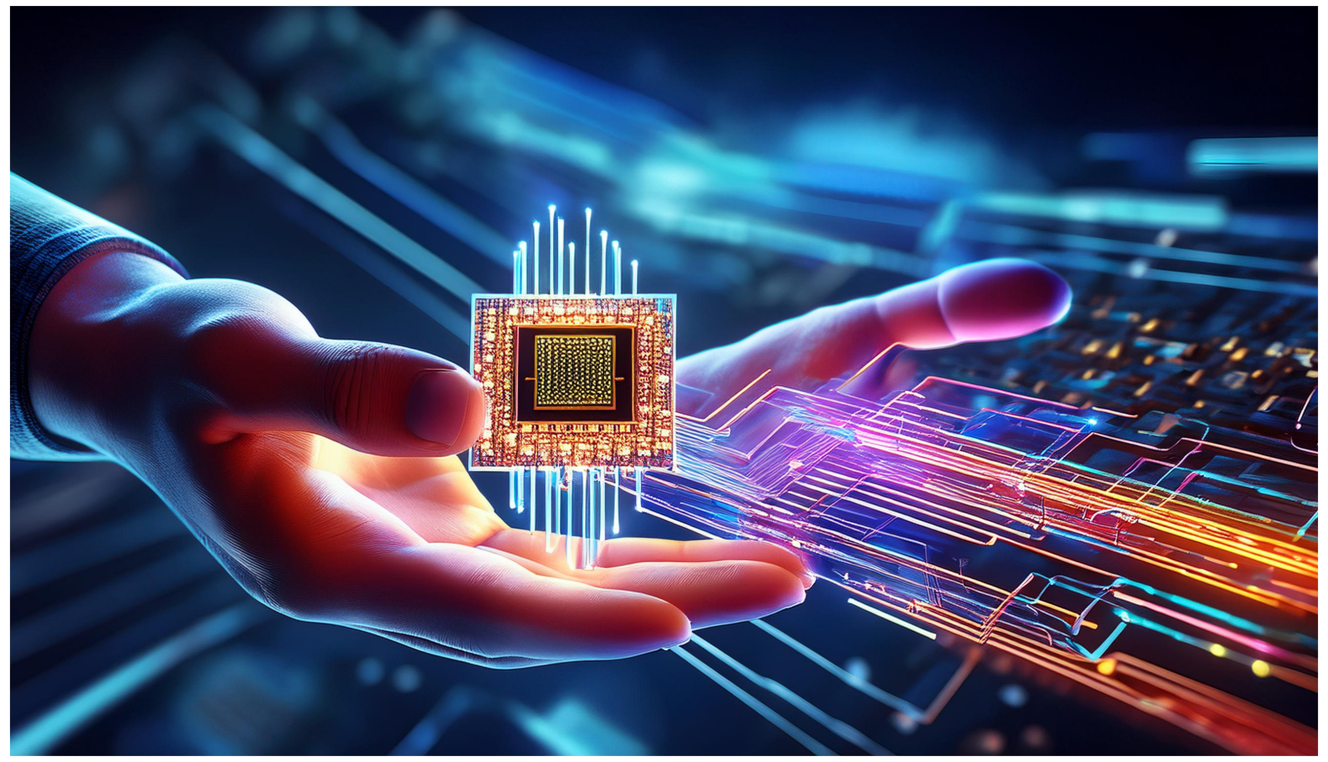
22.11.24 17:58

CSCI 250
Introduction to Computer Organisation
Lecture 5: Compiler Basics I



Jetic Gū

2024 Fall Semester (S3)

Overview

- Architecture: von Neumann
- Textbook: CO: 4.5
- Core Ideas:
  1. What is a compiler?
  2. Compiler Stages

