ArsDigita University
Month 8: Theory of Computation
Professor Shai Simonson

Lecture Notes

What is this Course About?

Theory of computation is a course of abstractions about what we can compute. It is the purest of all computer science topics attempting to strip away any details about the real computer and replacing in it with abstractions that give a hierarchy of capabilities culminating in a Turing Machine, the abstract model of our modern computer, invented by Alan Turing. Abstractions give us a way of saying *something* about what we are doing in a rigorous formal way, rather than flailing randomly through a haze of informality. There is great value in abstraction.

Applications

It turns out, as it often does with good abstractions, that the theory of computation provides us with many good applications. Finite state machines are used in string searching algorithms, compiler design, control unit design in computer architecture, and many other modeling applications. Context free grammars and their restricted forms are the basis of compilers and parsing. NP-Complete theory helps us distinguish the tractable from the intractable. We do *not* focus on these applications. They are deep enough to require a separate course, and hopefully you have already seen some of them. There is plenty of basic theory to learn, and we will concentrate on providing a thorough understanding of all the abstract ideas with a constructive focus. That is, when we prove things it will be through examples. When we define things, we will motivate their definitions through examples.

Overview

What can be computed? For the purposes of this course, we consider a very simple model of a *problem*. A problem is a set of strings (often binary strings) that we wish to distinguish. For example, one problem might be the set of all binary strings divisible by three. To solve this problem we need to be able to say *yes* or *no* to each candidate binary string as to whether it is in the set or not. If we have an algorithm to do this, then we have solved the problem. These kinds of problems are called *decidable*.

More complicated inputs over larger alphabets can be considered as well. For example, we can consider lists of numbers as the input, where each list has numbers written in binary with a # symbol

separating each one from the other. A problem is the set of all such strings where the numbers are in ascending order. Even more complicated examples are easy to imagine. Consider the set of all syntactically legal Java programs. These strings are over a large alphabet of symbols. This problem can be solved with a compiler. Consider the set of all syntactic legal Java programs that never go to into an infinite loop on any input. There is no way to *solve* this problem. This problem is *undecidable*.

For our purposes we think of a machine, or automaton, as a guard at the door of an exclusive club that admits only specific kinds of binary strings. The machine is given a program by the boss. It looks at each candidate string trying to enter the club, and it executes the program by looking at deciding for each one: yes or no.

There are different kinds of machines, each one a little more powerful than the rest. What kinds of sets are recognized by different machines? What features can you add to a machine to make it more powerful, that is able to recognize sets that it could not recognize before? Can we measure time and space requirements of a machine as a tradeoff to this power?

The chart below summarizes the levels of machines we will discuss in this class. The machines have equivalent formulations as grammars or other forms.

Machine	Grammar	Other
Finite State Machine	Right/Left Linear Grammars	Regular Expressions
Deterministic Pushdown Machine	LR Grammars	Syntax Diagrams
Non-Deterministic Pushdown Machine	Context Free Grammars	
Turing Machine	Unrestricted Grammars	Recursive Sets

The top level is the least powerful and we start from there.

Finite State Machines

These notes will stay away from formal definitions and proofs, and you are referred to the text for those. The notes will provide intuition and perspective, which can be an obstacle in trying to read any material like this.

A finite state machine is a kind of very limited type of computation that has no memory or data structure except for what you can encode directly into the machine. Every FSM looks like a directed

graph where the nodes are called states and the edges are called transitions. The edges are labeled with symbols of the alphabet, and we run the machine by reading the input one symbol at a time from left to right, and following the transitions in the graph. We start at a designated start node. When we are finished reading the string, we decide whether or not we accept that string (answer 'yes') depending on the state that we end up in. If the state is marked final, then we say yes, otherwise no.

Any set of strings accepted by an FSM is called a regular set.

