(Scanners), Grammars and Productions, v.2

CS 440: Programming Languages and Translators, Spring 2020

2/11 v.2

A. (Scanners — left over from the Regular Expression/FSM topic) [Added 2/11]

- A **lexical scanner** (= **lexer**) takes the input for a compiler or interpreter and breaks it up into convenient-sized **tokens**. E.g., identifiers, constants, operators, etc. Token structure is described using regular expressions, and a scanner basically just repeats running a regular expression search, capturing the text of each token, and returning the list of tokens. We use a scanner because it simplifies the work of the parser, which takes the list of tokens as its input.
- We can write a scanner by using our regular expression capturing routine to look for patterns of the form (`identifier` | `constant` | ... | etc). With each successful match, we take the captured string and add it to our list of tokens. (The simplest kind of token is just the string we matched; more complex tokens can be described by a data structure for scanner tokens, e.g. `data SToken = Id String | Const String` | etc; we write a `tokenize` routine that takes a string and returns the appropriate kind of token.
- There is one complication: Strings that don't become part of the program (whitespace or comments). Those have to be recognized somewhere and not added to the list of `SToken` output. We can try to skip them while matching regular expressions or when building the list of `STokens` or even later during parsing.

B. Grammars; Productions

- Regular expressions are a textual way of describing certain kinds of patterns of strings. They are handy and useful but don't describe everything we need in a programming language description. Instead we use a more general mechanism, grammars.
- A **grammar** describes a textual mechanism for finding and generating sentences of a language.
- Grammar $G = (V, T, P, S)$
  - V = Set of **nonterminal symbols** (Noun, Verb, …, Expression, Statement, …)
  - S = **Start symbol** (nonterminal we begin with)
  - T = Set of **terminal symbols** T (alphabet for actual sentences)
    - For regular expressions. we used Σ instead of T
    - "Terminal" as a noun means "terminal symbol"
    - A **terminal string** is a string from T^*.
  - P = Set of **production rules**
    - General form is lhs → rhs. We'll only use grammars where the lhs is a nonterminal; the rhs can be any **sentential form** (a mix of terminal &/or nonterminal symbols).
    - We say a rule is "for" the lhs symbol.
    - **Example 1**: A couple of production rules for S: $S \to$ `a` S `b` and $S \to \varepsilon$.
- Using a grammar, we can chain together uses of production rules to form a **production** that shows how to **derive** a sentence (= string of terminal symbols) in the language.

- We begin with the start symbol; the first step has to replace the start symbol with the rhs for one of its rules. We repeat step by step, replacing a nonterminal symbol by the rhs of one of its rules. When we have nothing but terminal symbols, we're done.
- **Example 2**: With rules $S \to$ `a` S `b` and $S \to \varepsilon$, we can take an S and apply the first rule a number of times and eventually apply the second rule to stop with a string of terminal symbols.

  $S \to \texttt{a}\,S\,\texttt{b} \to \texttt{a a}\,S\,\texttt{b b} \to \texttt{a a a}\,S\,\texttt{b b b} \to \texttt{a a a}\,\varepsilon\,\texttt{b b b} = \texttt{aaabbb}$
- In Example 2, there's only ever one nonterminal in the sentential form, so there's no choice of which nonterminal to replace (but there's a choice as to which rule to use).
- If we had a rule $S \to \texttt{a}\,S\,\texttt{b}\,S$, then we could have different sequences of forms based on what order we replace the occurrences of S. E.g.,

  $S \to \texttt{a}\,S\,\texttt{b}\,S \to \texttt{a a}\,S\,\texttt{b}\,S\,\texttt{b}\,S$ etc.

  $S \to \texttt{a}\,S\,\texttt{b}\,S \to \texttt{a}\,S\,\texttt{b a}\,S\,\texttt{b}\,S$ etc.

