

Abstract Thinking,  
Intro to the Theory of Computation

SC/COSC 2001 3.0 Lecture Notes

Jeff Edmonds

Winter 99-00, Version 0.2

# Contents

<b>1</b>	<b>Preface</b>	<b>2</b>
<b>2</b>	<b>DFAs, Iterative Programs, and Loop Invariants</b>	<b>5</b>
2.1	Different Models and Notations for Simple Devices and Processes . . . . .	5
2.2	Iterative Programs and Loop Invariants . . . . .	7
2.3	Mechanically Compiling an Iterative Program into a DFA . . . . .	9
2.4	More Examples . . . . .	11
<b>3</b>	<b>Converting NFA to DFA using the Clones and the Iterative Program Levels of Abstraction</b>	<b>22</b>
<b>4</b>	<b>Overview of Results</b>	<b>28</b>
<b>5</b>	<b>Pumping vs Bumping Lemma</b>	<b>30</b>
5.1	Pumping Lemma . . . . .	30
5.2	Bumping the Pumping Lemma . . . . .	32
<b>6</b>	<b>Context Free Grammars</b>	<b>36</b>
6.1	Proving Which Language is Generated . . . . .	38
6.2	Pumping Lemma . . . . .	41
<b>7</b>	<b>Parsing with Context-Free Grammars</b>	<b>42</b>

# Chapter 1

## Preface

These notes are by no means complete. They are just those that I have managed to scratch together so far.

### Intro:

**Thinking Abstractly:** On one level, this course can be viewed as teaching a sequence of simple models of computation and simple algorithms. On a higher level, the goal of the course is to teach you to think abstractly about computation. The more abstractions a person has from which to view the problem, the deeper his understanding of it will be, the more tools he will have at his disposal, and the better prepared he will be to design his own innovative ways to solve the problems that may arise in other courses or in the work place. Hence, we present a number of different notations, analogies, and paradigms within which to develop and to think about algorithms.

**Explaining:** To be able to prove yourself within a test or the world, you need to be able to explain the material well. In addition, explaining it to someone else is the best way to learn it yourself. Hence, I highly recommend spending a lot of time explain the material over and over again out loud to yourself, to each other, and to your stuffed bear.

**Dreaming:** When designing an algorithm, the tendency is to start coding. When studying, the tendency is to read or to do problems. Though these are important, I would like to emphasize the importance of thinking, even day dreaming, about the material. This can be done while going along with your day, while swimming, showering, cooking, or laying in bed. Ask questions. Why is it done this way and not that way? Invent other algorithms for solving a problem. Then look for input instances for which your algorithm gives the wrong answer. Mathematics is not all linear thinking. If the essence of the material, what the questions are really asking, is allowed to seep down into your subconscious then with time little thoughts will begin to percolate up. Pursue these ideas. Sometimes even flashes of inspiration appear.

### Course Outline:

**Useful Models:** We learn about regular languages, deterministic finite automata, context free grammars, Turing machines, and NP-completeness. We learn the definitions, how to construct, and how to manipulate them.

In addition to teaching us to think abstractly, these models of computation are very useful in themselves.

**Regular Languages** for describing simple string structures;

**Deterministic Finite Automata** for understanding simple machines and processes;

**Context Free Grammars** for describing languages such a English or Java and for describing recursive structures;

**Turing Machines** for understanding the bare bones of computation and for proving that certain computational problems are not computable;

**NP-Completeness** for understanding that large classes of useful computational problems require too much time to practically compute.

**Describing Algorithms:** Learning about different models of computation will help you. Each model provides a different notation for describing algorithms. Each provides a different abstract framework for thinking about algorithms.

The first model we learn is DFA. Because these perform simplistic tasks, like determining if the number of *as* in a string is even or odd, students think that they are not useful. However, it is their simplicity that makes them useful. In themselves, they have many applications. However, a main reason for teaching them is that students find it very difficult to think abstractly about computation. Hence, it is good to start with simple ideas.

**Compiling Between Models:** You will be taught algorithms for mechanically “compiling” an algorithm developed in one model into an equivalent algorithm described in another model.

**Iterative Loop Invariant Model  $\Leftrightarrow$  Deterministic Automata:** Loop Invariants provide a useful level of abstraction for developing, understanding, and describing iterative algorithms. Deterministic Automata provides another. It is useful and insightful to be able to translate between them.

**Deterministic Finite Automata (DFA)  $\Leftrightarrow$  Regular Expressions:** As said above, both are very useful levels of abstraction. Hence, it is important to be able to translate between them.

**Deterministic Finite Automata (DFA)  $\Leftrightarrow$  Nondeterministic Deterministic Automata (NFA):**

Another very useful level of abstraction is *nondeterminism*. This can be viewed as *help from above*. The idea is to write your algorithm assuming that you have some very powerful help from your favorite *higher power*. With this help, designing the algorithm is much easier. Once a correct algorithm has been designed within this level of abstraction, the next completely separate task is to *move between* the different levels of abstraction, by mechanically “compiling” your algorithm into an algorithm that does not require help from above.

**Nondeterministic Machine:** The *transition function* of these machines does not provide only one next state that the machine must be in based on its current configuration. Instead it gives a set of possible states. The result is each input instance does not have just one computation path, but has many. We say that the input instance is *accepted* if there exists at least one computation path that leads to an accept state.

**Help From A Higher Power:** One can view a computation on a nondeterministic machine as requiring nondeterministic help from a higher power. This power tells the computation the sequence of choices to make to lead it along one of the accepting computation paths to an accepting state. A key feature of this help is that a corrupt higher power can never convince you that there is an accepting computation when there is not one. The reason is that you are able to check that the sequence of choices given to you does in fact lead to an accept state.

**Context Free Grammars  $\Leftrightarrow$  Push Down Automata:** Context free grammars are very useful. Push down automata are only of passing interest.

**Turing Machines  $\Leftrightarrow$  Multi Tapes & Heads  $\Leftrightarrow$  Random Access  $\Leftrightarrow$  JAVA Code:**  
The Church/Turing thesis states that all reasonable models of computation are equivalent as far as which computational problems are computable. This is worth understanding.

**Resources Needed for Computation:** Suppose your boss wants you to design an algorithm for solving some new computational problem. You should be able to determine what resources this problem requires. Could it be hard wired into a small chip or does it require more computation time and more memory? Can it be solved quickly or will it require more time than the number of atoms in the universe or is it provably unsolvable even given unbounded time.

Learning about different models of computation will help you with this as well. Each model also places restrictions on the resources and types of actions that an algorithm is an is not allowed to do. Then we learn to classify computational problems based on which resources they require.

In this course, you will learn each of these separate levels, as well as ways of moving between the levels. Individually, no level is very difficult. The key to the entire course is to apply these abstract thinking skills to solving new problems that arise in other courses and in the workplace.

## Chapter 2

# DFAs, Iterative Programs, and Loop Invariants

DFAs, NFAs, and Regular Languages are all discussed very well in the Sipser book. Here we will discuss the connection between these and iterative programs and loop invariants.

## 2.1 Different Models and Notations for Simple Devices and Processes

This section studies simple algorithms, devices, processes, and patterns.

### Examples:

- simple iterative algorithms
- simple mechanical or electronic devices like elevators and calculators
- simple processes like the job queue of an operating system
- simple patterns within strings of characters.

**Similar Features:** All of these have the following similar features.

**Input Stream:** They receive a stream of information to which they must react. For example, the stream of input for a simple algorithm consists of the characters read from input; for a calculator, it is the sequence of buttons pushed; for the job queue, it is the stream of jobs arriving; and for the pattern within a string, one scans the string once from left to right.

**Read Once Input:** Once a token of the information has arrived it cannot be requested for again.

**Bounded Memory:** The algorithm/device/process/pattern has limited memory with which to remember the information that it has seen so far.

**Models/Notations:** Because they have these similar features, we are able to use the same *models of computation* or notations for describing them and thinking about them. They also help focus on the above listed aspects of the algorithm/device/process/pattern without being bogged down with details that are irrelevant to the current level of understanding.

There are, however, more than one such models/notations that can be used. Each provides different tools, insights, and levels of abstraction. One should know how to develop algorithm/devices/processes in each of these models and know how to mechanically compile an algorithm/device/process/pattern developed in any of these models into an equivalent one within any of the others. The models/notations considered are the following.

**Deterministic Finite Automaton (DFA):** A DFA,  $M = \langle Q, \Sigma, \delta, s, F \rangle$ , is an abstraction of an algorithm/device/process/pattern. It has a set of states  $Q = \{q_1, q_2, \dots, q_{|Q|}\}$  to represent the states that the device might be in and a transition function  $\delta : Q \times \Sigma \rightarrow Q$  that dictates how the device responds to the next input token. If the DFA's current state is  $q \in Q$  and the next input character is  $c \in \Sigma$ , then the next state of the DFA is given by  $q' = \delta(q, c)$ .

Though a DFA can be used to model any algorithm/device/process/pattern, this course will focus on using it as a model of computation. The input for the computation is fed character by character into the DFA. The computation on the DFA begins by initializing the DFA into the start state denoted by  $s$ . When the last character of the input has been read, the input is *accepted* if the DFA is in one of the accept states contained in the set denoted by  $F$ . If the last state is not an accept state, then the input is *rejected*.

**A path through a graph:** A DFA can be viewed as an edge labeled graph. The nodes of the graph correspond to the states of the DFA; the labeled edges of the graph to the transitions of the DFA; and paths through the graph correspond to computations on the DFA.

The graph representing the DFA has the property that for each node  $q$  and each character  $c$  of the alphabet  $\Sigma$  there is one edge labeled  $c$  coming out of node  $q$ . It follows that for every string  $\omega \in \Sigma^*$  there is exactly one path through the graph, labeled with  $\omega$ , starting at the start state (proof by induction on the length of the string). The DFA accepts a string  $\omega$  if this path labeled  $\omega$  ends in an accept state.

Two advantages of switching to this level of abstraction is that it is more visual and one then can draw on all the tools of graph theory.

**An Iterative Program with a Loop Invariant:** The above models sometimes seem esoteric to a beginning student. Hence, we will consider how such computations/devices/processes/patterns can be developed and expressed using standard code.

An algorithm that consists of small steps implemented by a main loop is referred to an *iterative algorithm*. Examples, will be given below. Formally, one proves that an iterative algorithm works correctly using what are called *loop invariants*. These same techniques can also be used to think about, develop and describe iterative algorithms so that they are easily understood.

We will see that the idea of a loop invariant also provides meaning to the states of a DFA. Instead of naming the states  $q_0, q_1, q_2, \dots$ , it is better to give the states names that explain what the computation remembers about the part of the input read so far.

**Nondeterministic Finite Automaton (NFA):** Another very useful level of abstraction is *nondeterminism*. The idea is to write your algorithm assuming that you have some very powerful help from your favorite *higher power*. With this help, designing the algorithm is much easier. Once a correct algorithm has been designed within this level of abstraction, the next completely separate task is to *move between* the different levels of abstraction, by mechanically “compiling” your algorithm into an algorithm that does not require this help.

**Nondeterministic Machine:** The *transition function* of these machines does not provide only one next state that the machine must be in based on its current configuration. Instead it gives a set of possible states. The result is that each input instance does not have just one computation path, but has many. We say that the input instance is *accepted* if there exists at least one computation path that leads to an accept state.

**Help From A Higher Power:** One can view a computation on a nondeterministic machine as requiring nondeterministic help from a higher power. This power tells the computation the sequence of choices to make to lead it along one of the accepting computation paths to an accepting state. A key feature of this help is that a corrupt higher power can never convince you that there is an accepting computation when there is not one. The reason is that you are able to check that the sequence of choices given to you does in fact lead to an accept state.

**Regular Languages:** Regular expressions are a very useful notation for describing simple string structures. We will see how they also correspond to the computational problems that are solved by DFAs.

## 2.2 Iterative Programs and Loop Invariants

**Structure:** An iterative program (with bounded memory) has the following structure.

```
routine()
  allocate "state" variables
  initialize "state" variables
  loop
    % ⟨loop invariant⟩           What we remember about the input read so far
    exit when end of input
    allocate "temp" variables, eg. c
    get(c)                       % Reads next character of input
    code                           % Compute new values for state variables
    deallocate "temp" variables
  end loop
  if("state" variables have an "accept" setting) then
    accept
  else
    reject
  end if
end routine
```

**Steps in Developing an Iterative Program:** You may want to read my optional document on writing iterative programs using loop invariants that can be found on the web. The following are the main steps.

**Assertions:** Assertions are very helpful when thinking through the logic of the algorithm and when explaining the algorithm to someone else.

An assertion is made at some particular point during the execution of an algorithm. It is a statement about the current state of the data structure that is either true or false. If it is false, then something has gone wrong in the logic of the algorithm.

An assertion is not a task for the algorithm to perform. It is only a comment that is added for the benefit of the reader.

Some languages allow you to insert them as lines of code. If during execution such an assertion is false, then the program automatically stops with a useful error message. This is very helpful when debugging your program. However, it is often the case that the computation itself is unable to determine whether or not the assertion is true.

**Loop Invariant:** Suppose that you, as the computation, have read in some large prefix of the input string. With bounded memory you cannot remember everything you have read. However, you will never be able to go back and to read it again. Hence, you must remember a few key facts about the prefix read. The most difficult part of designing such an algorithm is knowing what to remember. The *loop invariant* is the assertion that the state variables store this key information about this prefix. The following are different types of information that should be remembered.

**Whether to Accept This Prefix:** The prefix of the input that you have read so far may be the entire input. You do not yet know whether the end has been reached. Therefore, the foremost thing that you must know about the current prefix is whether you would accept it if the input would stop here. Is this prefix itself in the language?

**Progress:** You will likely need to remember additional information about the prefix of the string read so far. The language/computational problem in question may have a number of conditions that must be met before it accepts an input string. Perhaps the prefix of the string read so far is not in the language because it has not met all of these requirements, however, it has met some of them. In such case, you must remember how much progress has been made.

On the other hand, the language may reject a string as soon as a number of conditions are met. In this case, you must remember how much progress has been made towards rejection.

**Definitely Accept or Reject:** Sometimes the prefix is such that no matter what the rest of the input string is, you will definitely accept (or reject) it. This would be good to remember.

**Patterns:** Sometimes the strings that are accepted contain a pattern that must be repeated over and over. In this case, you must remember at what point in this pattern this prefix ends.

**Integrity Constraints:** Sometimes certain settings of the state variables are not allowed. For example, your the amount in your bank account cannot be negative. The loop invariant would state these constraints.

**State Variables:** What you remember about the prefix of the input read so far is stored in the variables that we will refer to as the *state* variables. You must define what variables are needed to store this information. They can have only a fixed number of bits or be able to take on only fixed number of different values.

**Maintaining Loop Invariant:** You must provide code for the loop that maintains the loop invariant. Let  $\omega$  denote the prefix of the input string read so far. You do not know all of  $\omega$ , but by the loop invariant you know something about it. Now suppose you read another character  $c$ . What you have read now is  $\omega c$ . What must you remember about  $\omega c$  in order to meet the loop invariant? Is what you know about  $\omega$  and  $c$  enough to know what you need to know about  $\omega c$ ?

You must write code to change the values of the state variables as needed. You may allocate as many temporary variables as needed to help you. For example, the variable  $c$  that reads in the next character of input is a temporary variable. These variables are deallocated at the bottom of the loop. Anything that you want to remember about  $\omega c$  when back at the top of the loop must be stored in the state variables.

**Initial Conditions:** Now that we have an idea of where we are going, we can start at the beginning knowing which direction to head. At the beginning of the computation, no input characters have been read and hence the prefix that has been read so far is the empty string  $\epsilon$ . Which values should the state variables have to establish the loop invariant?

When unsure, the initial conditions can be determined by working backwards. You know the loop invariant for when a single character has been read. Move backwards from there to see what conditions you want for the empty string.

**Ending:** When the input string has been completely read in, your loop invariant states that you know whether or not the string should be accepted. The steps outside the loop are to check whether the state variables have *accepting* or *rejecting* conditions and to accept or reject the string appropriately.

**Testing:** Try your algorithm by hand on a couple of examples. Consider both general input instances as well as special cases.

This completes the step for developing an iterative algorithm.

**Example - Mod Counting:** Consider the language

$$L = \{w \in \{0,1\}^* \mid [2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega)] \bmod 5 = 0\}$$

For example, with  $\omega = 10011010011001001$ ,  $2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega) = 2 \times 9 - 8 = 10 \bmod 5 = 0$ . Hence  $\omega \in L$ .

To construct an iterative program for this we follow the steps given above.

**Loop Invariant:** Suppose that you, as the computation, have read in some large prefix of the input string. The most obvious thing to remember about it would be the number of 0's and the number of 1's. However, with a large input these counts can grow arbitrarily large. Hence, with only bounded memory you cannot remember them. Luckily, the language is only concerned with these counts modulo 5, i.e. we count 1, 2, 3, 4, 0, 1, 2, 3, ... and  $28 \bmod 5 = 3$ . Even better we could remember  $[2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega)] \bmod 5$ .

