenata u definiciji funkcije. Tipovi arugmenata takođe moraju da se poklapaju, ili da mogu da se konvertuju.
  - U okviru tela funkcije, svako return mora da vraća vrednost čiji se tip poklapa ili može da se konvertuje u tip povratne vrednosti funkcije

Primer

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Primer

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

Semantika

- Postoje različite vrste semantičkih provera, važenje nekih semantičkih pravila je moguće utvrditi u vreme kompilacije, dok za neke to nije slučaj. Primeri:
  - Semantika izračunavanja vrednosti izraza zahteva da su vrednosti izraza odgovarajućeg tipa (npr aritmetički izrazi imaju smisla nad numeričkim vrednostima, množenje dva stringa obično nije semantički definisano) — ova provera može se izvršiti u fazi kompilacije
  - Semantika deljenja celobrojnih vrednosti zahteva da je delilac različit od nule — u vreme kompilacije često nije moguće utvrditi da li će delilac uvek biti različit od nule
  - Semantika pristupa elementu niza zahteva da je indeks u granicama rezervisane memorije za niz — takođe neodlučiv problem

Validnost vs korektnost

```
int main() {
    string x;
    if (false) {
        x = 137;
    }
}
```

Validnost vs korektnost

```
int main() {
    string x;
    if (false) {
        x = 137;
    }
}
```

Safe; can't happen

Validnost vs korektnost

```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

Validnost vs korektnost

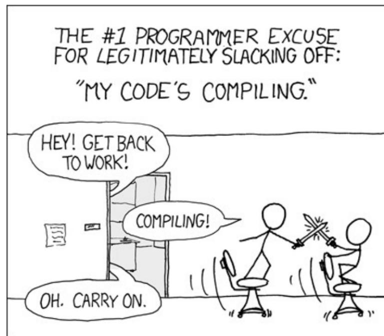
```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int main() {
    Print(Fibonacci(40));
}
```

Incorrect,
should be
"return n;"

Izazovi semantičke analize

- Odbaciti što više nekorektnih programa
- Prihvatiti što više korektnih programa
- Uraditi to brzo

Efikasnost kompajlera



Semantička upozorenja

- Rezultati semantičkih provera često se prijavljuju programeru kao upozorenja
- Mnoge semantičke provere i odgovarajuća upozorenja nastaju kao rezultat iskustva
- Na primer, ako se uvede promenljiva koja se ne koristi, najverovatnije je da je u pitanju neka greška i da to programer nije uradio namerno
- Naredba dodele umesto naredbe poređenja, često je greška koju je teško uočiti
- Ako klasa ima virtuelne metode, očekivano bi bilo da ima i virtuelan destruktor
- ...

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (9 == 9)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (9 == 9)
     printf("9");
  return 0;
}
```

```
primer.c:3: warning: unused variable ‘a’
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a = 9)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if (a = 9)
     printf("9");
  return 0;
}
```

```
primer.c:4: warning: suggest parentheses around assignment
            used as truth value [-Wparentheses]
   if (a = 9)
       ^
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  if ((a = 9))
     printf("9");
  return 0;
}
```

Eksplicirana namera eliminise upozorenje

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;

  if (a / 0)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;

  if (a / 0)
     printf("9");
  return 0;
}
```

```
primer.c:4: warning: division by zero
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  int b = 0;
  if (a / b)
     printf("9");
  return 0;
}
```

Primer — upozorenja

```
#include<stdio.h>
int main() {
  int a = 9;
  int b = 0;
  if (a / b)
     printf("9");
  return 0;
}
```

Nema upozorenja!

gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)

Semantička upozorenja

- Semantička upozorenja su često nepotpuna (kao u prethodnom primeru)
- Na izbor semantičkih provera koje kompajler implementira utiče pre svega efikasnost analize — kompajler mora da radi efikasno i kompleksna semantička analiza nije poželjna jer usporava proces kompilacije. Zbog toga se kompleksnije semantičke analize obično izdvajaju u posebne alate, ili ako se već implementiraju, ne uključuju se podrazumevano.

Semantička upozorenja

- Na izbor semantičkih provera koje kompajler sprovodi na nekom projektu utiče pre svega domen projekta
- Što je ispravnost aplikacije kritičnija i važnija, to je semantička analiza detaljnija i sveobuhvatnija, i pritom je potrebno napisati kôd koji prilikom prevođenja nema upozorenja, tj greške i upozorenja se tretiraju na isti način
- Najčešće se izbor semantičkih provera može kontrolisati opcijama kompajlera *Options to Request or Suppress Warnings*

Opcije uključivanja/isključivanja upozorenja

- gcc: `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`
- clang: `https://clang.llvm.org/docs/DiagnosticsReference.html`
- Možete da zahtevate konkretne analize koje daju različita upozorenja. To se kontroliše opcijama koje počinju sa '`-W`'.
- Na primer, opcija `-Wimplicit` zahteva upozorenja vezana za implicitne deklaracije
- Svaka od ovih opcija ima i negativnu varijantu koja omogćava njeno isključivanje. Negativna varijnata počinje sa '`-Wno-`'
- Na primer, `-Wno-implicit`.

Grupne opcije

- Neke opcije, npr `-Wall` i `-Wextra`, uključuju druge opcije ili uključuju grupe drugih opcija, na primer, `-Wunused`, koja uključuje opcije kao što je `-Wunused-value`.
- Kada postoji više opcija, onda specifične opcije imaju prednost u odnosu na one koje su manje specifične, nezavisno od mesta opcije u komandnoj liniji.
- Na primer, ako jedna opcija uključuje celu grupu opcija, a druga opcija isključuje neku konkretnu opciju iz te grupe, onda će ovo isključivanje imati efekta jer je u pitanju specifičnija opcija.
- Ako su opcije iste specifičnosti, onda se računa efekat poslednje.
- Na primer, ako uključite pa isključite istu opciju, računa se da je ona isključena.

Grupne opcije

- Opcija `-Wall` — This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning).
- Opcija `-Wextra` — This enables some extra warning flags that are not enabled by -Wall.
- Opcija `-Wpedantic` — Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++.

Primeri

- Opcija `-Wmain` — Warn if the type of `main` is suspicious. `main` should be a function with external linkage, returning `int`, taking either zero arguments, two, or three arguments of appropriate types. This warning is enabled by default in `C++` and is enabled by either `-Wall` or `-Wpedantic`.
- Opcija `-Wunused` — Some of the diagnostics controlled by this flag are enabled by default. Controls `-Wunused-argument`, `-Wunused-function`, `-Wunused-label`, `-Wunused-lambda-capture`, `-Wunused-local-typedef`, `-Wunused-private-field`, `-Wunused-property-ivar`, `-Wunused-value`, `-Wunused-variable`.
- Opcija `-Werror` — Make all warnings into errors.
- Opcija `-Werror=` — Make the specified warning into an error.

Implementacija semantičke analize

- Jednostavna semantička analiza zasniva se na ispitivanju karakteristika apstraktnog sintaksnog stabla
- Za dodavanje jednostavnih semantičkih provera, u okviru clang-a postoji čak tri interfejsa, dva koja su sastavni deo kompajlera, i treći koji je nezavisan alat ali u sklopu clang projekta
- Kompleksnija semantička analiza uključuje i analizu koda i grafa kontrole toka
- Za dodavanje kompleksnijih semantičkih analiza, u okviru samog clang-a ne postoji direktan interfejs (postoji samo interfejs za obilazak naredbi u okviru grafa kontrole toka koji se može iskoristiti u te svrhe), ali postoji nezavisan alat koji je u sklopu clang projekta.

Provera tipova

- Jedan od osnovnih zadataka semantičke analize je provera tipova (engl. *typechecking*).
- Tokom provere tipova proverava se da li je svaka operacija primenjena na operande odgovarajućeg tipa (npr. indeksni pristup nizu obično zahteva da je indeks celobrojnog tipa i ako se pokuša pristup elementu niza sa navedenim indeksom koji je realnog tipa, prijavljuje se greška).

Primer — greška

