

# Semantic analysis & type checking

The contents are copied from

<https://web.stanford.edu/class/cs143/lectures/lecture09.pdf>

<https://courses.cs.washington.edu/courses/cse401/22au/lectures/l-semantics.pdf>

<https://www3.nd.edu/~dthain/compilerbook/chapter7.pdf>

Chapter 4 of Engineering a Compiler,  
Cooper & Torczon, 3rd edition.

# Content

Reading: For an example abstract syntax tree implementation in C

<https://www3.nd.edu/~dthain/compilerbook/chapter6.pdf>

# So far

## Lexical analysis

- Detects inputs with illegal tokens

## Parsing

- Detects inputs with ill-formed parse trees

- ❖ Parsing cannot catch some errors
- ❖ Some language-constructs not context-free

# How to verify/know/check if it is a legal program?

```
class C {  
    int a;  
    C(int initial) {  
        a = initial;  
    }  
    void setA(int val) {  
        a = val;  
    }  
}
```

```
class Main {  
    public static void main(){  
        C c = new C(17);  
        c.setA(42);  
    }  
}
```

- Semantic analysis
  - ◆ Last “front end” phase
  - ◆ Catches all remaining errors

# Beyond Syntax: What Your Compiler Really Needs to Check

a level of correctness that is not captured by a context-free grammar



```
// Syntax says this is fine, but semantics say NO!  
String name = 42;           // Type mismatch  
user.printInfo(1, 2, 3);    // Wrong number of arguments  
x = y + z;                  // Are y,z declared? Compatible types?  
return "hello";             // Method expects int return type
```

The requirements depend on the language

# Real-World Semantic Errors

TypeScript (Static Checking):

```
let name: string = "Alice";  
name = 42; // Error: Type 'number' is not assignable to type 'string'  
  
function greet(person: string): string {  
    return `Hello, ${person}`;  
}  
greet(123); // Error: Argument of type 'number' is not assignable
```

```
# Runtime error (dynamic typing)  
def add(a, b):  
    return a + b  
  
add("hello", 5) # TypeError at runtime!  
  
# With type hints + mypy (static checking)  
def add(a: int, b: int) -> int:  
    return a + b  
  
add("hello", 5) # mypy: error: incompatible type
```

# Real-World Semantic Errors

Python (Runtime vs Static):

```
# Runtime error (dynamic typing)
def add(a, b):
    return a + b

add("hello", 5) # TypeError at runtime!

# With type hints + mypy (static checking)
def add(a: int, b: int) -> int:
    return a + b

add("hello", 5) # mypy: error: incompatible type
```

# Real-World Semantic Errors

Rust (Compile-time Safety):

```
let mut data: String = "hello".to_string();  
let reference = &data;  
data.push_str(" world"); // Error: cannot borrow as mutable  
                        // while borrowed as immutable
```

# What Does Semantic Analysis Do?

1. All identifiers are declared
2. Types
3. Inheritance relationships
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers are not misused And others . .

## Modern Tools & Approaches:

- Static Analysis: TypeScript, Rust, MyPy
- Symbol Management: Modern IDEs, Language Server Protocol
- Type Inference: Hindley-Milner (Haskell, OCaml), Flow Analysis

The requirements depend on the language

# Semantic Checks

For each language construct we want to know:

- What semantic rules should be checked

Specified by language definition (type compatibility, required initialization, etc.)

- For an expression, what is its type (used to check whether expression is legal in the current context)
- For declarations, what information needs to be captured to use elsewhere

# Examples of semantic checks

Consider an identifier  $x$ ; to generate executable code, we have to know

- What kind of value is stored in  $x$ ?
- How big is  $x$ ?
- if  $x$  is a procedure, what arguments does it take? What kind of value, if any, does it return?
- How long must  $x$ 's value be preserved?
- Who is responsible for allocating space for  $x$  (and initializing it)?

Appearance of a name: **id**

- Check: id has been declared and is in scope
- Compute: Inferred type of id is its declared type

Constant: v

- Compute: Inferred type and value are explicit

Binary operator: **exp1 op exp2**

- Check: **exp1** and **exp2** have compatible types
  - Either identical, or
  - Well-defined conversion to appropriate types
- Compute: Inferred type is a function of the operator and operand types

Field reference: **exp.f**

- Check: exp is a reference type (not primitive type)
- Check: The class of exp has a field named f
- Compute: Inferred type is declared type of f

Assignment: **exp1 = exp2**

- Check: exp1 is assignable (not a constant or expression)
- Check: exp1 and exp2 have (assignment-)compatible types
  - Identical, or
  - exp2 can be converted to exp1 (e.g., int to double), or
  - Type of exp2 is a subclass of type of exp1 (can be decided at compile time)
- Compute: Inferred type is type of exp1

## Cast: **(exp1) exp2**

- Check: exp1 is a type
- Check: exp2 either
  - Has same type as exp1
  - Can be converted to type exp1 (e.g., double to int)
  - Downcast: is a superclass of exp1 (in general this requires a runtime check to verify type safety; at compile time we can at least decide if it could be true)
- Upcast (Trivial): is the same or a subclass of exp1
- Compute: Inferred type is exp1

Method call: **exp.m(e1, e2, ..., en)**

- Check: exp is a reference type (not primitive type)
- Check: The type of exp has a method named m
  - (inherited or declared as part of the type)
- Check: The method m has n parameters
  - Or, if overloading is allowed, at least one version of m exists with n parameters
- Check: Each argument has a type that can be assigned to the associated parameter
  - Same as “assignment compatible” check for assignment
  - Overloading: need to find a “best match” among available methods if more than one is compatible – or reject if result is ambiguous (e.g., full Java, C++, others)
- Compute: Inferred (result) type is given by methoddeclaration (or could be void)

Return statement: **return exp;** or: **return;**

Check:

- If the method is not void: The expression can be assigned to a variable that has the declared return type of the method
  - exactly the same test as for assignment statement and method call-by-value argument/parameter types
- If the method is void: There is no expression

