Languages and Compilers
Type Checking and Type Inference

Adam Sampson
School of Design and Informatics
Abertay University

The problem

- When we do **semantic analysis** of our program – checking that the meaning is correct – we'll need to check that the types in the program are used consistently
  - e.g. we should complain at `int x = “hello”;`
- To do this, the compiler must reason about the types being used in the program
- … and, in particular, about the types of **expressions**

Type-checking a variable initialisation

```
int end = (height / num) * (i + 1);
```

Type-checking a variable initialisation

```
int end = (height / num) * (i + 1);
```

The type of the variable...

… must match the type of the expression

Type-checking a variable initialisation

```
int end = (height / num) * (i + 1);
```

The type of the variable...

… must match the type of the expression

But what is the type of the expression?
We must **infer** it...

Inferring the type

```
(height / num) * (i + 1)
```

- Start with the simplest parts of the expression (the **terminal symbols**), and work outwards
- `1` is a constant, of type `int`
- `height`, `num`, `i` are all variables, which must have been declared before...
- … so we can look up their types in the **symbol table** – let's say that `height` is a float, and `num` and `i` are ints

Inferring the type

```
(height / num) * (i + 1)
 float     int     int int
```

- Now we can tell a bit more, based on the typing rules for the operators `/` and `+`
- If you add an int to an int, you get an int
- If you divide a float by an int, you get a float

Inferring the type

```
(height / num) * (i + 1)
 float    int     int int
     float          int
```

- And again – if you multiply a float by an int, you get a float

Inferring the type

$$\underbrace{\underbrace{(\underbrace{\texttt{height}}_{\textit{float}}\ \texttt{/}\ \underbrace{\texttt{num}}_{\textit{int}})}_{\textit{float}}\ \texttt{*}\ \underbrace{(\underbrace{\texttt{i}}_{\textit{int}}\ \texttt{+}\ \underbrace{\texttt{1}}_{\textit{int}})}_{\textit{int}}}_{\textit{float}}$$

- So the type of this expression is `float`

Inferring the type

```
int end = (height / num) * (i + 1);
```

$\underbrace{\texttt{int end}}_{\textit{int}} = \underbrace{\texttt{(height / num) * (i + 1)}}_{\textit{float}};$

- The types on the left- and right-hand sides of the assignment don't match – so we've found a type error
- We say that the program is **not well-typed**

Syntax and typechecking

```
(height / num) * (i + 1)
  float    int     int int
      float           int
             float
```

- Note the order in which we made type decisions here: it's the same as the order of evaluation...
- … and the same as the order in which we completed **expanding rules** while parsing (remember derivation sequences in Practical 2)

Syntax and typechecking

```
(height / num) * (i + 1)
 float    int     int int
     float           int
            float
```

- So it would actually be possible to infer the type of an expression as it's parsed – each time you get to the end of a production, note down what the corresponding type is
- This is what a **single-pass** compiler would do

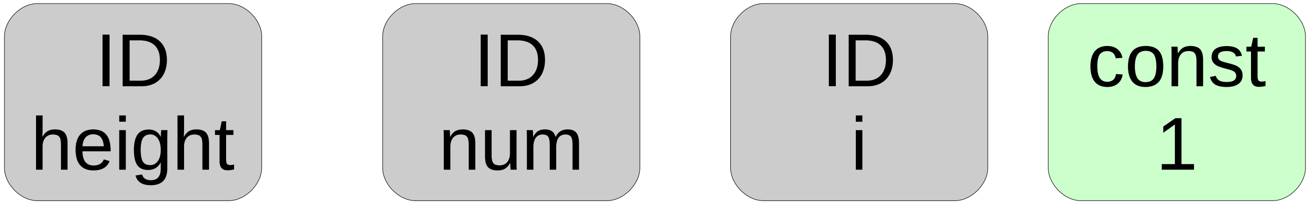
The AST

```
int end = (height / num) * (i + 1);
```

- It's more common, though, for the parser to build a data structure representing the program: an **abstract syntax tree (AST)**

