COMPILER DESIGN

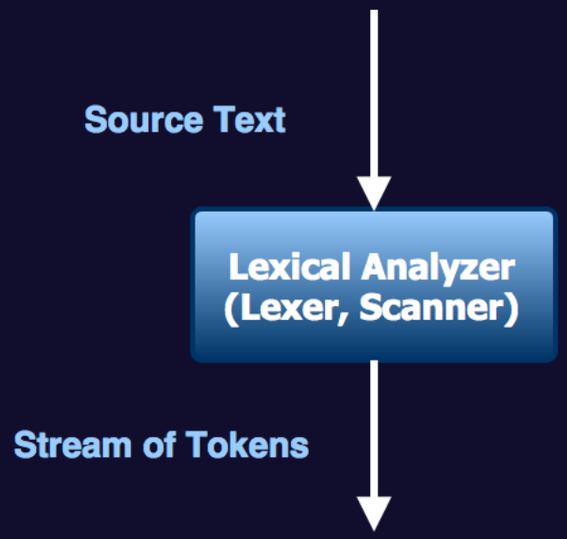
AND IMPLEMENTATION

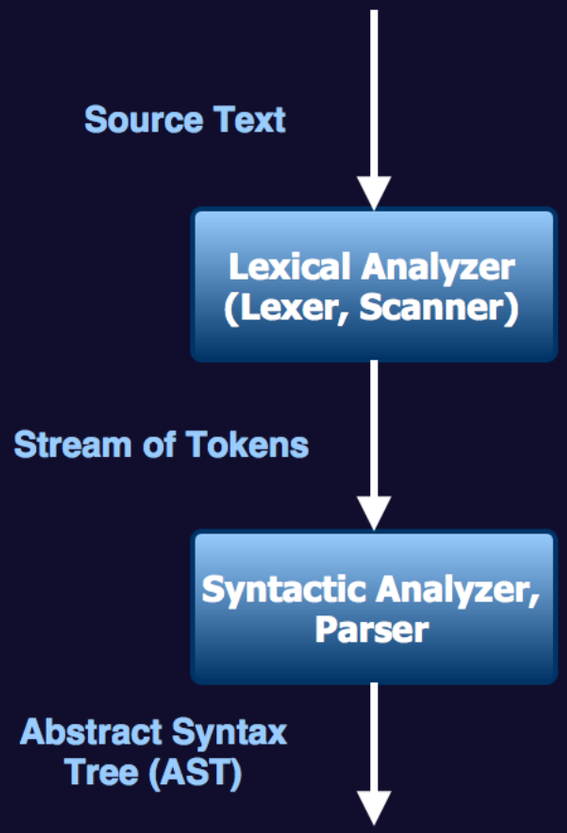
ÁRPÁD GORETITY
