Konstrukcija kompilatora

— Izvršno okruženje —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

1 Uvod **2**

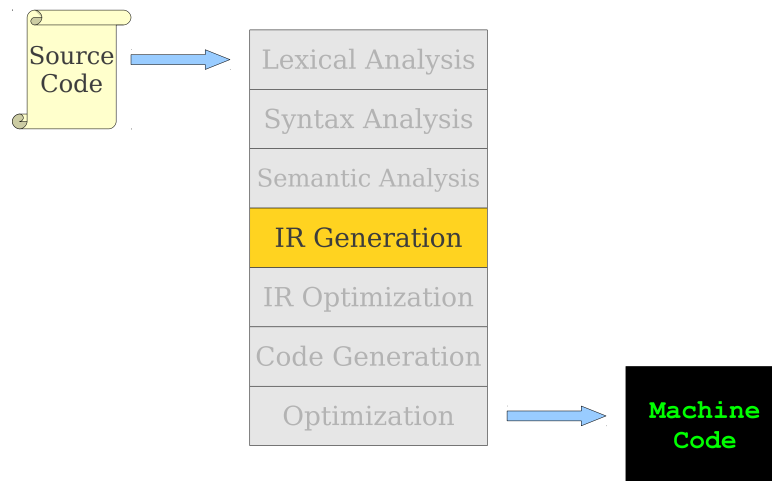
2 Podaci **4**
2.1 Enkodiranje osnovnih tipova 5
2.2 Nizovi 5
2.3 Višedimenzioni nizovi 5

3 Funkcije **6**
3.1 Aktivaciono stablo 7
3.2 Zatvorenja 8
3.3 Korutine 10
3.4 Stek izvršavanja 11

4 Objekti **13**
4.1 Strukture, objekti i nasleđivanje 13
4.2 Funkcije članice klasa 16
4.3 Tabela virtuelnih funkcija i tabela metoda 18
4.4 Višestruko nasleđivanje i interfejsi 21
4.5 Dinamička provera tipova 22

5 Literatura **23**

Na slajdovima obavezno pogledati animacije koje su ovde izostavljene!



1 Uvod

Generisanje međureprezentacije (IR)

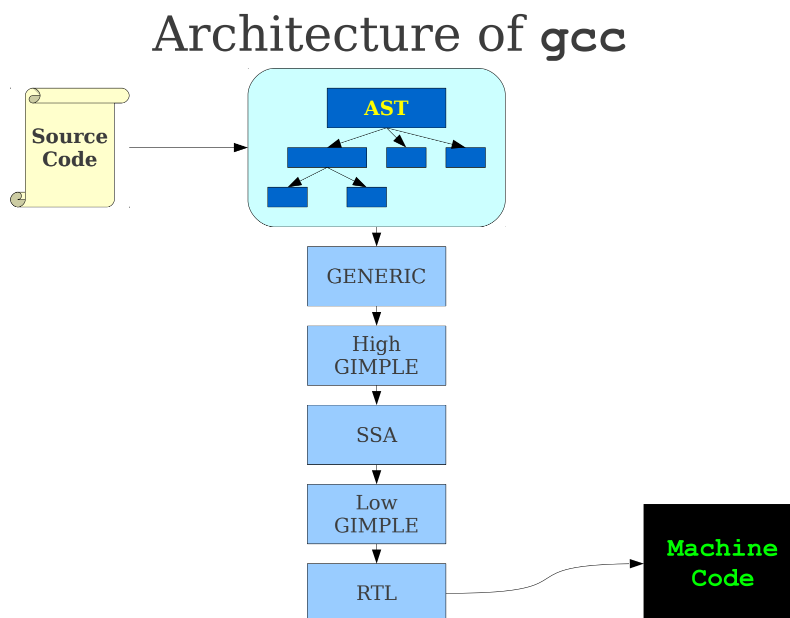
- Poslednja faza prednjeg dela kompajlera
- Cilj: prevesti program u format koji očekuje srednji/zadnji deo kompajlera
- Generisani kod ne mora da bude optimizovan, to se ostavlja za kasnije faze
- Generisani kod nije asembler, i to se ostavlja za kasnije faze

Zašto generišemo međukod

- Pojednostavljujemo optimizacije:
  - Mašinski kod ima razna ograničenja koja sprečavaju optimizaciju
  - Rad sa međureprezentacijom čini optimizaciju lakšom i čistijom
- Omogućavamo vezu većeg broja prednjih delova sa jednim zadnjim delom
  - I llvm i gcc mogu kompilirati više programskih jezika: svaki prednji deo prevodi kod u međureprezentaciju
- Omogućava emitovanje izvršnog koda za različite arhitekture, tj omogućava vezu većeg broja zadnjih delova sa jednim prednjim delom.
  - Veći deo optimizacija se izvrši nad srednjim delom pre emitovanja koda za ciljnu arhitekturu.