**State Variables:** To remember this we will have one state variable denoted  $r$  that takes values from  $\{0, 1, 2, 3, 4\}$ . The loop invariant will be that for the prefix  $\omega$ ,  $[ 2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega) ] \bmod 5 = r$ .

**Maintaining Loop Invariant:** Knowing the count  $r$  for the current prefix  $\omega$  and knowing new character  $c$ , we can compute what  $r$  will be for the new prefix  $\omega c$ . If we read a 0, then the count increases by  $2 \bmod 5$ . If we read a 1, then it decreases by  $1 \bmod 5$ .

**Initial Conditions:** Initially, the number of 0's and 1's is both zero.  $[ 2 \times \text{zero} - \text{zero} ] \bmod 5 = 0$ . Hence, we set  $r$  to 0.

**Ending:** The language accepts the string if the count  $r$  is 0.

**Code:**

```

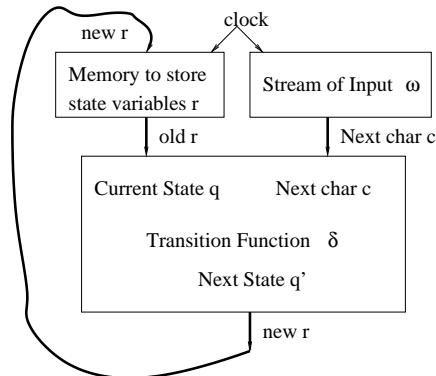
routine()
  allocate  $r \in \{0, 2, 1, 3, 4\}$ 
   $r = 0$ 
  loop
    % (loop invariant)            $[2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega)] \bmod 5 = r$ 
    exit when end of input
    allocate temp  $c$ 
    get( $c$ )                       % Reads next character of input
    if( $c = 0$ ) then                % Compute new values for state variables
       $r = r + 2 \bmod 5$ 
    else
       $r = r - 1 \bmod 5$ 
    end if
    deallocate  $c$ 
  end loop
  if( $r = 0$ ) then
    accept
  else
    reject
  end if
end routine

```

We will give many more examples below.

## 2.3 Mechanically Compiling an Iterative Program into a DFA

Consider any iterative program  $P$  with bounded memory and an input stream that has already been developed. We could build a circuit with *AND*, *OR*, *NOT* gates that computes it. See Figure ??.



We can also mechanically compile this iterative program  $P$  into a DFA  $M = \langle Q, \Sigma, \delta, s, F \rangle$  that solves the same problem. The steps are as follows.

### Steps To Convert:

**Alphabet  $\Sigma$ :** The input alphabet  $\Sigma$  of the DFA  $M$  will contain one “character” for each possible token that the program  $P$  may input. This may be  $\{0, 1\}$ ,  $\{a, b\}$ ,  $\{a, b, \dots, z\}$ , ASCII, or any other finite set of objects.

**Set of States  $Q$ :** When mapping a computation of program  $P$  to a computation of the DFA  $M$ , we only consider the computation of  $P$  at the points in which it is at the top of the loop. At these points in time, the *state* of the computation is given by the current values of all of  $P$ 's state variables. Recall that the temporary variables have been deallocated. The DFA will have a set of states  $Q$  consisting of all such possible states that program  $P$  might be in.

The program's loop invariant asserts that its state variables are remembering some key information about the prefix of the input read so far. The DFA remembers this same information using its states. Instead of assigning the meaningless names  $q_0, q_1, \dots, q_{|Q|}$  to the states, more insight is provided both for developing and for understanding the device by having names that associate some *meaning* to the state. For example, the state named  $q_{\langle \text{even, first } 0, \text{last } 1 \rangle}$  might remember that the number of 1's read so far is even, that the first character read was 0, and that the most recently read character was 1.

The set of states  $Q$  must contain a state for each such possible setting of the state variables. If the variables are allocated  $r$  bits of memory, then they can hold  $2^r$  different values. Conversely, with  $|Q|$  states a DFA can “remember”  $\log_2 |Q|$  bits of information. No more.

Formally, suppose that the state variables in  $P$  are  $v_1, v_2, \dots, v_k$  and that the variable  $v_i$  can take on values from the set  $S_i$ . Then the complete set of states is

$$\begin{aligned} Q &= S_1 \times S_2 \times \dots \times S_k \\ &= \{q_{\langle v_1=a_1, v_2=a_2, \dots, v_k=a_k \rangle} \mid a_1 \in S_1, a_2 \in S_2, \dots, a_k \in S_k\} \end{aligned}$$

For example, if the state variables remember a 6 bit string  $\alpha$ , two digits  $x, y \in \{0, 1, \dots, 9\}$ , and a character  $c \in \{a, b, c, \dots, g\}$  then the DFA would require  $2^6 \times 10 \times 10 \times 7$  states, one of which might be named  $q_{\langle \alpha=100110, x=4, y=7, c=f \rangle}$ .

Sometimes the set of states can be reduced. For example, any state that is not reachable by a path from the start state is clearly not useful. States that are “equivalent” in terms of future computation can be collapsed into one.

**Transition Function  $\delta$ :** The DFA's transition function  $\delta$  defines how the machine transitions from state to state. Formally, it is a function  $\delta : Q \times \Sigma \rightarrow Q$ . If the DFA's current state is  $q \in Q$  and the next input character is  $c \in \Sigma$ , then the next state of the DFA is given by  $q' = \delta(q, c)$ .

We define  $\delta$  as follows. Consider some state  $q \in Q$  and some character  $c \in \Sigma$ . Set program  $P$ 's state variables to the values corresponding to state  $q$ , assume the character read is  $c$ , and execute the code once around the loop. The new state  $q' = \delta(q, c)$  of the DFA is defined to be the state corresponding to the values of  $P$ 's state variables when the computation has reached the top of the loop again.

A great deal of complexity can be *hard wired* into the transition function  $\delta$  of a DFA. For this reason, we can allow the code within  $P$ 's loop to have any level of complexity. It can have if, select, or while statements. It can perform additions, multiplications, or any other operation. In fact, it can even perform uncomputable operations (like the *HALTING* problem) as long as the input to the operation depends only on the current values the program's state and temporary variables.

For example, suppose that program  $P$  when reading the character ' $f$ ' factors a 100 digit number stored in its state variables. The DFA would need at least  $2 \cdot 10^{100}$  different states. For each 100 digit number  $x$ , there are two states  $q_x$  and  $q_{\langle x, a, b \rangle}$  where  $x = a \times b$  is a suitable factoring. Using

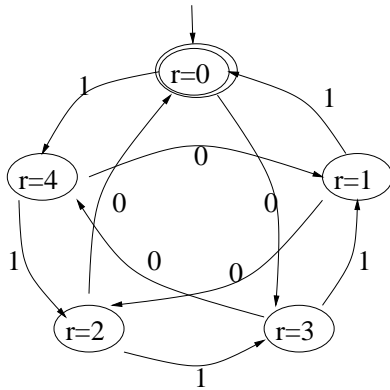
the transition function  $\delta(q_x, f) = q_{\langle x, a, b \rangle}$ , the DFA can factor in one time step. In fact, we can simply assume that the DFA automatically knows any information that can be deduced from the information that it knows.

**The Start State  $s$ :** The start state  $s$  of the DFA  $M$  is the state in  $Q$  corresponding to the initial values that  $P$  assigns to its state variables.

**Accept States  $F$ :** A state in  $Q$  will be considered an *accept* state of the DFA  $M$  if it corresponds to a setting of the state variables for which  $P$  accepts.

This completes the construction of a DFA  $M$  from an iterative program  $P$ .

**Example:** Consider again the iterative program developed for the language  $L = \{w \in \{0, 1\}^* \mid [2 \times (\# \text{ of } 0\text{'s in } \omega) - (\# \text{ of } 1\text{'s in } \omega)] \bmod 5 = 0\}$ . Following these steps produces the following DFA.



## 2.4 More Examples

We will now present DFAs and regular expressions for the following languages. We recommend that you attempt to do them yourself before reading on.

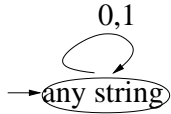
1.  $L = \emptyset$ .
2.  $L = \{0, 1\}^* - \{\epsilon\} =$  every string except  $\epsilon$ .
3.  $L = \{\omega \in \{0, 1\}^* \mid \omega \text{ begins with a } 1 \text{ and ends with a } 0\}$
4.  $L = 0^*1^*$ .
5.  $L = \{0^n1^n \mid n \geq 0\}$ .
6.  $L = \{w \in \{0, 1\}^* \mid \text{every } 3^{rd} \text{ character of } w \text{ is a } 1\}$ . Eg.  $10\underline{1}00\underline{1}11\underline{1}01\underline{1}0 \in L$ ,  $\epsilon \in L$ ,  $0 \in L$ , and  $10\underline{0} \notin L$ .
7.  $L = \{\omega \in \{0, 1\}^* \mid \omega \text{ has length at most three}\}$
8.  $L_{AND} = \{\omega \in \{0, 1\}^* \mid \omega \text{ has length at most three AND the number of } 1\text{'s is odd}\}$
9.  $L = \{\omega \in \{0, 1\}^* \mid \omega \text{ contains the substring } 0101\}$ . For example  $\omega = 1110101101 \in L$ , because it contains the substring, namely  $\omega = 111 \ 0101 \ 101$ .
10.  $L = \{J \in ASCII^* \mid J \text{ is a Java program that halts on input zero and } |J| \leq 10000\}$ .
11.  $L = \{\epsilon, 0, 010, 10, 11\}$ .
12. Dividing an integer by seven.

13. Adding two integers.

When developing a DFA for a complex language it is useful to follow all the steps outlined above. For simple languages all the steps should be thought about, but many of them need not be written down. In the examples below, think for yourself about the steps that have been skipped.

**Just Say No:**  $L = \emptyset$ .

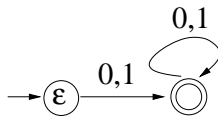
Nothing needs to be remembered about the prefix read so far because we simply reject every string.



(Zero information is remembered. The program uses zero bits for the state variables. The DFA needs  $2^{zero} = 1$  states.)

**Starting:**  $L = \{0, 1\}^* - \{\epsilon\} =$  every string except  $\epsilon$ .

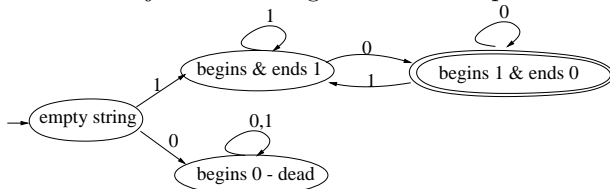
Here we must only remember whether or not we have seen any characters yet.



A regular expression representing  $L$  is  $\Sigma\Sigma^*$ .

**More Info:**  $L = \{\omega \in \{0, 1\}^* \mid \omega \text{ begins with a } 1 \text{ and ends with a } 0\}$

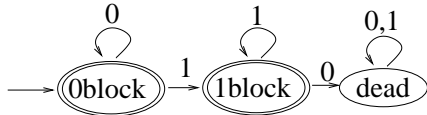
What the state variables must remember is which characters the prefix read so far begins and ends with (or that the prefix is empty and hence begins and ends with no characters). If the string begins with a 0, then there really is no reason to remember anything but this fact. In such a case, you enter what is sometimes referred to as a *dead* state. Once in such a state, the computation never leaves it and rejects the string when the computation is over.



A regular expression representing  $L$  is  $1\Sigma^*0$ .

**Different Phases:**  $L = 0^*1^*$ .

A string in this language contains a block of 0's followed by a block of 1's. You must remember which block or phase you are in. In the first phase you can read any number of 0's. When the first 1 is read, you must move to the second phase. Here you can read as many 1's as you like. If a 0 appears after the first 1 has been read, then the string is not in the language. Hence, the computation enters the dead state.



Note one should check special cases. For example, here either blocks could be empty, namely 000 and 111 are both valid strings. See how the DFA works for these cases.

**Finite Memory:**  $L = \{0^n1^n \mid n \geq 0\}$ .

**Phases:** A string in this language also must contain a block of 0's followed by a block of 1's. Hence, you must remember the same things remembered in the last example.

**Same Count:** However, for this language, the two blocks must have the same number of characters. For example,  $000111 \in L$  but  $00011 \notin L$ .

Suppose that a large block of 0's has been read in. What must you remember? Clearly, you need to know how many 0's have been read in. Otherwise, you will not know whether the number of 1's that follows is the same. Recall that you cannot go back and reexamine the block of 0's.

**Code:** An iterative program that works is as follows.

```

routine()
  allocate  $p \in \{0block, 1block, dead\}$ 
  allocate  $n_0, n_1 : int$ 
   $p = 0block$ 
   $n_0 = n_1 = 0$ 
  loop
    % (loop invariant)                 $p$  gives the phase,  $n_0$  the number of 0's,
    % and  $n_1$  the number of 1's

    exit when end of input
    allocate temp  $c$ 
    get( $c$ )                            % Reads next character of input
    % Compute new values for state variables

    if(  $p = 0block$  and  $c = 1$  ) then
       $p = 1block$ 
    end if
    if(  $p = 1block$  and  $c = 0$  ) then
       $p = dead$ 
    end if

    if( $c = 0$ ) then
       $n_0 = n_0 + 1$ 
    else
       $n_1 = n_1 + 1$ 
    end if

    deallocate  $c$ 
  end loop
  if( $p \neq dead$  and  $n_0 = n_1$ ) then
    accept
  else
    reject
  end if
end routine

```

**Infinite Set of States  $Q$ :** When converting this into a DFA, the first step is to construct the set of states  $Q$  consisting of all possible settings of the state variables. Here  $n_0$  is allocated so that it can take on any integer. Hence, the corresponding set of state  $Q$  is infinite.

This iterative program can be converted into a very good deterministic automata. However, because it does not have a finite set of states, it is not a deterministic *finite* automata.

**$0^n 1^n$  is Not Regular:** We will later prove that no DFA accepts this language. The intuition is that in order to know whether or not the blocks have the same size, the computation must remember how many 0's have been read. An input can have an arbitrarily large number of 0's. Hence, a DFA, with only a bounded amount of memory, is unable to distinguish between all of these different counts.

When told this, students always protest. Here are the two common statements.

**0\*1\***: Students give a DFA like the one above computing  $0^*1^*$  and say that this DFA accepts the language  $0^n1^n$ . I respond by saying that yes this DFA accepts all the strings that are in the language  $0^n1^n$ , but it also accepts many strings like 00111 that are not in the language. For a DFA to *accept* a language, it must accept a string if and *only if* it is in the language.

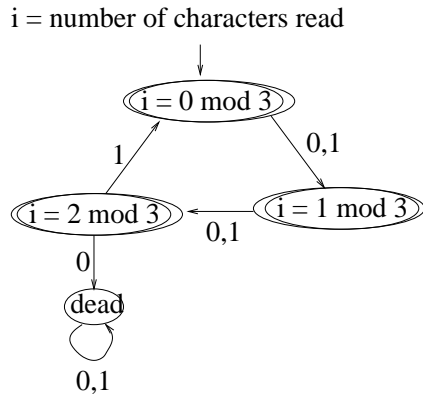
**The Computation Game:** Students also argue as follows. “Give me a string, say  $0^{1000}1^{1000}$ . I can construct a DFA with a finite number of states that determines whether or not the string is in the language.”

This, however, is not the way that the computation game is played. If you are claiming that a language or a computational problem is computable, then you must first give me a machine/algorithm that you claim solves the problem. Then I, as an adversary, get to choose an input string for which your machine must work. If you give me a machine with 20 states, I could give you a string with 21 0’s. However, if you give me a machine with a trillion states, I could give you a string with a trillion and one 0’s. Your machine must still work. That is how the game is played.

**Finite State Counting:** A DFA is unable to count arbitrarily high. The next is a second example demonstrating how a DFA is able to count modulo some fixed constant. The subsequent example shows how a DFA is also able to count up to some bounded amount.

**Mod Counting:**  $L = \{w \in \{0,1\}^* \mid \text{every } 3^{\text{rd}} \text{ character of } w \text{ is a } 1 \}$ . Eg.  $1010011110110 \in L$ ,  $\epsilon \in L$ ,  $0 \in L$ , and  $100 \notin L$ .

The computation must remember modulo 3 how many characters have been read in, so that it knows which characters are every third one. It also must remember whether the string has already failed to meet the conditions.



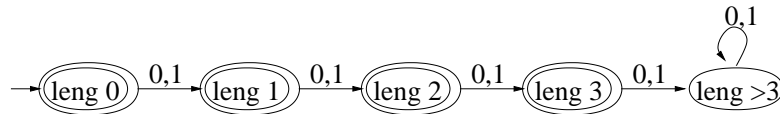
A regular expression representing  $L$  is  $(\Sigma\Sigma 1)^* (\epsilon \cup \Sigma \cup \Sigma\Sigma)$ . Here the  $(\Sigma\Sigma 1)^*$  ensures that every third character is a 1, but also forces the string’s length to be a multiple of three. The  $(\epsilon \cup \Sigma \cup \Sigma\Sigma)$  adds one zero, one, or two, more characters on the end so that the string can be any length.