```
int main() {
  int a[10];
  float i = 0.0;
  a[i] = 42;
  return 0;
}
```

Primer — greška

```
int main() {
  int a[10];
  float i = 0.0;
  a[i] = 42;
  return 0;
}
```

```
semantika.c: In function ‘main’:
semantika.c:5:4: error: array subscript is not an integer
    5 |   a[i] = 42;
      |     ^
```

Provera tipova

- U zavisnosti od jezika se u nekim slučajevima u sintaksičko drvo umeću implicitne konverzije, gde je potrebno (slabo tipizirani jezici) ili se prijavljuje greška (strogo tipizirani jezici).

Semantika

Razmatrajmo naredni fragment C koda.

```
float x, y;
x = 2 * y;
```

Semantički analizator prikuplja informacije o tipovima pojedinačnih promenljivih (obični ih skladišti u tablicu simbola). Nakon toga, proveravaju se tipovi operanada i pošto se zaključuje da se množe ceo broj i broj u pokretnom zarezu, ceo broj se konvertuje u realan.

Semantika

- Najbolji način da se to uradi je da se u sintaksičkom drvetu čvor koji predstavlja celobrojnu konstantu 2 zameni čvorom koji predstavlja konstantu 2 zapisanu u realnom zarezu.
- Drugi način je da se u drvo umetne nov čvor koji predstavlja operaciju konverzije vrednosti iz celobrojnog u realni tip i da se prilikom generisanja koda na osnovu tog čvora generiše instrukcija koja tu konverziju vrši.
- U slučaju kada je potrebno izvršiti konverziju tipa promenljive, to će se skoro neizbežno desiti, a u slučaju da se ispod tog čvora nađe konstantna vrednost veoma verovatno je da će se konverzija izvršiti tokom faze optimizacije.

Primer — kompilacija — semantička analiza

```
#include<stdio.h>
int main() {
  int a = 9;
  if ("a" * 9)
     printf("9");
  return 0;
}
```

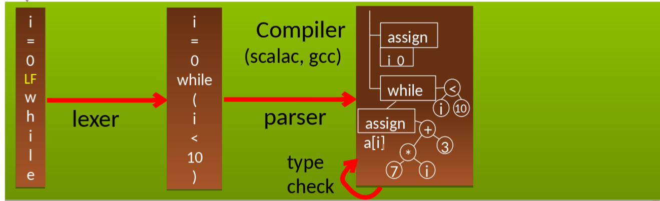
Primer — kompilacija — semantička analiza

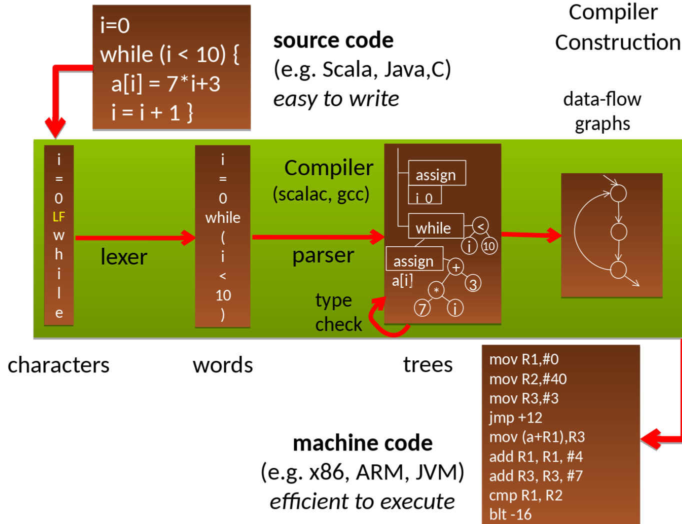
```
#include<stdio.h>
int main() {
  int a = 9;
  if ("a" * 9)
     printf("9");
  return 0;
}
```

```
primer.c:5:8: error: invalid operands to binary *
                     (have ‘char *’ and ‘int’)
```

Pregled

1. Leksička analiza
2. Sintaksička analiza
3. Semantička analiza
4. Generisanje međukoda
   - Uloga međukoda
   - Troadresni kôd
   - Primer: gcc


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
type check
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
data-flow graphs
Compiler
(scalac, gcc)
lexer
parser
type check
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Generisanje međukoda

- Većina kompilatora prevodi sintaksičko stablo provereno i dopunjeno tokom semantičke analize u određeni međukod (engl. *intermediate code* ili *intermediate representation*), koji se onda dalje analizira i optimizuje i na osnovu koga se u kasnijim fazama gradi rezultujući asemblerski i mašinski kod.

Čemu služi ova međureprezentacija?

- Pojednostavljivanje optimizacija
  - Mašinski kod ima razna ograničenja koje ometaju optimizaciju
  - Rad sa međureprezentacijom čini da su optimizacije lakše i čistije
- Da bi imali više prednjih delova za isti zadnji deo
  - gcc ima podršku za C, C++, Java, Fortran, Ada, i još neke druge jezike
  - Svaki front-end gcc-a prevodi izvorni kod u jezik koji se naziva *GENERIC*
- Da bi imali više zadnjih delova iz istog prednjeg dela
  - Poželjno je uraditi većinu optimizacija na međureprezentaciji pre nego što se emituje kôd za neku konkretnu ciljnu mašinu.

Generisanje međukoda

- Veoma je teško dizajnirati dobar IR jezik
- Potrebno je balansirati potrebe visokog jezika i potrebe jezika niskog nivoa mašine za koju je izvršavanje namenjeno
- Previsok nivo: nije moguće optimizovati neke implementacione detalje
- Prenizak nivo: nije moguće korisititi znanje visokog nivoa da se izvrše neke agresivne optimizacije
- Kompajleri često imaju više nego jednu međureprezentaciju

Generisanje međukoda

- U okviru gcc-a postoji više međureprezentacija
- U okviru llvm-a postoji više međureprezentacija, osnovni LLVM IR se vodi kao IR visokog nivoa `https://llvm.org/docs/LangRef.html`
- Slično, IRovi viskog nivoa su Java bytecode, CPython bytecode i Microsoft CIL.
- Zadržava strukturu jezika visokog nivoa
- Omogućava kompilaciju do ciljne mašine
- Omogućava JIT kompilaciju ili interpretaciju

Generisanje međukoda

- Postoje različiti oblici za ovu međureprezentaciju
  - Grafovska reprezentacija (such as variants of syntax trees and directed acyclic graphs)
  - Troadresna reprezentacija
  - Reprezentacija virtuelnih mašina (such as bytecodes and stack-machine code)
  - Linearna reprezentacija (such as postfix notation)

Generisanje međukoda

- Najčešći oblik međureprezentacije je tzv. troadresni kôd u kome se javljaju dodele promenljivama u kojima se sa desne strane dodele javlja najviše jedan operator.
- Naziv troadresni: u svakoj instrukciji se navode "adrese" najviše dva operanda i rezultata operacije.
- Naredbe kontrole toka (if, if-else, while, for, do-while) se uklanjaju i svode na uslovne i bezuslovne skokove (predstavljenih često u obliku naredbi goto).

