Introduction

Alex S.*

1 Introduction

> "Computer science is as much about computers as astronomy is about telescopes." - Edsger Wybe Dijkstra, 1930-2002

1.1 What is programming?

Programming is an intellectual activity that everyone is always involved in—it is a way of expressing ideas. Computer programming is a special type of programming, where you express your ideas to a computer. The computer in turn, if it understands what you mean, performs whatever you've asked of it.

This class is primarily concerned with ways of expressing your ideas to the computer, and with the techniques the computer uses to 'understand' what you mean.

1.2 Brief History

In computer science, history is not as important as in other subjects. Most of the time, programmers are more concerned with the future than the past. Discounting that, lets go for a brief historical introduction to computer science. If you think this section is boring and uninteresting, you're welcome to skip to the next one.

To the contrary of what you may think, computers were not invented in the past 50 years. As a matter of fact, they've been around for quite a while. It is however reasonable to assume that only now, computers have become truly popular, and necessary.

One of the first computers has been invented by a French mathematician, Pascal. (yes, just like the programming language) It was a mechanical digital calculator using gears and lots of other components which are not generally present in the current machines. It was called the Pascaline. Although crude, impractical, and expensive, it was good enough to make it into the history books.

A lot more activity went on about 200 years later. Joseph Marie Jacquard used punch cards to automate a weaving loom. Charles Babbage designed a Difference Engine; he

*alex@theparticle.com

later quit, and concentrated on his other invention: the Analytical Engine. Augusta Ada suggested the binary system, and George Boole developed Boolean logic, which was later (and still now) used to design computer circuitry.

In 1890, Dr. Herman Hollerith introduced the first electromechanical, punched-card data-processing machine which was used to compile information for the 1890 U.S. census. Hollerith's tabulator became so successful, that he started his own business to market it. His company would eventually become known as the International Business Machines. (yeah, the IBM!)

In the early 1900's, the vacuum tube was invented, giving rise to television, and lots of "computery" stuff. (yes, I just made that word up) In 1939, Dr. John V. Atanasoff and his assistant Clifford Berry build the first electronic digital computer. Their machine, the Atanasoff-Berry-Computer (ABC) provided the foundation for the advances in electronic digital computers.

A British mathematician Alan Turing developed a hypothetical device, the Turing machine, which would be designed to perform logical operations and could read and write. He also used vacuum technology to build British Colossus, a machine used to counteract the German code scrambling device, the Enigma. Turing's other popular achievement which earned him quite a lot of popularity with the science fiction writers, is The Turing Test. The test is suppose to determine if a machine is intelligent. A person, is talking to a machine, or another human, if that person can't determine which one is the machine, then the machine is intelligent. So far, nobody (or shall I say "nothing") has passed this test.

In 1944, Howard Aiken, in collaboration with engineers from IBM, constructed a large automatic digital sequence-controlled computer, called the Harvard Mark 1. This computer could handle all four arithmetic operations, and had special built-in programs for logarithms and trigonometric functions. (your pocket calculator is probably much more powerful ;-)

In 1945, Dr. John Von Neumann presented a paper outlining the stored program concept. This was almost the birth of programming. In 1947, came the transistor; invented by William Shockley, John Bardeen, and Walter Bratain of Bell Labs. It would rid computers of vacuum tubes and lots of bulky equipment.

After that, things started to fly. In 1949, Maurice V. Wilkes built the EDSAC (Electronic Delay Storage Automatic Computer), the first stored program computer. The EDVAC (Electronic Discrete Variable Automatic Computer), was the second stored program computer; built by the Mauchly, Eckert, and Von Neumann. The same year, An Wang developed magnetic core memory which Jay Forrester would later reorganize to be more efficient. A year later, Turing built the ACE, considered by some to the first programmable digital computer.

The first computer to be commercially designed specifically for business data processing applications was the UNIVAC 1, and was around in 1951. In later years, Dr. Grace Murray Hopper developed the UNIVAC 1 Compiler. Later that decade, John Backus, an IBM engineer, designed a cool programming language: FORTRAN (FORmula TRANslator).

