CSCI 340: Computational Models

The Chomsky Hierarchy

Chapter 24

Department of Computer Science

Grammars

- We have yet to discover the "language structure" that define recursively enumerable sets independent of Turing Machines
- *Question:* Why are context-free languages called "context-free"?

Grammars

- We have yet to discover the "language structure" that define recursively enumerable sets independent of Turing Machines
- *Question:* Why are context-free languages called "context-free"? If there is a production $N \to t$, where N is any nonterminal and T is any terminal, then the replacement of t for N can be made in **any** situation
- English is *not* context-free

$$\text{Base} \to \text{cowardly}$$

$$\text{Ball} \to \text{dance}$$

$$\text{Baseball} \Rightarrow \text{cowardly dance}$$

- We make use of the **context** of words — their adjacent words
- *Insight:* Instead of replacing one string character by a string of characters (CFG), we must consider replacing an *entire string* of characters (including both terminals and nonterminals)

Phase-Structure Grammars

A **phrase-structure grammar** is a collection of three things:

1. A finite alphabet Σ of letters called **terminals**
2. A finite set of symbols called **nonterminals** that includes the **start symbol** S
3. A finite list of productions of the form:

   $$\text{String } 1 \to \text{string } 2$$

   Where string 1 can be any string of terminals and nonterminals that contains at least one nonterminal and string 2 is any string of terminals and nonterminals whatsoever.

A **derivation** in a phrase-structure grammar is a series of working strings beginning with S, which, by making substitutions according to the productions, arrives at a string of all terminals.

The **language generated** by a phase-structure grammar is the set of all strings of terminals that can be derived starting at S.

Example

$$S \to XS \mid \Lambda \qquad S \text{ is the language of zero or more } X\text{'s}$$

$$X \to aX \mid a \qquad X \text{ is the language of one or more } a\text{'s}$$

$$aaaX \to ba \qquad \text{anytime we see } aaaX\text{, we can replace it with } ba$$

$$\begin{aligned}
S &\Rightarrow XS && \textit{By } 1 \\
&\Rightarrow XXS && \textit{By } 1 \\
&\Rightarrow XX && \textit{By } 1 \\
&\Rightarrow aXX && \textit{By } 2 \\
&\Rightarrow aaXX && \textit{By } 2 \\
&\Rightarrow aaaXX && \textit{By } 2 \\
&\Rightarrow baXX && \textit{By } 3 \\
&\Rightarrow baaXX && \textit{By } 2 \\
&\Rightarrow baaaX && \textit{By } 2 \\
&\Rightarrow bba && \textit{By } 3
\end{aligned}$$

Phase-Structure Grammars > CFG

Theorem

At least one language that cannot be generated by a CFG can be generated by a phase-structure grammar

Proof.

Consider the following phase-structure grammar over $\Sigma = \{ \, a \;\; b \, \}$

$$\text{Prod 1} \quad S \longrightarrow aSBA$$

$$\text{Prod 2} \quad S \longrightarrow abA$$

$$\text{Prod 3} \quad AB \longrightarrow BA$$

$$\text{Prod 4} \quad bB \longrightarrow bb$$

$$\text{Prod 5} \quad bA \longrightarrow ba$$

$$\text{Prod 6} \quad aA \longrightarrow aa$$

□

Showing the grammar generates $a^n b^n a^n$

To generate the word $a^m b^m a^m$ for some fixed number m ...

Apply Prod 1 exactly $(m - 1)$ times:

$a\,a\,a\,.\,.\,.\,a \qquad S \qquad BA\,BA\;\;BA\;\;.\,.\,.\;\;BA$

$(m-1)$ a's followed by S followed by $(m-1)$ BA's

Then Prod 2 once:

$a\,a\,a\,a\,.\,.\,.\,a \qquad b \qquad A\,BA\,BA\,BA\;\;.\,.\,.\;\;BA$

m a's followed by b followed by m A's and $(m-1)$ B's

Apply Prod 3 enough times such that all B's come before all A's

$a\,a\,a\,a\,.\,.\,.\,a \qquad b \qquad BBB\;\;.\,.\,.\;\;AAA\;\;.\,.\,.\;\;A$

m a's followed by b followed by $(m-1)$ B's then m A's

Apply Prod 4 until it can't, Prod 5 until it can't, Prod 6 until it can't

$a\,a\,a\,a\,.\,.\,.\,a \qquad b\,b\,b\,b\,.\,.\,.\,b \qquad a\,a\,a\,a\,.\,.\,.\,a$

m a's followed by m b's followed by m a's

Showing the grammar **only** generates $a^n b^n a^n$

- Consider some derivation *aSBA* — which is of the form:
  "some *a*'s" S "equal number of *A*'s and *B*'s"
- If we never apply Prod 2 then the working string will contain an S and not generate any words
- As soon as Prod 2 is applied, we have a string of the form:
  "m *a*'s" abA "collection of m *A*'s and m *B*'s"
- Prod 3 merely scrambles this collection of *A*'s and *B*'s by shifting all *B*'s to come before all *A*'s.
- Productions 4, 5, and 6 are converting with rules of the form: $tN \rightarrow tt$ where t is a terminal and N is a nonterminal
- All productions from 4, 5, and 6 are done one-at-a-time from left-to-right. The resulting string is of the form:
  $a^{(m+1)} \quad b \quad b^m \quad a^{(m+1)}$

