Konstrukcija kompilatora

— Semantička analiza: Određivanje dosega —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

1 Semantička analiza **1**

2 Provera dosega **4**

2.1 Ime, doseg i tabela simbola 4

2.2 Operacije nad tabelom simbola i statička implementacija 8

2.3 Doseg i tabela simbola u OOP 9

2.4 Jednoprolazni i višeprolazni kompajleri 14

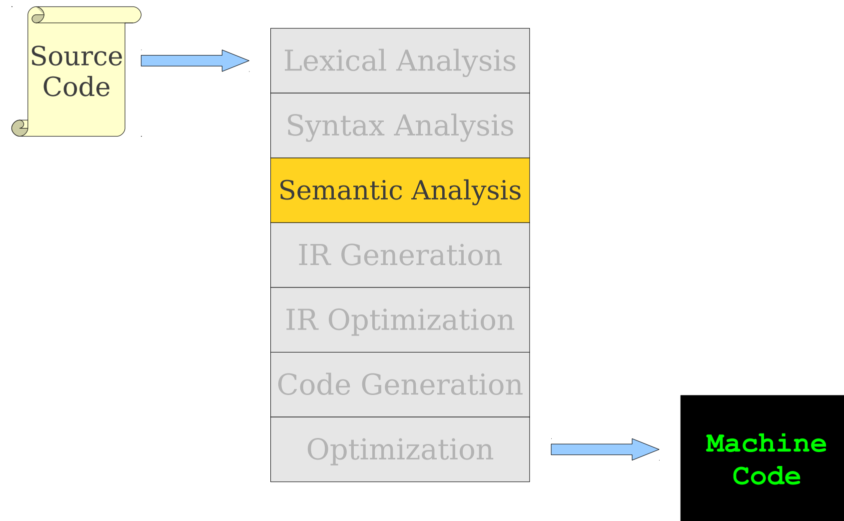
2.5 Dinamički dosezi 15

3 Zaključak i literatura **17**

1 Semantička analiza

Na slajdovima obavezno pogledati animacije koje su ovde izostavljene!

Where We Are



Primer

Postoje razne vrste semantičkih grešaka:

```
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);
    }
}
```


Razne vrste semantičkih provera

Semantičkom analizom proverava se, na primer:

- Da li su svi identifikatori deklarisani na mestima na kojima se upotrebljavaju?
- Da li se poštuju navedeni tipovi podataka?
- Da li su odnosi nasleđivanja u objektno orijetnisanim jezicima korektni?

- Da li se klase definišu samo jednom?
- Da li se metode sa istim potpisom u klasama definišu samo jednom?
- ...

Validnost vs korektnost

Ovaj program nije validan i kao takav će sigurno biti odbačen:

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

Safe; can't happen

Validnost vs korektnost

Ovaj program nije korektan, ali neće biti odbačen:

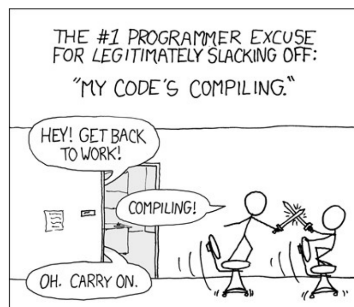
```
int Fibonacci(int n) {
    if (n <= 1) return 0;

    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



Izazovi semantičke analize

- Prikupiti i druge korisne informacije o programu koje su potrebne za kasnije faze
  - Odrediti koju promenljivu označava koja varijabla (razrešiti probleme sa dosegom)
  - Izgraditi interno predstavljanje hijerarhije nasleđivanja
  - Izbrojati koliko promenljivih je u dosegu u svakoj tački programa
- Razmotrićemo detaljnije dve vrste semantičke analize, koje se mogu implementirati rekurzivnim prolazom kroz AST:
  - Scope-Checking (provera dosega) — koji identifikator se odnosi na koji konkretan objekat (da li je svaki dentifikator deklarisan pre upotrebe?)
  - Type-Checking (provera tipova) — da li izrazi imaju validne tipove, da li parametri funkcija imaju argumente ispravnih tipova?

2 Provera dosega

2.1 Ime, doseg i tabela simbola

What's in a Name?

- The same name in a program may refer to fundamentally different things:
- This is perfectly legal Java code:

```
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A) A);
    }
}
```

What's in a Name?

- The same name in a program may refer to completely different objects:
- This is perfectly legal C++ code:

