

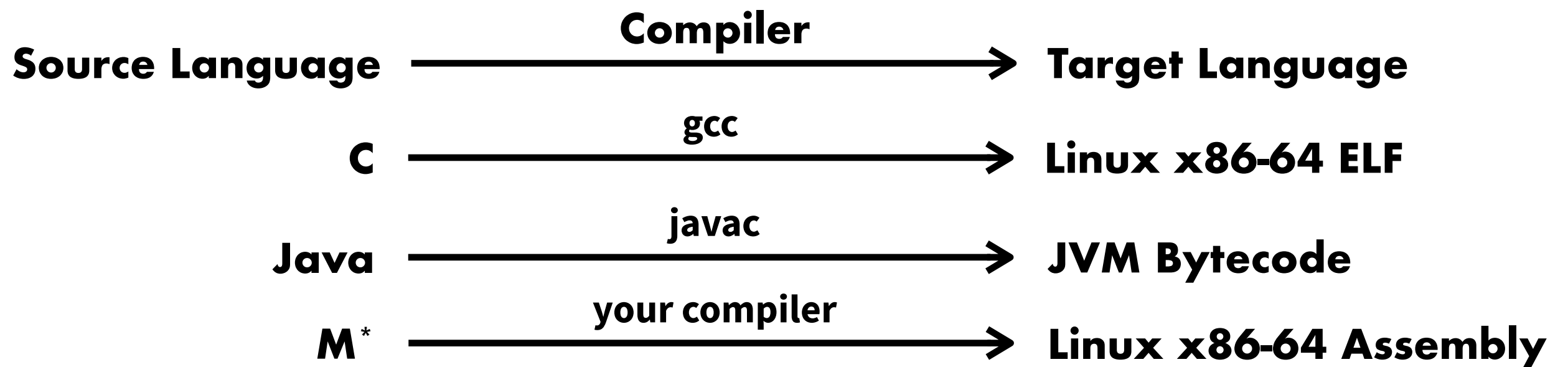
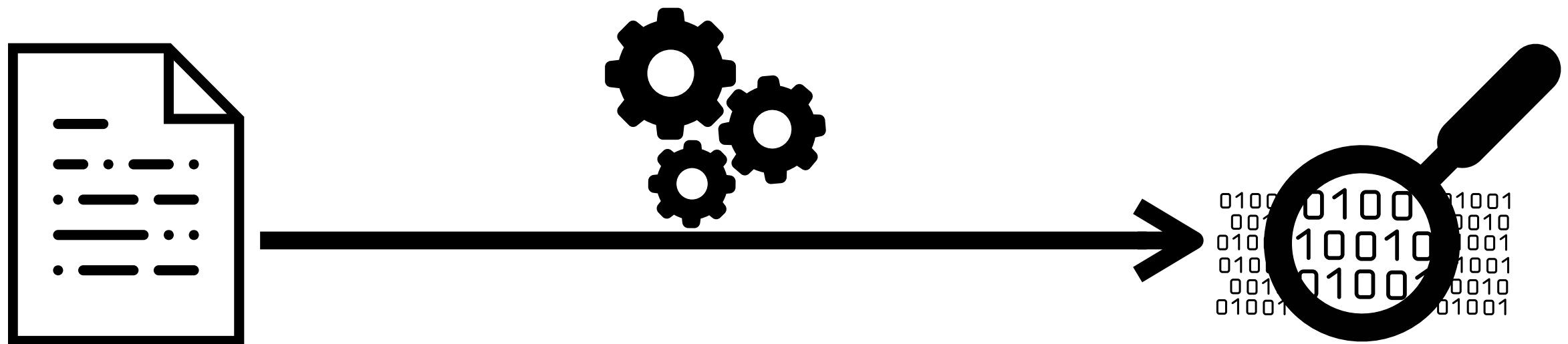
Compiler 2018: Big Picture

Lequn Chen
March 16, 2017

Why Compilers Course

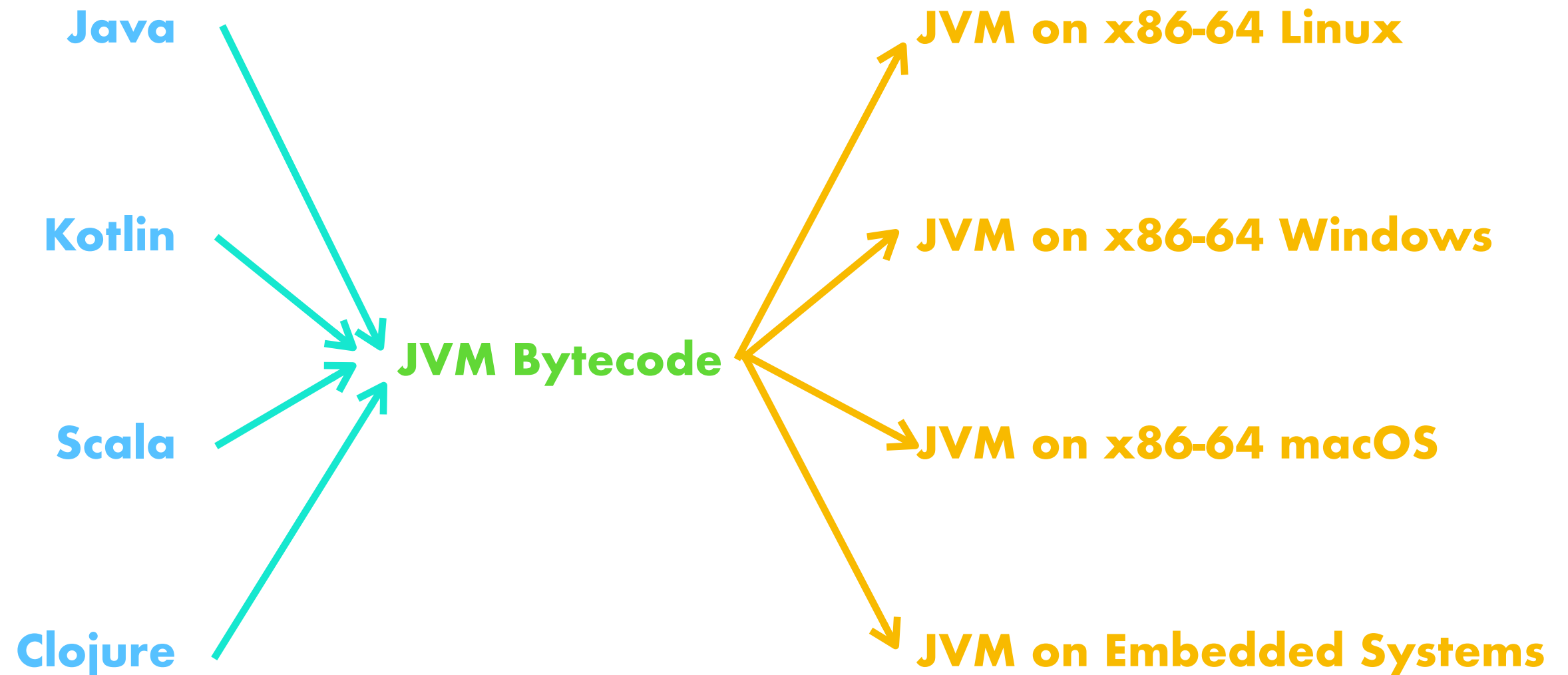
- Improve programming skills
 - ~10k loc
- Very important project experience in your CV
 - programming skills
 - perseverance

