Compilation

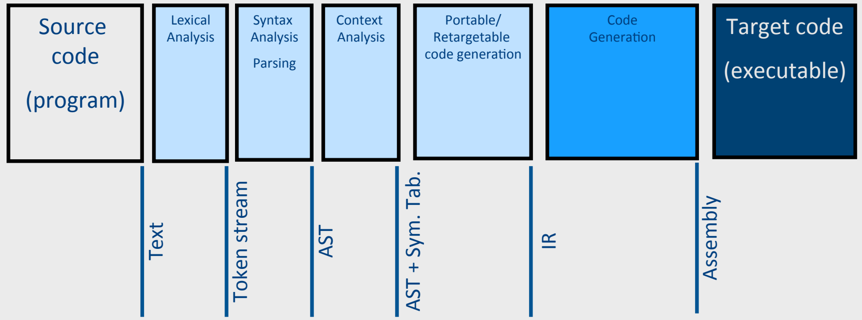
0368-3133 (Semester A, 2013/14)

Lecture 14: Compiling Object Oriented Programs

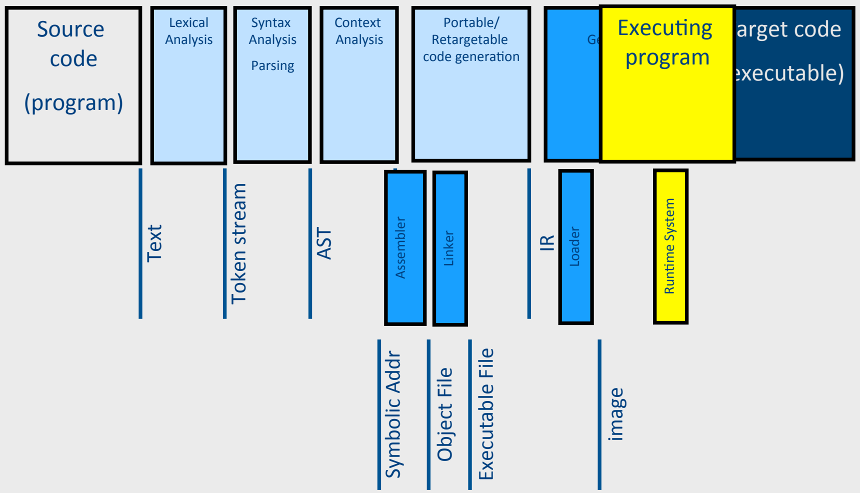
Noam Rinetzky

Slides credit: Mooly Sagiv

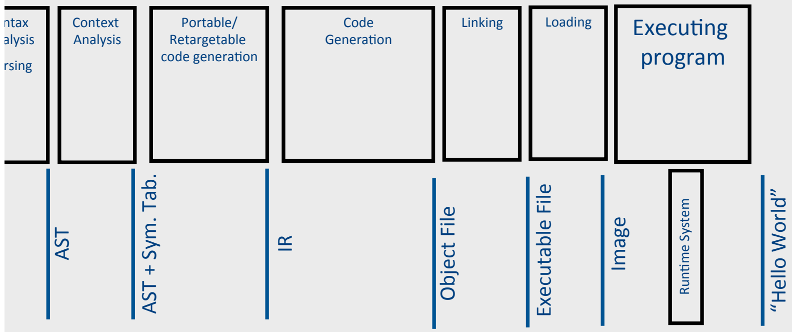
Stages of compilation



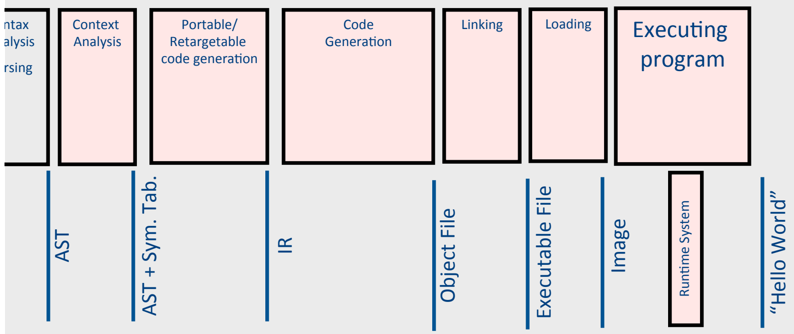
Compilation ➔ Execution



Compilation ➔ Execution



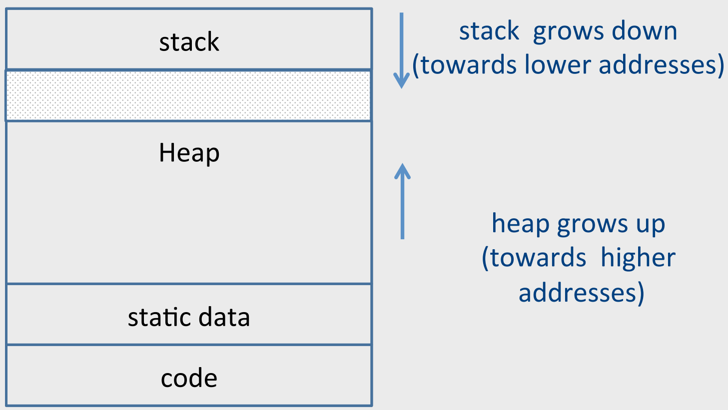
OO: Compilation ➔ Execution



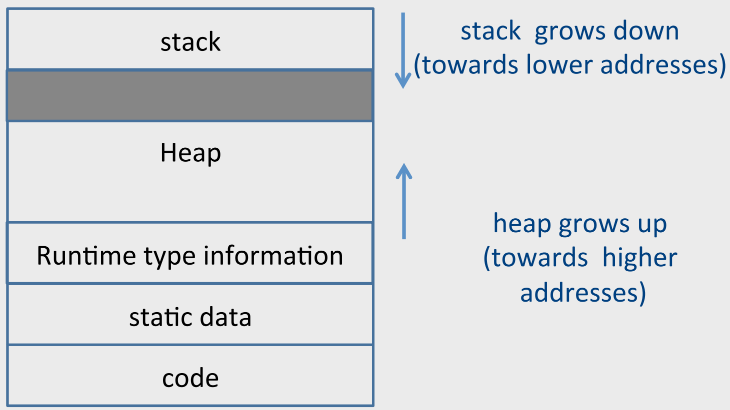
Runtime Environment

- 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)
  - Memory management
- Runtime type information
  - Method invocation
  - Type conversions

Memory Layout



Memory Layout



Object Oriented Programs

- Simula, Smalltalk, Modula 3, C++, Java, C#, Python
- Objects (usually of type called class)
  - Code
  - Data
- Naturally supports Abstract Data Type implementations
- Information hiding
- Evolution & reusability

