哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 6: Maintainability-Oriented Software Construction Approaches

6.3 Maintainability-Oriented Construction Techniques

Wang Zhongjie
rainy@hit.edu.cn

April 13, 2018

Outline

- **State-based construction**
  - Automata-based programming
  - Design Pattern: Memento provides the ability to restore an object to its previous state (undo).
  - Design Pattern: State allows an object to alter its behavior when its internal state changes.
- **Table-driven construction ***
- **Grammar-based construction**
  - Grammar and Parser
  - Regular Expression (regexp)
  - Design Pattern: Interpreter implements a specialized language.

学了这么多OO设计模式，不外乎都是delegation + subtying，万变不离其宗

除了OO，还有什么其他能够提升软件可维护性的构造技术？——本节从委派+子类型跳出来，学习以下三个方面：

(1) 基于状态的构造技术
(2) 表驱动的构造技术
(3) 基于语法的构造技术

Reading

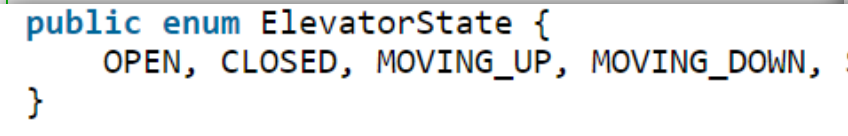
- **MIT 6.031：17、18**
- **Java编程思想：第13.6节**

1 State-based construction

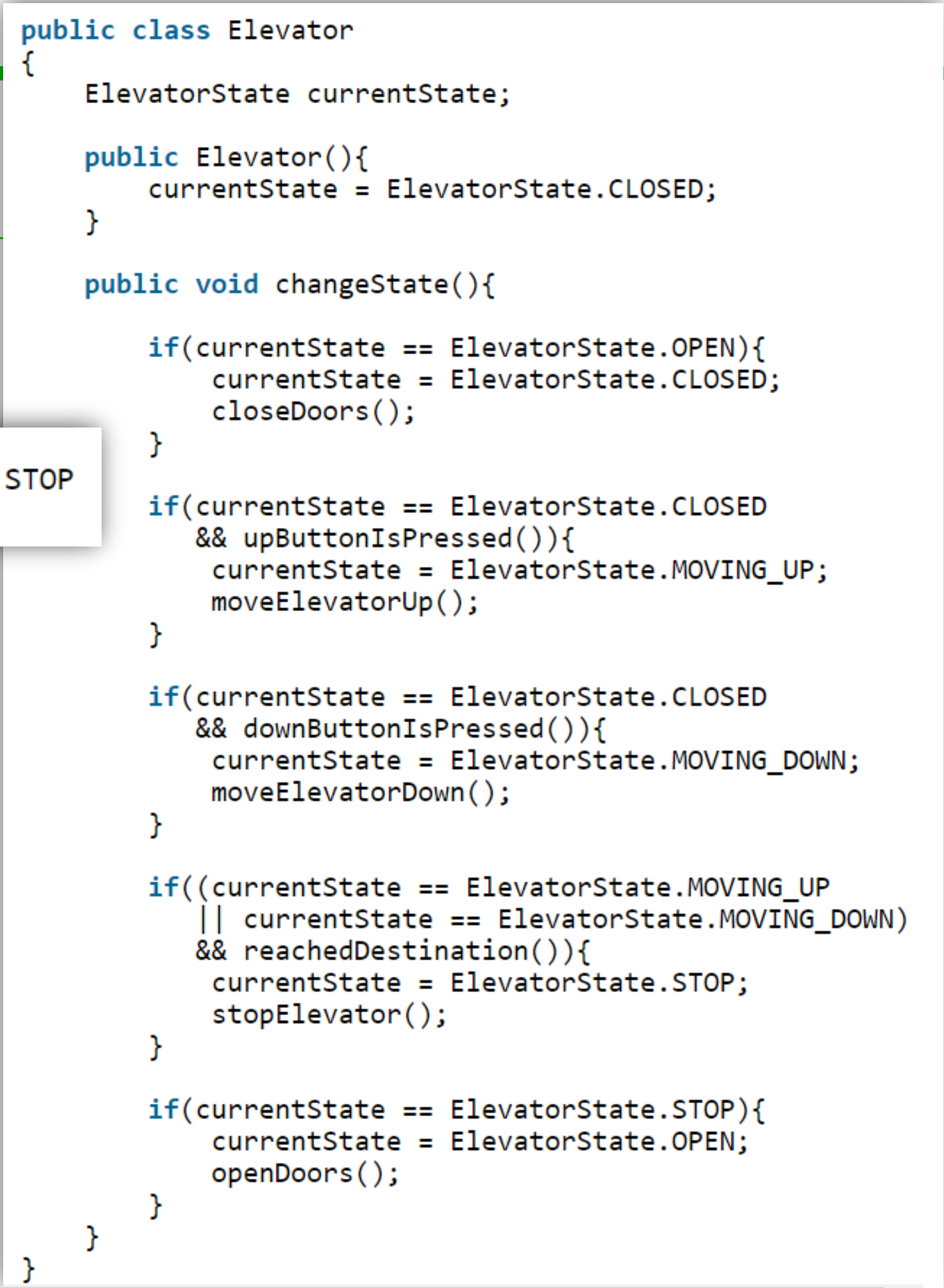
State-based programming

- **State-based programming** is a programming technology using finite state machines (FSM) to describe program behaviors, i.e., the use of "states" to control the flow of your program. 使用有限状态机来定义程序的行为、控制程序的执行
  - For example, in the case of an elevator, it could be `stop`, `moving up`, `moving down`, `stopping`, `closing the doors`, and `opening the doors`.
- Each of these are considered a state, and what happens next is determined by the elevator's current state. 根据当前状态，决定下一步要执行什么操作、执行操作之后要转移到什么新的状态
  - If the elevator has just closed its doors, what are the possibilities that can happen next? It can stop, move up, or move down.
  - When an elevator stops, you expect the next action to be the doors opening, moving up, or moving down.

The code



```java
public enum ElevatorState {
    OPEN, CLOSED, MOVING_UP, MOVING_DOWN, STOP
}
```

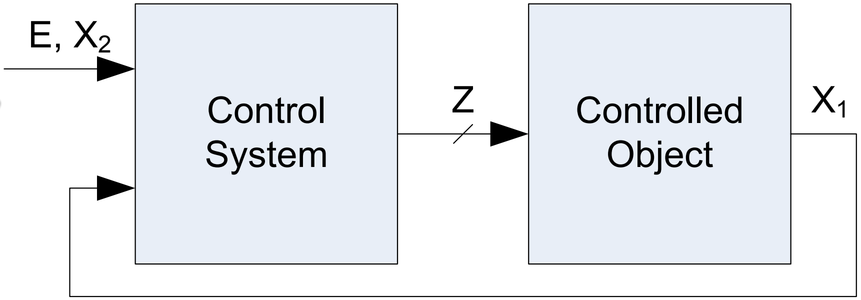


```java
public class Elevator
{
    ElevatorState currentState;

    public Elevator(){
        currentState = ElevatorState.CLOSED;
    }

    public void changeState(){

        if(currentState == ElevatorState.OPEN){
            currentState = ElevatorState.CLOSED;
            closeDoors();
        }

        if(currentState == ElevatorState.CLOSED
           && upButtonIsPressed()){
            currentState = ElevatorState.MOVING_UP;
            moveElevatorUp();
        }

        if(currentState == ElevatorState.CLOSED
           && downButtonIsPressed()){
            currentState = ElevatorState.MOVING_DOWN;
            moveElevatorDown();
        }

        if((currentState == ElevatorState.MOVING_UP
            || currentState == ElevatorState.MOVING_DOWN)
           && reachedDestination()){
            currentState = ElevatorState.STOP;
            stopElevator();
        }

        if(currentState == ElevatorState.STOP){
            currentState = ElevatorState.OPEN;
            openDoors();
        }
    }
}
```

(1) Automata-based programming

Automata-based programming

- **Automata-based programming is a programming paradigm in which the program or part of it is thought of as a model of a finite state machine (FSM) or any other formal automaton.**
  - Treat a program as a finite automata.
  - Each automaton can take one "step" at a time, and the execution of the program is broken down into individual steps.
  - The steps communicate with each other by changing the value of a variable representing "the state“.
  - Control flow of the program is determined by the value of that variable.
- **Application design approach should be similar to the design of control systems (Automata System).**
- **核心思想：将程序看作是一个有限状态自动机，侧重于对“状态”及“状态转换”的抽象和编程**

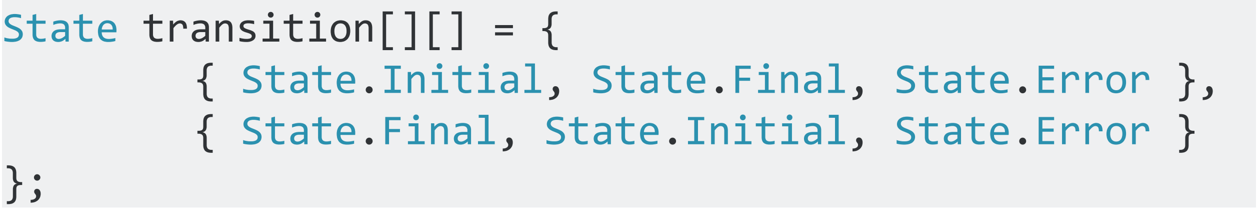


Automata-based programming

- **The time period of the program‘s execution is clearly separated down to the *steps of the automaton*.** 程序的执行被分解为一组自动执行的步骤
  - Each of the *steps* is effectively an execution of a code section (same for all the steps), which has a single entry point. Such a section can be a function or other routine, or just a cycle body.
- **Any communication between the steps is only possible via the explicitly noted set of variables named *the state*.** 各步骤之间的通讯通过“状态变量”进行
  - Between any two steps, the program can not have implicit components of its state, such as local (stack) variables' values, return addresses, the current instruction pointer, etc.
  - The state of the whole program, taken at any two moments of entering the step of the automaton, can only differ in the values of the variables being considered as the state of the automaton.

How to implement?