In 1959, Jack St. Clair Kilby and Robert Noyce of Texas Instruments manufactured the first integrated circuit, or chip, which is a collection of tiny little transistors. Without one of these, computers would still be the size of football fields.

1960's, have witnessed a lot of revolutionary innovations in the computer industry. Gene Amdahl designed the IBM System/360, a series of mainframe computers; first general-purpose digital computer to use integrated circuits. Dr. Hopper was instrumental in developing COBOL (COmmon Business Oriented Language) programming language. Ken Olsen, founder of DEC, produced the PDP-I, the first minicomputer. In the middle 1960's, Dr.Thomas Kurtz and Dr. John Kemeny developed BASIC (Beginner's All-purpose Symbolic Instruction Code). And the greatest thing that begun in the 1960's is the Internet.

The 1970's did even better in relation to 1960's... In 1970, Dr. Ted Hoff developed the famous Intel 4004 chip. In 1971, Intel released the first microprocessor, a specialized integrated circuit which was able to process four bits of data at a time. It also included it's own arithmetic-logic unit. In the same year, Pascal, a structured programming language was developed by Niklaus Wirth.

In 1975, Ed Roberts, the "father of microcomputer" designed the first microcomputer, the Altair 8800, which was produced by Micro Instrumentation and Telemetry Systems (MITS). The same year, two young hackers, William Gates and Paul Allen approached MITS and promised to deliver a BASIC compiler. So they did and from the same, Microsoft was born.

A year later, Cray developed the Cray-I supercomputer, and Apple Computers, Inc. was founded by Steven Jobs and Stephen Wozniak. They later built the first Apple microcomputer.

In 1980, IBM offers Bill Gates the opportunity to develop the operating system for it's new IBM personal computer. Microsoft has achieved it's tremendous growth and success largely due to the development of MS-DOS. Same year, Apple III was also released.

In 1981, IBM PCs have become 16 bit. and in 1982, the Time magazine chooses the computer instead of a person for it's "Machine of the Year."

In 1984, Apple introduced the Macintosh computer, which incorporated a unique graphics interface, making it easy to use. The same year, IBM released the 286-AT, the first x86 chip that could work in Protected Mode.

In 1986, Compaq released the DeskPro 386 computer, the first to use the 80386 microprocessor. It expanded on the idea of 286s protected mode, and was the first 32 bit x86 chip. The 386 chip has virtually begun the computer gaming industry. It was fast and powerful enough for the complex calculations needed in most computer games.

In 1987, IBM announced the OS/2 operating-system technology, but by that time, it was way behind Microsoft's OS, the MS-DOS. A year later, a nondestructive worm was introduced onto the Internet, bringing thousands of computers to a halt.

In 1989, the world witnessed the arrival of the 486 chip, it began a more powerful resurrection of the 386 chip. It was also the world's first 1,000,000 transistor CPU.

In 1993, The Energy Star program, endorsed by the Environmental Protection Agency (EPA), encouraged manufacturers to build computer equipment that met power consumption guidelines. When guidelines are met, equipment displays the Energy Star logo. The same year, several companies introduced computer systems using the Pentium microprocessor from Intel that contains 3.1 million transistors and is able to perform 112 million instructions per second (MIPS). Pentium's ability to perform fast floating point calculations has made it

quite popular with computer gamers (with it, revolutionary games like Quake by id Software were possible).

With this innovating technology, and increased processing power of newer and newer chips, the Internet has become a new medium of communication. Almost everybody owning a computer has some sort of connection to the Internet, able to send e-Mail, and view web sites. Computery history is changing every day, with every year (as you can see), newer and newer generations of computers are being invented, and with them, come almost instantaneous changes.

(eventually, I'll add more history here)

2 Language Landscape

We've defined programming as an intellectual activity—where we express our ideas. There are, however, many ways that we can express our ideas.

Computers, being relatively primitive machines, cannot speak our language (English, and so forth), so we need to speak their language in order to facilitate communication. As computers get more advanced (faster, with more memory), computer languages also become more expressive, and natural.

2.1 Assembly to the Rescue

As the history section indicates, first computers were mechanical. To 'program' that computer, you might have needed to rearrange gears. Later computers were electronic; to program those, you might have had to solder wires together. A bit later, when a concept of a stored program was developed, you could program computers by creating a series of bits that the processors would understand (and fed that into the computer via a punch card). A program may look something like Figure 1.