# Scope of an identifier

- 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 scope

Most languages have static scope

- Scope depends only on the program text, not run-time behavior

A few languages are dynamically scoped

- Lisp, SNOBOL
- Lisp has changed to mostly static scoping
- **Scope depends on execution of the program**

```
# A perl code with dynamic scoping
```

```
$x = 10;
```

```
sub f{
```

```
    return $x;
```

```
}
```

```
sub g{
```

```
    # with local, x uses dynamic scoping.
```

```
    local $x = 20;
```

```
    return f();
```

```
}
```

```
print g()."\n";#outputs 20
```

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.6>

# scope

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

- In some languages, a class name can be used before it is defined
- Attribute names are global within the class in which they are defined

```
class Point {  
    int x, y;           // coordinates  
    PointColor color;   // color of this point  
    Point next;         // next point with this color  
}  
class PointColor {  
    Point first;        // first point with this color  
    PointColor(int color) { this.color = color; }  
    private int color;  // color components  
}
```

## **Method/attribute names have complex rules**

- A method need not be defined in the class in which it is used, but in some parent class
- Methods may also be redefined (overridden)

# Semantics checking

Much of semantic analysis can be expressed as a recursive descent of an AST

When performing semantic analysis on a portion of the AST,

we need to know which identifiers are defined

## Implementing the Most-Closely Nested Rule

- **Before:** Process an AST node  $n$
- **Recurse:** Process the children of  $n$
- **After:** Finish processing the AST node  $n$

# Symbol tables

A symbol table is a data structure that tracks the current bindings of identifiers

Symbol tables will hold environment information – i.e., properties of every name declared or used in the code

Add fields to AST nodes to refer to appropriate attributes

- symbol table entries for identifiers, types for expressions including identifiers, etc.

Map identifiers to

<type, kind, location, other properties>

```
static int interest;
```

- <interest, int, static>

Operations

- Lookup(id) => information
- Enter(id, information)
- Open/close scopes

Build & use during semantics pass

- Build first from declarations
- Then use to check semantic rules

Use (and augment) in later compiler phases

# Implementing symbol table

Stack, Linear (sorted or unsorted) list, Binary Search Tree, Hash table

Big topic in classical (i.e., ancient) compiler courses: implementing a **hashed symbol table**

- ★ use the collection classes that are provided with the standard language libraries (Java, C#, C++, ML, Haskell, etc.)

# Semantic analysis requires more than one pass

Class names can be used before being defined

We can't check class names

- using a symbol table
- or even in one pass

Solution

- Pass 1: Gather all class names
- Pass 2: Do the checking

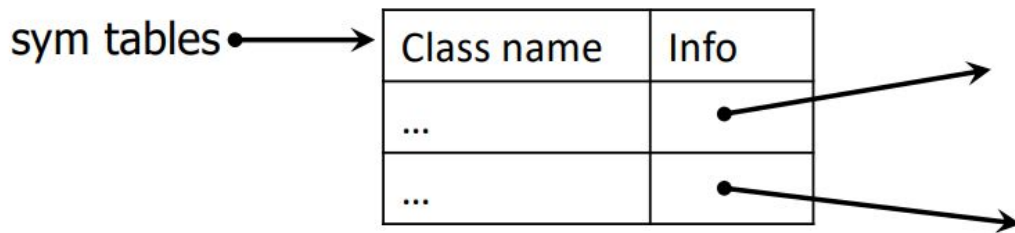
# Symbol table implementations for miniJava

## Global – Per Program Information

Single global table to map class names to per-class symbol tables

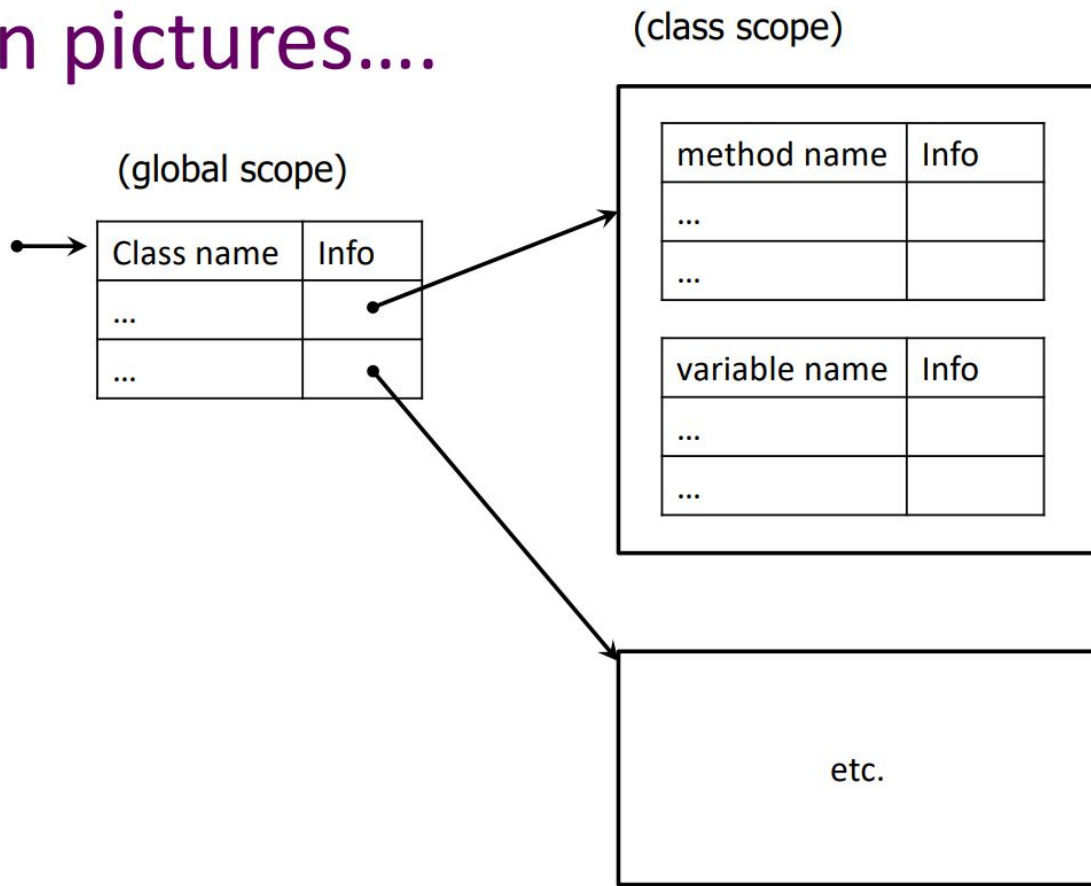
Created in a pass over class definitions in AST