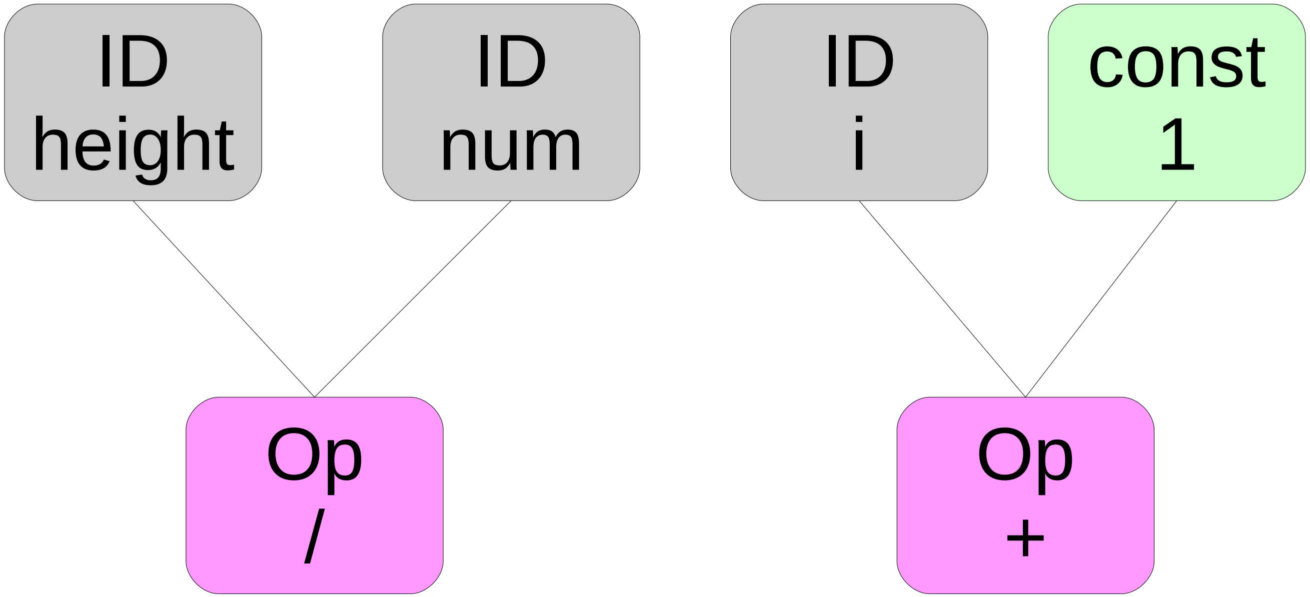
The AST

```
int end = (height / num) * (i + 1);
```