```
int Awful() {
   int x = 137;
   {
      string x = "Scope!"
      if (float x = 0)
         double x = x;
   }
   if (x == 137) cout << "Y";
}
```

Doseg

- Doseg nekog objekta (na primer, promenljive ili funkcije) je skup lokacija u programu gde se uvedeno ime može koristiti za imenovanje datog objekta
- Uvođenje nove promenljive u doseg može da sakrije ime neke prethodne promenljive
- Na koji način pratimo vidljivost promenljivih? Za to postoji **tabela simbola** (symbol table)
- Tabela simbola je preslikavanje koje za svako ime određuje čemu to ime konkretno odgovara.
- Kako se izvršava semantička analiza, tabela simbola se osvežava i menja.
  - Kako to praktično izgleda i kako se to tačno implementira?
  - Koje operacije treba da definišemo na tabeli simbola?

Određivanje dosega

Razmotrimo primer određivanja dosega po pravilu najbliže ugnježdenosti. U okviru primera imamo i globalne promenljive.

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
```

Symbol Table	
x	0
z	1

2.2 Operacije nad tabelom simbola i statička implementacija

Operacije nad tabelom simbola

- Tabela simbola je tipično implementirana kao stek kataloga (mapa)
- Svaka mapa odgovara jednom konkretnom dosegu
- Osnovne operacije su
  - Push scope — ulazak u novi doseg
  - Pop scope — napuštanje dosega, izbacivanje svih deklaracija koje je doseg sadržao
  - Insert symbol — ubacivanje novog unosa u tabelu tekućeg dosega
  - Lookup symbol — traženje čemu neko konkretno ime odgovara

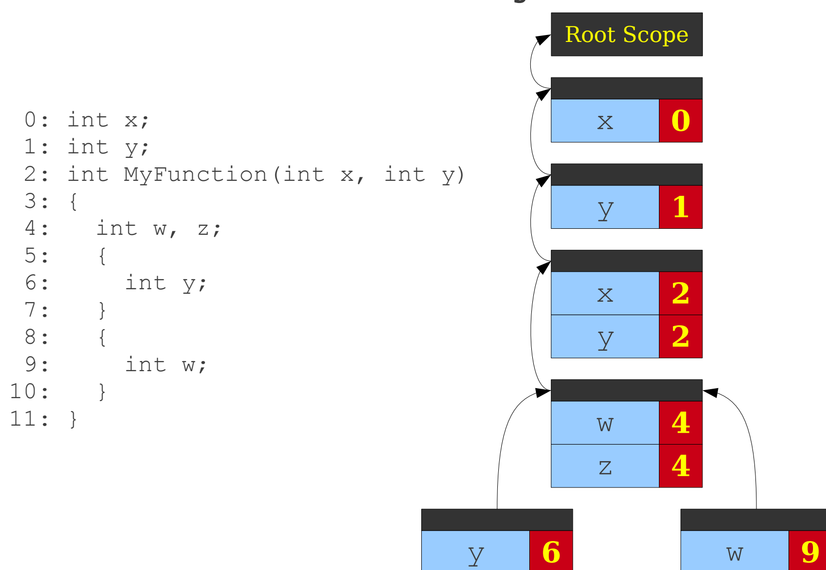
Obrada dosega

- Da bi se obradio deo programa koji kreira neki doseg (blok naredbi, poziv funkcije, klase...) potrebno je
  - Ući u doseg
  - Dodati sve deklarisane promenljive u tabelu simbola
  - Obraditi telo bloka/funkcije/klase
  - Izaći iz dosega
- Veliki deo semantičke analize se definiše na ovaj način: rekurzivnim prolaskom kroz AST

Malo drugačiji pristup praćenju dosega

- U okviru ove interpretacije, tretira se tabela simbola kao uvezana struktura dosega
- Svaki doseg čuva pokazivač na svog roditelja, ali ne i obrnuto
- Iz svake tačke programa, tabela simbola izgleda kao stek

Another View of Symbol Tables



Špageti stek

- Ovakva struktura se zove špageti stek i bolje i preciznije hvata strukturu dosega
- Špageti stek je statička struktura, dok je eksplicitni stek dinamička struktura

2.3 Doseg i tabela simbola u OOP

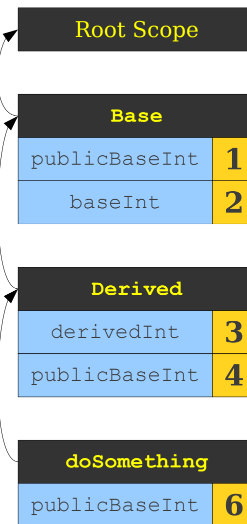
Doseg u okviru objektno-orijentisanog programiranja

- Doseg izvedene klase obično čuva link na doseg njene bazne klase
- Traženje polja klase prolazi kroz lanac dosega i zaustavlja se kada se pronađe odgovarajući identifikator ili kada se pojavi semantička greška

Scoping with Inheritance

```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

