Context-Free Grammars

A Way to Specify Some Nonregular Languages

Monday, October 29, 2007
Reading: Stoughton 4.1, Sipser 2.1

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Overview

- Introduce **Context-Free Grammars** (**CFGs**), a new formalism for specifying languages = so-called **Context-Free Languages** (**CFLs**).
- Define the set of strings denoted by a Context-Free Grammar via the notions of **derivations** and **parse trees**.
- Show that CFGs can specify some simple nonregular languages.

A Sample Context-Free Grammar (CFG)

LHS		RHS
S	→	AB
A	→	0A1
A	→	%
B	→	1B0
B	→	%

Informally, a context-free grammar (CFG) is a collection of **productions** = **substitution rules** for rewriting **variables** (a.k.a. **nonterminals**) to strings of variables and **terminals** (non-variable symbols).

Each rule has a **left-hand side** (**LHS**) consisting of a single variable and a **right-hand side** (**RHS**) consisting of variables and terminals.

A CFG has a **start variable**, which is conventionally the variable in the LHS of the first rule.

A string of variables and terminals can be rewritten to another by substituting the RHS of a rule for the variable in the LHS. E.g.:

$0A11B0 \Rightarrow 00A111B0$ $\quad 0A11B0 \Rightarrow 0\%11B0 = 011B0$

$0A11B0 \Rightarrow 0A111B00$ $\quad 0A11B0 \Rightarrow 0A11\%0 = 0A110$

Derivations Generate Strings

A sequence of substitution steps that rewrites the start variable to a string of terminals is called a **derivation**. A CFG **generates** a string of terminals *s* if there is a is derivation of *s*. E.g.:

S	→	AB
A	→	0A1
A	→	%
B	→	1B0
B	→	%

$S \Rightarrow AB \Rightarrow \%B = B \Rightarrow \%$

$S \Rightarrow AB \Rightarrow 0A1B \Rightarrow 0\%1B = 01B \Rightarrow 01\% = 01$

$S \Rightarrow AB \Rightarrow A1B0 \Rightarrow A1\%0 = A10 \Rightarrow \%10 = 10$

$S \Rightarrow AB \Rightarrow 0A1B \Rightarrow 0A11B0 \Rightarrow 00A111B0$

$\Rightarrow 00A111\%0 = 00A1110 \Rightarrow 00\%1110 = 001110$

Explicit %s are usually omitted from a derivations unless % is the final string:

$S \Rightarrow AB \Rightarrow B \Rightarrow \%$

$S \Rightarrow AB \Rightarrow 0A1B \Rightarrow 01B \Rightarrow 01$

Leftmost and Rightmost Derivations

S	→	AB
A	→	0A1
A	→	%
B	→	1B0
B	→	%

There are often multiple derivations generating the same string that differ inconsequentially in the order of substitutions performed. E.g.:

S ⇒ AB ⇒ 0A1B ⇒ 0A11B0 ⇒ 00A111B0 ⇒ 00A1110 ⇒ 001110

S ⇒ AB ⇒ A1B0 ⇒ 0A11B0 ⇒ 0A110 ⇒ 00A1110 ⇒ 001110

We can standardize the sequence by always substituting for the leftmost variable (resulting in a **leftmost derivation**):

S ⇒ AB ⇒ 0A1B ⇒ 00A11B ⇒ 0011B ⇒ 00111B0 ⇒ 001110

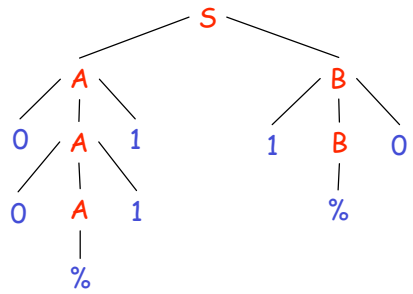
or the rightmost variable (resulting in a **rightmost derivation**):

S ⇒ AB ⇒ A1B0 ⇒ A10 ⇒ 0A110 ⇒ 00A1110 ⇒ 001110

Parse Trees

S	→	AB
A	→	0A1
A	→	%
B	→	1B0
B	→	%

Any sequence of rewriting steps can be depicted as a **parse tree** in which each internal node shows how a variable rewrites to the children of the node.



The **yield** of a parse tree = the string consisting of the leaves of the tree from from left to right = the result of the rewriting steps.

