Theory of computation: Grammars and Machines

As mentioned before, computation is elegantly modeled with simple mathematical objects:

Turing machines, finite automata, pushdown automata, and such.

We have discussed finite automata (DFA's, NFA's, NFA-Λ's). We will take a look at other "machines" that model computation.

Before we do that, we will examine methods of generating languages:

regular expressions, grammars.

We have already discussed regular expressions. Now we will examine several kinds of grammars.

§3.3 Grammars, §3.3.1 English grammar

We are familiar with natural language grammars. For example, a sentence with a transitive verb has the "creation" rule:

$$\text{<sentence>} \longrightarrow \text{<subject>}\ \text{<predicate>}$$

Then there are other rules that we apply before getting to the actual words:

$$\text{<subject>} \longrightarrow \text{<article>}\ \text{<adjective>}\ \text{<noun>}$$
$$\text{<predicate>} \longrightarrow \text{<verb>}\ \text{<object>}$$
$$\text{<object>} \longrightarrow \text{<article>}\ \text{<noun>}$$

In grammar theory, we call <sentence>, <subject>, <predicate>, <verb>, <object>, <article>, <adjective>, <noun>, etc., *variables* or *non-terminals*. The variable <sentence> is special — it is the "start variable", i.e. where we start constructing a sentence.

Then there are other rules that allow us to replaces variables by actual dictionary words, which we call *terminals*:

$$\text{<noun>} \longrightarrow \text{dog},\ \text{<noun>} \longrightarrow \text{cat},\ \text{<verb>} \longrightarrow \text{chased, etc.}$$

We sometimes diagram sentences using these substitution rules, and thus build a *parse tree*, with the start variable at the root. Figure 3.7 page 180 shows the parse tree for the sentence:

The big dog chased the cat.

§3.3.2 General grammars

Definition An *unrestricted grammar*, G, (also called a *phrase-structure grammar*) is a 4-tuple:

$$G = (N, \Sigma, P, S)$$

where:

- N is a finite set of "non-terminals" or "variables",
- Σ is a finite set of "terminals" where $N \cap \Sigma = \phi$,
- P is a finite set of "rules" or "productions",
- S is the "start variable" or "starting non-terminal".

The rules or productions are of the form $\alpha \longrightarrow \beta$, where α and β are strings over the alphabet $N \cup \Sigma$, with the following restrictions: $\alpha \neq \Lambda$, S appears alone on the left side of some rule, and each non-terminal appears in the left side of some rule.

For natural languages the non-terminals are the parts of grammar, and the terminals are the words in the dictionary of that language.

First example of a grammar

We describe a grammar, G_1 for the language

$$L = \{a^n b^n \mid n \geq 0\}$$

over $\Sigma = \{a, b\}$, the set of terminals.

Here $N = \{S\}$, where S is the start variable, and the rules are:

$$S \longrightarrow aSb$$

$$S \longrightarrow \Lambda$$

When we have several rules with the same left side, we can combine them with the vertical line $|$, which we read as "or":

$$S \longrightarrow aSb \mid \Lambda$$

§3.3.3 Derivations

Definition: A string in $(N \cup \Sigma)^*$ is called a *sentential form*.

Definition of Derivation If x and y are sentential forms and $\alpha \longrightarrow \beta$ is a rule or production, then the replacement of α by β in the sentential form $x\alpha y$ is called a *derivation step* and is denoted by:

$$x\alpha y \Longrightarrow x\beta y$$

A *derivation* is a sequence of derivation steps, with notation:

- $\Longrightarrow$ means to derive in one step
- $\Longrightarrow^+$ means to derive in one or more steps
- $\Longrightarrow^*$ means to derive in zero or more steps

Here is a derivation of *aabb* using the sample grammar given above:

$$S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aa\Lambda bb = aabb$$

Definition: A derivation is called *left-most* if the left-most non-terminal is replaced at each step.

The Language of a Grammar

The Language of a Grammar If G is a grammar with start variable S and terminals Σ, then $L(G)$, the language generated by G is the set:

$$L(G) = \{w \mid w \in \Sigma^* \text{and } S \Longrightarrow^+ w\}$$

For example, for the grammar G_1 above:

$$L(G_1) = \{a^n b^n \mid n \geq 0\}$$

Note that the rule $S \longrightarrow aSb$ is recursive.

