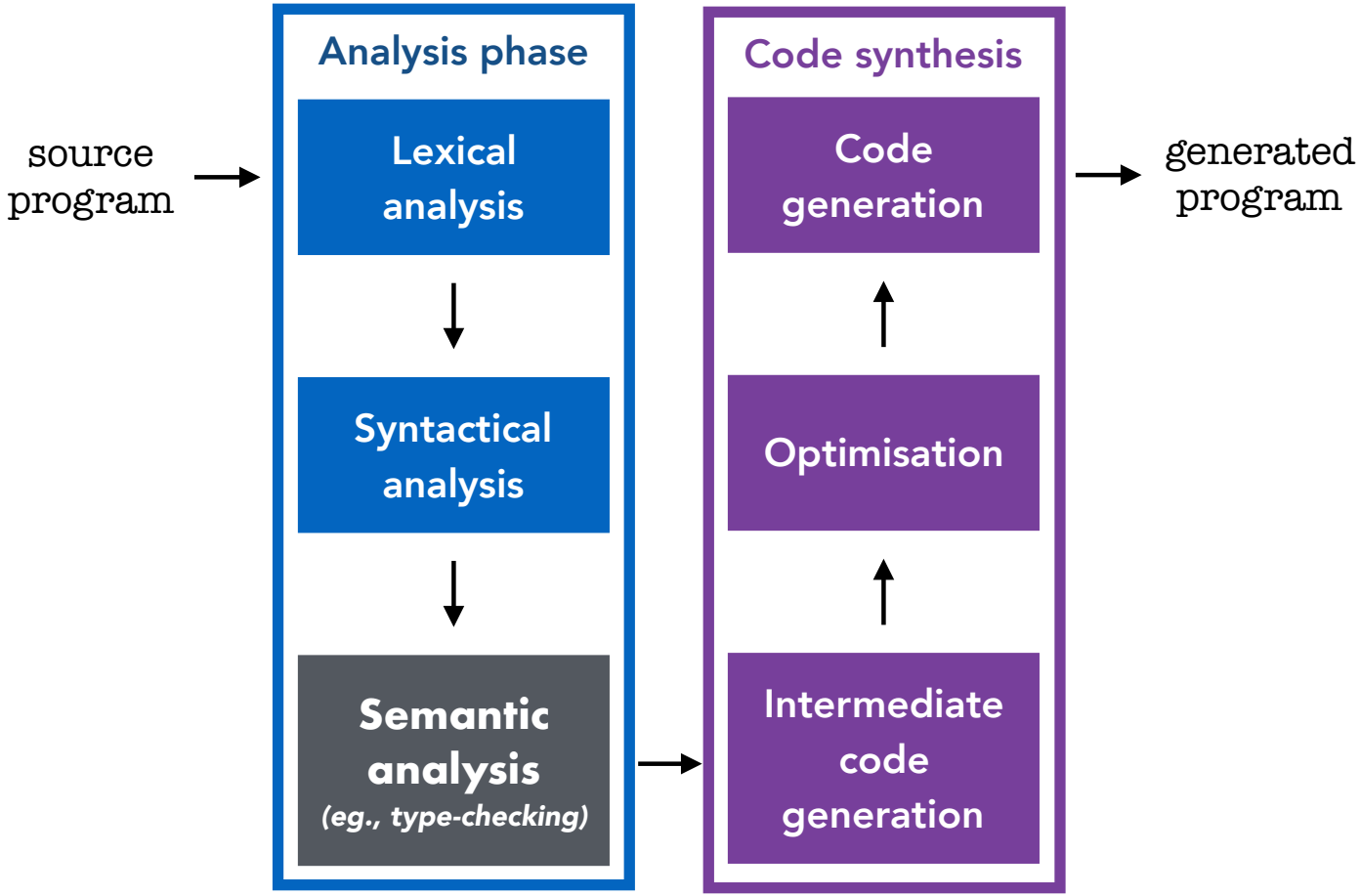
Compilers and computer architecture

9. Type-checking

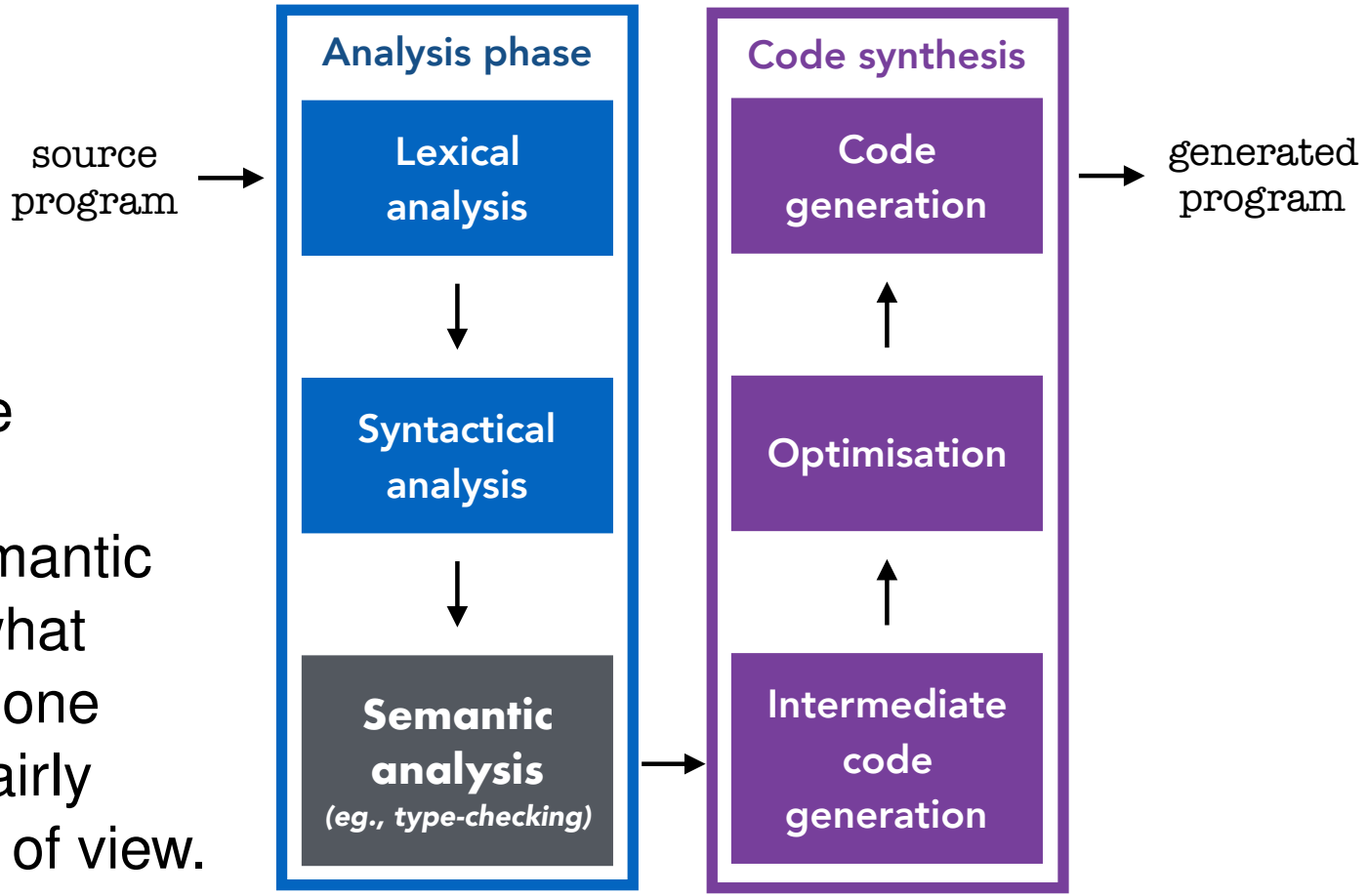
