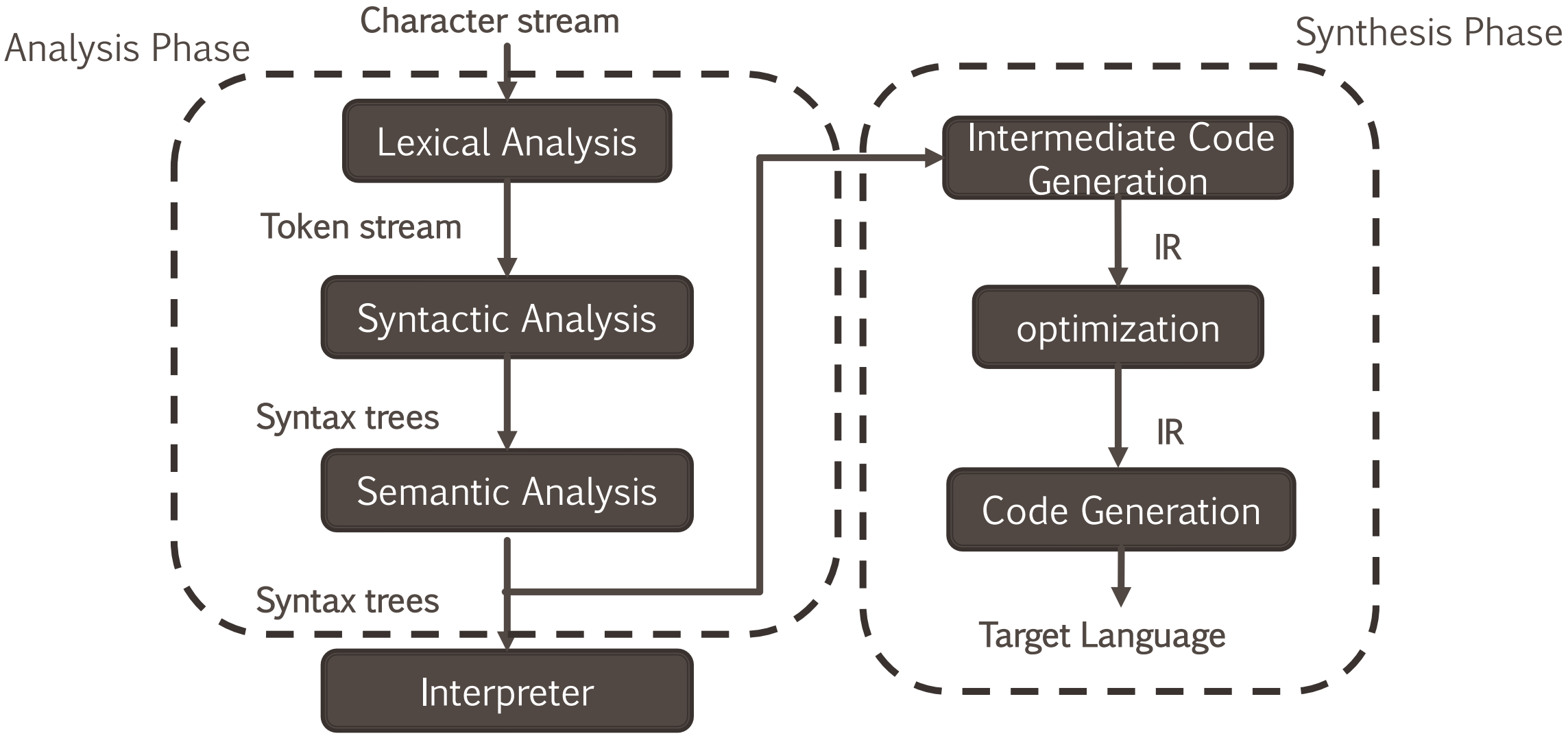
# SEMANTIC ANALYSIS

Baishakhi Ray

These slides are motivated from Prof. Alex Aiken and Prof. Stephen Edward

# Structure of a Typical Compiler



# The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last “front end” phase
  - Catches all remaining errors

# What's Wrong With This?

```
a + f(b, c)
```

# What's Wrong With This?

```
a + f(b, c)
```

Is `a` defined?

Is `f` defined?

Are `b` and `c` defined?

Is `f` a function of two arguments?

Can you add whatever `a` is to whatever `f` returns?

