Compilers: Week 3 - Parsing and Top-Down parsing

Ian Quah (itq)

June 15, 2017

Topic 1

Parsing - Introduction

Regular Languages

1. Weakest formal languages - most languages aren't regular

   $\{(^i)^i \mid i \geq 0\}$, the set of all balanced parens, which can't be represented as a regular language

2. What **CAN** a regular language express?

   (a) count mod k

   can't calculate to some arbitrarily high value, like in balanced parens problem

3.

Phase	Input	Output
Lexer	String of Chars	String of Tokens
Parser	String of tokens (output of lexer)	Parse Tree

Topic 2

Context Free Grammars

1. **The Problem**

   (a) Not all strings of tokens are programs

   Thus, need a language for describing valid strings of tokens and method to distinguish invalid and valid strings of tokens

   Note: RegExp $\subset$ CFG

   A regular language can be recognized by a finite automaton. A context-free language requires a stack

   (b) Programming Languages have a recursive structure, and CFGs are a natural notation for describing these structures.

2. **What IS a CFG?**

   (a) A set of terminals, T

   (b) A set of non-terminals, N

   (c) A start symbol, S (S $\in$ N)

   (d) A set of productions: (A symbol, followed by an arrow then a list of symbols)

   $X \rightarrow Y_1, \ldots Y_n$

   X $\in$ N and $Y_i \in N \cup T \cup \{\epsilon\}$

3. **CFG example: Balanced Parens Problem**

   Productions / Rules

$S \rightarrow (S)$	S, our start state then becomes another state with two parens around it
$S \rightarrow \epsilon$	

   Our States

   N = {S}

   T = {(,)}

   S = start symbol in general the first production will specify the start symbol for that CFG

Topic 2 continued on next page...

4. **Productions as rules:**

   (a) Begin with a String that only has start symbol, S

   (b) Replace any non-terminal X in start by RHS of some production

   (c) Repeat (2) until there are no non-terminals

   If we consider a single step as α, then a program can be thought of as

   $\alpha_0 \to \alpha_1 \to ... \to \alpha_n$ equiv $\alpha_0 \to^* \alpha_n$

5. **CFG formal definition**

   Let G be a CFG with start symbol S, then L(G) of G is:

$$\{a_1, ..., a_n | \forall i, a_i \in T, S \to^* a_1, ..., a_n\}$$

Topic 3

Derivations Part 1

1. **Derivation:** a sequence of productions

2. **Representations**

   - A derivation can be drawn as a tree instead of a linear path

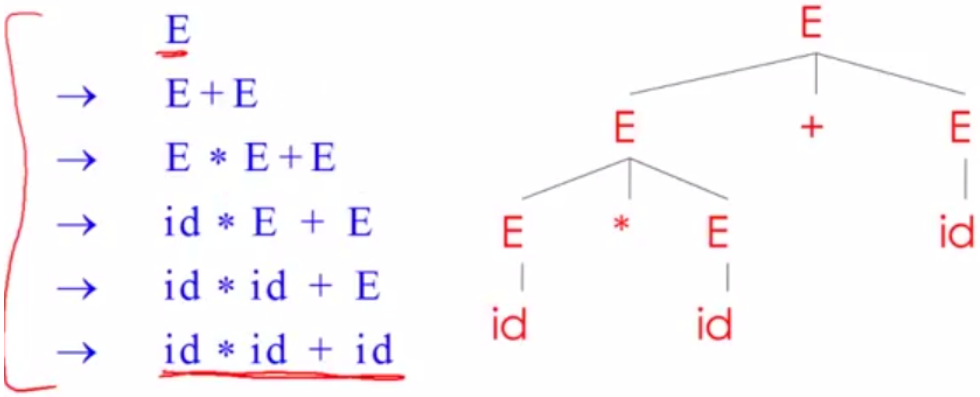
   - A derivation can be drawn as a tree

   (a) Start symbol is the root

   (b) For a production, add the children

   Example:

   Grammar: E → E + E | E * E | (E) | id



   LHS: Derivation intermediates (Left-most derivation)

   RHS: Parse tree of input string

3. **Parse Trees**

   (a) Terminals at the leaves

(b) Non-terminals at interiors

(c) in-order traversal of tree is original input

(d) Parse tree shows association of operations, input string does not (in tree we see multiplication is more tight than addition)

(e) Left-most and right-most derivation produces same parse tree, but the main difference is the intermediate step

Topic 4

Ambiguity in CFGs

1. **Ambiguity**: if there is more than one parse tree for some string (more than 1 left-most or right-most derivation)

   This is bad, because it means that the decision is left to the compiler

2. Determining if it's ambiguous

   Construct a string, then work backwards and see if there is more than 1 way to get the tree

3. **Removing ambiguity**

   id * id + id

   E → E' + E — E'

   E' → id * E' — id — (E) * E' — (E)

   rewrite to enforce precedence of * over +, by separating multiplication and + s.t * is handled by E'

Topic 5

Error Handling

Requirements of a Compiler

1. Report errors accurately
2. Not slow down compilation of valid code

Error Handling methods

1. Panic Mode
   (a) Simplest, most popular
   (b) When an error is detected:
   High-level Algorithm
      i. Discated tokens until one with a clear role is found
      ii. continue from there
   (c) Details
      i. Looks for syncrhonizing tokens
      ii. typically statement or expression terminators
2. Error productions

Topic 5 continued on next page...

