Programming Language Design

9/16 Design Dimensions and Code Modularity

Programming languages

- The languages we use to talk with machine
- Similar to natural language, in a sense
  - Have syntax and grammar
  - Vocabulary (libraries or functions?)
- Although the evolution of programming languages is controlled with intentions!
- Somehow reasonable
  - *According to a draft*
  - *Someone says that in a book*

Programming languages (cont.)

- But different
  - Manually created, of course
- No “exception” and “convention”
  - Well, someone might say actually we have a lot!
  - And both the two terms are used in programming languages… ;)
    - *“Exceptions” in Java*
    - *We say “programming convention”, “calling convention”*

What we will see in this course

- A programming language consists of
  - Theory
  - Design ← here we are
  - Implementation

Why?

- Tell you that there are a lot of languages besides C, C++, Java, and C#
- Not only imperative style object-oriented programming in the world!
- Suggest you learn more programming languages for fun…
  - Knowing at least two languages is really helpful!

We will see the design

- From smaller things to larger things
  - Name Binding, Typing, and Evaluation Strategy
  - Method Delegation and Class Composition
  - The Observer Pattern and the Visitor Pattern
  - Programming Paradigms

What I call design dimensions

- They determine how you can write a program in the language
  - Programming paradigms it supports! --- we will see them later
- Let us look into the design dimensions first
  - The fundamental things in the design of programming languages

When we start writing a program

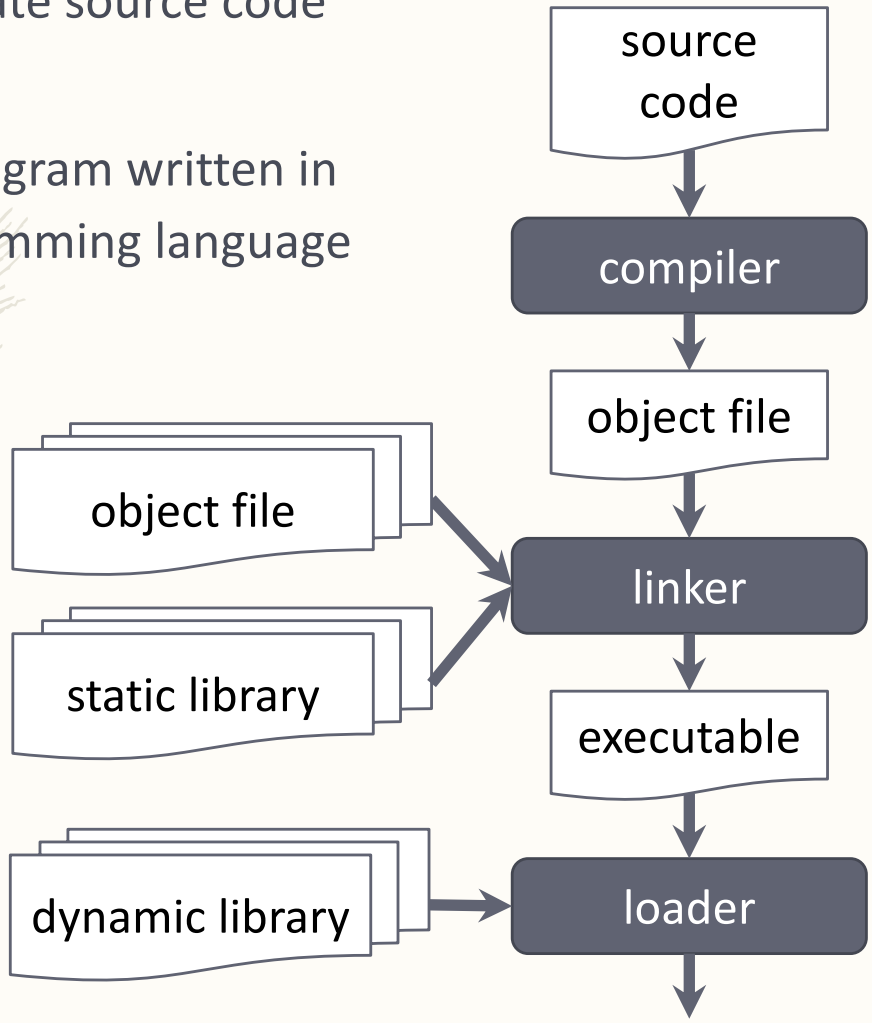
- How to define an identifier?
- Where can we use it?

- When we are reading a program
  - where to find the value for the identifier?
  - is it still available?

