CPSC 323 Compilers and Languages

Instructor: Susmitha Padda

Credits : Doina Bein

Chapter 3. Syntactic Analysis I
(Top down parsing)

3.1 Introduction
3.2 Grammar
3.3 Chomsky Hierarchy
3.4 Left-Recursion and Left Factorization (Backtracking)
3.5 Top-Down Parsers
    a. Recursive Descent Parser (RDP)
    b. Predictive Recursive Descent Parser (PRDP)
    c. Table Driven Predictive Parser

Context-free languages

- The class of CFLs contain all regular languages, as well as some nonregular languages such as $a^n b^n$, #a=#b which are not regular
- These languages have some sort of recursive structure
- We know that the language $A = \{a^n b^n \mid n \geq 0\}$ is NOT regular; but it is a CFL. The recursive definition of A:
  - Basis: ε is in A
  - Recursive: if x is in A, then axb is in A
  - Exclusive: nothing else

- We also know that *pal* is not regular; but *pal* is CFL. Assume the alphabet Σ={a,b}.
  - The recursive definition of *pal*:
    - Basis: ε, a, b are in *pal*
    - Recursive: if x is in *pal*, then axa and bxb are in *pal*
    - Exclusive: nothing else
- We also know that C = { w | w has equal number of 1s and 0s} is not regular; but it is CFL.
  - The recursive definition of C:
    - Basis: ε is in C
    - Recursive: if x is in C, then 1x0 and 0x1 are in C
      if x and y are in C, then xy is in C
    - Exclusive: nothing else

- For a FA, the automaton has finite memory
- For a CFL, the automaton needs an infinite memory, but it is organized as a stack, with strict rule of in-out
- A CFL is accepted by some Context-Free Grammar (CFG); a CFG accepts a CFL
- A CFL is accepted by some Pushdown Automaton (PDA)
- Pushdown automata are deterministic or nondeterministic
- But the set of all deterministic PDAs ≠ set of all nondeterministic PDAs
- Most programming languages have a CFG base, but with extra conditions
- A CFL is language is generated by a CFG, so in order to study the properties of CFLs, we start with CFG

- For regular languages we have regular expressions, for CFL we have CFGs
- Later we will see that regular languages are generated by regular grammars, that are a special type of CFG
- Going back to our examples
- For the language A = $\{0^n1^n \mid n \geq 0\}$, we can rewrite the recursive definition as the following rules:
  - S can take the value ε: $S \rightarrow \varepsilon$
  - S can take the new value of 0S1 computed from the old S: $S \rightarrow 0S1$
  - And these two rules generate A:
    $S \rightarrow \varepsilon$
    $S \rightarrow 0S1$

- Language *pal*? The rules are:
  $S \rightarrow \varepsilon$
  $S \rightarrow a$
  $S \rightarrow b$
  $S \rightarrow 0S0$
  $S \rightarrow 1S1$
- How about the language B? The rules are:
  $S \rightarrow \varepsilon$
  $S \rightarrow 1S0$
  $S \rightarrow 0S1$
  $S \rightarrow SS$
- These rules belong to a CFG

Context Free Grammars

- A CFG is a 4 tuple G = (V, Σ, R, S) where
  Σ is called the *set of terminal symbols*; it is finite
  V is called the *set of variables* (non-terminal symbols); it is finite and Σ and V are disjoint: $V \cap \Sigma = \emptyset$
  S $\in$ V is called the *start variable* (or start symbol); it is unique
  R is a finite set, whose elements are called *rules* (or productions); each rule is of the form $A \rightarrow w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$, i.e w is a string of terminals and/or variables, including empty string
- For our three examples, the grammars are …

Derivations

- If we want to see how a string is generated, let's take $abba \in pal$:
  S → aSa by applying the rule S → aSa
  →abSba by applying the rule S → bSb
  →abba by applying the rule S → ε
  We have S → aSa →abSba →abba. These are called *derivations*.
- A derivation is a transformation of a variable using one of the existing rules.
- When writing derivations, we use the symbol $\underset{G}{\Rightarrow}$:
  S $\underset{G}{\Rightarrow}$ aSa $\underset{G}{\Rightarrow}$ abSba $\underset{G}{\Rightarrow}$ abba
- Formally, a string $\beta \underset{G}{\Rightarrow} \gamma$ is there exists a rule $A \to w \in R$ such that
  β = u A v
  γ = u w v
  so u$Av \underset{G}{\Rightarrow} uvw$

Language generated by a CFG

- In our example, $S \underset{G}{\Rightarrow} aSa \underset{G}{\Rightarrow} abSba \underset{G}{\Rightarrow} abba$ can be shorten it to $S \underset{G}{\Rightarrow}^* abba$
- More generally, we write $\beta \underset{G}{\Rightarrow} *\gamma$ is the string γ can be derived in zero or more steps from β.
- Let G = (V, Σ, R, S) be a CFG. The language generated by G is
  $L(G) = \{ x \in \Sigma^* \mid S \underset{G}{\Rightarrow}^* x \}$
- A language L is a *context free language* (CFL) if there is a CFG G such that L = L(G).

Examples of context-free grammars

- Properly nested parentheses: consider a to be (and b to be). The grammar would be G = (V, Σ, R, S) where V = {S},
  R = { S → ε, S → aSb, S → SS}.
  We can also write R = { S → ε |aSb | SS} where | is the shorthand for “or”.
  - Examples of strings in the language. Examples of strings not in the language.

Example

- The program:

  x * y + z

- Input to parser:

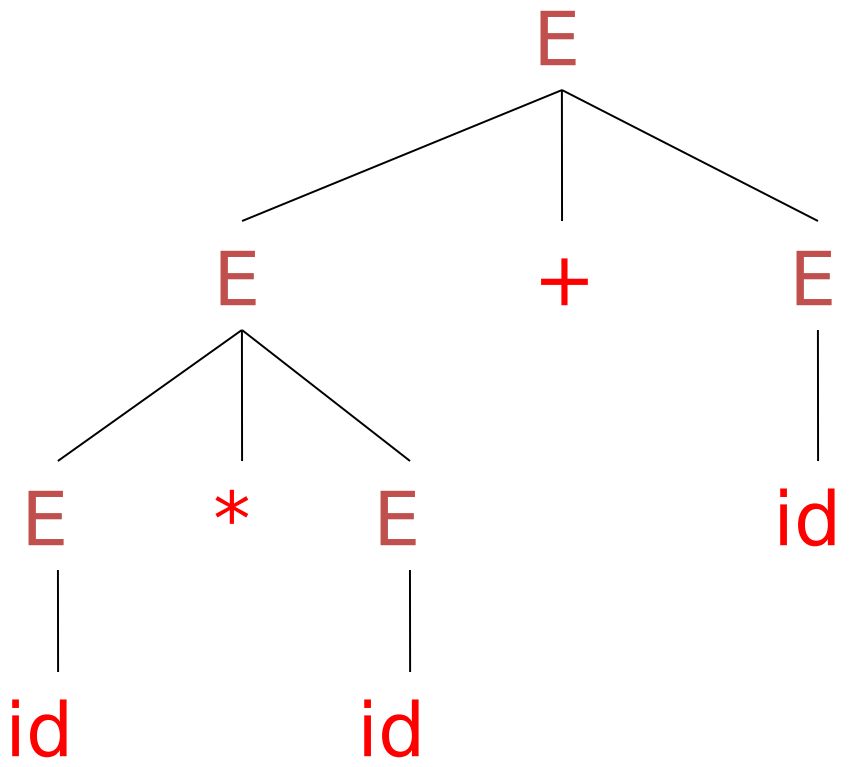
  ID TIMES ID PLUS ID

  we'll write tokens as follows:

  id * id + id

- Output of parser:

  a parse tree



What must a parser do?

1. Recognizer: not all sequences of tokens are programs
   - must distinguish between valid and invalid strings of tokens
2. Translator: must expose program structure
   - e.g., associativity and precedence
   - hence must return the syntax tree

We need:

- A language for describing valid sequences of tokens
  - *context-free grammars*
  - (analogous to regular expressions in the scanner)
- A method for distinguishing valid from invalid strings of tokens (and for building the syntax tree)
  - *the parser*
  - (analogous to the finite state machine in the scanner)

3.1 Introduction

- The *syntactic analyzer*, or *parser*, is the heart of the front-end of a compiler
- Most programming languages are not completely described by the grammars we will learn in this chapter (context-free grammars - CFGs)
- Example: CFGs cannot described the restriction of strongly-typed languages that every variable must be declared before being used
- Informally, *a grammar* is a finite set of rules for generating an infinite set of sentences; finite languages are of little interest
- The rules impose some desired structure of sentences
- In natural languages, sentences are made up of words; in programming languages, sentences are made up of tokens
- *Syntax Analysis* is a phase where the structure of the source code is recognized and built using a finite set of rules called *productions*.

Cont.