- **The whole execution of the automata-based code is a (possibly explicit) cycle of the automaton's steps.** 程序执行就可看作是各自动步骤的不断循环
- **The "state" variable can be a simple** `enum` **data type, but more complex data structures may be used.** 使用枚举类型enum定义状态
- **A common technique is to create a state transition table, a two-dimensional array comprising rows representing every possible state, and columns representing input parameter.** 使用二维数组定义状态转换表
  - The value of the table where the row and column meet is the next state the machine should transition to if both conditions are met.



```
State transition[][] = {
        { State.Initial, State.Final, State.Error },
        { State.Final, State.Initial, State.Error }
};
```

See Wikipedia: https://en.wikipedia.org/wiki/State_transition_table

Applications Areas

- **High reliability systems**
  - Military applications
  - Aerospace industry
  - Automotive industry
- **Embedded systems**
- **Mobile systems**
- **Visualization systems**
- **Web applications**
- **Client-server applications**

(2) State Pattern

状态模式 (behavioral pattern)

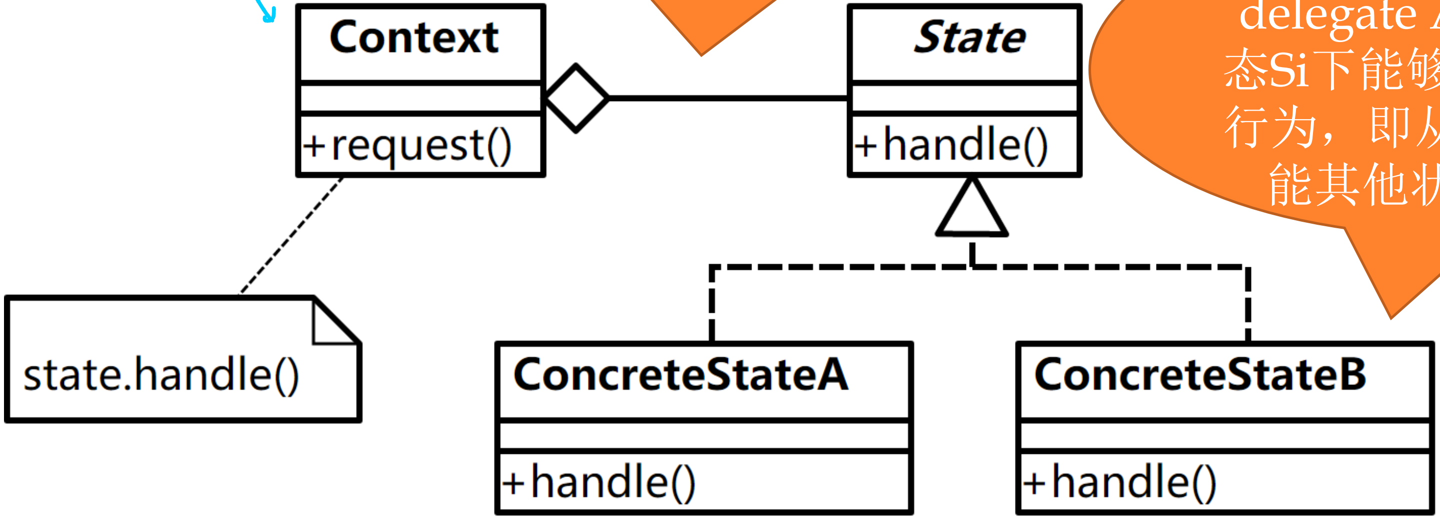
State pattern

- **Suppose an object is always in one of several known states**
- **The state an object is in determines the behavior of several methods**
- **Could use if/case statements in each method**
- **Better solution: state pattern**

- **Have a reference to a state object**
  - Normally, state object doesn’t contain any fields
  - Change state: change state object
  - Methods delegate to state object

Structure of State pattern

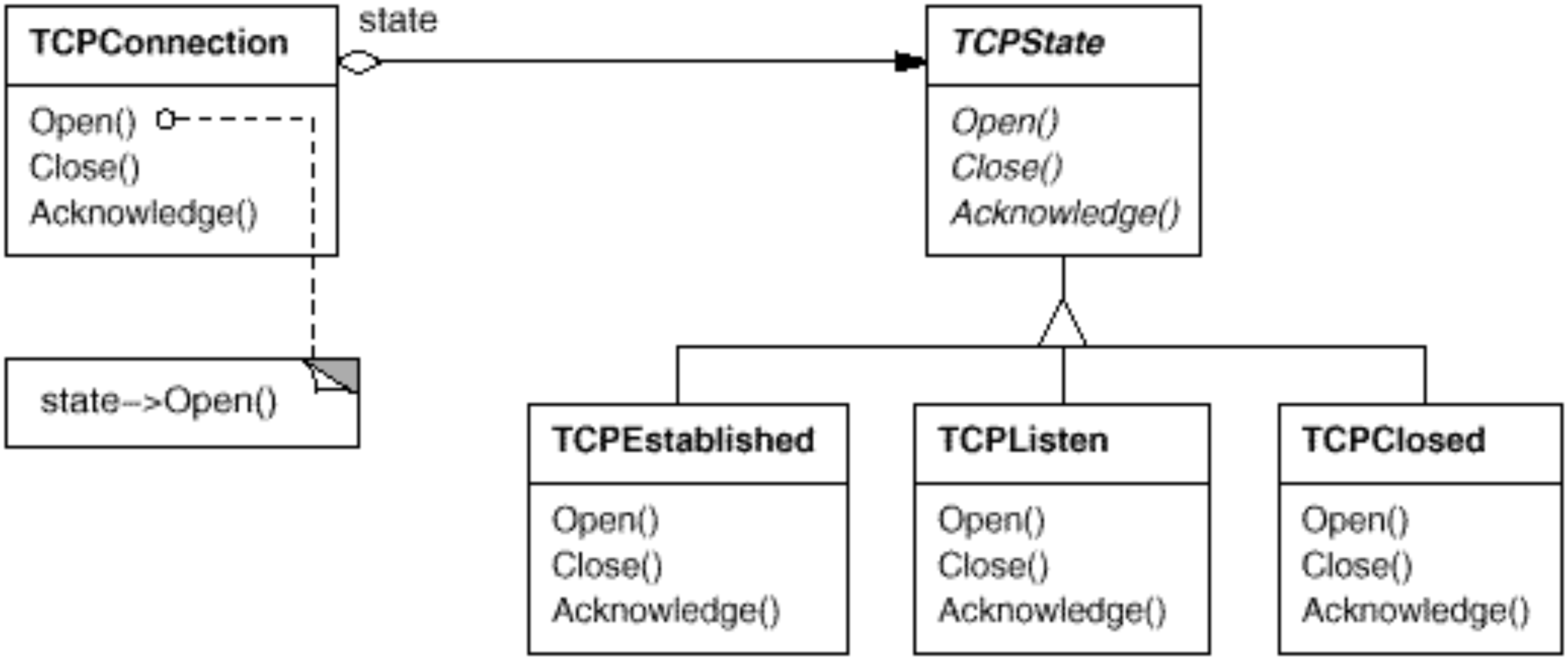
原始ADT



依旧逃离不了无处不在的delegation：ADT的操作看作是状态转换，都委派给了外部的state对象

不同的“状态”子类：delegate ADT在该状态Si下能够发生的所有行为，即从Si到所有可能其他状态的转换

Instance of State Pattern



State pattern notes

- **Can use singletons for instances of each state class**
  - State objects don’t encapsulate state, so can be shared -- immutable
- **Easy to add new states**
  - New states can extend other states
  - Override only selected functions

Example – Finite State Machine

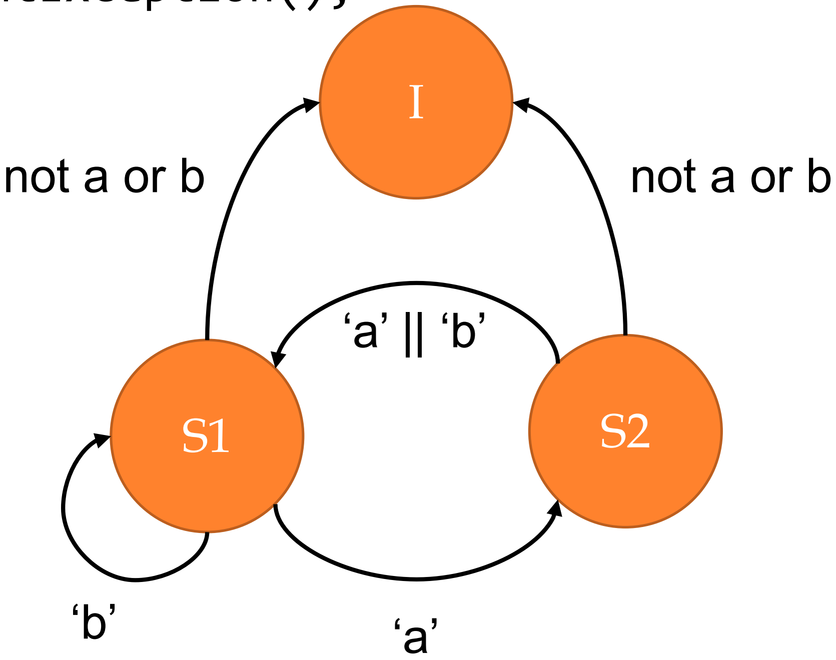
将改变状态的“动作”
delegate到state对象

state 由 Context( ) 传入,

```
class Context {
    State state; //保存对象的状态
    //设置初始状态
    public Context(State s) {state = s;}
    //接收外部输入，开启状态转换
    public void move(char c) {  state = state.move(c); }
    //判断是否达到合法的最终状态
    public boolean accept() {  return state.accept();  }
    public State getState() {  return this.state; }
 }


 //状态接口
 public interface State {
      State move(char c);
      boolean accept();
 }
```

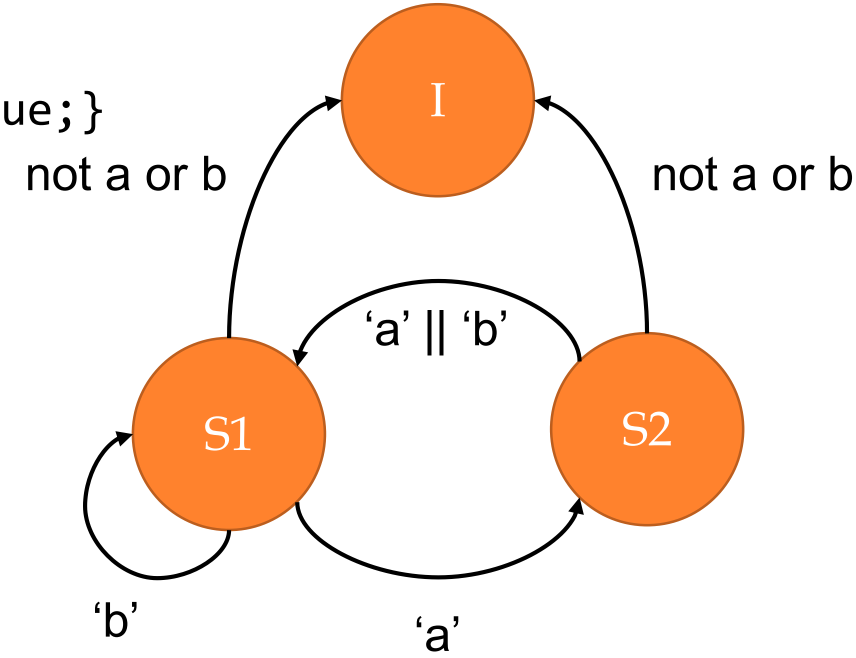
FSM Example – cont.

```
class State1 implements State {
  static State1 instance = new State1();  //singleton模式
  private State1() {}
  public State move (char c) {
     switch (c) {
      case 'a': return State2.instance;
      case 'b': return State1.instance;
      default: throw new IllegalArgumentException();
     }
  }
  public boolean accept() {
        return false;
  } //该状态非可接受状态
}
```



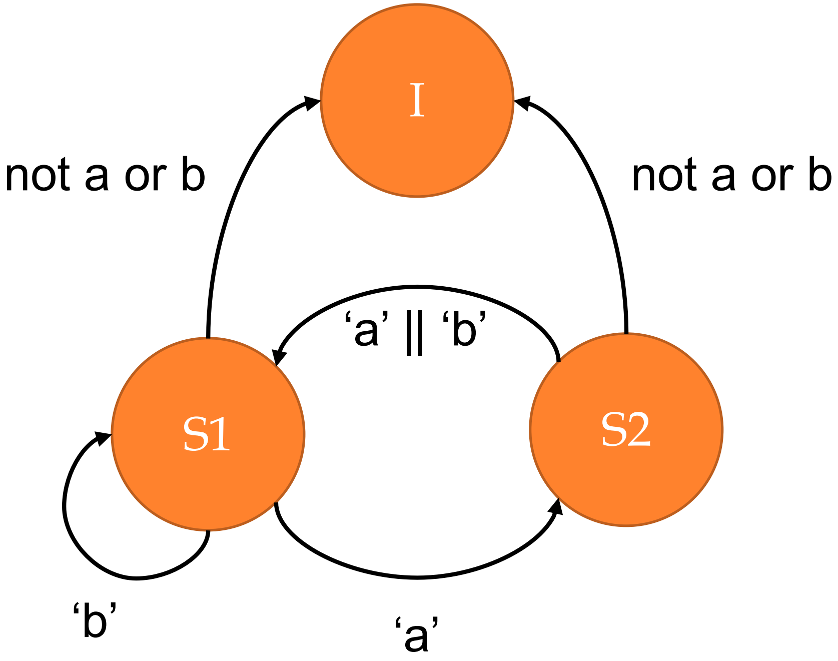
FSM Example – cont.

```
class State2 implements State {
   static State2 instance = new State2();
   private State2() {}
   public State move (char c) {
       switch (c) {
           case 'a': return State1.instance;
           case 'b': return State1.instance;
           default: throw new IllegalArgumentException();
          }
    }
   public boolean accept() {return true;}
}
```