BUDAPEST SWIFT MEETUP

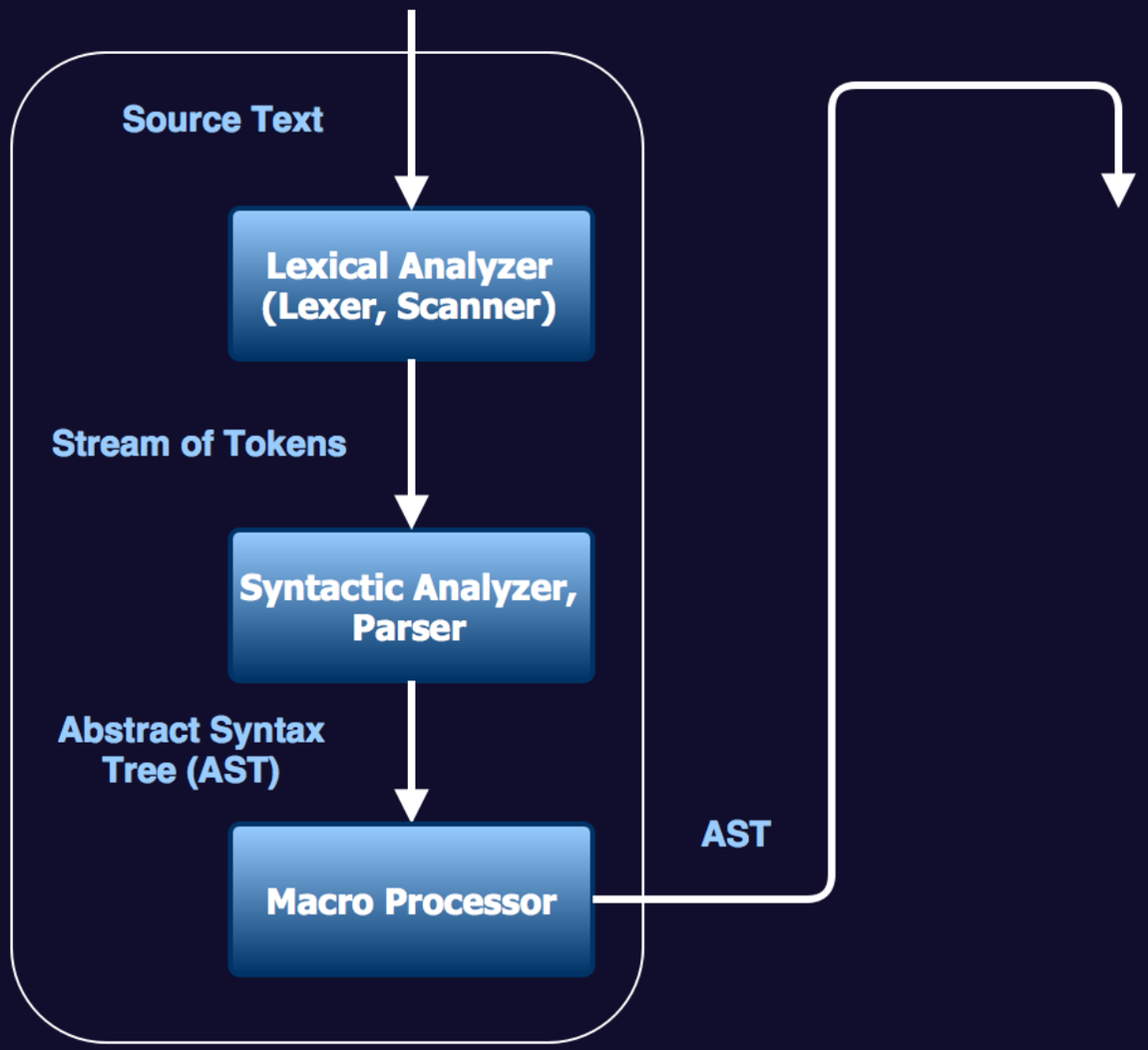
CODE GENERATION

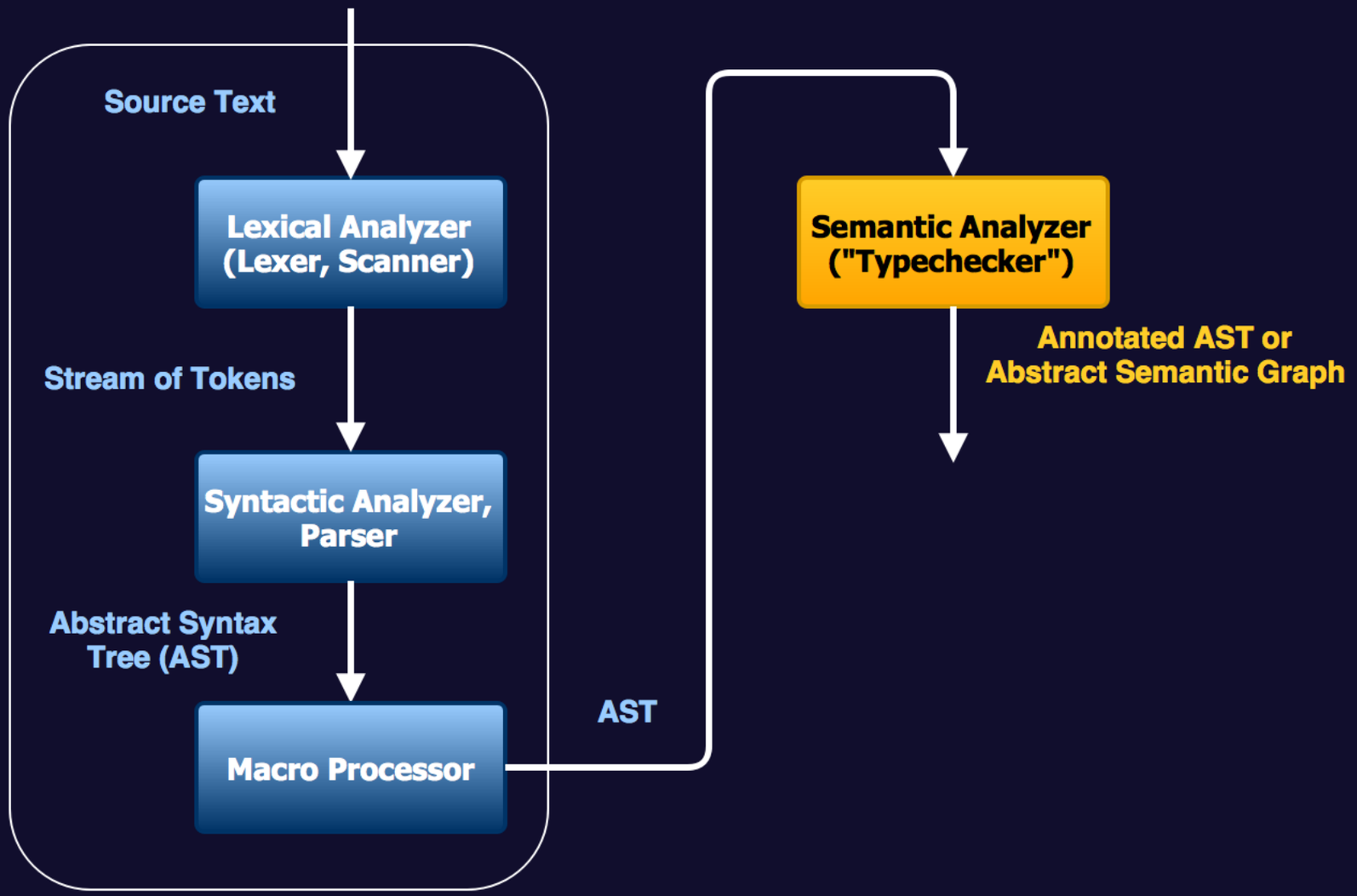
