THEORY OF COMPUTATION
UNIT 2

Learning Objective

- Explain the concepts of Context free grammar and their closure properties.
- Explain the concept of parse tree and ambiguity of grammar.
- Understand Normal forms i.e. CNF and BNF.
- Concept of PDA and Relationship between CFG and PDA.
- Discuss the concept of parsing and types of parser.

Introduction

Context-Free Grammar

- A Grammar G=(V, T, P, S) is said to be context free where

  V=A finite set of Non-terminal , generally represented by capital letters.

  T=A finite set of terminals, generally represented by small letters.

  P= Set of production in CFG

  S= Starting Variable or Start Symbol

All productions in CFG are in following form:

$$\alpha \rightarrow \beta \text{ Where } \alpha \in V \text{ and } \beta \in (V+T)^*$$

Context-Free Grammar

- Derivation : A derivation of a string for a grammar is a sequence of grammar rule applications hat transforms the start symbol into a string.
- Symbolically we represent the derivation as $S \overset{*}{\underset{G}{-}}> w$ which means that the string w is derived from root S.
- A derivation proves that the string belongs to the grammar's language.
- Example: Consider the following grammar

  $S \rightarrow S + S$

  $S \rightarrow 1$

  $S \rightarrow a$

Context-Free Grammar

- Let us derive the string "1+1+a"
- S $\rightarrow$ S+S (rule 1 on first S)

  $\rightarrow$ S+S+S (rule 1 on second S)

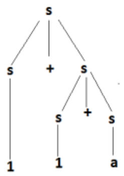
  $\rightarrow$ S+1+S (rule 2 on second S)

  $\rightarrow$ S+1+a (rule 3 on third S)

  $\rightarrow$ 1+1+a (rule 2 on first S)
- It is useful to derivation as a tree hence called "*derivation tree*" or "*parse tree*".
- The derivation tree is a pictorial representation of derivation of string from the grammar.

Context-Free Grammar

- The Derivation Tree of "1+1+a"



- Formally a tree is a derivation tree for G if:
  - ✓Every vertex has a level of V U T U {λ}
  - ✓The level of root is S.
  - ✓All the leaf nodes are terminals

Context-Free Grammar

- Now the question is "What would be substitution order of the variables?"
- There are two possibilities.
  - ✓Left Most Derivation
  - ✓Right Most Derivation
- Left most derivation: always substitutes the left most variable at each step.
- Right most derivation: always substitutes the right most variable at each step.

Context-Free Grammar

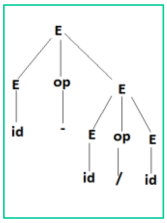
- If *w* is in L(G) for CFG G, then w has at least one parse tree.
- A string *w* may have several left most derivation trees or right most derivation trees.
- A context-free grammar G such that some word has two parse tree is said to be ambiguous.
- A grammar is ambiguous when the same variable appears twice on a right hand side.
- For example, if we have the grammar

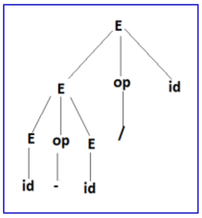
  E->E op E | id

  op-> - | /

Context-Free Grammar

- The sentence id –id / id has two different parse tree.





- However both are left most derivation trees but we can see that for the string we have two parse trees. Hence the given grammar is ambiguous.

Context-Free Grammar

- Problems due to ambiguity in context-free grammars
  - ✓ Problem to decide associativity
  - ✓ Problem to decide precedence
  - ✓ Problem to determine which one should be preferred to other.
- Weather a given grammar is ambiguous or not is undecidable problem.
- A context-free language for which every CFG is ambiguous is said to be an inherently ambiguous CFL.
- For example L= $\{a^n b^n c^m\} \cup \{a^n b^m c^m\}$ is an inherently ambiguous context-free language.

