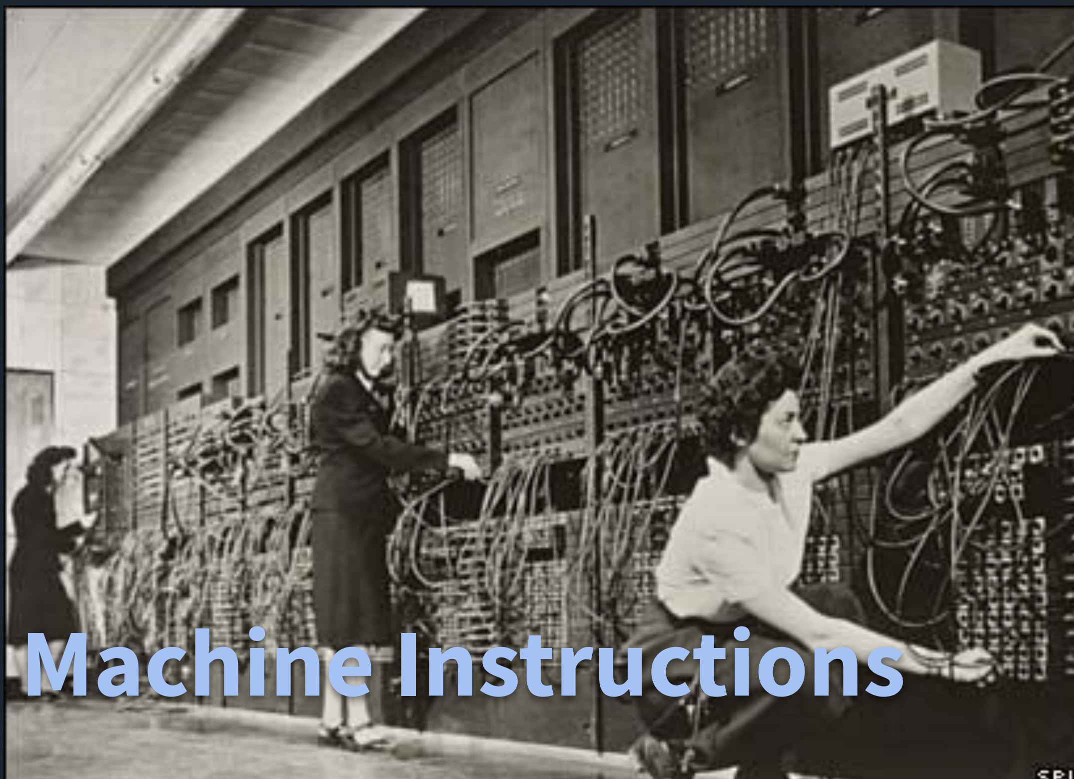
Building a Static Type-Inferring Compiler

Please follow these instructions to get set up:

`github.com/rtfeldman/compiler-workshop-v1` roc-lang.org

zed.dev

Before There Were Compilers



1946 UPenn

ENIAC

Electronic
Numerical
Integrator
And
Computer

Source Representation

Runtime
Interpreter

Ahead-of-Time
Compiler

“Assembler”