Dizajniranje dobrog IR

- IR je kao sistem tipova: veoma je teško dizajnirati dobar IR
- Potrebno je balansirati između potreba jezika visokog nivoa (izvornog koda) i potreba jezika niskog nivoa (asembler ciljne arhitekture)
- Ukoliko je previše visokog nivoa, ne mogu se optimizovati različiti implementacioni detalji
- Ukoliko je previše niskog nivoa, ne može se koristiti znanje sa višeg nivoa za korišćenje agresivnih optimizacija
- Obično postoje više različitih IRova u jednom kompajleru



IR visokog nivoa

- Primeri: Java bytecode, CPython bytecode, LLVM IR, Microsoft CIL.
- U velikoj meri zadržavaju strukturu programa.
- Omogućavaju i kompilaciju, i JIT kompilaciju i interpretiranje

Izvršno okruženje

- Kako da dizajniramo dobar IR?
- Kako da implementiramo osobine jezika u mašinskom kodu?
- Koje strukture podataka su nam potrebne?

Važna dualnost

- Programski jezici sadrže strukture visokog nivoa:
  - Funkcije
  - Objekte
  - Izuzetke
  - Dinamičko određivanje tipova
  - Lenja evaluacija
  - itd.
- Računar operše samo u terminima nekoliko primitivnih operacija:
  - Aritmetičke operacije
  - Premeštanje podataka
  - Skokovi
- **Potrebna nam je reprezentacija koncepata visokog nivoa korišćenjem dostupnih struktura niskog nivoa mašine**

Izvršno okruženje

- **Izvršno okruženje predstavlja skup struktura podataka koje se održavaju u fazi izvršavanja sa ciljem omogućavanja koncepata jezika visokog nivoa** (npr stek, hip, statički prostor, tabela vrituelnih funkcija i slično)
- Izvršno okruženje zavisi od osobina izvornog i ciljanog jezka
- Generisanje IR zavisi od toga kako se postavi izvršno okruženje

Izvršno okruženje

- Potrebno je razmotriti:
  - Kako izgledaju objekti u memoriji?
  - Kako izgledaju funkcije u memoriji?
  - Gde se smeštaju objekti i funkcije u memoriji?
- Ne postoji ispravan odgovor na ova pitanja već samo različite opcije sa svojim prednostima i manama koje je potrebno izbalansirati

2 Podaci

Prikaz podataka

- Kako izgledaju različiti tipovi u memoriji?
- Mašine obično podržavaju samo ograničen skup tipova
  - Celobrojne vrednosti fiksirane širine: 8-bitni, 16-bitni, 32-bitni, 64-bitni signed, unsigned i slično
  - Brojeve u pokretnom zarezu 32-bitne, 64-bitne, 80-bitne IEEE 754
- Na koji način se enkodiraju objekti korišćenjem raspoloživih tipova?

2.1 Enkodiranje osnovnih tipova

Enkodiranje osnovnih tipova

- Osnovni celobrojni tipovi, kao što su byte, char, short, int, long, unsigned, uint16_t i slično obično se preslikavaju direktno u odgovarajuće mašinske tipove
- Osnovni realni tipovi, kao što su float, double, long double se takođe obično preslikavaju u odgovarajuće mašinske tipove
- Pokazivači se obično implementiraju kao celobrojni tip koji čuva memorijske adrese
- Veličina celobrojnih tipova zavisi od arhitekture računara
- Nizovi? Mogu se predstaviti na razne načine

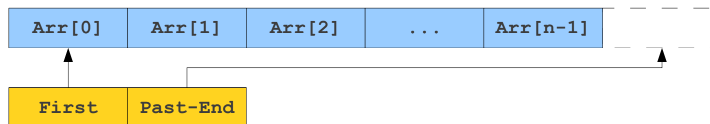
2.2 Nizovi

Encoding Arrays

- C-style arrays: Elements laid out consecutively in memory.

- Java-style arrays: Elements laid out consecutively in memory with size information prepended.

- D-style arrays: Elements laid out consecutively in memory; array variables store pointers to first and past-the-end elements.



2.3 Višedimenzioni nizovi

Višedimenzioni nizovi

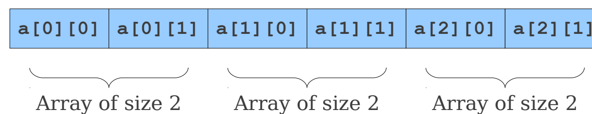
- Višedimenzioni nizovi se obično predstavljaju kao nizovi nizova
- Oblik zavisi od nizova koji se koriste

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```

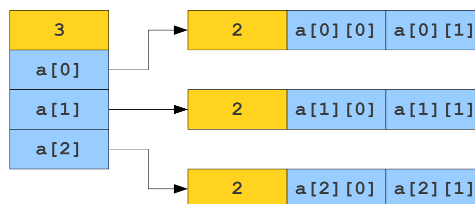
How do you know where to look for an element in an array like this?



Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

```
int[][] a = new int [3][2];
```



3 Funkcije

Predstavljanje funkcija

- Potrebno je odgovoriti na veliki broj potanja:
  - Kako izgleda dinamičko izvršavanje funkcija?
  - Gde se izvršivi kod za funkcije nalazi?
  - Kako se prosleđuju parametri u/iz funkcije?
  - Gde se čuvaju lokalne promenljive?
- Odgovori zavise od željenih karakteristika jezika

Stek - podsetnik

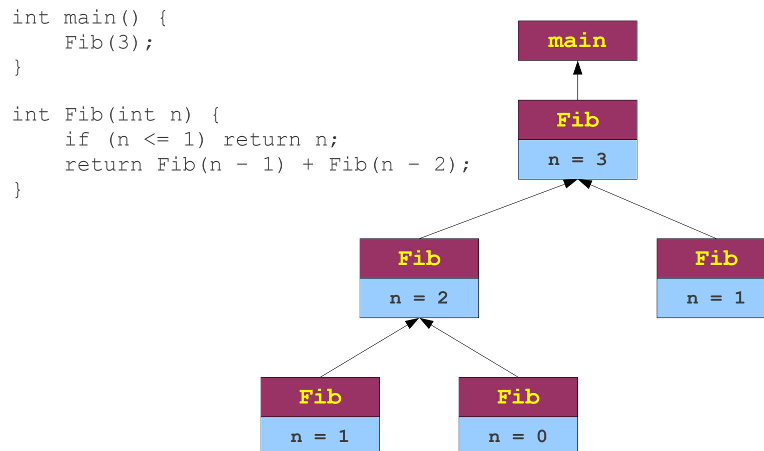
- Pozivi funkcija se obično implementiraju korišćenjem steka aktivacionih slogova (ili stek okvira)
- Poziv funkcije gura novi aktivacioni slog na stek
- Povratak iz funkcije skida aktivacioni slog sa vrha steka
- Pitanja: Da li ovo uvek funkcioniše?

3.1 Aktivaciono stablo

Aktivaciono stablo

- Aktivaciono stablo je stablo struktura koje predstavljaju sve pozive funkcija prilikom nekog konkretnog izvršavanja programa
- Aktivaciono stablo zavisi od ponašanja programa, ne može se uvek odrediti u fazi kompilacije
- Statički ekvivalent je graf poziva funkcija
- Svaki aktivacioni slog čuva kontrolni link prema aktivacionom slogu koji ga je inicirao (pozvao)

Activation Trees



Osobine stabla u fazi izvršavanja

- Aktivaciono stablo je špageti stek
- Stek u fazi izvršavanja je optimizacija špageti steka (?)
- Zašto (da li) možemo uvek da optimizujemo špageti stek? Pretpostavke:
    1. Jednom kada se funkcija vrati, njen aktivacioni slog ne može da bude ponovo referenciran. Prema tome, nema potrebe da čuvamo stare čvorove u aktivacionom stablu
    2. Svaki aktivacioni slog je ili završio izvršavanje ili je predak tekućeg aktivacionog sloga. Nema potrebe da čuvamo više grani izvršavanja u jednom trenutku.
- Međutim, ove pretpostavke ne moraju uvek da budu tačne!

3.2 Zatvorenja

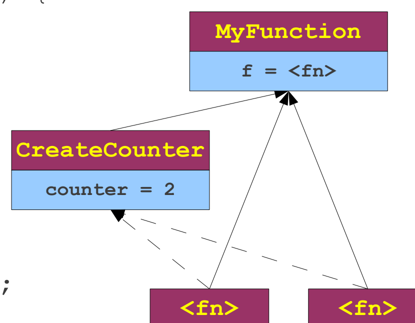
Pretpostavka 1

- **Jednom kada se funkcija vrati, njen aktivacioni slog ne može da bude ponovo referenciran.**
- Ova pretpostavka ne važi za zatovrenja (closures)!
- Programski jezici koji podržavaju zatvorenja: Scheme, JavaScript, Swift, Ruby, Python...

- Zatvorenje Z je ugnežđena funkcija u funkciju F koja ima mogućnost pristupa slobodnim promenljivama iz funkcije F (promenljivama koje nisu vezane za lokalni doseg funkcije Z), pri čemu je funkcija F završila sa svojim izvršavanjem. Osnovne karakteristike zatvorenja su:
  - to je ugnežđena funkcija
  - ona ima pristup slobodnim promenljivama iz spoljašnjeg dosega
  - ona je vraćena kao provratna vrednost funkcije u koju je ugnežđena

Closures

```
function CreateCounter() {
    var counter = 0;
    return function() {
        counter ++;
        return counter;
    }
}

function MyFunction() {
    f = CreateCounter();
    print(f());
    print(f());
}
```




```
> 1
  2
```

Control and Access Links

- **Kontrolni link** funkcije je pokazivač na funkciju koja ju je pozvala, koristi se da se odredi gde se izvršavanje nastavlja kada funkcija završi sa radom
- **Pristupni link** funkcije je pokazivač prema aktivacionom slogu u kojem je funkcija kreirana. Koriste ga ugnežđene funkcije da odrede lokaciju promenljivih u spoljašnjem dosegu

Zatvorenja i stek

- Jezici koji podržavaju zatvorenja obično **nemaju stek izvršavanja**
- Aktivacioni slogovi se tipično dinamički alociraju i budu oslobođeni sakupljačem otpadaka
- Interesantan izuzetak: gcc dozvoljava ugnežđene funkcije, iako koristi stek izvršavanja. Ipak, te funkcije nisu zatvorenja i ponašanje je nedefinisano ako ugnežđena funkcija pristupa podacima svoje funkcije kada ona završi sa radom