Examples of Regular sets for the alphabet {0,1}.

a. Even number 1's.
b. Strings containing 101.
c. Even number of 0's and contains 101.
d. Even number of 0's or contains 101.
e. All strings.
f. Divisible by three as a binary number.
g. Every one has at least two zeros that follow it.
h. Not divisible by three.
i. A finite set.
j. Second symbol not a one.
k. Some two zeros are separated by a number of symbols that is a multiple of three.
l. End with 00 or 01.

These examples will be done in class and will teach four important lessons:

1. To design FSM's, give semantic information to the states.
2. Closure properties of a class of sets, help identify sets in the class. Constructive closure proofs provide actual FSM's for these sets.
3. Equivalence of FSM variations provides us with high-level tools for designing FSM's.
4. There is more than one FSM to recognize a set, but there is a unique one with the minimum number of states.

FSM's are under complement (h), union (c), intersection (d) and reverse (l). We will explain the first three in general with examples in class. The proof of the fourth has a bug that will motivate the following variation of an FSM. Are FSM's closed under double? That is double(L) is the set of all strings xx where x is in L? Note interestingly that FSM's are closed under half(L).

Non-deterministic Finite State Machines

We now consider a variation of the FSM that will make it much easier to design FSM's. The notion of non-determinism is a fundamental one that we will revisit many times in this class. You already are

familiar with non-determinism from month 5 (Algorithms) and the theory of NP-Complete problems. Here we will be more formal and careful with the definition.

We will show that anything you can do with a non-deterministic FSM you can also do with a deterministic FSM, implying that using non-determinism in the design of an FSM always accepts a regular set.

A deterministic FSM has an arrow coming out of each state for each symbol in the alphabet. If an arrow is missing, we assume that it actually goes to a dead state, that is a state with two self-looping arrows (for 0 and 1). A non-deterministic machine may have any number of arrows coming out of each states with 0 and 1 appearing on the transitions a arbitrary number of times.

How do we run a non-deterministic machine? We have potentially many choices of directions to move after reading each symbol of the input. Well we don't really run the machine, at least not in the sense that we run a deterministic machine. However, there is a very formal and precise definition as to which strings are accepted by a given non-deterministic FSM. If we can read the symbols of a string and find some path that ends up in a final state then we accept that string. On the other hand, if every choice of path ends up in a non-final state then we reject the string.

For example, we will consider (b), (k) and (l) above and show how non-determinism helps construct the machines more easily.

There are more complicated regular sets that do not yield to a casual attempt at designing the FSM, but yield readily to a non-deterministic FSM. You will see some hard ones in your psets. This should remind you of how hard it is to solve problems like Towers of Hanoi without recursion, even though in principle recursion is just a convenient tool that can always be simulated with stacks and iteration.

It is common to discuss variations of machines in order to determine which variations actually allow us to accept sets that we could not accept before. Almost all variations of FSM's do not add power. These include 2-way FSM's, non-deterministic FSM's, alternating FSM's, and FSM's with lambda productions.

We show constructively how to take a non-deterministic FSM and turn it into a deterministic FSM. This lets us complete the proof that Regular sets are closed under reversal. Regular sets are closed under many operations, some of which are hard to prove.

One interesting implication of this is that a binary string is divisible by three if and only if its reverse is divisible by three. We can prove this by taking the FSM for the former machine, constructing its reverse, and check that it is identical to the original. The only thing is, that it may accept the same set and not be identical. In order to do this right, we have to minimize both machines and then they are identical if and only if they accept the same set. This means we need to learn how to minimize FSM's.

Minimizing FSM's

The algorithm to minimize an FSM is based on the fact that the states can be partitioned into disjoint sets where all the states in each state are equivalent. This equivalence relation is defined as follows: two states A and B are equivalent if a given string could start in either place and its acceptance status would be identical. There is a more precise way to say this, but it is obscure and hard to state. Examples will be done in class.

There is a folklore method to minimize an FSM. Reverse it, convert it back to a deterministic FSM and throw out unreachable useless states. Then repeat the process. The reason why this works is not readily apparent. The complexity is worst case exponential.

