

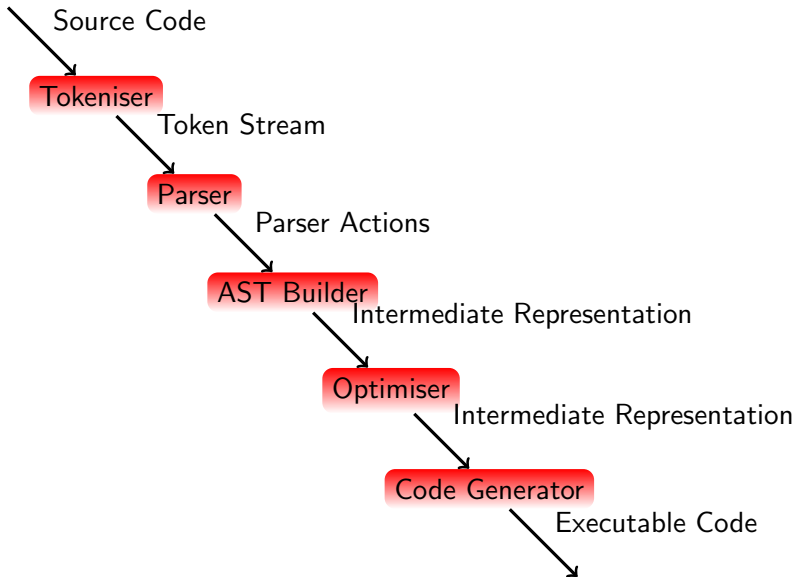
# Modern Intermediate Representations (IR)

L25: Modern Compiler Design

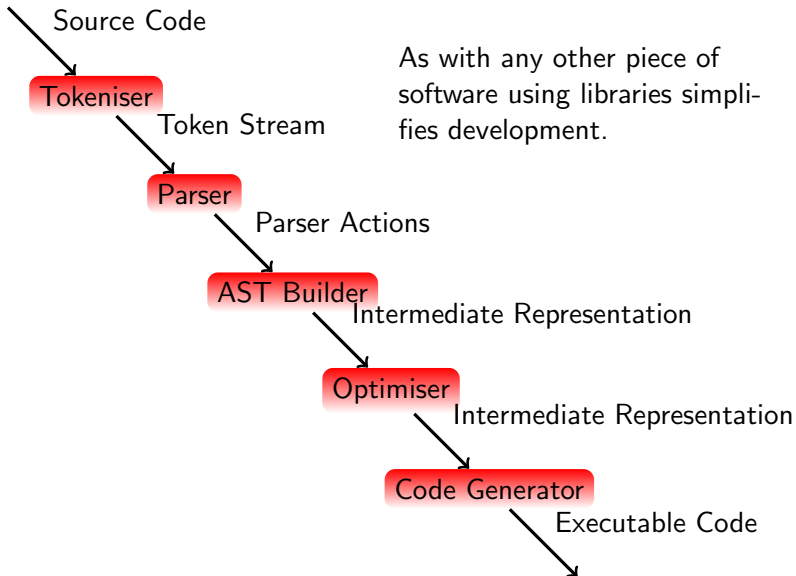
# Reusable IR

- Modern compilers are made from loosely coupled components
- Front ends produce IR
- Middle 'ends' transform IR (optimisations)
- Back ends generate native code

# Structure of a Modern Compiler



# Structure of a Modern Compiler



# Optimisation Passes

- Modular, transform IR (Analysis passes just inspect IR)
- Can be run multiple times, in different orders
- May not always produce improvements in the wrong order!
- Some intentionally pessimise code to make later passes work better

# Register vs Stack IR

- Stack makes interpreting, naive compilation easier
- Register makes various optimisations easier
- Which ones?

# Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r4 + r5  
r7 = r3 * r6  
store a r6
```

# Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r5  
r7 = r3 * r6  
store a r7
```



# Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r2  
r7 = r3 * r6  
store a r7
```

# Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r2  
r7 = r3 * r3  
store a r7
```

# Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b  
load c  
add  
load b  
load c  
add  
mul  
store a
```

# Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b  
load c  
add  
dup  
mul  
store a
```

## Problems with CSE and Stack IR

- Entire operation must happen at once (no incremental algorithm)
- Finding identical subtrees is possible, reusing results is harder
- If the operations were not adjacent, must spill to temporary

## Hierarchical vs Flat IR

- Source code is hierarchical (contains structured flow control, scoped values)
- Assembly is flat (all flow control is by jumps)
- Intermediate representations are supposed to be somewhere between the two

# Hierarchical IR

- Easy to express high-level constructs
- Preserves program semantics
- Preserves high-level semantics (variable lifetime, exceptions) clearly
- Example: WHRIL in MIPSPro/Open64/Path64 and derivatives

# Flat IR

- Easy to map to the back end
- Simple for optimisations to process
- Examples: LLVM IR, CGIR, PTX



Questions?