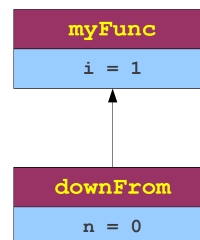
3.3 Korutine

Pretpostavka 2

- **Svaki aktivacioni slog je ili završio izvršavanje ili je predak tekućeg aktivacionog sloga.**
- Podsetimo se:
  - **Podrutine** su funkcije koje, kada se pozovu, izvršavaju se do završetka svog posla i vraćaju kontrolu funkciji pozivaocu (odnos master/slave između pozivaoca i pozvane funkcije)
  - **Korutine** su funkcije koje, kada se pozovu, urade deo posla, i onda vraćaju kontrolu pozivajućoj funkciji, ali mogu da kasnije ponovo nastave sa radom od tog istog mesta (odnos peer/peer između pozivaoca i pozvane funkcije)
  - Podrutine su specijalni slučaj korutina
  - Korutine se nekada koriste i za brzi izlazak iz rekurzije (preskakanje stek okvira)
- Ova pretpostavka ne važi za korutine

Coroutines

```
def downFrom(n):
   while n > 0:
      yield n
      n = n - 1


def myFunc():
   for i in downFrom(3):
       print i
```




```
> 3
  2
  1
```

Korutine i stek izvršavanja

- Korutine često ne mogu da budu implementirane preko steka izvršvanja (šta se dešava ako funkcija ima nekoliko korutina koje se izvršavaju zajedno sa njom?)
- Malo jezika podržava korutine (npr Python)

Zaključak

- Čak i najosnovniji koncepti poput steka mogu da budu veoma kompleksni
- Kada se dizajnira kompajler ili programski jezik, mora da se ima na umu na koji način osobine programskog jezika utiču na izvršno okruženje

3.4 Stek izvršavanja

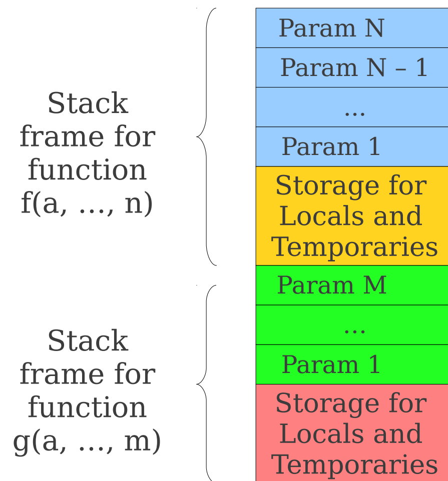
Stek izvršavanja

- U nastavku ćemo koristiti primere koji podrazumevaju korišćenje steka izvršavanja i nekih najčešćih odluka: Svaki aktivacioni slog mora da čuva
  - Sve svoje parametre
  - Sve svoje lokalne promenljive
  - Sve privremene promenljive uvedene sa IR generatorom
- Gde se sve te promenljive smeštaju?
- Ko alocira prostor za njih?

Izgled stek okvira

- **Logički izgled stek okvira** obično kreira IR generator. Logički izgled obično ignoriše detalje o mašinski specifičnim konvencijama poziva funkcija
- **Fizički izgled stek okvira** kreira generator koda. On je zasnovan na logičkom izgledu koji postavlja IR generator i uključuje *frame pointere*, *caller-saved* registre i slične detalje

A Logical Decaf Stack Frame



IR pozivna konvencija

- Pozivaoc je odgovoran za stavljanje na stek i skidanje sa steka prostora za argumente pozivajuće funkcije
- Pozivajuća funkcija je odgovorna za stavljanje i skidanje sa steka svojih lokalnih promenljivih i privremenih promenljivih

Prenos parametara

- Dva česta pristupa
  - Call-by-value — parametri su kopije zadatih vrednosti
  - Call-by-reference — parametri su pokazivači na vrednosti koje su zadate kao parametri
- Postoje i razni drugi načini prenosa parametara u funkciju
  - Na primer, JavaScript — funkcije mogu da budu pozvane sa bilo kojim brojem argumenata: parametri se inicijalizuju zadatim odgovarajućim argumentima, ili su nedefinisani ako nije zadato dovoljno arugmenata

Rezime poziva funkcija

- Stek izvršavanja je optimizacija aktivacionog stabla špageti steka
- Većina jezika koriste stek izvršavanja, ali neki jezici imaju osobine koje onemogućavaju ovu optimizaciju

- Aktivacioni slogovi logički čuvaju kontrolni link na funkciju pozivaoca kao i pristupni link na funkciju koja ju je kreirala
- Za svaki programski jezik važno je definisati konvenciju pozivanja funkcija, tj obaveze funkcije pozivaoca i obaveze funkcije koja je pozvana
- Najčešće pozivaoc upravlja prostorom za parametre dok pozvana funkcija upravalja prostorom za svoje lokalne i privremene promenljive
- Poziv po vrednosti i poziv po referenci mogu da se implementiraju kopiranjem i pokazivačima

4 Objekti

Implementacija objekata

- Implementacija objekata je veoma složena: veoma je teško napraviti izražajan i efikasan objektno-orijentisan jezik
- Koncepti koje je teško implementirati efikasno su
  - Dinamičko raspoređivanje (virtualne funkcije)
  - Intefejsi i višestruko nasleđivanje
  - Dinamičku proveru tipova (*instanceof*)

