Class 10
Type Checking

What?

```
x = a + b * doTask(c,d);
```

Verify that types of construct match that expected by its context

- Operands are compatible with each other
- Operands are compatible with the operator

Why?

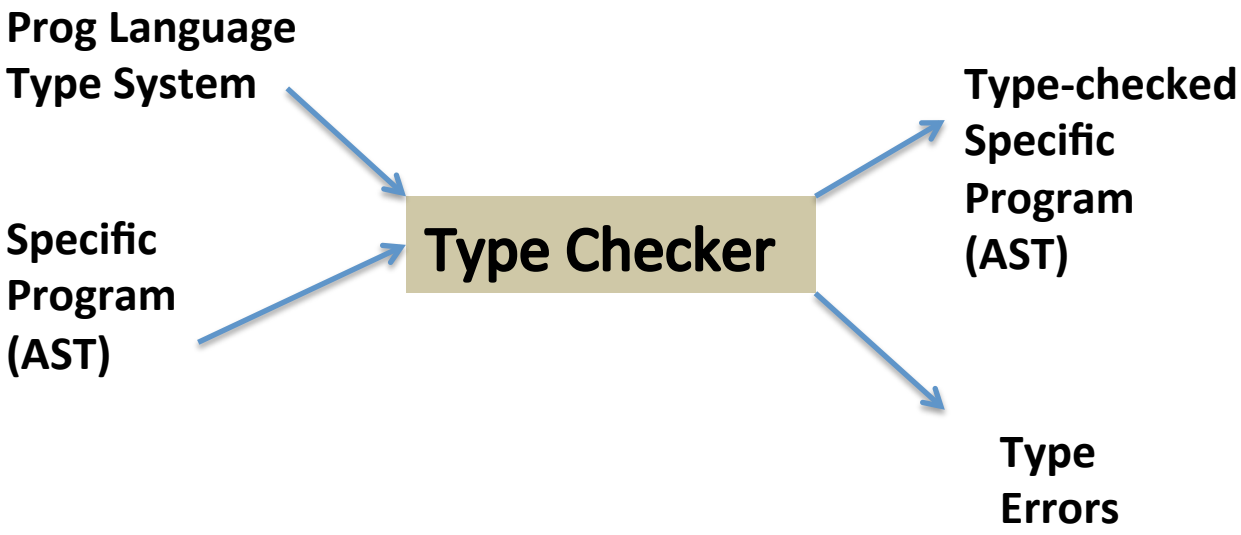
The alternative ...

→ allow operation to act on representation of value, even if it does not have a semantically well-defined result

Advantages of alternative:

Disadvantages of alternative:

How?


Prog Language
Type System
Specific
Program
(AST)
Type Checker
Type-checked
Specific
Program
(AST)
Type
Errors

Static versus Dynamic Type Checking

Static:

Dynamic:

Tradeoffs?

Static Checking

- **What does a data type determine?**

- **What is a type error? Example?**

A PL usually provides:

* Base Types
* Type constructors

How do you determine the type of an identifier?

Type Systems

- The rules governing permissible operations on types form a **type system**.

- **Strong type systems** never allow for a type error to happen at run-time unchecked. (all checked either at compile time or runtime)
- Java, Python, JavaScript, LISP, Haskell, etc.

- **Weak type systems** can allow type errors at runtime.
- C (casting any pointer type to any other pointer type), C++, perl

What about **types of intermediate values**?

```
x = a + b * doTask(c,d);
```

– Need to keep track of AND **infer** expression type from operations. Then check if matches expected type.

→Type expressions AND type rules

== PL's Type System

→ Type Checker **implements** the Type System

Defining a Type System

To formally define a type system...

– We define **axioms** and **inference rules.**

n : number Axiom N is of type number

$$\frac{e_1 : \text{number} \qquad e_2 : \text{number}}{\{+\ e_1\ e_2\} : \text{number}}$$

Inference rule

Meaning of the inference rule:

If expression e_1 has type number and expression e_2 has type number

then expression $\{+\ e_1\ e_2\}$ can be assigned type number

Example Language SIMPLE

BAE ::= true
| false
| *n* — *n* is an integer literal
| {+ *BAE BAE*}
| {< *BAE BAE*}
| {or *BAE BAE*}

Exercise!

Derive typing rules and axioms for this language!

Reminder:

n : number — Axiom

$$\frac{e_1 : \text{number} \qquad e_2 : \text{number}}{\{+\ e_1\ e_2\} : \text{number}}$$

Inference rule

How the Rules Work

Case 1: syntactically correct? Type correct?

{ < { + 3 4} { + 1 2}}

Case 2: syntactically correct? Type correct?

{+ {< 1 2} 3}

Thus, a Simple Type Checker

Type checking `E1 op E2`:

1. `TypeCheck(E1)` return inferred `type(E1)`
2. `TypeCheck(E2)` return inferred `type(E2)`
3. Type rule: Are these what are expected?
   1. `CheckCompatibility(E1, E2)`
   2. `CheckCompatibility(E1,E2,op)`
   3. Emit type errors appropriately
4. `InferType(E1 op E2)`

Type Equivalence

Suppose checking `E1` op `E2`
And
  `E1` is type int and `E2` is type subrange

Consider Pascal:

```
Type T = array[1..100] of int;
Var X,Y: array[1..100] of int;
Z: array[1..100] of int;
W: T;
A: T;
```

Are they all equivalent? Some of them? None of them?

Name vs Structural Equivalence

- **Name**: 2 names are of the same type iff they are declared together or declared using the same type name.

- **Structural:** 2 names are of the same type iff the components of their type are identical in all respects (when all names substituted out)

Comparing Name & Structural Equivalence

- Type checking effort?
- Strictness in type checking?

Consider:

```
struct {
    int: id;
    string: employee_name;
    } employee_record;


struct {
    int: zipcode;
    string: address;
    } address_record;
```

Type Rules for Function Calls

f is an identifier.
f is a non-member function in scope S.
f has type $(T_1, \ldots, T_n) \rightarrow U$

$$\frac{S \vdash e_i : T_i \text{ for } 1 \leq i \leq n}{S \vdash f(e_1, \ldots, e_n) : U}$$

Where is the type signature?
What the the checks to be done here?
Inference to be done?