Used in remaining parts of compiler to check class types and their field/method names and extract information about them



# One symbol table for each class

In pictures....



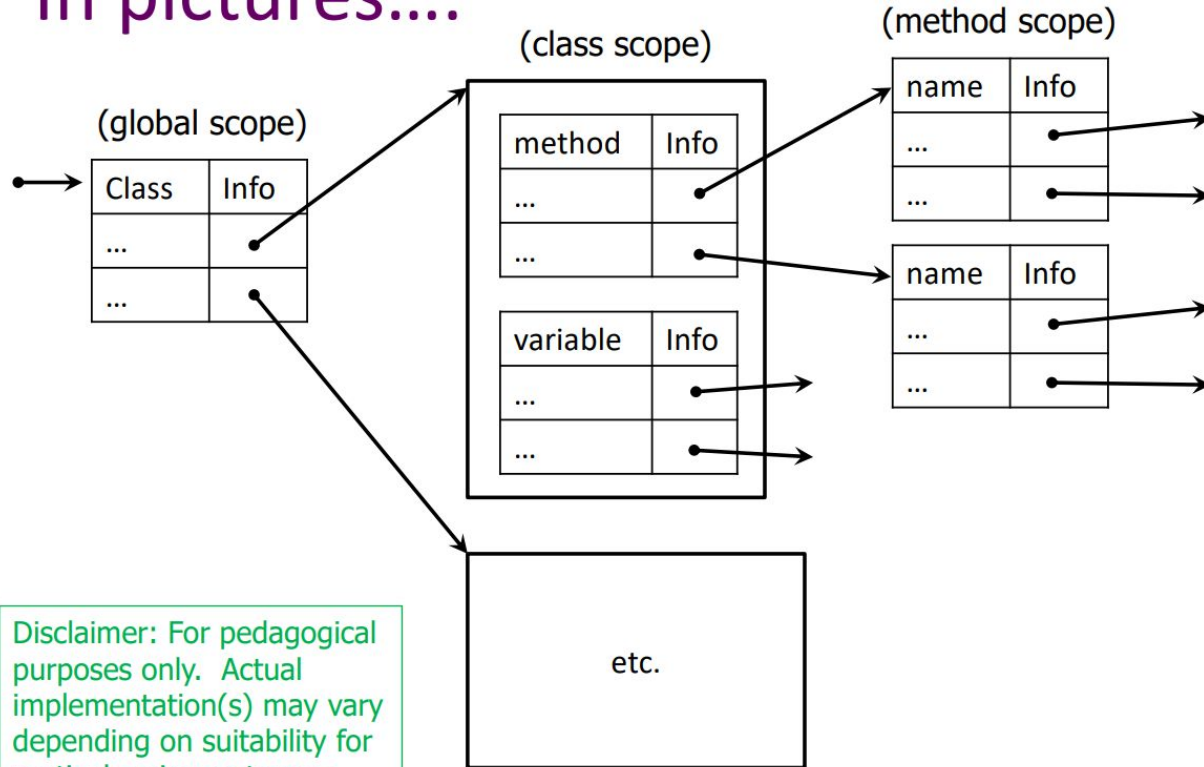
All global tables persist throughout the compilation

And beyond in a real compiler...

- Symbolic information in Java .class or MSIL files, link-time optimization information in gcc .o files)
- Debug information in .o and .exe files
- Some or all of this information in library files (.a, .so)
- Type information for garbage collector

# One local symbol table for each method

in pictures....



Disclaimer: For pedagogical purposes only. Actual implementation(s) may vary depending on suitability for particular circumstances.

# What is missing in miniJava symbol tables

What we aren't dealing with: nested scopes

- Inner classes
- Nested scopes in methods
- reuse of identifiers in parallel or inner scopes (most languages); nested functions (ML etc. ...)
- Lambdas and function closures (Racket, JavaScript, Java, C#, , ...)

Basic idea: new symbol table for inner scopes, linked to surrounding scope's table (i.e., stack of symbol tables, top = current innermost scope, bottom = global scope)

- Look for identifier in inner scope (top); if not found look in surrounding scope (recursively)
- Pop symbol table when we exit a scope

Also ignoring static fields/methods, accessibility (public, protected, private), package scopes, ...

# Engineering issues

In multipass compilers, inner scope symbol table needs to persist to use in later passes

- Can't really delete symbol tables on scope exit
- Retain tables and add a pointer to the parent scope table (effectively a reverse tree of symbol tables for nested scopes with root = global table)

Keep a pointer to current innermost scope (usually a leaf but could be interior node) and start looking for symbols there

In practice, often want to retain  $O(1)$  lookup or something close to it

- Would like to avoid  $O(\text{depth of scope nestings})$
- Use hash tables when possible with linked list

# Modern Symbol Table Implementations

From Hash Tables to  
Advanced Scoping

Traditional Approach  
(Stack of Scopes):

```
class SymbolTable:
    def __init__(self):
        self.scopes = [{}] # Start with global scope

    def enter_scope(self):
        self.scopes.append({})

    def exit_scope(self):
        self.scopes.pop()

    def add_symbol(self, name, info):
        self.scopes[-1][name] = info

    def lookup(self, name):
        for scope in reversed(self.scopes):
            if name in scope:
                return scope[name]
        return None
```

## Modern Functional Approach (Immutable):

```
type symbol_info = { name: string; type: typ; scope: int }  
type symbol_table = symbol_info list  
  
let add_symbol info table = info :: table  
  
let lookup name table =  
    List.find_opt (fun sym -> sym.name = name) table  
  
let enter_scope table = table (* No mutation! *)  
let exit_scope table old_table = old_table
```

## Real-World Example: TypeScript Compiler:

```
// TypeScript uses complex symbol merging
interface Symbol {
  flags: SymbolFlags;
  name: string;
  declarations: Declaration[];
  valueDeclaration: Declaration;
  members?: SymbolTable;      // For classes/interfaces
  exports?: SymbolTable;     // For modules
}
```

# Types

What is a type?