A Simple Example

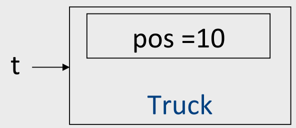
```
class Vehicle extends object {
  int pos = 10;
  void move(int x) {
    position = position + x ;
  }
}

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;
    c.move(60);
    v.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);
  }
}
```

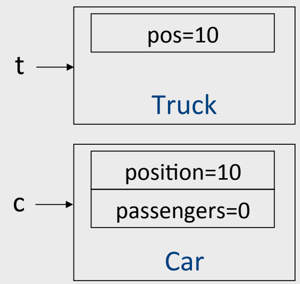
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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);
  }
}
```

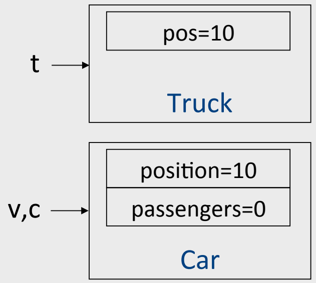
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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);
  }
}
```

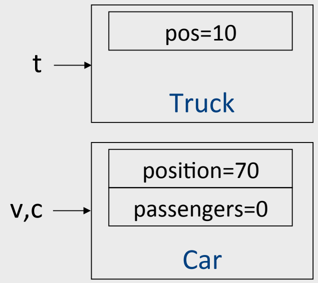
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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);
  }
}
```

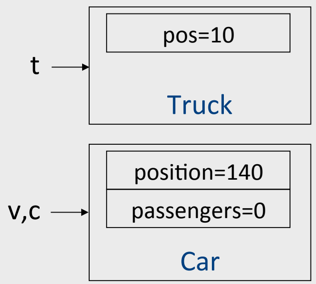
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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);
  }
}
```