**Counting to an Upper Bound:**  $L = \{\omega \in \{0,1\}^* \mid \omega \text{ has length at most three} \}$

The state variables must remember whether the length of the prefix read so far is 0, 1, 2, 3, or more than 3. An iterative program might use a state variable *leng* with values from  $\{0, 1, 2, 3, \text{more}\}$ . The code in the loop is simply

if(*leng*  $\neq$  *more*) then *leng* = *leng* + 1

A DFA would have the corresponding set of states and corresponding transitions.



A regular expression representing  $L$  is  $\epsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma$ .

**Intersection  $\approx$  AND:**  $L_{AND} = \{\omega \in \{0,1\}^* \mid \omega \text{ has length at most three AND the number of 1’s is odd} \}$

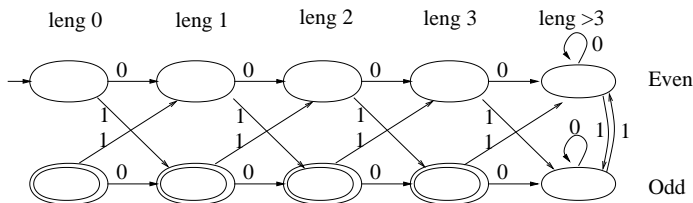
Membership within some languages, like this one, depends on more than one condition. This requires remembering more than one separate piece of information about the prefix. In an iterative program, this is done using separate state variables. The set of states of the DFA is constructed by taking the Cartesian product of these separate sets of values.

This iterative program will remember the length with the state variable  $l \in \{0, 1, 2, 3, \text{more}\}$  and the parity of the number of 1's with  $r \in \{\text{even}, \text{odd}\}$ . The set of states of the DFA is the cross product of these sets of values, namely the  $5 \times 2 = 10$  states  $Q = \{0, 1, 2, 3, \text{more}\} \times \{\text{even}, \text{odd}\} = \{q_{\langle 0, \text{even} \rangle}, q_{\langle 0, \text{odd} \rangle}, q_{\langle 1, \text{even} \rangle}, q_{\langle 1, \text{odd} \rangle}, q_{\langle 2, \text{even} \rangle}, q_{\langle 2, \text{odd} \rangle}, q_{\langle 3, \text{even} \rangle}, q_{\langle 3, \text{odd} \rangle}, q_{\langle \text{more}, \text{even} \rangle}, q_{\langle \text{more}, \text{odd} \rangle}\}$ .

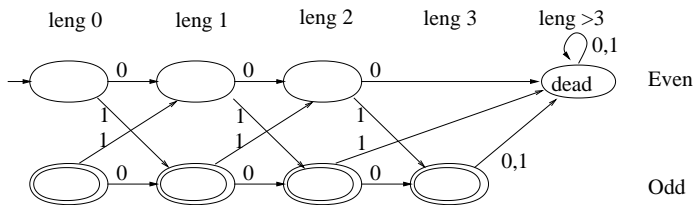
Within the loop, the program has separate code for updating  $l$  and  $r$ , namely  
 if( $l \neq \text{more}$ ) then  $l = l + 1$   
 $r = r + c \bmod 2$

The transition function  $\delta$  for the DFA changes the state mirroring how these two separate pieces of code change the values of the state variables, namely  $\delta(q_{\langle l, r \rangle}, c) = q_{\langle l+1, r+c \bmod 2 \rangle}$  unless it is already more,  $r=r+c \bmod 2$ .

Initially, the program sets  $l = 0$  and  $r = \text{even}$ . In the end, the program accepts if  $l \neq \text{more}$  AND  $r = \text{odd}$ . The start state  $s$  and the set of accept states  $F$  reflect these settings.



Or more simply



How would the DFA change for the language  $L_{OR} = \{\omega \in \{0, 1\}^* \mid \omega \text{ has length at most three OR the number of 1's is odd}\}$ ?

A regular expression representing the language of strings with an odd number of 1's is  $0^*10^*(10^*10^*)^*$ , representing  $L_{AND}$  is  $\{1, 01, 10, 100, 010, 001, 111\}$ , and representing  $L_{OR}$  is  $(\epsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma) \cup (0^*10^*(10^*10^*)^*)$ .

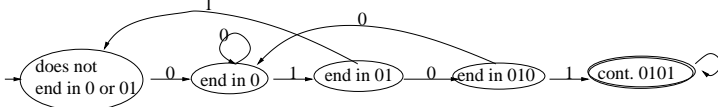
**Partial Progress:**  $L = \{\omega \in \{0, 1\}^* \mid \omega \text{ contains the substring } 0101\}$ . For example  $\omega = 1110101101 \in L$ , because it contains the substring, namely  $\omega = 111 \ 0101 \ 101$ .

If the prefix of the string read so far contains the substring 0101, this certainly should be remembered. However, if it does not, some *progress* may have been made towards this goal. A prefix grows one character to the right at a time. Therefore, if the prefix ends in 010 a lot of progress would have been made because reading a 1 would complete the 0101 task. The levels of progress are

1.  $q_{\langle \text{contains } 0101 \rangle}$
2.  $q_{\langle \text{ends in } 010, \text{ but does not contain } 0101 \rangle}$
3.  $q_{\langle \text{ends in } 01, \text{ but does not contain } 0101 \rangle}$
4.  $q_{\langle \text{ends in } 0, \text{ but does not end in } 010 \text{ and does not contain } 0101 \rangle}$
5.  $q_{\langle \text{does not ends in } 0 \text{ or in } 01 \text{ and does not contain } 0101 \rangle}$

Note how we were careful to ensure that these classifications are disjoint and cover all strings so that every possible prefix is contained in exactly one of these. The loop invariant asserts that when we have read some prefix, we are in the correct corresponding state.

When defining the code for the loop or the transition function for the DFA, you must determine when a character moves you forward to a higher level of progress and when you are moved backwards. As said, if the prefix  $\omega$  so far ends in 010 and you read the character  $c = 1$ , then progress is made because the prefix  $\omega 1$  ends in and hence contains 0101. On the other hand, if  $\omega$  ends in 010 and you read a  $c = 0$ , then this destroys the chance that the ending 010 will lead to a 0101. However, it is not a complete disaster. The prefix  $\omega 0$  now ends in 0100. This last 0 begins again the pattern 0101. Care needs to be taken to determine all of these transitions.



A regular expression representing  $L$  is  $\Sigma^*0101\Sigma^*$ .

**Finite Language:** Every language that contains only a finite number of strings can be computed by a DFA. The intuition is that every finite language has a longest string some length  $l_{max}$ . Hence, a DFA, which can remember any bounded amount of information, can either reject an input for being too long or remember it in its entirety.

Consider the example,

$$L = \{J \in ASCII^* \mid J \text{ is a Java program that halts on input zero and } |J| \leq 10000\}$$

The iterative algorithm simply reads in the input  $J$  and remembers the first 10000 characters. At the end of the computation, if the string is shorter than this, then the DFA, knowing the entire input, can magically know whether or not to accept it. If the string is longer, the DFA can simply reject it.

The DFA is constructed as follows. Its graph is a balanced tree of state nodes each representing a different input string.

$$Q = \{q_{\langle J \rangle} \mid J \in ASCII^* \text{ and } |J| \leq 10000\} \cup \{q_{\langle tooLong \rangle}\}$$

$$s = q_{\langle \epsilon \rangle}$$

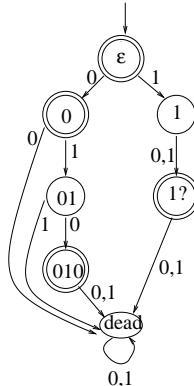
$$\delta(q_{\langle J \rangle}, c) = q_{\langle Jc \rangle}, \text{ if } |J| < 10000$$

$$= q_{\langle tooLong \rangle}, \text{ otherwise}$$

$$\delta(q_{\langle tooLong \rangle}, c) = q_{\langle tooLong \rangle}$$

$$F = \{q_{\langle J \rangle} \mid J \in ASCII^*, J \text{ is a Java program that halts on input zero, and } |J| \leq 10000\}$$

**Eliminating States:** As said, languages containing a finite number of strings can be accepted by a DFA whose graph is a complete balanced tree. However, for some finite languages some of these states can be eliminated, because they contain the same information. Consider for example the language  $L = \{\epsilon, 0, 010, 10, 11\}$ . The following DFA accepts it.



**Dividing:** Dividing an integer by seven is reasonably complex algorithm. It would be surprising if it could be done by a DFA. However, it can. Consider the language  $L = \{w \in \{0, 1, \dots, 9\}^* \mid w \text{ is divisible by } 7 \text{ when viewed as an integer in normal decimal notation}\}$ .

Consider the standard algorithm to divide an integer by 7. Try it yourself on say 39462. You consider the digits one at a time. After considering the prefix 394, you have determined that 7 divides into 394, 56 times with a remainder of 2. You likely have written the 56 above the 394 and the 2 at the bottom of the computation.

The next step is to “bring down” the next digit, which in this case is the 6. The new question is how 7 divides into 3946. You determine this by placing the 6 to the right of the 2, turning the 2 into a 26. Then you divide 7 into the 26, learning that it goes in 3 times with a remainder of 5. Hence, you write the 3 next to the 56 making it a 563 and write the 3 on the bottom. From this we can conclude that 7 divides into 3946, 563 times with a remainder of 5.

We do not care about how many times 7 divides into our number, but only whether or not it divides evenly. Hence, we remember only the remainder 2. One can also use the notation  $395 = 2 \pmod{7}$ . To compute  $3956 \pmod{7}$ , we observe that  $3956 = 395 \cdot 10 + 6$ . Hence,  $3956 \pmod{7} = (395 \cdot 10 + 6) \pmod{7} = (395 \pmod{7}) \cdot 10 + 6 \pmod{7} = (2) \cdot 10 + 6 \pmod{7} = 26 \pmod{7} = 3$ .

More formally, the algorithm is as follows. Suppose that we have read in the prefix  $\omega$ . We store a value  $r \in \{0, 1, \dots, 6\}$  and maintain the loop invariant that  $\omega = r \pmod{7}$ , when the string  $\omega$  is viewed as an integer. Now suppose that the next character is  $c \in \{0, 1, \dots, 9\}$ . The current string is then  $\omega c$ . We must compute  $\omega c \pmod{7}$  and set  $r$  to this new remainder in order to maintain the loop invariant. The integer  $\omega c$  is  $(\omega \cdot 10 + c)$ . Hence, we can compute  $r = \omega c \pmod{7} = (\omega \cdot 10 + c) \pmod{7} = (\omega \pmod{7}) \cdot 10 + c \pmod{7} = r \cdot 10 + c \pmod{7}$ . The code for the loop is simply  $r = r \cdot 10 + c \pmod{7}$ .

Initially, the prefix read so far is the empty string. The empty string as an integer is 0. Hence, the initial setting is  $r = 0$ . In the end, we accept the string if when viewed as an integer it is divisible by 7. This is true when  $r = 0$ . This completes the development of the iterative program.

The DFA to compute this will have seven states  $q_0, \dots, q_6$ . The transition function is  $\delta(q_r, c) = q_{\langle r \cdot 10 + c \pmod{7} \rangle}$ . The start state is  $s = q_0$ . The set of accept states is  $F = \{q_0\}$ .

**Adding:** The input to the problem consists of two integers  $x$  and  $y$  represented as strings of digits. The output is the sum  $z$  also represented as a string of digits. The standard algorithm is a classic iterative algorithm that depends on a loop invariant. As a reminder, add the following numbers.

```

1896453
+ 7288764
-----

```

The input can be view as a stream if the algorithm is first given the lowest digits of  $x$  and of  $y$ , then the second lowest, and so on. The algorithm outputs the characters of  $z$  as it proceeds. The only memory required is a single bit to store the carry bit. Because of these features, the algorithm can be modeled as a DFA.

**Alphabets  $\Sigma$  and  $\Sigma_{out}$ :** A single token inputted in the Adding example, consists of a pair of digits,  $\langle x_i, y_i \rangle$ , one from  $x$  and one from  $y$ . The alphabet is

$$\Sigma = \{0..9\} \times \{0..9\} = \{\langle x_i, y_i \rangle \mid x_i, y_i \in \{0..9\}\}$$

The input  $x = 312$ ,  $y = 459$  is given as the string of characters  $\langle x_1, y_1 \rangle \langle x_2, y_2 \rangle \dots \langle x_n, y_n \rangle = \langle 2, 9 \rangle \langle 2, 5 \rangle \langle 3, 4 \rangle$ .

The algorithm will output the digits  $z_i$  of the sum  $z = x + y$ . They will be outputted from the low to the high order digits. Hence, output alphabet  $\Sigma_{out}$  is  $\{0, 1, \dots, 9\}$ .

**Code:**

```

routine()
  allocate carry ∈ {0, 1}
  carry = 0
  loop
    % ⟨loop invariant⟩ The sum of the characters read has been calculated and outputted.
    % The state variable carry contains the final carry.
    exit when end of input
    allocate temp variables xi, yi, zi, and s
    get(⟨xi, yi⟩)
    s = xi + yi + carry
    zi = low order digit of s
    carry = high order digit of s
    put(zi)
    deallocate temp variables xi, yi, zi, and s
  end loop
  if(carry = 1) then
    put(carry)
  end if
end routine

```

**DFA with Output:** One can also construct a DFA that produces a string of output each time step. To do this the transition function is changed to  $\delta : Q \times \Sigma \rightarrow Q \times \Sigma_{out}^*$ , where  $\Sigma_{out}$  is the output alphabet. If the DFA's current state is  $q \in Q$  and the next input character is  $c \in \Sigma$ , then the DFA outputs the string  $\alpha$  and its next state is  $q'$ , where  $\delta(q, c) = \langle q, \alpha \rangle$ . In a graph representation of such a DFA, the edge  $\langle q, q' \rangle$  would be labeled with  $\langle c, \alpha \rangle$ .

**The DFA:**

$$Q = \{q_{(carry=0)}, q_{(carry=1)}\}.$$

$$s = q_{(carry=0)}$$

$$\delta(q_{(carry=c)}, \langle x_i, y_i \rangle) = \langle q_{carry=c'}, z_i \rangle$$

where  $c'$  is the high order digit and  $z_i$  is the low order digit of  $x_i + y_i + c$ .

We do not need accept states, because the problem does not require the machine to accept or reject the input.

**Calculator:** Invariants can be used to understand a computer system that, instead of simply computing one function, continues dynamically to take in inputs and produce outputs. See Chapter ?? for further discussion of such systems.

In a simple calculator, the keys are limited to  $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$ . You can enter a number. As you do so it appears on the screen. The  $+$  key adds the number on the screen to the accumulated sum and displays the sum on the screen. The  $clr$  key resets both the screen and the accumulator to zero. The machine only can store positive integers from zero to 99999999. Additions are done mod  $10^8$ .

**algorithm** *Calculator*()

⟨*pre-cond*⟩: A stream of commands are entered.

⟨*post-cond*⟩: The results are displayed on a screen.

```

begin
  allocate accum, current ∈ {0..108 - 1}
  allocate screen ∈ {showA, showC}
  accum = current = 0
  screen = showC
  loop

```

**(loop-invariant):** The bounded memory of the machine remembers the current value of the accumulator and the current value being entered. It also has a boolean variable which indicates whether the screen should display the current or the accumulator value.

```

get(c)
if( c ∈ {0..9} ) then
    current = 10 × current + c mod 108
    screen = showC
else if( c = ' +' ) then
    accum = accum + current mod 108
    current = 0
    screen = showA
else if( c = ' clr' ) then
    accum = 0
    current = 0
    screen = showC
end if
if( screen = showC ) then
    display(current)
else
    display(accum)
end if
end loop
end algorithm

```

The input is the stream keys that the user presses. It uses only bounded memory to store the eight digits of the accumulator and of the current value and the extra bit. Because of these features, the algorithm can be modeled as a DFA.

**Set of States:**  $Q = \{q_{\langle acc, cur, scr \rangle} \mid acc, cur \in \{0..10^8 - 1\} \text{ and } scr \in \{showA, showC\}\}$ . Note that there are  $10^8 \times 10^8 \times 2$  states in this set so you would not want to draw the diagram.

**Alphabet:**  $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$ .

**Start state:**  $s = q_{\langle 0, 0, showC \rangle}$ .

**Transition Function:**

- For  $c \in \{0..9\}$ ,  $\delta(q_{\langle acc, cur, scr \rangle}, c) = q_{\langle acc, 10 \times cur + c, showC \rangle}$ .
- $\delta(q_{\langle acc, cur, scr \rangle}, +) = q_{\langle acc + cur, cur, showA \rangle}$ .
- $\delta(q_{\langle acc, cur, scr \rangle}, clr) = q_{\langle 0, 0, showC \rangle}$ .