```
if (v < 120)
  s = s0 + v*t;
```

```
ifFalse v < 120 goto L
t1 = v * t
s = s0 + t1
L:
```

Generisanje međukoda

Ako je, na primer, promenljiva t celobrojna, a s i v realne, onda bi se tokom semantičke analize u drvo ubacio čvor konverzije tipa i to bi se oslikalo i u međukodu.

```
ifFalse v < 120 goto L
t1 = int2float(t)
t2 = v * t1
s = s0 + t2
L:
```

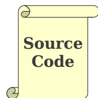
Generisanje međukoda

- Da bi se postigla troadresnost, u međukodu se vrednost svakog podizraza smešta u novouvedenu privremenu promenljivu (engl. *temporary*).
- U prethodnom primeru takve su t1 i t2.
- Njihov broj je potencijalno neograničen, a tokom faze generisanja koda (tj. registarske alokacije) svim promenljivama se dodeljuju fizičke lokacije gde se one skladište (to su ili registri procesora ili lokacije u glavnoj memoriji).

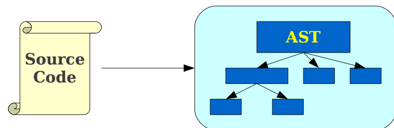
Generisanje međukoda

- GCC koristi tri osnovna jezika međureprezentacije koji predstavljaju program za vreme kompilacije: **GENERIC**, **GIMPLE** and **RTL**.
- Dodatno, GIMPLE može da se podeli na tri dela, visoki nivo, SSA (engl. *Static Single Assignment*) i niski nivo

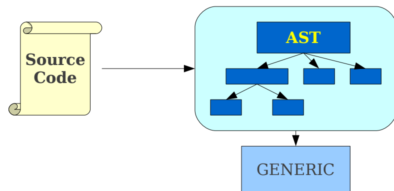
Architecture of gcc



Architecture of gcc



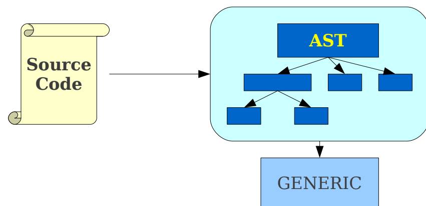
Architecture of gcc



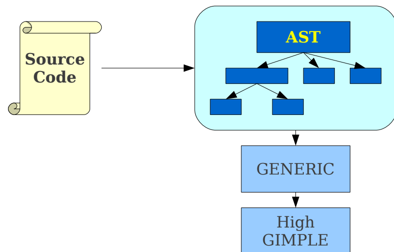
Primer: gcc

- GENERIC se koristi kao interfejs izmedju parsera i optimizatora.
- GENERIC je jezički nezavisna reprezentacija koju generiše svaki fornt-end. GENERIC je zajednička reprezentacija za sve jezike koje GCC podržava.
- Uloga GENERICa je da obezbedi način prikazivanja stabla funkcija u formatu koji je nezavisan od jezika.
- https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html

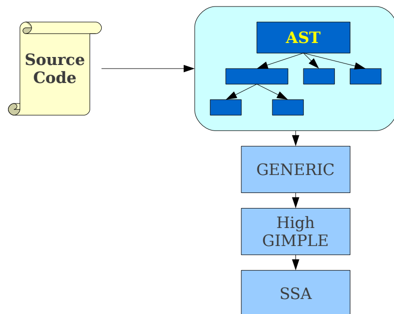
Architecture of gcc



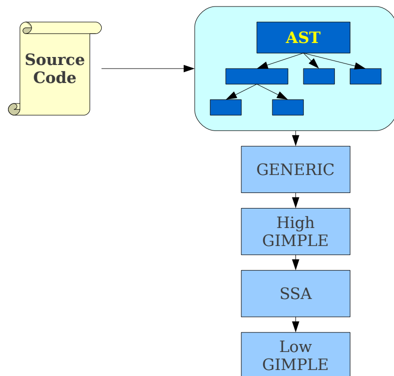
Architecture of gcc



Architecture of gcc



Architecture of gcc



Primer: gcc

- GIMPLE je troadresna reprezentacija koja je izvedena is GENERICa razbijanjem izraza u torke sa najviše tri operanda (osim u slučaju poziva funkcija)
- Ako želite da vidite kako izgleda GIMPLE za neki program, koristite opciju `-fdump-tree-gimple`.
- `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`

Primer: gcc

- The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable. Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. **The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created.** Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment.
- `https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html`

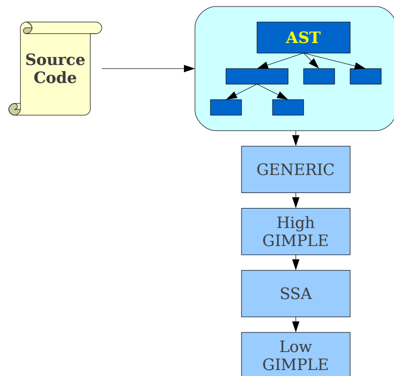
Primer: gcc

```
gcc -fdump-tree-gimple -fsyntax-only program.c
```

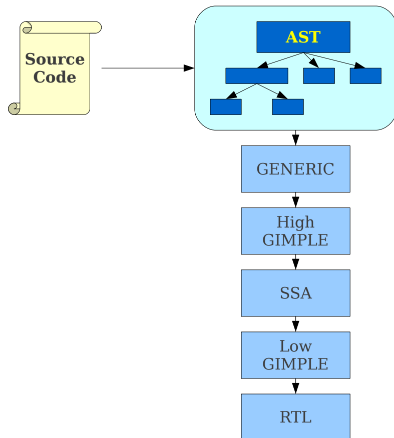
```
int function(int a, int b) {
  return (a+1)*b+1;
}
```

```
function (int a, int b)
{
  int D.1797;