Example

```
public static void main(String[] args) {
    Context context = new Context(new State1());
    for (int i = 0; i < args.length; i++) {
        context.move(args[i]);
        if(context.accept())
            break;
    }
}
```



(3) Memento Pattern

备忘录模式 (behavioral)

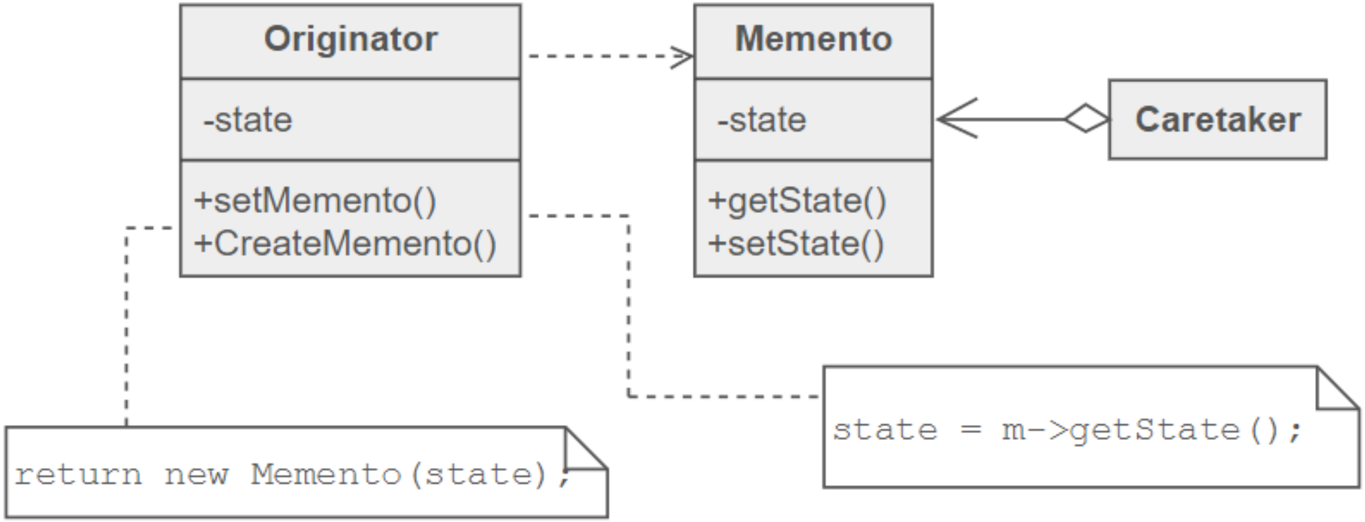
Memento Pattern

- **Intent**
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
  - A magic cookie that encapsulates a "check point" capability.
  - Promote undo or rollback to full object status.
- **Problem:** Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

- 记住对象的历史状态，以便于“回滚”

Memento Pattern

- **Memento design pattern defines three distinct roles:**
  - `Originator` - the object that knows how to save itself. 需要“备忘”的类
  - `Caretaker` - the object that knows why and when the `Originator` needs to save and restore itself. 添加originator的备忘记录和恢复
  - `Memento` - the lock box that is written and read by the `Originator`, and shepherded by the `Caretaker`. 备忘录，记录originator对象的历史状态



Memento Pattern

```
class Memento {
    private State state;

    public Memento(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }
}
```

```
class Originator {
    private State state;

    public void setState(State state) {
        System.out.println("Originator: Setting state to " + state.toString());
        this.state = state;
    }

    public Memento save() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restore(Memento m) {
        state = m.getState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }
}
```

Memento Pattern

```
class Caretaker {
    private ArrayList<Memento> mementos
                  = new ArrayList<>();

    public void addMemento(Memento m) {
        mementos.add(m);
    }

    public Memento getMemento() {
        return mementos.get(1);
    }
}
```

mementos.size()-1

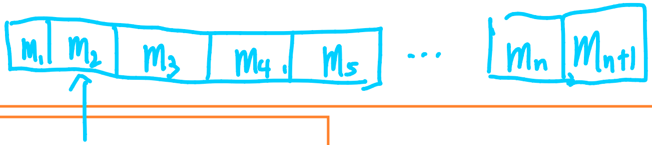
```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from
                  Memento: State3
```

```
public class Demonstration {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Originator originator = new Originator();
        originator.setState("State1");
        originator.setState("State2");
        caretaker.addMemento( originator.save() );
        originator.setState("State3");
        caretaker.addMemento( originator.save() );
        originator.setState("State4");
        originator.restore( caretaker.getMemento() );
    }
}
```

如何rollback两步、三步、…

Memento Pattern

```
class Caretaker {
    private ArrayList<Memento> mementos = new ArrayList<>();
    public void addMemento(Memento m) {mementos.add(m);}
    public Memento getMemento(int i) {
        if(mementos.size()-i < 0)
             throw new RuntimeException("Cannot rollback so many back!");
        return mementos.get(mementos.size()-i);
    }
}
```



ADT: m2.

```
public class Demonstration {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Originator originator = new Originator();
        originator.setState("State2");
        caretaker.addMemento( originator.save() );
        originator.setState("State3");
        caretaker.addMemento( originator.save() );
        originator.setState("State4");
        originator.restore( caretaker.getMemento(2) );
    }
}
```

2* Table-driven construction

What is “Table-Driven”?

- **A table-driven method is a schema that uses tables to look up information rather than using logic statements (such as `if-else` and `switch-case`).**

- **In simple cases, it's quicker and easier to use logic statements, but as the logic chain becomes more complex, table-driven code:**
  - Simpler than complicated logic
  - Easier to modify
  - More efficient

- **表驱动编程的核心思想：将代码中复杂的if-else和switch-case语句从代码中分离出来，通过“查表”的方式完成，从而提高可维护性**

An example

- **Suppose you wanted to classify characters into letters, punctuation marks, and digits; you might use a complicated chain of logic:**

```
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||
     ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {
        charType = CharacterType.Letter;
}
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||
          ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||
          ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||
          ( inputChar == '?' ) || ( inputChar == '-' ) ) {
        charType = CharacterType.Punctuation;
}
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {
        charType = CharacterType.Digit;
}
```

- **If you used a lookup table instead, you'd store the type of each character in an array that's accessed by character code. The complicated code fragment just shown would be replaced by this:**

```
charType = charTypeTable[ inputChar ];
```

Insurance Rates Example

```
if ( gender == Gender.Female ) {
 if (maritalStatus ==
           MaritalStatus.Single){
   if (smokingStatus ==
           SmokingStatus.NonSmoking) {

         if ( age < 18 ) {
            rate = 200.00;
         }

         else if ( age == 18 ) {
            rate = 250.00;
         }

         else if ( age == 19 ) {
            rate = 300.00;
         }

         ...

         else if ( 65 < age ) {
            rate = 450.00;
      }
```

```
 else {

   if ( age < 18 ) {
        rate = 250.00;
   }
   else if ( age == 18 ) {
        rate = 300.00;
   }
   else if ( age == 19 ) {
        rate = 350.00;
   }

   ...

   else if ( 65 < age ) {

        rate = 575.00;
   }
 }

 else if (maritalStatus ==
                 MaritalStatus.Married)

   ...

}
```

Insurance Rates Example

```
enum SmokingStatus {
    SmokingStatus_First = 0,
    SmokingStatus_Smoking = 0,
    SmokingStatus_NonSmoking = 1,
    SmokingStatus_Last = 1
}
enum Gender {
    Gender_First = 0,
    Gender_Male = 0,
    Gender_Female = 1,
    Gender_Last = 1,
}
```

```
enum MaritalStatus {
    MaritalStatus_First = 0,
    MaritalStatus_Single = 0,
    MaritalStatus_Married = 1,
    MaritalStatus_Last = 1
}

#define MAX_AGE = 125
Double rateTable = [
SmokingStatus_Last, Gender_Last,
MaritalStatus_Last, MAX_AGE ]
```

```
    rate = rateTable(smokingStatus, gender, maritalStatus, age)
```

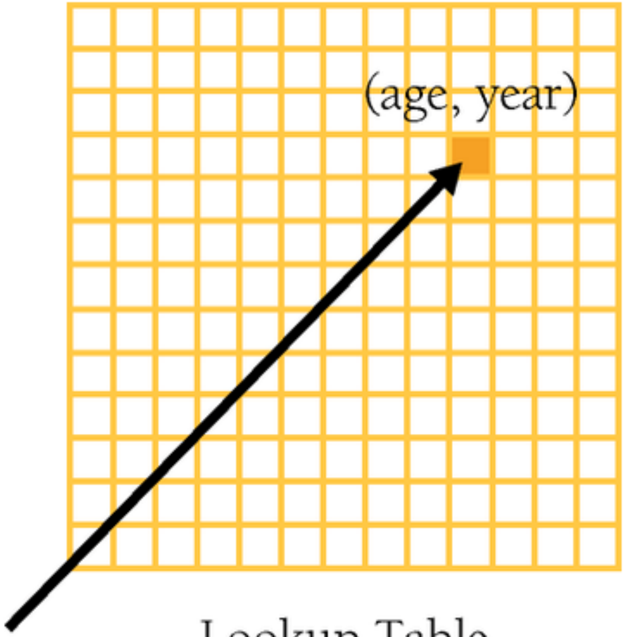
Methods of looking things up

- **Direct access**
- **Indexed access**
- **Stair-step access**

- **Selecting one of these depends on the nature of the data, and the size of the domain of the data.**

(1) Direct Access Tables

- **Simple – you just "look things up" by an index or indexes.**
  - Like all lookup tables, direct-access tables replace more complicated logical control structures.
  - They are "direct access" because you don't have to jump through any complicated hoops to find the information you want in the table.



