Compilation

0368-3133 2014/15a

Lecture 13

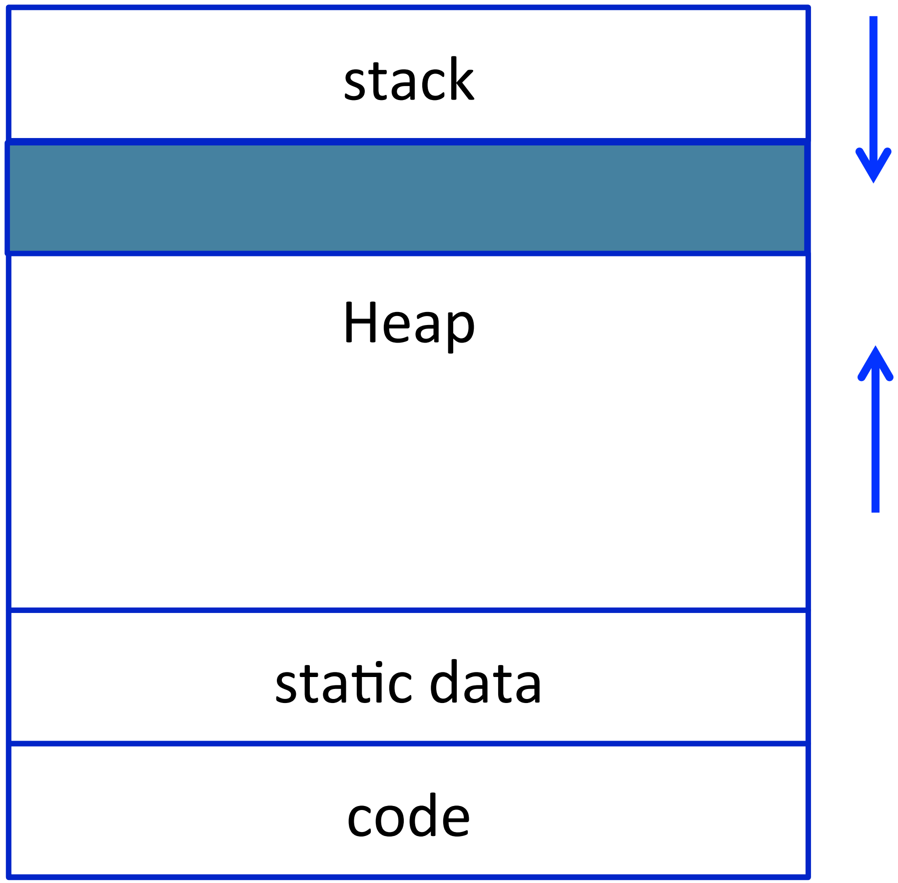
Compiling Object-Oriented Programs

Noam Rinetzky

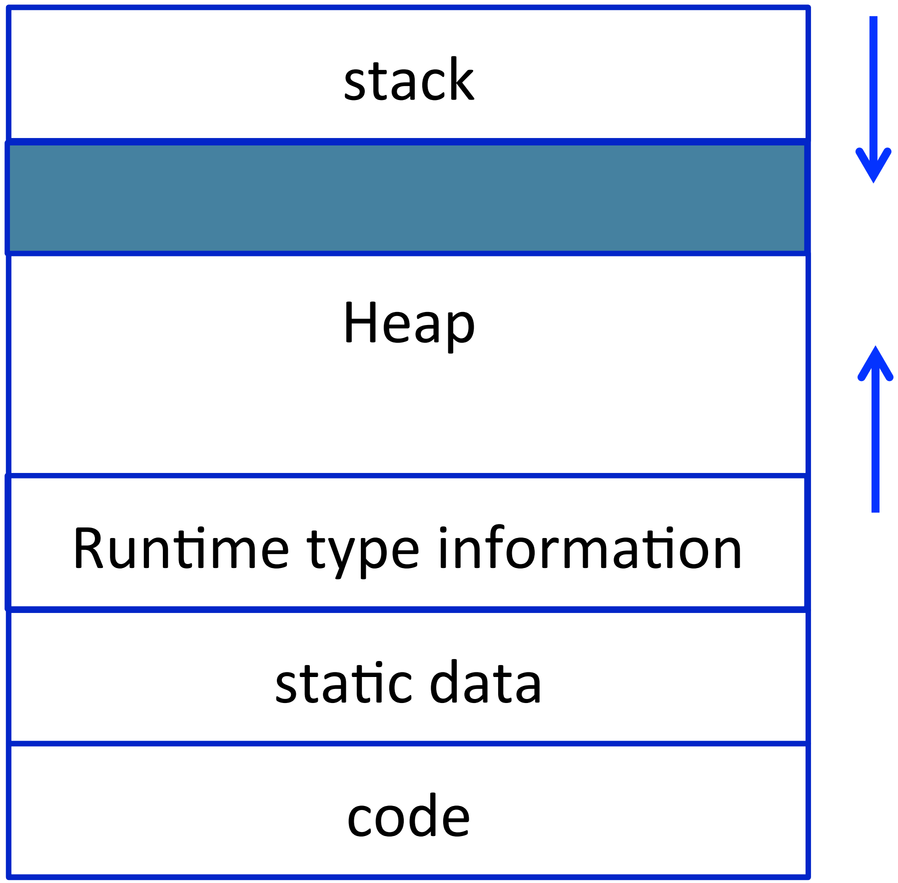
Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Memory management
- Runtime type information
  - Method invocation
  - Type conversions

Memory Layout



Memory Layout



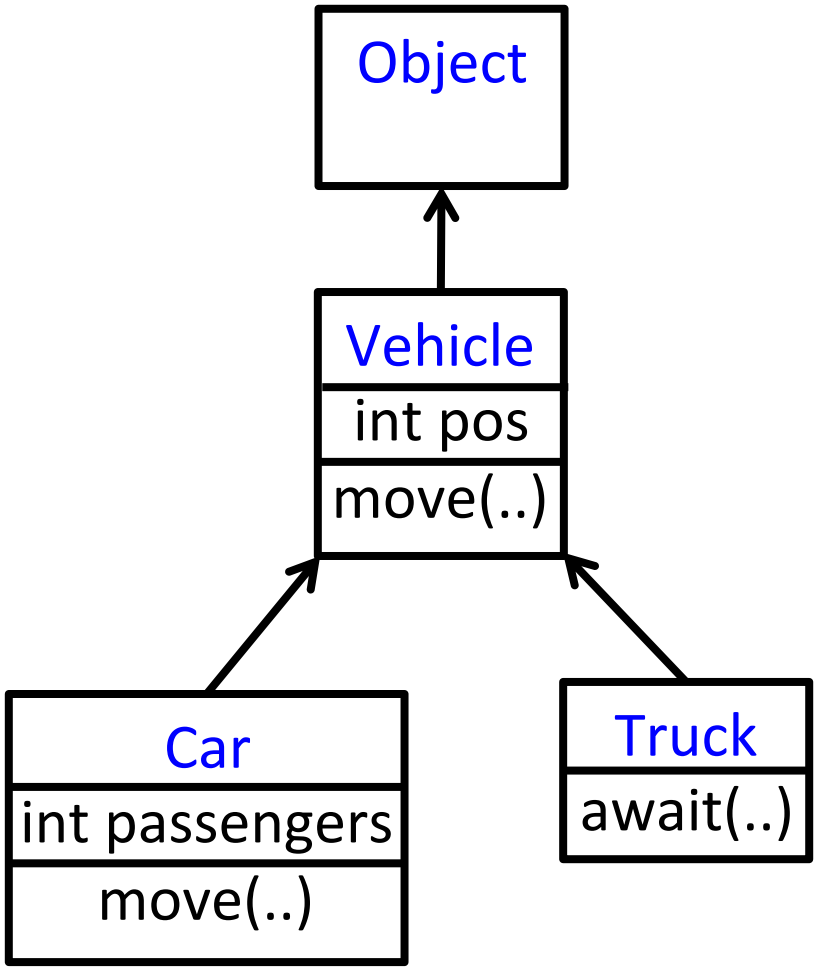
Object Oriented Programs

- C++, Java, C#, Python, …

- Main abstraction: **Objects** (usually of type called class)
  - Code
  - Data

- Naturally supports **A**bstract **D**ata **T**ype implementations
- Information hiding
- Evolution & reusability

A Simple Example



A Simple Example

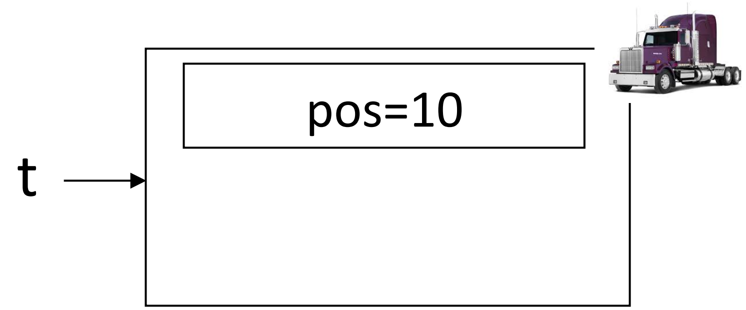
```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

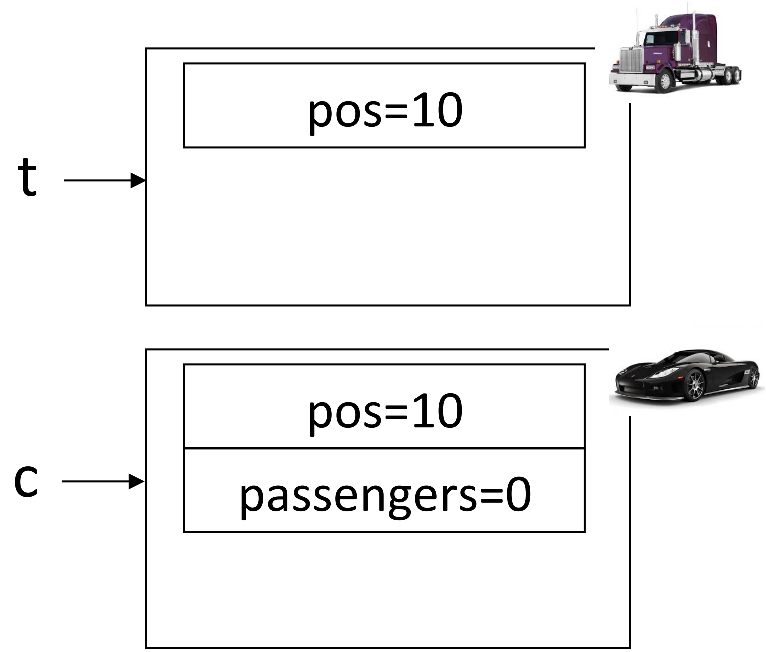
```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

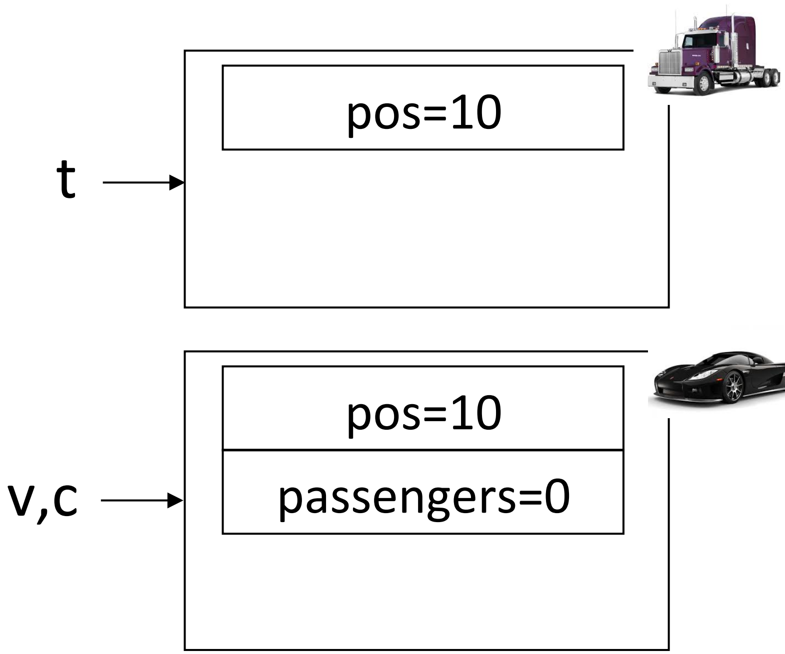
```
class main extends Object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

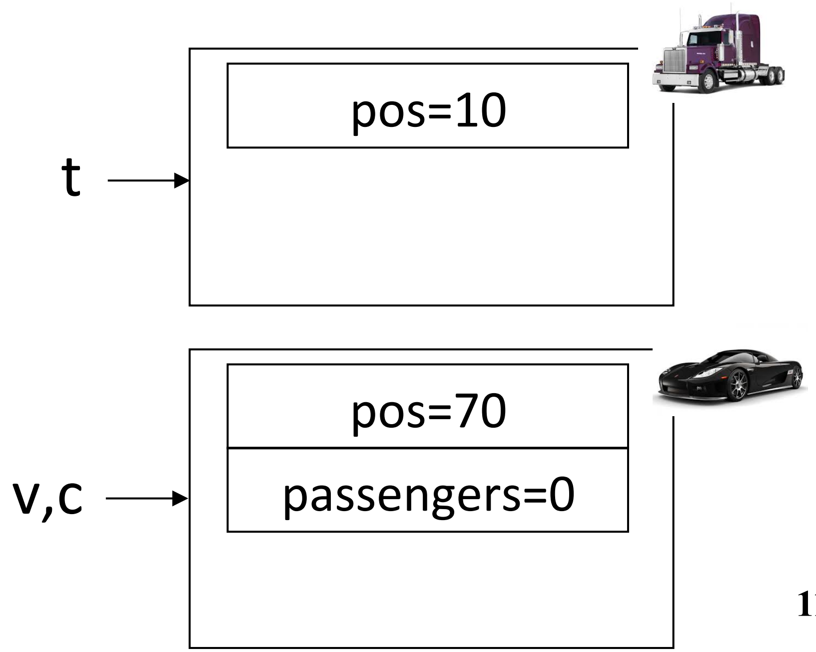
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
     pos = pos + x;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

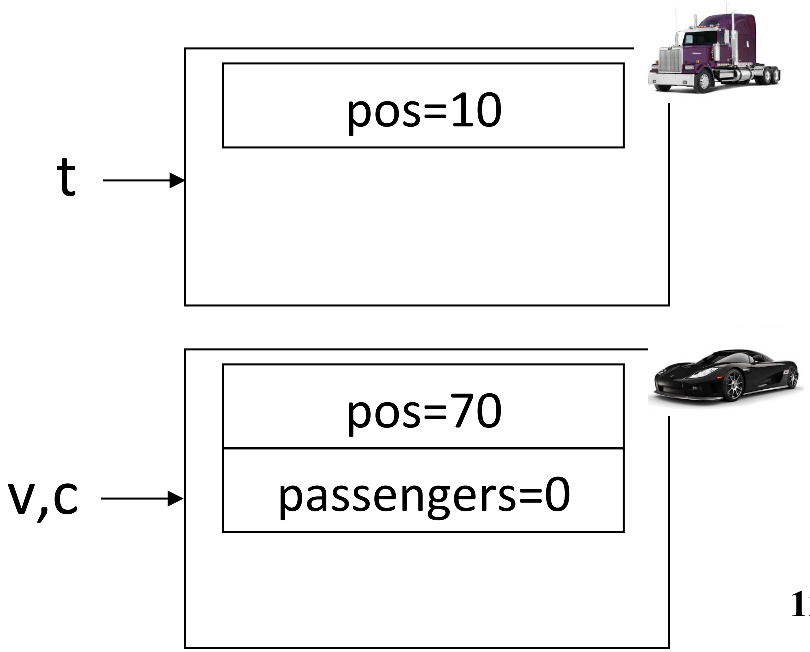
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

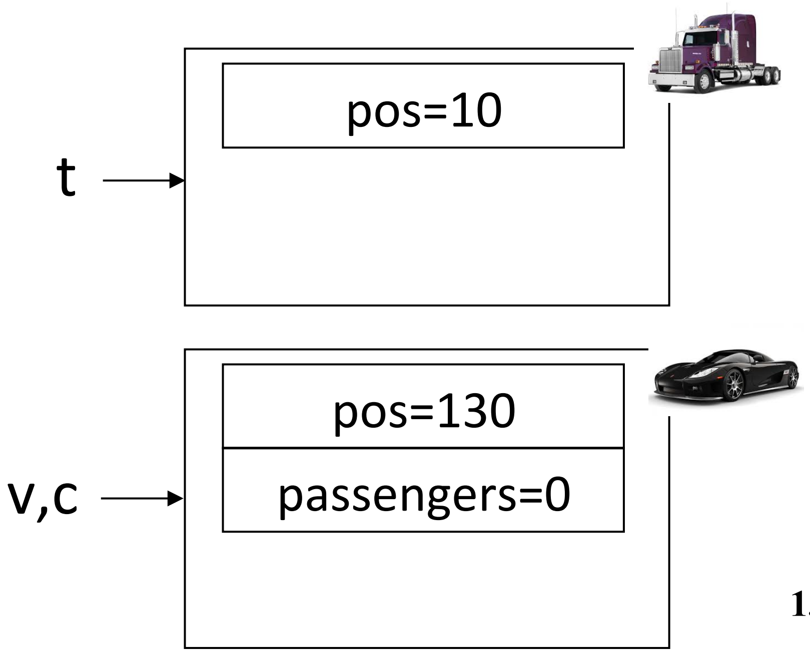
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

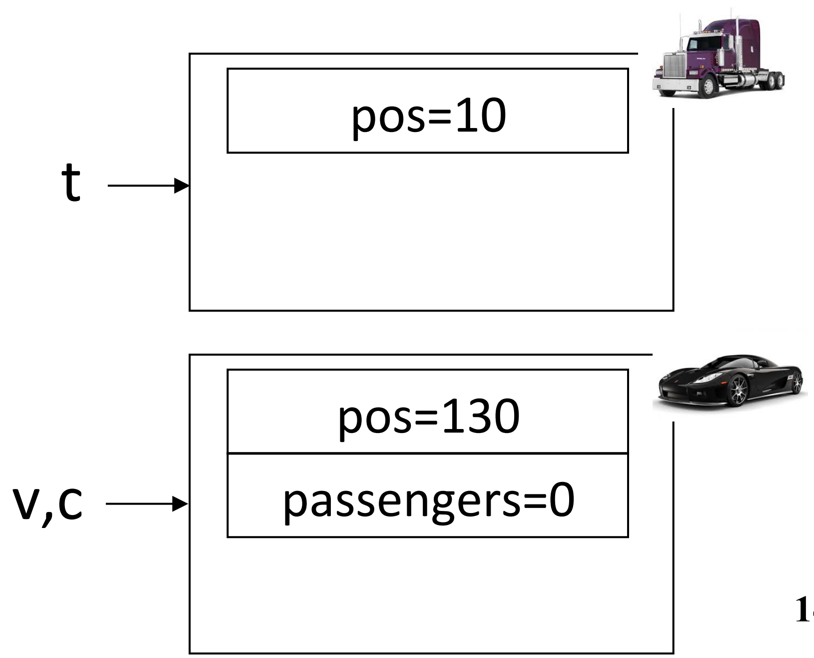
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```



A Simple Example

```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}

class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}

class Car extends Vehicle {
  int passengers = 0;
  void await(vehicle v){
    if (v.pos < pos)
      v.move(pos - v.pos);
    else
      this.move(10);
  }
}
```

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



Translation into C (Vehicle)

```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
struct Vehicle {
  int pos;
}
```

Translation into C

Translation into C (Vehicle)

```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;
```

Translation into C (Vehicle)

```
class Vehicle extends Object {
  int pos = 10;
  void move(int x) {
    pos = pos + x ;
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;

void NewVe(Ve *this){
  this→pos = 10;
}

void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Translation into C (Truck)

```
class Truck extends Vehicle {
  void move(int x){
    if (x < 55)
      pos = pos + x;
  }
}
```

```
typedef struct Truck {
  int pos;
} Tr;

void NewTr(Tr *this){
  this→pos = 10;
}

void moveTr(Ve *this, int x){
  if (x<55)
    this→pos = this→pos + x;
}
```

Naïve Translation into C (Car)

```
class Car extends Vehicle {
 int passengers = 0;
 void await(vehicle v){
  if (v.pos < pos)
    v.move(pos - v.pos);
  else
    this.move(10);
 }
}
```

```
typedef struct Car{
  int pos;
  int passengers;
} Ca;

void NewCa (Ca *this){
  this→pos = 10;
  this→passengers = 0;
}

void awaitCa(Ca *this, Ve *v){
  if (v→pos < this→pos)
    moveVe(this→pos - v→pos)
  else
    MoveCa(this, 10)
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
typedef struct Vehicle {
  int pos;
} Ve;
```

```
typedef struct Car{
  int pos;
  int passengers;
} Ca;
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
void moveCa() ?
```

```
void moveVe(Ve *this, int x){
  this→pos = this→pos + x;
}
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
Vehicle x = t;
x.move(20);
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
Ve *x = t;
moveTr((Tr*)x, 20);
```

Naïve Translation into C (Main)

```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    v.move(60);
    t.move(70);
    c.await(t);
  }
}
```

```
Vehicle x = t;
x.move(20);
```

```
void mainMa(){
  Tr *t = malloc(sizeof(Tr));
  Ca *c = malloc(sizeof(Ca));
  Ve *v = (Ve*) c;
  moveVe(v, 60);
  moveVe(t, 70);
  awaitCa(c,(Ve*) t);
}
```

```
Ve *x = t;
moveTr((Tr*)x, 20);
```

```
void moveVe(Ve *this, int x){…}
```

```
void moveTr(Ve *this, int x){…}
```

Translation into C

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {
   field a1;
   field a2;
   method m1() {…}
   method m2(int i) {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA  *this, int i) {
   // Body of m2 with any object
   // field f as this→f
   …
}
```

Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

```
class A {
   field a1;
   field a2;
   method m1() {…}
   method m2(int i) {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A
m2A

```
void m2A(classA  *this, int i) {
   // Body of m2 with any
   // object-field f as this→f
   …
}
```

```
a.m2(5)
```

```
m2A(a,5)
```

Features of OO languages

- **Inheritance**
  - Subclass gets (inherits) properties of superclass
- **Method overriding**
  - Multiple methods with the same name with different signatures
- **Abstract** (aka **virtual**) **methods**
- **Polymorphism**
  - Multiple methods with the same name and different signatures but with different implementations
- **Dynamic dispatch**
  - Lookup methods by (runtime) type of target object

