哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 6: Maintainability-Oriented Software Construction Approaches

6.3 Maintainability-Oriented Construction Techniques

Ming Liu

April 15, 2018

Outline

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

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.**
- For example, in the case of an elevator, it could be 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.
  - If the elevator has just closed its doors, what are the possibilities that can happen next? It can either move up, or move down. You wouldn't expect the elevator to stop after closing its doors.
  - When an elevator stops, you expect the next action to be the doors opening.

The code

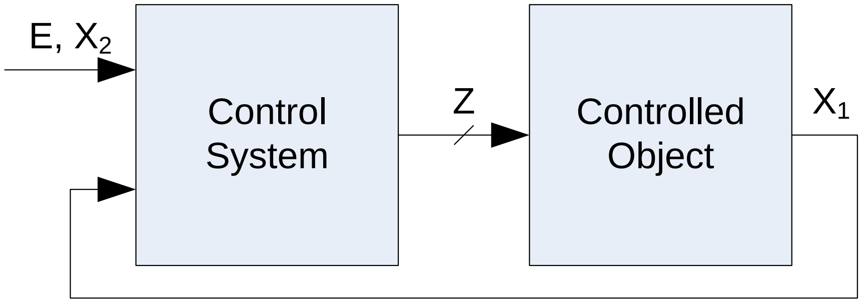
```java
public enum ElevatorState {
    OPEN, CLOSED, MOVING_UP, MOVING_DOWN, STOP
}
```

```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).**



Automata-based programming

- **The time period of the program's execution is clearly separated down to the *steps of the automaton*.**
  - 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*.**
  - 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.
- **The whole execution of the automata-based code is a (possibly explicit) cycle of the automaton's steps.**

How to implement?

- **The "state" variable can be a simple enum data type, but more complex data structures may be used.**
  - 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.
  - 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 }
    };
```

