Compiler Design and Implementation

Árpád Goretity
@H2CO3_iOS

Budapest Swift Meetup 2015

Part 1:

Introduction
The Lexer

What is this course about?

- Writing a compiler for a programming language
- Tiny toy language, but an actual, real compiler
- Little theory, mostly practice

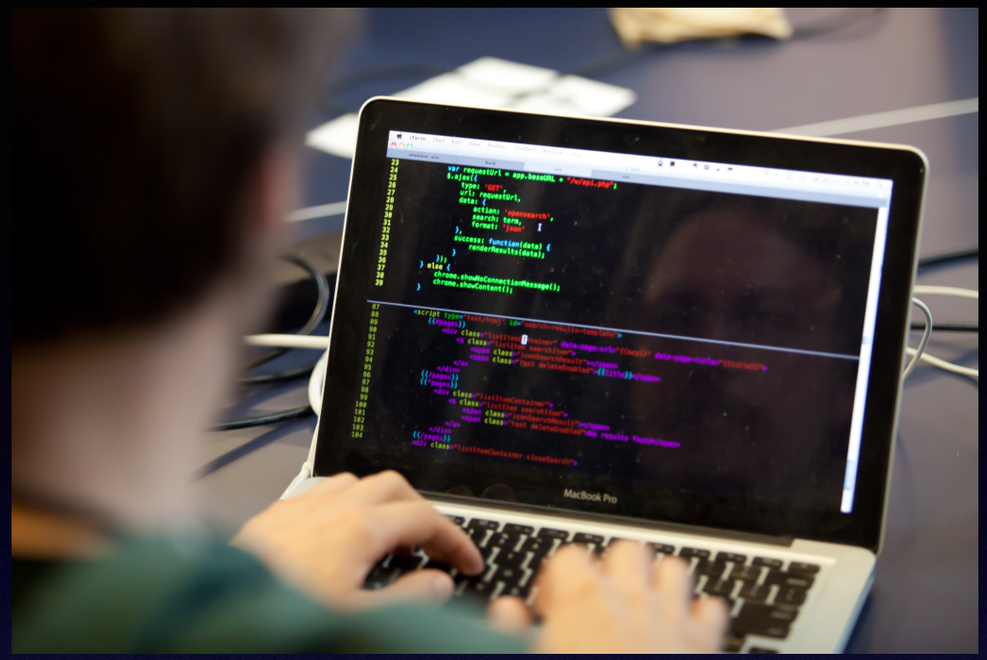
Goal:

- Become familiar with practical, useful implementation techniques

Learning by Doing

What's a compiler?

1.



2. Apply Magic

3.

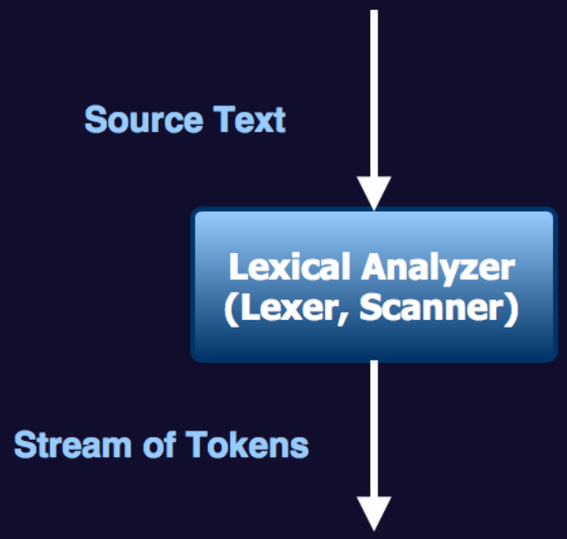
What's a compiler?

What's a compiler?

- Can anyone give a brief definition?
- Tool that translates code written in one programming language to another
- For our purposes:
  - Source: high-level, statically-typed language
  - Target: native executable ("machine code")

Compiler Stages

- Old compilers: monolithic – Fast
- Modern compilers: modular – Maintainable
- Pipeline-like architecture
- Every stage performs some transformation
- Different data structures for each transformation


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

The Lexer

- Strings are tedious to deal with – abstract them away!
- Human languages: unit of information is a **word** and not individual *characters*
- Lexer:
  - Scans every character ("scanner")
  - Matches lexical rules, breaks text into *lexemes*
  - Outputs list of *tokens* ("tokenizer")

Lexemes and Tokens

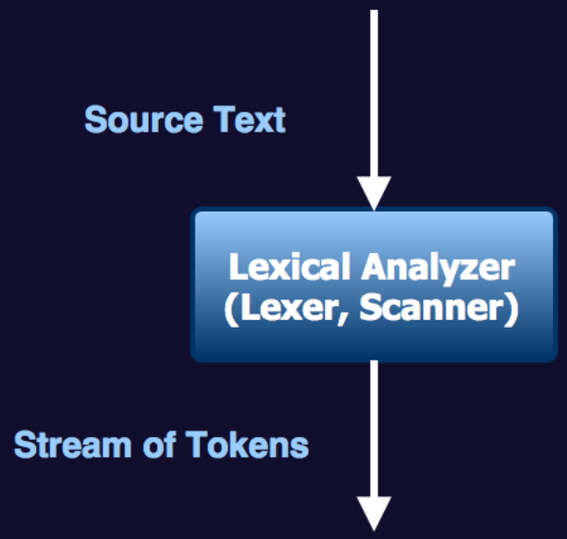
- Lexeme: textual unit of information in the language
  - Equivalent of written or spoken "word"
- Token: data structure describing a lexeme
  - Equivalent of the **meaning** of a word
  - Typically: ***type-value-location*** tuple

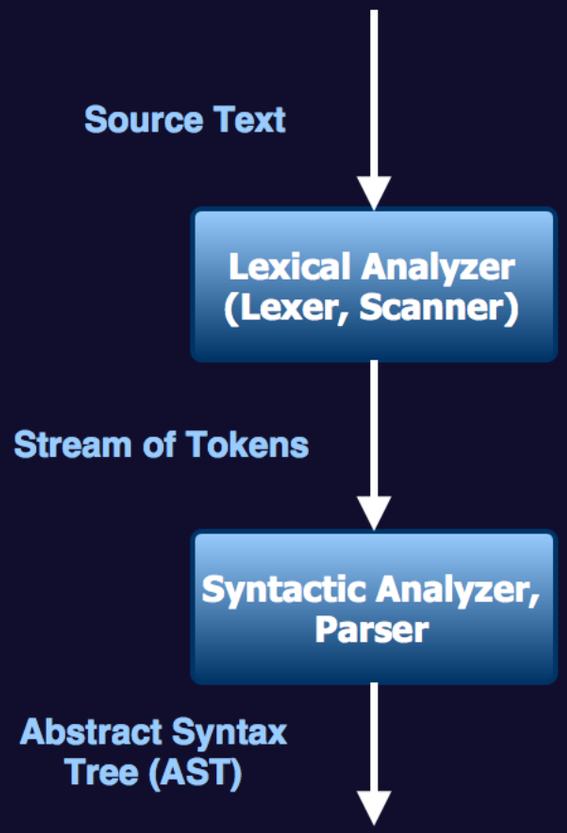
```
struct Token {
    enum Type {
        case Keyword
        case Identifier
        case Number
        case Symbol
    }

    let type: Type
    let value: String
    let location: (Int, Int)
}
```

```
let myNum = 1 - 2
```

```
(Keyword,    "let",    (1, 1))
(Identifier, "myNum",  (1, 5))
(Symbol,     "=",      (1, 11))
(Number,     "1",      (1, 13))
(Symbol,     "-",      (1, 15))
(Number,     "2",      (1, 17))
```