```
> 4
  2
  3
  6
```

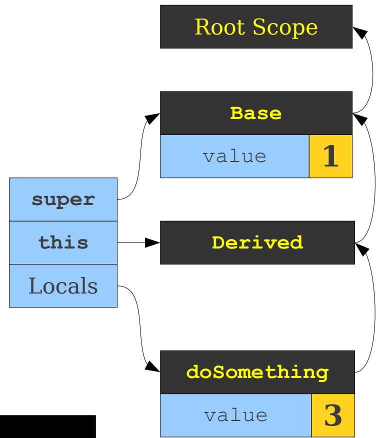


Razjašnjenje kod nasleđivanja

- Kod nasleđivanja je potrebno održavati još jednu tabelu pokazivača koja pokazuje na stek dosega
- Kada se traži vrednost u okviru specifičnog dosega, počinje se pretraga od tog konkretnog dosega
- Neki jezici omogućavaju skakanje do proizvoljne bazne klase (npr C++)

Explicit Disambiguation

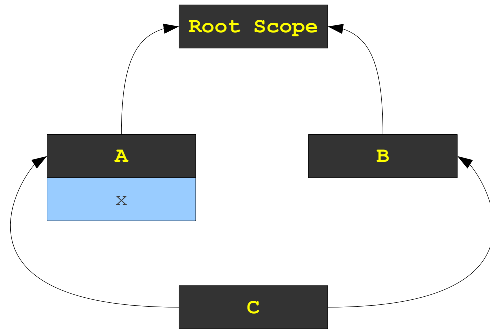
```
public class Base {
    public int value = 1;
}

public class Derived extends Base {


    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```



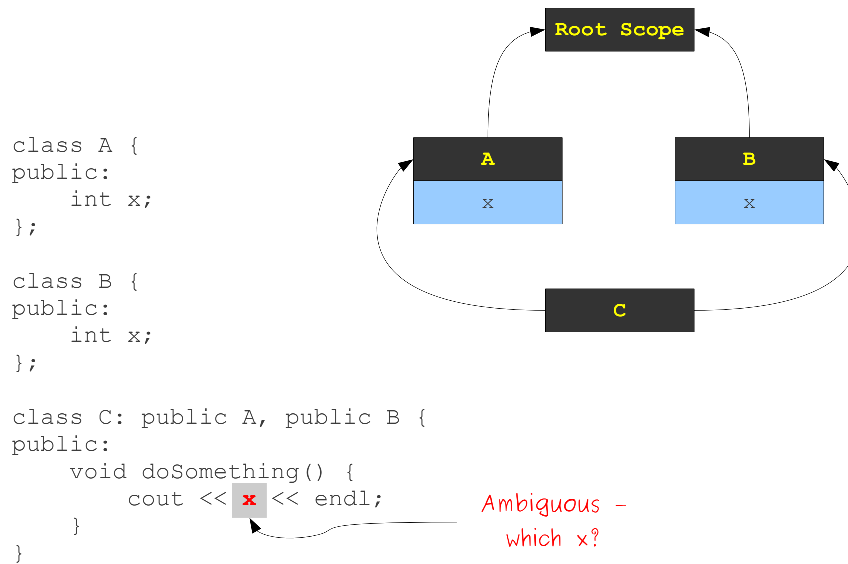

```
> 3
  1
  1
```

Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {


};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

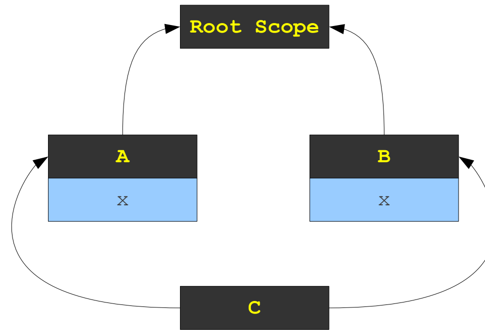


Scoping with Multiple Inheritance



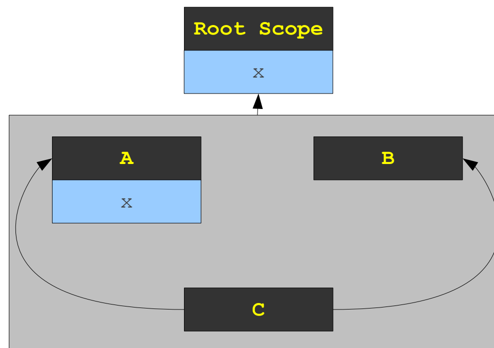
Scoping with Multiple Inheritance

```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << A::x << endl;
    }
}
```



