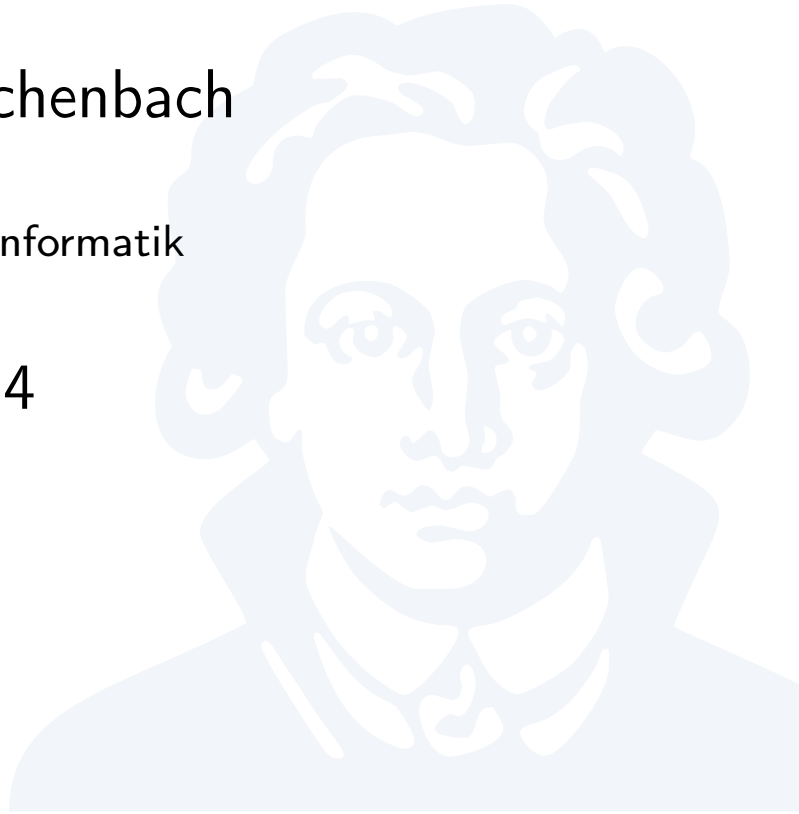
Foundations of Programming Languages

Overview: Compilers

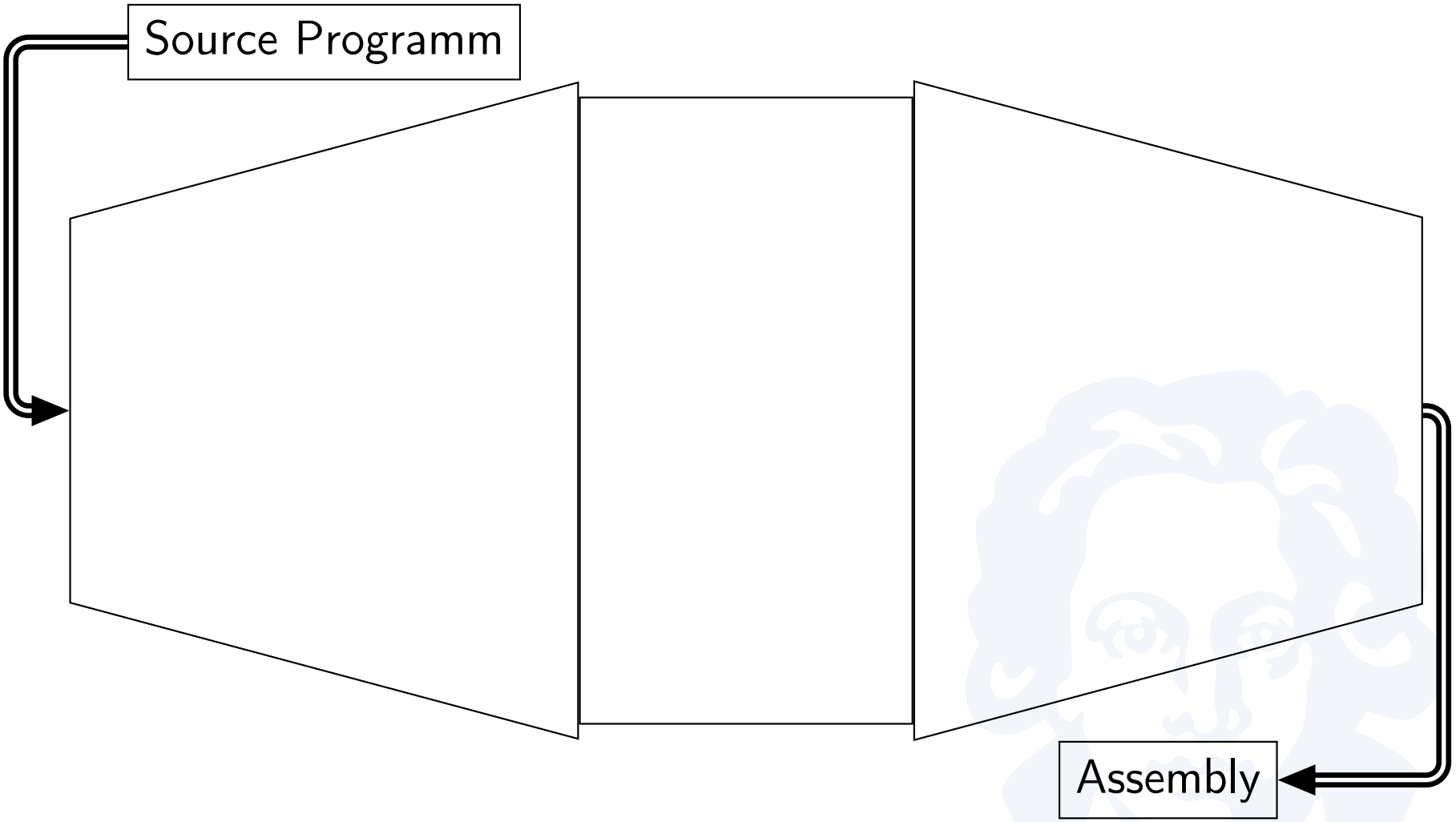
Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

