The Life and Adventures of LLVM

From Bytecode to the Executables

Rithik Sharma, PhD Student

Motivation?

Motivation?

- How does this talk align with the compiler class?

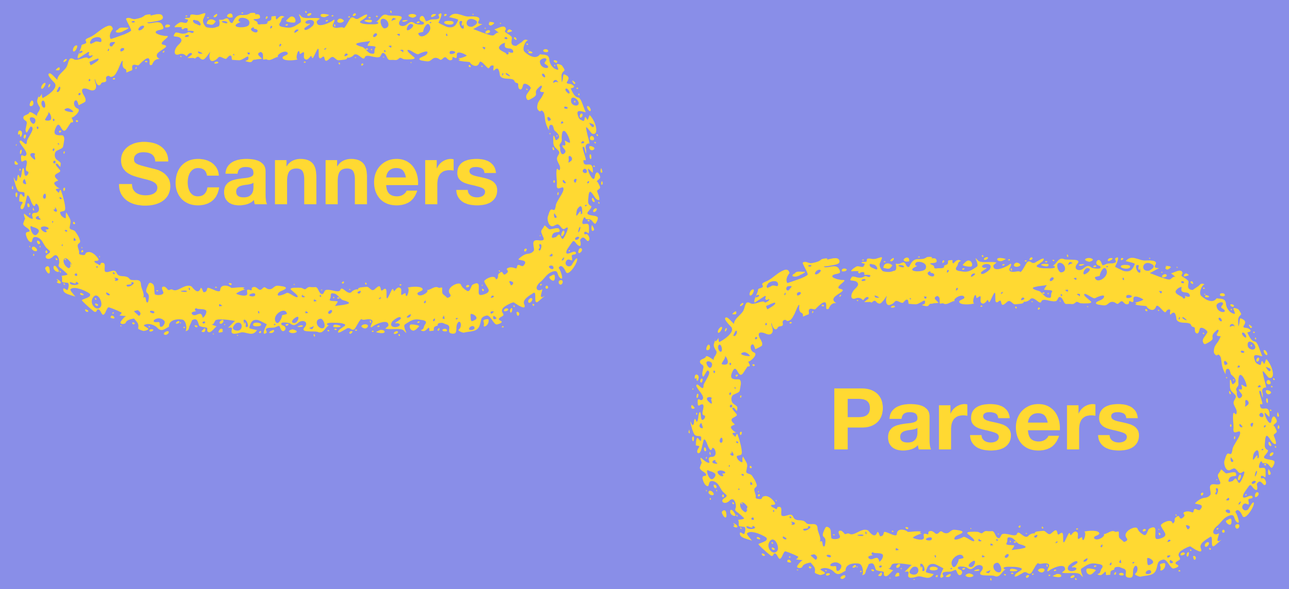
Motivation?

- How does this talk align with the compiler class?

Scanners

Motivation?

- How does this talk align with the compiler class?



Scanners

Parsers

Motivation?

- How does this talk align with the compiler class?

Scanners

Parsers

SSA/Three address code

Motivation?

- How does this talk align with the compiler class?

Scanners

Parsers

SSA/Three address code

Non- Ambiguous grammars

Motivation?

- How does this talk align with the compiler class?

Scanners

Parsers

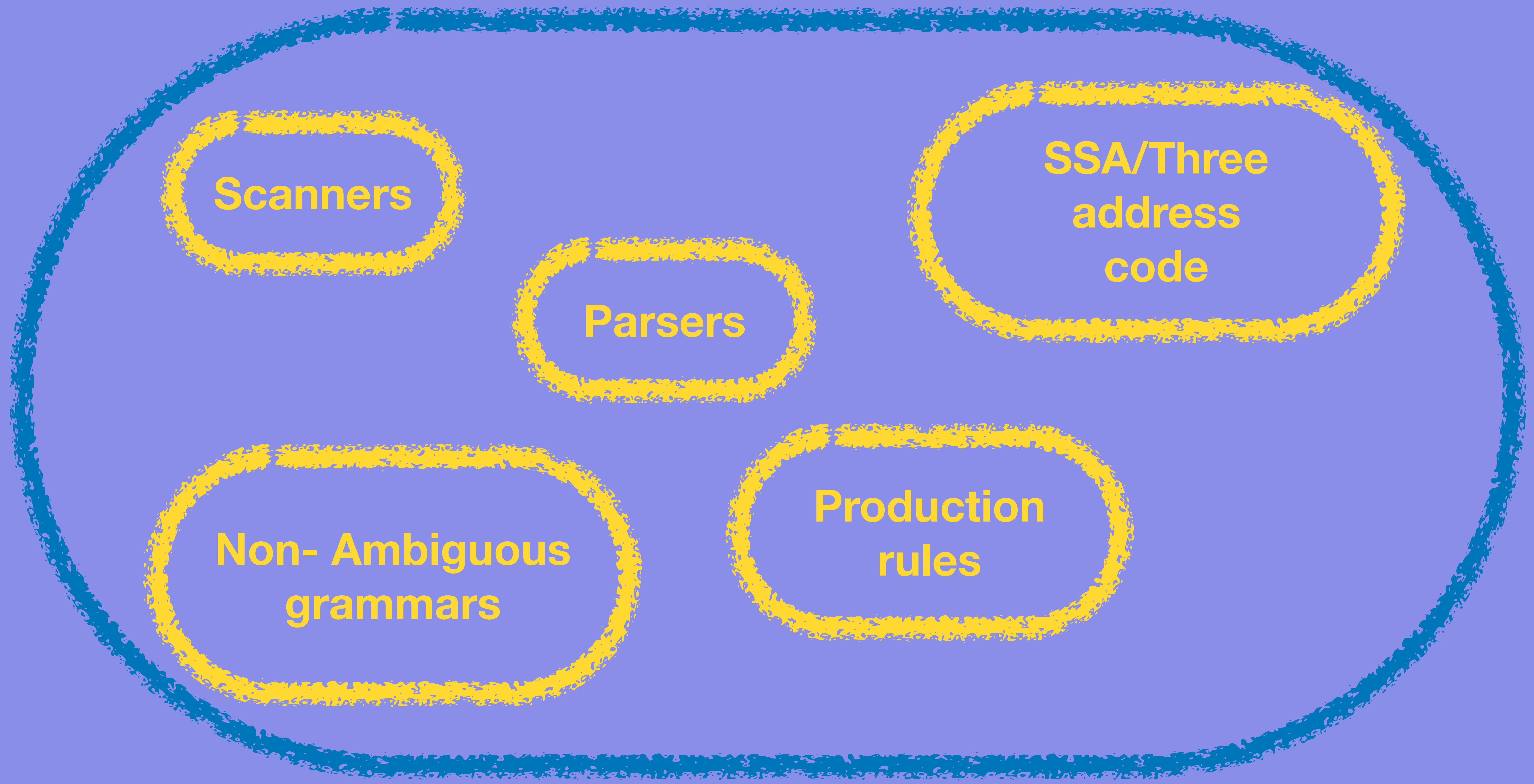
SSA/Three address code

Non- Ambiguous grammars

Production rules

Motivation?

- How does this talk align with the compiler class?



Scanners

Parsers

SSA/Three address code

Non- Ambiguous grammars

Production rules

Motivation?

- How does this talk align with the compiler class?

Techniques used by compilers

Motivation?

- How does this talk align with the compiler class?

Motivation?

- How does this talk align with the compiler class?
- What are some shortcomings of early compilers?

Motivation?

- What are some shortcomings of early compilers?

Motivation?

- What are some shortcomings of early compilers?
  - Performance

Motivation?

- What are some shortcomings of early compilers?
  - Performance
  - Re-usability

Motivation?

- What are some shortcomings of early compilers?
    - Performance
    - Re-usability
    - Optimizations

Motivation?

- What are some shortcomings of early compilers?
    - Performance
    - Re-usability
    - Optimizations
    - Correctness

Motivation?

- What are some shortcomings of early compilers?
    - Performance
    - Re-usability
    - Optimizations
    - Correctness
    - Scaling

Motivation?

- What are some shortcomings of early compilers?

Even Bob the Builder, is
confused about where to start

Picture credits: Bob the Builder

Motivation?

- What are some shortcomings of early compilers?

We need a modern compiler!

Picture credits: Bob the Builder

Motivation?

- Introduction to LLVM

LLVM

LLVM

LLVM

Low Level Virtual Machine

2023
EURO LLVM
DEVELOPERS' MEETING

2022
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2023
EURO LLVM
DEVELOPERS' MEETING

2022
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2022
EURO LLVM
DEVELOPERS' MEETING

2023
EURO LLVM
DEVELOPERS' MEETING

20
22
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2022
EURO LLVM
DEVELOPERS' MEETING

2021
LLVM DEVELOPERS' MEETING

2023
EURO LLVM
DEVELOPERS' MEETING

20
22
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2022
EURO LLVM
DEVELOPERS' MEETING

2021
LLVM DEVELOPERS' MEETING

2020
LLVM
VIRTUAL
DEVELOPERS' MEETING
OCTOBER 6-8

2023
EURO LLVM
DEVELOPERS' MEETING

20
22
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2022
EURO LLVM
DEVELOPERS' MEETING

2021
LLVM DEVELOPERS' MEETING

2020
LLVM
VIRTUAL
DEVELOPERS' MEETING
OCTOBER 6-8

2019
LLVM DEV
MEETING

2023
EURO LLVM
DEVELOPERS' MEETING

2022
LLVM
DEVELOPERS' MEETING
SAN JOSE, CALIFORNIA • HAYES MANSION

2022
EURO LLVM
DEVELOPERS' MEETING

2021
LLVM DEVELOPERS' MEETING

2020
LLVM VIRTUAL
DEVELOPERS' MEETING
OCTOBER 6-8

2019
LLVM DEV
MEETING

2018
LLVM
DEVELOPERS' MEETING

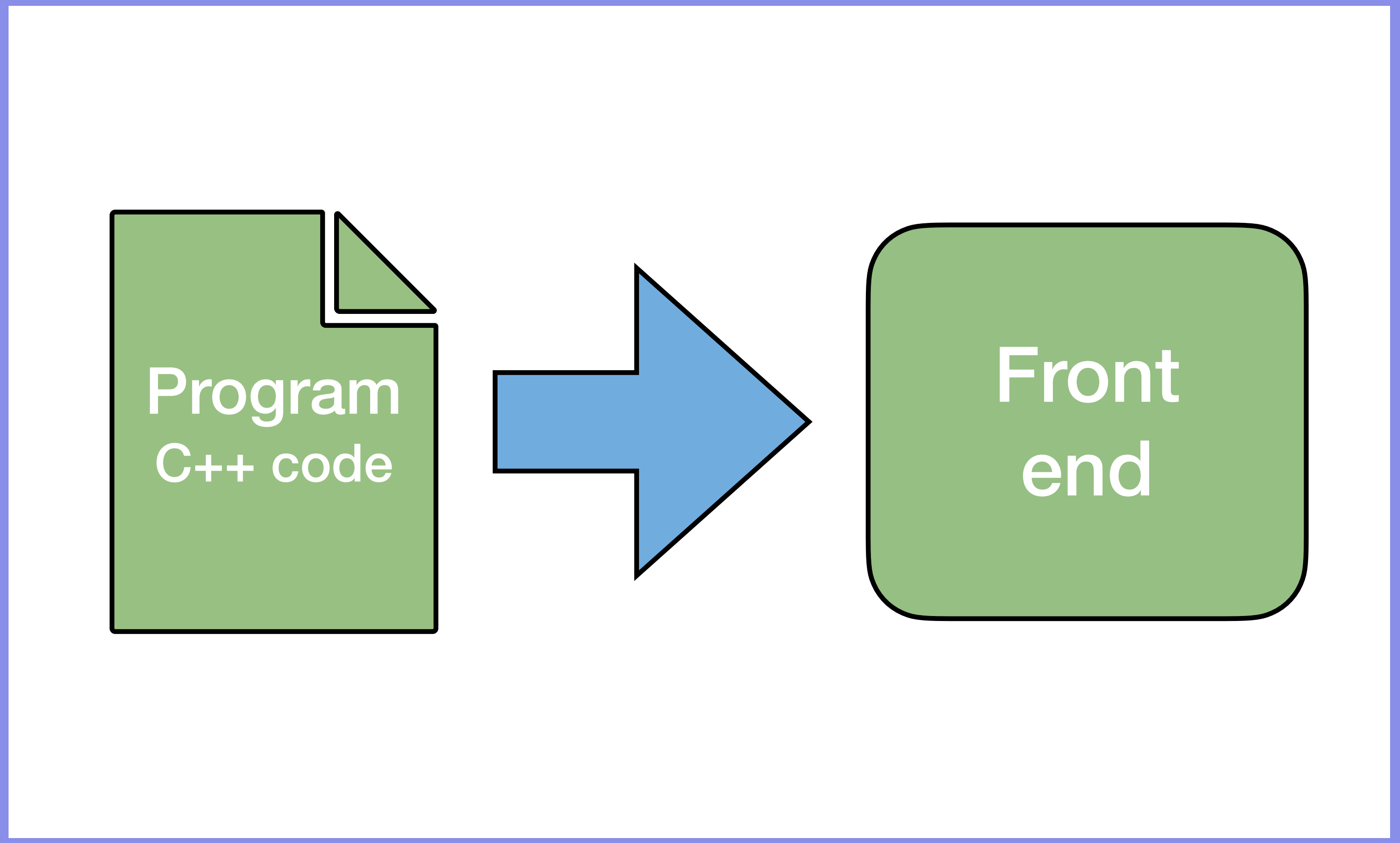
Motivation?

- Introduction to LLVM

Picture credits: Bob the Builder

Motivation?

- Introduction to LLVM



What happens inside the front end?

- Lexical Analysis

LLVM

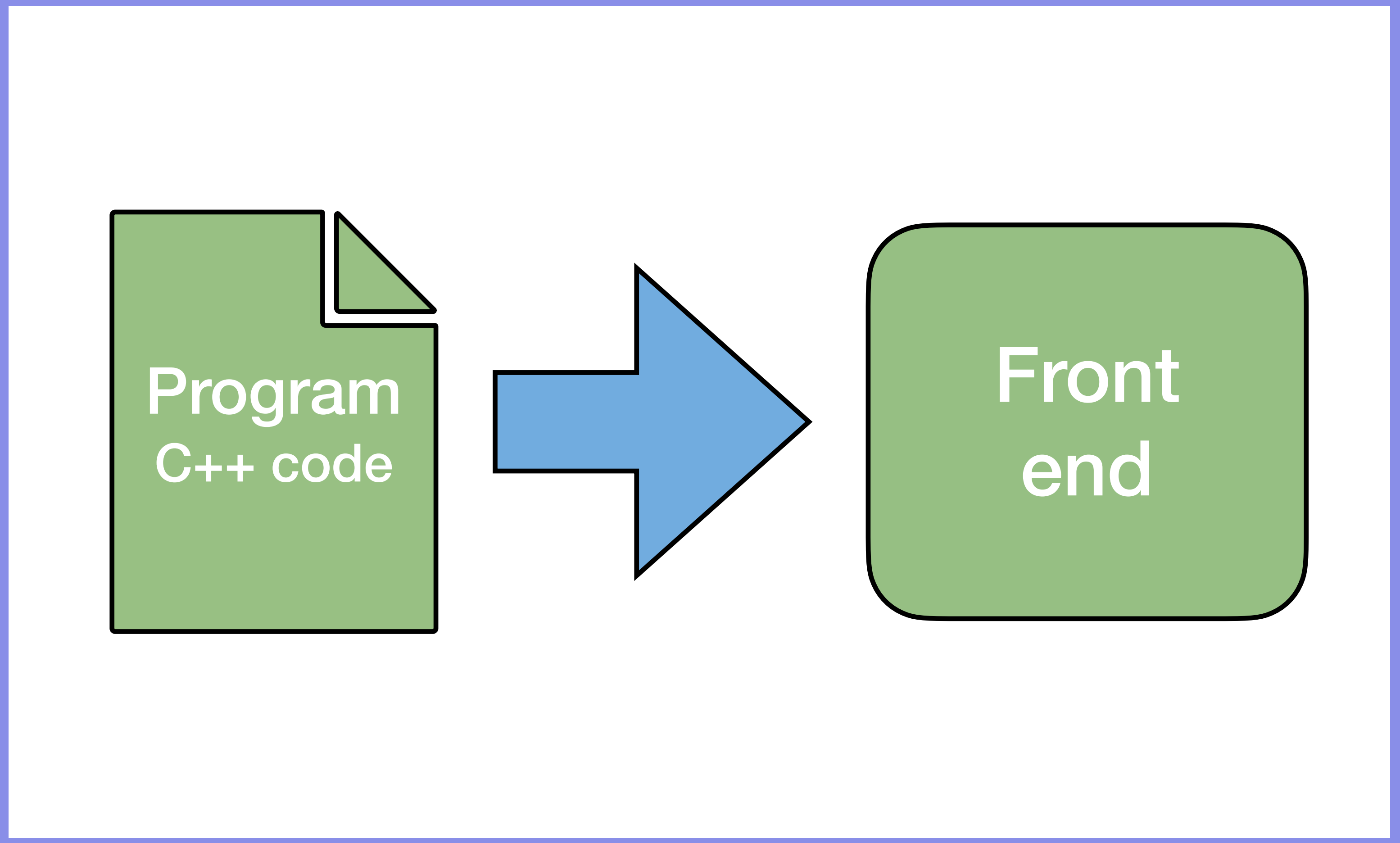
- **Lexical Analysis (tokenization or scanning)**
  - It breaks the source code into individual tokens, such as identifiers, keywords, literals, and operators.
  - Example of lexical analysis for a simple arithmetic expression: "5 + 3 * (7 - 2)"

Token: Integer Value: 5
Token: Operator Value: +
Token: Integer Value: 3
Token: Operator Value: *
Token: Left Parenthesis
Value: (

Token: Integer Value: 7
Token: Operator Value: -
Token: Integer Value: 2
Token: Right Parenthesis Value:)

Motivation?

