Lesson 5 - Declarative Programming

CSC330 Language Implementation and Design

Lesson 5: Imperative Programming

Lesson Objectives

By the end of this lesson, you will be able to:

- ✓ Use a declarative programming language.
- ✓ Describe declarative programming.
- ✓ Describe the use of *finite state machines* in the implementation of regular expressions.
- ✓ Analyze choices in syntax and structure in declarative programming.

Describe declarative programming.

Unlike imperative programming, **declarative programming** tries to solve a problem by focusing on what the solution should look like rather than the steps required to find the solution. It may sound confusing, but through the use of **formal logic**, declarative programming can perform calculations by using a set of defined rules to produce an output.

For example, hypertext markup language (HTML) is a declarative syntax that purely represents the markup of data presented to a Web browser. HTML, by itself, does not describe *how* a Web browser will render the content to users; instead, it simply provides markup descriptions about *what* data the browser should present. In combination with other languages, such as Cascading Style Sheets (CSS) and JavaScript, a Web browser will compute and render text and images to the screen. In this example, HTML does the *what* while CSS and JavaScript are responsible for the *how*.

Another example, called **regular expressions**, is a programming language built for string recognition. The syntax of the language is a special set of characters and symbols that can very quickly calculate whether a string exists within a body of text. Let's take a look at a few examples.

To learn more about the definition of a regular expression, visit the Wikipedia page, "Regular expression."

Grep

In these examples, we'll use a command-line utility called *grep*. Grep was originally built for Unix, but it is now available for all operating systems. The program accepts a body of text and a regular expression. It will then match the expression against the text and return any matches of the expression within the text. This type of program is called a **regular expression engine**. By default, grep will return the entire line of text in which the regular expression is matched, but the "-o" switch can be included for grep to only return the actual matching string.

For example, suppose we want to use grep to look for the word *dog* in the sentence "A quick, brown fox jumps over the lazy dog." We can use the sentence as the parameter in the command, and grep confirms a positive match for containing the string *dog*[1]:

```
$> echo A quick brown fox jumps over the lazy dog | grep -o 'dog'
dog
```

This might sound very simple, but we made a declarative statement that said our *solution* should look like *dog* without telling grep *how* to find the word *dog* within the sentence.

Another example:

```
$> echo A quick brown fox jumps over the lazy dog | grep -o '[0-9]'
output is blank
```

In this example, we wanted to find examples of a single digit (0–9) in the sentence. Of course, the sentence doesn't have any numbers, so the output is blank. The code "[0–9]" says to search for a single digit, with the range of 0–9.

Using Regular Expressions

Let's look at a more complex example of a creative search:

```
$ echo A quick brown fox jumps over the lazy dog | grep -oE "(\w*[aeiouAEIOU]\w*)"
A
quick
brown
fox
jumps
over
the
lazy
dog
```

This example is a little more complex. The regular expression is: (\w*[aeiouAEIOU]\w*), which it searches for strings that contain alphanumeric characters and at least one vowel within the string. Note how this search matches every single word because they all have at least one vowel.

In these examples, there are no control flow statements, no loops, no conditional statements, and no functions or procedures. Again, with declarative programming, the idea is to logically prove a solution through rules rather than apply a series of instructions on how to accomplish a task.

Regular expressions are incredibly useful in any task involving bodies of text, and most high-level programming languages have native regular expression libraries, most often included in their standard libraries.

Check yourself: Declarative Programming

1. Which of the following best describes the focus of declarative programming?

[1] Grep for Windows is available at "Grep for Windows." Mac users can use the terminal utility in Mac OS. Anyone can use the online regular expression engines included in the assignment instructions for this lesson.

a. Declarative programming focuses on the end result of the program.
b. Declarative programming focuses on the steps required to solve the problem.
c. Declarative programming focuses on the underlying subroutines that the code will use to solve the problem.

Correct Answer: Declarative programming languages allow developers to focus on what the program should do rather than on how the tasks are completed.

Describe the use of finite state machines in the implementation of regular expressions.

Regular expression engines typically use a computational model called **finite state machines** to implement an efficient and accurate algorithm to search text against an expression. A finite state machine models a machine that contains many states. Between each state is a transition, which can cause the current state of the machine to move to another state based on a specific condition or event.

Finite state machines are very useful in logic-based systems, including declarative programs and the software used in many common devices. Vending machines, elevators, combination locks, and traffic signals are all examples of finite state machines that have only one active state and require a certain event or condition to move to another state.

