College of Engineering and Computer Science

UNIVERSITY OF CENTRAL FLORIDA

Context-Free Grammars

COP-3402 Systems Software

Paul Gazzillo

The Parsing Problem

- Lexing recognizes "words" (lexemes)
- Programming languages have nested structure

  while (x > 3) { print x * (3 + y); }

- Regular languages cannot express nested structure
- How do we recognize languages with nesting?
- Terminology note: parsers vs recognizers
  - Recognizer - check whether input is in language
  - Parser - generates a parse tree, is also a recognizer
  - (I may abuse these terms, interchanging them)

UCF

Limitation of Regular Languages

- Example: unlimited matching parentheses, "((( )))"
- Formal proof with the *Pumping Lemma*
    - All regular languages can be "pumped"
        - (Converse not always true, though)
    - Pump: language's strings are repeating pattern
        - For any sufficiently long string in the language
- Intuition: finite states for infinite set of strings
    - States will eventually be repeated (*pigeonhole principle*)
    - Matching parentheses will need infinite states

Recognizing Nested Structure in Language

- Read one symbol at-a-time
  - Just like lexers
  - For compilers, one token at-a-time (instead of characters)
- Match patterns of symbols (just like lexers)
  - But track nested structure using a stack
  - Equivalent to a *pushdown automaton*
  - Finite automaton plus a stack
- Express nested structure using *grammar*
  - We will look at *context-free grammar*

Context-Free Grammars Capture Nesting

- Language definition for arithmetic expressions
    - Expressions can contain nested expressions

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | LPAREN expression RPAREN
  | NUMBER
```

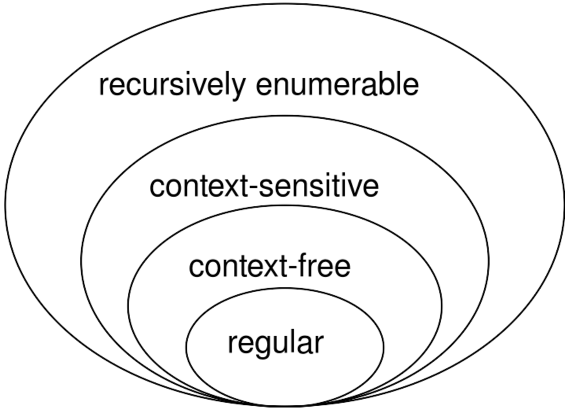
$$\boxed{\boxed{2} \; * \; \boxed{(\boxed{\boxed{3} + \boxed{4}})}}$$

Languages are Structured

- Noam Chomsky's *generative grammars*,
  - Innate language ability
- Captures hierarchical nature of languages
- Intuition: syntactic correctness, even without meaning

"Colorless green ideas sleep furiously."
VS.
"Furiously sleep ideas green colorless."

(Chomsky, Syntactic Structures)



Context-Free Grammars Use Symbols to Represent the Structures Themselves

- Recall definition of a language
  - Potentially infinite set of strings over a finite alphabet
- A new kind of symbol represents language structures
  - *Nonterminals* represent constructs, not part of the string
  - *Terminals* represent the symbols, part of strings
- Language generated by substitution rules
  - Nonterminals symbols expand to "smaller" symbols
  - Terminals "terminate" the expansion

Demo: Arithmetic Expressions

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | LPAREN expression RPAREN
  | NUMBER
```

```
2  *  ( 3  +  4 )
```

Definition of Context-Free Grammars

- *Terminals* are the "words"
  - For our compiler, terminals are tokens from the lexer
  - Terminals cannot be broken down, "terminate" expansion
- *Nonterminals* represent structures
  - Can always be further broken down into symbols
- *Productions* are substitution rules
  - Nonterminals are replaced with a sequence of symbols
- The *starting symbol* is the first nonterminal to replace

Terminal vs Nonterminal

- IDENTIFIER?
- statement?
- program?
- SEMI?
- NUMBER?
- expression?

Notations for Context-Free Grammars

- Chomsky normal form as used in Dragon Book
  - Uppercase letters for nonterminals
  - **Bold** or punctuation for terminals
  - Arrows **->** for productions
  - Pipe **|** for alternate productions
  - Gotcha: parentheses are terminals

```
E -> E + number
   | E - number
   | ( E )
```

(In the original, `number` is printed in bold.)

- Notation for this class
  - Blend of Extended Backus-Naur Form and regular expressions
  - lowercase for nonterminals
  - UPPERCASE for terminals
  - Equals sign **=** for productions
  - Pipe **|** for alternate productions
  - Gotcha parentheses are **not** terminals
    they group symbols as in regular expressions

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | LPAREN expression RPAREN
```

Derivation: Generating Strings from the Grammar

Derivation:
Generating a String from the Grammar

- Generative grammar means utterances are created by following productions from starting symbol to string
- Each step uses a production, replacing the nonterminal
  - Replaced with a sequence of nonterminals and terminals
- Grammar may be recursive
  - E.g., expressions nested within expressions

Demo: Arithmetic Expressions

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | LPAREN expression RPAREN
  | NUMBER
```

```
2 * ( 3 + 4 )
```


UCF

Leftmost vs Rightmost Derivations

- Each step may have multiple nonterminals to replace
- A leftmost derivations always replaces the leftmost first
  - Conversely, rightmost derivations replace rightmost first

Parsing Is Finding the Derivation of a String

- Generative grammar: produce *string* from *derivation*
- Parser: produce *derivation* from *string*
- The parser infers how string must have been created

Derivations as Trees

- The parser constructs *parse trees*
- The root node is the starting symbol
- Each inner node is a nonterminal
- Child nodes are the symbols from the production
- Leaf nodes are terminals

Demo: Parse Trees for Expressions

Ambiguity: Multiple (Leftmost) Derivations

- A grammar is ambiguous when there is a string s.t.,
  - There is more than derivation of the same string
  - (leftmost or rightmost derivation)
- For example: 2 + 3 * 7
  - Two interpretations, addition first or multiplication first
- Corresponds to different parse trees, i.e., derivations

Demo: Ambiguous Expressions

Resolving Ambiguity

- What does ambiguity reflect in the expression?
- How can we resolve the ambiguity?
- How can we implement the solution in the grammar?

Operator Precedence

- Enforce operator precedence with the grammar
- Add extra nonterminals and productions
  - Adds factors and terms
  - Restricts operators to specific productions

Demo: Operator Precedence

Conclusion

- Context-free grammars express language structures
  - Programming languages expressed with grammars
- The parsing problem
  - Given a string, how is it derived from the grammar?
  - Your project will parse a structured language
- Ambiguities mean different derivations (parse trees)
- Resolve ambiguities by rewriting the grammar