- Introduction to LLVM



What happens inside the front end?

- Lexical Analysis
- Syntax Analysis

LLVM

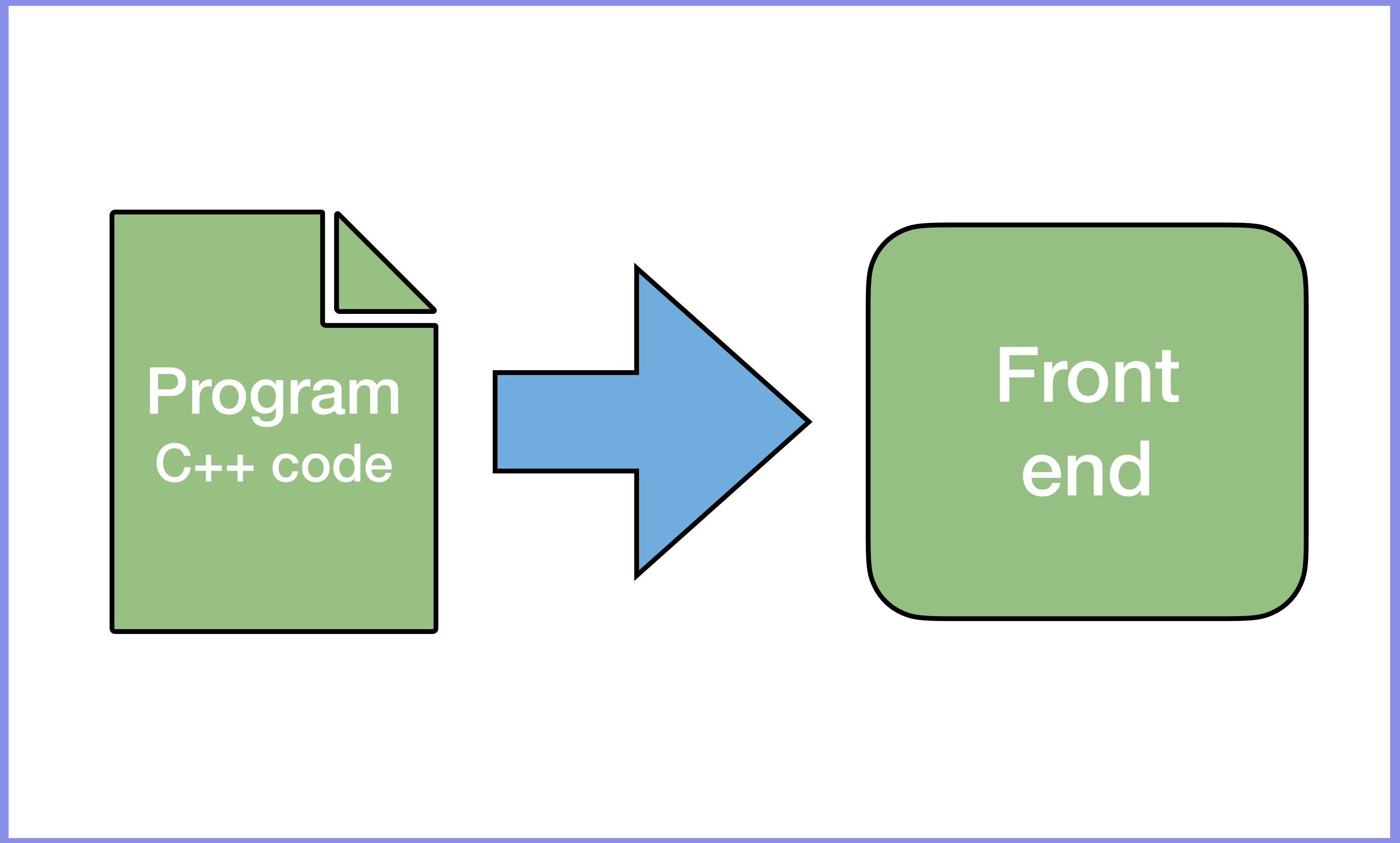
- Syntax Analysis
  - It builds the abstract syntax tree (AST) from the tokens.
  - AST represents the hierarchical structure of the source code.
  - Capturing the relationships between different elements and their corresponding expressions, statements, and declarations.

    expr -> term
    expr -> expr + term
    term -> factor
    term -> term * factor
    factor -> Integer
    factor -> (expr)

```
   expr
    / \
  term  +
   / \  |
factor term
     / \
  factor *
      |
    factor
```

Motivation?

- Introduction to LLVM



What happens inside the front end?

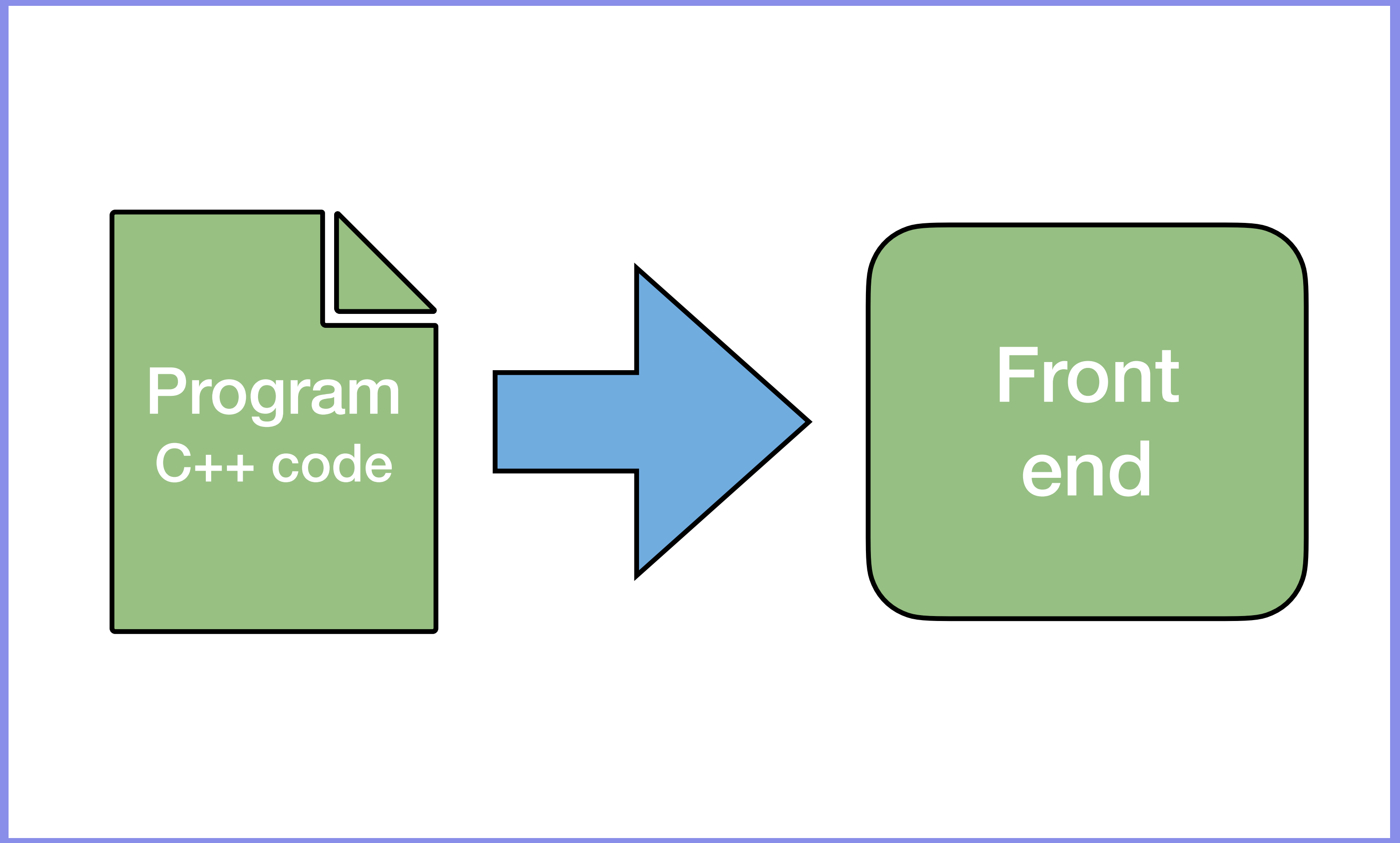
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

LLVM

- Semantic Analysis
    - Semantic analysis ensures the program is well-formed and meaningful according to the language's rules and specifications.
    - It helps catch errors and inconsistencies that may not be detected during lexical and syntax analysis alone.

Motivation?

- Introduction to LLVM

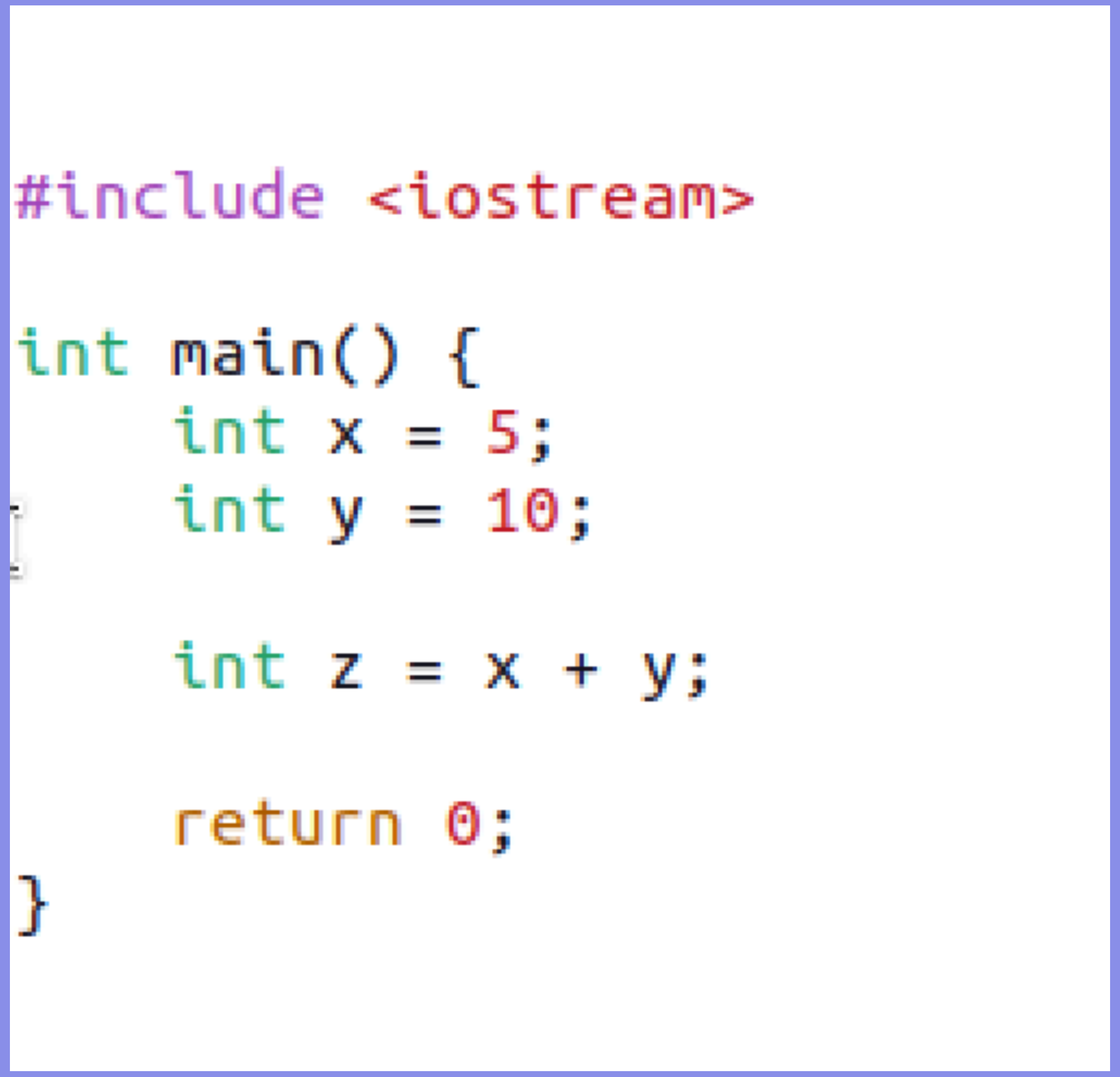


What happens inside the front end?

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- LLVM IR generation

LLVM

- **LLVM IR generation.**



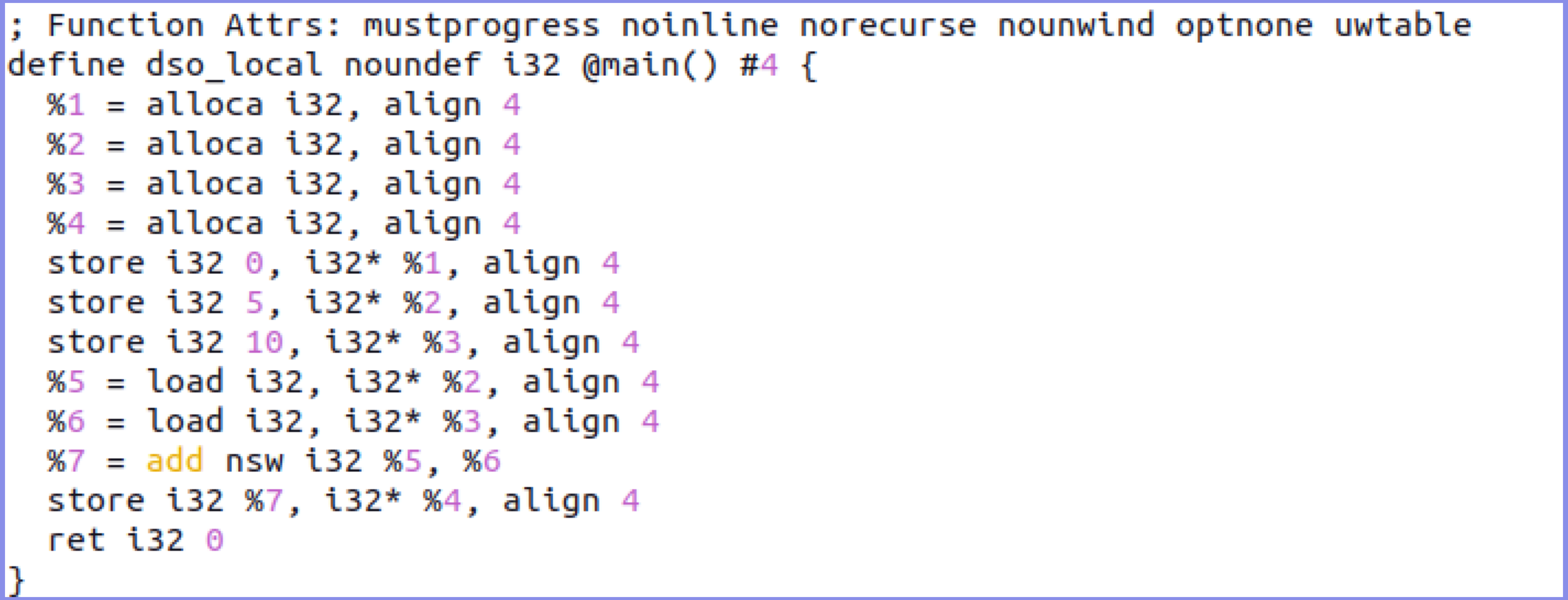
```
#include <iostream>

int main() {
    int x = 5;
    int y = 10;

    int z = x + y;

    return 0;
}
```