Notation

• $T = \{\texttt{a}, \texttt{b}, \ldots\}$	(`a`, `b`,… **are** members of T)
• $a, b, \ldots \in T$	(a, b, … **stand for** members of T)
• $x, y, \ldots \in T^*$	(A **word** is a list of terminals)
• $A, B, \ldots \in V$	(Individual nonterminals)
• $W, X, \ldots \in T \mid V$	(Individual terminal or nonterminal symbols)
• $\alpha, \beta, \ldots \in (T \mid V)^*$	(A **sentential form** is a sequence of terminal and/or nonterminal symbols)

C. Kinds of grammars

- **Regular Grammar**: $A \to a\,B$, or $A \to B$ (plus $A \to a$, or $A \to \varepsilon$)
  - Correspond to NFA: States = V, rules are transitions → on a or on ε; $A \to a$ or ε for accepting state.
- **Context-Free Grammar (CFG)**: $A \to \alpha$; context-free because we can substitute for any A regardless of where it is in 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 $\alpha\,A\,\beta \to \alpha\,B\,\beta$ (you can replace A by B in the context of α … β
  - More powerful than CFG's.
  - Typechecking example: You can use x as an `int` variable if it's been declared:

    `int` x`;` $\alpha\ \mathit{Var}\ \beta \to$ `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.

D. Productions, the Language of a Grammar

- Let's go back to the notion of "production" and make it more formal.
- **Definition**: A **production** is a sequence $S \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n$ with each $\alpha \in (T \mid V)^*$ a sentential form **produced by** S. Production ends when $\alpha_n \in T^*$. Each $\to$ step is a lhs $\to$ rhs replacement.
  - For a CFG, all the rules are of the form $A \to \alpha$ (a single nonterminal can be replaced by a sequence of terminal and/or nonterminal symbols). The rhs α can have zero, one, or more nonterminals. In a production, all the forms except for the last includes at least one nonterminal.
- Often we want to concentrate on the first and last steps of a production.
- **Definition** (of $\to^*$) We write $S \to^* \gamma$ to abbreviate a production like $S \to \alpha \to \beta \to \ldots \to \gamma$. (Note γ doesn't have to be in T^*; i.e., it doesn't have to be a terminal string.)
  - More generally we can have $\alpha \to^* \beta$ (so α could come from the middle of a sequence headed by S).
  - We say α **produces** or **yields** β or that β can be **produced by** or **derived from** α.
  - **Note**: $\to^*$ is the "reflexive transitive closure" of $\to$ (it's what you get if you have a sequence of zero or more $\to$ steps.[1]
- **Definition**: $L(G)$ is the **language generated by grammar *G*.** $L(G) = \{x \in T^* \mid S \to^* x\}$ 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. E.g., $S \to \varepsilon \mid \texttt{a}\, S$ generates same language as the reg expr $\texttt{a}^*$; so does $S \to \varepsilon \mid S\, \texttt{a}$.
- **Example 3**: Language $\{\texttt{a}^n\, \texttt{b}^n \mid n \geq 0\}$ uses rules $S \to \varepsilon \mid \texttt{a}\, S\, \texttt{b}$
  - This is a classic example of language that is context-free but not regular.
  - Using or bar to abbreviate two rules $S \to \varepsilon$ and $S \to \texttt{a}\, S\, \texttt{b}$
  - Infer G = (nonterminals, terminals, rules, start symbol) = ($\{S\}$, $\{\texttt{a}, \texttt{b}\}$, 2 rules above, S)
  - $S \to \varepsilon$ so $\varepsilon \in L(G)$
  - $S \to \texttt{a}\, S\, \texttt{b} \to \texttt{a}\, \varepsilon\, \texttt{b}$, so $\texttt{ab} \in L(G)$.
  - More generally, $S \to^n \texttt{a}^n\, S\, \texttt{b}^n \to \texttt{a}^{n+1}\, S\, \texttt{b}^{n+1} \to \texttt{a}^{n+1}\, \varepsilon\, \texttt{b}^{n+1}$ [2/11]
- **Example 4**: Language $\{x\, y \mid y = \text{reverse of } x\}$, the language over T of palindromes.
  - $S \to \varepsilon \mid \texttt{a}\, S\, \texttt{a} \mid \texttt{b}\, S\, \texttt{b}$ (one rule for each member of T).
  - To get odd-length palindromes, we need rules $S \to \texttt{a}$ and $S \to \texttt{b}$.