In fact if the language defined by a grammar is infinite, that grammar must be recursive.

An Inductive Definition for $L(G)$

- *Basis*: If $S \Longrightarrow^{+} w$ without using a recursive derivation, put w in $L(G)$
- *Induction*: If $w \in L(G)$ with derivation $S \Longrightarrow^{+} w$ and contains a non-terminal A from a recursive production $A \longrightarrow \alpha A \beta$ or from an indirect recursive production corresponding to $A \Longrightarrow^{+} \alpha A \beta$, then modify the original derivation using the recursive production or the indirect recursive production to obtain a new derivation $S \Longrightarrow^{+} x$ and put x in $L(G)$.

§3.3.4 Constructing Grammars

1. Let $L = \{a^n \mid n \geq 0\}$ a grammar for L is given by the following productions:
$$S \longrightarrow aS \mid \Lambda$$
2. The palindrome language $PAL = \{w \in \{a, b\}^* \mid w = w^R\}$ is generated by:
$$S \longrightarrow aSa \mid bSb \mid a \mid b \mid \Lambda$$

Combining Grammars

Let L_1 and L_2 be language generated by grammars $G_1 = (N_1, \Sigma, P_1, S_1)$ and $G_2 = (N_2, \Sigma, P_2, S_2)$ respectively. Further assume that $N_1 \cap N_2 = \phi$ and that a new non-terminal S is not in N_1 or N_2. Then we can define grammars that generate $L_1 \cup L_2$, $L_1 L_2$, and L_1^* with start variable S as follows:

- **union** $G = (N, \Sigma, P, S)$ where $N = N_1 \cup N_2 \cup \{S\}$, and $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1 \mid S_2\}$.
- **concatenation** $G = (N, \Sigma, P, S)$ where $N = N_1 \cup N_2 \cup \{S\}$, and $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1 S_2\}$.
- **star** $G = (N, \Sigma, P, S)$ where $N = N_1 \cup \{S\}$, and $P = P_1 \cup \{S \longrightarrow S_1 S \mid \Lambda\}$.

§3.3.5 Meaning and Ambiguity

Usually, by using prececence, we understand that

$$3 - 4 - 2 = (3 - 4) - 2 = -3$$

and not

$$3 - 4 - 2 = 3 - (4 - 2) = 1$$

But the grammar with the following rules would allow either interpretation:

$$E \longrightarrow E - E \mid a \mid b$$

Definition: A grammar is *ambiguous* if its language contains some string with more than one parse tree, or equivalently, with more than one left-most (or right-most) derivation.

The grammar above is ambiguous since $a - b - a$ has two left-most derivations:
$E \Longrightarrow E - E \Longrightarrow a - E \Longrightarrow a - E - E \Longrightarrow a - b - E \Longrightarrow a - b - a$
$E \Longrightarrow E - E \Longrightarrow E - E - E \Longrightarrow a - E - E \Longrightarrow a - b - E \Longrightarrow a - b - a$

The first means $a - (b - a)$ and the second means $(a - b) - a$; this is also shown by their respective parse trees in Figure 3.10.

Curing Ambiguity

A grammar that is not ambiguous and gives the usual interpretation of $a - b - a$ is:

$$E \longrightarrow E - T \mid T$$

$$T \longrightarrow a \mid b$$

Here is a general unambiguous grammar for algebraic expressions:

$$E \longrightarrow E + T \mid E - T \mid T$$

$$T \longrightarrow T * F \mid T/F \mid F$$

$$F \longrightarrow a \mid b \mid (E)$$

Where E stands for "expression", T stands for "term", and F stands for "factor".

"Dangling else" ambiguity

Here is a natural grammar for the if-then-else construct:

$$S \longrightarrow I \mid L \mid A$$

$$I \longrightarrow \textit{if } C \textit{ then } S$$

$$L \longrightarrow \textit{if } C \textit{ then } S \textit{ else } S$$

$$A \longrightarrow a := 1$$

$$C \longrightarrow x > 0$$

where S stands for any statement (the start variable), I stands for the "if-then" statement, L stands for the "if-then-eLse" statement, A stands for any assignment statement, and C stands for any conditional expression.