What Are Compilers



What Are Virtual Machines

- **Interpreter**
- **JIT Optimization focused on Code Generation**



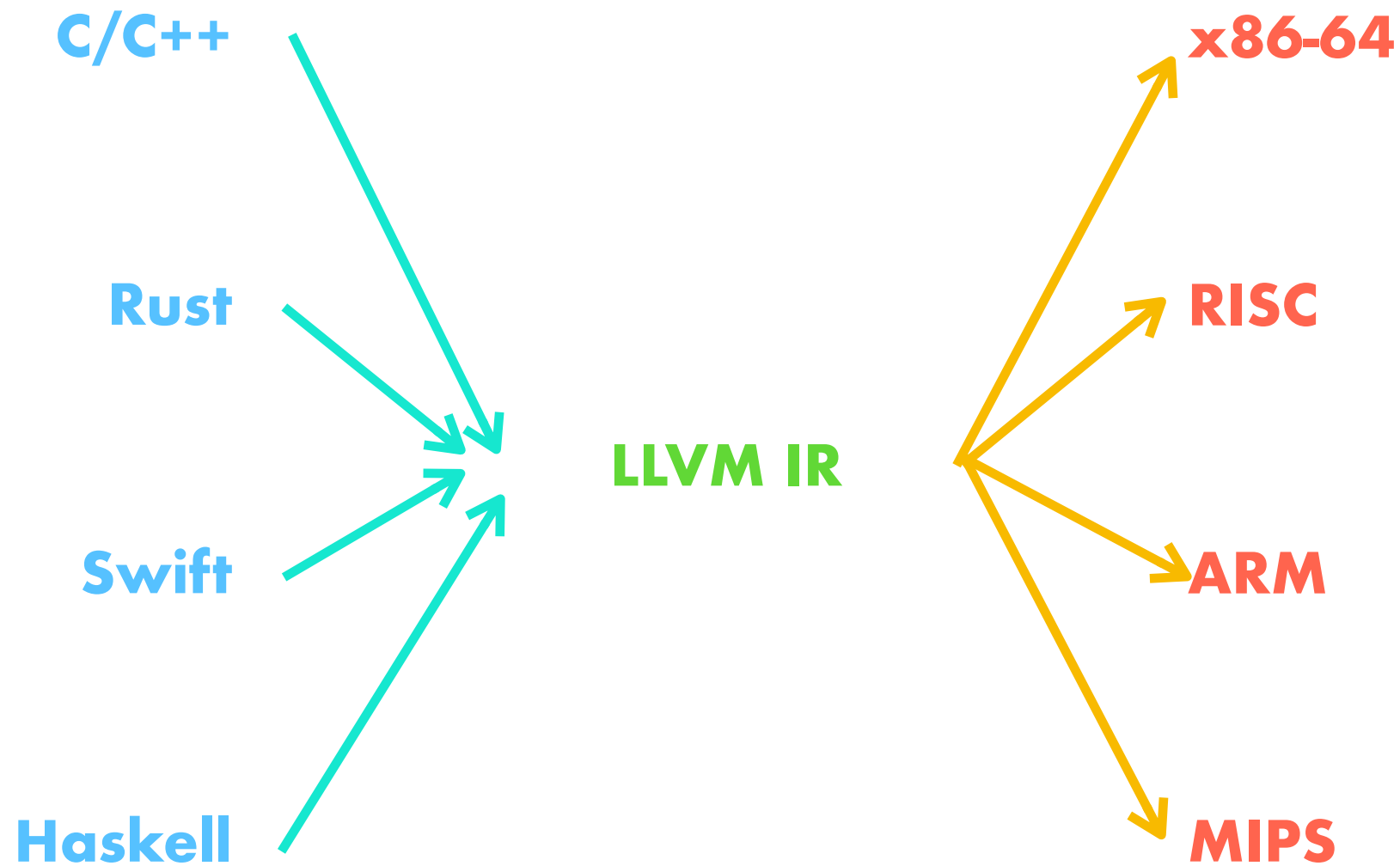
- **Language Features**
- **Optimization based on High Level Semantics**