There is a more intuitive algorithm that has an O(n^2) time algorithm which is basically a dynamic programming style solution. We will do an example in class.

By the way, if you use the right data structure and are clever, you can implement the previous algorithm in O(n log n). Extra credit for any solutions.

Regular Expressions

We now consider a completely different way to think about regular sets called regular expressions. Regular expressions are defined inductively. Every single alphabet symbol is a regular expression. The empty string, lambda, is a regular expression. The empty set, that is nothing, is a regular expression. These are the base cases. If R and S are regular expressions then so are R+S, RS, R* and S*. For example 0*1 + 1 is a regular expression. The semantic interpretation of R+S is the union of all strings in R and S. RS is the set of strings xy, where x is in R and y is in S. R* consists of all the strings which are zero or more concatenations of strings in R. For example. 00011101 is in (00+01)*(11+00)01.

It turns out that any FSM can be converted to a regular expression, and every regular expression can be converted into a non-deterministic FSM.

We will prove this constructively in class, showing the equivalence of the three items in the first row of our original table.

Right Linear Grammars

A grammar is another way to represent a set of strings. Rather than define the set through a notion of which strings are accepted or rejected, like we do with a machine model, we define the set by describing a collection of strings that the grammar can *generate*.

A grammar is described by *productions*. Best to start with an example and then give some terminology. S ✍ 0B B ✍ 0S S ✍ 1S B ✍ 1B S ✍ ?

S and B (normally all upper case letters) are called non-terminal symbols because they do not appear in the final generated strings. 0, 1, and ? are called terminal symbols, where ? is the empty string. All strings generated by the grammar consist of non- ? terminal symbols.

There is a unique non-terminal symbol called the start symbol usually designated as S. A production is a substitution of one set of symbols for another. The left side is replaced by the right side. A sequence of productions starting with S and ending with a terminal string x, is said to generate the string x. For example, S ✍ 0B ✍ 00S ✍ 001S ✍ 0011S ✍ 0011, shows that the grammar generates the string 0011. See if you can figure out which production was used in each step of the *derivation* of this string.

The grammar above is a very special kind of grammar called a right-linear grammar. In a right-linear grammar, every production has a single non-terminal symbol on the left, and either a single terminal symbol on the right or a combo terminal followed by non-terminal symbol.

It is not too tough to prove that every right-linear grammar generates a regular set, and that every regular set can be generated by a right-linear grammar. Examples will be done in class. The grammar above corresponds to the set of strings over the alphabet {0,1} that have an even number of zeros. The idea is that the non-terminal symbols correspond to states, and the terminal productions correspond to final states.

General Information about Grammars

There is a hierarchy of grammars, each of which can recognize more languages than its predecessor. Left-linear grammars are the same level as right-linear grammars and you are asked to show this in a problem set. Context free grammars keep the restriction on the left side of each production but lose the restriction on the right side. This lets us use a production S ✍ 0S1, which combined with S ✍ ? generates the language 0^n1^n, which is not a regular set. There are grammars in between context free and right-linear grammars called LR-grammars. There are less restricted grammars called context-sensitive grammars, which demand only that the length of the left side be smaller than the length of the right side. There are unrestricted grammars, which can recognize any set that a Turing machine can recognize. The LR grammars have a restriction that is complicated to describe but they represent the most practical use of grammars, and are fundamental in the design of compilers and for parsing.

Sets that are Not Regular

What sets cannot be recognized by finite state machines? We mentioned 0^n1^n in the last paragraph as a non-regular set. How do we know this? If I ask you to try to design an FSM to recognize 0^n1^n, you will not succeed. What do you discover as you try? One possibility is that you can do if you had an unlimited number of states (an infinite state machine). Another possibility is that you design a machine that accepts all the strings in 0^n1^n but also accepts plenty of other strings. But none of these failures point to a convincing reason why I should not just send you back to try harder.

