Chomsky Grammars

CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2021

Grammars

To this point, we have studied languages that can be generated by regular expressions. These languages are call *regular languages*. We will now expand our study to include more expressive languages, which are generated by *grammars*. Grammars can generate all regular languages; regular expressions cannot generate all languages that grammars can.

Vocabulary: Definition 1, Section 13.1.2*

A *vocabulary* (or *alphabet* V is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence* over V is a string of finite length of elements of V. The *empty string* or *null string*, denoted by λ (and sometimes ϵ), is the string containing no symbols. The set of all words over V is denoted by V^*. A *language over* V is a subset of V^*.

Phrase-Structured Grammar: Definition 2, Section 13.1.2*

$G = (V, T, S, P)$
- V = Vocabulary (set that contains all terminal and nonterminal symbols)
- T = Terminals (set of all possible symbols in all strings generated by the grammar)
- S is in V (S is a symbol, not a set; it is the starting nonterminal)
- $P = \{A \to a, \dots\}$ (set of production rules: left-hand side *produces* right-hand side)

More useful definitions:
- N = nonterminals, V–T (set all symbols in the vocabulary that are *not* terminals)
- $V = N \cup T$ (the vocabulary is all terminals and nonterminals)

Language Generated: Definition 4, Section 13.1.2*

Let $G = (V, T, S, P)$ be a phrase-structured grammar. The *language generated by* G (or the *language of* G), denoted by $L(G)$, is the set of all strings of terminals that are derivable from the starting state S. In other word,

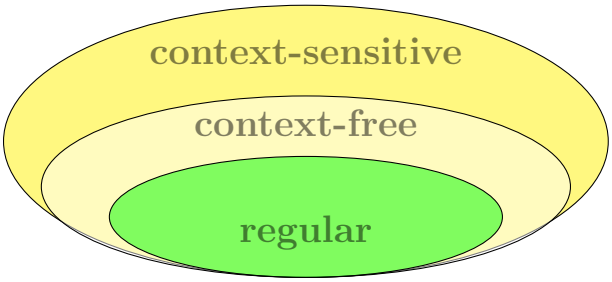
$$L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$$

Context-Free Grammars

Context-free grammars are the subset of all phrase-structured grammars that have the following form: all production rules only have a *single, nonterminal* on the left-hand side. Grammars that have more than a single, nonterminal on the left-hand side for any rule (e.g. $aAb \rightarrow \ldots$) are known as *context-sensitive* grammars. We will only study context-free grammars.

Chomsky Grammar Hierarchy

The hierarchy of phrase-structured grammars is shown below:



Thus context-sensitive grammars can generate all languages also generated by context-free and regular grammars, but the reverse is not true. We will focus our study on context-free (also known as Type 2) grammars and we have already studied regular (also known as Type 3) grammars through the lens of regular expressions. We will not study context-sensitive (Type 1) grammars.

Grammar Example:

Let's do an example to see how to create a grammar. First, let's start with a simple language $L = \{abc, def, ghi\}$. There are three possible strings in this language abc, def, and ghi. This language is a regular language, since we could create the following regular expression to generate L: $abc \cup def \cup ghi$. But recall from our hierarchy of grammars, a context-free grammar can generate any language generated by a regular expression. Remember that a grammar G is made up of several sets. Let's start with the set T, all the terminals. L contains the following unique symbols or vocabulary (not to be confused with the vocabulary of the grammar): $\{a, b, c, d, e, f, g, h, i\}$. All these symbols are the terminals of our grammar: all terminal strings (the output of our grammar) can only contain terminals. Thus $T = \{a, b, c, d, e, f, g, h, i\}$.

Using these terminals, let's create some production rules that will allow us to generate the three terminal strings abc, def, ghi. Note how the vocabulary of the language is the same as the set of terminals of the grammar. Similarly, the strings of a language are the terminal strings of the grammar (i.e the strings the grammar produces). These last two connections between grammars and the languages they generate are always true. Back to production rules. We'll pick a starting nonterminal – there's no real rhyme or reason here; let's call it S. We know there are three possible terminal strings, so we'll have a production rule from S to each possible string. This leads us to discuss nonterminals. nonterminals

are used to generate new nonterminals and terminals. In essence, nonterminals give us more expressiveness, which is what allows grammars to express more than regular languages. Here is the single production rule we'll create: $S \rightarrow abc \mid def \mid ghi$. Here is our final grammar:

$G_L = (V, T, S, P)$ where
$V = \{S, a, b, c, d, e, f, g, h, i\}$
$T = \{a, b, c, d, e, f, g, h, i\}$
$S = S$
$P = \{S \rightarrow abc \mid def \mid ghi\}$

Context-Free Language Recognizer

We've used finite-state machines to recognize regular languages, but finite-state machines cannot recognize context-free languages generated by grammars. We will use stack-based machines, known as *pushdown automata*, to recognize context-free languages. Pushdown automata are essentially finite-state automata that can make use of a stack (this means they can push and pop information to the stack at each transition). The stack is intentional – this is what allows pushdown automata to recognize context-free languages. We'll see an example of this below.