For example, a finite state machine for a vending machine sits in a "waiting" state for a customer to insert coins. Upon the event of a customer inserting a coin, the vending machine moves into a "waiting for correct change" state. The customer continues to drop in coins until he or she meets the condition of correct change. The vending machine then moves into a "waiting for customer selection" state, which waits for customer input. The customer selects a drink, which is the event that triggers the machine to go into a "disposing of drink" state and then eventually back to the "waiting" state.

Finite State Machines Explained

The following video explains how a finite state machine can be used with regular expressions.

Regular Expressions and Finite State Machines

Let's talk about regular expressions as they relate to a finite state machine:

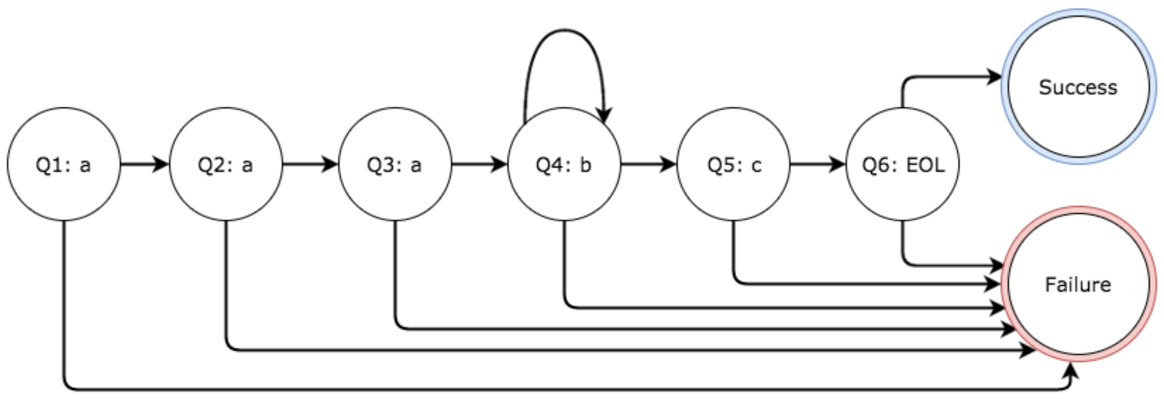
- Regular expressions are declarative and rule-based.
- Regular expressions either match or don't match a string, ending in a success state or a failure state.
- There are no control structures or flow changes, but only successful matching of an event to a condition or not.
- A finite state machine matches strings character by character, using a state-to-state process.
- The finite state machine only needs to match the next character if the current one matches, relying on a single condition to transition to a new state.
- Regular expressions are composable, meaning a regular expression can contain other regular expressions.

Because of all these characteristics, programmers often use a finite state machine in the implementation of a regular expression engine. The language will push a string into the state machine, and character by character, conditions will either be met or unmet and the engine will place itself within a failure or success state.

Example of a Finite State Machine

Let's take a look at an example:

- Regular expression: a{3}b*c
- Example string: aaabbc
- Finite state machine:



As the flow chart for the finite state machine describes, we can break down the regular expression into individual states that are required to match a string. The machine will look at each character in the new string one at a time from left to right, until it finds a character that does not match the corresponding character in the example string or until it reaches the end of the string:

- Q1: a
  - Condition: 'a'
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q2
- Q2: a
  - Condition: 'a'
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q3
- Q3: a
  - Condition: 'a'
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q4
- Q4: b*
  - Condition: 'b'
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q4 (loop back)
  - Condition: *empty*

  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q5

- Q5: c
  - Condition: ‘c’
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Q6

- Q6: *end of string*
  - Condition: *empty*
  - Condition unmet: Transition to Fail State
  - Condition met: Transition to Success State

Using A Specific Example

If we work through the string aaabbc, we can see these states being triggered and a successful run occur:

- Q1: String: aaabbc
  - First letter: a
  - Q1 condition check? Met
  - Transition to Q2

- Q2: String: aabbc
  - First letter: a
  - Q2 condition check? Met
  - Transition to Q3

- Q3: String: abbc
  - First letter: a
  - Q3 condition check? Met
  - Transition to Q4

- Q4: Substring: bb
  - First letter: b
  - Q4 condition check? Met
  - Transition to Q4
  - Q4: Substring: b
  - First letter: b
  - Q4 condition check? Met
  - Transition to Q4
  - Q4: Substring: *empty*
  - First letter: empty
  - Q4 condition check? Met
  - Transition to Q5

- Q5: String: c
    - First letter: c
    - Q5 condition check? Unmet
    - Transition to Q6
- Q6: String: c
    - First letter: c
    - Q6 condition check? Met
    - Transition to Success State

Regular expression engines can become quite complex with even the simplest of expressions. Researchers have proven finite state machines to be mathematically more efficient and effective than any other algorithm for creating a regular expression engine.[2]

Check Yourself: Finite State Machines

2. True or false? A finite state machine can be in more than one state at any given time.

a. True
b. False

Correct Answer: A finite state machine can be in exactly one state at any given time.