Phase-Structure Grammars

Theorem

If we have a phase-structure grammar that generates the language L, then there is another grammar that also generates L which has the same alphabet of terminals and in which each production is of the form:

$$\textit{string of nonterminals} \longrightarrow \textit{string of terminals and nonterminals}$$

Where the left side cannot be Λ *but the right side can*

Proof.

1. For each terminal, introduce a new nonterminal and change every occurrence of the "old" symbol to the "new" symbol. For example, $aSbXb \longrightarrow bbXYX$ becomes $ASBXB \longrightarrow BBXYX$
2. Add the new productions. From the example above, introduce $A \longrightarrow a$ and $B \longrightarrow b$

These new productions are now of the form $N^{+} \longrightarrow N^{*}$ or $N \longrightarrow t$ □

Example of Phase-Structure Modification

Consider the phase-structure grammar over $\Sigma = \{\ a\ \ b\ \}$:

$$
\begin{aligned}
S &\longrightarrow aSBA \mid abA \\
AB &\longrightarrow BA \\
bB &\longrightarrow bb \\
bA &\longrightarrow ba \\
aA &\longrightarrow aa
\end{aligned}
$$

Is transformed into:

$$
\begin{aligned}
S &\longrightarrow XSBA \mid XYA \\
AB &\longrightarrow BA \\
YB &\longrightarrow YY \\
YA &\longrightarrow YX \\
XA &\longrightarrow XX \\
X &\longrightarrow a \\
Y &\longrightarrow b
\end{aligned}
$$

Type 0 Grammars

Definition

A phase-structure grammar is called **type 0** if each production is:

non-empty string of nonterminals $\rightarrow$ any string of terminals and nonterminals

- We cannot allow the production "anything $\rightarrow$ anything"

  This would allow a terminal to yield some other string (even a nonterminal!) This goes against the philosophy of what a terminal is

- We do not want to allow any Λ on the left hand side

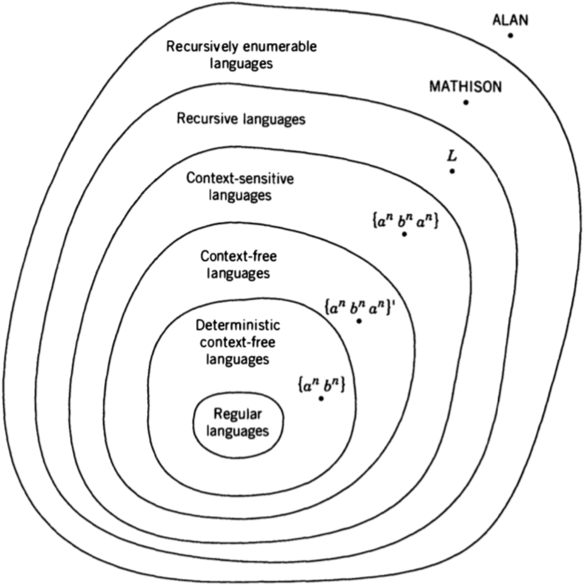
  This could arbitrarily have letters pop into words indiscriminately (see Genesis 1:3 for "$\Lambda \rightarrow$ light")

The Chomsky Hierarchy

The Chomsky Hierarchy of Grammars

Type	Name of Languages Generated	Production Restrictions $X \rightarrow Y$	Acceptor
0	Phrase-structure = recursively enumerable	X = any string with nonterminals Y = any string	TM
1	Context-sensitive	X = any string with nonterminals Y = any string as long as or longer than X	TMs with bounded (not infinite) TAPE, called linear-bounded automata LBAs*
2	Context-free	X = one nonterminal Y = any string	PDA
3	Regular	X = one nonterminal $Y = t\,N$ or $Y = t$, where t is terminal and N is nonterminal	FA

The Chomsky Hierarchy



Type 0 = TM

Theorem

If L is generated by a type 0 grammar G, then there is a TM that accepts L

Proof.