Lookup Table

```
int daysPerMonth[ ] = { 31, 28,
31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };

days = daysPerMonth[month-1];
```

Insurance Rates Example

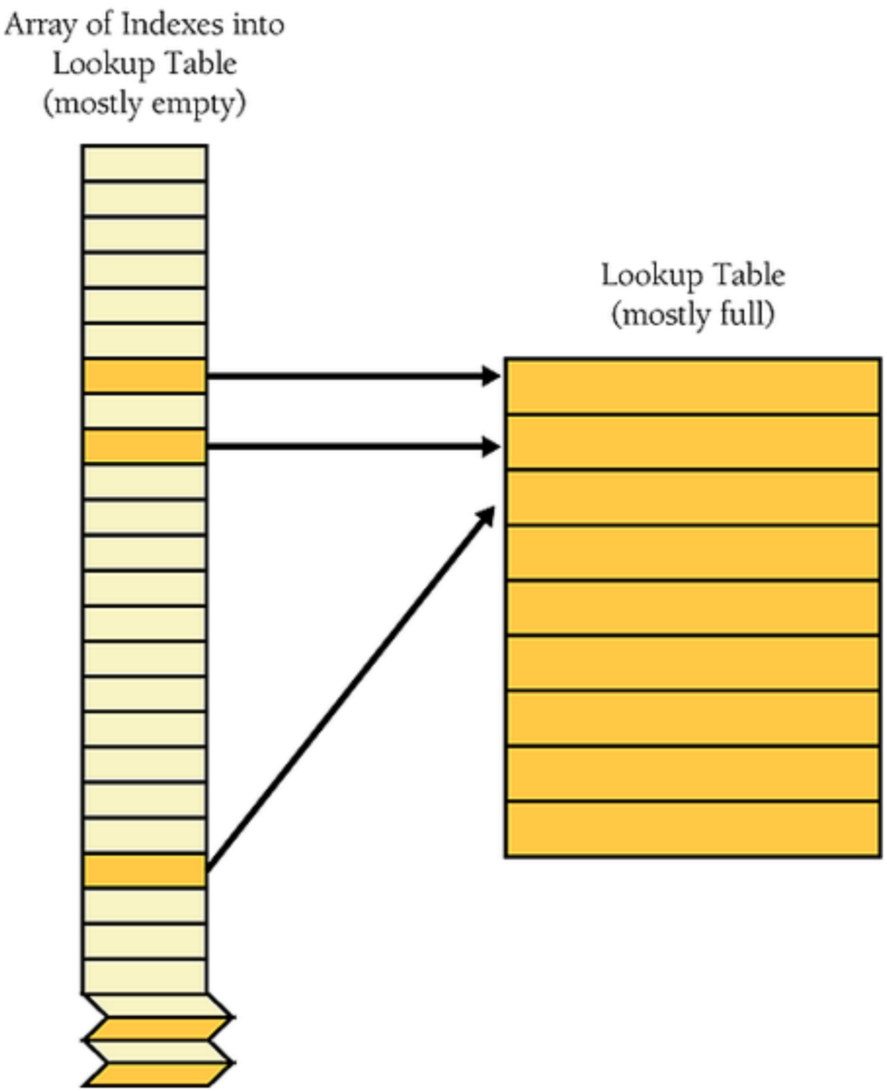
```
enum SmokingStatus {
    SmokingStatus_First = 0,
    SmokingStatus_Smoking = 0,
    SmokingStatus_NonSmoking = 1,
    SmokingStatus_Last = 1
}
enum Gender {
    Gender_First = 0,
    Gender_Male = 0,
    Gender_Female = 1,
    Gender_Last = 1,
}
```

```
enum MaritalStatus {
    MaritalStatus_First = 0,
    MaritalStatus_Single = 0,
    MaritalStatus_Married = 1,
    MaritalStatus_Last = 1
}

#define MAX_AGE = 125
Double rateTable = [
SmokingStatus_Last, Gender_Last,
MaritalStatus_Last, MAX_AGE ]
```

```
    rate = rateTable(smokingStatus, gender, maritalStatus, age)
```

(2) Indexed Access Tables

- Sometimes direct indexing is a problem, especially if the domain of possible values is huge.
- For example what if you wanted to use the product id (8 digits let's say), and make a table mapping 200 products.

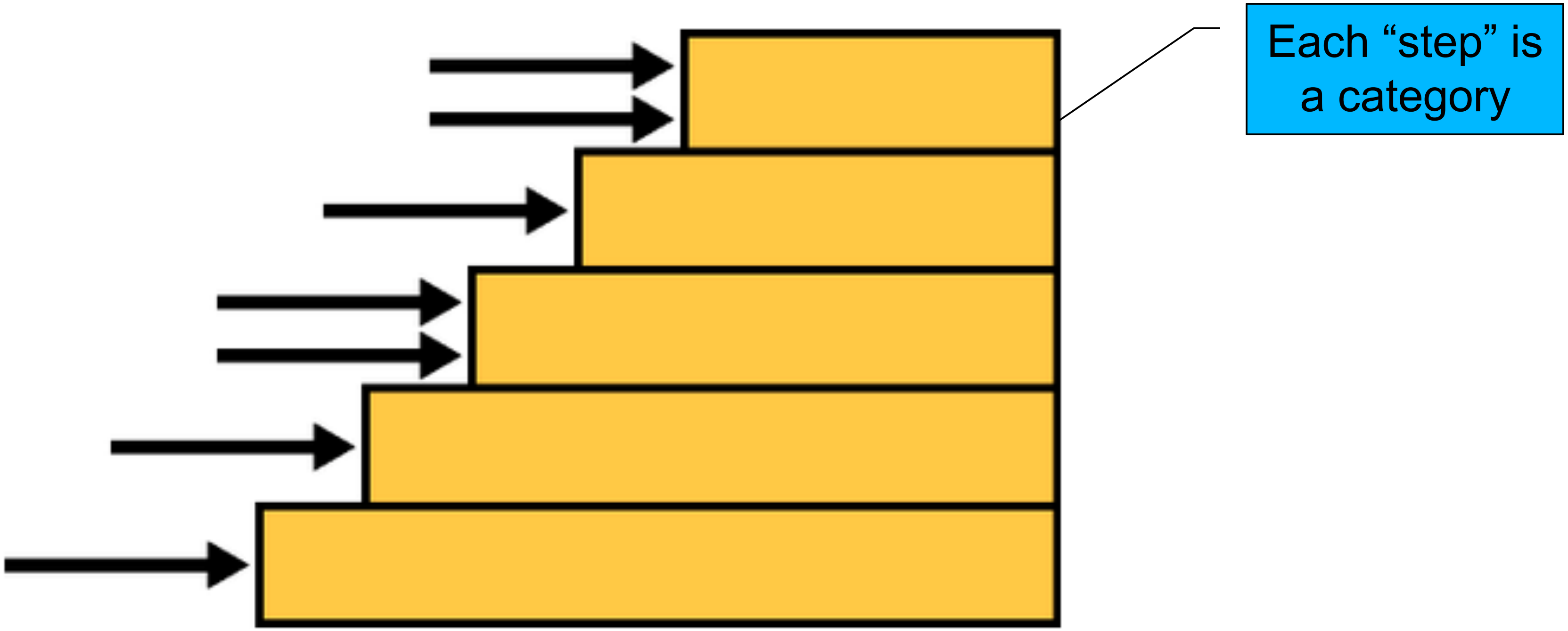

Array of Indexes into Lookup Table (mostly empty)
Lookup Table (mostly full)

Lookup Index vs Direct

- **Index elements are small (integers), values can efficiently be large (only as many as you need), such as a string (names, descriptions, error messages, etc).**
- **Multiple indexes can access the same data (employee info can be mapped to by name, hire date, salery, etc.)**
- **Maintainable – isolate lookup method from the application interface.**

(3) Stair-Step Access Tables

- **Entries in a table are valid for ranges of data rather than for distinct data points**



Each "step" is a category

Stair-Step Access Tables

Grade Table

Range	Grade
≥ 90.0%	A
< 90.0%	B
< 75.0%	C
< 65.0%	D
< 50.0%	F

Grade Lookup

```
// set up data for grading table
float rangeLimit[] = { 50.0, 65.0, 75.0, 90.0, 100.0 };
String grade[] = { "F", "D", "C", "B", "A" };
#define maxGradeLevel = sizeof(rangeLimit) / sizeof(rangeLimit[0])
...
// assign a grade to a student based on the student's score
int gradeLevel = 0;
string studentGrade = "A”;
While (( studentGrade == "A" ) and ( gradeLevel < maxGradeLevel )) {
   If ( studentScore < rangeLimit[ gradeLevel ] ) {
        studentGrade = grade[ gradeLevel ];
   }
   gradeLevel = gradeLevel + 1;
}
```

Statistics – Irregular Data

Probability	Insurance Claim Amount
0.458747	$0.00
0.547651	$254.32
0.627764	$514.77
0.776883	$747.82
0.893211	$1,042.65
0.957665	$5,887.55
0.976544	$12,836.98
0.987889	$27,234.12
…	

Key Points

- **Tables provide an alternative to complicated logic and inheritance structures. If you find that you're confused by a program's logic or inheritance tree, ask yourself whether you could simplify by using a lookup table.**
- **One key consideration in using a table is deciding how to access the table. You can access tables by using direct access, indexed access, or stair-step access.**
- **Another key consideration in using a table is deciding what exactly to put into the table.**

3 Grammar-based construction

语法驱动的构造

Objective of Grammar-based Construction

- Understand the ideas of grammar productions and regular expression operators
- Be able to read a grammar or regular expression and determine whether it matches a sequence of characters
- Be able to write a grammar or regular expression to match a set of character sequences and parse them into a data structure

String/Stream based I/O

- **Some program modules take input or produce output in the form of a sequence of bytes or a sequence of characters, which is called a *string* when it's simply stored in memory, or a *stream* when it flows into or out of a module. 有一类应用，从外部读取文本数据，在应用中做进一步处理。**
- **Concretely, a sequence of bytes or characters might be:**
  - A file on disk, in which case the specification is called the *file format* 输入文件有特定格式，程序需读取文件并从中抽取正确的内容
  - Messages sent over a network, in which case the specification is a *wire protocol* 从网络上传输过来的消息，遵循特定的协议
  - A command typed by the user on the console, in which case the specification is a *command line interface* 用户在命令行输入的指令，遵项特定的格式
  - A string stored in memory 内存中存储的字符串，也有格式需要

The notion of a grammar

- **For these kinds of sequences, the notion of a grammar is a good choice for design:**
  - It can not only help to distinguish between legal and illegal sequences, but also to parse a sequence into a data structure a program can work with. 使用grammar判断字符串是否合法，并解析成程序里使用的数据结构
  - The data structure produced from a grammar will often be a recursive data type. 通常是递归的数据结构
- **Regular expression 正则表达式**
  - It is a widely-used tool for many string-processing tasks that need to disassemble a string, extract information from it, or transform it.
- **A parser generator is a kind of tool that translate a grammar automatically into a parser for that grammar. 根据语法生成它的解析器，用于后续的解析**

(1) Constituents of a Grammar

Terminals: Literal Strings in a Grammar

- To describe a string of symbols, whether they are bytes, characters, or some other kind of symbol drawn from a fixed set, we use **a compact representation called a grammar.**
- **A grammar defines a set of strings.**
  - For example, the grammar for URLs will specify the set of strings that are legal URLs in the HTTP protocol.
- **The literal strings in a grammar are called terminals 终止节点、叶节点**
  - They're called terminals because they are the leaves of a parse tree that represents the structure of the string. 语法解析树的叶子节点
  - They don't have any children, and can't be expanded any further.
  - We generally write terminals in quotes, like `'http'` or `':'`. 无法向下扩展了、

Nonterminals and Productions in a Grammer

- **A grammar is described by a set of productions 产生式节点, where each production defines a nonterminal 非终止节点**
  - A nonterminal is like a variable that stands for a set of strings, and the production as the definition of that variable in terms of other variables (nonterminals), operators, and constants (terminals). 遵循特定规则，利用操作符、终止节点和其他非终止节点，构造新的字符串
  - Nonterminals are internal nodes of the tree representing a string.
- **A production in a grammar has the form**
  - nonterminal ::= expression of terminals, nonterminals, and operators
- **One of the nonterminals of the grammar is designated as the root.**
  - The set of strings that the grammar recognizes are the ones that match the root nonterminal.
  - This nonterminal is often called root or start. 根节点

(2) Operators in a Grammar

Three Basic Grammar Operators

- **The three most important operators in a production expression are:**
  - Concatenation, represented not by a symbol, but just a space:

    `x ::= y z`   an `x` is a `y` followed by a `z`

  - Repetition, represented by *:

    `x ::= y*`   an `x` is zero or more `y`

  - Union, also called alternation, represented by `|`:

    `x ::= y | z`   an `x` is a `y` or a `z`

Combinations of three basic operators

- **Additional operators are just syntactic sugar (i.e., they're equivalent to combinations of the big three operators):**
  - Optional (0 or 1 occurrence), represented by ?:

    `x ::=  y?`　　an x is a y or is the empty string

  - 1 or more occurrences: represented by +:

    `x ::= y+`　　an x is one or more y

    (equivalent to `x ::= y y*`)

    从中括号里选一个.

