Scoping and Types

Martin Kellogg

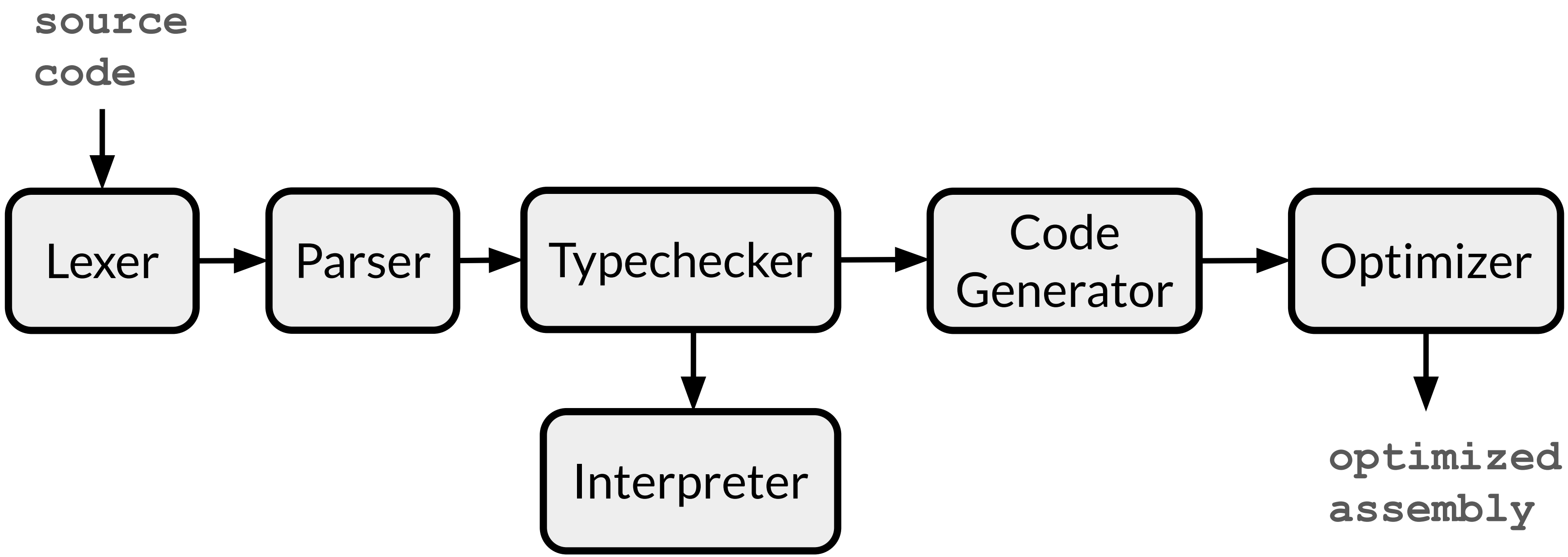
Today's Agenda

- Overview of the role of semantic analysis in a compiler
- Scoping and symbol tables
- Introduction to types