Ian Mackie

Autumn 2022

What are we up to now?

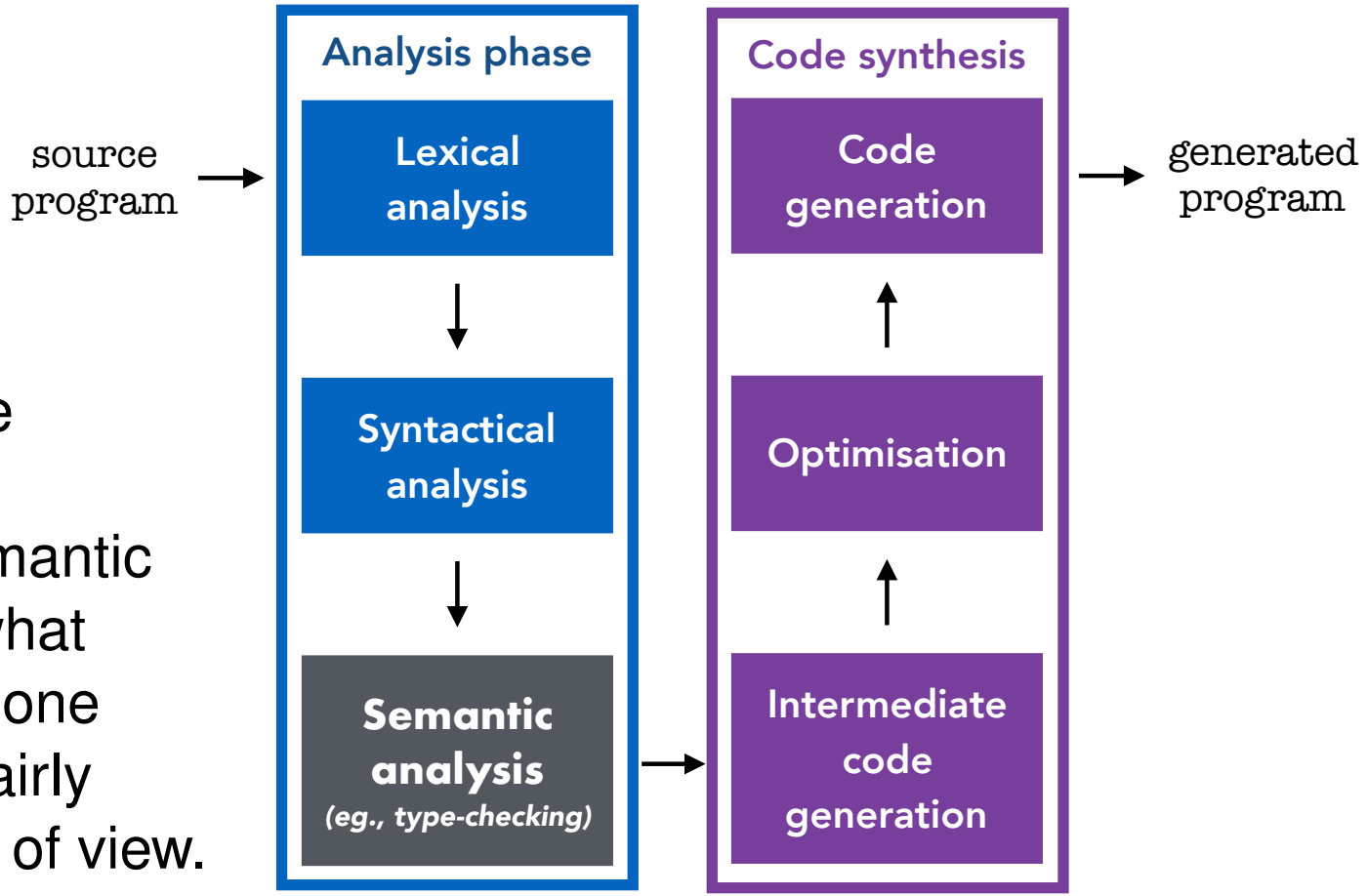


What are we up to now?