Push Down Automata

A Push Down Automata can be defined by seven tuple

$$\{Q, \Sigma, \Gamma, \delta, q_0, Z, F\}$$

Where

Q = A finite non-empty set of state

Σ = A finite non-empty set of input symbol

Γ = A finite non-empty set of push down symbol

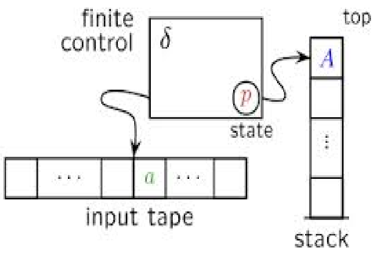
δ is a transition function mapping $\mathbf{Q x(\Sigma \cup \{\lambda\}) x \Gamma \rightarrow Q x \Gamma^*}$ in deterministic case whereas in non-deterministic case it is $\mathbf{Q\ x\ (\Sigma \cup \{\lambda\})\ x\ (\Gamma \cup \lambda) \rightarrow 2^{(Q\ x (\Gamma \cup \lambda)}}$

$\mathbf{q_0}$ is a special state called initial state

Z is a set of push down symbol

F = Set of final states where $F \subseteq Q$

Push Down Automata



Model of PDA

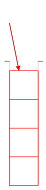
Push Down Automata

- PDA= DFA + Memory (Stack)

a	l	p	h	a	b	e	t

The tape is divided into finitely many cells. Each cell contains a symbol in an alphabet Σ.

Push Down Automata



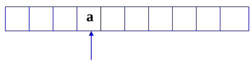
The stack head always scans the top symbol of the stack. It performs two basic operations:

Push: add a new symbol at the top.

Pop: read and remove the top symbol.

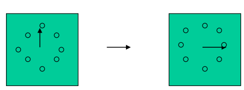
Alphabet of stack symbols: Γ

Push Down Automata



- The head scans at a cell on the tape and can *read* a symbol on the cell. In each move, the head can move to the right cell.

Push Down Automata

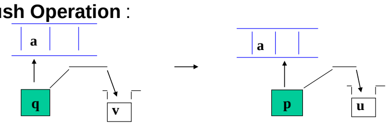


- The finite control has finitely many states which form a set Q. For each move, the state is changed according to the evaluation of a *transition function*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^{Q \times (\Gamma \cup \{\varepsilon\})}.$$

Push Down Automata

- Operation on Push Down Automata
- As we use stack in PDA we only can perform following operations
  - **Push Operation** :

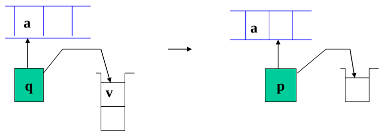


  - On state q tape head reads "a" and stack head reads "v" then it moves to state "p" and "v" is replaced by "u".
  - Note that push down symbols may be different from tape symbol.
  - By transition function we show this operation as

$$\delta(q, a, v)=(p, u)$$

Push Down Automata

- POP Operation :

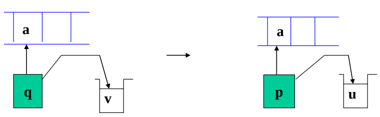


- On state "q" tape head reads "a" and stack head reads "v" then PDA moves to state "p" pop operation is performed and "v" is removed.
- By transition function we show this operation by

  $\delta(q, a, v) = (p, \varepsilon)$

Push Down Automata

- ***ε* move Operation** :



- On state "q", tape head reads "a" and stack head reads "v" then it moves to state "p" and "v" is replaced by "u".
- Note that tape head does not move
- By transition function we show this operation as

  $\delta(q, \varepsilon, v) = (p, u)$

Acceptance of String in Push Down Automata

- We define the string accepted by PDA in two ways :

  ✓ Acceptance by Final State
  - *A string "w" is accepted by a finite state iff* $(q_0, w, z_0) \vdash^* (p, \varepsilon, \gamma)$ *where* $p \in F$, $\gamma \in \Gamma$.
  - *We can define L(M) as* $L(M) = \{w \mid (q_0, w, z_0) \vdash^* (p, \varepsilon, \gamma)\}$

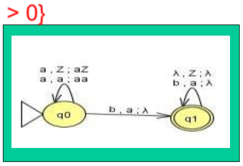
  ✓ Acceptance by Empty Stack
  - *A string "w" is accepted by empty stack iff* $(q0, w, z0) \vdash^* (p, \varepsilon, \varepsilon)$ *for some "p" in Q.*