It is (or rather was) extraordinarily tedious and error prone to program like this, so assembly languages were created. An assembly language has a one-to-one correspondence with the binary representation, but is human readable. Figure 2 presents an example of assembly language.

An assembly program is translated to machine language with the help of a program called, appropriately enough, "assembler."

Assembly languages were very popular when programmer time was cheaper than computer's time. Soon however, a higher level language was needed; and the world witnessed the birth of Fortran, Lisp, Prolog, COBOL, Pascal and a bunch of others. The key thing about "high level" languages is that they must be translated into assembly (or machine language) - and this is where the whole point of this book comes into play.

```
0000000 d6bf be7d 7d6b 00b8 8ea0 b8c0 0013 10cd
0000010 01b4 16cd 3274 e432 16cd 713c 0775 03b8
0000020 cd00 cd10 3320 33db 3eed 863a 0199 0f75
0000030 d155 d1ed 3ee5 968b 01a9 9789 01b9 835d
0000040 02f3 8345 10fd e17c 3e03 01b9 8026 003d
0000050 0574 8eba eb01 2622 05c6 0304 bb36 2601
0000060 3c80 7400 ba05 0184 0feb c626 0104 c933
0000070 00ba b4f0 cd86 eb15 b897 0003 10cd 09b4
0000080 21cd 20cd 6552 2064 6957 736e 2421 6c42
0000090 6575 5720 6e69 2173 3824 3665 3266 3463
00000a0 3773 3977 3372 3176 c078 01fe 4000 ff01
00000b0 bfff c1fe 41fe 3f01 c001 c0fe 00fe 0000
```

Figure 1: A computer program.

```
.MODEL TINY         ;generate a .com
.CODE
.STARTUP            ;start of execution
mov     ax,0305h    ;set keyboard speed function
xor     bx,bx       ;fastest possible
int     16h         ;BIOS service
END
```

Figure 2: Assembly language program.

2.2 Different Language Types

There are three [1] major "high-level" programming language types—each with its own basic philosophy behind it. There are *declarative* languages—these concentrate on *what* the computer should do, and *imperative* languages - and these concentrate on *how* the computer should do something. There are also *functional* languages—where you program by creating functions from other functions (for the most part).

Here's an example of the difference; if you have a list of people and you'd like to find someone named "John." In a declarative languages, you may have a program that says (hypothetically): *display everyone named "John."* Notice that you just say *what* you want, now *how* the computer should accomplish it. In an imperative language, you might go about saying something like: *loop through every person in the list, compare their name to "John," if they match, then display their record.*

Declarative languages are often thought of as higher level than imperative languages. Imperative languages are most widely used however, partly because of performance implications, and partly because that's what schools teach their students - very few programmers ever learn declarative languages (C# or Java sound cooler than Prolog).

Figure 3 presents an example of a declarative language (in this case, Prolog).

Don't worry if it seems cryptic and unfamiliar, we will go over it a bit later. As an example of an imperative language, Figure 4 presents a C program to find the Greatest Common Divisor.

3 Compilation Overview

Compilation is just a fancy word for *translation*. A program in a high-level language needs to be compiled (translated) before it can be utilized by the computer. There are many variations of what exactly happens (and how it happens), but the general flow of events is that a high-level program (say in C language) enters the *compiler*, which translates it to assembly, which is then fed into an *assembler*, which outputs object code. This object code is then fed into a *linker*, which creates the actual executable you can run.

The compiler usually has many modules (or phases). Usually, input is read by a lexical analyzer (sometimes called *scanner* or *lexer*). This has the effect of breaking the input character stream into tokens. These tokens are then analyzed for syntax, by a *parser*. The result of these two phases is a *parse tree* of the input program (in some internal memory representation). The lexer and parser are sometimes called the *front end* of the compiler. They determine what the input language is. You then usually have some semantic analysis phase (figuring out what the program means).

Most compilers then run all sorts of optimizations - eliminating invariants from loops (variables that are not changed), etc. Then they proceed onto code generation.