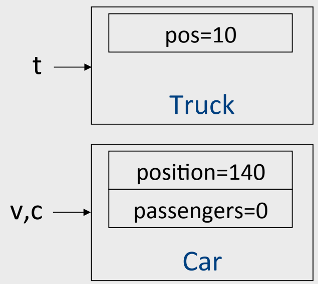
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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);
  }
}
```

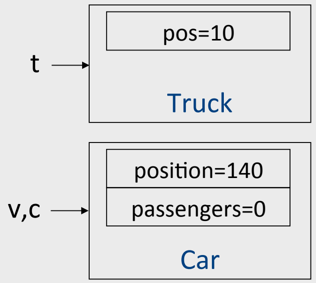
```
class main extends object {
  void main() {
    Truck t = new Truck();
    Car c = new Car();
    Vehicle v = c;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```



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;
    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 (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;
    c.move(60);
    v.move(70);
    c.await(t);
  }
}
```

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

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

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

```
void moveCa() ?
```

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


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 m2_A(classA  *this, int i) {
  // Body of m2 with any object
  // field f as this→f
  …
}
```

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

```
m2A(a,5)   //m2A(&a,5)
```


Features of OO languages

- Inheritance
- Method overriding
- Polymorphism
- Dynamic binding

Handling Single Inheritance

- Simple type extension
- Type checking module checks consistency
- Use prefixing to assign fields in a consistent way

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

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


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

m2 is redefined

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


Method Overriding

- Redefines functionality
- Affects semantic analysis

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

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

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

declared

defined

Method Overriding

```
a.m2(5)   // class(a) = A
m2A_A(a, 5)
```

```
b.m2(5)   // class(b) = B
m2A_B(b, 5)
```

```
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

```
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

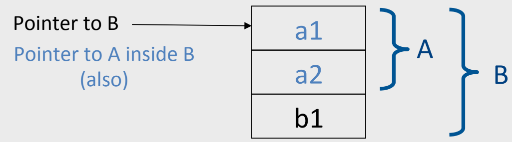
- Declared separately
  - Defined in child classes
  - E.g., Java abstract classes
    - Abstract classes cannot be instantiated
- Handled similarly
- Textbook uses “virtual” for abstract

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 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() {…}
}
```

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

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

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

```
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	Compile-Time Table
a1	m1A_A
a2	m2A_A

Runtime object	Compile-Time Table
a1	m1A_A
a2	m2A_B
b1	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	Compile-Time Table
a1	m1A_A
a2	m2A_A

Runtime object	Compile-Time Table
a1	m1A_A
a2	m2A_B
b1	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

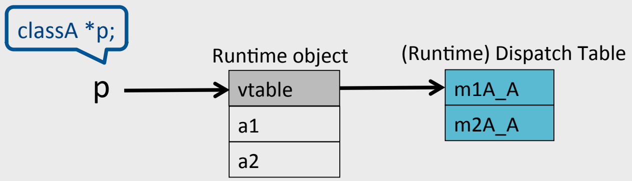
```
typedef struct  {
    field a1;
    field a2;
} A;

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){…}
```

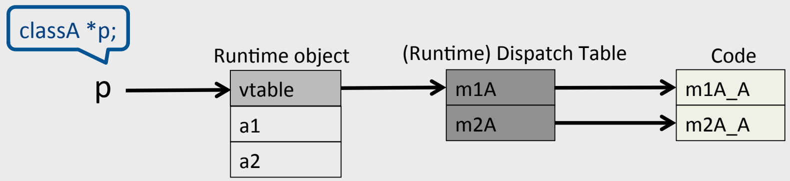
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

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){…}
```

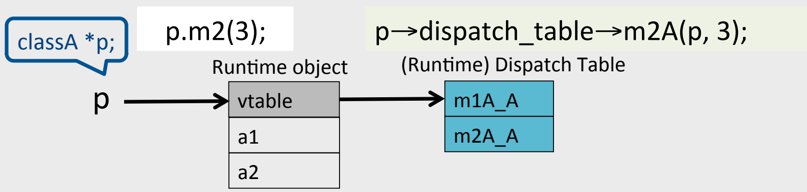
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

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){…}
```

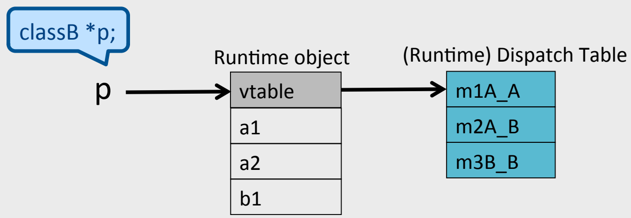
```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

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

void m3B_B(B* this){…}
```