LLVM

not *Low-Level Virtual Machine* **anymore**



- **Code Generation**
- **Transforms and Optimizations**

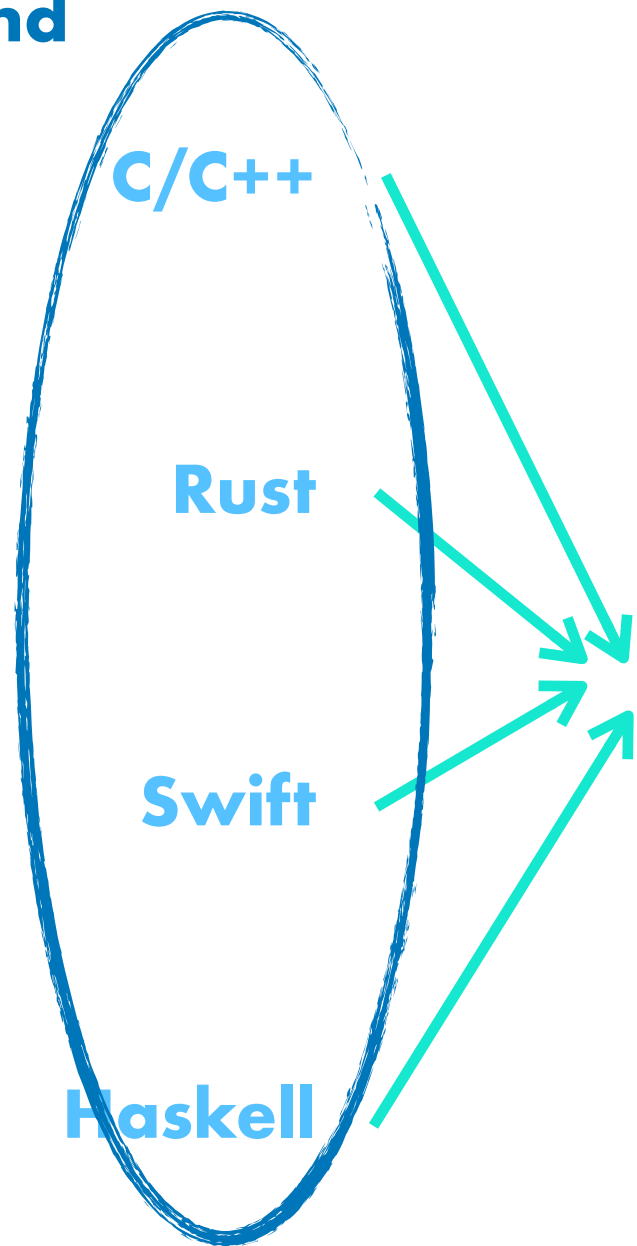


- **Language Features**
- **Optimization based on High Level Semantics**

LLVM

not Low-Level Virtual Machine **anymore**

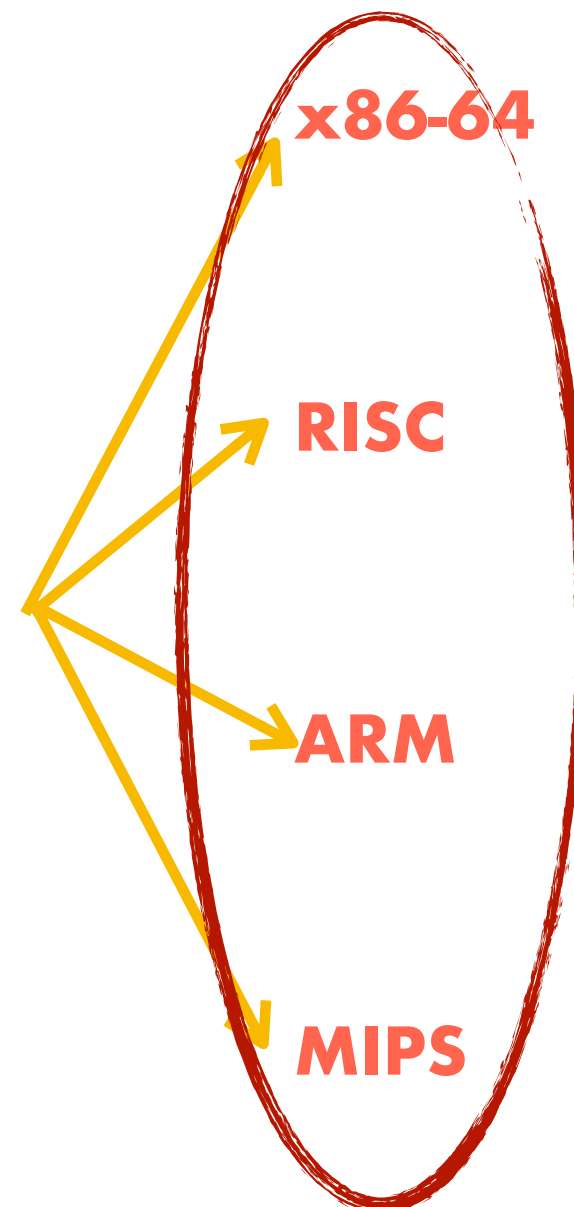
Front-end



LLVM IR

- Code Generation
- Transforms and Optimizations

Back-end



- Language Features
- Optimization based on High Level Semantics

Compilers

- Overview
 - Front-end → Intermediate Representation → Back-end
- More detail?
 - Lexing
 - Parsing
 - Semantic Analysis
 - IR Generation
 - IR Optimization
 - Code Generation
 - Target-dependent Optimizations

About the Course

- Language: whatever you want
- Lexing and parsing library: whatever you want
- Source language: M^* (C-and-Java-like)
- Target platform: Linux x86-64 Assembly in NASM
- Additional language features: whatever you want
 - as long as it is compatible with the manual
- Optimizations: whatever you want
 - as long as you can pass the tests

Lexing & Parsing

```
while f3 < 100 {  
    f3 = f1 + f2;  
    f1, f2 = f2, f3;  
}
```

Source Code

```
while f3 < 100 {  
    f3 = f1 + f2;  
    f1, f2 = f2, f3;  
}
```