Last lecture, we considered the **purpose** of semantic analysis, and what things can be done with it, from a fairly high-level point of view.

What are we up to now?



Last lecture, we considered the **purpose** of semantic analysis, and what things can be done with it, from a fairly high-level point of view.

This lecture, we look at a single extended example — to see something of how semantic analysis is **actually done.**

Plan for this lecture

- Present a simple programming language —
  but one that is rich enough to actually be useful
- Describe a type system for this language
- Present some theory to help us describe the type checking
  (and a bit of other semantic analysis)
- Sketch how to realise semantic analysis for this language

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{aligned}
P \rightarrow\ & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} \\
\mid\ & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P \\
\mid\ & \texttt{if } P \texttt{ then } P \texttt{ else } P && \text{// if statements} \\
\mid\ & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} && \text{// 'scoped' var. decl.}
\end{aligned}
$$

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{aligned}
P \rightarrow\ & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} \\
\mid\ & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P \\
\mid\ & \texttt{if } P \texttt{ then } P \texttt{ else } P && \text{// if statements} \\
\mid\ & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} && \text{// 'scoped' var. decl.} \\
\text{EXPR} \rightarrow\ & \text{NUM} \mid \text{IDFR} \mid P \texttt{ + } P \mid P \texttt{ * } P && \text{// integer expressions}
\end{aligned}
$$

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{aligned}
P \to\ & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} \\
\mid\ & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P \\
\mid\ & \texttt{if } P \texttt{ then } P \texttt{ else } P && \text{// if statements} \\
\mid\ & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} && \text{// 'scoped' var. decl.} \\
\text{EXPR} \to\ & \text{NUM} \mid \text{IDFR} \mid P \texttt{ + } P \mid P \texttt{ * } P && \text{// integer expressions} \\
\text{COND} \to\ & \texttt{true} \mid \texttt{false} \mid P \texttt{ = } P \mid P \texttt{ < } P && \text{// boolean expressions}
\end{aligned}
$$

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{array}{rll}
P \to & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} & \\
\mid & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P & \\
\mid & \texttt{if } P \texttt{ then } P \texttt{ else } P & \text{// if statements} \\
\mid & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} & \text{// 'scoped' var. decl.} \\
\text{EXPR} \to & \text{NUM} \mid \text{IDFR} \mid P \texttt{ + } P \mid P \texttt{ * } P & \text{// integer expressions} \\
\text{COND} \to & \texttt{true} \mid \texttt{false} \mid P \texttt{ = } P \mid P \texttt{ < } P & \text{// boolean expressions} \\
\text{LIST} \to & \texttt{new List<} \alpha \texttt{>} & \text{// generic lists} \\
\mid & P\texttt{.append(}P\texttt{)} \mid P\texttt{.get(}P\texttt{)} &
\end{array}
$$

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{array}{rll}
P \to & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} & \\
\mid & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P & \\
\mid & \texttt{if } P \texttt{ then } P \texttt{ else } P & \text{// if statements} \\
\mid & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} & \text{// 'scoped' var. decl.} \\
\text{EXPR} \to & \text{NUM} \mid \text{IDFR} \mid P \texttt{ + } P \mid P \texttt{ * } P & \text{// integer expressions} \\
\text{COND} \to & \texttt{true} \mid \texttt{false} \mid P \texttt{ = } P \mid P \texttt{ < } P & \text{// boolean expressions} \\
\text{LIST} \to & \texttt{new List<} \alpha \texttt{>} & \text{// generic lists} \\
\mid & P\texttt{.append(}P\texttt{)} \mid P\texttt{.get(}P\texttt{)} & \\
\text{LOOP} \to & \texttt{for } \text{IDFR} \texttt{ = } P \texttt{ to } P \texttt{ \{ } P \texttt{ \}} & \text{// for and while loops} \\
\mid & \texttt{while } P \texttt{ do \{ } P \texttt{ \}} &
\end{array}
$$

Case study: a simple imperative language

Let us look at a simple programming language. Here's it's CFG:

. . . where IDFR ranges over identifiers, NUM over integers, and α over types (see below).

$$
\begin{array}{rll}
P \to & \text{EXPR} \mid \text{COND} \mid \text{LIST} \mid \text{LOOP} & \\
\mid & \text{IDFR} \texttt{ := } P \mid P \texttt{ ; } P & \\
\mid & \texttt{if } P \texttt{ then } P \texttt{ else } P & \text{// if statements} \\
\mid & \texttt{let } \text{IDFR} \texttt{ : } \alpha \texttt{ = } P \texttt{ in \{ } P \texttt{ \}} & \text{// 'scoped' var. decl.} \\
\text{EXPR} \to & \text{NUM} \mid \text{IDFR} \mid P \texttt{ + } P \mid P \texttt{ * } P & \text{// integer expressions} \\
\text{COND} \to & \texttt{true} \mid \texttt{false} \mid P \texttt{ = } P \mid P \texttt{ < } P & \text{// boolean expressions} \\
\text{LIST} \to & \texttt{new List<} \alpha \texttt{>} & \text{// generic lists} \\
\mid & P\texttt{.append(}P\texttt{)} \mid P\texttt{.get(}P\texttt{)} & \\
\text{LOOP} \to & \texttt{for } \text{IDFR} \texttt{ = } P \texttt{ to } P \texttt{ \{ } P \texttt{ \}} & \text{// for and while loops} \\
\mid & \texttt{while } P \texttt{ do \{ } P \texttt{ \}} &
\end{array}
$$

- A Java snippet `int x = 3; x = x+1; ···` translates as

  `let x : Int = 3 in { x := x+1 ; ··· }`

Types in our simple imperative language

We wish to distinguish between boolean values and integer values — so, disallowing expressions such as `true + 7`.