However, the statement *if C then if C then A else A* has two meanings, i.e. two left-most derivations (or parse trees), and hence is ambiguous:
if C then (*if C then A else A*) — the "else" matches the second "if", or
if C then (*if C then A*) *else A*— the "else" matches the first "if".

"Dangling else" ambiguity — Cured

Here is an unambiguous grammar for the if-then-else construct:

$$S \longrightarrow M \mid U$$

$$M \longrightarrow \mathit{if}\ \ C\ \ \mathit{then}\ M\ \ \mathit{else}\ M \mid A$$

$$U \longrightarrow \mathit{if}\ \ C\ \ \mathit{then}\ M\ \ \mathit{else}\ U \mid \mathit{if}\ \ C\ \ \mathit{then}\ S$$

$$A \longrightarrow a := 1$$

$$C \longrightarrow x > 0$$

where S stands for any statement (the start variable), M stands for any statement in which all "if"s have Matched "else"s, U stands for any statement in which at least one "if"s is "Unmatched" — it doesn't have a matching "else" A stands for any assignment statement, and C stands for any conditional expression.

§12.1 Context-Free Languages

Definition of a Context-Free Grammar: A grammar is *context-free* if every production/rule is of the form:

$$A \longrightarrow \alpha$$

where A is a non-terminal and $\alpha \in (N \cup \Sigma)^*$.
A language L is said to be *context-free* if $L = L(G)$ for some context-free grammar G.

All of the grammars we have seen so far have been context-free.

If L_1 and L_2 are context-free, so are $L_1 \cup L_2$, $L_1 L_2$, and L_1^*, from the "Combining Grammars" construction above since the "combining" productions:

$$S \longrightarrow S_1 \mid S_2$$

$$S \longrightarrow S_1 S_2$$

$$S \longrightarrow S_1 S \mid \Lambda$$

are of the context-free form.

Regular languages are context free

Theorem: Regular languages are context free. We can assume the regular language L is recognized by a DFA M, which we use to specify a context free grammar G.

Idea: the variables of G will be the states of M i.e. $N = Q$ where we will assume the states are represented by capital letters, and the transitions of M will become rules of G as follows:

If $T(A, a) = B$ then

$$A \longrightarrow aB$$

and S is the both the start state of M and the start variable of G. Furthermore for each state $A \in F$, there is a rule:

$$A \longrightarrow \Lambda$$

With this definition, $L(G) = L(M)$.

§12.2 Pushdown Automata

A *PushDown Automaton*, PDA, is a 6-tuple $M = (S, \Sigma, \Gamma, T, s_0, F)$ that is like an NFA with an intially empty stack added to it. where:

- S is a finite set of "states",
- Σ is an alphabet — the "input alphabet",
- Γ is an alphabet — the "stack alphabet",
- $T : S \times \Sigma_\Lambda \times \Gamma_\Lambda \longrightarrow \text{power}(S \times \Sigma_\Lambda)$ is the "transition function",
- $s_0 \in S$ is the "initial state",
- $F \subset S$ is the set of "final" or "accepting" states.

where $\Sigma_\Lambda = \Sigma \cup \{\Lambda\}$ and $\Gamma_\Lambda = \Gamma \cup \{\Lambda\}$.

It works as follows: if $(q, c) \in T(p, a, b)$, M reads input symbol a, replaces the top symbol b on the stack with the symbol c, goes to state q, and moves to the next input symbol (if $a \neq \Lambda$). Special cases:

- $(q, c) \in T(p, a, \Lambda)$ pushes c onto the stack
- $(q, \Lambda) \in T(p, a, b)$ pops b from the stack
- $(q, b) \in T(p, a, b)$ is a "no-op" — it leaves the stack unchanged

Pushdown Automata — continued

Notes:

M accepts an input string w if it ends up in a state in F after all input has been consumed. Otherwise, it rejects w.

