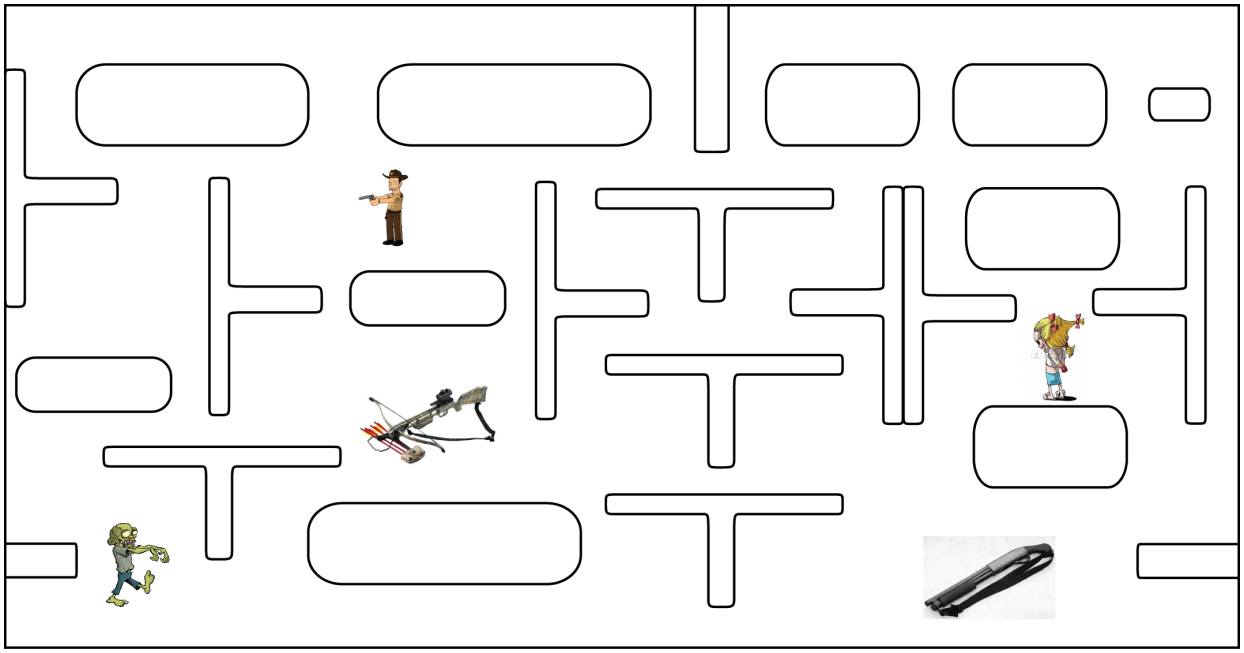
Introduction to Languages
Theory of Computation (CSCI 3500)

Prof. Harley Eades (heades@gru.edu).

The first topic we will study in this course is called "regular languages" that can be seen as a certain class of problems with a property called "regularity." If a problem can be characterized as regular, than there exists an efficient algorithm in both space and time that solves the problem. In fact, the space complexity of any algorithm that solves a "regular problem" is $O(1)$. Hence, they are extremely optimal.

Suppose we are working for a startup company that designs and implements different types of web-based video games. Furthermore, suppose we are designing a modern version of Pacman that consists of a main character trying to navigate a maze that has weapons placed throughout it, and there are zombies wondering around the maze. The following image is an example mock up of the game featuring Rick Grimes as the main character:



We are in charge of implementing the following algorithm:

- If Rick gets close to a zombie and has no amo, then Rick must run and loose the zombie.
- If Rick gets close to a zombie and has amo, then Rick must fight the zombie.
- If Rick is running and finds amo, then he must fight the zombie.
- If Rick runs out of amo while fighting a zombie, then he must run and loose the zombie.
- If Rick neutralizes the zombie, then he becomes safe.
- If there are no close zombies, then Rick is safe.

Now we can state this algorithm as a set of commands that operate a state based machine. Suppose we have the following commands:

$NCZ :=$ no close zombies

$ZCNA :=$ zombie close and no amo

$ZCFA :=$ zombie close and found amo

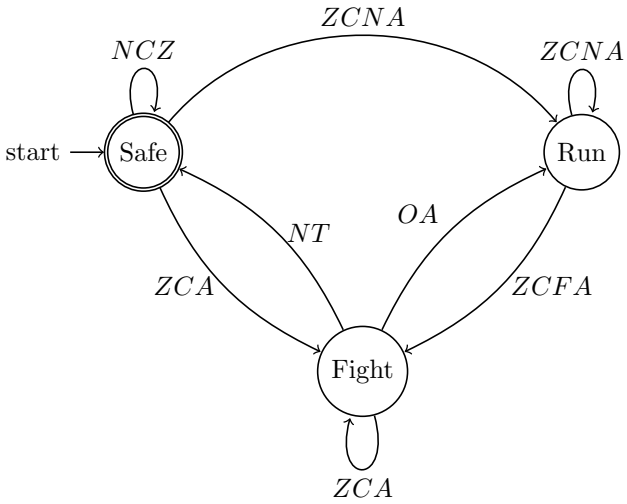
$OA :=$ out of amo

$ZCA :=$ zombie close and have amo

$NT :=$ neutralized threat

$LZ :=$ lost zombie

The following state machine describes our algorithm using the previous commands:



The previous diagram is called a deterministic finite automata. This machine describes everything we need to implement our algorithm that essentially uses no memory, and is very time efficient. We will see how these machines work formally later in the semester. The set of commands

$$\{NCA, ZCNA, ZCFA, OA, ZCA, NT, LZ\}$$

is called a formal language where each command is a letter of an alphabet, and sequences of commands are called words. In this lecture note we give the formal definition of a formal language, and then an example of how languages correspond to lists.