Generating code from a parse tree is basically just walking the parse tree, and outputting appropriate assembly code for every node that you encounter. There might also be some

[1] There are actually many different types, but they're subsets of the two described.

```
% some simple 'standard' definitions
% member~relation
member(X,[X|Tail]).
member(X,[Head|Tail]):-
    member(X,Tail).

% do list concatenation
conc([],L,L).
conc([X|L1],L2,[X|L3]):-
    conc(L1,L2,L3).

% select X from a list
select([X|Xs],X,Xs).
select([X|Xs],Y,[X|Ys]):-
    select(Xs,Y,Ys).

% generate permutations
perm([],[]).
perm(Xs,[X|Ys]):-
    select(Xs,X,Xs1),
    perm(Xs1,Ys).
```

Figure 3: Example of Prolog code.

```
#include <stdio.h>
#include <stdlib.h>

/* function to find Greatest Common Divisor of x and y */
int gcd(int x,int y){
    int g;
    if(x < 0)
        x =-x;
    if(y < 0)
        y =-y;
    if(x + y == 0)
        return 0;
        g = y;
    while(x > 0){
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

int main(int argc,char** argv){
    int g,a,b;
    if(argc < 3){
        printf("usage: %s [number1] [number2]\n",argv[0]);
        return 1;
    }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    g = gcd(a,b);
    if(g != 0){
        printf("GCD of %d and %d is: %d\n",a,b,g);
    }else{
        printf("Error.\n");
    }
    return 0;
}
```

Figure 4: C program to find the Greatest Common Divisor.

machine specific optimizations involved. This code generating phase is often referred to as the *back end* of the compiler, which determines on which machine the program will run.

4 Interpreters

When we think of interpreters we think of a person who translates from one language to another, a sentence or two at a time. You say something, they translate that, you say something more, and they translate that as well. In a computer sense, the meaning is very similar.

While a compiler translates the whole program to assembly, (then machine language), which is then executed, interpreters translate small parts of the program at a time, which are then executed[2]. This has the seeming effect of executing source code (or executing human readable code on the fly). The techniques for reading the program and getting to the binary representation are nearly identical for a compiler and an interpreter; the interpreter however has an extra module to actually execute the code. Execution is often done by a *virtual machine*, which is just a program that simulates the execution of another program.

Interpreters have become very robust, and for certain applications rival the perceived speed of compiled programs. Some examples of interpreters include the Perl, and Basic.

4.1 Hybrid Systems

Several recent systems were not exactly compilers nor interpreters - they're both. Java for example, has a compiler that accepts Java language programs, and outputs a byte-code representation that is machine independent. The Java VM (virtual machine) then takes that byte-code and executes it anywhere. It is the same idea behind Microsoft's CLR (Common Language Runtime) and their C# compiler (and in fact all of the .NET languages).

5 A Bit of Theory

> "People don't understand computers. Computers are magical boxes that do things. People believe what computers tell them." - Bruce Schneier, Secrets & Lies

5.1 Intro

Perhaps in no other area in Computer Science that theory is so paramount than in programming languages, and the structure of compilers. This section will present a brief introduction

[2]Actually, most modern interpreters usually translate the whole program to machine code - for virtual machine, then run it; they never output the compiled binary to disk and so appear to be executing source code.

to set, language, and automata theories - just enough for us to make sense of the rest of this book. It is by far not the most complete reference (ie: read the book)

6 Sets

A set is just an object that includes other objects. There are two primary ways of specifying one, either by listing the elements, or providing some property that is characteristic to elements of the set.

$$A = \{2, 3, 5, 7, 11\}$$
$$B = \{a, b, c, d, e, f, g\}$$
$$C = \{2, 4, 6, 8, 10, 12, ...\}$$
$$D = \{x \mid x \text{ is a positive even integer}\}$$
$$E = \{x \in A \mid x < 5\}$$
$$F = \{x \mid x \in A \text{ and } x \notin C\}$$

Figure 5: Examples of Sets

In Figure 5, the A set has five elements (the first five primes). Set B is a set of seven letters. Note that sets can contain any objects, even other sets; which is why they're so useful in math and science. Set C is an infinite set of all even integers. We don't specifically list all even integers (which is impossible) but we provide a few examples, and the '...' indicate that the sequence continues. Some of these can be ambiguous, but usually the context makes it clear what is meant. Set D is also an infinite set of all positive even integers. We can also use other sets in our definition, as in set E, which contains elements $\{2, 3\}$.