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){…}
```



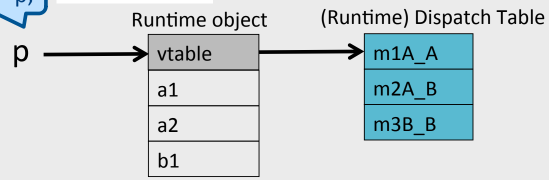
More efficient implementation

```
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){…}
```

classB *p;

p.m2(3);



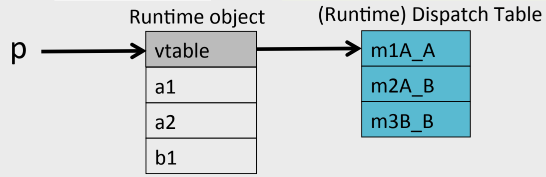
More efficient implementation

```
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

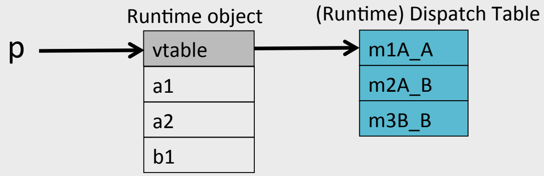
```
typedef struct  {
    field a1;
    field a2;
} A;

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){…}
```

convert_ptr_to_B_to_ptr_to_A(p)

p.m2(3);

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

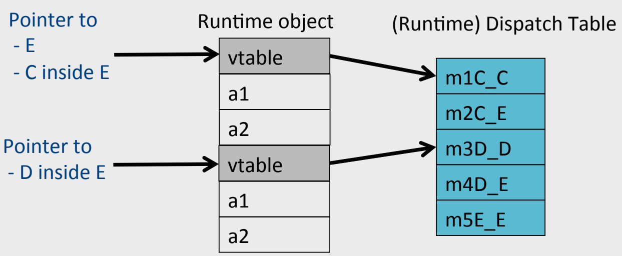
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

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)

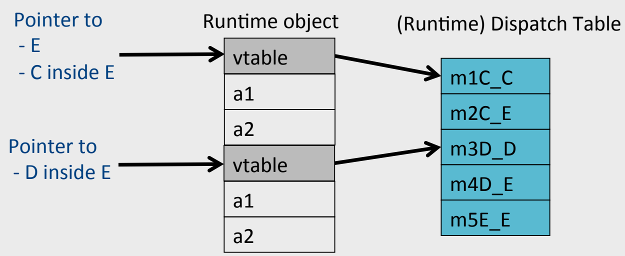
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
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)

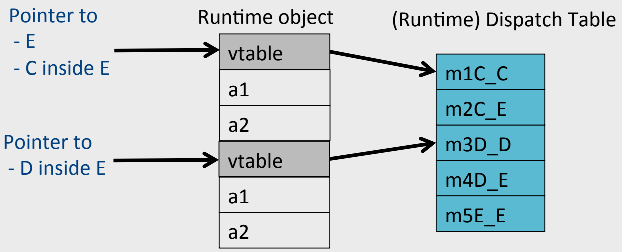
```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
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);



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(){…}
}
```


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

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

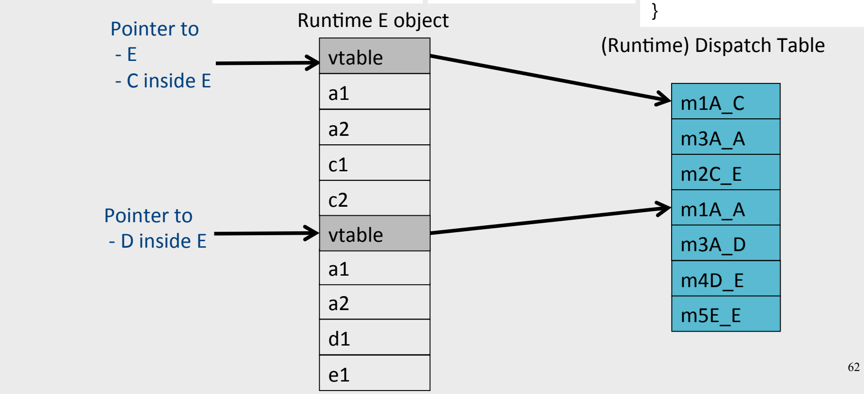
Independent Inheritance

```
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(){…}
}
```