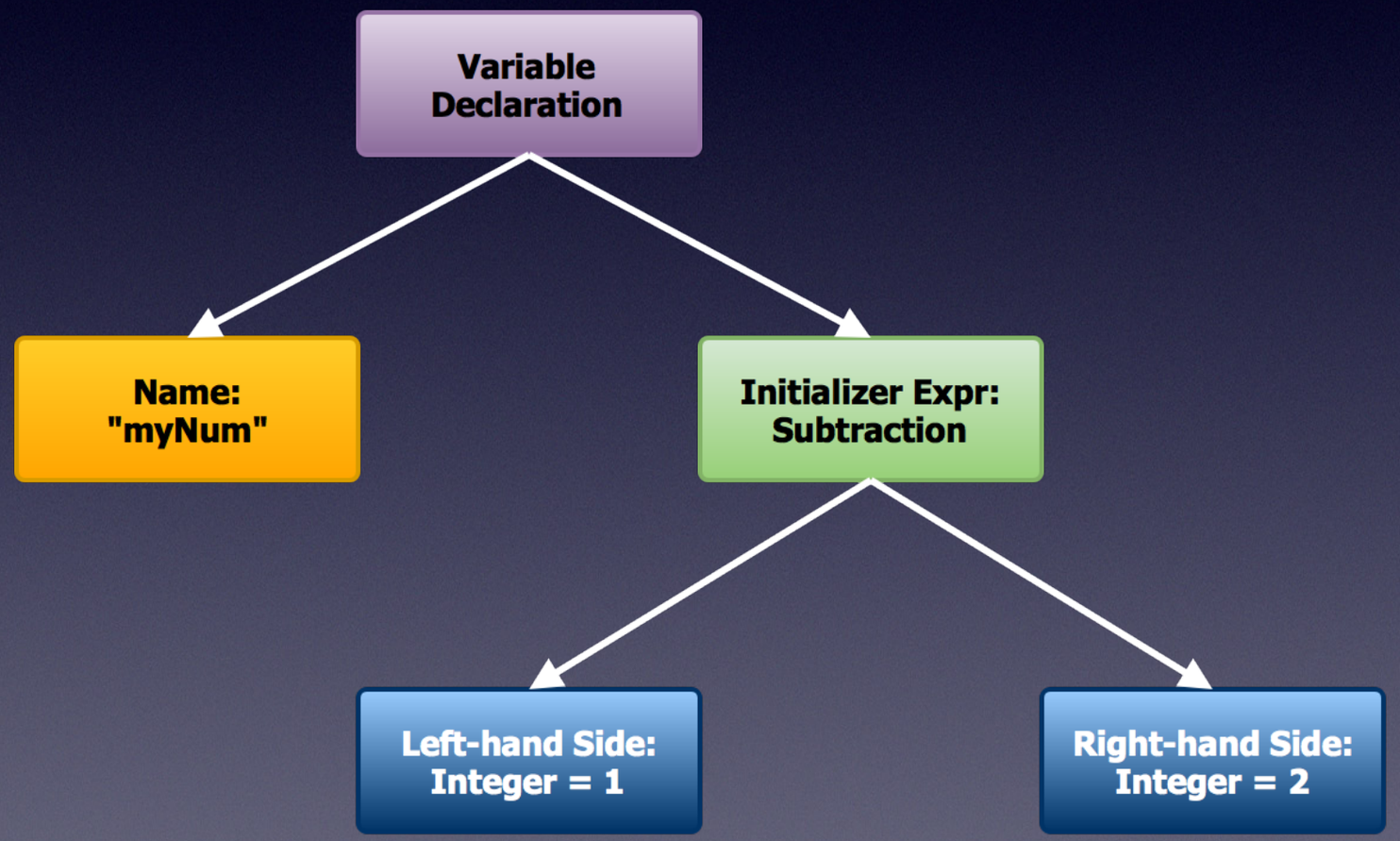

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens

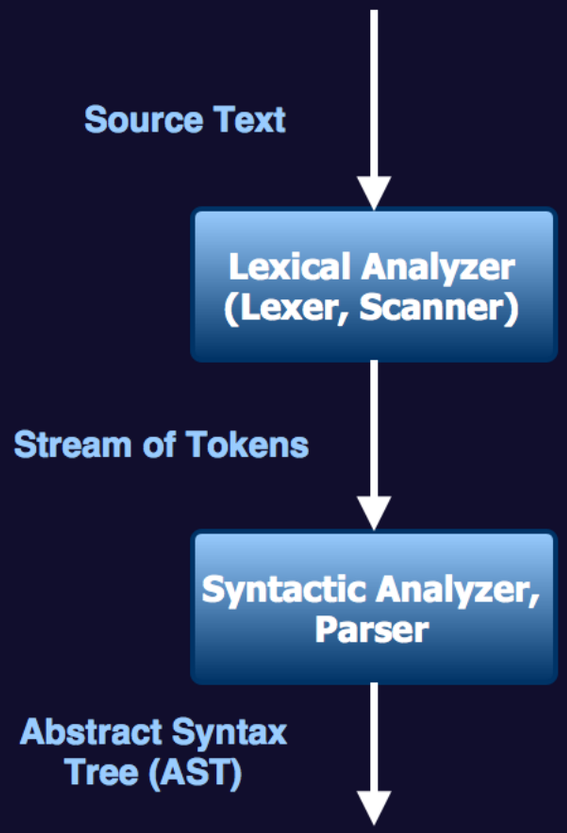

Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)