  - A character class [...], representing the length-1 strings containing any of the characters listed in the square brackets:

    `x ::= [abc]` is equivalent to `x ::= 'a' | 'b' | 'c‘`

    ^→除了

  - An inverted character class [^...], representing the length-1 strings containing any character not listed in the brackets:

    `x ::= [^abc]` is equivalent to `x ::= 'd' | 'e' | 'f' | ...`
    (all other characters in Unicode)

Grouping operators using parentheses

- By convention, the postfix operators `*`, `?`, and `+` have highest precedence, which means they are applied first.
- Concatenation is applied next.
- Alternation `|` has lowest precedence, which means it is applied last.

- Parentheses can be used to override precedence:
  - `x ::=  (y z | a b)*` an `x` is zero or more `y-z` or `a-b` pairs
  - `m ::=  a (b|c) d` an `m` is `a`, followed by either `b` or `c`, followed by `d`

abd; acd

Exercise

- **Consider this grammar:**

  S ::= (B C)* T

  B ::= M+ | P B P

  C ::= B | E+

- **What are the nonterminals in this grammar?** → S, B, C
- **What are the terminals in this grammar?** T, M, P, E.
- **Which productions are recursive?** B

Root → S

Exercise

- **Which strings match the root nonterminal of this grammar?**

```
root ::= 'a'+ 'b'* 'c'?
```

↙ 1个 ↓ 0个 ↓ 0∨1个C

- **Strings**

aabcc x c多了

bbbc x a没有

aaaaaaaa ✓

abc ✓

abab x 开始了b就不会有a了.

aac ✓

Exercise

- **Which strings match the root nonterminal of this grammar?**

```
root    ::= integer ('-' integer)+
        integer ::= [0-9]+
```

(- integer) 一次或多次出现.

[0~9] 1次以上.

- **Strings**

617 ✗

617-253 ✓

617-253-1000 ✓

--- ✗

integer-integer-integer ✗

5--5 ✗

3-6-293-1 ✓

Exercise

- **Which strings match the root nonterminal of this grammar?**

```
root    ::= (A B)+ 一次或多个.
A       ::= [Aa]
B       ::= [Bb]
```

- **Strings**

```
aaaBBB ✗
abababab ✓
aBAbabAB ✓
AbAbAbA ✗
```

(3) Example 1: URL

URL as an example

- **To write a grammar that represents URLs.**
- **Here's a simple URL:**

```
http://mit.edu/
```

- **A grammar that represents the set of strings containing only this URL would look like:**

```
url ::= 'http://mit.edu/'
```

- **But let's generalize it to capture other domains, as well:**

```
http://stanford.edu/
http://google.com/
```

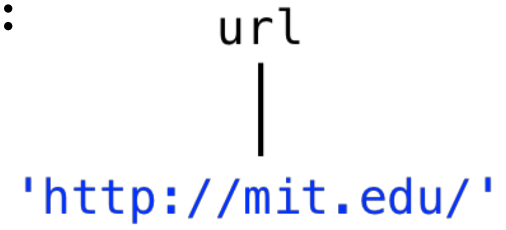
- **We can write this as one line, like this:**

```
url ::= 'http://' [a-z]+ '.' [a-z]+  '/'
```

URL as an example

```
url ::= 'http://' [a-z]+ '.' [a-z]+  '/'
```

- **This grammar represents the set of all URLs that consist of just a two-part hostname, where each part of the hostname consists of 1 or more letters.**
- **So** `http://mit.edu/` **and** `http://yahoo.com/` **would match, but not** `http://ou812.com/`
- **Since it has only one nonterminal, a parse tree for this URL grammar would look like this:**



url

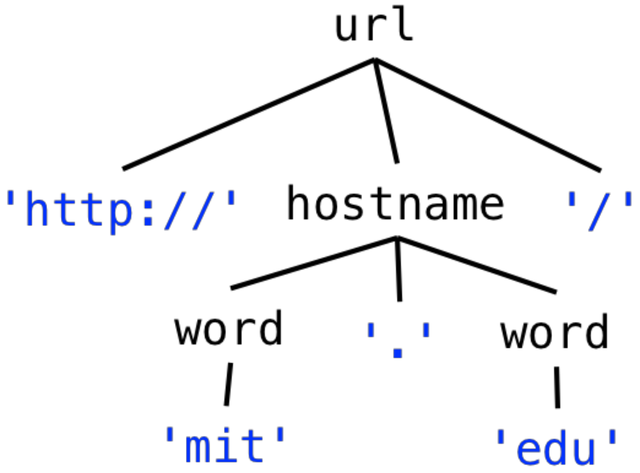
'http://mit.edu/'

- **In this one-line form, with a single nonterminal whose production uses only operators and terminals, a grammar is called a regular expression.**

Grammars with multiple nonterminals

- It will be easier to understand if we name the parts using new nonterminals:

```
url ::= 'http://' hostname '/'

hostname ::= word '.' word

word ::= [a-z]+
```



- **The leaves of the tree are the parts of the string that have been parsed.**
  - Concatenating the leaves together, we would recover the original string.
  - The `hostname` and word `nonterminals` are labeling nodes of the tree whose subtrees match those rules in the grammar.
  - The immediate children of a nonterminal node like `hostname` follow the pattern of the `hostname` rule, `word '.' word`.

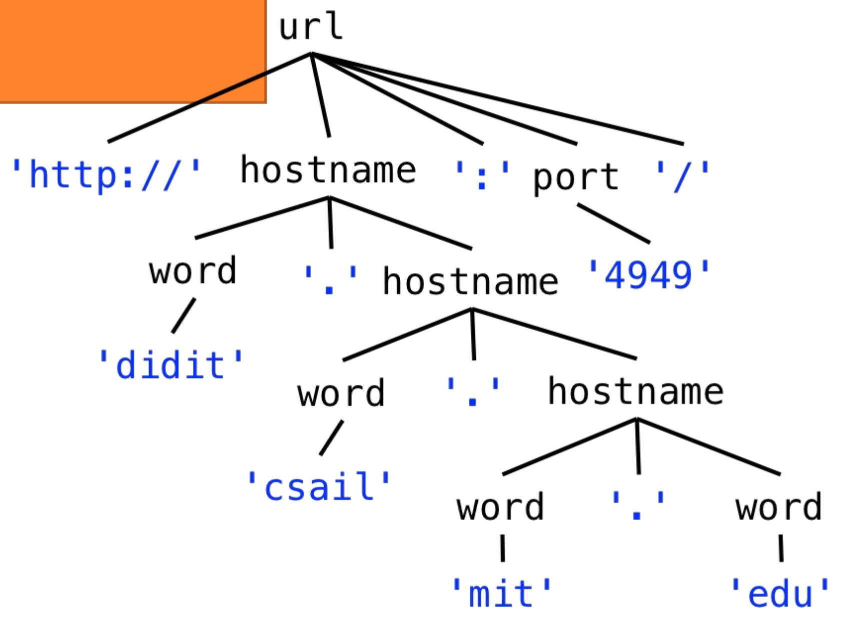
Again generalizing…

- **Hostnames can have more than two components, and there can be an optional port number:**

```
http://didit.csail.mit.edu:4949/
```

- **To handle this kind of string, the grammar is now:**

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```



- `hostname` **is now defined recursively in terms of itself.**
- **Another way:**

```
hostname ::= (word '.')+ word
```

More generalizations…

- **There are more things we should do to go farther:**
  - Generalizing `http` to support the additional protocols that URLs can have, such as `ftp`, `https`, …
  - Generalizing the `/` at the end to a slash-separated path, such as `http://didit.csail.mit.edu:4949/homework/lab1/`
  - Allowing hostnames with the full set of legal characters instead of just `a-z` such as `http://ou812.com/`

- **Can you do these?**

```
url ::= pptotype   hostname   (':'  port )? '/' (word '/') *
prototype ::= ['http://'  'ftp://'  'https://']
hostname ::= (host word . )+ host word.
word     ::= [a-z]+
host word ::= [a-z 0- 9]+
```

Exercise

- **We want the URL grammar to also match strings of the form:**
  - https://websis.mit.edu/
  - ftp://ftp.athena.mit.edu/
- **but not strings of the form:**
  - ptth://web.mit.edu/
  - mailto:bitdiddle@mit.edu
- **So we change the grammar to:**

```
url ::= protocol '://' hostname (':'
        port)? '/'
protocol ::= TODO
hostname ::= word '.' hostname |
             word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

- **What could you put in place of TODO to match the desirable URLs but not the undesirable ones?**
  - `word`
  - `'ftp' | 'http' | 'https'` ✓
  - `('http' 's'?) | 'ftp'` ✓
  - `('f' | 'ht') 'tp' 's'?` → ftps

(4) Example 2: Markdown and HTML

Markdown and HTML

- **Markdown**

```
This is _italic_.
```

- **HTML**

```
Here is an <i>italic</i> word.
```

- **For simplicity, these HTML and Markdown grammars will only specify italics, but other text styles are of course possible.**
- **For simplicity, we will assume the plain text between the formatting delimiters isn't allowed to use any formatting punctuation, like _ or <.**

HTML ::= (normal | text)*

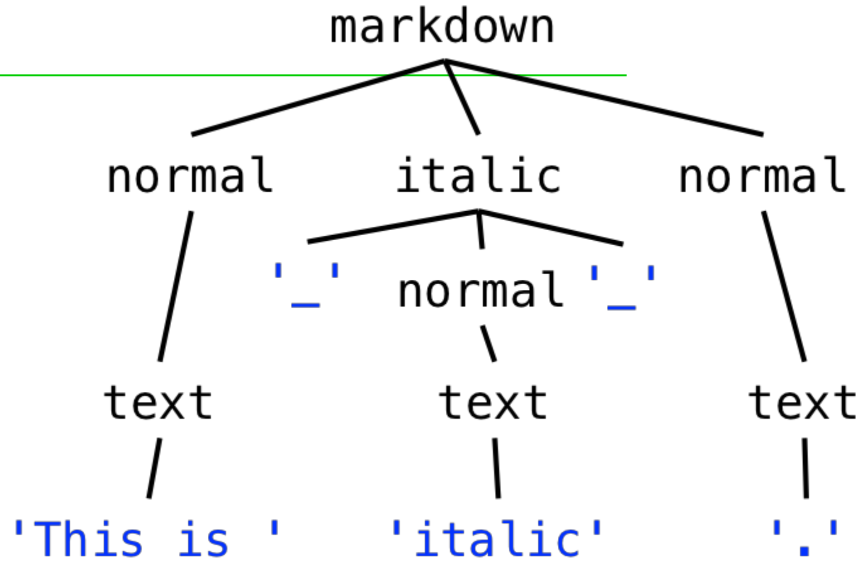
text::= <i> ~~normal~~ HTML </i>

normal::= [^<>]*

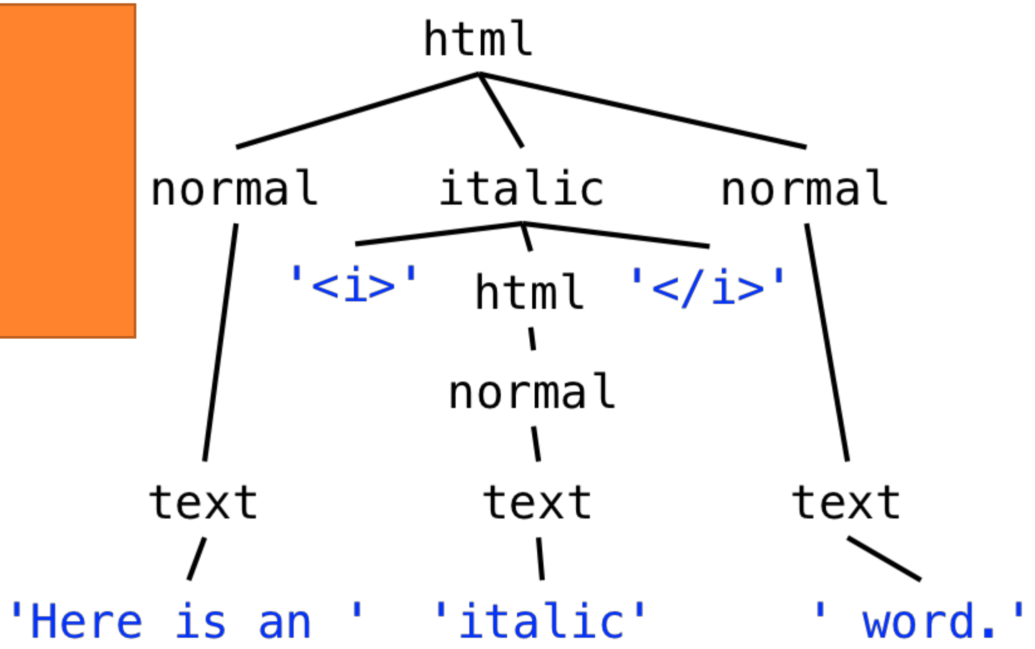
- **Can you write down their grammars?**

Markdown and HTML

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```