Today’s Agenda

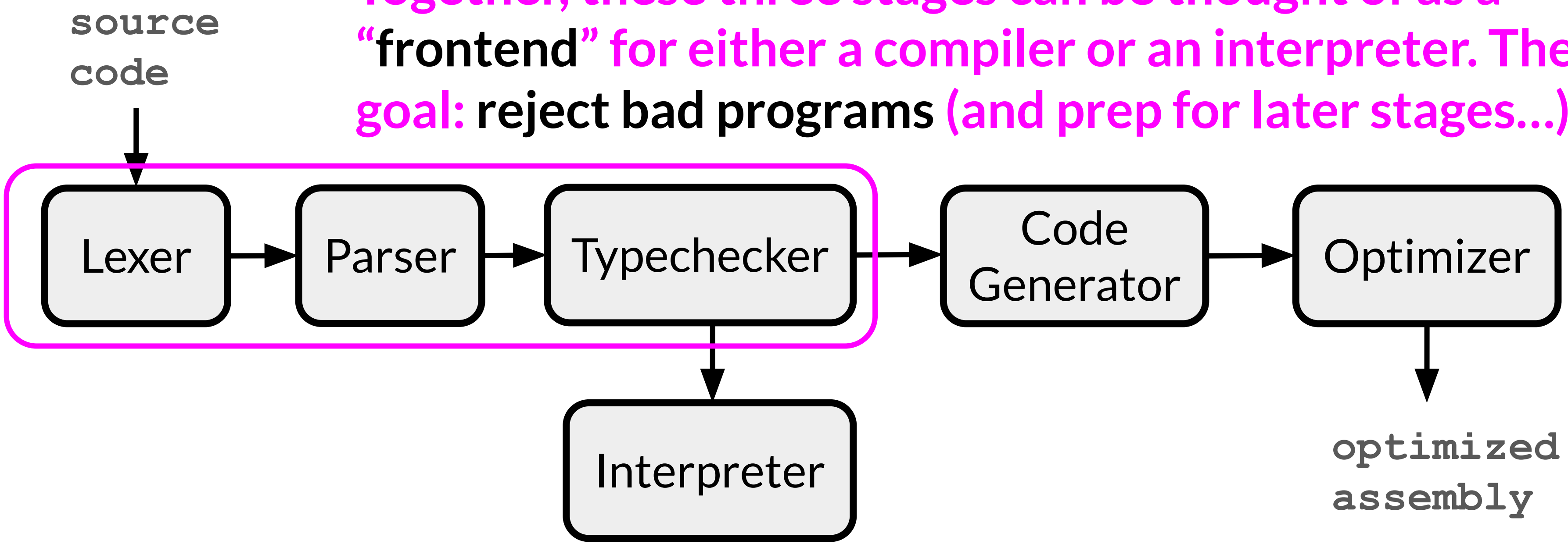
- **Overview of the role of semantic analysis in a compiler**
- Scoping and symbol tables
- Introduction to types

Traditional compiler/interpreter structure



Traditional compiler/interpreter structure

Together, these three stages can be thought of as a "**frontend**" for either a compiler or an interpreter. Their goal: **reject bad programs** (and prep for later stages...).



The Role of Semantic Analysis

Three “Compiler Frontend” stages that reject bad programs:

The Role of Semantic Analysis

Three “Compiler Frontend” stages that reject bad programs:

- Lexical analysis
  - Detects inputs with illegal tokens

The Role of Semantic Analysis

Three “Compiler Frontend” stages that reject bad programs:

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees

The Role of Semantic Analysis

Three “Compiler Frontend” stages that reject bad programs:

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last “frontend” phase
  - Catches more errors! But what kinds of errors…

Why a Separate Semantic Analysis?

- Lexing and parsing cannot catch some errors

Why a Separate Semantic Analysis?

- Lexing and parsing cannot catch some errors
  - Why? Some language constructs are **not context-free**!

Why a Separate Semantic Analysis?

- Lexing and parsing cannot catch some errors
  - Why? Some language constructs are **not context-free**!
- Examples:

Why a Separate Semantic Analysis?

- Lexing and parsing cannot catch some errors
  - Why? Some language constructs are **not context-free**!
- Examples:
  - All used variables must have been declared (i.e. ***scoping***)
  - A method must be invoked with arguments of proper type (i.e. ***typing***)
  - A class must not be defined more than once
  - etc.

What Does Semantic Analysis Do?

What Does Semantic Analysis Do?

Many checks! For example, `cool` checks:

1. All identifiers are declared
2. Static Types
3. Inheritance relationships (no cycles, etc.)
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers are not misused
7. And others (check the CRM)!

What Does Semantic Analysis Do?

Many checks! For example, `cool` checks:

1. All identifiers are declared
2. Static Types
3. Inheritance relationships (no cycles, etc.)
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers are not misused
7. And others (check the CRM)!

These requirements are **language-dependent**! For example, which of the above are checked by Python?

What Does Semantic Analysis Do?