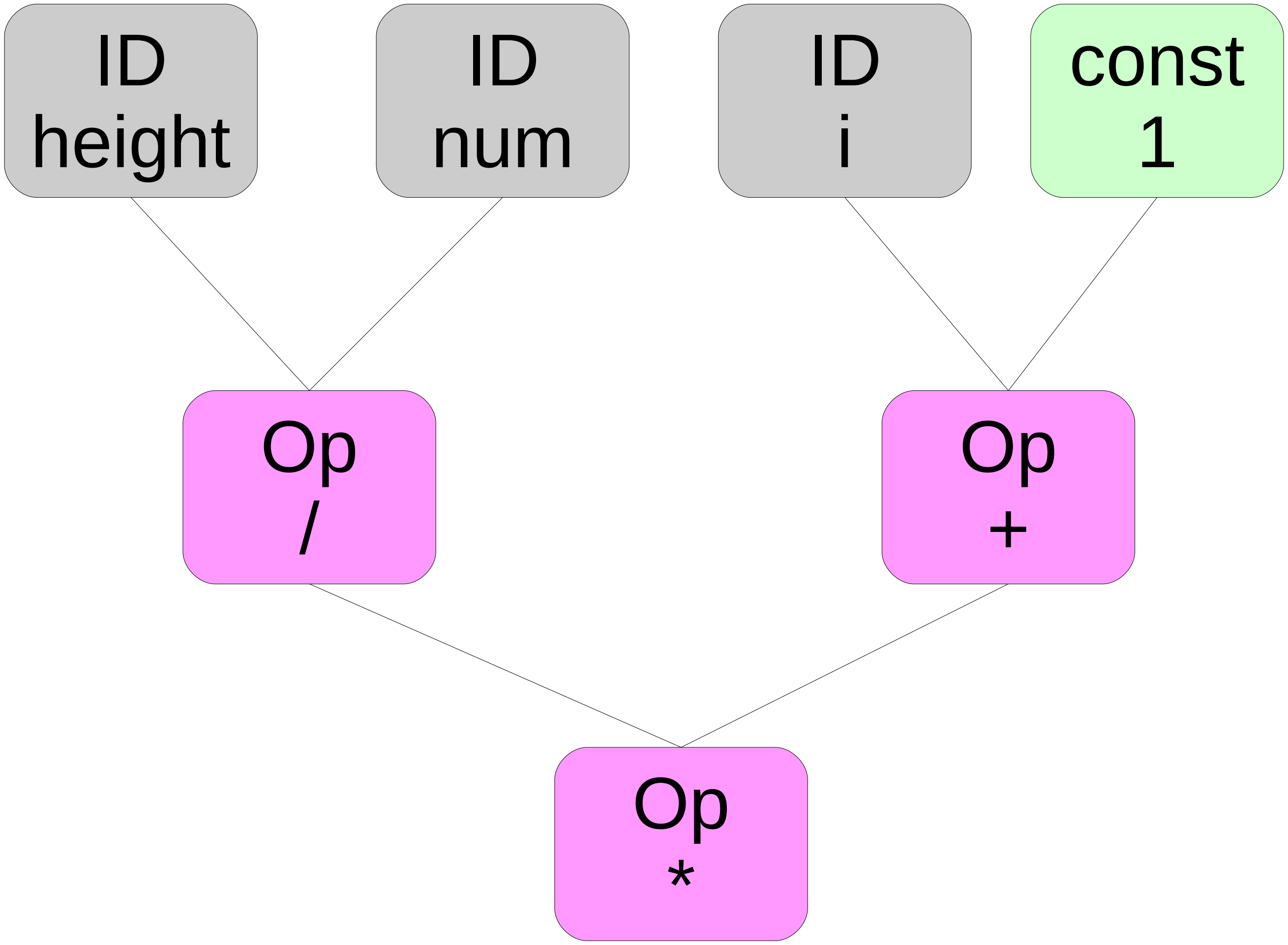
The AST

```
int end = (height / num) * (i + 1);
```