Machine Instructions

`01110010011011110110011`

Source Representation

Runtime Interpreter

SH JS

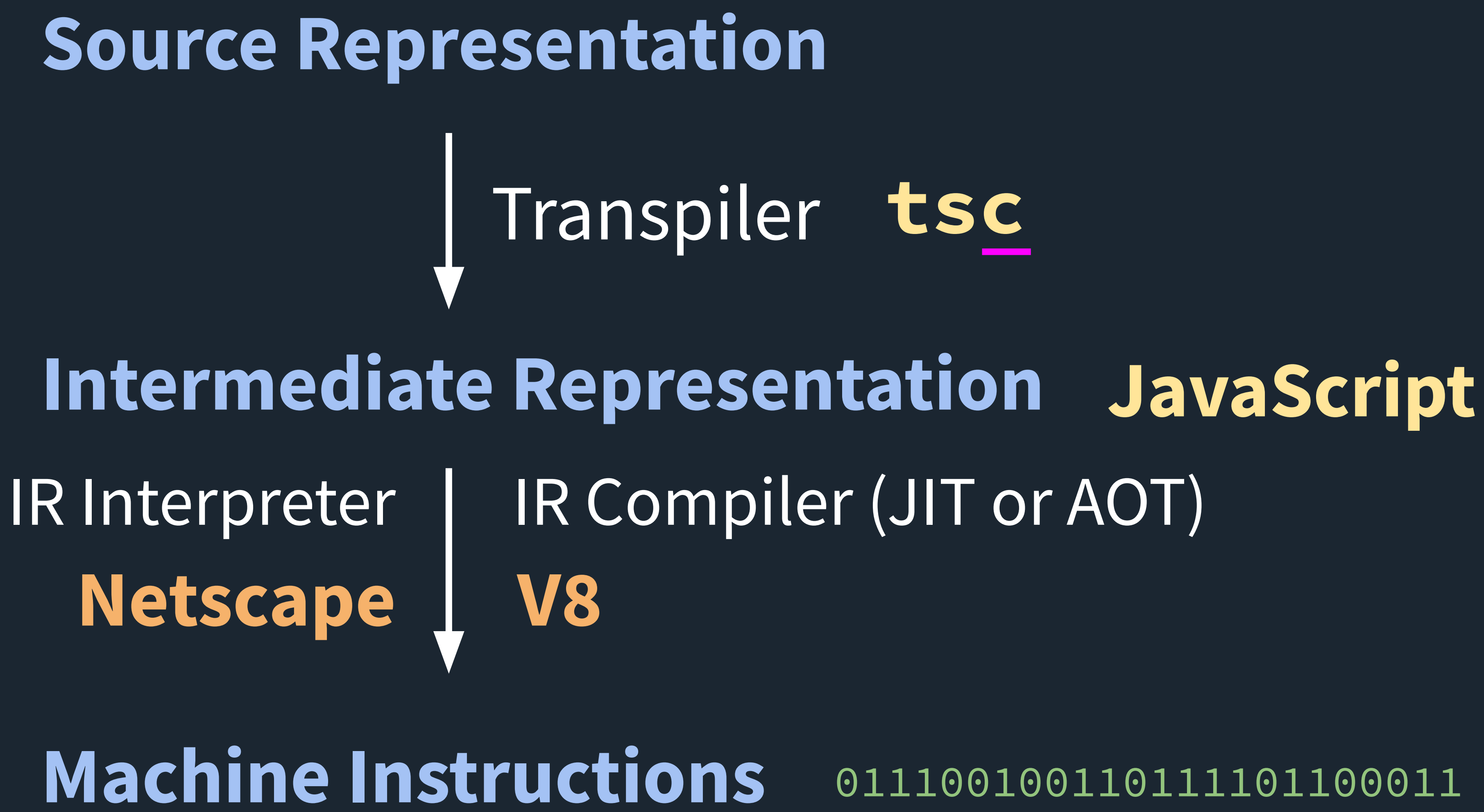
Just-in-Time Compiler

JS WA

Streaming WebAssembly Compilation

Machine Instructions

011100100110111101100011


Source Representation
Transpiler
tsc
Intermediate Representation
JavaScript
IR Interpreter
IR Compiler (JIT or AOT)
Netscape
V8
Machine Instructions
01110010011011110110011

Source Representation

Bytecode
Compiler

Java

Intermediate Representation

JVM
Bytecode
01101010…

IR Interpreter
JVM

IR Compiler
HotSpot

Machine Instructions

01110010011011110110011

Source Representation

Bytecode
Compiler

This Course!

Intermediate Representation

WebAssembly Bytecode

01110111…

IR Interpreter
Browser

IR Compiler
Browser

Machine Instructions

0111001001101111101100011

Terminology

Assembler (Low-Level → Low-Level)

Transpiler (High-Level → High-Level)

Interpreter (Generates at runtime)

Compiler (Take input, generate output)

Analysis

Names

Types

Optimization

Course Overview

Compiler for small subset of JS

Formatting

Name Resolution

Type Inference (Hindley-Milner, not TypeScript!)

WASM Generation

Part 1: Formatting

Lexing

Parsing

Formatting

Lexing

```
const x = 5 // x is 5

{
  type: "CONST",
  value: null,
  position: 0,
}
```

Lexing

```
const x = 5 // x is 5

{
  type: "IDENTIFIER",
  value: "x",
  position: 6,
}
```

Lexing

```
const x = 5 // x is 5

{
  type: "EQUALS",
  value: null,
  position: 8,
}
```

Lexing

```
const x = 5 // x is 5

{
  type: "NUMBER",
  value: "5",
  position: 10,
}
```

Lexing

```
const x = 5 // x is 5

{
  type: "COMMENT",
  value: " x is 5",
  position: 12,
}
```

Formatting

```
{
  type: "CONST",
  value: null,
  position: 0,
}

"const "
```

Formatting

```
{
  type: "IDENTIFIER",
  value: "x",
  position: 6,
}

"const "
```

Formatting

```
{
  type: "IDENTIFIER",
  value: "x",
  position: 6,
}

"const x"
```

Formatting

```
{
  type: "EQUALS",
  value: null,
  position: 8,
}

"const x"
```

Formatting

```
{
  type: "EQUALS",
  value: null,
  position: 8,
}

"const x ="
```

Formatting

```
{
  type: "NUMBER",
  value: 5,
  position: 10,
}

"const x ="
```

Formatting

```
{
  type: "NUMBER",
  value: 5,
  position: 10,
}

"const x = 5"
```

Formatting

```
{
  type: "COMMENT",
  value: " x is 5",
  position: 12,
}

"const x = 5"
```

Formatting

```
{
  type: "COMMENT",
  value: " x is 5",
  position: 12,
}

"const x = 5 // x is 5"
```

Formatting

```
const x = 5 // x is 5
```

```
"const x = 5 // x is 5"
```

Formatting

```
const      x   =    5      // x is 5
```

```
"const x = 5 // x is 5"
```

Formatting Newlines

```
{
    a: 1,
    b: 2,
    c: 3,
}
```

```
{ a: 1, b: 2, c: 3 }
```

how do we know back here to do multiline mode?

Tokens → Parse Tree

```
const x = 5 // x is 5
```

```
CONST
IDENTIFIER "x"
EQUALS
NUMBER 5
COMMENT "…"
```

Tokens → Parse Tree

```
const x = 5 // x is 5
```

```
CONST
IDENTIFIER "x"
EQUALS
NUMBER 5
```

```
{
  type: "ConstDecl",
  id: "x",


}
```

Tokens → Parse Tree

```
const x = 5 // x is 5
```

```
CONST
IDENTIFIER "x"
EQUALS
NUMBER 5
```

```
{
  type: "ConstDecl",
  id: "x",
  value: {


  },
}
```

Tokens → Parse Tree

```
const x = 5 // x is 5
```

```
CONST
IDENTIFIER "x"
EQUALS
NUMBER 5
```

```
{ Parse Tree Node
  type: "ConstDecl",
  id: "x",
  value: {
      type: "Number",
      value: "5",
  },
}
```

Formatting Newlines

```
{
  a: 1,
  b: 2,
  c: 3,
}
```

```
{ a: 1, b: 2, c: 3 }
```

Q: How do we know back here to do multiline mode?

A: Set `isMultiline: true` in the parse tree node when we see the comma token at the end

Pipeline

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Syntax Sugar **Formatter**

```
x+++      IDENTIFIER "x"
          PLUS
          PLUS
{         PLUS
  type: "PlusPlusPlus",
  identifier: "x",
}
```

```
"x = x + 2"
```

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Syntax Sugar

```
x+++      IDENTIFIER "x"
          PLUS
          PLUS
          PLUS
{
  type: "PlusPlusPlus",
  identifier: "x",
}
```

~~Formatter~~ Transpiler

```
"x = x + 2"
```

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Summary of Part 1

Lexing (input string, output tokens)

Parsing (input tokens, output parse tree)

Formatting (input parse tree, output string)

Part 1 Exercises

Open `exercises/1/formatter.js`

See comments with 👉 in them

Try doing what those comments say!