- The notion varies from language to language

Consensus

- A set of values
- A set of operations on those values

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**?

# Types and Operations

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!

The type system plays role in

- **Correctness (Runtime Safety),**
- **Performance (Better Code),**
- **Expressiveness (e.g. operator overloading)**

# terminology

## Static vs dynamic typing

- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

## Weak vs strong typing

- strong: guarantees no illegal operations performed
- Weak can't make guarantees

## Hybrids are common

“untyped,” “typeless” could mean dynamic or weak

```
/* This is C++11 code */  
auto x = 32.5;  
cout << x << endl;
```

	static	dynamic
strong	Java, SML	Scheme, Ruby
weak	C	PERL

# Dynamic vs static types

Competing views on static vs. dynamic typing

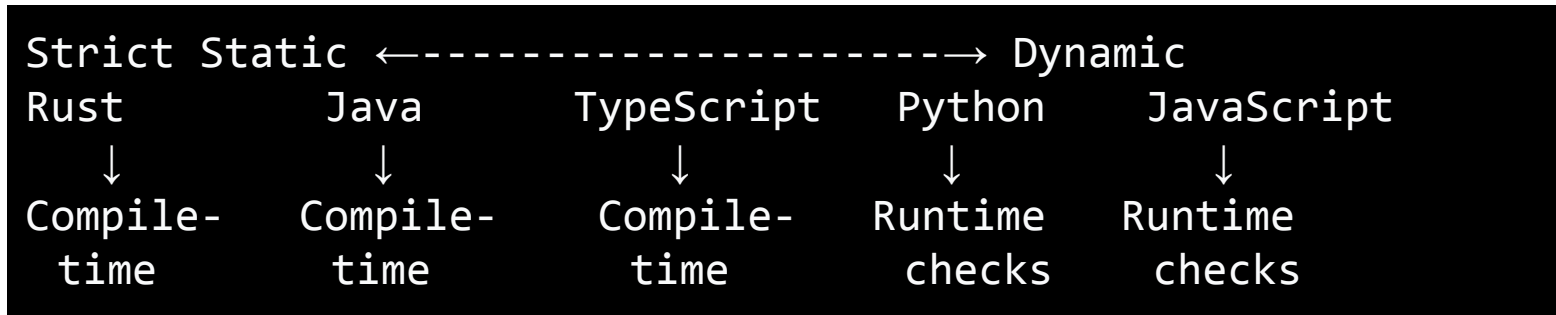
- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping difficult within a static type system

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 Systems in Modern Languages

## Static vs Dynamic Typing Spectrum



# Type Systems in Modern Languages

Gradual Typing (Best of Both Worlds):

```
# Python with gradual typing
def process_data(data: List[int]) -> Dict[str, float]:
    result = {}          # Type inferred as Dict[str, float]
    for item in data:    # item inferred as int
        result[str(item)] = item * 1.5
    return result

# Mixed typing allowed
untyped_data = [1, 2, "3"]  # No type checking
process_data(untyped_data)  # Runtime error!
```

# Type Systems in Modern Languages

**Nullability:** the ability of a variable to hold a null value.

**Null Safety (Modern Trend):** Catching potential null-related issues at compile time rather than runtime

```
// Kotlin - null safety at compile time  
  
var name: String = "John"    // Non-nullable  
name = null                  // Compilation error  
  
var nullableName: String? = "John" // Nullable  
nullableName = null           // OK
```

# Practical Type Checking Implementation

## Type Checker Architecture

```
class TypeChecker:
    def __init__(self):
        self.symbol_table = SymbolTable()
        self.current_class = None
        self.current_method = None

    def visit_binary_op(self, node):
        left_type = self.visit(node.left)
        right_type = self.visit(node.right)

        # Check type compatibility
        if not self.is_compatible(left_type, right_type, node.op):
            self.error(f"Type mismatch: {left_type} {node.op} {right_type}")
        return self.result_type(node.op, left_type, right_type)
```

```
def visit_assignment(self, node):
    target_type = self.visit(node.target)
    value_type = self.visit(node.value)

    if not self.is_assignable(target_type, value_type):
        self.error(f"Cannot assign {value_type} to {target_type}")

    return target_type

def visit_method_call(self, node):
    obj_type = self.visit(node.object)
    method_info = self.lookup_method(obj_type, node.method_name)

    if not method_info:
        self.error(f"Method {node.method_name} not found in {obj_type}")

    # Check arguments
    self.check_arguments(method_info, node.arguments)
    return method_info.return_type
```

# Modern Type Inference Examples

How Compilers "Guess"

Types

TypeScript Type

Inference:

```
let name = "Alice";           // inferred as string
let age = 25;                  // inferred as number
let scores = [95, 87, 92];     // inferred as number[]

// Function return type inference
function calculateTotal(scores: number[]) {
    return scores.reduce((a, b) => a + b, 0);
    // return type inferred as number
}

// Generic type inference
function identity<T>(value: T): T {
    return value;
}

let result = identity("hello"); // T inferred as string
```

# Modern Type Inference Examples

## Rust Type Inference:

```
let numbers = vec![1, 2, 3]; // inferred as Vec<i32>

let doubled: Vec<_> = numbers.iter().map(|x| x * 2).collect();

// Rust infers the type inside Vec based on x * 2

// Pattern matching with type inference
if let Some(name) = get_name() {
    println!("Hello, {}", name); // name's type inferred from get_name()
}
```

# Modern Type Inference Examples

Hindley-Milner type inference  
(OCaml):

```
let rec map f lst =  
  match lst with  
  | [] -> []  
  | x::xs -> (f x) :: (map f xs)  
  
(* Type inferred as: ('a -> 'b) -> 'a list -> 'b list *)
```

[Hindley–Milner type system - Wikipedia](#)

A Hindley–Milner (HM) type system is a classical type system for the lambda calculus with parametric polymorphism.