Types in our simple imperative language

We wish to distinguish between boolean values and integer values — so, disallowing expressions such as `true + 7`. We use types for this purpose, and enforce them during semantic analysis.

Types in our simple imperative language

We wish to distinguish between boolean values and integer values — so, disallowing expressions such as `true + 7`. We use types for this purpose, and enforce them during semantic analysis.

We define the types as follows:

$$\alpha \;\rightarrow\; \texttt{Unit} \;\big|\; \texttt{Int} \;\big|\; \texttt{Bool} \;\big|\; \texttt{List<}\alpha\texttt{>}$$

Types in our simple imperative language

We wish to distinguish between boolean values and integer values — so, disallowing expressions such as `true + 7`. We use types for this purpose, and enforce them during semantic analysis.

We define the types as follows:

$$\alpha \;\rightarrow\; \texttt{Unit} \;\big|\; \texttt{Int} \;\big|\; \texttt{Bool} \;\big|\; \texttt{List<}\alpha\texttt{>}$$

- `Unit` is the type of statements (like `void` in Java).
- `Int` is the type of integers and integer expressions.
- `Bool` is the type of booleans and integer comparisons.
- `List<`α`>` is the type of lists storing things of type α.

Types in our simple imperative language

We wish to distinguish between boolean values and integer values — so, disallowing expressions such as `true + 7`. We use types for this purpose, and enforce them during semantic analysis.

We define the types as follows:

$$\alpha \;\rightarrow\; \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{List<}\alpha\texttt{>}$$

- `Unit` is the type of statements (like `void` in Java).
- `Int` is the type of integers and integer expressions.
- `Bool` is the type of booleans and integer comparisons.
- `List<`α`>` is the type of lists storing things of type α.

So, `List<Int> a = new List<Int>(); a.append(2); ···`
in Java translates to

```
let a:List<Int> = new List<Int> in { a.append(2) ; ··· }
```

Types in our simple imperative language

Let's look at some examples.

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type ...

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type ... `Int`

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type ... `Int`
- `3 + x` has type...

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type ... `Int`
- `3 + x` has type... hold on a minute, what's `x`?

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type . . . `Int`
- `3 + x` has type. . . hold on a minute, what's `x`?

  — a **free variable:** undeclared (not bound) in this scope. The type of `3 + x` cannot be determined from the snippet alone.

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type . . . `Int`
- `3 + x` has type. . . hold on a minute, what's `x`?

  — a **free variable:** undeclared (not bound) in this scope. The type of `3 + x` cannot be determined from the snippet alone.

  - If `x` was declared as an `Int`, then `3 + x` has type `Int`
  - If `x` was declared as a `Bool`, then `3 + x` **has no type**
  - And if `x` was not declared at all?

Types in our simple imperative language

Let's look at some examples.

- `3` has type `Int`
- `true` has type `Bool`
- `3 + 4` has type . . . `Int`
- `3 + x` has type. . . hold on a minute, what's `x`?

  — a **free variable:** undeclared (not bound) in this scope. The type of `3 + x` cannot be determined from the snippet alone.

  - If `x` was declared as an `Int`, then `3 + x` has type `Int`
  - If `x` was declared as a `Bool`, then `3 + x` **has no type**
  - And if `x` was not declared at all? — again, **no type.**

Variable declarations
in the simple imperative language

An executable program has no free variables,
since **all variables have to be declared before use.**

If all variables have to be declared before use, why do we have to worry about programs with free variables at all?
(The programs we run don't have free variables!)

Variable declarations
in the simple imperative language

An executable program has no free variables,
since **all variables have to be declared before use.**

If all variables have to be declared before use, why do we have to worry about programs with free variables at all?
(The programs we run don't have free variables!)

Because: we want to type-check in a **compositional** way — determining types of a program from the types of its components.

Variable declarations
in the simple imperative language

An executable program has no free variables,
since **all variables have to be declared before use.**

If all variables have to be declared before use, why do we have to worry about programs with free variables at all?
(The programs we run don't have free variables!)

Because: we want to type-check in a **compositional** way — determining types of a program from the types of its components.

For our simple language, the key construct is

$$\texttt{let x : } \alpha \texttt{ = } P \texttt{ in \{ } Q \texttt{ \}}.$$

In type-checking this construct, we use the information that `x` is of type α, to type-check Q (*by itself*).

Storing type information for type-checking

Assume `y` has type `Int`. Under this assumption, the two programs

```
y := 7
```

```
let x : Int = y + 1 in { x := x+2 }
```

are well-typed, and both have type `Unit`.

Storing type information for type-checking

Assume `y` has type `Int`. Under this assumption, the two programs

```
y := 7
```

```
let x : Int = y + 1 in { x := x+2 }
```

are well-typed, and both have type `Unit`.

But — we have to type-check them, using **premises** about the types of free variables.

Storing type information for type-checking

Assume `y` has type `Int`. Under this assumption, the two programs

```
y := 7
```

```
let x : Int = y + 1 in { x := x+2 }
```

are well-typed, and both have type `Unit`.

But — we have to type-check them, using **premises** about the types of free variables.

To do this, we are going to need to **store** this type — in an 'assumption store' (called a **symbol table** or **environment**).

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

1. Initialise symbol table.

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

1. Initialise symbol table.
2. Type-check each statement:

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

1. Initialise symbol table.
2. Type-check each statement:
    2.1 Use symbol table as a reference for types of identifiers.