```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```



Markdown and HTML

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```

```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```

markdown:

`a_b_c_d_e`

html:

`a<i>b<i>c</i>d</i>e`

If you match the specified grammar against it, which letters are inside matches to the italic nonterminal?

(5) Regular Grammars and Regular Expressions

Regular grammar

根节点可以转化为叶子结点的组合,

- **A regular grammar** has a special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side. 正则语法：简化之后可以表达为一个产生式而不包含任何非终止节点

- Which of them are regular grammars?

```
url ::= 'http://' hostname (':' port)? '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

```
markdown ::=  ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```

```
html ::=  ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```

Regular grammar

```
url ::= 'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

Regular!

```
markdown ::= ([^_]* | '_' [^_]* '_' )*
```

Regular!

```
html ::=  ( [^<>]* | '<i>' html '</i>' )*
```

Not regular!

Regular Expressions (*regex*)

- **The reduced expression of terminals and operators can be written in an even more compact form, called a regular expression. 正则表达式**
- **A regular expression does away with the quotes around the terminals, and the spaces between terminals and operators, so that it consists just of terminal characters, parentheses for grouping, and operator characters. 去除引号和空格，从而表达更简洁(更难懂)**

```
markdown ::= ([^_]* | '_' [^_]* '_' )*
```

```
markdown ::= ([^_]*|_[^_]*_)*
```

- **Regular expressions are also called *regex* for short.**
  - A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression.
  - But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions.

Some special operators in regex

- `.` any single character
- `\d` any digit, same as [0-9]
- `\s` any whitespace character, including space, tab, newline
- `\w` any word character, including letters and digits
- `\., \(, \), \*, \+, ...`

  escapes an operator or special character so that it matches literally

An example

- **Original:**

```
'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

- **Compact:**

```
http://([a-z]+.)+[a-z]+(:[0-9]+)?/
```

转义.

- **With escape:**

```
http://([a-z]+\.)+[a-z]+(:[0-9]+)?/
```

Exercise

- **Consider the following regular expression:**

```
[A-G]+(♭|♯)?
```

- **Which of the following strings match the regular expression?**
  - A♭ ✓
  - C♯ ✓
  - ABK♭ ✗
  - A♭B ✗
  - GFE ✓

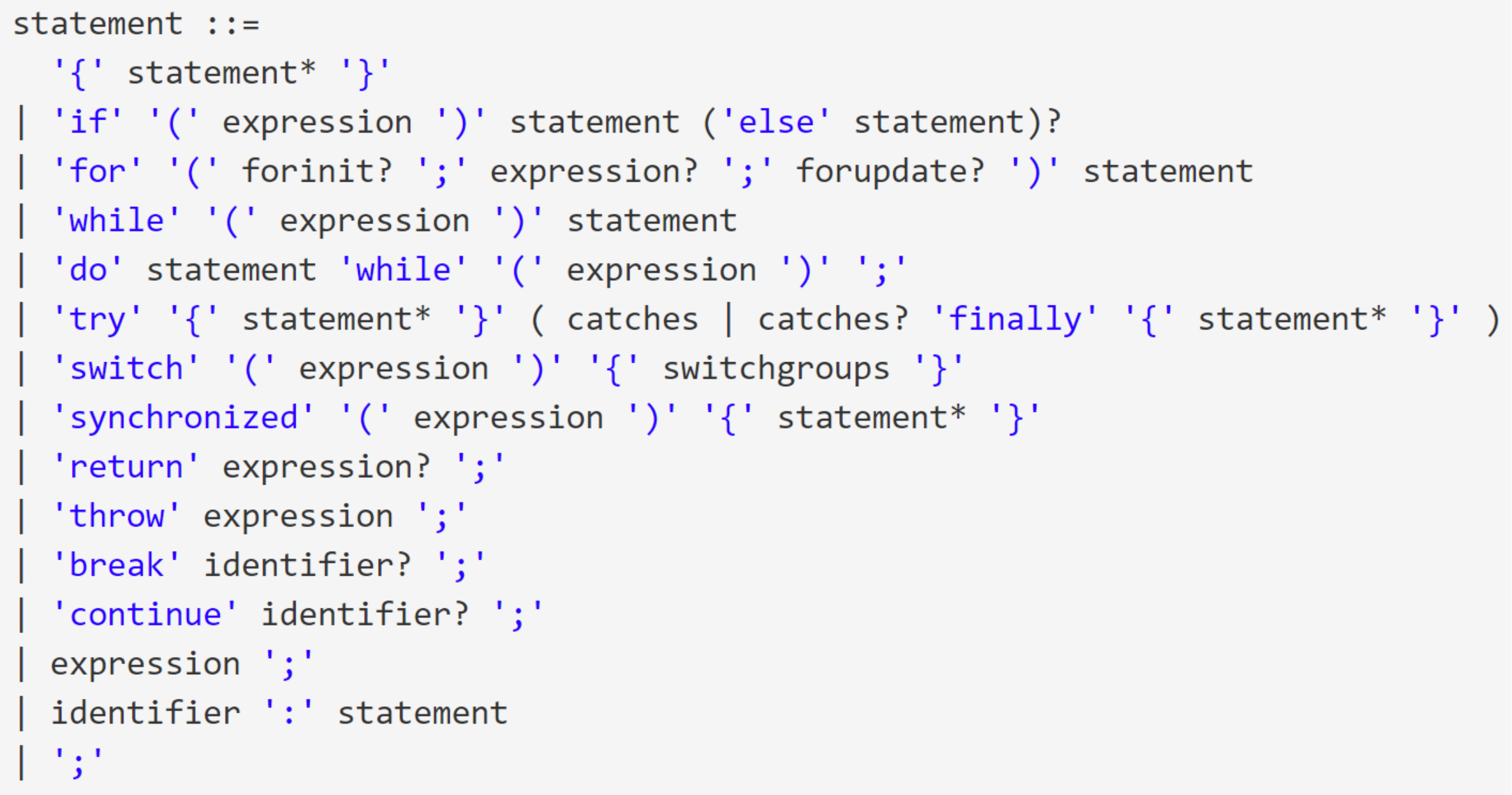
Context-Free Grammars

- In general, a language that can be expressed with our system of grammars is called context-free.
  - Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonrecursive productions.
  - The HTML grammar is context-free but not regular.

- The grammars for most programming languages are also context-free.
- In general, any language with nested structure (like nesting parentheses or braces) is context-free but not regular.

课程《形式语言与自动机》

Java grammar



```
statement ::=
  '{' statement* '}'
| 'if' '(' expression ')' statement ('else' statement)?
| 'for' '(' forinit? ';' expression? ';' forupdate? ')' statement
| 'while' '(' expression ')' statement
| 'do' statement 'while' '(' expression ')' ';'
| 'try' '{' statement* '}' ( catches | catches? 'finally' '{' statement* '}' )
| 'switch' '(' expression ')' '{' switchgroups '}'
| 'synchronized' '(' expression ')' '{' statement* '}'
| 'return' expression? ';'
| 'throw' expression ';'
| 'break' identifier? ';'
| 'continue' identifier? ';'
| expression ';'
| identifier ':' statement
| ';'
```

(6) * Parsers

Grammar, Parser, and Parser Generator

- **Objectives:**
  - Be able to use a grammar in combination with a parser generator, to parse a character sequence into a parse tree
  - Be able to convert a parse tree into a useful data type

Parser 将输入文本转为parse tree

- **A parser takes a sequence of characters and tries to match the sequence against the grammar. parser：输入一段文本，与特定的语法规则建立匹配，输出结果**
- **The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence. parser文本转化为parse tree**
  - The root of the parse tree is the starting nonterminal of the grammar.
  - Each node of the parse tree expands into one production of the grammar.
- **The final step of parsing is to do something useful with this parse tree. 利用产生的parse tree，进行下一步的处理**
- **A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).**

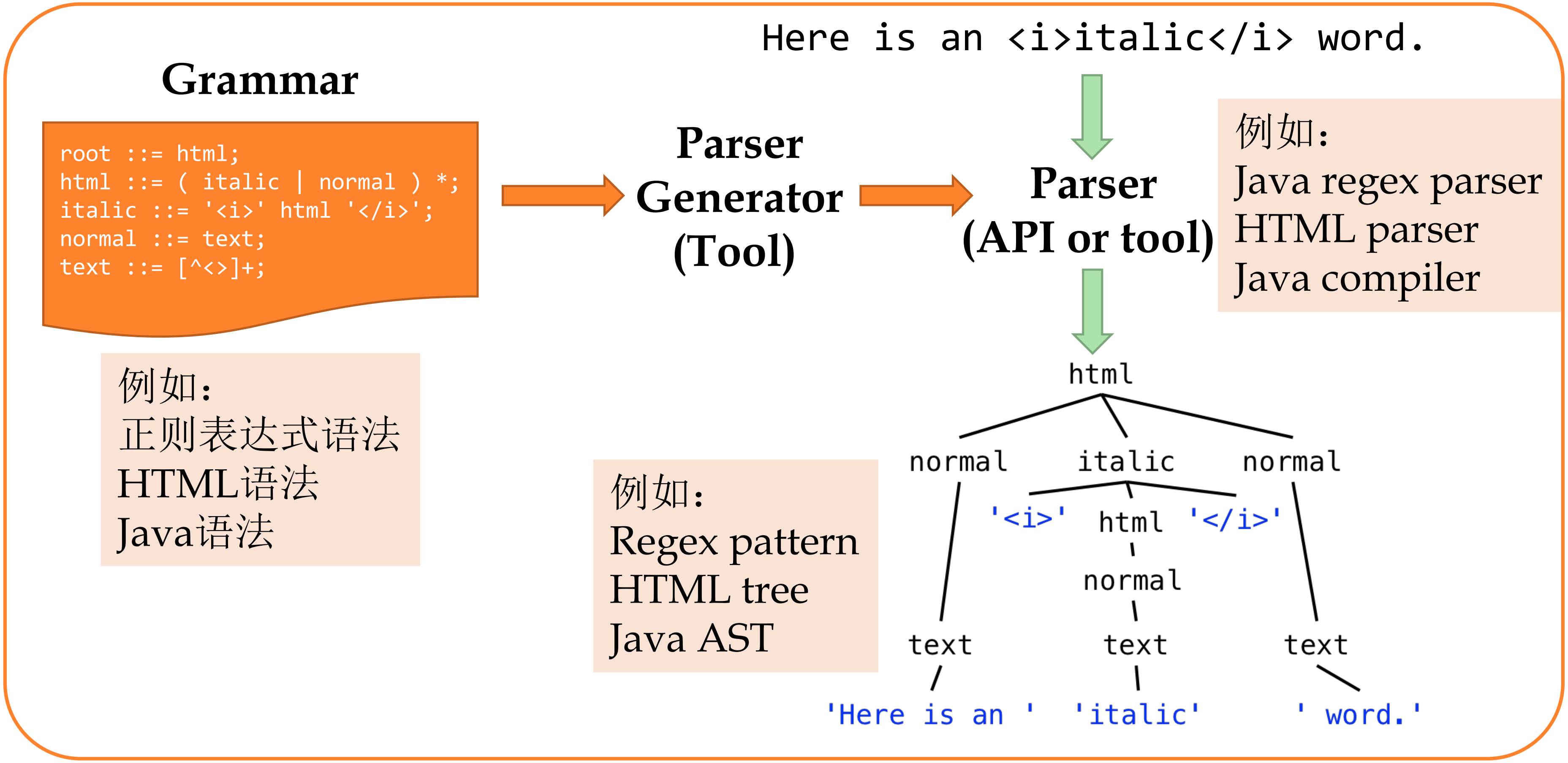
Parser Generator 根据语法定义生成parser

- **A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.**
  - Read http://web.mit.edu/6.031/www/sp17/classes/18-parsers
    This is not the mandatory contents of this course.

