Class 8

On to how the parser actually works under the hood…

Parsing Methods

Universal: For every CFG, there exists a parser that will take at most $O(n^3)$ time, $O(n^2)$ space.

Cocke-Young-Kasami: check whether each consecutive substring is possible: dynamic programming

Early: build all possible trees in parallel

BUT we want linear time in input size:

- single left to right scan of input program
- lookahead of 1 token at a time
- no backtracking

Linear-time Parsers:

Top-down: root “expanded” to leaves

- recursive-descent
- **LL(1) predictive parsers**

Bottom-up: leaves “reduced” to root

- LR family: SLR, LALR, LR(1) canonical

Top-down Parsing

Goal: Find leftmost derivation starting at root and building tree in preorder.

Why leftmost derivation?

What do we mean by avoiding the backtracking to be linear?

Consider:

S -> aAd | aB
A -> b | c
B -> ccd | ddc

Input: accd

Exploring Top Down Parsing Challenges

Consider: procedure id (param list) ; param list is optional

where param list => param : type; param : type;…param:type
param => var id, id, …, id
var is optional

Context-free Grammar:

```
S -> procedure id P ; | ε
P -> ( L ) | ε
L -> R : T | R : T ; L
R -> V D
V -> var | ε
D -> D , id | id
T -> int | real
```

String: procedure print (var x,y,z: int; a,b: real);

Recursive-descent Parsing

CFG: T -> T * F | F
F -> (E) | id

=> T -> F T'
T' -> * F T' | ε
F -> (E) | id

```
T( )
{


}

T' ( )
{


}
```

```
F( )
{




}
```

Consider input string: a * b * c