1. Loosely based on Prof. Anoop Sarkar's slides

Introduction to Compilers

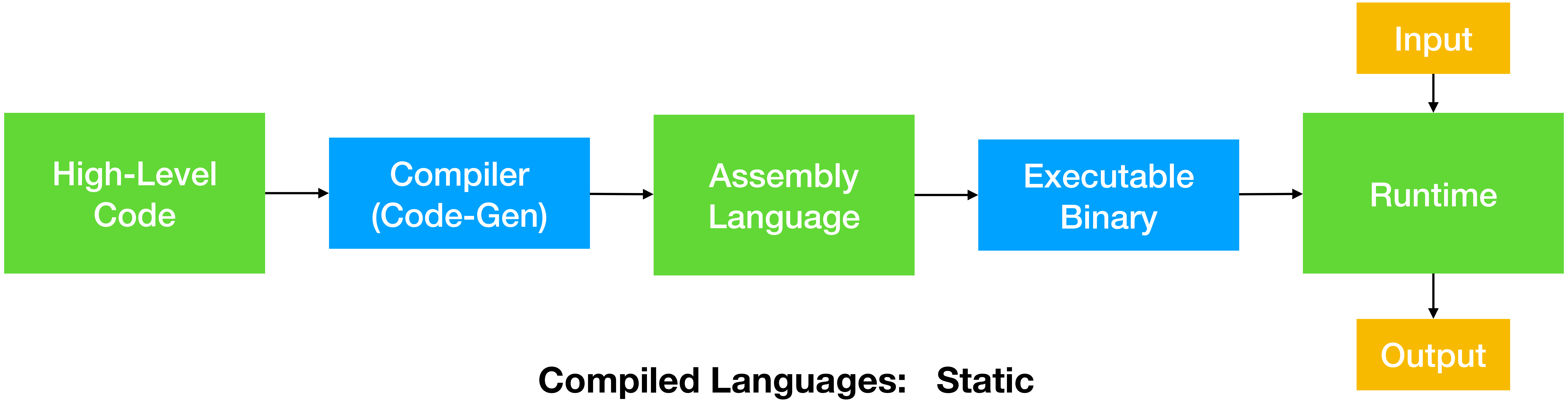
What Languages use Compilers?

- C, C++, C#, Swift, Compiler, etc.
- Interpreted Languages
  - On-the-fly interpretation
  - Bash (Shell script)
  - Python, Javascript, etc.

