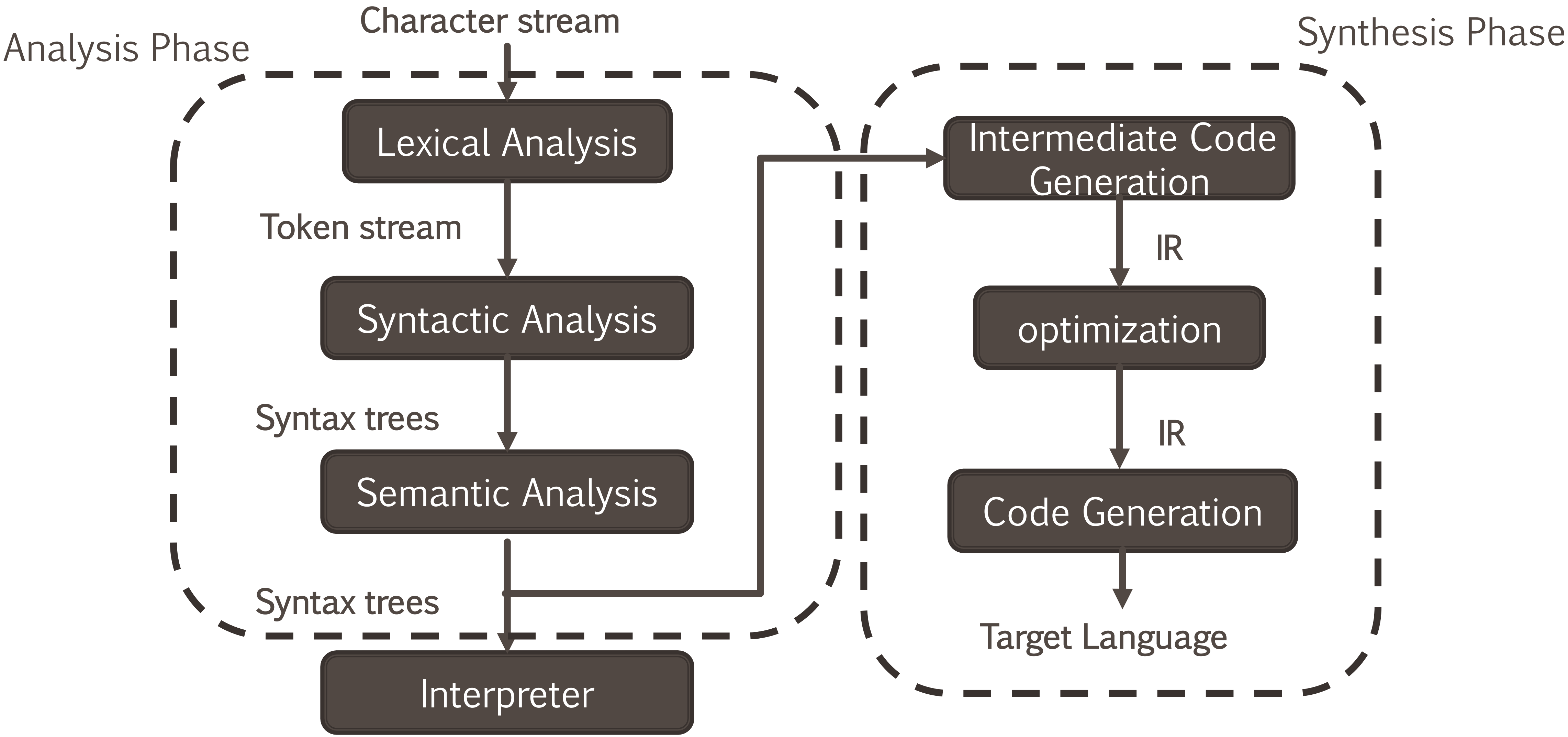
Programming Languages & Translators

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?