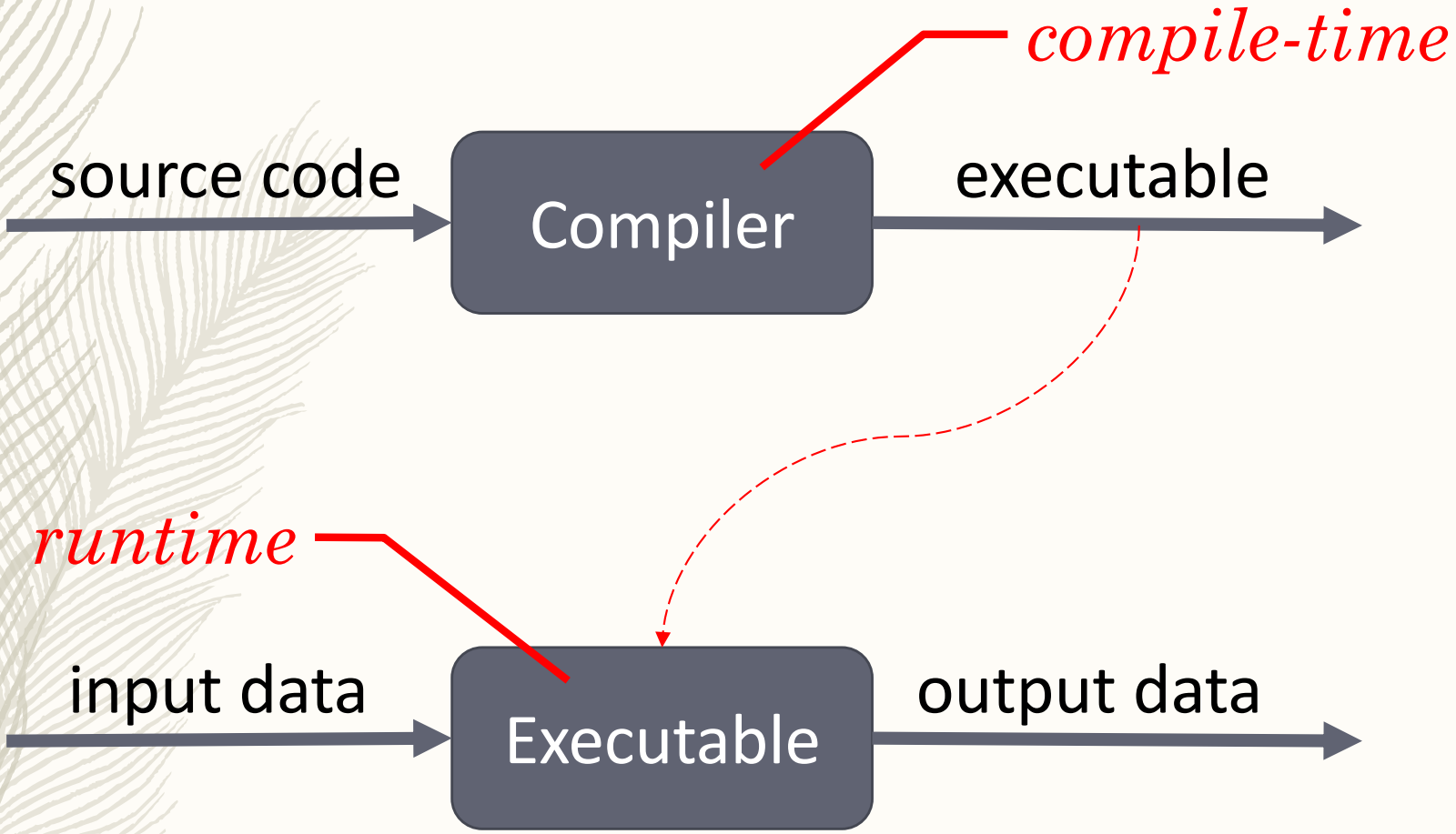
→ Need to know how programs will be processed

How your code is processed: compilation

- Compiler: translate source code to object code
- Source code: program written in a specific programming language



From source code to executable



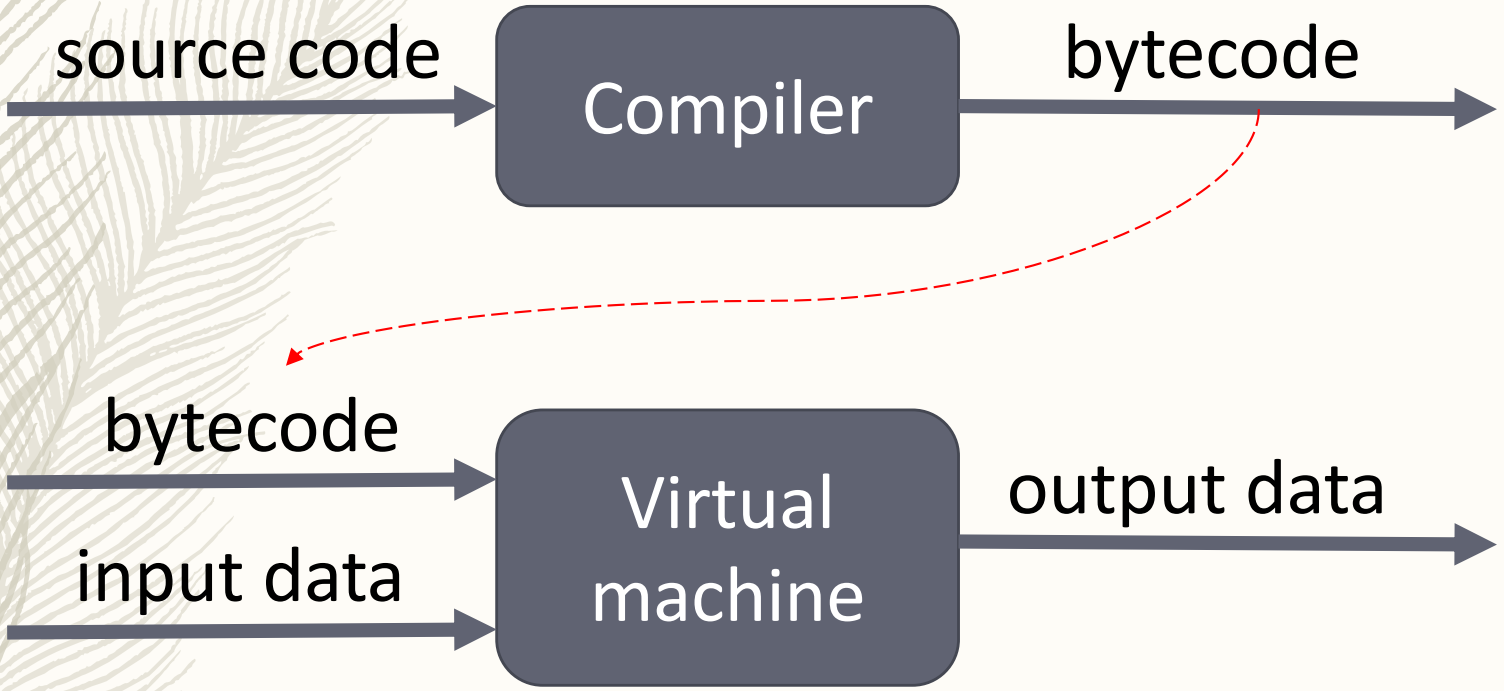
– (Here we ignore linker to simplify the explanation)