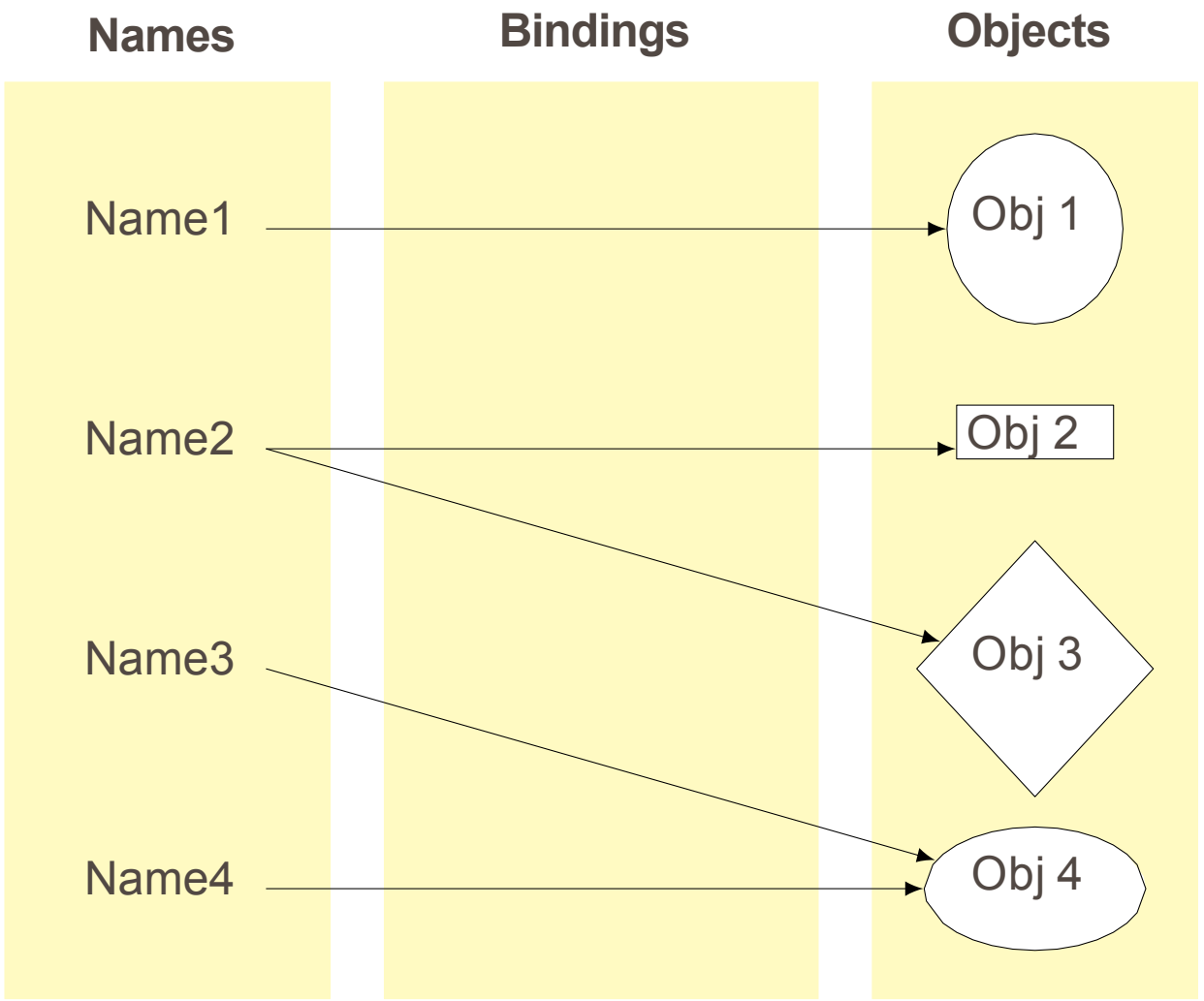
Does `f` accept whatever `b` and `c` are?

Scope questions  Type questions

parsing alone cannot answer these question.

# Scope

- 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.
- An identifier may have restricted scope.



# Static Vs. Dynamic Scoping

- Most modern languages have static scope
  - Scope depends only on the program text, not runtime behavior
  - Most modern languages use static scoping. Easier to understand, harder to break programs.

- A few languages are dynamically scoped
  - Scope depends on execution of the program
  - Lisp, SNOBOL (Lisp has changed to mostly static scoping)
  - Advantage of dynamic scoping: ability to change  environment.
  - A way to surreptitiously pass additional parameters.

# Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, "The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block."

```
void foo()
{
    int x;
}
```

# Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, "If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block."

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

# Dynamic Definitions in TEX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
     \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```

# Open vs. Closed Scopes

- An *open scope* begins life including the symbols in its outer scope.
- Example: blocks in Java

```
{
 int x;
 for (;;){
    /* x visible here */
 }
}
```

- A *closed scope* begins life devoid of symbols. Example: structures in C.

```
struct foo { int x; float y; }
```

# Symbol Tables

- A symbol table is a data structure that tracks the current bindings of identifiers
- Can be implemented as a stack
- Operations
  - add_symbol(x) push x and associated info, such as x's type, on the stack
  - find_symbol(x) search stack, starting from top, for x. Return first x found or NULL if none found
  - remove_symbol() pop the stack when out of scope
- Limitation:
  - What if two identical objects are defined in the same scope multiple times.
  - Eg: foo(int x, int x)

# Advanced Symbol Table

- enter_scope() start a new nested scope
- find_symbol(x) finds current x (or null)
- add_symbol(x) add a symbol x to the table
- check_scope(x) true if x defined in current scope
- exit_scope() exit current scope

# Advanced Symbol Table

- Class names can be used before they are defined.