y

C++ Code

LLVM

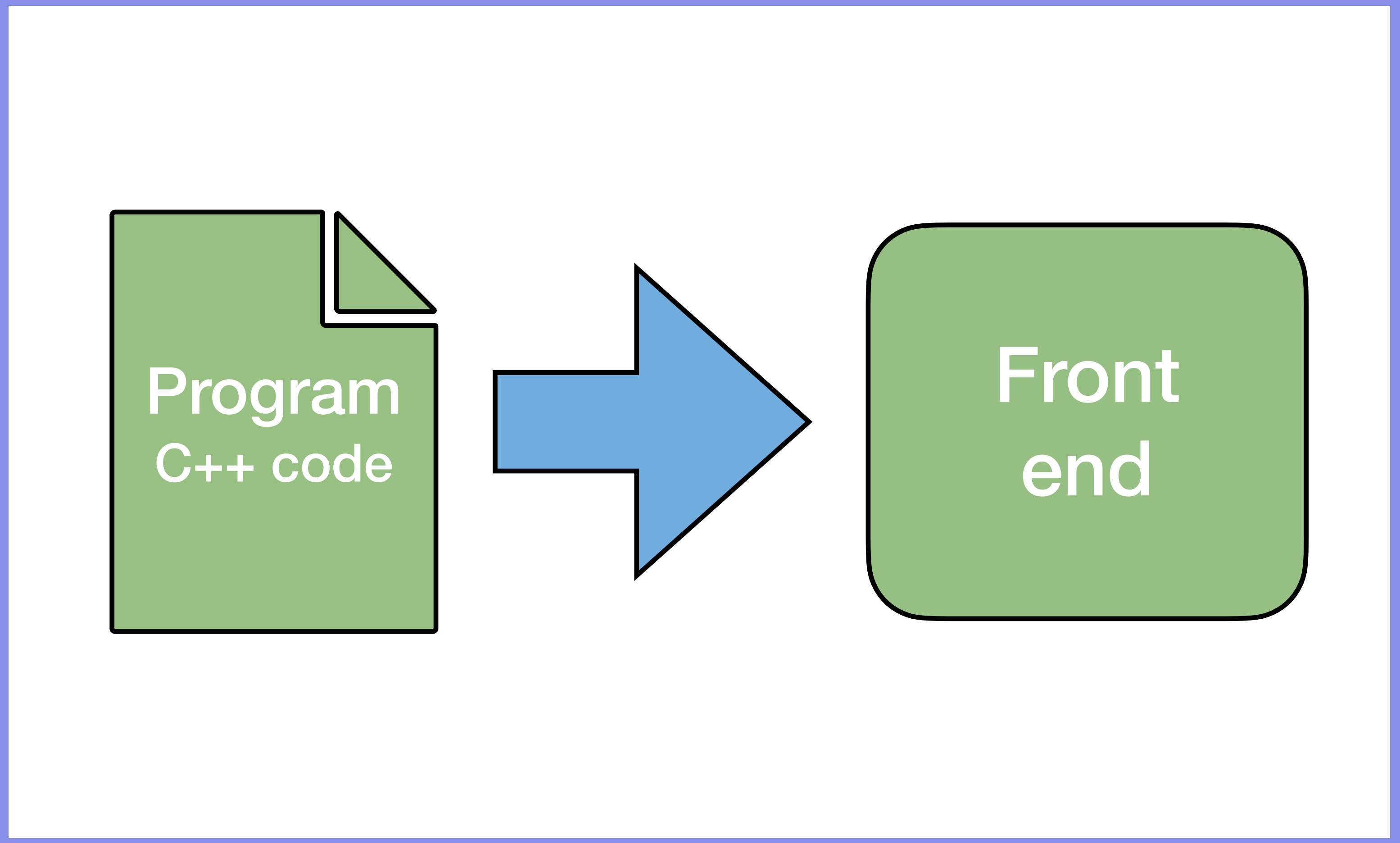
- **LLVM IR generation.**



```
; Function Attrs: mustprogress noinline norecurse nounwind optnone uwtable
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  store i32 10, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  ret i32 0
}
```

Motivation?

- Introduction to LLVM

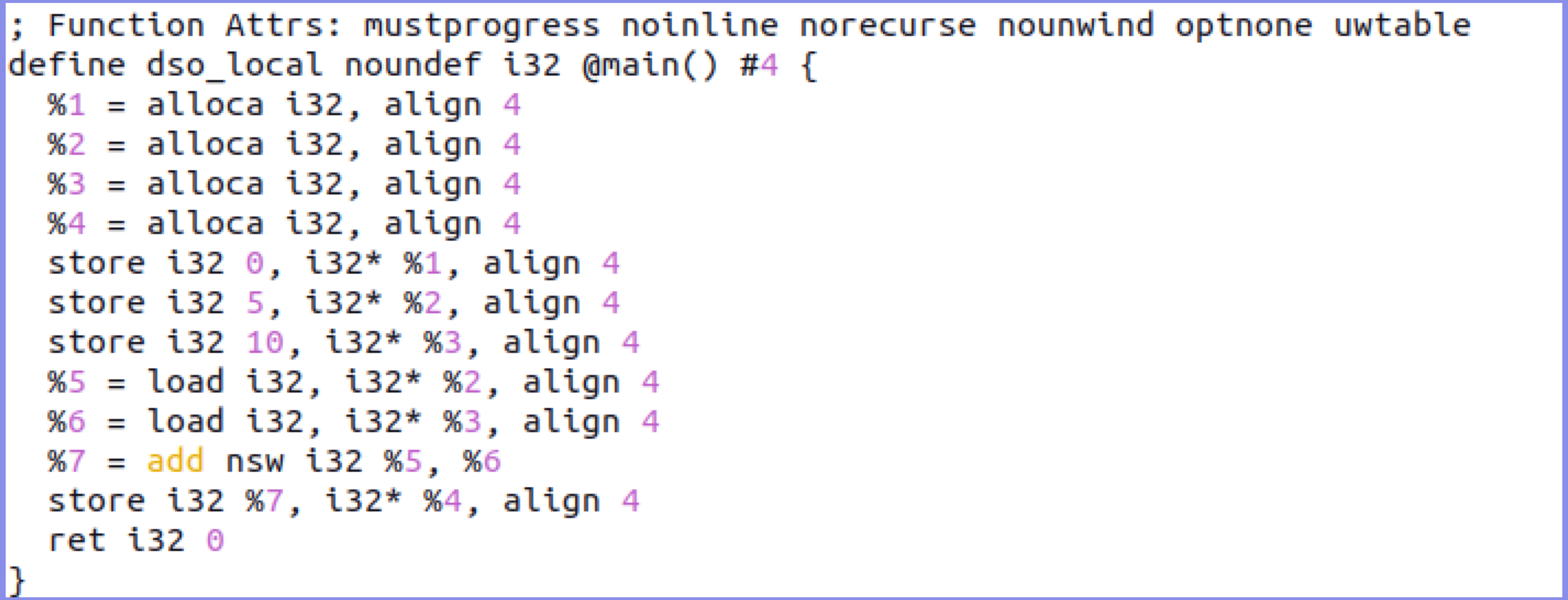


What happens inside the front end?

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- LLVM IR generation
- Optional Optimizations

LLVM

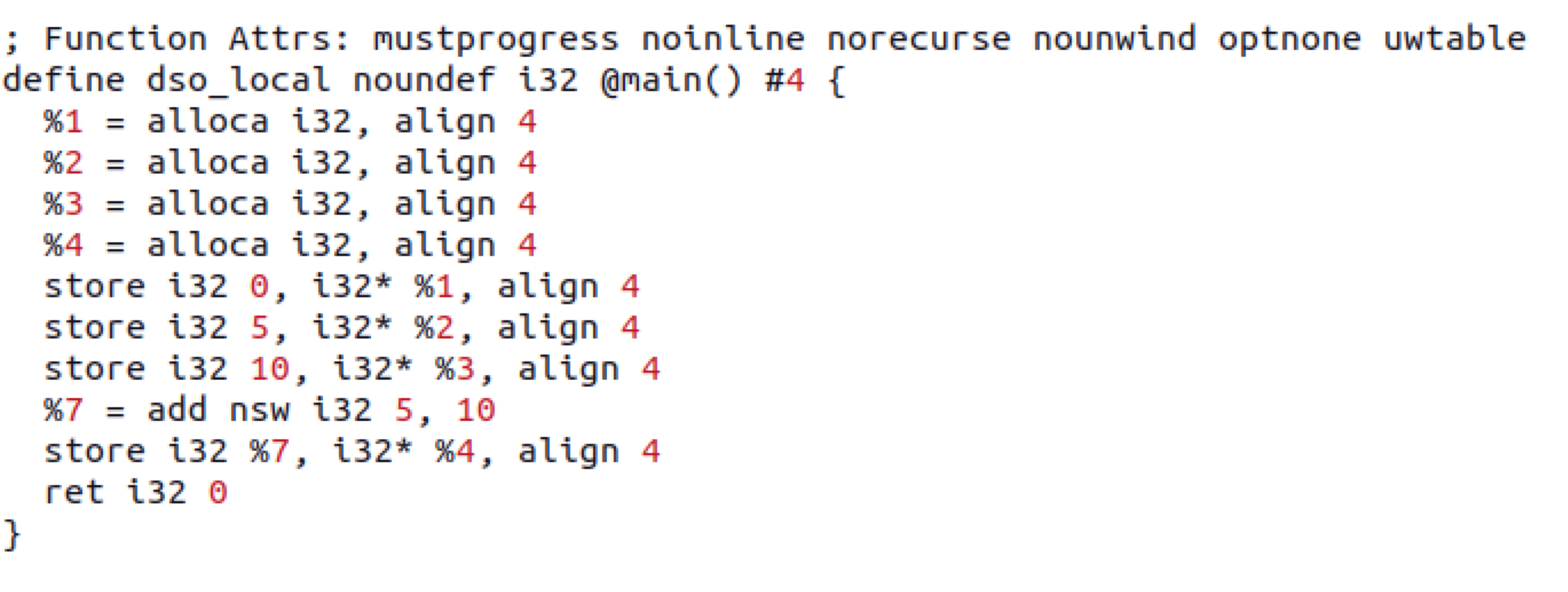
- **Optional Optimizations**
    - **Constant folding - simplifies expressions involving constants and replaces them with their computed values.**



```
; Function Attrs: mustprogress noinline norecurse nounwind optnone uwtable
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  store i32 10, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  ret i32 0
}
```

LLVM

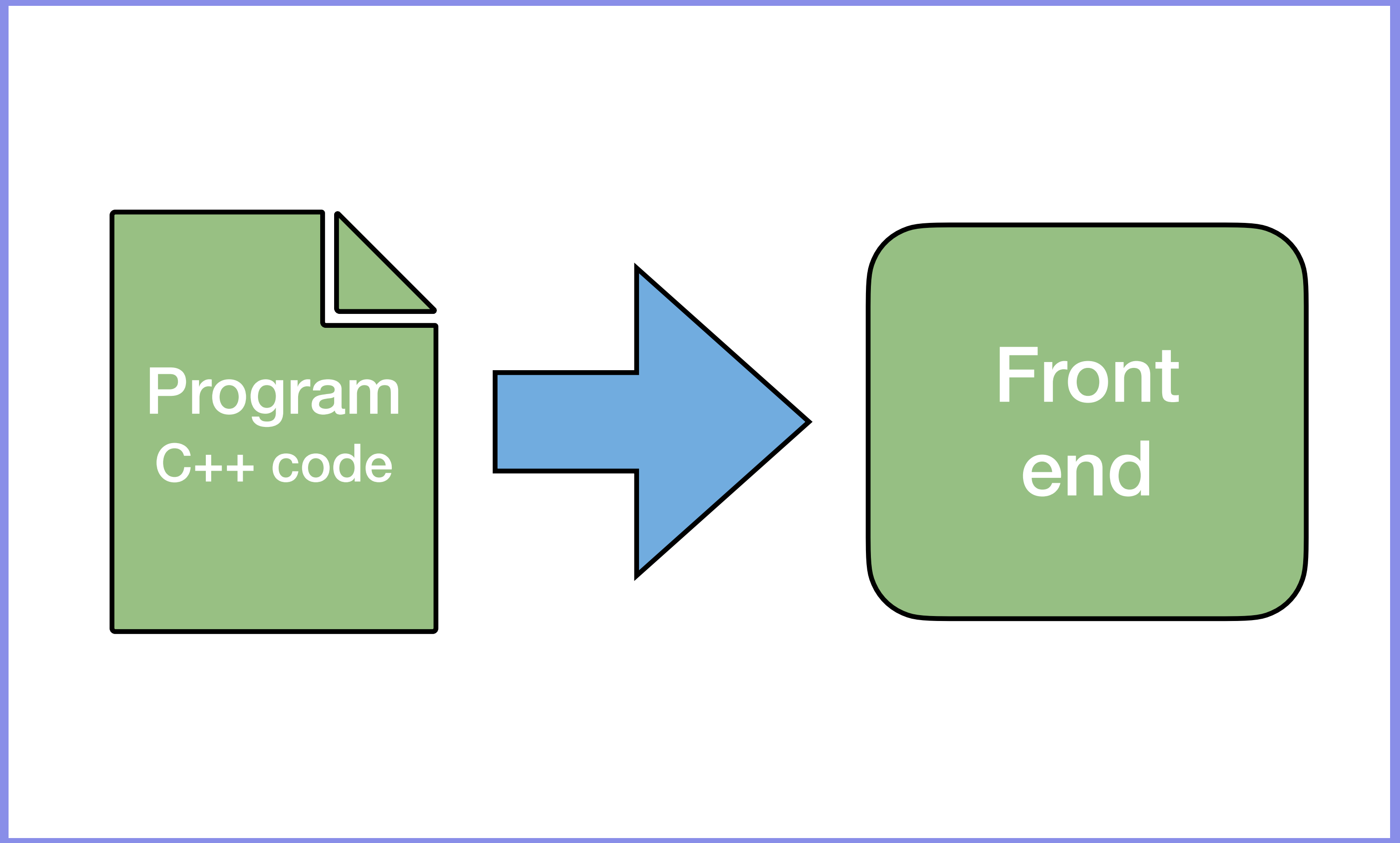
- Optional Optimizations
  - Constant folding - simplifies expressions involving constants and replaces them with their computed values.



```
; Function Attrs: mustprogress noinline norecurse nounwind optnone uwtable
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  store i32 10, i32* %3, align 4
  %7 = add nsw i32 5, 10
  store i32 %7, i32* %4, align 4
  ret i32 0
}
```

Motivation?

- Introduction to LLVM

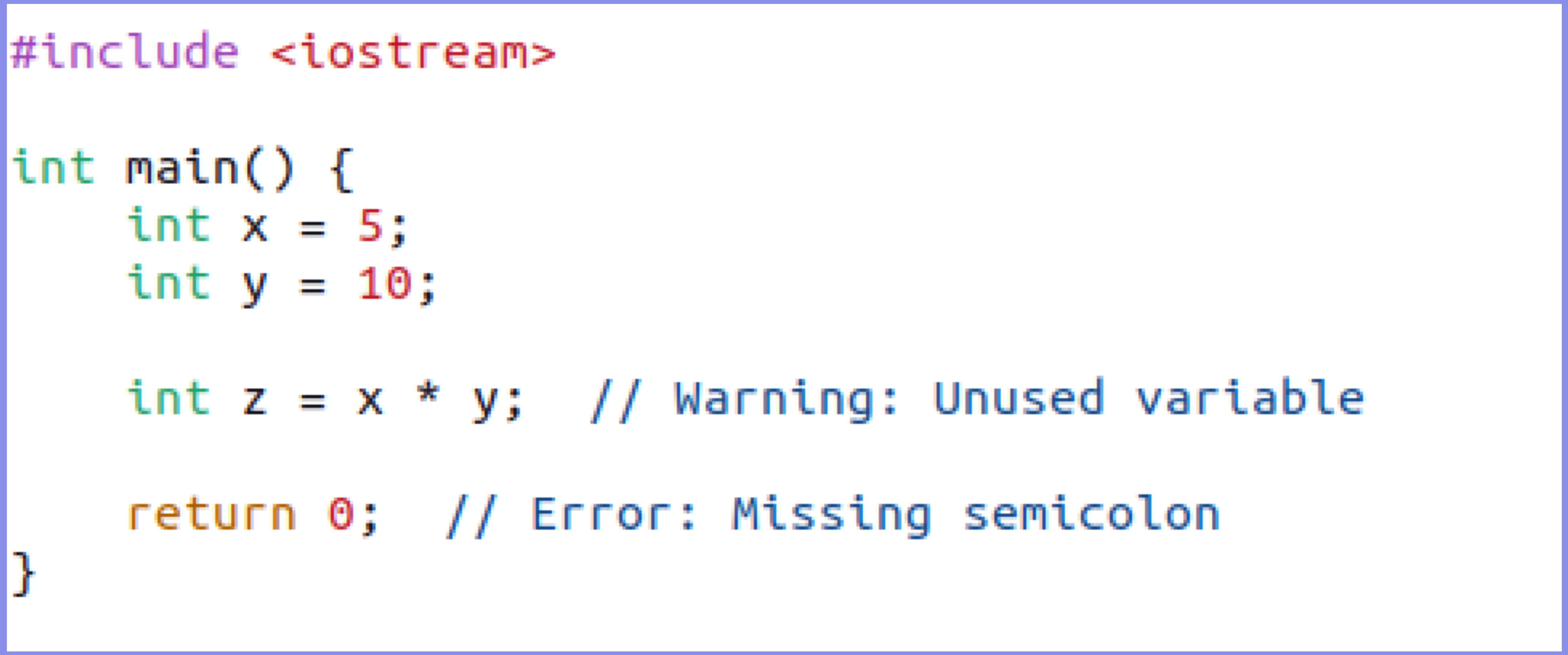


What happens inside the front end?