PART 4

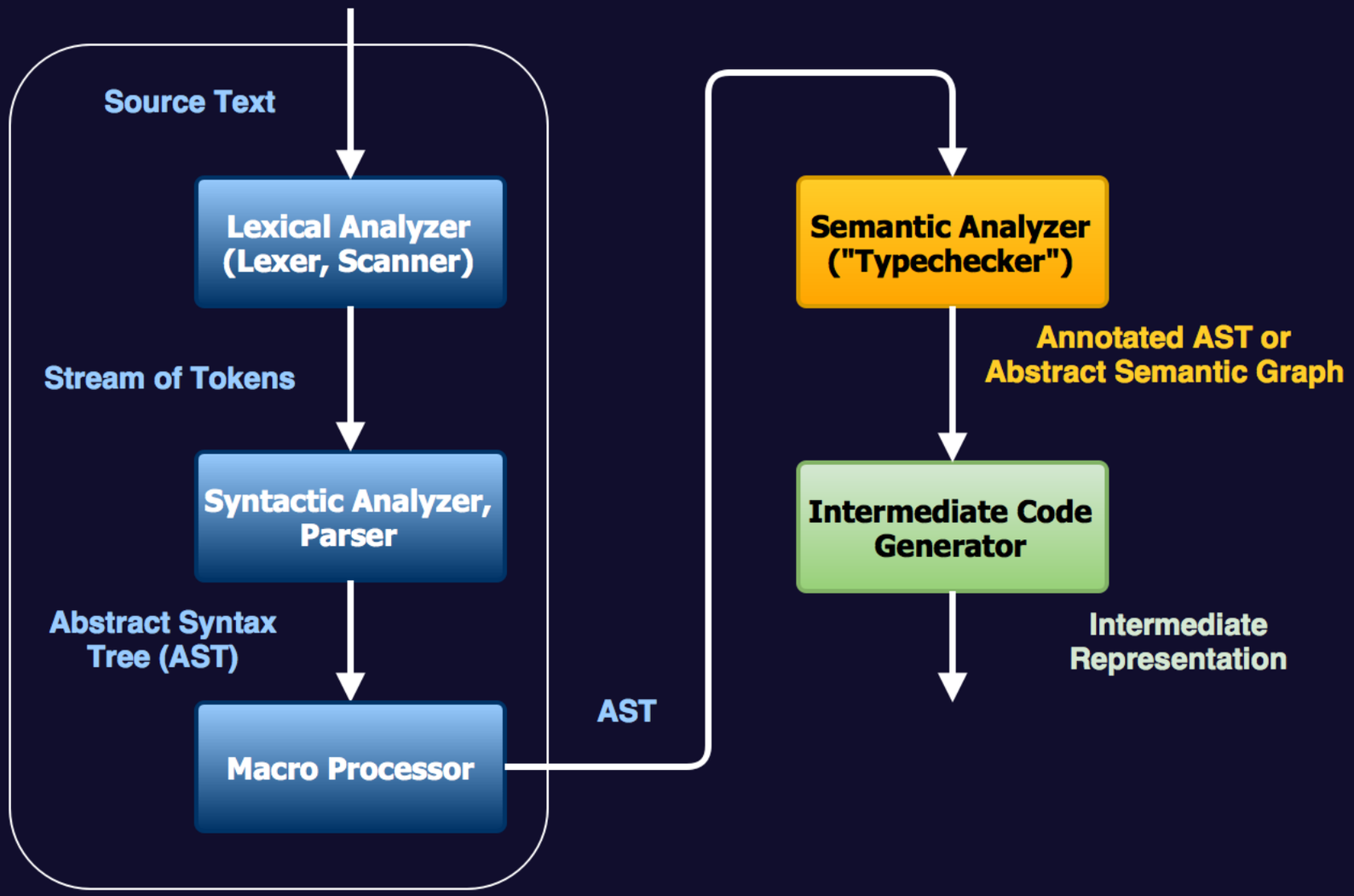
REMINDER

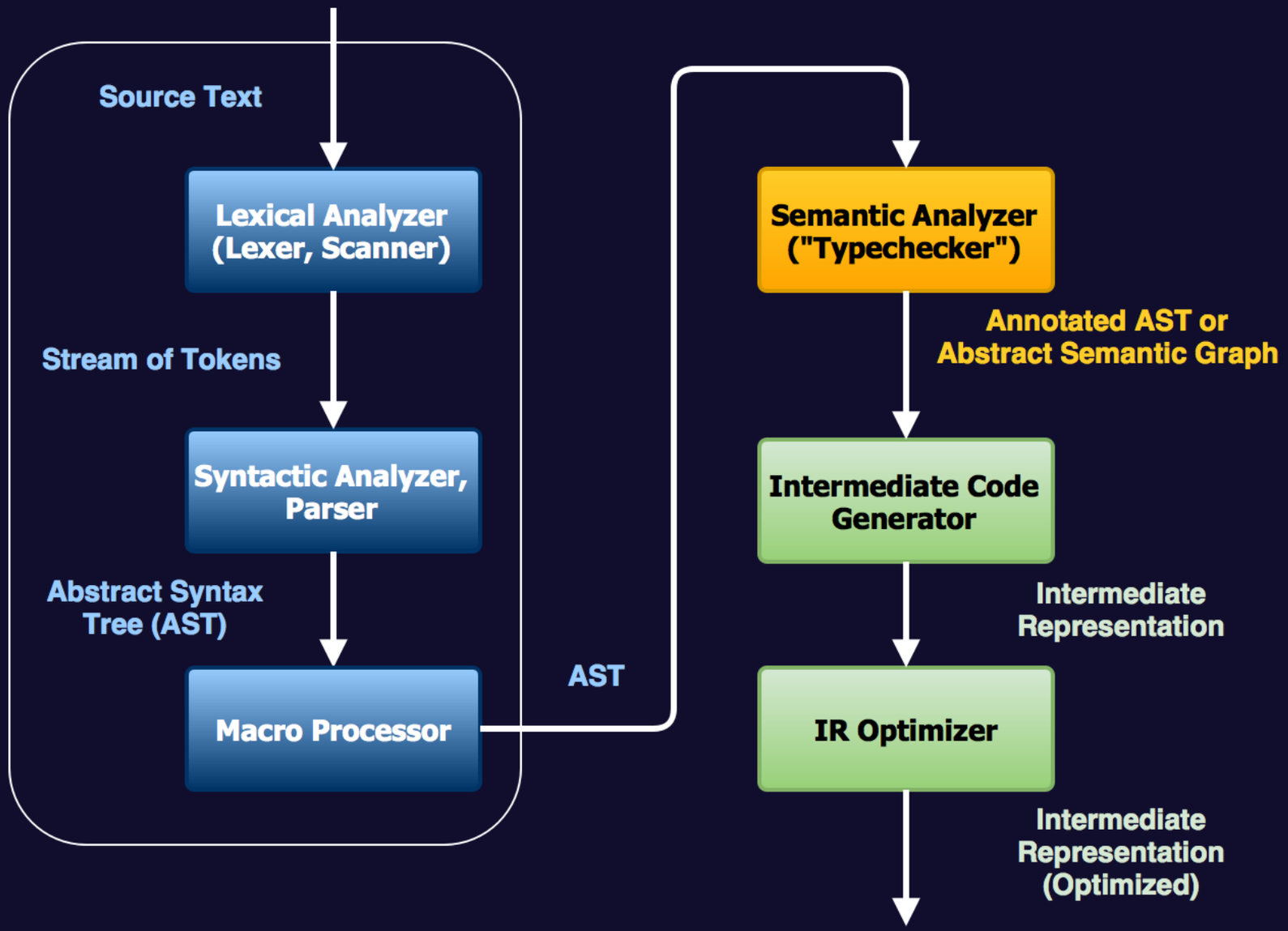

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens

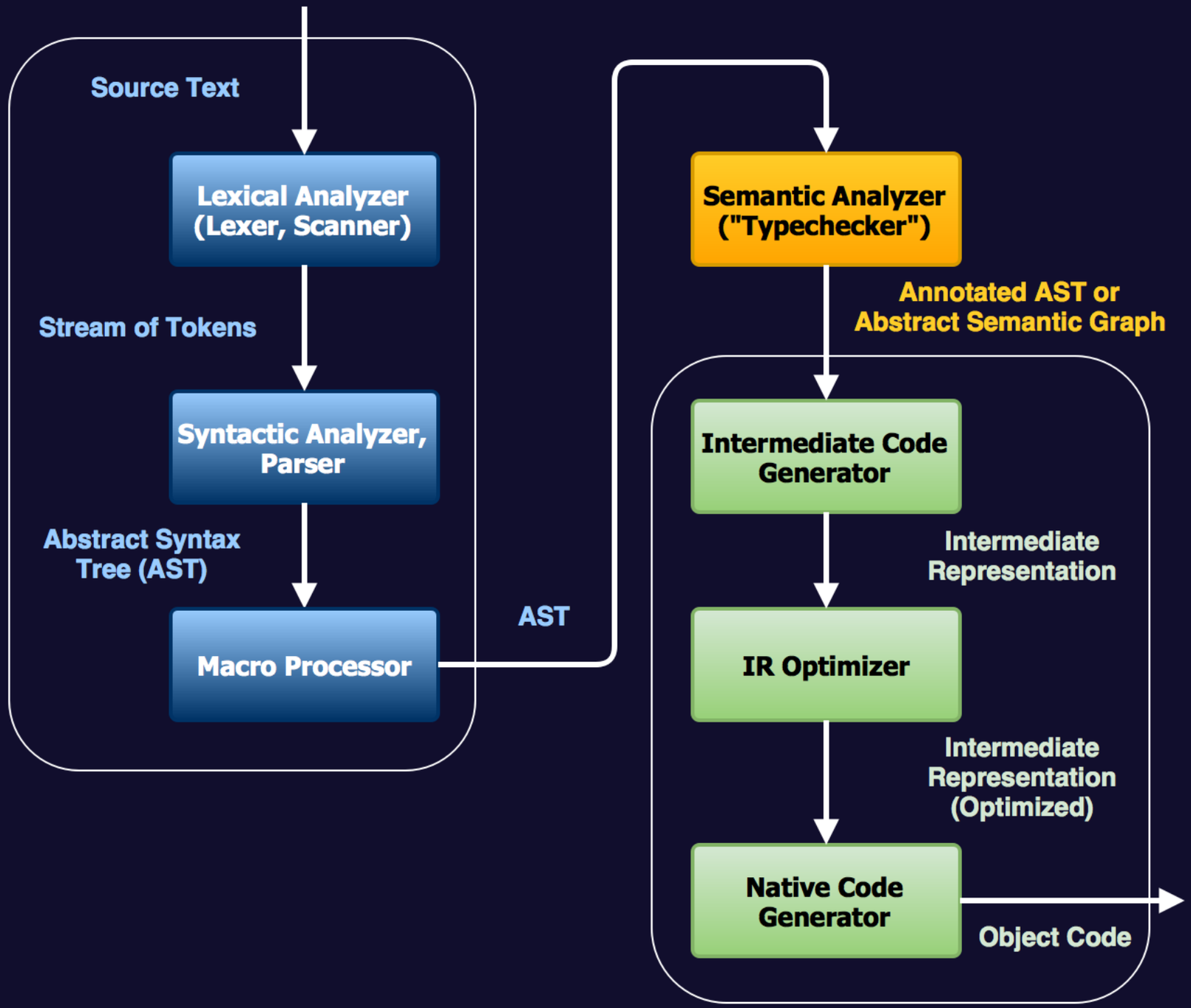

Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)