- Ex: a set of rules for a subset of English sentences

  <Sentence> -> <Noun Phrase> <Verb Phrase>

  <Noun Phrase> -> <Article> <Noun>

  <Verb Phrase> -> <Verb> <Noun Phrase>

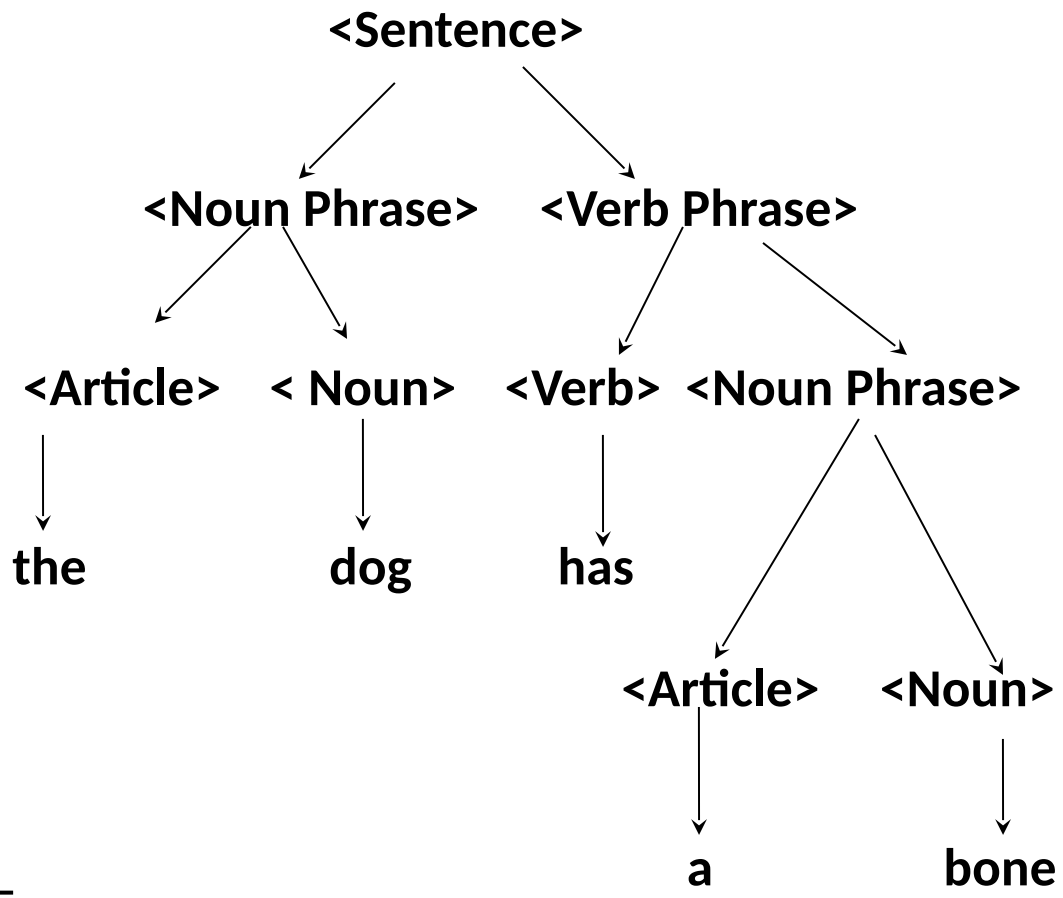
  <Noun> -> dog, cat, bone, school

  <Article> -> the, a, an

  <Verb> -> goes, has

- Each production uses a finite number of *terms*; a term is either
  - a *non-terminal symbol (or nonterminal)*that is the lhs of one or more productions
  - a *terminal symbol* (or *terminal*) that is not the lhs of any production
- We use a type of grammar called a *generative grammar* that builds a sentence in a series of steps, starting with the concept of sentence and refining it using productions until the actual sentence emerges by applying the rules
- Example on next slide

Sentence: The dog has a bone



However, the 2nd sentence:

The dog goes a bone

has no meaning!

Cont.

- Such a tree is called a *parse tree*
- The root of the tree is our starting point, a sentence
- The tree is oriented: the children are read off from left to right
- The actual words are terminal symbols
- Syntax versus semantics:
  - Syntax deals with the way a sentence is put together
  - Semantics deals with what the sentence means
  - Grammars define the proper structure of a sentence, but have no bearing on the meaning of it
  - The parser looks for grammatical errors but the programmer must see if the program written makes sense

3.2 Grammar

- Def: A *grammar* G = (T, N, S, R) where

  T is a finite set of terminal symbols

  N is a finite set of non-terminal symbols; N and T are disjoint

  S ∈ N is a unique starting symbol

  R is a finite set of productions of the form α -> β where α, β are strings of terminals and nonterminals

- Chomsky convention:
  - nonterminals are represented by capital letters,
  - terminals by lowercase letters early in the alphabet
  - Strings of terminals by lowercase letters late in the alphabet, and
  - Mixtures of nonterminals and terminals by lowercase Greek letters
- Def: The *language* of grammar G is the set of all sentences that can be generated by G and it is written L(G).
- If each string α is a single nonterminal then the grammar is called *context free (CFG)*
- The language of a CFG is called a *context free language.*

How to represent productions in CFGs

- BNF (Backus-Naur(Normal) Form) notation;

BNF grammars = Context free grammars

- Extended BNF
- Chomsky Normal Form (CNF)
- Syntax diagrams

BNF Notation

- Is a specific notation to describe the productions in which nonterminals are placed in angle brackets, the symbol ::= is used in place of the arrow, several productions share the same lhs, and the symbol | means “or”
- Example

```
<Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>

<Term> ::= <Term> * <Factor> | <Term> / <Factor>  | <Factor>

<Factor> ::=  <Identifier>   | <Number>  |  ( <Expression> )
```

Example of a grammar

G = (

T = {id, +, -, *, /, (,) },

N = {<Expr>, <Term>, <Factor> }

S = <Expr>,

R = { 1. <Expr> -> <Expr> + <Term>

2. <Expr> -> <Expr> - <Term>

3. <Expr> -> <Term>

4. <Term> -> <Term> * <Factor>

5. <Term> -> <Term> / <Factor>

6. <Term> -> <Factor>

7. <Factor> -> id

8. <Factor> -> (<Expr>)

}

)

Extended BNF (EBNF)

- *Extended BNF*: allows additional notations; such as

  { } 0 or more times,

  [ ] optional

- Example:

```
<Number> ::= <Digit> { <Digit> }
<Identifier> ::= <Letter> { <Letter> |
<Digit> }
<Factor>  ::=    [ - ]  <Primary>
```

Derivation

- Def: *Derivation* is replacing one non-terminal symbol at a time in order to recognize a sentence.
- Ex. Source code: a / (c – d)

Set of rules:

<Expr> => <Term>

=> <Term> / <Factor>

=> <Factor> / <Factor>

=> id(a) / <Factor>

=> a / (<Expr>)

=> a / (<Expr> - <Term>)

=> a / (<Term> - <Term>)

=> a / (<Factor> - <Term>)

=> a / (c - <Term>)

=> a / (c - <Factor>)

=> a / (c – d)

- In general, we say <Expr> $\Rightarrow^*$ a / (c-d)

Cont.

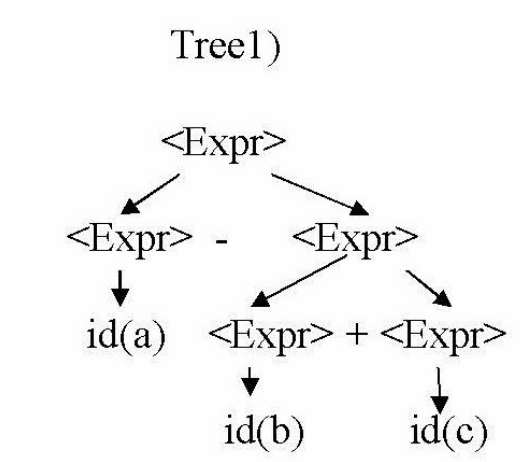
- A *sentential form* is a string of symbols (terminal or nonterminal) appearing in various steps in derivation
- The *left most derivation* (LMD): in each step, always replace the leftmost nonterminal.
- The *right most derivation* (RMD): in each step, always replace the right most nonterminal
- Def: The *language accepted* by a CFG grammar is the set of all the strings for which there exists a derivation starting from the starting symbol S
  L(G) = {x | S =>* x }
- Given a CFG G and a string $x \in L(G)$, *parsing* x means finding the derivation S =>* x .