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- LLVM IR generation
- Optimizations
- Warnings and errors

LLVM

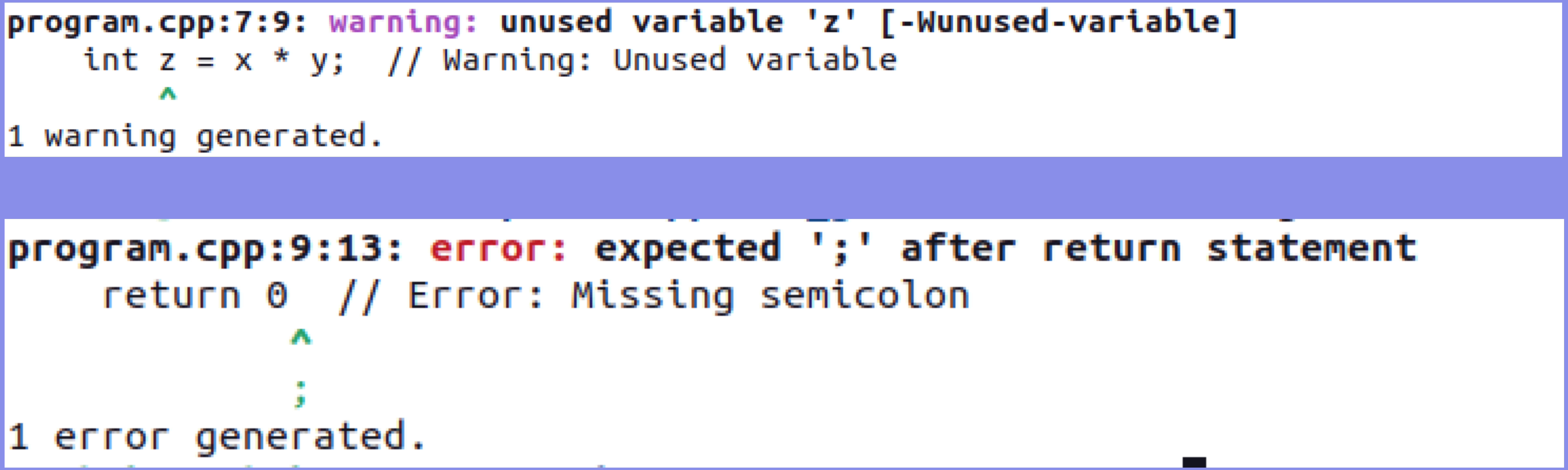
- **Warnings and errors.**



```
#include <iostream>

int main() {
    int x = 5;
    int y = 10;

    int z = x * y;  // Warning: Unused variable

    return 0;  // Error: Missing semicolon
}
```

LLVM

- Warnings and errors.



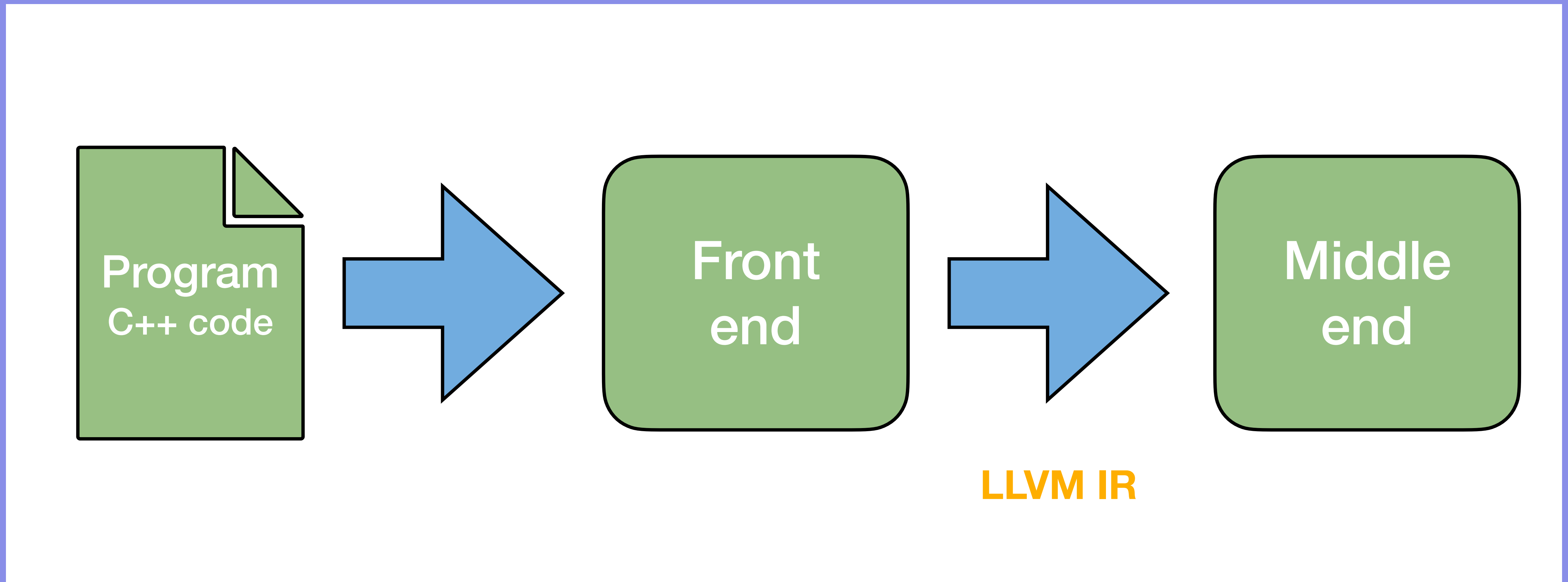
```
program.cpp:7:9: warning: unused variable 'z' [-Wunused-variable]
    int z = x * y;  // Warning: Unused variable
        ^
1 warning generated.
```

```
program.cpp:9:13: error: expected ';' after return statement
    return 0  // Error: Missing semicolon
            ^
            ;
1 error generated.
```

Clang error and warnings

Motivation?

- Introduction to LLVM

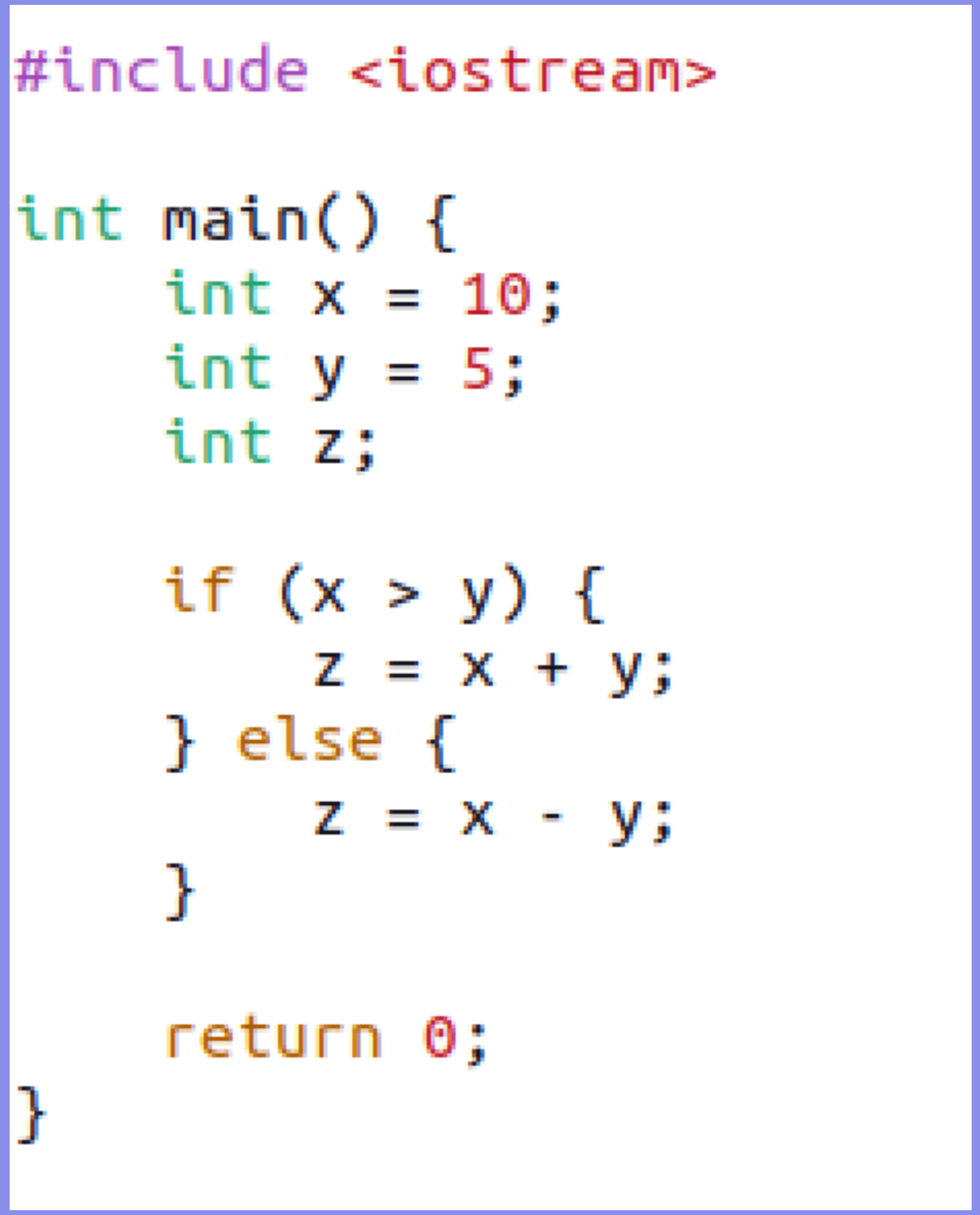


What happens inside the middle end?

- Data Flow Analysis (DFA)

LLVM

- **Data Flow Analysis (DFA)**

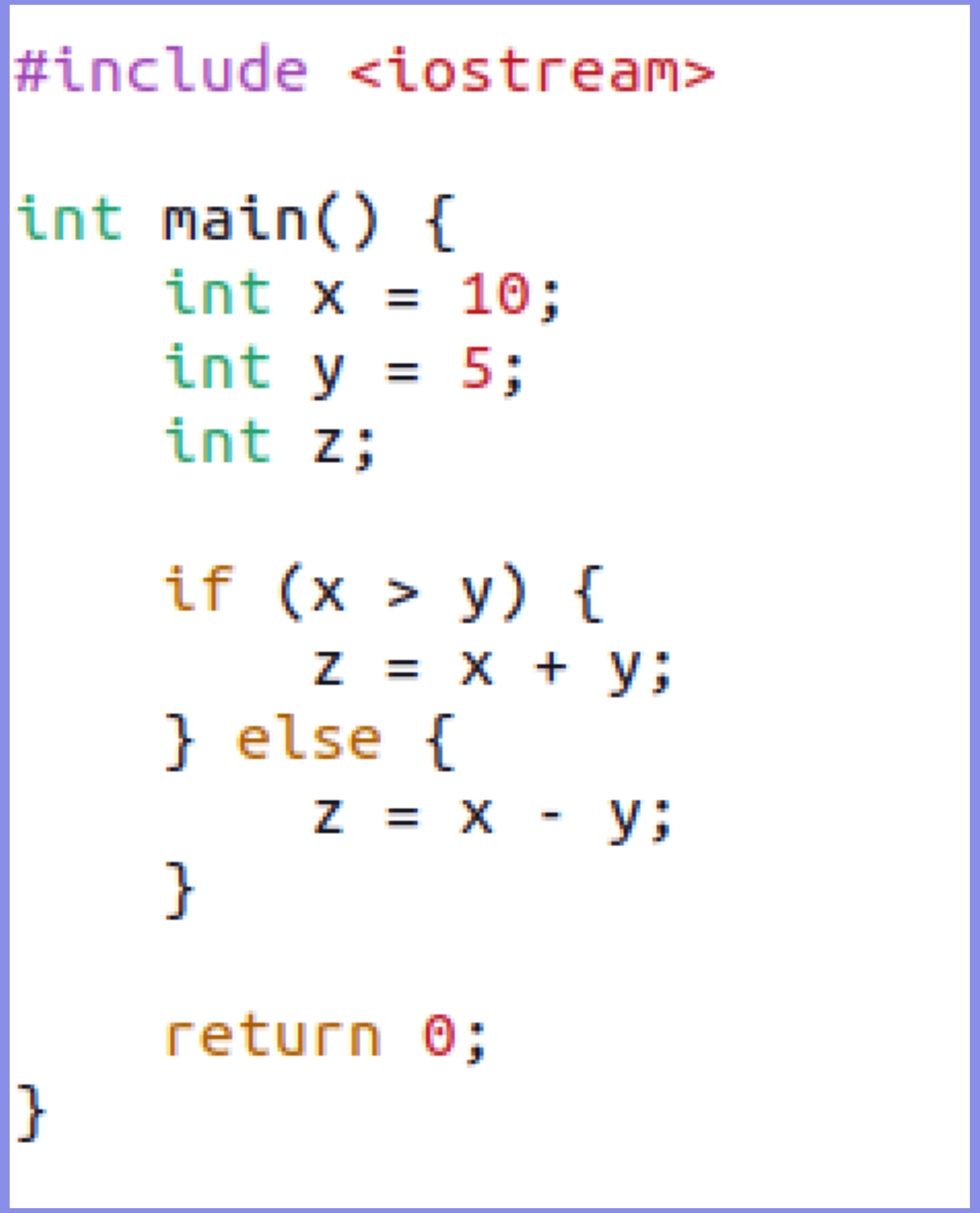


```
#include <iostream>

int main() {
    int x = 10;
    int y = 5;
    int z;

    if (x > y) {
        z = x + y;
    } else {
        z = x - y;
    }

    return 0;
}
```

C++ code

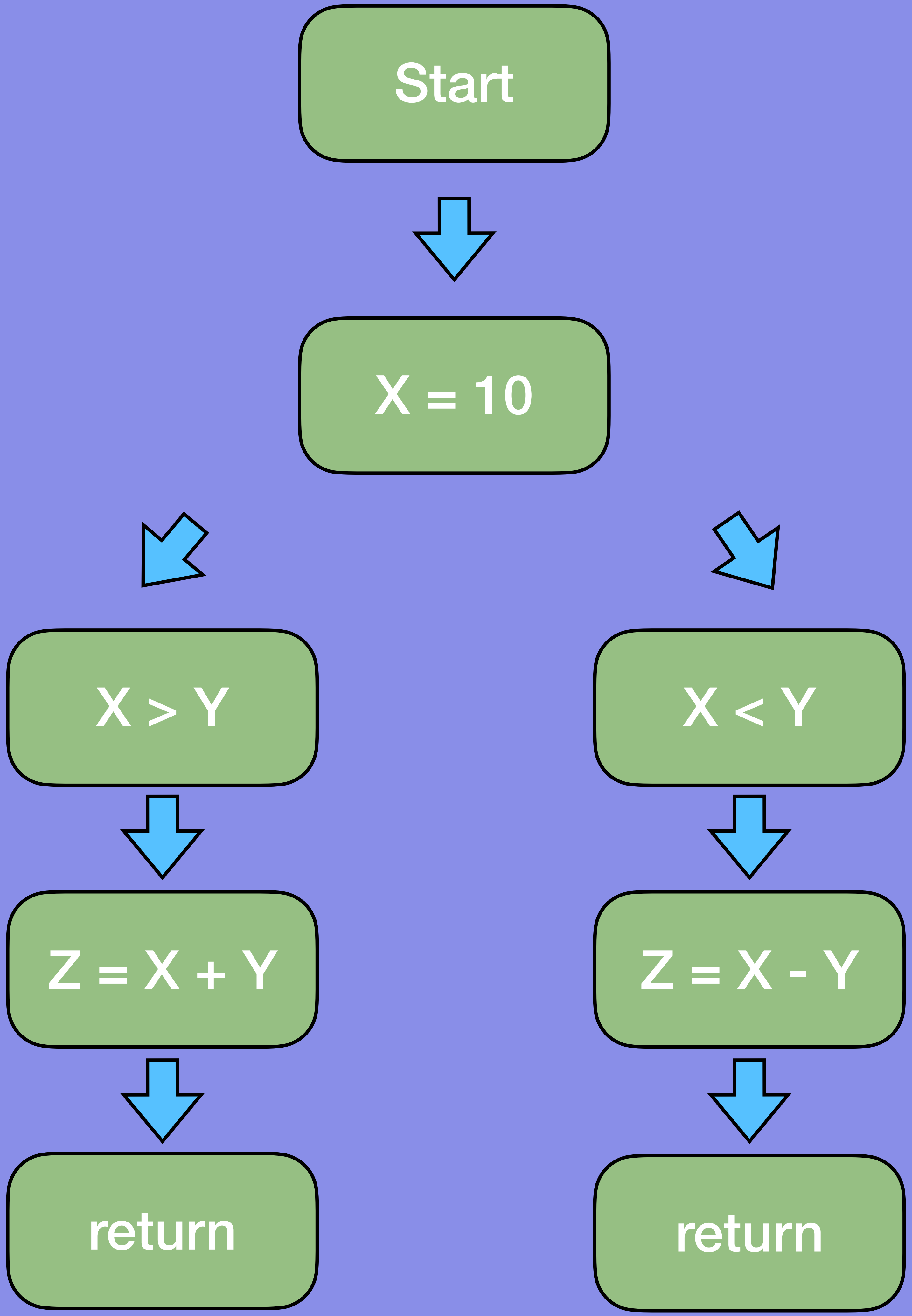
LLVM

- Data Flow Analysis (DFA)