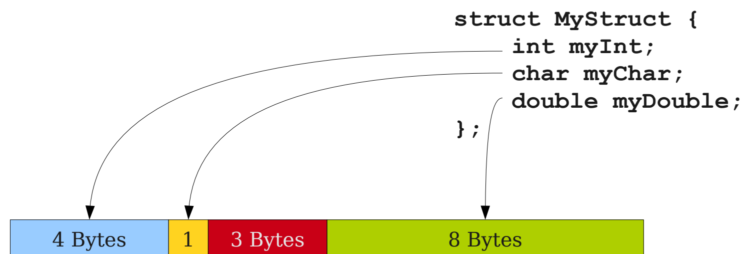
4.1 Strukture, objekti i nasleđivanje

Strukture

- Počnimo od C-ovskih struktura
- Struktura je tip koji sadrži kolekciju imenovanih vrednosti
- Najčešći pristup, postaviti svako polje da leži redom kojim su polja deklarisana

Encoding C-Style `struct`s

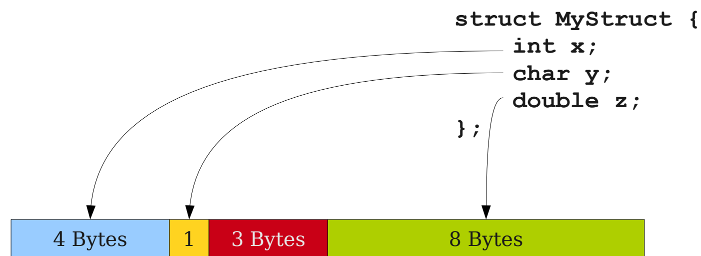
- A **`struct`** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.



Pristup poljima strukture

- Jednom kada se objekat postavi u memoriju, on je samo serija bajtova
- Potrebno je znati gde tražiti neko konkretno polje?
- Ideja: Čuvati internu tabelu u okviru kompajlera koja sadrži pomeraje za svako polje
- Da bi se pronašlo željeno polje, počni od osnovne adrese objekta i pomeri je unapred za odgovarajući pomeraj (engl. offset)

Field Lookup




```
MyStruct* ms = new MyStruct;
ms->x = 137;   store 137   0 bytes after ms
ms->y = 'A';   store 'A'   4 bytes after ms
ms->z = 2.71   store 2.71  8 bytes after ms
```

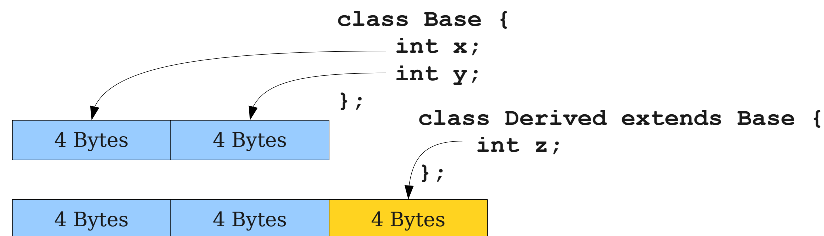
OOP bez metoda

- Razmotrimo dve klase bez metoda

```
class Base {
  int x;
  int y;
}

class Derived extends Base {
  int z;
}
```

- Kako će izvedena klasa da izgleda u memoriji?

Field Lookup With Inheritance




```
Base ms = new Base;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms

Base ms = new Derived;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms
```

Jednostruko nasleđivanje

- Pojednostavljeno, izgled memorije za izvedenu klasu D je zadat kao izgled memorije za baznu klasu B za kojom sledi izgled memorije za preostale članove klase D
- Objašnjenje: pokazivač na baznu klasu koji pokazuje na D i dalje vidi objekat B na početku memorije
- Operacije koje se izvode na objektu D kroz pokazivač na objekat B su garantovano ispravne, nema potrebe da se proverava na šta tačno B dinamički ukazuje

4.2 Funkcije članice klasa

Funkcije članice klasa (metode)

- Funkcije članice klasa su kao i obične funkcije, ali sadrže dve komplikacije:
  - Kako znati za koji objekat je funkcija vezana?
  - Kako znati koju funkciju pozvati u fazi izvršavanja (dynamic dispatch)?
- U okviru funkcije članice klase, ime *this* se koristi za tekući objekat. Ova informacija treba da se iskomunicira funkciji
- Ideja: tretiraj *this* kao implicitni prvi parametar funkcije. Svaka funkcija koja ima n-argumenata će u stvarnosti imati n+1 argument pri čemu je prvi parametar pokazivač *this*

`this` is Clever

```
class MyClass {
    int x;
}
void MyClass_myFunction(MyClass this, int arg){
    this.x = arg;
}

MyClass m = new MyClass;
MyClass_myFunction(m, 137);
```

Pokazivač *this*

- Kada se generiše kod, za poziv funkcije članice prosleđuje se još jedan parametar *this* koji predstavlja odgovarajući objekat
- U okviru funkcije članice, *this* je samo još jedan parametar funkcije
- Kada se implicitno referiše na neko polje od this, koristi se ovaj dodatni parametar kao objekat u kojem treba da se traži odgovarajuće polje