Lexing

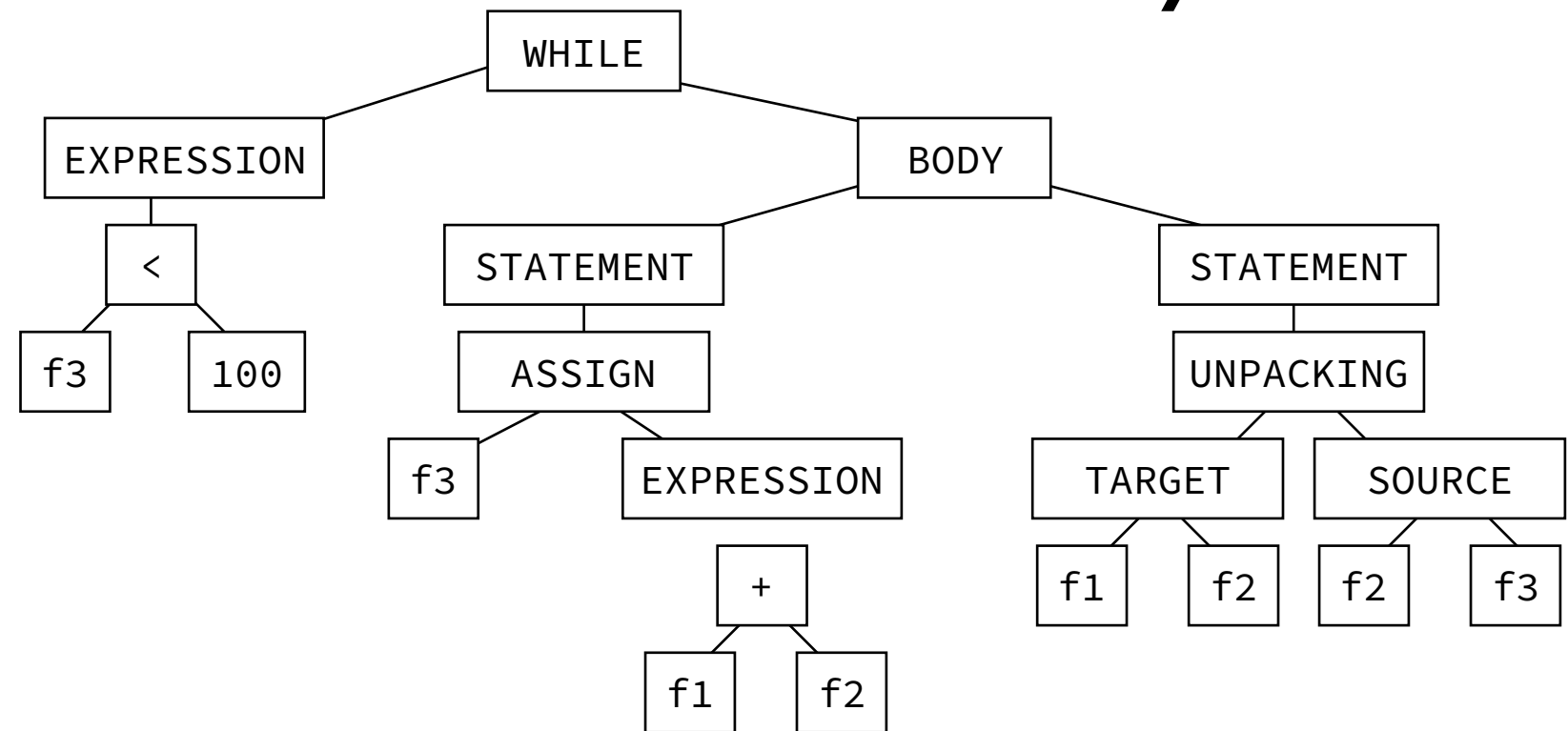
KEYWORD	while
IDENTIFIER	f3
SYMBOL	<
LITERAL	100
SYMBOL	{
IDENTIFIER	f3
SYMBOL	=
IDENTIFIER	f1
SYMBOL	+
IDENTIFIER	f2
SYMBOL	;
IDENTIFIER	f1
SYMBOL	,
IDENTIFIER	f2
SYMBOL	=
IDENTIFIER	f2
SYMBOL	,
IDENTIFIER	f3
SYMBOL	;
SYMBOL	}

```
while f3 < 100 {
    f3 = f1 + f2;
    f1, f2 = f2, f3;
}
```

Parsing

KEYWORD	while
IDENTIFIER	f3
SYMBOL	<
LITERAL	100
SYMBOL	{
IDENTIFIER	f3
SYMBOL	=
IDENTIFIER	f1
SYMBOL	+
IDENTIFIER	f2
SYMBOL	;
IDENTIFIER	f1
SYMBOL	,
IDENTIFIER	f2
SYMBOL	=
IDENTIFIER	f2
SYMBOL	,
IDENTIFIER	f3
SYMBOL	;
SYMBOL	}

Abstract Syntax Tree

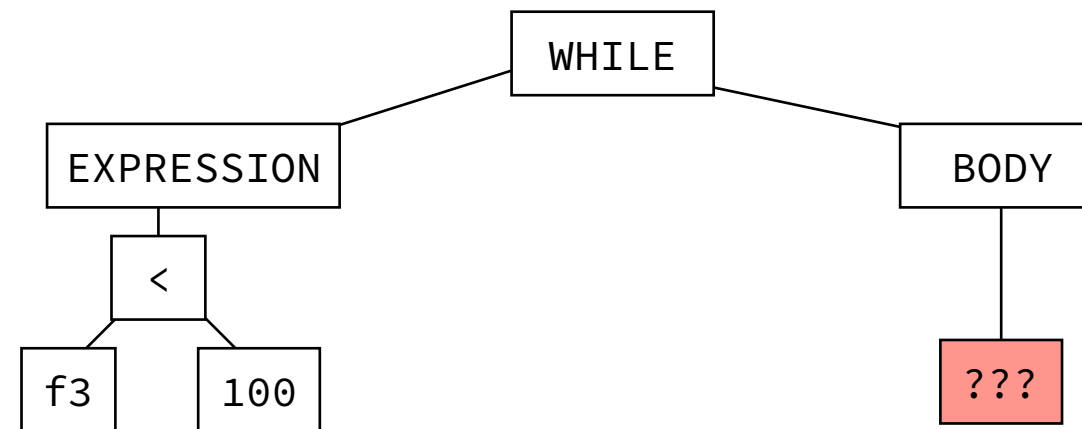


```
while f3 < 100 {  
    f3 = f1 + f2;  
    f1, f2 = f2, f3;  
}
```