Designing of PDA

Example 1. Construct PDA to accept
$L = \{0^n 1^n \mid n > 0\}$

Solution

$\partial(q_0,a,Z)=(q, aZ)$
$\partial(q_0,a,a)=(q_1, aa)$
$\partial(q_0,b,a)=(q_1, \lambda)$
$\partial(q_1,b,a)=(q_1, \lambda)$
$\partial(q_1, \lambda,Z)=(q_1, \lambda)$



Designing of PDA

Exercise Problems

- Design PDA for the following
  - $L = \{ a^n b^{2n} \mid n>=1\}$
  - $L = \{wcw^R \mid w \in \Sigma^*\}$
  - $L = \{ww^R \mid w \in \Sigma^*\}$
  - $L=\{a^3 b^n c^n \mid n >=1\}$
  - $L=\{a^n b^n c^m d^m \mid n, m >=1\}$
  - $L=\{a^n b^m \mid n>m\}$

Instantaneous Description of PDA

- A configuration of PDA at a given instance is called *instantaneous description or ID*.
- An ID is defined to be a member of $Q \times \Sigma^* \times \Gamma^*$ such that the first component is the state of machine, the second is a input symbol yet to be read and third is the contents of stack.
- More precisely we can say that An ID is 3 tuple (q, w, z) where q= current state, w=string to be read, z= content of stack
- We can define it as $(p, ax, z\alpha) \vdash (q, x, \gamma a)$

PDA To CFG Conversion

- Each Context Free Grammar is accepted by PDA.
- Let grammar G is accepted by PDA M where S is the initial symbol and is accepted by empty stack.
- We construct the transition function by following rules
  1. $\delta(q, \varepsilon, A) = \{(q, a_1), (q, a_2), \ldots\ldots, (q, a_n)\}$ where $\forall A \in V$ and $a_i \in (V \cup \Sigma^*)$ for $i=1,2,3,\ldots\ldots n$ such that $A \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n$
  2. $\delta(q, a, a) = (q, \varepsilon)$ where $\forall a \in \Sigma$
- For example Let us take following grammar

  S->aSb

  S->bSb

  S->ε

PDA To CFG Conversion

- Now construct the transition

$$\delta(q, \varepsilon, S) = \{(q, aSa)\} \ldots\ldots\ldots\ldots\ldots\ldots.. Using\ rule\ 1$$
$$\delta(q, \varepsilon, S) = \{(q, bSb)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots Using\ rule\ 1$$
$$\delta(q, \varepsilon, S) = \{(q, \varepsilon)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots. Using\ rule\ 1$$
$$\delta(q, a, a) = \{(q, \varepsilon)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots. Using\ rule\ 2$$
$$\delta(q, b, b) = \{(q, \varepsilon)\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots. Using\ rule\ 2$$

Exercise : Construct the PDA for following grammar

S -> ε

S -> SS

S -> (S)

CFG To PDA Conversion

- If M is a pushdown automata which recognizes the language L then there exists a context-free grammar G such that L(G) = N(M)
- We construct context-free grammar by following rules
- Let us consider Q= {p, q} and S is the start symbol
  1. Write the following production rules for start symbol

     S-> [p, z, p]

     S-> [p, z, q]
  2. For transition $\delta(q, a, z) = (p, \varepsilon)$ [Pop operation], write following production rule

     [q, z, p] -> a

CFG To PDA Conversion

3. For transition $\delta(q, a, z) = (p, z)$ [No operation in stack] , write the following production rules..

   [q , z, q] -> a [p, a, q]
   [q, z, p] -> a [p, a, p]

4. For transition $\delta(p, a, a) = (p, aa)$ [Push operation], write following production rule

   [p, a, p] -> a [p, a, p] [p, a, p]
   [p, a, p] -> a [p, a, q][q, a, p]
   [p, a, q] -> a [p, a, p][p, a, q]
   [p, a, q] -> a [p, a, q][q, a, q]