Part 2: Name Resolution

Declarations & Lookups

Hoisting

Nested Scopes

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Declarations & Lookups

```
const x = 5

const y = x + 1

if (scope.has("y")) {
    // duplicate const
}
```

```
const scope = new Set()

scope.add("x")

if (!scope.has("x")) {
    // naming error
}

scope.add("y")
```

Nested Scopes

```
const inc = (n) => {
  return n + 1
}



const x = 5

const y = inc(3)
```

```
const scope = new Set()

scope.add("inc")
scope.add("n")
scope.add("x") ⚠️
scope.add("y") ⚠️
```

Nested Scopes

```
const inc = (n) => {
  return n + 1
}


const x = 5

const y = inc(3)
```

```
const scope = new Set()
scopes.push(scope)
scope.add("inc")
const inner = new Set()
scopes.push(inner)
inner.add("n")

if (!scopes.some(s =>
  s.has("n")
)) { /* naming error */

scopes.pop()
```

Hoisting

```
const x = 5

const y = inc(3)

function inc(arg) {
  return arg + 1
}
```

```
const scope = new Set()
scopes.push(scope)
scope.add("x")
scope.add("y")

if (!scopes.some(scope =>
  scope.has("inc")
)) { /* naming error */ }
```

Hoisting Pass

```
const x = 5

const y = inc(3)



function inc(arg) { … }
```

```
const scope = new Set()
scopes.push(scope)
scope.add("inc")


if (!scopes.some(scope =>
  scope.has("inc")
)) { /* naming error */ }
```

Hoisting Pass

```
const x = 5

const y = inc(3)

function inc(arg) { … }
```

```
const scope = new Set()
scopes.push(scope)
scope.add("inc")
scope.add("x")

if (!scopes.some(scope =>
  scope.has("inc")
)) { /* naming error */ }
```

Hoisting Pass

```
const x = 5

const y = inc(3)



function inc(arg) { … }
```

```
const scope = new Set()
scopes.push(scope)
scope.add("inc")
scope.add("x")

if (!scopes.some(scope =>
  scope.has("inc")
)) { /* naming error */ }

scope.add("y")
```

Hoisting Pass

```
const x = 5

const y = inc(3)



function inc(arg) { … }
```

```
const scope = new Set()
scopes.push(scope)
scope.add("inc")
scope.add("x")

if (!scopes.some(scope =>
  scope.has("inc")
)) { /* naming error */ }

scope.add("y")
const inner = new Set()
```

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Code Gen}}$ Output String

Optional: intermediate representation

$$\text{Source String} \xrightarrow{\text{Lexing}} \text{Tokens}$$

$$\text{Tokens} \xrightarrow{\text{Parsing}} \text{Parse Tree}$$

$$\text{Parse Tree} \xrightarrow{\text{Naming}} \text{Canonical IR}$$

$$\text{Canonical IR} \xrightarrow{\text{Code Gen}} \text{Output String}$$

Summary of Part 2

Declarations & Lookups (adding to scope set)

Hoisting (extra pass just for functions)

Nested Scopes (removing entries from scope)

Part 2 Exercises

Open `exercises/2/naming.js`

See comments with 👉 in them

Try doing what those comments say!

Part 3: Inferring Constants

Type Variables

Type Relationships

Path Compression

Why Hindley-Milner Inference?

Used by languages like Elm, Haskell, Roc…

Simple type system (no co/contravariance etc.)

Sound (no runtime type mismatches)

Decidable (no annotations required, ever)

Polymorphic (supports generics)

Inferring Constants

```
const a = "hi"
const b = a
const c = b
```

Goal: infer that c is a String

Types Database

```
const a = "hi"
const b = a
const c = b
```

id	type	
0	null	const a
1	null	"hi"
2	null	const b
3	null	const c

Assign type variables

Types Database

```
const a = "hi"
const b = a
const c = b
```

id	type	
0	null	const a
1	null	"hi"
2	null	const b
3	null	const c

Assign type relationships

Types Database

```
const a = "hi"
const b = a
const c = b
```

id	type	
0	db[1]	const a
1	null	"hi"
2	null	const b
3	null	const c

Assign type relationships

Types Database

```
const a = "hi"
const b = a
const c = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	null	const b
3	null	const c

Assign type relationships

Types Database

```
const a = "hi"
const b = a 0
const c = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	null	const c

Assign type relationships

Types Database

```
const a = "hi"
const b = a
const c = b  2
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[2]	const c

Assign type relationships

Types Database

```
const a = "hi"
const b = a
const 3 = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[2]	const c

Goal: infer that c is a String

Tokens → Parse Tree → Types

```
const a = "hi"
```

Tokens

```
{                              {
  type: "CONST",                 type: "IDENTIFIER",
  value: null,                   value: "a",
  position: 0,                   position: 6,
}                              }
```

Tokens → Parse Tree → Types

```
const a = "hi"

{ Parse Tree Node
  type: "ConstDecl",
  id: "a",
  value: {
      type: "String",
      value: "hi",
  },
}
```

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	null
1	null
2	null
3	null

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	String
2	db[0]
3	db[2]

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	null
1	null
2	null
3	null

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

```
db[0] = null
db[1] = null
db[2] = null
db[3] = null
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	null
2	null
3	null

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

```
db[0] = { symlink: 1 }
db[1] = null
db[2] = null
db[3] = null
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	String
2	null
3	null

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

```
db[0] = { symlink: 1 }
db[1] = { concrete: "String" }
db[2] = null
db[3] = null
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

id	type
0	db[1]
1	String
2	db[3]
3	null

```
db[0] = { symlink: 1 }
db[1] = { concrete: "String" }
db[2] = { symlink: 0 }
db[3] = null
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	String
2	db[3]
3	db[0]

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

```
db[0] = { symlink: 1 }
db[1] = { concrete: "String" }
db[2] = { symlink: 0 }
db[3] = { symlink: 2 }
```

Path Compression

```
const a = "hi"
const b = a
const 3 = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[2]	const c

Goal: infer that c is a String

Path Compression

```
const a = "hi"
const b = a
const 3 = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[2]	const c

Goal: infer that c is a String

Path Compression

```
const a = "hi"
const b = a
const 3 = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[0]	const c

Goal: infer that c is a String

Path Compression

```
const a = "hi"
const b = a
const 3 = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[0]	const b
3	db[1]	const c

Goal: infer that c is a String

Path Compression

```
const a = "hi"
const b = a
const c = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[1]	const b
3	db[1]	const c

Unification, not Subtyping

Editor Tooling