Implementation

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

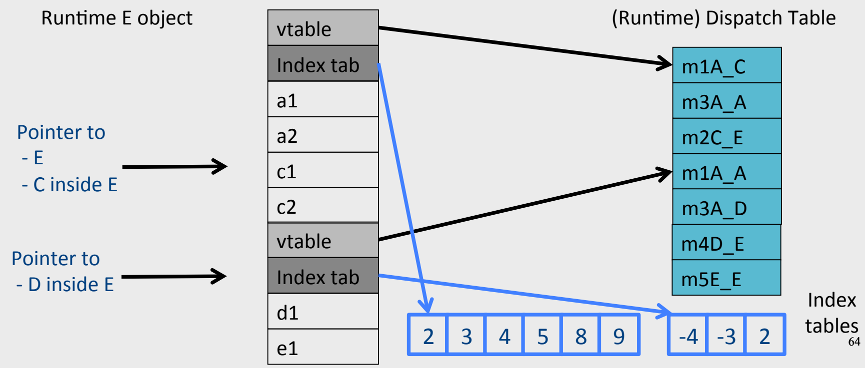
Implementation

```
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

```
public interface Comparable {
    public int compare(Comparable o);
}
```

- 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

Interface Types

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 (Semester A, 2013/14)

Noam Rinetzky

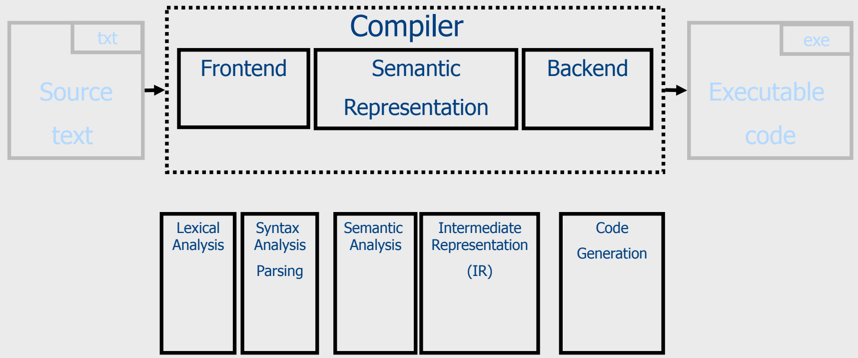
What is a compiler?

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

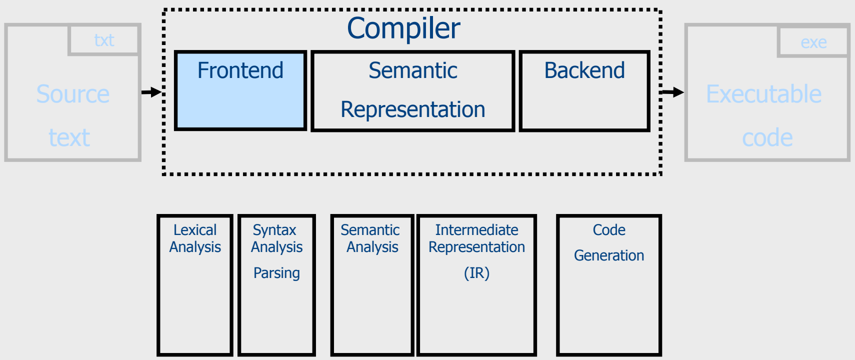