Review

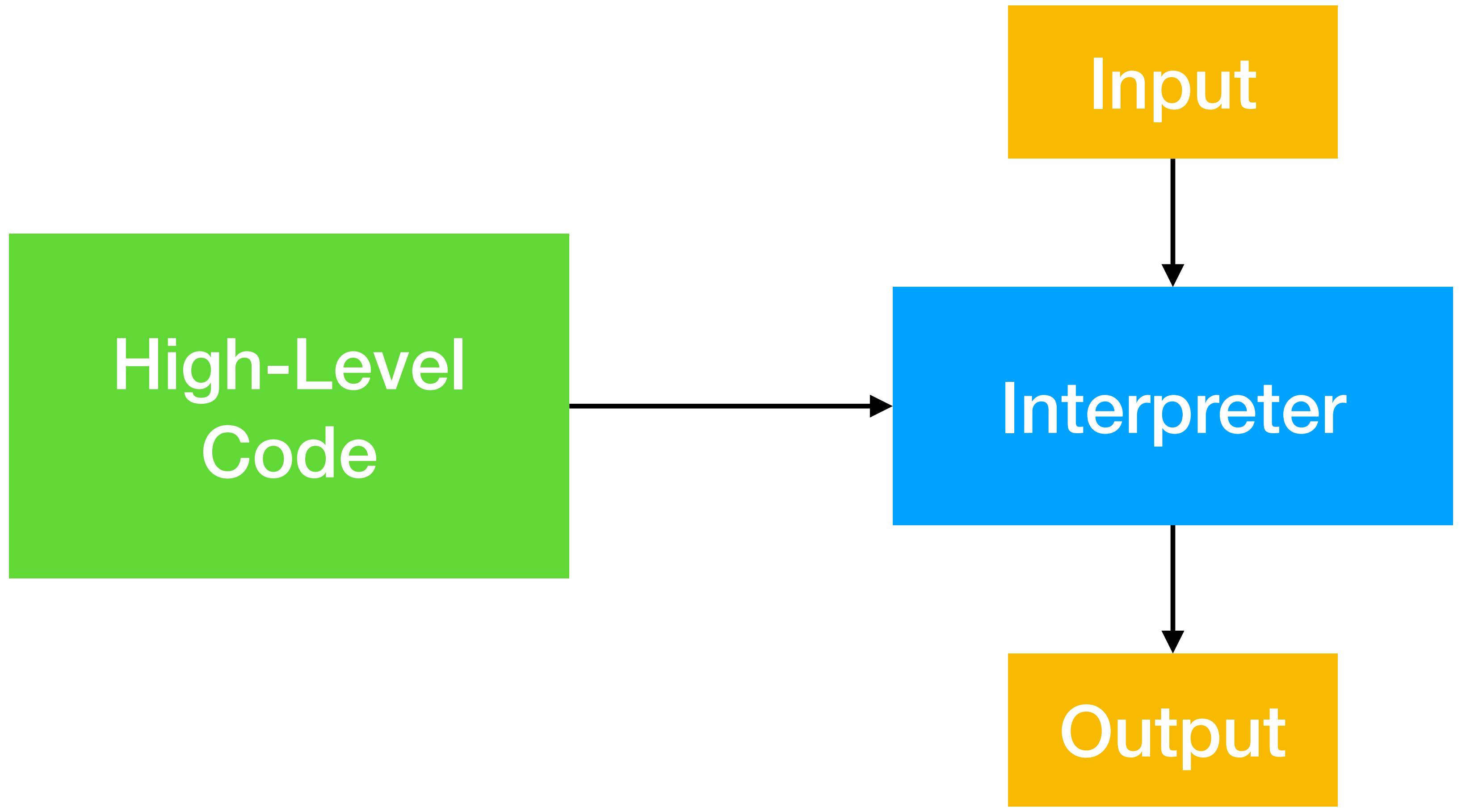
P1
Compiler Intro

General Pipeline



Compiled Languages: Static

Interpreted Languages: Dynamic



1. Blue: Programme; Green: Code, either in text or binary; Orange: IO

Technical

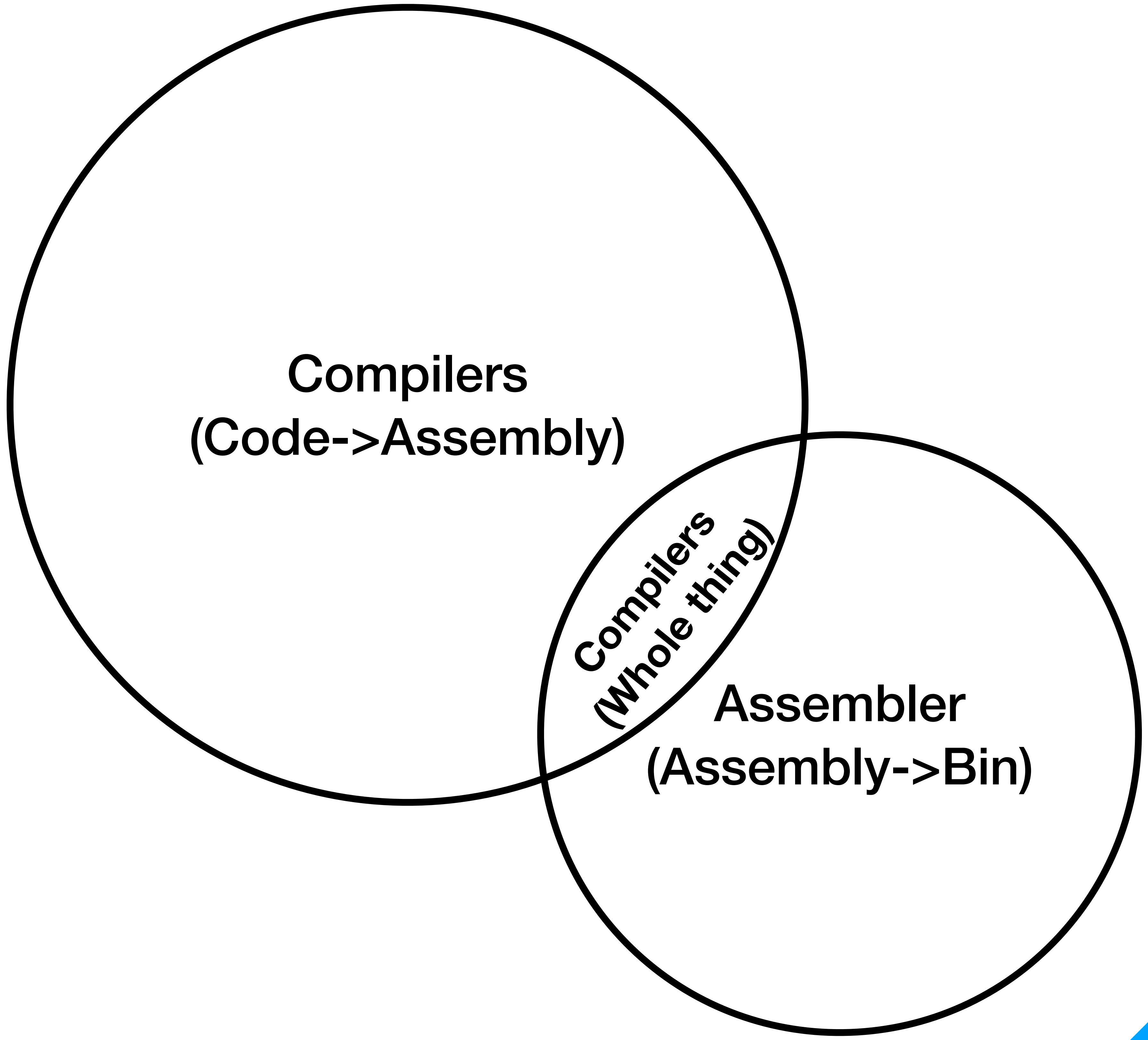
Building a Compiler

- Compilers are computer programmes
- Each programming language compiler could be built using a compiler-compiler
  - yacc: yet another compiler compiler
  - bison: version of yacc from the GNU project (GNU is short for **GNU is Not Unix**)
- **Compiler**: take high-level programming language code as input, generate assembly code or binary code (through assembler)
  - Assemblers sometimes are shared between multiple compilers, and sometimes treated as part of the compiler. Is this technically correct?
- **Assembler**: architecture-specific, translates assembly code to binary machine code

Concept

Compiler & Assembler

- Context Matters!
- Assembler
  - No ambiguity: assembly code goes in, machine code goes out
- Compiler
  - Sometimes people include the assembler as part of the compiler (as a whole product), sometimes not (as a separable component)



Technical

P1
Compiler Intro

Compilers (Code -> Assembly)

- Analyse the source code
  - Syntactic analysis
  - Translation of higher-level language to low-level assembly
- What is assembly?
  - Pretty much equivalent to machine code, but still readable text
  - One line of Assembly equals to usually 1 or 2 lines of machine code (binary)
  - Direct translation, line-by-line

Concept

P1
Compiler Intro

What can be shared?

- Programming Languages have a lot in common
- e.g. C/C++/Obj-c/C#: very very similar
  - C++ began as a *fork* of C
- Compilation targets (what compilers generate)
  - Assembly Code, specific to the instruction set of your target machine
  - e.g. x64, ARM64, etc.