```
String
const a = "hi"
const b = a
const c = b
```

id	type	
0	db[1]	const a
1	String	"hi"
2	db[1]	const b
3	db[1]	const c

1. Find source position of c
2. Find that parse tree node
3. Get that node's type ID
4. Look up that ID in the DB

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Summary of Part 3

Type Variables

Type Relationships

Path Compression

Part 3 Exercises

Open `exercises/3/typing.js`

See comments with 👉 in them

Try doing what those comments say!

Part 4: Type Unification

Unifying Operators

Type Mismatches

The `unify()` function

Type Unification

A way to say "these 2 types should be **identical**"

A way to impose **multiple constraints** on types

A way to detect **type mismatches** at build time

Unifying Operators

```
const a = "hi"
const b = a

a * b
```

id	type
0	null
1	null
2	null

Goal: infer two **type mismatches**

Unifying Operators

```
const a = "hi"
const b = a
a * b
```

id	type
0	null
1	null
2	null
3	null

Goal: infer two **type mismatches**

Unifying Operators

```
const a = "hi"
const b = a
```

```
a * b
```

id	type
0	db[1]
1	null
2	null
3	null

Goal: infer two **type mismatches**

Unifying Operators

```
          0        1
const a = "hi"
const b = a
          2     0
0     2
a * b
------
   3
```

id	type
0	db[1]
1	String
2	null
3	null

Goal: infer two **type mismatches**

Unifying Operators

```
const a = "hi"
const b = a

a * b
```

id	type
0	db[1]
1	String
2	db[0]
3	null

Goal: infer two **type mismatches**

Unifying Operators

```
const a = "hi"
const b = a
a * b
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Goal: infer two

type mismatches

Unifying Operators

```
const a = "hi"
const b = a
```

$$\frac{a \;*\; b}{\;}$$

id	type
0	db[1]
1	String
2	db[0]
3	Number

Unify both * operator args with the Number type

Unifying Operators

```
unify(null, null) → symlink
unify("String", "String") → "String"
unify("Number", "Number") → "Number"
unify("Number", "String") →
```

Unify both `*` operator args with the `Number` type

Unifying Operators

```
unify(null, null) → symlink
unify("String", "String") → "String"
unify("Number", "Number") → "Number"
unify("Number", "String") → TYPE MISMATCH!
```

Unify both `*` operator args with the `Number` type

Unifying Operators

```
const a = "hi"
const b = a
```

```
0     2
a  *  b
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Unify both * operator args with the Number type

Unifying Operators

```
const a = "hi"
const b = a
0     2
a  *  b

unify("Number",
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Unifying Operators

```
const a = "hi"
const b = a
0     2
a  *  b

unify("Number", "String") →
    TYPE MISMATCH!

unify("Number",
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Unifying Operators

```
const a = "hi"
const b = a
0   2
a * b

unify("Number", "String") →
    TYPE MISMATCH!
unify("Number", "String") →
    TYPE MISMATCH!
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Unifying Operators

```
const a = "hi"
const b = a
```

$\boxed{0}$ $\boxed{2}$

```
a * b
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

Goal: infer two **type mismatches**

Tokens → Parse Tree → Types

```
const a = "hi"
```

Tokens

```
{
  type: "CONST",
  value: null,
  position: 0,
}
```

```
{
  type: "IDENTIFIER",
  value: "a",
  position: 6,
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

```
{ Parse Tree Node
  type: "ConstDecl",
  id: "a",
  value: {
      type: "String",
      value: "hi",
  },
}
```

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	null
1	null
2	null
3	null

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  id: "a",
  typeId: 0,
  value: {
      type: "String",
      value: "hi",
      typeId: 1,
  },
}
```

Tokens → Parse Tree → Types

```
const a = "hi"
```

id	type
0	db[1]
1	String
2	db[0]
3	Number

```
{ Typed Parse Tree Node
  type: "ConstDecl",
  typeId: 0,
  …
}
```

```
db[0] = { symlink: 1 }
db[1] = { concrete: "String" }
db[2] = { symlink: 0 }
db[3] = { concrete: "Number" }
```

```
const unify = (aTypeId, bTypeId) => {
  const aType = resolveSymlinksAndCompress(aTypeId)
  const bType = resolveSymlinksAndCompress(bTypeId)

  if (aType == null) {
    db[aTypeId] = (bType == null)
      ? { symlink: bTypeId }
      : { concrete: bType.concrete }
  } else if (bType == null) {
    return unify(bTypeId, aTypeId) // Swap the args
  } else if (aType.concrete != bType.concrete) {
    reportTypeMismatch(…)
  }
}
```

Summary of Part 4

Unifying Operators

Type Mismatches

The `unify()` function

Part 4 Exercises

Open `exercises/4/typecheck.js`

See comments with 👉 in them

Try doing what those comments say!

Part 5: Inferring Functions

Inferring Conditionals

Inferring Function Declarations

Inferring Function Calls

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "other"
```

must be boolean

must be the same type

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "other"
```

these must all be the same type

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "…"
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(4, 8)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool
4	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "…"

unify(4, 8)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool
4	db[8]

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "…"

unify(4, 8)
unify(5, 2)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool
4	db[8]
5	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."
```

```
unify(4, 8)
unify(5, 2)
unify(5, Bool)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool ✅
4	db[8]
5	db[2]

Inferring Conditionals

```
const a = "hi"
const cond = "not a bool"
const b = cond ? a : "…"

unify(4, 8)
unify(5, 2)
unify(5, Bool)
```

id	type
0	db[1]
1	String
2	db[3]
3	String
4	db[8]
5	db[2]

Inferring Conditionals

```
const a = "hi"
const cond = "not a bool"
const b = cond ? a : "..."

unify(4, 8)
unify(5, 2)
unify(5, Bool) TYPE MISMATCH
```

id	type
0	db[1]
1	String
2	db[3]
3	String
4	db[8]
5	db[2]

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(4, 8)
unify(5, 2)
unify(5, Bool)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool ✅
4	db[8]
5	db[2]

path compression

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "…"

unify(4, 8)
unify(5, 2)
unify(5, Bool)
```

id	type
0	db[1]
1	String
2	db[3]
3	Bool ✅
4	db[8]
5	db[3]

path compression

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(6, 0)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	null
7	null
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(6, 0)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[0]
7	null
8	null

path compression

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(6, 0)
unify(7, String)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[1]
7	null
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(6, 0)
unify(7, String)
unify(6, 7)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[1]
7	String
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."
```