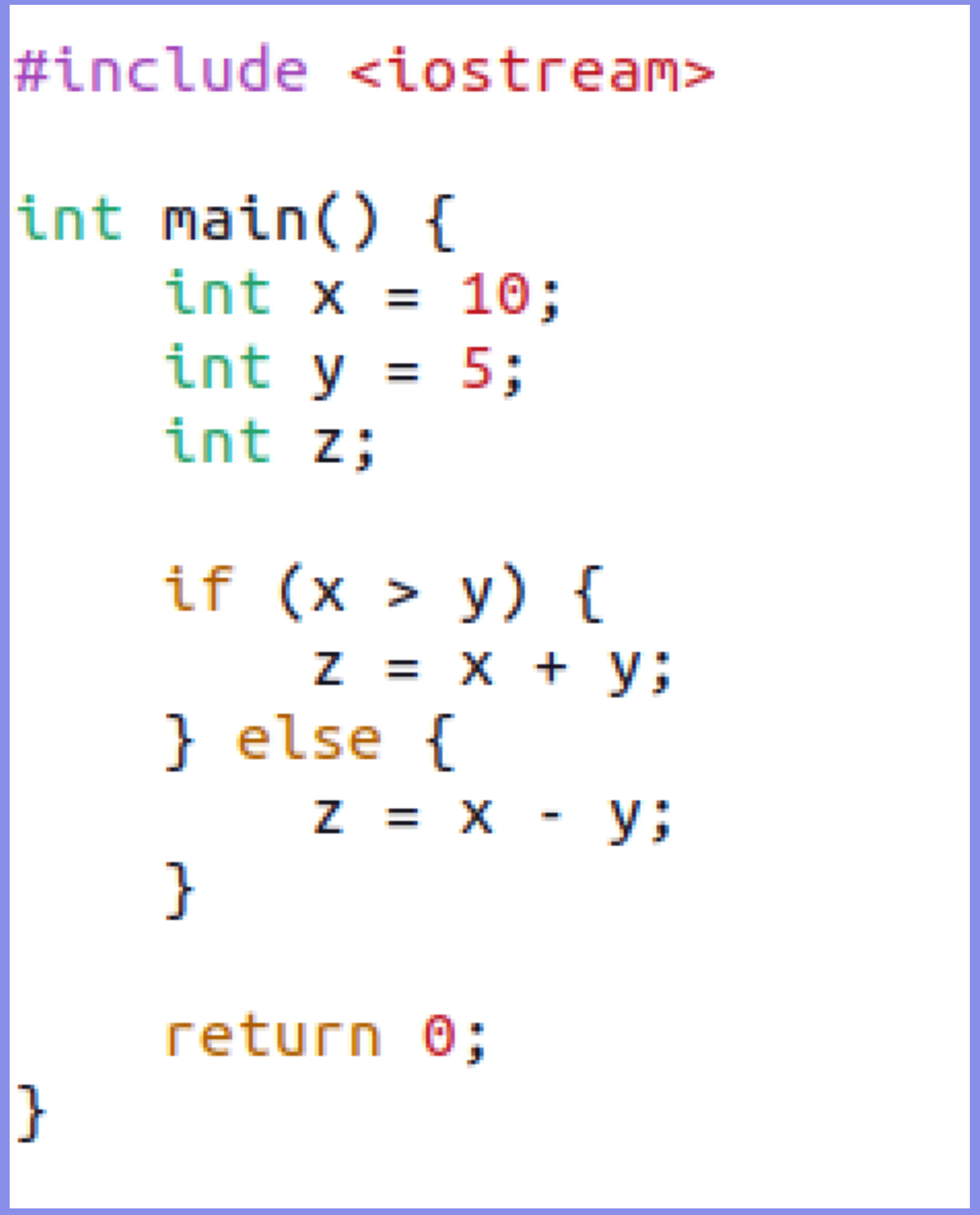
```
#include <iostream>

int main() {
    int x = 10;
    int y = 5;
    int z;

    if (x > y) {
        z = x + y;
    } else {
        z = x - y;
    }

    return 0;
}
```

C++ code



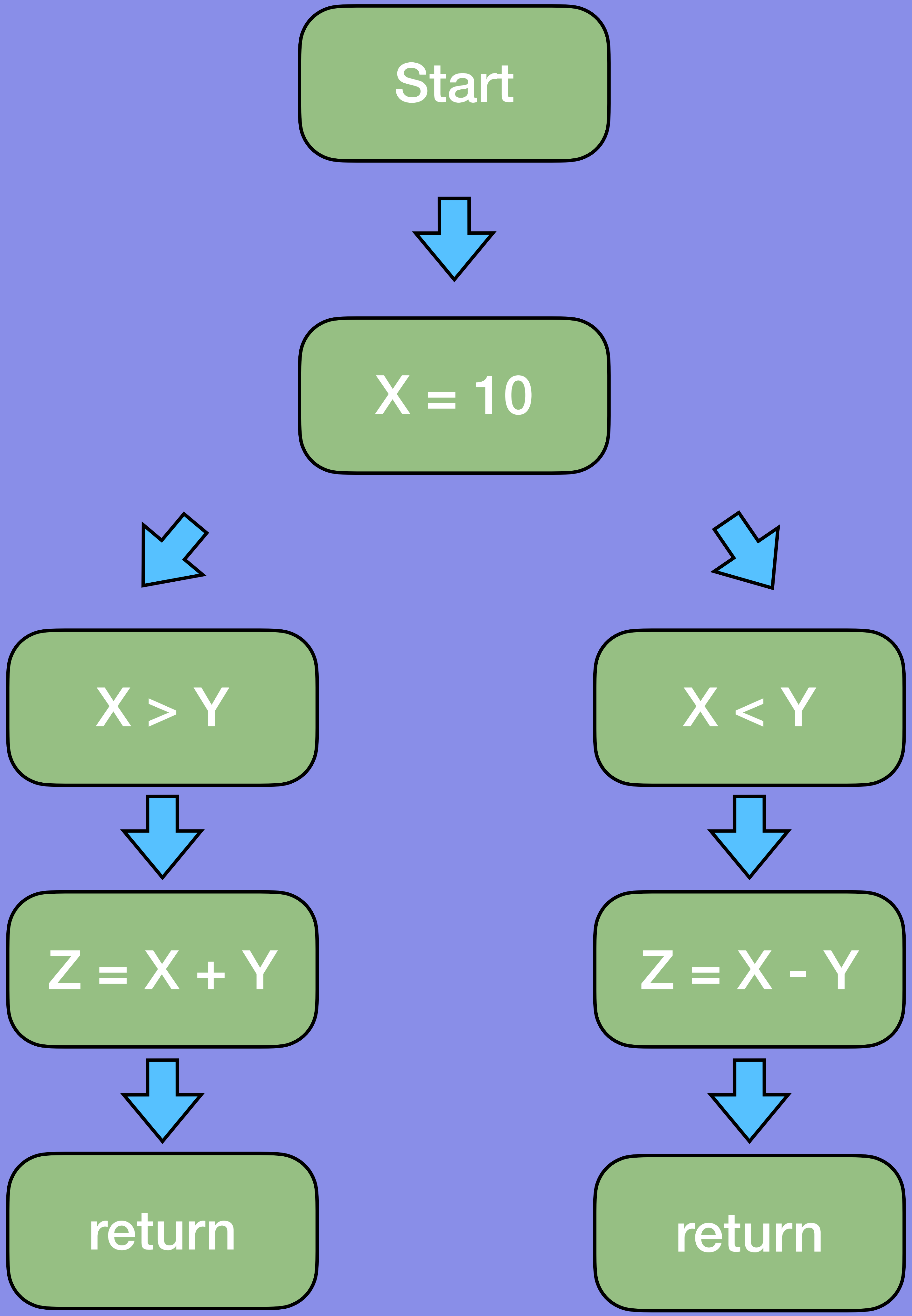
LLVM

- **Is there a dead code?**



```
#include <iostream>

int main() {
    int x = 10;
    int y = 5;
    int z;

    if (x > y) {
        z = x + y;
    } else {
        z = x - y;
    }

    return 0;
}
```

C++ code



Start

X = 10

X > Y

Z = X + Y

return

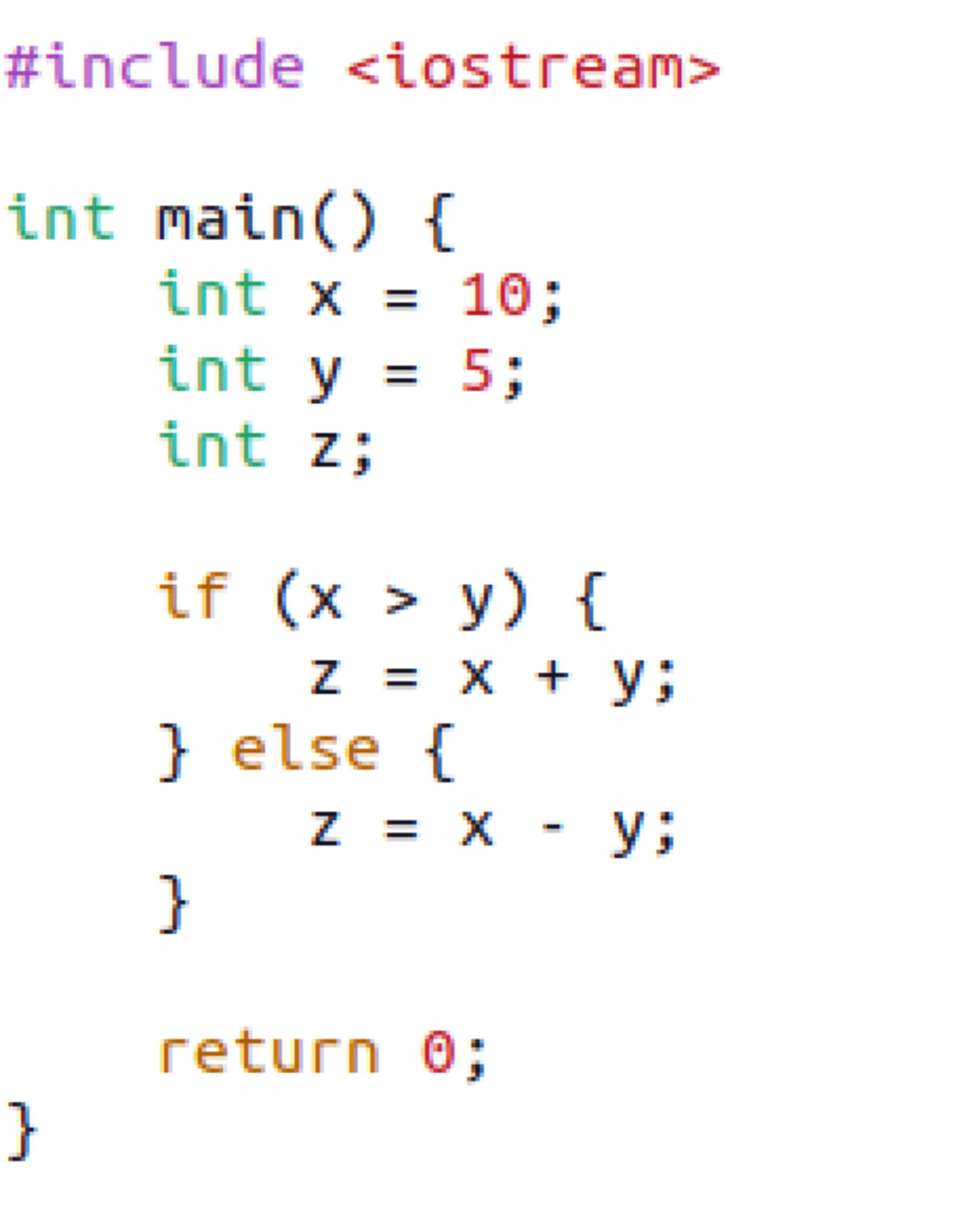
X < Y

Z = X - Y

return

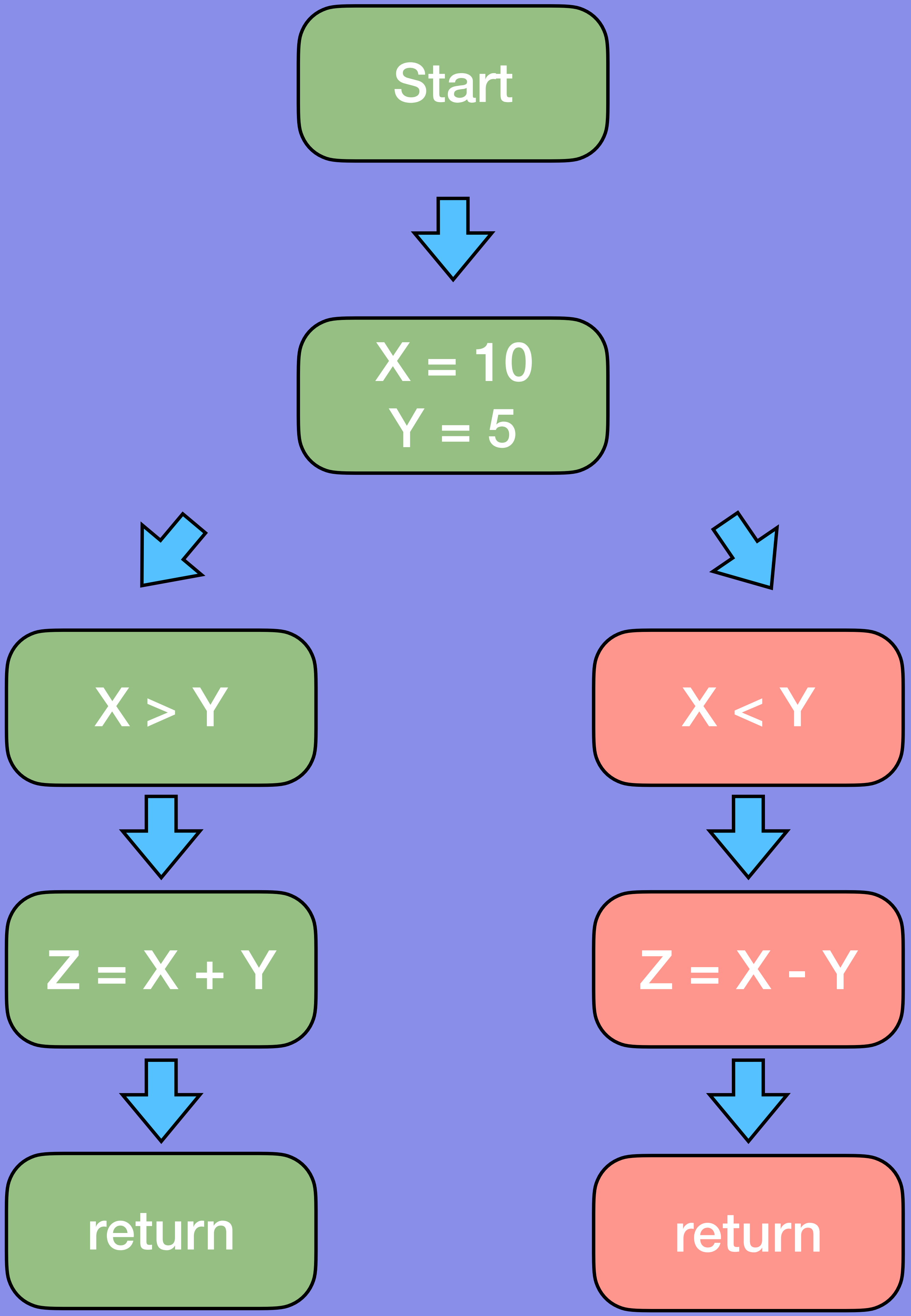
LLVM

- Yes!