Concept

P1
Compiler Intro

General Challenges

- Instruction Pipelining (software solutions to pipeline hazards)
  - reordering; branch prediction
- Parallel algorithms
- Memory Management
- Architectural differences
- Hardware synthesis (FPGA stuff)

Concept

Requirements for Compiler

- Cost of Compiling and execution should be optimised
- No program that violates the definition of the language should compile successfully
  - Error detection and reporting
- No program that is valid should fail to compile

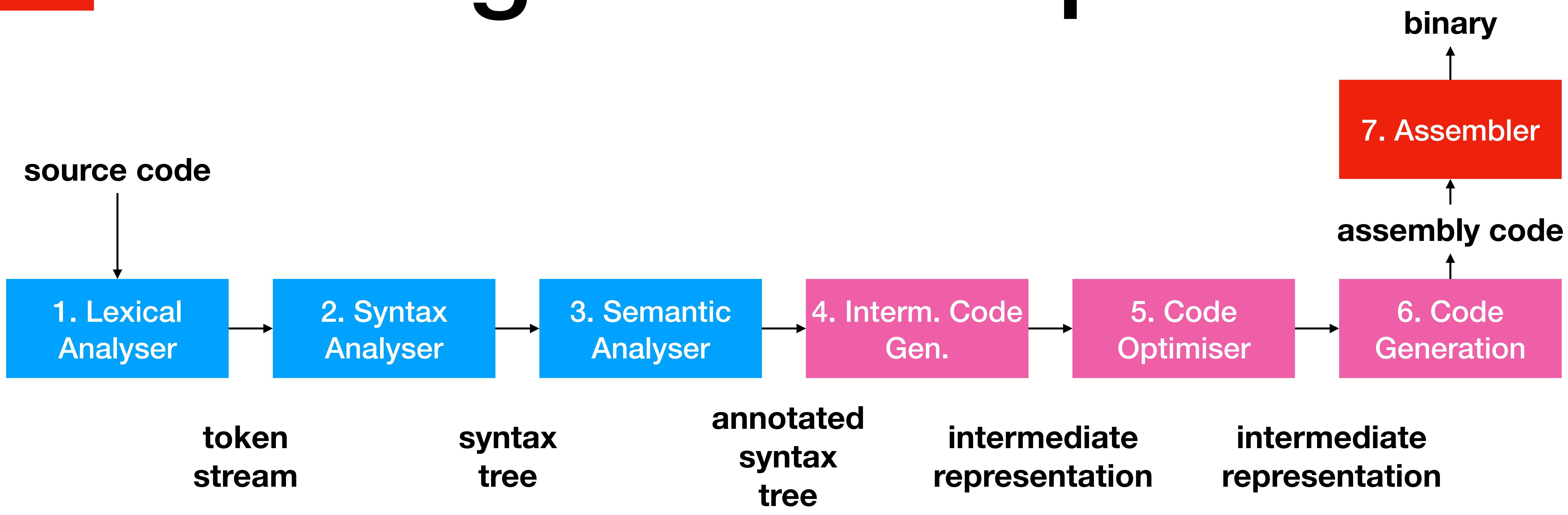
Stages of a Compiler

Stages of Compiler

- Analysis (Front-end)
  1. Lexical analysis
  2. Syntax analysis (parsing)
  3. Semantic analysis (type-checking)
- Synthesis (Back-end)
  4. Intermediate code generation
  5. Code optimisation
  6. Code generation