Simplification of context-free grammar

- One can restrict the format of production without reducing the generating power of context free grammar.
- If L is a non-empty context-free language then it can be generated by context-free grammar G with the following properties:
  - Each variable and each terminal of G appears in the derivation of some word in L.
  - There is no production of the form A->B where A and B are variables.
- Means that if ε is not in L then there is no need of production of the form $A \to \varepsilon$

Simplification of context-free grammar

- Summarily, if ε is not in L then we can restrict the grammar by following operations.
  - Removing useless symbol
  - Removing Unit production
  - Removing lambda (ε) or Null production
- Removing useless symbol :
  - A symbol X is a useful symbol if a word can be derived from it or it takes part in the derivation of a word, otherwise X is useless.
  - We can remove useless symbol by following rule
    - ✓Remove the production which is not reachable from start symbol. For this we can construct connectivity graph.
    - ✓Identify non-generating symbol.

Simplification of context-free grammar

- Algorithm for non-generating useless symbol

    begin

        oldv = ϕ

        newv = { A | A->w for some w in T*}

        while oldv ≠ newv

        begin

            oldv := newv

            newv := oldv ∪ { A | A-> α for α in (T∪ oldv)*

        end;

        V' := newv

    end

Simplification of context-free grammar

- Removing Unit production
    - A production of the form A->B means *single variable -> single variable*
    - To remove unit production of the form A -> B , just substitute B by its RHS value.
    - For example, if we have the grammar like *A->B, B->b* then just substitute the B by b and the grammar will be *A->b*.
- Removing lambda (ε) or Null production

Simplification of context-free grammar

- A production of the form $A \rightarrow \varepsilon$ is called ε-production
- In a given CFG, a non-terminal or variable A is said *Nullable* if there is a production $A \rightarrow \varepsilon$ or A -*> ε.
- To eliminate the ε-production we use following technique
    - For each production in CFG which contains "A" in RHS, substitute "A" with ε and add the resulting string into the production

        ✓For example : Let's take a CFG

            S-> aA ……….. P1

            A->b | ε ……… P2

        In P1, substitute "A" with ε we get the new string "a ε"= "a"

        Hence new production will be S->aA | a and A->b

Chomsky Normal Form

- Chomsky Normal Form (CNF) : A grammar where every production is either of the form

  A -> BC

  or

  A -> c

  where A, B, C are arbitrary variables and c an arbitrary symbol.
- Example:

  S -> AS | a

  A -> SA | b

Chomsky Normal Form

- The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly 2n − 1 steps.
- The conversion to Chomsky Normal Form has four main steps:
  - Remove ε-production
  - Remove unit production
  - Remove useless symbol
  - Replace long production by shorter one

  For example, if we have production like A->BCD, then replace it with A->BE and E->CD.

  If we have production like A->bC, then replace it with A->BC and B->b

Greibach Normal Form

- Greibach Normal Form (GNF) : A CFG is said to in GNF it its all production rules are of type

  A->aα where $\alpha \epsilon$ V*
- The RHS of any production must start with a single terminal symbol followed by variables.
- For example :

  S->ABC------------------------ Not in GNF (Why?)

  S->A+B-------------------------Not in GNF (Why?)

  S->a------------------------------In GNF

  S->bBCD------------------------In GNF

  S->A----------------------------Not in GNF

Greibach Normal Form

- Advantages of GNF
  - Avoid left recursion
  - Helps in selecting the suitable production in derivation of a string.
  - Guarantees derivation length no longer than string length.
- Conversion of CFG into GNF
  - Convert CFG into CNF
  - Change the label of the variables
  - Modify the grammar so that every production are in the form

    $A_i \rightarrow aY$ or,

    $A_i \rightarrow A_jY$ where $j > i$ and $Y \in V^*$
  - Eliminate the left recursions
  - Substitute the productions to get the GNF form

Greibach Normal Form

Example:

CNF	New Labels	Updated CNF
$S \rightarrow XA \mid BB$	$S = A_1$	$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$B \rightarrow b \mid SB$	$X = A_2$	$A_4 \rightarrow b \mid A_1A_4$
$X \rightarrow b$	$A = A_3$	$A_2 \rightarrow b$
$A \rightarrow a$	$B = A_4$	$A_3 \rightarrow a$

