Scanning & Regular Languages: Part 1

Motivation

- A string like “add $3, $2, $1” is stored as a sequence of characters:

  `'a'  'd'  'd'  ' '  '$'  '3'  ','  ' '  '$'  '2'  ','  ...`

- The string “add $3, $2, $1” is how we represent an Add instruction ($3 = $2 + $1) in **MIPS assembly language**.
- The corresponding **MIPS machine language** instruction is 000000 00010 00001 00011 00000 100000.
- If we could break down the string “add $3, $2, $1” into meaningful chunks of information, it would be easier to translate it:

  `[add]  [$3]  [$2]  [$1]`

Motivation

- If we could break down the string “add $3, $2, $1” into meaningful chunks of information, it would be easier to translate it:

  `[add]  [$3]  [$2]  [$1]`

- A **scanner** is a tool that splits strings into meaningful **tokens**, while discarding portions of the string that are unimportant.
- For example, “add $3, $2, $1” might become:

[Identifier: “add”] [Register: “$3”] [Comma: “,”] [Register: “$2”] [Comma: “,”] [Register: “$1”]

- The commas are kept for syntax checking purposes, but whitespace in the string is discarded. Once syntax checking is done, we can discard the comma tokens and only keep the important information.

Tokens & Tokenizations

- Aside from splitting the input string into smaller strings, a scanner also assigns a **kind** to each string, describing the string's meaning.
- This combination of a string and a "kind" is called a **token**.
- The string is called the **lexeme** of the token.
- The goal of scanning is to find a **tokenization**, a sequence of tokens whose lexemes can be put together to form the input string.
- Scanning requires a definition of what strings are considered valid token lexemes, and what kind to assign to each lexeme.
- There might be multiple valid tokenizations for a single input string.

Maximal Munch Scanning

- Consider the string “241”. There are four ways to split up this string:
  - “241”
  - “2” “41”
  - “24” “1”
  - “2” “4” “1”
- Not all tokenizations make sense in all contexts.
- For example, in the C statement “int x = 241;” we probably want to interpret “241” as a single number, not two or three numbers.
- **Maximal munch** is a scanning algorithm that produces a unique tokenization by always extracting the longest token possible.

Maximal Munch: Overview

- Definition: A **prefix** of a string is a substring that begins at the start (for example, “token” is a prefix of “tokenization”).
- Maximal munch works as follows:

1. Find the longest prefix of the input that is a valid token lexeme. If no such prefix exists, halt with an error (scanning failed).
2. If a prefix was found, remove it from the front of the input and generate a token for this prefix.
3. Repeat the above steps until either an error occurs, or the input becomes empty (scanning successful).

Problems

- Normally, the set of valid token lexemes is *infinite*.
- For example, to tokenize C, you would want all possible variable names to be valid token lexemes. Unless you impose a length limit on variable names, there are infinitely many.
- Even if you impose length limits, the set of valid token lexemes will likely still be very large for a practical programming language.
- How do we efficiently check membership in an infinite or very large set of strings?
- How do we even *represent* sets like this in code?

Regular Expressions

- These are a string matching tool that many programming languages offer as part of their standard library.
- As an example, we could describe the set of all non-empty strings of digits from 0 to 9 with the expression: “[0-9]+”
  - [0-9] matches any single character in the range 0 to 9
  - The + character means “one or more repetitions of the preceding expression”
- You can describe infinite sets of strings this way, as long as they have a simple enough structure.
- But how are these actually implemented? How do you check if a string matches a regular expression?

Regular Languages

- Regular expressions are based on **regular languages**, a concept from theoretical computer science.
- “Language” does not mean programming language here, it refers to a **formal language**, which is simply a set of strings.
- It may seem strange to just call any set of strings a “language”. We’ll revisit this definition later in the course when we cover *formal language theory.*
- A regular language is a restricted kind of formal language that is constructed according to certain rules.

Regular Languages: Definition

- A language (set of strings) is regular if:

1. It is the empty set: { }
2. It is the set containing only the empty string: { "" }
3. It is a set containing only a single string, and this string consists only of one character.
4. It is the *union* of two regular languages.
5. It is formed from two regular languages by (pairwise) *string concatenation*.
6. It is the *Kleene star* of a regular language.

