Grammars and Productions

CS 440: Programming Languages and Translators, Spring 2019
Lecture 9, Mon 2/18

Grammars

- Grammar uses production rules to show how to create utterances (= sentence) in the language.
- Grammar $G = (V, T, P, S)$
  - $V = \text{Set of nonterminal symbols (Noun, Verb, …, Expression, Statement, …)}$
  - $S = \text{Start symbol (nonterminal we begin with)}$
  - $T = \text{Set of terminal symbols} T (\text{alphabet for actual sentences})$
  - $P = \text{Production rules lhs} \rightarrow \text{rhs} (\text{any lhs can be replaced by rhs})$
    - lhs restricted by kind of grammar
    - rhs = string of terminal and/or nonterminal symbols, can be restricted by kind of grammar

Notation

- $a, b, \ldots \in T$ ($a$ and $b$ stand for members of $T$; $a$, $b$, … are members of $T$)
- $\Sigma$ for regular expressions.
- $x, y, \ldots \in T^*$
- $A, B, \ldots \in V$
- $W, X, \ldots \in T \mid V$
- $\alpha, \beta, \ldots \in (T \mid V)^*$ Sentential form

Kinds of grammars

- **Regular Grammar**: $A \rightarrow a B$, or $A \rightarrow B$ (plus $A \rightarrow a$, or $A \rightarrow \varepsilon$)
  - Correspond to NFA: States = $V$, rules are transitions $\rightarrow$ on $a$ or on $\varepsilon$; $A \rightarrow a$ or $\varepsilon$ for accepting state.
- **Context-Free Grammar (CFG)**: $A \rightarrow \alpha$; context-free because we can substitute for any $A$ regardless of where it is in a string $\in (T \mid V)^*$
  - What we use for programming languages; covers more languages than regular grammars, possible to write decent parsers for certain subset of CFG’s.
- **Context-Sensitive Grammar (CSG)**: Allows $\alpha A \beta \rightarrow \alpha B \beta$ (you can replace $A$ by $B$ in the context of $\alpha$ … $\beta$
  - More powerful than CFG’s.
    - Typechecking example: You can use $x$ as an $\text{int}$ variable if it’s been declared:
      $$\text{int } x; \text{ Var } \beta \rightarrow \text{int } x; \alpha x \beta ()$$
    - Hard to parse efficiently. We fake them by using *semantic analysis* after the initial parsing.
- **General Grammar**: No restriction on lhs; as powerful as Turing machines.
  - Set of languages recognized by TMs = Set of languages generated by general grammars
  - Completely impractical to use.
Productions, Derivations, Language of Grammar

- To use a grammar we start with \( S \), replace it by a rhs of \( S \to \text{rhs} \) rule, replace a nonterminal \( A \) in rhs by using \( A \to \text{lhs} \) its rhs, repeat until we have all terminal symbols.
- A production is a sequence \( S \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \) with each \( \alpha \in (T \cup V)^* \) a sentential form produced by \( S \). Production ends when \( \alpha_n \in T^* \). Each \( \to \) step is a lhs \( \to \) rhs replacement.
  - For CFG, each \( \alpha \) after \( S \) can have \( >1 \) nonterminal, since rhs can have \( >1 \) nonterminal,
  - Write \( S \to \star \alpha \) to indicate zero or more \( \to \) steps (reflexive transitive closure of \( \to \))
  - Example of reflexive transitive closure: \( \leq \) is reflexive transitive closure of relation \( m+1 = n \)
  - \( S \to \star \alpha \) can be pronounced "\( \alpha \) can be produced by \( S \)". "you can derive \( \alpha \) from \( S \)", "\( S \) produces \( \alpha \)", "\( S \) yields \( \alpha \)."
  - Note \( \alpha \to \star \beta \) is a production, \( \beta \) is a yield of \( \alpha \).

- Language generated by grammar \( L(G) = \{ x \mid S \to \star x \} \) the terminal strings produced by \( S \).
  - Arbitrary language \( L \) is a context-free if it's \( L(G) \) for some context-free \( G \).
  - Similar for regular language, etc.
  - Every regular language is also context-free because all regular grammar rules are context-free.

