Programski prevodioci (vežbe)

Semantička analiza

Tabele simbola

Semantika programskog jezika

- Dok se leksički elementi i sintaksa programskog jezika definišu strogo formalno (pomoću regularnih izraza i gramatike jezika), ne postoje formalna pravila za definisanje semantike programskih jezika, tj. semantika jezika se opisuje na neformalni način.
- Primer nekih semantičkih pravila programskog jezika C++:
  - U programu postoji samo jedna main funkcija (definisana na globalnom nivou).
  - Main funkcija moze da bude bez argumenata ili da ima 2 argumenta (prvi je tipa int, a drugi tipa char**).
  - Ne može se koristi promenljiva koja nije deklarisana,
  - Ne može se koristiti promenljiva kojoj nije dodeljena vrednost,
  - U pozivu funkcije lista stvarnih argumenata mora da se slaže po broju i tipu argumenata sa listom fiktivnih argumenata te funkcije.
  - Operatori treba da budu primenjeni nad operandima odgovarajućeg tipa (npr. operator % je primenljiv nad celobrojnim tipovima podataka).

Osnovni zadaci semantičkog analizatora

- Kreiranje tabele simbola
- Provera opsega važenja simbola (scope checking)
- Provera tipova podataka (type checking)
- Provera main metoda

Tabela simbola

- Održava informacije o deklarisanim imenima i tipovima.
- Tabela simbola je dinamička struktura podataka koja sadrži čvorove objekata i čvorove tipova.
- U tabeli simbola postoji po jedan čvor za sve osnovne tipove i sve izvedene tipove podataka (bibliotečke i korisnički definisane).

Uobičajeni podaci o deklarisanim objektima (imenovanim entitetima)

- Obavezni za sve:
  - Ime,
  - Tip.
- Specifični za pojedine tipove objekata:
  - Vrednost – za konstante,
  - Adresa – za promenljive i metode,
  - Lista parametara – za metode,
  - ...

Pristupi za predstavljanje simbola u tabelama simbola

- „Flat“ pristup:
  - Jedinstvena struktura podataka za predstavljanje svih tipova imenovanih entiteta
- Objektno-orijentisani pristup:
  - Zajedničke karakteristike svih imenovanih tipova entitieta izdvojiti u jednu apstraktnu klasu
  - Definisati izvedene klase za predstavljanje svih konkretnih tipova entiteta

Predstavljanje čvorova objekata

- “flat” implementation:

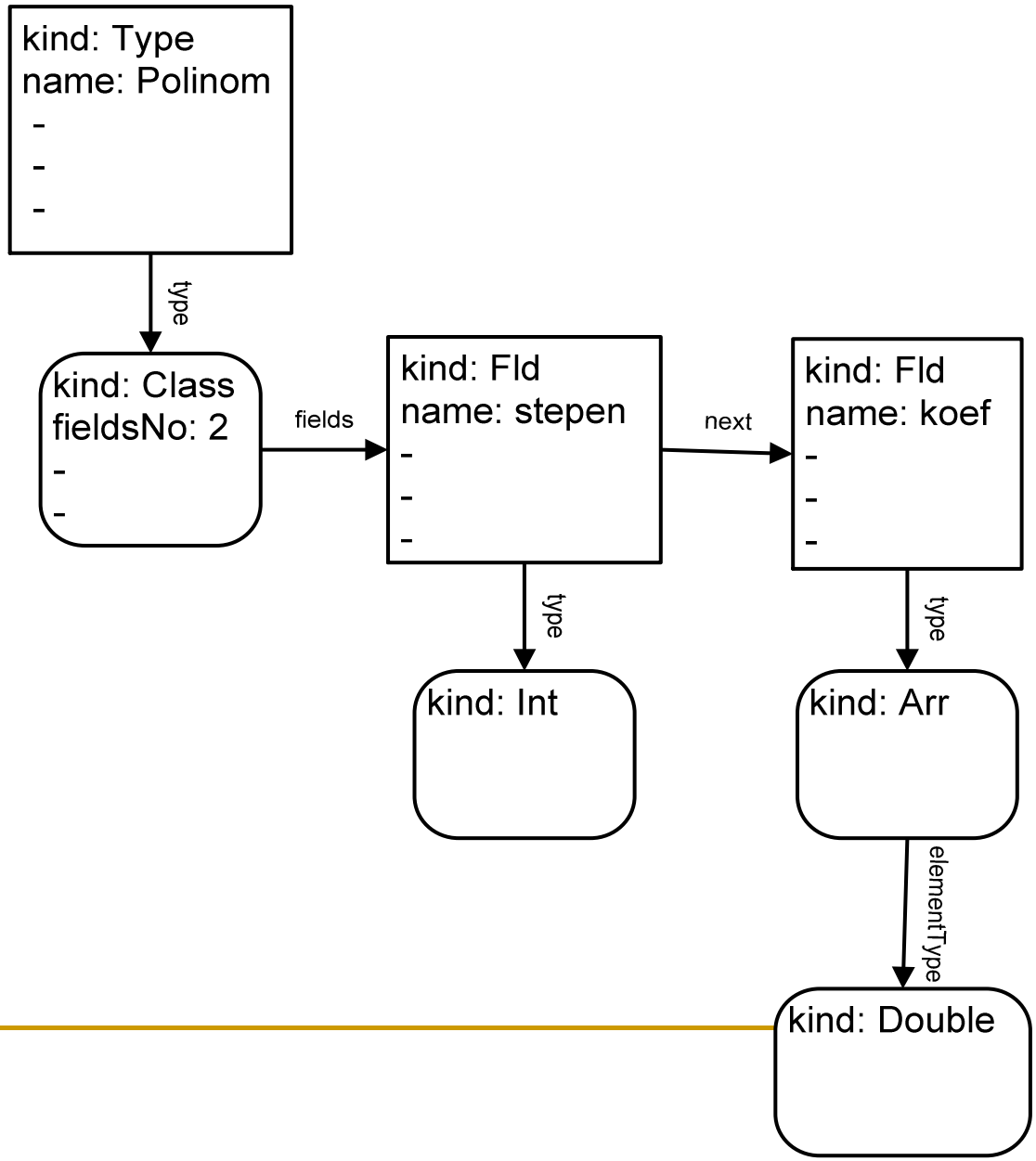
```
class ObjectNode
{
  static const int Con = 0, Var = 1, Type = 2,
     Fld = 3, Meth = 4;
  String name;
  int kind;      // Con, Var, Type, Fld, Meth
  TypeNode* type;
  ObjectNode* next;
  int adr;  // Var, Fld, Meth: memorijska adresa
  int parametersNo;
  ObjectNode* parameters; // Meth: argumenti
  void* value;  // Con: vrednost
}
```

