## Part 5

# Type Checking

# Type Systems

- Programming languages have different types of data and objects

```
a = 42                # int
b = 4.2               # float
c = "fortytwo"        # str
d = [1,2,3]           # list
e = {'a':1,'b':2}     # dict
...
```

- Each type has different capabilities

```
>>> a - 10
32
>>> c - "ten"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'st
>>>
```

# What is a Type?

- It partly relates to data representation

```
int a = 42;
short b = 42;
long c = 42;
float d = 4.2;
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | 00 | 00 | 00 | 2a |
| | | | | | | 00 | 2a |
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 2a |
| 40 | 10 | cc | cc | cc | cc | cc | cd |

- Must map to low-level operations
- Different kinds of instructions (int vs. float)
- Also: Input/Output encoding

# Error Checking

- A type system also encodes semantic rules
- A lot of it is common sense
  - Can't perform operations (+,-,*,/) if not supported by the underlying type
  - Can't overwrite immutable data
  - Array indices must be positive integers

# Dynamic Typing

- Rules are enforced at run-time
- Objects carry their type around

```
>>> a = 42
>>> a.__class__
<class 'int'>
>>> a + 10
52
>>> a.__add__(10)
52
>>> a.__add__('hello')
NotImplemented
>>> a + 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
```

# Static Typing

- Rules are enforced at compile-time
- Code is typically annotated with type labels

```
/* C */
int fact(int n) {
    int result = 1;
    while (n > 0) {
        result *= n;
        n--;
    }
}
```

- Compiler executes a "proof of correctness"
- Types are discarded (erased) during execution

# Note

- Most programming languages involve a mix of both techniques (static/dynamic)
- Compiler checks as much as possible
- Certain checks may be forced to run-time
- Example: Array-bounds checking

# Elements of a Type System

- A Few Minimal Requirements:
  - Must be able to specify types
  - Must be able to compare types
  - Must be able to compose types
  - Must be able to check capabilities

# Type Specification

- There is a set of "primitive" given types
- They are labels that you attach to values

```
var x float;
func fact(n int) int {
    if n == 1 {
        return 1;
    } else {
        return n*fact(n-1);
    }
}
```

- Implicitly present in literals

```
42     (int)
4.2    (float)
```

# Type Comparison

- Types must be comparable

```
int != float
```

- A major part of checking is finding type-mismatches in the code (type errors)
- Common solution: compare type names
- "Nominal Typing"

# Type Composition

- Can define new types from existing types

```
struct Position {
    float x;
    float y;
};
```

- Example: Structures/Records
- Sometimes known as a "product type"
- "Product" terminology refers to the total number of possible values (all floats * all floats)

# Type Composition

- Enums

```
enum Speed {
    Stopped;
    Slow;
    Fast;
}
```

- An instance picks only <u>one</u> of the values

```
x = Speed::Stopped;
y = Speed::Slow;
```

# Type Composition

- Advanced Enums (parameterized)

```
enum Speed {
   Stopped;
   Moving(int);
}
```

- Choices may have a value attached

```
x = Speed::Stopped;
y = Speed::Moving(300);
```

- Note: Sometimes called a "sum type"

# Function Types

- Functions may also represent a type

```
func mul(x int, y int) int {
   return x * y;
}
```

- Type consists of argument types and result type

```
(int, int) -> int
```

- Note: Functions might be first-class objects just like integers, floats, etc.

```
var m = mul;
...
z = m(x, y);     # Requires x=int, y=int, z=int
```

# Type Capabilities

- Types have different capabilities (operators)

```
int:
   binary_ops = { '+', '-', '*', '/' },
   unary_ops = {'+','-'}

string:
   binary_ops = {'+'},
   unary_ops = {}
```

- A type checker will consult

```
231 * 42       # OK!
'a' * 'b'      # ERROR!
```

# Type Coercion

- There may be well-defined type conversions

```
bool -> int -> float
```

- Explicit casts

```
var x int = 42;
var y float = float(x);       // y = 42.0
```

- Implicit casts