Analyze choices in syntax and structure in declarative programming.

The ability to use a regular expression to search text has become commonplace in technology. Before the wide adoption of regular expressions, programmers often used **brute-force searches** to find a search term within a large document. Users would provide a list of terms to find within a large body of text, and the search program would go through the text, character by character, comparing the text against the users' search phrases. Using a bit of math and formal language research, programmers invented regular expressions to describe more efficiently the terms to be matched in a body of text.

In the previous example, where the solution was to find all words containing at least one vowel, we could have written an imperative program that went line by line through code and found examples for each vowel. Using a declarative solution and a regular expression engine (in our examples, we used grep), we were able to concisely state our regular expression as "(\w*[aeiouAEIOU]\w*)" and have the engine do the hard work for us.

Regular Expression Syntax

Let's go over the basic regular expression syntax and some useful features that most implementations carry. Unfortunately, not all regular expression engines are created equally. There are different algorithms on how to compute regular expressions as well as special syntax introduced by some of the programming languages to provide additional features. Any programming language's documentation should describe the common regular expression syntax as well as any special features or requirements. This section covers some of the more basic syntax of regular expressions.

[2] See the video at "Finite State Machines: Part 1" for another example of the use of finite state machines outside of regular expressions.

A regular expression is generally composed of a series of literal characters, special symbols, and **metacharacters** combined to represent a pattern of the solution string to match. Metacharacters are modifiers that specify occurrences of patterns or repeated strings. Developers can apply them to literal characters, special symbols, and subexpressions. They are important because they help programmers build a regular expression that is concise or flexible.

Given that a regular expression uses a declarative syntax, the search string describes how the solution *should* look using literal characters. When defined alone without the special symbols, each character in the regular expression represents a literal character. If we have a regular expression of *w*, it means search for the literal character *w*. If we search for *\w*, which happens to be a special symbol and the letter *w*, the regular expression engine does not search for the literal character *w*. Instead, it searches for any single word character, which is considered any letter, number, or underscore.

Special Symbols

So with that in mind, let's look at some of the special symbols and what they mean:

- ^
  This signifies the start of the line. If the strings must match starting at the beginning of the line, use this symbol to start the search.
    - Example regular expression: '^A brown dog'
    - Matches: *A brown dog.*
    - Does not match: *A dirty dog. A brown dog.*
- $
  This signifies the end of the line.
    - Example regular expression: 'dog$'
    - Matches: A brown dog<EOL>
    - Does not match: A brown dog barks<EOL>
- .
  Matches a single character.
    - Example regular expression: "a..."
    - Read as "an 'a' with any three characters"
    - Matches: "abcd"
    - Does not match: "a"
    - Note: To match a literal '.' within a regular expression, you must escape the '.' with '\'. Example: "The end of sentence\."
- \s
  Matches any single whitespace character, including tab, space, and new lines.
    - Example regular expression: "123\sabc"
    - Matches: "123 abc"
    - Does not match: "123abc"
- \S
  Matches any single non-whitespace character.
    - Example regular expression: "\Sabc"

        - Matches: “zabc”
        - Does not match: “ abc”
- \d
  Matches any single digit.
    - Example regular expression: “\d\d\d”
        - Matches: “999”
        - Does not match: “45a”
- \D
  Matches any single non-digit.
    - Example regular expression: “\D\D\D”
        - Matches: “abc”
        - Does not match: “999”
- \w
  Matches any single word character, including letters, numbers, and underscores.
    - Example regular expression: “\w\w”
        - Matches: “aa”
        - Does not match: “a?”
- \W
  Matches any single non-word character.
    - Example regular expression: “\W\W”
        - Matches: “??”
        - Does not match: “b9”
- \b
  Matches a single word character and a non-word character boundary or edge.
    - Example regular expression: “dog\b”
        - Matches: “dog.”
        - Does not match: “dogs”
- [...]
  Matches a single character out of the set defined within the brackets.
    - Example regular expression: “[aeiou]”
        - Matches: “a”
        - Does not match: “b”
- [^...]
  Matches a single character not in the specified set within the brackets.
    - Example regular expression: “[^aeiou]”
        - Matches: “b”
        - Does not match: “a”
- (...)
  Groups the patterns within the parentheses into a single result.
    - Example regular expression: “a(dog)”
        - Matches: “adog” (and will **capture** “dog”)

        - Does not match: "doggie"
    - Regular expression engines will often *capture* the results within a group for easy extraction. That way developers can match *and* extract the useful data from a large body of text.