Compiling OO languages

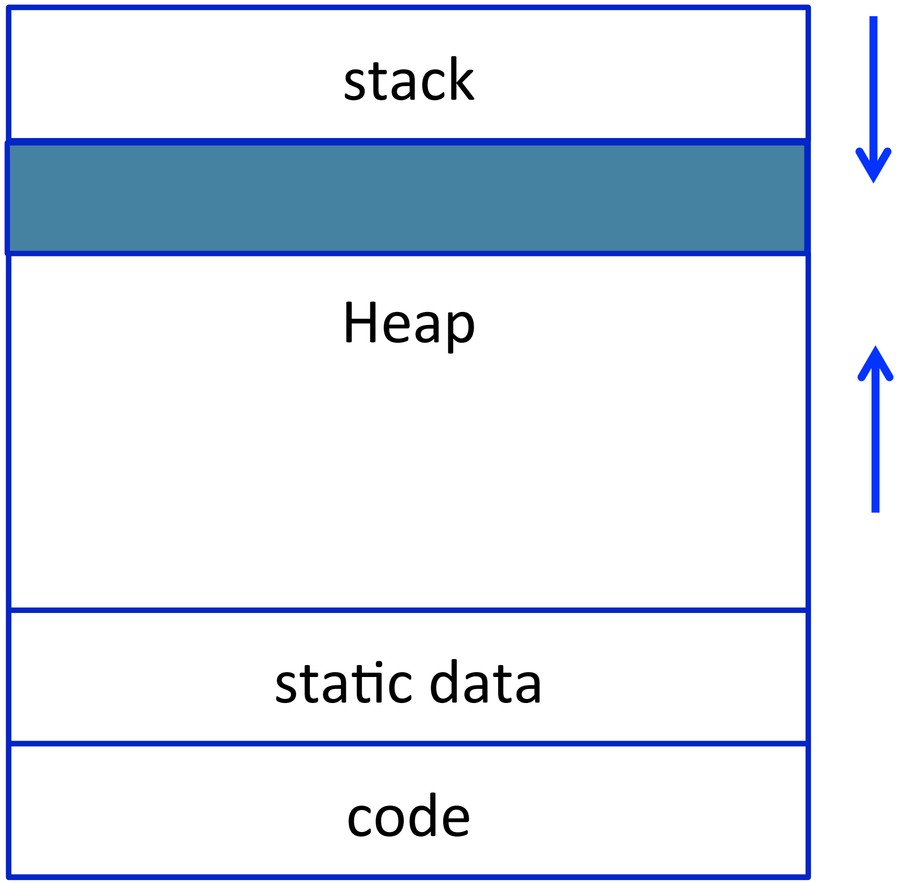
- “Translation into C”
- Powerful runtime environment

- Adding “gluing” code

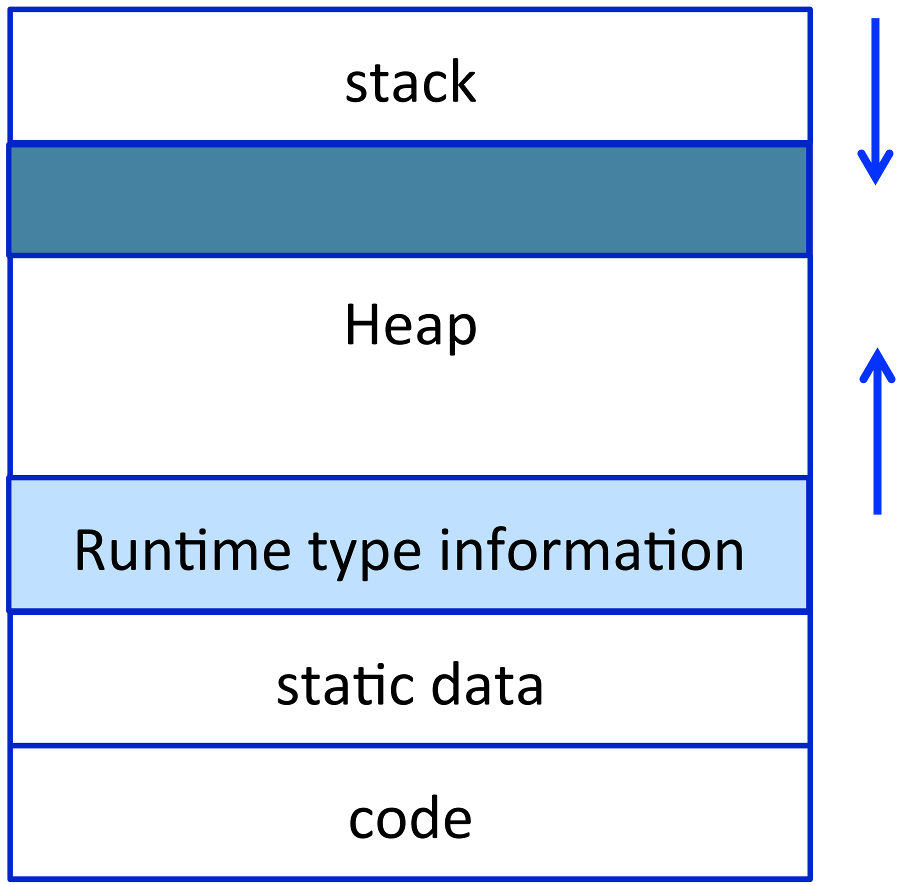
Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Memory management
- **Runtime type information**
  - **Method invocation**
  - **Type conversions**

Memory Layout



Memory Layout



Handling Single Inheritance

- Simple type extension

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m3() {…}
}
```

Adding fields

Fields aka Data members, instance variables

- Adds more information to the inherited class
  - “Prefixing” fields ensures consistency

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
     field a1;
     field a2;
     method m1() {…}
     method m2() {…}
}
```

m2 is declared and defined

```
class B extends A {
     field b1;
     method m2() {
          … b1 …
     }
     method m3() {…}
}
```

m2 is redefined

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

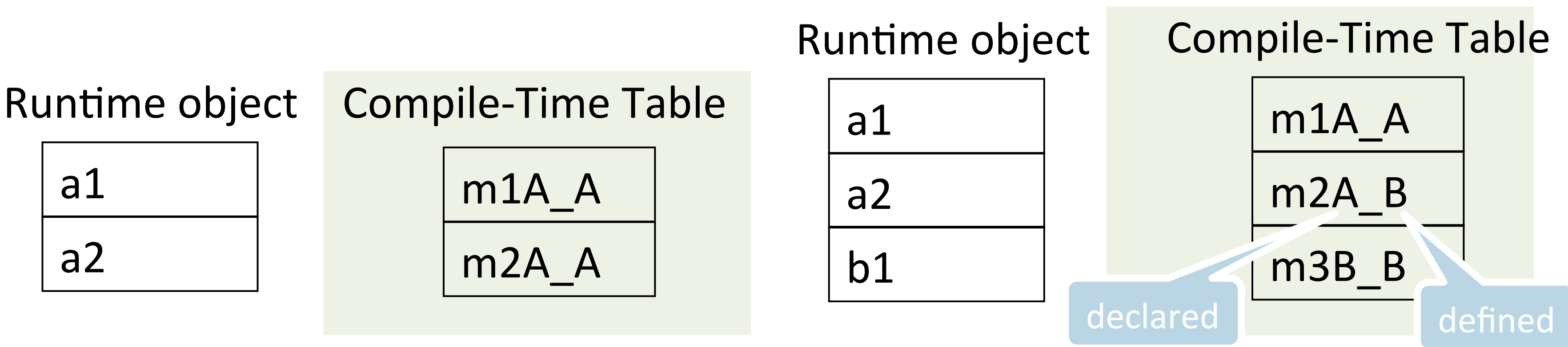
m1A_A
m2A_B
m3B_B

Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```



Method Overriding

```
a.m2(5)  // class(a) = A
```

```
m2A_A(a, 5)
```

```
class A {
     field a1;
     field a2;
     method m1() {…}
     method m2() {…}
}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

```
b.m2(5)  // class(b) = B
```

```
m2A_B(b, 5)
```

```
class B extends A {
     field b1;
     method m2() {
          … b1 …
     }
     method m3() {…}
}
```

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
     … b1 …
    }
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Method Overriding

a.m2(5) // class(a) = A

m2A_A(a, 5)

b.m2(5) // class(b) = B

m2A_B(b, 5)

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Abstract methods & classes

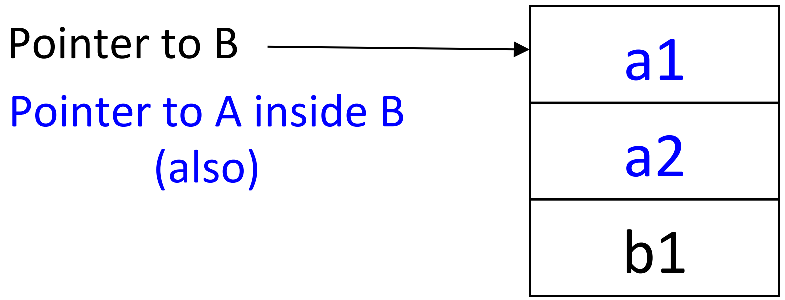
- Abstract methods
  - Declared separately, defined in child classes
  - E.g., C++ pure virtual methods, abstract methods in Java
- Abstract classes = class may have abstract methods
  - E.G., Java/C++ abstract classes
  - Abstract classes cannot be instantiated
- Abstract aka "virtual"
- Inheriting abstract class handled like regular inheritance
  - Compiler checks abstract classes are not allocated

Handling Polymorphism

- When a class B extends a class A
  - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Prefixing fields guarantees validity

```
class B *b = …;

class A *a = b ;
```

```
classA *a = convert_ptr_to_B_to_ptr_A(b) ;
```



Dynamic Binding

- An object (“pointer”) o declared to be of class A can actually be (“refer”) to a class B

- What does ‘o.m()’ mean?
  - Static binding
  - Dynamic binding
- Depends on the programming language rules

- How to implement dynamic binding?
  - The invoked function is not known at compile time
  - Need to operate on data of the B and A in consistent way

Conceptual Impl. of Dynamic Binding

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m2() {
     … a3 …
    }
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B:m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

Conceptual Impl. of Dynamic Binding

?

```
switch(dynamic_type(p)) {
  case Dynamic_class_A: m2_A_A(p, 3);
  case Dynamic_class_B:m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

Runtime object

a1
a2

Compile-Time Table

m1A_A
m2A_A

Runtime object

a1
a2
b1

Compile-Time Table

m1A_A
m2A_B
m3B_B

More efficient implementation

- Apply pointer conversion in sublasses
  - Use dispatch table to invoke functions
  - Similar to table implementation of case

```
void m2A_B(classA *this_A) {
    Class_B *this = convert_ptr_to_A_ptr_to_A_B(this_A);
   …
}
```

More efficient implementation

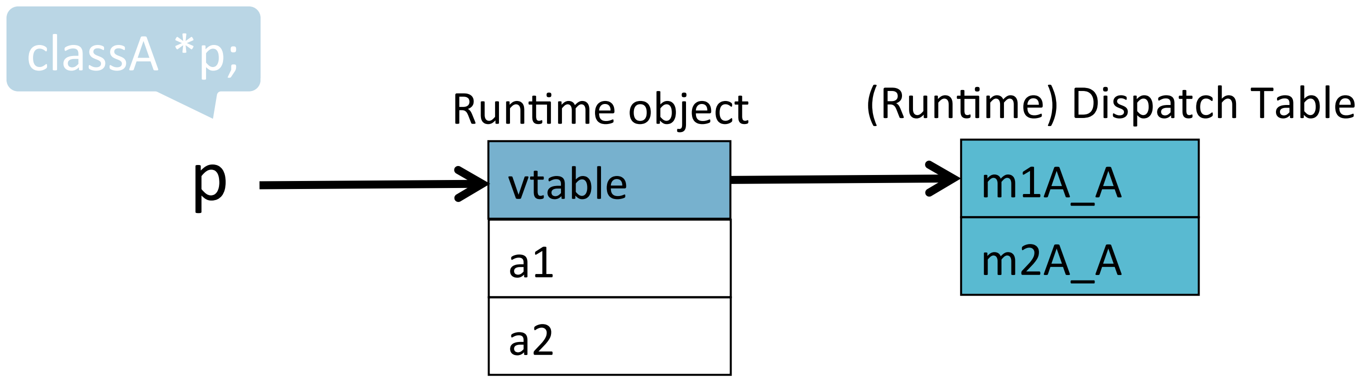
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

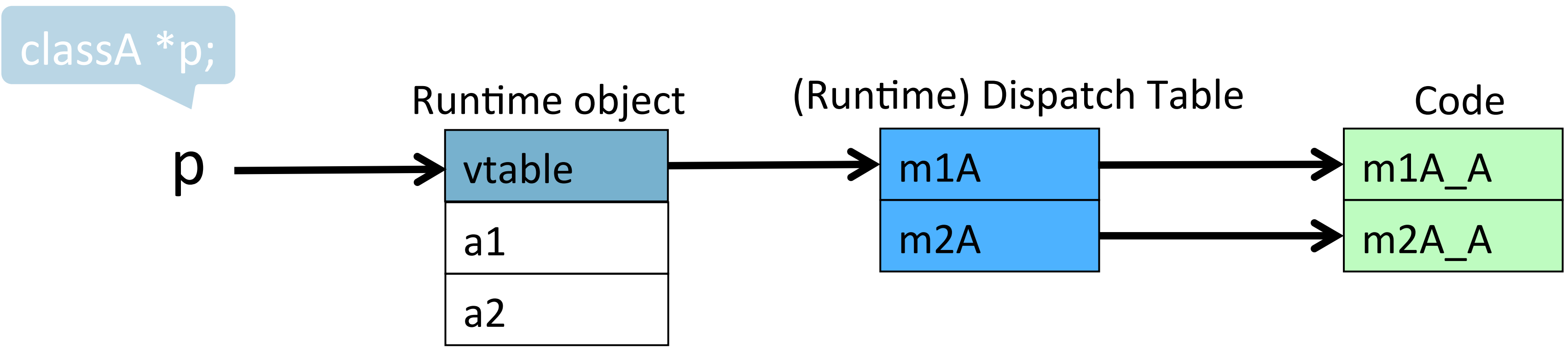
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```



More efficient implementation

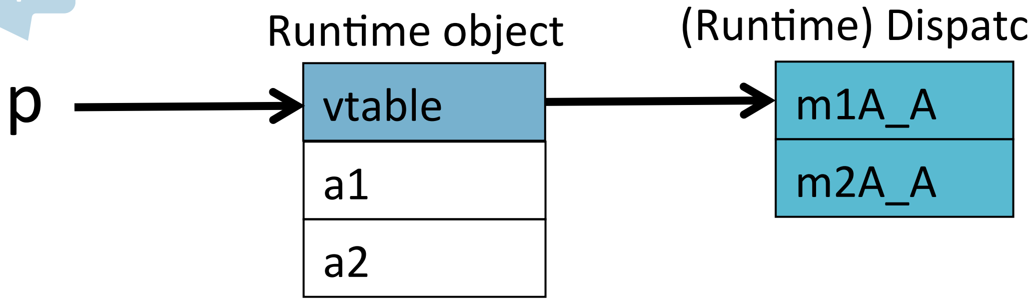
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

classA *p;

p.m2(3);

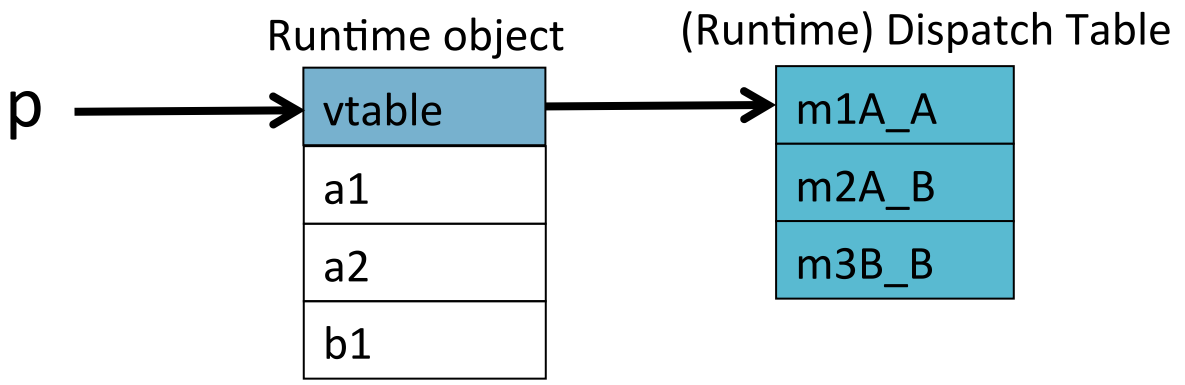
p→dispatch_table→m2A(p, 3);



More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

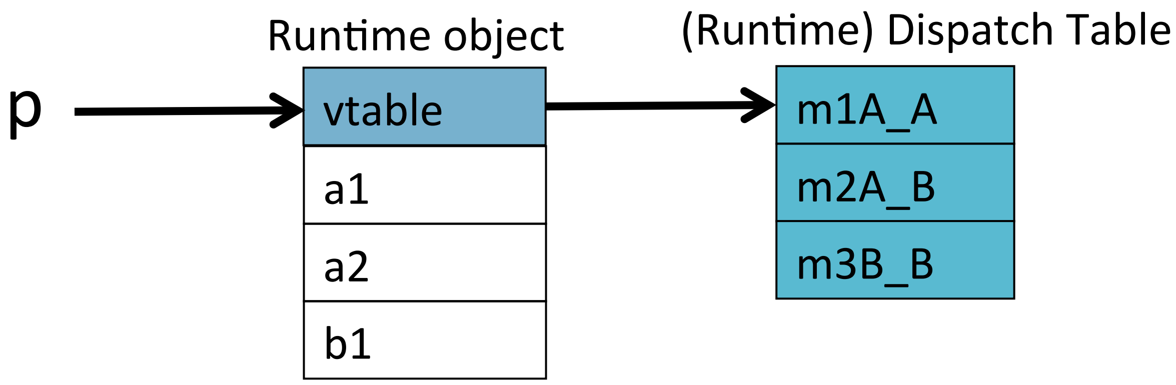


More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);



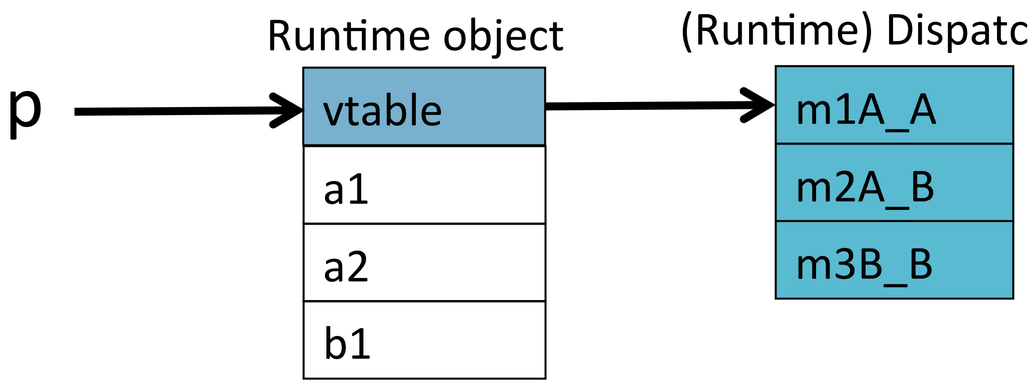
More efficient implementation

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);

p→dispatch_table→m2A(~~p~~, 3);



More efficient implementation

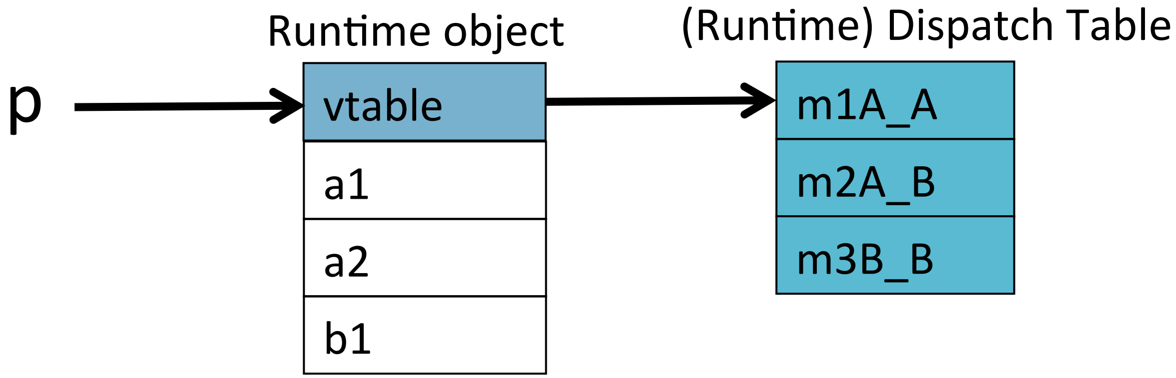
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
  Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);

convert_ptr_to_B_to_ptr_to_A(p)

p→dispatch_table→m2A(, 3);



Multiple Inheritance

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Multiple Inheritance

- Allows unifying behaviors
- But raises semantic difficulties
  - Ambiguity of classes
  - Repeated inheritance
- Hard to implement
  - Semantic analysis
  - Code generation
    - Prefixing no longer work
    - Need to generate code for downcasts
- Hard to use

A simple implementation

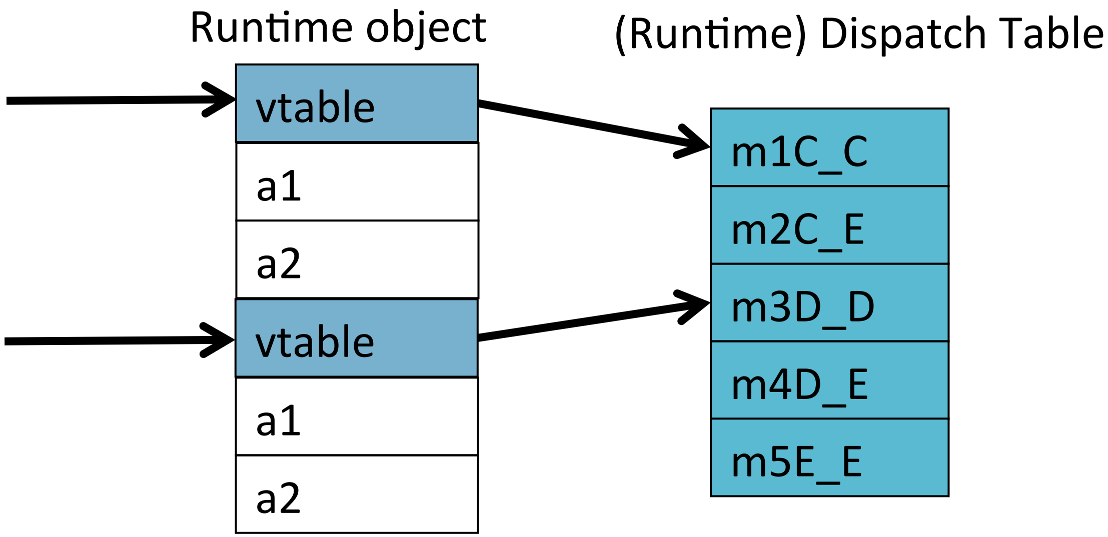
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

A simple implementation

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```



Downcasting (E→C,D)

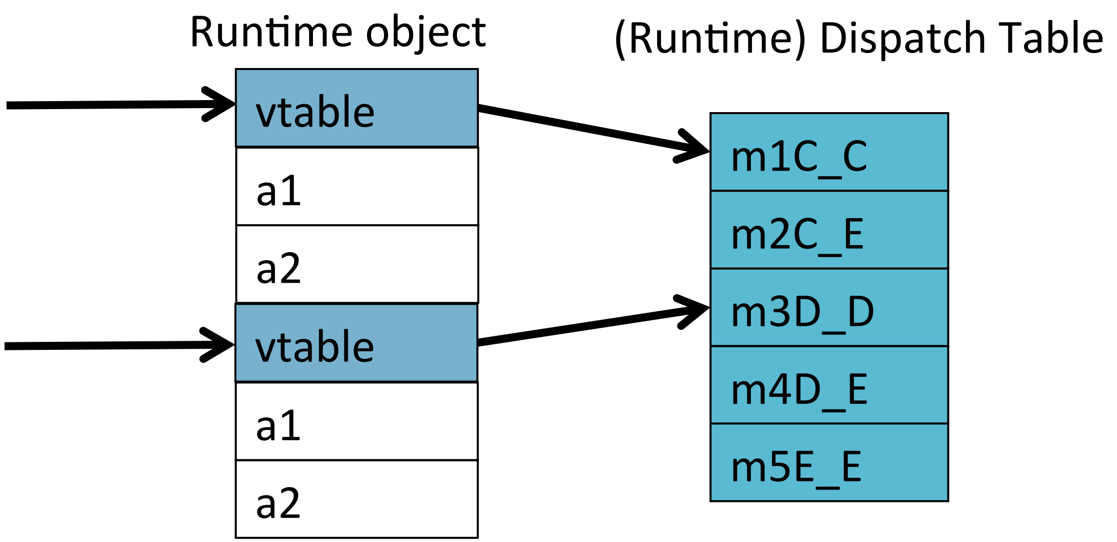
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

convert_ptr_to_E_to_ptr_to_C(e) = e;

convert_ptr_to_E_to_ptr_to_D(e) = e + sizeof(C);