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

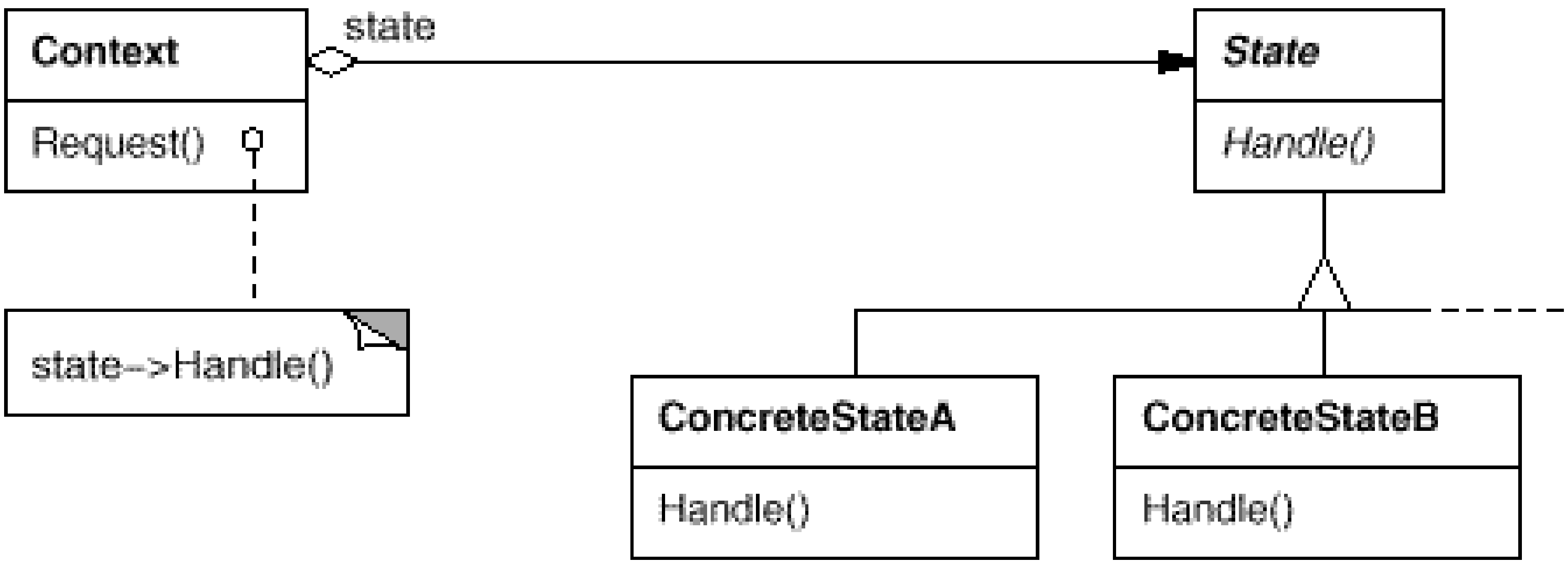
State pattern

- **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**

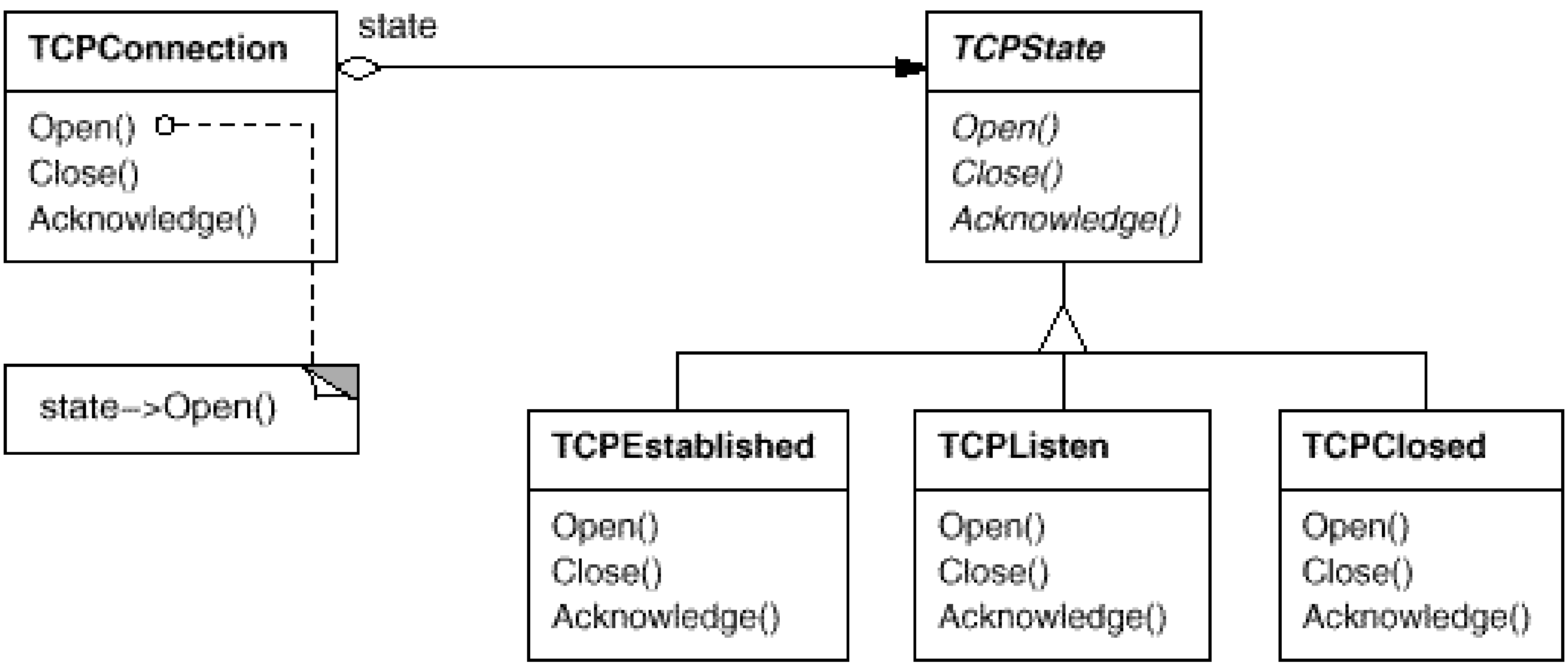
State pattern

- **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



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
- **Easy to add new states**
  - New states can extend other states
    - Override only selected functions