Regular Language Operations

- **Union:** All elements that are in at least one of the two languages.
  - $L_1 \cup L_2 = \{ x : x \in L_1 \text{ or } x \in L_2 \}$
- **(Pairwise) Concatenation:** All strings formed by concatenating a string from the first language with a string from the second language.
  - $L_1 \cdot L_2 = \{ xy : x \in L_1 \text{ and } y \in L_2 \}$
  - Like multiplication, the dot is often omitted and we just write L_1L_2
- **Kleene Star:** All strings that can be formed by concatenating any combination of strings in a given language (including the empty string, formed by concatenating "nothing" or "no strings").
  - $L^* = \{""\} \cup L \cup LL \cup LLL \cup LLLL \cup \cdots$

Regular Language Operations: Examples

- Union: {"a"} ∪ {"b"} ∪ {"c"} = {"a","b","c"}
- Concatenation:
  - {"a","b","c"}{"a","b","c"} = {"aa","ab","ac","ba","bb","bc","ca","cb","cc"}
  - {"c"}{"a"}{"b"} = {"cab"}
  - {""}{"cab"}{""} = {"cab"} *(concatenation with empty string does nothing)*
  - {"aaa","aba","aca"}{} = {} *(concatenation with empty set is empty)*
- With union and concatenation together, we can form any finite set.
- ({"up"} ∪ {"down"})({"hill"} ∪ {"load"}) = {"uphill","upload","downhill","download"}

Regular Language Operations: Examples

- Star: {"a"}* = {"", "a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", …}
- {"a","b","c"}* = {"", "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", "aaa", "aab", "aac", "aba", "abb", "abc", "aca", "acb", "acc", "baa", "bab", "bac", "bba", "bbb", "bbc", "bca", "bcb", "bcc", "caa", …}
  - Contains **all** strings formed by concatenating a, b and c.
  - Including the empty string (represents "concatenating nothing").
- {"up","down","hill","load"}* = {"", "up", "down", "hill", "load", "upup", "updown", "uphill", "upload", "downup", "downdown", "downhill", …}
- (L*)* = L* for all languages L.
- {""}* = {""} and { }* = {""}.

Regular Language Operations: Examples

- Can the star of an infinite set be a different infinite set?
- Yes. Consider L = {"a"}* ∪ {"b"}* = {"", "a", "aa", …} ∪ {"", "b", "bb", …}
- L does not contain "ab" or "baba" but L* does. In fact, L* = {"a","b"}*.
- {"a"}{"cc"}*{"a"} = {"aa", "acca", "acccca", "acccccca", …}
  - Strings that start and end with a, and have an even number of c's in between.
- {"aa"}*{"bb"}* = {"", "aa", "aaaa", …}{"", "bb", "bbbb", …}
  = {"", "aa", "bb", "aaaa", "aabb", "bbbb", "aaaaaa", "aaaabb", …}
  - Strings with an even number of a's *followed* by an even number of b's.
  - For example, "bbaa" is not in this language.

Regular Expressions

- Regular expressions originated as a simplified text notation for describing regular languages.
- We drop the “set” notation: {"a"} is just written "a".
- Union is written using | instead of ∪. This is easier to type on a standard ASCII keyboard and avoids confusion with the letter U.
- Concatenation is normally written without the dot.
- Although we dropped the { } set notation, we can use ( ) parentheses to group expressions when necessary.
- For example, {"b"}{"ab"}*{"a"} can be written as "b(ab)*a".

Regular Expressions: Order of Operations

- The order of operations is: star, concatenation, then union.
- Does "ab|c "mean {"ab"} ∪ {"c"} or {"a"}({"b"} ∪ {"c"})
  - Answer: {"ab"} ∪ {"c"} = {"ab", "c"}
- Does "ba*" mean {"ba"}* or {"b"}({"a"}*)?
  - Answer: {"b"}({"a"}*) = {"b", "ba", "baa", "baaa", "baaaa", …}
- What does "a|b*(a|c)a*" mean?
  - Star has highest precedence: "a|(b*)(a|c)(a*)"
  - Concatenation is next: "a|((b*)(a|c)(a*))"
  - Union is last, so the answer is: {"a"} ∪ ({"b"}*)({"a"} ∪ {"c"})({"a"}*)
    = {"a", "c", "aa", "ba", "bc", "ca", "aaa", "baa", "bba", "bbc", "bca", "caa", …}

Regular Expressions in Practice

- Most modern implementations of regular expressions typically have a lot of additional syntax and features.
- Some features allow you to describe languages that are *not regular!*
- For example, egrep supports “backreferences”: Referring to an earlier part of the matched string.
- Many features are just for convenience though, and do not grant additional expressive power.
- For example, practical implementations usually have the + operator, where R+ is the same as RR*.