Another way to translate

– Interpreter: translate immediately and then execute

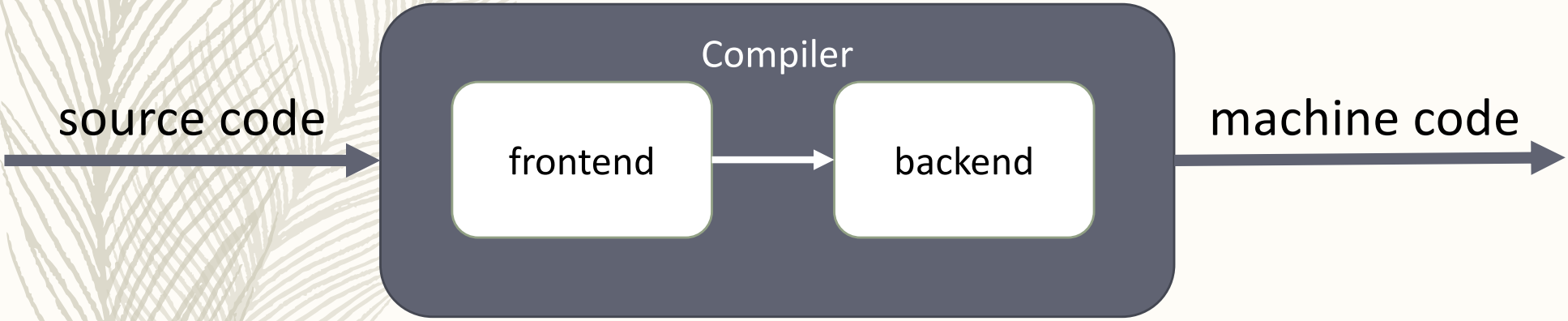
Virtual machine

– Bytecode are the instructions for virtual machine



Typical compiler architecture

- Frontend: from source code to abstract syntax tree (AST) or intermediate representation (IR)
- Backend: from AST or IR to machine code



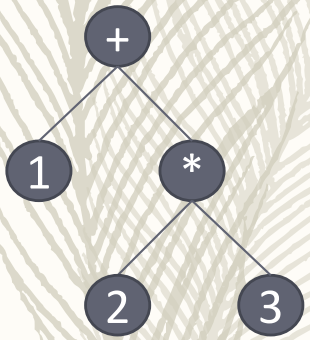
Reference

- *Charles N. Fischer, Ron K. Cytron, Richard J. LeBlanc, Jr. Crafting a Compiler, Global edition, Pearson.*
- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*
- *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, 2nd edition, Addison Wesley. (the Dragon book)*

What compilers do for us?

1. Separate tokens, and
   simply remove meaningless annotations
2. Try to organize tokens into a structure
3. Traverse the structure to get the meaning
4. Accept or reject!
5. Translate to machine code for target platform

for example,
1 + 2 * 3



Frontend: analysis

- Lexical analysis: recognize tokens in source code
- Syntactic analysis: use tokens to build a tree
- Semantic analysis: attach meaning to nodes in the tree

Lexical analysis: lexer

- Scan and tokenize the input into tokens
  - Take a stream of characters
  - Produce a stream of names, keywords, and punctuation marks
  - Discard white space and comments
- Pass tokens to parser for syntax analysis

Some token types might look like

Type	Examples
ID	i foo bar
NUM	0 10 99
IF	if
FOR	for
INT	int
LPAREN	(
RPAREN	)
LBRACE	{
RBRACE	}
LT	<
ASSIGN	=
INC	++
SEMI	;