1 Alphabets, Words, and Languages

Every formal language has a set of generators called an **alphabet**. An alphabet is simply a set of symbols. It can contain any symbols at all. The following are some examples:

$$A = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$
$$B = \{0, 1\}$$
$$H = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$
$$\Sigma = \{+, \times, \cup, \cap, \wedge, \vee\}$$

Suppose Σ is an arbitrary alphabet. Then a **word** over Σ is simply a sequence of alphabet symbols juxtaposed together. Here are a few examples:

Words over A: $the, jenny, computerscience$
Words over B: $0101010101001, 00, 0, 1, 1110101$
Words over H: $123A, FEEDFACE, A9$

There is also a special word with no symbols called the **empty word** and it is denoted ϵ. We will often need to discuss **subwords** of words. Subwords are to words as subsets are to sets. Here are a few examples:

Subword of *computerscience* is *computerscience*.
Proper Subword of *computerscience* is *science*.

The following definition defines when one word is a subword of another.

Definition 1. *Suppose Σ is an arbitrary alphabet, and w is a word over Σ. Then a word, w', over Σ is a subword of w if and only if the following holds:*

- *w' and w are the same word,*
- *w' is the empty word, or*
- *if $w' = b_1 b_2 \cdots b_j$, then $w = a_1 a_2 \cdots a_i b_1 b_2 \cdots b_j a_{j+1} \cdots a_n = a_1 a_2 \cdots a_i w' a_{j+1} \cdots a_n$.*

We call a subword w' of w proper if and only if w' is distinct from w.

A **formal language** is any set of words over some alphabet. Consider the following examples:

$\Sigma = \{a, b\}$
$L_\Sigma = \{ab, aabb, aaabbb, \ldots\}$
$L'_\Sigma = \{a^n \mid n \text{ is a Fibonacci number}\}$

1.1 Languages in the Real World

A string – or word – in a language consists of the juxtaposition of some number of the alphabet symbols. We can think of this juxtaposition as an actual operation on sets called **concatenation**.

Suppose A is some alphabet. Furthermore, suppose A^* is the set of all possible words of any length composed of the symbols in A. Notice that $A \subset A^*$ and $\epsilon \in A^*$. Then symbol concatenation is defined as follows:

$$\begin{aligned} &\cdot_s : A \to A^* \to A^* \\ &a \cdot_s w = aw \end{aligned}$$

The above operation takes an alphabet symbol and a word and simply adds the symbol to the front of the word. We can now define any word in A^* using our new concatenation operation defined above.

Lets consider an example. Suppose $A = \{a, b\}$. We know that the word $aabbba \in A^*$ because it is a word consisting of only symbols in A. Now we can define this word using concatenation by

$$\begin{aligned} a \cdot_s (a \cdot_s (b \cdot_s (b \cdot_s (b \cdot_s a)))) &= a \cdot_s (a \cdot_s (b \cdot_s (b \cdot_s ba))) \\ &= a \cdot_s (a \cdot_s (b \cdot_s bba)) \\ &= a \cdot_s (a \cdot_s bbba) \\ &= a \cdot_s abbba \\ &= aabbba \end{aligned}$$

What happens when we try and concatenate a symbol of the alphabet to the empty string?

$$a \cdot_s \epsilon = a$$

This description of words using concatenation gives a very nice mathematical definition of a language. However, does this have any practical implications? It turns out that it does.

In computer programming – especially functional programming – there is a common data structure called a list. Lists are like sets but with an order – much like sequences. An example of a list is $[2, 4, 6, 8, 10]$.

Lists have one primitive binary operation called :: (cons) whose first element is an element one would like to add to the front of a list, and the second is the list to add the first argument to the beginning of. In addition, there is a special empty list denoted [] – the list with no elements. A list can be defined like $(A, ::, [])$ where A is some set of elements we want to build lists out of. We can also think of A as the type of the elements of the list. In C# lists are called `List<T>` where the empty list is `List<T>()`, and :: is defined by using `List<T>.insert` at position 0.

Now suppose A again is some alphabet. Then we can take any word of A^* constructed using symbol concatenation, and translate it into a list using :: and []. That is, we can define the following function:

$$\begin{aligned} &f : (A^*, \cdot_s, \epsilon) \to (A, ::, []) \\ &f(\epsilon) = [] \\ &f(a \cdot_s w) = a :: f(w) \end{aligned}$$

We can also define a function in the opposite direction:

$$\begin{aligned} &g : (A, ::, []) \to (A^*, \cdot_s, \epsilon) \\ &g([]) = \epsilon \\ &g(a :: l) = a \cdot_s g(l) \end{aligned}$$

Lastly, it is possible to prove the following:
Lemma 2.

i. For any $w \in A^*$ we have g(f(w)) = w.

ii. For any list l of elements of A we have $f(g(l)) = l$.

Proof. Informally, it is easy to see that by definition g undoes what f does. Thus, if we apply both in sequence we will end up where we started. There is a more formal proof, but we will not go into the details of that. □

Now this lemma has a an impact here. It shows that list construction has all the same mathematical properties that languages do!

In a perfect world when a programmer finishes writing a program they would prove that it is correct. This means that they prove mathematically that given any input no error state is ever reached. If a programmer had to do this for list concatenation, then they could use f to move over into the very mature world of languages. Then prove the properties they are looking for about languages, and then use g to move back to lists.