You can interpret the '|' to mean *such as*. The $\in$ symbol indicates that something is an *element of* a set. Similarly, $\notin$ indicates that something is not an element of a set. For example, saying $a \in S$, not even knowing what the set S is, we know that a is a member of it.

For finite sets, we can speak of set length as $\mid A \mid$. For example, for set A from Figure 5, the $\mid A \mid = 5$. For infinite sets, size gets a bit more complicated [**?**]. There is also a concept of an *empty set*, (a set that has no elements) written as . Naturally, for an empty set, $\mid \mid = 0$.

6.1 Set Operations

Sets are only useful when we consider them in context of other sets. This section presents some of the more common operations; there are certainly others that are not mentioned here. For a more detailed view, consider reading [**?**].

For this section, lets use these four example sets: $S1 = \{a, b, c\}$, $S2 = \{c, a, a, b, c, a\}$, $S3 = \{a, b, c, d\}$, and $S4 = \{d, e, f, g\}$.

When we write $A = B$, then what we're saying is that both set A and set B have the same elements. For example, $S1 = S2$. Notice that they have the same elements; order nor number of occurances of members don't matter.

When we write $A \subseteq B$, we're saying that A is a *subset* of B, and that set B has all the elements of A (and possibly others). Now, convince yourself that if $A \subseteq B$ and $B \subseteq A$ then $A = B$. An example is: $S1 \subseteq S2$ and $S2 \subseteq S1$ then $S1 = S2$.

Where the $\subseteq$ is not precise enough, we have $\subset$, which is a *proper subset*. We say $A \subset B$ to mean $A \subseteq B$ but $A \neq B$. An example might be $S1 \subset S3$.

An important property is that an empty set is a subset of any set: $\subseteq S$ for any set S. So saying $\subseteq S1$ is perfectly valid.

The minus (or set difference) operation is $A - B$, and is about removing all the common elements from the first set. For example, the set $S1 - S3 =$, while $S3 - S1 = \{d\}$.

Given two sets, A and B, we say $A \cup B$ to mean the *union*, a set that contains all elements of set A and set B. The sets $S1 \cup S2 = \{a, b, c\}$, and $S1 \cup S3 = \{a, b, c, d\}$.

We say $A \cap B$ to mean the *intersection*, a set that contains elements that are common to both set A and set B. The set $S1 \cap S3 = \{a, b, c\}$.

Figure 6 presents these more formally.

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$
$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

Figure 6: Important set operations

Sets A and B are said to be *disjoint* if they contain no common elements. We can express this by writing $A \cap B =$. For example, sets $S1$ and and $S4$ are disjoint, since $S1 \cap S4 =$.

Given any set A, we define the set A' to be a set of all elements that are not in A. This only makes sense in the context of a *universal set* U. More formally, $A' = \{x \in U \mid x \notin A\}$. Note that universal set is sometimes tricky; if you're dealing with numbers, than universal set includes all numbers, if you're dealing with strings, then universal set includes all strings, etc. The universal set depends on the context of what is it a universal set of.

Figure 7 illustrates some basic set laws.

Sometimes it is useful to refer to the set of all subsets of a given set. It is written as 2^S for any set S. For example, what are all the subsets of $S = \{a, b, c\}$?

$$2^S = \{, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

The number of subsets is actually $2^{|S|}$, so since $\mid S \mid = 3$, the $2^3 = 8$, and thus, there are 8 subsets (notice that is a subset too).