Scoping with Multiple Inheritance

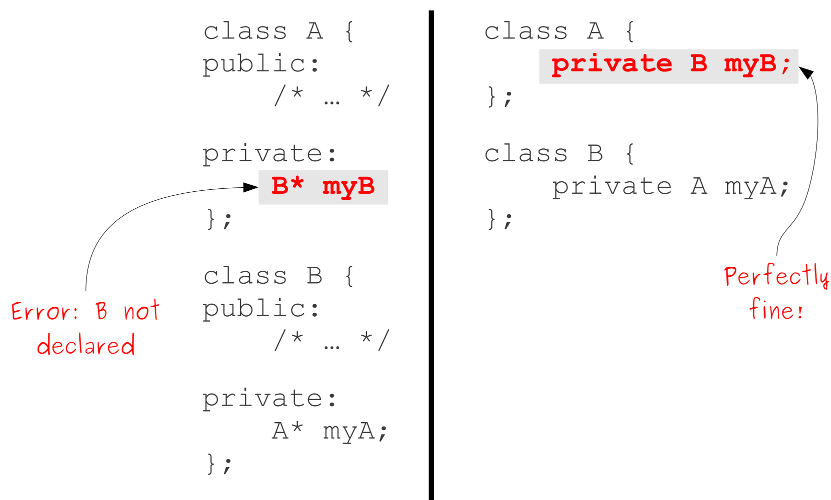
```
int x;

class A {
public:
    int x;
};

class B {


};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```



Pojednostavljena pravila dosega za C++

- U okviru klase, pretraži celu hijejrhiju da pronađeš koji skupovi imena se tu mogu naći (koristeći standardnu pretragu dosega)
- Ako se pronađe samo jedno odgovarajuće ime, onda je pretraga završena bez dvosmislenosti
- Ako se pronađe više nego jedno odgovarajuće ime, onda je pretraga dvosmislena i mora se zahtevati razrešavanje pretrage
- U suprotnom, počni ponovo pretragu ali van klase

2.4 Jednoprolazni i višeprolazni kompajleri



Jednoprolazni i višeprolazni kompajleri

- Na prethodnom primeru smo videli razlike u pravilima dosega. Suština je u tome da li se analiza može obaviti u jednom prolazu ili u više prolaza.
- Skeniranje i parsiranje je moguće uraditi u jednom prolazu.
- Neki kompajleri kombinuju skeniranje, parsiranje, semantičku analizu i generisanje koda u jednom prolazu kroz kod. To su **jednoprolazni kompajleri**.
- Većina kompajlera ipak prolazi kroz kod više puta i to su **višeprolazni kompajleri**.

Jednoprolazni i višeprolazni kompajleri

- Neki jezici su dizajnirani tako da mogu da podrže jednoprolazne kompajlere (npr C ili C++)
- Neki jezici su dizajnirani tako da zahtevaju višeprolazne kompajlere (npr Java)
- Većina modernih kompajlera koristi veoma veliki broj prolaza kroz kod

Pravila dosega u višeprolaznim kompajlerima

- Prvi prolaz: kompletno parsiranje ulaznog koda i kreiranje ASTa
- Drugi prolaz: prolazak kroz AST i skupljanje informacija o klasama
- Treći prolaz: prolazak kroz AST i provere raznih osobina
- Prolazi se mogu kombinovati, ali su logički to nezavisne celine

2.5 Dinamički dosezi

Dinamički dosezi

- Do sada smo videli primere statičkog određivanja dosega, tj određivanje dosega u fazi kompilacije
- Neki jezici koriste dinamičko određivanje dosega, koje se sprovodi u fazi izvršavanja: ime odgovara varijabli sa tim imenom koja je najbliže ugnježdena u fazi izvršavanja

Dynamic Scoping

```
int x = 137;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**

```
> 179
>
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	**137**
y	**42**
x	**0**

```
> 179
> 42
>
```

Dynamic Scoping

```
int x = 137;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function3();
```

Symbol Table	
x	137
y	42
y	0
x	0

```
> 179
> 42
> 0
>
```

Dinamički dosezi

- Primeri jezika sa dinamičkim dosezima: Perl, Common Lisp
- Implementacija dinamičkih dosega uključuje čuvanje tabele simbola u fazi izvršavanja
- Obično je to manje efikasno od statičkog određivanja dosega jer kompajleri ne mogu da hardkoduju lokacije promenljivih već imena moraju da razrešavaju u fazi izvršavanja

3 Zaključak i literatura

Zaključak

- Semantička analiza provera da li sintaksno ispravni program korektrno formirani i računa dodatne informacije o značenju programa
- Proveravanje dosega utvrđuje na koje objekte ili klase se referiše imenima u programu
- Provera dosega se obično radi sa tabelom simbola koja se implementira ili kao stek ili kao špageti stek

Zaključak

- U objektno orijentisanim programima, doseg izvedenih klasa je obično smešten u doseg njihovih baznih klasa
- Neki semantički analizatori rade u višestrukim prolazima kroz AST sa ciljem da skupe više informacija o programu
- U dinamičkom određivanju dosega, u fazi izvršavanja programa se određuje na šta se odnosi koje ime
- Ukoliko imamo višestruko nasleđivanje, ime može da se traži kroz različite putanje

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/