**Syntax Preprocessor:** Your task here is to define a finite state transducer that will read strings representing  $C$  programs and write onto the output tape the same string/program with all comments removed. Comments in a  $C$  program start with  $/*$  and continue until the next  $*/$ . For example,

```

input:
    if(x=5) /* This is a "great" comment */ then /* add 1 */ x=x+1
output:
    if(x=5) then x=x+1

input:
    x=5; /* Nested comments /* do not */ work like you might want. */ y=7;
output:
    x=5; work like you might want. */ y=7;

input:

```

```

    x=5; /* Comments that do not end should still be deleted.
output:
    x=5;

```

Comments cannot begin within a string literal, i.e. inside a quoted piece of text like "the /\* here is not the start of a quote."

```

input:
    printf("This is not a /* comment */ ok"); /* This is */
output (of your machine, not of printf):
    printf("This is not a /* comment */ ok");

```

Do not worry about the syntax of the *C* program. Just treat it as a string that may have some occurrences of /\* ... \*/ inside it. To really simplify things, assume that the only symbols in the input string are the following:

```

/ (slash)
* (star)
" (open quote or close quote - use same symbol)
a,b (to represent other characters)

```

The *C* compiler has a preprocessor option that strips away comments. You can try it using

```
cc -E file
```

**The Iterative Algorithm:** Having the computation remember the last character read makes the code easier, but then there would be more states. My algorithm remembers only whether the last character is special or not.

```

routine()
    allocate mode ∈ {none, comment, quote}
    allocate lastchar ∈ {notspecial, special}
    mode = none
    lastchar = notspecial
    loop
        % {loop invariant}
        % What the computation remembers about the prefix of the string read in so far is
        % whether at this point in the string it is with in a comment or quote,
        % and whether the last character was a "special".
        % If you are in neither a comment nor a quote,
        % then '/' is "special", because it might start '/*'.
        % If you are in a comment, then '*' is "special", because it might start '*/'.
        % If you are in a quote, then there are no special characters.
    exit when end of input
    allocate temp variables c and thischar
    get(c)

    % Determine whether this character c is special
    if((mode = none and c = '/') or (mode = comment and c = '*')) then
        thischar = special
    else
        thischar = notspecial
    endif

```

```

% Check if comment before outputting
if( mode = none and lastchar = special and c = ' *') mode = comment

% Output characters. Note "special" characters always get printed next step.
if( mode ≠ comment ) then
    if( lastchar = special ) output('/')
    if( thischar ≠ special ) output(c)
endif

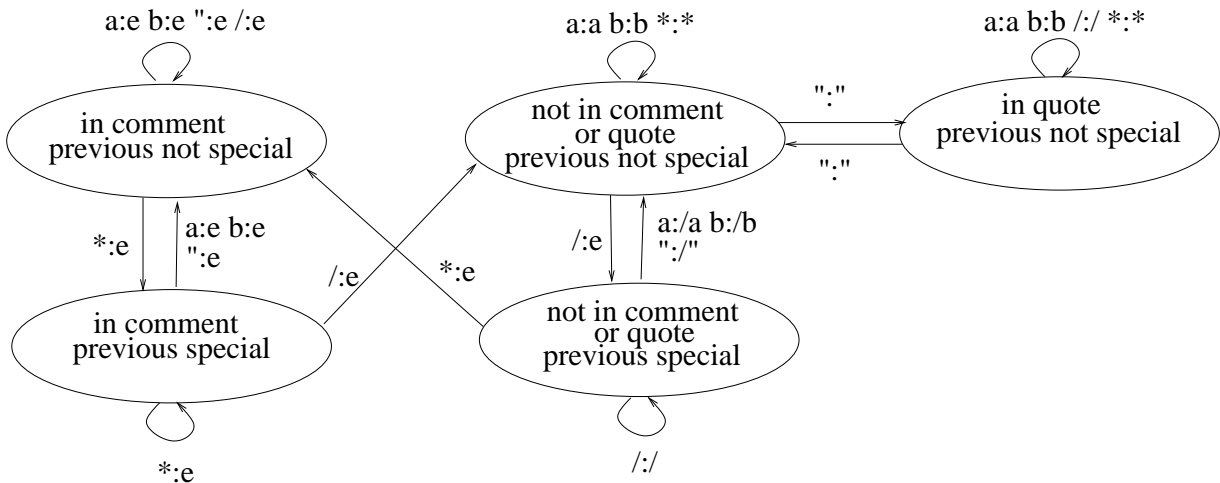
% Change mode
if( mode = comment and lastchar = special and c = '/' ) mode = none
if( mode = quote and c = '"' ) mode = none
if( mode = none and c = '"' ) mode = quote

lastchar = thischar
deallocate temp variables c and thischar
end loop

% Output last special character.
if(mode ≠ comment and lastchar = special output('/')
end routine

```

**A DFA with Output:** Recall that a DFA that produces an output has a transition function  $\delta : Q \times \Sigma \rightarrow Q \times \Sigma_{out}^*$ .



**Harder Problem:** Above we implied that comments that begin with a `/*` yet do not contain an ending `*/` should be deleted. Suppose that we change the description of the problem so that such comments are NOT deleted.

```

input:
  x=5; /* Comments that do not end should not be deleted.
output:
  x=5; /* Comments that do not end should not be deleted.

```

If possible construct a DFA for this new problem. If it is not possible give intuition as to why. There is NO DFA to solve this problem. The reason is that you must remember the comment so that you can print it out if the line ends without a closing `*/`. You cannot remember this with only a finite memory.

## Chapter 3

# Converting NFA to DFA using the Clones and the Iterative Program Levels of Abstraction

**Clones Level of Abstraction:** A computation on an NFA can be viewed as a parallel computation in which many sub-processors are computing simultaneously. We will refer to these sub-processors as *clones* because they are all identical and because one is able to split into many. Given an NFA  $M_{NFA}$  and an input string  $\alpha$ , the clones compute as follows.

**Starting:** Initially a single clone starts on the start state  $s_{NFA}$  of  $M_{NFA}$ .

**Operations:** Clones are able to repeatedly do either of following:

- Read the next input character and follow an edge labeled with the character read or
- Follow an edge labeled  $\epsilon$ .

**Splitting:** Whenever a clone has a choice it splits into one clone for each option.

- If current state of a clone has two edges leaving it labeled  $\epsilon$ , then the clone will split into three copies. One will read the next character, one will follow the one  $\epsilon$  edge, and one will follow the other  $\epsilon$  edge.
- Suppose a clone reads the next input character and it is a  $c$ . If his current state has five edges leaving it labeled  $c$ , then the clone will split into five copies, one for each edge.

**Separate Inputs:** Because these clones are acting independently, they need different copies of the input. When a clone reads the *next* input, he gets the next character that he has not seen.

**Accepting The Input:** We say that this computation of the NFA  $M_{NFA}$  on input  $\alpha$  accepts if and only if at least one of the clones has read the entire input and is on an accept state.

**Synchronizing the Clones:** In asynchronous parallel computations, the sub-processors, having their own clocks, run at their own speeds. Sometimes, if they want to share information, they must synchronize. This involves some sub-processors waiting for others to catch up.

**NFA to DFA:** A DFA can only read the next character of the input once. If we are to simulate an NFA computation on a DFA, then all the clones must read the next character of the input all at the same time. If two clones follow a different number of  $\epsilon$  edges, then they are not ready to read the next character at the same time. Hence, they must synchronize before reading it.

**Super-Step:** We will call the time from one synchronization to the next a *super-step*. For  $i \geq 1$ , the  $i^{th}$  super-step consists of the following sub-steps in the following order:

- All the clones simultaneously read the  $i^{th}$  input character.

- If the character read is a  $c$ , then each clone must first follow one edge labeled  $c$ .
- Each clone can then follow as many edges labeled  $\epsilon$  as it likes, i.e. zero or more.

**$\epsilon$  Moves First:** One may ask why a clone within a super-step is not able to follow  $\epsilon$  edges before reading the next character. The answer is that one could define a super-step to allow this, however, we have chosen not to. This restriction does not restrict what the clones are able to do for the following reason. Suppose within one super-step a clone reads the input character  $c$  and follows a few  $\epsilon$  edges. Suppose that in the next super we allow it to follow a few more  $\epsilon$  edges before reading the next input character. We can keep the sequence of sub-steps the same, but redefine the super-steps to have all of these  $\epsilon$  moves within the first super-step. This ensures that the second super-step begins with reading an input character.

**Before the First Super-Step:** At the beginning of the computation, a clone may follow a number of  $\epsilon$  edges from the start state before reading the first input character. Because of this, we allow a zeroth super-step within which each clone can follow as many  $\epsilon$  edges as it likes.

**At Most One Clone per State:** As stated, a clone splits whenever it has a choice. The following are ways of getting rid of excess clones.

**Dying:** A clone dies if it reads a character for which there are no edges leaving the clone's current state.

**Merging:** Suppose that two clones are on the same state at the end of a super-step. Note they have read the same prefix  $\omega$  of the input. Even if their pasts are completely different, everything that they remember about their past and about the prefix  $\omega$  is characterized by which state of the NFA they are on. Being in the same state, what they remember is the same. Assuming that they have free will, what they choose to do in the future may be different. However, their potential futures are the same. Hence, at the moment, these clones are indistinguishable. Because of this, we will merge them into one clone. (If given a choice, they will split again.) The advantage of this is that at the beginning of each super-step each state of the NFA will either have zero or one clone.

**Super-State:** At any given point in time during a computation on a parallel machine, each sub-processor is in some internal state. The *super-state* of the entire parallel machine is determined by which state each of its sub-processors is in.

Similarly during a computation on an NFA, each clone is in one of the NFA states. The state of the entire NFA computation is determined by which sub-state each clone is in. There is nothing distinguishing between the clones. Hence, we do not need to say that clone Joe is in NFA state  $q_{38}$ . We only need to know how many clones are in each of the NFA's states. Because clones in the same NFA state merge into one clone, each NFA state contains either zero or one clone. Therefore, we can describe the super-state of the NFA computation by specifying for each NFA state, whether or not it contains a clone. An equivalent way of doing this is by specifying the set of NFA states  $S = \{q \mid \text{a clone is on state } q\} \subseteq Q_{NFA}$ . The set of possible super-states is the set of all subsets of  $Q_{NFA}$ , which is denoted either  $Power(Q_{NFA})$  or  $2^{Q_{NFA}}$ . The second notation refers to the fact that there are  $2^{|Q_{NFA}|}$  different subsets of  $Q_{NFA}$ .

**NFA to Iterative Program:** We can use a simple iterative program to simulate and to better understand the computation of the clones on the NFA. The steps in constructing any iterative program are the following.

1. We must decide what data structures or "state" variables are needed.
2. The loop invariant states what is remembered about the prefix  $\omega$  of the input read so far. We must decide what will be remembered and how the "state" variables store this information.
3. We must figure out how this loop invariant is to be maintained as the next input character is read.

4. We must figure out how this loop invariant is initially obtained before the first input character is read.
5. When the entire input has been read in, we must determine based solely on the contents of the “state” variables whether or not to accept the input.

We will now consider each of these steps.

**“State” Variables:** The state variables of our iterative program must store the current super-state of the NFA. This consists of a subset  $S \subseteq Q_{NFA}$  of the NFA states.

**The Loop Invariant:** For  $i \geq 0$ , after  $i$  times around the loop, the state variable  $S$  is storing the set  $\{q \mid \text{a clone is on state } q \text{ after the } i^{th} \text{ super-step, i.e. before reading the } i + 1^{st} \text{ input character}\}$  =  $\{q \mid \text{there is a path from the start state of the NFA to state } q \text{ labeled } \omega, \text{ where } \omega \text{ is the prefix consisting of the first } i \text{ characters of the input}\}$ .

**Maintaining the Loop Invariant:** Suppose that at the beginning of the  $i^{th}$  super-step,  $S_{old}$  specifies which NFA states contain clones. Suppose that the next character read is a  $c$ . We must determine the locations  $S_{new}$  of the clones after this super-step. We know that the clones must follow an edge labeled  $c$  followed by any number of  $\epsilon$  moves. Hence  $S_{new}$  is computed as follows.

$\delta(q, c)$ : A clone on NFA state  $q$  after reading the input character  $c$  will split into clones which will move onto the NFA states in  $\delta(q, c) = \{q' \mid \text{the NFA has an edge from state } q \text{ to state } q' \text{ labeled } c\}$ .

$\delta(S_{old}, c)$ : After reading the  $c$ , the clones on states in  $S_{old}$  will split and move to the states in  $\delta(S_{old}, c) = \cup_{q \in S_{old}} \delta(q, c) = \{q' \mid \text{the NFA has an edge from some state } q \in S_{old} \text{ to state } q' \text{ labeled } c\}$ .

**Closure:** A set is said to be closed under an operation if for any object (objects) the result of the operation is still in the set. For example, the set of integers is closed under addition.

The closure of a set  $R$  under an operation is defined to be the smallest set  $E(R)$  that contains the original  $R$  and is closed under the operation. For example, the closure  $E(R)$  of the set  $R = \{1\}$  under additions is the set of positive integers. 1 must be in the closure because it is in  $R$ . Because 1 is in the closure,  $1 + 1 = 2$  must be in it. Because 1 and 2 are in the closure,  $1 + 2 = 3$  must be in it, and so on. There is no reason for 3.5 or -4 to be in the closure. Hence, they are not.

$E(R)$ : Given a set  $R \subseteq Q_{NFA}$  of NFA states, define the set  $E(R)$  to be those states reachable from  $R$  by zero or more  $\epsilon$  moves, namely  $E(R) = \{q'' \mid \text{there is a path of any number of } \epsilon \text{ edges from } q' \text{ to } q'' \text{ where } q' \in R\}$ .

$S_{new} = E(\delta(S_{old}, c))$ :  $S_{new}$  consists of the NFA states that can be reached from  $S_{old}$  with one  $c$  edge followed by any number of  $\epsilon$  edges.

**The Starting Super-State:** The loop invariant states that after 0 times around the loop, the state variable  $S$  is storing the set  $\{q \mid \text{a clone is on state } q \text{ after the } 0^{th} \text{ super-step, i.e. before reading the } 1^{st} \text{ input character}\}$ . Before reading this first input character, the clones can travel from the NFA’s start state  $s_{NFA}$  along any number of  $\epsilon$  edges. Hence, the set of NFA states that the clones might be in is  $E(s_{NFA})$ . Setting  $S = E(s_{NFA})$  makes the loop invariant initially true.

**Finishing:** When the entire input has been read, the loop invariant states that the state variable  $S$  is storing the set  $\{q \mid \text{a clone is on state } q \text{ after the last super-step}\}$ . We are to accept the input if there is a clone on an accept state of the NFA. Hence, we accept if  $S$  is such that  $S \cap F_{NFA} \neq \emptyset$ .

**The Iterative Program:** Combining these pieces gives the following iterative program.

```

routine()
  allocate “state” variable S
  S = E(sNFA)
  loop
    % {loop invariant}           Clones are on the NFA states specified in S
  exit when end of input

```

```

    get(c)                                % Reads next character of input
    S = E( $\delta(S, c)$ )
end loop
if( $S \cap F_{NFA} \neq \emptyset$ ) then
    accept
else
    reject
end if
end routine

```

**NFA to DFA:** To construct a DFA from an NFA one could convert the NFA into the iterative program and then convert the iterative program into a DFA as we have learned. Alternatively, one could construct the DFA directly.

**Formal Translation:** Given an NFA  $M_{NFA} = \langle Q_{NFA}, \Sigma, \delta_{NFA}, s_{NFA}, F_{NFA} \rangle$  construct a DFA  $M_{DFA} = \langle Q_{DFA}, \Sigma, \delta_{DFA}, s_{DFA}, F_{DFA} \rangle$  as follows.

- $Q_{DFA} = \text{powerset}(Q_{NFA}) = 2^{Q_{NFA}} = \{q_S \mid S \subseteq Q_{NFA}\}$ .
- $\delta_{DFA}(q_{S_{old}}, c) = q_{S_{new}}$ , where  $S_{new} = E(\delta(S_{old}, c))$  consists of the NFA states that can be reached from  $S_{old}$  with one  $c$  edge followed by any number of  $\epsilon$  edges.
- $s_{DFA} = q_{S_{start}}$ , where  $S_{start} = E(s_{NFA})$  consists of the NFA states that can be reached from the NFA's start state  $s_{NFA}$  with any number of  $\epsilon$  edges.
- $F_{DFA} = \{q_S \mid S \cap F_{NFA} \neq \emptyset\}$  consist of those super-states that have clones on one of the NFA's accepts states.

**A Exponential Blow up in Size:** Note that if the NFA has  $|Q_{NFA}|$  states, then the DFA constructed this way has  $2^{|Q_{NFA}|}$  states. Some times much smaller DFA can be constructed. Some times the smallest DFA is really exponentially bigger than the smallest NFA for the same language.

**Constructing the DFA in a Practical Way:** If you are given a picture of an NFA and you must quickly draw a picture of an equivalent DFA, the above definition of the DFA is not that useful. Instead follow the following practical steps.

