# **6.110** Computer Language Engineering

## Re-lecture 2

February 21, 2024

**High-level IR ←**

Semantic Analysis

Directed Acyclic Graphs

Parse Tree

Abstract Syntax Tree

High-level IR

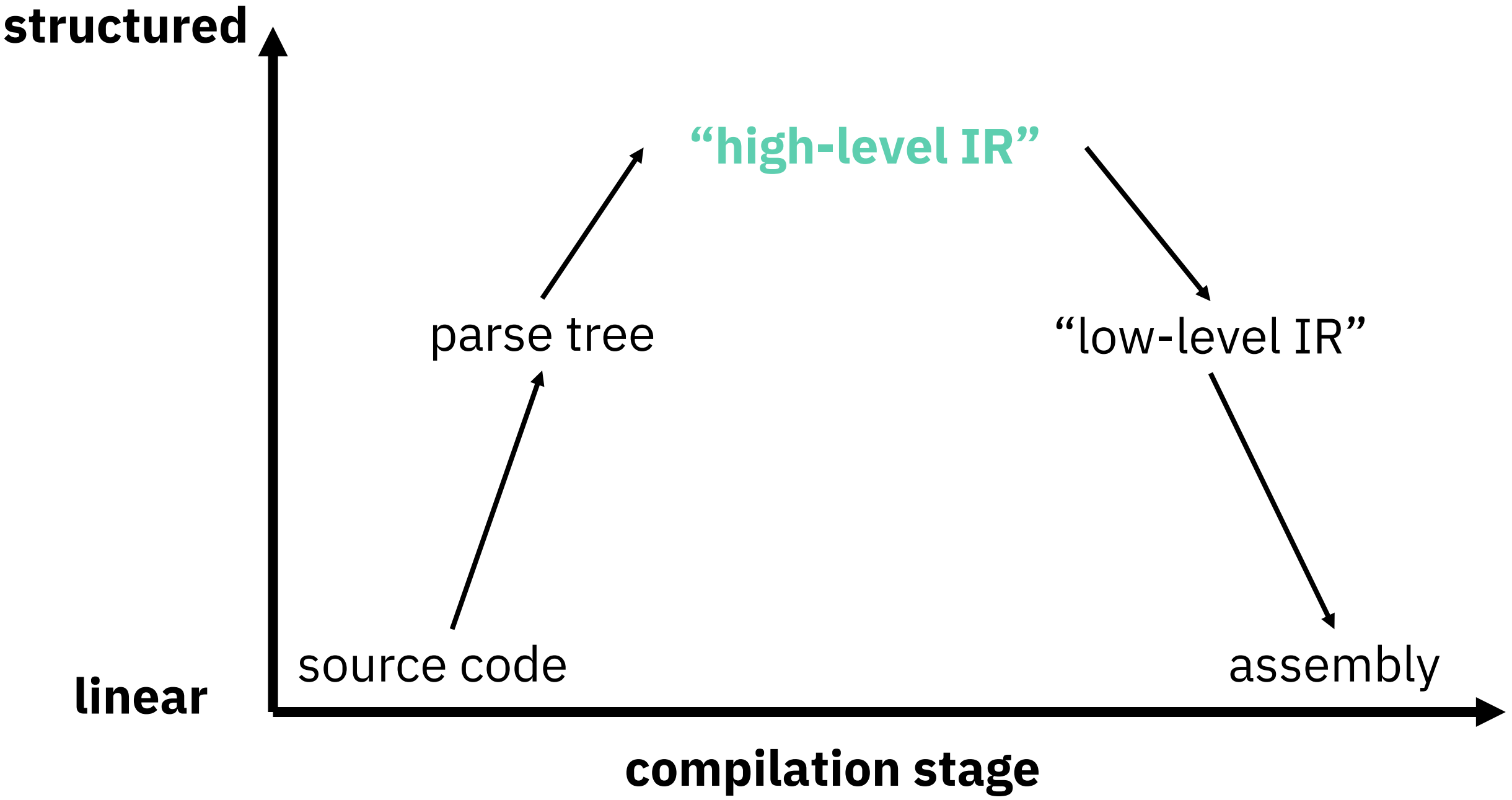
# Intermediate Representations

Basic Blocks

Single Static Assignment

Low-level IR

Control-flow Graph


structured
“high-level IR”
parse tree
“low-level IR”
source code
assembly
linear
compilation stage

# High-level IR

- Goal: **semantic checking** and **program analysis**

# High-level IR

- Goal: **understand what the code is doing**

```
x = 4 + f(true);
```

- What is `4`? What is `true`?
- What is `x`?
- What is `f`?