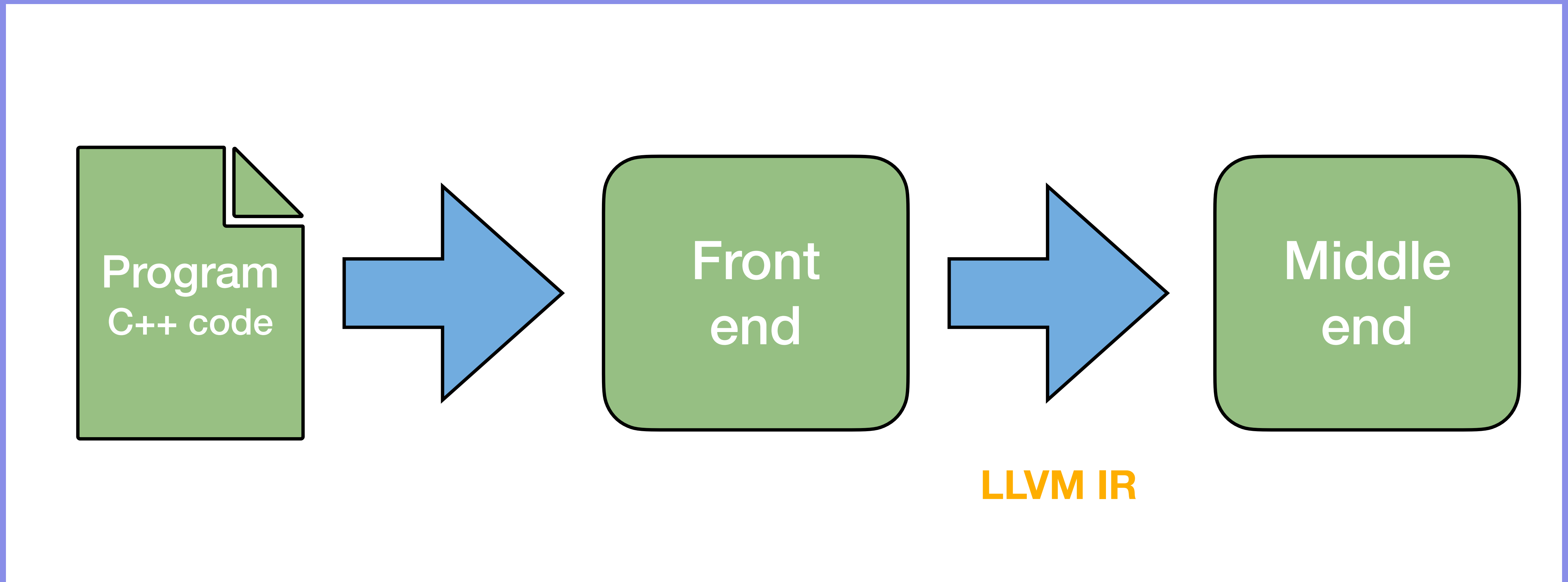
```
#include <iostream>

int main() {
    int x = 10;
    int y = 5;
    int z;

    if (x > y) {
        z = x + y;
    } else {
        z = x - y;
    }

    return 0;
}
```

C++ code



Motivation?

- Introduction to LLVM

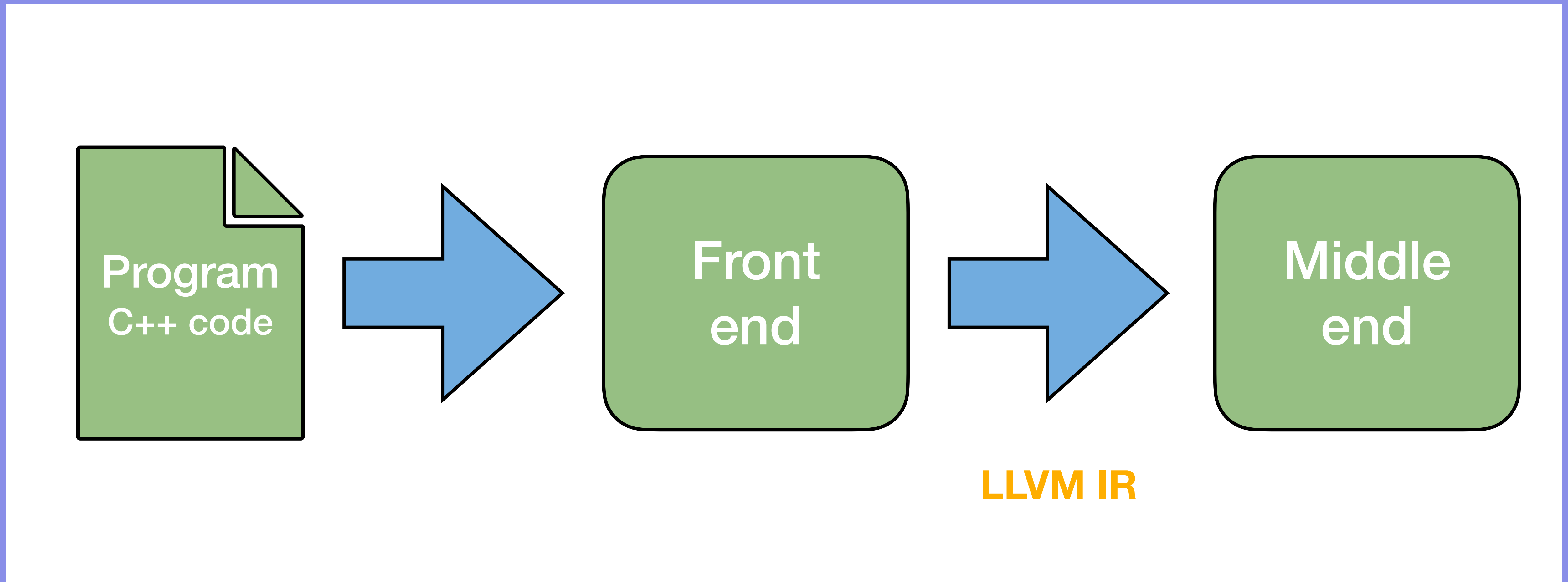


What happens inside the middle end?

- Data Flow Analysis (DFA)
- Control Flow Analysis (CFA)

Motivation?

- Introduction to LLVM

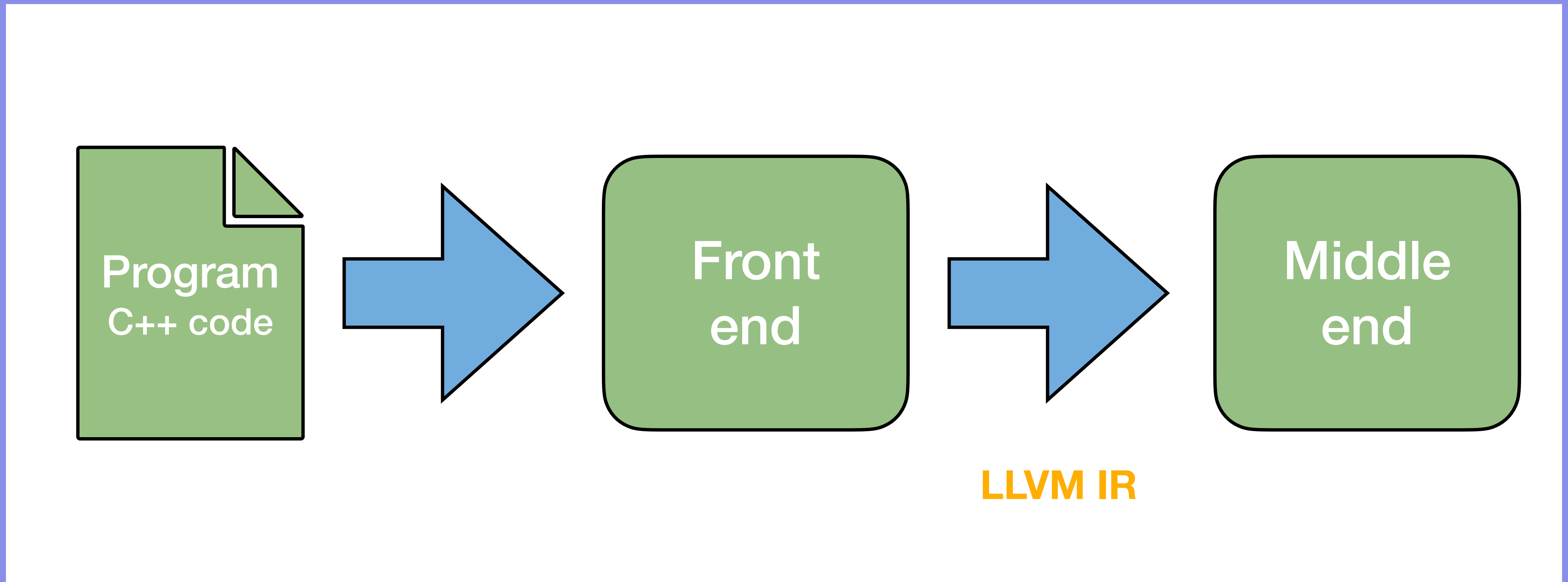


What happens inside the middle end?

- Data Flow Analysis (DFA)
- Control Flow Analysis (CFA)
- Alias Analysis (AA)

Motivation?

- Introduction to LLVM

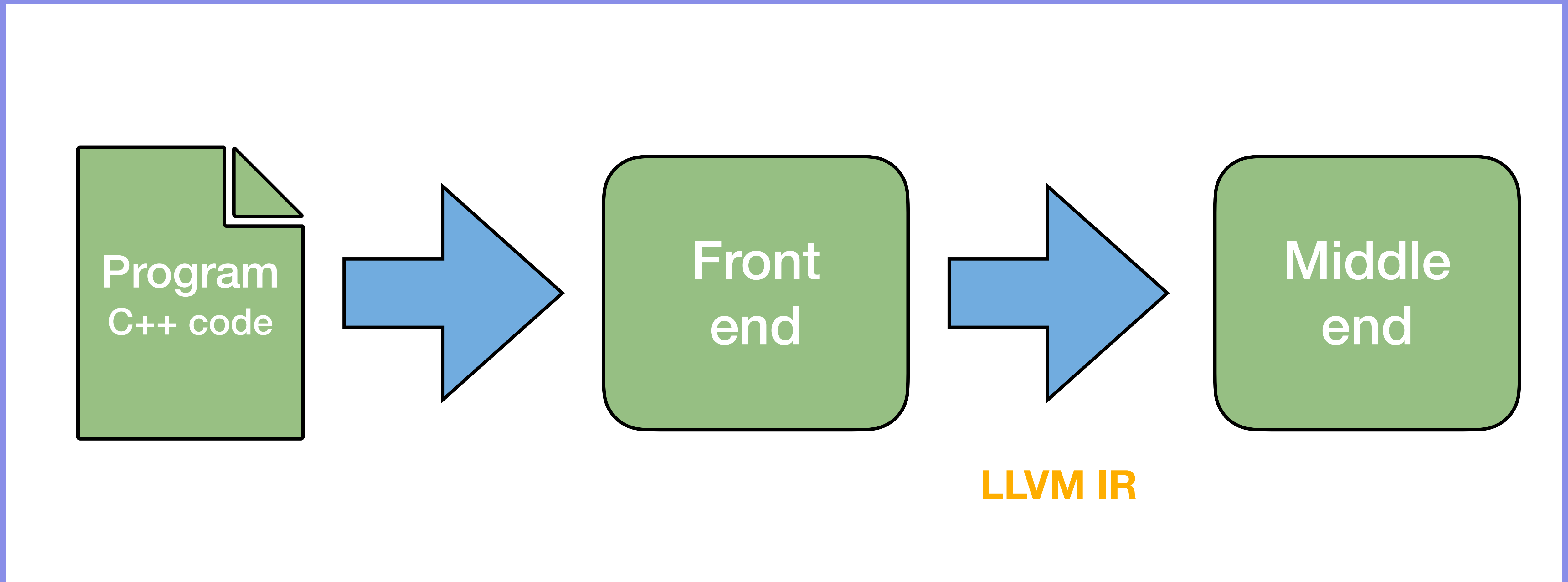


What happens inside the middle end?

- Data Flow Analysis (DFA)
- Control Flow Analysis (CFA)
- Alias Analysis (AA)
- Data Dependence Analysis (DDA)

Motivation?

- Introduction to LLVM

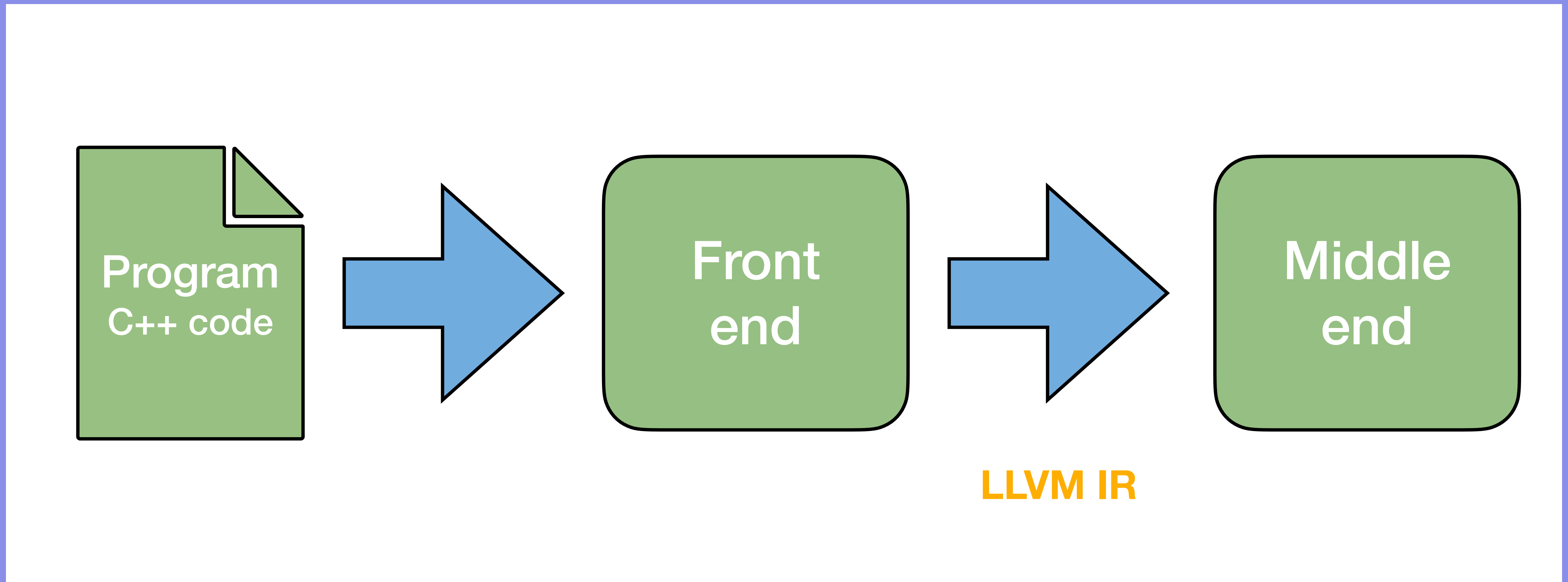


What happens inside the middle end?

- Optimizations
  - Transformation passes
  - Analysis passes

Motivation?

- Introduction to LLVM



What happens inside the middle end?

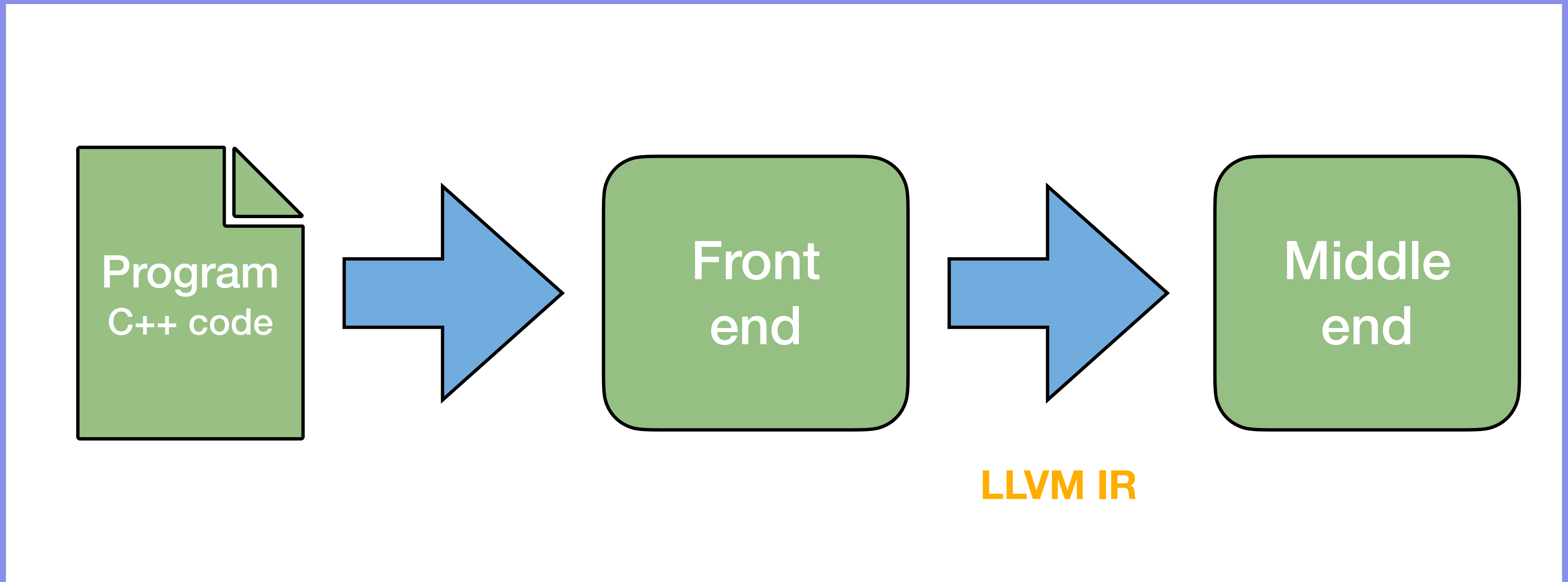
- Optimizations
- Optimization ordering

LLVM

- Optimization ordering
  - The reason behind ordering?
  - What if there is a functionality change?
  - Transformation and Analysis passes.