Example – Finite State Machine

```
class FSM {
    State state;
    public FSM(State s) { state = s;  }
    public void move(char c) {  state = state.move(c); }
    public boolean accept() {  return state.accept();}
  }

  public interface State {
      State move(char c);
      boolean accept();
  }
```

FSM Example – cont.

```
class State1 implements State {
  static State1 instance = new State1();
  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;}
}
```

```
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;}
}
```

```
public class Client {
    public static void main(String[] args) {
        FSM fsm =new FSM(state1.instance); //start with state1
        fsm.move("a");  //change to state2
        fsm.move("a");  //change to state1
        fsm.move("b");  //change to state1
        fsm.move("b");  //change to state1
        ….
    }
}
```

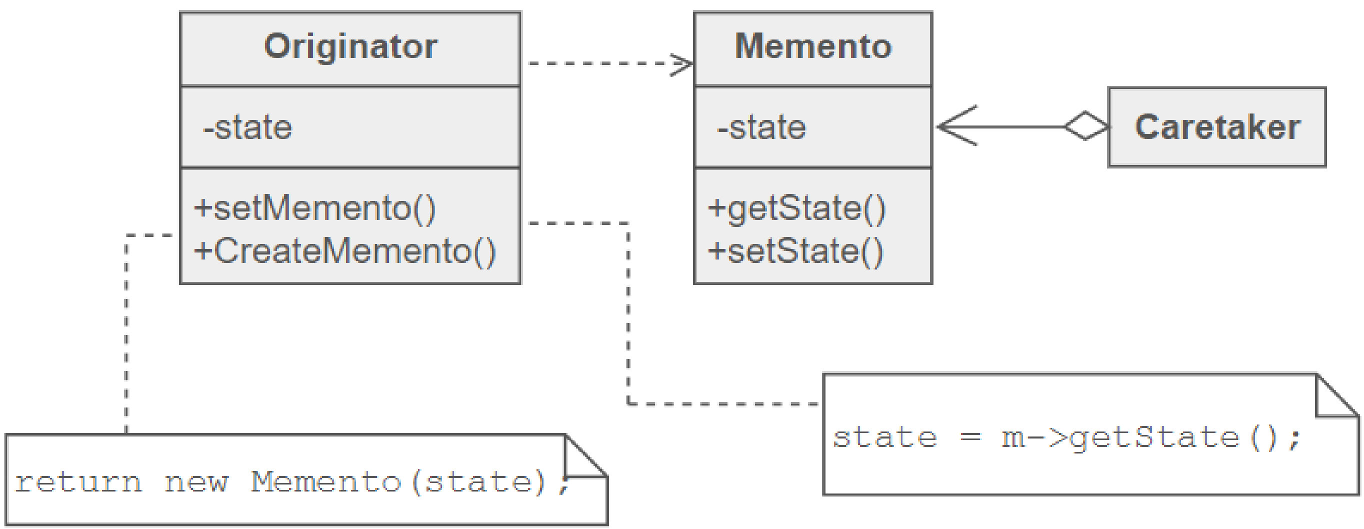
(3) Memento Pattern

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).

Memento Pattern

- **Memento design pattern defines three distinct roles:**
  - *Originator* - the object that knows how to save itself.
  - *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
  - *Memento* - the lock box that is written and read by the Originator, and shepherded by the Caretaker.



Memento Pattern

```
class Memento {
    private String state;

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

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

```
class Originator {
    private String state;

    public void setState(String state) {
        System.out.println("Originator: Setting state to " + state);
        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);
    }
}
```

```
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() );
    }
}
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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 and 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