```
3 + 2.5     -> float(3) + 2.5  -> 5.5
```

# Type Hierarchies

- There may be a concept of inheritance

```
class Parent:
    ...

class Child(Parent):
    ...
```

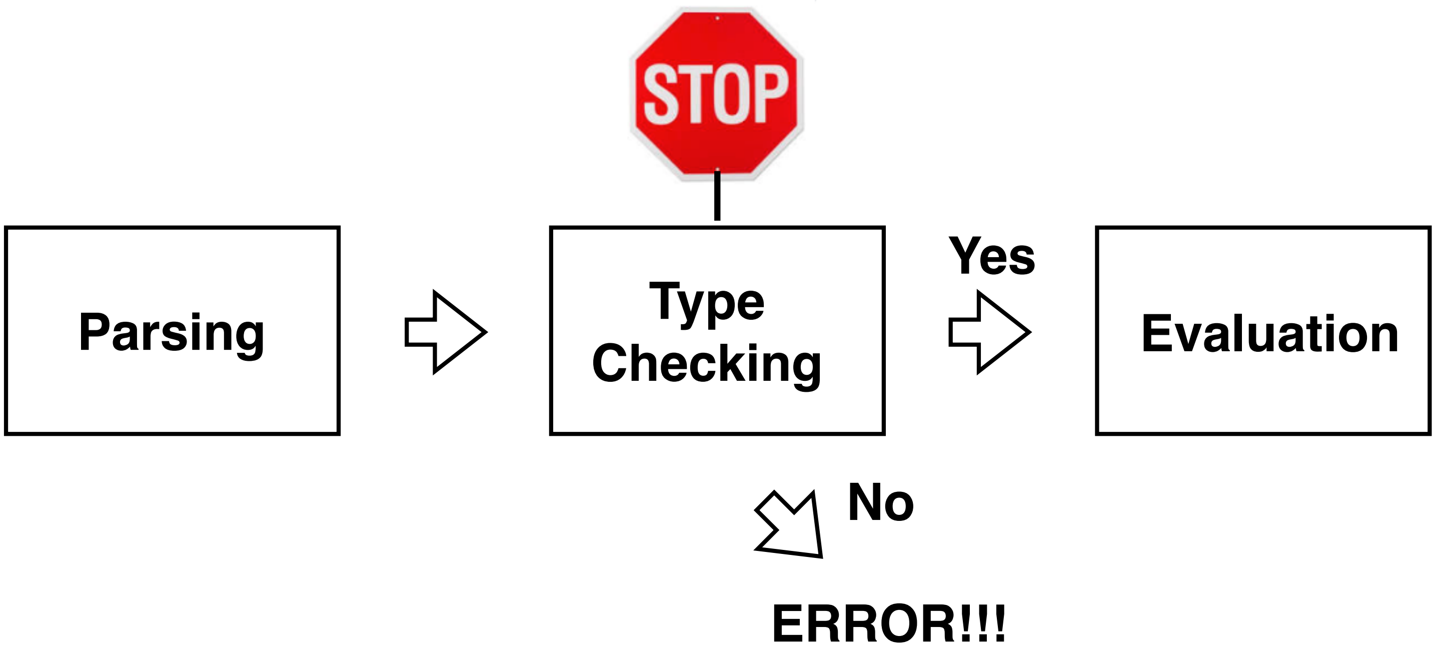
- Type "Child" is compatible with "Parent"