[1] Another example: If we have the relation $n \mapsto n+1$, then $n \leq m$ is the reflexive transitive closure of $\mapsto$.

- In general, if $S \to^* x\ S\ y \to x\ y$ where $y = x$ reversed and x and y are of length $\texttt{n}$, then to add new characters between x and y,
- $x\ S\ y \to x\ \texttt{a}\ S\ \texttt{a}\ y \to x\ \texttt{a}\ \texttt{a}\ y$ and $x\ S\ y \to x\ \texttt{b}\ S\ \texttt{b}\ y \to x\ \texttt{b}\ \varepsilon\ \texttt{b}\ y$ are the only possible extensions of $x\ y$ with two characters.

E. Leftmost and Rightmost Derivations

- The grammars of Examples 3 and 4 above (for $\texttt{a}^n\ \texttt{b}^n$ and for palindromes) are pretty easy to analyze because each sequence in $S \to^* w$ has just one nonterminal, so there's no choice involved in which nonterminal to expand. (There is a choice of how to expand it, however.)
- In general, the α in $S \to^* \alpha$ 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.

[2/11] Start some rewriting:

- **Notation**: To reduce the number of symbols inside a sentential form, ellipses (…) can stand for an unnamed sentential form. E.g., if we're not interested in discussing α or β, we can rewrite $\alpha\ X\ \beta$ as $\ldots X \ldots$.
  - With $S \to^* \ldots A \ldots B \ldots$ with $A \to \alpha'$ and $B \to \beta'$, we can have
    $S \to \ldots A \ldots B \ldots \to \ldots \alpha' \ldots B \ldots \to \ldots \alpha' \ldots \beta' \ldots$ or
    $S \to \ldots A \ldots B \ldots \to \ldots A \ldots \beta' \ldots \to \ldots \alpha' \ldots \beta' \ldots$
  - But technically, the derivations are different because as sequences, they differ.
- In a **leftmost derivation**, we always choose the leftmost nonterminal to replace.
  - $S \to w\ A\ \gamma \to w\ \alpha\ \gamma$ uses $A \to \alpha$. Since left of the A we have only terminal symbols, this is a leftmost derivation step. (Recall the notation: $w \in T^*$.)
  - The next leftmost step would be the leftmost nonterminal in $\alpha\ \gamma$ (the leftmost nonterminal in α if there is one, else the leftmost nonterminal in γ if α is a terminal string). If $\alpha\ \gamma$ has has no nonterminal then it's a terminal string and there is not next derivation step.
- Symmetrically, in a **rightmost derivation**, we always choose the rightmost nonterminal to replace.
  - $S \to \beta\ A\ w \to \beta\ \alpha\ w$ again uses $A \to w$ but this time the A is the rightmost nonterminal in $\beta\ A\ w$.
  - The next rightmost step (if there is one) would expand the rightmost nonterminal in $\beta\ \alpha$ (if there is one).
- If we have two derivations that use the same production rules but in different orders, then they will produce the same terminal string. E.g., $S \to^* w$ whether we use a leftmost or rightmost derivation, so long as all we do is reorder the productions. In this case, the leftmost and rightmost derivations are handy standard ways to order the rule applications.
- **Example 5**: With rules $S \to \texttt{s} \mid S\ T \mid T\ S$ and $T \to \texttt{t}$, the two derivations below produce the same yield but in leftmost or rightmost order.
  - $S \to S\ T \to \texttt{s}\ T \to \texttt{s}\ \texttt{t}$ (leftmost)
  - $S \to S\ T \to S\ \texttt{t} \to \texttt{s}\ \texttt{t}$ (rightmost)

  On the other hand, the following two derivations are both leftmost but produce different yields because they use different rules.

    - $S \to S\,T \to \texttt{s}\;T \to \texttt{s t}$ (leftmost using $S \to S\,T$)
    - $S \to T\,S \to \texttt{t}\;S \to \texttt{t s}$ (leftmost using $S \to T\,S$)
- 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.