(a) Specify known common mistakes in the grammar

(b) E.g :
Write 5x instead of 5 * x
Add the production E → ... | E E which makes it now well-formed

3. Automatic Error correction

Idea: find some correct "nearby" program

(a) Try token insertion and deletion

(b) Or do an exhaustive search

Disadvantages

(a) Hard to implement

(b) Storage of all the state is expensive

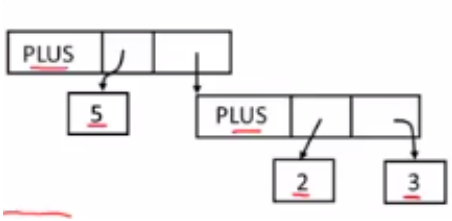
(c) Final output is not necessarily what the programmer intended. Safer to just error out

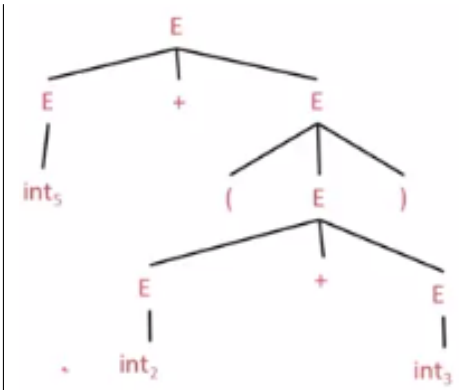
Topic 6

Abstract Syntax Trees

Why not use the Parse Tree

1. Contains a lot of redundant information.





See that in the parent of root nodes in parse tree, might as well just store the values themselves there, and that the parens were important in parsing but now they're not as important

Topic 7

Recursive Descent

Top-Down Construction

1. From the top
2. From Left to Right

Work our way down the productions, trying to match from the most general to most specific, then if it doesn't match at the root, we do backtracking up until the most general again

Recursive Descent Algorithm

1. A given token terminal

   bool term (TOKEN tok) {return *next++==tok;}

2. The nth production of S:

   bool S_n() {...}

3. Try all productions of S:

   bool S() {...}

Example case:

1. For production E → TOKEN

   bool E_1(){return T();}

2. For production E → T + E

   bool E_2{return T() && term(PLUS) && E();}

3. For all productions of E(with backtracking)

   bool E(){ TOKEN *save = next; return (next =save, E_1()) || (next=save, E_2();) }

4. Functions for non-terminal T

   bool T_1(){return term(INT);} T → int

   bool T_2() {return term(INT) && term(TIMES) && T();} T→int * T

   bool T_3(){return term(OPEN) && E() && term(CLOSE); } T → (E)

   bool T(){ TOKEN *save = next; return (T_1()) || T_2() || T_3); }

Limitations of RDA

1. There is no backtracking for a production meaning even though we may have succeeded in matching a rule, we may not consume all the strings.
2. Sufficient for grammars where for any non-terminal at most one production can succeed then.
3. Thus, we need to introduce another method.
4. * The method we were given earlier was given because it's mechanical and easy to do by hand.

Topic 8

Left Recursion

Addresses the issues we brought up in the previous video

1. Consider the production S → S a
   (a) bool S_1() { return S() && term(a);}
   (b) bool S() {return S_1();}
   (c) S() goes into an infinite loop
2. This is called a left-recursive grammar: it has a non-terminal S

   S $\rightarrow^+$ S α for some α

3. **Consider a more general production** S $\rightarrow$ S α | β

   S generates all strings starting with a β and followed by any number of α

   S generates all strings starting with a β followed by any number of αs

4. **The solution: rewriting using right-recursion**

   S $\rightarrow$ β S'

   S' $\rightarrow$ α S' | ϵ

5. **General form**

   (a) In General
   S $\rightarrow$ S α_1 | ... | S α_n | β |... β_m

   (b) All strings derived from S start with one of $\beta_1, ...\beta_m$ and continue with several instances of $\alpha_1, ..., \alpha_n$

   (c) Rewrite as
   S $\rightarrow$ β_1 S' | ... | β_m S'
   S' $\rightarrow$ α_1S' | ... | α_n S' | ϵ

Topic 9

Quiz hints

1. Expand the production rules, then think of binary
2. Build off the previous one, then consider the number of times you can add epsilon as well as the number of unique strings it will generate
3. Think of whether it would be possible to write a RegExp that would accept a string generated by the grammar

   (a) The answer was discussed in a video

   (b) Construct a regexp

   (c) My advice here is to start from the simplest case, then work your way backwards (I.e consider D, then C from D and so forth)

4. unfortunately here there is no good trick that I know of. You need to substitute in all the values and see if it generates something left-recursive
5. Substitutions
6. Throw all you know about order of operations out and build different trees from there (the parenthesis rule still holds though). Write down the different configurations you generated - will be useful for next question
7. Review the definition of associates left, then consider each case given.
8.
9.
10.

11. I think of ambiguous as not being able to determine the order again, but if you keep the same ordering (hint hint), and the rules are uniform then you can generate the original (I *think*)

12. Think of short circuiting, and review the RD algorithm thoroughly.