- Input:

```
for (int i=0; i<10; i++) {
    // do somethings in the loop
}
```

- Lexer returns:

```
FOR LPAREN INT ID(i) ASSIGN
NUM(0) SEMI ID(i) LT NUM(10)
SEMI ID(i) INC RPAREN LBRACE
RBRACE
```

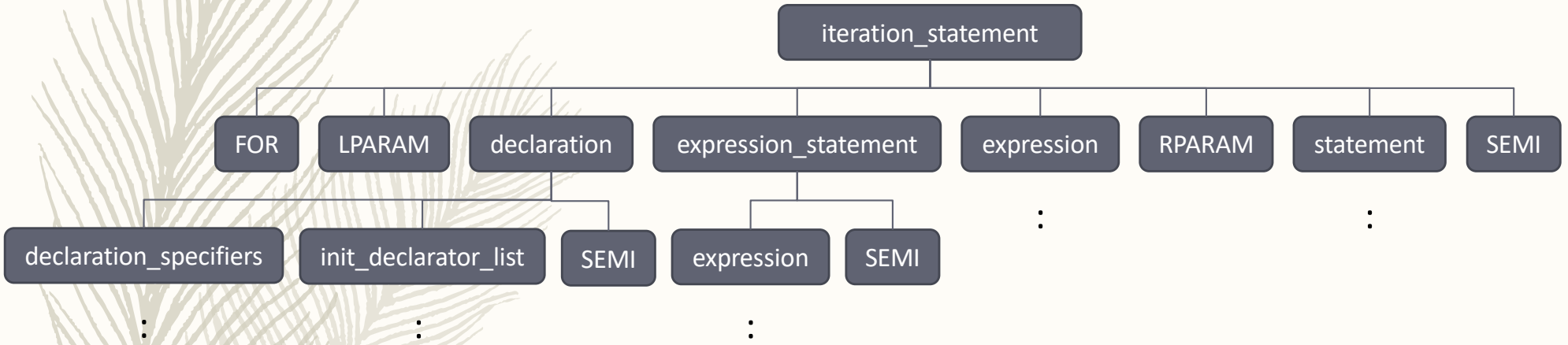
Syntactic analysis: parser

- Use tokens to build a syntactic structure
  - Concrete syntax tree (parse tree)
  - According to a formal grammar
  - Check the syntax of this language
- Eventually build abstract syntax tree for semantic analysis
  - Simplified syntactic representation
  - Meaningful data structure rather than the whole syntactic clutter

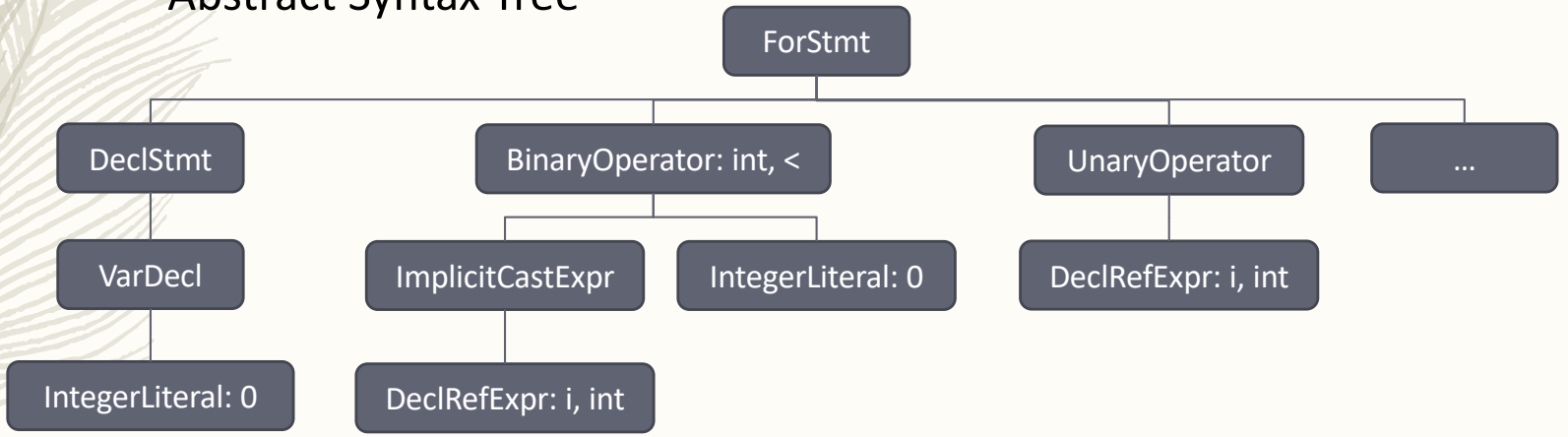
An example of CST and AST

```
for (int i=0; i<10; i++) {
        // do somethings in the loop
}
```