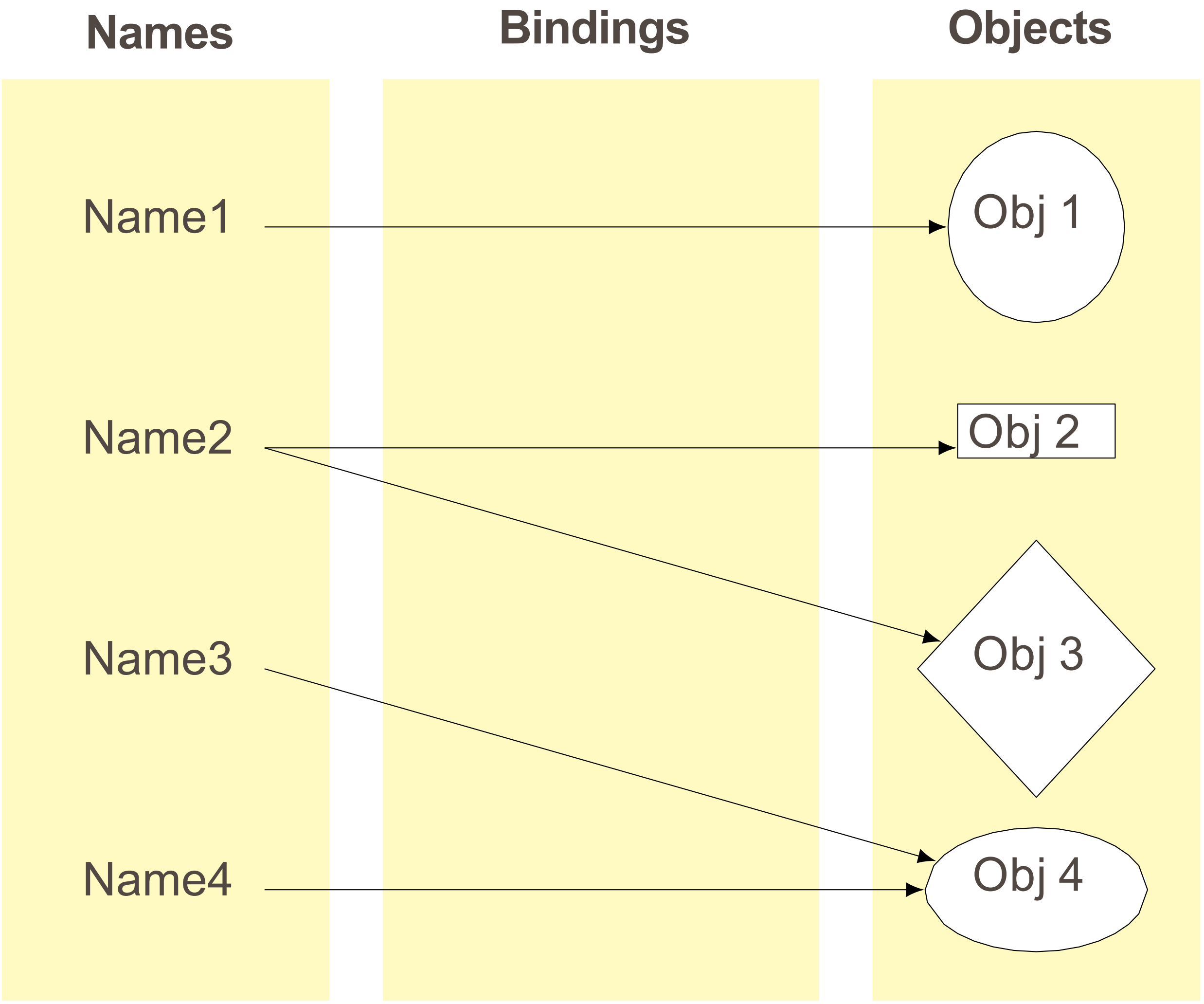
Does `f` accept whatever `b` and `c` are?

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.