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)
```

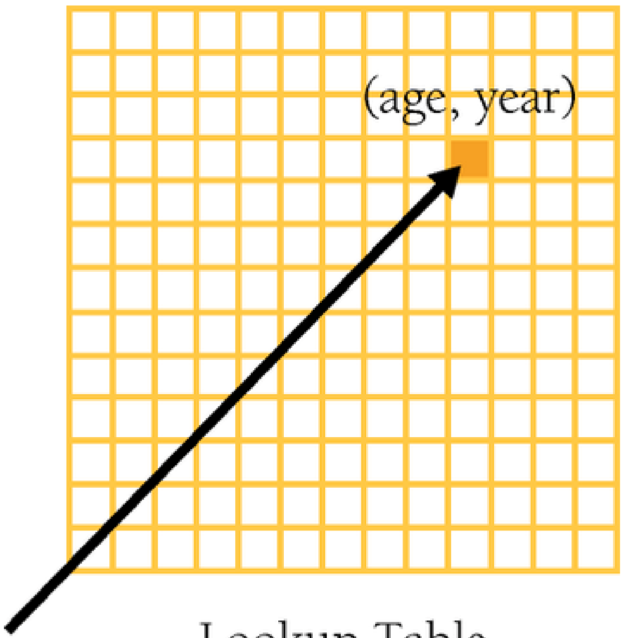
Methods of looking things up

- **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,
}
```