The most common reason for wanting to transform source code is to create an executable program.”

--Wikipedia

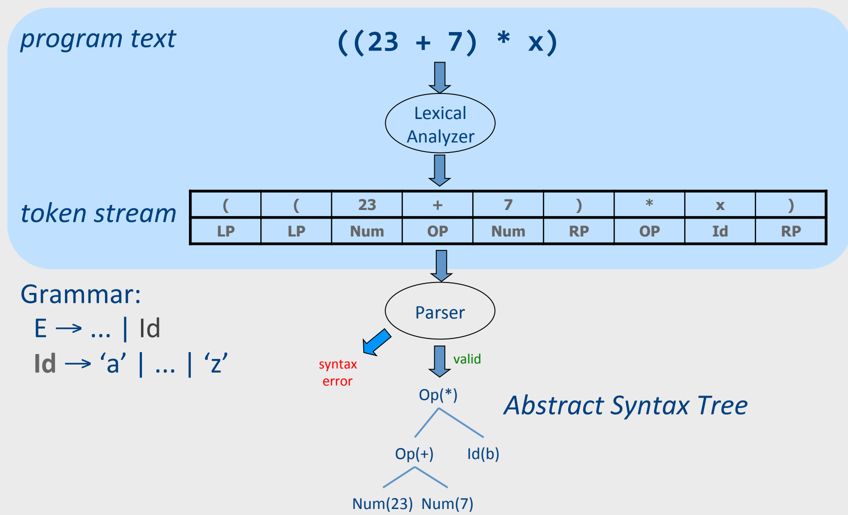
Conceptual Structure of a Compiler



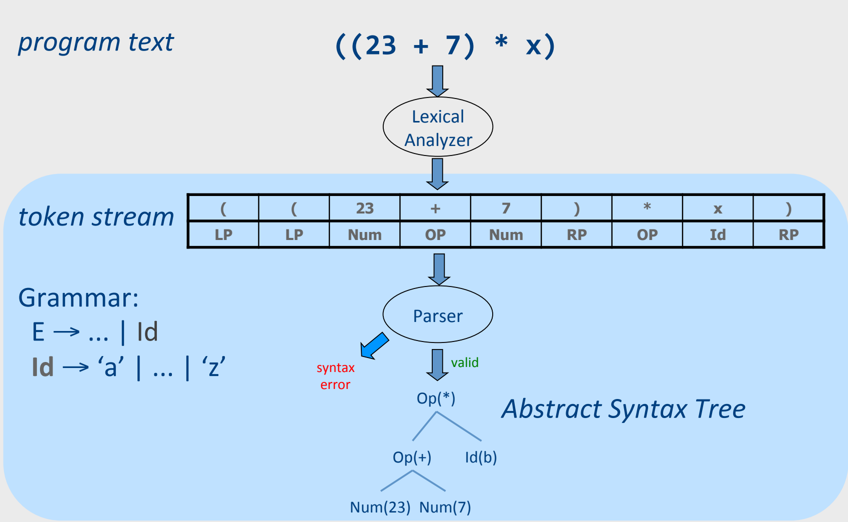
Conceptual Structure of a Compiler



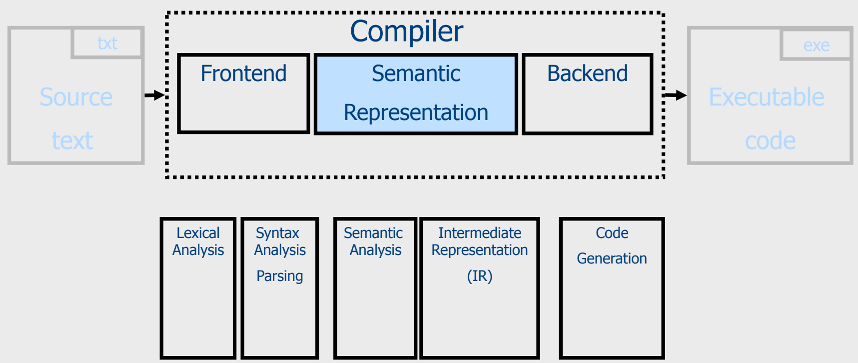
From scanning to parsing



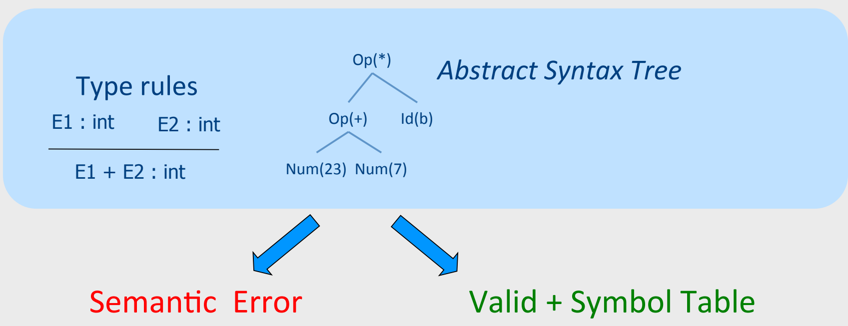
From scanning to parsing



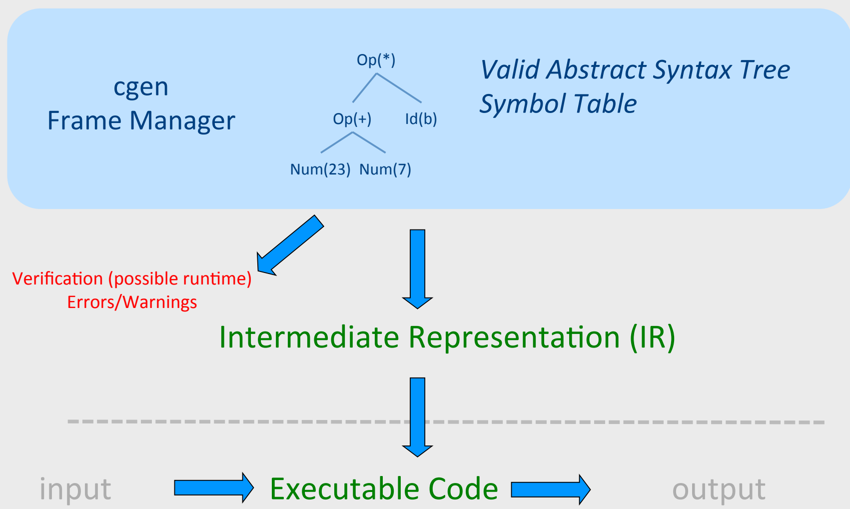
Conceptual Structure of a Compiler



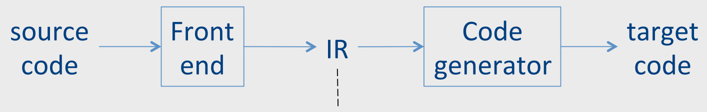
Context Analysis



Code Generation



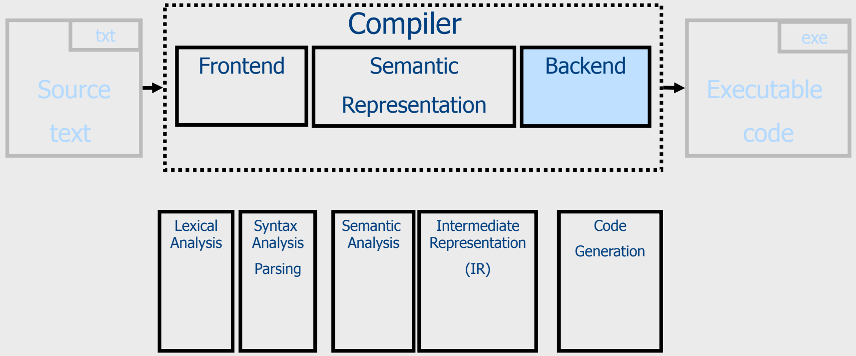