The Pumping Lemma

The pumping lemma for Regular sets is a way to convince me that a set is not Regular. I say it like that, because the best way to understand the lemma, is to think of it as an adversarial conversation. You say it is not Regular, and I say you are not trying hard enough and that I have an FSM that accepts the language 0^n1^n. You then ask me how many states there are in my machine. I answer, say 238. You then ask me to run my machine on the string $0^{238}1^{238}$. While I start working, you point out that somewhere in the first 238 symbols, there must be a time where I will enter the same state a second time. (This is due to the pigeonhole principle, because we have 238 movements and only 237 different places we can go from the start state). You ask me to notice the length of this loop, and you point out that the symbols on this loop are all zeros. I tell you that the length of this loop is 32, and then you make the final winning point that if so, then my machine will also accept $0^{238+32}1^{238}$, which it should most certainly not.

This whole adversarial argument can be made more rigorous by having your arguments work in general regardless of my answers. In this case, my machine has k states, and your string is 0^k1^k. I must indicate a loop in the first 238 symbols. That is $0^k1^k = vwx$, where $|vw| <= k$ and $w >= 1$, (v is the computation before the first loop, w is the loop, and x is the rest of the computation). You point out that $vwwx = 0^{k+|w|}1^k$ must also be accepted by my machine, but it should not be!

The Pumping lemma formally looks like this:

Let R be a Regular set. For any z in R, there exists an n (specifically, the number of states in a machine accepting R), such that z can be written as vwx, where $|vw| <= n$ and $|x| >= 1$, and for $i >= 0$ vw^ix is also in R.

The lemma is usually used in its contrapositive form. That is, if the *pumping* condition is not true then the set is not Regular. Note that if the pumping condition is true, the set might be Regular and might not. That is, the converse of the pumping lemma is not true. It is not if and only if. You will be asked in a problem set to identify a non-Regular set where the pumping lemma is true.

There are many sets which can be shown to be non-Regular using the pumping lemma. We will do some examples in class, and you will have plenty of practice in your Psets.

Diagonalization – Another Way to Identify Sets that are Not Regular

The pumping lemma is very convenient for showing that certain sets are not regular, but it is not the only way. A more general method that works for all kings of different machines is called diagonalization. The trick of diagonalization goes back to Cantor in the late 19th century who used the technique to show that there are infinites that are bigger than the infinite of the integers.

Here is the trick in a nutshell. Later we will explain why it is called diagonalization.

We consider binary strings that represent FSM's. Imagine that we encode an FSM itself in binary. There are many ways to do this and one will be discussed in class. Then some FSM's will accept there own representation and some will not. The ones that do are called self-aware, and the ones that don't are called self-hating. Now consider the set of all binary strings that represent self-hating FSM's. Is this set Regular?

If it were Regular, then there is an FSM Q that accepts the set. Does Q accept itself? If it does, then it is a self-hating FSM, which means that it should reject itself. But if it rejects itself, then it is not a self-hating machine, and it should accept itself!

Strange, Q can neither accept or reject itself logically, hence Q cannot exist!

The neat thing about this trick is that it does not depend on the fact that the machine is a FSM. It could just as well be any kind of a machine. There are generalizations of the pumping lemma but the technique of diagonalization works generally.

Why is it Called Diagonalization

If we think of a 2-dim array D where the binary strings are listed on the top in lexicographic order, and the FSM's are listed and indexed on the left, then we can fill in the chart with 0's and 1's, where D(i,j) = 1 if and only if the machine indexed by i accepts the string j. Every row in this array represents a different Regular language, and all the Regular sets are listed (there are infinitely many). Are there any languages that do not appear in this list? Consider the language constructed by toggling the 0's and 1's on the diagonal. This language is precisely the self-hating FSM's. This language is not in the current list because it differs from the ith language in the list in the ith column. Hence there exists a language that is not acceptable by any FSM, hence not regular.