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

1. Initialise symbol table.
2. Type-check each statement:
   - 2.1 Use symbol table as a reference for types of identifiers.
   - 2.2 Add type information to symbol table, as identifiers are declared.

High-level approach to type-checking
in the simple imperative language

How does the compiler go about type-checking?

There are different approaches, but for single-pass type-checkers the basic approach is:

1. Initialise symbol table.
2. Type-check each statement:
   - 2.1 Use symbol table as a reference for types of identifiers.
   - 2.2 Add type information to symbol table, as identifiers are declared.

To dig deeper, let's formalise the way that we type-check statements, making use of the symbol table.

Formal type inference

Even for **type-checking** (as opposed to *type inference)*, a concept of “inference” is useful to us here.

Formal type inference

Even for **type-checking** (as opposed to *type inference)*,
a concept of “inference” is useful to us here.

We introduce some notation.

- Γ (“Gamma”): represents the symbol table, considered as a **collection of premisses** about types of identifiers

Formal type inference

Even for **type-checking** (as opposed to *type inference)*, a concept of "inference" is useful to us here.

We introduce some notation.

- Γ ("Gamma"): represents the symbol table, considered as a **collection of premisses** about types of identifiers
- We write $\Gamma(x) = \alpha$ to indicate that Γ stores the premise that `x` has type α.

  — Not unlike a function: pass an identifier name `x` to Γ, to get
  - a type α, if Γ contains a premise that `x` has type α; or
  - an error, if Γ has no premisses about `x`.

Formal type inference

Even for **type-checking** (as opposed to *type inference)*,
a concept of "inference" is useful to us here.

We introduce some notation.

- Γ ("Gamma"): represents the symbol table, considered as a **collection of premisses** about types of identifiers
- We write $\Gamma(x) = \alpha$ to indicate that Γ stores the premise that `x` has type α.

  — Not unlike a function: pass an identifier name `x` to Γ, to get
  - a type α, if Γ contains a premise that `x` has type α; or
  - an error, if Γ has no premisses about `x`.
- We write $\Gamma \vdash P : \alpha$ meaning that program P has type α, *under the premisses* given by Γ.

Formal type inference

Even for **type-checking** (as opposed to *type inference)*, a concept of "inference" is useful to us here.

We introduce some notation.

- Γ ("Gamma"): represents the symbol table, considered as a **collection of premisses** about types of identifiers
- We write $\Gamma(x) = \alpha$ to indicate that Γ stores the premise that `x` has type α.

  — Not unlike a function: pass an identifier name `x` to Γ, to get
  - a type α, if Γ contains a premise that `x` has type α; or
  - an error, if Γ has no premisses about `x`.
- We write $\Gamma \vdash P : \alpha$ meaning that program P has type α, *under the premisses* given by Γ.
  - We sometimes use **additional** premisses, such as `y` : β . We write Γ, `y` : $\beta \vdash P : \alpha$ in this case.

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

  These remain true in the presence of a symbol table: for instance, we also have $\Gamma \vdash$ `7` : `Int` for any Γ.

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

  These remain true in the presence of a symbol table: for instance, we also have $\Gamma \vdash$ `7` : `Int` for any Γ.
- The role of Γ comes in for more complicated inferences:

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

  These remain true in the presence of a symbol table: for instance, we also have $\Gamma \vdash$ `7` : `Int` for any Γ.
- The role of Γ comes in for more complicated inferences:
  - If $\Gamma($`x`$) = \alpha$ then $\Gamma \vdash$ `x` : α

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

  These remain true in the presence of a symbol table: for instance, we also have $\Gamma \vdash$ `7` : `Int` for any Γ.
- The role of Γ comes in for more complicated inferences:
  - If $\Gamma(\mathtt{x}) = \alpha$ then $\Gamma \vdash \mathtt{x} : \alpha$
  - If $\Gamma \vdash P$: `Int` and also $\Gamma \vdash Q$: `Int`, then $\Gamma \vdash P$ `+` Q : `Int`.

Formal type inference
for our simple imperative language

- Some inferences can be made without a set of premisses. We may write these **without** a symbol table on the left:
  - $\vdash$ `true` : `Bool`
  - $\vdash$ `false` : `Bool`
  - $\vdash$ `7` : `Int`

  These remain true in the presence of a symbol table: for instance, we also have $\Gamma \vdash$ `7` : `Int` for any Γ.

- The role of Γ comes in for more complicated inferences:
  - If $\Gamma(\mathtt{x}) = \alpha$ then $\Gamma \vdash \mathtt{x} : \alpha$
  - If $\Gamma \vdash P$: `Int` and also $\Gamma \vdash Q$: `Int`, then $\Gamma \vdash P$ `+` Q : `Int`.
  - If $\Gamma \vdash P$: `Int` and also $\Gamma \vdash Q$: `Int`, then $\Gamma \vdash P$ `=` Q : `Bool`.

Formal type inference

— “rules of inference”

Writing “ If ⟨`foo`⟩ and also ⟨`bar`⟩ then ⟨`bla`⟩ ” quickly gets unwieldly.

Formal type inference

— “rules of inference”

Writing “If ⟨`foo`⟩ and also ⟨`bar`⟩ then ⟨`bla`⟩” quickly gets unwieldly. Let’s introduce some new notation:

$$\frac{\textit{Assumption}\ \#1 \qquad \ldots \qquad \textit{Assumption}\ \#n}{\textit{Conclusion}}$$

— meaning: whenever [*Assumption* #1] and ... and [*Assumption* #n] are true, then [*Conclusion*] is true.