1. The first step is to consider one at a time each tuple  $\langle q, c \rangle$ , where  $q \in Q_{NFA}$  is a state of the NFA and  $c \in \Sigma$  is a character. Assume that at the beginning of a super-step there is only one clone and he is one state  $q$  and that he reads the character  $c$ . Determine the set  $E(\delta(q, c))$  of states that clones will be on at the end of the super-step. Recall that the clone must take a  $c$  move, possibly followed by any number of  $\epsilon$  moves. Put all of these results into a table.
2. The set of states of the DFA is stated to be  $Q_{DFA} = 2^{Q_{NFA}}$ , however, it may not need all these states. A primary reason for not needing a state  $q_S$  is because it is not reachable from the start state. The following technique considers only DFA states that are reachable from the DFA's start state.

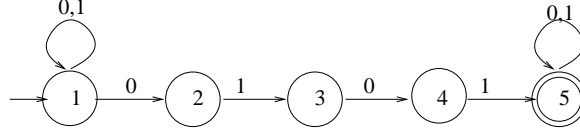
Start by writing down the start state  $s_{DFA} = E(s_{NFA})$  of  $M_{DFA}$ . Repeat the following until every state of  $M_{DFA}$  has one edge leaving it for each character of  $\Sigma$ .

- Choose an a super-state  $q_S$  of your  $M_{DFA}$  and a character  $c$  for which your  $M_{DFA}$  does not have a  $c$  transition.
  - Put a clone on the NFA states in the set  $S$  corresponding to DFA state  $q_S$ . Have them take a super-step, i.e. make a  $c$  followed by as many  $\epsilon$  transitions as they like.
  - Determine where the clones will be as follows.  $S_{new} = E(\delta(S, c)) = \cup_{q \in S} E(\delta(q, c))$ . You can read each  $E(\delta(q, c))$  from your table.
  - If your DFA does not already contain the super-state  $q_{S_{new}}$ , then add it.
  - Add the transition edge from  $q_S$  to  $q_{S_{new}}$  labeled  $c$ .
3. Determine which states of your  $M_{DFA}$  are accept states. Remember  $q_S$  is an accept state if  $S$  contains an NFA accept state.

4. The  $M_{DFA}$  that you constructed may still be too big. Is there any obvious way of simplifying  $M_{DFA}$ ? Why does  $M_{DFA}$  still work with this simplification?

Recall what was said about the closure of a set under an operation. The above technique is forming the closure under the  $\delta$  operation starting with the start symbol  $s_{DFA}$ .

**An Example:** Consider the NFA  $M_{NFA}$ :



1. What language does it accept?

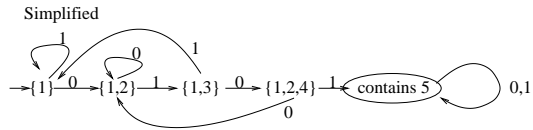
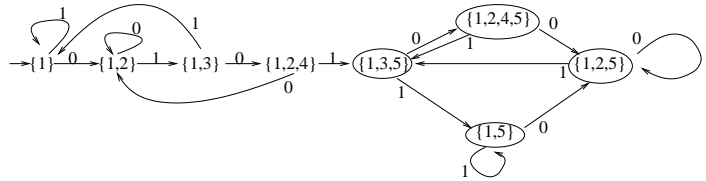
- Answer:  $L = \{w \in \{0,1\}^* \mid w \text{ contains the substring } 0101 \}$ .

2. Fill in the table of  $E(\delta(q, c))$ .

- Answer:

$Q \setminus \Sigma$	0	1
1	{1, 2}	{1}
2	$\emptyset$	{3}
3	{4}	$\emptyset$
4	$\emptyset$	{5}
5	{5}	{5}

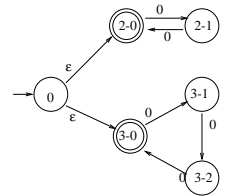
3. Construct the DFA  $M_{DFA}$  for the states that are reachable. Is there any obvious way of simplifying it?



- Answer:

4. Does this DFA look familiar?

- Answer: This DFA is in the notes.



**Another Example:** Consider the NFA  $M_{NFA}$

1. What language does it accept?

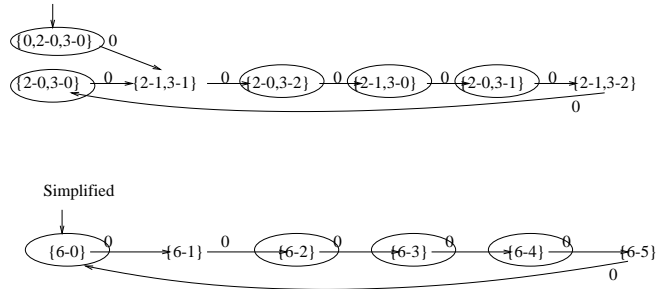
- Answer:  $L(m) = \{w \in \{0\}^* \mid |w| = 0 \text{ mod } 2 \text{ or } |w| = 0 \text{ mod } 3\}$

2. Fill in the table of  $E(\delta(q, c))$ .

- Answer:

$Q \setminus \Sigma$	0
0	$\emptyset$
2-0	$\{2-1\}$
2-1	$\{2-0\}$
3-0	$\{3-1\}$
3-1	$\{3-2\}$
3-2	$\{3-0\}$

3. Construct the DFA  $M_{DFA}$  for the states that are reachable. Is there any obvious way of simplifying it?



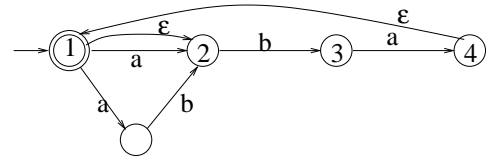
- Answer:

4. Make a DFA  $M''$  for the following language  $L'' = \{w \in \{0\}^* \mid |w| \text{ is } 0, 2, 3, \text{ or } 4 \pmod 6\}$ .

- Answer: See above fig.

5. Is there a connection between  $M'$  and  $M''$ ? Why? (If you are in the mood, see references for the Chinese Remainder Theorem of number theory.)

- Answer: They compute the same language. The Chinese Remainder Theorem tells you for example that if  $x = 1 \pmod 2$  and  $x = 2 \pmod 3$  then  $x = 5 \pmod 6$ .

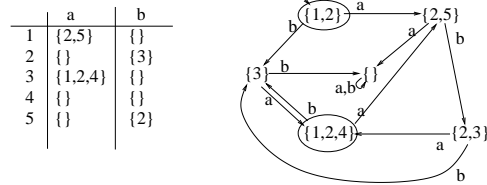


**Another Example:** Consider the NFA  $M_{NFA}$

1. What language does it accept?

- Answer:  $((\epsilon \cup a \cup ab)ba)^*$  or  $(ba \cup aba \cup abba)^*$

2. Construct an equivalent DFA  $M_{DFA}$ .



- Answer:

## Chapter 4

# Overview of Results

For each of the following theorems, give a two or three sentence sketch of how the proof goes or why it is not true.

1. Every DFA  $M$  can be converted into an equivalent NFA  $M'$  for which  $L(M) = L(M')$ .
  - Answer: True. A DFA  $M$  is automatically also a NFA.
2. Every NFA  $M$  can be converted into an equivalent DFA  $M'$  for which  $L(M) = L(M')$ .
  - Answer: True.  $M'$  computing is like having many clones computing on  $M$ . The current state of  $M'$  is the subset of the states of  $M$  that the clones are currently on.
3. Every DFA  $M$  can be converted into an equivalent TM  $M'$  for which  $L(M) = L(M')$ .
  - Answer: True. A TM  $M'$  can simulate a DFA  $M$  simply by not using its tape except for reading the input in.
4. Every TM  $M$  can be converted into an equivalent DFA  $M'$  for which  $L(M) = L(M')$ .
  - Answer: False. A TM can compute  $L = \{0^n 1^n \mid n \geq 0\}$  and a DFA cannot.
5. Every regular expression  $R$  can be converted into an equivalent NFA  $M$  for which  $L(R) = L(M)$ .
  - Answer: True. NFA are closed under union, concatenation, and kleene star. Hence, the NFA  $M$  is built up by following these operations within the  $R$ .
6. Every DFA  $M$  can be converted into an equivalent regular expression  $R$  for which  $L(M) = L(R)$ .
  - Answer: True. This is the complex algorithm in which states of the NFA are removed one at a time and edges are allowed to be labeled with regular expressions.
7. Every NFA  $M$  can be converted into an equivalent one  $M'$  that has a single accept state.
  - Answer: True. Given  $M$ , construct  $M'$  by adding a new state  $f$  that will be the only accept state of  $M'$ . Add an  $\epsilon$  transition from each accept state of  $M$  to  $f$ .
8. The set of languages computed by DFA is closed under complement.
  - Answer: True. Given a DFA  $M$  that computes  $L$ , construct  $M'$  by switching the accept and not accept states of  $M$ .  $L(M') = \overline{L(M)}$ .

9. The set of languages computed by NFA is closed under complement.

- Answer: True. Given a NFA  $M$  that computes  $L$ , constructing  $M'$  for which  $L(M') = \overline{L(M)}$  cant be done directly. Instead, convert the NFA  $M$  into a DFA  $M''$  as done above and then construct from the DFA  $M''$  the DFA  $M'$  by switching the accept state, so that  $L(M') = \overline{L(M'')} = \overline{L(M)}$ .

## Chapter 5

# Pumping vs Bumping Lemma

The goal here is to present a technique for proving that a language is not decidable by a DFA. The method found in every text and course I have seen is the *Pumping Lemma*. I think that the students find this technique hard, esoteric, useless, ... and I tend to agree with them. We first give the pumping lemma. We then give an alternative that I call the bumping lemma. You choose.

### 5.1 Pumping Lemma

For every language  $L$ , the pumping lemma (as given in class) defines a game  $G_L$  between a prover and an adversary. Then the lemma states “If  $L$  is regular, then the adversary has a strategy for the game  $G_L$  in which he is guaranteed to win”. The contra-positive of this statement is “If there are no winning strategies for the adversary to the game  $G_L$ , then  $L$  is not regular”. If a game has a winning strategy for the prover than it cannot have a winning one for the adversary. It follows that one can prove that a language  $L$  is not regular simply by presenting a strategy for the prover in which he is guaranteed to win.

For each of the following languages, attempt on your own to prove whether is regular or not.

1.  $L = \{a^n b^m \mid n \geq m\}$
2.  $L = \{a^n b^m \mid n \leq m\}$
3.  $L = \{a^n b^m \mid n \geq 100, m \leq 100\}$
4.  $\{a^n \mid n = k^2 \text{ for some } k \geq 0\}$
5.  $L = \{a^n b a^m b a^{n+m} \mid n, m \geq 0\}$
6.  $L = \{u w w \mid u, w \in \{a, b\}^*\}$
7.  $L = \{w \in \{0, 1\}^* \mid w \text{ when viewed as an integer in binary is divisible by } 7\}$ .

1.  $L = \{a^n b^m \mid n \geq m\}$

- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Adversary gives pumping number  $p$ .
  - I give  $s = a^p b^p$ . Note  $s \in L$  and  $|s| \geq p$ .
  - Adversary gives split  $s = xyz$ , where  $|xy| \leq p$  and  $|y| > 0$ . Due to the fact that  $|xy| \leq p$  and  $s$  begins with  $p$   $a$ s, it follows that  $y$  must contain all  $a$ s.
  - I set  $i = 0$ .

–  $xy^iz = xz = a^{p-|y|}b^p$ . This is not in  $L$  because  $n = p - |y| < p = m$ .

2.  $L = \{a^n b^m \mid n \leq m\}$

- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Same as above, except I set  $i = 2$ .
  - $xy^2z = a^{p+|y|}b^p$ . This is not in  $L$  because  $n = p + |y| > p = m$ .

3.  $L = \{a^n b^m \mid n \geq 100, m \leq 100\}$

- Answer: It is regular.  $\{a^n \mid n \geq 100\}$  and  $\{b^m \mid m \leq 100\}$  are both clearly regular.  $L$ , which is their concatenation, is also regular.

4.  $L = \{a^n \mid n = k^2 \text{ for some } k \geq 0\}$

- Answer: It is not regular.

In class, we proved that  $\{a^n \mid n \text{ is prime}\}$ . My method was using the fact that for every integer  $p$  there is a pair of consecutive primes  $p_0$  and  $p_1$  that are further apart than  $p$ . We then set  $s$  to be  $a^{p_1}$ . We will apply the same trick here.

Claim for every integer  $p$  there is a pair of consecutive squares  $p_0$  and  $p_1$  that are further apart than  $p$ . For example, let  $p_0$  be  $p^2$  and  $p_1$  be  $(p+1)^2$ . Then  $p_1 - p_0 = (p+1)^2 - p^2 = 2p+1 > p$ .

The proof that  $L$  is not regular is the following strategy to win the game  $G_L$ .

- Adversary gives pumping number  $p$ .
- I give  $s = a^{p^2}$ . Note  $s \in L$  and  $|s| \geq p$ .
- Adversary gives split  $s = xyz$ ,  $|xy| \leq p$  and  $|y| > 0$ . Clearly  $y$  contains only  $as$ .
- I set  $i = 2$ .
- $xy^2z$  has length  $p^2 + |y|$ . Note  $p^2 < |xy^2z| = p^2 + |y| \leq p^2 + p < (p+1)^2$ . Therefore, this length  $p^2 + |y|$  is not a square and hence the string  $xy^2z$  is not in  $L$ .

5.  $L = \{a^n b a^m b a^{n+m} \mid n, m \geq 0\}$

- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Adversary gives pumping number  $p$ .
  - I give  $s = a^p b a^p b a^{2p}$ . Note  $s \in L$  and  $|s| \geq p$ .
  - Adversary gives split  $s = xyz$ , where  $|xy| \leq p$  and  $|y| > 0$ . Due to the fact that  $|xy| \leq p$  and  $s$  begins with  $p$   $as$ , it follows that  $y$  must contain all  $as$  from the first block.
  - I set  $i = 0$ .
  - $xy^iz = xz = a^{p-|y|} b a^p b a^{2p}$ . This is not in  $L$  because  $(p - |y|) + p \neq 2p$ .

6.  $L = \{uww \mid u, w \in \{a, b\}^*\}$

- Answer: Trick question. This language IS regular. I claim that  $L$  is in fact the language  $\{a, b\}^*$  containing all strings. Why? Consider any string  $u \in \{a, b\}^*$ . This is in  $L$  simply by setting  $w$  to  $\epsilon$ .

7.  $L = \{w \in \{0, 1\}^* \mid w \text{ when viewed as an integer in binary is divisible by } 7\}$ .

- Answer: This language IS regular. We did it in the notes.

## 5.2 Bumping the Pumping Lemma

The goal here is to present a technique for proving that a language is not decidable by a DFA. The method found in every text and course I have seen is the *Pumping Lemma*. I think that the students find this technique hard, esoteric, useless, ... and I tend to agree with them. Here is an alternative that I think is more intuitive, easier to prove, and easier to use. Let me know if anyone has any reason that it is not as good as the pumping lemma. Let me know if you have a language that is not regular and this technique either does not prove or does not prove as easily as the pumping lemma.

Consider a language  $L$  that we want to prove is not regular. We say that the prefixes  $\alpha$  and  $\beta$  can be *distinguished* by  $L$  if there exists a postfix  $\gamma$  such that  $\alpha\gamma$  and  $\beta\gamma$  are given different answers by  $L$ . Eg. if  $L_{names}$  is the set of names of professors at York, then the first names  $\alpha = jeff$  and  $\beta = gordon$  are distinguished by  $L$  because the last name  $\gamma = edmonds$  is such that  $\alpha\gamma = jeffedmonds$  is in  $L_{names}$  and  $\beta\gamma = gordonedmonds$  is not. Similarly, if  $L_{0^n 1^n} = \{0^n 1^n \mid n \geq 0\}$ , the prefixes  $\alpha = 0^i$  and  $\beta = 0^j$  for  $i \neq j$  are distinguished by  $L$  because the postfix  $\gamma = 1^i$  is such that  $\alpha\gamma = 0^i 1^i$  is in  $L_{0^n 1^n}$  and  $\beta\gamma = 0^j 1^i$  is not.

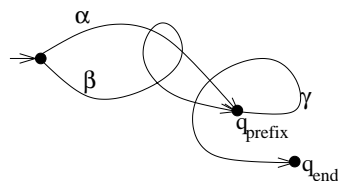
We say that the set  $S \subseteq \Sigma^*$  of prefixes can be *distinguished* by  $L$  if every pair of prefixes  $\alpha, \beta \in S$  can be distinguished. Likely the set  $S_{names}$  of first names of professors at York can be distinguished by  $L_{names}$ . We have proved that  $S = 0^*$  can be distinguished by  $L_{0^n 1^n}$ .

**Theorem 1** *If language  $L$  distinguishes the prefixes  $S$ , then any DFA that decides  $L$  requires at least  $|S|$  states.*

The intuition is that after the DFA has read in some prefix in  $S$ , it needs enough memory to remember which one it has seen or equivalently it needs enough states to distinguish between them.