Another way to look at this is that there are more languages than there are FSM's to accept them. This is similar to the fact that there are more real numbers than there are integers.

Using Closure Properties to Show a Set is not Regular

One can also use closure properties to show that a set is not regular. For example, to show that the set of binary strings with an equal number of zeros and ones is not regular, we note that the intersection of that set with 0*1* is 0^n1^n. If the equal 0's and 1's set was regular then so would 0^n1^n, and we know that is not the case, hence equal zeros and ones is also not regular. There are many other examples of this kind of proof.

Context-free Languages

We now jump a level in power and we consider a new collection of sets from a grammar point of view. A context-free grammar is a grammar where every production has a single non-terminal symbol on the left side. Context free grammars can generate non-regular states. For example the 0^n1^n set is described by the language S ✍ 0S1 | ?. The design of context-free grammars is more of an art than a science but there are least two identifiable strategies. One is inductive, and the other semantic.

If a set is described inductively, then you can often leverage the inductive definition and turn it into a grammar. For example, consider the set of balanced parentheses. This can be defined inductively: ? is a balanced string. If x is a balanced string then so is (x). If and x and y are balanced strings then so is xy (concatenation not multiplication). This can be turned into a grammar by having each inductive rule be a production. We get S ✍ (S) | S ✍ SS | S ✍ ?.

If a set is described semantically, we can sometimes make a grammar by assigning semantic information to each non-terminal symbol. For example, consider the set of strings that have an equal number of zeros and ones, but not necessarily in any order. Let S generate these strings. Then we have S ✍ 0A and S ✍ 1B, where A generates strings that have one more 1 than 0, and B generates strings that have one more 0 than 1. We then continue A ✍ 1S and B ✍ 0S, and A ✍ 0AA and B ✍ 1BB. Finally, S ✍ ?, A ✍ 1 and B ✍ 0. All this is consistent with the semantic interpretation of A, B and S.

There is a third strategy that is used for grammars that are not necessarily context-free. The strategy is to use the grammar to simulate a machine-like computation. For example, consider the set of binary strings $0^n1^n0^n$.

In class we will consider the following context sensitive grammar and explain how it is essentially simulating a computation. D and C move back and forth like duck in a shooting gallery. L and R are bookends making sure that no symbols move off the edges. A, B and C represent what will eventually be the terminal symbols 0^n 1^n and 0^n respectively.

Here is the grammar:

S ✍ LDABCR	ADA ✍ AAD	BDB ✍ BBD	CDC ✍ CCD	
DR ✍ ER	LDA ✍ LAAD	ADB ✍ ABBD	BDC ✍ BCCD	
CE ✍ EC	BE ✍ EB	AE ✍ EA	LE ✍ LD	
LD ✍ ?	R ✍ ?	A ✍ 0	B ✍ 0	C✍ 0

Parse Trees and Ambiguity

A parse tree is a way to represent the derivation of a string from a grammar. We will show examples in class. A given string may have more than one parse tree. If every string in the language has exactly one parse tree then the grammar is called unambiguous. If there is even one string with two or more parse trees, then the grammar is called ambiguous.

What is considered a different parse tree? Two parse trees are distinct if they have a different structure, which means you cannot lay one on top of the other and have the symbols match up. An equivalent formulation of this is to say that there are two different leftmost (or rightmost) derivations of the string. Note it is not correct to say that two parse trees being distinct is the same as the string having two different variations. That is many different derivations can give the same parse tree, but every different leftmost derivation gives a unique parse tree.

In recitation Dimitri, will show you some cool uses of parse trees that apply to XML and compilers in general.

Chomsky Normal Form

It is convenient to assume that every context-free grammar can without loss of generality be put into a special format, called a normal form. One such format is Chomsky Normal Form. In CNF, we expect every production to be of the from A ✍ BC or D ✍ d, where A, B, C and D are non-terminal symbols and d is a non-lambda terminal symbols. If lambda (the empty string) is actually part of the language, then S ✍ ? is allowed.