Dinamičko određivanje poziva

- Dinamičko određivanje poziva znači odabir poziva funkcije u fazi izvršavanja u zavisnosti od tekućeg dinamičkog tipa objekta
- Kako postaviti izvršno okruženje tako da se ovo omogući na efikasan način?
- Jednostavna ideja: u fazi kompilacije napravi listu svih klasa i generiši naredni kod

```
if (the object has type A)
  call A's version of the function
else if (the object has type B)
  call B's version of the function
...
else if (the object has type N)
  call N's version of the function.
```

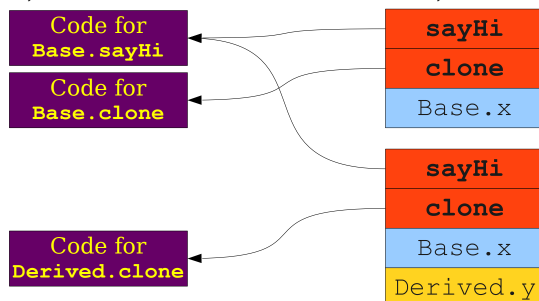
Analiza ideje

- Prethodna ideja ima nekoliko ozbiljnih problema:
  - Veoma je spora: broj provera je jednak O(C) gde je C broj klasa na koje se ovo razrešavanje odnosi. Takođe, što više klasa imamo, to je ovo rešenje sporije
  - Nije uvek ostvarivo rešenje: nije ostvarivo ako imamo više izvornih fajlova, ili ako postoji podrška za dinamičko učitavanje klasa
- Umesto toga, ideja je da se za funkcije napravi sličan pristup kao za podatke

More Virtual Function Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base {
    int y;


    Derived clone() {
        return new Derived;
    }
}
```



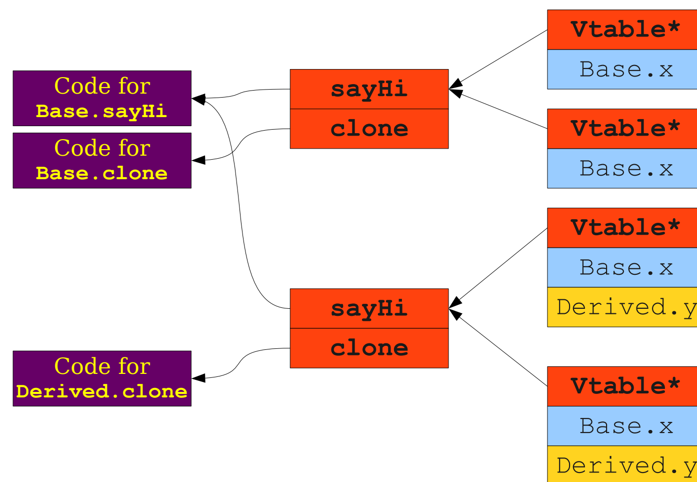
4.3 Tabela virtuelnih funkcija i tabela metoda

Tabela virtuelnih funkcija

- Tabela virtuelnih funkcija *vtable* je niz pokazivača na implementacije funkcija članica neke klase
- Da bi se pozvala funkcija članica klase
  - Odredi statički indeks u virtuelnoj tabeli
  - Prati pokazivač koji se nalazi na tom indeksu u okviru vtable objekta da bi došao do koda funkcije
  - Pozovi tu funkciju

Analiza ovog pristupa

- Prednosti: vreme da se odredi funkcija koja će biti pozvana je O(1)
- Mane: Objekti su veći jer svaki objekat mora da ima prostor da skladišti O(M) pokazivača, gde je M broj funkcija članica. Zbog toga i kreiranje postaje sporije.



Dinamičko rešavanja u vremenu O(1)

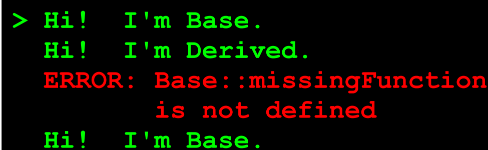
- Kreira se jedinstvena instaca vtable za svaku klasu
- Svaki objekat čuva pokazivač na svoj vtable
- Svaki objekat može da prati taj pokazivač u vremenu O(1) i može da prati index u tabeli u vremenu O(1)
- Za svaki novi objekat postavljanje pokazivača na vtable se može uraditi u O(1) vremenu
- Povećava se veličina svakog objekta za O(1)
- **Ovo rešenje se koristi u većini C++ i Java implementacija**

Zahtevi

- Za prethodno rešenje napravljene su implicitne pretpostavke o jeziku koje dozvoljavaju da vtable radi korektno:
  - Svi metodi su poznati statički: možemo utvrditi u fazi kompilacije koji metod je planiran na mestu poziva (čak i kada nismo sigurni koji tačno će metod biti pozvan)
  - Jednostruko nasleđivanje: ne moramo da brinemo o građenju jedinstvene vtable tabele prilikom višestrukog nasleđivanja
- Nasleđivanje se može implementirati i na drugačiji način

Inheritance in PHP

```
class Base {
    public function sayHello() {
        echo "Hi!  I'm Base.";
    }
}

class Derived extends Base {
    public function sayHello() {
        echo "Hi!  I'm Derived.";
    }
}

$b = new Base();
$b->sayHello();

$d = new Derived();
$d->sayHello();

$b->missingFunction();