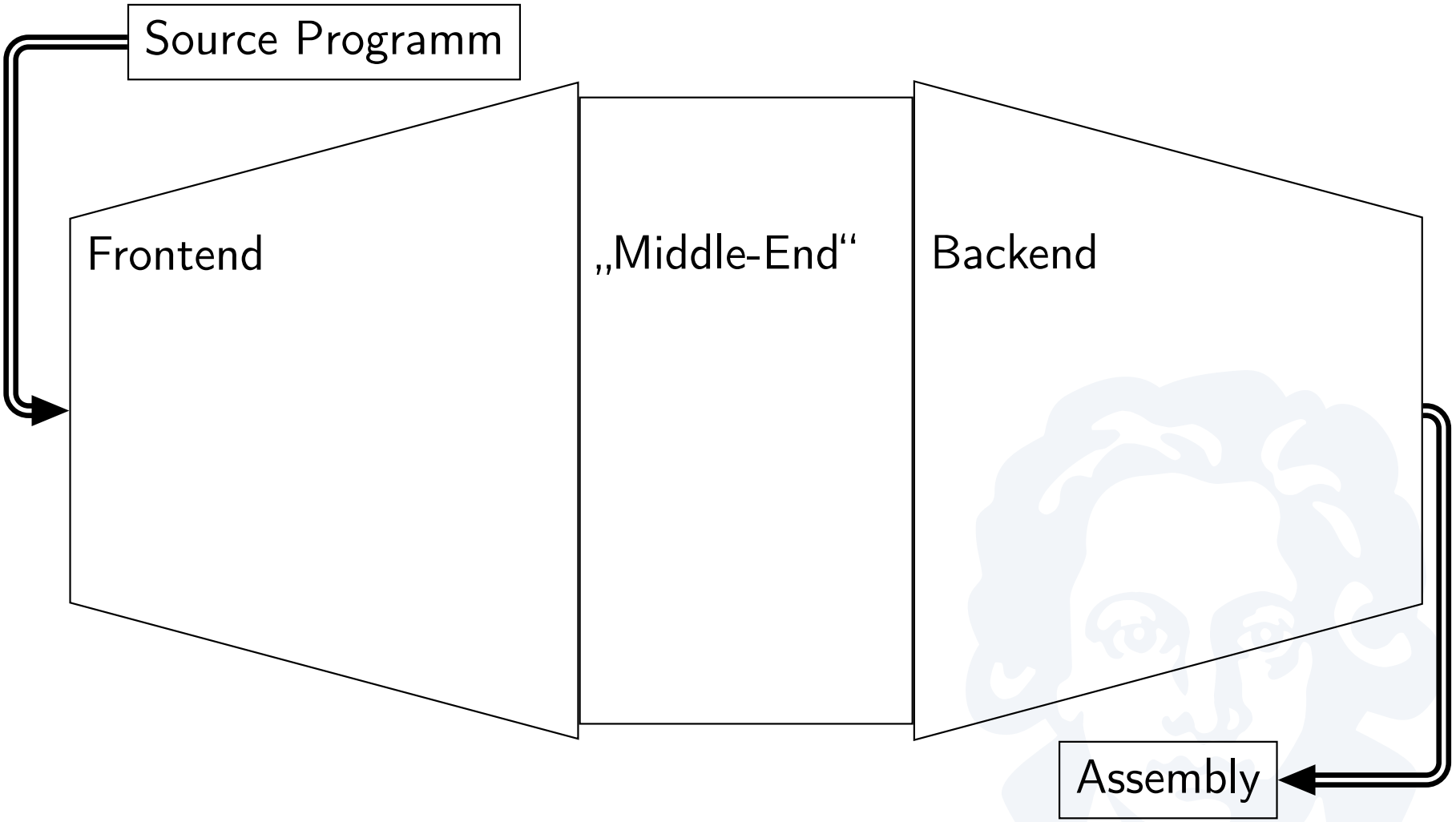
17. Oktober 2014



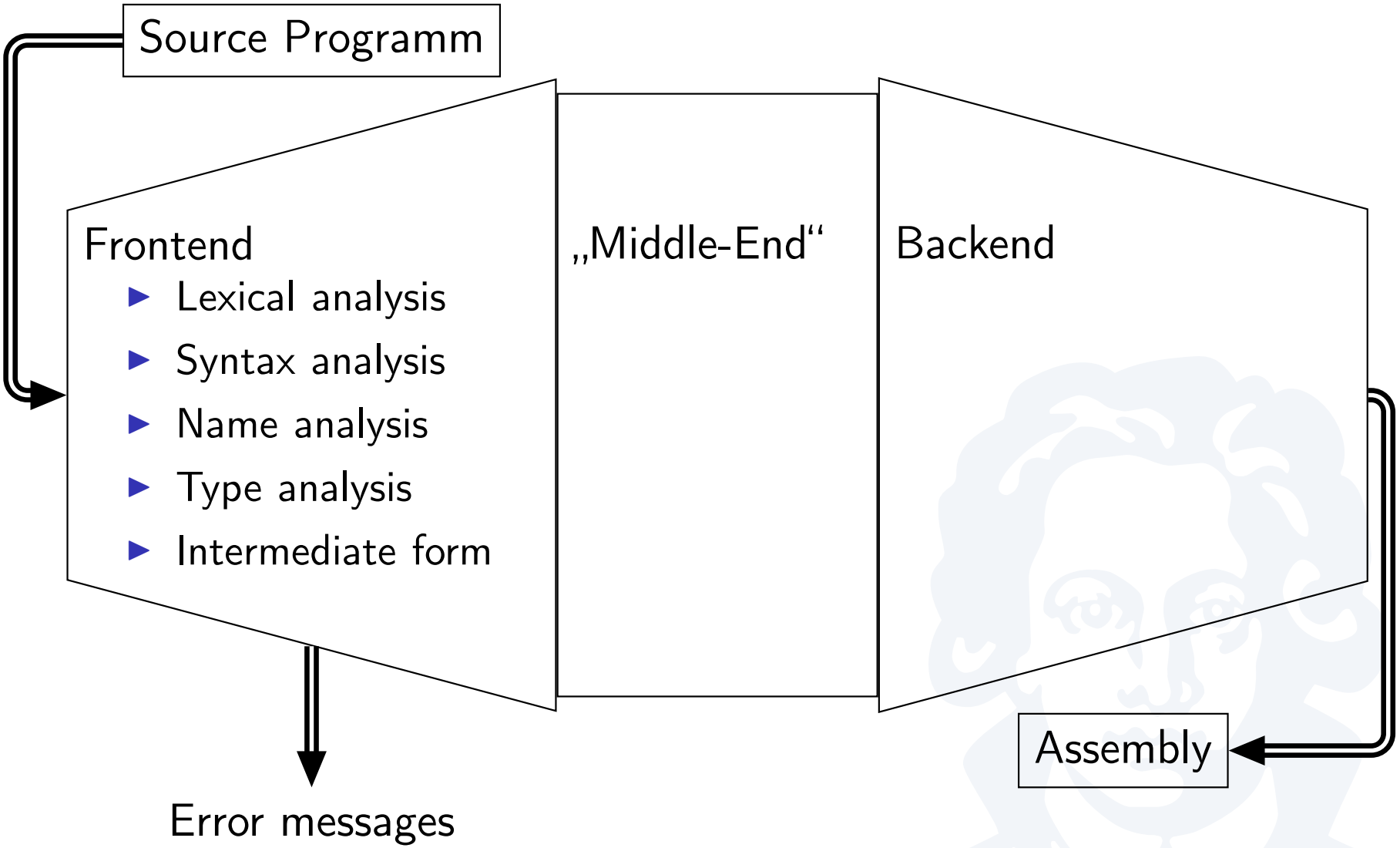
Compilers



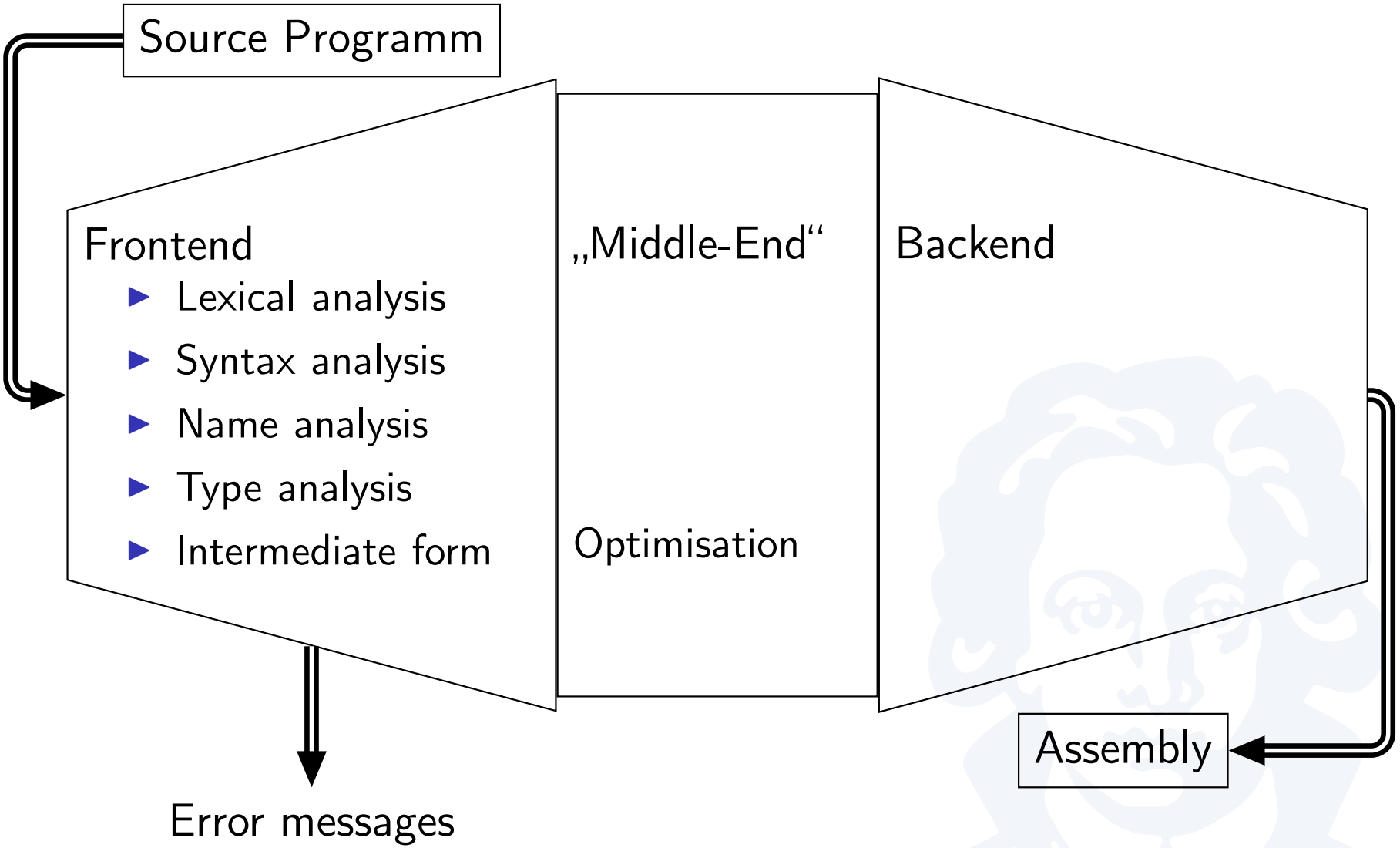
Compilers



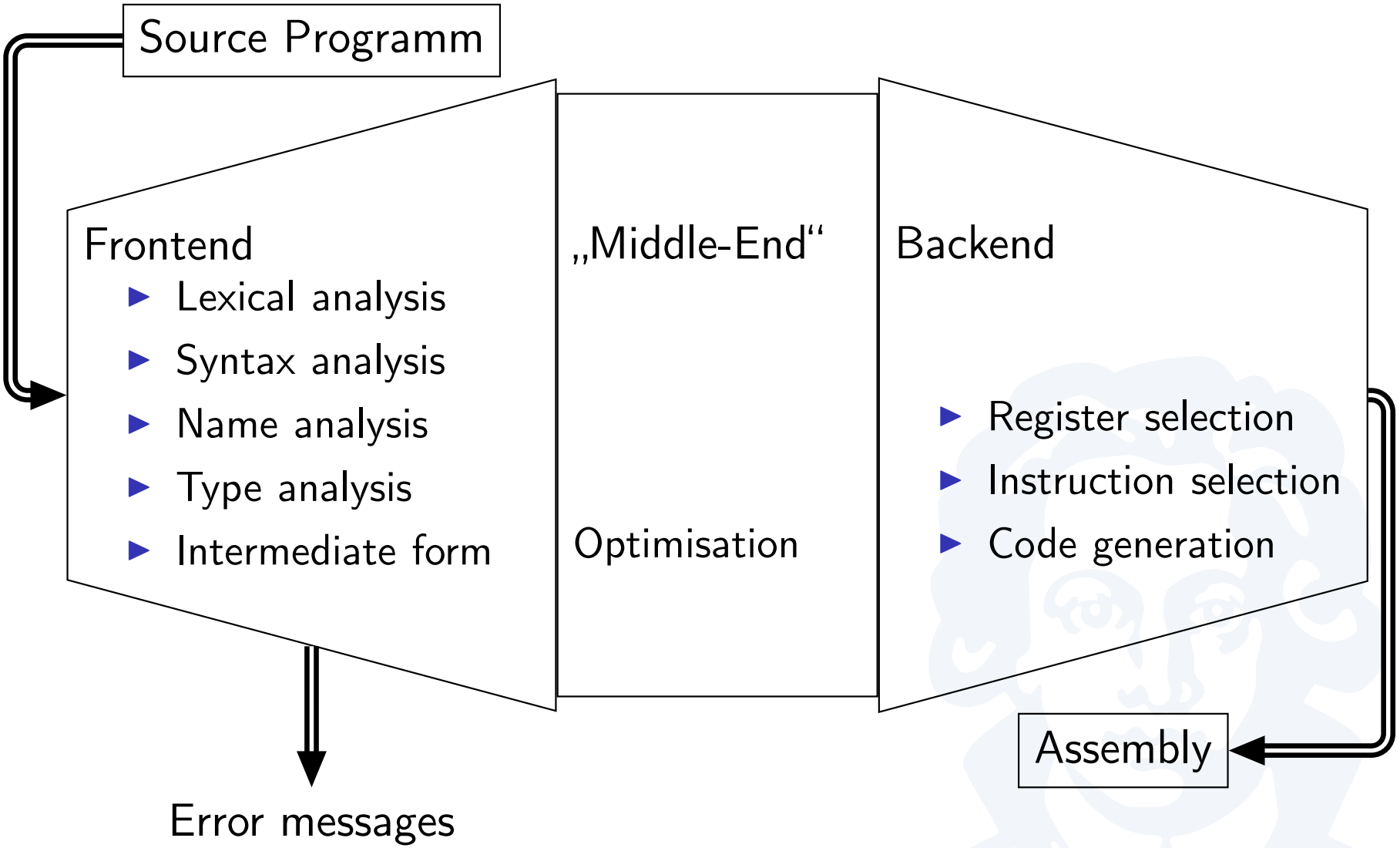
Compilers


Source Programm
Frontend
Lexical analysis
Syntax analysis
Name analysis
Type analysis
Intermediate form
„Middle-End“
Backend
Assembly
Error messages