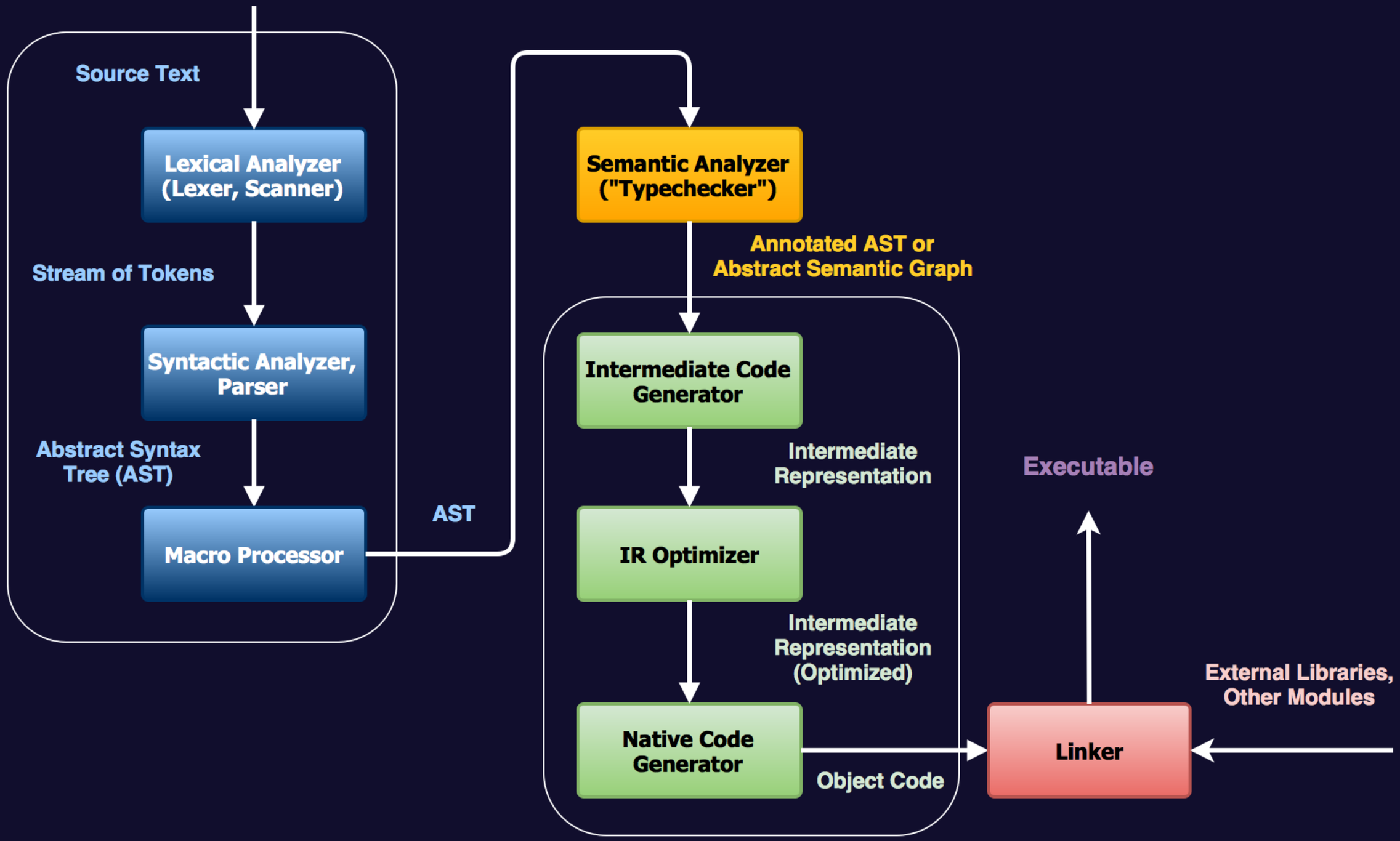

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer,
Parser
Abstract Syntax
Tree (AST)
Macro Processor
AST