  _1 = a + 1;
  _2 = b * _1;
  D.1797 = _2 + 1;
  return D.1797;
}
```

Architecture of gcc



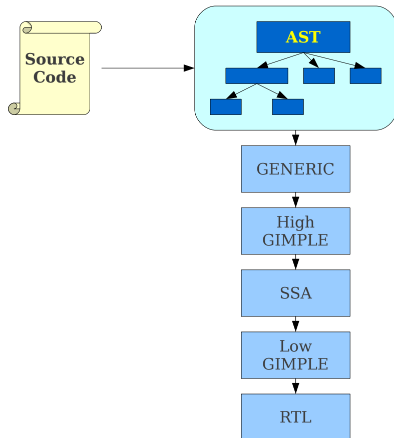
Architecture of gcc



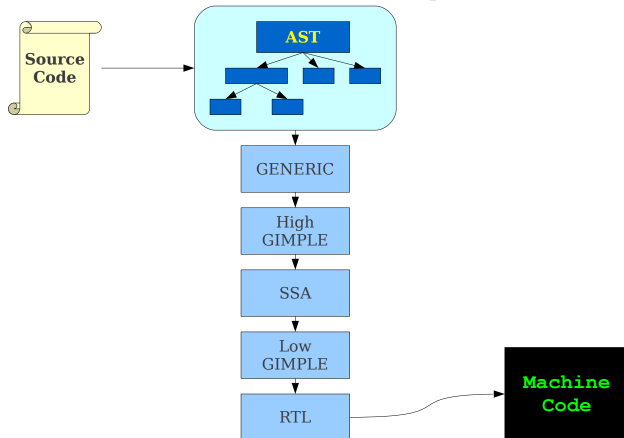
Primer: gcc

- The last part of the compiler work is done on a low-level intermediate representation called **Register Transfer Language**.
- GCC was designed to use RTL internally only. Correct RTL for a given program is very dependent on the particular target machine. And the RTL does not contain all the information about the program.
- RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.
- `https://gcc.gnu.org/onlinedocs/gccint/RTL.html`

Architecture of gcc



Architecture of gcc



Pregled

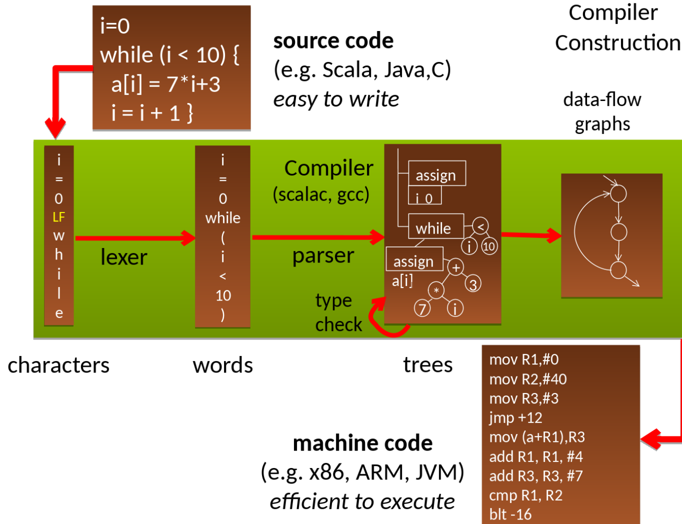
1 Leksička analiza

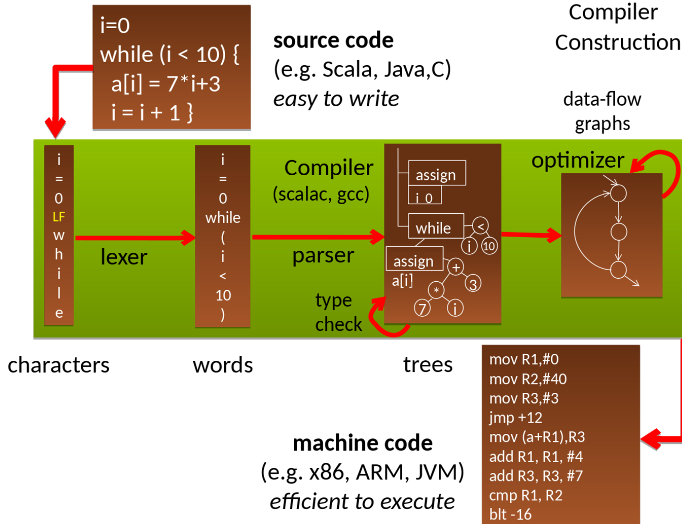
2 Sintaksička analiza

3 Semantička analiza

4 Generisanje međukoda

5 Optimizacija
- Constant folding
- Constant propagation


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
Construction
data-flow
graphs
Compiler
(scalac, gcc)
lexer
parser
type
check
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
Compiler
Construction
data-flow
graphs
i
=
0
LF
w
h
i
l
e
lexer
i
=
0
while
(
i
<
10
)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
type
check
optimizer
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Optimizacija

- Optimizacija podrazumeva poboljšanje performansi koda, zadržavajući pri tom ekvivalentnost sa polaznim (optimizovani kôd za iste ulaze mora da vrati iste izlaze kao i originalni kôd i mora da proizvede iste sporedne efekte - npr. ako originalni kôd nešto ispisuje na ekran ili skladišti na disk, to mora da uradi i optimizovani).
- Fazi optimizacije prethodi faza analize na osnovu koje se donose zaključci i sprovode optimizacije

Cilj

- Cilj: poboljšati IR koji je generisan u prethodnim koracima sa ciljem da se bolje iskoriste resursi.
- Jedan od najvažnijih i najkompleksnijih delova modernih kompajlera.
- Veoma aktivna oblast istraživanja.

Zašto je potrebna optimizacija?

- Da bismo optimizovali IR, moramo najpre da razumemo zašto je to potrebno.
- Prvi razlog: Generisanje IRa uvodi redundantnost.
  - Naivna translacija jezika višeg nivoa unosi u IR dodatna izračunavanja.
  - Ova izračunavanja često mogu da se ubrzaju, podele između različitih izračunavanja ili eliminišu u potpunosti.
- Drugi razlog: Programeri su lenji.
  - Na primer, izvršavanja u okviru petlje često može da se izvuče van petlje.

Optimizacija na osnovu generisanja IR-a

```
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

Optimizacija na osnovu generisanja IR-a

```
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizacija na osnovu generisanja IR-a

```
int x;
int y;
bool b1;
bool b2;
bool b3;
b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

b2 = _t0 == _t1;

b3 = _t0 < _t1;
```

Optimizacija petlje

```
while (x < y + z) {
 x = x - y;
}
```

```
_L0:
  _t0 = y + z;
  _t1 = x < _t0;
  IfZ _t1 Goto _L1;
  x = x - y;
  Goto _L0;
_L1:
```

Optimizacija petlje

```
while (x < y + z) {
 x = x - y;
}
```

```
_L0:
  _t0 = y + z;
  _t1 = x < _t0;
  IfZ _t1 Goto _L1;
  x = x - y;
  Goto _L0;
_L1:
```

```
  _t0 = y + z;
_L0:
  _t1 = x < _t0;
  IfZ _t1 Goto _L1;
  x = x - y;
  Goto _L0;
```

Optimizacija

- Optimizacija se najčešće odvija na dva nivoa:
  - optimizacija međukoda se generiše na početku faze sinteze i podrazumeva mašinski nezavisne optimizacije tj. optimizacije koje ne uzimaju u obzir specifičnosti ciljne arhitekture.
  - optimizacija ciljnog koda se izvršava na samom kraju sinteze i zasniva se na detaljnom poznavanju ciljne arhitekture i asemblerskog i mašinskog jezika na kome se izražava ciljni program.
- Slede primeri nekih poznatih i jednostavnih optimizacija

Constant folding

Konstantni izrazi se mogu izračunati (engl. *constant folding*).