Regular Expressions in CS 241

- If we ask you to write a regular expression in this course, we are expecting a **formal regular expression**, meaning only the operations of union, concatenation, and Kleene star are allowed.
- Formal regular expressions are built up from base languages (empty language, language containing the empty string, languages containing single-character strings) and these three operations.
- If asked to write a regular expression on an exam, do not use other operations/features from tools like egrep or from regular expression engines in programming languages you know.
- If you want to use regexes on a C++ or Racket programming question though, you're allowed to use whatever features you like.

Recognizing Regular Languages

- The **recognition problem** for a language is to determine, given a string as input, whether the string belongs to the language.
- If we solve this, we can implement maximal munch. We can determine whether a prefix of the string-to-scan is a valid token, and find the longest such prefix.
- However, the recognition problem is not always solvable.
  - Example: the language of all strings which are valid C programs that halt (do not loop infinitely) when run with no input.
- For **regular languages**, the recognition problem is solvable, so we can use these languages for maximal munch scanning.

Regular Language Recognition Programs

- Let's create recognition programs for each kind of regular language:

1. The empty set.
2. A set containing only an empty string.
3. A set containing only one string, which has only one character.
4. The union of two regular languages that we already have recognition programs for.
5. The concatenation of two regular languages that we already have recognition programs for.
6. The Kleene star of a regular language that we have a recognition program for.

Simple Cases

1. Empty Set
   - Our recognition program just rejects all inputs.
2. Set Containing Only The Empty String
   - Our recognition program rejects all non-empty inputs and accepts the empty input.
3. Set Containing A Single-Character String
   - Our recognition program rejects all inputs that are not single-character strings. For single-character strings, it compares the input against the expected character, and accepts or rejects accordingly.

Union

- Let's suppose we have two regular languages, and we've developed recognition programs for both of them.
- It is straightforward to develop a recognition program for the union of the two languages.
- Given regular languages L_1 and L_2 and an input string x:
  1. Run the recognition program for L_1 on x.
  2. Run the recognition program for L_2 on x.
  3. If one or both programs accepted, then accept x. Otherwise, reject x.

Concatenation

- Assume we have recognition programs for two regular languages, and we want to develop a recognition program for their concatenation.
- Given regular languages L_1 and L_2 and an input string x, we consider all possible ways of splitting x into two strings.
- For each pair of strings (y, z) such that concatenating y with z produces x:
  1. Run the recognition program for L_1 on y.
  2. Run the recognition program for L_2 on z.
  3. Accept x if both y and z were accepted in the above steps.
- If you test every pair and they *all* fail, reject x.

Kleene Star

- Assume we have a recognition program for a regular language, and we want to develop a recognition program for its Kleene star.
- Like before, we consider ways of splitting up the input string x, but we have to consider *all possible decompositions* into *any number of parts* instead of just ways of splitting the string into two parts.
- Fortunately, the number of decompositions is finite.
  - We can ignore decompositions where some of the "parts" are empty strings.
  - Therefore, if a string has length n, it can be split into at most n parts.
  - So, we only need to test decompositions into n parts or fewer.
- It's a brute force approach, but we can solve the recognition problem.

Kleene Star

- Let L be a regular language and assume we have a recognition program for L.
- Given an input string x of length n, find all ways to split x into at most n substrings.
- For each k-tuple of strings (x_1, …, x_k) where $k \leq n$ and concatenating the strings x_1 through x_k together produces *x:*
  1. Run the recognition program for L on each string x_i in the k-tuple.
  2. Accept if *all* x_i strings were accepted in the previous step.
- If you test every tuple and they *all* fail, reject x.

From Regular Expressions to Programs

- Given a regular language, described using either regular expressions or set notation, we have just seen a procedure to construct a recognition program for this language.
- First break it down into individual characters (or the special cases of the empty set and empty string, if these are used in the expression). It is straightforward to write recognition programs for these parts.
- Then, use the techniques we described for union, concatenation and Kleene star to combine these recognition programs together into one big program.
- This will work in theory. But is this practical? Is it efficient?