Concrete Syntax Tree



Abstract Syntax Tree

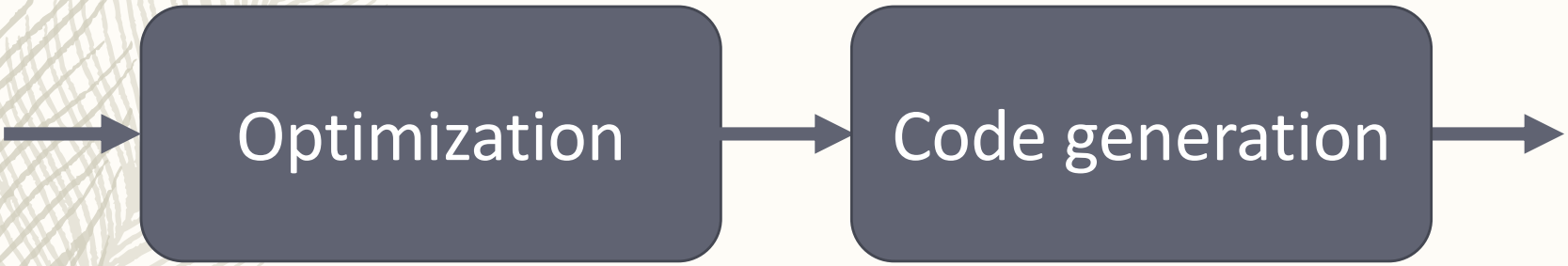


Semantics analysis

- Identifiers are bound to “meanings” in symbol tables
  - Connect variable definitions to their uses
  - Check each expression has a correct type
- Translate abstract syntax into a simpler representation
  - For generating machine code

Backend: synthesis

- Optimization: data-flow analysis and apply optimization
- Code generation: generate the code for target platform



When we start writing a program (again)

- How to define an identifier?
- Where can we use it?

- When we are reading a program
  - where to find the value for the identifier?
  - is it still available?

→ Given an identifier, what is it?

Name binding

- for example, given a pseudo-code,

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

Name binding (cont.)

- Bind an identifier to a value

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

What is “x”?

Scope

– Um, which x I should refer to?

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

Where to find “x”?

Scope

- Where the identifier can be used to refer to a value

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

Where to find “x”?

Static binding (lexical binding)

- Look up in the lexical environment
  - Hierarchical structure
  - Can understand by reading code

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

print out “5”

Dynamic binding

- All bindings are global
  - Determined at runtime
  - Hard to understand, but flexible

```
int x;
def printx() {
  print x;
}
def setx() {
  int x = 10;
  printx();
}
x = 5;
setx();
```

③ x = 10

② x = 10

① x = 5

print out “10”

Use Emacs to try it

- Emacs Lisp supports lexical binding since version 24.3
  - Enable it by "`(setq-local lexical-binding t)`"
  - Default is dynamic binding

- An example
  - Define a lambda that takes an argument and plus 3
    - `(lambda (arg) (+ 3 arg))`
  - We can give the lambda "5"
    - `((lambda (arg) (+ 3 arg)) 5)` → *the result is 8*

Reference

- https://www.emacswiki.org/emacs/LexicalBinding
- https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding

LISP 101

- LISt Processor
  - list is the major data structure in it
- a very old high-level language
  - since 1958
- has many important features
  - you will see some of them in later lessons
- its dialects include Common Lisp and Scheme

LISP 101 (cont.)

- written in S-expressions
  - parenthesized lists
  - use Polish notation (prefix notation)
  - for example, *1 + 2* in LISP:

```
(+ 1 2)
```

*1 * (2 + 3)* in LISP:

```
(* 1 (+ 2 3))
```

func(1, 2) in LISP:

```
(func 1 2)
```

let expression

- ”let” variables in the list be available in body

- Syntax

```
(let ((variable value)
      (variable value) …)
  body…)
```

- E.g.

```
(let ((x 1)
      (y 2) …)
  you-can-use-x-and-y-here…)
```

let expression (cont.)

– ”let” variables in the list be available in body

– variables can only be used in the body

```
(let ((x 1)
       (y x) …)
  you-can-use-x-and-y-here…)
```

the “x” in “(x 1)” is not available!

– use let in let…

```
(let ((x 1))
  (let ((y x) …)
  you-can-use-x-and-y-here…))
```

refer to the “x” in “(x 1)”

let* expression

– ”let” the variables that are defined in the same binding form be available

– Or using let* instead

```
(let* ((x 1)
       (y x) …)
  you-can-use-x-and-y-here…)
```

refer to the “x” in “(x 1)”

→ A syntax sugar of nested let expressions

lambda expression

- Define an anonymous function object, but can be given to a variable
- Syntax

```
(lambda (arg) (body))
```

- E.g.

```
(lambda (a) (* 7 a))
```

- Can be called by

```
((lambda (a) (* 7 a)) 3)
```

- Or called by funcall

```
(let ((foo (lambda (a) (* 7 a))))
  (funcall foo 3))
```

Use Emacs to try it (again)

- Emacs Lisp supports lexical binding since version 24.3
  - Enable it by "`(setq-local lexical-binding t)`"
  - Default is dynamic binding

- An example
  - Define a lambda that takes an argument and plus 3
    - `(lambda (arg) (+ 3 arg))`
  - We can give the lambda "5"
    - `((lambda (arg) (+ 3 arg)) 5)` → *the result is 8*

Reference

- https://www.emacswiki.org/emacs/LexicalBinding
- https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding

Use Emacs to try it (cont.)

