

# Overview of Compilation

Readings: EAC2 Chapter 1



EECS4302 A:  
Compilers and Interpreters  
Summer 2025

CHEN-WEI WANG

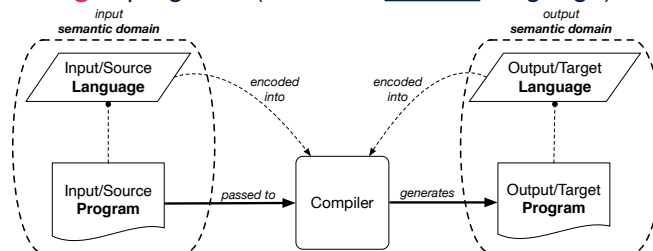
## What is a Compiler? (2)

- The idea about a **compiler** is extremely powerful:  
You can turn anything to anything else,  
as long as the following are **clear** about these two things:
  - SYNTAX [ **specifiable** as CFGs ]
  - SEMANTICS [ **programmable** as mapping functions ]
- Mental Exercise.** Let's consider an A+ challenge.
- A compiler should be constructed with good **SE principles**.
  - Modularity** [ interacting components ]
  - Information Hiding** [ hiding unstable, revealing stable ]
  - Single Choice Principle** [ a change only causing minimum impact ]
  - Design Patterns** [ polymorphism & dynamic binding ]
  - Regression Testing** [ e.g., unit-level, acceptance-level ]

3 of 20

## What is a Compiler? (1)

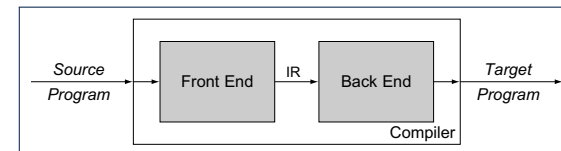
A software system that **automatically translates/transforms** **input/source** programs (written in one language) to **output/target** programs (written in another language).



- Semantic Domain**: Context with its own vocabulary & meanings  
e.g., OO (EECS1022/2030/2011), database (3421), predicates (1090)
- Source** and **target** may be in **different semantic domains**.  
e.g., Java programs to SQL relational database schemas/queries  
e.g., C procedural programs to MISP assembly instructions

2 of 20

## Compiler: Typical Infrastructure (1)



- FRONT END**:
  - Encodes: knowledge of the **source** language
  - Transforms: from the **source** to some **IR** (**intermediate representation**)
  - Principle: **meaning** of the source must be **preserved** in the **IR**.
- BACK END**:
  - Encodes knowledge of the **target** language
  - Transforms: from the **IR** to the **target**
  - Principle: **meaning** of the **IR** must be **reflected** in the **target**.

**Q.** How many **IRs** needed for building a number of compilers:

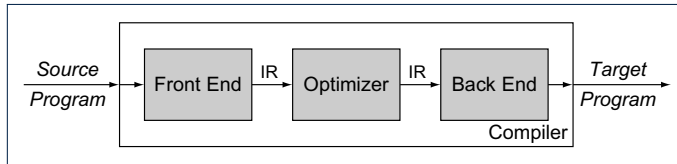
JAVA-TO-C, C#-TO-C, JAVA-TO-PYTHON, C#-TO-PYTHON?

**A.** Two **IRs** suffice: One for **OO**; one for **procedural**.

⇒ IR should be as **language-independent** as possible.

4 of 20

## Compiler: Typical Infrastructure (2)

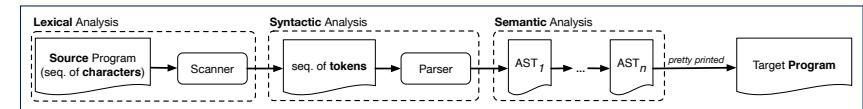


### OPTIMIZER:

- An **IR-to-IR** transformer that aims at “improving” the **output** of front end, before passing it as **input** of the back end.
  - Think of this transformer as attempting to discover an “**optimal**” solution to some computational problem.  
e.g., runtime performance, static design
- Q.** Behaviour of the **target** program depends upon?
- Meaning** of the **source** preserved in **IR**?
  - IR-to-IR** transformation of the optimizer **semantics-preserving**?
  - Meaning** of **IR** preserved in the generated **target**?
- (1) – (3) necessary & sufficient for the **soundness** of a compiler.