Pushdown Automata Example:

Let's consider the language $L = \{0^n 1^n | n \geq 0\}$ where all strings in the language have some number of 0's followed by the exact same number of 1's (including no zeros or ones). We don't have a concept of exponents in regular expressions (just the Kleene closure), so this language cannot be generated by a regular expression. Can we create a finite-state automaton that counts? No. But if we have a machine that contains a stack (i.e. a pushdown automaton) we can push every '0' we read onto the stack. Then, when we read a '1', we pop a '0' off the stack. The stack should be empty when we are done reading the input. If the stack has more '0's, then we reject the input. If the stack is empty when we read another '1', then we also reject the input. Recall that a grammar is defined as $G = (V, T, S, P)$. Let's consider the context-free grammar G that generates L:

$G_L = (V, T, S, P)$ where
$V = \{S, \lambda, 0, 1\}$
$T = \{\lambda, 0, 1\}$
$S = S$
$P = \{S \rightarrow 0S1, S \rightarrow \lambda\}$

This grammar, G_L, can generate any string with some number of '0's followed by the exact same number of '1's, where that number is greater than or equal to zero. Thus, the following strings are all legal members of $L : \{\lambda, 01, 0011, 000111, \ldots\}$. *Derivation* is the process of starting at the start symbol and then using the production rules until there are only terminals in the resulting string. Thus we call these strings of a language *terminal strings* because they are composed strictly of terminals (no nonterminals are allowed in the strings of our language). The terminal strings are also call words of the language. Let's derive the terminal string λ using the grammar.

- String before: S
- Production: $S \to \lambda$
- String after: λ

Thus our grammar produces λ, denoted by $S \Rightarrow \lambda$. Now, let's derive the terminal string 01:

- String before: S
- Production: $S \to 0S1$
- String after: $0S1$
- Production: $S \to \lambda$
- String after: 01

In this case we needed more than one production. We denote this by $S \stackrel{*}{\Rightarrow} 01$ (by definition, the symbol $\stackrel{*}{\Rightarrow}$ denotes *zero* or more productions). Work through the derivation for $S \stackrel{*}{\Rightarrow} 000111$. Notice that we repeatedly use the production rule $S \to 0S1$ for the number of zeros and ones we want. Finally, the last production is always $S \to \lambda$. Note that we only include λ in the terminal string when there are no other terminals present.

Grammar Example:

Consider the following grammar for a simple arithmetic language:
$G = (V, T, S, P)$ where
$N = \{E, D\}$
$T = \{0, 1, 2, \ldots, 9, +, -, *, /, (,)\}$
$S = E$
$P = \{E \to D|(E)|E + E|E - E|E * E|E/E$
$D \to 0|1|2|\ldots|9\}$

Recall that $V = N \cup T$ (all elements in both sets). Note that we can use the '|' to separate all the productions produced by the same nonterminal. Also, note that the start symbol can be any nonterminal. Let's derive the terminal string $4 + 3$:

E
$E + E$
$E + D$
$E + 3$
$D + 3$
$4 + 3$

$E \stackrel{*}{\Rightarrow} 4 + 3$.

Backus-Naur (or Normal) Form

Backus-Naur form (BNF) is a standardization for identifying terminals and nonterminals in a grammar. It was created to support development of programming languages. All nonterminals are identified by the angle brackets as in <id>, all terminals are not in angle brackets as in id. Production rules use ::= instead of →. Consider the following example:

```
<definition> ::= <type> <id> = <idOrint>;
<type> ::= int | byte
<idOrint> ::= <id> | <int>
<id> ::= a | b | c | ... | x | y | z
<int> ::= 0 | 1 | 2 | ... | 9
```

The following are all terminals: int, byte, '=', ';', all letters of the alphabet, and digits '0' to '9'. The nonterminals are <definition>, <type>, <idOrint>, <id>, and <int>. The starting nonterminal is usually the first one. If not, it is specified.

BNF Example:

Let's use the above grammar to derive the terminal string: int b = 3;

```
<definition>
<type> <id> = <idOrint>;
int <id> = <idOrint>;
int b = <idOrint>;
int b = 3;
```

Thus <definition> $\stackrel{*}{\Rightarrow}$ int b = 3;

See if you can derive these strings: int a = b; and byte z = 1; in the same way.

Conclusion

Grammars are a tool used in the creation of programming languages. See the book* for further examples and details.

*All definitions are from *Discrete Mathematics and Its Applications*, by Kenneth H. Rosen.