- If we want to store 3 in a variable...
  - `(let ((to_add 3)) (lambda (arg) (+ to_add arg)) )`
- And then call it
  -
```
(funcall
 (let ((to_add 3)) (lambda (arg) (+ to_add arg))) 5)
```
- Lexical binding
  - It prints out ”8”
- Dynamic binding
  - `Debugger entered--Lisp error: (void-variable to_add)`
  - →
```
(require 'cl)
 (funcall (lexical-let ((to_add 3))
                       (lambda (arg) (+ to_add arg))) 5)
```

the next: type

- Now we know where to find the value for an identifier
  - find out it in outer scope
  - according to the latest binding
- But, what is it, eventually?
  - an integer, a string? or?

the next: type (cont.)

- We need further information
  - to understand what it is
  - Otherwise, we don't know how to use the identifier properly
    - how to assign it a proper value
    - when to use its value

- Type is such an information to help you

Type system

"A type system is a syntactic method for enforcing levels of abstraction in programs. The study of type systems--and of programming languages from a type-theoretic perspective--has important applications in software engineering, language design, high-performance compilers, and security."

--- Benjamin C. Pierce

Reference

- *Benjamin C. Pierce. Types and Programming Languages, The MIT Press.*

An informal explanation of type

- A property associated with variables, functions, expressions, etc.

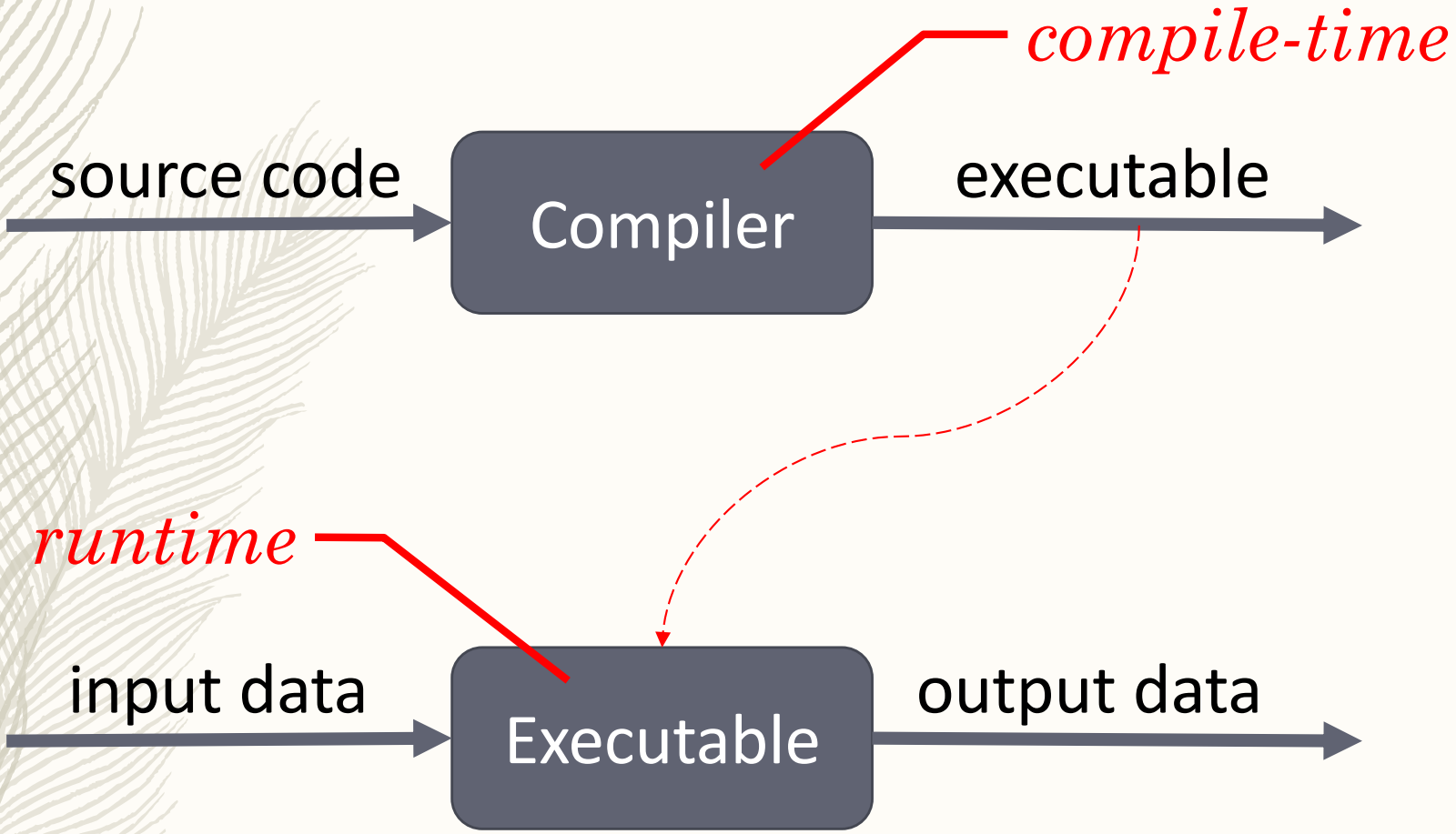
- Type system: rules used in a programming language
  - to reduce bugs, of course not all, but
  - ensure no type-error, at least

Type checking

- Enforce the constraints of types in a program
  - Statically at compile-time
  - Dynamically at runtime