Predstavljanje čvorova tipova

```
class TypeNode
{
  static const Int = 0, Char=1, Float=2,
      Double=3, Class=4, Arr=5;
  int kind;                    // Int, Char, Float,…
  TypeNode* elementType;       // tip elemenata u nizu
  int parentsNo;               // broj roditeljskih klasa
  TypeNode* parentTypes;       // niz roditeljskih klasa
  int fieldsNo;                // broj atributa
  ObjectNode* fields;          // niz atributa
};
```

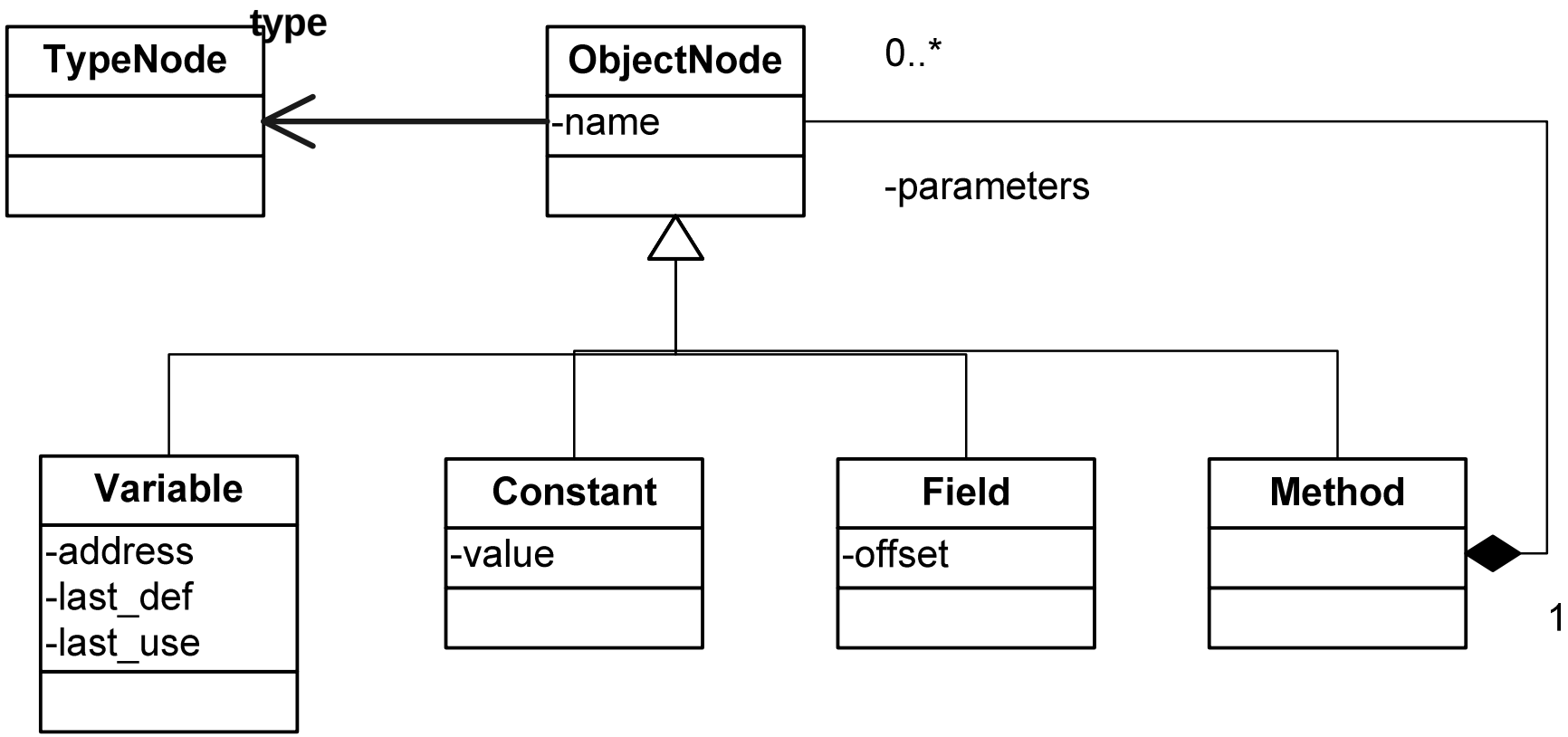
Primer: Predstavljanje klasa u tabeli simbola

```
class Polinom
{
   int stepen;
   double koef[10];
}
```