Compilers


Source Programm
Frontend
Lexical analysis
Syntax analysis
Name analysis
Type analysis
Intermediate form
„Middle-End“
Optimisation
Backend
Assembly
Error messages

Compilers

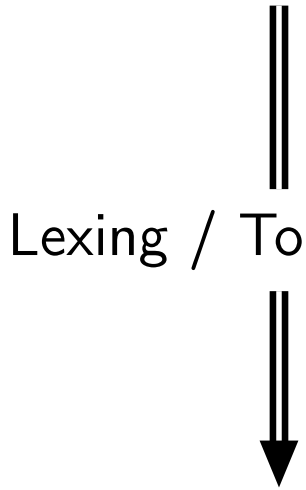

Source Programm
Frontend
Lexical analysis
Syntax analysis
Name analysis
Type analysis
Intermediate form
„Middle-End“
Optimisation
Backend
Register selection
Instruction selection
Code generation
Assembly
Error messages

Java lexing

```
int i;
if (2 > 0) {
  i = "One";
}
return i;
```

Java lexing

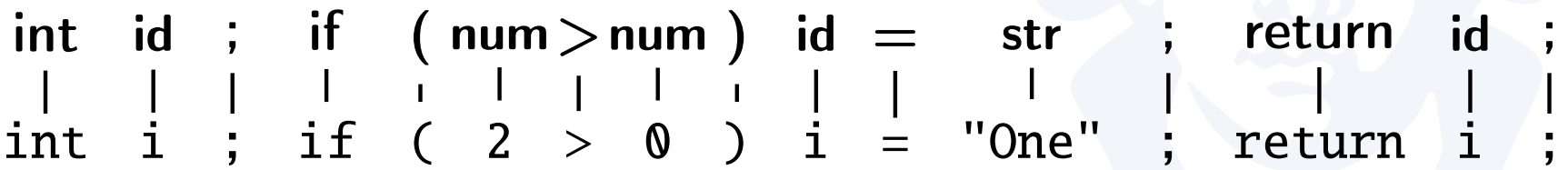
```
int i;
if (2 > 0) {
  i = "One";
}
return i;
```



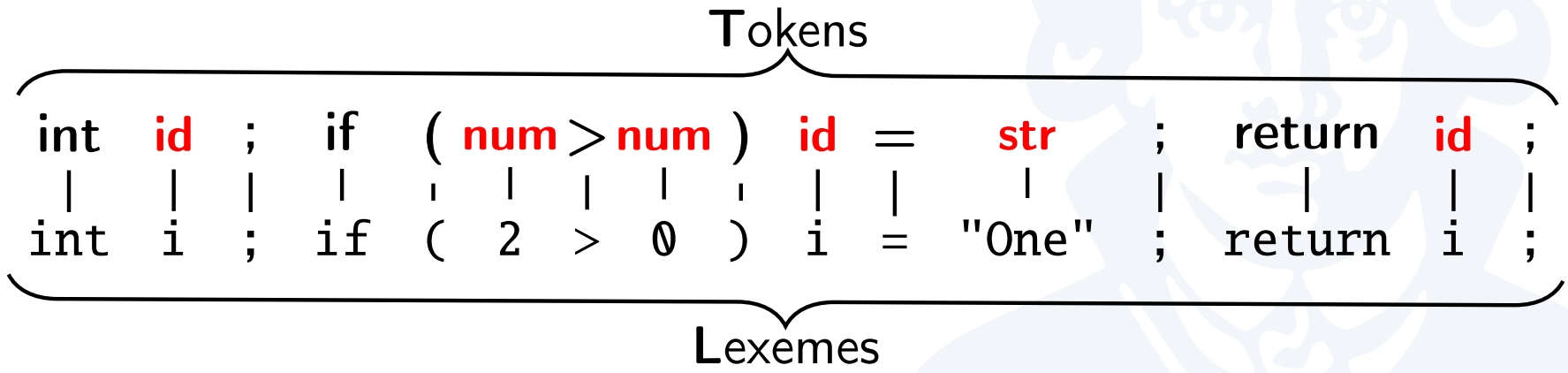
Lexing / Tokenisation


```
int  i  ;  if  (  2  >  0  )  i  =  "One"  ;  return  i  ;
```