We will use CNF in three different places:

1. A proof of a pumping lemma for CFG's.

2. A proof that every language generated by a CFG can be accepted by a non-deterministic pushdown machine.
3. An algorithm (dynamic programming style) for determining whether a given string is generated by a given context free grammar.

There are many steps needed to turn an arbitrary CFG into CNF. The steps are listed below:

1. Get rid of Useless symbols.
2. Get rid of lambda-productions.
3. Get rid of Unit Productions.
4. Get rid of Long Productions.
5. Get rid of Terminal symbols.

The steps are completely algorithmic with step one repeatable after each of the other steps if necessary. We will do a complete example in class.

Useless Symbols –

Delete all productions containing non-terminal symbols that cannot generate terminal strings.
Delete all productions containing non-terminal symbols that cannot be reached by S.

The details of the two steps for finding useless symbols are very similar and each is a bottom-up style algorithm. To find all non-terminal symbols that generate terminal strings, we do it inductively starting with all non-terminal symbols that generate a single terminal string in one step. Call this set T. Then we iterate again looking for productions whose right sides are combinations of terminal symbols and non-terminal from T. The non-terminals on the left sides of these productions are added to T, and we repeat. This continues until T remains the same through an iteration.

To find all non-terminal symbols that can be reached by S, we do a similar thing but we start from S and check which non-terminals appear on the right side of its productions. Call this set T. Then we check which non-terminals appear on the right side of productions whose left side is a non-terminal in T. This continues until T remains the same through an iteration.

The steps need to be done in this order. For example, if you do it in the opposite order, then the grammar S ✍ AB, S ✍ 0, A ✍ 0, would result in the grammar S ✍ 0, A ✍ 0. If we do it in the correct order, then we get the right answer, namely just S ✍ 0.

Subsequent steps may introduce new useless symbols. Hence Useless symbol removal can be done after each of the upcoming steps to ensure that we don't waste time carrying Useless symbols forward.

Lambda-Production Removal

The basic idea is to find all non-terminals that can eventually produce a lambda (nullable non-terminals), and then take every production in the grammar and substitute lambdas for each subset of such non-terminals. We add all these new productions. We can then delete the actual lambda productions. For example A ✍ 0N1N0 N ✍ ?. We add A ✍ 0N10 | 01N0 | 010, and delete N ✍ ?

The problem with this strategy is that it must be done for all nullable non-terminals simultaneously. If not, here is a problem scenario: S ✍ 0 | X1 | 0Y0 X ✍ Y | ? Y ✍ 1 | X. In this case, when we try to substitute for X ✍ ?, we add S ✍ 1 and Y ✍ ?, and then we sub for Y ✍ ?, we add S ✍ 00 and X ✍ ?. The trick is to calculate all nullable non-terminals at the start which include X and Y, and then sub for all simultaneously, deleting all resulting lambda productions, except perhaps for S ✍ ?. If lambda is actually in the language, then we add a special start symbol S' ✍ S | ?, where S remains the old start symbol.

Unit Productions

To get rid of Unit productions like A ✍ B, we simply add A✍ anything, for every production of the form B ✍ anything, and then delete A ✍ B. The only problem with this is that B might itself have Unit productions. Hence, like we did in lambda productions, we first calculate all Unit non-terminals that A can generate in one or more steps. Then the A ✍ *anything* productions are added for all the Unit non-terminals X in the list, where X ✍ anything, as long as *anything* is not a Unit production. The original Unit productions are then deleted. The Unit non-terminals that can be generated by A can be computed in a straightforward bottom-up manner similar to what we did earlier for lambda productions.

For example, consider S ✍ A | 11 A ✍ B | 1 B ✍ S | 0. The Unit non-terminals that can be generated by S, A and B are A,B and B,S and A,S respectively. So we add S ✍ 1 and S ✍ 0; A ✍ 11 and A ✍ 0; and B ✍ 1 and B ✍ 11. Then we delete S ✍ A, A ✍ B and B ✍ S.