Many checks! For example, `cool` checks:

1. All identifiers are declared
2. Static Types
3. Inheritance relationships (no cycles, etc.)
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers are not misused
7. And others (check the CRM)!

Let's look at one example: ***scoping***

These requirements are **language-dependent**! For example, which of the above are checked by Python?

Scoping

Definition: The ***scope*** of an identifier is the portion of a program in which that identifier is accessible

Scoping

Definition: The ***scope*** of an identifier is the portion of a program in which that identifier is accessible

- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don't overlap

Scoping

Definition: The ***scope*** of an identifier is the portion of a program in which that identifier is accessible

- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don’t overlap
- ***Scoping rules*** match identifier uses with identifier declarations

Scoping

Definition: The ***scope*** of an identifier is the portion of a program in which that identifier is accessible

- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don't overlap
- ***Scoping rules*** match identifier uses with identifier declarations
- Checking scoping rules is an important semantic analysis step in most languages
  - including Cool, Java, and C++ (and even Python has `global`)

Static vs. Dynamic Scope

- Most languages have ***static scope***
    - Scope depends only on the program text, not run-time behavior

Static vs. Dynamic Scope

- Most languages have ***static scope***
  - Scope depends only on the program text, not run-time behavior
  - Cool, Java, C++, C#, etc., have static scope

Static vs. Dynamic Scope

- Most languages have ***static scope***
  - Scope depends only on the program text, not run-time behavior
  - Cool, Java, C++, C#, etc., have static scope
- Ancient history: ***dynamically scoped*** languages
  - Scope depends on execution of the program

Static vs. Dynamic Scope

- Most languages have ***static scope***
    - Scope depends only on the program text, not run-time behavior
    - Cool, Java, C++, C#, etc., have static scope
- Ancient history: ***dynamically scoped*** languages
    - Scope depends on execution of the program
    - e.g., Lisp, SNOBOL, Tex, Perl, PostScript
        - though modern Lisp has changed to mostly static scoping

Static Scoping Example

```
let x: Int <- 0 in
  {
    x;
    { let x: Int <- 1 in
          x; } ;
    x;
  }
```

Static Scoping Example

```
let x: Int <- 0 in
  {
    x;
    { let x: Int <- 1 in
          x; } ;
    x;
  }
```

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
  {
    x;
    { let x: Int <- 1 in
          x; } ;
    x;
  }
```

which definition of x is used?

- Recall static scoping = uses of x refer to the closest enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
     x;
      { let x: Int <- 1 in
              x; } ;
     x;
   }
```

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
     x;
     { let x: Int <- 1 in
            x; } ;
     x;
   }
```

which definition of x is used?

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
     x;
      { let x: Int <- 1 in
               x; } ;
     x;
   }
```

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
     x;
      { let x: Int <- 1 in
              x; } ;
     x;
   }
```

which definition of x is used?

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
    x;
     { let x: Int <- 1 in
               x; } ;
    x;
   }
```

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Static Scoping Example

```
let x: Int <- 0 in
   {
     x;
      { let x: Int <- 1 in
               x; } ;
     x;
   }
```

Redefining a variable like x in this example is sometimes called "*shadowing* x"

- Recall static scoping = uses of x refer to the **closest** enclosing definition

Scope in Cool

- Cool identifier bindings are **introduced** by

Scope in Cool

- Cool identifier bindings are **introduced** by
  - Class declarations (introduce class names)
  - Method definitions (introduce method names)
  - Let expressions (introduce object ids)
  - Formal parameters (introduce object ids)
  - Attribute definitions in a class (introduce object ids)
  - Case expressions (introduce object ids)

Implementing the Most-Closely Nested Rule

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
  - Process an AST node *n*

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
    - Process an AST node *n*
    - Process the children of *n*

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*
- Example: the scope of let bindings is **one subtree**

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*
- Example: the scope of let bindings is **one subtree**
  - consider:

```
let x: Int <- 0 in e
```