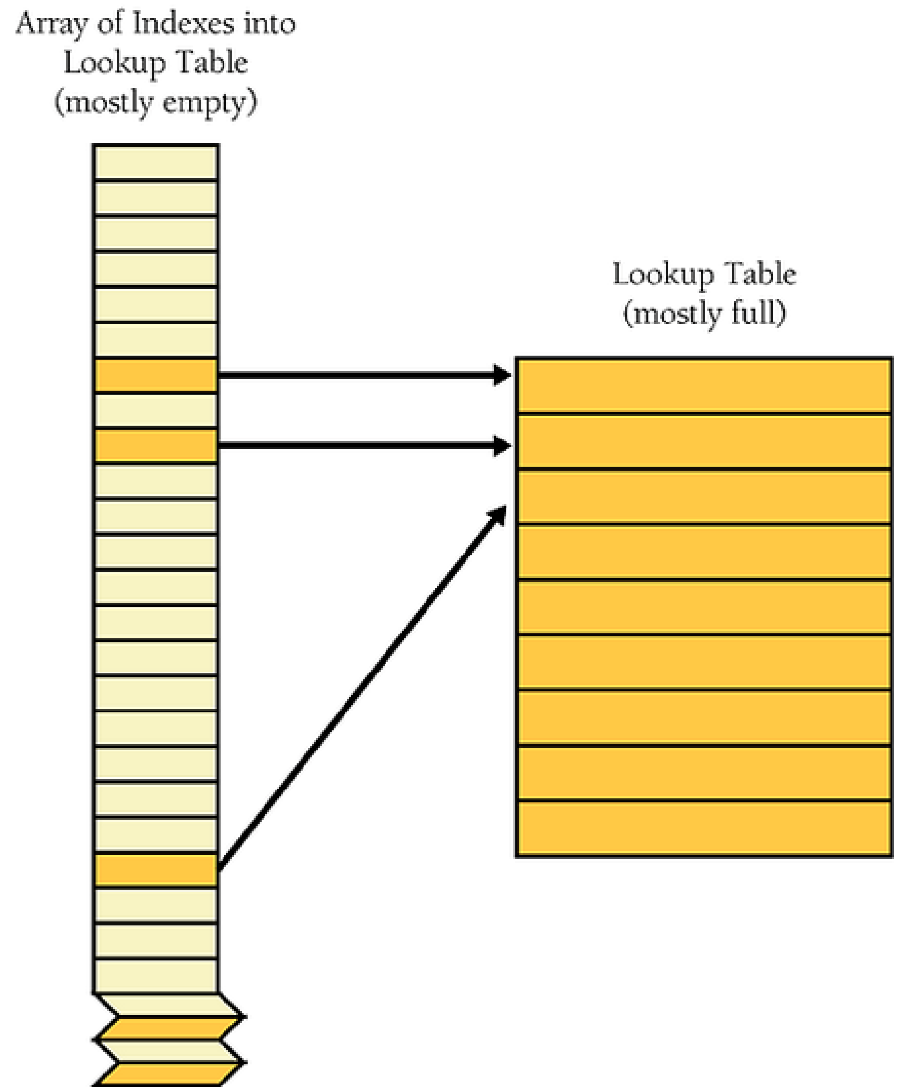
```
enum MaritalStatus {
   MaritalStatus_First = 0,
   MaritalStatus_Single = 0,
   MaritalStatus_Married = 1,
   MaritalStatus_Last = 1
}
```

```
#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.**

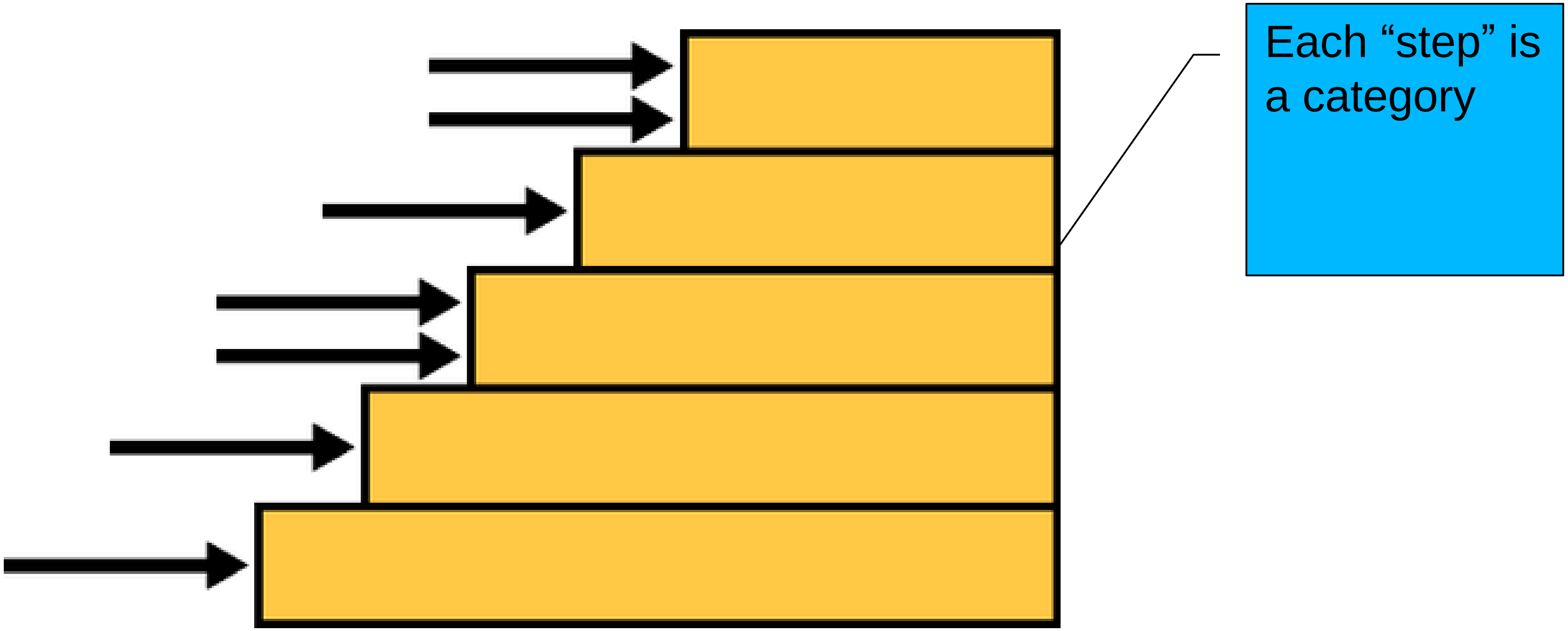


Lookup Index vs Direct

- **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**



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

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
- 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

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.**
- **Concretely, a sequence of bytes or characters might be:**
  - A file on disk, in which case the specification is called the *file format*
  - Messages sent over a network, in which case the specification is a *wire protocol*
  - A command typed by the user on the console, in which case the specification is a *command line interface*
  - A string stored in memory

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.
  - The data structure produced from a grammar will often be a recursive data type.
- **A specialized form of a grammar called a regular expression.**