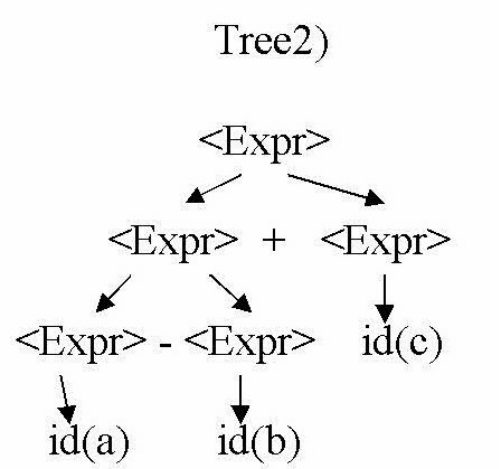
Ambiguity

- Def: A grammar is *ambiguous* if there are two different parse trees for some words in L(G)
- Ex: consider productions

```
<Expr> -> <Expr> + < Expr>
<Expr> -> <Expr> - <Expr>
<Expr> ->  id
```

- Consider the string w= a-b+c

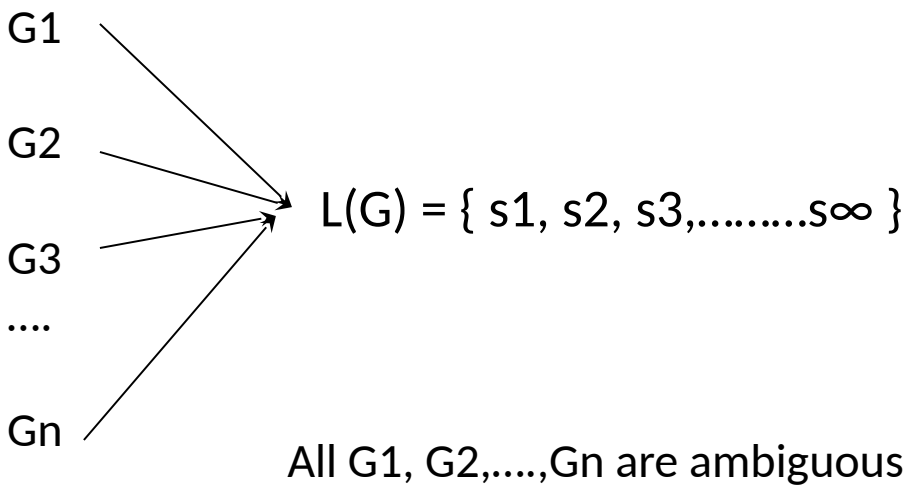




- Two different parse trees => the grammar is ambiguous

(contd.)

- Def: A language for which NO UNAMBIGUOUS grammar exists is called an *inherently ambiguous language*.



- Most Programming languages are ambiguous
- Ex. if-then-else statement (Dangling-Else Problem)

```
<stmt> -> if (<expr>) <stmt> | if (<expr>) <stmt> else <stmt> | <other>
```

Consider the following statement:

```
if (x > y) then
    if ( x >  z ) then
          a = b
else a = c;
```

It can be derived in two different ways, two distinct parse trees

Q: Where does “else” belong?

A: Belongs to the closest “if” => impose an outside rule! or
consider { } to remove ambiguity

Chomsky Normal Form (CNF)

- All productions must have either exactly two nonterminals or only one terminal on the right-hand side
- Not all CFGs can be brought to CNF; the grammar cannot have ε or unit productions
- Algorithm:

For every production A -> $\propto$, and | $\propto$| >1, if $\propto$ contains terminals create new nonterminals for each terminal: if $a \in \Sigma, a \in \propto$ then we introduce a new nonterminal X_a -> a and we replace a by X_a in $\propto$. Let $\propto'$ be the new right-hand side with only nonterminals.

The new set of productions will contain only rules of the form:

A -> $B_1B_2...B_k$

X_a -> a

We have now to break $B_1B_2...B_k$ in groups of two variables:

A -> B_1X_1

X_1 -> B_2X_2

...

X_{k-1}-> $B_{k-1}B_k$

- Most Programming languages are ambiguous
- Ex. if-then-else statement (Dangling-Else Problem)

```
<stmt> → if (<expr>) <stmt> | if (<expr>) <stmt> else <stmt> | <other>
```

Consider the following statement:

```
if (x > y) then
    if ( x >  z ) then
          a = b
else a = c;
```

It can be derived in two different ways, two distinct parse trees

Q: Where does “else” belong?

A: Belongs to the closest “if” => impose an outside rule! or
consider { } to remove ambiguity

Push Down Automaton (PDA)

- Consider the following language $\{a^n b^n \mid n \geq 0\}$.
- If we consider a stack, initially empty (or with a bottom symbol Z_0) and we process the input string in the following way:

```
while (there exists a symbol in the input) {
    read(x);
    if (x==a) push(a);
    else break;
}
while (there exists a symbol in the input) {
    read(x);
    if (x==b) pop();
    else break;
}
if (stack == Z0 and no error from pop() ) accept string
else reject string;
```

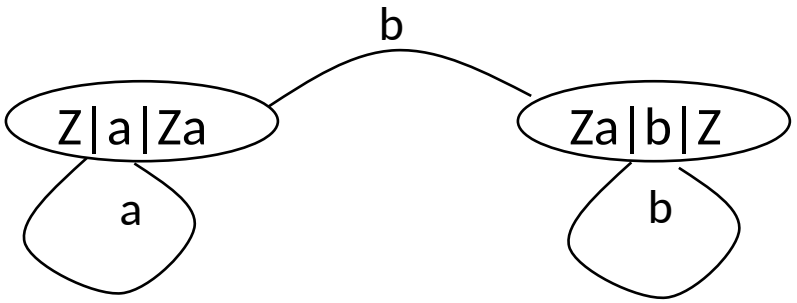
- Graphically the content of the stack for a^3b^3 will look like:

String a^3b^3 is accepted

- Graphically the content of the stack for a^4b^3 will look like:

String is not accepted

- What is important at any step?
  initial stack, the symbol read, and the updated stack



with initially $Z = Z_0$. If $Z_0|b$ we have an error

- We can represent all the transitions in a table:

	Input	Top of Stack current	Top of Stack updated
State 0:	a	Z_0	Z_0 a
State 0:	a	Z	Za
State 0:	b	Za	Z
State 1	b	Z_0a	Z_0
State 1:	b	Z_0	error
State 2:	λ (lambda) or ε	Z_0	accept

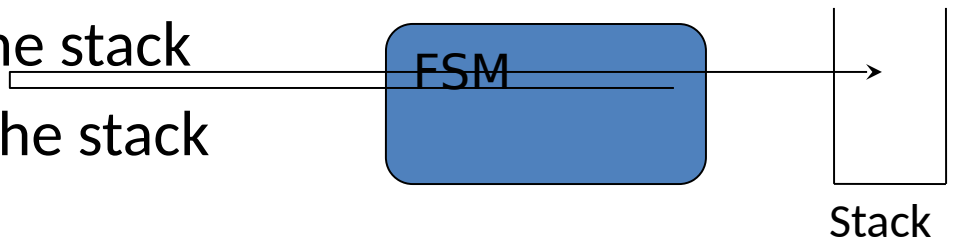
- Or:

	state	Input	Stack symbol (top)	Move
1.	q_0	a	Z_0	(q_0, aZ_0)
2.	q_0	a	A	(q_0, aa)
3.	q_0	b	A	(q_1, λ)
4.	q_1	b	A	(q_1, λ)
5.	q_1	λ (lambda) or ε	Z_0	(q_2, Z_0)
All other combinations				none

New state

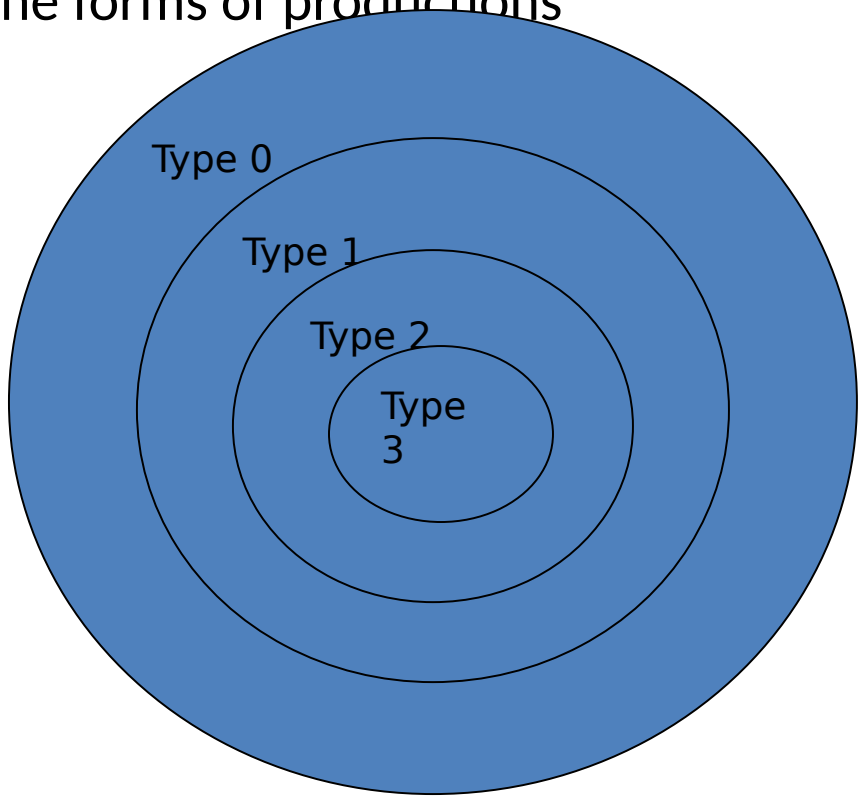
New stack

- For a PDA:
  1. Read a symbol from the input or not
  2. Push a string onto the stack
  3. Pop a symbol from the stack
  4. Change state
  5. Accept if stack is Z_0 and input stream is Ø