$fnName = "sayHello";
$b->$fnName();
```

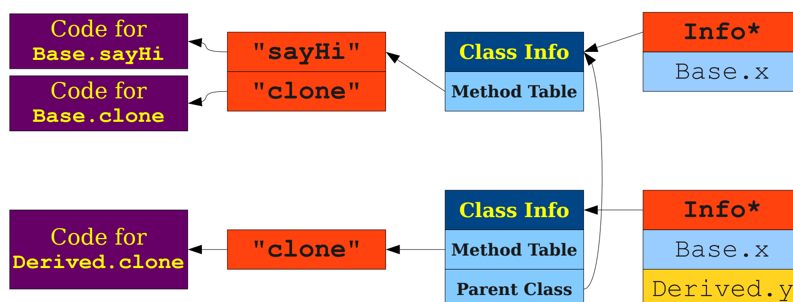


Druga rešenja

- Neki programski jezici ne mogu da rade sa virtuelnim tabelama — na primer PHP
  - Call-by-string onemogućava vtable optimizaciju — nemoguće je statički odrediti sadržaj stringa, bilo bi potrebno odrediti index u fazi izvršavanja
  - Nemamo statičke informacije o objektu — nemoguće je odrediti da li neki metod uopšte postoji
  - eval ključna reč izvršava proizvoljan PHP kod, može da uvede nove klase ili metode

Inheritance without Vtables

```
class Base {                          class Derived extends Base {
    int x;                                int y;
    void sayHi() {                        Derived clone() {
        Print("Hi!");                        return new Derived;
    }                                     }
    Base clone() {                    }
        return new Base;
    }
}
```



Opšti okvir nasleđivanja

- Svaki objekat čuva pokazivač na deskriptor svoje klase
- Svaki deskriptor klase čuva
  - Pokazivač na deskriptor bazne klase
  - Pokazivač na tabelu metoda
- Da bi se pozvao metod
  - Prati pokazivač do tabele metoda
  - Ako metod postoji, pozovi ga
  - U suprotnom, navigiraj na baznu klasu i ponovi postupak
- Ovo je sporo, jer pretražujemo tabelu stringova, ali se često može optimozovati

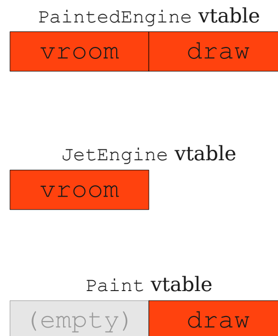
4.4 Višestruko nasleđivanje i interfejsi

Višestruko nasleđivanje i interfejsi

- Šta se dešava kada imamo višestruko nasleđivanje i interfejse?
- Kako i da li se mogu koristiti virtuelne tabele?

Vtables and Interfaces

```
interface Engine {
    void vroom();
}
interface Visible {
    void draw();
}
class PaintedEngine implements Engine, Visible {
    void vroom() { /* … */ }
    void draw() { /* … */ }
}
class JetEngine implements Engine {
    void vroom() { /* … */ }
}
class Paint implements Visible {
    void draw() { /* … */ }
}

Engine e1  = new PaintedEngine;
Engine e2  = new JetEngine;
e1.vroom();
e2.vroom();
Visible v1 = new PaintedEngine;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```



Višestruko nasleđivanje i interfejsi

- Višestruko nasleđivanje i interfejsi komplikuju izgled virtuelne tabele jer zahtevaju da metodi imaju konzistentne pozicije kroz sve virtuelne tabele: može da se desi da virtuelna tabela ima neiskorišćene unose (prazna mesta)
- Zbog toga, prilikom višestrukog nasleđivanja ili interfejsa, obično se ne koriste čiste virtuelne tabele
- Postoje različiti načini da se ovo efikasno implementira, jedan način je hibridni pristup: korišćenje virtuelnih tabela za standardno nasleđivanje, a za interfejse koristiti opisani metod zasnovan na poređenju stringova
- U nekim jezicima može se koristiti optimizacija poređenja stringova pravljenjem heš tabela (kako bi se smanjilo vreme pronalaženja odgovarajuće funkcije)

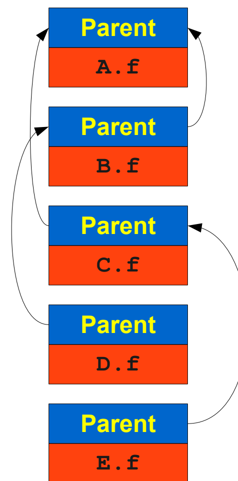
4.5 Dinamička provera tipova

Implementiranje dinamičkih provera tipova

- Mnogi jezici imaju potrebu za nekom vrstom provere dinamičkih tipova: Javin metod *instanceof*, u C++-u metod *dynamic_ cast*, jezici koji su dinamički tipizirani
- Ono što možemo da želimo da odredimo je da li je dinamički tip pretvoriv u neki drugi tip (odnos $\leq$)
- Kako to implementirati?