7. Assembler
   - Pretty much just a for `loop`, and a bunch of `if` conditions

Stages of Compiler



Technical

P2
Compiler Stages

1. Lexical Analysis

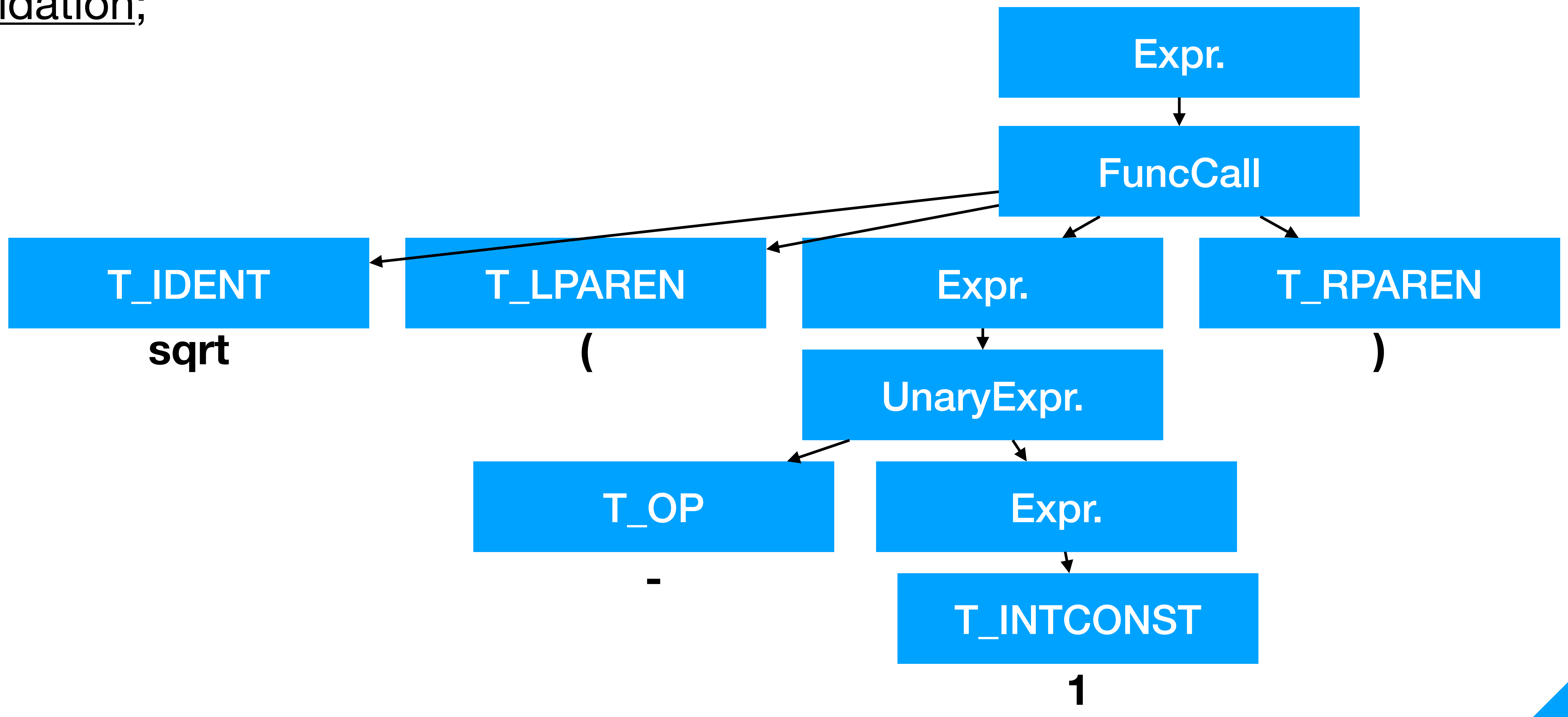
- Also called **scanning**, takes the input programme as a *string*, and convert into *tokens*
- e.g.: `double f = sqrt(-1);`

double	f	=	sqrt	(	-	1	)	;
T_DOUBLE	T_IDENT	T_OP	T_IDENT	T_LPAREN	T_OP	T_INTCONST	T_RPAREN	T_SEP

Technical

2. Syntax Analysis

- Also called **parsing**
  - use a grammar to convert the tokens into a structural representation (parse tree); and
  - perform structural validation;
- e.g.: `sqrt(-1)`



Technical

P2
Compiler Stages

2. Syntax Analysis (Abstract Syntax Tree)

- e.g.: `sqrt(-1)`

```
MethodCall (
   sqrt,
   UnaryExpr( UnaryMinus,
              Number(1),
   )
)
```

Technical

P2
Compiler Stages

3. Semantic Analysis

- "does it make sense"? Checking semantic rules,
    - A function call to `sqrt` is mentioned. Has this function been declared?
    - Are the variables declared? Is the `return` expression there?
    - Are the operands type-compatible? (e.g. `int` -> `float` in arithmetics)
    - Do function arguments' type (e.g. `-1`, 1 argument) match function declaration? (e.g. for `sqrt`, you need 1 argument)
- Type Checking
- Static vs runtime semantic checks
    - Array bounds, return values' types matching function definition, etc.