Long Productions

Now that we have gotten rid of all length zero and length one productions, we need to concentrate on length > 2 productions. Let A ✍ ABC, then we simply replace this with A ✍ XC and X ✍ AB, where X is a new non-terminal symbol. We can do this same trick inductively (recursively) for longer productions.

If we have long terminal productions or mixed terminal/non-terminal productions, then for each terminal symbol, say our alphabet is {0,1}, we add productions M ✍ 0 and N ✍ 1. Then all 0's and 1's are replaced by M's and N's, where M and N are new non-terminal symbols.

Pushdown Machines

A pushdown machine is just like a finite state machine, except is also has a single stack that it is allowed to manipulate. (Note, that adding two stacks to a finite state machine makes a machine that is as powerful as a Turing machine. This can be proved later in the course). PDM's can be deterministic or nondeterministic. The difference here is that the nondeterministic PDM's can do more than the deterministic PDM's.

In class we will design PDM's for the following languages:

a. 0^n 1^n
b. 0^n 1^2n
c. The union of a and b.
d. ww^R (even length pallindromes).
e. The complement of d.
f. The complement of ww.

The collection of examples above should be enough to give you a broad set of tools for PDM design.

Closure Properties

Deterministic PDMs have different closure properties than non-deterministic PDM's. Deterministic machines are closed under complement, but not under union, intersection or reverse. Non-deterministic machines are closed under union and reverse but not under complement or intersection. They are however closed under intersection with regular sets. We will discuss all these results in detail in class.

Non-CFL's

Not every language is a CFL, and there are NPDM's which accept languages that no PDM can accept. An example of the former is 0^n1^n0^n, and and example of the latter id the union of 0^n 1^n and 0^n 1^2n.

.

CFG = NPDM
Every CFL Has a Non-deterministic PDM that Accepts It

One side of this equivalence is easier than the other. We omit the proof of theharder direction. The easier direction shows that every CFG G has a NPDM M where M accepts the language generated by G. The proof uses Chomsky Normal Form.

Without loss of generality, assume that G is given in CNF. We construct a NPDM M with three main states, an initial, a final and a *simulating* state. The initial state has one arrow coming out of it labeled lambda, lambda, SZ. All the rest of the arrows come out of the simulating state and all but one loops back to the simulating state. A detailed example will be shown in class using the grammar S ✍ AB A ✍ BB | 0 B ✍ AB | 1 | 0.

The idea of the proof is that the machine will simulate the grammar's attempt at generating a leftmost derivation string. The stack holds the current state of the derivation, namely the non-terminals that remain. At any point in the derivation, we must guess which production to use to substitute for the current leftmost symbol, which is sitting on top of the stack. If the production is a double non-terminal then that symbol is popped off the stack and the two new symbols pushed on in its place. If it is a single terminal, then a symbol on the tape is read, while we just pop the symbol off the stack.

CYK Algorithm

The most important decision algorithm about CFG's is given a string x and a CFG G, is x generated by G? This is equivalent to given a text file, and a Java compiler, is the file a syntactically correct Java program?

The details of such an algorithm is essentially one third of a compiler, called parsing.

One way to solve this problem is to turn the CFG into CNF and then generate all parse trees of height 2n-1 or less, where n is the length of x. (In a CNF grammar, all strings of length n are generated by a sequence of exactly 2n-1 productions). We check to see if any of these trees actually derive x. This method is an algorithm but runs in exponential time.

There are linear time algorithms which need special kinds of CFG's called LR(k) grammars. There is a cubic time algorithm by Cocke, Younger and Kasami which is described below:

(copy details from alg notes)

Other Decision Algorithms for CFGs

Almost anything else about CFG's that you want to know in undecidable. An exception is whether or not a CFG generates the empty set. This is decidable by checking whether or not the start symbol is useful, ala CNF. Undecidable questions include:

Does a given CFG generate everything?
Do two given CFG's generate the same language?
Is the intersection of two CFL's empty?
Is the complement of a CFL also a CFL?