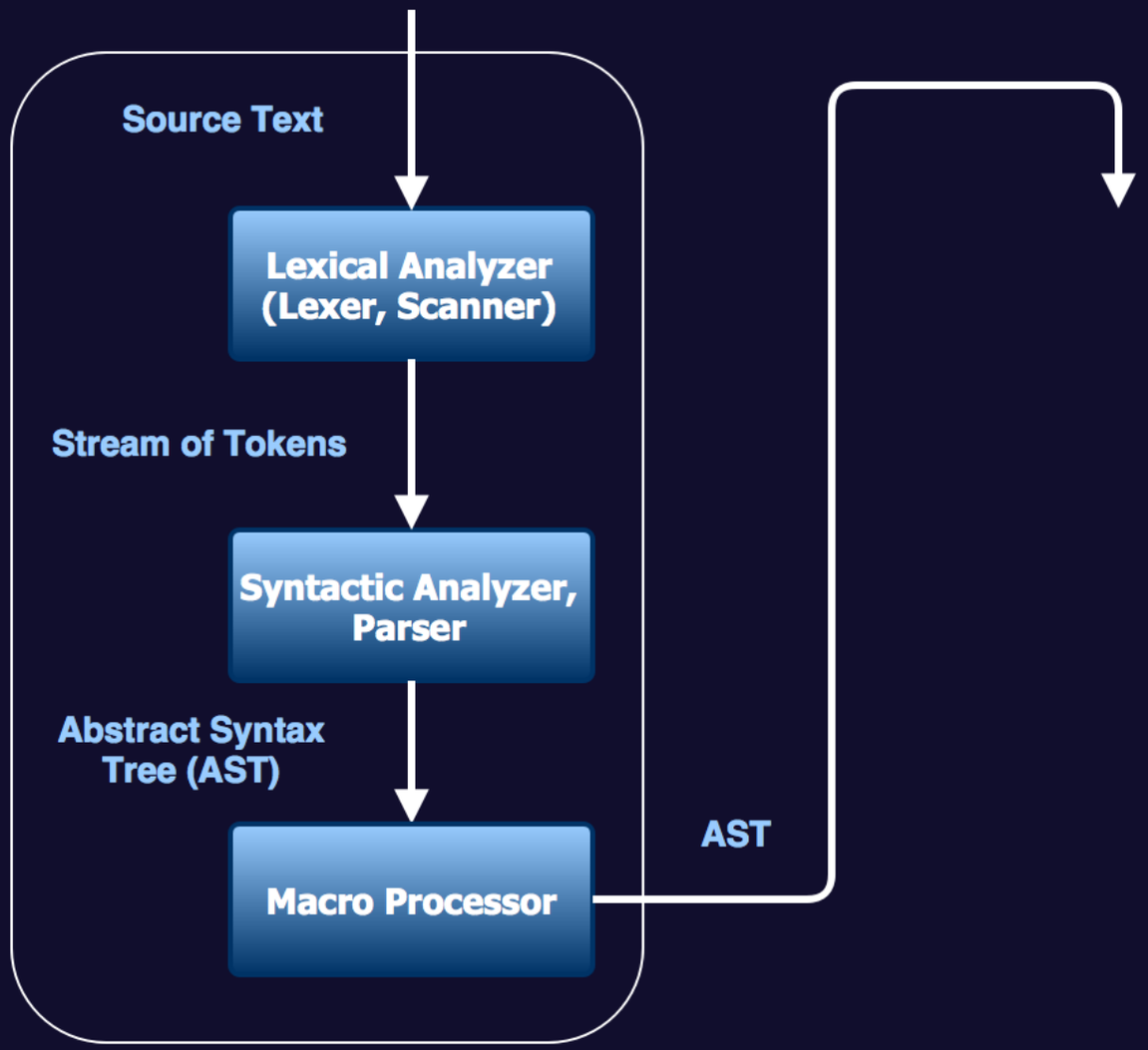
The Parser

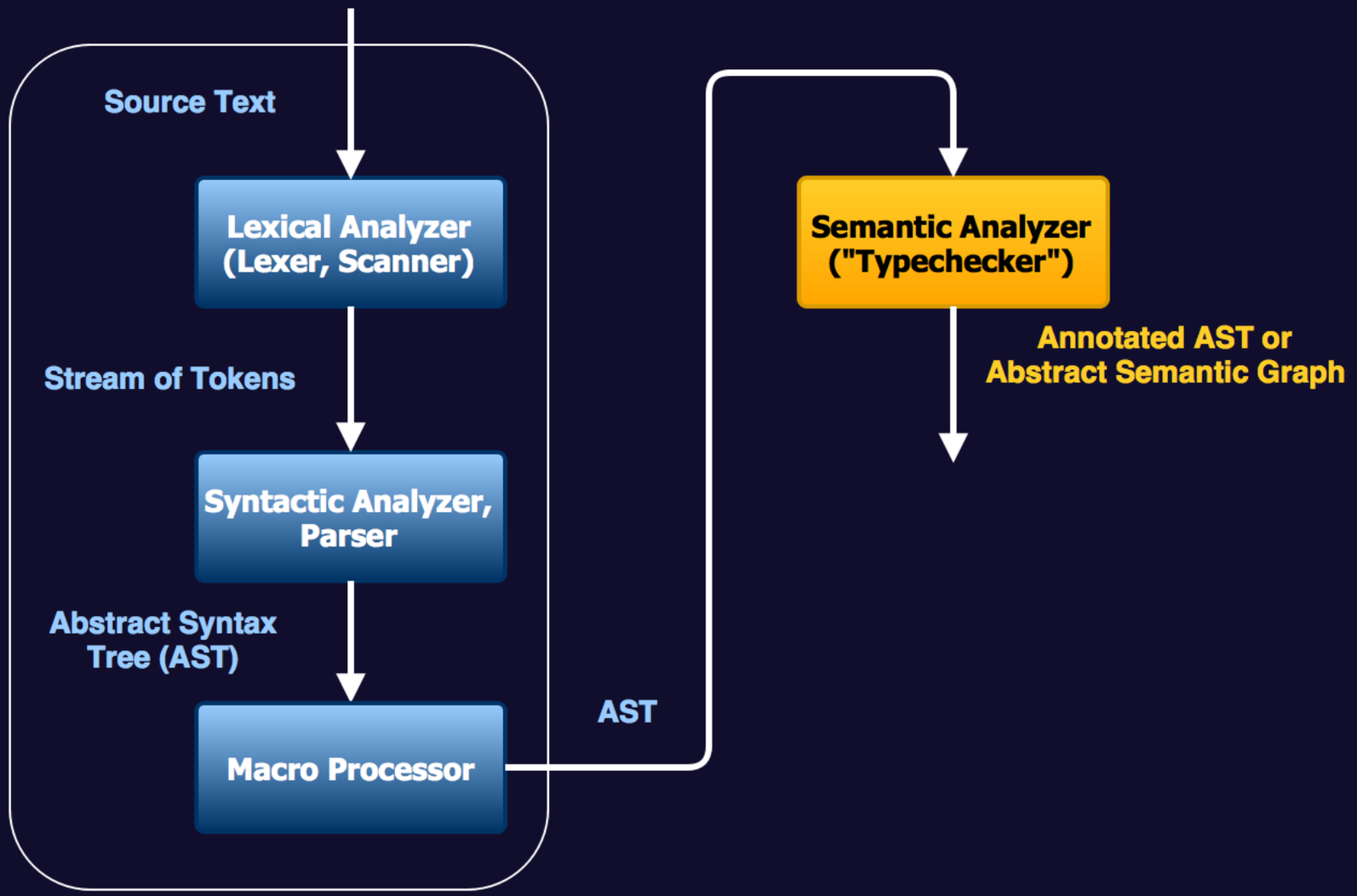
- Matches order of tokens with syntactic rules
- Arranges tokens in a tree called the AST
  - Hierarchical structure
  - Directly represents the source code
  - However, more abstract
    – e.g. parentheses are implicit

```
let myNum = 1 - 2
```