Formal type inference

— “rules of inference”

Writing “If $\langle$`foo`$\rangle$ and also $\langle$`bar`$\rangle$ then $\langle$`bla`$\rangle$” quickly gets unwieldly. Let’s introduce some new notation:

$$\frac{\textit{Assumption}\ \#1 \qquad \ldots \qquad \textit{Assumption}\ \#n}{\textit{Conclusion}}$$

— meaning: whenever [*Assumption* #1] and `...` and [*Assumption* #n] are true, then [*Conclusion*] is true.

Example.

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

We describe these as **rules of inference.**

Rules of inference
in our simple imperative language

- Some rules without assumptions:

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \mathrm{NUM} : \texttt{Int}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \text{NUM} : \texttt{Int}}$$

- A simple rule from symbol-table lookup:

$$\frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \mathrm{NUM} : \texttt{Int}}$$

- A simple rule from symbol-table lookup:

$$\frac{\Gamma(\mathbf{x}) = \alpha}{\Gamma \vdash \mathbf{x} : \alpha}$$

- Some simple rules for numerical expressions:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \mathrm{NUM} : \texttt{Int}}$$

- A simple rule from symbol-table lookup:

$$\frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

- Some simple rules for numerical expressions:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}} \qquad \frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \text{NUM} : \texttt{Int}}$$

- A simple rule from symbol-table lookup:

$$\frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

- Some simple rules for numerical expressions:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}} \qquad \frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

- Some simple rules for statements:

$$\frac{\Gamma \vdash P : \texttt{Unit} \qquad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P \texttt{ ; } Q : \texttt{Unit}}$$

Rules of inference
in our simple imperative language

- Some rules without assumptions:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \mathrm{NUM} : \texttt{Int}}$$

- A simple rule from symbol-table lookup:

$$\frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

- Some simple rules for numerical expressions:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}} \qquad \frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

- Some simple rules for statements:

$$\frac{\Gamma \vdash P : \texttt{Unit} \qquad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash P \texttt{ ; } Q : \texttt{Unit}} \qquad \frac{\Gamma \vdash C : \texttt{Bool} \qquad \Gamma \vdash Q : \texttt{Unit}}{\Gamma \vdash \texttt{while } C \texttt{ do \{ } Q \texttt{ \}} : \texttt{Unit}}$$

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

Other rules apply for not just one possible type, but for all possible types:

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

Other rules apply for not just one possible type, but for all possible types:

$$\frac{\Gamma \vdash C : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R\ :\ \alpha}$$

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

Other rules apply for not just one possible type, but for all possible types:

$$\frac{\Gamma \vdash C : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R : \alpha} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x :=}\ P : \texttt{Unit}}$$

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

Other rules apply for not just one possible type, but for all possible types:

$$\frac{\Gamma \vdash C : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R : \alpha} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x :=}\ P : \texttt{Unit}}$$

Some rules have both of these features:

More complicated rules of inference

in our simple imperative language

Some rules of inference depend on what variables are defined in the symbol table — for instance:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i = } P \texttt{ to } Q \texttt{ \{ } R \texttt{ \}} : \texttt{Unit}}$$

Other rules apply for not just one possible type, but for all possible types:

$$\frac{\Gamma \vdash C : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } C \texttt{ then } Q \texttt{ else } R : \alpha} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \texttt{Unit}}$$

Some rules have both of these features:

$$\frac{\Gamma \vdash P : \alpha \qquad \texttt{x} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{x} : \alpha \vdash Q : \beta}{\Gamma \vdash \texttt{let x : } \alpha \texttt{ = } P \texttt{ in \{ } Q \texttt{ \}} : \beta}$$

Rules of inference for lists
in our simple imperative language

- Creating a new list:

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

Rules of inference for lists
in our simple imperative language

- Creating a new list:

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

- Adding items to a list:

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \qquad \Gamma \vdash Q : \alpha}{\Gamma \vdash P\texttt{.append(}Q\texttt{)} : \texttt{List<}\alpha\texttt{>}}$$

Rules of inference for lists
in our simple imperative language

- Creating a new list:

$$\frac{}{\Gamma \vdash \texttt{new List<}\alpha\texttt{>} : \texttt{List<}\alpha\texttt{>}}$$

- Adding items to a list:

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \qquad \Gamma \vdash Q : \alpha}{\Gamma \vdash P\texttt{.append(}Q\texttt{)} : \texttt{List<}\alpha\texttt{>}}$$

- Looking up items in a list:

$$\frac{\Gamma \vdash P : \texttt{List<}\alpha\texttt{>} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P\texttt{.get(}Q\texttt{)} : \alpha}$$

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad\qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

- We *could* even define apparently silly rules like

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \texttt{Bool}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

- We *could* even define apparently silly rules like

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \texttt{Bool}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

  if we have a reasonable approach to define what numerical values such expressions as, *e.g.*,

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

- We *could* even define apparently silly rules like

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \texttt{Bool}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

  if we have a reasonable approach to define what numerical values such expressions as, *e.g.*, "`7 + false`"

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

- We *could* even define apparently silly rules like

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \texttt{Bool}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

  if we have a reasonable approach to define what numerical values such expressions as, *e.g.*, "`7 + false`" or even "`for i = 1 to 3 { i=0 } + true`"

Alternative rules?
(for our simple imperative language)

The ways in which we could define the type inferences — these are **design choices** (though most **were** the most sensible choices).

- Some alternative rules are also meaningful — *e.g.*,

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \beta}{\Gamma \vdash P \texttt{ ; } Q : \beta} \qquad \frac{\Gamma \vdash \texttt{x} : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \texttt{x := } P : \alpha}$$

  - What expressions should be returned in these two cases?

