Type Checking

Warning ☺

- On the next slides you will see several examples of type checking and how types work in different languages
- There is no single language that does all the things that we will see here. You don't have to implement all this in the project ☺

Type Checking

- *Type Checking* = the process of verifying that the operations and statements in the source code respect the type system of the language

- **Type checking an expression in the AST usually consists of two steps**
  1. **Infer the type of the expression**
  2. **Check whether its type matches what we expected**

- The type checking can fail for two reasons:
  1. We cannot infer the type of the expression. For example because there is an unknown identifier in the expression
  2. The type of the expression doesn’t match the expected type

Examples of the two steps of type checking

- In practice you will have to infer the types of several expressions and then check
- Array access: `x[i]`
  1. Infer the type of `x` and `i`
  2. Check that `x` is an array and `i` is an integer
- Arithmetic expression: `3*a`
  1. Infer the type of 3 and `a`
  2. Check whether the multiplication supports the types
- Function call: `f(3,x)`
  1. Infer the type of 3 and `x`
  2. Check whether the types match the parameter types of `f`
- Assignment: `x[3] = a`
  1. Infer the type of `x[3]` and `a`
  2. Check whether the types of left and right side match
- Same if-statements, return statements,...

Strongly vs Weakly typed languages

- Not all languages do type checking
- Two broad classes of programming languages
  - **Strongly typed**: all variables have a declared type and all type errors can be (in theory) found by the compiler or runtime system
    - Examples: Java, Python
  - **Weakly typed**: “everything is allowed”, runtime system tries to execute program as good as possible
    - Example: Pearl

      `"20a" + 10` gives `30`
    - Example: C

      `(time*)i` interpreting an int variable as a pointer
- And there is also assembly language: **untyped** (everything is bytes)

Static vs Dynamic type checking

- For strongly typed language, type checking can be done...
  - ...at compile time: **Static typing**
    - Examples: Java, C, C++
  - ...at runtime: **Dynamic typing**
    - Examples: Python, Java (JVM performs type checks, too)
    - Requires that we keep information about the type of values and objects during runtime

- Note that type checking sometimes comes with compromises for performance reasons
  - C does not check array accesses during runtime (buffer overflows)
  - C and Java do not check for numeric overflows during integer operations (for example, when adding two very large integers)

How to represent types during semantic analysis

- During the semantic analysis, we can represent types with tree-like structures, similar to expressions
- Possible implementation:

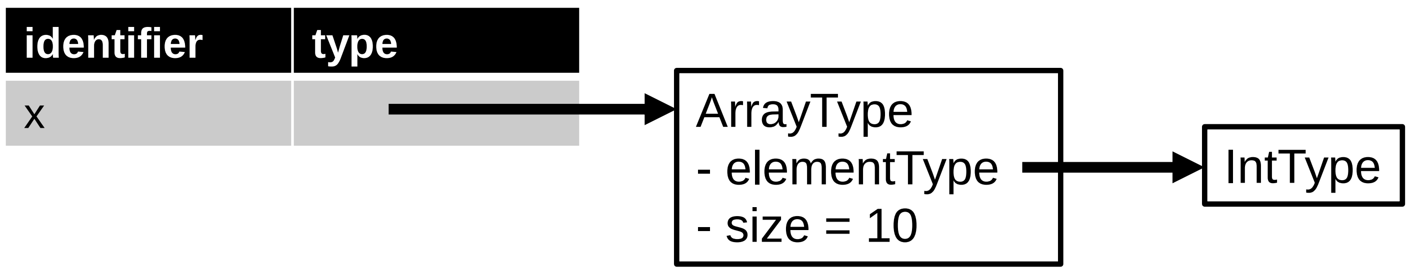
```
abstract class Type { String name; }

class IntType extends Type { }

class StringType extends Type { }

class ArrayType extends Type {
    Type elementType;
    int size;
}

class RecordType extends Type {
    Map<String,Type> fields;
}
```

Example C

- For the declaration

```
int x[10]
```

the entry in the symbol table would look like this



User-defined types

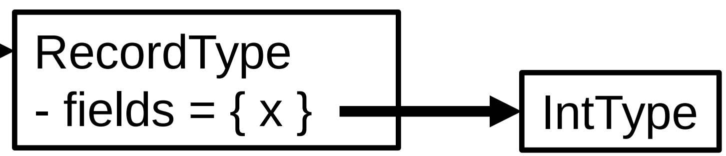
- In many languages, the programmer can define new types, e.g.

```
struct MyNewType {
    int x;
};
```

- We also have to keep a table of new types during the semantic analysis, similar to the symbol table for variables and functions
- For example:

This is not the symbol table

identifier	type
MyNewType	



- This means that we also have to perform some kind of type checking on type declarations:
  - Do the types used in type declarations exist?
  - No duplicate type names?
  - No duplicate field names in records?

Names of variables vs names of types

- Having different tables for variables/functions (the symbol table) and user defined types raises a practical question:

  Do we allow that a variable has the same name as a type?

- In Java, this is indeed allowed:

```
List List;  // variable List of type List
```

- Some programming languages do not separate variables and types and they allow to use types in expressions:

```
List x = new List(Float); // the type Float is used as an
                          // argument (not possible in Java)
```

Obviously, such languages do not allow that a variable has the same name as a type

Type Equivalence

Type checking an expression in the AST usually consists of two steps

1. *Infer the type of the expression*
2. *Check whether the type matches what we expected*

- Let's first look at step 2. So far, we assumed that it is easy to do
  For example, when type-checking an assignment *s*:

```
var leftType = getTypeOfExpression(s.leftSide);
var rightType = getTypeOfExpression(s.rightSide);
if(!leftType.equals(rightType))
    throw new TypeErrorException();
```

- This works well for basic types like int, double, etc.:

```
double x;
x = 123.2;  // Easy! x is a double and 123.2 is a double
```

Implicit type conversion

- Many programming languages define automatic type conversions to make life easier for the programmer
- In Java:

```
double x = 123; // not identical types, but still works
                // because of implicit type conversion
                // in Java
```

- More examples:
  - C, Java: `3 + 4.5` (integer+double) is converted to (float+double)
  - Java: `“Hello” + 3` (string+integer) gives a string
- In C++ you can even define your own conversions
- Some languages (Ada, Pascal) are very strict and have almost no automatic conversion

Type Equivalence for compound types

- Let’s consider this situation in C:

```
struct A { int a; }
struct B { int b; }
A x;
B y;
```

- Do we allow this assignment?

```
x = y;
```

- Two possible answers:
  - No, because A and B are types with different names.
    This is called *named equivalence*
  - Yes, because A and B have the same structure.
    This is called *structural equivalence*

- Again, this is a design decision of the language designer

Type compatibility

- In addition to type equivalence, many programming languages have the concept of *type compatibility*
  - We have already seen implicit type conversion that makes integers "compatible" to doubles in Java
  - There are also *sub-types*
    - In Java, the class B is a subtype of class A:

      ```
      class B extends A { ...}
      ```

      For that reason, this statement is correct:

      ```
      A x = new B();
      ```

    - In C, you have enums:

      ```
      enum week {Mon, Tue, Wed, Thur, Fri, Sat, Sun};
      ```

      which are seen as subtypes of integers (Mon=0, Tue=1,...)