  - 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.**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(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.**
  - They're called terminals because they are the leaves of a parse tree that represents the structure of the string.
  - They don't have any children, and can't be expanded any further.
  - We generally write terminals in quotes, like 'http' or ':'.

Nonterminals and Productions in a Grammer

- **A grammar is described by a set of productions, where each production defines a nonterminal.**
  - 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).
  - 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.

(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*`)

  - 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'`

  - 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 yz or ab pairs
  - `m ::= a (b|c) d` an `m` is `a`, followed by either `b` or `c`, followed by `d`

(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:**

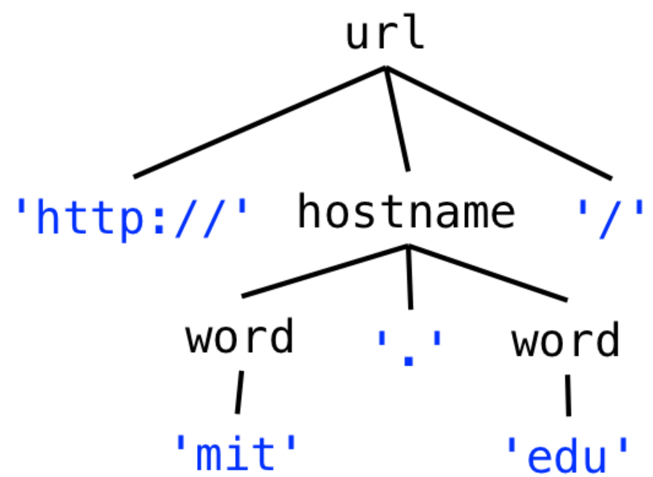
```
       url
        |
'http://mit.edu/'
```

- **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…

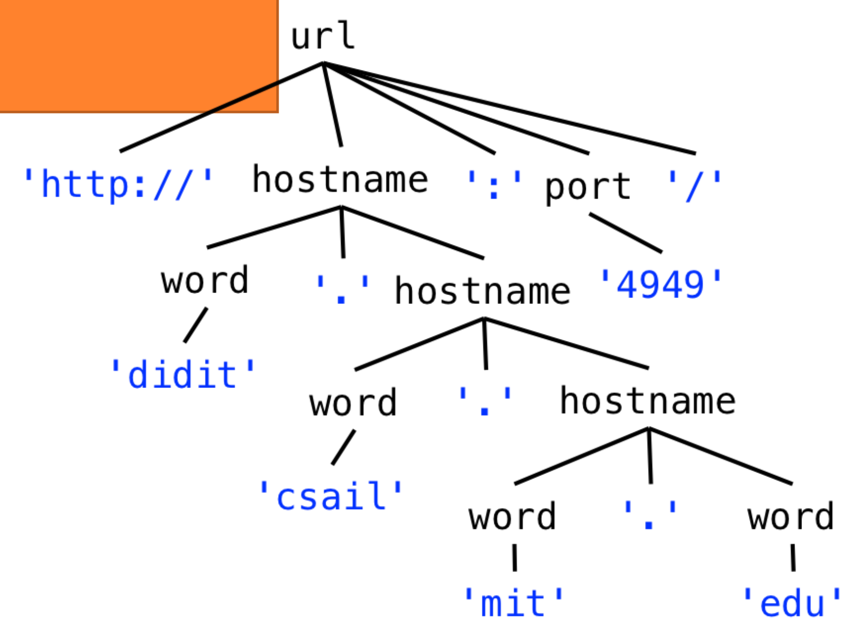
- **Hostnames can have more than two components, and there can be an optional port number:**