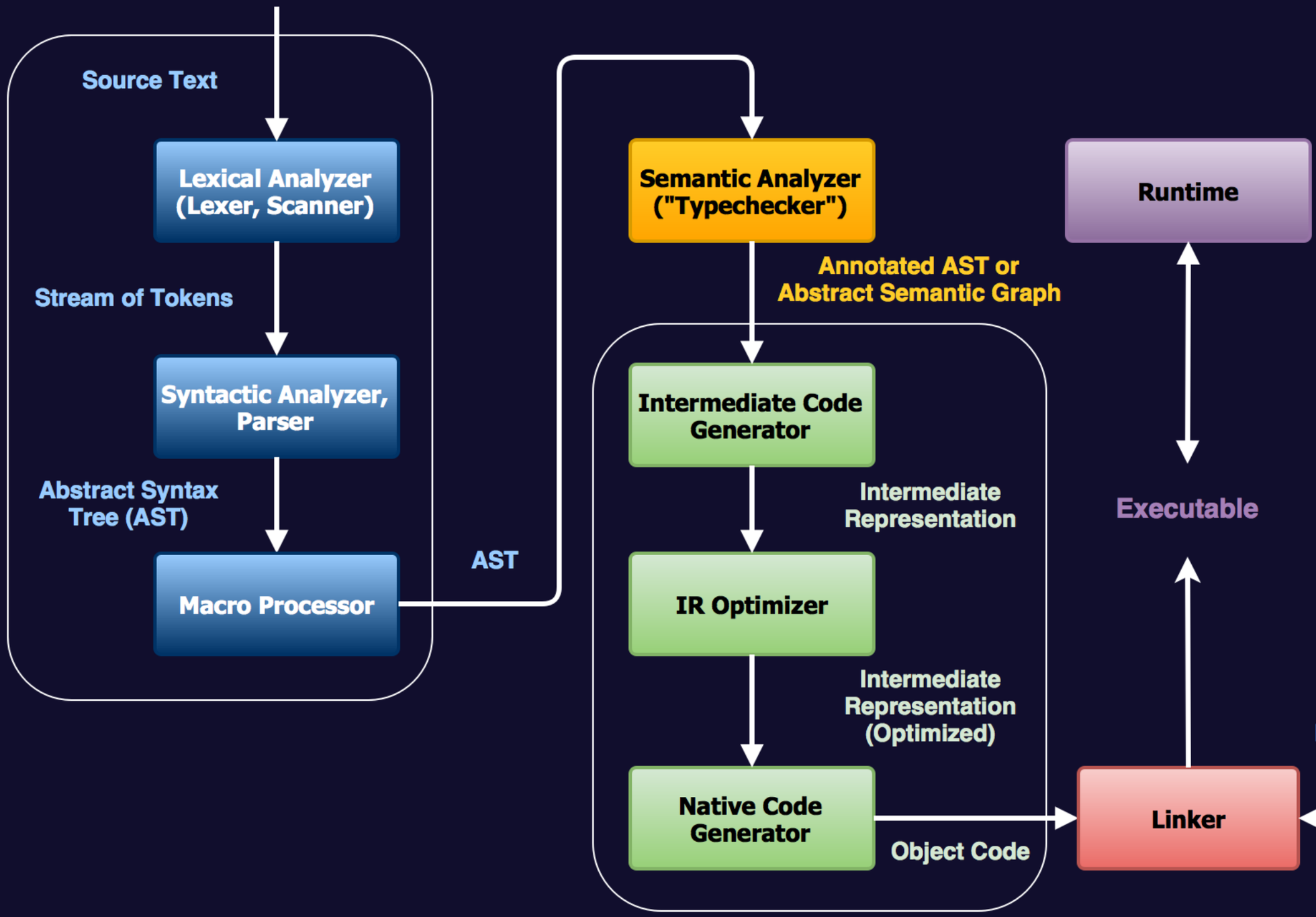

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer,
Parser
Abstract Syntax
Tree (AST)
Macro Processor
AST
Semantic Analyzer
("Typechecker")
Annotated AST or
Abstract Semantic Graph


Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation


Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation
IR Optimizer
Intermediate Representation (Optimized)


Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation
IR Optimizer
Intermediate Representation (Optimized)
Native Code Generator
Object Code


Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation
IR Optimizer
Intermediate Representation (Optimized)
Native Code Generator
Object Code
Linker
Executable
External Libraries, Other Modules


Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation
IR Optimizer
Intermediate Representation (Optimized)
Native Code Generator
Object Code
Runtime
Executable
Linker
External Libraries, Other Modules

REMINDER

- Semantic Analysis
  - Input: AST
  - Output: IR (high-level or low-level)
- If succeeds: program "makes sense"
  - At this point, abstractions are ready to be collapsed into executable code

CODE GENERATION

- Process that transforms language-dependent IR into language-agnostic executable code
  - Directly binary machine code: rare, e.g. TCC
  - Assembly: was very popular, e.g. GCC
  - Low-level but machine-independent IR: e.g. Clang
    - Most popular: LLVM

OH NOES, NOT ANOTHER IR!

- Machine-independent, yet low-level optimizations
- Native code generation –> **Some** of ABI abstracted away
- Linker support (native and IR)
  - Opportunity for intermodule optimization, "LTO"
    - Not possible or very hard in native code
- Many languages, same back-end –> less work

THE LLVM COMPILER INFRASTRUCTURE

- IR with 3 first-class representations
  - Binary "bitcode"; Textual "assembly"; C++ objects
- Division of labor – beneficial for compiler writers:
  - Front- / middle-end: generate LLVM IR
  - LLVM: heavy lifting (optimization, native codegen)
  - Implementing the Runtime: still our duty…

LLVM IR

- High-level model of a modern von Neumann architecture
  - Registers – arbitrarily many
  - Memory – explicitly allocated, implicitly deallocated (stack)
    - Explicit loads and stores – Stateful
    - Pointer arithmetic
- Control flow model: Control Flow Graph and Functions
- Data flow model: Static Single Assignment (-ish…)