Technical

4. Intermediate Code Generation

- Step 1-3: Src -> AST

```
extern void print_int(int);

class C {
  bool foo() { return(true); }
  int main() {
    if (foo()) {
      print_int(1); }
    }
}
```

Source Code

```
Program(
    ExternFunction(print_int,VoidType,VarDef(IntType)),
    Class(  C,
            None,
            Method(  foo,
                     BoolType,
                     None,
                     MethodBlock(      None,
                                       ReturnStmt(BoolExpr(True)))),
            Method(  main,
                     IntType,
                     None,
                     MethodBlock(
                              None,
                              IfStmt(  MethodCall(foo,None),
                              Block(   None,
                                       MethodCall(print_int,Number(1))),
                              None)))))
```

AST

Technical

4. Intermediate Code Generation / 5. Optimisation

- Step 4-5: AST -> Intermediate Representation

```
Program(
   ExternFunction(print_int,VoidType,VarDef(IntType)),
   Class(  C,
         None,
         Method(  foo,
                  BoolType,
                  None,
                  MethodBlock(        None,
                                      ReturnStmt(BoolExpr(True)))),
         Method(  main,
                  IntType,
                  None,
                  MethodBlock(
                            None,
                            IfStmt(   MethodCall(foo,None),
                            Block(    None,
                                      MethodCall(print_int,Number(1))),
                            None)))))
```

AST

```
; ModuleID = 'C'

declare void
@print_int(i32)

define i1 @foo() {
entry:
  ret i1 true
}
```

```
define i32 @main() {
entry:
  br label %ifstart
ifstart:
%calltmp = call i1 @foo()
  br i1 %calltmp, label %iftrue, label %end
iftrue:
call void @print_int(i32 1)
  br label %end
end:
  ret i32 0
}
```

Technical

P2
Compiler Stages

6. Code Generation

- Intermediate Representation -> Assembly

```
; ModuleID = 'C'

declare void
@print_int(i32)

define i1 @foo() {
entry:
  ret i1 true
}


define i32 @main() {
entry:
  br label %ifstart
ifstart:
%calltmp = call i1 @foo()
  br i1 %calltmp, label %iftrue, label %end
iftrue:
call void @print_int(i32 1)
  br label %end
end:
  ret i32 0
}
```

```
          .section
          __TEXT,__text,regular,pure_i
nstructions
          .globl    _foo
          .align    4, 0x90
@foo
          .cfi_startproc
%entry
          mov       al, 1
          ret
          .cfi_endproc

          .globl    _main
          .align    4, 0x90
```

```
@main
          .cfi_startproc
%entry
          push      rax
Ltmp0:
          .cfi_def_cfa_offset 16
          call      _foo
          test      al, 1
          je        LBB1_2
%iftrue
          mov       edi, 1
          call      _print_int
%end
          xor       eax, eax
          pop       rdx
          ret
          .cfi_endproc
```

x86 assembly

Technical

P2
Compiler Stages

7. Assembling

- Assembly -> Bin

```
        .section
        __TEXT,__text,regular,pure_i
nstructions
        .globl    _foo
        .align    4, 0x90
@foo
        .cfi_startproc
%entry
        mov       al, 1
        ret
        .cfi_endproc

        .globl    _main
        .align    4, 0x90
```

x86 assembly

```
@main
        .cfi_startproc
%entry
        push      rax
Ltmp0:
        .cfi_def_cfa_offset 16
        call      _foo
        test      al, 1
        je        LBB1_2
%iftrue
        mov       edi, 1
        call      _print_int
%end
        xor       eax, eax
        pop       rdx
        ret
        .cfi_endproc
```

0101001010101010101010100100010
1010101101010101010101010101001
0100101010010100101001001010010
0101001010010101001010010010100
1101101011111101010101011010 1
1110101011111011001100100100 10
1001001111101010010100111001 00
1010010100101001001010010101 0
00101001010......

Technical