A Pretty Good Approach

```
class A {
   void f() {}
}

class B extends A {
   void f() {}
}

class C extends A {
   void f() {}
}

class D extends B {
   void f() {}
}

class E extends C {
   void f() {}
}
```



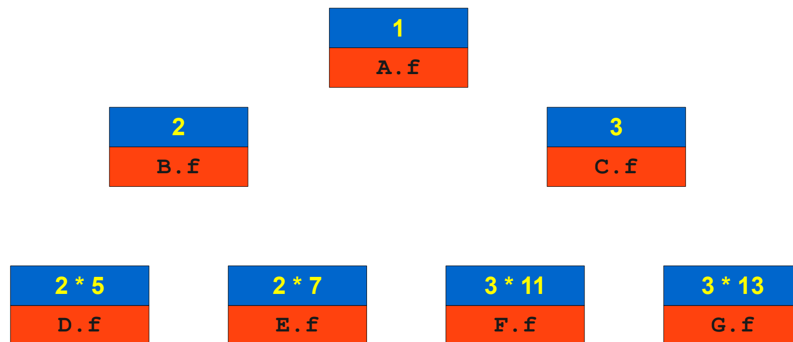
Jednostavna dinamička provera tipova

- Implementirati da svaka virtuelna tabela čuva i pokazivač na svoju osnovnu klasu
- Provera da li se objekat može pretvoriti u neki tip S u fazi izvršavanja se svodi na praćenje pokazivača na roditelja u okviru virtuelne tabele sve dok se ne naiđe na tip S ili dok se ne dođe do tipa koji nema roditelja
- Vreme izvršavanja ove provere je O(d) gde je d dubina hijerarhije klasa
- Da li se ovo može uraditi brže?

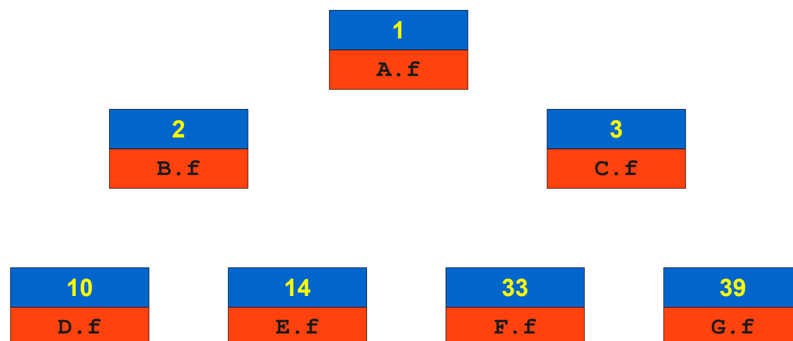
Ideja

- Postoji fantastična ideja provere pretvorivosti objekata u vremenu O(1), pod pretpostavkom da postoji konstantan i ne previše veliki broj klasa u hijerarhiji, kao i da su svi tipovi poznati statički
- Ideja je zasnovana na činjenici: objekat koji je statički tipa A je u fazi izvršavanja tipa B samo ako je A pretvorivo u B, tj $A \leq B$

A Marvelous Idea

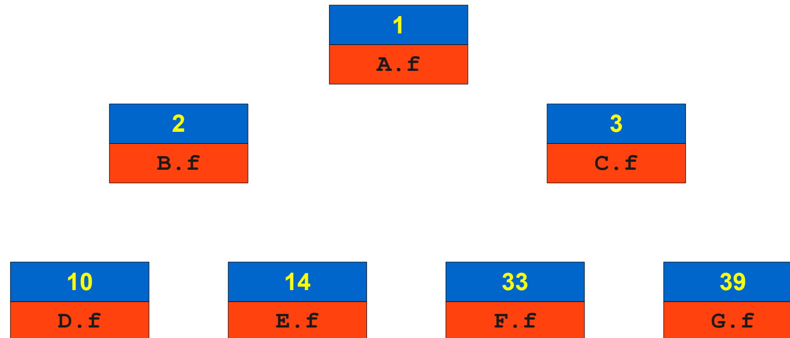


A Marvelous Idea




```
A myObject = /* … */
if (myObject instanceof C) {
    /* … */
}
```

A Marvelous Idea




```
A myObject = /* … */
if (myObject->vtable.key % 3 == 0) {
    /* … */
}
```

Dinamičko određivanje tipova kroz proste brojeve

- Dodeli svakoj klasi jedinstven prost broj (mogu se koristiti isti prosti brojevi kroz različite nezavisne hijerarhije)
- Postavi ključ svake klase da bude proizvod njenog prostog broja i svih prostih brojeva njenih nadklasa
- Da bi se proverilo u fazi izvršavanja da li se objekat može pretvoriti u tip T
  - Pogledaj ključ objekta
  - Ako je ključ od T deljiv sa ključem objekta, onda se objekat može konvertovati u T
  - Ako ključ nije deljiv, onda ne može
- Ako se proizvod dva prosta broja može smestiti u integer, ova provera se može uraditi u vremenu O(1)
- Metod se može koristiti i kod višestrukog nasleđivanja (neke implementacije C++ ovo koriste)

Zaključak

- Važno je razumeti na koji način se karakteristike višeg programskog jezika mogu implementirati strukturama koje su na raspolaganju mašinama
- Posebno bitne odluke su u okviru načina rešavanja dinamičkog poziva funkcija gde postoje dva pristupa, prvi zasnovan na virtuelnim tabelama i drugi zasnovan na pretraživanju tabela stringova
- Takođe, važno je efikasno implementirati i dinamičku proveru tipova

5 Literatura

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/