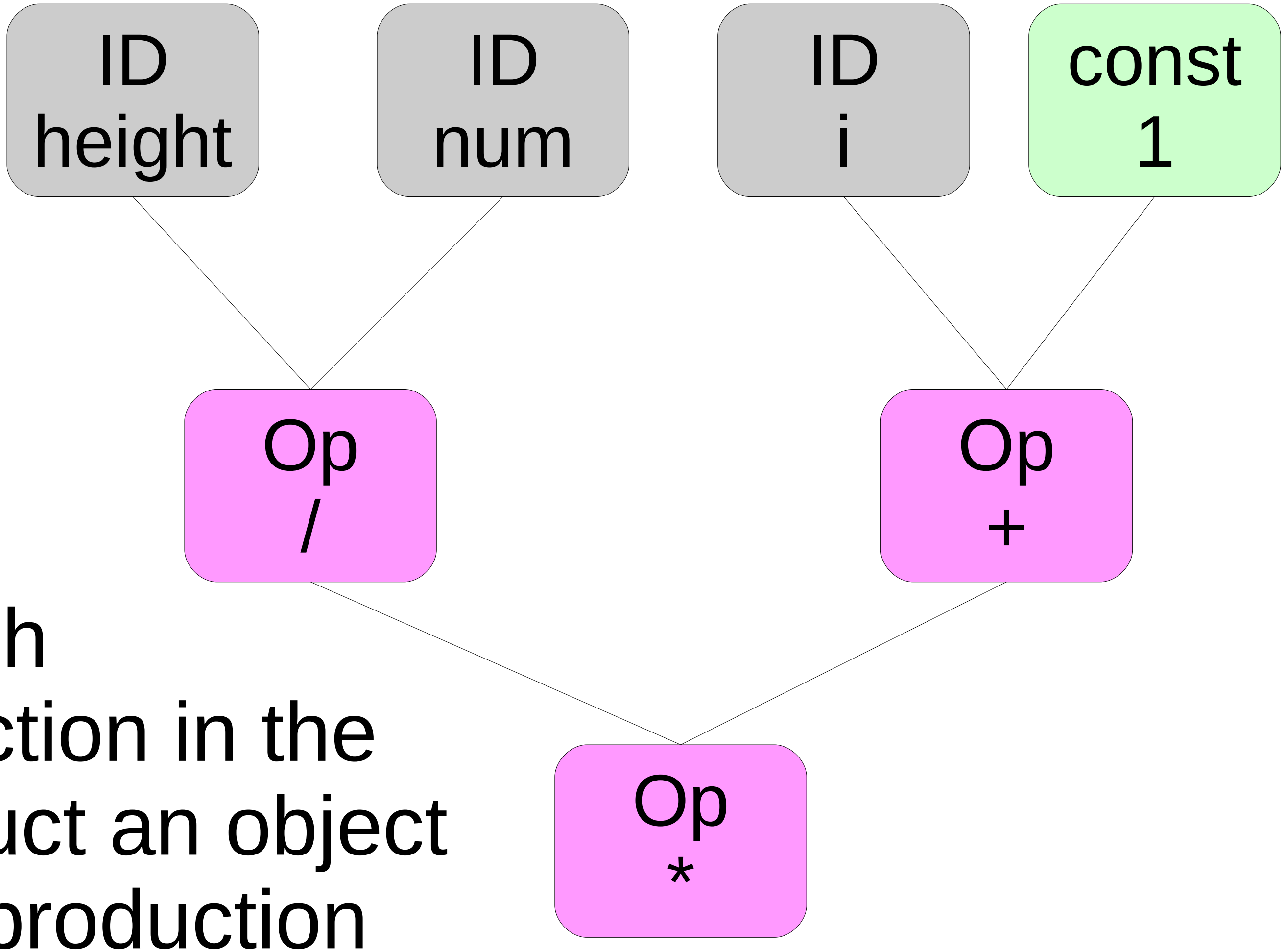
The AST

```
int end = (height / num) * (i + 1);
```