Upcasting (C,D→E)

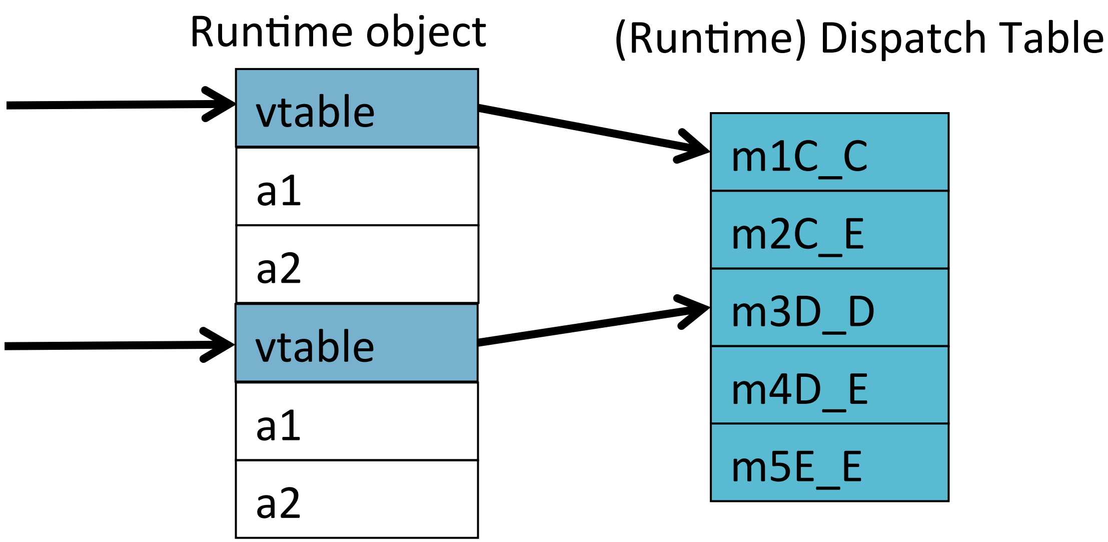
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

convert_ptr_to_C_to_ptr_to_E(c) = c;

convert_ptr_to_D_to_ptr_to_E(d) = d - sizeof(C);



Inependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Inependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Inependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Inependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;
    method m3(){…}  //alt explicit qualification
    method m2()  {…}
    method m4()  {…}
    method m5(){…}
}
```

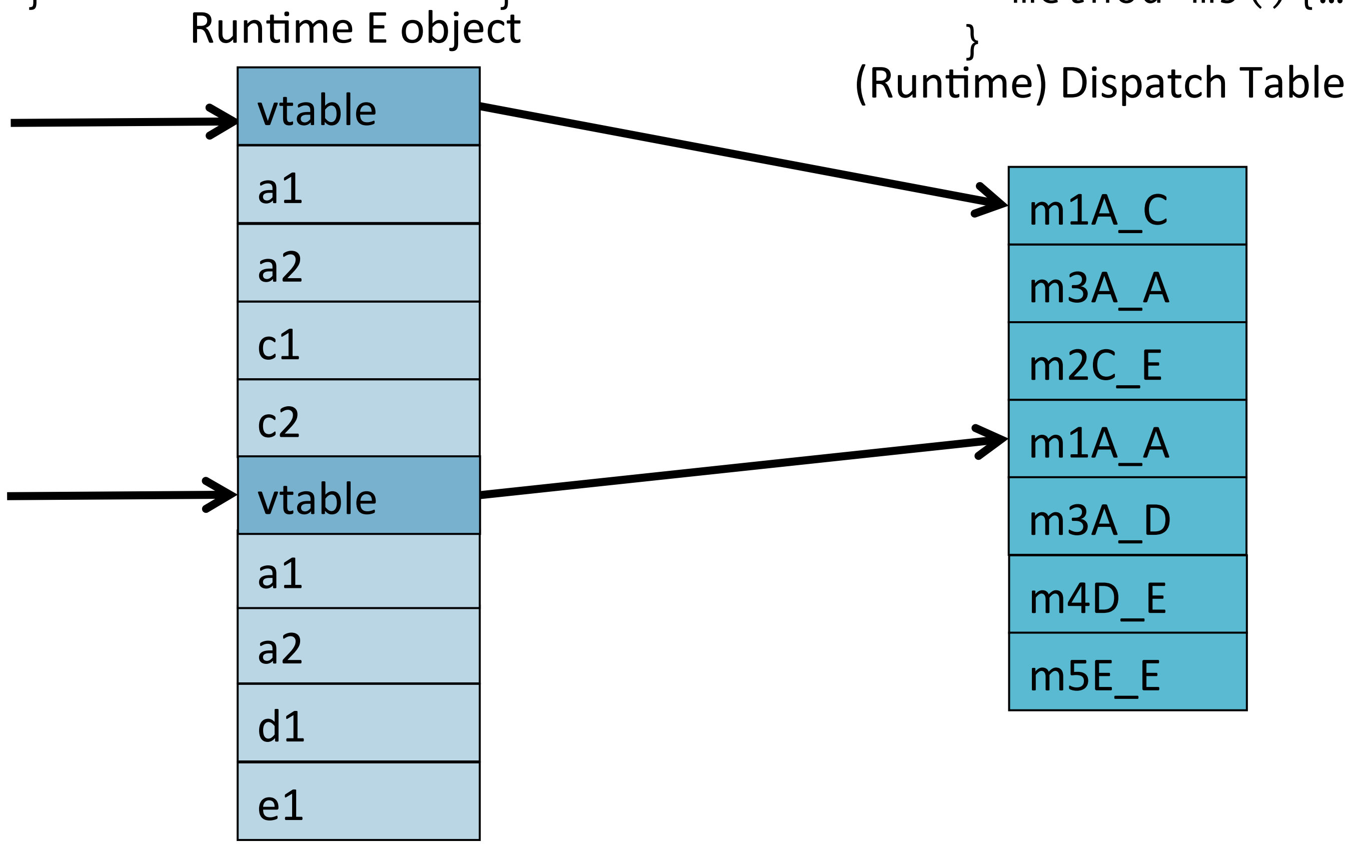
Independent Inheritance

```
class A{
  field a1;
  field a2;
  method m1(){…}
  method m3(){…}
}
```

```
class C
    extends A{
  field c1;
  field c2;
  method m1(){…}
  method m2(){…}
}
```

```
class D
    extends A{
  field d1;

  method m3(){…}
  method m4(){…}
}
```

```
class E
    extends C,D{
  field e1;

  method m2() {…}
  method m4() {…}
  method m5(){…}
}
```



Dependent multiple inheritance

- Superclasses share their own superclass

- The simple solution does not work
- The positions of nested fields do not agree

Dependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Dependent multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Dependent Multiple Inheritance

```
class A{
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}

class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}

class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}

class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

Dependent Inheritance

- Superclasses share their own superclass

- The simple solution does not work
- The positions of nested fields do not agree

Implementation

- Use an index table to access fields
- Access offsets indirectly

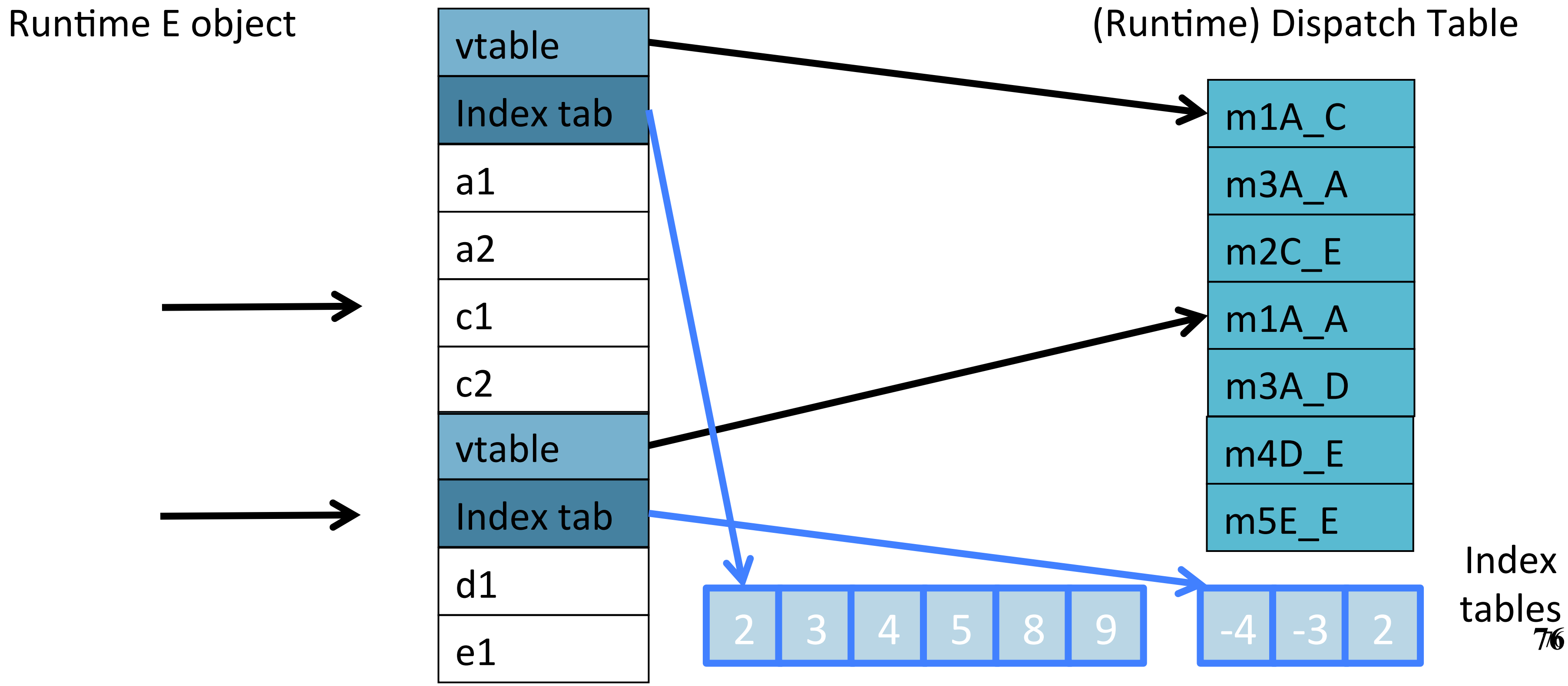
Implementation

```
class A{
  field a1;
  field a2;
  method m1(){…}
  method m3(){…}
}
```

```
class C
    extends A{
  field c1;
  field c2;
  method m1(){…}
  method m2(){…}
}
```

```
class D
    extends A{
  field d1;

  method m3(){…}
  method m4(){…}
}
```

```
class E
    extends C,D{
  field e1;

  method m2()  {…}
  method m4()  {…}
  method m5(){…}
}
```



Class Descriptors

- Runtime information associated with instances
- Dispatch tables
  - Invoked methods
- Index tables
- Shared between instances of the same class

- Can have more (reflection)

Interface Types

- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields
- A class can implement multiple interfaces
  Simpler to implement/understand/use
- Implementation: record with 2 pointers:
  - A separate dispatch table per interface
  - A pointer to the object

Dynamic Class Loading

- Supported by some OO languages (Java)
- At compile time
  - the actual class of a given object at a given program point may not be known
- Some addresses have to be resolved at runtime
- Compiling c.f() when f is dynamically loaded:
  - Fetch the class descriptor d at offset 0 from c
  - Fetch the address of the method-instance f from (constant) f offset at d into p
  - Jump to the routine at address p (saving return address)

Other OO Features

- Information hiding
  - private/public/protected fields
  - Semantic analysis (context handling)

- Testing class membership

Optimizing OO languages

- Hide additional costs
  - Replace dynamic by static binding when possible
  - Eliminate runtime checks
  - Eliminate dead fields

- Simultaneously generate code for multiple classeså
- Code space is an issue

Summary

- OO is a programming/design paradigm
- OO features complicates compilation
  - Semantic analysis
  - Code generation
  - Runtime
  - Memory management
- Understanding compilation of OO can be useful for programmers

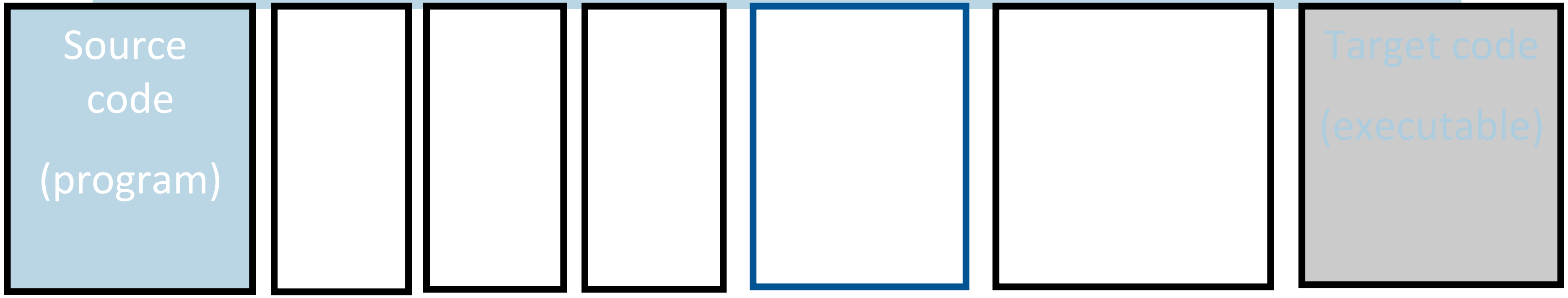
Compilation

0368-3133 2014/15a
Lecture 13

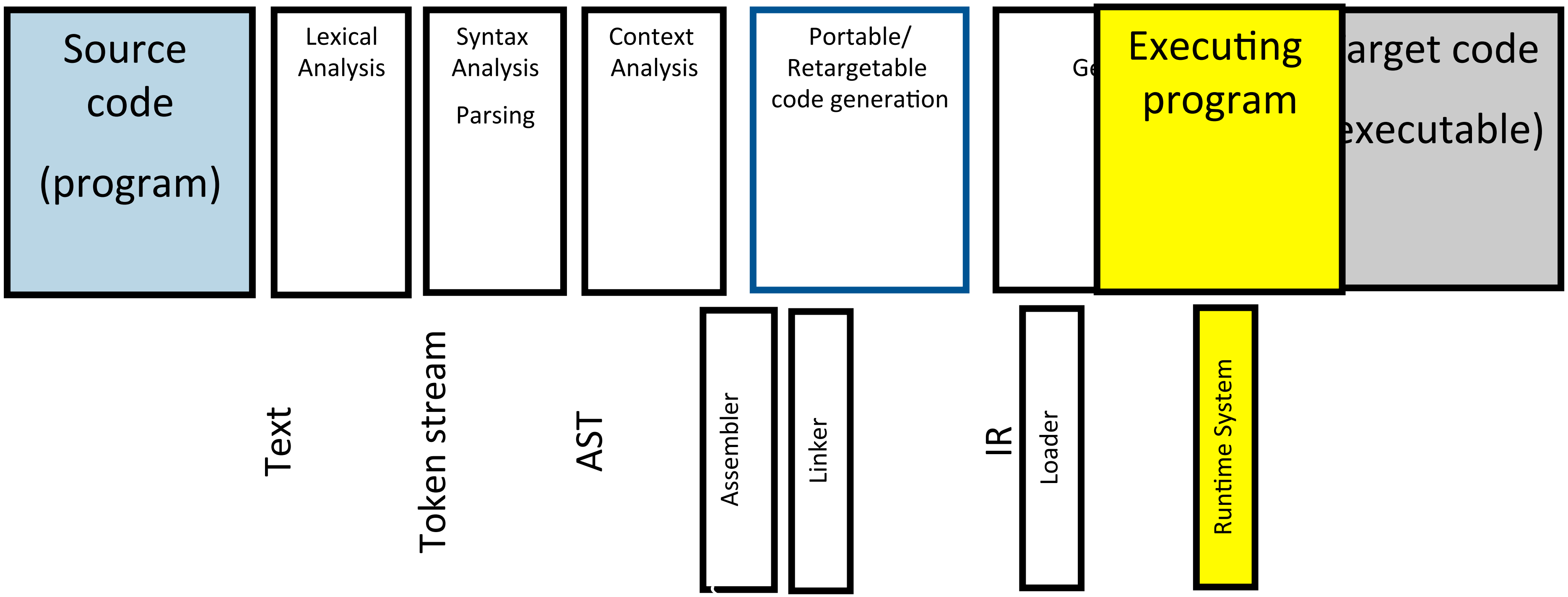
Memory Management

Noam Rinetzky

Stages of compilation



Compilation → Execution



Runtime Environment

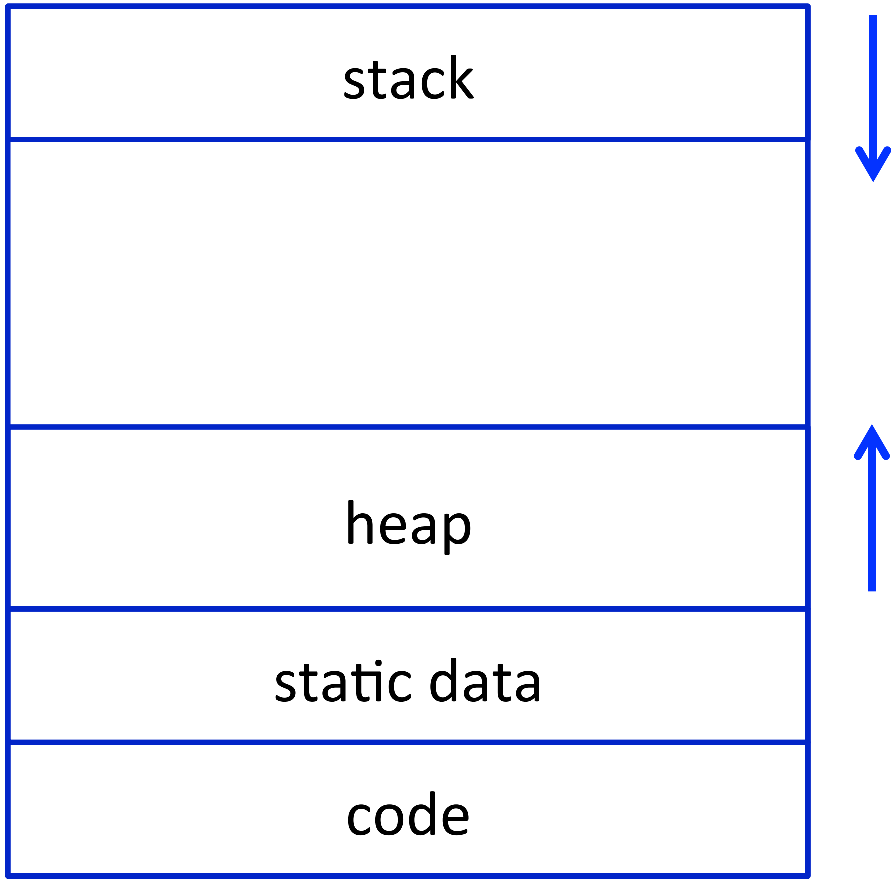
- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Dynamic optimization
  - Debugging
  - …

Where do we allocate data?

- Activation records
  - Lifetime of allocated data limited by procedure lifetime
  - Stack frame deallocated (popped) when procedure return

- **Dynamic memory allocation on the heap**

Memory Layout



Alignment

- Typically, can only access memory at aligned addresses
  - Either 4-bytes or 8-bytes
- What happens if you allocate data of size 5 bytes?
  - **Padding** – the space until the next aligned addresses is kept empty
- (side note: x86, is more complicated, as usual, and also allows unaligned accesses, but not recommended)

Allocating memory

- In C - malloc
- `void *malloc(size_t size)`

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its type
- How does malloc guarantee alignment?
  - After all, you don't know what type it is allocating for
  - It has to align for the largest primitive type
  - In practice optimized for 8 byte alignment (glibc-2.17)

Memory Management

- Manual memory management
- Automatic memory management

Manual memory management

- malloc
- free

malloc

- where is malloc implemented?
- how does it work?

free

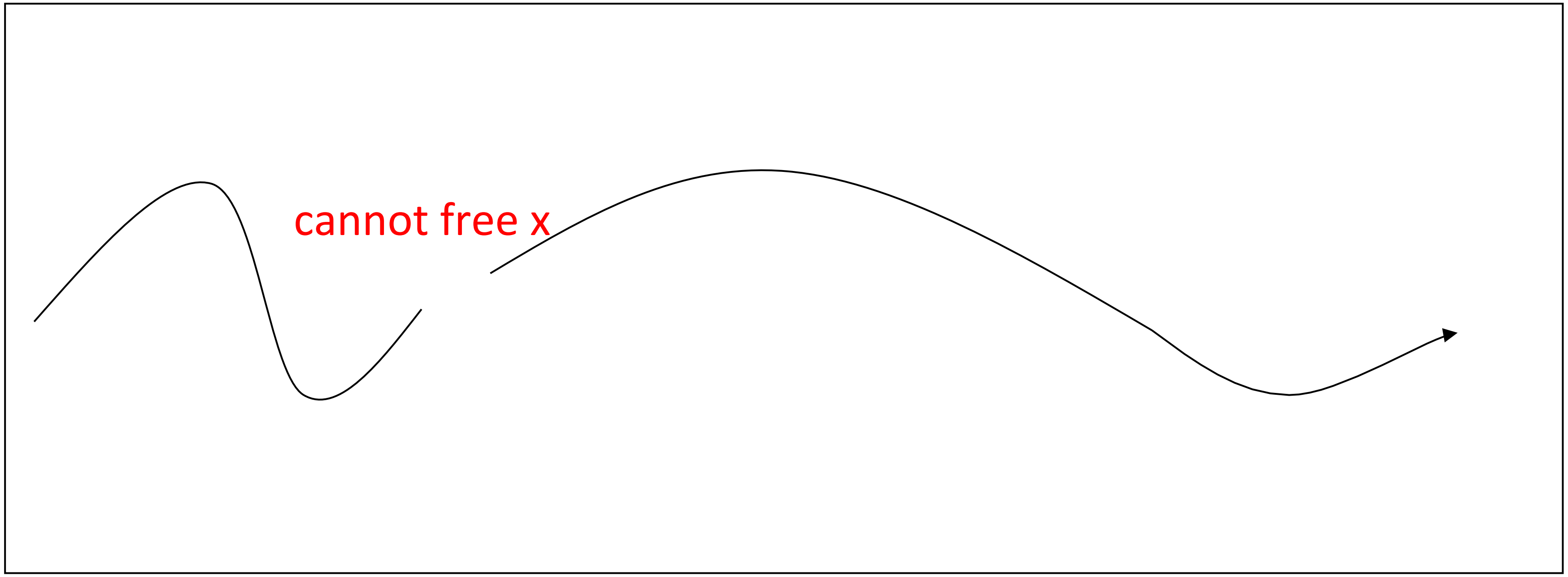
- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

When can we free an object?

```
// free (a); ?
```

Cannot free an object if it has a reference with a future use!