Variable Declaration
Name: "myNum"
Initializer Expr: Subtraction
Left-hand Side: Integer = 1
Right-hand Side: Integer = 2


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

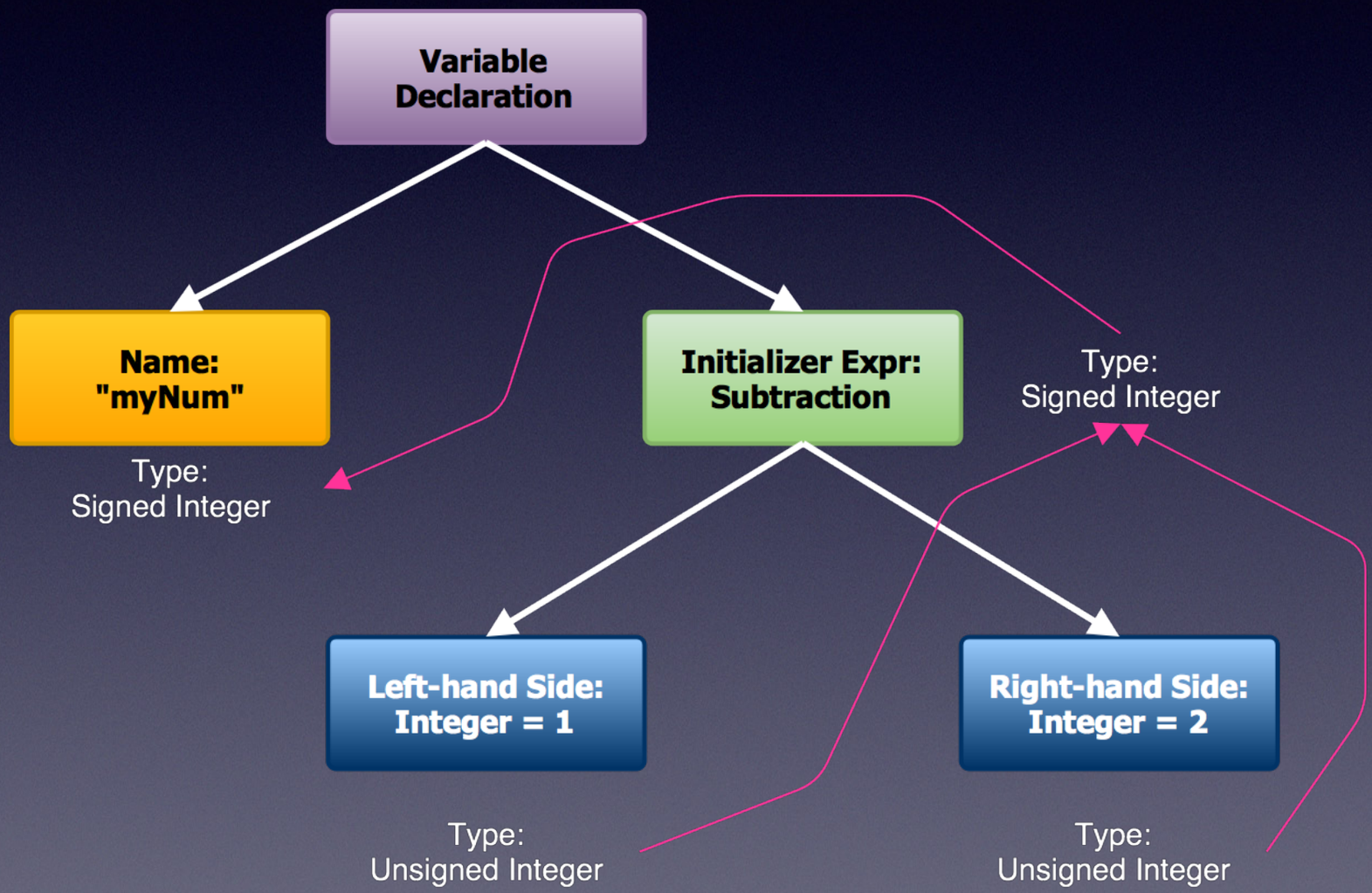

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer,
Parser
Abstract Syntax
Tree (AST)
Macro Processor
AST