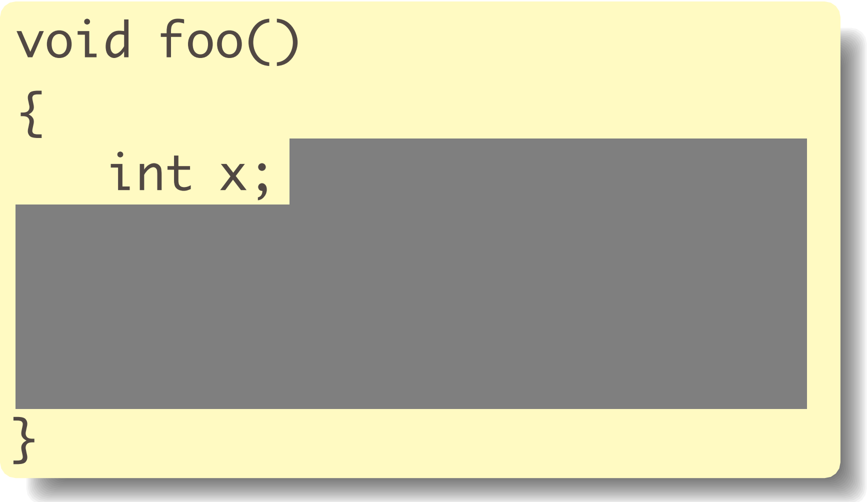
Static Vs. Dynamic Scoping

- 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.



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

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.”

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.

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but the terms are often used interchangeably
- Rules of Inference
  - We have seen two examples of formal notation specifying parts of a compiler : Regular expressions, Context-free grammars
  - The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form If Hypothesis is true, then Conclusion is true
- Type checking computes via reasoning

If E1 and E2 have certain types, then E3 has a certain type

- Rules of inference are a compact notation for “If-Then” statements

From English to an Inference Rule

- The notation is easy to read with practice
- Start with a simplified system and gradually add features
- Building blocks
    - Symbol $\wedge$ is “and”
    - Symbol $\Rightarrow$ is “if-then”
    - x:T is “x has type T”
- If e_1 has type Int and e_2 has type Int, then $e_1 + e_2$ has type Int
    - (e1 has type Int $\wedge$ e2 has type Int) $\Rightarrow$ e1 + e2 has type Int
    - (e1: Int $\wedge$ e2: Int) $\Rightarrow$ e1 + e2: Int
    - It is a special case of $\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$ (This is an inference rule).

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis} \ldots \vdash \text{Hypothesis}}{\vdash \text{Conclusion}}$$

$\vdash$ e:T means “it is provable that e is of type T

Two Rules

$$\frac{\vdash \ \text{i is } \mathit{an\ integer\ literal}}{\vdash \ \text{i: Int}} \ \text{[Int]}$$

$$\frac{\vdash \ \text{e1: Int} \qquad \vdash \ \text{e2: Int}}{\vdash \ \text{e1+e2: Int}} \ \text{[Add]}$$

$$\frac{\vdash \ \text{e: Boo}l}{\vdash \ !e: Bool} \ \text{[Not]}$$

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions
- Example: 1 + 2?

Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e' s sub-expressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

A Problem

- What is the type of a variable reference?
- $$\frac{x\ is\ a\ variable}{\vdash x : ?}$$
- The local, structural rule does not carry enough information to give x a type.

A solution

- Put more information in the rules!

- A type environment gives types for free variables
  - A type environment is a function from ObjectIdentifiers to Types
  - A variable is free in an expression if it is not defined within the expression

- Type Environments
  - Let O be a function from ObjectIdentifiers to Types

  The sentence O $\vdash$ e: T

  is read: Under the assumption that free variables have the types given by O, it is provable that the expression e has the type T

  - $$\frac{O(x) = T}{\vdash x : T}$$

Implementing Type Checking

$$\frac{O, M, C \vdash e1\colon Int \quad O, M, C \vdash e2\colon Int}{O, M, C \vdash e1 + e2\colon Int}$$

```
TypeCheck(Environment, e1 + e2) = {
T1 = TypeCheck(Environment, e1);
T2 = TypeCheck(Environment, e2);
Check T1 == T2 == Int;
return Int; }
```

Binding Time

When are bindings created and destroyed?

Binding Time

When a name is connected to an object.