Motivation?

- Introduction to LLVM

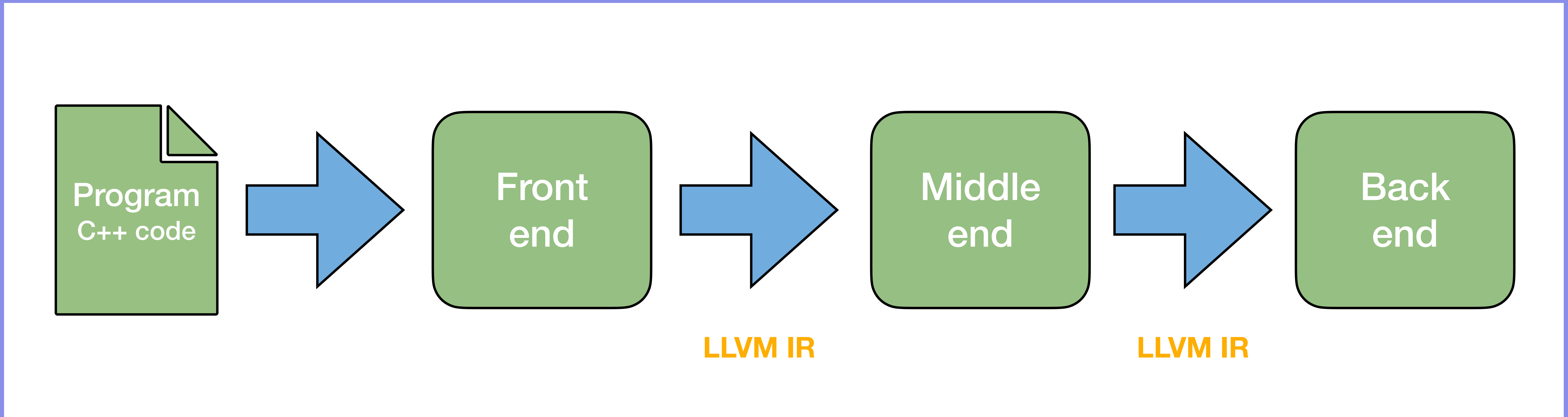


What happens inside the middle end?

- Optimizations
- Optimization ordering
- Generating optimized LLVM-IR

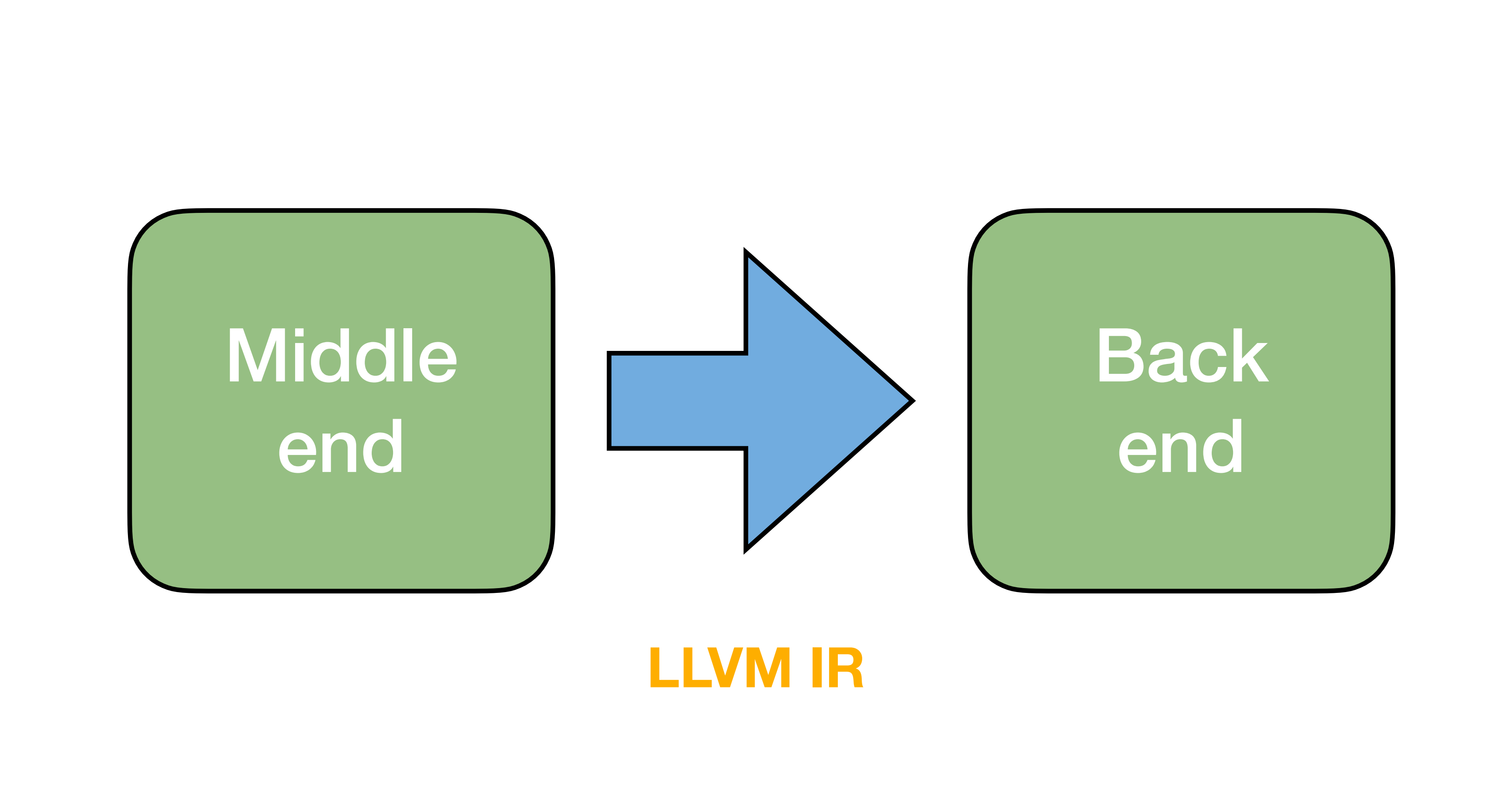
Motivation?

- Introduction to LLVM



Motivation?

- Introduction to LLVM

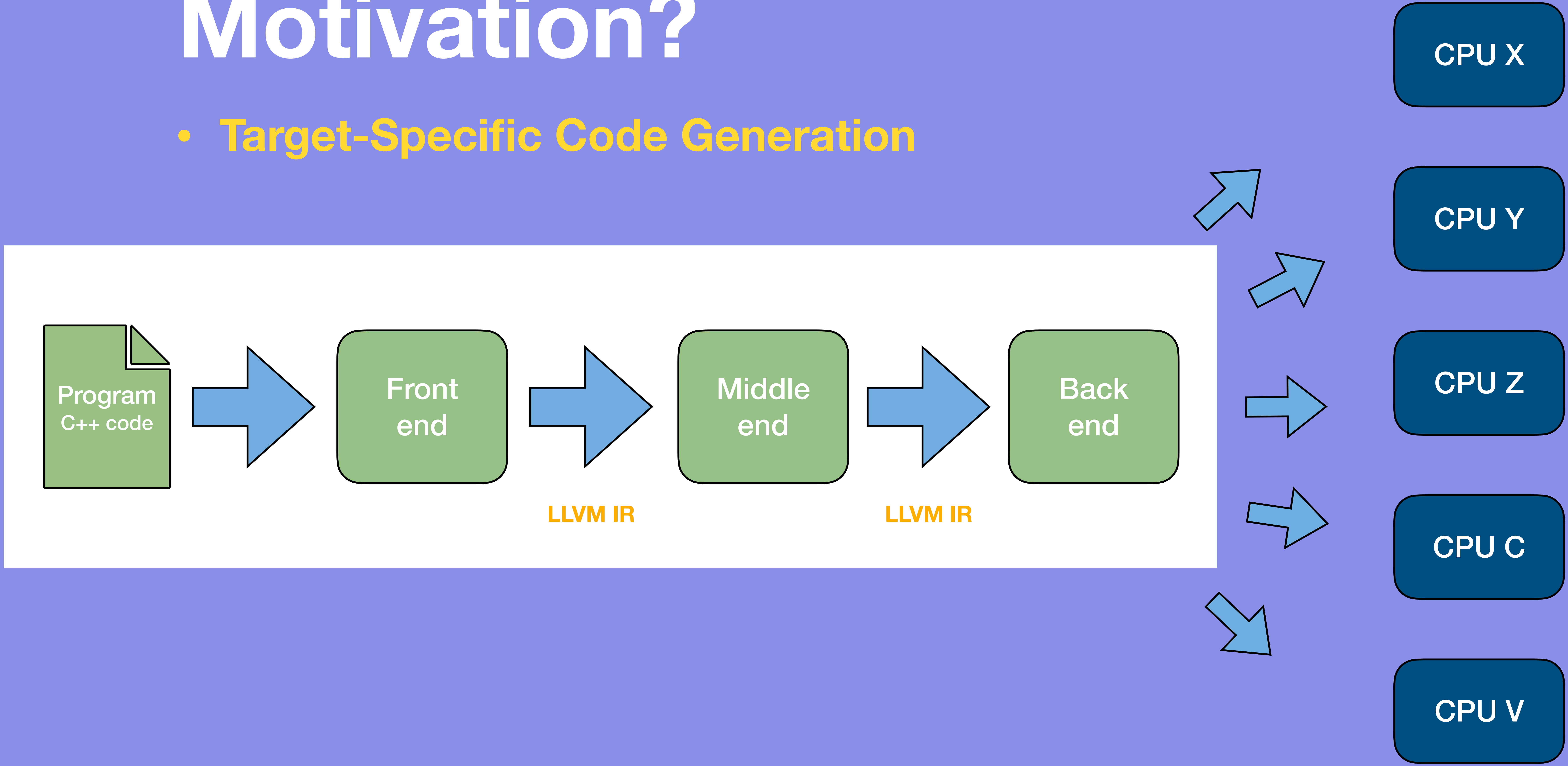


What happens inside the back end?

- Target-Specific Code Generation

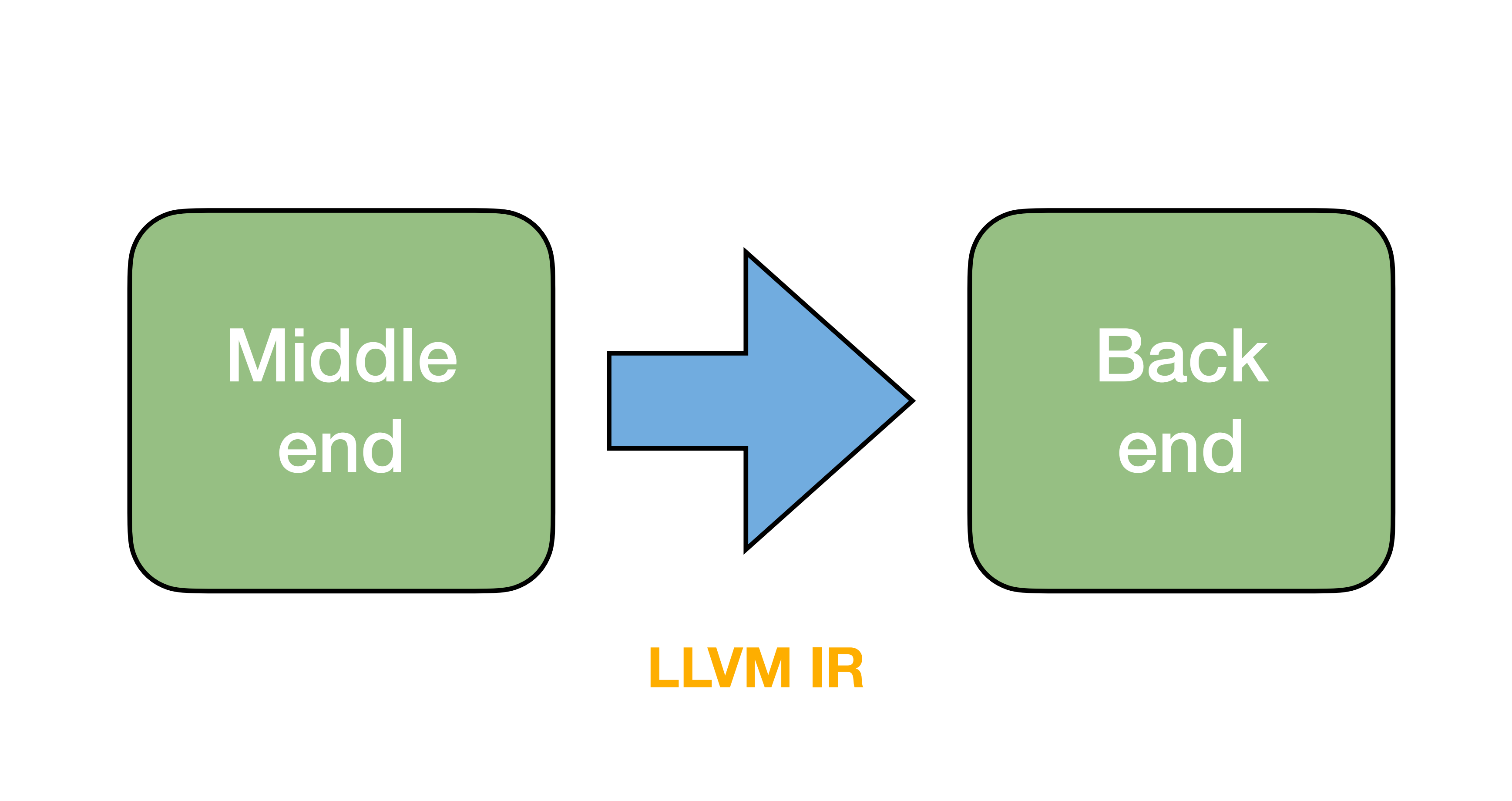
Motivation?

- **Target-Specific Code Generation**



Motivation?

- Introduction to LLVM



What happens inside the back end?

- Target-Specific Code Generation
- Instruction Selection

Motivation?

- Instruction Selection
    - Instruction Matching
    - Cost analysis and pattern matching
    - DAG (Directed Acyclic Graph)

Motivation?