# Components of a type system

## Base Types

- Fundamental, atomic types
- Typical examples: int, double, char, bool

## Compound/Constructed Types

- Built up from other types (recursively)
  - records/structs/classes,
  - arrays,
  - pointers,
  - enumerations,
  - functions, modules, ...

Most language provide a small collection of these

\*Strings may be also constructed type (C array of chars)

# How to Represent Types in a Compiler?

One solution: create a shallow class hierarchy

- Should not need too many of these

- Example:

```
abstract class Type { ... } // or interface
```

```
class BaseType extends Type { ... }
```

```
class ClassType extends Type { ... }
```

```
class ClassType extends Type {  
    Type baseClassType; // ref to base class  
    Map fields; // type info for fields  
    Map methods; // type info for methods  
}
```

```
class ArrayType extends Type {  
    int nDims;  
    Type elementType;  
}
```

```
class MethodType extends Type {  
    Type resultType; // type or “void”  
    List<Type> parameterTypes;  
}
```

# 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

- Start with a simplified system and gradually add features

Symbols for Building blocks

$\wedge$

- “and”

$\Rightarrow$

- “if-then”

$x:T$

- “x has type T”

$$(e1: \text{Int} \wedge e2: \text{Int}) \Rightarrow e1 + e2: \text{Int}$$

if **e1** has type **Int** and **e2** has type **Int**,

- then **e1 + e2** has type **Int**

**(e1 has type Int  $\wedge$  e2 has type Int)  $\Rightarrow$**

- **e1 + e2** has type **Int**

$$(e1: \text{Int} \wedge e2: \text{Int}) \Rightarrow e1 + e2: \text{Int}$$

is a special case of

$$\text{Hypothesis1} \wedge \dots \wedge \text{Hypothesisn} \Rightarrow \text{Conclusion}$$

**This is an inference rule.**

# Notation for Inference Rules

Modern inference rules are written

$\vdash$  **Hypothesis** ...  $\vdash$  **Hypothesis**

---

$\vdash$  **Conclusion**

Type rules

$\vdash e:T$

$\vdash$  means “it is provable that . . .”

[int]

**i is an integer literal**

---

$\vdash i : \text{Int}$

[Add]

$\vdash e1 : \text{Int} \vdash e2 : \text{Int}$

---

$\vdash e1 + e2 : \text{Int}$

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

By filling in the templates, we can produce complete typings for expressions

# Example 1+2 (type of an expression)

1 is an int literal

2 is an int literal

For Cool programming language

---

$\vdash 1 : \text{Int}$

---

$\vdash 2 : \text{Int}$

---

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

## Soundness

A type system is sound if

- Whenever  $\vdash e : T$
- Then  $e$  evaluates to a value of type  $T$
- We only want sound rules
- But some sound rules are better than others:

**$i$  is an integer literal**

---

**$\vdash i : \text{Object}$**

## 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 subexpressions
- Conclusion is the type of  $e$

Types are computed in a bottom-up pass over the AST

# More Rules

[false]

---

$\vdash \text{false: boolean}$

[String]

s is a string literal

---

$\vdash s: \text{String}$

**[New]**

---

$\vdash \text{new } T: T$

new T Produces object of type T

**[Not]**

$\vdash e: \text{Bool}$

---

$\vdash !e: \text{Bool}$

**[Loop]**

$\vdash e1: \text{Bool}$

$\vdash e2: T$

---

$\vdash \text{while } e1 \text{ loop } e2 \text{ pool} : \text{Object}$

[var]

**x is a variable**

---

$\vdash x : ?$

The local, structural rule does not carry enough information to give x a type

Put more information in the rules!

$O \vdash e : T$

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

- $O$  a function from **ObjectIdentifiers** to **Types**

[int]

**i is an integer literal**

---

$O \vdash i : \text{Int}$

[var]

$O(x) = T$

---

$O \vdash x: T$

If then else

if  $e_0$  then  $e_1$  else  $e_2$  fi

The result can be either  $e_1$  or  $e_2$

The type is either  $e_1$ 's type or  $e_2$ 's type

The best we can do is the smallest supertype larger than the type of  $e_1$  or  $e_2$

[if-then-else]

$O \vdash e_0: \text{Bool}$

$O \vdash e_1: T_1$

$O \vdash e_2: T_2$

---

$O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi}: \text{lub}(T_1, T_2)$

(LUP= least upper bound)

## Methods

### [dispatch]

$O \vdash e_0: T_0$

$O \vdash e_1: T_1$

$\dots$

$O \vdash e_n: T_n$

---

$O \vdash e_0.f(e_1, \dots, e_n): ?$

$M(C, f) = (T_1, \dots, T_n, T_{n+1})$

in class  $C$  there is a method  $f$

$f(x_1:T_1, \dots, x_n:T_n): T_{n+1}$

Example:

`unsigned int strlen(const char *s);`

`strlen : const char *  $\rightarrow$  unsigned int`

## Methods

In most cases,  $M$  is passed down but not actually used

[dispatch]

$$\begin{array}{l} O, M \vdash e_0: T_0 \\ O, M \vdash e_1: T_1 \\ \dots \\ O, M \vdash e_n: T_n \\ M(T_0, f) = (T'_1, \dots, T'_n, T_{n+1}) \\ \quad T_i \leq T'_i \text{ for } 1 \leq i \leq n \end{array}$$