For the above parse tree, the yield = 00%111%0 = 001110

Alternative Ways to Write CFGs

Can combine multiple productions from the same variable using |

```
S → AB
A → 0A1 | %
B → 1B0 | %
```

In the literature, the → in productions is often replaced by ::=, especially in so-called Backus-Naur Form (BNF).

```
S ::= AB
A ::= 0A1 | %
B ::= 1B0 | %
```

Formally, a CFG is a quadruple:

```
({S,A,B},   (1) set of variables
 {0,1},     (2) set of terminals
 S,         (3) start variable
 {(S,AB), (4) productions
  (A,0A1), (A,%)
  (B,1B0), (B,%)}
 )
```

Forlan format

```
{variables}
S, A, B
{start variable}
S
{productions}
S -> AB;
A -> % | 0A1;
B -> % | 1B0
```

Only three parts are specified because the terminals are implicitly defined as **Sym** - variables

The Language of a CFG

The language of a CFG is the set of all terminal strings generated by the language.

What is the language of our sample CFG?

```
S → AB
A → 0A1
A → %
B → 1B0
B → %
```

A language that can be specified by a CFG is called a **context-free language** (CFL).

Designing CFGs for Some Simple Languages

What is a CFG for $\{0^n1^n \mid n \in Nat\}$?

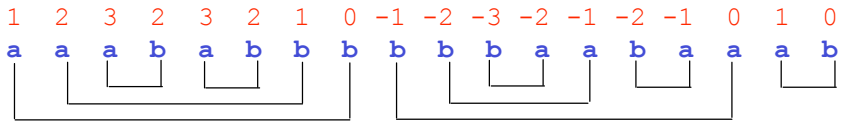
What is a CFG for $\{0^m1^n \mid m \geq n\}$?

What is a CFG for $\{0^m1^n \mid m > n\}$?

{x | x in {a,b}* contains equal # of as & bs}

What is a CFG for the above language?

For intuition, consider annotating each symbol with #as - #bs so far:



Note that each a matches a particular b.

Balanced Parentheses

Consider a language in which the only two terminals are (and).

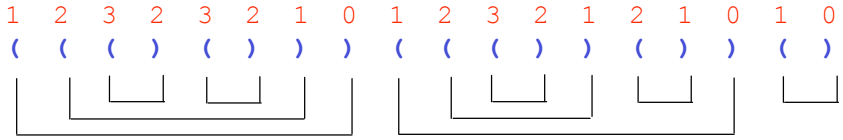
Let L(x) = # of left parens in x; R(x) = # of right parens in x

A string of parentheses is **balanced** iff

(1) L(x) = R(x) (alternatively, L(x) – R(x) = 0.)

(2) For every prefix y of x, L(y) ≥ R(y) (alternatively, L(y) – R(y) ≥ 0)

This is just like the language with equal # of as and bs, except that difference can never be < 0.



Note that each (matches a particular) after it.

What is a CFG for Balanced Parentheses?

Intuitively, why is the CFG correct?

(For a formal proof of correctness, see Kozen Lecture 20)

CFGs can Specify Natural Languages

<Sentence> → <NounPhrase><VerbPhrase>
<NounPhrase> → <Article><NounUnit>
<NounUnit> → <Noun> | <Adjective><NounUnit> | <NounUnit> that <VerbPhrase>>
<VerbPhrase> → <Verb> <NounPhrase>
<Article> → a | the
<Adjective> → big | small | black | gray | furry
<Noun> → dog | cat | mouse | bug
<Verb> → loves | chases | eats

(Imagine the nonterminals are tokens, not strings.)

Give a parse tree for the following sentence:
The big black dog that chases the gray cat loves a furry mouse that eats a bug

CFGs can Specify Programming Languages

Here is a CFG for SLiP:

<Stm> → <Stm> ; <Stm> | [ID(*string*)] := <Exp> | print (ExpList)
<Exp> → [ID(*string*)] | [INT(*integer*)]
| <Exp> [OP(*binop*)] <Exp> | (<Stm> , <Exp>)
<ExpList> → Exp | <ExpList> , <Exp>

(Note: ; stands for [SEMI] token, (stands for [LPAREN] token, etc.)

Give a parse tree for the following statement:
prod := (print (sum, sum-1), 10*sum);