Syntax Error

KEYWORD	while
IDENTIFIER	f3
SYMBOL	<
LITERAL	100
SYMBOL	{
IDENTIFIER	f3
SYMBOL	=
IDENTIFIER	f1
SYMBOL	+
IDENTIFIER	f2
SYMBOL	;
IDENTIFIER	f1
SYMBOL	,
IDENTIFIER	f2
SYMBOL	=
IDENTIFIER	f2
SYMBOL	,
IDENTIFIER	f3
SYMBOL	;
SYMBOL	}

Abstract Syntax Tree



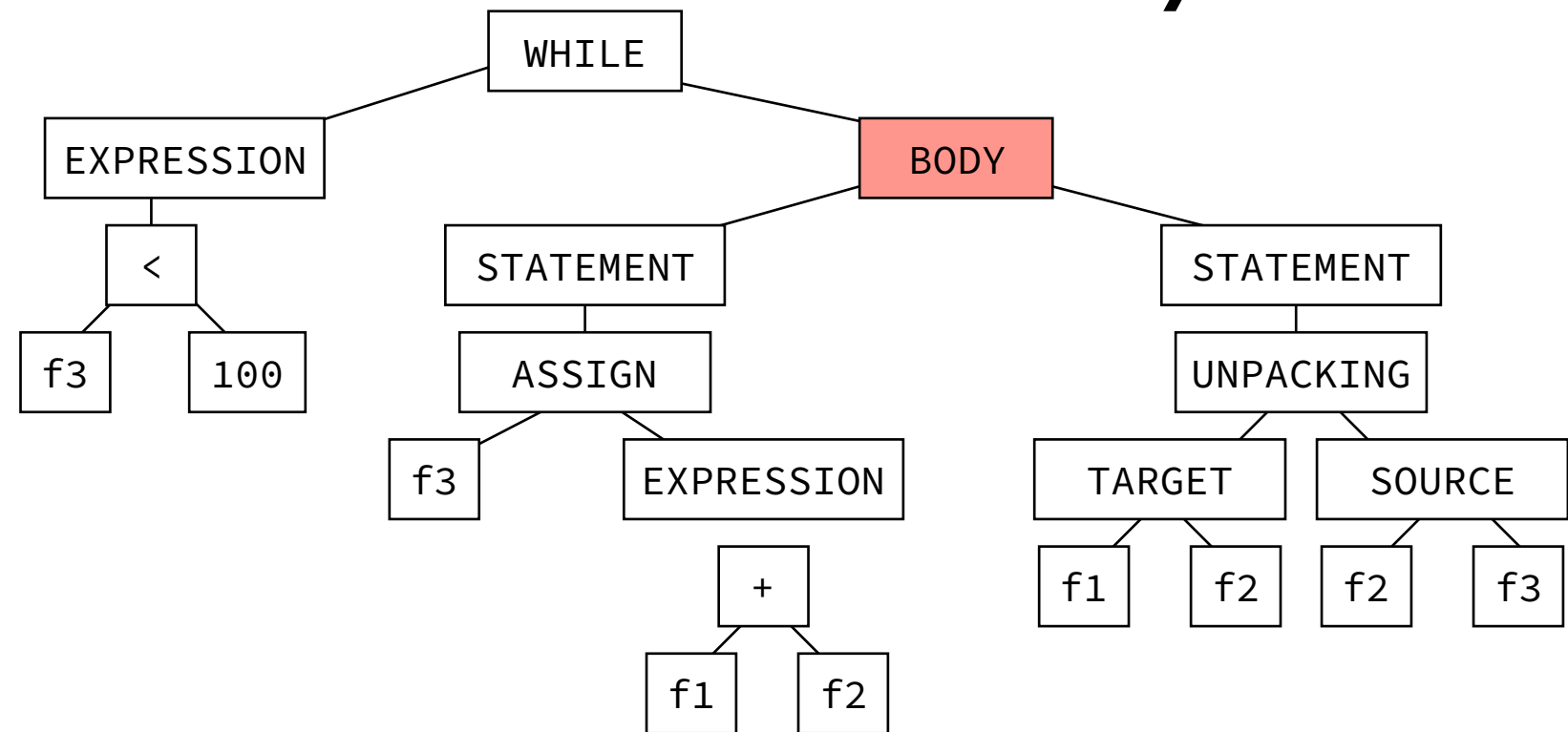
Syntax Error: Expect loop body

```
while f3 < 100 {
    f3 = f1 + f2;
    f1, f2 = f2, f3;
}
```

Syntax Error

KEYWORD	while
IDENTIFIER	f3
SYMBOL	<
LITERAL	100
SYMBOL	{
IDENTIFIER	f3
SYMBOL	=
IDENTIFIER	f1
SYMBOL	+
IDENTIFIER	f2
SYMBOL	;
IDENTIFIER	f1
SYMBOL	,
IDENTIFIER	f2
SYMBOL	=
IDENTIFIER	f2
SYMBOL	,
IDENTIFIER	f3
SYMBOL	;
SYMBOL	}