No, it's a huge pain

- Consider the following relatively short regular expression: "b*ab*(ab*ab*)*"
- How many small programs would we need to create and compose together just to recognize this simple expression?
- How long and complicated would the final code be?
- Considering concatenation and star used brute-force algorithms, would this even be efficient?
- We now know it's theoretically possible to solve the recognition problem for regular languages. But is there a better approach?

A More Direct Method

- What strings does the expression "b*ab*(ab*ab*)*" describe?
- If we remove all the "b*" parts we get "a(aa)*", which matches strings of a's that have odd length.
- This expression is basically built by taking "a(aa)*" and inserting "b*" in between every "a".
- "b*" matches any number of b's (including zero).
- In other words, this expression describes strings that have an odd number of a's, and any number of b's, but no other characters.
- It is easy to just write a recognition program for these strings directly.

A More Direct Method

- The expression "b*ab*(ab*ab*)*" describes strings with have an odd number of a's and an arbitrary number of b's.
- A recognition program for these strings could work as follows:
  - Read each character one at a time.
  - If a character other than "a" or "b" is seen, reject immediately.
  - Track the parity (odd or even) of the number of a's by keeping a counter modulo 2. When finished reading the string, if this counter is 1 (odd), accept. If the counter is 0 (even), reject.
- Coming up with this program is probably easier than coming up with a regular expression describing these strings!

An Important Insight

- Some regular languages are easy to describe with regular expressions. But some are actually a little unintuitive to describe using regular expressions, and are easier to describe by just **specifying the recognition program directly**.
- We're going to explore how to specify simple recognition programs using **finite automata** (also known as finite state machines).
- Recognition programs based on finite automata are efficient and easy to implement, and can actually recognize any regular language (though this fact is not obvious!)
- We will ultimately use finite automata to implement our scanner.

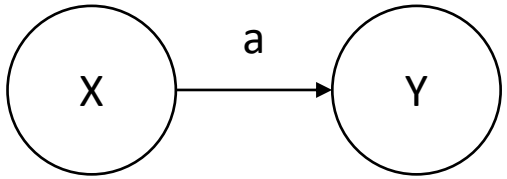
Deterministic Finite Automata

Deterministic Finite Automata

- **Deterministic finite automata (DFAs)** are a tool for describing simple language recognition programs, which work as follows:
  - The program has a *finite* number of distinct *states*.
  - The program can occupy one state at a time.
  - The program reads one character of input at a time, and cannot backtrack in the input.
  - Each time the program reads a character, the state is updated, following a *deterministic* process. The new state is completely determined by the previous state and the character that was read.
  - Some states are designated *accepting states*. Once all input has been read, the string is accepted if the current state is accepting, and rejected otherwise.

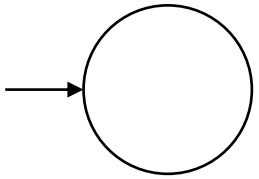
DFA State Diagrams

- Although DFAs represent recognition programs, we often represent them with *state diagrams,* rather than code.
- Each state in the program is represented by a circle.
  - The state can have a *name* written inside the circle.
  - Names are optional and don't have any meaning, but can make it easier to understand.
- If reading character "a" takes the program from state X to state Y, we draw it like this:

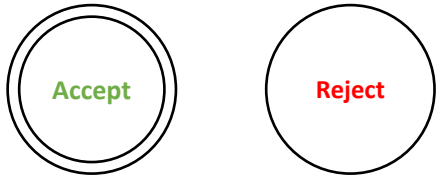


DFA State Diagrams, Continued

- The initial state of the program is represented by a circle with an arrow pointing inwards.

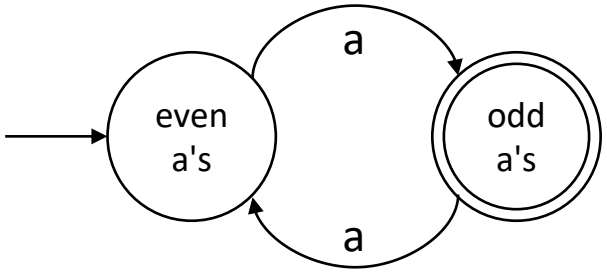


- If a state is “double-circled”, it is an accepting state. If the program ends up in this state after reading all input, it will accept the input. Otherwise, it will reject the input.



DFA Examples

- The following DFA recognizes strings of a's that have odd length.



- The following DFA recognizes strings of a's and b's with an odd number of a's and an arbitrary number of b's.