---

$$O, M \vdash e_0.f(e_1, \dots, e_n): T_{n+1}$$

[add]

$$\begin{array}{l} O, M \vdash e_1: \text{Int} \\ O, M \vdash e_2: \text{Int} \end{array}$$

---

$$O, M \vdash e_1 + e_2 : \text{Int}$$

In most cases, M is passed down from parent to child in AST but not actually used

[add]

type environments

- A mapping O giving types to object ids
- A mapping M giving types to methods
- The current class C

Sentence

$O, M, C \vdash e : T$

$O, M, C \vdash e1 : \text{Int}$

$O, M, C \vdash e2 : \text{Int}$

---

$O, M, C \vdash e1 + e2 : \text{Int}$

Note:

Type environment is passed down the tree  
– From parent to child

Types are passed up the tree  
– From child to parent

In most cases,  $M$  is passed down from parent to child in AST but not actually used

[add]

## Implementing type systems

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

See also

<https://www3.nd.edu/~dthain/compilerbook/chapter7.pdf> for an implementation in C

$$O, M, C \vdash e1 : \text{Int}$$
$$O, M, C \vdash e2 : \text{Int}$$

---

$$O, M, C \vdash e1 + e2 : \text{Int}$$

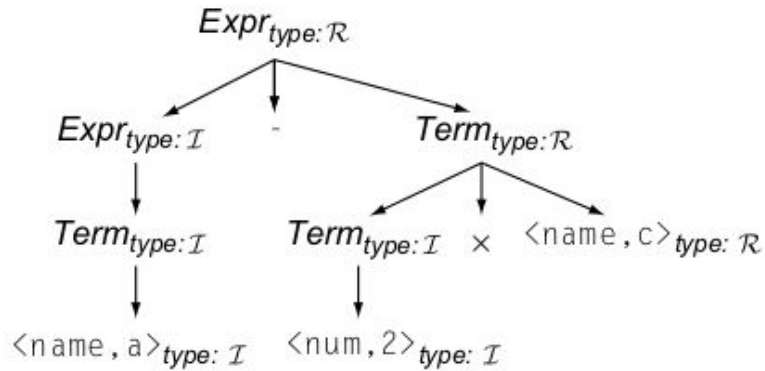
Note:

Type environment is passed down the tree  
– From parent to child

Types are passed up the tree  
– From child to parent

# Some more examples

## Inferring Expression Types



Production	Attribution Rules
$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$
$  Expr_1 - Term$	$Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$
$  Term$	$Expr_0.type \leftarrow Term.type$
$Term_0 \rightarrow Term_1 Factor$	$Term_0.type \leftarrow \mathcal{F}_\times(Term_1.type, Factor.type)$
$  Term_1 Factor$	$Term_0.type \leftarrow \mathcal{F}_\div(Term_1.type, Factor.type)$
$  Factor$	$Term_0.type \leftarrow Factor.type$
$Factor \rightarrow (Expr)$	$Factor.type \leftarrow Expr.type$
$  num$	$num.type \text{ is already defined}$
$  name$	$name.type \text{ is already defined}$

■ FIGURE 4.7 Attribute Grammar to Infer Expression Types.

# A simple execution time estimator

Production	Attribution Rules
$Block_0 \rightarrow Block_1 \text{ Assign}$	$\{ Block_0.cost \leftarrow Block_1.cost + Assign.cost \}$
$\quad   \text{ Assign}$	$\{ Block_0.cost \leftarrow Assign.cost \}$
$Assign \rightarrow name = Expr;$	$\{ Assign.cost \leftarrow Cost(store) + Expr.cost \}$
$Expr_0 \rightarrow Expr_1 + Term$	$\{ Expr_0.cost \leftarrow Expr_1.cost + Cost(add) + Term.cost \}$
$\quad   Expr_1 - Term$	$\{ Expr_0.cost \leftarrow Expr_1.cost + Cost(sub) + Term.cost \}$
$\quad   Term$	$\{ Expr_0.cost \leftarrow Term.cost \}$
$Term_0 \rightarrow Term_1 \times Factor$	$\{ Term_0.cost \leftarrow Term_1.cost + Cost(mult) + Factor.cost \}$
$\quad   Term_1 \div Factor$	$\{ Term_0.cost \leftarrow Term_1.cost + Cost(div) + Factor.cost \}$
$\quad   Factor$	$\{ Term_0.cost \leftarrow Factor.cost \}$
$Factor \rightarrow (Expr)$	$\{ Factor.cost \leftarrow Expr.cost \}$
$\quad   num$	$\{ Factor.cost \leftarrow Cost(loadI) \}$
$\quad   name$	$\{ Factor.cost \leftarrow Cost(load) \}$

■ FIGURE 4.8 Simple Attribute Grammar to Estimate Execution Time.

# Error recovery

As with parsing, it is important to recover from type errors

- Detecting where errors occur is easier than in parsing
  - There is no reason to skip over portions of code
- The Problem:
  - What type is assigned to an expression with no legitimate type?
  - This type will influence the typing of the enclosing expression

# Semantic Analysis in Project Paths

Implementation Approaches  
for Your Compiler

**C/Flex/Bison Path:**

```
// Traditional symbol table
struct Symbol {
    char *name;
    Type *type;
    int scope_level;
    struct Symbol *next;
};

struct Type {
    enum { INT, FLOAT, ARRAY, STRUCT } kind;
    union {
        struct Array *array;
        struct Struct *structure;
    };
};

// Manual memory management, but complete control
```

# Semantic Analysis in Project Paths

Implementation Approaches  
for Your Compiler

Python/PLY Path:

```
# Modern Pythonic approach
@dataclass
class Symbol:
    name: str
    type: 'Type'
    scope: int
    defined_at: tuple[int, int] # line, column

@dataclass
class FunctionType:
    parameters: List[Type]
    return_type: Type

# Use dictionaries for O(1) lookup
self.symbols: Dict[str, List[Symbol]] = {}
```