5 of 20

## Compiler Infrastructure: Scanner vs. Parser vs. Optimizer



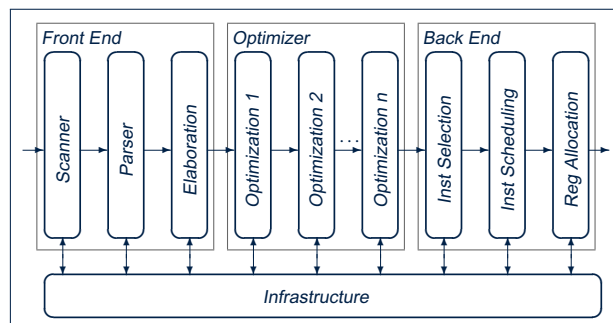
- The same input program may be perceived differently:
  - As a **character sequence** [ subject to **lexical** analysis ]
  - As a **token sequence** [ subject to **syntactic** analysis ]
  - As a **abstract syntax tree (AST)** [ subject to **semantic** analysis ]
- (1) & (2) are routine tasks of lexical/grammar rule specification.
- (3) is where the **most creativity** is used to a compiler:  
A series of **semantics-preserving AST-to-AST** transformations.

7 of 20

## Example Compiler 1



- Consider a conventional compiler which turns a **C-like program** into executable **machine instructions**.
- The **source** and **target** are at different levels of **abstractions**:
  - C-like program is like “high-level” **specification**.
  - Machine instructions are the low-level, efficient **implementation**.



6 of 20

## Compiler Infrastructure: Scanner



- The source program is perceived as a sequence of **characters**.
  - A scanner performs **lexical analysis** on the input character sequence and produces a sequence of **tokens**.
  - ANALOGY: Tokens are like individual **words** in an essay.  
⇒ Invalid tokens ≈ Misspelt words
  - e.g., a token for a useless delimiter: e.g., space, tab, new line
  - e.g., a token for a useful delimiter: e.g., (, ), {, }, ,
  - e.g., a token for an identifier (for e.g., a variable, a function)
  - e.g., a token for a keyword (e.g., int, char, if, for, while)
  - e.g., a token for a number (for e.g., 1.23, 2.46)
- Q.** How to specify such **pattern of characters**?
- A. Regular Expressions (REs)**
- e.g., RE for keyword **while** [ while ]
  - e.g., RE for an identifier [ [a-zA-Z][a-zA-Z0-9\_]\* ]
  - e.g., RE for a white space [ [ \t\r ]+ ]

8 of 20

## Compiler Infrastructure: Parser

- A parser's input is a sequence of **tokens** (by some scanner).
- A parser performs **syntactic analysis** on the input token sequence and produces an **abstract syntax tree (AST)**.
- ANALOGY: ASTs are like individual **sentences** in an essay.  
⇒ Tokens not **parseable** into a valid AST ≈ Grammatical errors

**Q.** An essay with no spelling and grammatical errors good enough?

**A.** No, it may talk about non-sense (sentences in wrong contexts).

⇒ An input program with no lexical/syntactic errors should still be subject to **semantic analysis** (e.g., type checking, code optimization).