Abstract Syntax Tree



Syntax Error: Missing }

Parsing: Grammars

```

stmt:  expr NEWLINE
      | ID '=' expr NEWLINE
      | NEWLINE
      ;

expr:  <assoc=right> expr op='^' expr
      | expr op=('*' | '/') expr
      | expr op=('+' | '-') expr
      | INT
      | ID
      | '(' expr ')'

MUL  : '*';
DIV  : '/';
ADD  : '+';
SUB  : '-';
ID   : Letter LetterOrDigit*
fragment Letter: [a-zA-Z_]
fragment Digit: [0-9]
fragment LetterOrDigit: Letter | Digit
NEWLINE: '\r'? '\n'
WS    : [ \t]+ -> skip

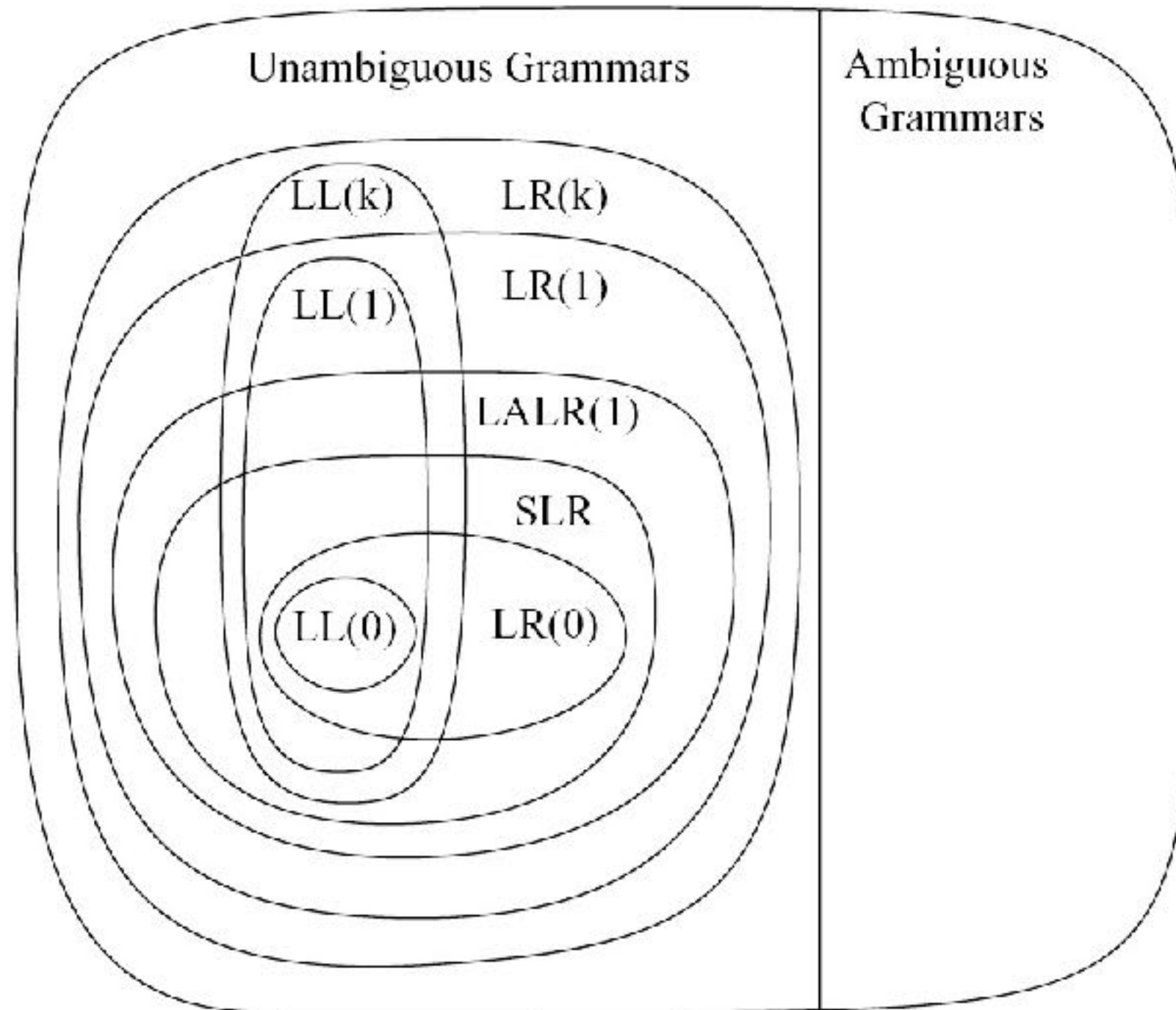
```

```

a = 2
b = 3
c = 4 + a * b
d = 5 - c * (3 + b) / a
e = a ^ (b + 10) ^ c ^ d

```

Parsing: Grammars



Parsing: Grammars

LL ✗
LR ✓

Expr \rightarrow Expr + Term
 | Expr - Term
 | Term

 Term \rightarrow Term * Factor
 | Term / Factor

 Factor \rightarrow (Expr)
 | Integer

Expr \rightarrow Term Expr'
 Expr' \rightarrow + Term Expr'
 | - Term Expr'
 | ϵ

 Term \rightarrow Factor Term'
 Term' \rightarrow * Factor Term'
 | / Factor Term'
 | ϵ

 Factor \rightarrow (Expr)
 | Integer

LL ✓
LR ✓

```
def Expr():
    Expr()
    match('+')
    Term()
```

Infinite Recursion!

Lexer & Parser?

- Usually, lexer and parser can be completely separated.
- However,
 - `vector<pair<int, int>>`

Pragmatic Solution

- What to do
 - Build AST
 - Check syntax errors
- Use parser generators, especially, *ANTLR 4*
- Check <https://github.com/antlr/grammars-v4>
- Read if you want
 - <https://abcdabcd987.com/using-antlr4/>
 - <https://abcdabcd987.com/notes-on-antlr4/>