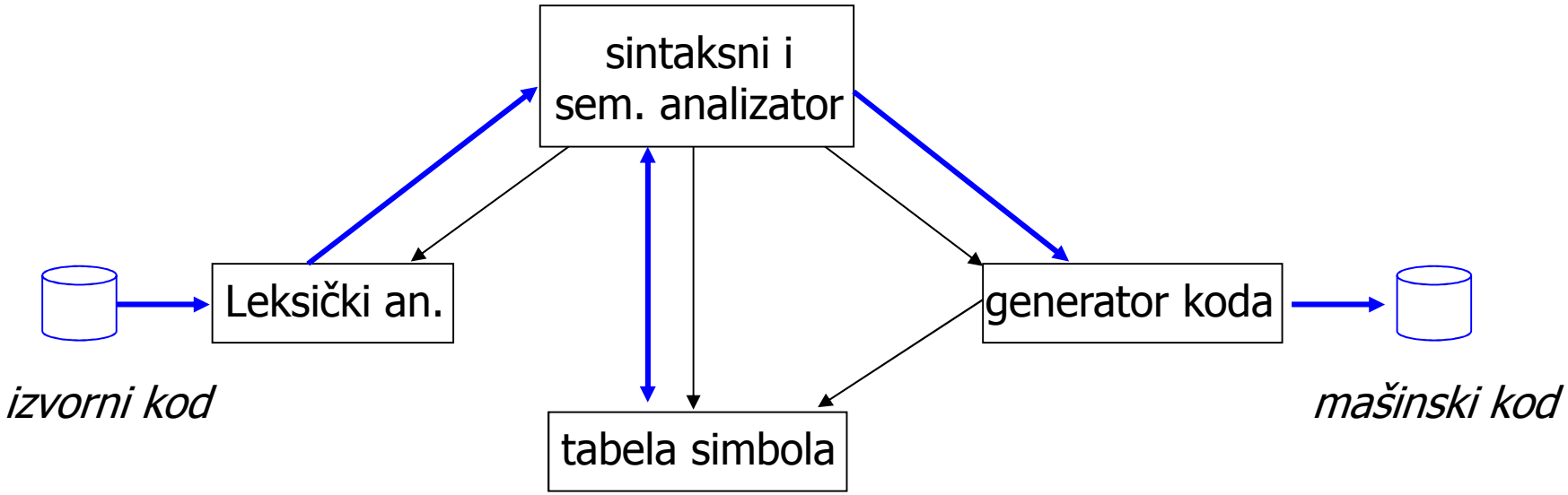


Predstavljanje čvorova objekata

- Objektno-orijentisani pristup:



Arhitektura kompilatora



Načini implementacije tabele simbola

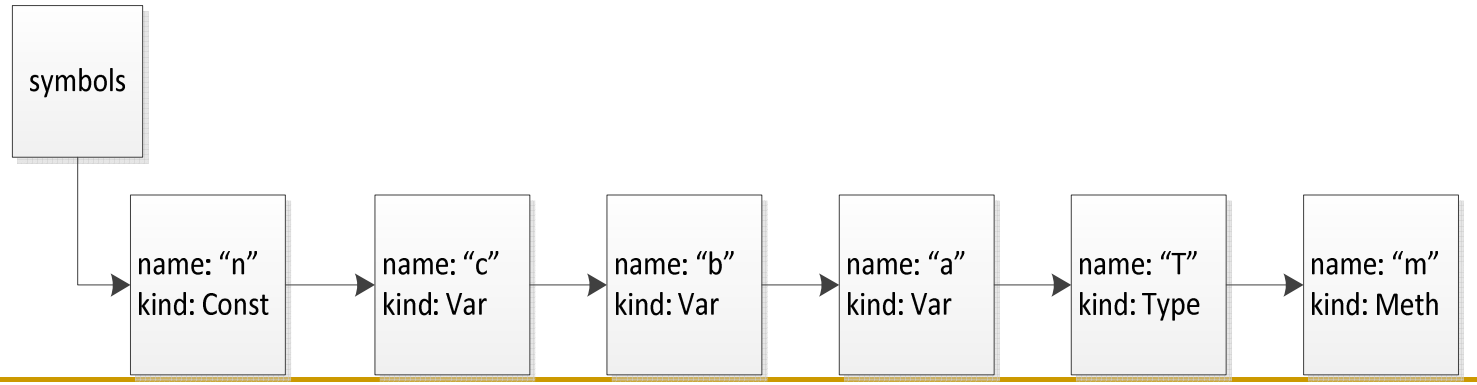
- U obliku lančane liste,
- Kao binarno stablo,
- Kao rasuta tablica.

Primer: Različite implementacije tabele simbola

Posmatrajmo sledeći deo koda:

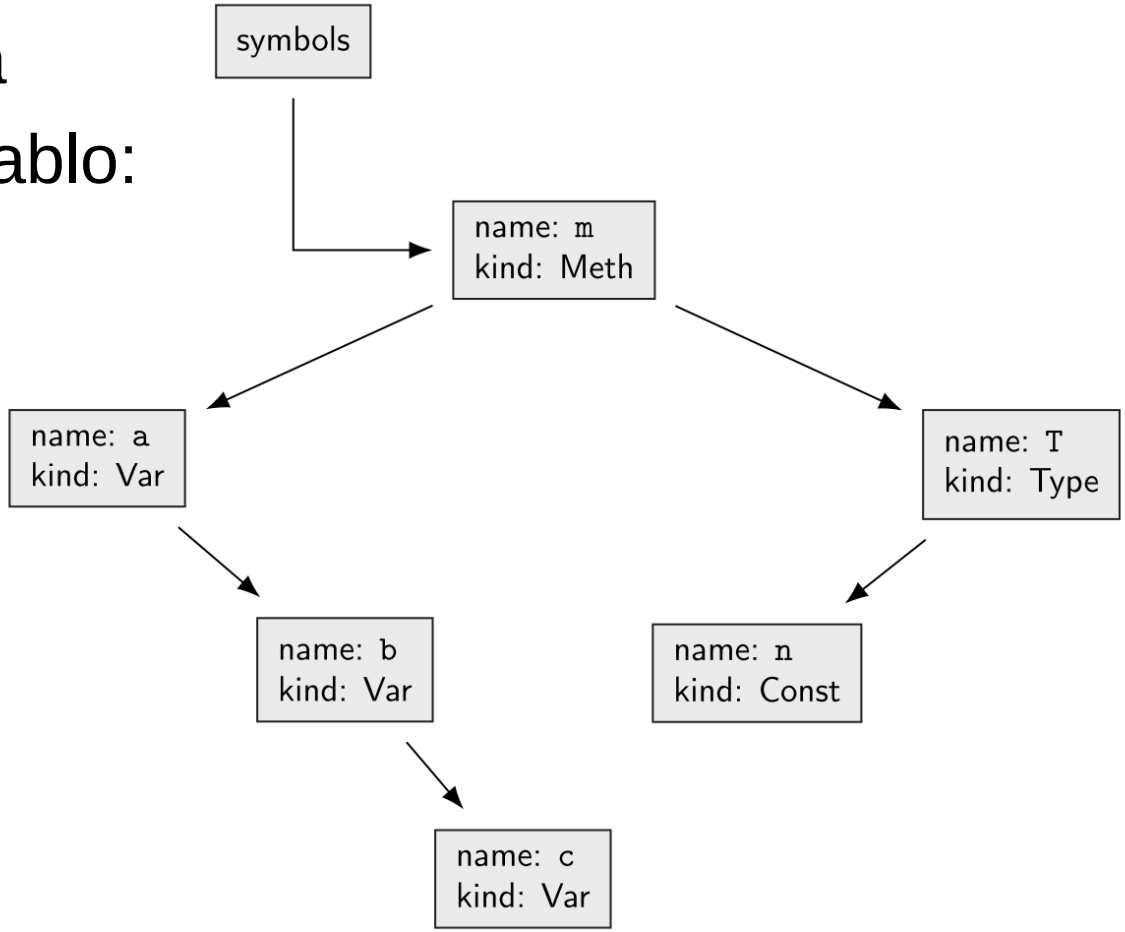
```
void m() { … }
class T { … };
int a, b, c;
const int n=10;
```

1. Tabela simbola kao lančana lista:



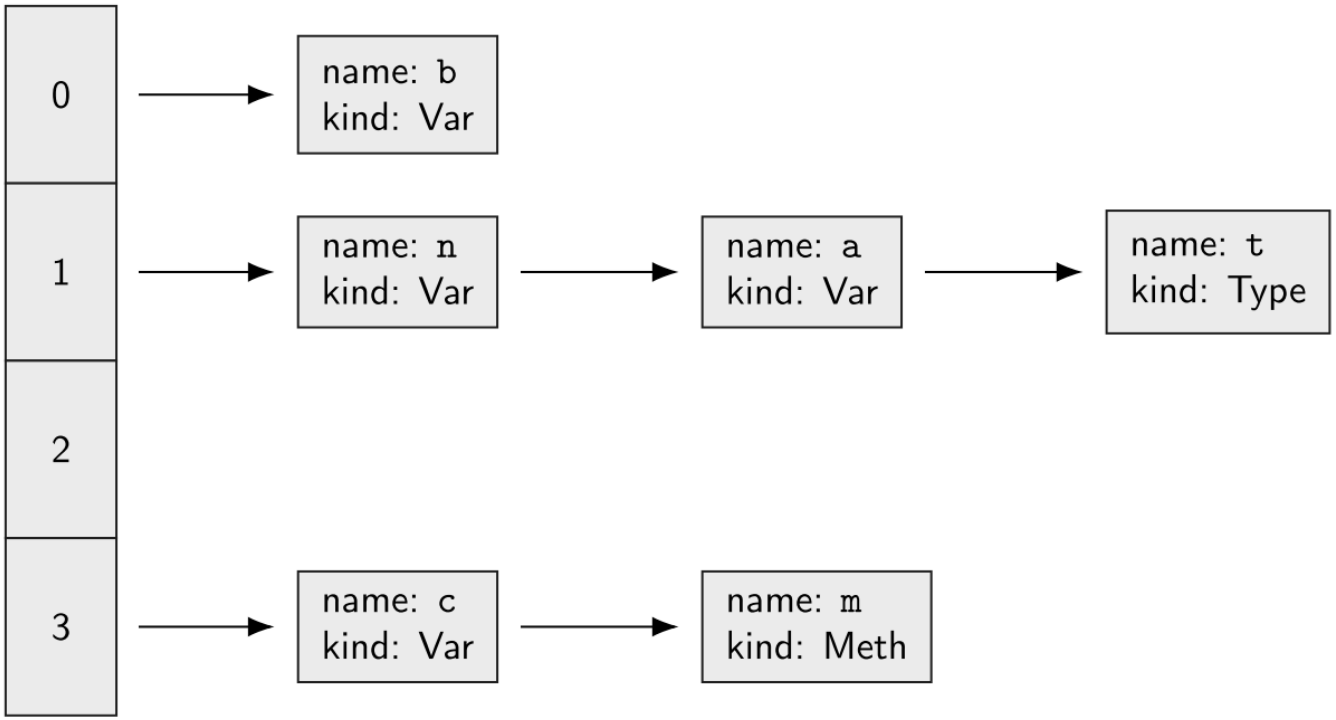
Primer: Različite implementacije tabele simbola

2. Tabela simbola kao binarno stablo:



Primer: Različite implementacije tabele simbola

3. Tabela simbola kao rasuta tablica:



Osobine različitih implementacija tabele simbola

- Lančana lista:
  - Jednostavna struktura,
  - Zadržava originalni redosled deklaracija,
  - Veoma sporo pretraživanje ukoliko postoji veći broj deklaracija.
- Binarno stablo:
  - Pretraživanje brže,
  - Utrošak memorijskog prostora veći,
- Rasuta tablica:
  - Pretraživanje veoma brzo,
  - Memorijska struktura mnogo komplikovanija od lančane liste,
  - Redosled deklarisanja narušen.