**Q.:** How to specify such **pattern of tokens**?

**A.:** **Context-Free Grammars (CFGs)**

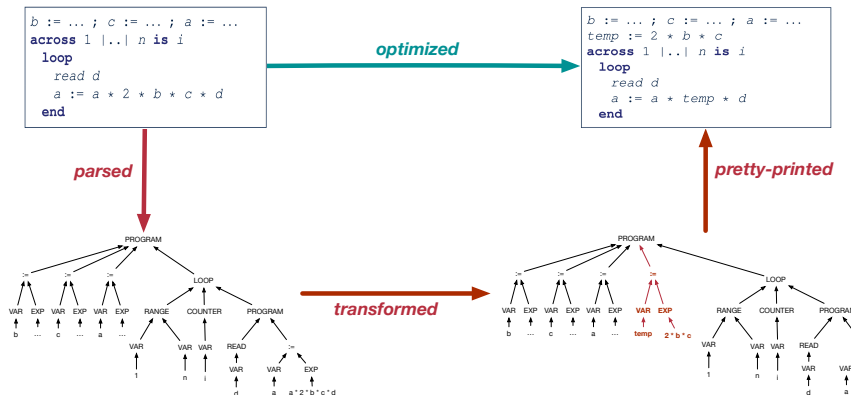
e.g., CFG (with **terminals** and **non-terminals**) for a while-loop:

```
WhileLoop ::= WHILE LPAREN BoolExpr RPAREN LCBRAC Impl RCBRAC
Impl      ::= Instruction SEMICOL Impl
```

9 of 20

## Compiler Infrastructure: Optimizer (2)

**Problem:** Given a user-written program, **optimize** it for best runtime performance.



11 of 20

## Compiler Infrastructure: Optimizer (1)

- Consider an input **AST** which has the pretty printing:

```
b := ... ; c := ... ; a := ...
across 1 |..| n is i
loop
  read d
  a := a * 2 * b * c * d
end
```

**Q.** AST of above program **optimized** for performance?

**A.** No ∴ values of 2, b, c stay invariant within the loop.

- An **optimizer** may **transform** AST like above into:

```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across 1 |..| n is i
loop
  read d
  a := a * temp * d
end
```

10 of 20

## Example Compiler 2

- Consider a compiler which turns an object-based **Domain-Specific Language (DSL)** into a **SQL database**.
- Why is an **object-to-relational compiler** valuable?

**Hint.** Which semantic domain is better for high-level specification?

**Hint.** Which semantic domain is better for data management?

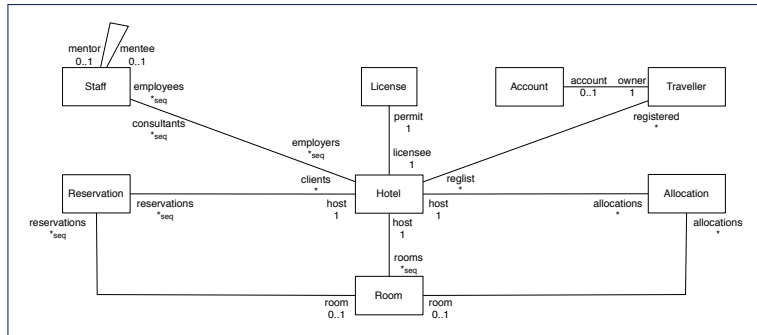
	managing big data	specifying data & updates
object-oriented environment	×	✓
relational database	✓	×

- Challenge:** **Object-Relational Impedance Mismatch**

12 of 20

## Example Compiler 2

- The input/source contains 2 parts:
  - DATA MODEL:** *classes & associations*  
e.g., data model of a Hotel Reservation System:



- BEHAVIOURAL MODEL:** update methods specified as *predicates*

13 of 20

## Example Compiler 2: Input/Source

- Consider a **valid** input/source program:

```
class Account {
  attributes
  owner: Traveller . account
  balance: int
}
```

```
class Traveller {
  attributes
  name: string
  reglist: set(Hotel . registered)[*]
}
```

```
class Hotel {
  attributes
  name: string
  registered: set(Traveller . reglist)[*]
  methods
  register {
    t? : extent(Traveller)
    & t? /: registered
    ==>
    registered := registered \/ {t?}
    || t?.reglist := t?.reglist \/ {this}
  }
}
```

- How do you specify the *scanner* and *parser* accordingly?