```
unify(6, 0)
unify(7, String)
unify(6, 7)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	✅ String
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : 123

unify(6, 0)
unify(7, String)
unify(6, 7)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[1]
7	Number
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : 123

unify(6, 0)
unify(7, String)
unify(6, 7) TYPE MISMATCH
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[1]
7	Number
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(6, 0)
unify(7, String)
unify(6, 7)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7] ✅
7	String
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(8, 6)
unify(8, 7) ?
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	String
8	null

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(8, 6)
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	String
8	db[6]

path compression

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."

unify(8, 6)
unify(8, 7) ?
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	String
8	db[7]

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."
```

```
unify(8, 6)
```

~~unify(8, 7)~~ **not needed!**

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	String
8	db[7]

Inferring Conditionals

```
const a = "hi"
const cond = true
const b = cond ? a : "..."
```

id	type
3	Bool
4	db[8]
5	db[3]
6	db[7]
7	String
8	db[7]

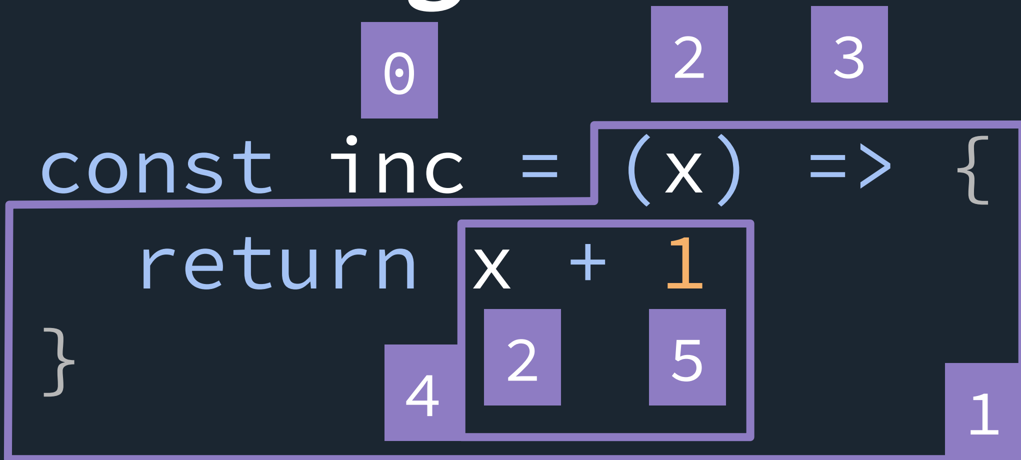
Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)           =
unify(4, Number)      +
unify(2, Number)      +
unify(5, Number)      +
unify(5, Number)      1
unify(4, 3)           return
```

id	type
0	null
1	null
2	null
3	null
4	null
5	null

Inferring Functions




```
unify(0, 1)          =
unify(4, Number)     +
unify(2, Number)     +
unify(5, Number)     +
unify(5, Number)     1
unify(4, 3)          return
```

id	type
0	db[1]
1	null
2	null
3	null
4	null
5	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)          =
unify(4, Number)     +
unify(2, Number)     +
unify(5, Number)     +
unify(5, Number)     1
unify(4, 3)          return
```

id	type
0	db[1]
1	null
2	null
3	null
4	Number
5	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)         =
unify(4, Number)    +
unify(2, Number)    +
unify(5, Number)    +
unify(5, Number)    1
unify(4, 3)         return
```

id	type
0	db[1]
1	null
2	Number
3	null
4	Number
5	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)          =
unify(4, Number)     +
unify(2, Number)     +
unify(5, Number)     +
unify(5, Number)     1
unify(4, 3)          return
```

id	type
0	db[1]
1	null
2	Number
3	null
4	Number
5	Number

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)          =
unify(4, Number)     +
unify(2, Number)     +
unify(5, Number)     +
unify(5, Number)     1
unify(4, 3)          return
```

id	type
0	db[1]
1	null
2	Number
3	null
4	Number
5	Number

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
unify(0, 1)         =
unify(4, Number)    +
unify(2, Number)    +
unify(5, Number)    +
unify(5, Number)    1
unify(4, 3)         return
```

id	type
0	db[1]
1	null
2	Number
3	db[4]
4	Number
5	Number

Inferring Functions

```
                     2   3
const inc = (x) => {
  return x + 1
}
                               1
```

```
{
    concrete: "Function",
    args: [2],
    returns: 3,
}
```

id	type
0	db[1]
1	null
2	Number
3	db[4]
4	Number
5	Number

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
{
    concrete: "Function",
    args: [2],
    returns: 3,
}
```

id	type
0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
4	Number
5	Number

Inferring Functions

```
              0         2    3
const inc = (x) => {
  return x + 1
}


inc(42)


inc("hi")
```

GOAL:

1 success

1 mismatch

id	type
0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
4	Number
5	Number

Inferring Functions

0 2 3

```
const inc = (x) => {
  return x + 1
}
```

0 6 7

```
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

0 8 9

```
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	null
7	null
8	null
9	null

Inferring Functions

```
          0          2    3
const inc = (x) => {
  return x + 1
}
```

```
 0   6   7
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
 0     8   9
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	null
8	null
9	null

Inferring Functions

```
              0         2    3
const inc = (x) => {
  return x + 1
}
```

```
 0   6     7
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
 0    8    9
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	null
8	null
9	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	db[3]
8	null
9	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	db[3]
8	String
9	null

Inferring Functions

```
        0          2    3
const inc = (x) => {
  return x + 1
}
```

TYPE MISMATCH!

```
  0    6    7
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
  0     8   9
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	db[3]
8	String
9	null

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

TYPE MISMATCH!