When can free x be inserted after p?



after p no uses of references to the object

referenced by x free x p valid

Automatic Memory Management

- automatically free memory when it is no longer needed
- not limited to OO languages
- prevalent in OO languages such as Java
  - also in functional languages

Garbage collection

- approximate reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead

Garbage Collection – Classical Techniques

- reference counting
- mark and sweep
- copying

GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when (rc==0) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)

Managing Reference Counts

- Each object has a reference count o.RC
- A newly allocated object o gets o.RC = 1
  - why?

- write-barrier for reference updates

```
update(x,old,new) {
  old.RC--;
  new.RC++;
  if (old.RC == 0) collect(old);
}
```

- collect(old) will decrement RC for all children and recursively collect objects whose RC reached 0.

Cycles!

- cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - periodically invoke a tracing algorithm to collect cycles
  - specialized algorithms for collecting cycles

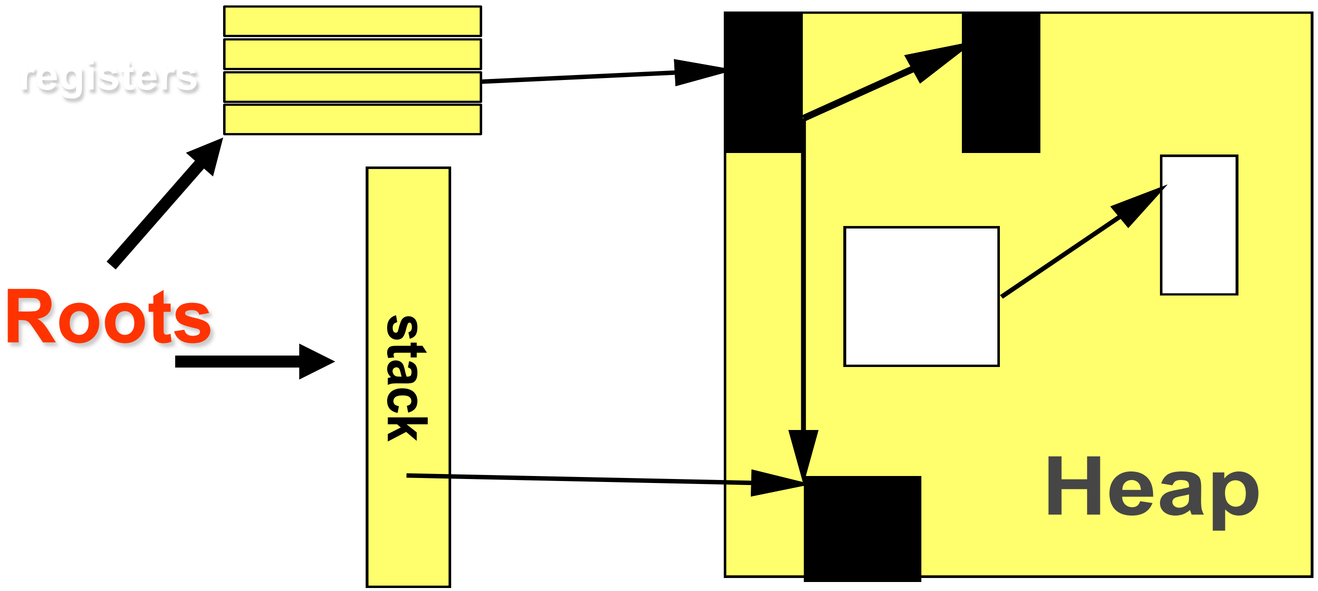
The Mark-and-Sweep Algorithm [McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object

- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.



heap

Triggering

```
New(A)=
   if free_list is empty
          mark_sweep()
           if free_list is empty
                    return ("out-of-memory")
   pointer = allocate(A)
   return (pointer)
```

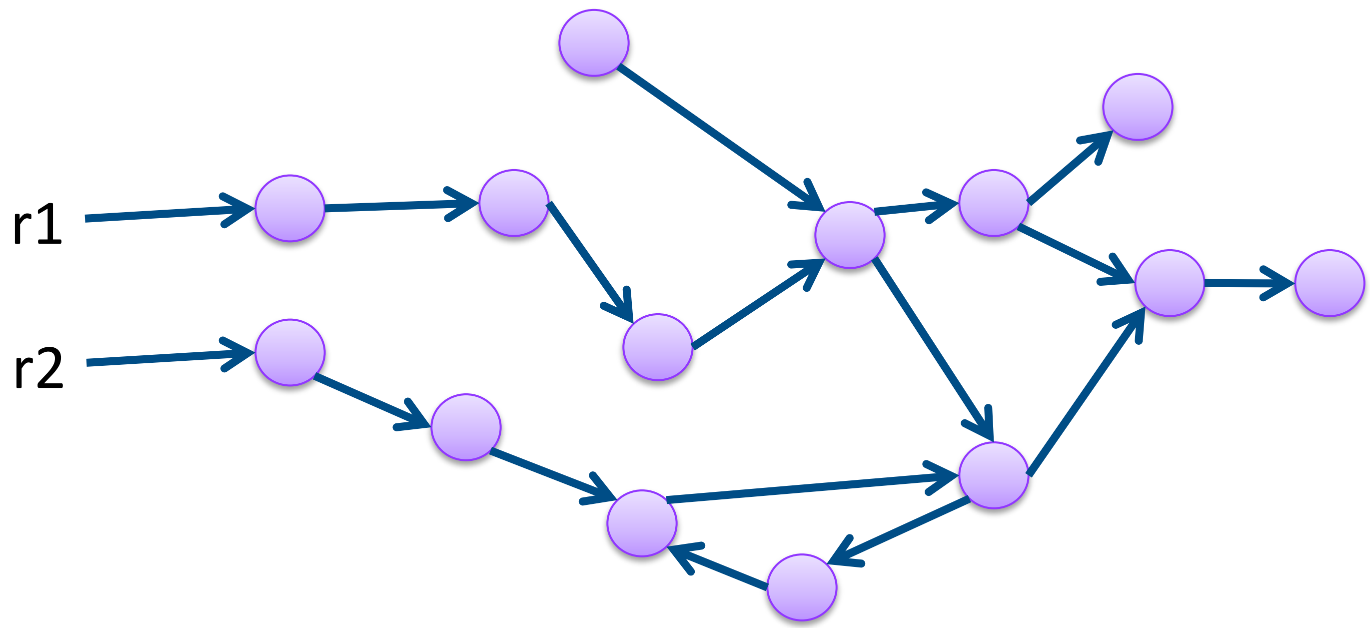
Basic Algorithm

```
mark_sweep()=
for Ptr in Roots
        mark(Ptr)
sweep()
```

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

```
Sweep()=
p = Heap_bottom
while (p < Heap_top)
        if (mark_bit(p) == unmarked) then free(p)
        else mark_bit(p) = unmarked;
        p=p+size(p)
```

Mark&Sweep Example



Mark&Sweep in Depth

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

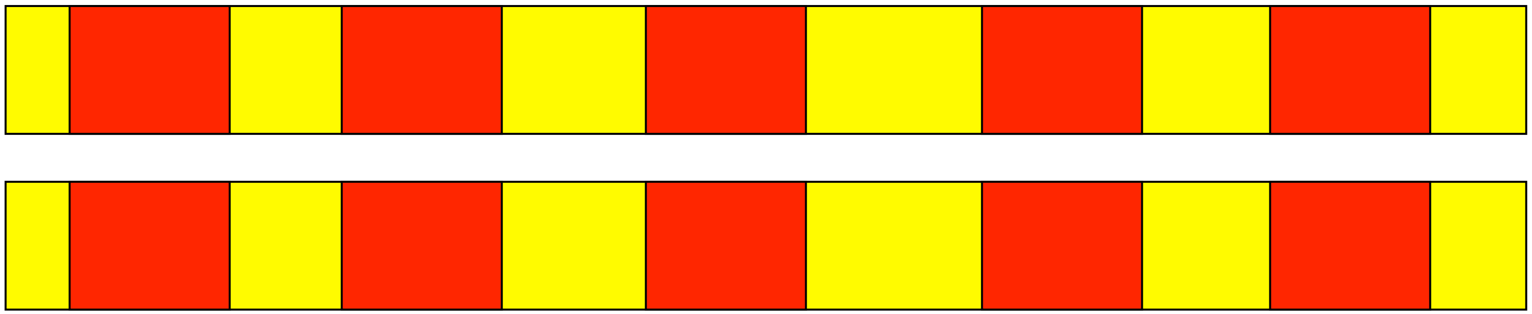
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Properties of Mark & Sweep

- Most popular method today
- Simple
- Does not move objects, and so **heap may fragment**
- Complexity
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size
- Termination: each pointer traversed once
- Engineering tricks used to improve performance

Mark-Compact

- During the run objects are allocated and reclaimed
- Gradually, the heap gets fragmented
- When space is too fragmented to allocate, a compaction algorithm is used
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations
- Compaction is very costly and we attempt to run it infrequently, or only partially



Mark Compact

- Important parameters of a compaction algorithm
  - Keep order of objects?
  - Use extra space for compactor data structures?
  - How many heap passes?
  - Can it run in parallel on a multi-processor?

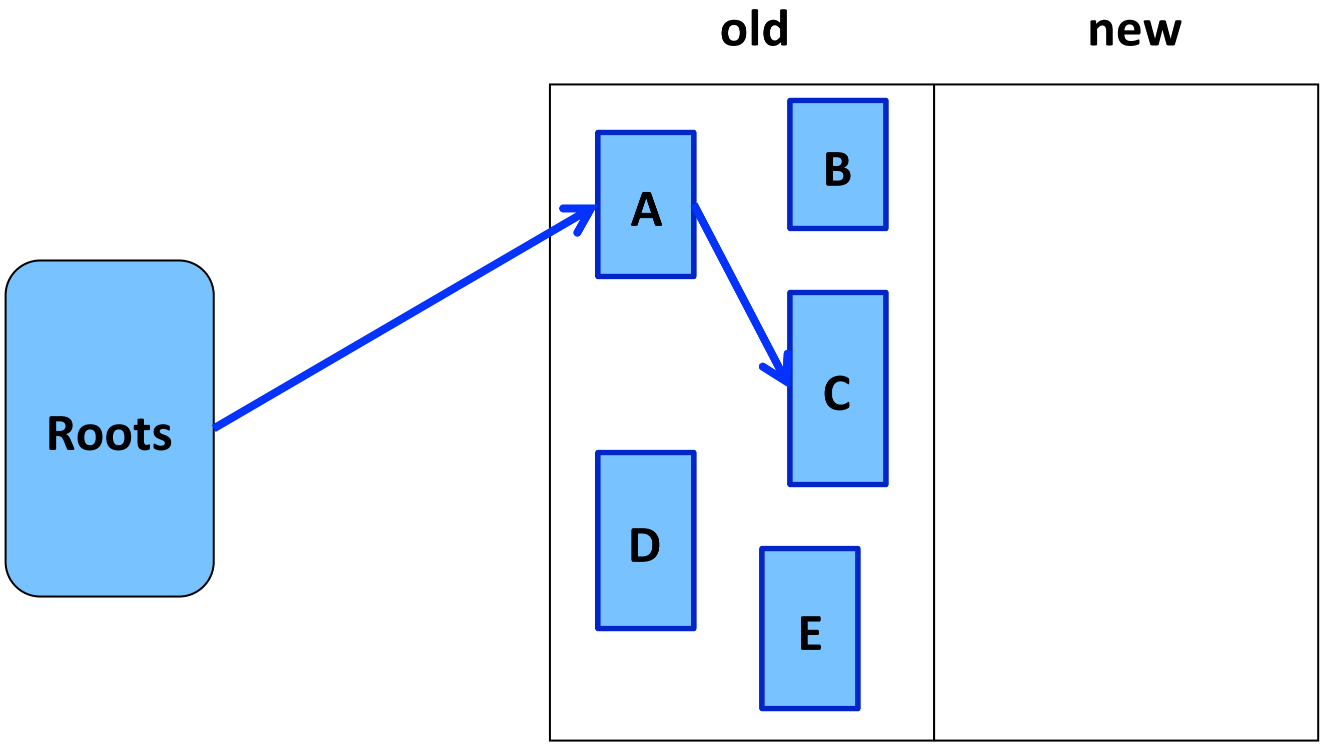
- We do not elaborate in this intro

Copying GC

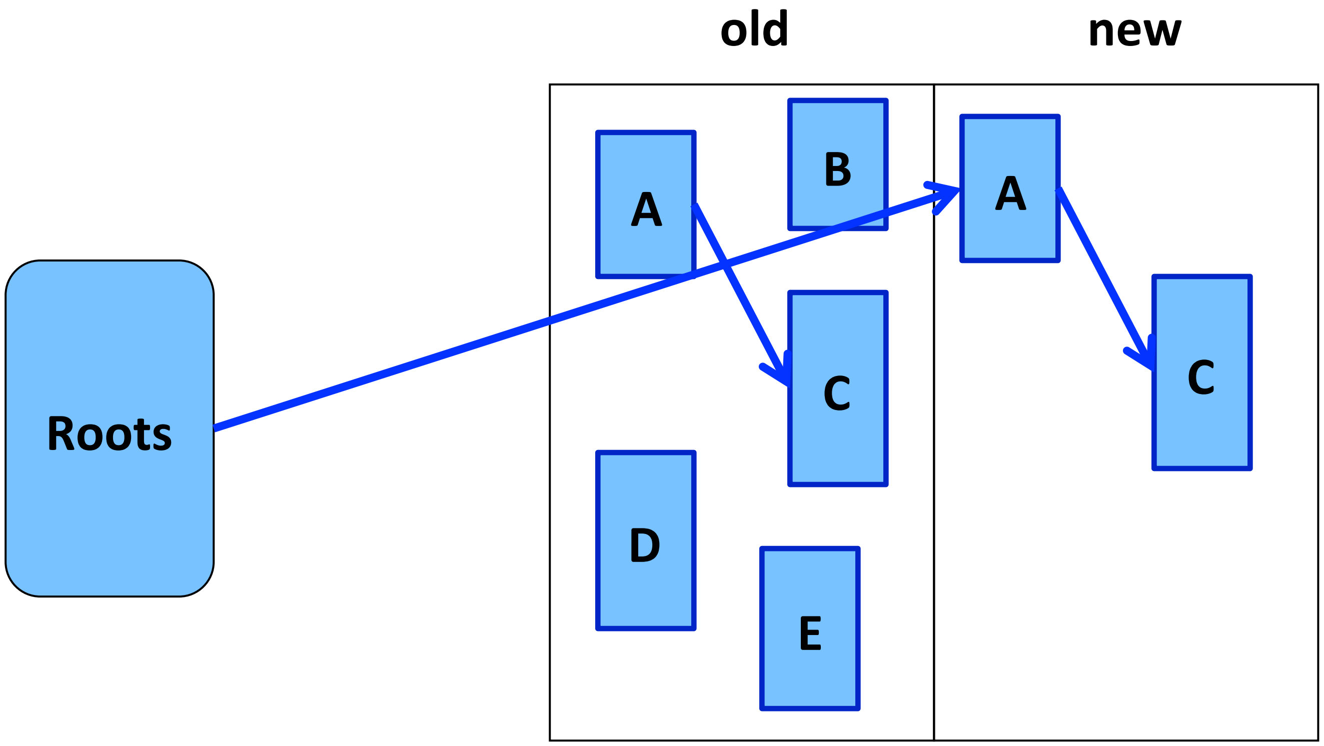
- partition the heap into two parts
  - old space
  - new space

- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - swap roles of old/new space

Example



Example



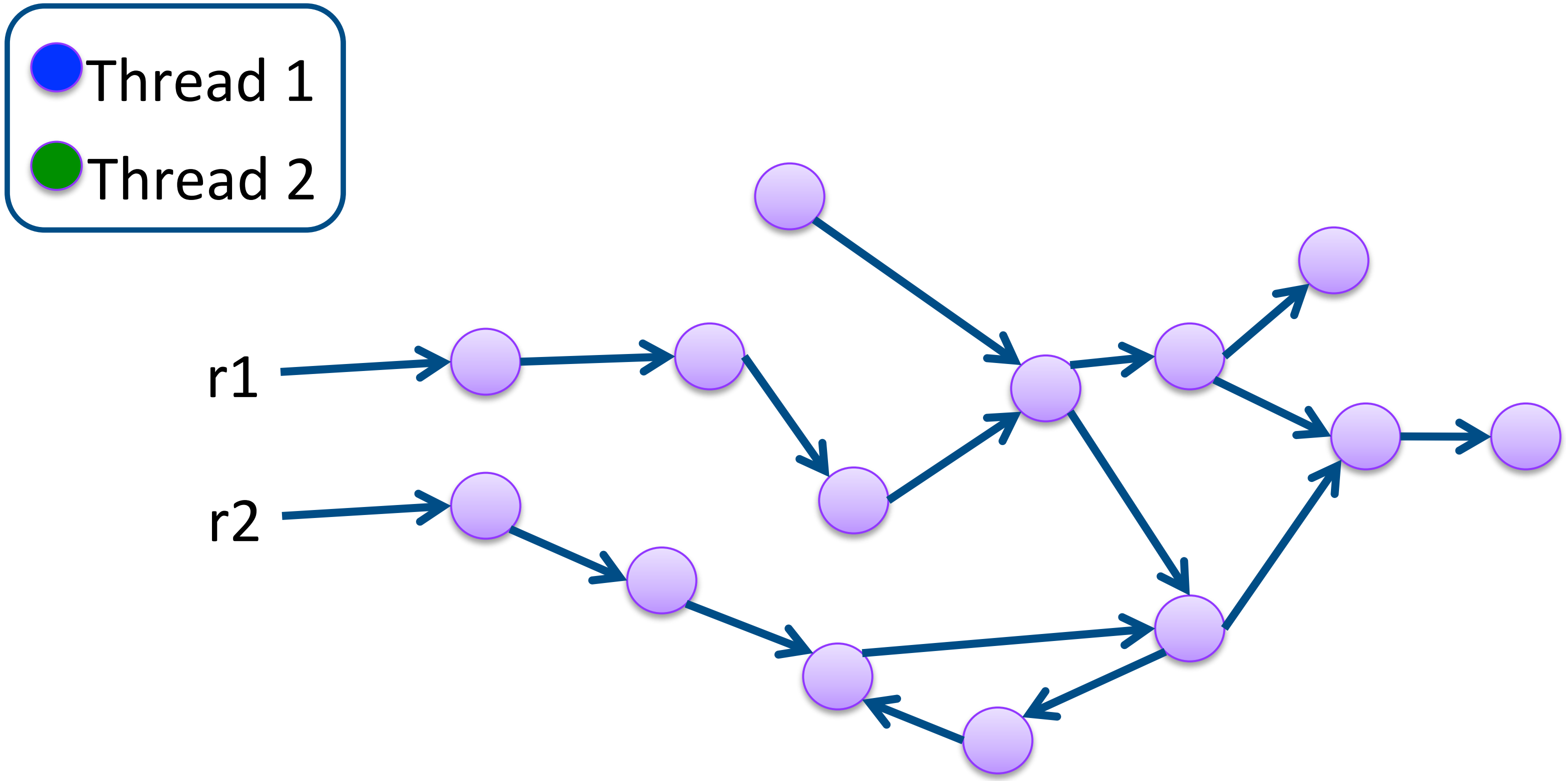
Properties of Copying Collection

- Compaction for free
- Major disadvantage: **half of the heap is not used**
- "Touch" only the live objects
  - Good when most objects are dead
  - Usually most new objects are dead
    - Some methods use a small space for young objects and collect this space using copying garbage collection

A very simplistic comparison

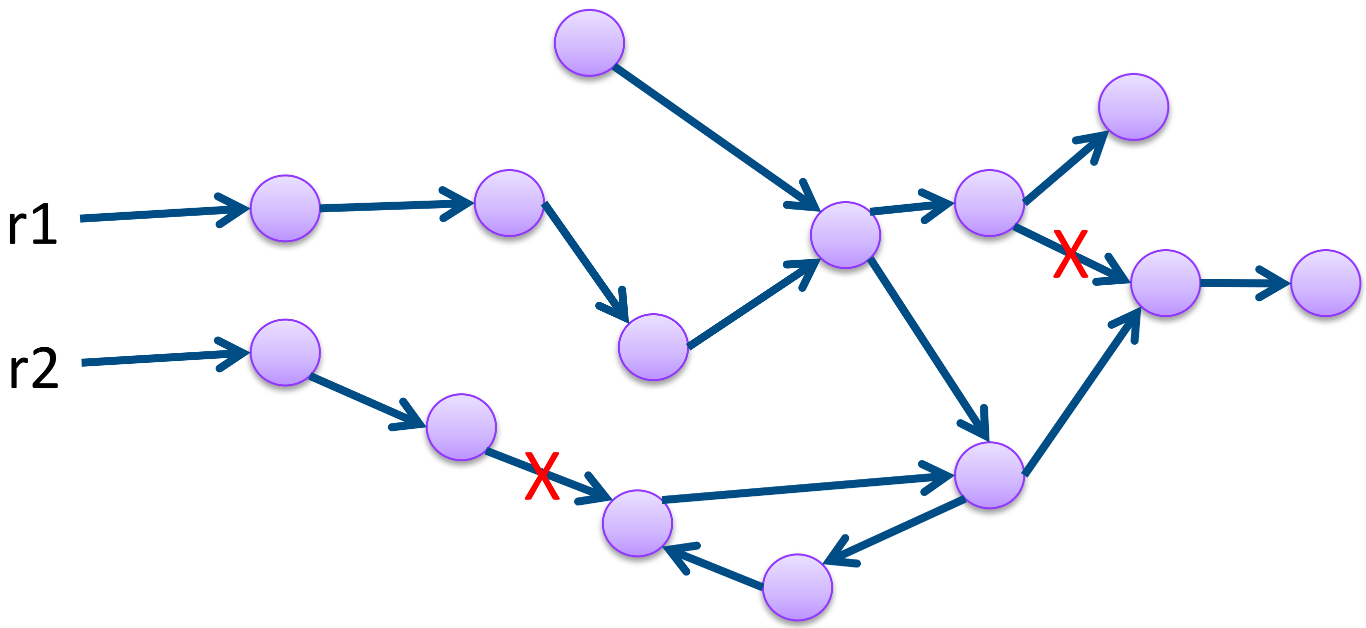
	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Size of heap (live objects)	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