Opseg važenja imena

- U objektno-orijentisanim jezicima (tipa C++) postoje sledeći opsezi važenja imena:
  - C++ – unapred definisana imena (npr. int, char, float, ...)
  - program - globalna imena,
  - klasa – članovi klase,
  - metod – lokalna imena,
  - blok – imena definisana unutar bloka.
- Dozvoljeno je ugnježdavanje opsega imena.
- Tekući opseg imena je opseg važenja definisan trenutnim najdubljim ugnježdenjem.

Pravila vidljivosti imena

- U bilo kojoj tački programa vidljiva su imena deklarisana u tekućem opsegu važenja i u opsezima važenja koji sadrže tekući opseg važenja.
- Ako je ime deklarisano u više od jednog opsega važenja, značenje tog imena definisano je deklaracijom koja se nalazi na najvećoj dubini ugnježdavanja, tj. deklaracijom koja je najbliža tački posmatranja.
- Nove deklaracije su vidljive samo u tekućem opsegu posmatranja.

Implementacija tabele simbola u zavisnosti od opsega važenja

Postoje 2 osnovna pristupa:

- Posebna tabela simbola za svaki opseg važenja,
- Jedinstvena tabela simbola.

Posebna tabela simbola za svaki opseg važenja

- U ovom slučaju tabele simbola (ili bar reference na njih) se čuvaju u magacinu i po zatvaranju tekućeg opsega, odgovarajuća tabela se izbacuje iz magacina.
- Nedostaci ovakvog pristupa:
  - Ako se pristupa nekom imenu koje nije definisano u tekućem opsegu, pretražuje se veći broj tabela simbola što usporava traženje.
  - Ako su tabele realizovane kao hash tabele, za svaki opseg se kreira nova tabela pa kada je broj imena u opsegu mali dolazi do velikog rasipanja memorijskog prostora. Zbog toga se u ovakvim slučajevima tabele simbola obično realizuju kao lančane liste.

Jedinstvena tabela simbola

- Svi simboli su smešteni u jedinstvenu tabelu i uz svaki simbol se pamti nivo ugnježdenja opsega važenja u kojem je to ime deklarisano.
- Uvodi se posebna premenljiva koja broji trenutno otvorene opsege.
- Prednosti ovakvog pristupa:
  - Brže traženje simbola,
  - Potreban memorijski prostor u slučaju korišćenja hash tabela je manji.
- Nedostaci ovakvog pristupa:
  - Uz svako ime se pamti još jedno polje (opseg važenja).

Zadatak

- Prikazati sadržaj tabele simbola u obeleženim tačkama datog C++ koda. (Posmatrati slučaj jedinstvene tabele i posebne tabele simbola za svaki opseg važenja).

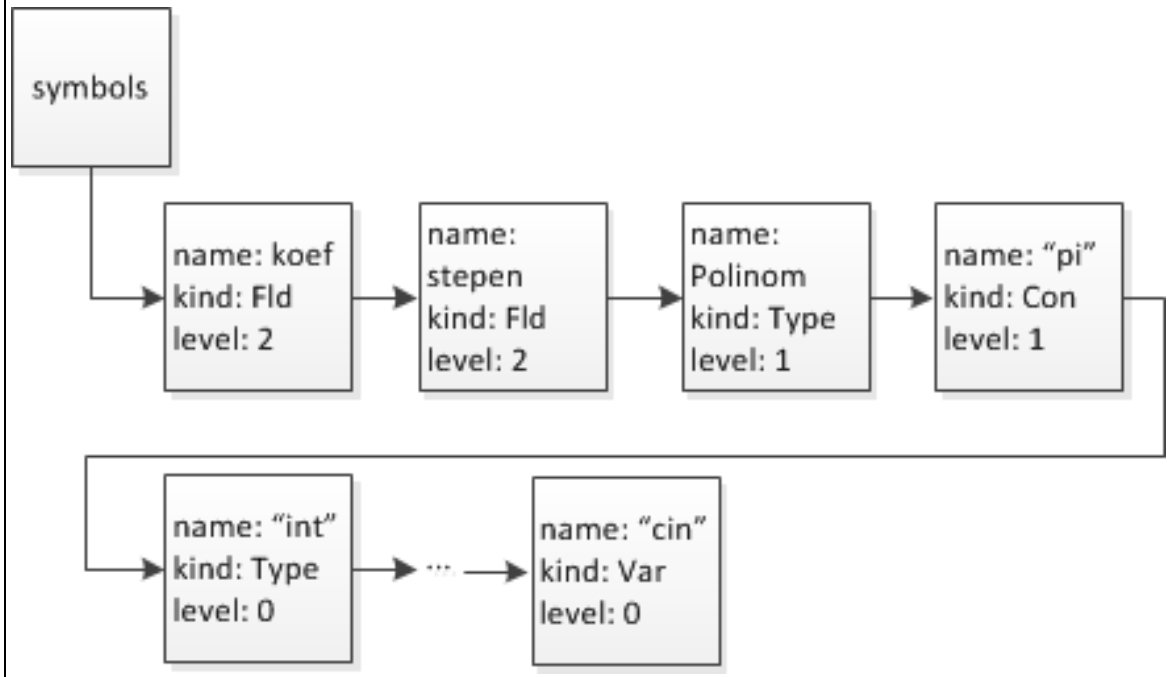
```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```

Tačka 1 jedinstvena tabela