# Semantic Analysis in Project Paths

Implementation Approaches  
for Your Compiler

**OCaml/LLVM Path:**

```
(* Functional, type-safe approach *)
type typ =
  | TInt | TFloat | TBool
  | TArray of typ
  | TFunction of typ list * typ

type symbol_info = {
  name: string;
  typ: typ;
  scope: int;
  is_mutable: bool;
}

type environment = symbol_info list

(* Immutable updates *)
let add_symbol env symbol = symbol :: env
```

# Common Semantic Patterns

## Recurring Problems & Solutions

### 1. Forward References:

```
class Calculator {  
    int result = add(5, 3); // Using add before declaration?  
  
    int add(int a, int b) {  
        return a + b;  
    }  
}
```

Solution: Multi-pass analysis

# Common Semantic Patterns

Recurring Problems & Solutions

## 2. Method Overloading:

```
class MathUtils {  
    int add(int a, int b) { return a + b; }  
    double add(double a, double b) { return a + b; }  
    String add(String a, String b) { return a + b; }  
}
```

Solution: Argument type matching

# Common Semantic Patterns

Recurring Problems &  
Solutions

## 3. Generic Types:

```
// TypeScript
interface Box<T> {
    value: T;
    getValue(): T;
    setValue(value: T): void;
}

let numberBox: Box<number> = { value: 42 };
numberBox.setValue("hello"); // Error: string not assignable to
number
```

Solution: Type parameter substitution

# Testing Semantic Analysis

How to Verify Your Type Checker

**Test Cases Structure:**

```
# Positive test cases (should pass)
valid_programs = [
    """
    let x: int = 5;
    let y: int = x + 10;
    """,
    """
    def add(a: int, b: int) -> int {
        return a + b;
    }
    let result = add(5, 3);
    """
]
```

```
# Negative test cases (should fail with specific errors)
invalid_programs = [
    ("let x: int = 'hello';", "TypeMismatchError"),
    ("undefined_var = 5;", "UndeclaredVariableError"),
    ("return 42;", "ReturnOutsideFunctionError")
]
```

```
def test_type_checker():  
    for program in valid_programs:  
        assert type_check(program).success == True  
  
    for program, expected_error in invalid_programs:  
        result = type_check(program)  
        assert result.success == False  
        assert expected_error in result.errors
```

Use property based testing

```
# Generate random valid programs to test
@given(lists(integers()))
def test_addition_commutative(numbers):
    program = f"""
    def test() -> bool {{
        return {numbers[0]} + {numbers[1]} == {numbers[1]} + {numbers[0]};
    }}
    """
    assert type_check(program).success == True
```

# Details for Bison/C project

File Structure  
(lecture-notes  
/8-semantic-a  
nalysis-type-c  
hecking)

```
lecture-notes/8-semantic-analysis-type-checking/  
├── lexer.l  
├── parser.y  
├── ast.h  
├── symbol.h  
├── type.h  
├── main.c  
├── ast.c  
├── type.c  
├── symbol.c  
└── Makefile
```

To compile(for step by step see Makefile)

```
$ make
```

```
$ make test
```

```
$ ./compiler any_test_file
```

# ast.h

```
#ifndef AST_H
#define AST_H

typedef enum {
    NODE_NUMBER,
    NODE_VARIABLE,
    NODE_BINARY_OP,
    NODE_ASSIGNMENT,
    NODE_DECLARATION,
    NODE_PROGRAM
} NodeType;

typedef enum {
    OP_ADD,
    OP_SUBTRACT,
    OP_MULTIPLY,
    OP_DIVIDE
} Operator;
```

```
struct ASTNode {
    NodeType type;
    int line_number;
    union {
        // For numbers
        struct {
            int value;
        } number;

        // For variables
        struct {
            char* name;
        } variable;
```

```
        // For binary operations
        struct {
            Operator op;
            struct ASTNode* left;
            struct ASTNode* right;
        } binary_op;

        // For assignments
        struct {
            char* var_name;
            struct ASTNode* value;
        } assignment;

        // For declarations
        struct {
            char* var_name;
            char* type_name;
        } declaration;

        // For program (list of statements)
        struct {
            struct ASTNode** statements;
            int statement_count;
        } program;
    } data;
};
```

# ast.h

```
// Constructor functions
struct ASTNode* create_number(int value, int line);
struct ASTNode* create_variable(char* name, int line);
struct ASTNode* create_binary_op(Operator op, struct ASTNode* left,
                                struct ASTNode* right, int line);
struct ASTNode* create_assignment(char* var_name, struct ASTNode* value, int line);
struct ASTNode* create_declaration(char* var_name, char* type_name, int line);
struct ASTNode* create_program(struct ASTNode** statements, int count);

// Utility functions
void print_ast(struct ASTNode* node, int indent);
void free_ast(struct ASTNode* node);

#endif
```

# Symbol Table (symbol.h)

```
#ifndef SYMBOL_H
#define SYMBOL_H

#include "type.h"

typedef struct Symbol {
    char* name;
    Type* type;
    int line_declared;
    int is_initialized;
    struct Symbol* next;
} Symbol;

typedef struct SymbolTable {
    Symbol* symbols;
    struct SymbolTable* parent; // For nested scopes
    int scope_level;
} SymbolTable;

// Symbol table management
SymbolTable* create_symbol_table(SymbolTable* parent);
void destroy_symbol_table(SymbolTable* table);

// Symbol operations
Symbol* add_symbol(SymbolTable* table, char* name, Type* type, int line);
Symbol* lookup_symbol(SymbolTable* table, char* name);
Symbol* lookup_symbol_current_scope(SymbolTable* table, char* name);

// Type checking helper
int is_assignable(Type* target, Type* source, int line);

#endif
```