- We can’t check class names using
  - Symbol Tables and One pass
- Solution:
  - Pass1: Gather all class names
  - Pass2: Do the checking

- Semantic Analysis often require multiple passes

# Types

- What is a type?
  - A set of values
  - A set of operations defined on those values
  - However,  the notion may vary from language to language
- Classes are one instantiation of the modern notion of type

# Why Do We Need Type Systems?

- Consider the assembly language fragment

  add $r1, $r2, $r3

- What are the types of $r1, $r2, $r3?
- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

# Type Systems

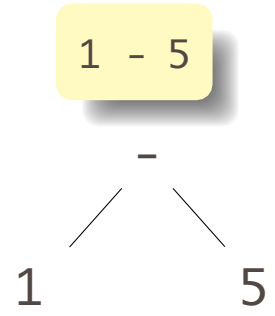
- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!
- Three kinds of languages:
  - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Python)
  - Untyped: No type checking (machine code)

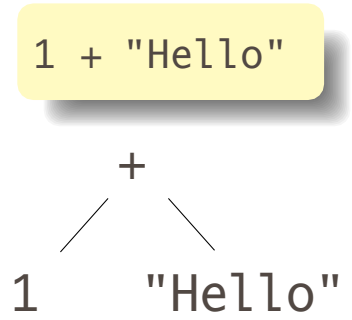
# Static vs. Dynamic Typing

- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping difficult within a static type system
- In practice
  - code written in statically typed languages usually has an escape mechanism •
    - Unsafe casts in C, Java
  - Some dynamically typed languages support “pragmas” or “advice” • i.e., type declarations.

# How To Check Expressions: Depth-first AST Walk

Checking function: environment → node → type



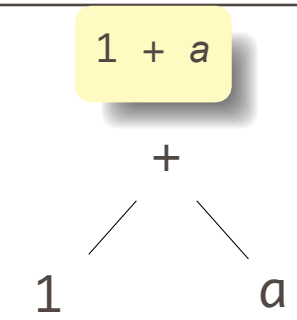

check(–)
- check(1) = int
- check(5) = int
- Success: int – int = int

check(+)
- check(1) = int
- check("Hello") = string
- FAIL: Can't add int and string

Ask yourself: at each kind of node, what must be true about the nodes below it? What is the type of the node?

# How To Check: Symbols

Checking function: environment → node → type




check(+)
    check(1) = int
    check(a) = int  Success:
    int + int = int

The key operation: determining the type of a symbol when it is encountered.

The environment provides a “symbol table” that holds information about each in-scope symbol.

# A Static Semantic Checking Function

A big function: “check: ast → sast”

Converts a raw AST to a “semantically checked AST”

Names and types resolved

```
AST
type expression =
          IntConst  of int
     | Id of string
     | Call of string * expression list
     | ...
```

→

```
SAST
type expr_detail =
          IntConst  of int
     | Id of variable_decl
     | Call of function_decl * expression list
     | ...type expression = expr_detail * Type.t
```

# Strong vs. Weak Typing

- A program introduces type-confusion when it attempts to interpret a memory region populated by a datum of specific type T1, as an instance of a different type T2 and T1 and T2 are not related by inheritance.

- Strongly typed if it explicitly detects type confusion and reports it as such
  - (e.g., with Java).

- Weakly typed if type-confusion can occur silently (undetected), and eventually cause errors that are difficult to localize.
  - C and C++ are considered weakly typed since, due to type-casting, one can interpret a field of a structure that was an integer as a pointer.

# Question

1. #include <stdio.h> int main() { int i = 0; char j = '5'; printf("%d\n", (i+j)); return 0; }
( Single Choice)

Answer 1: error

Answer 2: 5

Answer 3: 53

Answer 4: None

2. int main() { float p = 0.5; char* q = "hello"; int c = p + q; printf("%d\n",c); return 0; }
( Single Choice)

Answer 1: error

Answer 2: 4195796

Answer 3: other

# Question

1. What would be the output of the following Python Code?

```
def type_check(a):
    p = 7
   return (p + a)
print(type_check('4'))
```

( Single Choice)

Answer 1: error

Answer 2: 11

Answer 3: 74

2. What would be the output of the following Python Code?

```
def type_check(a):
    p = 7
   return (p + a)
print(type_check(4))
```

( Single Choice)

Answer 1: error

Answer 2: 11

Answer 3: 74

# Question

1. What will be the output of the following Java code?

```
class Test {
 public static void main(String args[]) {
   for (int x = 0; x < 4; x++) { ... }
      System.out.println(x); }
   }
```

Answer 1: 3

Answer 2: error

Answer 3: 4