- Statically typed programming languages:
  - C, C++, Fortran, Haskell, Java, Scala, …
- Dynamically typed programming languages:
  - Common Lisp, JavaScript, Perl, Python, Ruby, Scheme, Smalltalk, …

Compile-time vs. Runtime (again)



Static typing

- Check the type by analyzing the source code
  - The types of expressions are determined at compile-time
  - Basically the compiler can generate code with better optimization
- For example,

```
int x;
x = "x is an integer"  // error!
```

Dynamic typing

- Check the type according to the evaluation at runtime
  - The type of a variable can be changed

- For example,

```
x = 1;  // x is an integer
x = "now x is a string"  // ok!
```

Static typing vs. Dynamic typing

- Static typing
  - Detect errors before execution
  - Help compilers to generate an efficient code
- Dynamic typing
  - Easier to write and read
    - *Without a lot of type declaration*
  - Code is more flexible
    - *Pass variables without knowing their types*

Strong typing vs. Weak typing

- The terms are not very well-defined
  - How strong? how weak?
- But, in general, strong typing means
  - Ask programmers to do explicit type conversion (casting)
- Weak typing means
  - Do implicit type conversion

The things that might confuse you

- A language in which type declarations can be omitted, it must be a dynamically typed language?
  - NO, some compilers can do “type inference” for programmers
- In some language like Scala:

```
object InferenceTest1 extends Application {
  val x = 1 + 2 * 3          // the type of x is Int
  val y = x.toString()       // the type of y is String
  def succ(x: Int) = x + 1   // succ returns Int values
}
```

Reference

- *http://www.scala-lang.org/old/node/127*

The things that might confuse you (cont.)

- Why I see types in some JavaScript code? Is it the type inference mechanism?
  - Not exactly
  - Some languages like TypeScript provide type inference
  - Some languages might allow to use type annotations for only warnings or optimization
    - *Still leave type checking at runtime*

The things that might confuse you (cont.)

- Does dynamically typed mean untyped?
  - NO, since we say dynamically “typed”
  - Types exist and are associated with expressions in dynamically typed languages
  - Although some people might use “untyped” for the context of dynamically typed…

The things that might confuse you (cont.)

- Does statically typed imply strong typed, and dynamically typed imply weak typing?
  - NO, a statically typed language might allow implicit type conversion and a dynamically typed language might check type strongly
- In Python (dynamic, strong):

```
1 + "2"
→ TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

- In JavaScript (dynamic, weak):

```
var msg = 1 + "2";
→ "12"
```

the next: given to functions

- functions are pieces of code
  - get some values, do some operations, and return some values
  - we already know how to find the values for the variables directly used in “some operations”
- How about the ones given to functions?
  - “some values” got by functions

Parameter and Argument

- Parameter
  - i.e. Formal parameter
  - The parameter in the function definition
    "p" in the following example:

```
void foo(int p) {
        :
}
```

- Argument
  - i.e. Actual parameter
  - The parameter given to the function call
    "a" in the following example:

```
foo(a);
```

Parameter-passing variations

- How to bind its parameter to a value when executing a function body
- Call-by-value
  - Pass the value
- Call-by-reference
  - Pass the reference

Reference

- *Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages, 3rd edition, The MIT Press.*

Call-by-value

- For example,

```
void setx(int x) { x = 4; };
int a = 3;
setx(a);
printf("%d", a);
```

→ print out “3”

- The denoted value associated with x is a reference
  - that initially contains the same value as the reference associated with a
  - but these references are distinct

Call-by-reference

- For example,

```
void setx(int &x) { x = 4; };
int a = 3;
setx(a);
printf("%d", a);
```

→ print out “4”

- Pass the function a reference to the location of the caller’s variable
  - Rather than the contents of the variable
  - The parameter is then bound to the location

A confusing thing: pointer

- Looks like call-by-reference??

```
void setx(int *x) { *x = 4; };
int a = 3;
setx(&a);
printf("%d", a);
```

→ print out “4”

- Actually it is a kind of call-by-value! Why?

A confusing thing: pointer (cont.)

- What it passed is the value of reference
  rather than the reference itself
- If we assign it the reference of a new variable…