Implementing the Most-Closely Nested Rule

- Many (but not all) semantic analyses can be expressed as **recursive descent** over the AST, including static scoping
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*
- Example: the scope of let bindings is **one subtree**
  - consider:

```
let x: Int <- 0 in e
```

  - x can be used in exactly the AST subtree corresponding to **`e`**

Symbol Tables

- Consider again: **`let x: Int <- 0 in e`**

Symbol Tables

- Consider again: `let x: Int <- 0 in e`
- Idea:

Symbol Tables

- Consider again: `let x: Int <- 0 in e`
- Idea:
  - **before** processing `e`, **add** definition of `x` to the current definitions, overriding any other definition of `x`

Symbol Tables

- Consider again: `let x: Int <- 0 in e`
- Idea:
  - before processing `e`, add definition of `x` to the current definitions, overriding any other definition of `x`
  - after processing `e`, remove the definition of `x` and restore the old definition of `x`

Symbol Tables

- Consider again: `let x: Int <- 0 in e`
- Idea:
  - **before** processing `e`, add definition of `x` to the current definitions, overriding any other definition of `x`
  - **after** processing `e`, remove the definition of `x` and **restore** the old definition of `x`
- A *symbol table* is a data structure that tracks the current **bindings** of identifiers in this manner

Symbol Tables

- Consider again: `let x: Int <- 0 in e`
- Idea:
  - **before** processing **`e`**, add definition of **`x`** to the current definitions, overriding any other definition of **`x`**
  - **after** processing **`e`**, remove the definition of **`x`** and **restore** the old definition of **`x`**
- A *symbol table* is a data structure that tracks the current bindings of identifiers in this manner
  - You'll need to make one for PA2
  - OCaml's **`Hashtbl`** is specifically designed to be a symbol table

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program
- In other words, a class name can be **used before it is defined**

Scope in Cool (continued)

- Not all kinds of identifiers follow t
- For example, class definitions in Co
  - Cannot be nested
  - Are **globally visible** throughou
- In other words, a class name can be **used before it is defined**

Cool UBD example (classes):

```
class Foo {
   ... let y : Test in ...
};

class Test {
   ...
};
```

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program
- In other words, a class name can be **used before it is defined**
- Attribute names are **global** *within* the class where they are defined

Scope in Cool (continued)

- Not all kinds of identifiers follow t
- For example, class definitions in Co
  - Cannot be nested
  - Are globally visible throughout the program
- In other words, a class name can be **used before it is defined**
- Attribute names are **global** *within* the class where they are defined

Cool UBD example (attributes):

```
class Foo {
   f(): Int { tm };
   tm: Int <- 0;
}
```

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program
- In other words, a class name can be **used before it is defined**
- Attribute names are **global** *within* the class where they are defined
- Methods and attribute names have complex rules

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program
- In other words, a class name can be **used before it is defined**
- Attribute names are **global** *within* the class where they are defined
- Methods and attribute names have complex rules
  - E.g., a method can be defined in a **parent class** rather than in the class wherein it is used! (**inheritance**)

Scope in Cool (continued)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool:
  - Cannot be nested
  - Are **globally visible** throughout the program
- In other words, a class name can be **used before it is defined**
- Attribute names are **global** *within* the class where they are defined
- Methods and attribute names have complex rules
  - E.g., a method can be defined in a **parent class** rather than in the class wherein it is used! (**inheritance**)
  - Methods may also be redefined (overridden)

Class Definitions

- We know that class names can be used before being defined

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?
  - We **cannot**: the symbol table relies on the locality of the scoping rules

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?
  - We **cannot**: the symbol table relies on the locality of the scoping rules
- Solution:

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?
  - We **cannot**: the symbol table relies on the locality of the scoping rules
- Solution:
  - Pass 1: collect all class names
  - Pass 2: do the checking

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?
  - We **cannot**: the symbol table relies on the locality of the scoping rules
- Solution:
  - Pass 1: collect all class names
  - Pass 2: do the checking
- In other words, semantic analysis often requires **multiple passes**
  - commonly more than two!