**Corollary 2** *If language  $L$  distinguishes the prefixes  $S$ , where  $S$  contains an infinite number of prefixes, then  $L$  is not a regular language, because any DFA deciding it requires an infinite number of states.*

**Proof of Theorem 1:**



By way of contradiction, assume that the DFA  $M$  decides  $L$  and has fewer than  $|S|$  states. For each prefix  $\alpha \in S$ , run  $M$  on it and place it on the state of  $M$  at which the machine ends up. The *pigeon hole principle* states that you can't put  $n + 1$  pigeons into  $n$  pigeon holes without putting more than one pigeon in the same hole. By this principle, there is a state of  $M$  which we will denote  $q_{prefix}$  that receives two prefixes  $\alpha, \beta \in S$ . Because the prefixes  $\alpha$  and  $\beta$  can be distinguished by  $L$ , there exists a postfix  $\gamma$  such that  $\alpha\gamma$  and  $\beta\gamma$  are given different answers by  $L$ . I claim that the state  $q_{end}$  the  $M$  ends on when run on  $\alpha\gamma$  is the same as when run on  $\beta\gamma$ . The reason is that after reading  $\alpha$  and  $\beta$ ,  $M$  is in the same state  $q_{prefix}$  and then it continues in the same way from there when reading  $\gamma$ . This last state  $q_{end}$  is either an accept state or is not. Hence, it cannot give the correct answer on both input instances  $\alpha\gamma$  and  $\beta\gamma$ , which have different answers. ■

Hi Jeff,

I finally got a chance to look at the handout you sent me a while ago (Bumping the Pumping Lemma). I really think it is much easier to use than the Pumping Lemma, and it captures my intuition about why languages aren't regular much better. How did students like it?

I believe I can prove a converse to your Corollary 2:

Claim: If there is no infinite set distinguished by  $L$ , then  $L$  is regular.

This means that your "Bumping Lemma" can be used (in principle) to prove that any non-regular language is non-regular (provided you can find the right set). I suspect that this lemma might be part of the general folklore in automata theory. Nevertheless, it might actually be worth it to publicise the Bumping Lemma as a pedagogical tool, since all the automata textbooks seem to give only the Pumping Lemma. (In view of the claim above, it's actually better because it's an "iff" characterization of regular languages.) Maybe a little note in the Education Forum of SIGACT News?

Proof of claim: Let  $S$  be a maximal distinguished set. Construct a DFA whose states are labelled by elements of  $S$ .

For any  $x$  in  $S$  and  $a$  in  $\Sigma$ :

If  $xa$  is in  $S$ , then  $\delta(x,a)=xa$ .

Otherwise, there is a  $y$  in  $S$  s.t.  $L$  does not distinguish  $xa,y$  (if there were no such  $y$ ,  $S \cup \{xa\}$  would be a larger distinguished set). Let  $\delta(x,a)=y$ .

If  $\epsilon$  is in  $S$ , use  $\epsilon$  as the starting state  $s_0$ . Otherwise, there is some string  $y$  in  $S$  that is indistinguishable from  $\epsilon$ . Use that as the starting state  $s_0$ .

The accepting states of the DFA are those strings of  $S$  that are in  $L$ .

Lemma: For any string  $w$ ,  $L$  does not distinguish  $w$  from  $\delta(s_0, w)$ .

Proof: (It's actually pretty trivial, but I'll jot it down formally just to be careful.)

Base case ( $w = \text{empty string}$ ) follows from defn of  $s_0$ .

Suppose lemma holds for  $w$ . Let  $a \in \Sigma$ . Prove it for  $wa$ .

Case I:  $\delta(s_0,w)a \in S$ . Then  $\delta(s_0,wa)=\delta(s_0,w)a$ .

So if some postfix  $z$  could be used to distinguish  $wa$  from  $\delta(s_0,wa)$ , then the postfix  $az$  could be used to distinguish  $w$  from  $\delta(s_0,w)$ , contrary to ind. hyp.

Case II:  $\delta(s_0,w)a \notin S$ . Then  $\delta(s_0,wa)=y$ , where  $y$  is some string in  $S$  that  $L$  does not distinguish from  $\delta(s_0,w)a$ .

If there is a postfix  $z$  s.t.  $L$  would give different answers on  $waz$  and  $yz$ , then (since  $L$  gives same answer on  $yz$  and  $\delta(s_0,w)az$ )  $L$  would give different answers on  $waz$  and  $\delta(s_0,w)az$ . This means that the postfix  $az$  could be used to distinguish  $w$  from  $\delta(s_0,w)$ , contrary to the ind. hyp.

It follows from the lemma and the defn of accepting states that the

DFA decides the language  $L$ .

Eric

For each of the following languages, attempt on your own to prove whether is regular or not. For contrast, we provide for each a proof using both this technique and using the pumping lemma.

1.  $L = \{a^n b^m \mid n \geq m\}$
2.  $L = \{a^n b^m \mid n \geq 100, m \leq 100\}$
3.  $\{a^n \mid n = k^2 \text{ for some } k \geq 0\}$
4.  $L = \{a^n b a^m b a^{n+m} \mid n, m \geq 0\}$
5.  $L = \{ww \mid w \in \{a, b\}^*\}$
6.  $L = \{uww \mid u, w \in \{a, b\}^*\}$
7.  $L = \{w \in \{0, 1\}^* \mid w \text{ when viewed as an integer in binary is divisible by } 7\}$ .

1.  $L = \{a^n b^m \mid n \geq m\}$

- Answer: This  $L$  is not regular because it distinguishes between the infinite number of prefixes in the set  $S = a^*$ . The prefixes  $\alpha = a^i$  and  $\beta = a^j$  for  $i < j$  are distinguished by  $L$  as follows. Let  $\gamma = b^i$ . Then  $\alpha\gamma = a^i b^i$  is in  $L$  and  $\beta\gamma = a^j b^i$  is not.
- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Adversary gives pumping number  $p$ .
  - I give  $s = a^p b^p$ . Note  $s \in L$  and  $|s| \geq p$ .
  - Adversary gives split  $s = xyz$ , where  $|xy| \leq p$  and  $|y| > 0$ . Due to the fact that  $|xy| \leq p$  and  $s$  begins with  $p$   $a$ 's, it follows that  $y$  must contain all  $a$ 's.
  - I set  $i = 0$ .
  - $xy^i z = xz = a^{p-|y|} b^p$ . This is not in  $L$  because  $n = p - |y| < p = m$ .

2.  $L = \{a^n b^m \mid n \geq 100, m \leq 100\}$

- Answer: It is regular.  $\{a^n \mid n \geq 100\}$  and  $\{b^m \mid m \leq 100\}$  are both clearly regular.  $L$ , which is their concatenation, is also regular.

3.  $L = \{a^n \mid n = k^2 \text{ for some } k \geq 0\}$

- Answer: This  $L$  is not regular because it distinguishes between the infinite number of prefixes in the set  $S = a^*$ . The prefixes  $\alpha = a^i$  and  $\beta = a^j$  for  $i < j$  are distinguished by  $L$  as follows. Let  $\gamma = a^r$  for some  $r$  where  $i + r$  is a perfect square and  $j + r$  is not. Then  $\alpha\gamma = a^{i+r}$  is in  $L$  and  $\beta\gamma = a^{j+r}$  is not. There are lots of  $r$  that work. One is  $r = j^2 - i$ . Trivially  $i + r = j^2$  is a perfect square. We show that  $j + r$  is not a perfect square because it lies strictly between two consecutive perfect squares, namely  $j + r = j^2 + (j - i) > j^2$  because  $i < j$  and  $j + r = j^2 + (j - i) < j^2 + 2j + 1 = (j + 1)^2$ .
- Answer: It is not regular. Claim for every integer  $p$  there is a pair of consecutive squares  $p_0$  and  $p_1$  that are further apart than  $p$ . For example, let  $p_0$  be  $p^2$  and  $p_1$  be  $(p + 1)^2$ . Then  $p_1 - p_0 = (p + 1)^2 - p^2 = 2p + 1 > p$ .  
The proof that  $L$  is not regular is the following strategy to win the game  $G_L$ .

- Adversary gives pumping number  $p$ .
- I give  $s = a^{p^2}$ . Note  $s \in L$  and  $|s| \geq p$ .
- Adversary gives split  $s = xyz$ ,  $|xy| \leq p$  and  $|y| > 0$ . Clearly  $y$  contains only  $as$ .
- I set  $i = 2$ .
- $xy^2z$  has length  $p^2 + |y|$ . Note  $p^2 < |xy^2z| = p^2 + |y| \leq p^2 + p < (p + 1)^2$ . Therefore, this length  $p^2 + |y|$  is not a square and hence the string  $xy^2z$  is not in  $L$ .

4.  $L = \{a^n b a^m b a^{n+m} \mid n, m \geq 0\}$

- Answer: This  $L$  is not regular because it distinguishes between the infinite number of prefixes in the set  $S = a^*$ . The prefixes  $\alpha = a^i$  and  $\beta = a^j$  are distinguished by  $L$  as follows. Let  $\gamma = b a b a^{i+1}$ , then  $\alpha\gamma = a^i b a b a^{i+1}$  is in  $L$  and  $\beta\gamma = a^j b a b a^{i+1}$  is not.
- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Adversary gives pumping number  $p$ .
  - I give  $s = a^p b a^p b a^{2p}$ . Note  $s \in L$  and  $|s| \geq p$ .
  - Adversary gives split  $s = xyz$ , where  $|xy| \leq p$  and  $|y| > 0$ . Due to the fact that  $|xy| \leq p$  and  $s$  begins with  $p$   $as$ , it follows that  $y$  must contain all  $as$  from the first block.
  - I set  $i = 0$ .
  - $xy^i z = xz = a^{p-|y|} b a^p b a^{2p}$ . This is not in  $L$  because  $(p - |y|) + p \neq 2p$ .

5.  $L = \{ww \mid w \in \{a, b\}^*\}$

- Answer: This  $L$  is not regular, but the obvious proof is buggy.  $L$  distinguishes between the infinite number of prefixes in the set  $S = \{a, b\}^*$ . The prefixes  $\alpha = w$  and  $\beta = w'$  are distinguished by  $L$  with  $\gamma = w$  because  $\alpha\gamma = ww$  is in  $L$  and  $\beta\gamma = ww'$  is not. This is buggy because if  $w = 00$  and  $w' = 0000$  then  $ww' = 000\ 000$  is in  $L$ . The same proof works by changing  $S$  to  $ab^*$  so that with  $\alpha = \gamma = w = ab^i$  and  $\beta = w' = ab^j$ ,  $\beta\gamma = ww' = ab^i ab^j$  is not in  $L$ .
- Answer: It is not regular. The proof is the following strategy to win the game  $G_L$ .
  - Adversary gives pumping number  $p$ .
  - I give  $s = ab^p ab^p$ . Note  $s \in L$  and  $|s| \geq p$ .
  - Adversary gives split  $s = xyz$ , where  $|xy| \leq p$  and  $|y| > 0$ . Due to the fact that  $|xy| \leq p$  and that the second  $a$  is at index  $p + 2$ , it follows that  $y$  does not contain the second  $a$  or the  $bs$  after it.
  - I set  $i = 0$ .
  - $xy^i z = xz$  either only has one  $a$  or is of the form  $ab^{p-|y|} ab^p$  which is not in  $L$ .

6.  $L = \{uww \mid u, w \in \{a, b\}^*\}$

- Answer: Trick question. This language IS regular. I claim that  $L$  is in fact the language  $\{a, b\}^*$  containing all strings. Why? Consider any string  $u \in \{a, b\}^*$ . This is in  $L$  simply by setting  $w$  to  $\epsilon$ .

7.  $L = \{w \in \{0, 1\}^* \mid w \text{ when viewed as an integer in binary is divisible by } 7\}$ .

- Answer: This language IS regular. We did it in the notes.

## Chapter 6

# Context Free Grammars

Context Free Grammars consists of set of rules and a start *non-terminal* symbol. Each rule specifies one way of replacing a non-terminal symbol in the current string with a string of terminal and non-terminal symbols. When the resulting string consists only of terminal symbols, we stop. We say that any such resulting string has been *generated* by the grammar.

Context free grammars are used to understand both the syntax and the semantics of many very useful languages like mathematical expressions, JAVA, and English.

The *Syntax* of a language involves which strings of tokens are valid sentences in the language. We will say that a string is in the language if it can be generated by the grammar.

The *Semantics* of a language involves the “meaning” associated with strings. In order for a compiler or natural language recognizer to determine what a string means, it must *parse* the string. This involves deriving the string from the grammar and in doing so determining which parts of the string are “noun phrases”, “verb phrases”, “expressions”, or “terms”.

Most first algorithmic attempts to parse a string from a context free grammar requires  $2^{\Theta(n)}$  time. However, there is an elegant dynamic programming algorithm (we may consider it in this course) that parses a string from any context free grammar in  $\Theta(n^3)$  time. Though this is impressive, it is much too slow to be practical for compilers and natural language recognizers.

Some context free grammars have a property called *look ahead one*. Strings from such grammars can be parsed in linear time by what I consider to be one of the most amazing and magical recursive algorithms.

This recursive algorithm very nicely demonstrates the importance of the recursive technique described in the notes: Carefully write the specs for each program, believe by magic that the programs work, write the programs calling themselves as if they already work, and make sure that as you recurse the instance being inputted gets “smaller”.

**The Grammar:** As an example, we will be considering a very simple grammar that considers expressions over  $\times$  and  $+$ .

```
exp  $\Leftarrow$  term
     $\Leftarrow$  term + exp
term  $\Leftarrow$  fact
     $\Leftarrow$  fact * term
fact  $\Leftarrow$  int
     $\Leftarrow$  ( exp )
```

**A Derivation of a String:**

$s = ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 )$

```

|-exp-----|
|-term-----|
|-fact-----|
( |-exp-----| )
  |-term-----| + |-exp-----|
  |-fact---| * |-term---|   |-term-----| + |-exp-|
  ( |-ex-| )   |-fac---|     f * |-t---|   |-term-|
    t + e     ( |-ex-| )   978   f * t     f * t
    f t       t + e       7   f       15   f
    2 f       f t               123       54
      42       5 f
                12
s = ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 )

```

**A Parsing of an Expression:** The following are different forms that a parsing of an expression could take.

- A binary tree data structure with each internal node representing either '\*' or '+' and leaves representing integers.
- A text based picture of the above described tree.

```

s = ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 ) =
p =
      |-- 2
      | - + -
      |   |-- 42
      | - * -
      |   | |-- 5
      |   | - + -
      |   |   |-- 12
-- + -
      |   |-- 987
      |   | - * -
      |   |   | |-- 7
      |   |   | - * -
      |   |   |   |-- 123
      | - + -
      |   |-- 15
      |   | - * -
      |   |   |-- 54

```

- A string with more brackets to indicated the internal structure.

```

s = ( (2+42) * (5+12) + 987*7*123 + 15*54 )
p = (((2+42) * (5+12)) + ((987*(7*123)) + (15*54)))

```

- An integer evaluation of the expression.

```

s = ( ( 2 + 42 ) * ( 5 - 12 ) + 987 * 7 * 123 + 15 * 54 )
p = 851365

```

**Another Example:** Consider the following grammar. The start symbol is  $\langle Op \rangle$ .

```

⟨Op⟩ → ⟨If⟩ | ⟨IfElse⟩ | ⟨While⟩ | ⟨Assign⟩ | {⟨Ops⟩}
⟨Ops⟩ → ⟨Op⟩ | ⟨Op⟩⟨Ops⟩
⟨If⟩ → if(⟨Boolean⟩) then ⟨Op⟩
⟨IfElse⟩ → if(⟨Boolean⟩) then ⟨Op⟩ else ⟨Op⟩
⟨While⟩ → while(⟨Boolean⟩) ⟨Op⟩

```

$$\begin{aligned}
\langle \text{Assign} \rangle &\rightarrow \langle \text{Var} \rangle = \langle \text{Equ} \rangle \\
\langle \text{Boolean} \rangle &\rightarrow \langle \text{Equ} \rangle = \langle \text{Equ} \rangle \mid \langle \text{Equ} \rangle < \langle \text{Equ} \rangle \\
\langle \text{Equ} \rangle &\rightarrow \langle \text{Term} \rangle \mid \langle \text{Term} \rangle + \langle \text{Equ} \rangle \\
\langle \text{Term} \rangle &\rightarrow \langle \text{Factor} \rangle \mid \langle \text{Factor} \rangle \times \langle \text{Term} \rangle \\
\langle \text{Factor} \rangle &\rightarrow \langle \text{Var} \rangle \mid \langle \text{Num} \rangle \mid (\langle \text{Equ} \rangle) \\
\langle \text{Var} \rangle &\rightarrow i \mid j \mid k \mid x \mid y \mid z \\
\langle \text{Num} \rangle &\rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{Num} \rangle \\
\langle \text{digit} \rangle &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Below are three code fragments. View each as one entire string. Ignore spaces and line breaks. For each, do the following. If possible give a parsing for it. If not explain why. If possible give a second parsing for it. Is there anything about this code fragment that indicates a potential danger for programmers? How do the C and the JAVA compilers deal with these problems? How do programmers deal with these problems?