- **Note**: A grammar only has one language (a subset of \( T^* \)), but an arbitrary language can be the language of more than one grammar.

- **Example 1**: Language \( \{ a^n b^n \mid n \geq 0 \} \) uses rules \( S \to \epsilon \mid a \ S \ b \) (Classic example of language that is context-free but not regular)
  - Using or bar to abbreviate two rules \( S \to \epsilon \) and \( S \to a \ S \ b \)
  - Infer \( G = \) (nonterminals, terminals, rules, start symbol) = (\( \{ S \} \), \( \{ a, b \} \), 2 rules above, \( S \))
  - \( S \to \epsilon \) so \( \epsilon \in L(G) \)
  - \( S \to a \ S \ b \to a \epsilon \ b \), so \( a b \in L(G) \).
  - More generally, \( S \to a^n \ S \ b^n \to a^{n+1} \ S \ b^{n+1} \to a^{n+1} \epsilon \ b^{n+1} \)

- **Example 2**: Language \( \{ x y \mid y = \text{reverse of } x \} \), the palindrome language over \( T \).
  - \( S \to \epsilon \mid a \ S \ a \mid b \ S \ b \) (one rule for each member of \( T \)).
  - In general, if \( S \to x \ S \ y \to x \ y \) where \( y = x \) reversed and \( x \) and \( y \) are of length \( n \), then to add new characters between \( x \) and \( y \),
    - \( x \ S \ y \to x \ a \ S \ a \ y \to x \ a \ a \ y \) and \( x \ S \ y \to x \ b \ S \ b \ y \to x \ b \ b \ y \) are the only possible extensions of \( x y \) with two characters.

Leftmost and Rightmost Derivations, Parse Trees

- The two example grammars above (for \( a^n b^n \) and for palindromes) are pretty easy to analyze because each sequence in \( S \to \star \) \( w \) has just one nonterminal, so though there's a choice as to which to apply, the nonterminal to apply it to is unique.
- In general, the \( \alpha \) in \( S \to \star \alpha \) can have more than one nonterminal, in which case we could choose to substitute for one or the other, but if we want to do both, the order doesn't matter in the long run
  - With \( S \to \star \alpha A \beta B \gamma \) with \( A \to \alpha' \) and \( B \to \beta' \), we can have
• $S \rightarrow \alpha A \beta B \gamma \rightarrow \alpha \alpha' \beta B \gamma \rightarrow \alpha \alpha' \beta' \beta' \gamma$ or
• $S \rightarrow \alpha A \beta B \gamma \rightarrow \alpha A \beta' \beta \gamma \rightarrow \alpha \alpha' \beta' \gamma$
• But technically, the derivations are different because as sequences, they differ.

• In a **leftmost derivation**, we always choose the leftmost nonterminal to replace; in a **rightmost derivation**, we always choose the rightmost nonterminal to replace.

• $S \rightarrow \alpha A \beta B \gamma \rightarrow \alpha \alpha' \beta B \gamma \rightarrow \alpha \alpha' \beta' \beta' \gamma$ is uses leftmost derivation on $A$ and $B$
• $S \rightarrow \alpha A \beta B \gamma \rightarrow \alpha A \beta' \beta \gamma \rightarrow \alpha \alpha' \beta' \gamma$ is uses rightmost derivation on $A$ and $B$
• (We’re assuming that $\alpha, \alpha', \beta$ and $\gamma$ are all strings of terminals. (I.e., $\in T^*$).

• If we want, we can always use leftmost (or rightmost) derivations if we want to distinguish between derivations that apply fundamentally different rules from derivations that simply reorder the same rules.

• **Example 3:**
  • $S \rightarrow S T \rightarrow s T \rightarrow s t$ versus $S \rightarrow S t \rightarrow s t$ just reordered the rule applications
  • $S \rightarrow S T \rightarrow s T \rightarrow s t$ versus $S \rightarrow T S \rightarrow t S \rightarrow t s$ use different rules
  • (Quick question: What’s the implicit grammar here?)