```
void setx(int *x) { int y; x = &y; *x = 4; };
int a = 3;
setx(&a);
printf("%d", a);
→ print out “3” but not “4”!
```

- Named call-by-sharing
  - An informal name you might see: call-by-address?

Reference

- *Barbara Liskov. A Note on CLU. Computation Structures Group Memo 112*

Evaluation strategy

- We have seen three kinds of parameter passing
  - Call-by-value
  - Call-by-reference
  - Call-by-sharing
- They are all eager evaluation
  - Always find a value for each argument for a function

Evaluation strategy (cont.)

- Lazy evaluation
  - Arguments given to a function call is not evaluated
    - *Until it is needed by the function body*
  - How could we call a function without evaluating its arguments?
  - Both the expression and the environment are included in a ”thunk” and passed
    - *thunk: a value that is yet to be evaluated [computer jargon]*

Lazy: call-by-name

- The argument is “frozen” (unevaluated) when passed to the function
  - and it is “thawed” when the function evaluates it
- Leave the argument unevaluated until it is used inside the function
  - Avoid unnecessary evaluation

Lazy: call-by-need

- Remember the result once we evaluated a thunk
  - Replace the thunk with the value
  - Avoid to evaluate it again
- An instance of the general strategy “memoization”
- Heavily used in functional programming languages like Haskell

Strict semantics

- Strict function
  - Fail when
    - *encounter an error*
    - *encounter endless loop in evaluation*
  - Strictly evaluate the arguments given to the function call

- Usually innermost-first evaluation, e.g.

  `a(b(c))`

  → evaluate c

  → evaluate b with the result of c

  → evaluate a with the result of b

Non-strict semantics

- Non-strict functions
  - Usually outermost-first evaluation, e.g.
    a(b(c))
    → evaluate a, b, c, in order
  - Evaluate a without strictly evaluate b and c
    - It continue to evaluate a
      even if b causes an infinite loop
      even if c result in divide-by-zero error

Reference

- *https://wiki.haskell.org/Non-strict_semantics*
- *https://wiki.haskell.org/Lazy_vs._non-strict*

Does lazy means non-strict?

- In most cases, the two terms are used interchangeably
- Not exactly, they refer to different things
  - but has a strong correspondence
  - Strict and non-strict are used when talking about the semantics
  - Eager/lazy describe how we evaluate the expressions

Abstract data type

- "An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects"
- When a programmer uses of an abstract data object
  - Concerned only with the behavior but not with implementation details
- Examples
  - List, Map, Set, Stack, etc.

Reference

- *Barbara Liskov and Stephen Zilles, Programming with Abstract Data Types, in the proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, 1974.*

Abstract data type (cont.)

- A kind of abstraction
  - Abstraction is always an important thing in computer science
    - *especially in programming language and software engineering*
  - Separate the functionality and implementation
    - *How to use it is one thing*
    - *How it does in detail in another thing*

Data structure

- How data are organized
  - Related to implementation details and efficiency
  - Examples: Array, Linked List, Hash Table, etc.
- Can be used to implement abstract data type

e.g. Map and HashMap in Java

- the Map interface
  - map keys to values
  - can be regarded as abstract data type
- the HashMap class
  - Hash table based implementation of the Map interface

- Other Map implementations?
  - TreeMap: a Red-Black tree based implementation

e.g. List and ArrayList in Java

- the List interface
  - An ordered collection, aka a sequence
  - can be regarded as abstract data type
- the ArrayList class
  - Resizable-array implementation of the List interface

- Other List implementations?
  - LinkedList: doubly-linked list implementation

Modular

consisting of separate parts or units which can be put together to form something, often in different combinations

--- Longman Dictionary of Contemporary English

Modular (cont.)

Modular means relating to a part of a machine, especially a computer, which performs a particular function.

--- Collins COBUILD Advanced American English Dictionary

Code Modularity

- Modularity
  - How well the code is modularized
- How to modularize the code in a system?
  - based on functionality

Code Modularity (cont.)

- Some terms related to “module”, from large to small, roughly
  - library, module
  - package
  - file, unit
  - class, function
  - block

→ Let’s start from the smallest one

Language construct

- The built-in elements given by a programming language to construct your programs
  - Rather than a piece of code
- For programming paradigms
  - class, method, variable, function, etc.
- For control flow
  - if, while, for, etc.

without “for” construct?

- We have to repeat the code
  - annoying
- for example, add values in arrays
  - with for

```
for(int i=0; i<5; i++)
  C[i] = A[i] + B[i];
```

  - without for

```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
C[4] = A[4] + B[4];
```

Not just annoying...

- Error-prone
  - we must carefully check if we have typos

```
C[0] = A[0] + B[0];
C[1] = A[0] + B[1];    // a typo
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
C[4] = A[4] + B[4];
```

  - how to ensure the numbers in lines are consistent?

Not just annoying... (cont.)

- How to easily and correctly modify?
  - e.g. replace A[i] with D[i]

```
C[0] = D[0] + B[0];
C[1] = D[1] + B[1];
C[2] = D[2] + B[2];
C[3] = D[3] + B[3];
C[4] = D[4] + B[4];
```

  - I just want to replace A with D, but cannot avoid the risk of modifying the array index...
    - *since the code are put together*

Not just annoying... (cont.)

- How to reuse the operation?
  - give two arrays
  - add each elements in them
  - store the result in another array
- copy-and-paste??
  - never do that...

Root cause of this problem

– without “for” construct

```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
C[4] = A[4] + B[4];
```

– Similar code are repeated

  – so we have to ensure the consistency among them

– Different code are mixed up

  – so we have to apply modifications very carefully

Code Modularity (cont.)

- Code should be modularized
  - according to their functionalities
- Programmers should always
  - avoid repeating the same code
  - avoid mixing code for different purpose together

Think about...

- How to reuse parts of the program?
  - we don't want to implement everything every time, right?
- How to make the program easier to maintain?
  - modify a specific functionality
    - *without touching code for different functionalities*
  - apply a specific functionality
    - *without too much efforts*

Language construct (again)

- The built-in elements given by a programming language to construct your programs
  - Rather than a piece of code

- Let you easily use a mechanism in the language
  - make code clear and shorter
  - easy to reuse and unplug

Next lecture

9/23 **Practice 1**

9/30 Object-Oriented Programming