Class Definitions

- We know that class names can be used before being defined
- Can we check this property with a **symbol table**?
  - Why or why not?
  - We **cannot**: the symbol table relies on the locality of the scoping rules
- Solution:
  - Pass 1: collect all class names
  - Pass 2: do the checking
- In other words, semantic analysis often requires **multiple passes**
  - commonly more than two!

For PA2, use **as many passes as you'd like** - we aren't evaluating you on efficiency, but on correctness.

Trivia Break:

Today's Agenda

- Overview of the role of semantic analysis in a compiler
- Scoping and symbol tables
- **Introduction to types**

What is a type system, anyway?

What is a type system, anyway?

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

What is a type system, anyway?

A type can also encode the set of ***valid operations*** on values of that type

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

What is a type system, anyway?

A type can also encode the set of ***valid operations*** on values of that type

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values

What is a type system, anyway?

A type can also encode the set of ***valid operations*** on values of that type

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems

What is a type system, anyway?

A type can also encode the set of ***valid operations*** on values of that type

Definition: a ***type system*** is a set of rules that give every program element a ***type***, which is an upper bound on the set of possible values that that element can take on at run time

- goal of a type system: **prevent errors** at run time due to unexpected values
- **type theory** is the discipline of math (yes!) that studies the formal properties of type systems
- most programming languages include some kind of type system
  - exceptions: assembly, Lisp, a few others

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?
  - e.g., are they integers in the program?

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?
    - e.g., are they integers in the program? Are they pointers and offsets?

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?
  - e.g., are they integers in the program? Are they pointers and offsets? If so, to data or to other code?

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?
    - e.g., are they integers in the program? Are they pointers and offsets? If so, to data or to other code?
- Regardless of their **logical** types, all of these have the **same assembly language implementation**!

Why Do We Need Type Systems?

- Consider the following fragment of assembly:

```
add r1 <- r2 + r3
```

- What are the types of **`r1`**, **`r2`**, and **`r3`**?
    - e.g., are they integers in the program? Are they pointers and offsets? If so, to data or to other code?
- Regardless of their **logical** types, all of these have the **same assembly language implementation**!
    - one goal of typechecking: prevent mixing these up

Primary Goal of Type Systems

- A language's type system specifics **which operations** are valid for **which types**

Primary Goal of Type Systems

- A language's type system specifics **which operations** are valid for **which types**
- The primary goal of typechecking is to **ensure that operations are only used on the correct types**

Primary Goal of Type Systems

- A language's type system specifics **which operations** are valid for **which types**
- The primary goal of typechecking is to **ensure that operations are only used on the correct types**
  - This enforces the intended interpretation of values, which fundamentally **all look the same** to the machine
    - i.e., everything is a bit string at the machine code level…

Primary Goal of Type Systems

- A language's type system specifics **which operations** are valid for **which types**
- The primary goal of typechecking is to **ensure that operations are only used on the correct types**
  - This enforces the intended interpretation of values, which fundamentally **all look the same** to the machine
    - i.e., everything is a bit string at the machine code level...
- A type system provides a concise **formalization** for a set of semantic checking rules

What Kinds of Errors Can A Type System Detect?

What Kinds of Errors Can A Type System Detect?

- Memory errors:
  - Reading from an invalid pointer, etc.

What Kinds of Errors Can A Type System Detect?

- Memory errors:
  - Reading from an invalid pointer, etc.
- Invalid operations
  - e.g., calling “meow()” on a Dog

What Kinds of Errors Can A Type System Detect?

- Memory errors:
  - Reading from an invalid pointer, etc.
- Invalid operations
  - e.g., calling “meow()” on a Dog
- Violations of abstraction boundaries
  - e.g., ->

```
class FileSystem {
 open (x: String) : File {
   ...
 }
 ...
}
```

```
class Client {
 f(fs : FileSystem) {
  File fd <- fs.open("foo")
   ...
 }  -- f cannot see inside fd!
}
```

What Kinds of Errors Can A Type System Detect?