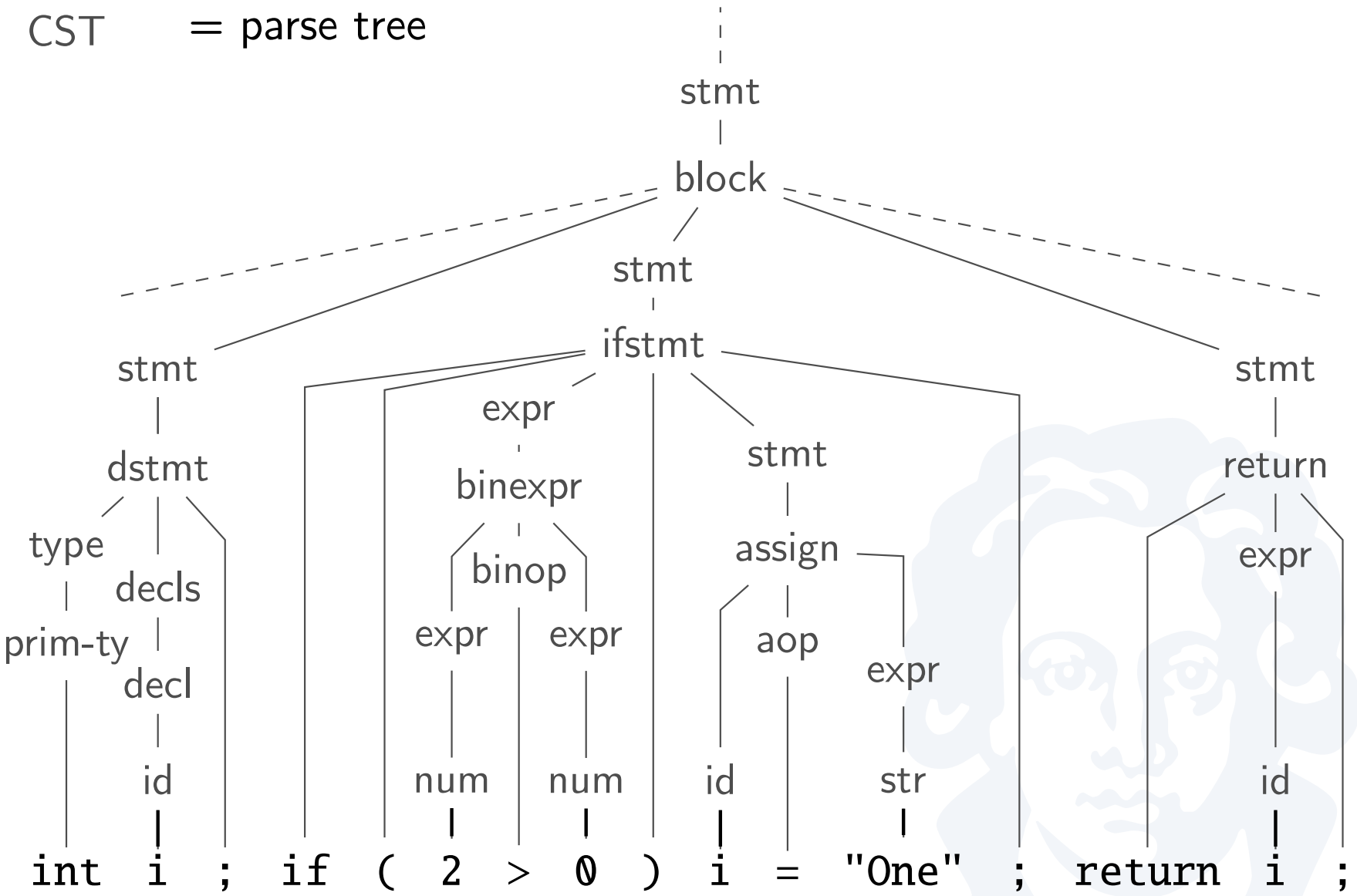
Java lexing & parsing



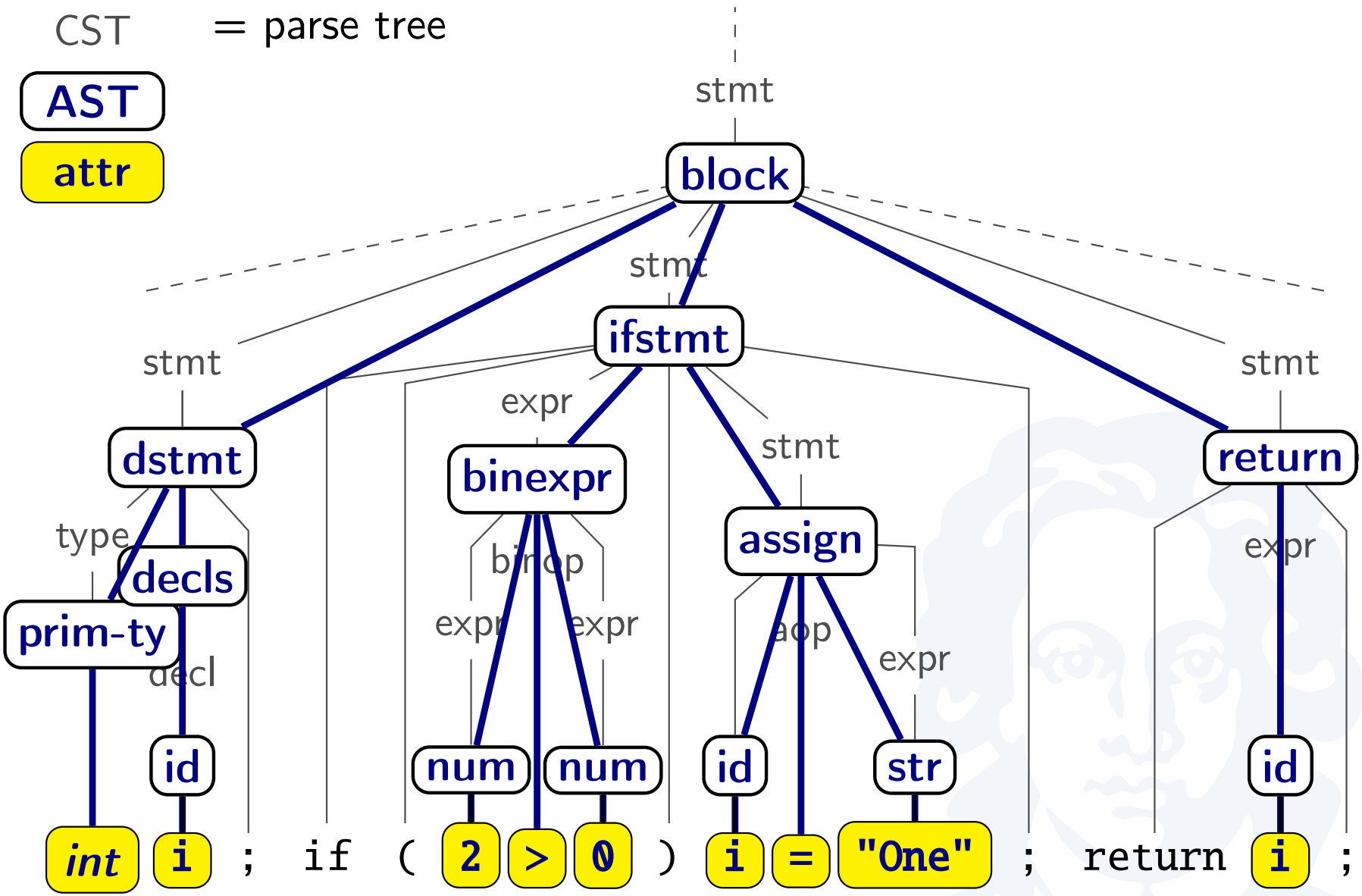
Java lexing & parsing



Java lexing & parsing



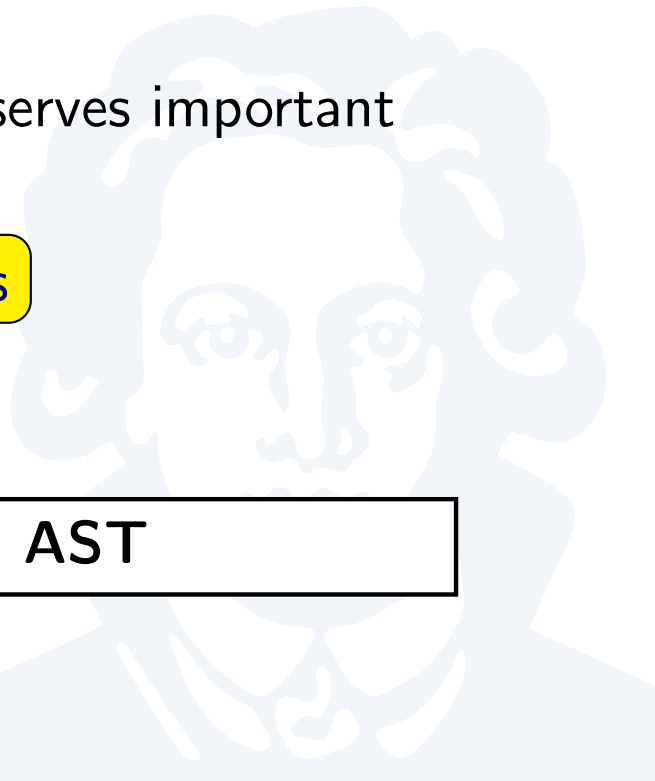
Java lexing & parsing



Parsing in general

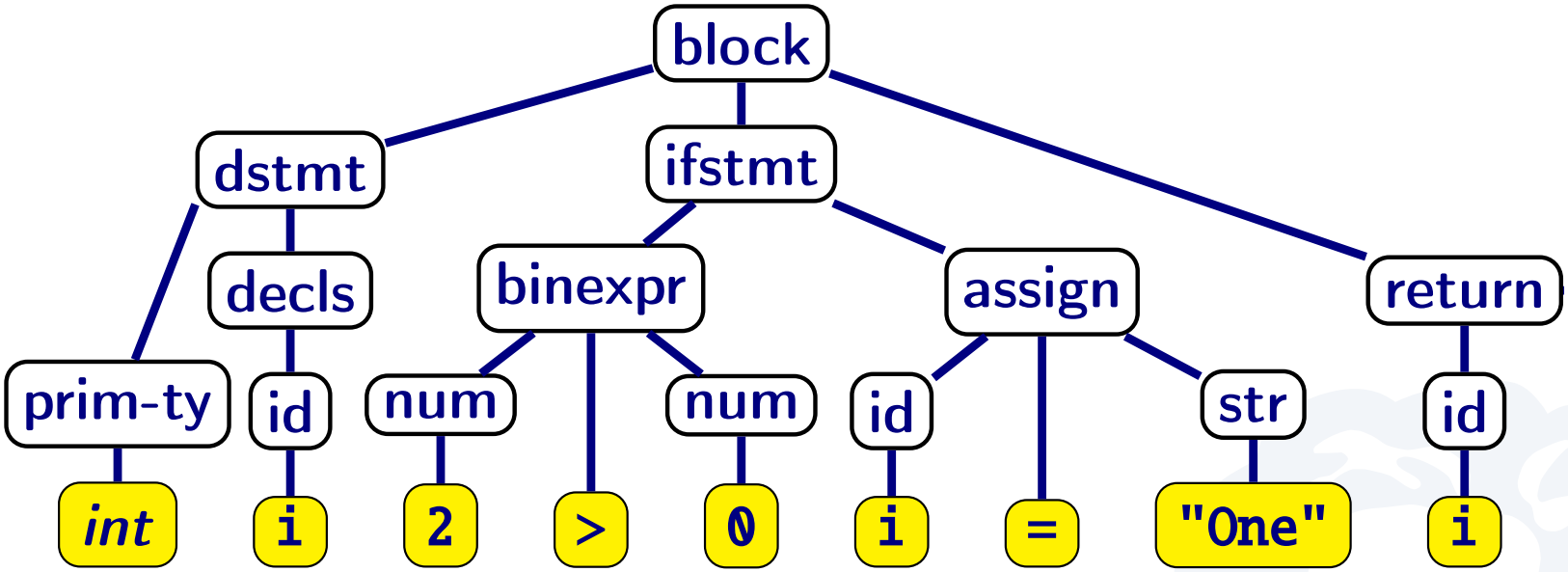
*Translate text files into **meaningful** in-memory structures*

- CST = Concrete Syntax Tree
  - Full "parse", cf. language BNF grammar
  - Not usually materialised in memory
- AST = Abstract Syntax Tree
  - Standard in-memory representation
  - Avoids syntactic sugar from CST, preserves important nonterminals as **AST nodes**
  - Converts useful tokens into attributes



Program analysis starts on the AST

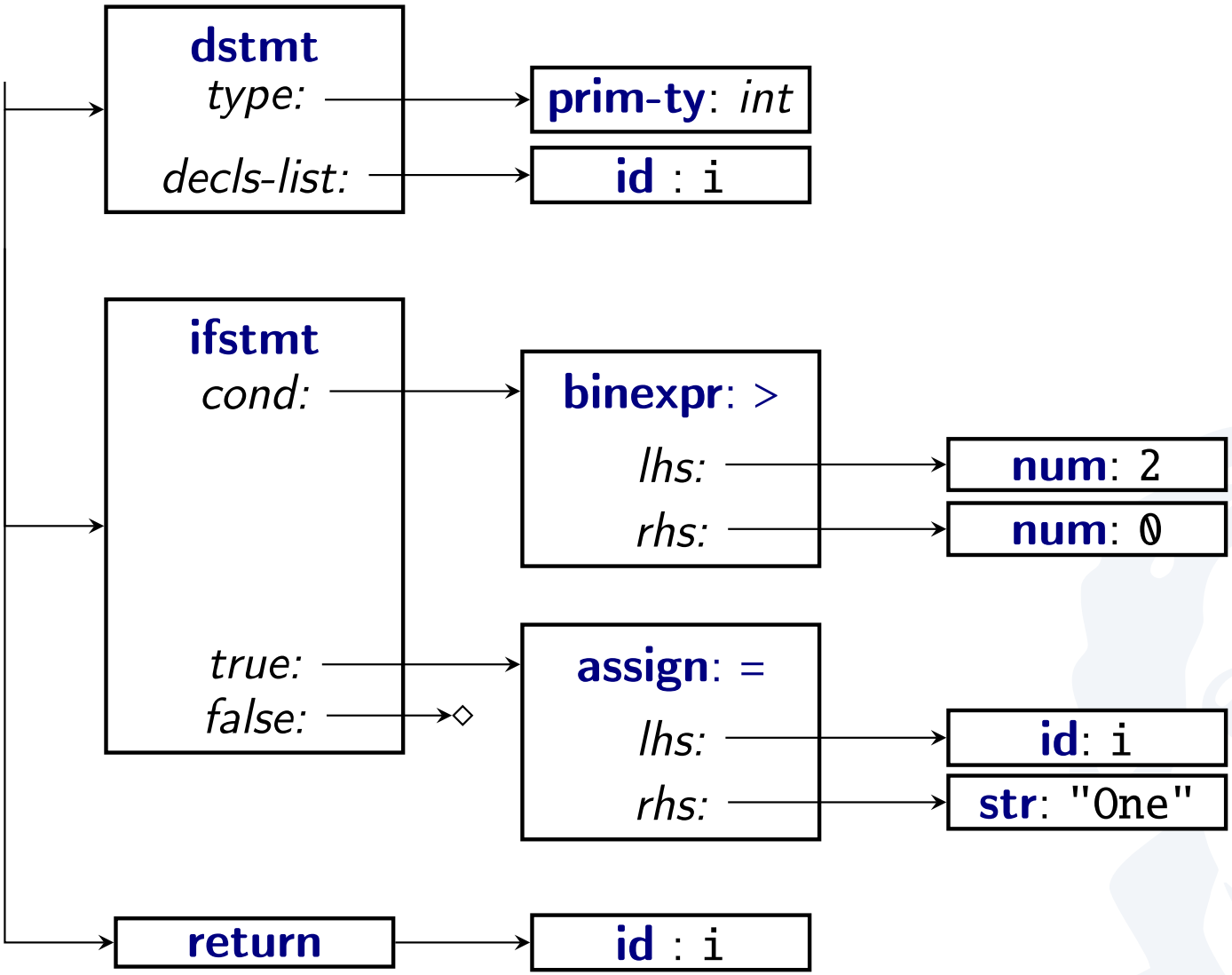
In-Memory Representation



Typical in-memory representations for this AST:

- Algebraic values (functional)
- Records (imperative)

In-Memory Representation

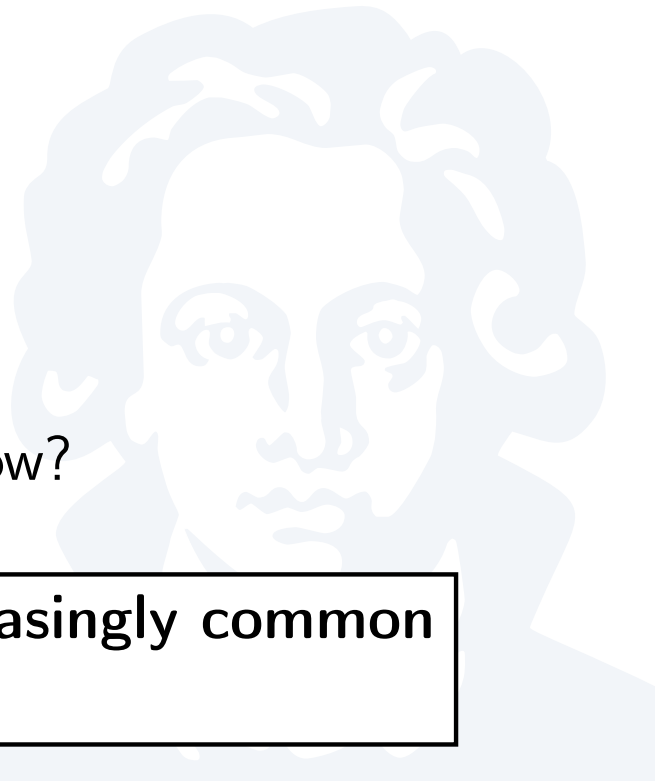


Program Analysis

We run numerous code analyses on the AST:

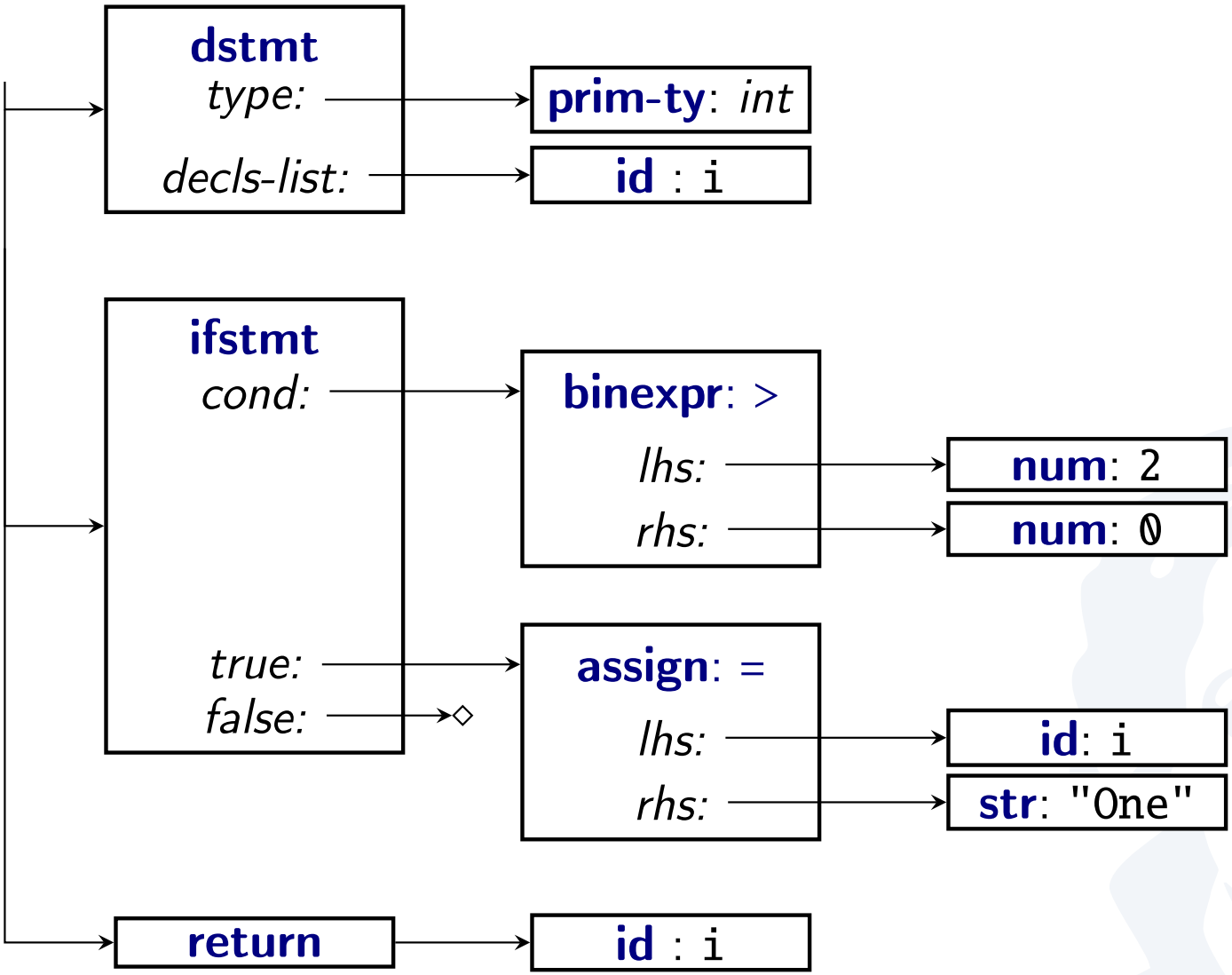
- *Name Analysis*:
  - Which name *use* binds to which *declaration*?
- *Type Analysis*:
  - What are the types of all expressions?
- *Static Correctness Checks*:
  - Are there type errors?
  - Is a variable unused?
  - Are we initialising all variables?

  ...
- *Optimisations*:
  - Can we speed up the program somehow?