# Symbol tables

- Stores relevant information about each identifier

| identifier → | *descriptor* |
| --- | --- |
| `x` | local variable id 1, type int |
| `f` | method id 3, type bool → int |

# Scope

*global scope*

```
import printf;
int x = 0;
```

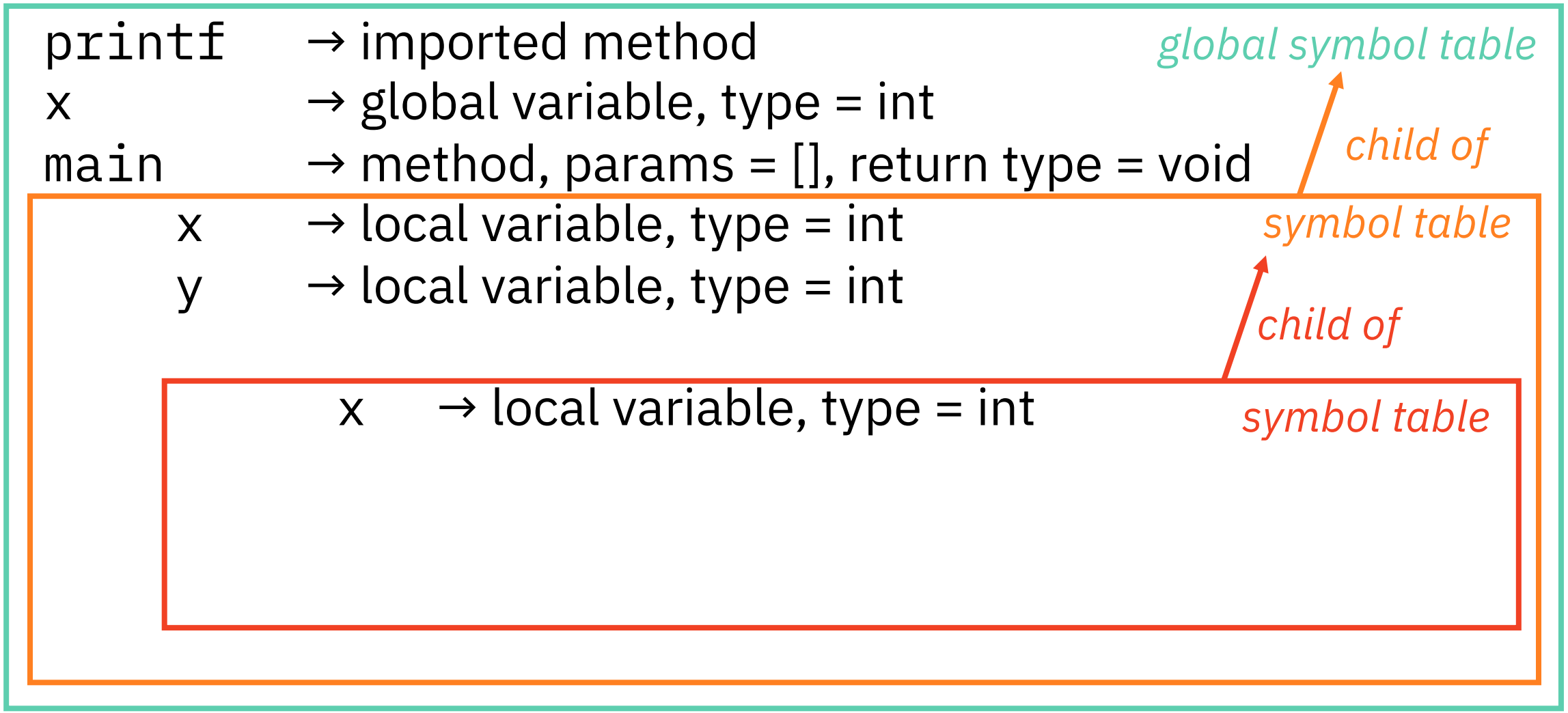
*method scope*

```
void main() {
    int x = 1, y = 2;
    if (x > 0)
```