- We *could* even define apparently silly rules like

$$\frac{\Gamma \vdash P : \alpha \qquad \Gamma \vdash Q : \texttt{Bool}}{\Gamma \vdash P \texttt{ + } Q : \texttt{Int}}$$

  if we have a reasonable approach to define what numerical values such expressions as, *e.g.*, "`7 + false`" or even "`for i = 1 to 3 { i=0 } + true`" should evaluate to.

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();
```

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();

env.get(x);                    // returns the type associated with x.
```

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();

env.get(x);              // returns the type associated with x.

env.put(x,T);            // associates x with type T
```

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();

env.get(x);              // returns the type associated with x.

env.put(x,T);            // associates x with type T

env.containsKey(x);      // returns true if there is an association for x
```

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();

env.get(x);              // returns the type associated with x.

env.put(x,T);            // associates x with type T

env.containsKey(x);      // returns true if there is an association for x

env.remove(x);           // returns association for x, if it exists
```

Symbol-tables in (pseudo-)code

Remember the symbol table Γ, mapping identifiers (names, variables) to their types.

We use the symbol table to track the following:

- Do we define every identifier that we use (and, define it exactly once)?
- Can every expression involving the identifier, be successfully typed?

In Java we could use the following functionality:

```
HashMap<String, Type> env = new HashMap<String, Type>();

env.get(x);              // returns the type associated with x.

env.put(x,T);            // watch out: does the language allow re-declaration?

env.containsKey(x);      // returns true if there is an association for x

env.remove(x);           // returns association for x, if it exists
```

Overriding type associations

```
env.put ( x, TypeInt )   // env.get(x) == TypeInt
```

Overriding type associations

```
env.put ( x, TypeInt )  // env.get(x) == TypeInt

env.put ( x, TypeBool ) // env.get(x) == TypeBool
```

This might or might not be behaviour that we should allow in the source language which we are compiling from.

Overriding type associations

```
env.put ( x, TypeInt )  // env.get(x) == TypeInt

env.put ( x, TypeBool ) // env.get(x) == TypeBool
```

This might or might not be behaviour that we should allow in the source language which we are compiling from.

- If variables can be defined more than once, this is fine.

Overriding type associations

```
env.put ( x, TypeInt )  // env.get(x) == TypeInt

env.put ( x, TypeBool ) // env.get(x) == TypeBool
```

This might or might not be behaviour that we should allow in the source language which we are compiling from.

- If variables can be defined more than once, this is fine.
- If variables can only be defined once, we should first test for whether `x` already has an association, and produce an error if so.

Overriding type associations

```
env.put ( x, TypeInt )  // env.get(x) == TypeInt

env.put ( x, TypeBool ) // env.get(x) == TypeBool
```

This might or might not be behaviour that we should allow in the source language which we are compiling from.

- If variables can be defined more than once, this is fine.
- If variables can only be defined once, we should first test for whether `x` already has an association, and produce an error if so.
- If overriding variables in narrower 'scopes' is allowed, *e.g.*

```
let x : List<Int> = new List<Int> in {
  let y : Int = x*x in {
    let x :  Int = y*y in {
      x*x } } }
```

then we must do something subtler.

Pseudocode for type-checking
(for our simple imperative language)

We want to write the following method in pseudo-code:

```
Type check (HashMap<String,Type> env, Prog p) {
    /* ... */ }
```

Pseudocode for type-checking
(for our simple imperative language)

We want to write the following method in pseudo-code:

```
Type check (HashMap<String,Type> env, Prog p) {
    /* ... */ }
```

- Returns the type of `p` if possible
  (*i.e.*, under the premises of the identifiers stored in `env`)

Pseudocode for type-checking
(for our simple imperative language)

We want to write the following method in pseudo-code:

```
Type check (HashMap<String,Type> env, Prog p) {
    /* ... */ }
```

- Returns the type of `p` if possible
  (*i.e.*, under the premises of the identifiers stored in `env`)
- Otherwise, it will throw an exception.

Pseudocode for type-checking
(for our simple imperative language)

We want to write the following method in pseudo-code:

```
Type check (HashMap<String,Type> env, Prog p) {
    /* ... */ }
```

- Returns the type of `p` if possible
  (*i.e.*, under the premises of the identifiers stored in `env`)
- Otherwise, it will throw an exception.

The implementation of such a function depends on the inference rules of the types (*e.g.*, that "`Int + Int`" is again `Int`)

Pseudocode for type-checking
(for our simple imperative language)

We want to write the following method in pseudo-code:

```
Type check (HashMap<String,Type> env, Prog p) {
    /* ... */ }
```

- Returns the type of `p` if possible
  (*i.e.*, under the premises of the identifiers stored in `env`)
- Otherwise, it will throw an exception.

The implementation of such a function depends on the inference rules of the types (*e.g.*, that "`Int + Int`" is again `Int`), and also knowledge of the **absence** of such rules (*e.g.*, that `Int` is the only type allowed with additions).

Pseudocode for type-check base cases
(in our simple imperative language)

Implementing the following rules is simple:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{7} : \texttt{Int}} \qquad \frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

Pseudocode for type-check base cases
(in our simple imperative language)

Implementing the following rules is simple:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{7} : \texttt{Int}} \qquad \frac{\Gamma(\texttt{x}) = \alpha}{\Gamma \vdash \texttt{x} : \alpha}$$

```
Type check ( HashMap<String,Type> env,  Prog p ) {
  if (p instanceOf BooleanLiteral)
    return TypeBool;
  else if (p instanceOf IntLiteral)
    return TypeInt;
  else if (p instanceOf Idfr)
    return env.get(x);
  else /* ... */
}
```

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown).

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown). As

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

is the **only** rules of inference involving '`=`' , we should write:

```
Type check ( HashMap<String, Type> env, Prog p ) {
```

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown). As

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

is the **only** rules of inference involving '`=`' , we should write:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf Comparison) {
```