```
x = 2 + 2
y = z * x
```

Constant folding

Konstantni izrazi se mogu izračunati (engl. *constant folding*).

```
x = 2 + 2          x = 4
y = z * x          y = z * x
```

Constant propagation

Izbegava se upotreba promenljivih čija je vrednost konstantna (engl. *constant propagation*).

```
x = 2 + 2        x = 4
y = z * x        y = z * x
```

Constant propagation

Izbegava se upotreba promenljivih čija je vrednost konstantna (engl. *constant propagation*).

```
x = 2 + 2          x = 4
y = z * x          y = z * x          y = z * 4
```

Strength reduction

Operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže (engl. *strength reduction*).

```
x = 2 + 2        x = 4
y = z * x        y = z * x        y = z * 4
```

Strength reduction

Operacije se zamenjuju onim za koje se očekuje da će se izvršiti brže (engl. *strength reduction*).

```
x = 2 + 2        x = 4
y = z * x        y = z * x        y = z * 4        y = z << 2
```

Common subexpression elimination

Izbegava se vršenje istog izračunavanja više puta (engl. *common subexpression elimination*).

```
a = x + y
b = x - y
c = x + y
d = a + c
```

Common subexpression elimination

Izbegava se vršenje istog izračunavanja više puta (engl. *common subexpression elimination*).

```
a = x + y          a = x + y
b = x - y          b = x - y
c = x + y          c = a
d = a + c          d = a + c
```

Copy propagation

Izbegava se uvođenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih (engl. *copy propagation*).

```
a = x + y
b = x - y
c = x + y
d = a + c
```

```
a = x + y
b = x - y
c = a
d = a + c
```

Copy propagation

Izbegava se uvođenje promenljivih koje samo čuvaju vrednosti nekih postojećih promenljivih (engl. *copy propagation*).

```
a = x + y
b = x - y
c = x + y
d = a + c
```

```
a = x + y
b = x - y
c = a
d = a + c
```

```
a = x + y
b = x - y

d = a + a
```

Dead code elimination

Izračunavanja vrednosti promenljivih koje se dalje ne koriste, se eliminišu (engl. *dead code elimination*). Npr. ako se nakon prethodnog bloka koda koristi d, ali ne i b, tada se kôd uprošćava na sledeći način.

```
a = x + y        a = x + y        a = x + y
b = x - y        b = x - y        b = x - y
c = x + y        c = a
d = a + c        d = a + c        d = a + a
```

Dead code elimination

Izračunavanja vrednosti promenljivih koje se dalje ne koriste, se eliminišu (engl. *dead code elimination*). Npr. ako se nakon prethodnog bloka koda koristi d, ali ne i b, tada se kôd uprošćava na sledeći način.

```
a = x + y
b = x - y
c = x + y
d = a + c
```

```
a = x + y
b = x - y
c = a
d = a + c
```

```
a = x + y
b = x - y

d = a + a
```

```
a = x + y


d = a + a
```

Optimizacija petlji

Posebno je značajna optimizacija petlji, jer se kôd u njima očekivano izvršava veći broj puta i čak i mala ušteda može biti značajnija nego neka veća ušteda u kodu koji se izvršava samo jednom.

Optimizacija petlji

- Jedna od najznačajnijih optimizacija petlji je izdvajanje izračunavanja vrednosti promenljivih koje su invarijantne za tu petlju (čija je vrednost ista u svakom koraku petlje) ispred same petlje.
- Pokažimo ovu ideju na C kodu (mada se optimizacija izvršava na nivou međukoda).