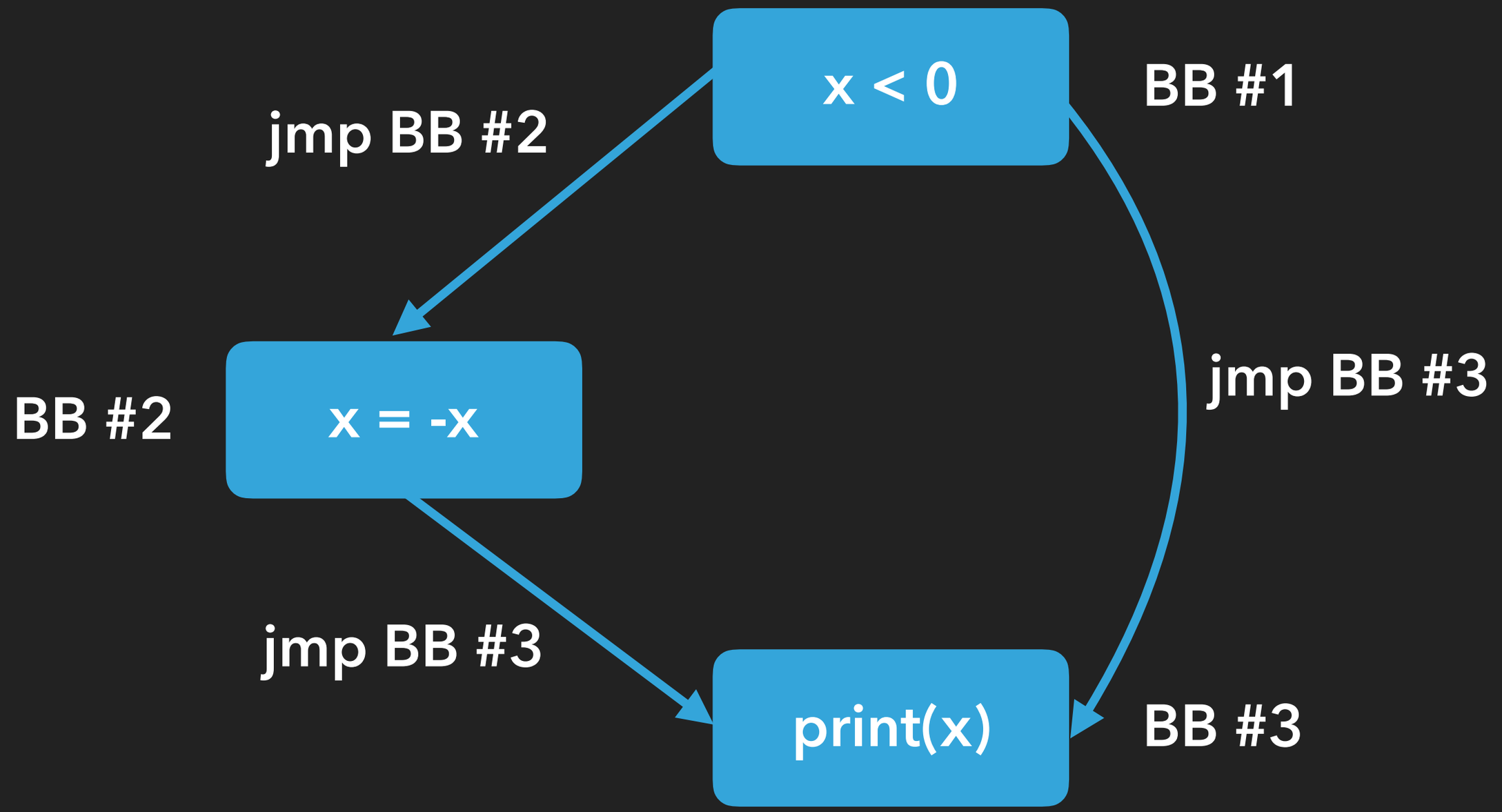
CONTROL FLOW GRAPH (CFG)

- Fundamental abstraction: Basic Blocks
  - Sequence of instructions without control flow inside
  - Jumps and returns must be at the end
  - Target of jump must be the first instruction
  - Function calls are still allowed inside BBs

CONTROL FLOW GRAPH (CFG)

- CFGs are Directed Cyclic Graphs
- Nodes: Basic Blocks
- Edges point from jump instructions to their targets

CONTROL FLOW GRAPH (CFG)



STATIC SINGLE ASSIGNMENT (SSA)

- No mutation: names are permanently bound to one value
- Assignment is represented using versioning
  - New version of variable is created by renaming (typically, by appending integer version numbers)
  - Mentions of original variable refer to the new version thereafter
- Assignment is definition; definition must precede uses

STATIC SINGLE ASSIGNMENT (SSA)