Parallel Mark&Sweep GC



Parallel GC: mutator is stopped, GC threads run in parallel

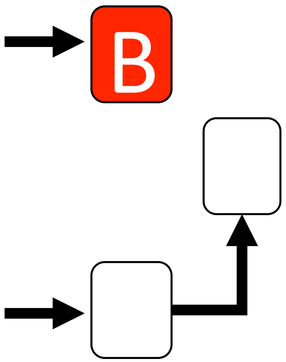
Concurrent Mark&Sweep Example



Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator

Problem: Interference

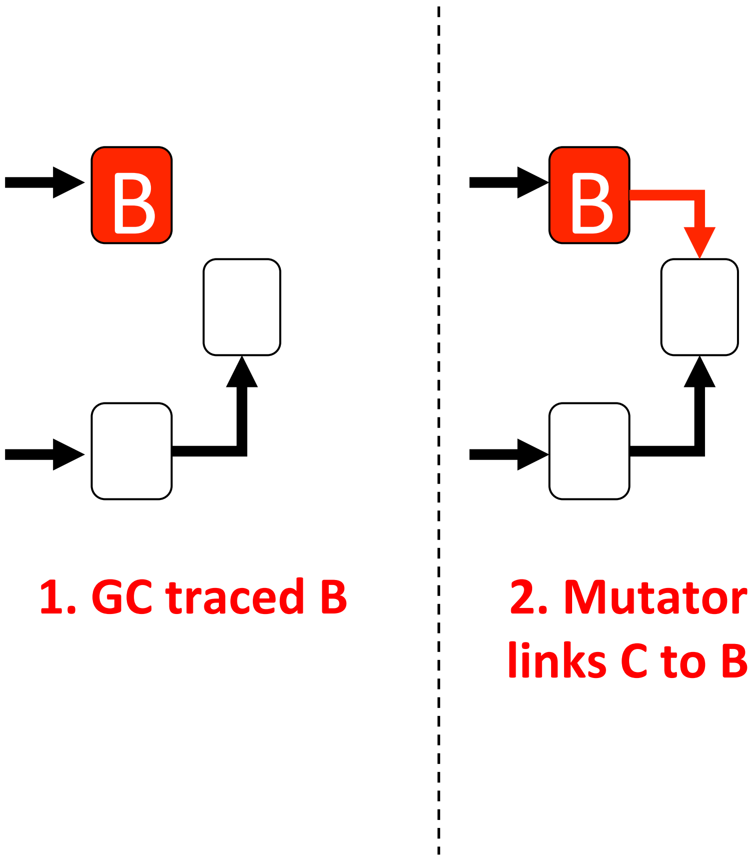
SYSTEM = MUTATOR || GC



1. GC traced B

Problem: Interference

SYSTEM = MUTATOR || GC

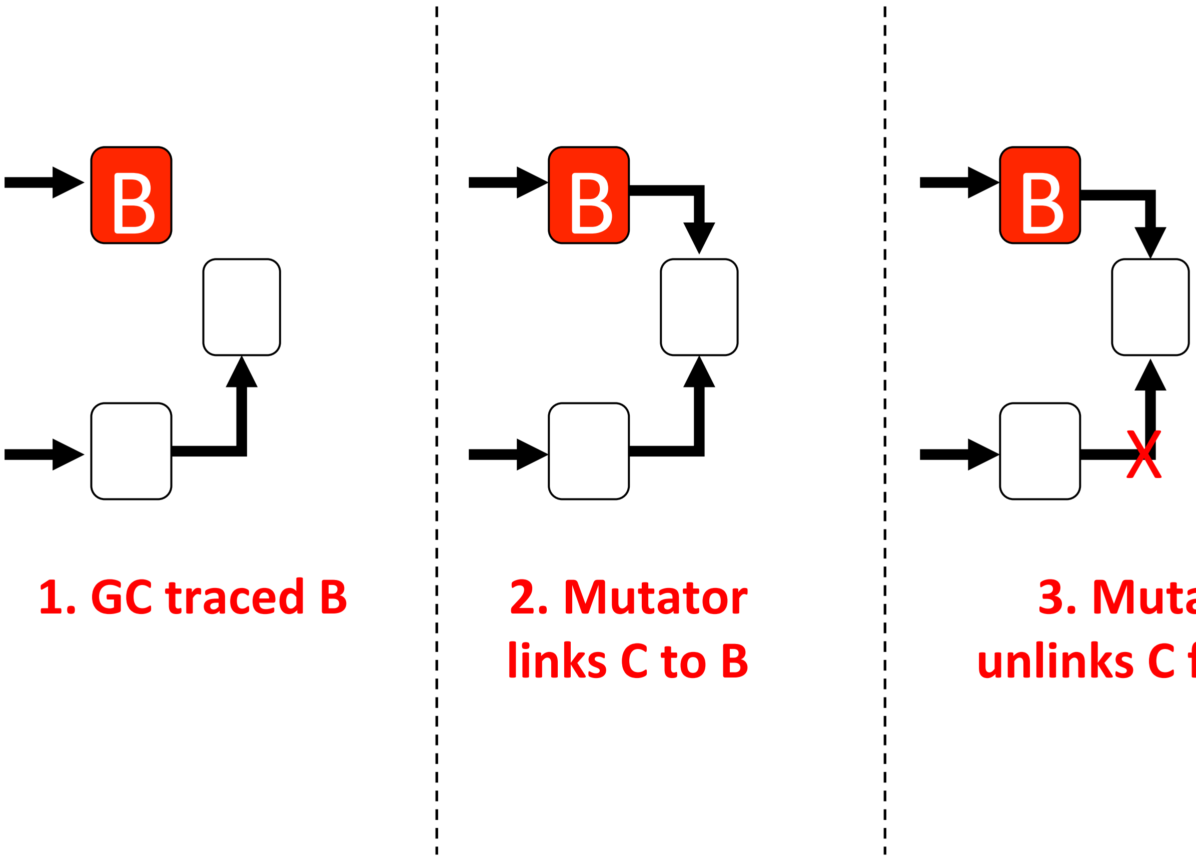


1. GC traced B

2. Mutator links C to B

Problem: Interference

SYSTEM = MUTATOR || GC



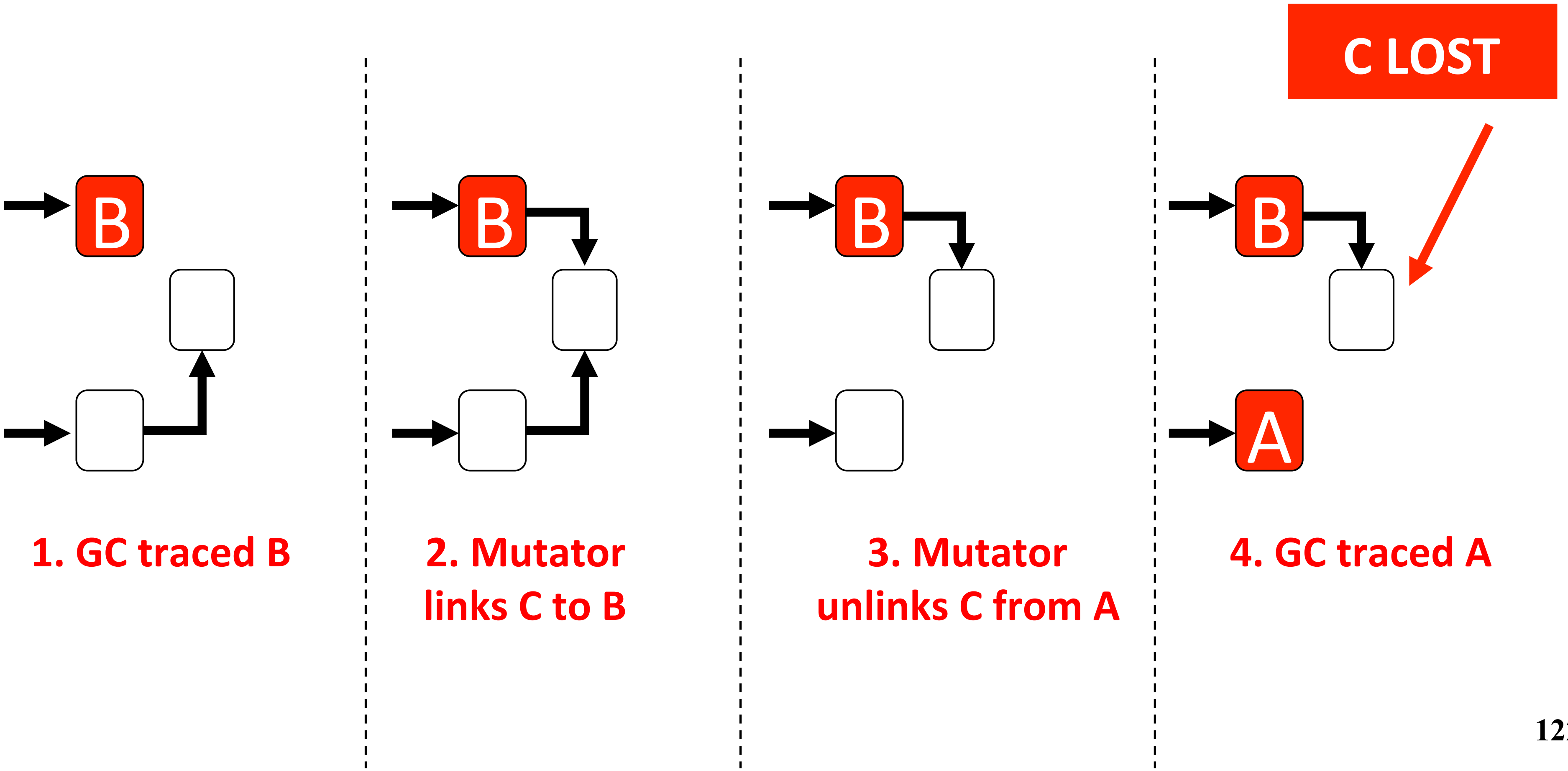
1. GC traced B

2. Mutator links C to B

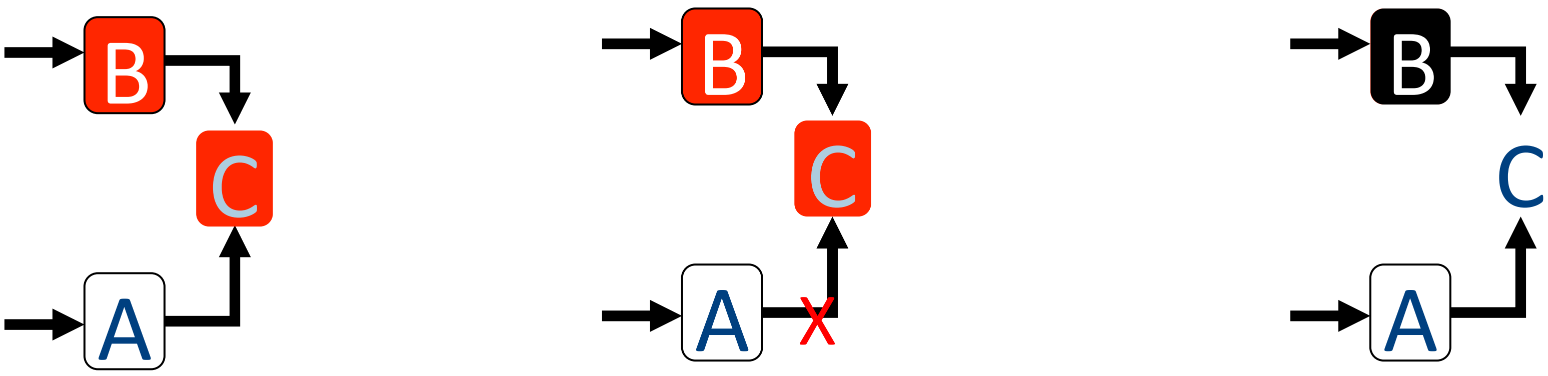
3. Mutator unlinks C from A

Problem: Interference

SYSTEM = MUTATOR || GC



The 3 Families of Concurrent GC Algorithms



Conservative GC

- How do you track pointers in languages such as C ?
  - Any value can be cast down to a pointer
- **How can you follow pointers in a structure?**
- Easy – be conservative, consider anything that can be a pointer to be a pointer
- Practical! (e.g., Boehm collector)

Conservative GC

- Can you implement a conservative **copying GC**?
- What is the problem?
- **Cannot update pointers to the new address… you don't know whether the value is a pointer, cannot update it**

Modern Memory Management

- Considers standard program properties
- Handle parallelism
  - Stop the program and collect in parallel on all available processors
  - Run collection concurrently with the program run
- Cache consciousness
- Real-time

Terminology Recap

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, compaction, tracing algorithms
- Fragmentation

Compilation

0368-3133 2014/15a

Lecture 13

Assembler, Linker and Loader

Noam Rinetzky

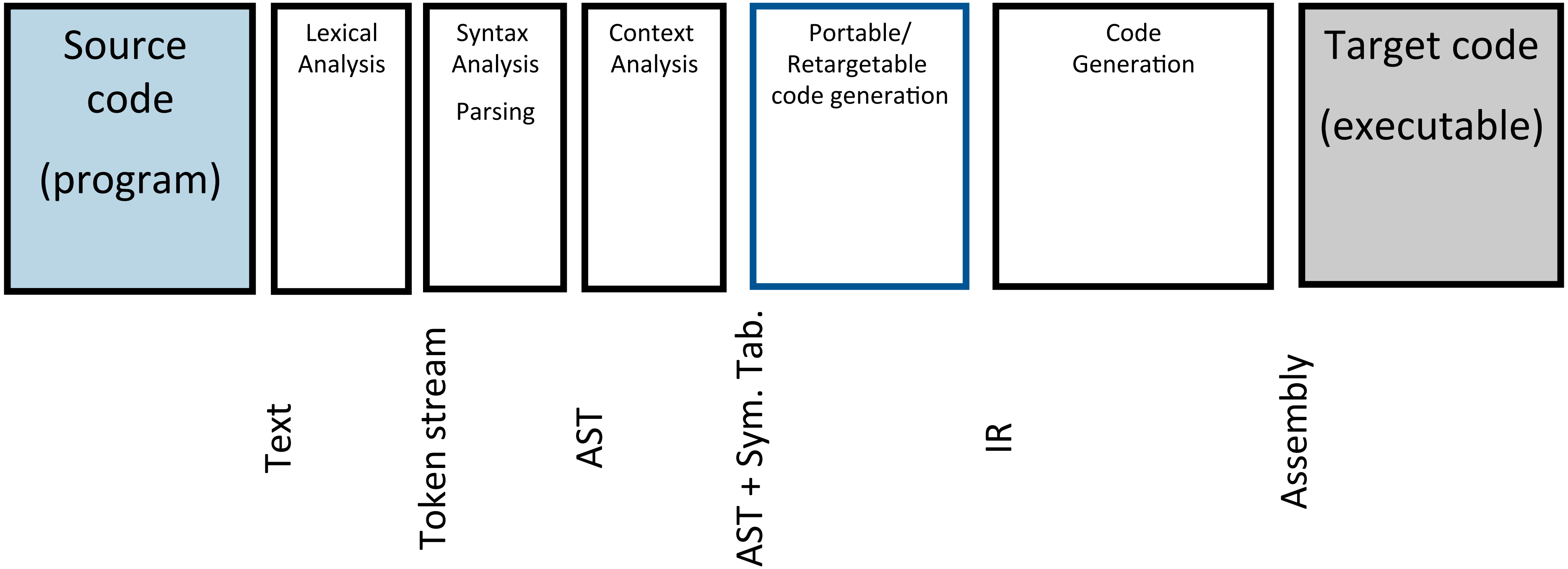
What is a compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

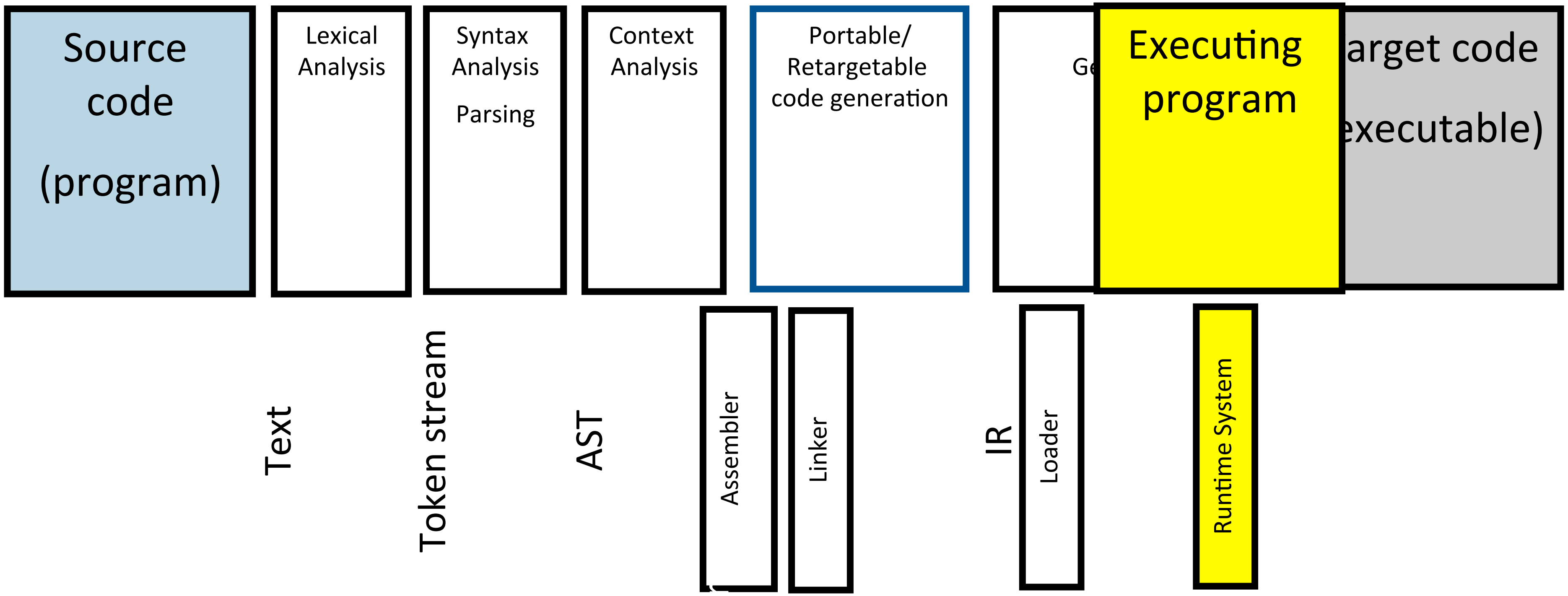
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

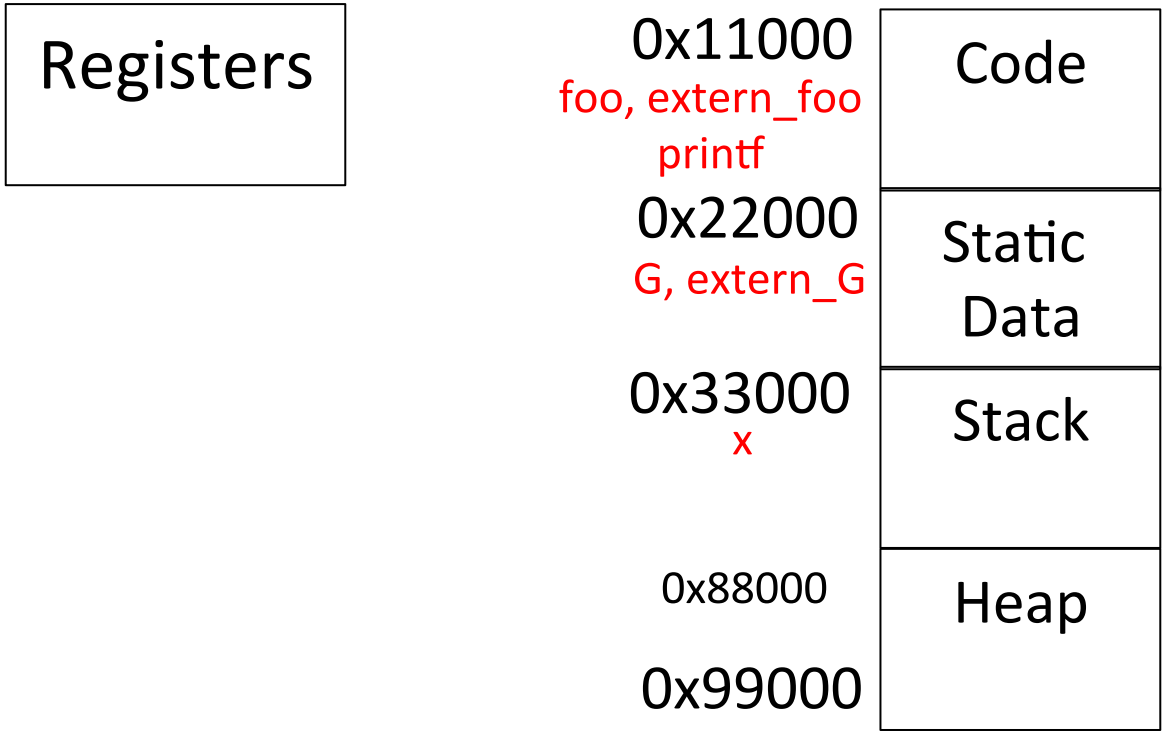
Stages of compilation



Compilation → Execution



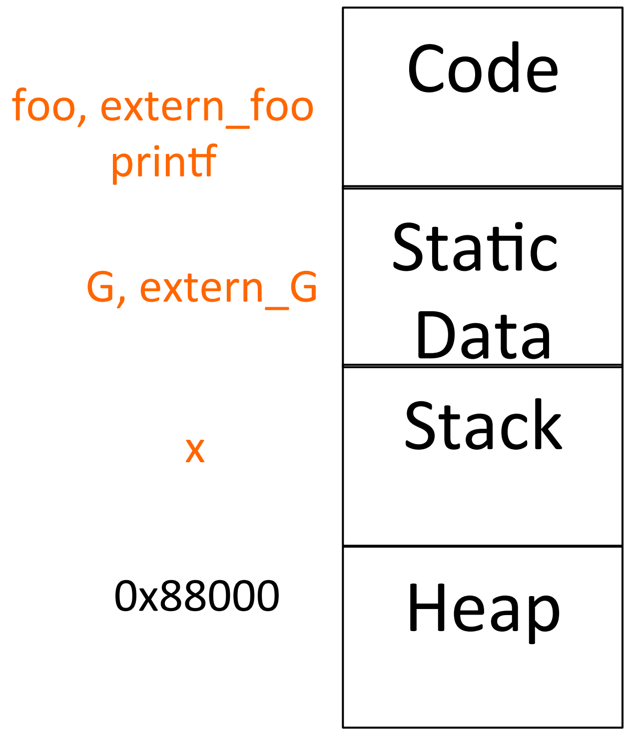
Program Runtime State



Challenges

- goto L2 → JMP 0x110FF
- G:=3 → MOV 0x2200F, 0..011
- foo() → CALL 0x130FF
- extern_G := 1 → MOV 0x2400F, 0..01
- extern_foo() → CALL 0x140FF
- printf() → CALL 0x150FF

- x:=2 → MOV FP+32, 0…010
- goto L2 → JMP [PC +] 0x000FF



Assembly → Image

Source program

Compiler

Assembly lang. program (.s)

Assembler

Machine lang. Module (.o): program (+library) modules

Linker

"compilation" time

Executable (".exe"):

"execution" time

Loader

Image (in memory):

Libraries (.o)
(dynamic loading)

Outline

- Assembly
- Linker / Link editor
- Loader

- Static linking
- Dynamic linking

Assembly → Image

Source file (*e.g., utils*)	Source file (*e.g., main*)	library
Assembly (.s)	Assembly (.s)	Assembly (.s)
Object (.o)	Object (.o)	Object (.o)

Executable (".elf")

Image (in memory):

Assembler

- Converts (symbolic) assembler to binary (object) code
    - Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory
  - Yet another(simple) compiler
    - One-to one translation
- Converts constants to machine repr. (3➔0…011)
- Resolve internal references
- Records info for code & data relocation

Object File Format

Header	Text Segment	Data Segment	Relocation Information	Symbol Table	Debugging Information

- Header: Admin info + “file map”
- Text seg.: machine instruction
- Data seg.: (Initialized) data in machine format
- Relocation info: instructions and data that depend on absolute addresses
- Symbol table: “exported” references + unresolved references