```
for (int i = 0; i < n; i++)
  a[i] = r[i] * sin(alpha);
```

```
t1 = sin(alpha)
for (int i = 0; i < n; i++)
  a[i] = r[i] * t1;
```

Napomena: ovo sme da se uradi za funkciju sin ali ne i za getchar() - optimizacija to mora da prepozna!

Optimizacije

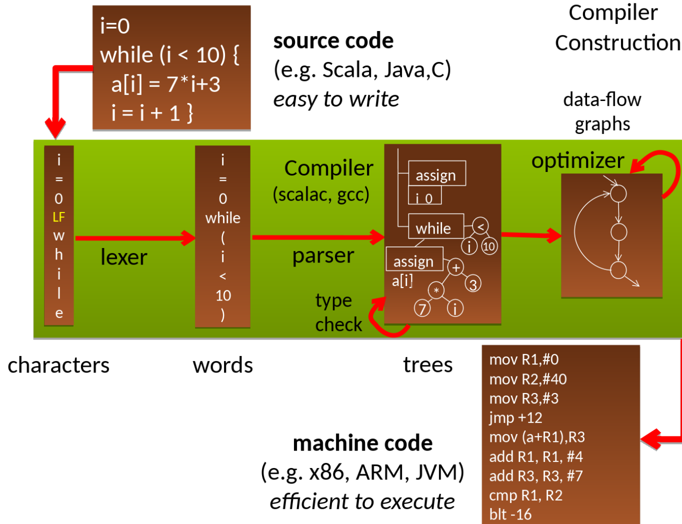
- Optimizacija uvek ide po nekom kriterijumu (vreme, memorija, energetska efikasnost...). Optimizacija po jednom kriterijumu najčešće narušava drugi
- Na primer, da bi se dobilo na brzini, radi se duplikacija koda (time se povećava upotreba memorije jer kod postaje veći)
- Duplikacija koda može da bude kod petlji, da bi se izbegla provera brojača ili umetanje kod poziva funkcija, da bi se izbegla cena kreiranja pozivnog steka
- Optimizacija umetanja je jedna od najbitnijih optimizacija — propuštanje umetanja kod frekventnog koda može da vodi drastičnom padu performansi
- Da bi se smanjila upotreba memorije, može da se radi optimizacija izdvajanja koda (Master rad Vladimir Vuksanović — *Unapređenje infrastrukture LLVM čuvanjem originalne lokacije pri debagovanju izdvojenog koda* `http://poincare.matf.bg.ac.rs/~milena/masteri/2023_VladimirVuksanovic/MasterRadVladimirVuksanovic.pdf`)

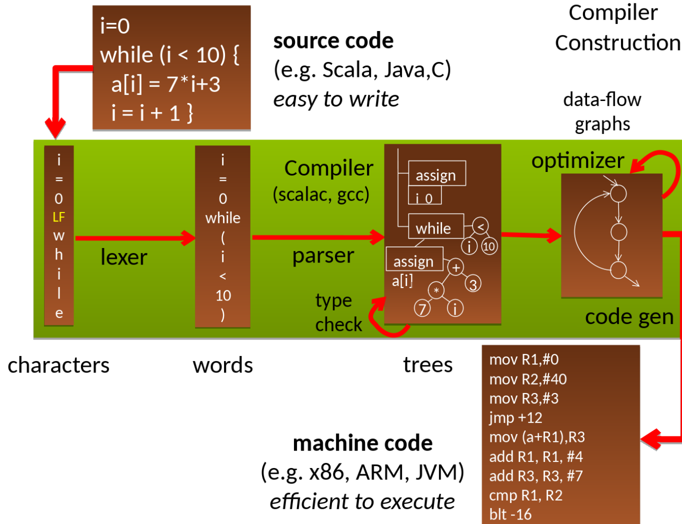
Optimizacije

- Optimizacije vođene profilima
- Suštinski je bitno koje delove koda optimizovati kako bi optimizacija imala pozitivan efekat na kod koji se izvršava i kako bi opravdala cenu koja je njena posledica
- Na primer, dobro je korišćenje duplikacije koda u delu koda koji se često izvršava jer se time dobija na brzini, ali nije dobro korišćenje duplikacije u delu koda koji se retko izvršava, jer se time ne dobija značajno na brzini, a ostaje cena povećane veličine koda koja se u tom slučaju ne isplati
- *Profile Guided Optimizations for Native Image* `https://medium.com/graalvm/profile-guided-optimization-for-native-image-f015f853f9a8`
- Optimizacije vođene mašinskim učenjem

Pregled

1. Leksička analiza
2. Sintaksička analiza
3. Semantička analiza
4. Generisanje međukoda
5. Optimizacija
6. Generisanje koda


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
data-flow graphs
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Compiler
(scalac, gcc)
parser
assign
i 0
while
assign
a[i]
type check
optimizer
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
data-flow graphs
Compiler
(scalac, gcc)
optimizer
lexer
parser
type check
code gen
assign
i 0
while
assign
a[i]
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Faze generisanja koda

- Tokom generisanja koda optimizovani međukod se prevodi u završni asemblerski tj. mašinski kod.
- Tri osnovne faze (koje mogu da budu isprepletane):
  - Faza odabira instrukcija (tada se određuje kojim mašinskim instrukcijama se modeluju instrukcije troadresnog koda),
  - Faza alokacije registara (tada se određuje lokacija na kojoj se svaka od promenljivih skladišti)
  - Faze raspoređivanja instrukcija (tada se određuje redosled instrukcija koji doprinosi kvalitetnijem iskorišćavanju protočne obrade i paralelizacije na nivou instrukcija).

Najčešće se prvo radi odabir instrukcija dok se raspoređivanje instrukcija nekada radi i pre i posle alokacije registara.

Izazovi generisanja koda

- Osnovni problem generisanja koda je što je **generisanje optimalnog programa** za dati izvorni kôd **neodlučiv problem**
- Mnogi problemi na koje se nailazi u generisanju koda (npr alokacija registara) su *computationally intractable* (bez efikasnih računarskih rešenja, teško računarski rešivi, računarski kompleksni)
- To znači da se u praksi koriste razne heurističke tehnike koje generišu dobar ali ne garantuju da će izgenerisati optimalan kod
- Razvoj heuristika je uznapredovao pa sada pažljivo dizajnirani generatori koda mogu da proizvedu kôd koji je neuporedivo brži nego kôd koji generiše naivnim pristupima

Izazovi generisanja koda — ispravnost

- Najbitniji kriterijum za generator koda je da on mora da proizvede **ispravan kôd**.
- Ispravnost ima specijalni značaj posebno zbog velikog broja specijalnih slučajeva sa kojima se generator koda susreće i koje mora adekvatno da obradi.
- Imajući u vidu važnost ispravnosti koda, posebno je važno dizajniranje generatora koda tako da on može biti lako implementiran, testiran i održavan.

Više prolaza kroz IR radi generisanja koda

- Ulaz u generator koda je IR izvornog programa koji je proizveo prednji deo kompajlera (i potencijalno već optimizovao srednji deo kompajlera), zajedno sa tablicama simbola koje se koriste za utvrđivanje run-time adresa objekata koji se koriste po imenu u okviru IRa.
- Generatori koda, (ili već na nivou generisanja međureprezentacije) razdvajaju IR instrukcije u "basic blocks", koje se sastoje od sekvenci instrukcija koje se uvek izvršavaju zajedno.
- U okviru faza optimizacije i generisanja koda najčešće postoje višestruki prolazi kroz IR koji se izvršavaju pre finalnog generisanja ciljnog programa.

Ulaz u generator koda

- Frontend je skenirao, parsirao i translirao izvorni program u IR niskog nivoa, što znači da se vrednosti imena koje se pojavljuju u IRu mogu predstaviti sa veličinama sa kojima ciljna mašina može direktno da manipuliše (npr celobrojne vrednosti, realni brojevi)
- Takođe, sintaksne, statičke i semantičke greške su već otkrivene, provere tipova su izvršene, konverzije su umetnute tamo gde je trebalo.
- Prema tome, generator koda dalje nastavlja pod pretpostavkom da je ulaz ispravan, ili barem slobodan od pomenutih grešaka

Ciljni program

- Arhitektura procesora definiše, pre svega, skup instrukcija i registara.
- Skup instrukcija ciljne mašine ima značajan uticaj na teškoće u konstruisanju dobrog generatora koda koji je u stanju da proizvede mašinski kôd visokog kvaliteta.
- Broj i uloge registara takođe imaju značajan uticaj
- Java kôd se kompilira u bytecode koji se dalje interpretira, ili se koriste JIT kompajleri — u ovom slučaju bitan je dizajn bytecode-a
- Postoje i kompajleri za Javu — tzv *Ahead Of Time* AOT kompilacija (Native Image)

ISA (Instruction Set Architecture)

ISA (*Instruction Set Architecture*) definiše izvršive mašinske instrukcije za dati tip hardvera. Kompleksnost ISA zavisi od

- formata instrukcije,
- formata podataka,
- načina adresiranja,
- registara opšte namene,
- specifikacije opkoda i
- mehanizama za upravljanje tokom izvršavanja programa.

Najčešće arhitekture su RISC (*Reduced Instruction Set Computer*) i CISC (*Complex Instruction Set Computer*).

CISC (Complex Instruction Set Computer)

- CISC arhitekuru procesora karakteriše bogat skup instrukcija.
- Bogat skup instrukcija ima za cilj da se smanje troškovi memorije za skladištenje programa.
- Broj instrukcija po programu se smanjuje žrtvovanjem broja ciklusa po instrukciji, tj ugradnjom više operacija u jednu instrukciju, praveći tako različite kompleksnije instrukcije
- Na primer, instrukcija može vršiti učitavanje vrednosti iz memorije, zatim primenu neke aritmetičke operacije, i zapisivanje rezultata u memoriji.
- Instrukcije mogu biti različitih dužina

Primer add — Integer Addition

The add instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.