Challenge Yourself

- Hand-written lexer and parser
- Check *Parsing Techniques: A Practical Guide*

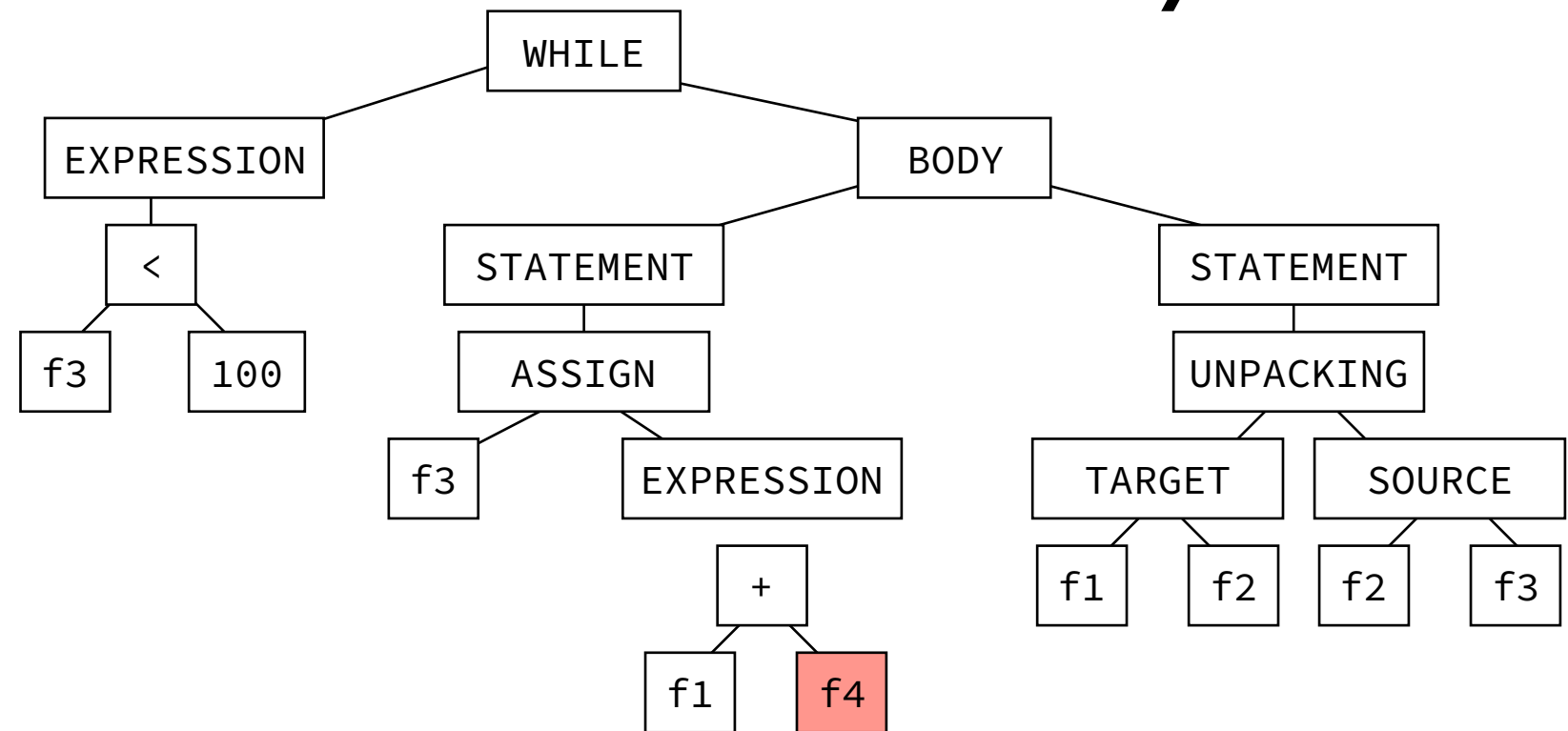
Semantic Analysis

```
while f3 < 100 {
    f3 = f1 + f4;
    f1, f2 = f2, f3;
}
```

Semantic Error

KEYWORD	while
IDENTIFIER	f3
SYMBOL	<
LITERAL	100
SYMBOL	{
IDENTIFIER	f3
SYMBOL	=
IDENTIFIER	f1
SYMBOL	+
IDENTIFIER	f2
SYMBOL	;
IDENTIFIER	f1
SYMBOL	,
IDENTIFIER	f2
SYMBOL	=
IDENTIFIER	f2
SYMBOL	,
IDENTIFIER	f3
SYMBOL	;
SYMBOL	}

Abstract Syntax Tree



Semantic Error: f4 used before declaration

Language Features

- `x, y = y, x`
- `c = sum(x * y for x in a for y in b)`
- `a.sort(key=lambda x: x[0])`

Pragmatic Solution

- What to do
 - Walk the AST tree
 - Build symbol table
 - Check all kinds of semantic errors

Challenge Yourself

- Add features to the language
 - unpacking
 - list comprehension
 - lambda
 - lifetimes
 - ...

IR Generation

IR: What & Why

- Intermediate Representation
- Focus less on the source language
- Pay more attention to the target platform
- Most of transformation and analysis are done in IR

IR Design

- IR design is closely related to
 - Source language
 - Target machine
 - Transforms / Analysis

IR: Multiple Levels

- A compiler can use more than one IR, and of course, there are more than one level.
- HIR/MIR: Carry more information. May have type system similar to the source language. Higher level analysis & transforms can be performed on. (Alias analysis works better with type knowledge)
 - `point1.x => (LoadField point1 "x")`
- LIR: Closer to the target machine. Don't have much type information (General/FP Reg). Focus on code generation.
 - `point1.x => (LoadMem (Mem baseAddr 4))`

IR: Multiple Levels

- A compiler can use more than one IR, and of course, there are more than one level.
- LLVM: Low Level Virtual Machine
 - Actually, its level is not that low.
 - And it happens that LLVM use a single representation.
- The more information you own, the more chances you have to do analysis and transforms.

LLVM IR

- LLVM: almost keep everything!

```
struct RT {  
    char A;  
    int B[10][20];  
    char C;  
};  
struct ST {  
    int X;  
    double Y;  
    struct RT Z;  
};
```

```
int *foo(struct ST *s) {  
    return &s[1].Z.B[5][13];  
}
```