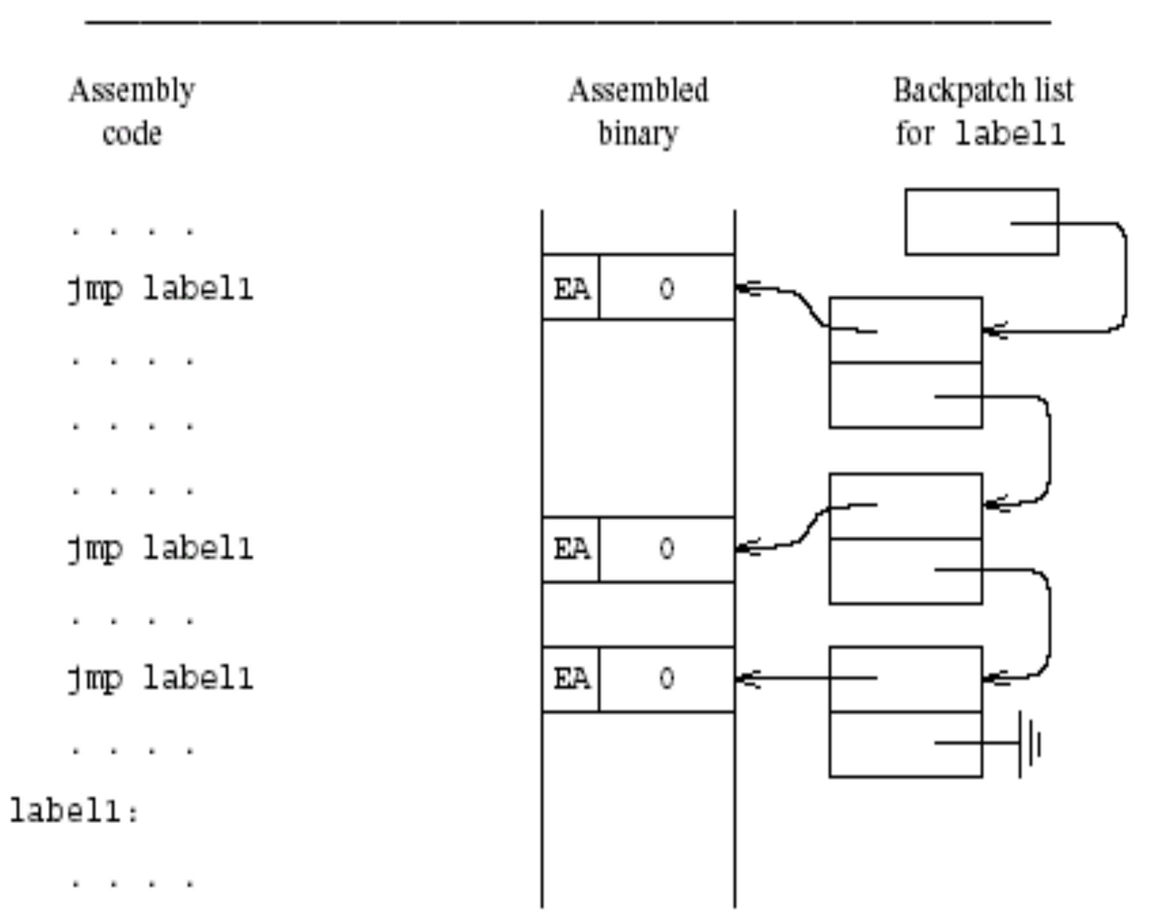
Handling Internal Addresses

```
.data
            ...
            .align 8
var1:
            .long 666
            ...
.code
            ...
            addl var1,%eax
            ...
            jmp label1
            ...
label1:
            ...
            ...
```

Resolving Internal Addresses

- Two scans of the code
  - Construct a table label → address
  - Replace labels with values
- One scan of the code (Backpatching)
  - Simultaneously construct the table and resolve symbolic addresses
    - Maintains list of unresolved labels
  - Useful beyond assemblers

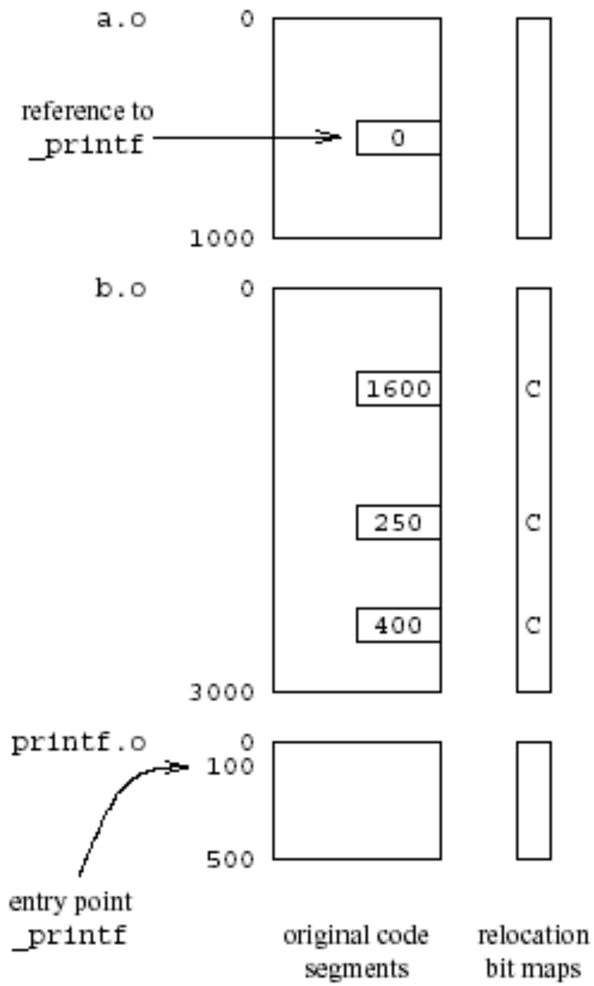
Backpatching



Handling External Addresses

- Record symbol table in “external” table
  - Exported (defined) symbols
    - G, foo()
  - Imported (required) symbols
    - Extern_G, extern_bar(), printf()
- Relocation bits
  - Mark instructions that depend on absolute (fixed) addresses
    - Instructions using globals,

Example



External references resolved by the Linker using the relocation info.

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

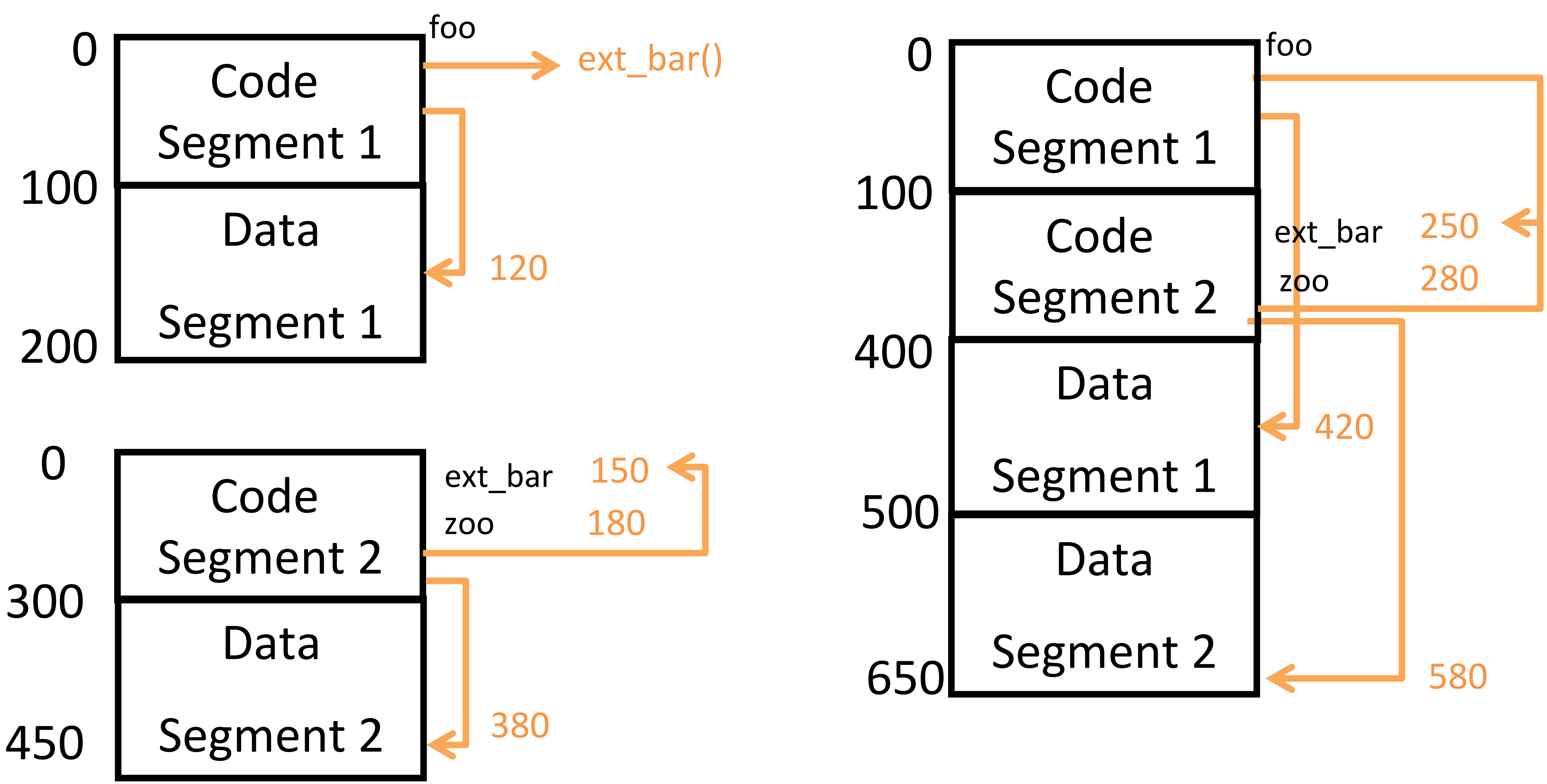
Assembler Summary

- Converts symbolic machine code to binary
  - addl %edx, %ecx ⇒ 000 0001 11 010 001 = 01 D1 (Hex)
- Format conversions
  - 3 ➔ 0x0..011 or 0x000000110…0
- Resolves internal addresses
- Some assemblers support overloading
  - Different opcodes based on types

Linker

- Merges object files to an executable
  - Enables separate compilation
- Combine memory layouts of object modules
  - Links program calls to library routines
    - printf(), malloc()
  - Relocates instructions by adjusting absolute references
  - Resolves references among files

Linker



0
Code
Segment 1
100
Data
Segment 1
200
foo
ext_bar()
120
0
Code
Segment 2
300
Data
Segment 2
450
ext_bar
zoo
150
180
380
0
Code
Segment 1
100
Code
Segment 2
400
Data
Segment 1
500
Data
Segment 2
650
foo
ext_bar
zoo
250
280
420
580

Relocation information

- Information needed to change addresses

- Positions in the code which contains addresses
  - Data
  - Code

- Two implementations
  - Bitmap
  - Linked-lists

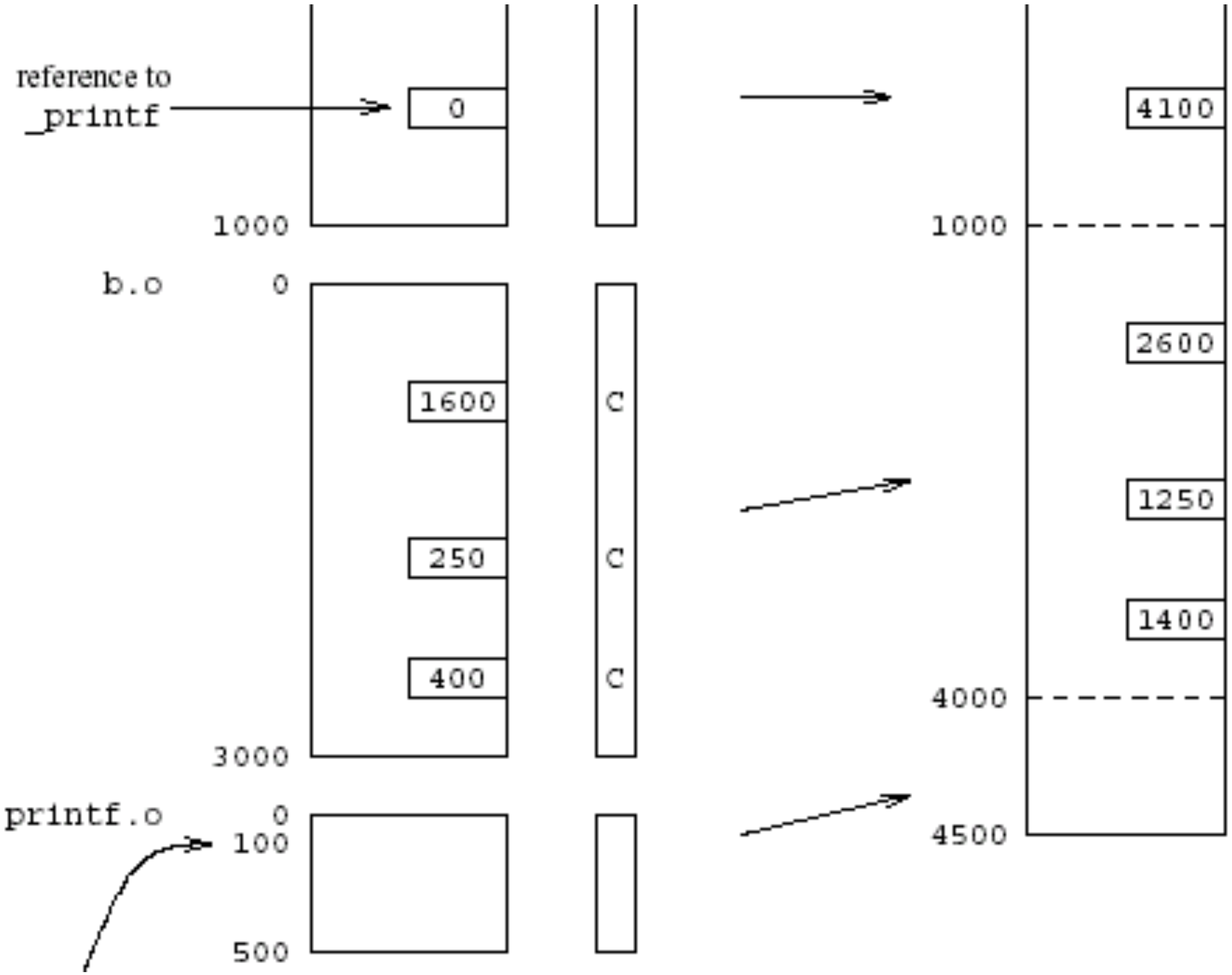
External References

- The code may include references to external names (identifiers)
  - Library calls
  - External data
- Stored in external symbol table

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Example


reference to
_printf
0
1000
b.o
0
1600
250
400
c
c
c
3000
printf.o
0
100
500
4100
1000
2600
1250
1400
4000
4500

Linker (Summary)

- Merge several executables
  - Resolve external references
  - Relocate addresses
- User mode
- Provided by the operating system
  - But can be specific for the compiler
    - More secure code
    - Better error diagnosis

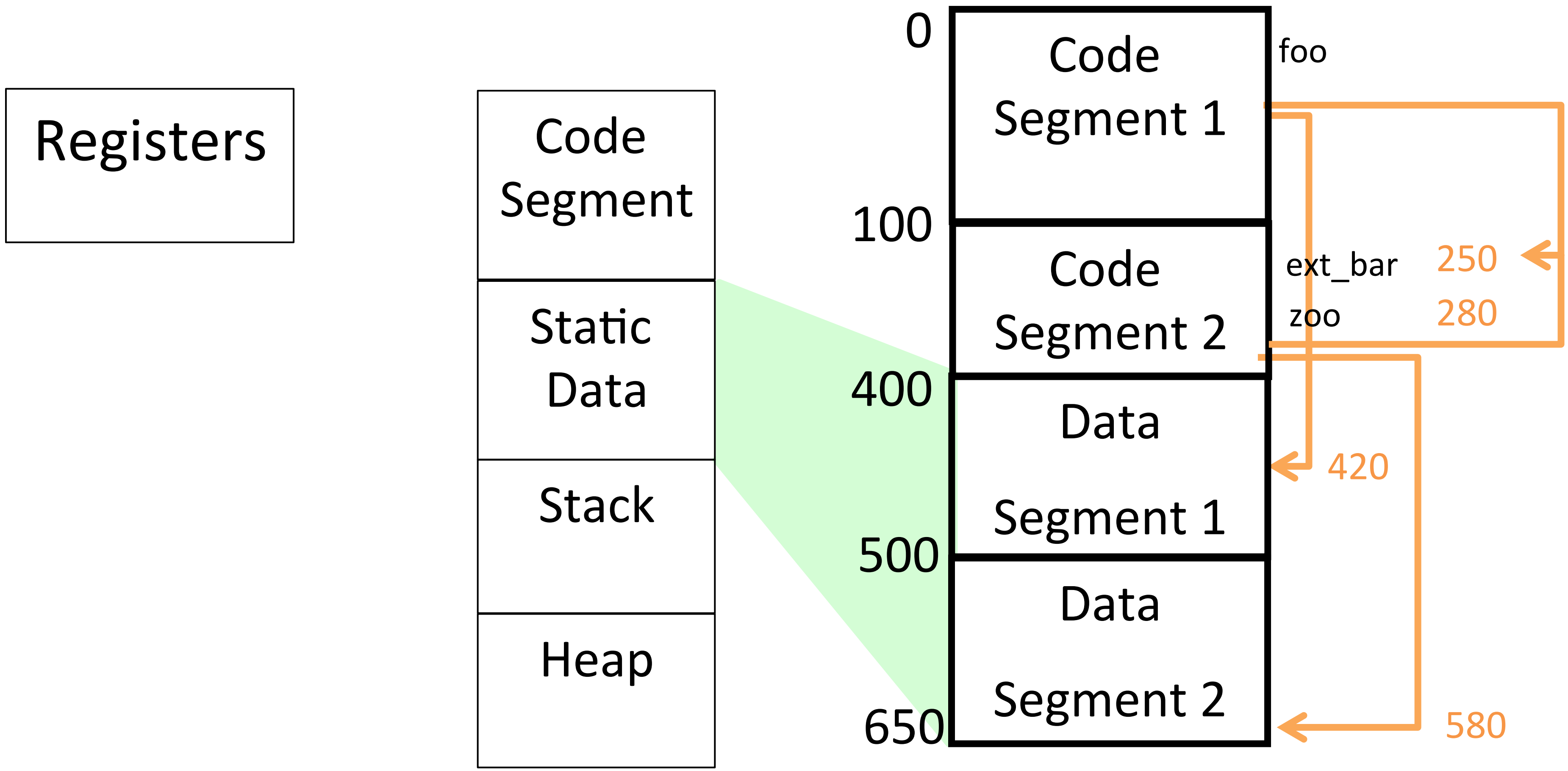
Linker Design Issues

- Merges
  - Code segments
  - Data segments
  - Relocation bit maps
  - External symbol tables
- Retain information about static length
- Real life complications
  - Aggregate initializations
  - Object file formats
  - Large library
  - Efficient search procedures

Loader

- Brings an executable file from disk into memory and starts it running
  - Read executable file's header to determine the size of text and data segments
  - Create a new address space for the program
  - Copies instructions and data into memory
  - Copies arguments passed to the program on the stack
- Initializes the machine registers including the stack ptr
- Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine

Program Loading



Loader (Summary)

- Initializes the runtime state
- Part of the operating system
  - Privileged mode
- Does not depend on the programming language
- “Invisible activation record”

Static Linking (Recap)

- Assembler generates binary code
  - Unresolved addresses
  - Relocatable addresses
- Linker generates executable code
- Loader generates runtime states (images)

Dynamic Linking

- Why dynamic linking?
  - Shared libraries
    - Save space
    - Consistency
  - Dynamic loading
    - Load on demand

What's the challenge?

Source program

Compiler

Assembly lang. program (.s)

Assembler

Machine lang. Module (.o): program (+library) modules

Linker

"compilation" time

Executable (".exe"):

"execution" time

Loader

Image (in memory):

Libraries (.o)
(dynamic linking)

Position-Independent Code (PIC)

- Code which does not need to be changed regardless of the address in which it is loaded
  - Enable loading the same object file at different addresses
    - Thus, shared libraries and dynamic loading
- “Good” instructions for PIC: use relative addresses
  - relative jumps
  - reference to activation records
- “Bad” instructions for : use fixed addresses
  - Accessing global and static data
  - Procedure calls
    - Where are the library procedures located?

How?

“All problems in computer science can be solved by another level of indirection"

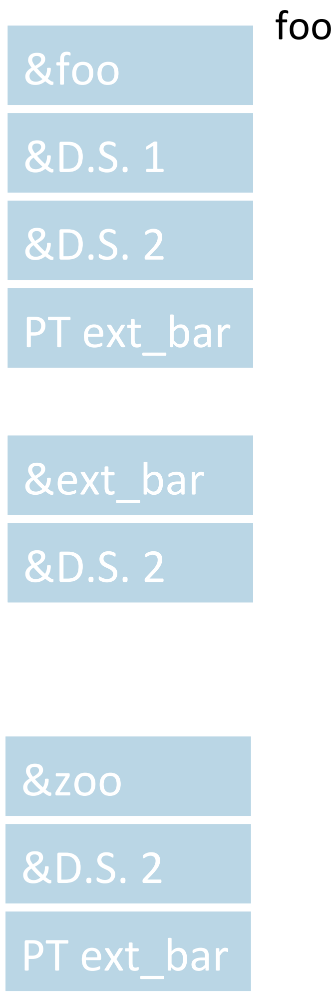
Butler Lampson

PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

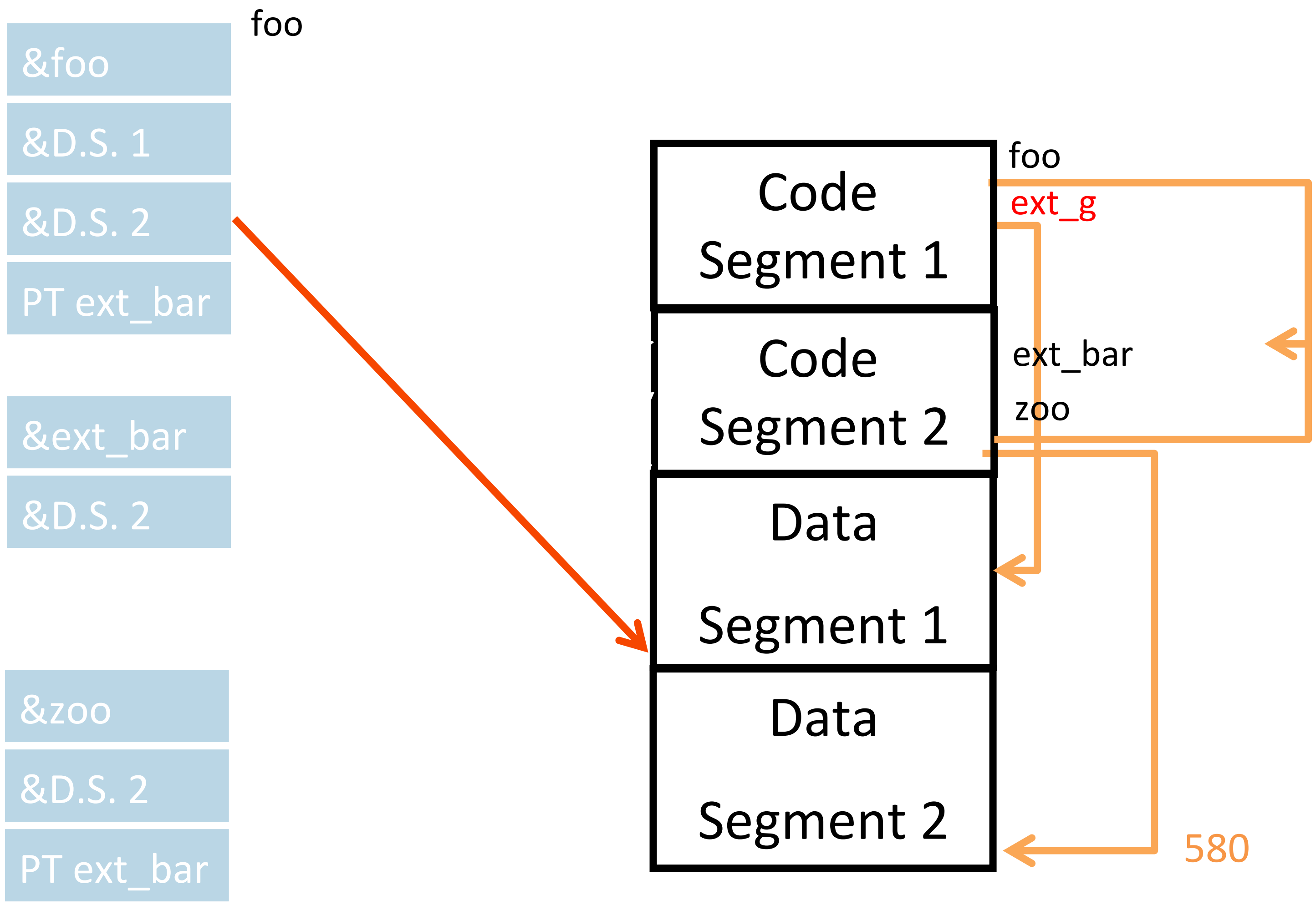
Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