```
inc(42)
```

```
unify(6, Num)
unify(6, 2)
unify(7, 3)
```

```
inc("hi")
```

```
unify(8, Str)
unify(8, 2)
unify(9, 3)
```

0	db[1]
1	db[2] => db[3]
2	Number
3	db[4]
…	…
6	Number
7	db[3]
8	String
9	db[3]

Inferring Functions

```
const inc = (x) => {
  return x + 1
}
```

```
inc(42)

inc("hi")
```

GOAL:
1 success
1 mismatch

0	db[1]
1	db[2] => db[3]
2	String
3	db[4]
…	…
6	Number
7	db[3]
8	String
9	db[3]

Summary of Part 5

Inferring Conditionals

Inferring Function Declarations

Inferring Function Calls

Part 5 Exercises

Open `exercises/5/typecheck.js`

See comments with 👉 in them

Try doing what those comments say!

Part 6: Polymorphism

Polymorphic Functions

Arrays

Type Annotations

Polymorphic Functions

```
const inc = (x) => {
  return x + 1
}
```

```
inc(42)
```

```
inc("hi")
```

GOAL:

1 success

1 mismatch

Polymorphic Functions

```
const identity =
  (a) => { return a }
```

```
identity(42)
```

GOAL:

2 successes

```
identity("hi")
```

Polymorphic Functions

```
                 0
const identity =
1 (a) => { return a }
    2    3             2

unify(0, 1)          =
unify(3, 2)          return
```

id	type
0	null
1	null
2	null
3	null

...that's it!

Polymorphic Functions

```
                0
const identity =
1 (a) => { return a }
   2    3         2
```

```
unify(0, 1)          =
unify(3, 2)          return
```

id	type
0	db[1]
1	null
2	null
3	null

Polymorphic Functions

```
const identity =
(a) => { return a }
```

id	type
0	db[1]
1	null
2	null
3	db[2]

```
unify(0, 1)        =
unify(3, 2)        return

{ concrete: "Function", args: [2], returns: 3 }
```

Polymorphic Functions

```
              0
const identity =
1 (a) => { return a }
   2    3           2
```

id	type
0	db[1]
1	db[2] => db[3]
2	null
3	db[2]

```
unify(0, 1)        =
unify(3, 2)        return
```

```
{ concrete: "Function", args: [2], returns: 3 }
```

path compression

Polymorphic Functions

```
const identity =
(a) => { return a }
```

```
unify(0, 1)          =
unify(3, 2)          return
```

id	type
0	db[1]
1	db[2] => db[2]
2	null
3	db[2]

```
{ concrete: "Function", args: [2], returns: 2 }
```

Polymorphic Functions

```
                0
const identity =
1 (a) => { return a }
    2   3             2
```

id	type
0	db[1]
1	db[2] => db[2]
2	null
3	db[2]

```
identity(42)
```

GOAL:

2 successes

```
identity("hi")
```

Polymorphic Functions

```
            0
const identity =
1 (a) => { return a }
   2    3           2

4    0       5
identity(42)

6    0       7
identity("hi")
```

```
unify(4, 2)
unify(5, 2)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	null
5	null
6	null
7	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(4, 2)
unify(5, 2)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	db[2]
5	null
6	null
7	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(4, 2)
unify(5, 2)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	db[2]
5	db[2]
6	null
7	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(4, 2)
unify(5, 2)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	Number ⚠️
3	db[2]
4	db[2]
5	db[2]
6	null
7	null

Polymorphic Functions

```
const identity =
  (a) => { return a }

identity(42)

identity("hi")
```

```
unify(6, 2)
unify(7, 2)
unify(7, S)
```

0	db[1]
1	db[2] => db[2]
2	Number ⚠️
3	db[2]
4	db[2]
5	db[2]
6	null
7	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(6, 2)
unify(7, 2)
unify(7, S)
```

0	db[1]
1	db[2] => db[2]
2	Number ⚠️
3	db[2]
4	db[2]
5	db[2]
6	db[2]
7	null

Polymorphic Functions

```
               0
const identity =
1 (a) => { return a }
   2   3            2

4     0      5
identity(42)

6     0       7
identity("hi")
```

```
unify(6, 2)
unify(7, 2)
unify(7, S)
```

0	db[1]
1	db[2] => db[2]
2	Number ⚠️
3	db[2]
4	db[2]
5	db[2]
6	db[2]
7	db[2]

Polymorphic Functions

```
        0
const identity =
1 (a) => { return a }
   2  3             2

4    0      5
identity(42)

6    0      7
identity("hi")
```

unify(6, 2)
unify(7, 2)
unify(7, S)

0	db[1]
1	db[2] => db[2]
2	Number ⚠️
3	db[2]
4	db[2]
5	db[2]
6	db[2]
7	db[2]

GOAL:
2 successes

Polymorphic Functions

```
                0
const identity =
1 (a) => { return a }
   2    3           2
4     0        5
identity(42)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	null
5	null

A fresh function type, just for this call!

Polymorphic Functions

```
            0
const identity =
1 (a) => { return a }
   2    3          2

4      0      5
identity(42)
```

```
unify(4, 7)
unify(5, 7)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	null
5	null
6	db[7] => db[7]
7	null

A fresh function type, just for this call!

Polymorphic Functions

```
                  0
const identity =
1 (a) => { return a }
    2     3              2

4      0         5
identity(42)
```

unify(4, 7)
unify(5, 7)
unify(5, N)

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	db[7]
5	null
6	db[7] => db[7]
7	null

A fresh function type, just for this call!

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)
```

```
unify(4, 7)
unify(5, 7)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	db[7]
5	db[7]
6	db[7] => db[7]
7	null

A fresh function type, just for this call!

Polymorphic Functions

```
const identity =
  (a) => { return a }

identity(42)
```

```
unify(4, 7)
unify(5, 7)
unify(5, N)
```

0	db[1]
1	db[2] => db[2]
2	null
3	db[2]
4	db[7]
5	db[7]
6	db[7] => db[7]
7	Number

A fresh function type, just for this call!

Polymorphic Functions

```
              0
const identity =
1 (a) => { return a }
   2   3            2

4   0      5
identity(42)

8   0       9
identity("hi")
```

```
unify(8,11)
unify(9,11)
unify(9, S)
```

4	db[7]
5	db[7]
6	db[7] => db[7]
7	Number
8	null
9	null
10	db[11] => db[11]
11	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(8,11)
unify(9,11)
unify(9, S)
```

4	db[7]
5	db[7]
6	db[7] => db[7]
7	Number
8	db[11]
9	null
10	db[11] => db[11]
11	null