3.3 Chomsky Hierarchy of Grammars

- Chomsky notation convention:
  - Capital Letter: A, B …. nonterminals
  - Lowercase letters: a,b,c … terminals
  - Greek Letters: α, β, δ, γ … strings of N and T
- Chomsky Hierarchy shows four different types of grammars (0,1,2 and 3) based on the forms of productions



Type 0: Unrestricted Grammar

Def: Production form: $\alpha \rightarrow \beta$ where both α and β are any string of N and T

Example:

R1) $S \rightarrow ACaB$

R2) $Ca \rightarrow aa\ C$

R3) $CB \rightarrow DB$

R4) $CB \rightarrow E$

R5) $aD \rightarrow D\ a$

R6) $AD \rightarrow AC$

R7) $aE \rightarrow Ea$

R8) $AE \rightarrow \varepsilon$

Example of strings accepted: aa, aaaa, aaaaaaaa etc

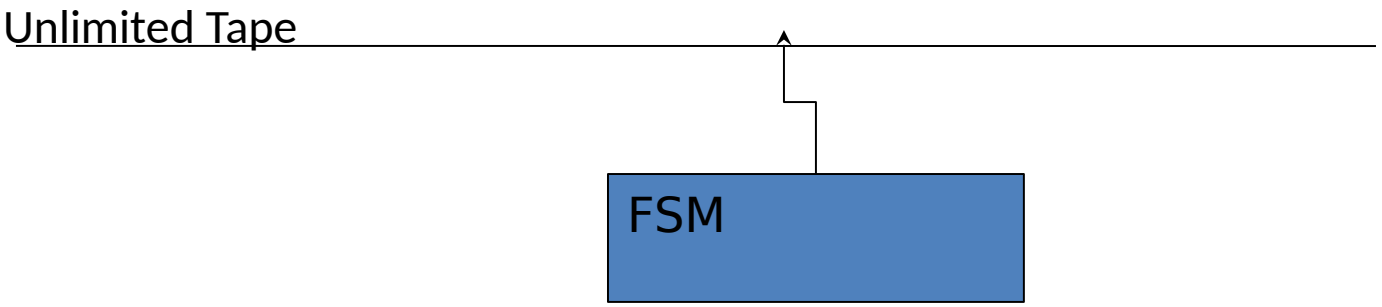
R1 R2 R4 R7 R7 R8

S => A C a B => A aa CB => AaaE => AaEa => AE aa => aa

- Note that L(G) = {w| a^i where i is a positive power of 2}

Machine for Unrestricted languages:

Turing Machine (By Allen Turing 1912-1954)



The Transition function:

N (q, a) → (qi, b ∈∑) /* goto a state and write to the tape */

→ (qi, {L,R}) /* goto a state and move to the write/read */

Type 1: Context Sensitive Grammar

Def: all the productions are of the form $\alpha \rightarrow \beta$ with $|\beta| \geq |\alpha|$ (where α and β are strings of non-terminals and terminals). Each rule transforms at most one nonterminal (A) at a time $\gamma A \delta \rightarrow \gamma \alpha \delta$

Similar to type 0 but

1) α cannot be ε i.e., $\rightarrow |\gamma\alpha\delta| >= |\gamma A\delta|$, RHS >= LHS

2) $A \rightarrow \alpha$ in the context of γ, δ.

Example

R1) $S \rightarrow aSBC$

R2) $S \rightarrow abC$

R3) $CB \rightarrow BC$ // no nonterminal is transformed

R4) $bB \rightarrow bb$

R6) $bC \rightarrow bc$

R6) $cC \rightarrow cc$

Example of strings accepted: abc, aabbcc etc

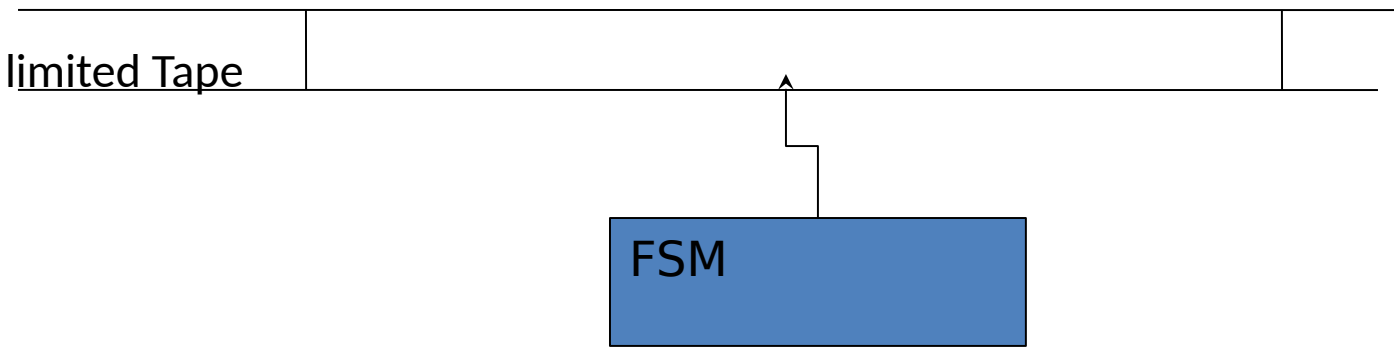
R1 R2 R3 R4 R5 R6

S => aSBC =>aabCBC =>aabBCC =>aabbCC => aabbcC => aabbcc

L(G) = $\{w \mid w = a^n b^n c^n\}$ equal number of a, b and c.

Machine for Context Sensitive languages:

LBA (Lineary Bounded Automaton): Similar to Turing machine but with



The transition function:

N (q, a) → (qi, b ∈∑) /* goto a state and write to the tape */

→ (qi, {L,R}) /* goto a state and move to the write/read */

Type 2: Context Free Grammar

Def: each production is of form: $A \rightarrow \beta$ where A is a single nonterminal

Example

R1) $S \rightarrow a\ B$

R2) $S \rightarrow bA$

R3) $A \rightarrow a$

R4) $A \rightarrow aS$

R5) $A \rightarrow bAA$

R6) $B \rightarrow b$

R7) $B \rightarrow bS$

R8) $B \rightarrow a\ BB$

Example of strings accepted: ab, ba, abab, aaabbb etc

L(G) ={ w | w has same number of a's and b's}

Type 3: Regular Grammar

Def: each production is of the form A → β where A is a single nonterminal and β is either all terminals or at most one nonterminal

(the last symbol on the RHS, first or last symbol)

Example

R1) S → bA

R2) S → aB

R3) S → ε

R4) A → abaS

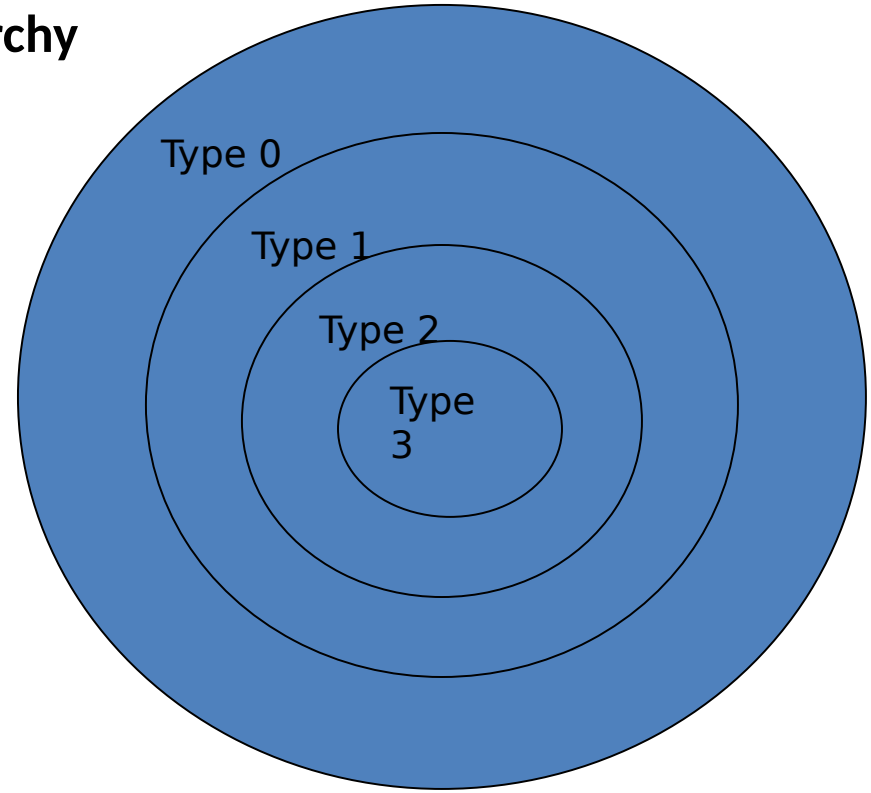
R5) B → babS

Example of accepted strings: abab, baba, abababab, babaabab etc

(abab | baba)* which is an RE

Machine: FSM

Chomsky Hierarchy



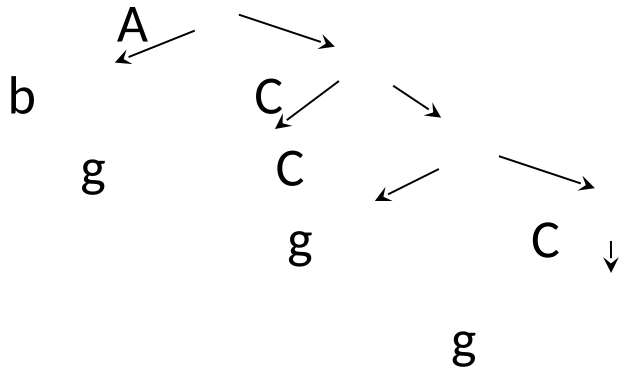
Parsers

- Question: How do we build a parser for CFL?
- Let's take an example and see how a parser works:

  A → Ba | bC

  B → d | eBf

  C → gC | g
- Let's consider the parse tree of a string `bggg`



- We consider two general types of parsers:
  - *Top-down parsers*: starts at the root of the parse tree (which is the starting nonterminal) and tries to reconstruct the growth of the tree that led to given token string as leaves, read from left to right; it doing so, it reconstructs the leftmost derivation
  - *Bottom-up parsers*: starts at the leaves of the parse tree and tries to work backward towards the root; in doing so, it reconstructs the rightmost derivation

- Both types of parsers have nothing but the string of tokens and the rules of the grammar
- They both scan the list of tokens from left to right
- Each token guides the parser in its choice of actions
- When the token is used up, a new token is at hand
- Tokens get “eaten” (used up) when they are matched to terminals in the productions (rules)
- We have linear time parsers for unambiguous grammars and something of the order of $O(n^3)$ for ambiguous ones
- We start with the top-down approach and later we present the bottom up parsing

Top Down Parsers

- A top down parser starts with the starting nonterminal and must determine, from the stream of tokens, how to grow the parse tree towards the leaves that results in the observed string of tokens.
- All it has is the stream of tokens and the rules of the grammar
- The list of tokens is scanned from left to right
- Before we build a top-down parser from the grammar, we have to consider two issues: eliminate left recursion & remove backtracking (do left-factorization)
- Combining elimination of left recursion and doing left factorization for a given CFG gives us a so called LL(k) grammar.
- A grammar is *LL(k)* if looking ahead k symbols in the input, it is enough to choose the next move of the PDA
- For a LL(k) grammar we can build a deterministic PDA
- A LL(1) parser looks ahead one symbol in the input to decide on the next move

3.4 Left-Recursion and Left Factorization (Backtracking)

- Tokens are used up when they are matched to terminals in the grammar
- If the RHS starts with a nonterminal, no token is used up and the parser must decide which rule of the grammar to apply next. This can lead to *left-recursion*.
- If the RHS starts with a terminal and there are several rules starting with the same terminal, the parser must decide which of then to apply. This can lead to b*acktracking*.
- Example: E → E+T
- The parser is presented with the nonterminal E and must decide what to do without using any tokens
- So it may decide to keep applying the rule
  E → E+T infinitely many often
- This creates left-recursion and is called *immediate (direct) left-recursion*

- Or the parser has the rules
  $A \rightarrow B\boldsymbol{\alpha} \mid \ldots$ and $B \rightarrow A\boldsymbol{\beta} \mid \ldots$

And the parser will use $B\boldsymbol{\alpha}$ for A, then $A\boldsymbol{\beta}$ for B, and so forth, and it will end up with an infinite length tree without using up any tokens.

This is called non-immediate or indirect left-recursion and needs to be eliminated.

- The solution is to rewrite the grammar in such a way to eliminate both types of left-recursion, immediate and indirect.
- Problems: left recursion and backtracking
  - Left-recursion: the parser grows the tree on a branch unnecessarily, because nothing in the grammar and in the input prevents it from doing so
  - Backtracking: the parser chooses the wrong rule to grow the tree because there are at least two rules that start the same way
- No top-down parser can handle left-recursion

Left Factorization (Backtracking)

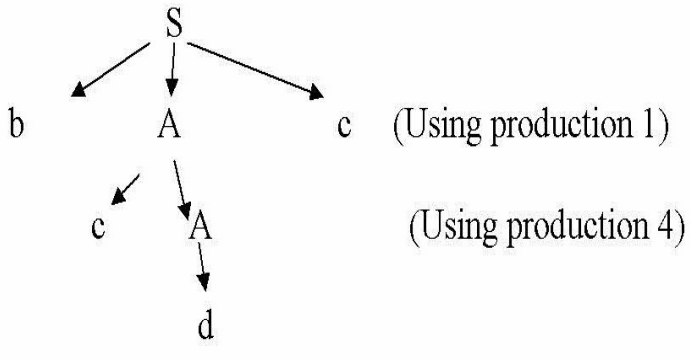
- *Backtracking* is reparsing of the same/previous tokens
- Example: Assuming the following productions:

  S → b A c

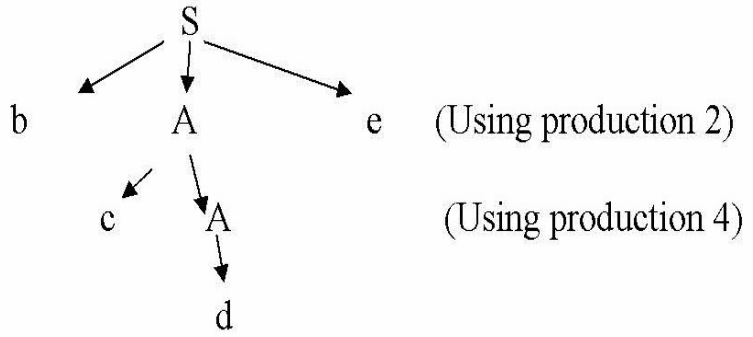
  S → b A e

  A → d

  A → c A
- and the following string `bcde`



Does not work.
String wrong?
=>
Consider other
Possibilities



Works!!

- How to solve backtracking?
- Use *left-factorization*: factor out same symbols of RHSs of the productions for the same nonterminal
- Example:

    R1) S → b A c

    R2) S → b A e

    R3) A → d

    R4) A → c A

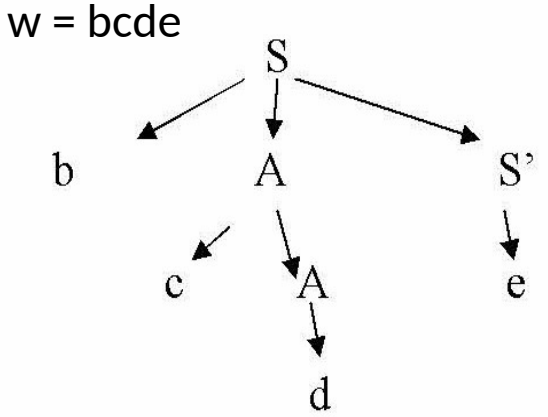
=>

R1) S → b A S’ (Factor out bA from R1 and R2)

R2) S’ → c | e

R3) A → d

R4) A → c A

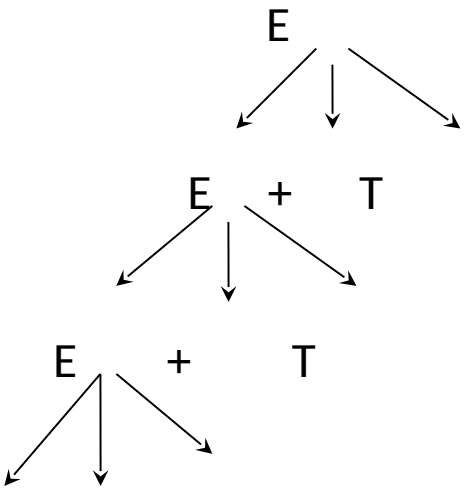


Left Recursion

- Example: E → E + T
- Why?
- Let's consider the following rules:

```
R1) E → E + T
R2) E → T
R3) T → id
```

Given string a + b + c , the parser tries to match the left-most symbol



Etc.