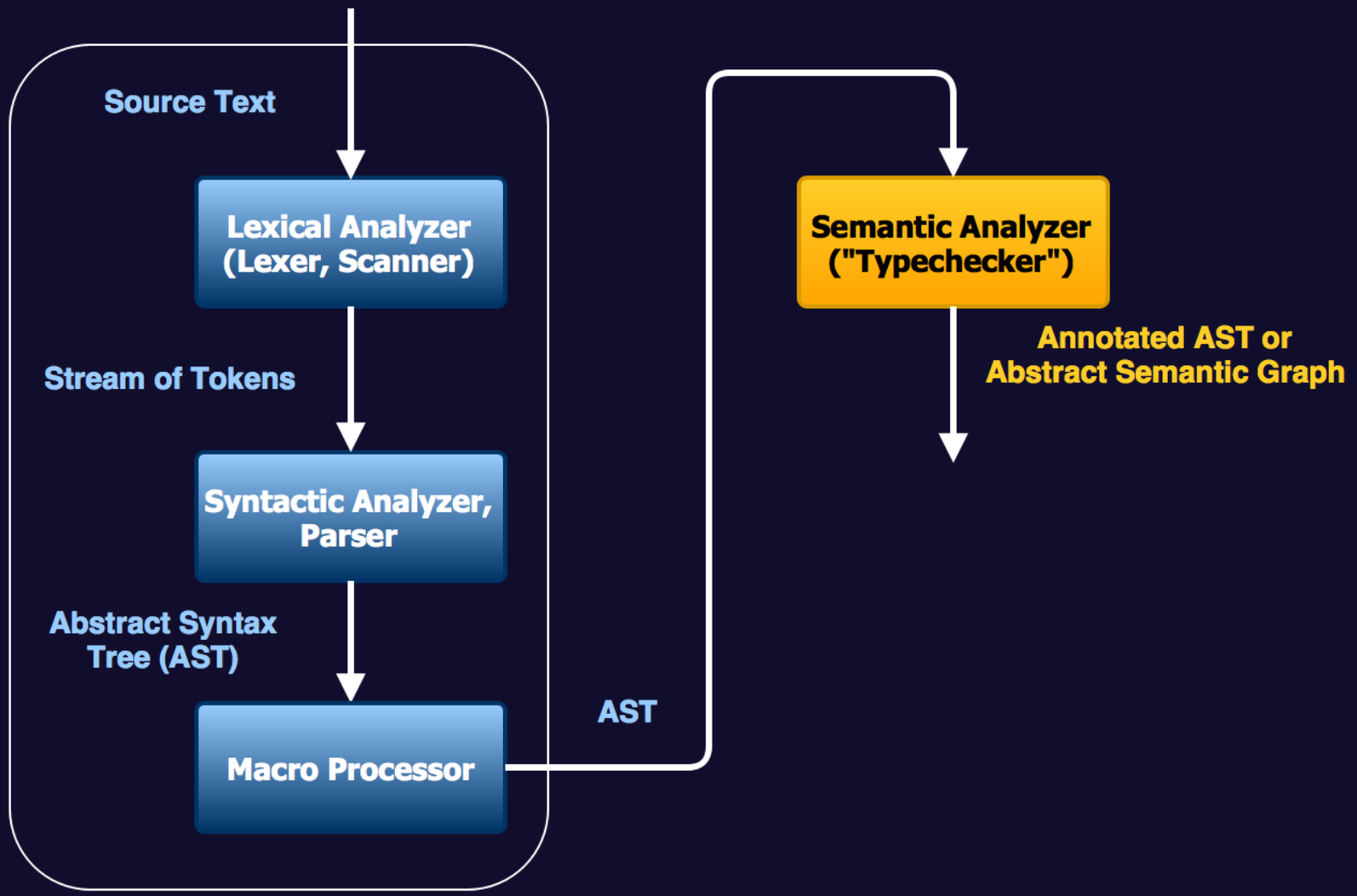

Source Text
Lexical Analyzer
(Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer,
Parser
Abstract Syntax
Tree (AST)
Macro Processor
AST
Semantic Analyzer
("Typechecker")
Annotated AST or
Abstract Semantic Graph