Pumping Lemma for CFG's

General information about our text and readings:

Our text is Introduction to the Theory of Computation by Sipser. It is a classical introduction to automata theory, formal grammars and complexity theory known collectively as the theory of computation. Like many theoretical computer science texts, it begins with a review of the relevant discrete mathematics in Chapter 0. I will assign only 0.1 from this chapter that serves as an overview of the subject, but you are expected to be familiar with the other sections and use them for reference when necessary. We will use recitations for reviewing these topics if needed.

We will cover chapters 1-5 in detail, comprising the core of the course. Chapter 7 on NP-Complete theory has been covered in the Algorithms course (month 5), but we will cover it again from a slightly different point of view. The basics of chapter 8 (section 1-3) and selections from chapter 10 will be covered as time allows.

A detailed syllabus with readings follows below:

Week 1: Introduction and Regular Sets.

General Introduction: Languages, Grammars, Automata (Machines), The Chomsky Hierarchy, Applications. Finite State Machines, Regular Expressions, Regular Grammars, Non-determinism, Non-regular sets, The Pumping Lemma, Decision Algorithms for Regular Sets.

Reading: Chapters 0.1, 1, 4.1a.

Lecture 1: Finite State Machines, Design, Non-Determinism. (Reading: 1.1-1.2)

Lecture 2: Closure Properties, Regular Expressions, Equivalence with Finite State Machines. (Reading: 1.3).

Lecture 3: The Pumping Lemma – Proving a Language is Non-Regular, The Adversary Game, Using Closure Properties. (Reading 1.4).

Lecture 4: Regular Grammars, Equivalence to FSMs, Other FSM Variations, Minimization of FSM's.

Lecture 5: Decision Algorithms for Regular Sets and Undecidability. (Reading 4.1a).

Week 2: Context-Free Languages.

Context-Free Grammars, Chomsky Normal Form, Pushdown Machines, Non-Context-Free Languages, Another Pumping Lemma, Decision Algorithms for Context Free Sets.

Reading: Chapter 2, 4.1b.

Lecture 1: Context-Free Grammars, Semantic and Inductive Design, Examples, Applications. (Reading 2.1).

Lecture 2: Chomsky Normal Form. Three Applications. (Reading 2.1).

Lecture 3: Pushdown Machines, Non-Determinism Adds Power, Equivalence to CFG's. (Reading 2.2).

Lecture 4: The Pumping Lemma and Non-Context-Free Sets. (Reading 2.3).

Lecture 5: Closure Properties, DCFL's and Decision Algorithms. (Reading 4.1b).

Week 3: Turing Machines.

Reading: Chapter 3, 4.2, 5.1, 5.2.

Lecture 1: Turing Machines, Design. (Reading 3.1).

Lecture 2: Variations and Equivalence: Non-Determinism, MultiTape, Two-Way. (Reading 3.2).

Lecture 3: Recursive and Recursively Enumerable Sets, Undecidability. (Reading 4.2)

Lecture 4: Diagonalization, The Halting Problem. (Reading 4.2).

Lecture 5: Other Undecidable Problems, PCP. (Reading 5.1, 5.2).

Week 4: Complexity Theory. NP-Completeness theory. Reductions.

Reading: 5.3, 7, 8.1-8.3, 10 (If time allows).

Lecture 1: Reductions: Many-One (Mapping) Reducibility, Rice's Theorem. (Reading: 5.3).

Lecture 2: What is NP? What is NP-Complete? What are reductions? Examples. (Reading: 7).

Lecture 3: What is PSPACE? Savitch's Theorem. (Reading 8.1-8.2).

Lecture 4: PSPACE-Complete Problems. (Reading 8.3).

Lecture 5: Advanced Topics: Alternation, IP. (Reading 10.3-10.4).