=> the parser does not know when to stop expanding the tree by continuously applying the production E → E +T since all it sees is the terminal `a`

Eliminating Immediate Left Recursion

We need to change the productions as follows:

- Assume we have the following productions:

  $A \rightarrow A\alpha$

  $A \rightarrow \delta$

- Steps:

1. Introduce a new nonterminal A'
2. Change $A \rightarrow \delta A'$
3. $A' \rightarrow \alpha A' \mid \varepsilon$

Without ε the rule A' will not terminate

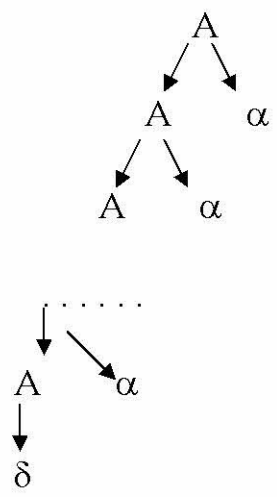
=> so the production rules become

$A \rightarrow \delta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

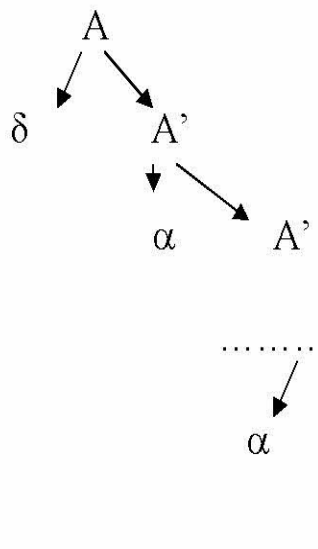
Let's see what we have done

$A \rightarrow A\alpha$
$A \rightarrow \delta$



$\delta\alpha\alpha\alpha\ldots.\alpha$ is the string

$A \rightarrow \delta\ A'$
$A' \rightarrow \alpha\ A'\ |\ \varepsilon$



Strings = $\delta\alpha\alpha\ldots..\alpha$

Note: We have two parser trees with same strings but the shapes are different.

Example of removing left recursion

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow id$

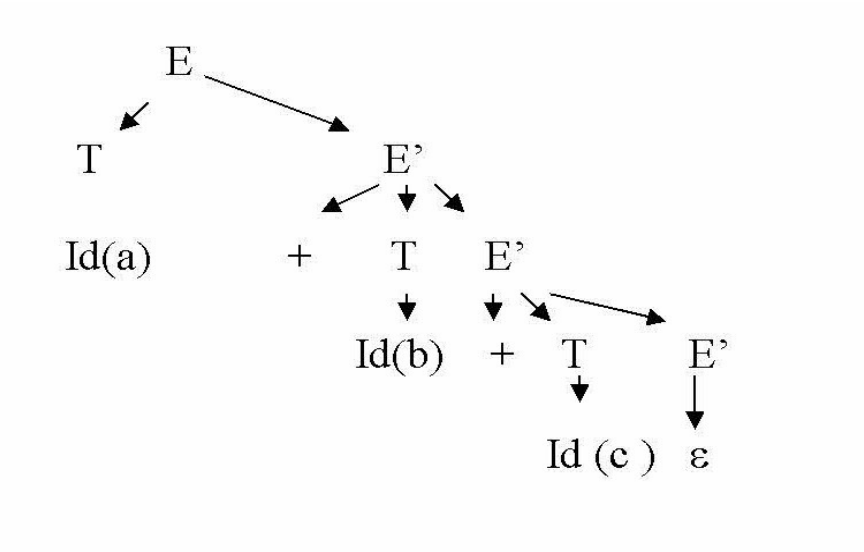
Remove left recursion =>

$E \rightarrow T\ E'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow id$

Let's further consider the string a + b + c



But there can be “indirect (non-immediate)” left-recursion

- Example
  A → B C
  B → A C
- Thus, we need an algorithm to remove all left-recursions:

Method (based on Aho, Sethi, and Ullman):

0. Get rid of any direct left recursion
1. List all non-terminals in the order they occur in the left-hand side of the rules
2. For each nonterminal N do
   If RHS begins with a nonterminal A earlier in the list (e.g., N → Aα) then look at the rules that have A in the LHS (e.g. A → γ_1| γ_2|…..)
   - Substitute A in the rule of N as follows: N → γ_1 α| γ_2 α |…..)
   - Remove any direct left recursion

Example A → N | β
…
N → Aγ
Then replace A in the rule N → A γ with the right hand side of all the A-rules
N → Nγ | βγ
After that, we need to remove direct left-recursion from the N-rules.

- Example

R1) E → E + T R2) E → T R3) T → E R4) T → id

- Method:

1) List of nonterminals => 1. E, 2. T

2) Direct left-recursion in (R1) and (R2)

E → TE’

E’ → + TE’ | ε

3) Need to replace E in (R3)

T → TE’

4) Here is left-recursion

T → id T’

T’ → E’T’ | ε

Therefore:

R1) E → TE’

R2) E’ → +TE’ | ε

R3) T → id T’

R4) T’ → E’T’ | ε

=> E, T, E’, T’

Another Round…

CONCLUSION:

1) **We need to eliminate Left-Recursion**
2) **Do left factorization (to remove backtracking)**

before
constructing the Top-Down parsers

- Combining elimination of left recursion and doing left factorization for a given CFG may give us a so called LL(k) grammar.
- A grammar is *LL(k)* if looking ahead k symbols in the input, it is enough to choose the next move of the PDA
- For a LL(k) grammar we can build a deterministic PDA
- A LL(1) parser looks ahead one symbol in the input to decide on the next move
- Note that eliminating left recursion and common prefixes does NOT make a grammar LL

3.5 Top-Down Parsers

- Recursive Descent Parser (RDP)
- The basic idea is that each <u>non-terminal has an associated parsing procedure</u> that can recognize any sequence of tokens generated by that non-terminal.
- For example:

1. $E \rightarrow E + T$
2. $E \rightarrow E - T$
3. $E \rightarrow T$
4. $T \rightarrow id$

1. Remove Left-Recursion

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid - TE' \mid \varepsilon$
$T \rightarrow id$

Procedure E ()

```
{
  T ();
   E'();
   If not eof marker then
      error-message
}
```

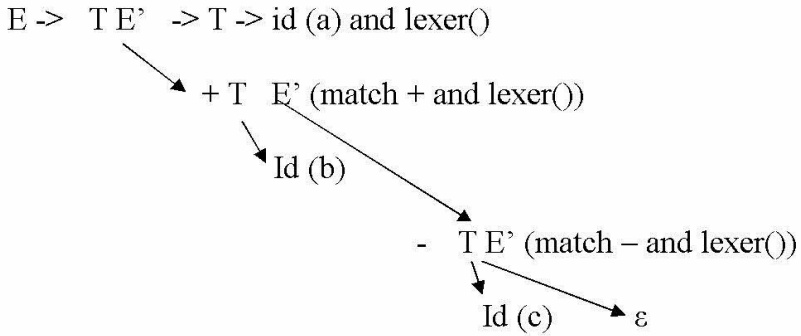
Procedure E'()

```
{
If token = + or – then
        {
        Lexer();
       T();
        E'();
        }
}
```

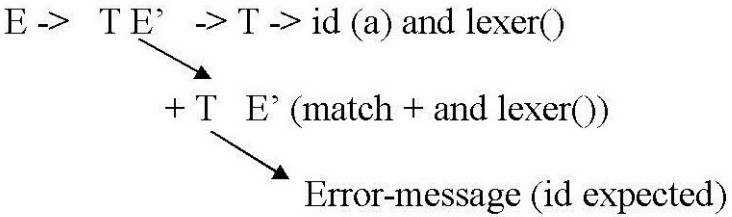
Procedure T();

```
{
   If token is id then
        lexer()
    else error-message
    (id expected)
}
```

Ex 1) Assume we have a string a+b-c



Ex 2) String a +



Parsing

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for 'Left-to-right, Leftmost derivation'.
- LR stands for 'Left-to-right, Rightmost derivation’

Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

LL Parsing

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are
  (1) match a terminal
  (2) predict a production
  (3) announce a syntax error

LL Parsing

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - what you *predict* you will see

LL Parsing

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:
  - (1) compute FIRST sets for symbols
  - (2) compute FOLLOW sets for non-terminals (this requires computing FIRST sets for some *strings*)
  - (3) compute predict sets or table for all productions

LL Parsing

- Algorithm First/Follow/Predict:
  - FIRST(α) == {a : α →* a β}
    ∪ (if α =>* ε THEN {ε} ELSE NULL)
  - FOLLOW(A) == {a : S →+ α A a β}
    ∪ (if S →* α A THEN {ε} ELSE NULL)
  - Predict (A → X_1 ... X_m) == (FIRST (X_1 ... X_m) - {ε}) ∪ (if X_1, ..., X_m →* ε then FOLLOW (A) ELSE NULL)
- Details following…

Motivation for First and Follow sets

- The removal of immediate left recursion entails the introduction of productions of the form $A \rightarrow \varepsilon$
- If the starting nonterminal has rules and none of them begin with a terminal, i.e. they all begin with nonterminals, then the parser does not know which rule to apply next
- It can uses a brute force approach and try all rules
- But there is a smart way to make a decision: use First and Follow sets
  - First sets tell us which rule(s) to select based on what is the first token in the list
  - Follow sets tell us what set of rules to apply in case the First set contains ε