• Note if two leftmost or rightmost derivations end in different terminal strings, then they definitely used different rules, but the converse doesn’t hold if we use a non-leftmost or non-rightmost derivation.

• **Definition:** A **parse tree** is a directed graph that represents a derivation by using $S$ as the root and every rule application $A \rightarrow X_1 X_2 \ldots X_n$ (each $X_i \in V | T$) turns into node $A$ having $X_1 X_2 \ldots X_n$ as its $n$ children.

• For Example 3, the first two derivations (reordering the rule application) have the same parse tree (the one on the left, below). The second two derivations have different parse trees (left vs. right below).

```
  S                 S
  / \               / \ 
 S   T          T   S
 |   |          |   |
 s   t          t   s
```

• The leftmost and rightmost derivations (and derivations generally) can differ as sequences, but their parse trees will be the same. Leftmost derivations are standard preorder traversals (top-down, children visited left-to-right), rightmost derivations visit children right-to-left.

• Comparing parse trees is a nice way to get rid of the rule-reordering differences so that we can concentrate on different rule applications.

**Ambiguous Grammars**

• There can be a problem where a grammar has two fundamentally different ways to generate the same string. (I.e., two parse trees for it.)

• **Example 4 (a standard example):**
  • $E \rightarrow \text{id} | E + E | E * E$. Here, $T = \{+, *, \text{id}\}$ (Once we get to programming languages, a string like $\text{id}$ is a terminal symbol instead of two concatenated characters.)
  • The terminal string $\text{id} + \text{id} * \text{id}$ has two parse trees (we probably want the one on the left).
• A grammar is **ambiguous** if there is a string that it generates with more than one different parse tree.
  • (In English, this is one kind of *pun*; the other kind is where we have one parse trees but the "terminal" symbols can stand for different things.)

• Ambiguous grammars cause problems because we don't have a unique answer to the question "Why is this terminal string in the language?"

• Ambiguous grammars can be nice for describing this but bad for parsing.
  • When we say an expression is an identifier or the + or * of subexpressions, translating to the rules

  \[ E \to \text{id} | E+E | E*E \]

  seems natural.

• There are basically two ways to **disambiguate** a grammar⁠¹:

  • **Technique 1**: *Keep the rules but break ties somehow*. This is where operator precedences and associativites get used.
    • This is the technique used by **parser-generators** (programs that take a grammar and output a parser program): If it’s a choice between two rules, pick the one that is higher in the list of rules. (Essentially, take the first rule that applies.)
    • Doesn't necessarily solve all problems.
      • With \( E \to E+E | E*E \), we don't know which rule we want until we get past the first \( E \) to the + or *.
      • If we just choose the first one in the list, it might be the wrong one.
    • Hmm. Use backtracking search? Nondeterministic search?

  • **Technique 2**: *Change the rules*. Find a non-ambiguous grammar that generates the same language.
    • Trying to avoid backtracking search.
    • This technique gets used too, but it can introduce many extra nonterminals and change the rules so that they're more obscure.

  • **Example 5**: \( (T = \text{term}, F = \text{factor}) \)
    • Grammar \( E \to T | T+E, T \to F | F*T, F \to \text{id} \)
      • In real life we'd include \( F \to (E) \)
    • Now \( \text{id} + \text{id} * \text{id} \) has just one parse tree, corresponding to the derivation

      \[
      E \to T + E \\
      \to F + E \\
      \to \text{id} + E
      \]

---

¹ It's "**ambiguous grammars**" we're interested in, so stop me if I accidentally say "**ambiguous language**". There are languages that are only generated by ambiguous grammars, but the languages we're interested in aren't that way.
→ id + T
→ id + F * T
→ id + id * T
→ id + id * F
→ id + id * id

• Activity questions: What is the parse tree? Was this a leftmost or rightmost derivation? If so, what is the opposite derivation? (Note in general, there can exist derivations that are neither leftmost nor rightmost.)