Polymorphic Functions

```
const identity =
(a) => { return a }

identity(42)

identity("hi")
```

```
unify(8,11)
unify(9,11)
unify(9, S)
```

4	db[7]
5	db[7]
6	db[7] => db[7]
7	Number
8	db[11]
9	db[11]
10	db[11] => db[11]
11	null

Polymorphic Functions

```
            0
const identity =
1 (a) => { return a }
   2   3          2
```

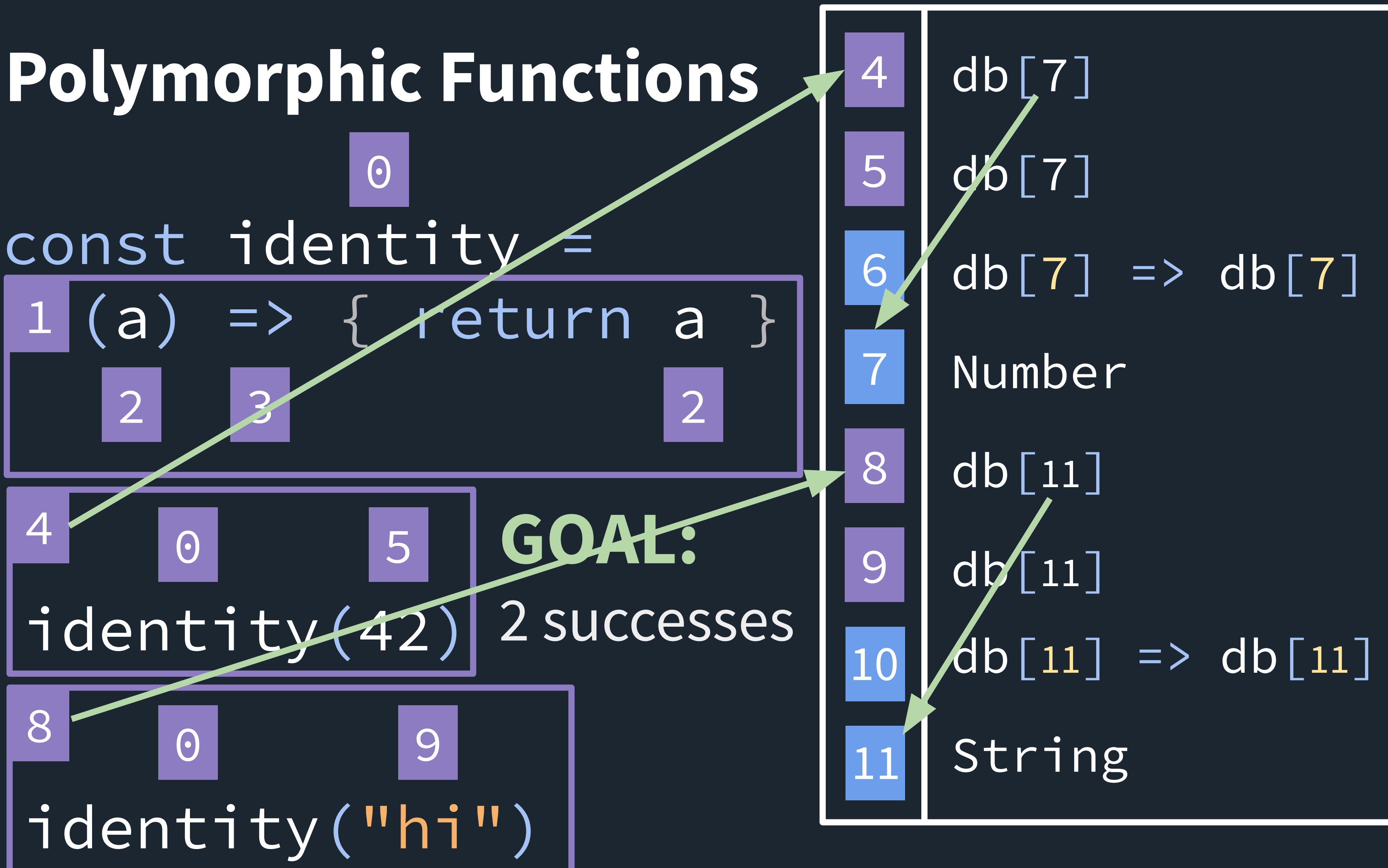
```
4    0        5
identity(42)
```

```
unify(8,11)
unify(9,11)
unify(9, S)
```

```
8    0        9
identity("hi")
```

4	db[7]
5	db[7]
6	db[7] => db[7]
7	Number
8	db[11]
9	db[11]
10	db[11] => db[11]
11	String

Polymorphic Functions



Arrays

```
               0
const concat =
   (a, b) => { … }
1   2    3    4

concat([1, 2, 3])

concat(["a", "b"])
```

0	db[1]
1	db[2], db[3] => db[4]
2	Array<5>
3	Array<5>
4	Array<5>
5	null

Arrays

```
                0
const concat =
    (a, b) => { … }
     2  3    4
1

concat([1, 2, 3])

concat(["a", "b"])
```

1	db[2], db[3] => db[4]
…	
4	Array<5>
5	null
6	db[7], db[8] => db[9]
7	Array<10>
8	Array<10>
9	Array<10>
10	Number

Polymorphic Type-Checking

Whenever calling polymorphic functions, make **fresh copies** of the function's type.

The new copies have new unbound (`null`) vars.

Call `unify()` on the copies instead of originals.

Type Annotations

```
      0          1
const x: String = "hi!"

      2         3
const y: Number = 5
```

0	null
1	null
2	null
3	null

Type Annotations

```
         0                 1
const x: String = "hi!"
             unify(0, String)

         2               3
const y: Number = 5
             unify(2, Number)
```

0	db[1]
1	String
2	db[3]
3	Number

Type Annotations

```
       0             1
const x: String = "hi!"
         unify(0, String)

       2          3
const y: Number = "hi!"
         unify(2, Number)
```

0	db[1]
1	String
2	db[3]
3	String

Principal Decidable Type Inference

This type system **always infers the correct type**, and never needs help from annotations.

It also always **infers the most generic type**. Adding annotations can never add flexibility.

Sound Type System

This type system has no runtime type mismatches.

If there are errors, they are reported at build time.

All types are **fully known at build time**.