- **More broadly:**
  - A parser generator is a programming tool that creates a parser, interpreter, or compiler from some form of formal description of a language.
  - The input may be a text file containing the grammar written in BNF or EBNF that defines the syntax of a programming language.
  - The output is some source code of the parser for the grammar.

Grammar, Parser Generator, and Parser

- **Grammar定义语法规则，Parser generator根据grammar定义的规则产生一个parser，client利用parser来解析文本，看其是否符合语法定义并对其做各种处理（例如转成parse tree）**



Grammar

```
root ::= html;
html ::= ( italic | normal ) *;
italic ::= '<i>' html '</i>';
normal ::= text;
text ::= [^<>]+;
```

例如：
正则表达式语法
HTML语法
Java语法

Parser Generator (Tool)

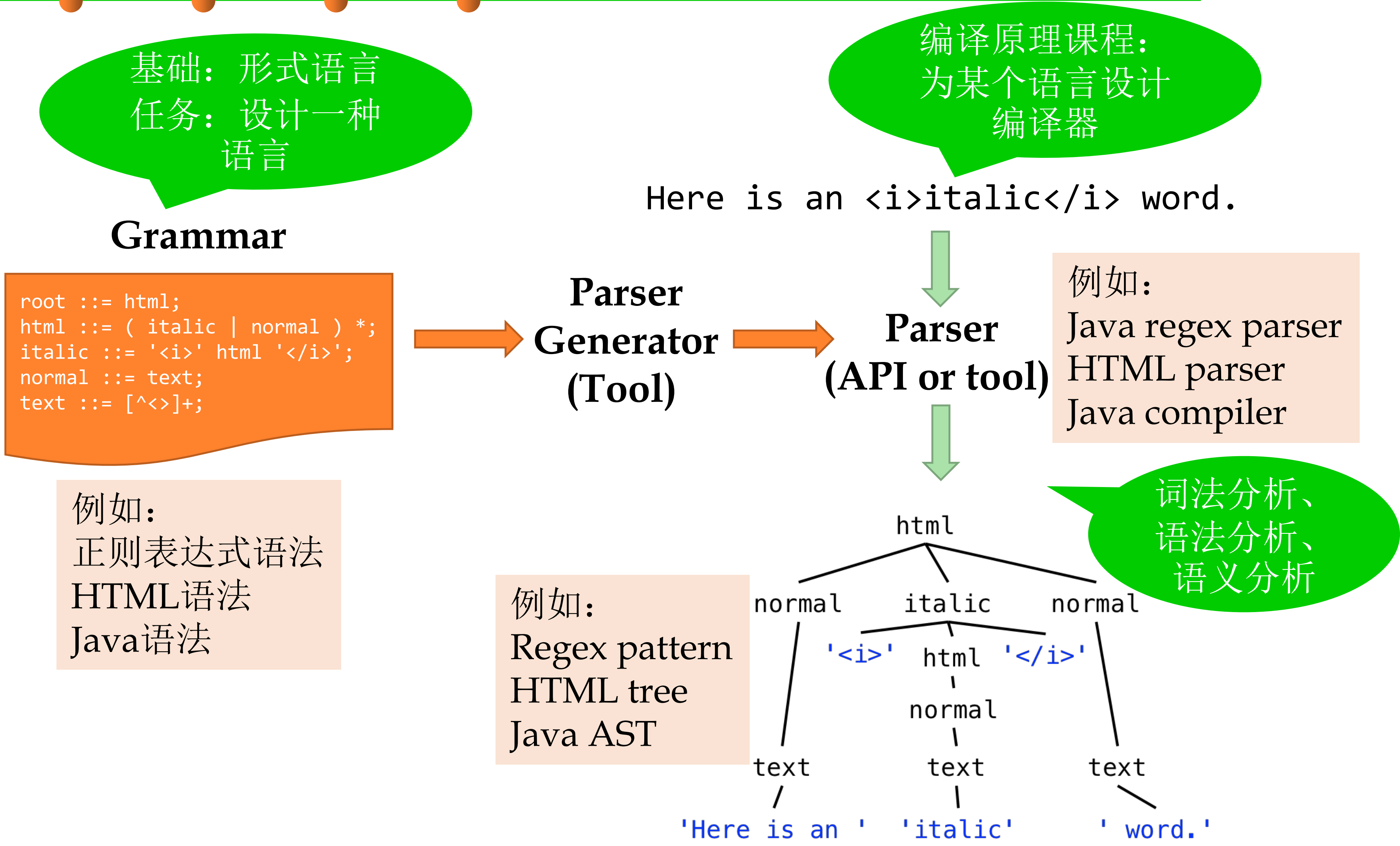
`Here is an <i>italic</i> word.`

Parser (API or tool)

例如：
Java regex parser
HTML parser
Java compiler

例如：
Regex pattern
HTML tree
Java AST

In your future study…



基础：形式语言
任务：设计一种语言

Grammar

```
root ::= html;
html ::= ( italic | normal ) *;
italic ::= '<i>' html '</i>';
normal ::= text;
text ::= [^<>]+;
```

例如：
正则表达式语法
HTML语法
Java语法

Parser Generator (Tool)

编译原理课程：
为某个语言设计编译器

```
Here is an <i>italic</i> word.
```

Parser (API or tool)

例如：
Java regex parser
HTML parser
Java compiler

词法分析、
语法分析、
语义分析

例如：
Regex pattern
HTML tree
Java AST

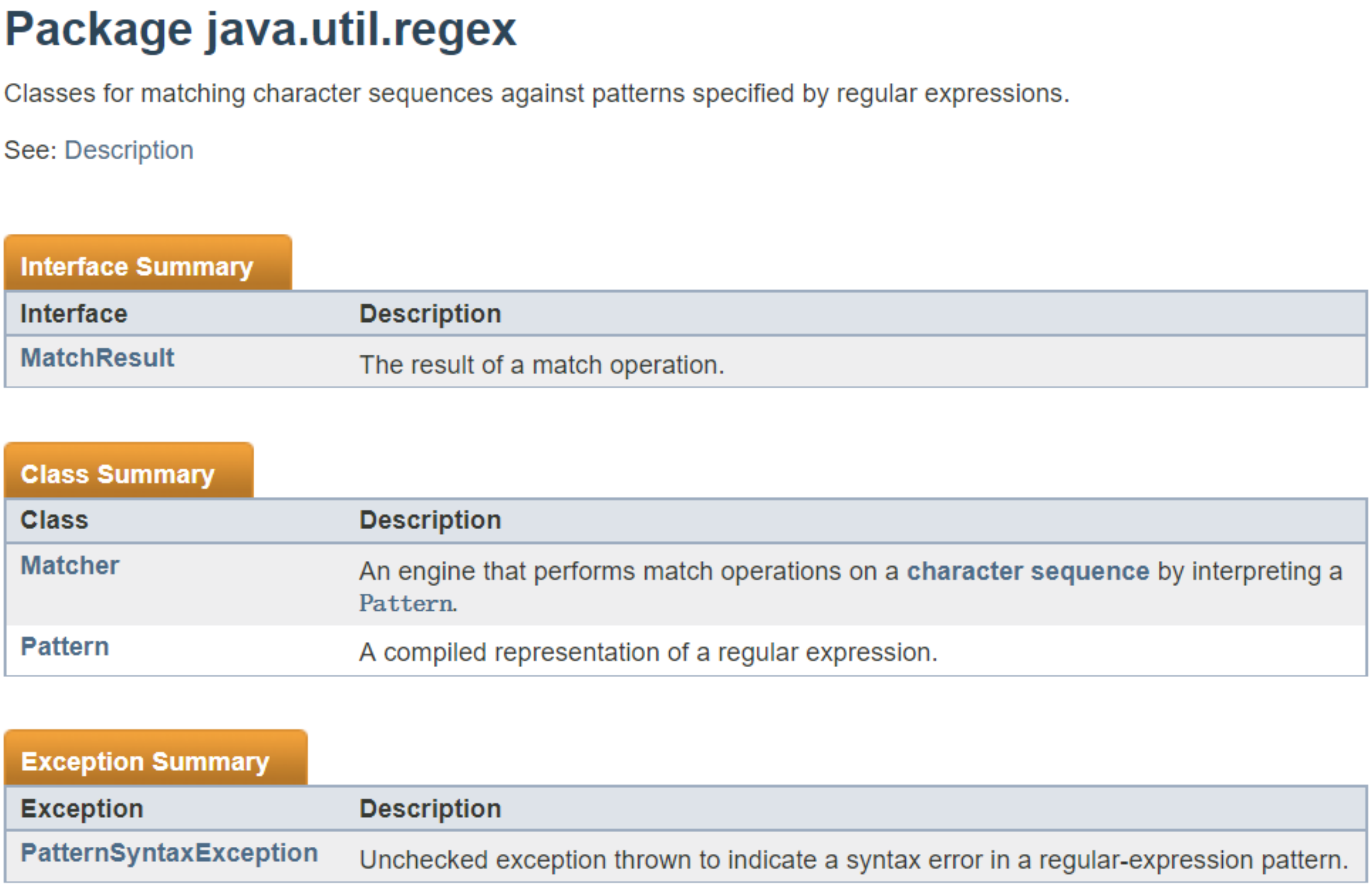
(7) Using regular expressions in Java

在本课程里，只需要能够熟练掌握正则表达式regex这种“基本语言”，并熟练使用其parser进行数据处理即可

java.util.regex for Regex processing

- **The `java.util.regex` package primarily consists of three classes:**
  - A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument.
  - A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the matcher method on a `Pattern` object.
  - A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.

`java.util.regex` for Regex processing



Package java.util.regex

Classes for matching character sequences against patterns specified by regular expressions.

See: Description

Interface Summary

Interface	Description
MatchResult	The result of a match operation.

Class Summary

Class	Description
Matcher	An engine that performs match operations on a **character sequence** by interpreting a `Pattern`.
Pattern	A compiled representation of a regular expression.

Exception Summary

Exception	Description
PatternSyntaxException	Unchecked exception thrown to indicate a syntax error in a regular-expression pattern.