15 of 20

## Example Compiler 2: Transforming Data

```
class A {
  attributes
  s: string
  bs: set(B . a) [*]
}
```

```
class B {
  attributes
  is: set(int)
  a: A . bs
}
```

- Each class is turned into a *class table*:
  - Column `oid` stores the object reference. [ PRIMARY KEY ]
  - Implementation strategy for attributes:

	SINGLE-VALUED	MULTI-VALUED
PRIMITIVE-TYPED	column in <i>class table</i>	<i>collection table</i>
REFERENCE-TYPED	<i>association table</i>	

- Each *collection table*:
  - Column `oid` stores the context object.
  - 1 column stores the corresponding primitive value or `oid`.
- Each *association table*:
  - Column `oid` stores the association reference.
  - 2 columns store `oid`'s of both association ends. [ FOREIGN KEY ]

14 of 20

## Example Compiler 2: Output/Target

- Class *associations* are transformed to *database schemas*.

```
CREATE TABLE 'Account'(
  'oid' INTEGER AUTO_INCREMENT, 'balance' INTEGER,
  PRIMARY KEY ('oid'));
CREATE TABLE 'Traveller'(
  'oid' INTEGER AUTO_INCREMENT, 'name' CHAR(30),
  PRIMARY KEY ('oid'));
CREATE TABLE 'Hotel'(
  'oid' INTEGER AUTO_INCREMENT, 'name' CHAR(30),
  PRIMARY KEY ('oid'));
CREATE TABLE 'Account_owner_Traveller_account'(
  'oid' INTEGER AUTO_INCREMENT, 'owner' INTEGER, 'account' INTEGER,
  PRIMARY KEY ('oid'));
CREATE TABLE 'Traveller_reglist_Hotel_registered'(
  'oid' INTEGER AUTO_INCREMENT, 'reglist' INTEGER, 'registered' INTEGER,
  PRIMARY KEY ('oid'));
```

- Method *predicates* are compiled into *stored procedures*.

```
CREATE PROCEDURE 'Hotel_register'(IN 'this?' INTEGER, IN 't?' INTEGER)
BEGIN
  ...
END
```

16 of 20

## Example Compiler 2: Transforming Updates



**Challenge**: Transform *dot notations* into *relational queries*.

e.g., The AST corresponding to the following dot notation  
(in the context of class `Account`,  
retrieving the owner's list of registrations)

```
this.owner.reglist
```

may be transformed into the following (nested) table lookups:

```
SELECT (VAR 'reglist')
  (TABLE 'Hotel_registered_Traveller_reglist')
  (VAR 'registered' = (SELECT (VAR 'owner')
    (TABLE 'Account_owner_Traveller_account')
    (VAR 'owner' = VAR 'this'))))
```

17 of 20

## Beyond this lecture ...



- Read Chapter 1 of EAC2 to find out more about Example Compiler 1
- Read this paper to find out more about Example Compiler 2:

<http://dx.doi.org/10.4204/EPTCS.105.8>

18 of 20

## Index (1)



What is a Compiler? (1)

What is a Compiler? (2)

Compiler: Typical Infrastructure (1)

Compiler: Typical Infrastructure (2)

Example Compiler 1

Compiler Infrastructure:

Scanner vs. Parser vs. Optimizer

Compiler Infrastructure: Scanner

Compiler Infrastructure: Parser

Compiler Infrastructure: Optimizer (1)

Compiler Infrastructure: Optimizer (2)

19 of 20

## Index (2)



Example Compiler 2

Example Compiler 2

Example Compiler 2: Transforming Data

Example Compiler 2: Input/Source

Example Compiler 2: Output/Target

Example Compiler 2: Transforming Updates

Beyond this lecture...

20 of 20