The AST

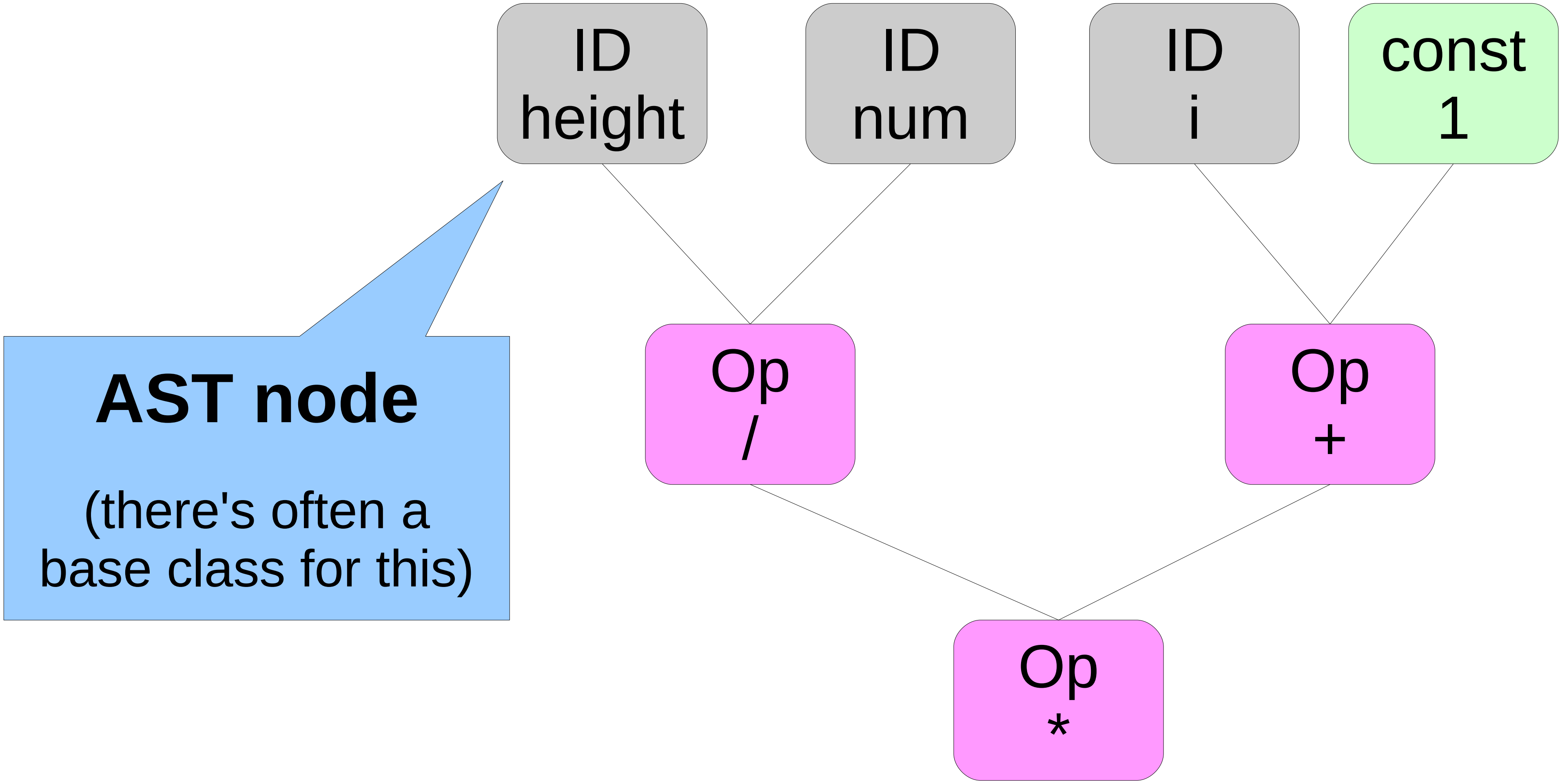
```
int end = (height / num) * (i + 1);
```



