CFG Introduction

Motivation

- Regular expressions are too limiting to express everything we need
- Some things are absolutely *impossible* with regexes

Example

- Produce regex for balanced parentheses: (), (()), ((())), etc.
- Can't be done!
  - Why not?
  - Pumping lemma

Idea

- Any regex is equivalent to some DFA
- DFA has only finite number of states
- That's the only mechanism DFA has to remember anything: Change current state
- Suppose the DFA has n states
- If we create a string with n+1 opening parens: DFA *must* re-enter a state twice
- But once the DFA re-enters that state: It's exactly the same as when it entered the state the first time (when it had less than n+1 parens)
  - No memory of anything else that happened before!
- Now the DFA doesn't know how many opening parentheses it's seen so far

Grammars

- We need something more powerful
- We use *grammars*
- Grammars have other advantages for us
  - Precise, concise, easy to grasp what it says
  - Easier to build parser from grammar than without it
  - Helps ensure language structure is correct
  - Can “evolve” language: add new constructs to grammar, ensure doesn’t break existing language features

Specification

- Terminals: These cannot change into anything else
  - Represent the tokens
- Nonterminals: These can change into other things
- Productions: Means for changing a nonterminal to one or more terminals and/or nonterminals
- Start symbol: Where to begin
  - By convention: first symbol or else called “S”

Example

- Terminals: a b c x y n
- $S \rightarrow A \mid B \mid n \mid x\ A\ y\ B$
- $A \rightarrow a\ b\ c$
- $B \rightarrow x\ S\ y$
- Ex: Can this grammar produce the string x a b c y x n y ?
- Yes: Derivation:
  - $S \rightarrow x\ A\ y\ B \rightarrow x\ a\ b\ c\ y\ B \rightarrow x\ a\ b\ c\ y\ x\ S\ y \rightarrow x\ a\ b\ c\ y\ x\ n\ y$
- This is usually the way the problem is posed in automata class.

Observe

- At each step of derivation:
  - We select exactly one nonterminal
  - We select a production
  - We replace that nonterminal with its production
- Repeat until entire string is all terminals

Notice

- Here, productions are always nonterminal $\rightarrow$ ...something...
- This is called a *context free grammar*
  - What a nonterminal can change into does not depend on what's next to it
- We can also create *context sensitive grammars*

Example

- Consider this grammar. The terminals are A, AN, FROG, APPLE, JUMPS, RIPENS, SLOWLY, and QUICKLY:
  article $\rightarrow$ A | AN
  noun $\rightarrow$ FROG | APPLE
  verb $\rightarrow$ JUMPS | RIPENS | ROTS
  adverb $\rightarrow$ SLOWLY | QUICKLY
  sentence $\rightarrow$ subject action adverb
  action $\rightarrow$ verb
  A FROG JUMPS $\rightarrow$ subject action
  AN APPLE RIPENS $\rightarrow$ subject action
  AN APPLE ROTS $\rightarrow$ subject action
- This is a context sensitive grammar

CSG

- CSG's are more difficult to work with
  - Linear bounded automaton vs. Pushdown automaton
- We generally stick with CFG's
- But...some constructs can't be expressed with CFG
  - Ex: "You must declare variables before you use them."
- We'll add rules outside the CFG for those cases.

Compilers

- In automata theory, problems are often posed as “What strings can we derive with this grammar”
- For compilers, we have a somewhat different question: *Here’s a string (a program). How do we derive it?*

Ambiguity

- Sometimes, a grammar gives us several ways to derive a string
- This means grammar is *ambiguous*
- Ex:
  $S \to x\ A \mid B\ y$
  $A \to y$
  $B \to x$
- We can derive
  $x\ y$
  in two ways (do you see how?)

λ Productions

- Often, it is useful to allow a nonterminal to simply "evaporate"
- We represent this as a λ-production
  - Some people use ε or Λ instead
- Ex:
  $S \rightarrow w\ S \mid \lambda$
- Derive: w w w
  $S \rightarrow w\ S \rightarrow w\ w\ S \rightarrow w\ w\ w\ S \rightarrow w\ w\ w$

Grammar

- Suppose we want to create a grammar that allows us to add two numbers
- How could we do that?
  - Do in class!

Grammar

- Suppose we want a grammar that allows us to add two things or multiply two things
- How could we do that?
  - Do in class!

Problem

- What if we want to allow adding or multiplying arbitrarily many things?
  - Do in class!

BNF

- Sometimes, we use BNF or EBNF to specify grammar
- Equivalent expressive power to CFG, but a little easier to read
  - ( ) are used for grouping
  - [ ] indicates optional item
  - { } indicates repetition
  - | indicates choice
  - ’xyz’ means “literal xyz”
  - Often use ::= instead of arrow
  - I’m going to introduce my own extension and use < > to delimit regular expressions

Example

- CFG for an if-else expression:
  if-stmt $\rightarrow$ IF LPAREN ID EQUALS NUM RPAREN LBRACE stmts RBRACE | IF LPAREN ID EQUALS NUM RPAREN LBRACE stmts RBRACE ELSE LBRACE stmts RBRACE
- EBNF for the same thing:
  if-stmt ::= ’if’ ’(’ ID ’==’ NUM ’)’ ’{’ stmts ’}’ [’else’ ’{’ stmts ’}’]

Example

- Suppose we want a list of statments
- CFG:
  stmts $\rightarrow$ stmt SEMICOLON stmts | λ
- EBNF:
  stmts ::= { stmt ';' }

Assignment

- Write a program which receives a single command line argument: This will be the name of a grammar specification file
- The file can be described with this EBNF grammar:

grammar ::= { terminal } '\n' { nonterminal }
identifier ::= <\w+>
terminal ::= { identifier } '->' { regex } '\n'
regex ::= ...any valid regular expression...
nonterminal ::= production [{ '|' production }] '\n'
production ::= 'lambda' | { identifier }

- Be careful of lambda!

Output

- For each nonterminal, report how many productions it has
- Also report the longest production in the entire grammar (the number of symbols and then the lhs and then an arrow and then the rhs)
- Your output should follow the form given in this EBNF grammar:

```
output ::= { line } longest
line ::= symbol ’ ’ number ’\n’
number ::= <\d+>
symbol ::= <\w+>
longest ::= number ’ ’ symbol ’->’ production
production ::= { symbol }
```

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, and Tools. 2nd ed.
- K. Louden. Compiler Construction: Principles and Practice
- Python Tutorial. http://www.python.org
- Java Documentation. http://java.sun.com
- Boost Documentation. http://www.boost.org
- PCRE Documentation
- https://en.wikipedia.org/wiki/ Extended_Backus%E2%80%93Naur_form
- Regexp. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
- https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm
- Microsoft. EBNF Overview. https://msdn.microsoft.com/en-us/library/aa597401.aspx

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org