$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$

This is different than the definition in the text, in that the text's definition only allows the special cases of transitions mentioned above. However both definitions are equally powerful, and the corresponding machines recognize the same set of languages.

PDA's are non-deterministic.

Also, PDA's may accept a string by final state as above, or by "empty stack" (i.e. the stack is empty when input is consumed). However the languages they accept are the same.

Theorem: The set of languages accepted by PDA's are exactly the context-free languages.

Ordinary (non-deterministic) PDA's are more powerful than deterministic PDA's (DPDA's), which are often used as parsers for programming languages (because they are more efficient and no power of the programming language is lost).

§13.1 Turing Machines

Definition: A *Turing Machine*, TM, is a 7-tuple $M = (S, \Sigma, \Gamma, T, s_0, s_a, s_r)$ where:

- S is a finite set of "states",
- Σ is an alphabet — the "input alphabet", not containing the blank symbol,
- Γ is an alphabet — the "tape alphabet" which contains the blank symbol and Σ,
- $T : S \times \Gamma \longrightarrow S \times \Gamma \times \{L, R\}$ is the "transition function",
- $s_0 \in S$ is the "initial state",
- $s_a \in S$ is the "accept state",
- $s_r \in S$ is the "reject state", which is differnt than s_a

It works as follows: Initially the input string w is on the tape (the rest of which is blank) and the read/write head points to the left-most symbol in w. If $T(p, a) = (q, b, L)$ the machine replaces a by b and moves one tape square to the left (L). If $T(p, a) = (q, b, R)$ the machine replaces a by b and moves one tape square to the right (R). If the machine ever comes to the "accept state" or "reject state" it halts immediately.

Turing Machines — continued

Notes:

Again this is a different definition than the one in the text, but both kinds of machines can cmpute the same things. Differences:

- The text uses only one "Halt" state, rather than s_a and s_r.
- We assume that the tape is only infinite to the right, not both to the left and right as in the text.

We say *M accepts w* if it ends up in s_a, so

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

Such a language is called *rcursively enumerable* or *Turing-recognizable*.

§13.2 The Church-Turing Thesis

Any computation that can be done with a multi-tape or non-deterministic TM can be done with one using our definition of a TM. In fact any computation that can be done on any computer with any language can also be done on a Turing machine. That is, a TM can implement anything we consider to be an algorithm. This leads to:

Church-Turing Thesis:
Anything that is intuitively computable can be computed by a Turing machine.

Alan Turing invented Turing machines in 1936. In the same year Alonzo Church invented the λ-calculus, which had computing power equal to that of a Turing machine.

The reason the statement above is called a "Thesis" rather than a "Theorem" is that "intuitively computable" is not a precise mathematical concept.

§14.1 Computability

For some input we can ask for a "yes or no" answer. This is nicely modeled by our definition of a TM M: the answer is "yes" if M ends up in the s_a state and "no" if M ends up in the s_r state.

For example we can write a TM that decides if $w \in \{a^n b^n c^n \mid n \geq 0\}$.

Any TM can be described by a string of symbols $< M >$ that specifies its 7-tuple. We can use $< M >$ as input to another TM M_1 that can be programmed to decide properites of M. In fact we could even use $< M >$ as input to M itself. This is actually done when using a word-processor to write the code for a word-processor, or when using a programming language to write a compiler for that language (the compiler takes the compiler description as input to build the executable compiler).

We will use this concept to show that there is no TM that can decide whether another TM accepts string w or not.

An Undecidable Problem

Theorem There is no TM H that decides if TM M accepts string w or not. Proof by contradiction. Assume there is such an H, and let $< M, w >$ be the string encoding of M and w, so:

- $H(< M, w >) =$ "yes" if M accepts w
- $H(< M, w >) =$ "no" if M rejects w

Now we use H to construct another TM D as follows: D takes a TM M as input and runs H on $< M, < M >>$, i.e. on M with $w =$ the encoding of M as input. Then D outputs the opposite of what H says:

- if $H(< M, < M >>) =$ "yes", then D outputs "no"
- if $H(< M, < M >>) =$ "no", then D outputs "yes"

So:

- if $D(M) =$ "yes" M rejects $< M >$
- if $D(M) =$ "no" M accepts $< M >$

Now what does D do with its own encoding as input?