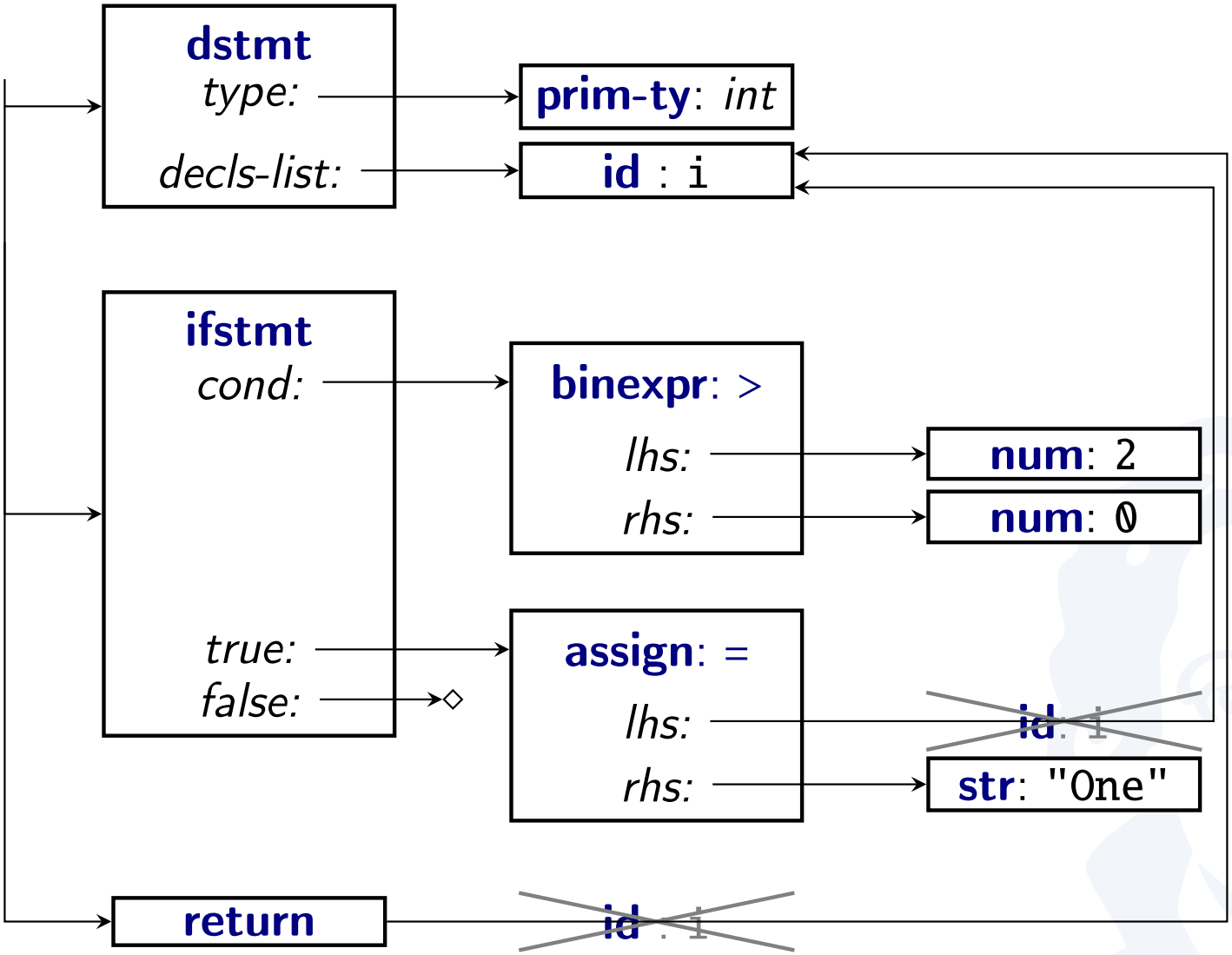


Advanced static correctness checks increasingly common in compilers

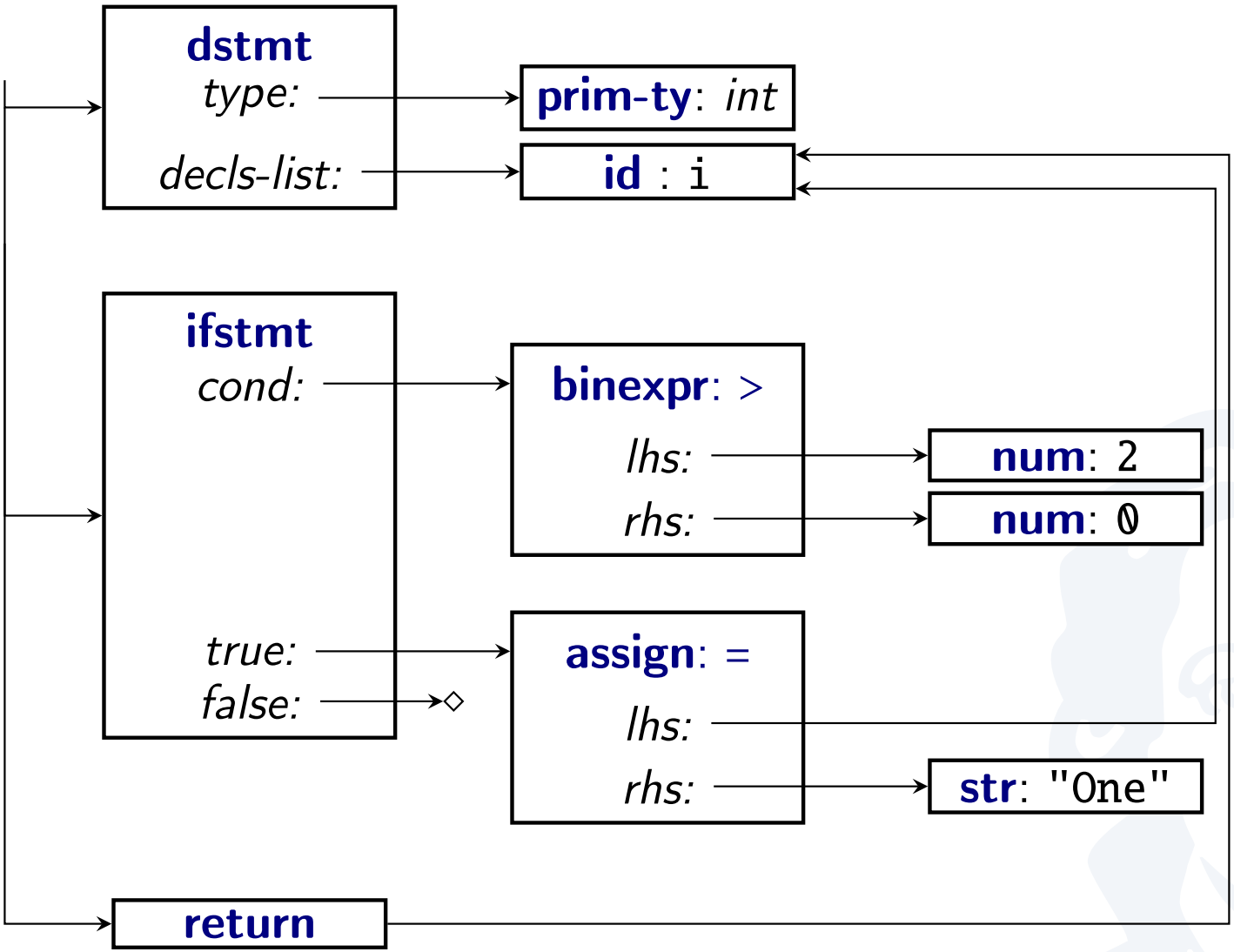
Name Analysis



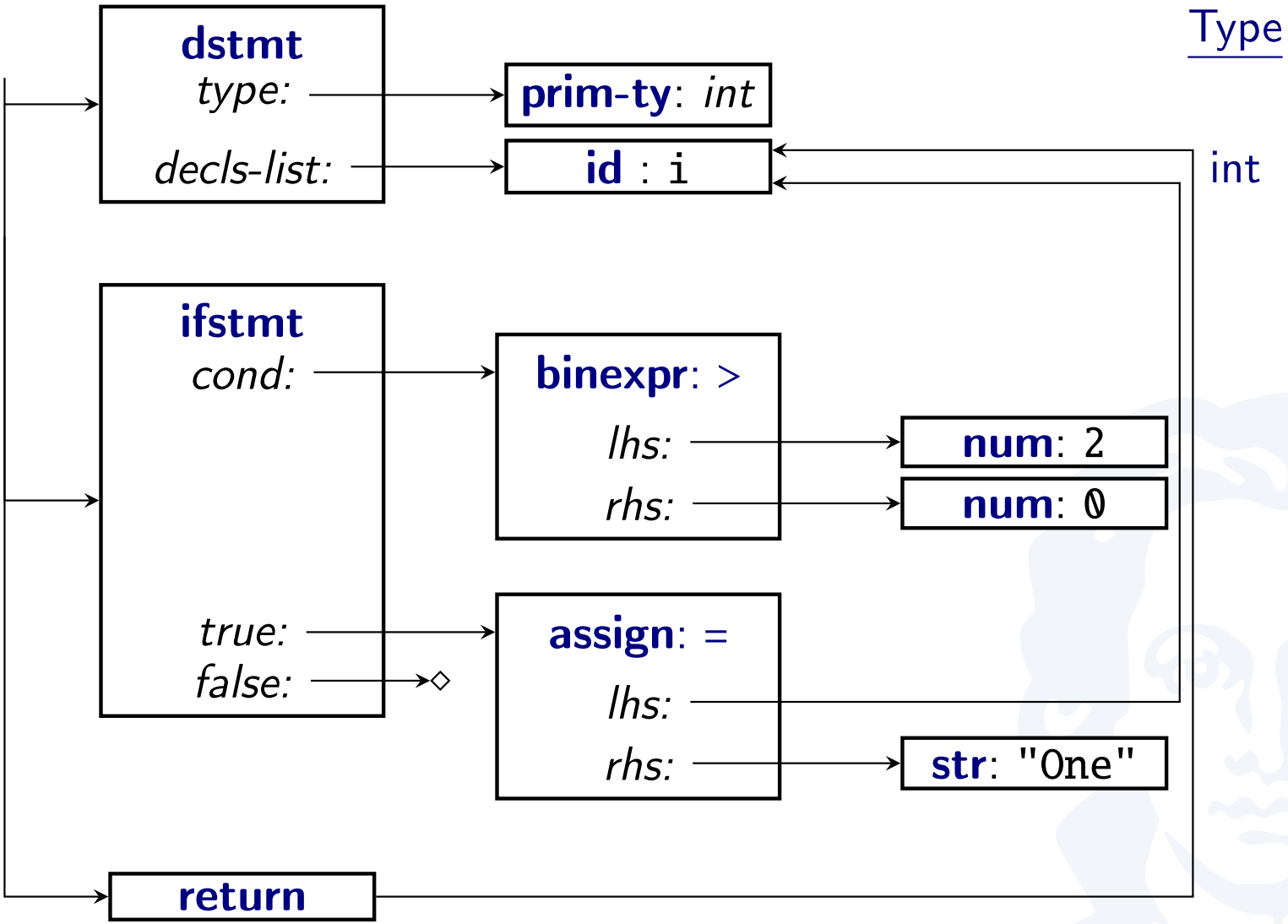
Name Analysis



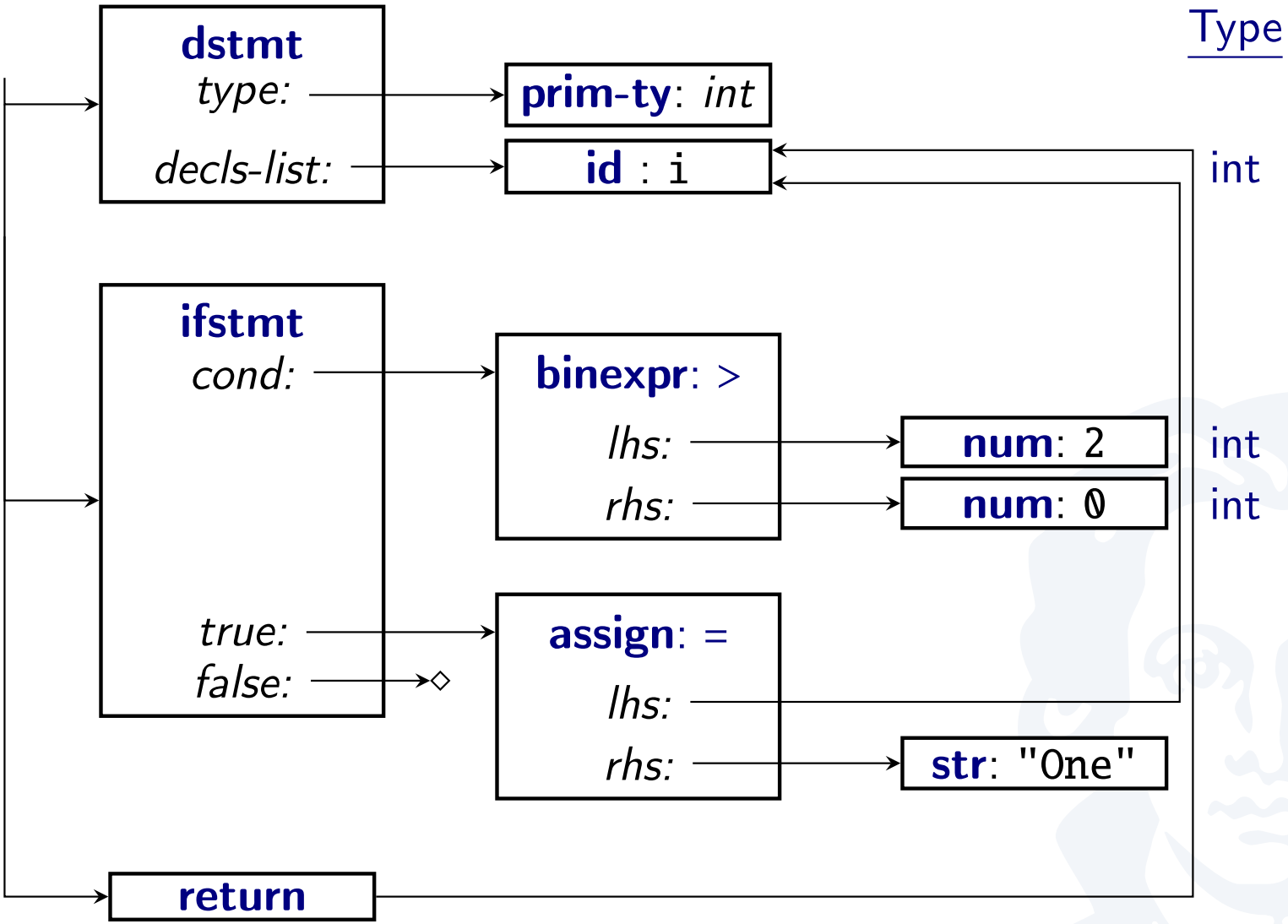
Type Analysis



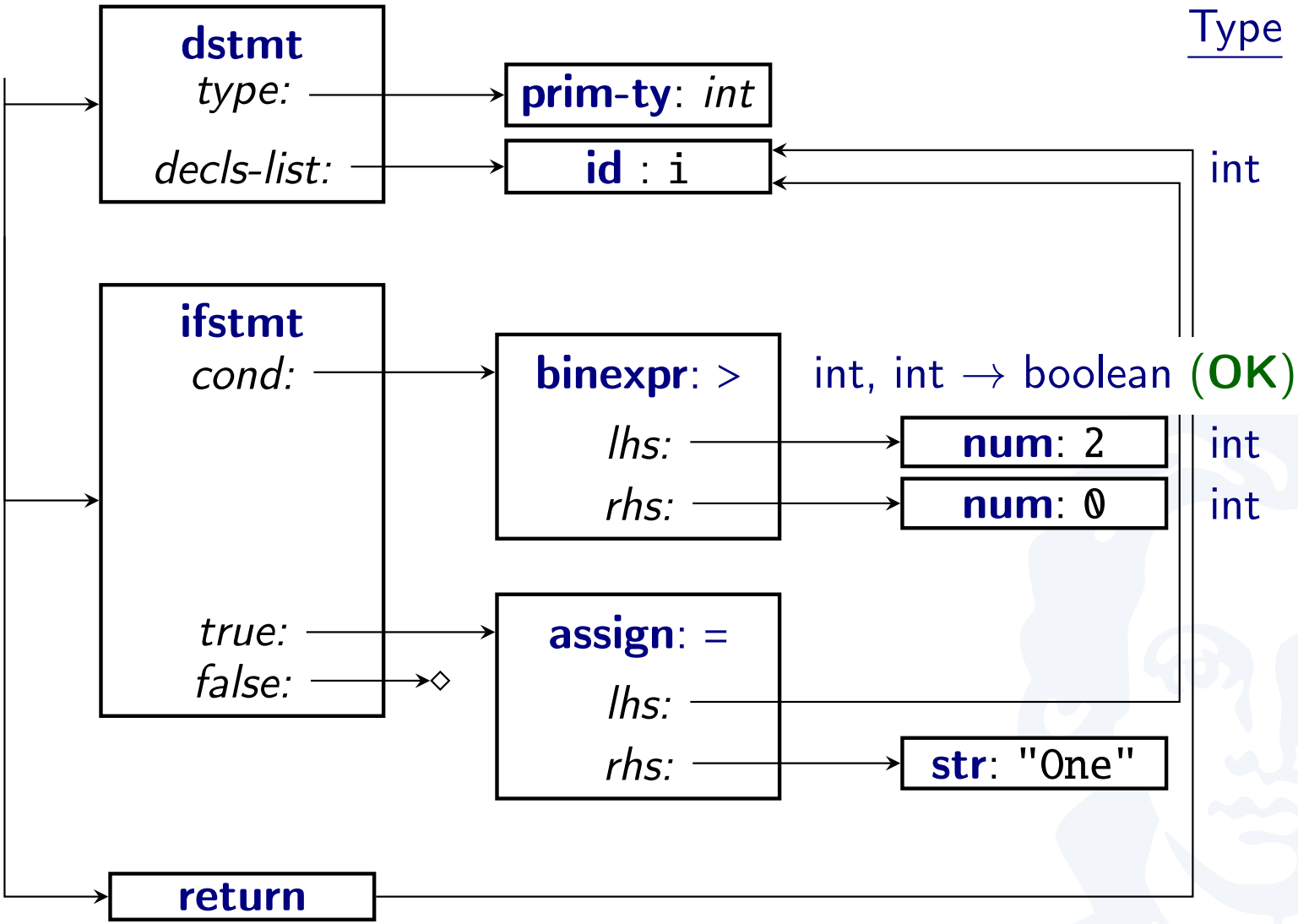
Type Analysis



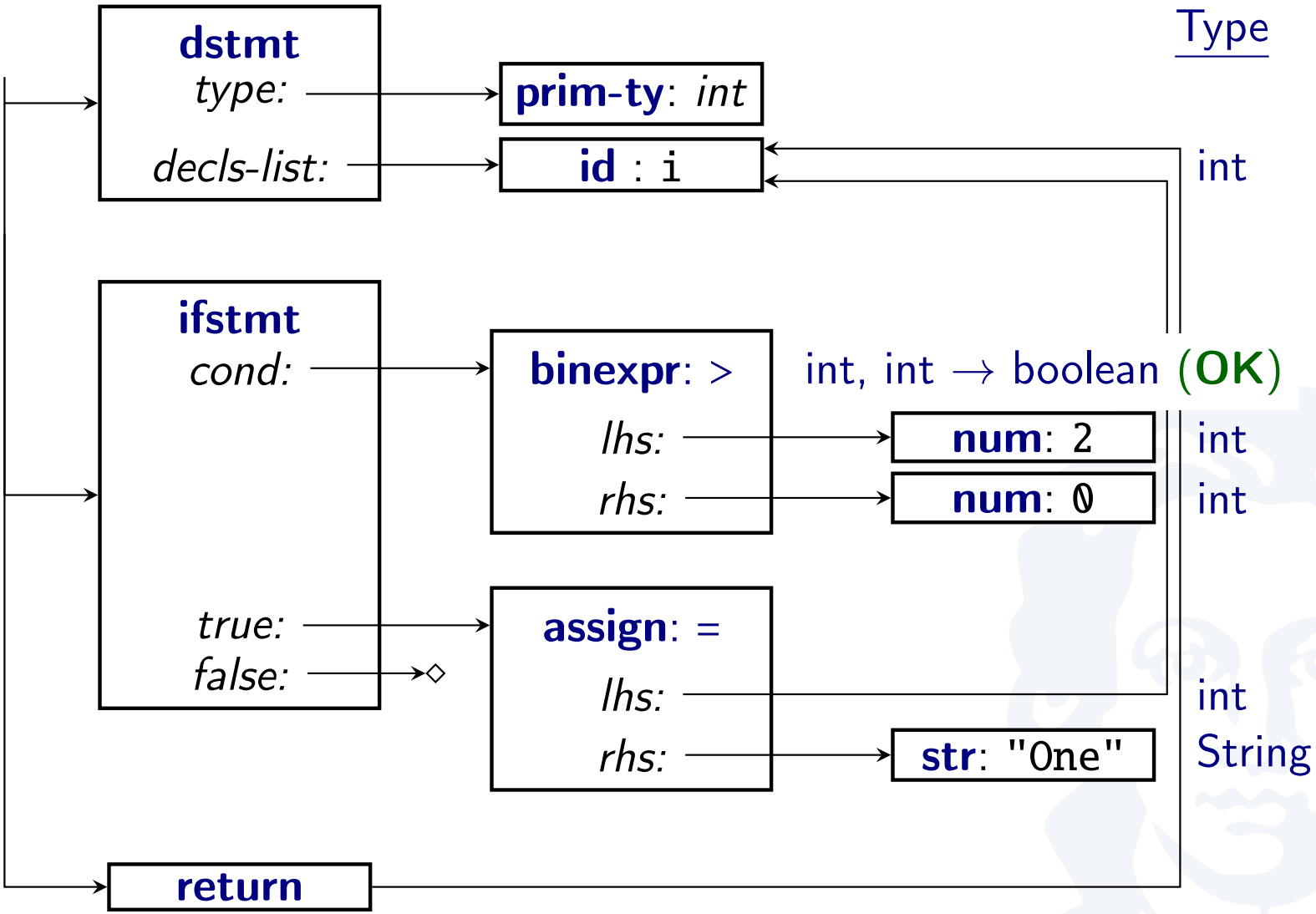