Cumulative Laws
$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$

Associative Laws
$$A \cup (B \cup C) = (A \cup B) \cup C$$
$$A \cap (B \cap C) = (A \cap B) \cap C$$

Distributive Laws
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

De Morgan Laws
$$(A \cup B)' = A' \cap B'$$
$$(A \cap B)' = A' \cup B'$$

Figure 7: Important Set Laws

7 Tuples

Two sets are equal if they have the same elements. So for example, sets $\{a, b, c\}$ and $\{c, c, a, b, a\}$ are equal. In fact, when dealing with sets, we have no way of distinguishing one from the other. Thus, when order (and number of occurrences) is important, we instead deal with n-tuples. The n in n-tuple stands for the number of elements in the tuple, for example: $(a_1, ..., a_n)$.[3]

So a tuple is basically just a list. A $(1, 2, 3, 4)$ is a tuple, and so is $(2, 4, 5, 2, 3, 4)$. We do not distinguish between a an element itself and a 1-tuple. For example, (x) is the same as x. A common name for 2-tuple is an *ordered pair*, and similarly, a 3-tuple is an ordered triple.

When comparing tuples, they are only equal when all their elements (and their order) is equal. For example, if we know that two tuples are equal,

$$(a_1, a_2, ..., a_n) = (b_1, b_2, ..., b_n)$$

then we know that

$$a_1 = b_1, a_2 = b_2, ..., a_n = b_n$$

You can think of tuples as being elements of a set formed by a *Cartesian product* of other sets. For example, if we have sets $S_1, S_2, ..., S_n$, and we form their product, $S_1 \times S_2 \times ... \times S_n$, the resulting set is the set of all tuples $(a_1, a_2, ..., a_n)$ such that $a_1 \in S_1, a_2 \in S_2, ..., a_n \in S_n$. As a shorthand, we may write S^n for $S_1 \times S_2 \times ... \times S_n$.

[3]We write tuples with parenthesis as opposed to curly braces.

8 Strings

Strings are key in the study of languages. Intuitively, a string is a *finite* sequance of symbols. A bit more formally, a string as a tuple of symbols, except we don't use parenthesis or commas. For example, a tuple (a, b, c) is the same as string abc. We get string lengths in a similar fashion as set lenghts, if s is some string, then $\mid s \mid$ is the length of s.

8.1 String Operations

Just like sets, strings have a few basic operations[4]. Given two strings, we can catenate them, for example, string foo catenated with string bar produces a string $foobar$ (the two strings stuck together).

There is also sometimes the notion of string union. This does not make sense at first (strings are tuples after all), but upon further reflection, nothing is stopping you from creating a set of the two tuples. For example:

$$\{foo\} \cup \{bar\} = \{foo, bar\}$$

It is probably important to to note that the result is no longer a string, but a set.

9 Languages

Consider for a moment the English language. When you open a dictionary, you encounter a bunch of words - is that 'language'? A collection of words doesn't exactly help you write in English (writing down a few random words doesn't necessarily mean that you've written something in English). What is needed is some notion of a valid string, or sentence. How about defining a language as a set of all valid sentences? The set may be infinite, but it will represent any possible utterance that we will consider as English.

The languages we deal with in this chapter are similar to English, except they're much simpler. Instead of having large alphabets, our languages will only have 2 - or so - letters (symbols), and the rules for constructing valid strings will be much more rigid than they are in English. So keeping that in mind, let us proceed.

An *alphabet* is a set of symbols or letters. An alphabet may be $\{0, 1\}$ or it may be:

$$\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

either way, it is just a set of characters. We form languages by doing Cartesian products of alphabets. Recall that a Cartesian products produces a set of tuples - or in our case, a set of strings.

If we have an alphabet $\sum = \{a, b\}$, and we let set $L = \sum \times \sum \times \sum$, (or $L = \sum^3$) then:

$$L = \{(a, a, a), (a, a, b), (a, b, a), (a, b, b), (b, a, a), (b, a, b), (b, b, a), (b, b, b)\}$$

[4]We will deal with more complicated operations when we deal with language grammars.

or basically a set of all 3-tuples formed by the cartesian product. We also say that L is a language. Note that usually in our dealings with such languages, we'll write them out as sets of strings (as opposed to tuples):

$$L = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$$

A *language* is a set of strings of some symbols (or *alphabet*). Formally, a language is a subset of $\sum^\star$ where $\sum$ is some alphabet (like $\{a, b, c\}$ for example), and the $\star$ indicates that we take some arbitrary number of Cartesian products.

