Overview of Compilation

Readings: EAC2 Chapter 1

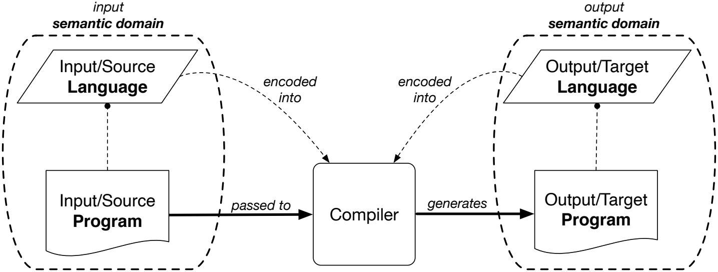
YORK UNIVERSITÉ UNIVERSITY

EECS4302 M:
Compilers and Interpreters
Winter 2020

CHEN-WEI WANG

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 and meanings
  e.g., OO, database, predicates
- 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

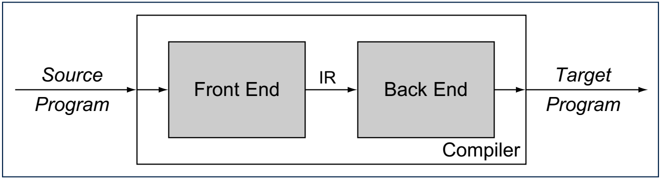
What is a Compiler? (2)

- The idea about a compiler is extremely powerful:

  You can turn anything to anything else,

  as long as the following are ***clear*** about them:
  - SYNTAX [***specifiable*** as CFGs]
  - SEMANTICS [***programmable*** as mapping functions]
- Construction of a compiler should conform to good *software engineering principles* .
  - Modularity & Information Hiding [interacting components]
  - Single Choice Principle
  - Design Patterns (e.g., composite, visitor)
  - Regression Testing at different levels: e.g., Unit & Acceptance

Compiler: Typical Infrastructure (1)



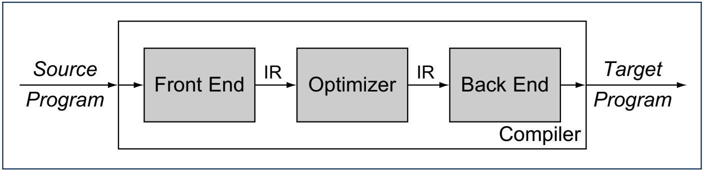
- **FRON END**:
  - Encodes: knowledge of the **source** language
  - Transforms: from the **source** to some ***IR*** (*intermediate representation*)
  - Principle: *meaning* of the source must be *preserved* in the ***IR***.
- **BACK END**:
  - Encodes knowledge of the **target** language
  - Transforms: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

Q. How many ***IRs*** needed for building a number of compilers: JAVA-TO-C, EIFFEL-TO-C, JAVA-TO-PYTHON, EIFFEL-TO-PYTHON?

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

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

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

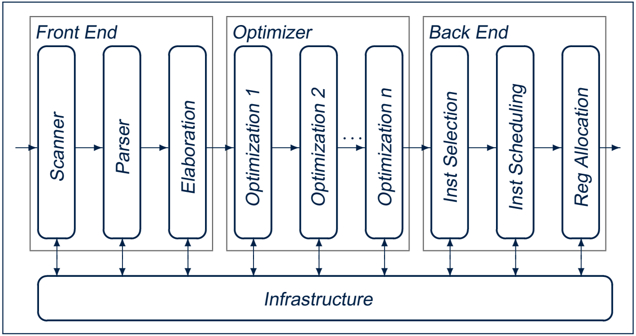
Q. Behaviour of the **target** program predicated upon?

1. ***Meaning*** of the **source** preserved in **IR**?
2. **IR**-to-**IR** transformation of the optimizer ***semantics-preserving***?
3. ***Meaning*** of **IR** preserved in the generated **target**?

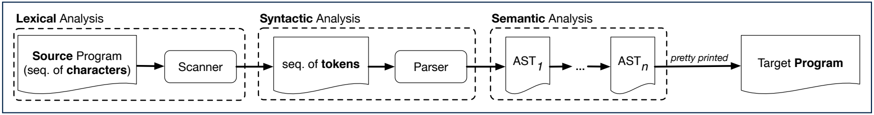
(1) – (3) necessary & sufficient for the *soundness* of a compiler.

Example Compiler One

- Consider a conventional compiler which turns a **C-like program** into executable **machine instructions.**
- The ***source*** (C-like program) and ***target*** (machine instructions) are at different levels of *abstraction*:
  - C-like program is like "high-level" ***specification***.
  - Macine instructions are the low-level, efficient ***implementation***.



Example Compiler One: Scanner vs. Parser vs. Optimizer



- The same input program may be treated differently:
  1. As a ***character sequence*** [subject to ***lexical*** analysis]
  2. As a ***token sequence*** [subject to ***syntactic*** analysis]
  3. 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 fun is about writing a compiler:

  A series of *semantics-preserving* AST-to-AST transformations.

Example Compiler One: Scanner

- The source program is treated 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 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]+`]

Example Compiler One: 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 speling 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 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*

Example Compiler One: Optimizer

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

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

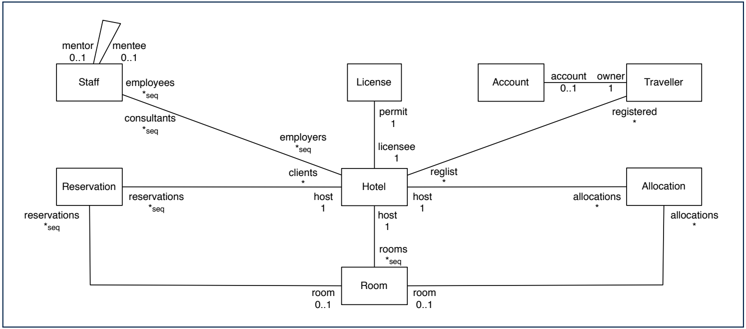
Q. AST of above program ***optimized*** for performance?
A. No $\because$ values of `2`, `b`, `c` stay invariant within the loop.

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

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

Example Compiler Two

- Consider a compiler which turns a ***Domain-Specific Language (DSL)*** of classes & predicates into a **SQL database.**
- The input/source contains 2 parts:
  - **DATA MODEL**: classes and associations (client-supplier relations) e.g., data model of a Hotel Reservation System:



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

Example Compiler Two: Mapping Data

```
class A {
  attributes
    s: string
    as: set(A . b) [*] }
```

```
class B {
  attributes
    is: set (int)
    b: B . as }
```

- 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 ***class table***	***collection table***
REFERENCE-TYPED	***association table***	

- 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]

Example Compiler Two: 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?

Example Compiler Two: Output/Target

- Class associations are compiled into ***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`));
```

- Predicate methods are compiled into ***stored procedures***.

```
CREATE PROCEDURE `Hotel_register`(IN `this?` INTEGER, IN `t?` INTEGER)
  BEGIN
    ...
  END
```

Example Compiler Two: Mapping Behaviour

- Challenge: Transform the OO dot notation into table queries.

  e.g., The AST corresponding to the following dot notation

  (in 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')))
```

- At the database level:
  - Maintaining a large amount of data is ***efficient***
  - Specifying **data** and **updates** is ***tedious*** & ***error-prone***.
  - RESOLUTIONS:
    - Define a DSL supporting the right level of ***abstraction*** for specification
    - Implement a DSL-TO-SQL compiler.

Beyond this lecture . . .

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

  `http://dx.doi.org/10.4204/EPTCS.105.8`

Index (1)

Index (2)