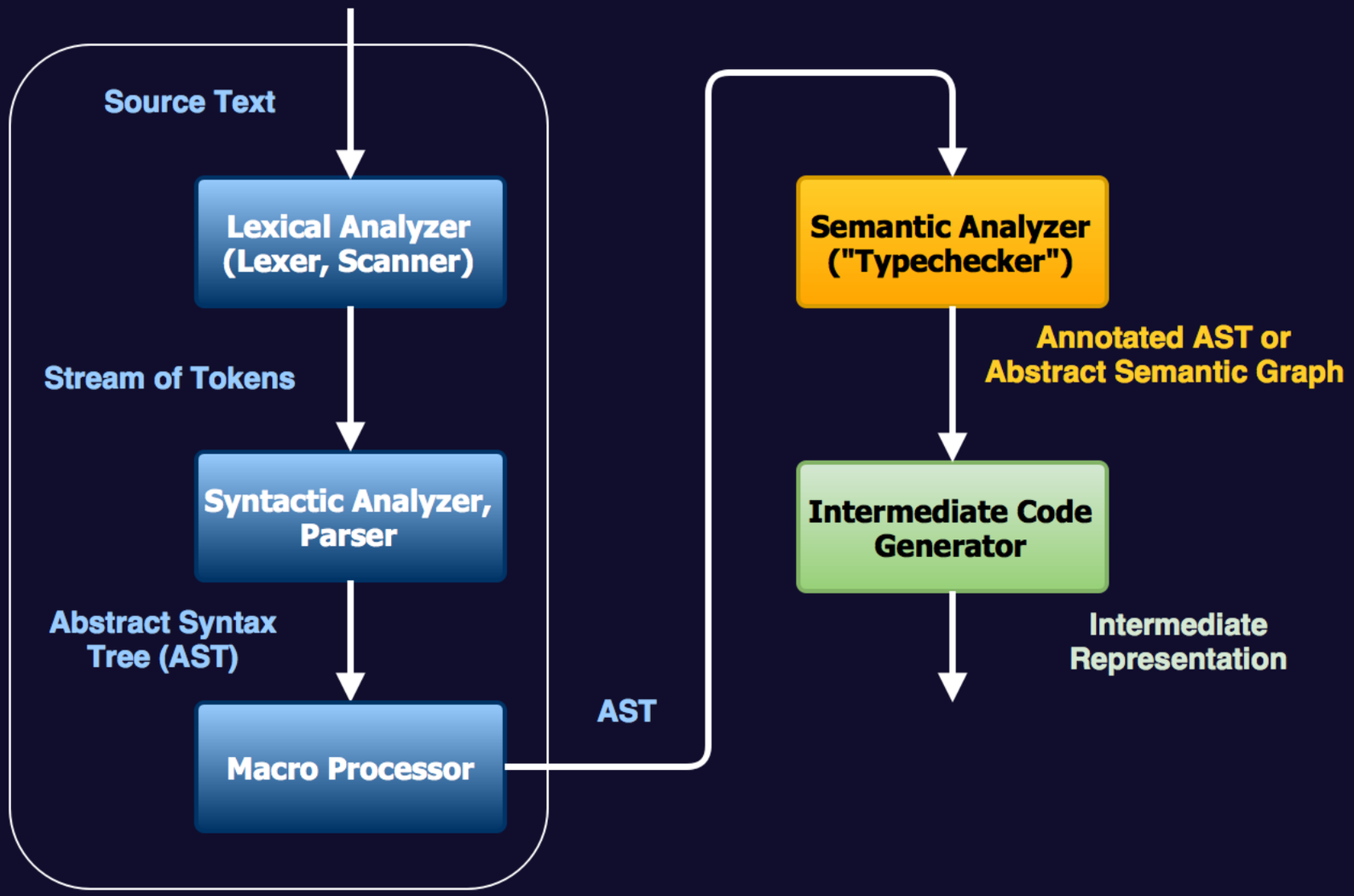
Semantic Analyzer

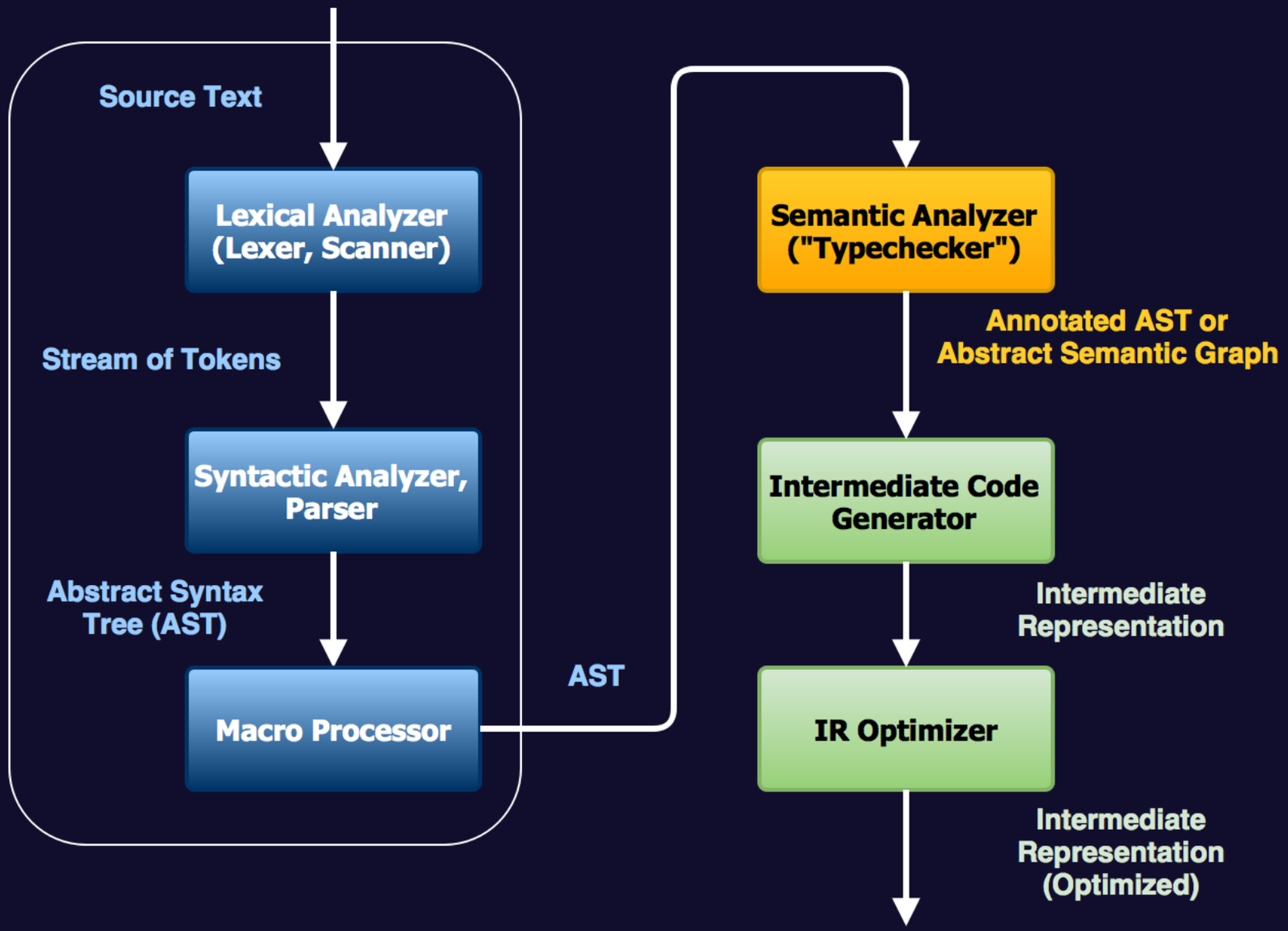
- Walks the AST, examines semantic properties
  - Checks correctness
    - undeclared variables, assignment to constant, etc.
  - Performs type inference, escape analysis, …
  - Annotates AST with e.g. type information

```
let myNum = 1 - 2
```


Variable Declaration
Name: "myNum"
Type: Signed Integer
Initializer Expr: Subtraction
Type: Signed Integer
Left-hand Side: Integer = 1
Type: Unsigned Integer
Right-hand Side: Integer = 2
Type: Unsigned Integer