- Memory errors:
  - Reading from an invalid pointer, etc.
- Invalid operations
  - e.g., calling “meow()” on a Dog
- Violations of **abstraction** boundaries
  - e.g., ->
- …and arbitrarily-complex other properties (wait for ***pluggable types*** lecture later)

```
class FileSystem {
 open (x: String) : File {
   ...
 }
  ...
}
```

```
class Client {
 f(fs : FileSystem) {
  File fd <- fs.open(“foo”)
   ...
 }  -- f cannot see inside fd!
}
```

Kinds of type systems

- Static vs dynamic checking

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime

Kinds of type systems

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime
    - shares advantages/disadvantages with other dynamic analyses

Kinds of type systems

> Dynamic typing is sometimes called ***duck typing***
> - "if it walks like a duck and quacks like a duck, you can treat it as a duck"

- Static vs dynamic checking
  - ***statically typed*** languages have their types checked before the program runs, typically **at compile time**
    - shares advantages/disadvantages with other static analyses
  - ***dynamically typed*** languages have their types checked **at run time**, typically by a special interpreter or language runtime
    - shares advantages/disadvantages with other dynamic analyses

Static vs dynamic types

- Both are **common in practice**

Static vs dynamic types

- Both are **common in practice**
  - examples of each?

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.

Static vs dynamic types

- Both are **common in practice**
    - examples of each?
        - Static: Java, C, Rust, OCaml, TypeScript, etc.
        - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits
  - Benefits of static typing:
    - ???
  - Benefits of dynamic typing:
    - ???

Static vs dynamic types

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits
  - Benefits of static typing:
    - early detection of errors, types are documentation
  - Benefits of dynamic typing:
    - faster prototyping, no false positives

Static vs dynamic types

Most "production" code written in a statically-typed language with **escape hatches**

- e.g., unsafe casts in C, native methods in Java

- Both are **common in practice**
  - examples of each?
    - Static: Java, C, Rust, OCaml, TypeScript, etc.
    - Dynamic: Python, Ruby, JavaScript, etc.
- **Ongoing debate** about the benefits
  - Benefits of static typing:
    - early detection of errors, types are documentation
  - Benefits of dynamic typing:
    - faster prototyping, no false positives

Other ways type systems differ

Other ways type systems differ

- **Implicit** vs **explicit**

Other ways type systems differ

- **Implicit** vs **explicit**
  - “do you write the types yourself”
  - almost all mainstream, static languages are explicit

Other ways type systems differ

- **Implicit** vs **explicit**
  - "do you write the types yourself"
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties

Other ways type systems differ

- **Implicit** vs **explicit**
  - "do you write the types yourself"
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties
  - e.g., Kotlin **guarantees no null-pointer dereferences**, but Java doesn't (both compile to Java bytecode)

Other ways type systems differ

- **Implicit** vs **explicit**
  - "do you write the types yourself"
  - almost all mainstream, static languages are explicit
- **Strength** of the type system
  - not all type systems can prove the same properties
  - e.g., Kotlin **guarantees no null-pointer dereferences**, but Java doesn't (both compile to Java bytecode)
  - stronger types can be added to a language (**ask me more**)
    - this is "pluggable types" from a few slides ago…

Cool Types

Cool Types

- Only two kinds:
  - Class names
  - **SELF_TYPE**

Cool Types

- Only two kinds:
  - Class names
  - **SELF_TYPE**
- There are **no unboxed base types** (like e.g., `int` in Java)

Cool Types

- Only two kinds:
  - Class names
  - **SELF_TYPE**
- There are **no unboxed base types** (like e.g., `int` in Java)
- The user must ***declare*** a type for all identifiers
  - “declare” here is just a fancy way to say “write down by hand”

Cool Types

- Only two kinds:
  - Class names
  - **SELF_TYPE**
- There are **no unboxed base types** (like e.g., `int` in Java)
- The user must ***declare*** a type for all identifiers
  - "declare" here is just a fancy way to say "write down by hand"
- The compiler then **infers** types for expressions
  - for *every* expression!
  - Java, C, C++, etc., do this too

Aside: Typechecking vs. Type Inference

Aside: Typechecking vs. Type Inference

Definition: ***Typechecking*** is the process of *verifying* that the types in a fully-annotated program are consistent.