Type Analysis



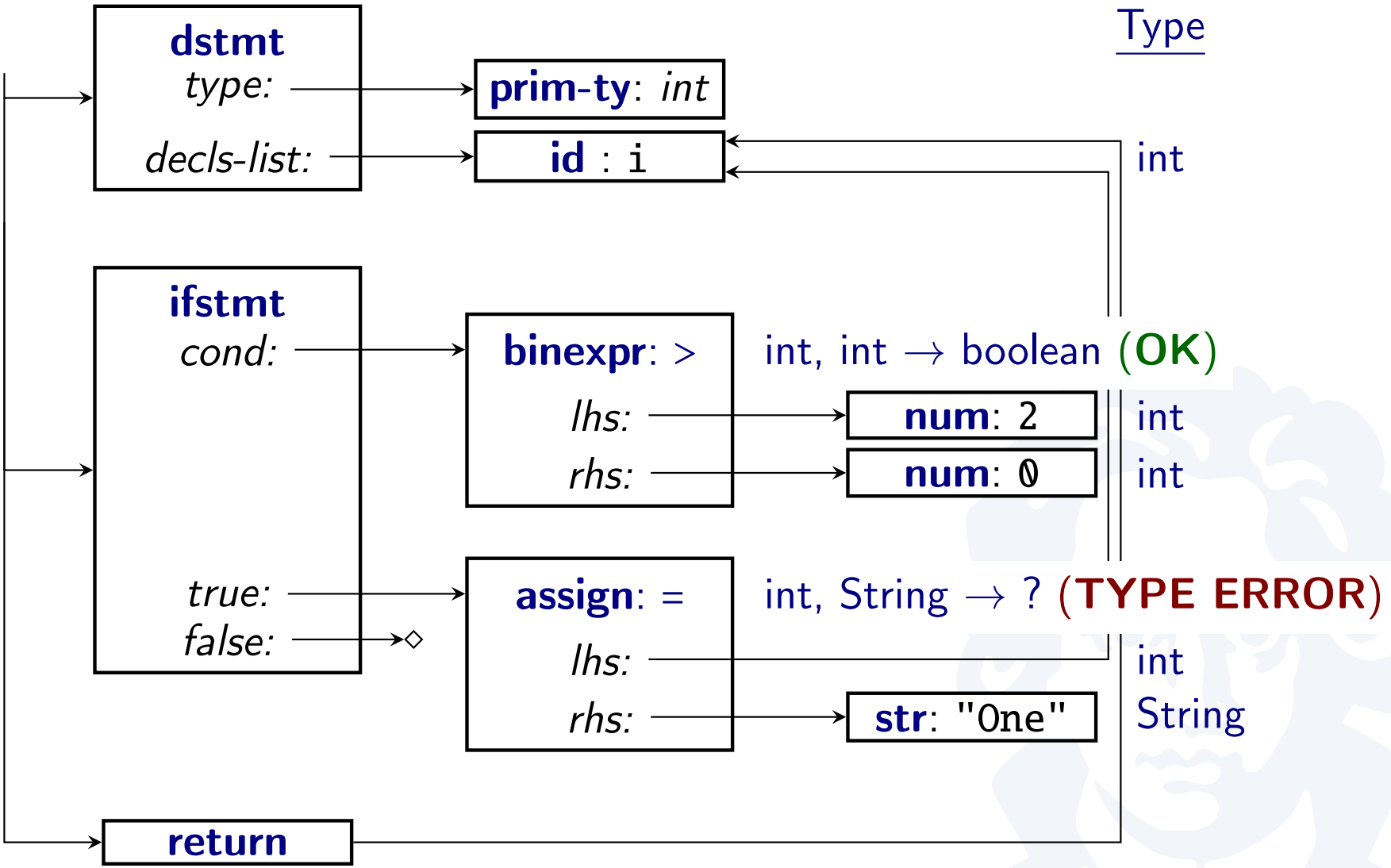
Type Analysis



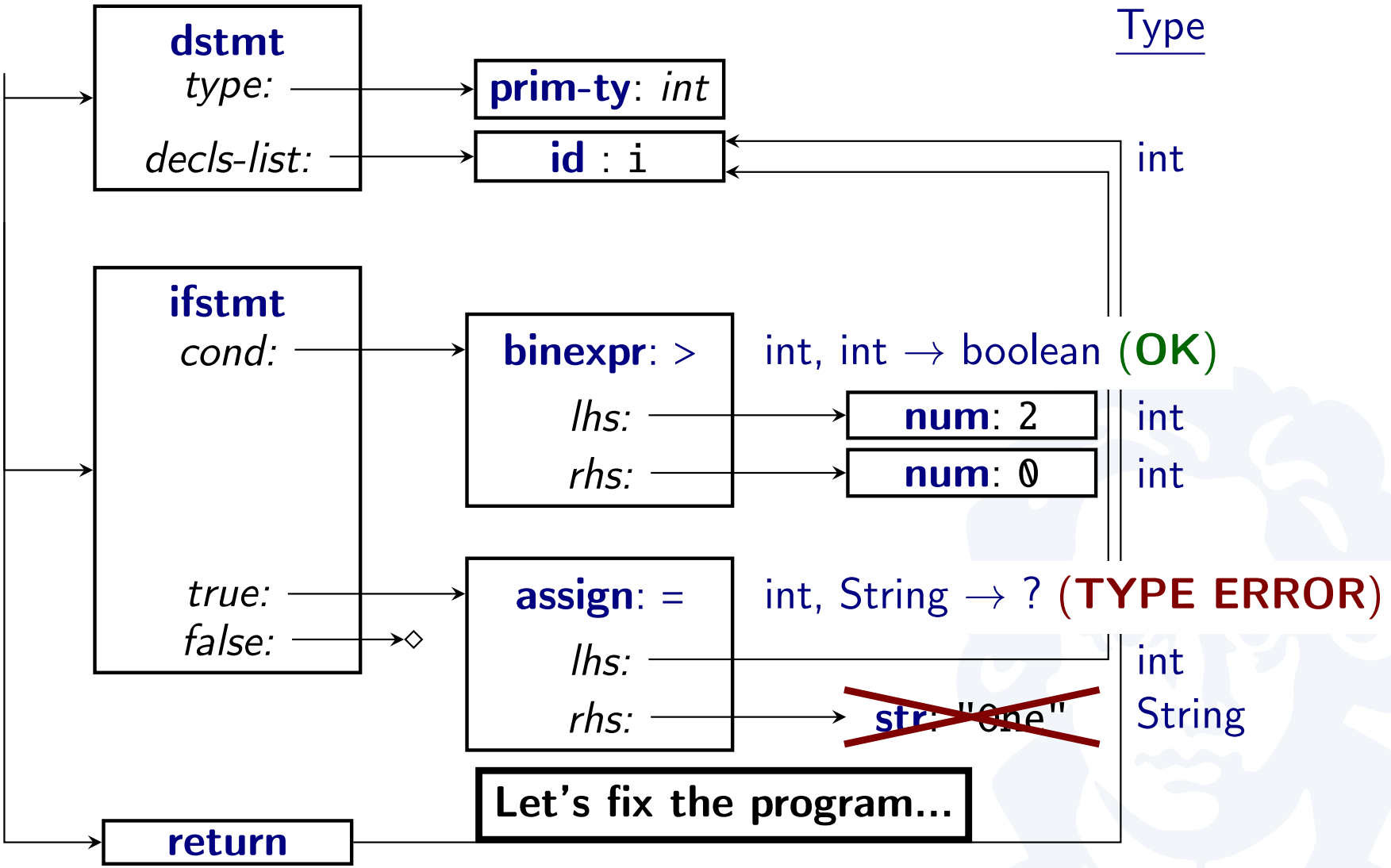
Type Analysis



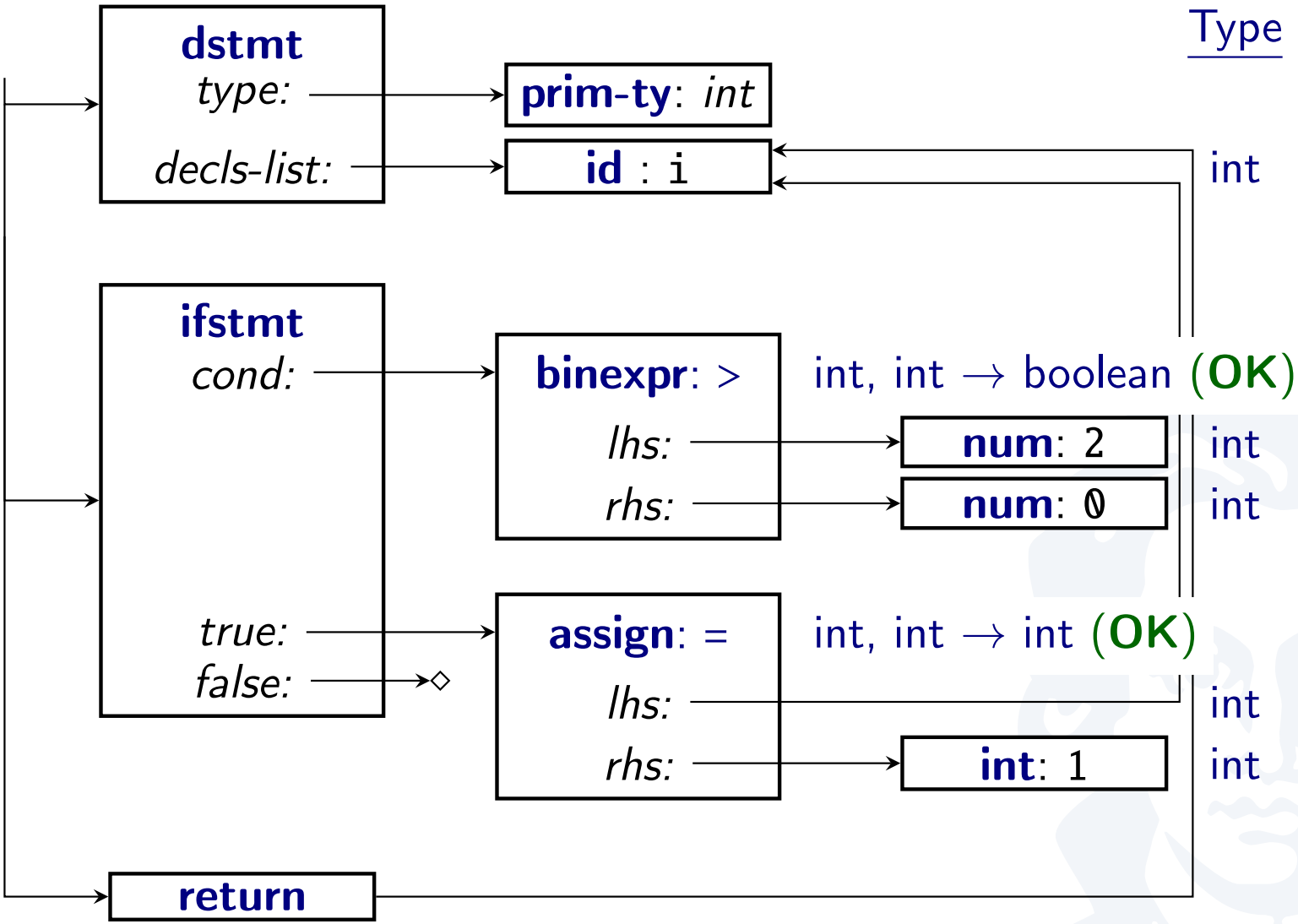
Type Analysis



Type Analysis



Type Analysis



Summary

- Compiler represents programs in *intermediate representations* (IRs)
- Compiler can be separated into:
  - *Frontend*: process incoming source code, generate IR
  - *Middle-end*: optimise IR
  - *Back-end*: translate IR into executable code
- Parser matches *concrete syntax tree* (CST), generates *abstract syntax tree* (AST)
- Typical analyses on AST:
  - *Name analysis*: which variable use belongs to which definition?
  - *Type analysis*: do variable/operator/function types agree? Any implicit conversions needed?