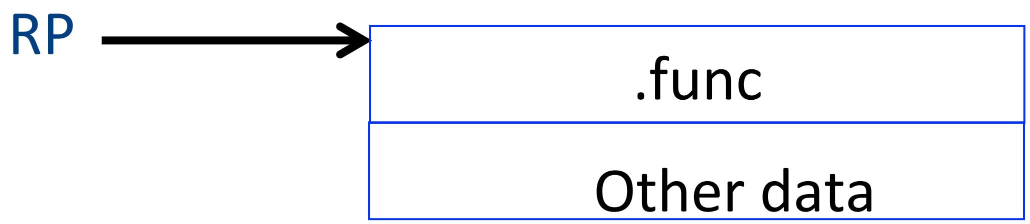
- Record for every routine in a table
- Record used as a address to procedure

Caller:

1. Load Pointer table address into RP
2. Load Code address from 0(RP) into RC
3. Call via RC

Callee:

1. RP points to pointer table
2. Table has addresses of pointer table for subprocedures



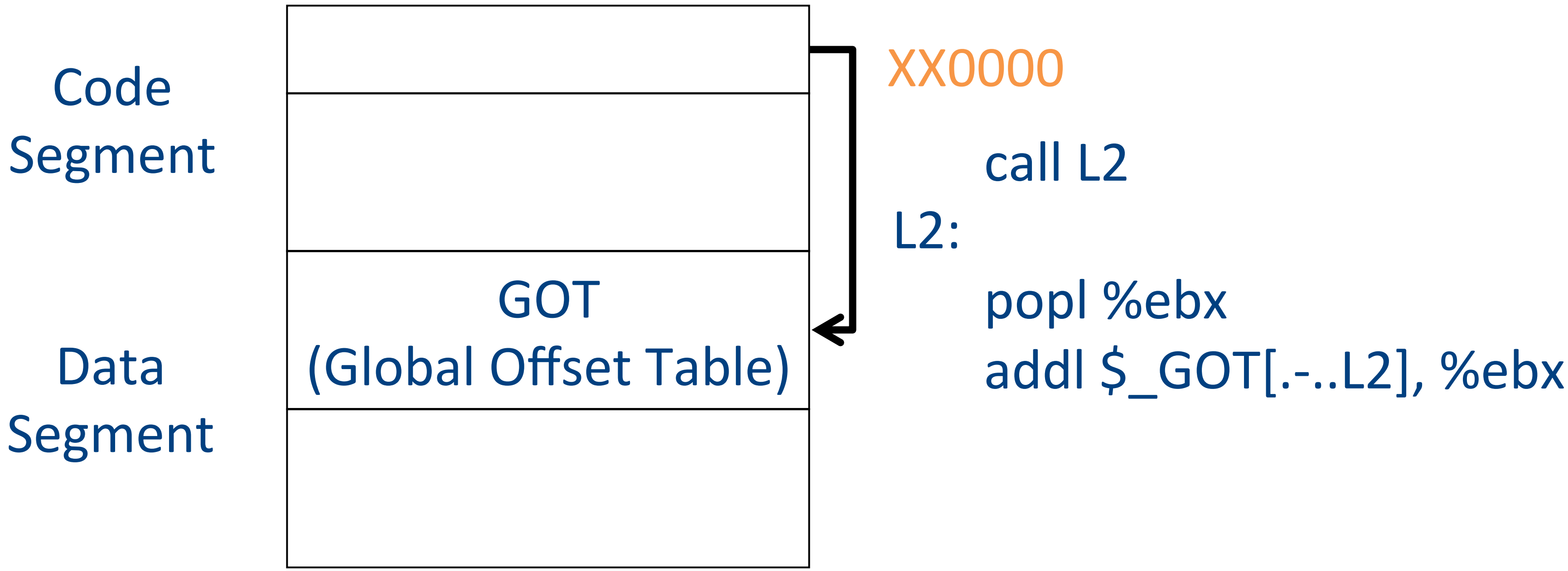
PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

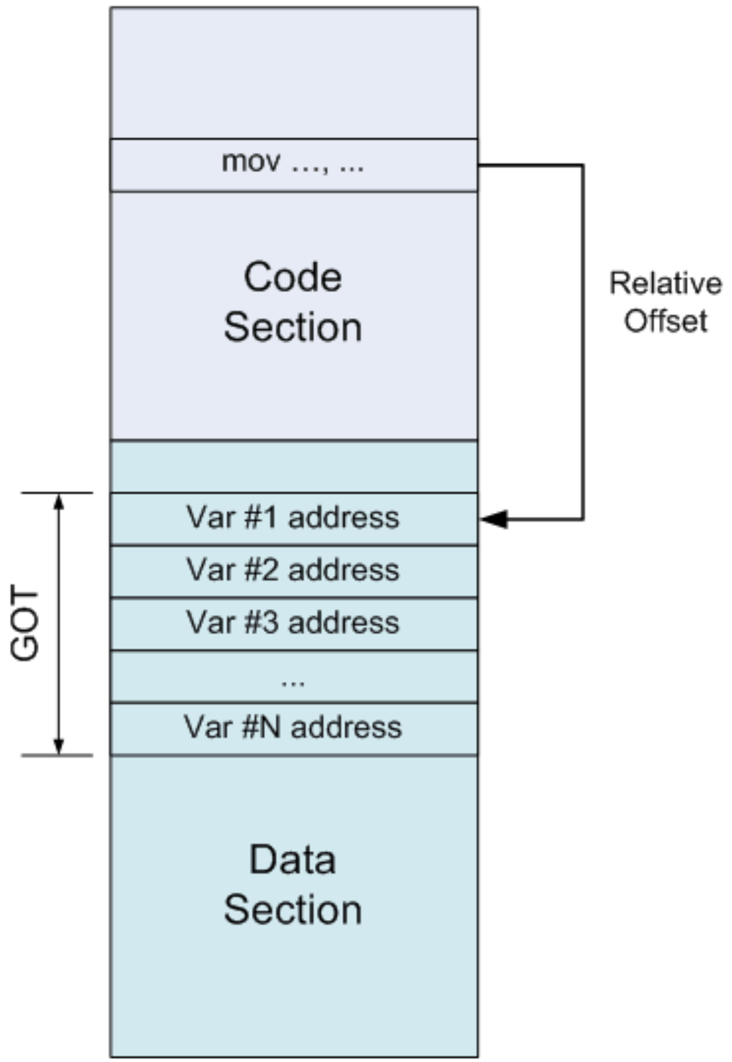
- Efficiency: use a register to point to the beginning of the table
  - Troublesome in CISC machines

ELF-Position Independent Code

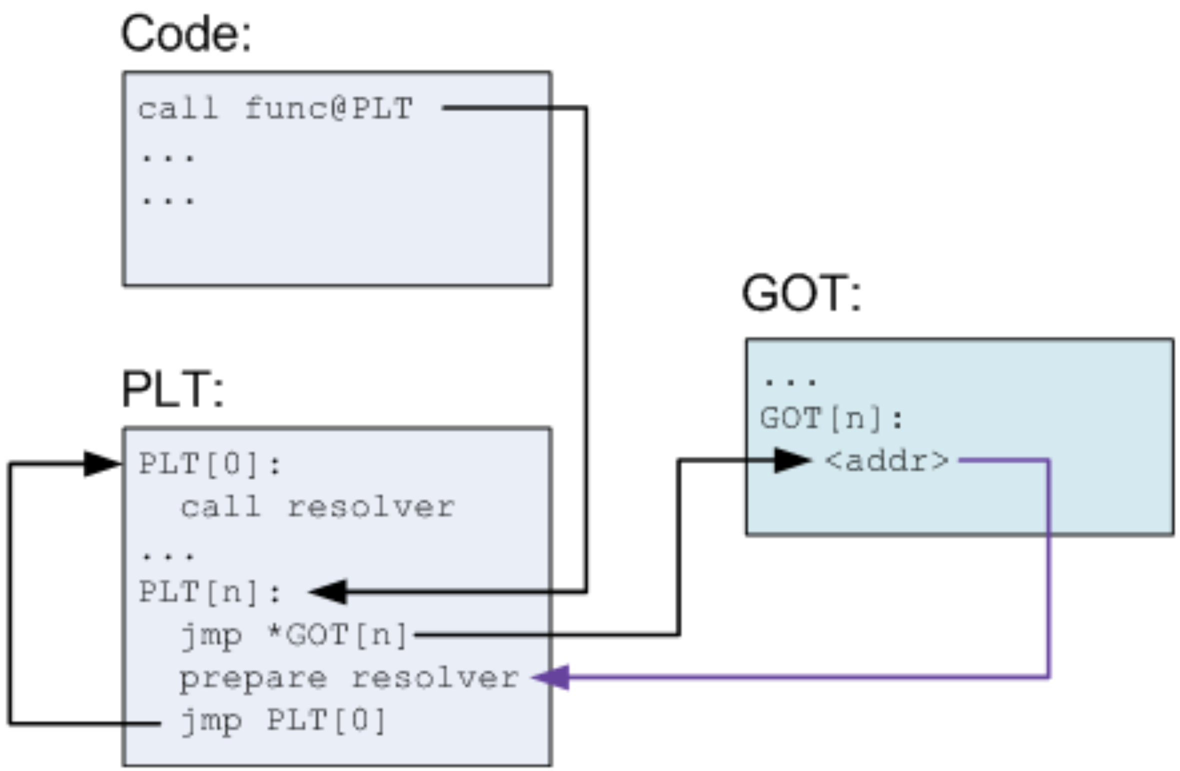
- Executable and Linkable code Format
  - Introduced in Unix System V
- Observation
  - Executable consists of code followed by data
  - The offset of the data from the beginning of the code is known at compile-time



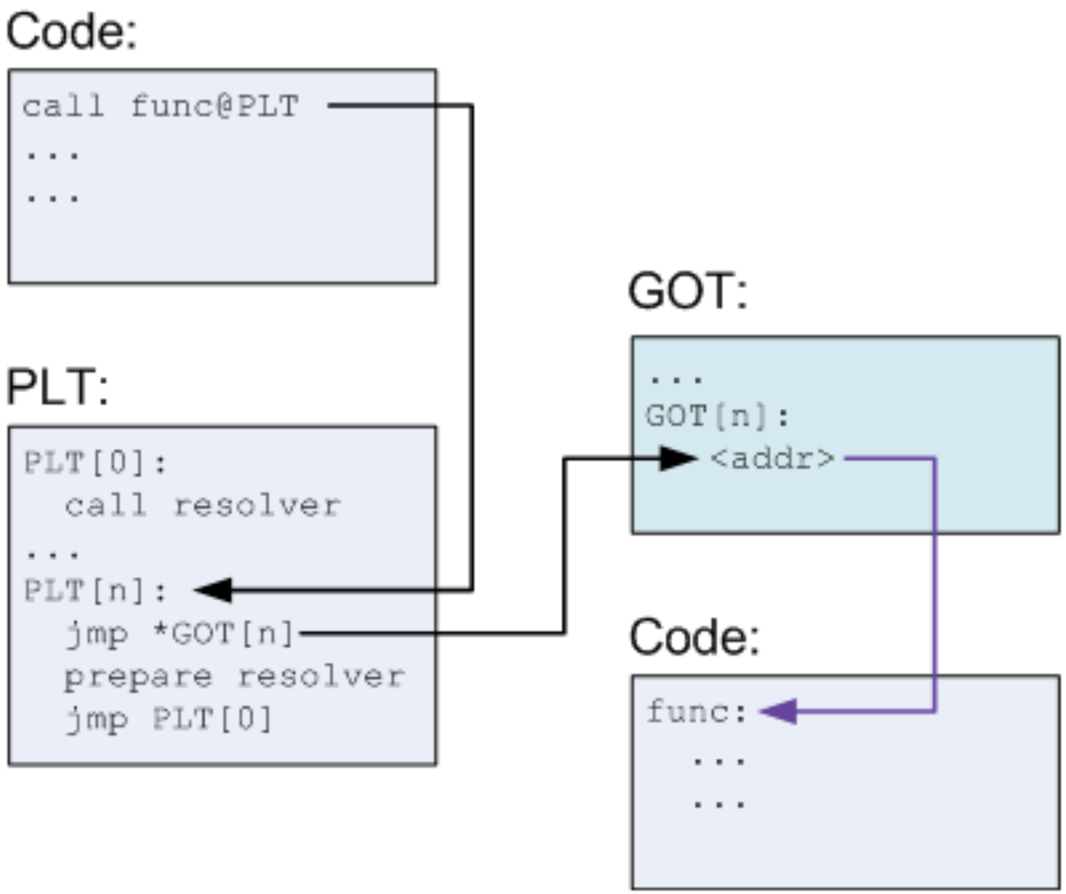
ELF: Accessing global data



ELF: Calling Procedures
(before 1st call)



ELF: Calling Procedures
(after 1st call)



PIC benefits and costs

- Enable loading w/o relocation
- Share memory locations among processes

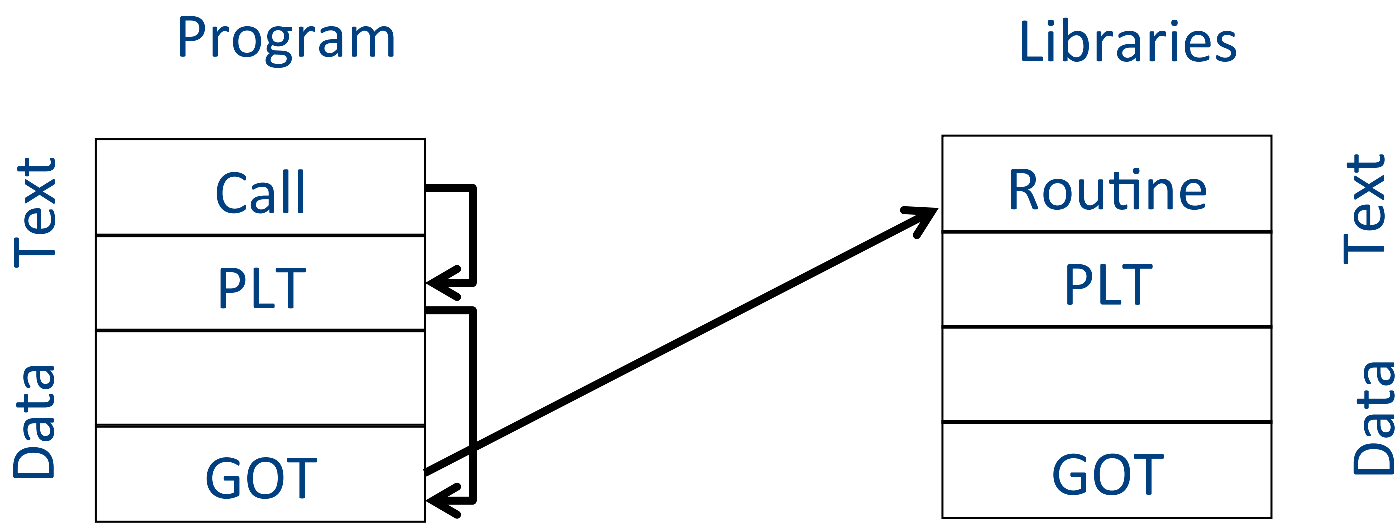
- Data segment may need to be reloaded
- GOT can be large
- More runtime overhead
- More space overhead

Shared Libraries

- Heavily used libraries
- Significant code space
  - 5-10 Mega for print
  - Significant disk space
  - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

- Can be implemented either with static or dynamic loading

Content of ELF file



Consistency

- How to guarantee that the code/library used the "right" library version

Loading Dynamically Linked Programs

- Start the dynamic linker
- Find the libraries
- Initialization
  - Resolve symbols
  - GOT
    - Typically small
  - Library specific initialization
- Lazy procedure linkage

Microsoft Dynamic Libraries (DLL)

- Similar to ELF
- Somewhat simpler
- Require compiler support to address dynamic libraries
- Programs and DLL are Portable Executable (PE)
- Each application has it own address
- Supports lazy bindings

Dynamic Linking Approaches

- Unix/ELF uses a single name space space and MS/PE uses several name spaces
- ELF executable lists the names of symbols and libraries it needs
- PE file lists the libraries to import from other libraries
- ELF is more flexible
- PE is more efficient

Costs of dynamic loading

- Load time relocation of libraries
- Load time resolution of libraries and executable
- Overhead from PIC prolog
- Overhead from indirect addressing
- Reserved registers

Summary

- Code generation yields code which is still far from executable
  - Delegate to existing assembler
- Assembler translates symbolic instructions into binary and creates relocation bits
- Linker creates executable from several files produced by the assembly
- Loader creates an image from executable

Compilation

0368-3133 2014/15a

Lecture 13

RECAP

Noam Rinetzky

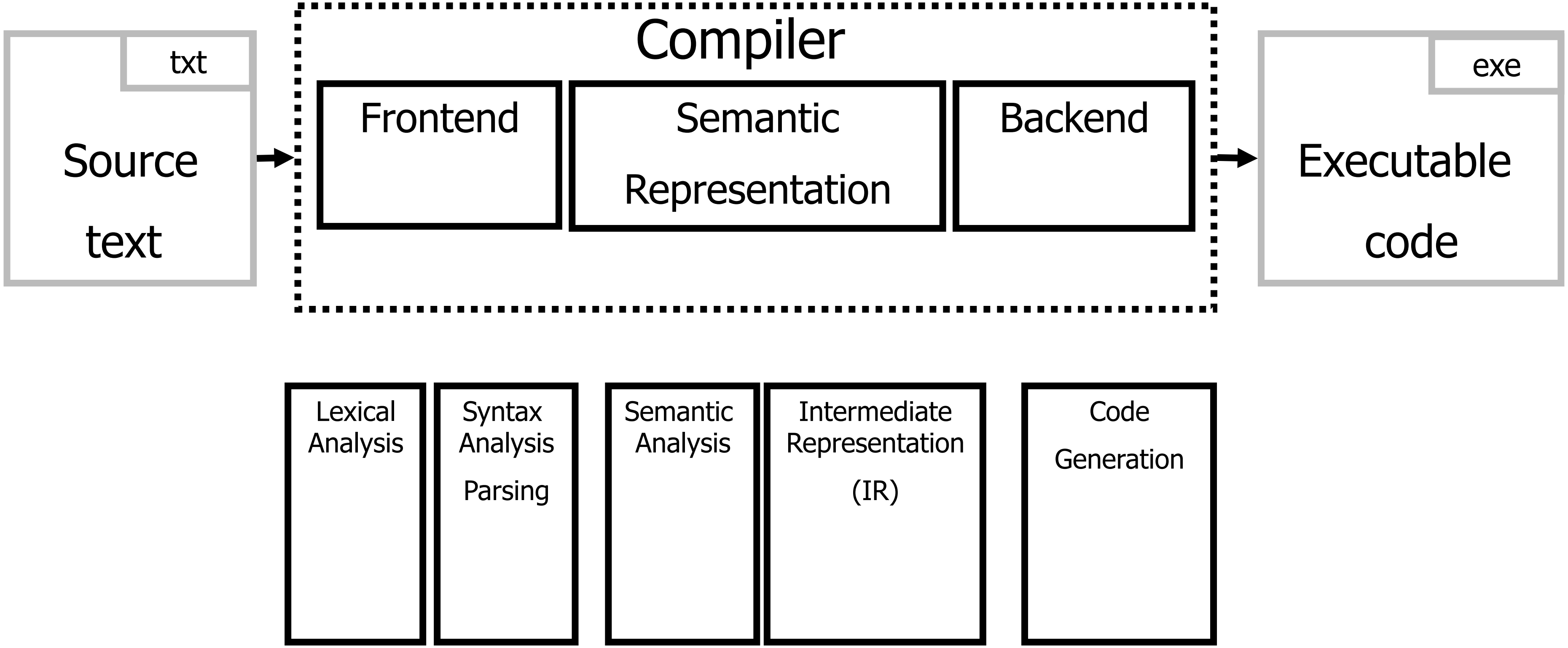
What is a compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

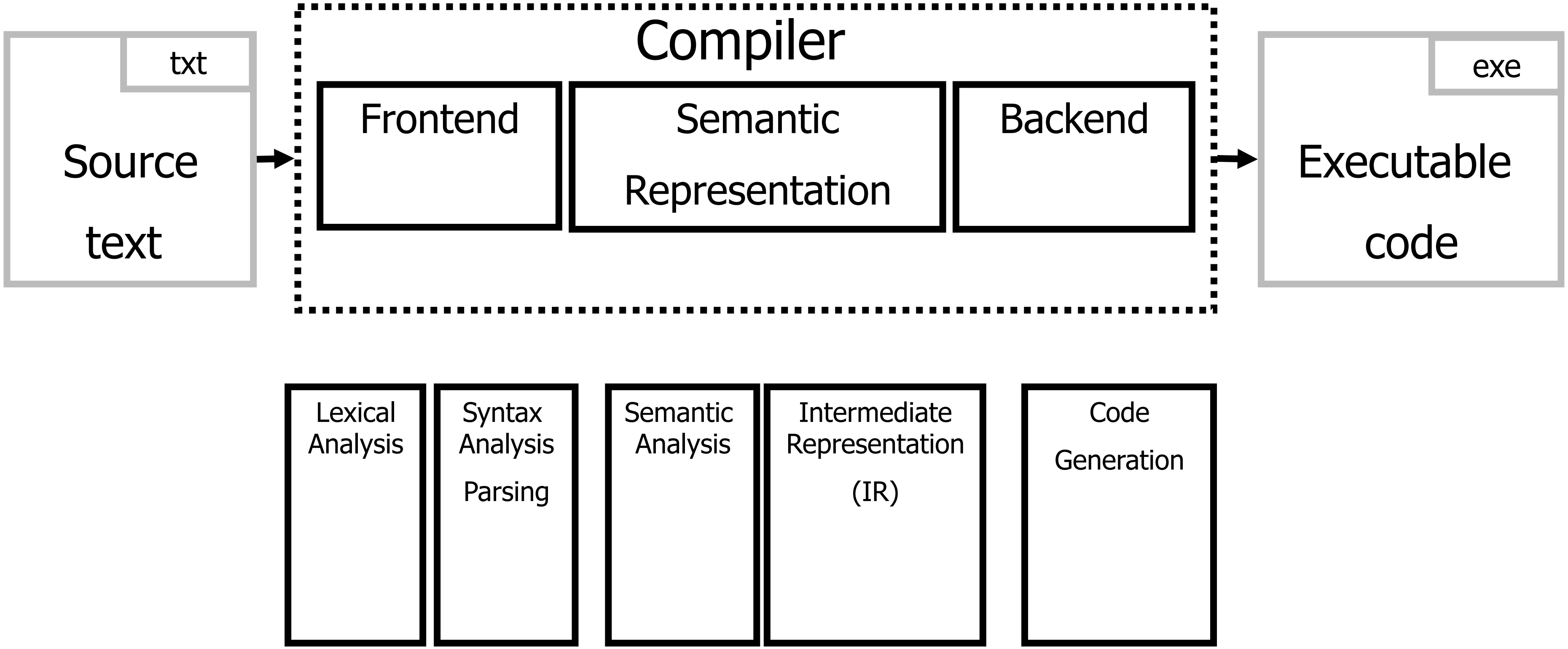
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