F. Parse Trees

- A parse trees is a way to summarize a set of derivations; each different derivation corresponds to a different traversal of nodes in the parse tree.
- **Definition**: A **parse tree** is a directed graph that represents a derivation. The root node is S (the start symbol), and each node has as its children the rhs of a rule application. E.g., for a node A, applying $A \to X_1\, X_2\, \dots\, X_n$ (each $X_i \in V \mid T$) is done by making $X_1\, X_2\, \dots\, X_n$ the children of the A node.
- For Example 5, the first two derivations (reordering the rule application) have the same parse tree (the one on the left, below) because they both used $S \to S\,T$ on the start symbol. The parse tree to the right is for the derivation that used $S \to T\,S$.

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

- Traversing the parse tree top-down, children left-to-right produces the leftmost derivations; traversing top-down with children right-to-left produces the rightmost derivation. The derivations will be sequences of different sentential forms (unless each node has only one nonterminal child), but the derivations have the same yield (i.e., the frontier of the tree).
- Using parse trees gives a nice way to see if derivations use different rule applications: If the parse trees differ as trees, the rule applications were different; if the parse trees are the same, then the difference was only in what order was used.
- Without parse trees, if we take two derivations and reorder each one to be leftmost (or both rightmost), then we can compare the derivations as sequences to see if there's a difference.

-------------------- 2020-02-11

Ambiguous Grammars

- It's not surprising that two different parse trees can have different yields; having two different parse trees for the same yield introduces possible confusion.
- **Example 6 (a standard example):**
    - $E \to \texttt{x} \mid E + E \mid E * E$. Here, $T = \{+, *, \texttt{x}\}$ (Here, $\texttt{x}$ stands for a general identifier.)

- The terminal string `x + x * x` has two parse trees (we probably want the one on the left).

```
   E              E
  /|\            /|\
 E + E          E * E
 |  /|\        /|\  |
 x E * E      E + E x
   |   |      |   |
   x   x      x   x
```

- 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 $E \to \mathtt{x} \mid E + E \mid E * E$ seems natural.
- There are basically two ways to **disambiguate** a grammar[2]:
- *Technique 1*: *Keep the rules but break ties somehow*. This is where operator precedences and associativities get used.
  - This is a 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.)
  - This doesn't necessarily solve all problems.
    - With $E \to E + E \mid E * E$, 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.
  - This technique gets used too, but it can introduce many extra nonterminals and change the rules so that they're more obscure.
  - Basically, we hard-code the precedences and associativities into the grammar. Since * has higher precedence than +, all other things equal, we want the * to appear lower in the tree.
- **Example 7**: (T = term, F = factor) Grammar (E is the start symbol)
    - $E \to T \mid T + E$
    - $T \to F \mid F * T$
    - $F \to \mathtt{x} \mid (E)$

[2] It's "*ambiguous grammars*" we're interested in, so stop me if I accidentally say "*ambiguous language*". A language is ambiguous if it's generated only by ambiguous grammars. (The languages we're interested in aren't that way.)

Now x + x * x has just one parse tree, corresponding to the derivation

$$E \rightarrow T + E \;\rightarrow\; F + E \;\rightarrow\; \mathtt{x} + E \;\rightarrow \mathtt{x} + T \;\rightarrow \mathtt{x} + F * T \rightarrow \mathtt{x} + \mathtt{x} * T \rightarrow \mathtt{x} + \mathtt{x} * F \rightarrow \mathtt{x} + \mathtt{x} * \mathtt{x}$$

The price to be paid for this is that the parse tree is much more complicated now:

```
      E
    / | \
   T  +  E
   |     |
   F     T
   |    /|\
   x   F * T
       |   |
       x   F
           |
           x
```

[2/11] End the rewriting

Activity Problems for Lecture 9

Scanners

1. (Scanners) [added 2/11] What does a lexical scanner do, why do we use one, and how does it do its work?