```
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
  - Generalizing the / at the end to a slash-separated path
  - Allowing hostnames with the full set of legal characters instead of just a-z

- **Can you do these?**

(4) Example 2: Markdown and HTML

Markdown and HTML

- **Markdown**

```
This is _italic_.
```

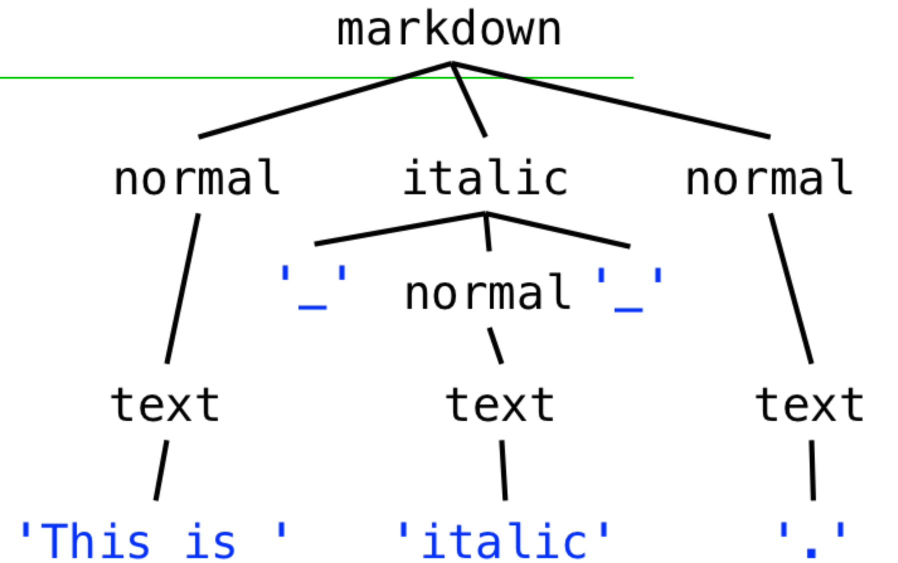
- **HTML**

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

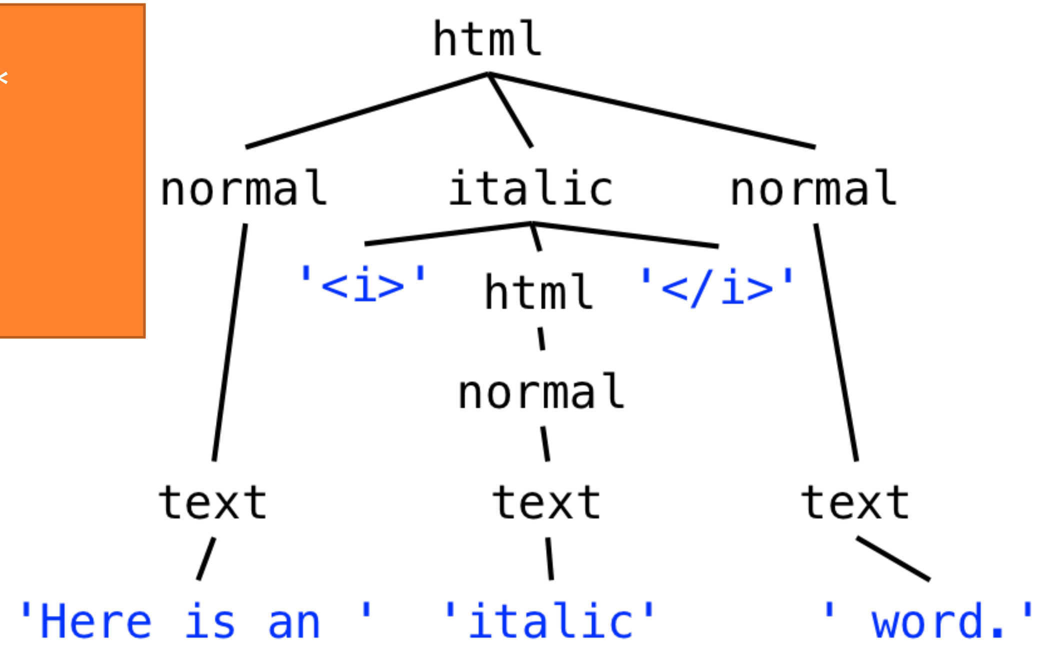
- **For simplicity, these HTML and Markdown grammars will only specify italics, but other text styles are of course possible.**
- **For simplicity, we will assume the plain text between the formatting delimiters isn't allowed to use any formatting punctuation, like _ or <.**
- **Can you write down their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

Markdown and HTML

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




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



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(5) Regular Grammars and Regular Expressions

Regular grammar

- **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.
- 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.**
- **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.**

```
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]+)?/
```

- **With escape:**

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

(6) Using regular expressions in Java

Using regular expressions in Java

- **Regex is very useful in programming languages.**
  - In Java, you can use regexes for manipulating strings (see `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]+>
```

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.

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
| ';'
```

(7) Parsers

Parser

- **A parser takes a sequence of characters and tries to match the sequence against the grammar.**
- **The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence.**
  - 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.
- **A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).**

Summary

Parser Generator

- **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.**

The end

April 15, 2018