Optimization



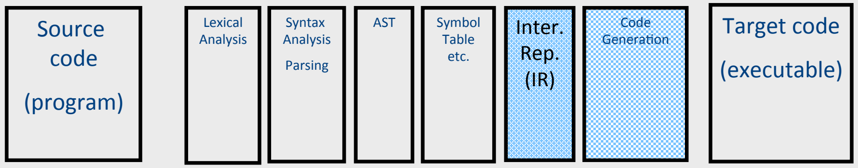
Program Analysis
Abstract interpretation

Can appear in later stages too

Conceptual Structure of a Compiler

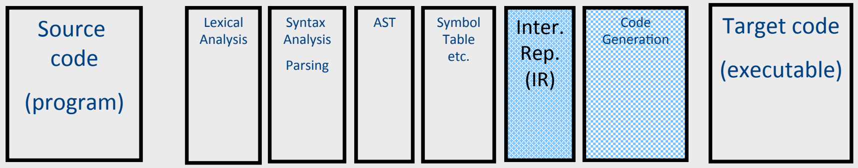


Register Allocation



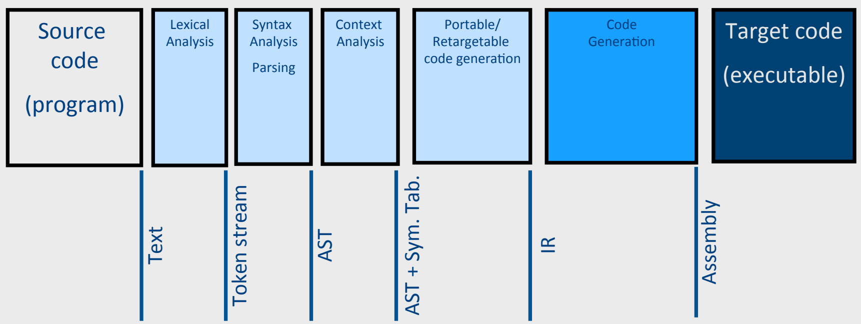
- The process of assigning variables to registers and managing data transfer in and out of registers
- Using registers intelligently is a critical step in any compiler
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation: Goals

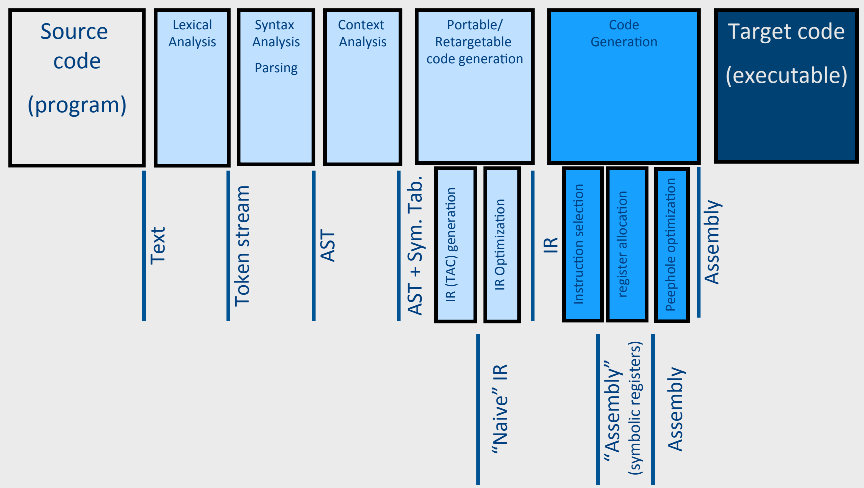


- Reduce number of temporaries (registers)