*block scope*

```
    {
        int x = 3;
        printf("%d %d", x + y);
    }
}
```

# Symbol tables



# Scope

*global scope*

```
import printf;
int x = 0;
```

*method scope*

```
void main() {
    int x = 1, y = 2;
    if (x > 0)
    {
        int x = 3;
        printf("%d %d", x + y);
    }
}
```

*block scope*

# Summary

- One symbol table per scope
  - Each symbol table links to symbol table of parent scope
- First search for identifier in current scope
  - If not found, go to parent symbol table
  - If not found in any table, *semantic error!*

# What goes in descriptors?

- Type (or signature for methods)
- Some identifying info (e.g. name, id, stack offset)
- Information about the “children” of the node
  - *Method descriptors*: method code, symbol table for method scope
  - *Class descriptors*: symbol table for class scope

Idea: **use descriptors to go down the tree**

# What goes in symbol tables?

- Everything at that given scope
  - *Global scope:* functions, imported functions, global variables
  - *Method scope:* parameters, local variables
  - *Block scope:* local variables
  - *Class scope:* class fields, class methods
- Link to symbol table of parent scope

Idea: **use symbol tables to go up the tree**

Other designs are also possible!

# Building high-level IR

- Recursively traverse parse tree to build corresponding IR nodes
  - Structure of high-level IR will be similar to language grammar
- Build up symbol tables as you go
  - Create a symbol table for each IR node corresponding to a scope

More practical tips in Recitation and Project 2 page (coming out soon!)

# For the quiz, you should know how to:

- Explain what descriptors are and describe what information they contain
- Construct symbol tables for simple programs, including programs with simple classes
- Identify the scope of each identifier

High-level IR

**Semantic Analysis ←**

# Semantic Analysis

- We want to make sure that our program *makes sense*.
- Here are some things that don’t make sense, and how to detect them.

# Name issues

```
void main() {
    int x, x; // x is defined twice
              // in the same scope
}
```

**Detection:** check for duplicates in each symbol table

# Name issues

```
void main() {
    y = 0; // y does not exist
}
```

**Detection:** look up each identifier, and check that it is in scope

# Type errors: operations

```
4 + true   // + : (int, int) → int
4 && 5     // && : (bool, bool) → bool
false < 1  // < : (int, int) → bool
```

**Detection:** recursively determine the type of each operand

# Type errors: assignments

```
int x = false; // x is int, not bool
int y[5];
y += 4; // y is int array, not int
```

**Detection:** check that LHS and RHS of each assignment has the same type

# Type errors: constants

```
const int x; // uninitialized const
const int y = 1;
y = 2; // assignment to a const
```

**Detection:** (kinda ad-hoc)

- check that each const declaration is initialized
- check that LHS of assignments is not const

# Type errors: methods

```
int f(int x) {} // should return int
void main() {
    f(0, 1); // wrong # arguments
    f(true); // wrong argument type
    return 1; // should not return
}
```

**Detection:** check method signature

# Type compatibility

```
class A {
    int x;
}
class B extends A {
    int y;
}
```

We say

- B is compatible with A
- B is a subtype of A
- B can substitute for A

(The reverse is not true!)

# Type compatibility

```
class A {
    int x;
}
class B extends A {
    int y;
}
B f(A a);
```

```
A a;
B b;
a.y = 1;  // invalid
b.x = 0;  // valid
a = b;    // valid
b = a;    // invalid
a = f(b); // valid
```

For more type theory, take 6.5110 [6.820] or 6.5120 [6.822]

# For the quiz, you should know how to:

- Determine what semantic checks need to be done for each given statement
- Perform semantic checks on a given program
- Determine compatibility of subclasses/superclasses

# Encore: more object-oriented stuff

(See lecture slides, lectures cover this for historical reasons)