Summary of Part 6

Polymorphic Functions

Arrays

Type Annotations

Part 6 Exercises

Open `exercises/6/typecheck.js`

See comments with 👉 in them

Try doing what those comments say!

Part 7: Generating WASM

Generating Bytecode

Code Tour

Generating Polymorphic Functions

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Generate Code as String ("Formatting")

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Parse Tree $\xrightarrow{\text{Code Gen}}$ String

Generate Code as WebAssembly

Source String	$\xrightarrow{\text{Lexing}}$	Tokens
Tokens	$\xrightarrow{\text{Parsing}}$	Parse Tree
Parse Tree	$\xrightarrow{\text{Naming}}$	Parse Tree + Scope
Parse Tree	$\xrightarrow{\text{Inference}}$	Parse Tree + Type DB
Parse Tree + Type DB	$\xrightarrow{\text{Code Gen}}$	WebAssembly Bytecode

CODE TOUR!

Generating Polymorphic Functions

A `Bool` is **1 byte** in memory. A `Number` is **8 bytes**.

This affects `Array<Bool>` and `Array<Number>`

```
push: <T>(array: Array<T>, arg: T)
```

How many bytes do we add to the end of this array?

Approach 1: Hidden Runtime Data

Whenever we call **push** with a different type for **T**,

we also pass some **hidden runtime data** about that **T** instance.

This can contain info like how many bytes T takes up.

Results in more runtime logic and memory usage.

Approach 2: Monomorphization

Whenever we call **push** with a different type for **T**,

we create a **specialized version** of the function.

Inside that function, **T** is no longer polymorphic. It's **concrete**.

Results in more functions being generated, but no runtime data.

Summary of Part 7

Generating Bytecode

Code Tour

Generating Polymorphic Functions

Part 7 Exercises

Open `exercises/7/wasm.js`

See comments with 👉 in them

Try doing what those comments say!

Wrap-Up

Recap

Further Learning

Source Representation

Runtime

Interpreter

SH JS

Ahead-of-Time

Compiler

ASM "Assembler"

Machine Instructions

01110010011011110110011

Source Representation

Runtime
Interpreter

Just-in-Time
Compiler

JS

JS

WA

Streaming
WebAssembly
Compilation

Machine Instructions

011100100110111101100011

Terminology

Assembler (Low-Level → Low-Level)

Transpiler (High-Level → High-Level)

Interpreter (Generates at runtime)

Compiler (Take input, generate output)

Compiler Passes

Tokenizing ("Lexing")

Parsing

Naming

Type Inference

Code Generation

Generate Code as String ("Formatting")

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Parse Tree $\xrightarrow{\text{Code Gen}}$ String

Generate Code as WebAssembly

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

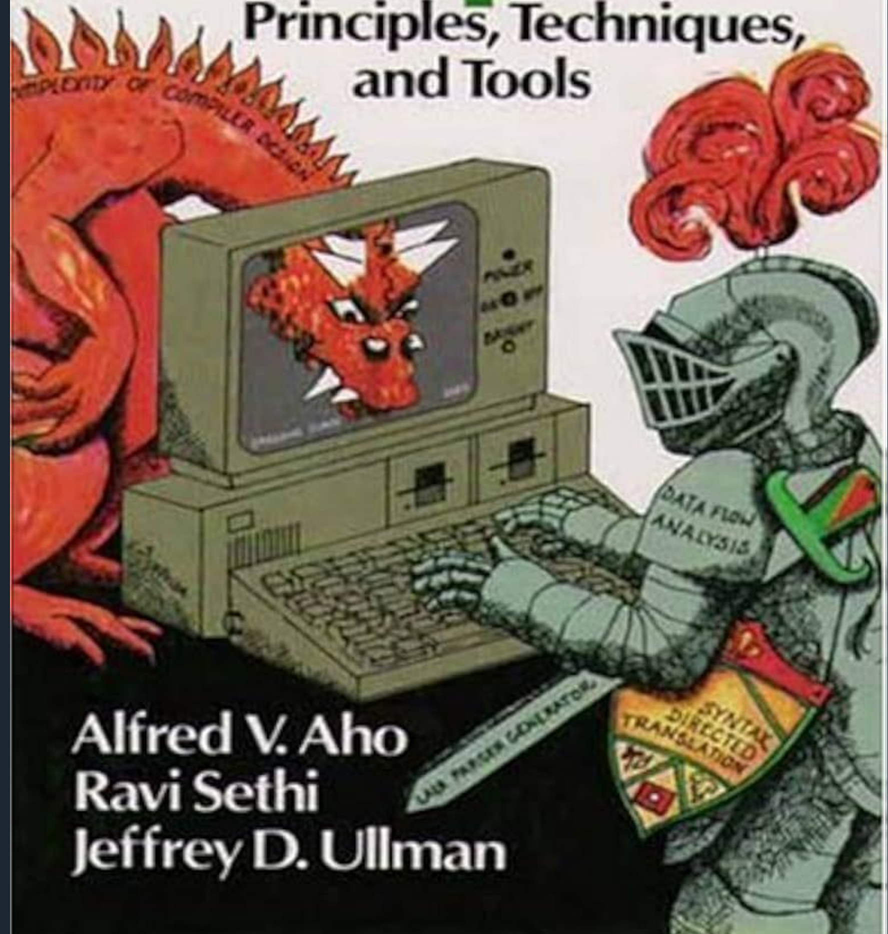
Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Parse Tree + Type DB $\xrightarrow{\text{Code Gen}}$ WebAssembly Bytecode

Compilers

Principles, Techniques, and Tools



Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

ENGINEERING A COMPILER

SECOND EDITION

Keith D. Cooper & Linda Torczon

PROGRAM

PROOF

Samuel MIMRAM

Theme: Traverse input, generate output

Source String $\xrightarrow{\text{Lexing}}$ Tokens

Tokens $\xrightarrow{\text{Parsing}}$ Parse Tree

Parse Tree $\xrightarrow{\text{Naming}}$ Parse Tree + Scope

Parse Tree $\xrightarrow{\text{Inference}}$ Parse Tree + Type DB

Parse Tree + Type DB $\xrightarrow{\text{Code Gen}}$ WebAssembly Bytecode

I host a podcast!

roc-lang.org

software-unscripted.com