1. Insert \$ at the beginning and end of the input, followed by an S
   $abb\Delta$ becomes $\$abb\$S\Delta$
2. In the TM, enter a "grand central state" similar to the POP state for PDA simulations of CFGs
   The field of the TAPE beginning with the second \$ will keep track of the working string. We want to simulate (nondeterministically) the application of all productions

Type 0 = TM

Proof. (continued)

3. If we were lucky enough to apply *just* the right productions at just the right points in the working string, we branch to a subprogram that compares the working string to the input string.
    - If the input was derivable, the machine HALTs
    - If the number of words generated was finite and none match, the machine will CRASH
    - If the grammar generates an infinite number of words where the input is not derivable, the machine will LOOP forever
4. This NTM accepts any word in the language generated by G and only those words $\square$

Theorem

If a language is r.e., it can be generated by a type 0 grammar

Proof is omitted due to scope and length... (10 pages)

Product and Kleene Closure of r.e. Languages

Theorem

If L_1 and L_2 are recursively enumerable languages, then so is L_1L_2. The recursively enumerable languages are closed under product.

Proof.

1. Add the subscript $_1$ to all nonterminals *and terminals* of L_1
2. Add the subscript $_2$ to all nonterminals *and terminals* of L_2
3. Introduce a new production $S \longrightarrow S_1S_2$
4. Introduce new productions $t_1 \longrightarrow t$ for all terminals in L_1
5. Introduce new productions $t_2 \longrightarrow t$ for all terminals in L_2

All derivations will be unique and independent between S_1 and S_2. The newly introduced production of $S \longrightarrow S_1S_2$ ensures the concatenation □

Product and Kleene Closure of r.e. Languages

Theorem

If L is recursively enumerable, then L^* is also. The recursively enumerable languages are closed under Kleene star.

Proof.

1. We'd want to introduce something like $S \longrightarrow S_1 S \mid \Lambda$ but this won't work!
   Multiple S_1's could potentially interact!
   Replicate **all** productions of L and append $_2$ to all nonterminals.
2. Then append $_1$ to all nonterminals found in L.
3. Introduce the following new productions:
   $S \longrightarrow S_1 S_2 S \mid S_1 \mid \Lambda$

From S we can only produce: $\Lambda \quad S_1 \quad S_1 S_2 \quad S_1 S_2 S_1 \quad S_1 S_2 S_1 S_2 \quad \ldots$ $\square$

Context-Sensitive Grammars

Definition

A generative grammar in which the left side of each production is not longer than the right side is called a **context-sensitive grammar**, denoted CSG, or type 1.

- We presume all human languages are CSGs but cannot mathematically prove it.
- All context-sensitive grammars are *recursive.*

Theorem

For every context-sensitive grammar G, there is some special TM that accepts all the words generated by G and crashes for all other inputs

Context-Sensitive Grammars

Proof.

1. All rules make the working string longer
2. Since G is recursive, the shortest derivation has no “loops”
3. We can iteratively apply all valid productions on a working string and ensure unique working strings
4. Our TM will generate all words less than an upper length w in a procedure similar to how a TM accepted type 0 grammars
5. In a finite number of steps it will either find a derivation for a string, determine there is none, or crash

$\square$

CSG Decidability

Knowing that a language is *recursive* translates into being able to decide membership for it

Theorem

Given G, a context-sensitive grammar, and w, an input string, it is decidable by a TM whether G generates w

Proof.

- Create the CWL code word for the TM based on G described in the previous theorem
- Feed the encoded turing machine of G and w into the Universal Turing Machine
- Because w either halts or crashes on the coded TM, membership is decidable

□

The Language L

Theorem

There is at least one language L that is recursive but not context sensitive

Proof.

- There is some method that exists of encoding an entire CSG into a single string of symbols.
- A TM can decide whether, given an input string, it is the "code word" for some CSG
- Let us define the language L (we ran out of Turing's names):
  L = {all code words for CSG grammars that cannot be
  generated by the very grammars they encode}
- L must be recursive — it will never loop
- L is not context-sensitive — if it were then all its words would be generated by some CSG G. If the code word is in L then it couldn't be generated by the grammar it represents. □

Homework 12a

1. Consider the grammar:

$$\begin{array}{ll} \textsc{Prod 1} & S \longrightarrow ABS \mid \Lambda \\ \textsc{Prod 2} & AB \longrightarrow BA \\ \textsc{Prod 3} & BA \longrightarrow AB \\ \textsc{Prod 4} & A \longrightarrow a \\ \textsc{Prod 5} & B \longrightarrow b \end{array}$$

   - [4pts each] Derive the following words: *abba* , *babbaaab*
   - [4pts] Prove every word generated by this grammar has equal number of *a*'s and *b*'s (EQUAL)

2. [4pts] Find a grammar that generates all words with more *a*'s than *b*'s (MOREA)

3. [4pts] Find a grammar that generates all words **not** in EQUAL