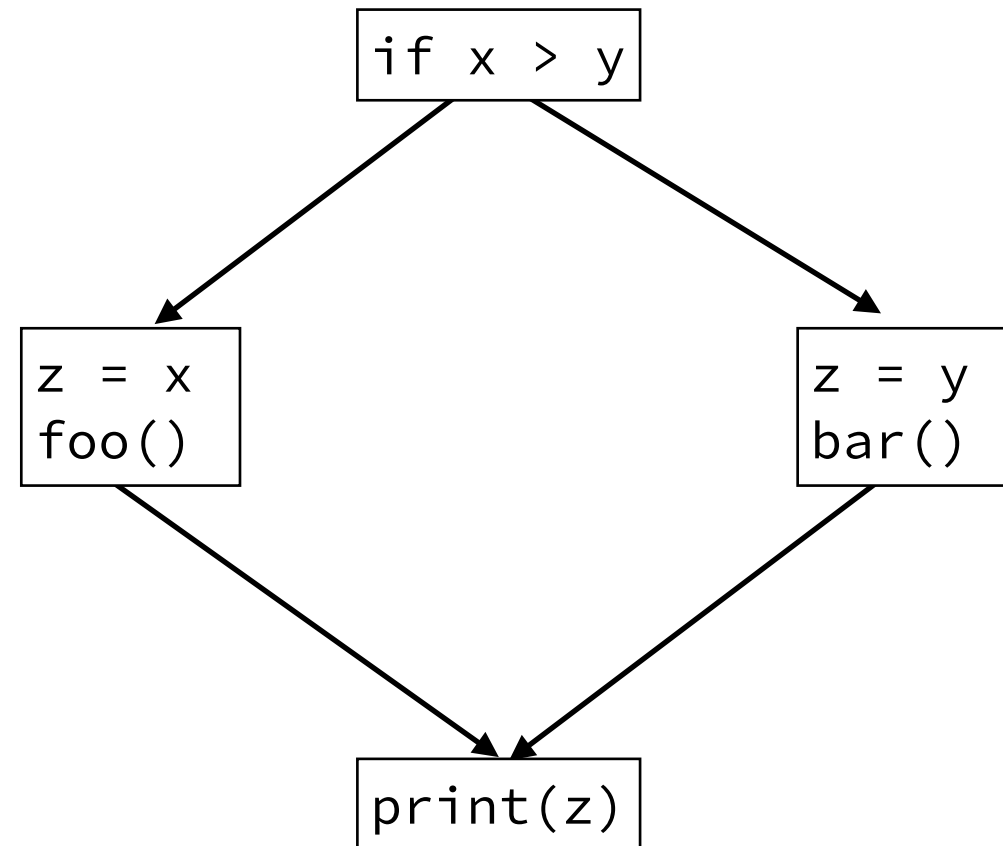
```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }  
%struct.ST = type { i32, double, %struct.RT }  
  
define i32* @foo(%struct.ST* %s) {  
entry:  
    %arrayidx = getelementptr inbounds %struct.ST,  
    %struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13  
    ret i32* %arrayidx  
}
```

IR Design: Structure

- Tree (the Tiger Book)
 - ✗ I cannot understand it
 - ✗ Hard to analyze and transform
- Linear (the Dragon Book)
 - ✗ Hard to analyze and transform
- Control Flow Graph
 - ✓ Easy to build CFG IR
 - ✓ Further analysis and transformations need CFG




Control Flow Graph

```
if x > y:  
    z = x  
    foo()  
else:  
    z = y  
    bar()  
print(z)
```




- Node: Basic Block
- BB: Straight-line piece of code without any jumps or jump targets
- Directed Edge: Jumps

Design: Memory Model?

- Memory-to-Memory
 - Reg Alloc: What should be kept in registers?
 -  Lots of students wasted lots of time on it
- Register-to-Register:
 - Unlimited virtual register
 - Reg Alloc: What should be spilled to memory?
 -  Easy to understand
 -  Similar to the target platform

Design: Function?

- Should the “function” and “function call” concept present in IR?
- I’m strongly in favor of it
 -  Simplify things
 - Function call doesn’t split basic block
 - In optimization’s language, “global” means inside a function, not the whole program.

Debugging

- I printed my IR in LLVM's format and run
 - Painful!
 - No direct memory arithmetic!
- I wrote my own interpreter
 - <https://github.com/abcdabcd987/LLIRInterpreter>
 - Life is much more easier!

Pragmatic Solution

- What to do
 - Walk the AST tree
 - Generate IR
- IR Design
 - Use CFG IR. Don't use tree IR or linear IR.
 - Use register-to-register memory model
- Check senior students' design for reference, for example
 - <https://github.com/abcdabcd987/LLIRInterpreter>

Challenge Yourself

- Design your own IR
- Read for your reference if you want:
 - <https://speakerdeck.com/abcdabcd987/compiler2016-by-abcdabcd987>

Optimizations

Optimizations

- Loop optimizations
 - Loop unrolling
 - Software pipelining
- Data-flow optimizations
 - Common subexpression elimination
 - Constant folding and propagation
- SSA-based optimizations
 - Global value numbering
 - Sparse conditional constant propagation
- Code generator optimization
 - Register allocation
 - Instruction selection
 - Instruction scheduling
- Others
 - Dead code elimination
 - Inlining
- ...

Register Allocation

- Register-to-register IR: infinite virtual registers
- Real machine: limited number of registers
- Register allocation: map virtual registers to real registers
- Spilling: which virtual registers should move to memory

Register Allocation

- Linear scan algorithm
 - ✓ Sounds easier?
 - ✓ Allocate faster
 - ✗ Slightly worse run time
- Graph coloring algorithm
 - ✗ Liveness analysis
 - ✗ Write more lines of code
 - ✓ Better run time performance
 - ✓ Actually, not hard at all. Way simpler than lots of OI/ACM algorithms.

Pragmatic Solution

- What to do
 - Analyze and transform IR
- Graph coloring register allocation
- Inlining

Challenge Yourself

- Try all kinds of optimizations

Code Generation

Pragmatic Solution

- What to do
 - Transform IR to target machine assembly
- Do it in a naïve way

Challenge Yourself

- Dive further into x86-64
- Instruction selection
- Instruction scheduling

Standard Library

Pragmatic Solution

- Use libc

Challenge Yourself

- Write your own standard library
- Write your own heap memory allocator

Wrap Up

Pragmatic Solution

- Use ANTLR 4. Imitate existing ANTLR 4 grammars.
- Use CFG IR. Use register-to-register model.
- Use graph coloring register allocation.
- Use libc
- Talk to classmates, TAs, senior students
- Ask for help. Don't plagiarize others' code.

Challenge Yourself



- And help others