Bound when	**Examples**
language designed	`if else` datatype
language implemented	widths `foo bar`
Program written	static addresses, code
compiled	relative addresses shared
linked	objects
loaded	heap-allocated objects
run	

Binding Time and Efficiency

Earlier binding time ⇒ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```
switch (statement) {

case add:
   r = a + b;
   break;

case sub:
   r = a - b;
   break;


   /* ... */
}
```

```
add %o1, %o2, %o3
```

Binding Time and Efficiency

Dynamic method dispatch in OO languages:

```
class Box : Shape {
  public void draw() { ...}
}

class Circle : Shape {
  public void draw() { ...}
}

Shape s;
s.draw(); /* Bound at run time */
```

Static Semantic Analysis

How do we validate names, scope, and types?

Static Semantic Analysis

Lexical analysis: Each token is valid?

```
if i 3 "This"                   /* valid Java tokens */
#a1123                          /* not a token */
```

Syntactic analysis: Tokens appear in the correct order?

```
for ( i = 1 ; i < 5 ; i++ ) 3 + "foo"; /* valid Java syntax */
for break                              /* invalid syntax */
```

Semantic analysis: Names used correctly? Types consistent?

```
int v = 42 + 13;        /* valid in Java (if v is new) */
return f + f(3);        /* invalid */
```

What To Check

Examples from Java:

Verify names are defined and are of the right type.

```
int i = 5;
int a = z;      /* Error: cannot find symbol */
int b = i[3]; /* Error: array required, but int found */
```

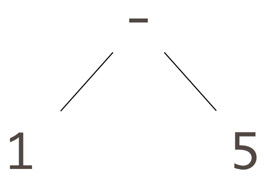
Verify the type of each expression is consistent.

```
int j = i + 53;
int k = 3 + "hello";     /* Error: incompatible types */
int l = k(42);           /* Error: k is not a method */
if ("Hello") return 5; /* Error: incompatible types */
String s = "Hello";
int m = s;               /* Error: incompatible types */
```

How To Check Expressions: Depth-first AST Walk

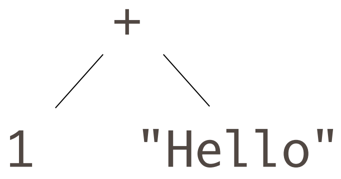
Checking function: environment → node → type

```
1 - 5
```



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

```
1 + "Hello"
```



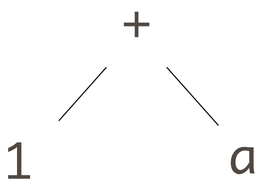
check(+)
- check(1) = int
- check("Hello") = string
- FAIL: Can’t add int and string

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

1 + *a*



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

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

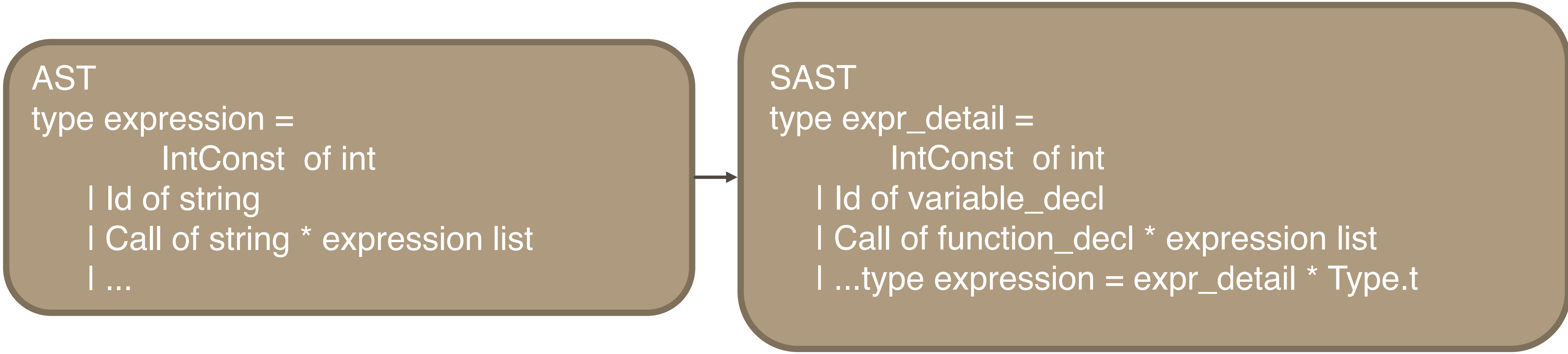
The environment provides a “symbol table” that holds information about each in-scope symbol.

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
```