Predictive Parsers

- Given a CFG with the starting nonterminal S and an input string w, we need to find out if S =>* w
- Given any nonterminal A in the CFG (including S), we call the rules of the CFG that have A in the left-hand side as *A-rules*
- We look in the grammar for the S-rules for which the righthand side starts with the terminal w1, which is the first symbol of w
- Since backtracking was eliminated, we can have at most one such rule
- If we do not find such a rule, then we look at the S-rules that start with non-terminals
- What happens if we have two or more S-rules that start with non-terminal? Which one to choose?
- Consider the CFG:

S → Ab | Bc

A → Df | CA

B → gA | e

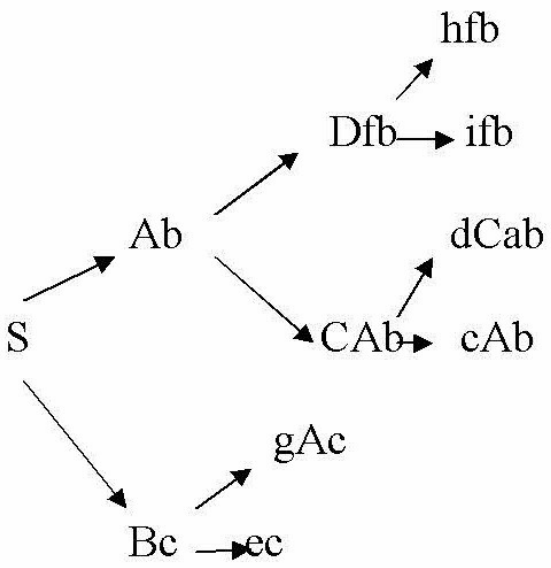
C → dC | c

D → h | i

- And the string w = gchfc
- We have two choices for S, two choices for A, and two choices for B
- Can we narrow down the choices?

Yes: anticipate what terminal symbols are derivable from each nonterminal symbol on the RHS of productions.

Example: Let's compute before we construct the parser.



if the parser goes (Ab) =>
    it will encounter terminals {h, i, d,c}
else (BC) => {g, e}

So, before parsing look ahead of the sets which way to go:
Ex. string "gchfc" => "g" is in (BC) route, so no need to go to the first route.
<u>These sets are called First sets.</u>
First (Ab) = { h, i, d, c}
First (Bc) = {g, e}

If the first character is {h,I,d,c} then choose the rule S -> Ab
If the first character is {g,e} then choose the rule S -> Bc
If the first character is neither, then stop with error.

- Def: First (α) Consider every string derivable from α by a left most derivation. If α => β where β begins with some terminal, then that terminal is in First (α).
- Computation of First (α)

1) if α begins with a terminal t , then First(α) = t

2) If α begins with a nonterminal A, then First (α) includes First(A) – ε

- and if A → ε then include First (γ) where α = Aγ
- and if α => ε, then First(α) includes ε

3) First (ε) = ε

- Example:

$E \rightarrow TE'$

$E' \rightarrow +\ TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *\ FT' \mid \varepsilon$

$F \rightarrow id \mid (E)$

First (F) = First (id) U First ((E)) = { id, (}

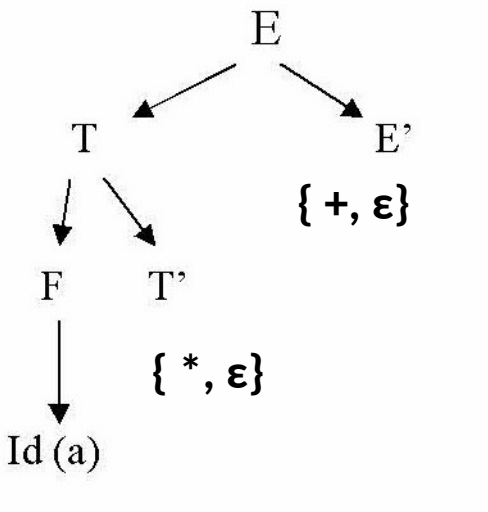
First (T') = First (*FT') | First (ε) = { *, ε }

First (T) = First (FT') = First (F) - ε = {id, (}

First (E') = First (+TE') U First (ε) = { +, ε}

First (E) = First (TE') = First (T) = First (id) U First ((E)) = { id, (}

However, there is a problem using just the First sets.
Consider the previous arithmetic grammar example and a string " a +b"



Next token = + but + is not in First (T) = { *, ε}
Does that mean it is wrong?
NO, because T => ε

In that case, we have to consider what can follow after T = { + } => acceptable tokens

So, because of the ε, we need to consider also Follow (N = nonterminal) = { terminals that can follow right after N}

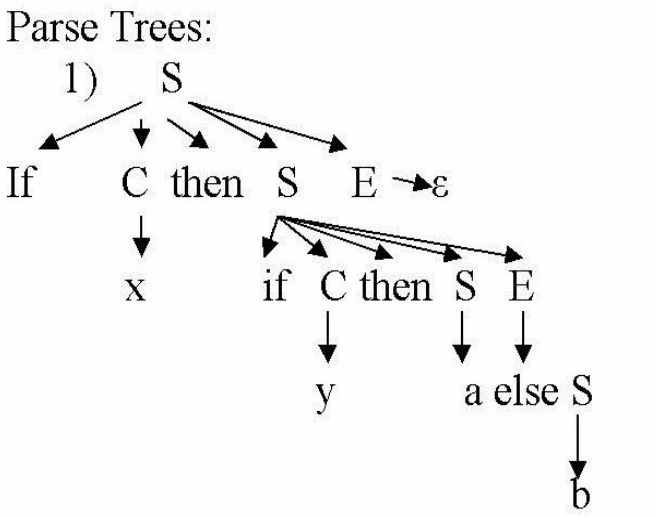
If a nonterminal is *nullable* (e.g. A => **ε**) then looking at the First sets in not enough, but we need to look at the terminals that follow A in order to choose the appropriate rule.

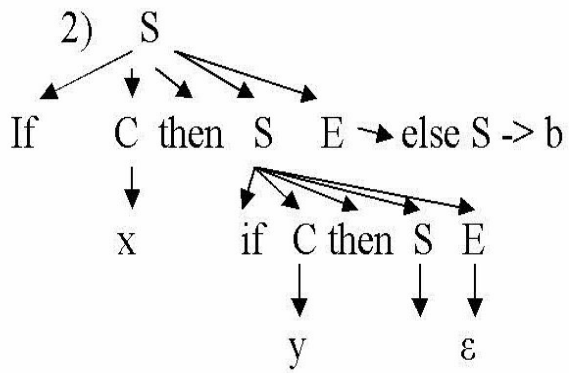
Why Conflict?

a sample sentence:

If x then
If y then a
else b

(where does this else belong????)





Grammar is Ambiguous

=> If a grammar is ambiguous, it will create a conflict

Motivation for Follow sets

- The removal of immediate left recursion entails the introduction of productions of the form $A \rightarrow \boldsymbol{\varepsilon}$
- First sets will not tell us when to choose $A \rightarrow \boldsymbol{\varepsilon}$ which translates into pushing a nonterminal on the stack without reading any symbol from the input (i.e. without processing any token from the input)
- To handle this, we need Follow sets
- We assume that the token string has an end marker appended to it, $

- Def: Follow (A) is the set of all terminal symbols that can come right after A in any sentential form of L(G). If A comes at the end of, the Follow(A) includes "$" = end of file marker
- Computation:

IF A is the starting symbol, then include $ in Follow(A)

For all occurrences of A on the RHS of productions do as follows:

Let $Q \rightarrow \alpha A \beta$ (means α before A and β after A), then

If β begins with a terminal t, then t is in Follow(A)

If β begins with a nonterminal , then include First (β) – ε

If $\beta =>^* \varepsilon$ or $\beta = \varepsilon$, then include Follow (Q) in Follow(A)

(** we ignore the case Q = A, e.g, $A \rightarrow \alpha A \beta$)

- Example:

  $E \rightarrow TE'$

  $E' \rightarrow +\ TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *\ FT' \mid \varepsilon$

  $F \rightarrow id \mid (E)$

Follow (E) = {$} U {) } = { $,)}

Follow (E') = Follow (E) U Follow (E') (ignore) = { $,)}

Follow (T) = First (E') – ε U Follow (E) U Follow (E') = {+, $,)}

Follow (T') = Follow (T) U Follow (T') (ignore) = {+, $,)}

Follow (F) = First (T') – ε U Follow (T) U First (T') - Follow (T')

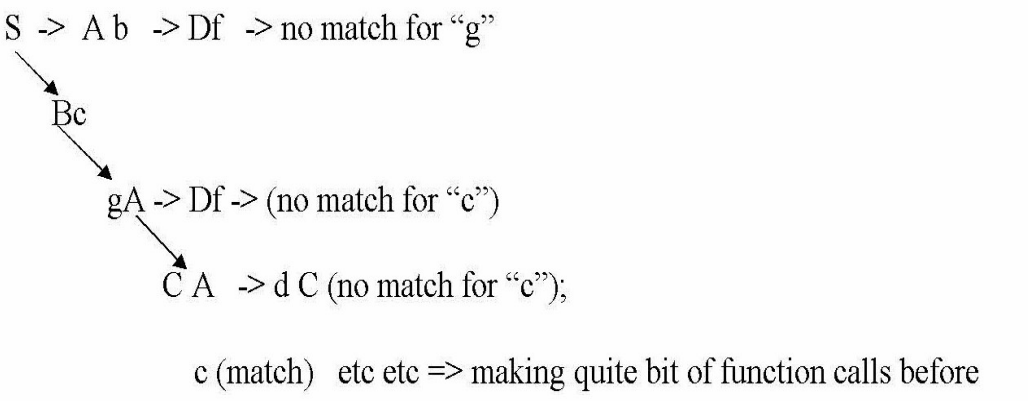
= {*} U {+ $,)} U {*} U {+, $,)} = { *, +, $,)}

Predictive Recursive Descent Parser (PRDP)

- A more efficient way of implementing RDP.
- Assume we have the following productions (no left-recursion or backtracking)

1. S → Ab | B c
2. A → Df | CA
3. B → gA | e
4. C → dC | c
5. D → h | i

and a string "gchfc" and parse



making quite bit of function calls before matching
=> a better way? A more efficient way?