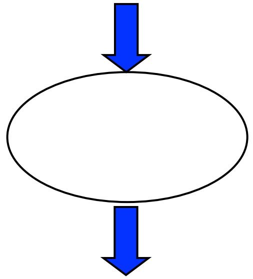
Conceptual Structure of a Compiler



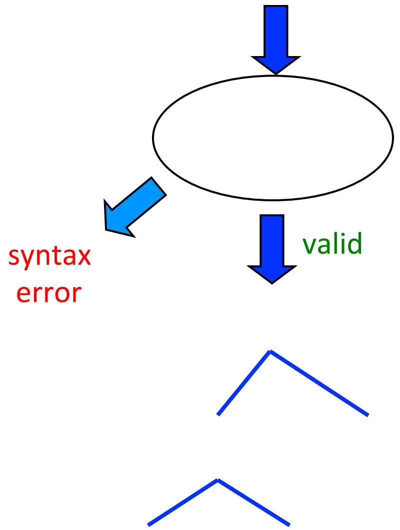
Conceptual Structure of a Compiler



From scanning to parsing



(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

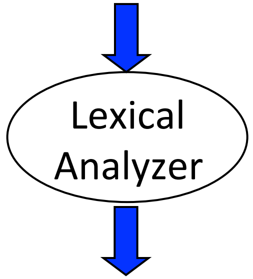


Id

Id

From scanning to parsing

program text **((23 + 7) * x)**



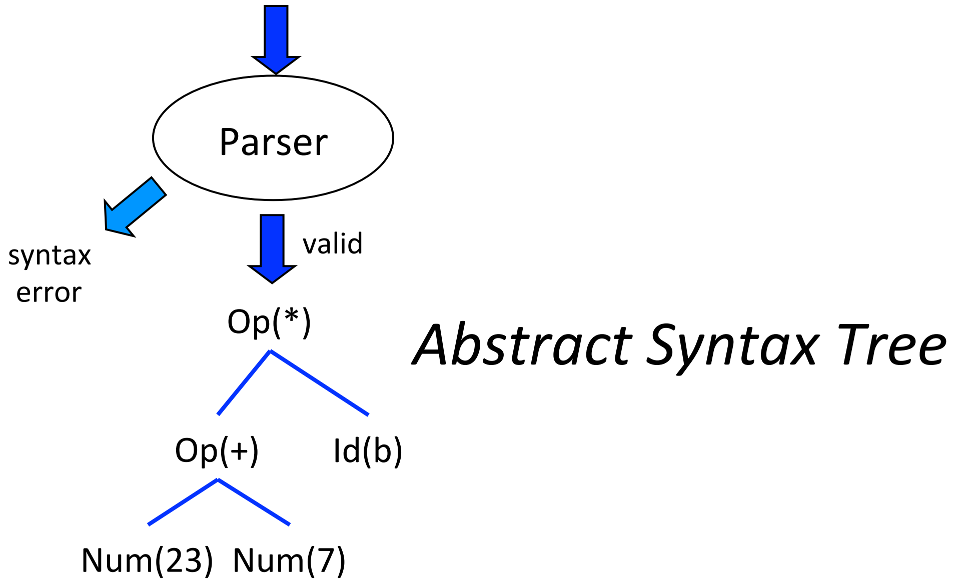
token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

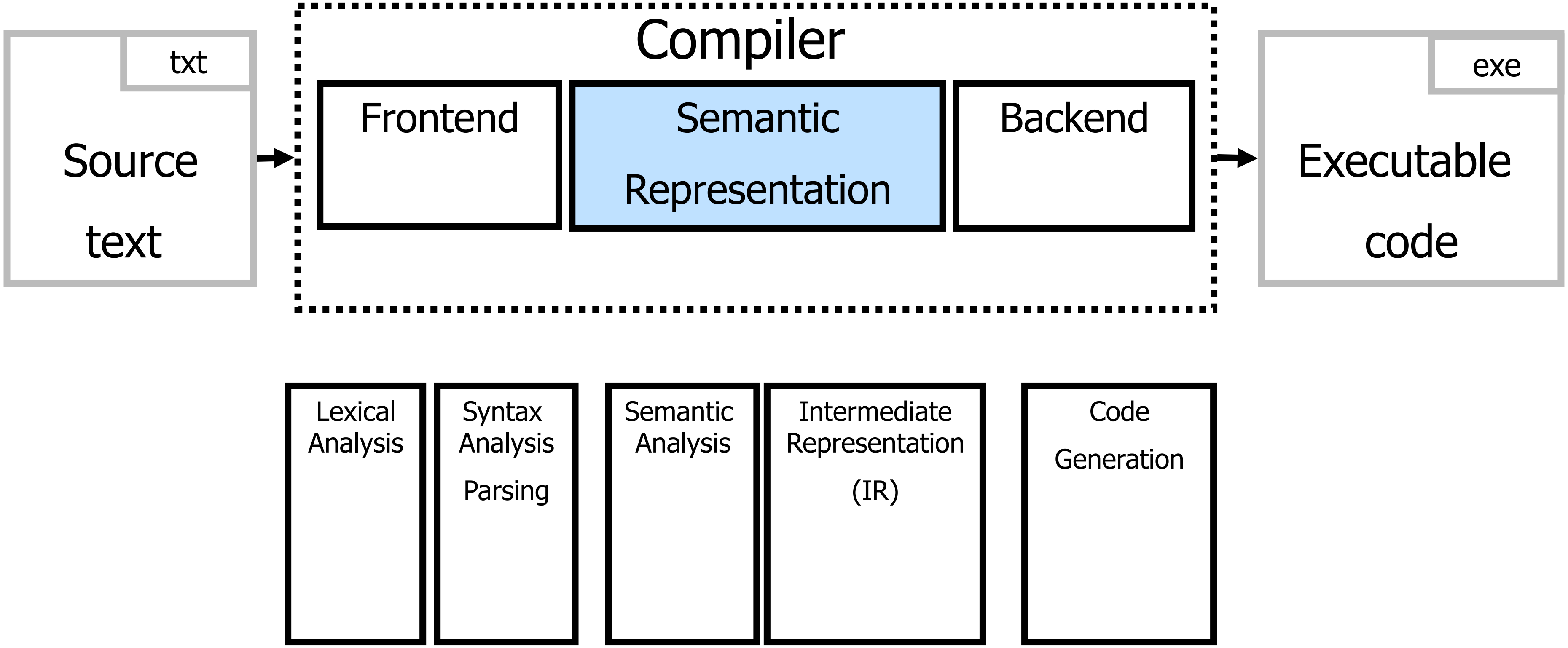
Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’



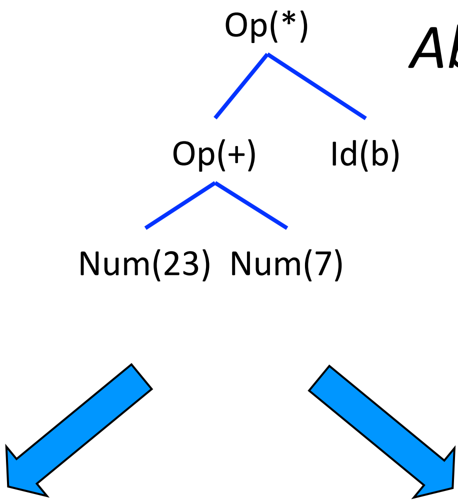
Conceptual Structure of a Compiler



Context Analysis

Type rules

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$



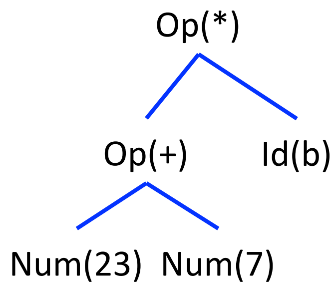
Abstract Syntax Tree

Semantic Error

Valid + Symbol Table

Code Generation

cgen
Frame Manager



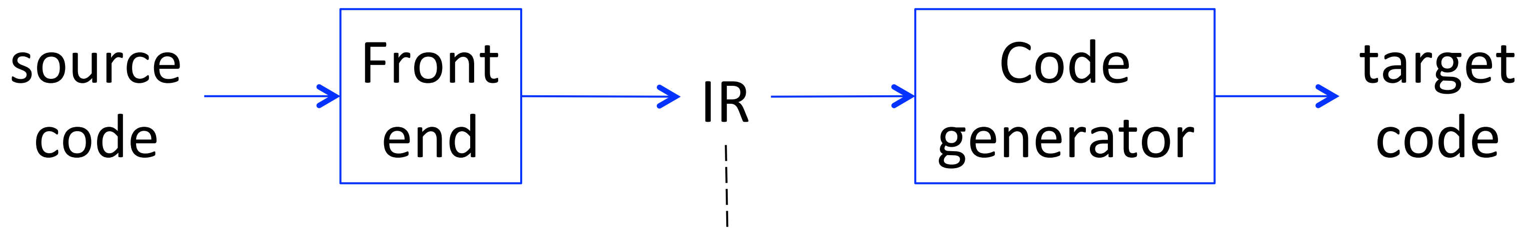
Valid Abstract Syntax Tree
Symbol Table

Verification (possible runtime)
Errors/Warnings

Intermediate Representation (IR)

input → Executable Code → output

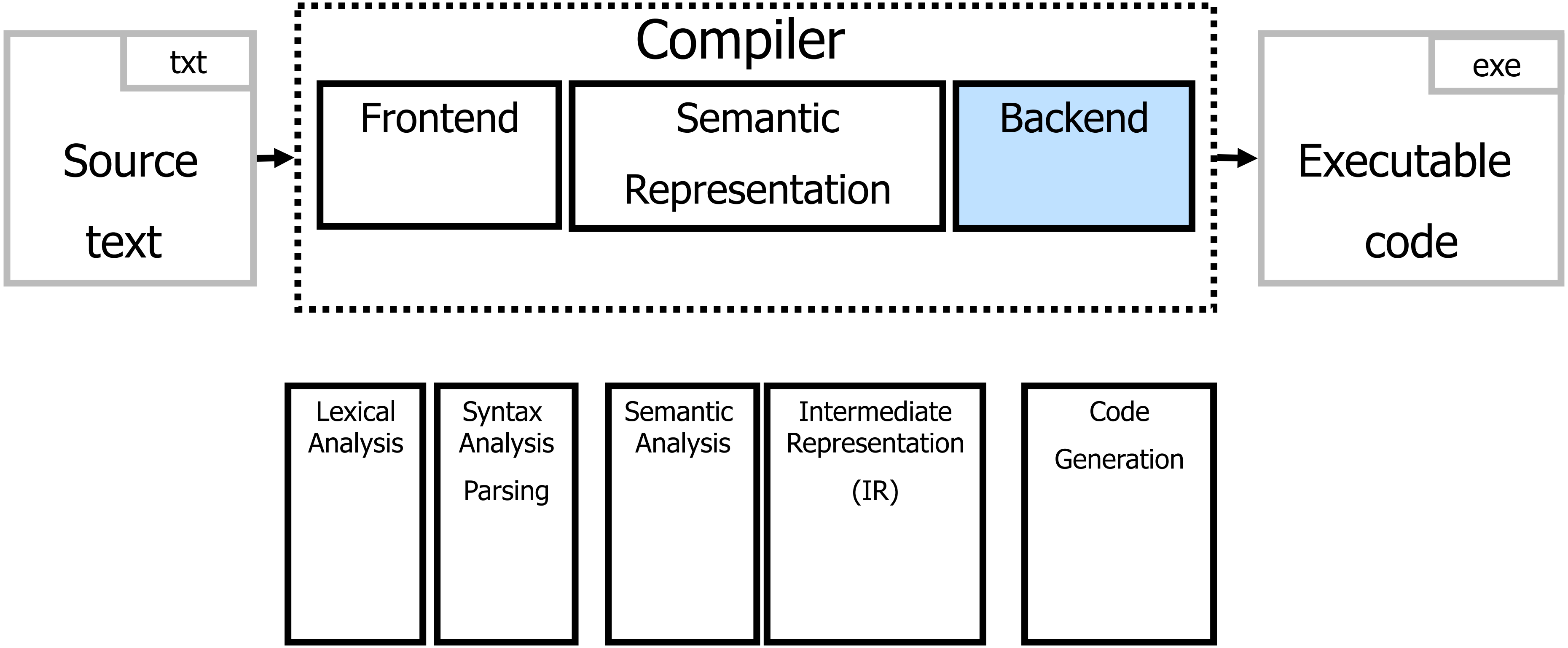
Optimization



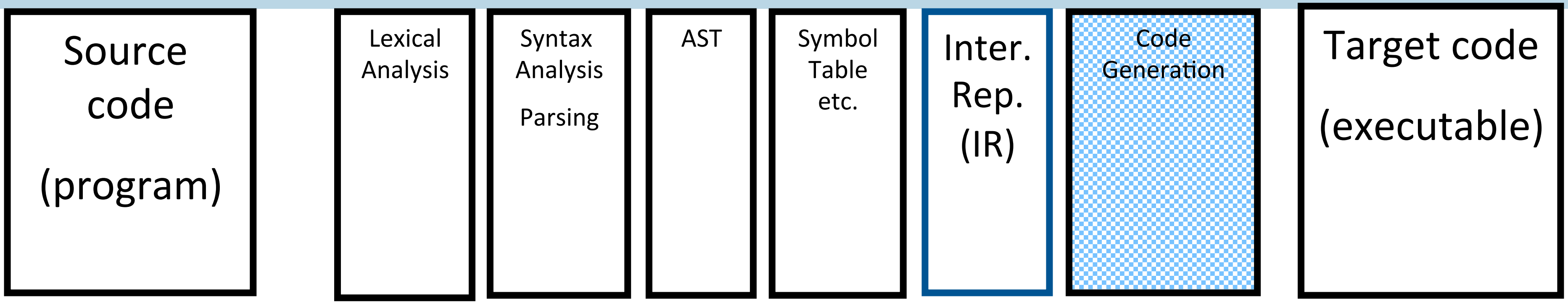
Program Analysis
Abstract interpretation

Can appear in later stages too

Conceptual Structure of a Compiler

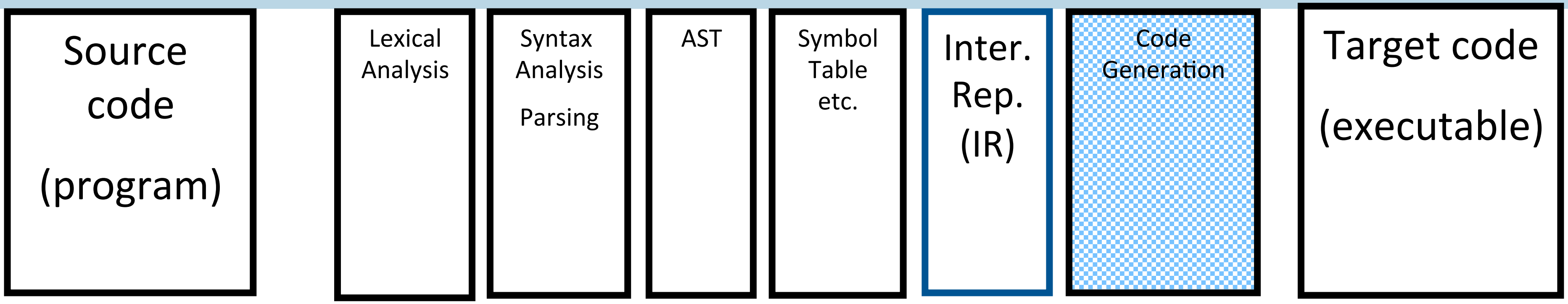


Register Allocation



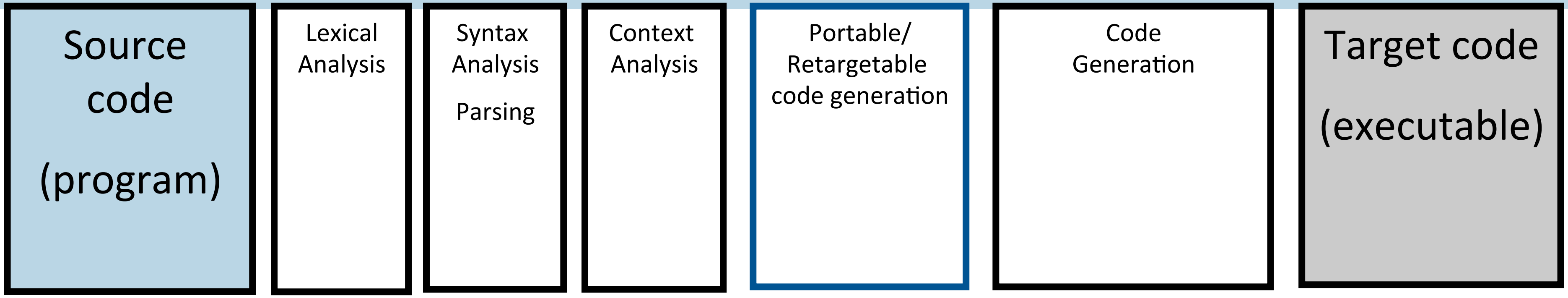
- The process of assigning variables to registers and managing data transfer in and out of registers
- Using registers intelligently is a critical step in any compiler
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation: Goals

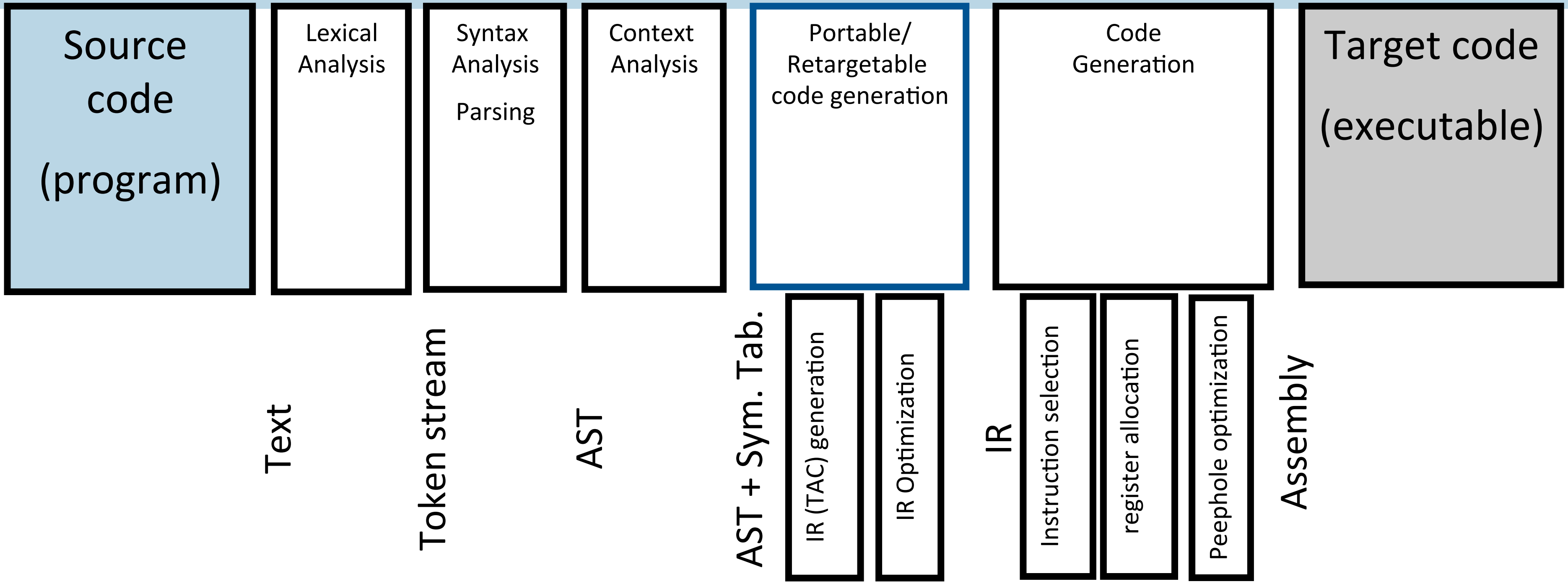


- Reduce number of temporaries (registers)
  - Machine has at most K registers
  - Some registers have special purpose
    - E.g., pass parameters
- Reduce the number of move instructions
  - MOVE R1,R2 // R1 ← R2

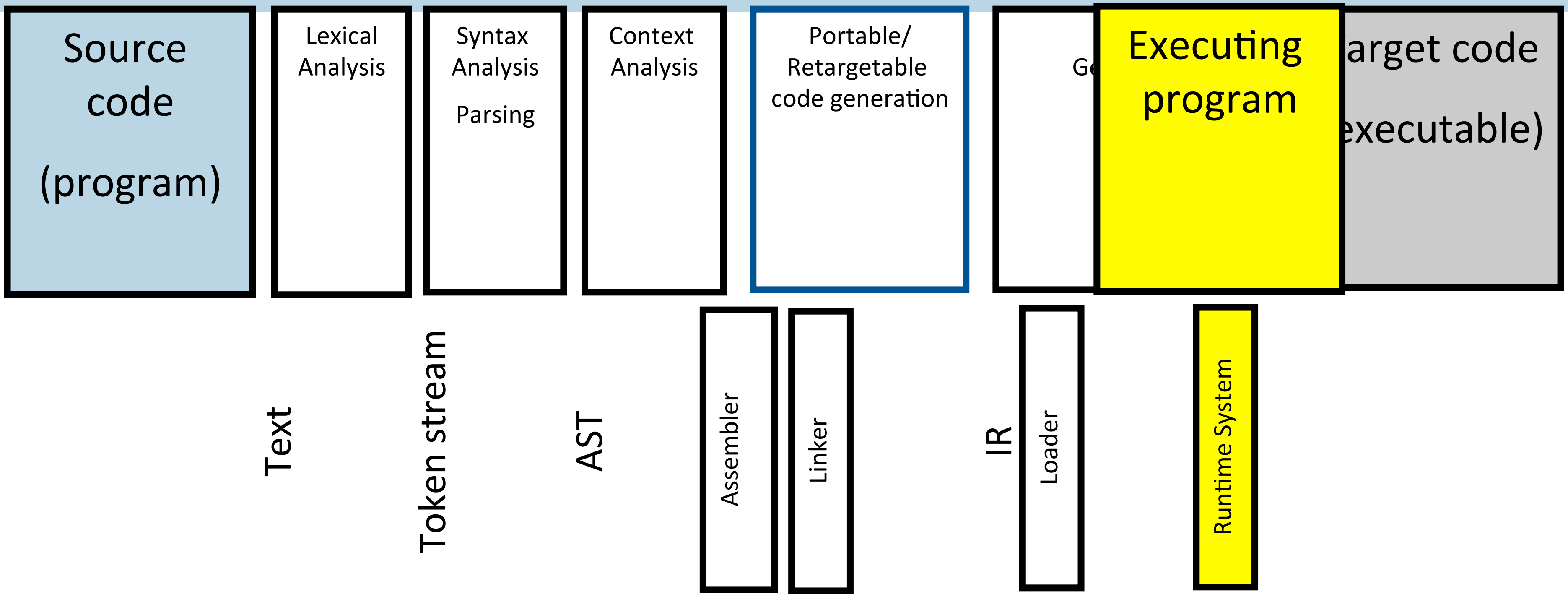
Code generation



Code generation



Runtime System (GC)



Compilation → Execution

Source code (program)	Lexical Analysis	Syntax Analysis Parsing	Context Analysis	Portable/ Retargetable code generation	Code Generation	Linking	Loading	Executing program

Text

Token stream

AST

AST + Sym. Tab.

IR

Object File

Executable File

Image

Runtime System

"Hello World"

Good Luck in the Exam!