Using regular expressions in Java

- **Regex is very useful in programming languages.**
  - In Java, you can use regexes for manipulating strings (such as `String.split`, `String.matches`, `java.util.regex.Pattern`).
  - They're built-in as a first-class feature of modern scripting languages like Python, Ruby, and Javascript, and you can use them in many text editors for find and replace.
- **Replace all runs of spaces with a single space:**

```
String singleSpacedString = string.replaceAll(" +", " ");
```

- **Match a URL:**

```
Pattern regex =
     Pattern.compile("http://([a-z]+\\.)+[a-z]+(:[0-9]+)?/");
Matcher m = regex.matcher(string);
if (m.matches()) { // then string is a url   }
```

Using regular expressions in Java

- **Extract part of an HTML tag:**

```
Pattern regex = Pattern.compile("<a href=\"([^\"]*)\">");
Matcher m = regex.matcher(string);
if (m.matches()) {
    String url = m.group(1);
    // Matcher.group(n) returns the nth parenthesized part of
    // the regex
}
```

- **The frequency of backslash escapes makes regexes still less readable!!!**

An example

- **Write the shortest regex you can to remove single-word, lowercase-letter-only HTML tags from a string:**

```
String input = "The <b>Good</b>, the <i>Bad</i>, and the
             <strong>Ugly</strong>";
String regex = "TODO";
String output = input.replaceAll(regex, "");
```

- **If the desired output is "The Good, the Bad, and the Ugly", what is shortest regex you can put in place of TODO?**

```
</?[a-z]+>
```

Character Classes

Construct	Description
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z, or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

Metacharacters: `<([{\^-=$!|]})?*+.>`

Two ways to force a metacharacter to be treated as an ordinary character:

- Precede the metacharacter with a backslash, or
- Enclose it within \Q (which starts the quote) and \E (which ends it).

Predefined Character Classes

Construct	Description
`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

Quantifiers

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

Boundary Matchers

Boundary Construct	Description
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Pattern method equivalents in `java.lang.String`

- `public boolean matches(String regex)`: **Tells whether or not this string matches the given regular expression.** `Pattern.matches(regex, str).`

- `public String[] split(String regex, int limit)`: **Splits this string around matches of the given regular expression.** `Pattern.compile(regex).split(str, n)`

- `public String[] split(String regex)`: **Splits this string around matches of the given regular expression.**

- `public String replace(CharSequence target,CharSequence replacement)`

Learn by yourself

- https://docs.oracle.com/javase/tutorial/essential/regex/index.html

Exercise

- **Write the shortest regex you can to remove single-word, lowercase-letter-only HTML tags from a string:**

```
String input = "The <b>Good</b>, the <i>Bad</i>, and the
<strong>Ugly</strong>";
String regex = "TODO";
String output = input.replaceAll(regex, "");
```

- **If the desired output is "The Good, the Bad, and the Ugly", what is shortest regex you can put in place of TODO?**

(8)* Interpreter

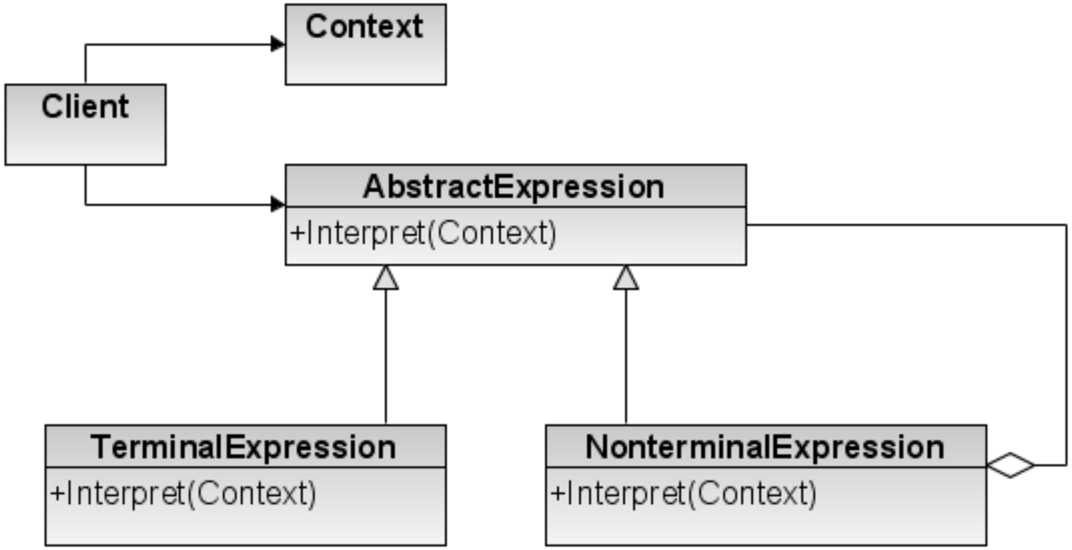
解释器模式

Interpreter

- **Interpreter pattern provides a way to evaluate language grammar or expression.**
- **Intent**
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. 给定一种语法，定义该语法的程序内部表示，形成该语法的解释器，将遵循语法规则的文本解释成程序内部的表示(例如一组object)
  - Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design
    解释语法的“引擎”：遵循语法的文本→OO表示
  - It is used exhaustively in defining grammars, tokenize input and store it.

Interpreter

- **Implementation**
  - It the use of the composite pattern applied to represent a grammar. 因为语法通常形成树结构，故使用composite模式来表达遵循语法的内容。
  - It defines the behavior while the composite defines only the structure. 针对该层次化树形结构，定义了一组行为来处理结构中的不同类型节点



Interpreter vs. grammar + parser

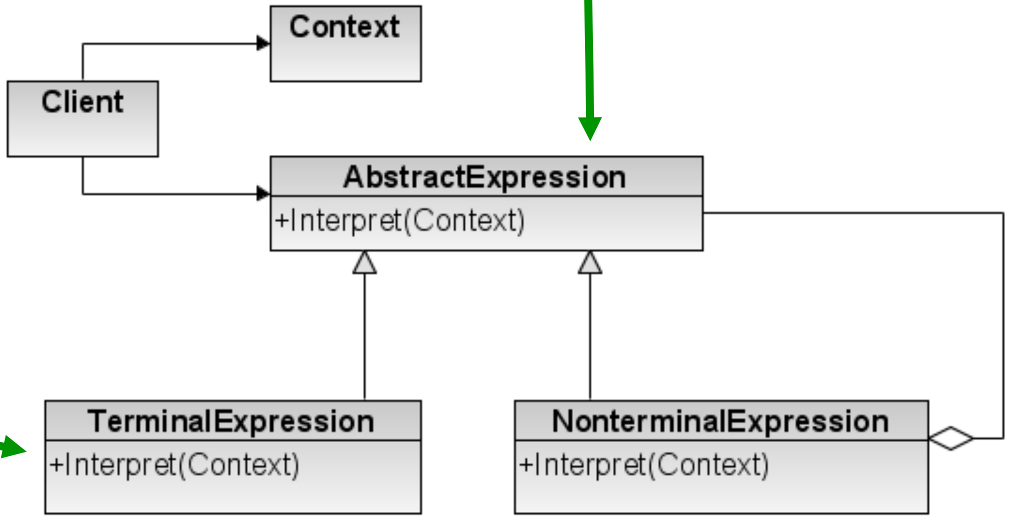
- **Grammar + Parser**
  - Grammar由BNF等形式定义
  - Parser读取用户输入的待解析的文本，判定其是否与grammar匹配，并转为符合grammar的parse tree，交给其他功能做后续处理
  - 可用于高度复杂的grammar rules

- **Interpreter pattern**
  - Grammar由程序员手工定义为一组interface/class及其之间的关系，对grammar的解释（即parser）由这组class的内部操作(`Interpret()`)负责
  - 每条语法规则(production)都要定义相应的类——相当于开发一个简单的parser
  - 用户使用的时候，调用这组类完成对输入文本的解释（这里的“解释”，其实相当于“翻译”）
  - 只适用于简单的grammar rules，过于复杂的grammar就需要引入大量的类

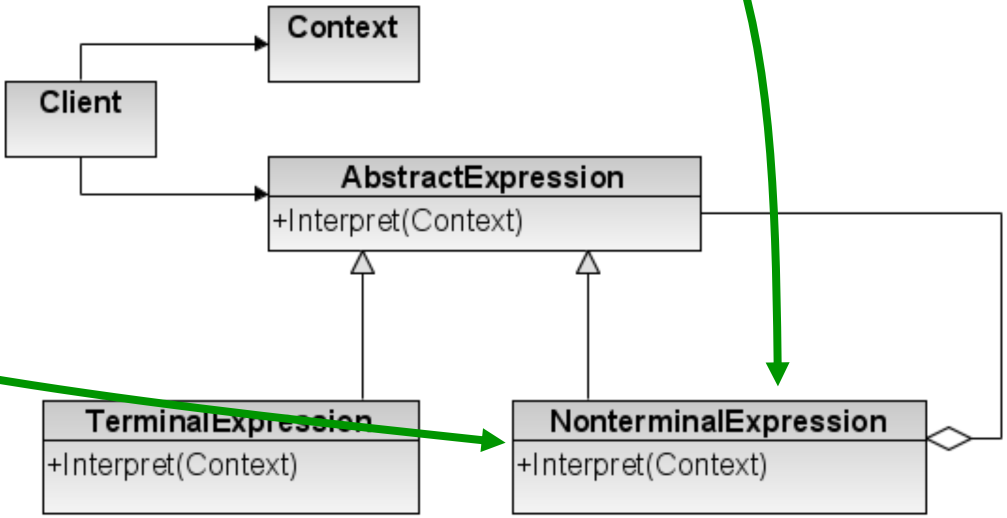
An example

- **Grammar**
  - `root ::= exp`
  - `exp ::= text | andtext | ortext`
  - `andtext ::= ‘(‘ text ‘AND’ exp ‘)’`
  - `ortext ::= ‘(‘ text ‘OR’ exp ‘)’`
  - `text ::= [a-zA-Z]+`
- **Legals:**
  - `(A AND (B OR (C OR D)))`
  - `(A AND B) AND ( (C OR D) AND E )`
- **Input: A B C，判断是否符合特定的语法规则**

```
public abstract class Expression {
    abstract public boolean interpret(String str);
}

public class TerminalExpression extends Expression {
    private String literal = null;
    public TerminalExpression(String str) {
        literal = str;
    }
    public boolean interpret(String str) {
        StringTokenizer st = new StringTokenizer(str);
        while (st.hasMoreTokens()) {
            String test = st.nextToken();
            if (test.equals(literal)) {return true;}
        }
        return false;
    }

}
```



```
public class OrExpression extends Expression{
    private Expression expression1 = null;
    private Expression expression2 = null;
    public OrExpression(Expression expression1, Expression expression2) {
        this.expression1 = expression1;
        this.expression2 = expression2;
    }
    public boolean interpret(String str) {
        return expression1.interpret(str) || expression2.interpret(str);
    }
}
public class AndExpression extends Expression{
    public boolean interpret(String str) {
        return expression1.interpret(str)
            && expression2.interpret(str);
    }
}
```



Interpreter

Here defines the rule "Owen and (John or (Henry or Mary))"

```
 static Expression buildInterpreterTree() {

        Expression terminal1 = new TerminalExpression("John");
        Expression terminal2 = new TerminalExpression("Henry");
        Expression terminal3 = new TerminalExpression("Mary");
        Expression terminal4 = new TerminalExpression("Owen");

        Expression alternation1 = new OrExpression(terminal2, terminal3);
        Expression alternation2 = new OrExpression(terminal1, alternation1);

        return new AndExpression(terminal4, alternation2);
}

public static void main(String[] args) {
        String context = "Mary Owen";
        Expression define = buildInterpreterTree();
        System.out.println(context + " is " + define.interpret(context));
}
```

Summary

Summary

- Machine-processed textual languages are ubiquitous in computer science.
- Grammars are the most popular formalism for describing such languages
- Regular expressions are an important subclass of grammars that can be expressed without recursion.

Summary

- **Safe from bugs**
  - Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and tools.
  - These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand**
  - A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code.
  - Regular expressions, alas, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change**
  - A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.

The end

April 13, 2018