(a) `if(y = 3 * x + 7) then {  
    i = 0  
    while(i < 10)  
        y = (y + 10) + 5  
        i = i + 1  
}`

(b) `if(i = y) then  
    if(i = x) then  
        i = 0  
    else  
        y = 7`

(c) `y = (3 * x + 7) * 53  
z = ((2 * 3 * 4 + 5 + 6) + 7)`

- Answer: The parsing is straight forward except that the line  $i = i + 1$  will not be within the *while* loop. One operation after a *while* or an *if* does not need brackets, but two does. Programmers should get into the habit of always putting brackets `{ }` around even a single *op*. Someone may later add a second *op* and not notice that the brackets are not there.
- Answer: There are two parsings. One associates the *else* with the first *if* and the other with the second *if*. The compiler defaults to the second. Again being in the habit of always putting brackets `{ }` around even a single command after an *if* would solve this problem.
- Answer: This has no parsing. It consists of two *ops* without a surrounding bracket.

## 6.1 Proving Which Language is Generated

Consider the grammar  $G$

$$S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$$

The goal of this question is to formally prove that this grammar generates the language  $L = \{s \mid \# \text{ of } a\text{'s in } s = \# \text{ of } b\text{'s in } s\}$ . This involves two steps.

$\Rightarrow$  Prove “ $G$  only generates strings that are in  $L$ .” See (a).

$\Leftarrow$  Prove “ $G$  generates all the strings in  $L$ ,” i.e., “If  $s \in L$ , then  $G$  generates it.” See (d).

1. Formally prove “if the string  $s$  can be generated by  $G$ , then it is in  $L$ ”. The proof will be by strong induction on the length  $|s|$  of the string. As with all proofs by strong induction, the format is as follows:

- (1 apple) For each integer  $n \geq 0$ , define  $H(n)$  to be the boolean statement “\_\_\_\_\_.”  
    – Answer: “If the string  $s$  has length  $n$  and can be generated by  $G$ , then it is in  $L$ .”
- (1 apple)  $H(0)$  is true because \_\_\_\_\_.

- Answer: the only string of length zero is  $s = \epsilon$ , which can be generated by  $G$  and is in  $L$ .
- (4 apples) Let  $n$  be an arbitrary integer bigger than zero. By way of strong induction, assume that each of  $H(0), H(1), H(2), \dots, H(n-1)$  is true. Using these assumptions, we prove  $H(n)$  as follows. \_\_\_\_\_
  - Answer: Let  $s$  be an arbitrary string of length  $n$  that can be generated by  $G$ . Let  $P$  be one of the shortest parsings of  $s$ . The first step of  $P$  must be one of the following.
    - $S \rightarrow aSb$ : In this case,  $s$  must be  $s = as_1b$  where  $s_1$  has length  $n-2$  and can be generated by  $G$ . Hence, by  $H(n-2)$ ,  $s_1$  is in  $L$ . Because  $s_1$  has the same number of  $a$ 's and  $b$ 's, it follows that  $s = as_1b$  also has the same number of  $a$ 's and  $b$ 's and hence  $s \in L$ .
    - $S \rightarrow bSa$ : Same as above.
    - $S \rightarrow SS$ : In this case,  $s$  must be  $s = s_1s_2$  where  $s_1$  has length  $i$ ,  $s_2$  has length  $n-i$ , and both  $s_1$  and  $s_2$  can be generated by  $G$ . Hence, by  $H(i)$  and  $H(n-i)$ , both  $s_1$  and  $s_2$  are in  $L$ . Because both  $s_1$  and  $s_2$  have the same number of  $a$ 's and  $b$ 's, it follows that  $s = s_1s_2$  also has the same number of  $a$ 's and  $b$ 's and hence  $s \in L$ .  
(Glitch: If  $i = n$ , then  $H(i) = H(n)$  has not been assumed. Similarly, if  $i = 0$ , then  $H(n-i) = H(n)$  has not been. If  $i = n$ , then the parsing  $P$  begins as follows.  $S \Rightarrow SS \Rightarrow S\epsilon$ . Removing these steps gives a shorter parsing that generates the same string  $s$ . This contradicts the fact that  $P$  is the shortest parsing. Hence,  $i \neq 0$ . Similarly  $i \neq n$ .)
- (1 apple) By way of strong induction we can conclude that  $\forall n \geq 0 H(n)$ . From this the statement, “if the string  $s$  can be generated by  $G$ , then it is in  $L$ ” follows because \_\_\_\_\_
  - Answer: if it is true for any  $s$  of each length, then it is true for any  $s$ .

2. Let  $s \in \{a, b\}^*$  be an arbitrary string. Let  $n = |s|$ . Define  $F_s$  to be the following function. For each integer  $i \in [0, n]$ , define  $F_s(i)$  to be ( $\#$  of  $a$ 's in the first  $i$  characters of  $s$ )  $-$  ( $\#$  of  $b$ 's in the first  $i$  characters of  $s$ ). For example,  $F_{abba}(3) = -1$ . To make the function  $F_s(x)$  continuous for all real values  $x$  interpolate between  $F_s(i)$  and  $F_s(i+1)$  with a diagonal line.

Note that for every string  $s$ ,  $F_s(0) = 0$ . Call this graph property (i).

(a) (2 apples) Graph  $F_{abbbba}$  and  $F_{aabab}$ .

(b) (1 apple) If  $s \in L$ , then what property does  $F_s$  have? Call this graph property (ii).

- Answer: Then  $F_s$  ends at zero, ie  $F_s(n) = 0$ .

(c) (2 apples) If  $s \in L$  and  $s$  begins and ends with the same character, then what property does  $F_s$  have? Call this graph property (iii).

- Answer: Define property (iii) to be property that  $F_s(1)$  and  $F_s(n-1)$  have opposite signs, i.e. one positive and the other negative.

If  $s$  begins with an  $a$ , then  $F_s$  starts at  $F_s(0) = 0$  and goes up to  $F_s(1) = 1$ . If  $s$  ends with an  $a$ , then  $F_s$  ends by going up from  $F_s(n-1) = -1$  to  $F_s(n) = 0$ . Note that this means that it has property (iii).

Similarly, if  $s$  begins with a  $b$ , then  $F_s$  starts at  $F_s(0) = 0$  and goes down to  $F_s(1) = -1$ . If  $s$  ends with a  $b$ , then  $F_s$  ends by going down from  $F_s(n-1) = 1$  to  $F_s(n) = 0$ . Again, it has property (iii).

(d) (2 apples) State a graph property (iv) of  $F_s$ . Prove that if  $F_s$  has properties (i), (ii), and (iv), then  $s$  can be split into two parts  $s = s_1s_2$  such that both  $s_1$  and  $s_2$  are in  $L$  and neither  $s_1$  nor  $s_2$  is the empty string.

- Answer: Define property (iv) to be property that there is an integer  $i$  such that  $i \neq 0$ ,  $i \neq n$ , and  $F_s(i) = 0$ .

Suppose that  $F_s$  properties (i), (ii), and (iv). Let  $s_1$  be the first  $i$  characters of  $s$  and  $s_2$  be the last  $n-i$  characters.

Note that because  $i \neq 0$  and  $i \neq n$  it follows that neither  $s_1$  nor  $s_2$  are the empty string.

Because  $F_s(0) = 0$  and  $F_s(i) = 0$ , it follows by definition that  $s_1$  has the same number of  $a$ 's and  $b$ 's and hence is in  $L$ . Similarly, because  $F_s(i) = 0$  and  $F_s(n) = 0$ , it follows by definition that  $s_2$  has the same number of  $a$ 's and  $b$ 's and hence is in  $L$ .

(e) (2 apples) Prove that any function that has graph property (iii) also has graph property (iv).

- Answer: If  $F_s(1)$  and  $F_s(n-1)$  have opposite signs and is continuous, then by the Mean Value Theorem from Calculus there must be an  $x$  in between for which  $F_s(x) = 0$ . All we need now is to prove that  $x$  is an integer. Note that for every integer  $i$ , the function  $F_s(x)$  either increases or decreases by one between  $F_s(i)$  and  $F_s(i+1)$ . Therefore, the function  $F_s(x)$  takes on integer values only on integers values. It follows that if  $F_s(x) = 0$ , then  $x$  is an integer. This  $x$  is the integer  $i$  required for property (iv).

3. (2 apples) Prove the following lemma. (Hint: Even if you were not able to answer the questions in (b), you may use the statements in the questions to prove this lemma. Do not leave any gaps in your logic.)

Lemma: If  $s \in L$  and begins and ends with the same character, then  $s$  can be split into two parts  $s = s_1s_2$  such that both  $s_1$  and  $s_2$  are in  $L$  and neither  $s_1$  nor  $s_2$  is the empty string.

- Answer: Proof:
  - All functions  $F_s$  have property (i).
  - Because  $s \in L$ , by question (bii),  $F_s$  has property (ii).
  - Because  $s \in L$  and  $s$  begins and ends with the same character, by question (biii),  $F_s$  has property (iii).
  - Because  $F_s$  has property (iii), by question (bv), it has property (iv).
  - Because  $F_s$  has properties (i), (ii), and (iv), by question (biv),  $s$  can be split into two parts  $s = s_1s_2$  such that both  $s_1$  and  $s_2$  are in  $L$  and neither  $s_1$  nor  $s_2$  is the empty string.

4. (4 apples) Construct a recursive algorithm whose input is a string  $s \in L$  and whose output is a parsing of  $s$  by the grammar. Note if  $s \notin L$ , then what the algorithm does is unspecified. Given an instance,  $s \in L$ , the algorithm will have a number of cases. All but one of these cases proceeds as follows.

- Assume that your friends can solve any instance to the same problem as long as it is strictly smaller than your instance.
- From your  $s$  construct some subinstances, ie construct one or more strings  $s_1$  and  $s_2$ . Be sure to prove that these strings are in  $L$  and are shorter than  $s$ .
- Have your friends give you a parsing for each of your strings  $s_1$  and  $s_2$ .
- Construct a parsing for  $s$  from your friends' parsings.

(Hint: Even if you were not able to answer question (c), you may use its statement to construct this algorithm.)

- Answer: Given an instance,  $s \in L$ , the recursive algorithm is as follows. There are four cases:
  - $s = \epsilon$ :** The parsing is  $S \Rightarrow \epsilon$ .
  - $s$  begins with an  $a$  and ends with a  $b$ :** Split  $s$  into  $s = as_1b$ . Because  $s = as_1b$  has the same number of  $a$ 's and  $b$ 's, it follows that  $s_1$  also has the same number of  $a$ 's and  $b$ 's, and hence  $s_1 \in L$ . Ask your friend to give you a parsing for  $s_1$ . Your parsing will be  $S \Rightarrow aSb \Rightarrow^* as_1b = s$ , where the  $\Rightarrow^*$  follows your friends parsing.
  - $s$  begins with a  $b$  and ends with an  $a$ :** Same as above.
  - $s$  begins and ends with the same character:** By the lemma we know that  $s$  can be split into two parts  $s = s_1s_2$  such that both  $s_1$  and  $s_2$  are in  $L$ . Because neither are empty, both are strictly smaller than  $s$ . Ask one friend to give you a parsing for  $s_1$  and another friend to give you a parsing of  $p_2$ . Your parsing will be  $S \Rightarrow SS \Rightarrow^* s_1S \Rightarrow^* s_1s_2 = s$ , where the first  $\Rightarrow^*$  follows your first friends parsing and the second follows that of your second friend.

5. (2 apples) Trace your algorithm to produce a parsing for  $s = babbaa$ .

• Answer:

Note  $babbaa$  begins with a  $b$  and ends with an  $a$ . Hence its parsing is

$S \Rightarrow bSa$ , where the  $S$  must generate  $abba$ .

Note  $abba$  begins with and ends with the same character. We split it into

$ab$  and  $ba$  both in  $L$ . Hence its parsing is  $S \Rightarrow SS$ , where the first  $S$  must generate  $ab$  and the second  $ba$ .

Note  $ab$  begins with an  $a$  and ends with a  $b$ . Hence its parsing is  $S \Rightarrow aSb$ , where the  $S$  must generate  $\epsilon$ .

$\epsilon$  is generated with  $S \Rightarrow \epsilon$ .

Note  $ba$  begins with a  $b$  and ends with an  $a$ . Hence its parsing is  $S \Rightarrow bSa$ , where the  $S$  must generate  $\epsilon$ .

$\epsilon$  is generated with  $S \Rightarrow \epsilon$ .

In conclusion the parsing is  $S \Rightarrow bSa \Rightarrow bSSa \Rightarrow baSbSa \Rightarrow ba\epsilon bSa = babSa \Rightarrow babbSaa \Rightarrow babb\epsilon aa = babbaa$

## 6.2 Pumping Lemma

Use the pumping lemma to show that  $L = \{a^n b^m c^n d^m \mid m, n \geq 0\}$  is not context free.

• Consider the following strategy for the game  $G_L$ .

– The adversary gives us the pumping length  $p$ .

– We choose the string  $s = \underline{\hspace{4cm}}$ .

\* Answer:  $s = a^p b^p c^p d^p$

– The adversary chooses a split  $s = uvxyz$  such that  $|vy| > 0$  and  $|vxy| \leq p$ .

– We separately consider the following cases  $\underline{\hspace{4cm}}$ .

\* Answer: Because  $|vxy| \leq p$ , the string  $vxy$  can only be contained in one of the four blocks of  $s$  or within two consecutive blocks.

– We set  $i = \underline{\hspace{4cm}}$ .

\* Answer: We set  $i = 0$ . Because  $|vy| > 0$ , we know that  $s$  has some of its characters removed to form  $uxz$ . More over, we know that the characters removed are in at most two consecutive blocks.

– We prove  $uv^i xy^i z \notin L$  as follows  $\underline{\hspace{4cm}}$ .

\* Answer: Consider one of the four blocks of  $s$  that has some of its characters removed. Call it block  $B$ . There is another block of  $s$  that is supposed to be the same size as this one. Call it block  $B'$ . Note  $B$  and  $B'$  are not consecutive to each other. Hence, they could not both have changed. Hence, after the change they cannot still be the same size. Hence,  $uxz \notin L$ .

– In each case we win the game.

• Hence, there is no strategy with which the adversary can win.

• Hence,  $L$  is not context free.

## Chapter 7

# Parsing with Context-Free Grammars

An important computer science problem is parsing a string according to a given context-free grammar. A *context-free grammar* is a means of describing which strings of characters are contained within a particular language. It consists of a set of rules and a start *non-terminal* symbol. Each rule specifies one way of replacing a non-terminal symbol in the current string with a string of terminal and non-terminal symbols. When the resulting string consists only of terminal symbols, we stop. We say that any such resulting string has been *generated* by the grammar.

Context-free grammars are used to understand both the syntax and the semantics of many very useful languages, such as mathematical expressions, JAVA, and English. The *syntax* of a language indicates which strings of tokens are valid sentences in that language. The *semantics* of a language involves the meaning associated with strings. In order for a compiler or natural language “recognizers” to determine what a string means, it must *parse* the string. This involves deriving the string from the grammar and, in doing so, determining which parts of the string are “noun phrases”, “verb phrases”, “expressions”, and “terms.”

Some context-free grammars have a property called *look ahead one*. Strings from such grammars can be parsed in linear time by what I consider to be one of the most amazing and magical recursive algorithms. This algorithm is presented in this chapter. It demonstrates very clearly the importance of working within the friends level of abstraction instead of tracing out the stack frames: Carefully write the specifications for each program, believe by magic that the programs work, write the programs calling themselves as if they already work, and make sure that as you recurse the instance being input gets “smaller.”

In Section ?? we will analyze an elegant dynamic-programming algorithm given that parses a string from any context-free grammar, not just look ahead one, in  $\Theta(n^3)$  time.

**The Grammar:** We will look at a very simple grammar that considers expressions over  $\times$  and  $+$ . In this grammar, a *factor* is either a simple integer or a more complex expression within brackets; a term is one or more factors multiplied together; and an expression is one or more terms added together. More precisely:

$$\begin{aligned} \text{exp} &\Rightarrow \text{term} \\ &\Rightarrow \text{term} + \text{term} + \dots + \text{term} \end{aligned}$$
$$\begin{aligned} \text{term} &\Rightarrow \text{fact} \\ &\Rightarrow \text{fact} * \text{fact} * \dots * \text{fact} \end{aligned}$$
$$\begin{aligned} \text{fact} &\Rightarrow \text{int} \\ &\Rightarrow (\text{exp}) \end{aligned}$$

**Non-Terminals, Terminals, and Rules:** More generally, a grammar is defined by a set of *non-terminals*,

a set of *terminals*, a *start non-terminal*, and a set of rules. Here the non-terminals are “exp”, “term”, and “fact.” The terminals are integers, the character “+”, and the character “\*.” The start non-terminal is “exp.” Above is the list of rules for this grammar.