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown). As

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

is the **only** rules of inference involving '`=`' , we should write:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf Comparison) {
    Type lhsType = check(env, (Comparison)p.lhs);
    Type rhsType = check(env, (Comparison)p.rhs);
```

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown). As

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

is the **only** rules of inference involving '`=`' , we should write:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf Comparison) {
    Type lhsType = check(env, (Comparison)p.lhs);
    Type rhsType = check(env, (Comparison)p.rhs);

    if (lhsType != TypeInt || rhsType != TypeInt)
```

Cases in which type-checking may yield errors

There will of course be times when type-checking should result in an error (in our case, an exception being thrown). As

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int}}{\Gamma \vdash P \texttt{ = } Q : \texttt{Bool}}$$

is the **only** rules of inference involving '`=`' , we should write:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf Comparison) {
    Type lhsType = check(env, (Comparison)p.lhs);
    Type rhsType = check(env, (Comparison)p.rhs);

    if (lhsType != TypeInt || rhsType != TypeInt)
       throw Exception ("type error")
    else return TypeBool; }
  else /* ... */ }
```

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

We might realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
```

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

We might realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf IfThenElseStmt) {
```

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

We might realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf IfThenElseStmt) {
    Type thenType = check(env, (IfThenElseStmt)p.thenStmt);
    Type elseType = check(env, (IfThenElseStmt)p.elseStmt);
```

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

We might realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf IfThenElseStmt) {
    Type thenType = check(env, (IfThenElseStmt)p.thenStmt);
    Type elseType = check(env, (IfThenElseStmt)p.elseStmt);

    if (thenType != elseType)
```

Type-checking rules allowing arbitrary types

Some rules allow some terms to have an arbitrary type, but for these types to be consistent with other types: for instance,

$$\frac{\Gamma \vdash P : \texttt{Bool} \qquad \Gamma \vdash Q : \alpha \qquad \Gamma \vdash R : \alpha}{\Gamma \vdash \texttt{if } P \texttt{ then } Q \texttt{ else } R : \alpha}$$

We might realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf IfThenElseStmt) {
    Type thenType = check(env, (IfThenElseStmt)p.thenStmt);
    Type elseType = check(env, (IfThenElseStmt)p.elseStmt);

    if (thenType != elseType)
       throw Exception ("type error")
    else return thenType; }
  else /* ... */ }
x
```

Checking for free names

Some rules may depend on whether a variable has already been declared

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
```

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf ForLoop) {
```

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf ForLoop) {
    Type startValType = check(env, (ForLoop)p.loopStart);
    Type endValType = check(env, (ForLoop)p.loopEnd);
```

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf ForLoop) {
    Type startValType = check(env, (ForLoop)p.loopStart);
    Type endValType = check(env, (ForLoop)p.loopEnd);

    if (startValType == TypeInt && endValType == TypeInt
        && !env.containsKey(i)) {
      env.put(i, TypeInt);
      if (check(env, (ForLoop)p.body) == UnitType) {
        env.remove(i);
        return UnitType; } }
```

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf ForLoop) {
    Type startValType = check(env, (ForLoop)p.loopStart);
    Type endValType = check(env, (ForLoop)p.loopEnd);

    if (startValType == TypeInt && endValType == TypeInt
        && !env.containsKey(i)) {
      env.put(i, TypeInt);
      if (check(env, (ForLoop)p.body) == UnitType) {
        env.remove(i);
        return UnitType; } }
    /* ... and if the types don't check out? ... */
```

Checking for free names

Some rules may depend on whether a variable has already been declared, such as:

$$\frac{\Gamma \vdash P : \texttt{Int} \qquad \Gamma \vdash Q : \texttt{Int} \qquad \texttt{i} \text{ undefined in } \Gamma \qquad \Gamma, \texttt{i} : \texttt{Int} \vdash R : \texttt{Unit}}{\Gamma \vdash \texttt{for i =}\ P\ \texttt{to}\ Q\ \texttt{\{}\ R\ \texttt{\}} : \texttt{Unit}}$$

We may realise this as follows:

```
Type check ( HashMap<String, Type> env, Prog p ) {
  if /* ... */
  else if (p instanceOf ForLoop) {
    Type startValType = check(env, (ForLoop)p.loopStart);
    Type endValType = check(env, (ForLoop)p.loopEnd);

    if (startValType == TypeInt && endValType == TypeInt
        && !env.containsKey(i)) {
      env.put(i, TypeInt);
      if (check(env, (ForLoop)p.body) == UnitType) {
        env.remove(i);
        return UnitType; } }
    /* ... and if the types don't check out? ... */ }
  /* ... */ }
```

A lot more could be said about type-checking

- Typing objects and classes, subtyping
  (structural vs nominal subtyping)
- Typing methods
- Inheritance
- Traits
- Higher-kinded types
- Dependent types, to catch non-trival logical bugs
  (*e.g.*, specifying “this is a sorting function” as a type)
- Faster type-checking algorithms
- Type-inference algorithms
- Rust-style lifetime inference
- Gradual typing (*c.f.* Typescript)
- Types for parallel computing

— and more!

Summary

- **Semantic analysis** is a level of error checking which helps make programs more predictable (and to catch errors)
- **Types** are weak (or partial) specifications of programs — or classifications of program fragments — which are one of the principal tools of semantic analysis

Summary

- **Semantic analysis** is a level of error checking which helps make programs more predictable (and to catch errors)
- **Types** are weak (or partial) specifications of programs — or classifications of program fragments — which are one of the principal tools of semantic analysis
- We can check them by walking the AST, and gathering information from sub-trees and checking for consistency with the rest of the sub-tree.
- **Symbol tables** are used to carry information about types in the program from from the beginning of the program to the end, overcoming the limitations of ASTs / CFGs in ensuring the coherence of a program/