```
Syntax
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>

Examples
add eax, 10            - eax <- eax + 10
add BYTE PTR [var], 10 - add 10 to the single byte stored
                         at memory address var
```

Primer - potprogrami — prolog i epilog

- L je broj bajtova potrebnih za rezervisanje prostora za lokalne promenljive

```
push ebp
mov  ebp, esp          <---->        enter L, 0
sub  esp, L
```

-
```
mov esp, ebp
pop ebp                <---->        leave
```

- Cilj prethodnih instrukcija je izdvajanje čestih blokova instrukcija u hardver zarad podizanja nivoa apstrakcije već na samom hardveru i smanjenja veličine programa

CISC (Complex Instruction Set Computer)

- Ovako složene instrukcije zahtevaju kompleksnost hardvera i rezultujuće arhitekture. To ima za posledicu i teže razumevanje i programiranje takvih čipova, a pored toga i veću cenu.
- CISC procesori se uglavnom koriste na ličnim računarima, radnim stanicama i serverima, a primer ovakvih procesora je arhitektura Intel x86.
- CISC procesori imaju više različitih načina adresiranja, od kojih su neki veoma kompleksni.
- CISC procesori obično nemaju veliki broj registara opšte namene

RISC (*Reduced Instruction Set Computer*)

- Na osnovu analiza izvršavanja programa, ustanovljeno je da se najveći deo kompleksnih instrukcija jako retko koristi i da je njihova implementacija na nivou hardvera zbog toga neopravdana (podiže cenu hardvera a da pritom od toga nema dovoljno benefita)
- Treba ubrzati ono što se najviše koristi
- Tako nastaje RISC — arhitektura sa procesorom smanjenog skupa instrukcija

RISC (*Reduced Instruction Set Computer*)

- Istraživačka grupa na univerzitetu Stanford je imala jako znanje kompajlera koje je iskoristla za razvoj procesora čija arhitektura treba da predstavlja spuštanje kompajlera na hardverski nivo, odnosno suprotno od postojećeg trenda koji je korišćen za CISC hardver i koji je zastupao podzanje hardvera na softverski nivo (što je dugo bila glavna filozofija dizajna hardvera u industriji). Razvoj počinje 80tih godina. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html`

RISC (*Reduced Instruction Set Computer*)

- RISC arhitektura procesora se zasniva na pojednostavljenom i smanjenom skupu instrukcija koji je visoko optimizovan.
- Zbog jednostavnosti instrukcija, potreban je manji broj tranzistora za proizvodnju procesora, pri čemu procesor instrukcije može brže izvršavati.
- Međutim, redukovanje skupa instrukcija umanjuje efikasnost pisanja softvera za ove procesore, što ne predstavlja problem u slučaju automatskog generisanja koda kompajlerom
- Ne postoje složene instrukcije koje pristupaju memoriji, već se rad sa memorijom svodi na *load* i *store* instrukcije

RISC (*Reduced Instruction Set Computer*)

- Najveća prednost je protočna obrada, koja se lako može implementirati. Protočna obrada (eng. *pipeline*) je jedna od jedinstvenih odlika arhitekture RISC, koja je postignuta preklapanjem izvršavanja nekoliko instrukcija.
- Zbog protočne obrade RISC arhitektura ima veliku prednost u performansama u odnosu na CISC arhitekture
- RISC procesori se uglavnom koriste za aplikacije u realnom vremenu.
- Primer RISC procesora su ARM i MIPS.

MIPS primer `i = N*N + 3*N`

Troadresni kôd:

```
t0 = N*N,

t1 = 3*N,


i = t0+t1
```

Neptimizovan kod:

```
lw      $t0, 4($gp)     # fetch N
mult    $t0, $t0, $t0   # N*N
lw      $t1, 4($gp)     # fetch N
ori     $t2, $zero, 3   # 3
mult    $t1, $t1, $t2   # 3*N
add     $t2, $t0, $t1   # N*N + 3*N
sw      $t2, 0($gp)     # i = ...
```

Optimizovan kod (28.6%):

```
lw      $t0, 4($gp)      # fetch N
add     $t1, $t0, $zero  # copy N to $t1
addi    $t1, $t1, 3      # N+3
mult    $t1, $t1, $t0    # N*(N+3)
sw      $t1, 0($gp)      # i = ...
```

Izbor instrukcija

Generator koda treba da preslika IR programa u sekvencu mašinskih instrukcija koje mogu da budu izvršene na ciljnoj mašini. Kompleksnost ovog preslikavanja je određena narednim faktorima:

- Nivo apstrakcija/preciznosti IRa
- Priroda instrukcija arhitekture
- Željeni kvalitet generisanog koda

Nivo apstrakcija/preciznosti IRa

- Ako je IR visokog nivoa, generator koda može da translira svaku naredbu IRa u sekvencu mašinskih instrukcija korišćenjem šablonskog koda. Takvo generisanje koda, naredba za naredbom, međutim, proizvodi kod lošeg kvaliteta za koji su neophodne dalje optimizacije.
- Ako je IR niskog nivoa takav da oslikava neke detalje niskog nivoa ciljne mašine, onda generator koda može da koristi ove informacije i da generiše efikasnije nizove instrukcija.

Priroda instrukcija arhitekture

- Priroda skupa instrukcija ciljne mašine ima jak uticaj na teškoće u implementaciji izbora instrukcija
- Na primer, bitni faktori su uniformnost (da li format instrukcija uniforman) i kompletnost instrukcija (da li nedostaju neke potrebne instrukcije)
- Ako ciljna mašina ne podržava svaki tip podataka na uniforman način, onda svaki izuzetak generalnog pravila zahteva specijalno rukovanje i dodatni kod
- Na primer, na nekim mašinama se operacije u pokretnom zarezu rade nad posebnim skupom registara

Željeni kvalitet generisanog koda

- Brzina pojedinačnih instrukcija i njihova veličina su takođe relevantni faktori
- Kada nam ne bi bila bitna efikasnost ciljnog programa, izbor instrukcija bi mogao da bude pravolinijski

Primer

- Za svaki tip troadresne naredbe, možemo da dizajniramo kod koji definiše ciljni kod koji će se generisati za tu naredbu
- Na primer, svaka troadresna naredba oblika `x = y + z`, gde su x, y, i z statički alocirani, može da se prevede u narednu sekvencu koda

```
LD R0 , y       // R0 = y       ( load y into register R0)
ADD R0, R0, z   // R0 = R0 + z  ( add Z to R0)
ST x , R0       // x = R0       ( store R0 into x )
```

Primer

Ova strategija obično daje redundantna čitanja i upisivanja u memoriju. Na primer:

```
a = b + c
d = a + e
```

```
LD  R0, b      // R0 = b
ADD R0, R0, c // R0 = R0 + c
ST  a, R0      // a = R0
LD  R0, a      // R0 = a
ADD R0, R0, e // R0 = R0 + e
ST  d, R0      // d = R0
```

Ovde je četvrta instrukcija redudantna jer sa njom učitavamo iz memorije vrednost koja je upravo tu upisana iz istog tog registra. Takođe, i treća je redudantna jer se vrednost a više ne koristi.

Željeni kvalitet generisanog koda

- Kvalitet generisanog koda se obično određuje na osnovu njegove brzine i veličine.
- Na većini mašina, zadati IR program može se implementirati sa puno različitih kodnih sekvenci, sa značajnim razlikama u ceni između samih implementacija
- Naivna translacija IR koda u izvršni kod može da vodi ka korektnom ali neprihvatljivo sporom i neefikasnom izvršnom kodu

Primer

Ako ciljna mašina ima instrukciju inkrementiranja (`INC`), onda troadresna naredba `a = a + 1` može efikasnije da bude implementirana korišćenjem instrukcije `INC a`, umesto korišćenja sekvence koja učitava a u registar, sabira jedinicu sa tim registrom, i onda upisuje reyultat nazad u a:

```
LD R0, a            // R0 = a
ADD R0, R0, # 1     // R0 = R0 + 1
ST a, R0            // a = R0
```

Željeni kvalitet generisanog koda

- Neophodno je da znamo cene instrukcija kako bi mogli da dizajniramo dobre sekvence koda. Nažalost, tačne cene je često veoma teško dobiti
- Odlučivanje koja sekvenca mašinskog koda je najbolja za dati troadresni konstrukt može takođe da zahteva znanje o kontekstu u kojem se taj konstrukt nalazi

Izbor registara

- Tokom faze registarske alokacije određuju se lokacije na kojima će biti skladištene vrednosti svih promenljivih koje se javljaju u međukodu (kako originalnih, tako i privremeno uvedenih tokom prevođenja u međukod).
- Cilj je da što više promenljivih bude skladišteno u registre procesora (jer je pristup registrima često za red veličine brži nego pristup memoriji), međutim, to je često nemoguće, jer je broj registara ograničen i često prilično mali.
- Dve promenljive mogu biti smeštene u isti registar ako im se životni vek (period u kome se koriste) ne preklapa.

Izbor registara

- Problem korišćenja registara je obično podeljen u dva podproblema:
  1. Alokacija registara — tokom koje se biraju skupovi promenljivih koji treba da borave u registrima u svakoj tački programa
  2. Dodela registara — tokom koje se biraju određeni konkretni registri u kojima će promenljiva boraviti

Izbor registara

- Izbor registara (pronalaženje optimalne dodele registara promenljivama) je **NP kompletan problem**
- Problem se dodatno komplikuje time što hardver i / ili operativni sistem ciljne mašine mogu da zahtevaju nekakve određene konvencije u korišćenju registara
- Na primer, sadržaji nekih registara moraju da budu sačuvani prilikom poziva funkcija ili registri u kojima se čuvaju rezultati deljenja, ili registri koji čuvaju neke specifične adrese i slično

Raspoređivanje instrukcija

- Faza raspoređivanja instrukcija pokušava da doprinese brzini izvršavanja programa menjanjem redosleda instrukcija. Naime, nekim instrukcijama je moguće promeniti redosled izvršavanja bez promene semantike programa.
- Neki rasporedi instrukcija zahtevaju manji broj registara za čuvanje privremenih rezultata.
- Jedan od ciljeva raspoređivanja je da se upotrebe pojedinačnih promenljivih lokalizuju u kodu, čime se povećava šansa da se registri oslobode dugog čuvanja vrednosti nekih promenljivih i da se upotrebe za čuvanje većeg broja promenljivih.

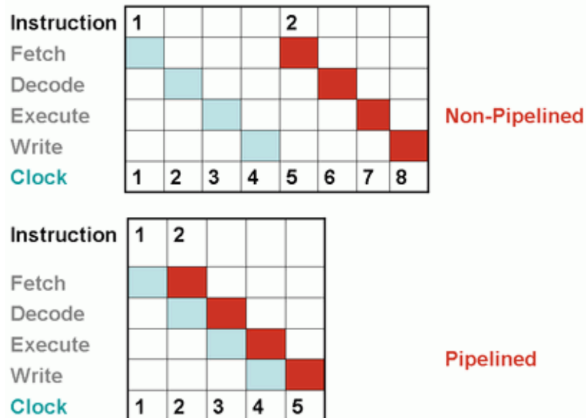
Raspoređivanje instrukcija

- Izbor najboljeg redosleda je u opštem slučaju **NP-kompletan problem**.
- Najjednsotavnije rešenje je ne menjati redosled instrukcija u odnosu na ono što je dao generator međukoda

Protočna obrada

- Savremeni procesori mogu da izvršavaju nekoliko instrukcija tokom jednog otkucaja sistemskog sata tj. imaju osobinu protočne obrade (engl. *pipelining*).
- Dok se jedna instrukcija dovlači iz memorije u procesor, druga se dekodira, treća se već izvršava, a četvrtoj se već skladište rezultati.

Protočna obrada

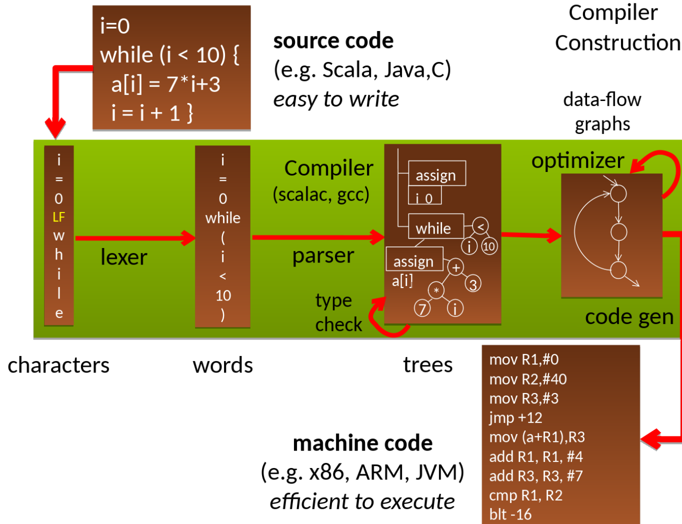


Protočna obrada

- Promena redosleda instrukcija može uticati na to da se protočna obrada bolje iskoristi tj. da se izbegnu čekanja i zastoji u protočnoj obradi (engl. *pipeline stalls*) nastala zbog zavisnosti između susednih instrukcija.
- Na primer, kada je narednoj instrukciji potreban rezultat prethodne, njeno izvršvanje mora da pričeka da prethodna instrukcija završi sa upisom svojih rezultata.
- Raspoređivač u taj zastoj može umetnuti neku drugu instrukciju, nezavisnu od ovih i tako ubrzati ukupno izvršavanje svih instrukcija zajedno (ta pomerena instrukcija će biti izvršena dok bi procesor praktično čekao u prazno).

Serijalizacija

- Kada su instrukcije odabrane, kada je izvršena alokacija registara i kada su instrukcije raspoređene, dobijeni kôd se serijalizuje tj. zapisuje se u datoteke (objektne module) u obliku konkretnog mašinskog koda, čime se faza kompilacije završava
- Nekada kompilator generiše asemblerski kôd koji se onda asemblerom prevodi u mašinski.
- Nakon kompilacije pojedinačni objektni moduli se povezuju (engl. *linking*) u izvršnu datoteku.


Compiler Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
data-flow graphs
i = 0 LF w h i l e
i = 0 while (i < 10)
Compiler
(scalac, gcc)
assign
i 0
while
<
i 10
assign
a[i]
+
3
*
7 i
optimizer
lexer
parser
type check
code gen
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Zaključak

Ovo je bio opšti pregled osnovnih faza kompilacije i njihovih izazova.

Literatura

- (The Dragon Book) Compilers: Principles, Techniques, and Tools Alfred V. Aho,Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Kompajleri Stanford https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/
- Konstrukcija kompilatora EPFL https://lara.epfl.ch/w/cc
- Skripta Filipa Marića http://poincare.matf.bg.ac.rs/~filip/kk/materijali/kk.pdf