**A Derivation of a String:** A grammar defines the *language* of strings that can be derived in the following way. A derivation of a string starts with the start the non-terminal. Then each rule, like those above, say that you can replace the non-terminal on the left with the string of terminals and non-terminals on the right. The following is an example.

```

|-exp-----|
|-t-| + |-term-----|
f * f  |-fact-----|
6  8  ( |-exp-----| )
      |-term-----| + |-term-----| + |-term-|
      |-fact---| * |-fact---|   f * f * f   f * f
      ( |-ex-| ) ( |-ex-| )   987  7  123  15  54 )
          t + t       t + t
          f  f       f + f
          2  42     5  12
s = 6 * 8 + ( ( 2 + 42 ) * ( 5 + 12 ) + 987 * 7 * 123 + 15 * 54 )

```

**A Parsing of an Expression:** Let  $s$  be a string consisting of terminals. A parsing of this string is a tree. Each internal node of the tree is labeled with a non-terminal symbol, the root with the start non-terminal. Each internal node must correspond to a rule of the grammar. For example, for rule  $A \Rightarrow BC$ , the node is labeled  $A$  and its two children are labeled  $B$  and  $C$ . The leaves of the tree read left to right give the input string  $s$  of terminals. The following is an example.

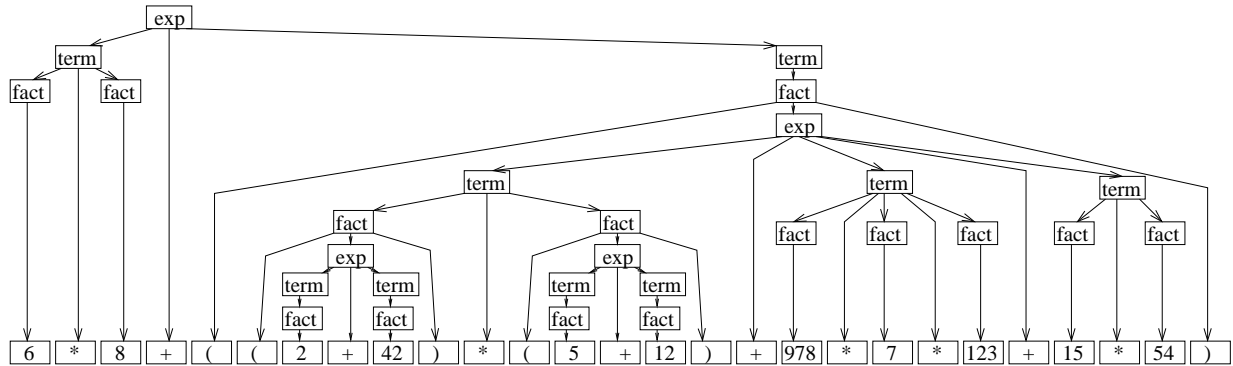


Figure 7.1: A parse tree for the string  $s = 6 * 8 + ((2 + 42) * (5 + 12) + 987 * 7 * 123 + 15 * 54)$ .

**The Parsing Abstract Data Type:** The following is an example where it is useful not to give the full implementation details of an abstract data type. If fact, we will even leave the specification of parsing structure open for the implementer to decide.

For our purposes, we will only say the following: When  $p$  is a variable of type parsing, we will use “ $p=5$ ” to indicate that it is assigned a parsing of the expression “5”. We will go on to *overload* the operations  $*$  and  $+$  as operations that join two parsings into one. For example, if  $p_1$  is a parsing of the expression “ $2*3$ ” and  $p_2$  of “ $5*7$ ”, then we will use  $p = p_1 + p_2$  to denote a parsing of expression “ $2*3 + 5*7$ ”.

The implementer defines the structure of a parsing by specifying in more detail what these operations do. For example, if the implementer wants a parsing to be a binary tree representing the expression, then  $p_1 + p_2$  would be the operation of constructing a binary tree with the root being a new ‘+’ node, the left subtree being the binary tree  $p_1$ , and the right subtree being the binary tree  $p_2$ . On the other hand, if the implementer wants a parsing to be simply an integer evaluation of the expression, then  $p_1 + p_2$  would be the integer sum of the integers  $p_1$  and  $p_2$ .

### Specifications for the Parsing Algorithm:

**Precondition:** The input consists of a string of tokens  $s$ . The possible tokens are the characters '\*', and '+' and arbitrary integers. The tokens are indexed as  $s[1], s[2], s[3], \dots, s[n]$ .

**Postcondition:** If the input is a valid "expression" generated by the grammar, then the output is a "parsing" of the expression. Otherwise, an error message is output.

The algorithm consists of one routine for each *non-terminal* of the grammar: *GetExp*, *GetTerm*, and *GetFact*.

### Specifications for *GetExp*:

**Precondition:** The input of *GetExp* consists of a string of tokens  $s$  and an index  $i$  that indicates a starting point within  $s$ .

**Postcondition:** The output consists of a parsing of the longest substring  $s[i], s[i+1], \dots, s[j-1]$  of  $s$  that starts at index  $i$  and is a valid expression. The output also includes the index  $j$  of the token that comes immediately after the parsed expression.

If there is no valid expression starting at  $s[i]$ , then an error message is output.

The specifications for *GetTerm* and *GetFact* are the same as for *GetExp*, except that they return the parsing of the longest term or factor starting at  $s[i]$  and ending at  $s[j-1]$ .

### Examples of *GetExp*, *GetTerm*, and *GetFact*:

*GetExp*:

$s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$					
$i$		$j$		$p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$	
$i$		$j$		$p = ( 2 * 8 + 42 * 7 ) * 5 + 8$	
$i$		$j$		$p = 2 * 8 + 42 * 7$	
$i$		$j$		$p = 42 * 7$	
$i$		$j$		$p = 5 + 8$	

*GetTerm*:

$s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$					
$i$		$j$		$p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$	
$i$		$j$		$p = ( 2 * 8 + 42 * 7 ) * 5$	
$i$		$j$		$p = 2 * 8$	
$i$		$j$		$p = 42 * 7$	
$i$		$j$		$p = 5$	

*GetFact*:

$s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$					
$i$		$j$		$p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )$	
$i$		$j$		$p = ( 2 * 8 + 42 * 7 )$	
$i$		$j$		$p = 2$	
$i$		$j$		$p = 42$	
$i$		$j$		$p = 5$	

**Reasoning for *GetExp*:** Consider some input string  $s$  and some index  $i$ . The longest substring  $s[i], \dots, s[j-1]$  that is a valid expression consists of some number of terms added together. In all of these case, it begins with a term. By magic, assume that the *GetTerm* routine already works. Calling *GetTerm*( $s, i$ ) will return  $p_{term}$  and  $j_{term}$ , where  $p_{term}$  is the parsing of this first term and  $j_{term}$  indexes the token immediately after this term. Specifically, if the expression has another term then  $j_{term}$  indexes the '+' that is between these terms. Hence, we can determine whether there is another term by checking  $s[j_{term}]$ . If  $s[j_{term}] = '+'$ , then *GetExp* will call *GetTerm* again to get the

next term. If  $s[j_{term}]$  is not a '+' but some other character, then *GetExp* is finished reading in all the terms. *GetExp* then constructs the parsing consisting of all of these terms added together.

The reasoning for *GetTerm* is the same.

**GetExp Code:**

**algorithm** *GetExp*( $s, i$ )

**⟨pre-cond⟩:**  $s$  is a string of tokens and  $i$  is an index that indicates a starting point within  $s$ .

**⟨post-cond⟩:** The output consists of a parsing  $p$  of the longest substring  $s[i], s[i + 1], \dots, s[j - 1]$  of  $s$  that starts at index  $i$  and is a valid expression. The output also includes the index  $j$  of the token that comes immediately after the parsed expression.

begin

if ( $i > |s|$ ) return “Error: Expected characters past end of string.” end if

$\langle p_{\langle term,1 \rangle}, j_{\langle term,1 \rangle} \rangle = GetTerm(s, i)$

$k = 1$

loop

**⟨loop-invariant⟩:** The first  $k$  terms of the expression have been read.

exit when  $s[j_{\langle term,k \rangle}] \neq '+'$

$\langle p_{\langle term,k+1 \rangle}, j_{\langle term,k+1 \rangle} \rangle = GetTerm(s, j_{\langle term,k \rangle} + 1)$

$k = k + 1$

end loop

$p_{exp} = p_{\langle term,1 \rangle} + p_{\langle term,2 \rangle} + \dots + p_{\langle term,k \rangle}$

$j_{exp} = j_{\langle term,k \rangle}$

return  $\langle p_{exp}, j_{exp} \rangle$

end algorithm

**GetTerm Code:**

**algorithm** *GetTerm*( $s, i$ )

**⟨pre-cond⟩:**  $s$  is a string of tokens and  $i$  is an index that indicates a starting point within  $s$ .

**⟨post-cond⟩:** The output consists of a parsing  $p$  of the longest substring  $s[i], s[i + 1], \dots, s[j - 1]$  of  $s$  that starts at index  $i$  and is a valid term. The output also includes the index  $j$  of the token that comes immediately after the parsed term.

begin

if ( $i > |s|$ ) return “Error: Expected characters past end of string.” end if

$\langle p_{\langle fact,1 \rangle}, j_{\langle fact,1 \rangle} \rangle = GetFact(s, i)$

$k = 1$

loop

**⟨loop-invariant⟩:** The first  $k$  facts of the term have been read.

exit when  $s[j_{\langle fact,k \rangle}] \neq '*'$

$\langle p_{\langle fact,k+1 \rangle}, j_{\langle fact,k+1 \rangle} \rangle = GetFact(s, j_{\langle fact,k \rangle} + 1)$

$k = k + 1$

end loop

$p_{term} = p_{\langle fact,1 \rangle} * p_{\langle fact,2 \rangle} * \dots * p_{\langle fact,k \rangle}$

$j_{term} = j_{\langle fact,k \rangle}$

return  $\langle p_{term}, j_{term} \rangle$

end algorithm

**Reasoning for GetFact:** The longest substring  $s[i], \dots, s[j - 1]$  that is a valid factor has one of the following two forms:

fact  $\Rightarrow$  int  
 fact  $\Rightarrow$  ( exp )

Hence, we can determine which form the factor has by testing  $s[i]$ .

If  $s[i]$  is an integer, then we are finished.  $p_{fact}$  is a parsing of this single integer  $s[i]$  and  $j_{fact} = i + 1$ . The +1 moves the index past the integer.

If  $s[i] = '('$ , then for  $s$  to be a valid factor there must be a valid expression starting at  $j_{term} + 1$ , followed by a closing bracket ')'. We can parse this expression with  $GetExp(s, j_{term} + 1)$ , which returns  $p_{exp}$  and  $j_{exp}$ . The closing bracket after the expression must be in  $s[j_{exp}]$ . Our parsed factor will be  $p_{fact} = (p_{exp})$  and  $j_{fact} = j_{exp} + 1$ . The +1 moves the index past the ')

If  $s[i]$  is neither an integer nor a '(', then it cannot be a valid factor. Give a meaningful error message.

**GetFact Code:**

**algorithm** *GetFac*( $s, i$ )

**<pre-cond>**:  $s$  is a string of tokens and  $i$  is an index that indicates a starting point within  $s$ .

**<post-cond>**: The output consists of a parsing  $p$  of the longest substring  $s[i], s[i + 1], \dots, s[j - 1]$  of  $s$  that starts at index  $i$  and is a valid factor. The output also includes the index  $j$  of the token that comes immediately after the parsed factor.

```
begin
  if ( $i > |s|$ ) return "Error: Expected characters past end of string." end if
  if ( $s[i]$  is an int)
     $p_{fact} = s[i]$ 
     $j_{fact} = i + 1$ 
    return  $\langle p_{fact}, j_{fact} \rangle$ 
  else if ( $s[i] = '('$ )
     $\langle p_{exp}, j_{exp} \rangle = GetExp(s, i + 1)$ 
    if ( $s[j_{exp}] = ')'$ )
       $p_{fact} = (p_{exp})$ 
       $j_{fact} = j_{exp} + 1$ 
      return  $\langle p_{fact}, j_{fact} \rangle$ 
    else
      Output "Error: Expected ')' at index  $j_{exp}$ "
    end if
  else
    Output "Error: Expected integer or '(' at index  $i$ "
  end if
end algorithm
```

**Tree of Stack Frames:** *GetExp* calls *GetTerm*, which calls *GetFact*, which may call *GetExp*, and so on. If one were to draw out the entire tree of stack frames showing who calls who, this would exactly mirror the parse tree that created.

**Running Time:** We prove that the running time of this entire computation is linear in the size to the parse tree produced and that this is linear in the size  $\Theta(n)$  of the input string.

To prove the first, it is sufficient to prove that the running time of each stack frame is either constant or is linear in the number of children of the node in the parse tree that this stack frame produces. For example, if stack frame for *GetFact* finds an integer than its node in the parse tree has no children, but *GetFact* uses only a constant amount of time. In contrast, if a stack frame for *GetExp* reads in  $t$  terms, then its running time will be some constant times  $t$  and its node in the parse tree will have  $t$  children.

We now prove that the size to the parse tree produced is linear in the size  $\Theta(n)$  of the input string. If the grammar is such that every non-terminal goes to at least one terminal or at least two non-terminals, then each node in the parse tree is either a leaf or has at least two children. It follows that the number of nodes in the parse tree will be at most some constant times the number of leaves, which is the size of the input string. In our grammar, however, an expression might go to a single term, which can go to a single factor. This creates a little path of out degree one. It cannot, however, be longer than this because a factor is either a leaf or has three children, one is “(”, the second an expression, and the third “)”. Such little paths can only increase the size of the parse tree by a factor of three.

In conclusion, the running time is  $\Theta(n)$ .

**Proof of Correctness:** To prove that a recursive program works, we must consider the “size” of an instance. The routine needs only consider the postfix  $s[i], s[i+1], \dots$ , which contains  $(|s| - i + 1)$  characters. Hence, we will define the size of instance  $\langle s, i \rangle$  to be  $|\langle s, i \rangle| = |s| - i + 1$ .

Let  $H(n)$  be the statement “Each of *GetFac*, *GetTerm*, and *GetExp* work on instances  $\langle s, i \rangle$  when  $|\langle s, i \rangle| = |s| - i + 1 \leq n$ .” We prove by way of induction that  $\forall n \geq 0, H(n)$ .

If  $|\langle s, i \rangle| = 0$ , then  $i > |s|$ : There is not a valid expression/term/factor starting at  $s[i]$ , and all three routines return an error message. It follows that  $H(0)$  is true.

If  $|\langle s, i \rangle| = 1$ , then there is one remaining token: For this to be a factor, term, or expression, this token must be a single integer. *GetFac* is written to give the correct answer in this situation. *GetTerm* gives the correct answer, because it calls *GetFac*. *GetExp* gives the correct answer, because it calls *GetTerm* which in turn calls *GetFac*. It follows that  $H(1)$  is true.

Assume  $H(n - 1)$  is true, that is, that “Each of *GetFac*, *GetTerm*, and *GetExp* work on instances of size at most  $n - 1$ .”

Consider *GetFac*( $s, i$ ) on an instance of size  $|s| - i + 1 = n$ . It makes at most one subroutine call, *GetExp*( $s, i + 1$ ). The size of this instance is  $|s| - (i + 1) + 1 = n - 1$ . Hence, by assumption this subroutine call returns the correct answer. Because all of *GetFac*( $s, i$ )’s subroutine calls return the correct answer, it follows that *GetFac*( $s, i$ ) works on all instances of size  $n$ .

Now consider *GetTerm*( $s, i$ ) on an instance of size  $|s| - i + 1 = n$ . It calls *GetFac* some number of times. The input instance for the first call *GetFac*( $s, i$ ) still has size  $n$ . Hence, the induction hypothesis  $H(n - 1)$  does NOT claim that it works. However, the previous paragraph proves that this routine does in fact work on instances of size  $n$ . The remaining calls are on smaller instances.

Finally, consider *GetExp*( $s, i$ ) on an instance  $\langle s, i \rangle$  of size  $|s| - i + 1 = n$ . We use the previous paragraph to prove that its first subroutine call *GetTerm*( $s, i$ ) works.

In conclusion, all three work on all instances of size  $n$  and hence on  $H(n)$ . This completes the induction step.

**Look Ahead One:** A grammar is said to be *look ahead one* if, given any two rules for the same non-terminal, the first place that the rules differ is a difference in a terminal. This feature allows our parsing algorithm to look only at the next token in order to decide what to do next. Thus the algorithm runs in linear time.

An example of a good set of rules would be:

$$\begin{aligned} A &\Rightarrow B \text{ 'b' } C \text{ 'd' } E \\ A &\Rightarrow B \text{ 'b' } C \text{ 'e' } F \\ A &\Rightarrow B \text{ 'c' } G H \end{aligned}$$

An example of a bad set of rules would be:

$$\begin{aligned} A &\Rightarrow B C \\ A &\Rightarrow D E \end{aligned}$$

With such a grammar, you would not know whether to start parsing the string as a B or a D. If you made the wrong choice, you would have to back up and repeat the process.

**Exercise:** Consider  $s = (( ( 1 ) * 2 + 3 ) * 5 * 6 + 7)$ .

1. Give a derivation of the expression  $s$ .
2. Draw the tree structure of the expression  $s$ .
3. Trace out the execution of your program on  $GetExp(s, 1)$ . In other words, draw a tree with a box for each time a routine is called. For each box, include only whether it is an expression, term, or factor and the string  $s[i], \dots, s[j - 1]$  that is parsed.