Grammars and Productions

1. (Terminology and definitions)
   a. What are the meanings of the following terms : grammar, terminal symbol, nonterminal symbol, production rule, word, sentential form?
   b. How do regular, context-free, context-sensitive, and general grammars differ?
   c. Can we write programs to deal with regular languages? (Generate or recognize strings in the language, for example.) How about context-free, context-sensitive, and general grammars?
   d. Which kind of grammar do we typically use for describing programming languages? Why? What features of parsing / compiling aren't expressed by context-free languages?
   e. What is a production; how do they involve sentential forms? What is the notation for a production? What does it mean to say something can be produced or derived or is yielded by from a nonterminal or sentential form?
   d. What is $L(G)$, the language of a grammar G? Can you have multiple languages for a grammar? Multiple grammars for a language?
   e. What are derivations? How do leftmost and rightmost derivations differ? What is a parse tree and how can it generalize a set of derivations?
   f. Do strings generated by a language always have only one parse tree? What is an ambiguous grammar? What are the advantages of ambiguous and non-ambiguous grammars?

2. Give the grammar rules for the derivations in Example 5.

3. Give leftmost and rightmost derivations for the trees in Example 6.

4. Give a CFG for the language $\{\mathtt{a}^n\,\mathtt{b}^m \mid n, \le m\}$, where each string has at least as many *b*'s as *a*'s. (Hint: take the grammar for $\mathtt{a}^n\,\mathtt{b}^n$ and add more steps that generate `b`'s.

5. Give a CFG for the language $\{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid w$ has an equal number of `a`'s and `b`'s$\}$. Note that unlike the language for $\mathtt{a}^n\,\mathtt{b}^n$, here the `a`'s and `b`'s can come in any order. For example, `aabb`, `abab`, `abba`, `baab`, `baba`, and `bbaa` are all in this language.

6. Was the derivation in Example 7 a leftmost or rightmost derivation? If it was, what does the opposite kind of derivation look like? What is the parse tree for this derivation? How can leftmost and rightmost derivations be recovered using a parse tree?

Solutions to Selected Activity Problems for Lecture 9

Scanners

1. (Omitted)

Grammars and Parsers

1. (Omitted)

2. $S \to S\,T,\ \ S \to T\,S,\ \ S \to \texttt{s},\ \ T \to \texttt{t}$

3. $E \to E + E \to \texttt{id} + E \to \texttt{id} + E * E \to \texttt{id} + \texttt{id} * E \to \texttt{id} + \texttt{id} * \texttt{id}$ (leftmost)

   $E \to E + E \to E + E * E \to E + E * \texttt{id} \to E + \texttt{id} * \texttt{id} \to \texttt{id} + \texttt{id} * \texttt{id}$ (rightmost)

   $E \to E * E \to E + E * E \to \texttt{id} + E * E \to \texttt{id} + \texttt{id} * E \to \texttt{id} + \texttt{id} * \texttt{id}$ (leftmost)

   $E \to E * E \to E * \texttt{id} \to E + E * \texttt{id} \to E + \texttt{id} * \texttt{id} \to \texttt{id} + \texttt{id} * \texttt{id}$ (rightmost)

4. (Grammar for $\{\texttt{a}^n\ \texttt{b}^{n+k} \mid n, k \geq 0\}$.)

$S \to E\,B$	// E generates strings with equal numbers of `a`'s followed by `b`'s
$E \to \texttt{a}\,E\,\texttt{b}$	// Get string $\texttt{a}^{n+1}\ \texttt{b}^{n+1}$ from string $\texttt{a}^n\ \texttt{b}^n$
$E \to \varepsilon$	
$B \to \varepsilon$	// B generates strings matching `b*`
$B \to \texttt{b}\,B$	

5. (Language for strings of `a`'s and `b`'s with an equal number of `a`'s and `b`'s, in any order.)

$S \to S\,S$	// S generates a string with equal nbr `a`'s and `b`'s
$S \to A\,S\,B\,S$	
$S \to B\,S\,A\,S$	
$S \to \varepsilon$	
$A \to \texttt{a}\,S$	// A generates strings with one more `a`'s than `b`'s
$A \to S\,\texttt{a}$	
$B \to \texttt{b}\,S$	// B generates strings with one more `b`'s than `a`'s
$B \to S\,\texttt{b}$	

6. Omitted