### 3. Type System (type.h)

```
#ifndef TYPE_H
#define TYPE_H

typedef enum {
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_BOOL,
    TYPE_VOID,
    TYPE_ERROR // For error recovery
} TypeKind;

typedef struct Type {
    TypeKind kind;
    char* name; // For user-friendly error messages
} Type;
```

```
// Type constructors
Type* create_type(TypeKind kind);
Type* create_named_type(char* name);

// Type operations
int types_equal(Type* t1, Type* t2);
Type* get_expression_type(Type* left, Type* right, char* op, int line);
char* type_to_string(Type* type);

// Built-in types
extern Type* TYPE_INTEGER;
extern Type* TYPE_FLOAT;
extern Type* TYPE_BOOLEAN;
extern Type* TYPE_VOID;
extern Type* TYPE_ERROR;
#endif
```

## 4. Lexer (lexer.l)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "parser.h"

int line_number = 1;
void yyerror(const char* s);
```

```
%}
```

```
%option noyywrap
```

```
DIGIT    [0-9]
LETTER   [a-zA-Z_]
ID        {LETTER}({LETTER}|{DIGIT})*
INTEGER  {DIGIT}+
FLOAT    {DIGIT}+\. {DIGIT}*
WS       [ \t\r]
```

```
%%
{WS}      /* skip whitespace */
\n        { line_number++; }
```

```
"int"     { return TOK_INT; }
"float"    { return TOK_FLOAT; }
"bool"     { return TOK_BOOL; }
"true"     { return TOK_TRUE; }
"false"    { return TOK_FALSE; }
```

```
"+"        { return TOK_PLUS; }
"_"        { return TOK_MINUS; }
"*"        { return TOK_MULTIPLY; }
"/"        { return TOK_DIVIDE; }
"="        { return TOK_ASSIGN; }
";"        { return TOK_SEMICOLON; }
"("        { return TOK_LPAREN; }
")"        { return TOK_RPAREN; }
```

```
{INTEGER} {
    yylval.int_val = atoi(yytext);
    return TOK_INTEGER;
}

{FLOAT}    {
    yylval.float_val = atof(yytext);
    return TOK_FLOAT;
}

{ID}        {
    yylval.string_val = strdup(yytext);
    return TOK_ID;
}
<<EOF>>    { return TOK_EOF; }
.           {
    printf("Error: Invalid character
' %c ' at line %d\n", yytext[0],
line_number);
    return TOK_ERROR;
}
```

```
%%
```

## 5. Parser with Semantic Actions (parser.y)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ast.h"
#include "symbol.h"
#include "type.h"

extern int line_number;
extern int yylex();
void yyerror(const char* s);

// Global symbol table and AST
SymbolTable* global_symtab = NULL;
struct ASTNode* program_ast = NULL;

// Helper function for type creation
Type* get_type_from_token(int token);

// Other external declarations
// Other external type variables
```

```
%union {
    int int_val;
    double float_val;
    char* string_val;
    struct ASTNode* node;
    Type* type;
}

%token TOK_INT TOK_BOOL
%token TOK_TRUE TOK_FALSE
%token TOK_PLUS TOK_MINUS TOK_MULTIPLY TOK_DIVIDE
%token TOK_ASSIGN TOK_SEMICOLON
%token TOK_LPAREN TOK_RPAREN
%token TOK_ERROR

%token <int_val> TOK_INTEGER
%token <float_val> TOK_FLOAT
%token <string_val> TOK_ID

%type <node> program statements statement expression term factor
%type <node> declaration assignment
%type <type> type_specifier

%%
```

```

program:
    statement_list TOK_EOF {
        program_ast = $1;
        YYACCEPT;
    }
;

    statement_list:
        statement {
            // Create a program node with single statement
            struct ASTNode** stmts = malloc(sizeof(struct ASTNode*));
            stmts[0] = $1;
            $$ = create_program(stmts, 1);
        }
    | statement_list statement {
        // Append statement to existing program
        int old_count = $1->data.program.statement_count;
        int new_count = old_count + 1;
        struct ASTNode** new_stmts = realloc($1->data.program.statements,
                                              new_count * sizeof(struct ASTNode*));

        new_stmts[old_count] = $2;
        $1->data.program.statements = new_stmts;
        $1->data.program.statement_count = new_count;
        $$ = $1;
    }
;

```

## 5. Parser with Semantic Actions (parser.y)

```
statement:
    declaration TOK_SEMICOLON { $$ = $1; }
    | assignment TOK_SEMICOLON { $$ = $1; }
    | expression TOK_SEMICOLON { $$ = $1; }
    ;

declaration:
    type_specifier TOK_ID {
        Symbol* existing = lookup_symbol_current_scope(global_syntab, $2);
        if (existing) {
            printf("Error at line %d: Variable '%s' already declared\n",
                line_number, $2);
        } else {
            add_symbol(global_syntab, $2, $1, line_number);
        }
        $$ = create_declaration($2, $1->name, line_number);
    }
    ....
```

## 6. Main Program (main.c)

```
int main(int argc, char** argv) {
    // Initialize type system
    initialize_types();

    // Create global symbol table
    global_symtab = create_symbol_table(NULL);

    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    // Open input file
    FILE* input_file = fopen(argv[1], "r");
    if (!input_file) {
        printf("Error: Cannot open file %s\n", argv[1]);
        return 1;
    }
}
```

```
// Set flex to read from file
yyin = input_file;

printf("=== Parsing and Semantic Analysis ===\n");

// Run parser
if (yyparse() == 0) {
    printf("Parse successful!\n");

    if (program_ast) {
        printf("\n=== Abstract Syntax Tree ===\n");
        print_ast(program_ast, 0);
    }
} else {
    printf("Parse failed with errors.\n");
}

// Cleanup
if (program_ast) {
    free_ast(program_ast);
}
destroy_symbol_table(global_syntab);
fclose(input_file);

return 0;
}
```

See lecture notes example:

**8-semantic-analysis-type-checking**