- \N
  Matches a *back reference* to the the *N*th group capture.
    - Example regular expression: "A\s(dog|cat)\sis\sa\s\1\."
        - Matches: "A dog is a dog." or "A cat is a cat."
        - Does not match: "A dog is a cat."
    - In the example, the \1 tells the regular expression engine to reuse and match the capture from the first group defined within the parentheses. In the first set of parentheses, it captures either dog or cat. The \1 then says to match dog or cat, but match it exactly as found in the first capture.
    - This is useful for code such as HTML tags, where each element must include an opening tag and a matching closing tag.
    - The *N* is a 1-based index, meaning the first capture result is 1, second is 2, and so on.
- [a-z]
  Matches a single character within a range defined by the first and last character.
    - Example regular expression: "[a-z]"
        - Matches: "f"
        - Does not match: "9"
- Example regular expression: "[a-zA-Z0-9_]"
    - Matches: "A"
    - Does not match: "?"
- These are equivalent regular expressions:
    - "[a-zA-Z0-9_]"
    - "\w"

Metacharacters

The basic list of metacharacters is as follows:

- |
  Separates alternative possibilities within an expression.
    - Example regular expression: "(dog|cat)"
    - Matches: "cat" or "dog"
    - Does not match: "fish"

- ?
  Matches an expression if it occurs zero or one times.
    - Example regular expression: "dogs?"
    - Matches: "dog" or "dogs"
    - Does not match: "hotdogs"

- *
  Matches an expression if it occurs zero or more times.
  - Example regular expression: "a*"
  - Matches: "" or "a" or "aa" or "aaa", etc.
  - Does not match: "b"

- +
  Matches an expression if it occurs one or more times.
  - Example regular expression: "a+"
  - Matches: "a" or "aa" or "aaa", etc.
  - Does not match: ""

- {M}
  Matches an expression if it occurs *M* consecutive times.
  - Example regular expression: "a{3}"
  - Matches: "aaa"
  - Does not match: "aa"

- {M,}
  Matches an expression if it occurs *M* or more times.
  - Example regular expression: "a{3,}"
  - Matches: "aaa" or "aaaa" or "aaaaa", etc.
  - Does not match: "aa"

- {M,N}
  Matches an expression if it occurs *M* to *N* times.
  - Example regular expression: "a{1,2}"
  - Matches: "a" or "aa"
  - Does not match: "aaa"

Again, the metacharacters can reference literal characters, special symbols, or subexpressions within a full regular expression. Developers can use these in combination to generate very concise and flexible regular expressions that can find many strings within large bodies of text.

Check Yourself: Using Regular Expressions

3. Which regular expressions find only the string aaa within a body of text? (Select all that apply.)

a. [aaa]
b. a*
c. a{3}
d. a+
e. aaa

Correct Answer: You can use either *a{3}* or *aaa* to find the string aaa.

References

Brown, B. (2014, November 23). *Finite state machines: Part 1*. [Video file]. Retrieved from https://youtu.be/4XEK7OU2gIw

GNU Grep. (n.d.). In *The Free Software Directory*. Retrieved February 13, 2015, from http://directory.fsf.org/wiki/Grep

Regular expression. (n.d.). Retrieved February 13, 2015, from https://en.wikipedia.org/wiki/Regular_expression

Udacity. (2012, June 3). *Finite state machines - Programming languages*. [Video file]. Retrieved from https://youtu.be/kXkd0qtJfpQ

Glossary

Brute-force search

A search algorithm that enumerates every possible candidate and checks it against the solution. Generally considered the slowest form of searching. Also known as *exhaustive search*.

Capture

A feature within regular expressions that allows developers to capture groups of expressions within a regular expression. Useful for finding a whole string and capturing specific parts of the string as results as well.

Declarative programming

A programming language whose syntax focuses on what a solution should look like rather than a series of instructions on *how* to solve the problem.

Hypertext markup language (HTML)

A declarative markup language that Web browsers use to interpret and compose text and images on a screen.

Metacharacters

Modifiers that specify occurrences of patterns or repeated strings.

Regular expression

A declarative programming language that is made up of a series of symbols and characters that form a search pattern, used for matching text.

Regular expression engine

A program or library that can take text and a regular expression as input and return the matches for the regular expression. Most modern programming languages provide a regular expression engine built in or as part of their standard library.

State machine

A computational model in which a machine is composed of a finite set of states. The machine is in only one state at a time, the *current state*. An *event* or *condition* can cause the machine to move from one state to the next.