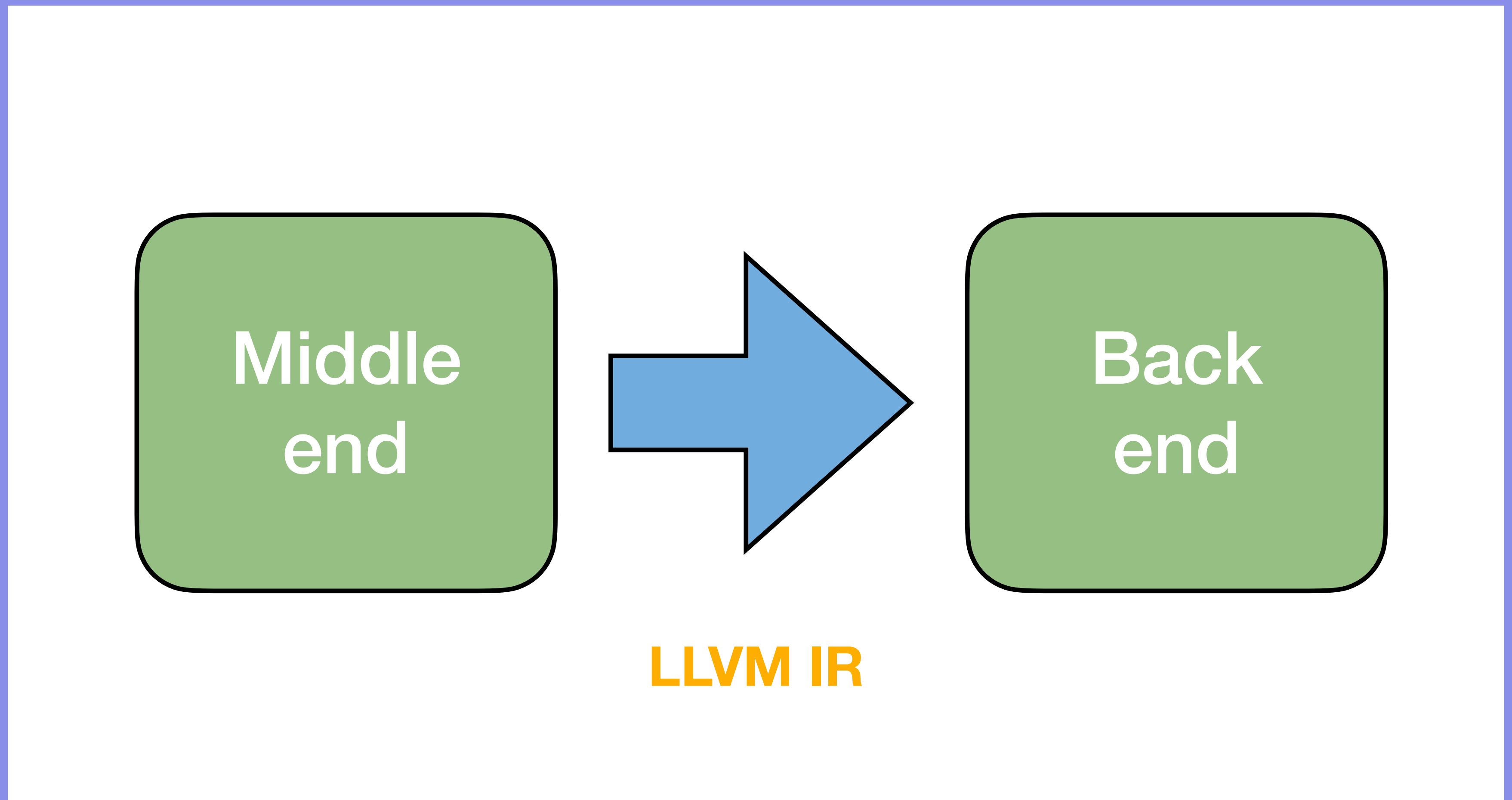
- Instruction Selection
  - DAG (Directed Acyclic Graph)
    - DAG (Directed Acyclic Graph) - It captures the dependencies and operations of the program as nodes and edges in a directed graph. Each node in the DAG represents an operation or value, and the edges represent the data flow between them.
    - Overall, DAG-based instruction selection in LLVM's backend is crucial in mapping high-level IR to target-specific machine code, enabling efficient and optimized code generation for a wide range of target architectures.

Motivation?

- Instruction Selection
  - Global ISEL (Global Instruction Selection)
    - It aims to improve instruction selection by performing the selection process across the entire function or module globally rather than on a per-basic-block basis, as done in the DAG approach.
    - Improved Code Quality: By considering a broader context and optimizing across the entire function or module.
    - Code Sharing: Global ISEL can identify opportunities for code sharing and reuse across different basic blocks and paths, leading to reduced code size and improved cache locality.
    - Simplified Code Generation: With Global ISEL, the instruction selection process becomes more unified and cohesive since it operates globally.

Motivation?

- Introduction to LLVM



What happens inside the back end?

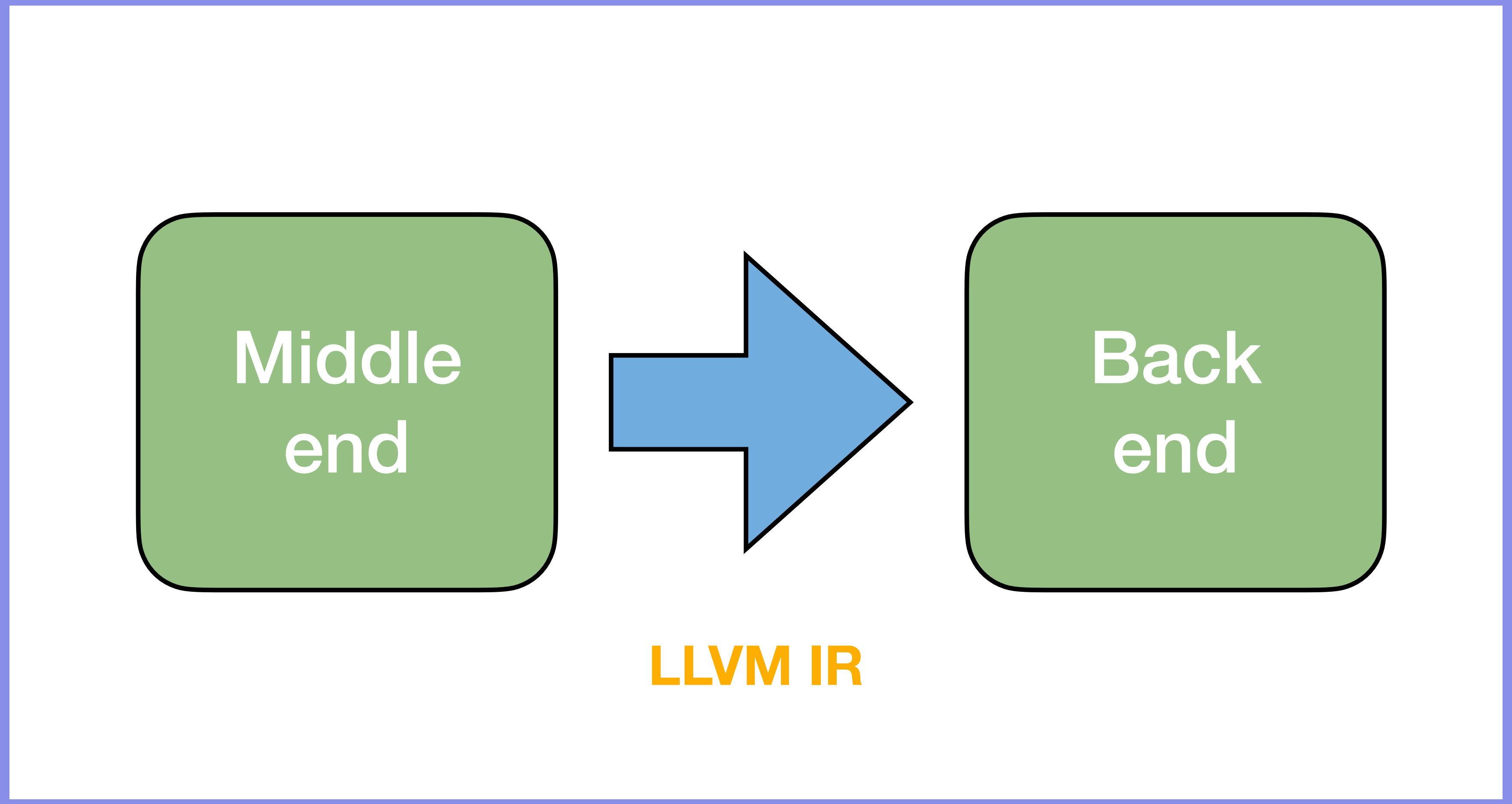
- Target-Specific Code Generation
- Instruction Selection
- Instruction Scheduling

Motivation?

- Instruction Scheduling
  - The backend determines the order of instructions to be executed to maximize the performance of the generated code.
  - Instruction scheduling considers factors such as instruction dependencies, pipeline hazards, and the target architecture's specific execution characteristics to minimize stalls and improve instruction-level parallelism.

Motivation?

- Introduction to LLVM



What happens inside the back end?

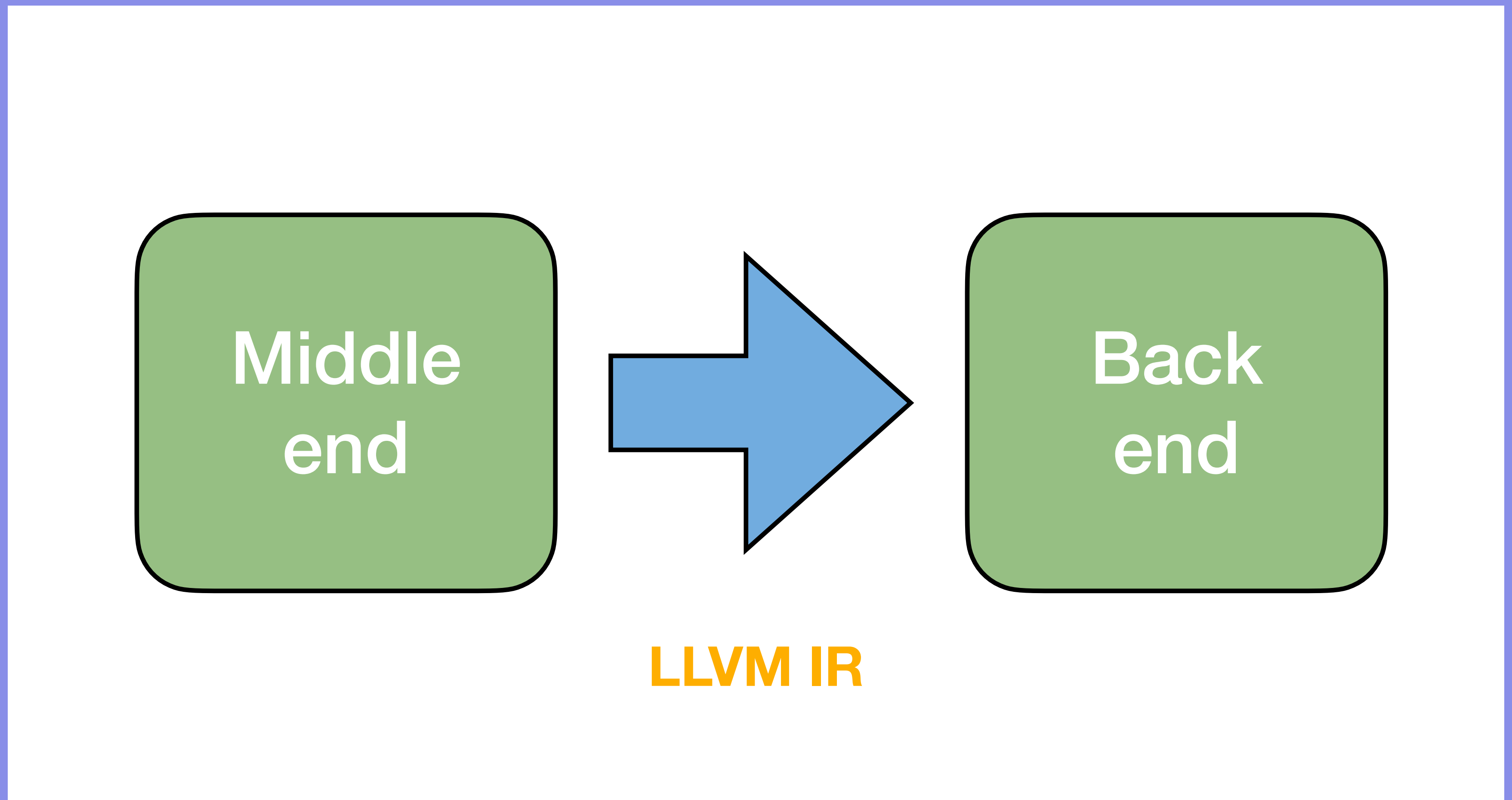
- Target-Specific Code Generation
- Instruction Selection
- Instruction Scheduling
- Register Allocation

Motivation?

- Register Allocation
  - Virtual Register Allocation - virtual registers are initially unlimited and can hold any value.
  - This allows for efficient analysis and optimization without the limitations of physical registers.
  - Register Interference Analysis - determine which virtual registers may conflict with each other.

Motivation?

- Introduction to LLVM

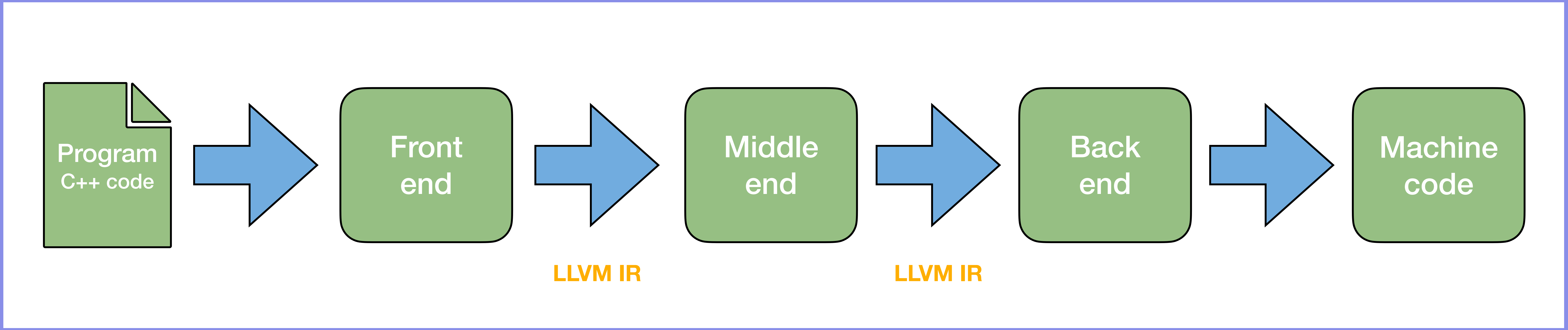


What happens inside the back end?

- Target-Specific Code Generation
- Instruction Selection
- Instruction Scheduling
- Register Allocation
- Code emission

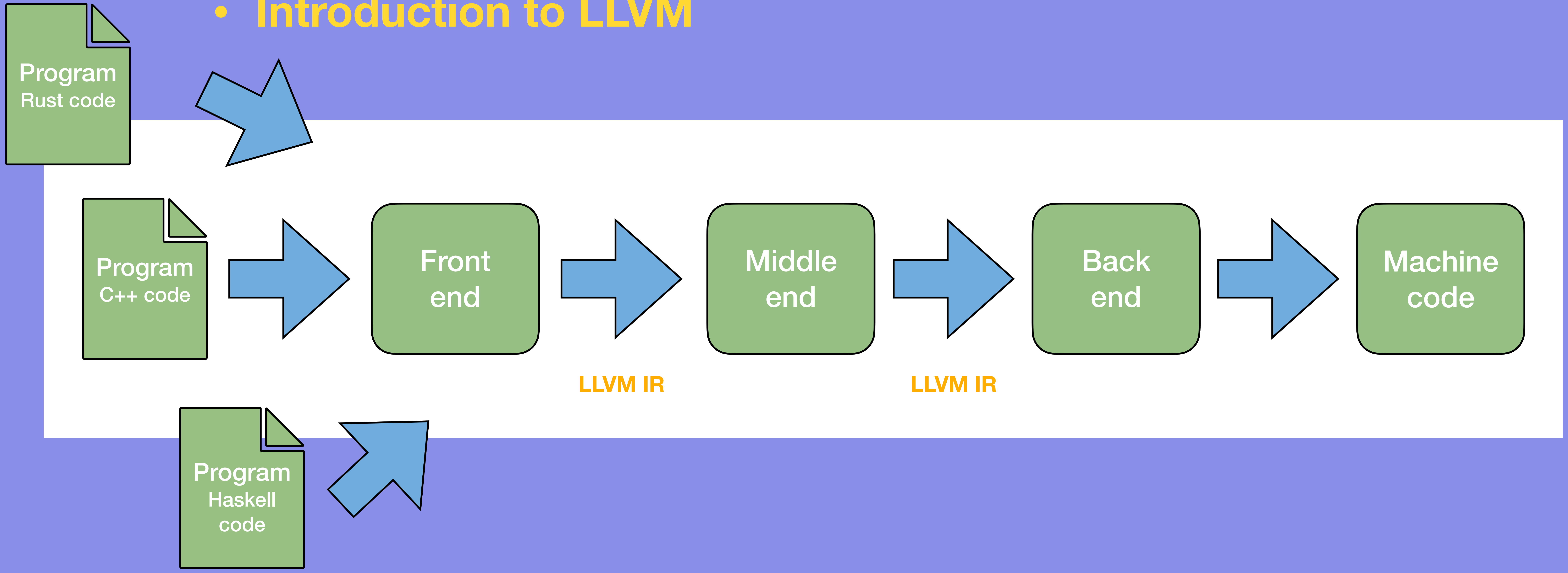
Motivation?

- Introduction to LLVM



Motivation?

- Introduction to LLVM



Motivation?

- Benefits of LLVM
  - Modularity and Extensibility

Motivation?

- Benefits of LLVM
  - Modularity and Extensibility
  - Portability

Motivation?

- Benefits of LLVM
  - Modularity and Extensibility
  - Portability
  - Optimizations

Motivation?

- Benefits of LLVM
  - Modularity and Extensibility
  - Portability
  - Optimizations
  - Just in time, execute the code on the fly

Motivation?

- Benefits of LLVM
  - Modularity and Extensibility
  - Portability
  - Optimizations
  - Just in time, execute the code on the fly
  - Supported tools (LLDB, GDB)

Motivation?

- Benefits of LLVM
  - Modularity and Extensibility
  - Portability
  - Optimizations
  - Just in time, execute the code on the fly
  - Supported tools (LLDB, GDB)
  - Community and easier Adoption

Thank you :)