```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```

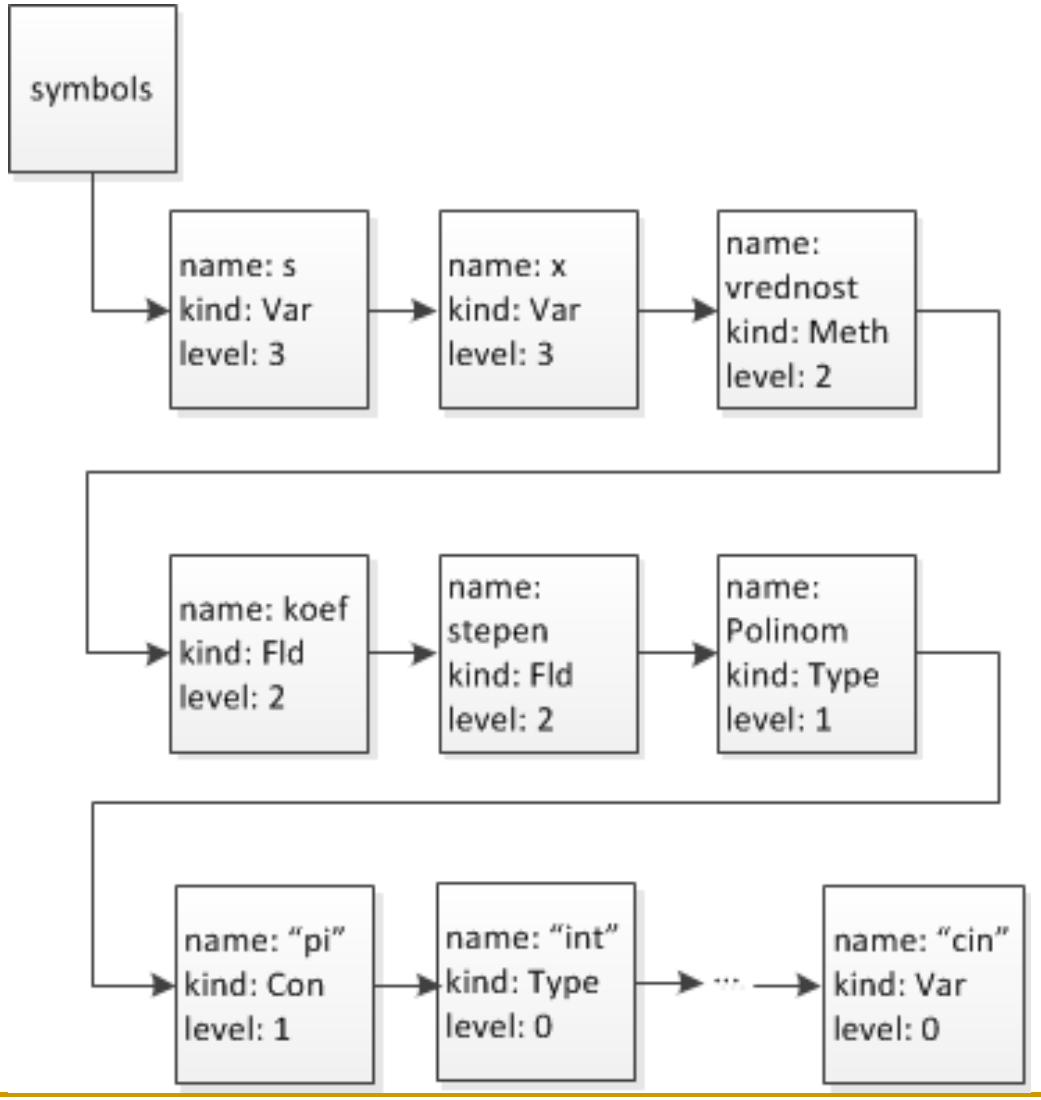
Tačka 2 jedinstvena tabela

```
// tacka 1
const double pi=3.14;
class Polinom {
 int stepen;
 double koef[100];
 // tacka 2
 double vrednost( double x )
 {
   double s=0;
   // tacka 3
   for ( int i=0;
      i<=stepen; i++)
     s = s*x+koef[i];
 }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



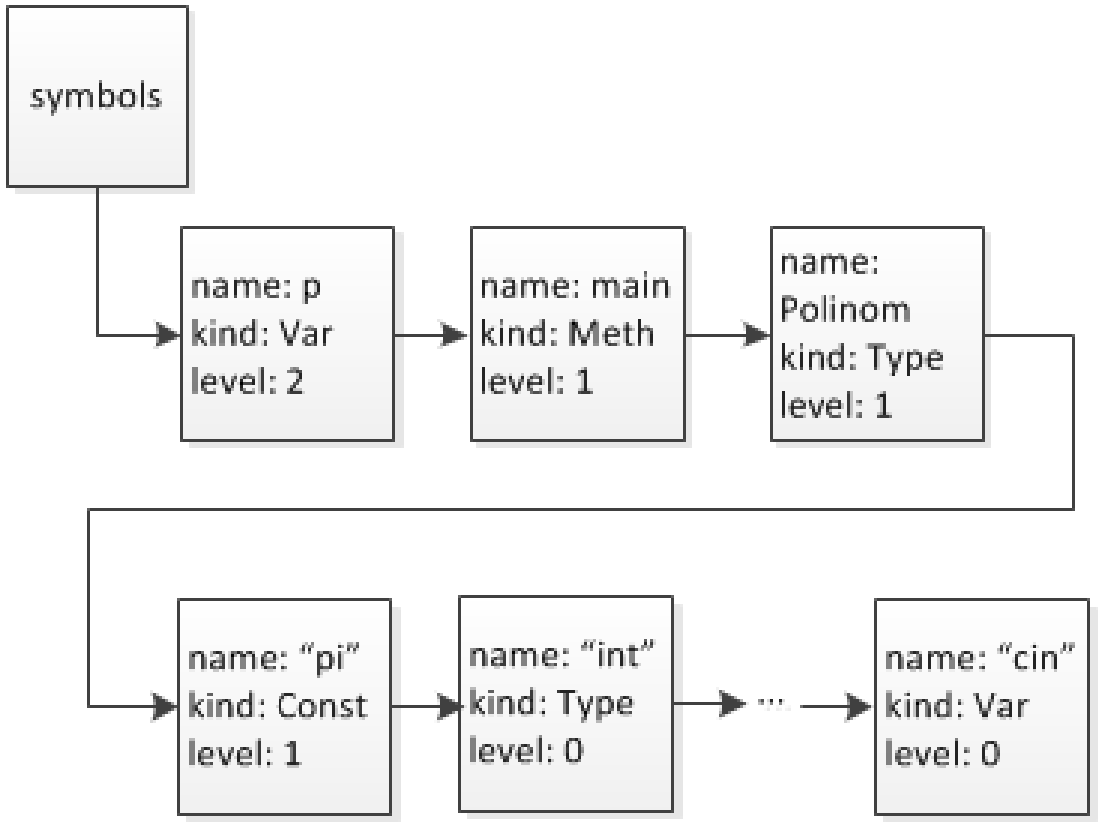
Tačka 3 jedinstvena tabela