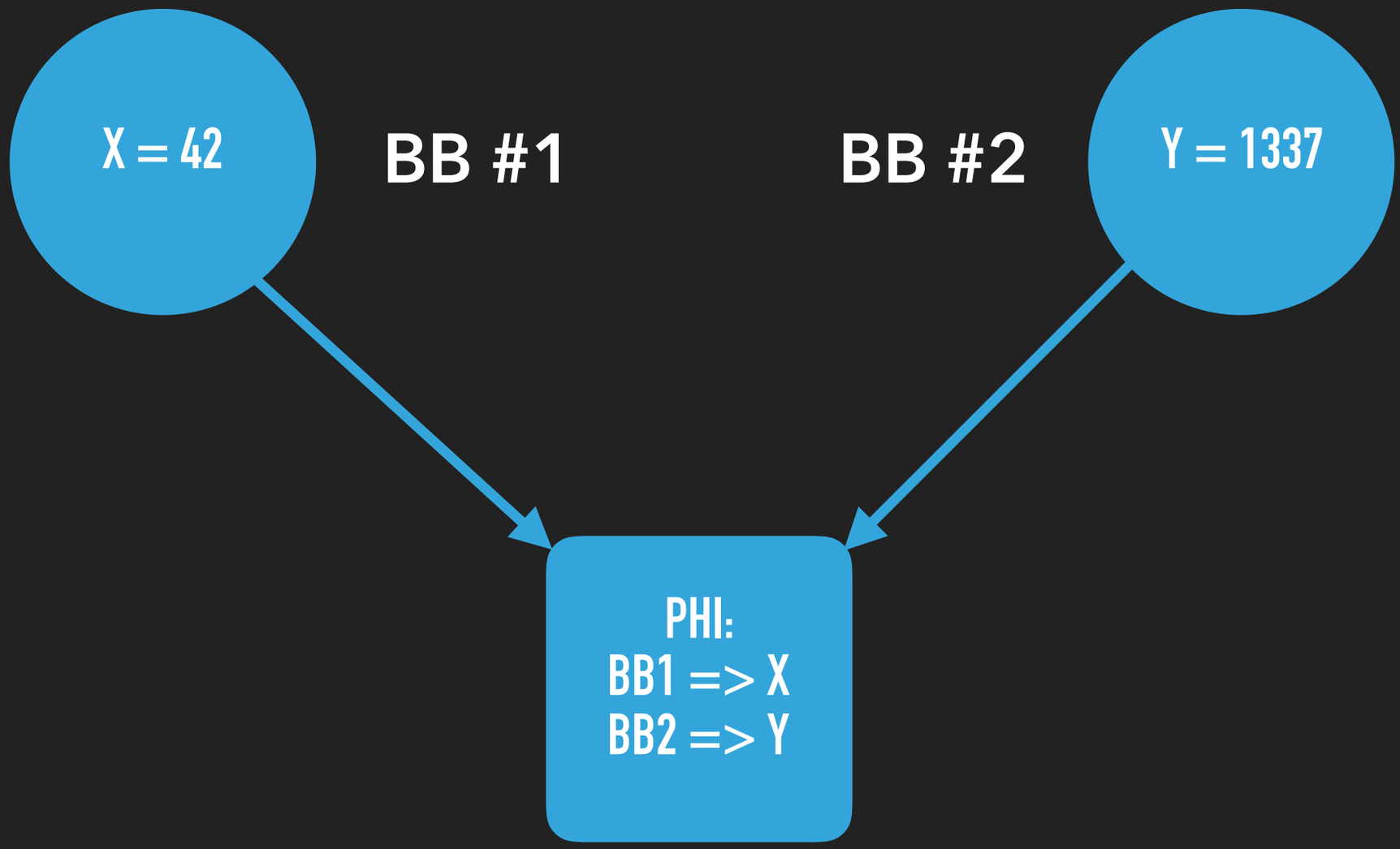
```
var x = 42
print(x)
x = x + 1
print(x)
x = x + 1
print(x)
```

```
x = 42
print(x)
x_1 = x + 1
print(x_1)
x_2 = x_1 + 1
print(x_2)
```

PHI NODES IN SSA

- Flow-sensitive value selection: PHI nodes
  - "Magic": output is one of the inputs, depending on where control flow comes from (incoming basic blocks)

PHI NODES IN SSA



LLVM BITCODE

- Our example: Print Absolute Value

```
void print_abs(int x) {
    if (x < 0) {
        x = -x;
    }
    print(x);
}
```

LLVM BITCODE

```
define void @print_abs(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32, i32* %1, align 4
    %3 = icmp slt i32 %2, 0
    br i1 %3, label %4, label %7

; <label>:4                                      ; preds = %0
    %5 = load i32, i32* %1, align 4
    %6 = sub nsw i32 0, %5
    store i32 %6, i32* %1, align 4
    br label %7

; <label>:7                                      ; preds = %4, %0
    %8 = load i32, i32* %1, align 4
    call void @print(i32 %8)
    ret void
}
```

LLVM BITCODE

- Compute Absolute Value

```
int abs(int x) {
    return x < 0 ? -x : x;
}
```

LLVM BITCODE

```
define i32 @abs(i32 %x) #0 {
    %1 = alloca i32, align 4
    store i32 %x, i32* %1, align 4
    %2 = load i32, i32* %1, align 4
    %3 = icmp slt i32 %2, 0
    br i1 %3, label %4, label %7

; <label>:4                                      ; preds = %0
    %5 = load i32, i32* %1, align 4
    %6 = sub nsw i32 0, %5
    br label %9

; <label>:7                                      ; preds = %0
    %8 = load i32, i32* %1, align 4
    br label %9

; <label>:9                                      ; preds = %7, %4
    %10 = phi i32 [ %6, %4 ], [ %8, %7 ]
    ret i32 %10
}
```

LLVM API ESSENTIALS

- Modules and Functions
- Values: Constants, Instructions
- Basic Blocks (are also Values)
- Builders: Convenience API for typical, basic usage
- Passes: Individual transform algorithms (e.g. optimizations)
- C++ API: cutting-edge, latest and greatest
- C API: Stable, can be bridged to Swift

LET'S GET TO WORK!