Aside: Typechecking vs. Type Inference

Definition: ***Typechecking*** is the process of *verifying* that the types in a fully-annotated program are consistent.

Definition: ***Type Inference*** is the process of *selecting* consistent types for a program, which typically is not fully annotated.

Aside: Typechecking vs. Type Inference

Definition: ***Typechecking*** is the process of *verifying* that the types in a fully-annotated program are consistent.

Definition: ***Type Inference*** is the process of *selecting* consistent types for a program, which typically is not fully annotated.

- These two concepts are closely related, but subtly different

Aside: Typechecking vs. Type Inference

Definition: ***Typechecking*** is the process of *verifying* that the types in a fully-annotated program are consistent.

Definition: ***Type Inference*** is the process of *selecting* consistent types for a program, which typically is not fully annotated.

- These two concepts are closely related, but subtly different
    - Which do you think is harder?

Rules of Inference

- Lexers and parsers have **formal notations** that specify how they work

Rules of Inference

- Lexers and parsers have **formal notations** that specify how they work
  - Regexps/DFAs (lexer), context-free grammars (parser)

Rules of Inference

- Lexers and parsers have **formal notations** that specify how they work
  - Regexps/DFAs (lexer), context-free grammars (parser)
- The appropriate formalism for typechecking is ***logical rules of inference***

Rules of Inference

- Lexers and parsers have **formal notations** that specify how they work
  - Regexps/DFAs (lexer), context-free grammars (parser)
- The appropriate formalism for typechecking is ***logical rules of inference***
- Why? A rule of inference has the form:
  - *"if Hypothesis is true, then Conclusion is true"*

Rules of Inference

- Lexers and parsers have **formal notations** that specify how they work
  - Regexps/DFAs (lexer), context-free grammars (parser)
- The appropriate formalism for typechecking is ***logical rules of inference***
- Why? A rule of inference has the form:
  - *"if Hypothesis is true, then Conclusion is true"*
- Typechecking computes via **similar reasoning**:
  - *"If E_1 and E_2 have certain types, then E_3 has a certain type"*

Rules of Inference

- Lexers and parsers have fo
  work
  - Regexps/DFAs (lexer),
- The appropriate formalism for typechecking is ***logical rules of inference***
- Why? A rule of inference has the form:
  - *"if Hypothesis is true, then Conclusion is true"*
- Typechecking computes via **similar reasoning**:
  - *"If E_1 and E_2 have certain types, then E_3 has a certain type"*

You can think of rules of inference as a compact notation for **If-Then** statements/conditionals

English to Inference Rules

- We'll start with a **simplified system** and gradually add features
  - I promise the notation is easy to read (with practice)

English to Inference Rules

- We'll start with a **simplified system** and gradually add features
  - I promise the notation is easy to read (with practice)
- Building blocks:
  - $\land$ is "and"

English to Inference Rules

- We'll start with a **simplified system** and gradually add features
  - I promise the notation is easy to read (with practice)
- Building blocks:
  - $\wedge$ is "and"
  - **->** is "if-then"

English to Inference Rules

- We'll start with a **simplified system** and gradually add features
  - I promise the notation is easy to read (with practice)
- Building blocks:
  - $\wedge$ is "and"
  - **->** is "if-then"
  - **x : T** is "x has type T"

English to Inference Rules

Building blocks:

- ∧ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

English to Inference Rules

If e_1 has type **Int** and e_2 has type **Int**,
then $e_1 + e_2$ has type **Int**

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

English to Inference Rules

Building blocks:

- ∧ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

If $\mathbf{e_1}$ has type **Int** and $\mathbf{e_2}$ has type **Int**, then $\mathbf{e_1 + e_2}$ has type **Int**