- if $D(D) =$ "yes" D rejects $< D >$, i.e. $D(D) =$ "no"
- if $D(D) =$ "no" D accepts $< D >$, i.e. $D(D) =$ "yes"

So such a D cannot exist, and therefore no such H can exist.

§14.1.2 The Halting Problem

Theorem There is no TM R that decides if TM M halts on string w or not. Proof by contradiction. Assume there is such an R, and let $< M, w >$ be the string encoding of M and w. We can use R to build another TM S that decides if M accepts w. S works as follows:

1. Run R on input $< M, w >$
2. If R rejects (M doesn't halt on input w), then S rejects — i.e. says "no"
3. If R accepts, then we know M will halt on input w, then we run M on input w
4. If M accepts w, S accepts (says "yes"); If M rejects w, S rejects (says "no");

Such an S cannot exist by the previous theorem, and therefore no such R can exist either.

§14.2 A Hierarchy of Languages

Regular Languages We have discussed these and shown that they can be generated by grammars with special rules of the form:

$$A \longrightarrow aB \text{ or } A \longrightarrow \Lambda.$$

They can also be recognized by DFA's, NFA's, and NFA-Λ's.

Context-Free Languages These can be generated by Context-Free Grammars, and can be recognized by Push-Down Automata (PDA's).

Context-Sensitive Languages These are generated by Context-Sensitive Grammars whose productions are of the form $xAy \longrightarrow x\alpha y$ where A is a non-terminal, and $x, y \in (N \cup \Sigma)^*$ and $\alpha \in (N \cup \Sigma)^+$, with $S \longrightarrow \Lambda$ allowed if Λ is in the language. The following language is not context-free.

Example of a context-sensitive grammar that generates $\{a^n b^n c^n \mid n \geq 0\}$:

1. $S \longrightarrow aSBC \mid aBC \mid \Lambda$
2. $CB \longrightarrow HB$
3. $HB \longrightarrow HC$
4. $HC \longrightarrow BC$
5. $aB \longrightarrow ab$
6. $bB \longrightarrow bb$
7. $bC \longrightarrow bc$
8. $cC \longrightarrow cc$

Hierarchy of Languages — Continued

Monotonic Grammars are those productions are of the form $u \longrightarrow v$ where $u, v \in (N \cup \Sigma)^+$, where $|u| \leq |v|$ and u contains at least one non-terminal. Again, the special case $S \longrightarrow \Lambda$ is allowed.

Theorem: Monotonic and context-sensitive grammars generate the same set of languages.

Theorem: The set of context-sensitive languages is the same as the set of languages recognized by **Linear Bounded Automata**, LBA's, which are non-deterministic Turing machines with markers L and R on the left and right ends of the input w (thus an LBA starts with LwR on its tape) which never go to the left of L or to the right of R, and they never replace L or R (p. 860).

Recursively Enumerable Languages, also called **Turing-Recognizable Languages** are the languages accepted (recognized) by a Turing machine.

Theorem: The set of recursively enumerable languages is the same as the set of languages generated by unrestricted (or phrase-structure) grammars.

The language $\{< M, w > |\ TM\ M\ \text{accepts}\ w\}$ is recursively enumerable but not context-sensitive. It is recognized by the **Universal Turing Machine** U that takes the string $< M, w >$ that describes M and w as input and uses M's description to "run" it on input w. U works like a development environment in which you create a program, and then run it, giving your program the appropriate input.

§14.2.2 Summary

In the late 1950's Noam Chomsky introduced this hierarchy of languages, which he called type-0, type-1, type-2, and type-3, and which correspond to recursively enumerable, context-sensitive, context-free, and regular languages respectively. The following table summarizes this hierarchy, indicating the correspondence between language gererators and language recognizers (machines).

Chomsky Type	Generator	Recognizer
0	Unrestricted Grammar	Turing Machine
1	Context-Sensitive Grammar	LBA
2	Context-Free Grammar	PDA
3	Regular Expression	DFA/NFA/NFA-Λ