```
// tacka 1
const double pi=3.14;
class Polinom {
 int stepen;
 double koef[100];
 // tacka 2
 double vrednost( double x )
 {
   double s=0;
   // tacka 3
   for ( int i=0;
      i<=stepen; i++)
     s = s*x+koef[i];
 }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



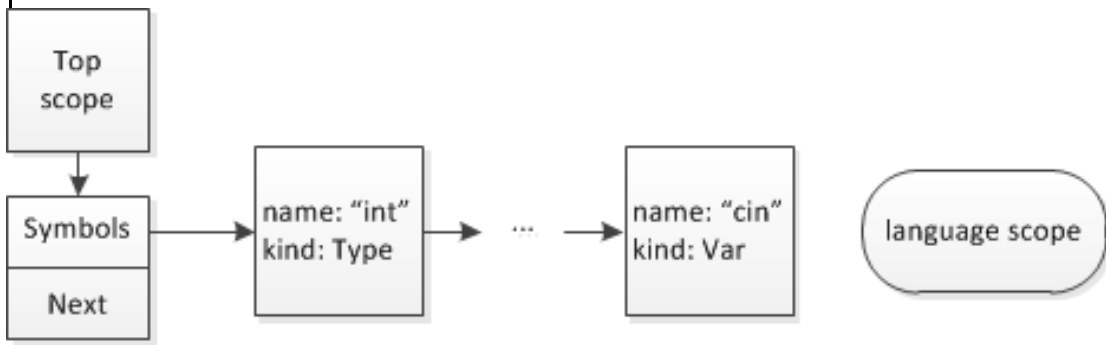
Tačka 4 jedinstvena tabela

```
// tacka 1
const double pi=3.14;
class Polinom {
 int stepen;
 double koef[100];
 // tacka 2
 double vrednost( double x )
 {
   double s=0;
   // tacka 3
   for ( int i=0;
      i<=stepen; i++)
     s = s*x+koef[i];
 }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



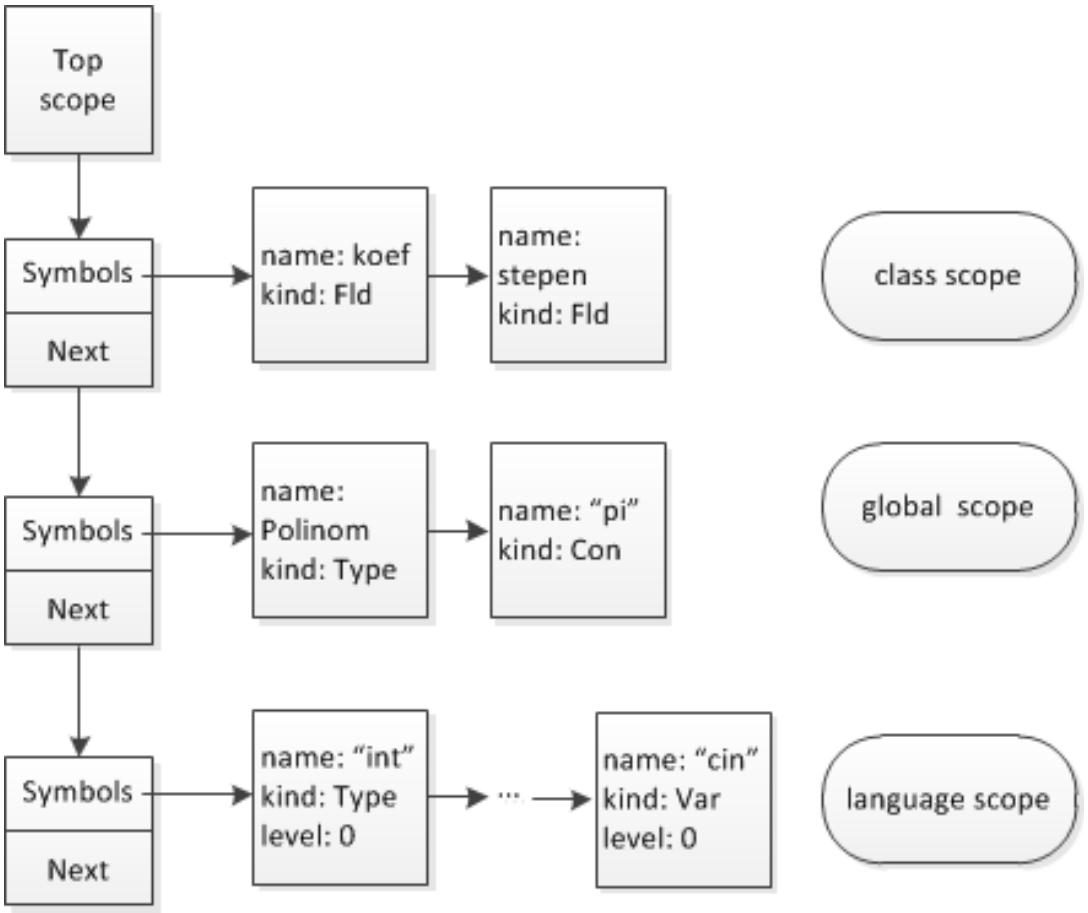
Tačka 1 posebna tabela za svaki scope