Also, just like sets have an empty set as their subset, the $\sum^\star$ has λ or an empty string; $\lambda \in \sum^\star$.

Here's an example: Let $\sum = \{0, 1\}$, and let $L = \{x \in \sum^\star \mid x$ has odd number of 0s and 1s$\}$. What are some of the strings of this language? Is λ in the language? Not really - it doesn't have any 0s or 1s. We can begin to list some strings of language L.

$$01, 10, 0001, 0010, 0100, 1000, 1110, 1101, 1011, 0111, ...$$

Notice that we can go on forever - the size of strings grows, but they all have odd number of 0s and 1s - and they are all members of $\{0, 1\}^\star$[5].

Most languages are quite large (infinite in fact), and it is beyond our ability to list all member strings. So how exactly do we describe languages? This is where we hit our next topic - regular languages. There are many types of languages, and they differ in complexity of what type of a machine you need to recognize them or what type of notation you need to describe them.

10 Regular Languages

Many languages are infinite, and we need a way to describe them in finite ways. One way is to start with a simple element of a language, and show how we can construct other elements of that language using simple string operations. The set of languages we will examine now are called regular languages, and are generated by three basic operations: catenation, union, and Kleene* (the Kleene star).

String union and catenation have been defined before (Section 8.1). The Kleene* signifies repetition of something that preceded it 0 or more times. For example, $a*$ can generate λ (if it repeats 0 times), a if it repeats 1 time, aa if it repeats twice, and so on and so on.

TODO...

10.1 Regular Expressions

TODO...

[5]This is just a shorthand for $\sum = \{0, 1\}$, and then $\sum^\star$.

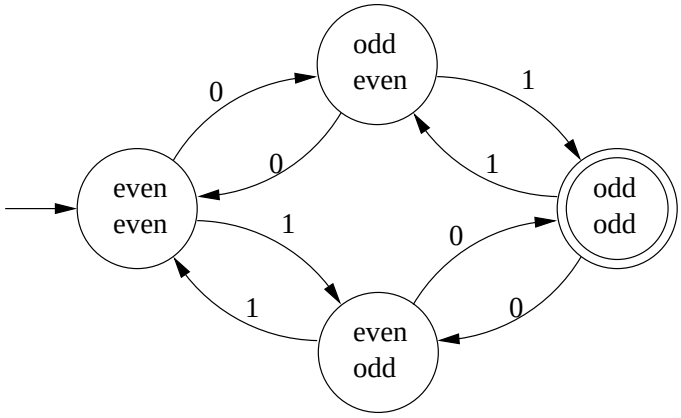

Figure 8: Finite Automaton that accepts strings with odd number of 0s and 1s.

10.2 Finite Automata

TODO; Here it is... Figure 8.

FA is a 5 tuple - $(Q, \sum, q_0, A, \delta)$, etc...

10.3 Nondeterminism

TODO; stuff about NFA-λ, NFA, and DFA, and converting from one to another.

λ-closure

Subset construction.

10.4 Kleene's Theorem

TODO; converting a regular expression to a DFA.

11 Context Free Languages

Pal (language of palindromes) is context free.

TODO; Same as regular languages, only with recursion. Have no context.

11.1 Context Free Grammars

Grammar is a 4-tuple, $(V, \sum, S, P)$, where V is a set of variables, $\sum$ is a set of terminals, S is the starting variable (S$\in V$), and P is a set of formulas of the form A-¿ something, where A is element of V, and something is element of (V union $\sum$)*

derivation trees, ambiguity

left recursion, common prefixes

getting rid of lambdas

11.2 Push-Down Automata

PDA is a 7-tuple, $(Q, \sum, \Gamma, q_0, Z_0, A, \delta)$, where Q are states, $\sum$ is input alphabet, Γ is the stack alphabet, q0 is the first state, Z0 is the initial stack symbol (element of Γ), A is a set of accepting states (subset of Q), and $\delta : Q \times (\sum \cup \{\lambda\}) \times \Gamma$ (finite subsets of $Q \times \Gamma*$.

We move from "configuration" to configuration.

configuration of PDA is current state, input string, and top of stack