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

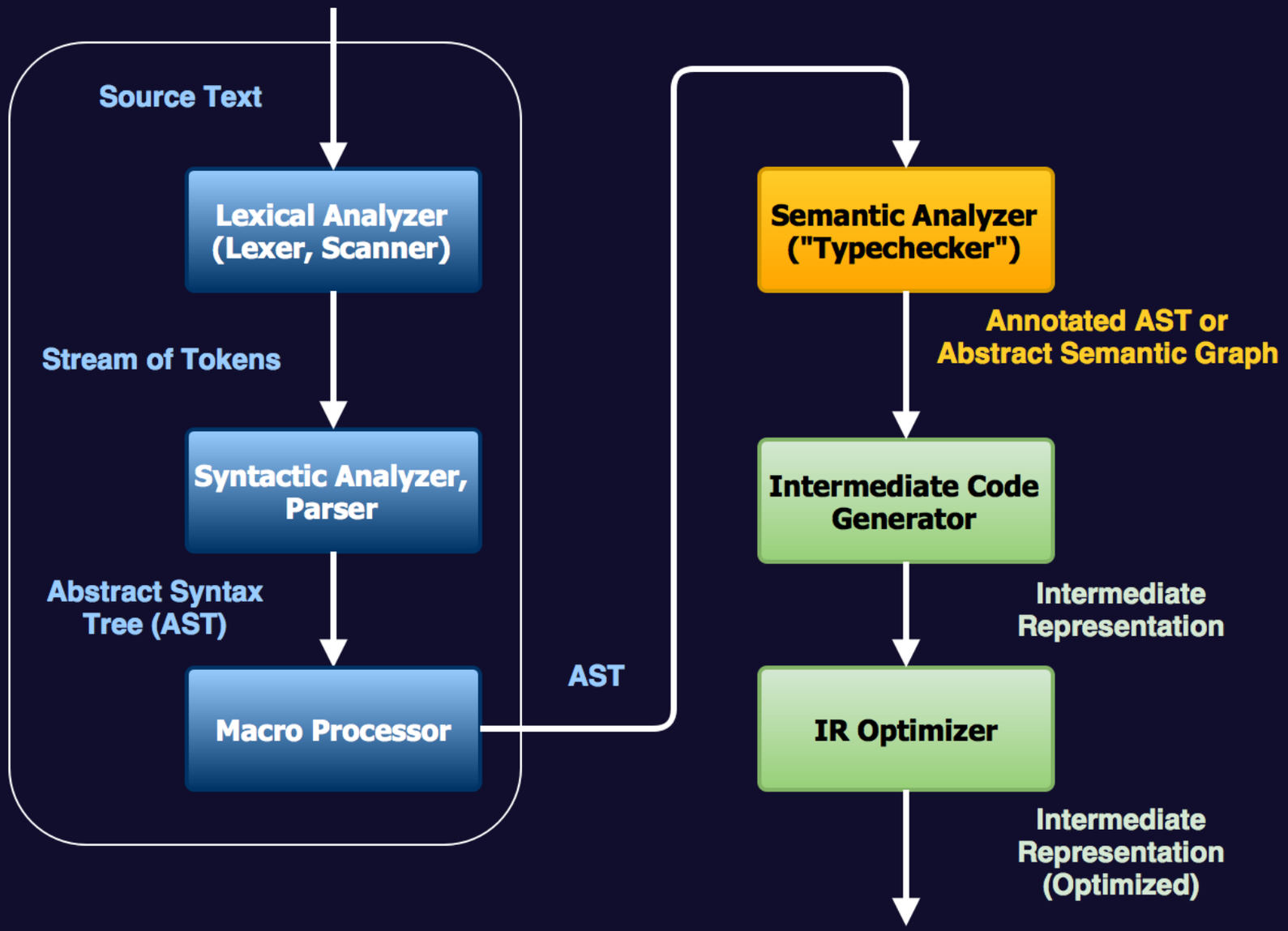

Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation

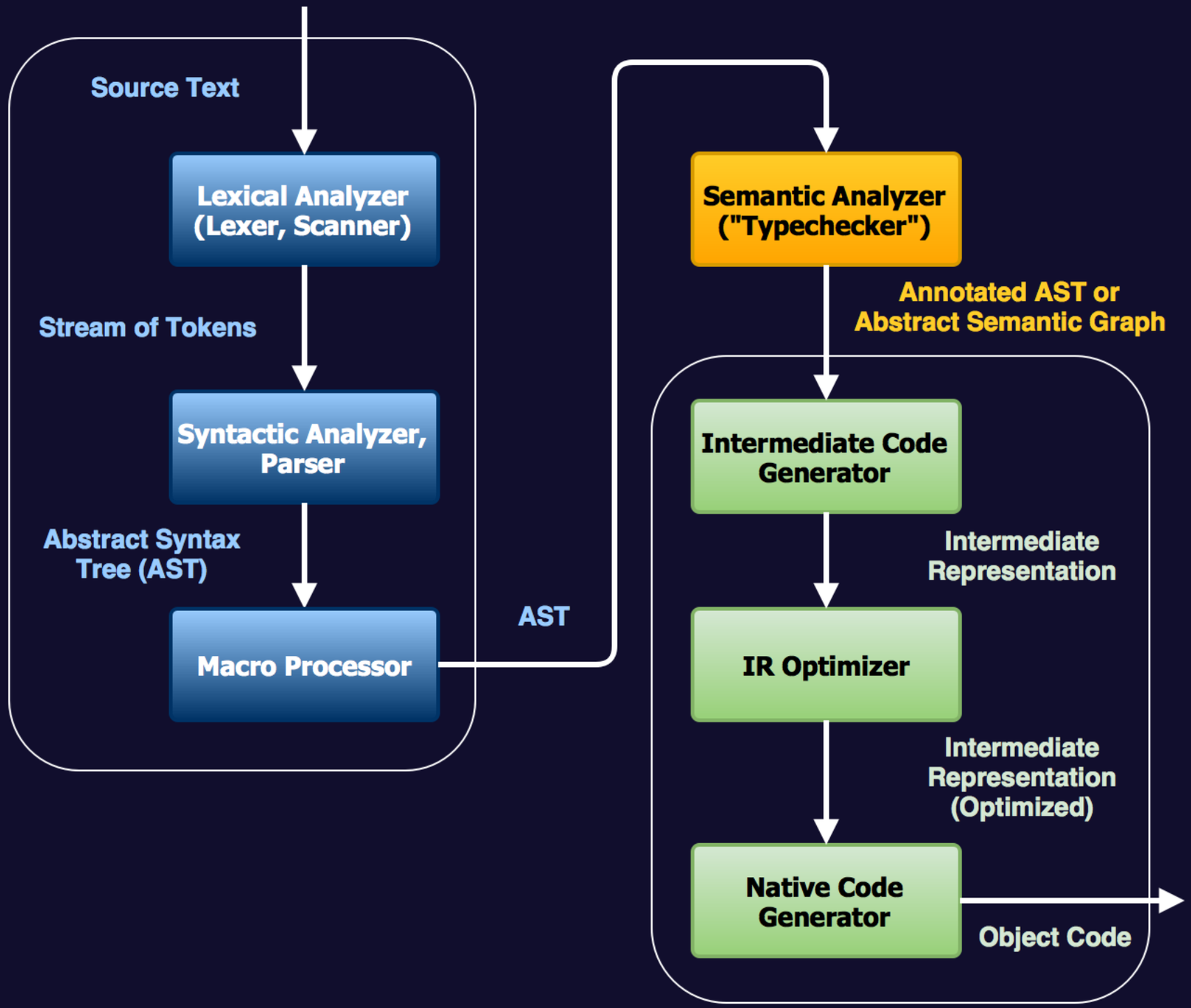

Source Text
Lexical Analyzer (Lexer, Scanner)
Stream of Tokens
Syntactic Analyzer, Parser
Abstract Syntax Tree (AST)
Macro Processor
AST
Semantic Analyzer ("Typechecker")
Annotated AST or Abstract Semantic Graph
Intermediate Code Generator
Intermediate Representation
IR Optimizer
Intermediate Representation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