Each time we finish matching a production in the syntax, we construct an object representing that production (hence “syntax tree”)

The AST

```
int end = (height / num) * (i + 1);
```



Inferring types from the AST

```
abstract class ASTNode {
  Type getType();
}
class PlusOp extends ASTNode {
  ASTNode left;
  ASTNode right;
  Type getType() {
    Type lt = left.getType();
    Type rt = right.getType();
    if (lt == rt) return lt;
    throw new TypeError();
  }
}
```

Inferring types from the AST

- Alternatively: we could have a `Type` field in each `ASTNode` object, and recurse through the tree filling in the `Types`
- This would be a **typechecking pass**
-
```
class AssignOp extends ASTNode {
  Variable lhs;  // LHS variable
  ASTNode rhs;   // RHS expression
  void checkTypes() {
    rhs.checkTypes();
    if (lhs.type != rhs.type)
      throw …;
  }
}
```

Why it's not that simple, part 1

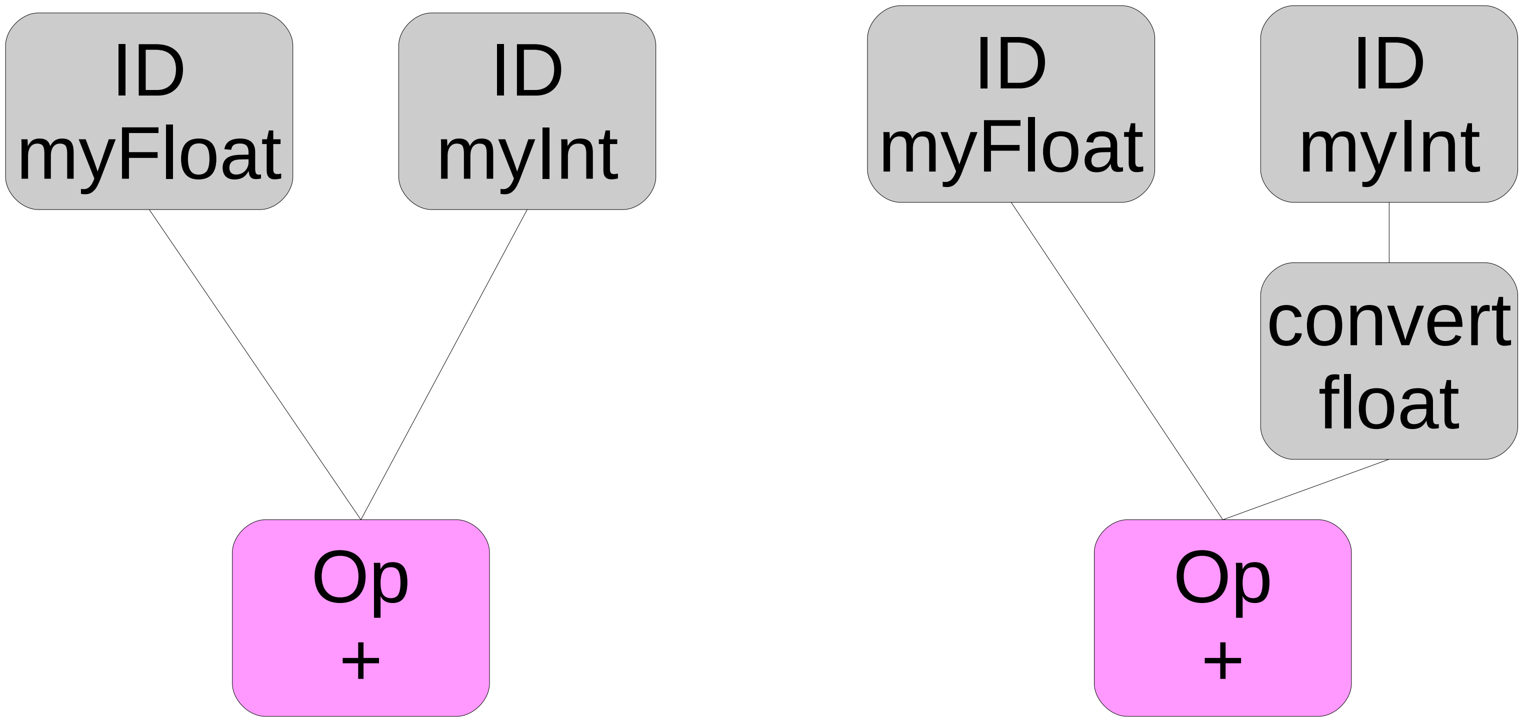
- In practice, you normally want to allow some kinds of implicit type conversion in a language
- e.g. supplying an `int` where a `float` was expected, or a `Dog` to a function that takes an `Animal`
- You then need to **convert** the `int` to the expected type...
- ... which might involve rewriting the AST, as if the programmer had included an explicit conversion (so that the code generator knows to perform the conversion)

Inserting conversions

… `myFloat + myInt` …

rewritten to:

… `myFloat + intToFloat(myInt)` …



Type inference and language design

- If you assume that the compiler is able to figure out the types of expressions, you can take advantage of this when designing a language
- C++98 is a famously verbose language:

```
vector<string>::iterator it =
   myNameList.begin();
```

- In C++11, we can write:

```
auto it = myNameList.begin();
```

- `auto` means “figure this type out automatically using type inference” – for initialisations this is always possible

More complex type inference

- C++11's type inference has limitations, though
- I can't write:

```
int myFunction() {
    auto a;            // error here
    cin >> a;
    return a + 42;
}
```

- … because the compiler can't figure out the type of a – despite there being plenty of “clues”...
- To handle this, you need a smarter type inference approach based on constraints

Why it's not that simple, part 2

- Sometimes the result of inference is not going to be a single fixed type (a **monotype**)
- For example, here's a very useful function:

```
def map(func, list):
  return [func(x) for x in list]
```

- What's the type of `map`? We can give it any function and list, provided the function can take the items in the list
- In Haskell we would describe this type as:

```
map :: (a → b) → [a] → [b]
```

- a and b are **type variables**

Hindley-Milner

- Both of those problems are solved by using a **Hindley-Milner type system**
- H-M type systems provide an efficient way of inferring generic types (types with type variables), by reasoning about the whole expression
- Loosely, you start with constraints on how each variable is used, and repeatedly unify the constraints until either you have a consistent type (success) or a contradiction (failure)

- I don't expect you to know how H-M works for the exam – we'll use an H-M language in practice next week!

Any questions?

- You need some kind of type inference in order to do typechecking...
- Many statically-typed languages use type inference to figure out the types of variables/functions/etc. automatically
- Languages with generic types need more complex type inference approaches...
- … and this may have a cost in terms of clarity of error messages! This is an argument for C++ only having limited type inference...
- **Next week**: implementing syntactic analysis