```
c = Child()
isinstance(c, Parent)     # --> True
```

- Note: Type ontologies are a complex subject (we're not doing this in the compiler project).

# Type Checking

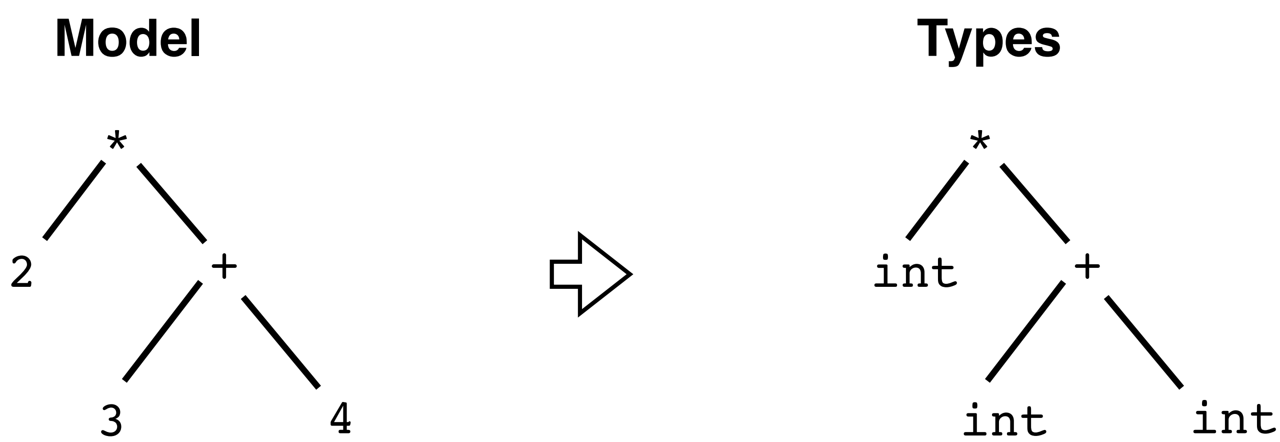
- It's a check in the middle of compilation




- Think of it as a filter:  "all systems go."

# How To Implement

- Run a type-level program simulation




- Watch how the <u>types</u> would evolve in evaluation
- Look for mismatches. Report errors.

# Example:

```
def check_integer(node, env):
    return "int"

_binops = {
    ('+', 'int', 'int') : 'int',
    ('-', 'int', 'int') : 'int',
    ('<', 'int', 'int') : 'bool',
    ...
}
def check_binop(node, env):
    left_type = check_expression(node, env)
    right_type = check_expression(node, env)

    result_type = _binops.get((node.op, left_type, right_type))
    if result_type is None:
        error("Type Error!")

    return result_type
```

# Confusion

- Type checking is NOT the same thing as running a program (that's an interpreter)
- No actual calculations are carried out
- It's at a higher level of abstraction

```
var x int;
var y int;

z = x + y;        // We know that z must be an "int"
                  // We know nothing about x, y values
```

- It's "meta". It fills in details about the program.

# Checking of Names

- Type checking also involves name checking
- Consider this expression:

```
42 + x
```

- What type is "x"? What is "x"?
- Languages usually require declarations

```
var x int;
```

- No declaration -> Name Error

# Symbol Tables

- Declarations are managed in a symbol table
- It represents the "environment"

```
env = {
        'x': Variable(type="int", ...)
      }
```

- Declarations insert definitions in the table
- Later name references consult the table

# Symbol Tables and Scopes

- You need nested symbol tables

```
var x int;          // Global

func spam() int {
    var y int;      // Locals
    var z int;
    ...
}
```

```
{ 'x': Variable('int') }
```

```
{
  'y': Variable('int')
  'z': Variable('int')
}
```

- Keep in mind: You're not running the program.
- You're tracking metadata (definitions).

# Control-Flow Analysis

- There are many common programming errors related to control-flow issues
- Often a control-flow check is performed
- In addition to type checking.
- Will illustrate some common scenarios.

# Dead Code

- There might be statements that never execute

```
while n > 0 {
    if n == 5 {
        break;
        print "Done!";    // <<<< Never executes
    }
    n = n - 1;
}
```

- Should it result in a compiler warning?

# Uninitialized Variable

- What is the value?

```
var z int;
print z;
```

- Or this...

```
var z int;
if x > 0 {
    z = 10*x;     // Only initialized on one branch
}
print z;
```

# Unused Variable

- What about this?

```
var x = 42;
var z = x + 10;   // z never reference ever again
...
<END>
```

- Does the compiler see the lack of use?
- Note: Such problems often the domain of linters/code checkers. But could be part of type-checking too.

# Project

- Find the file
  - `wabbit/typecheck.py`
- Follow instructions inside.