↓

($\mathbf{e_1}$ has type **Int** ∧ $\mathbf{e_2}$ has type **Int**) -> $\mathbf{e_1 + e_2}$ has type **Int**

English to Inference Rules

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

If $\mathbf{e_1}$ has type **Int** and $\mathbf{e_2}$ has type **Int**, then $\mathbf{e_1 + e_2}$ has type **Int**

↓

($\mathbf{e_1}$ has type **Int** $\wedge$ $\mathbf{e_2}$ has type **Int**) -> $\mathbf{e_1 + e_2}$ has type **Int**

↓

$(\mathbf{e_1} : \mathbf{Int} \wedge \mathbf{e_2} : \mathbf{Int}) \rightarrow \mathbf{e_1 + e_2} : \mathbf{Int}$

English to Inference Rules

If e_1 has type **Int** and e_2 has type **Int**, then $e_1 + e_2$ has type **Int**

↓

(e_1 has type **Int** ∧ e_2 has type **Int**) -> $e_1 + e_2$ has type **Int**

↓

(e_1 : **Int** ∧ e_2 : **Int**) -> $e_1 + e_2$: **Int**

Building blocks:

- ∧ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

Traditional notation (same meaning!):

English to Inference Rules

If e_1 has type **Int** and e_2 has type **Int**, then $e_1 + e_2$ has type **Int**

↓

(e_1 has type **Int** ∧ e_2 has type **Int**) -> $e_1 + e_2$ has type **Int**

↓

$(e_1 : \text{Int} \land e_2 : \text{Int}) \rightarrow e_1 + e_2 : \text{Int}$

Building blocks:

- ∧ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

Traditional notation (same meaning!):

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}}$$

English to Inference Rules

If e_1 has type **Int** and e_2 has type **Int**,
then $e_1 + e_2$ has type **Int**

↓

(e_1 has type **Int** $\wedge$ e_2 has type **Int**) ->
$e_1 + e_2$ has type **Int**

↓

$(e_1 : \text{Int} \wedge e_2 : \text{Int}) \rightarrow e_1 + e_2 : \text{Int}$

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

Traditional notation (same meaning!):

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}}$$

Pronounced "we can prove that…"

Inference Rule Examples

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{[Add]}$$

Inference Rule Examples

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{ [Add]}$$

$$\frac{\textit{i is any integer constant}}{\vdash i : \text{Int}} \text{ [Int]}$$

Inference Rule Examples

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \; [\text{Add}]$$

$$\frac{i \text{ is any integer constant}}{\vdash i : \text{Int}} \; [\text{Int}]$$

- These rules give templates describing how to type integers and + expressions

Inference Rule Examples

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}}\ \textbf{[Add]}$$

$$\frac{i \text{ is any integer constant}}{\vdash i : \text{Int}}\ \textbf{[Int]}$$

- These rules give **templates** describing how to type integers and + expressions
- By filling in the templates, we can produce **complete typings** for expressions

Inference Rule Examples

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{[Add]}$$

$$\frac{i \text{ is any integer constant}}{\vdash i : \text{Int}} \text{[Int]}$$

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions
- Note that we can fill the template with *any* expression!

Inference Rule Examples

Valid use of the [Add] rule:

$$\frac{\vdash \textbf{false} : \textbf{Int} \quad \vdash \textbf{true} : \textbf{Int}}{\vdash \textbf{false} + \textbf{true} : \textbf{Int}}$$

d]

$$\frac{\textit{i is any integer constant}}{\vdash \textbf{i} : \textbf{Int}} \textbf{[Int]}$$

describing how to type integers and + expressions

- By filling in the templates, we can produce **complete typings** for expressions
- Note that we can fill the template with ***any*** expression!

Baby's First Type Derivation

$$\frac{}{\vdash 1 + 2 : \text{Int}}$$

on the whiteboard...

Course Announcements

- My OH this week are modified:
  - no OH this afternoon (faculty meeting)
- Don't forget: PA2c1 is due **Friday**
  - this is a testing assignment: you'll just write Cool programs
- PA1 grades will come out "soon"