  - Machine has at most K registers
  - Some registers have special purpose
    - E.g., pass parameters
- Reduce the number of move instructions
  - `MOVE R1,R2` // R1 ← R2

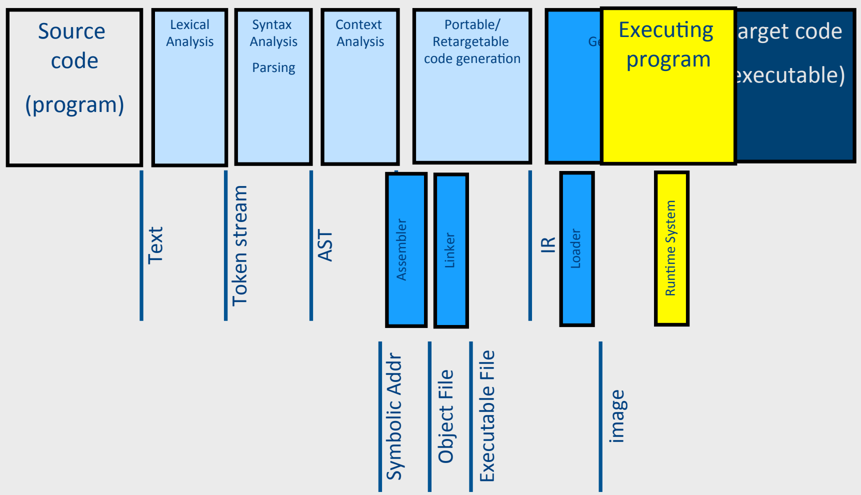
Code generation



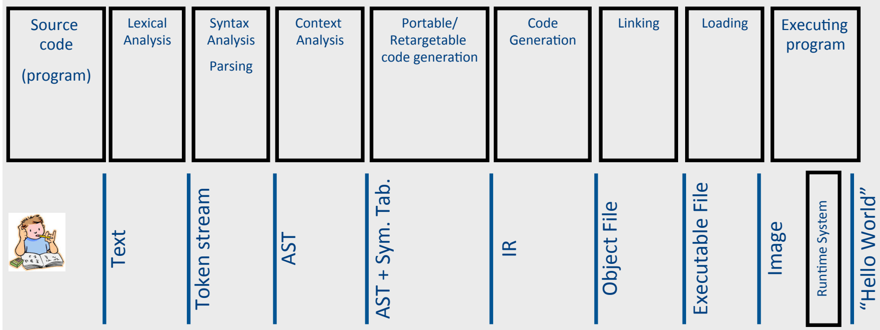
Code generation



Runtime System (GC)



Compilation ➔ Execution



The End

- And advanced course next semester
- And workshop on detecting malicious JavaScripts

The End

- And advanced course next semester
- And workshop on detecting malicious JavaScripts (using static analysis)
- And thanks you & good luck!