```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



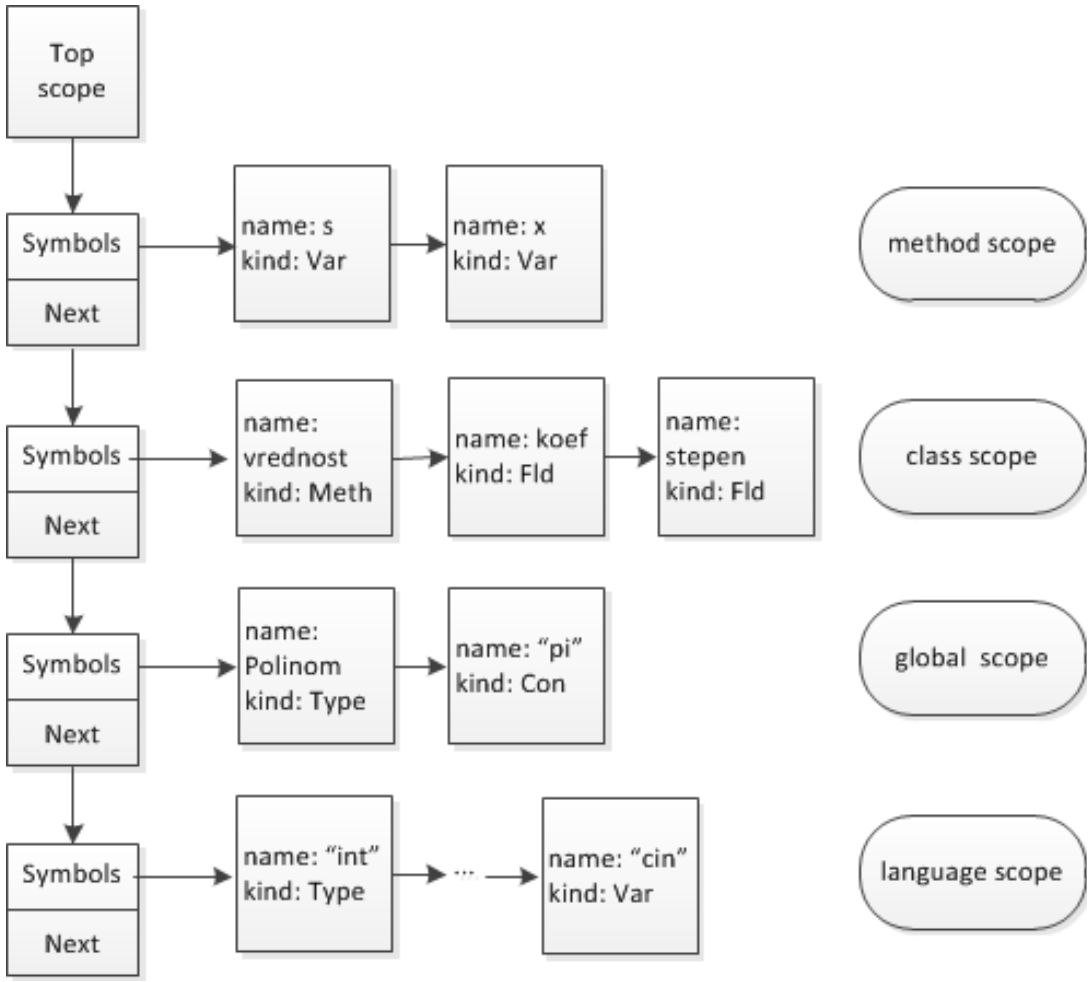
Tačka 2 posebna tabela za svaki scope

```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



Tačka 3 posebna tabela za svaki scope

```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```



Tačka 4 posebna tabela za svaki scope

```
// tacka 1
const double pi=3.14;
class Polinom {
   int stepen;
   double koef[100];
   // tacka 2
   double vrednost( double x ){
        double s=0;
        // tacka 3
        for ( int i=0;
              i<=stepen; i++)
           s = s*x+koef[i];
   }
…
};
void main() {
   Polinom p;
   // tacka 4
   …
}
```