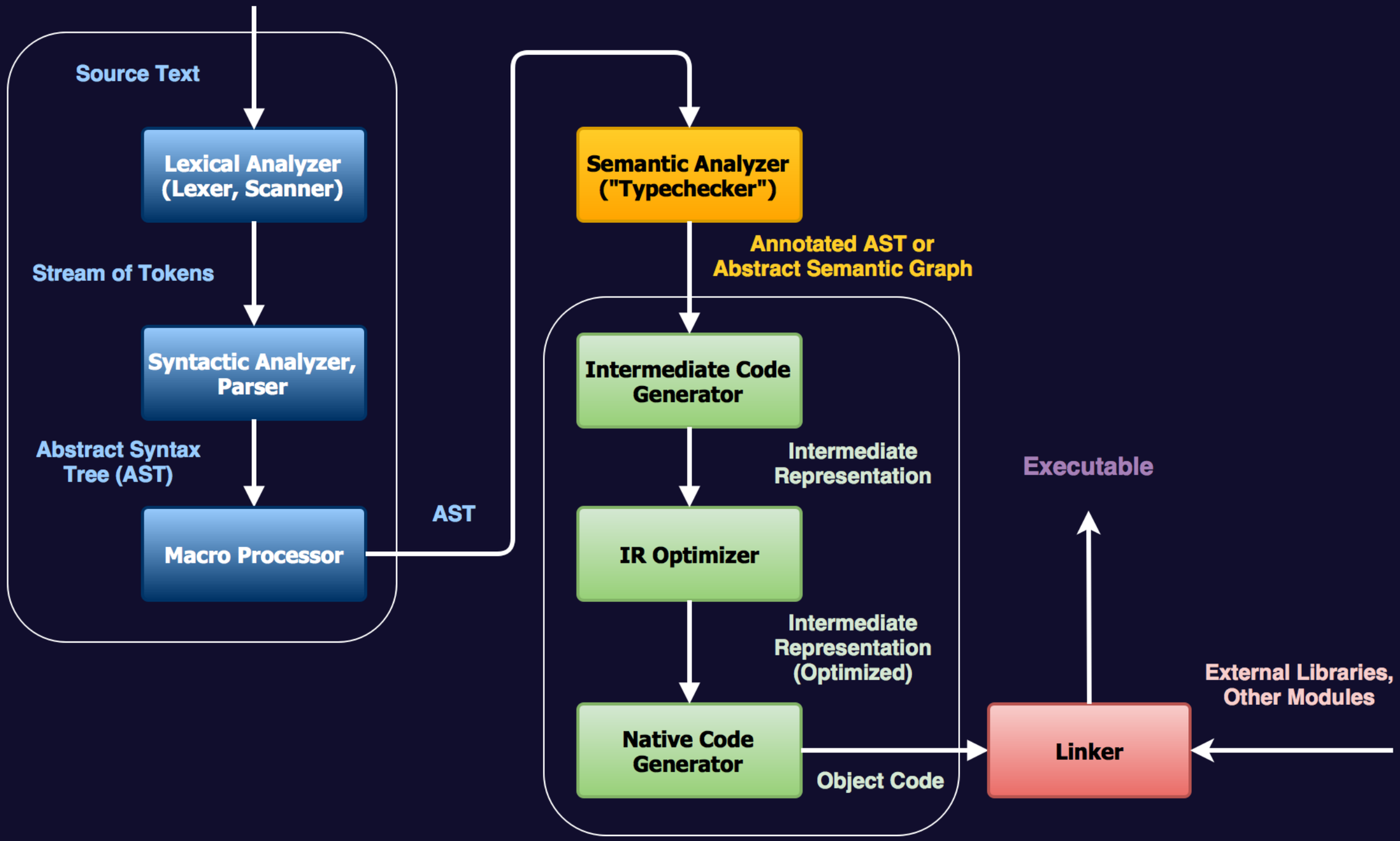
IR Generator & Optimizer

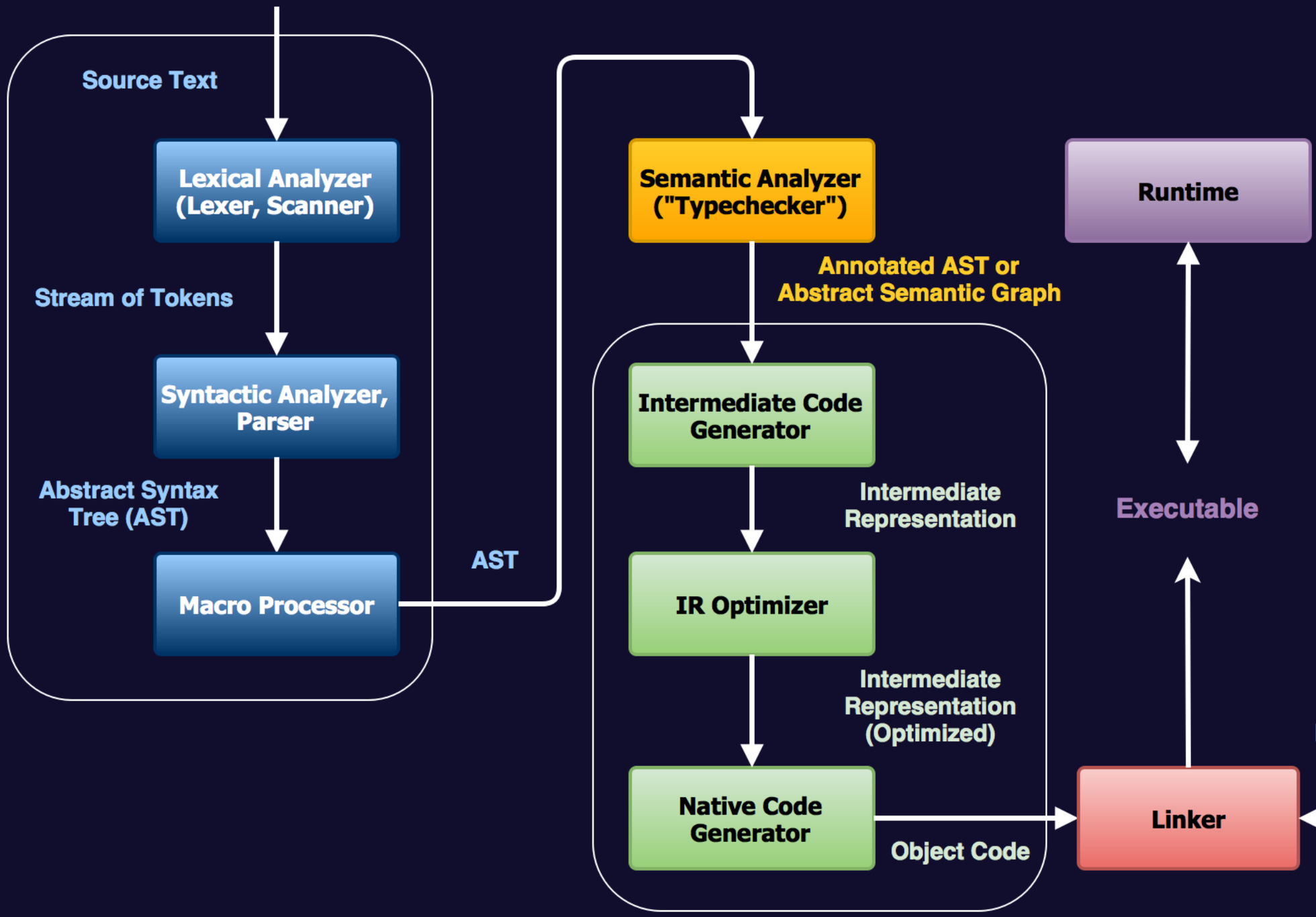
- IR: lower-level than AST or ASG
  - Assembly-like (registers, memory operations, jumps and labels, size and alignment info, …)
  - Better target for many optimizations
  - Functional languages: often lambda calculi
  - Debugging information can be generated here

```
104  %5 = icmp slt i32 %4, 1
105  br i1 %5, label %6, label %7
106
107; <label>:6                                       ; preds = %0
108  call void (i8*, ...)* @spn_die(i8* getelementptr inbounds ([27
109  unreachable
110
111; <label>:7                                       ; preds = %0
112  %8 = load i32* %2, align 4
113  %9 = load i8*** %3, align 8
114  %10 = call i32 @process_args(i32 %8, i8** %9, i32* %pos)
115  store i32 %10, i32* %args, align 4
116  %11 = load i32* %args, align 4
117  %12 = and i32 %11, 255
118  switch i32 %12, label %85 [
```


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

Lexing Algorithm

1. Determine type and boundaries of next lexeme
    - Examine first N characters (bounded lookahead)
2. Create a token representing the lexeme
3. Add token to output list / stream / array
4. Skip lexeme: update cursor / pointer

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

IT'S HOPELESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

Say No To RegEx!

Regular Expressions

- Lexer Generator tools use them (`lex`, `flex`, ...)
- Insufficiently powerful

  http://j.mp/regex-comment

- Hard to read, easy to get wrong

  http://j.mp/regex-error

Lexer Generators

- Regex-centric fixed mindset
  - Customization == lots of ugly hacking
- `lex, flex`: No Unicode support
- Others: Additional dependency
  (not part of Unix-like systems by default)

Hand-coding

- More powerful, more general
- Easier to customize special cases
- More verbose sometimes
- Intent is generally clearer
- Can still use regex for lexing simpler tokens (e.g. integers)

Let's Get to Work!