- FOLLOW sets tells us when to use ε productions
- Suppose that the nonterminal A is at the top of the stack and needs to be expanded
- We seek if the next token is in the FIRST set for some RHS rule of A
- If not, this normally means an error
- But if A => ε which means that ε is in First(A) then we check whether the token is in Follow(A)
- If it is, then it may not be an error and the rule used will be A $\rightarrow \varepsilon$. We expand A $\rightarrow \varepsilon$ by doing nothing and returning

Predictive RDP with First and Follow Sets :

```
Procedure E ()
{
If token in First (E) then
    T();
    E'();
else error-message (token in First
     of (E) expected)
}
```

```
Procedure T()
{
   If token in First (T) then
      F();
      T'();
  else error-message (……)
}
```

```
Procedure E' ()
{
   If token = + then
     Lexer();
     T();
     E'();
  else if token not in Follow E' then
        error-message (……)
}
```

```
Procedure T'()
{
   If token = '*' then
      Lexer();
      F();
      T'();
  else if token NOT in folllow (T') then
       error-message (……..)
}
```

```
Procedure F() {
…… same ….}
```

Table Driven Predictive Parser

- A nonrecursive form that uses a table to decide which rule or rules can be applied at each step
  - Can be constructed by hand for small grammars or computed for large grammars
  - Tells us which RHS to use for a nonterminal and if tokens need to be used up
- Blanks are error conditions
- Each row corresponds to a nonterminal; each column to a terminal
- The first row corresponds to the starting symbol
- For each RHS, add the nonterminal in right-to-left manner; this is due to the way the pushing and popping for the stack (we will see later)
- Example : the expression grammar

```
E → TE′
E′ → +TE′     | ε
T → FT′
T′ → * FT′    | ε
F →   (E)   |  id
```

	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	id			(E)		

- Consist of three components: parsing table, stack, program driver
- Table: to generate a table with all terminals (columns) and nonterminals (rows)
- The program driver uses its own stack; the rule is described on next slide
- Tokens in the input are processed left to right
- If the top of the stack is a terminal:
  - If it marches the token in the input, then the top of the stack is popped and the input token is used up
  - If it does not match, then error
- When the top of stack is a nonterminal A, a rule with that nonterminal in RHS must be used
  - The table will tell us with rule to choose, based on the token `t` in the input
  - If the entry is empty, then error
- Once a rule is found in Table[A,t] then we push the RHS in reverse order such that its leftmost symbol will be at the top of the stack

•Stack => well known with pop(), push(), etc

•Driver:

Push $ onto the stack
Put end-of-file marker ($) at the end of the input string
Push (Starting Nonterminal) on to the stack
While stack not empty do
    let t = top of stack symbol and i=incoming token
    if t = terminal symbol then
        if t=i then
            pop(t);
            lexer(); .// go to the next token
        else error-message (…._
    else begin
        if Table [t, i] has entry then
            pop(t);
            push Table[t, i] in reverse order
        else error
        end
endwhile

	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	id			(E)		

Ex: String b + c

Stack	***Input***	***Action***
$ E	**b+c$**	**pop(E), Push (E’, T)**
$E’T	**b+c$**	**pop(T), push(T’, F)**
$E’T’F	**b+c$**	**pop(F), push (id);**
$E’T’ id	**b+c$**	**pop(id), lexer();**
$E’T’	**+c$**	**pop(T’); push (ε)**
$E’	**+c$**	**pop(E’); push (E’, T, +)**
$E’T+	**+c$**	**pop(+), lexer()**
$E’T	**c$**	**pop(T), push (T’,F)**
$E’T’F	**c$**	**pop(F); push (id);**
$E’T’id	**c$**	**pop(id), lexer()**
$E’T’	**$**	**pop(T’), push (ε)**
$E’	**$**	**pop(E’), push (ε)**
$	**$**	**Stack empty**

How to Build a Predictive Parser Table

- We need the FIRST and FOLLOW sets
- In a top down parser, if there is no backtracking, the next incoming token will tell us what to do next
- We start with the starting nonterminal on top of the stack
- If the top of the stack and the input token are the same (both terminals, also) then we pop them
- But if the top of the stack is a nonterminal X and the input token is `a`, we need to decide which rule to apply
- We want to select a rule that either has `a` as the first symbol in RHS
- Can lead to a sentential form beginning with a
- So we want to select a rule A → **α** in which $a \in FIRST(\alpha)$
- But if $FIRST(\alpha)$ contains **ε**, since the table does not contain a column with **ε**, we use the Follow(A) to populate the table with **ε**

- To construct such a table:

for each nonterminal N do

{

Let N → β a typical production

Compute First (β);

Each terminal t in First (β) except ε do

Table [N, t] = β

if First (β) has ε then

for each terminal t in Follow (N) do

Table [N, t] = ε

}

Example:

E → TE'
E' → +TE' | ε
T → FT'
T' → * FT' | ε
F → (E) | id

Computer First Sets:

First (F) = {(, id}
First (T') = (*, ε)
First (T) = First (F) – ε = {(, id}
First(E') = {+, ε}
First (E) = First(T) – ε = {(, id}

If a FIRST set contains ε, then compute FOLLOW sets:

Follow (T') = { +, $,)}
Follow (E') = {), $}

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

Conflict:

In a table driven predictive parser, a conflict occurs if a cell Table [N, t] has more than one entry.

Ex. "Dangling else" problem

1) S → if C then S E | a | b
2) C → x | y
3) E → else S | ε

Let's try to construct the table in particular for E

First (E) = {else, ε}

Since it contains ε

We need to computer Follow (E) = Follow (S) =

= { $} U First (E) – ε U Follow(E)

= {S} U { else} = {$, else}

							else	$
S								
C								
E							else S ε	ε

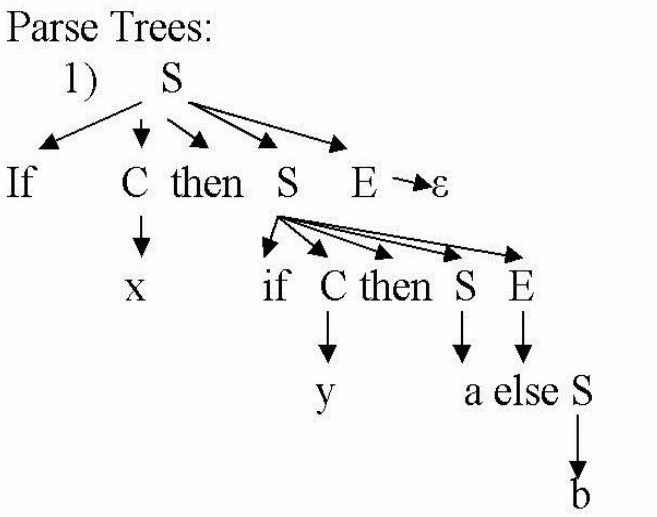
Two entries in Table [E, else] => conflict

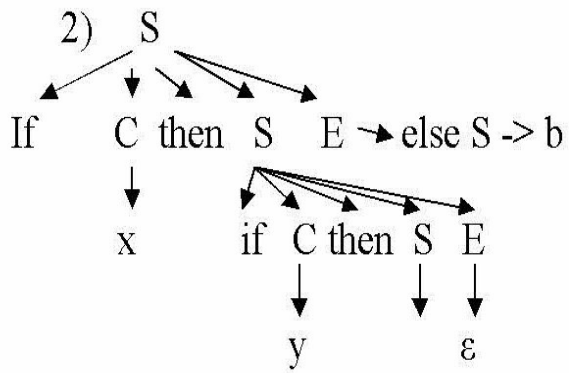
Why Conflict?

a sample sentence:

If x then
If y then a
else b

(where does this else belong????)





Grammar is Ambiguous

=> If a grammar is ambiguous, it will create a conflict