Greibach Normal Form

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

First Step $\boxed{A_i \rightarrow A_jX_k \quad j > i}$

X_k is a string of zero or more variables

✕ $A_4 \rightarrow A_1A_4$

Greibach Normal Form

First Step $A_i \rightarrow A_jX_k \quad j > i$

$A_4 \rightarrow A_1A_4$
$A_4 \rightarrow A_2A_3A_4 \mid A_4A_4A_4 \mid b$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Greibach Normal Form

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Second Step

Eliminate Left Recursions

$\times\; A_4 \rightarrow A_4A_4A_4$

Greibach Normal Form

Eliminate Left Recursions

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Greibach Normal Form

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow A_4A_4 \mid A_4A_4Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

$\boxed{A \rightarrow \alpha X}$

GNF

Greibach Normal Form

$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Grammar in Greibach Normal Form

Pumping lemma for CFL

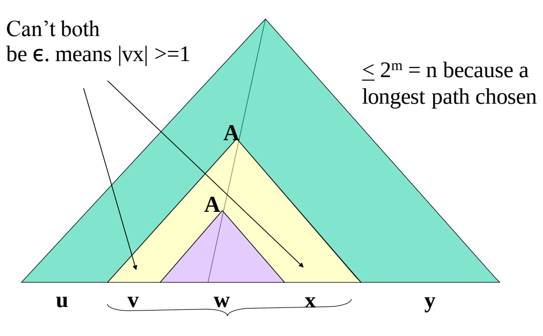
- Lemma: *For every context-free language L, there is an integer n such that for every string z in L of length > n there exists z = uvwxy such that:*
  1. $|vx| \geq 1$
  2. $|vwx| \leq n$ *and*
  3. *For all* $i \geq 0$, uv^iwx^iy *is in L.*

- *Proof:*
  - Start with a CNF grammar for $L - \{\varepsilon\}$.
  - Let the grammar have m variables.
  - Pick $n = 2^m$.
  - Let $|z| \geq n$

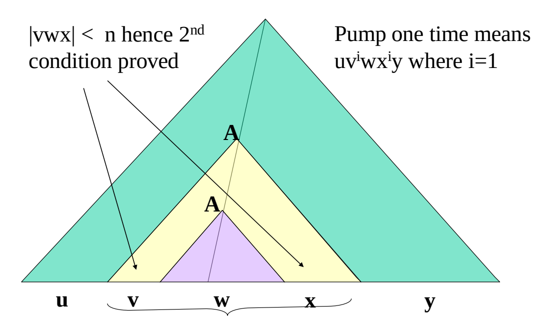
Pumping lemma for CFL

- Since $|z| > 2^{m-1}$, any parse tree for z must have a path of length at least $m+1$.
- Such a path has at least $m+2$ vertices.
- Thus there must be some variable that appears twice on the path
- Consider some longest path.
- There are only *"m"* different variables, so among the lowest *"m+1"* we can find two nodes with the same label, say A.
- The parse tree thus looks like:

Pumping lemma for CFL

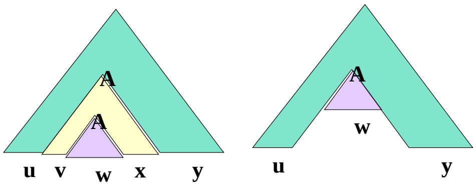


Pumping lemma for CFL



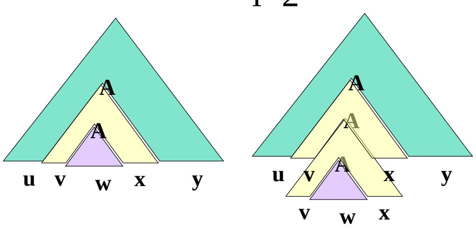
Pumping lemma for CFL

Pump Zero Times i.e. uv^iwx^iy where i=0



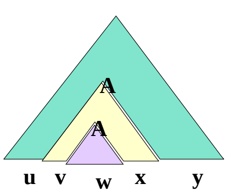
Pumping lemma for CFL

Pump Twice i.e. uv^iwx^iy where i=2

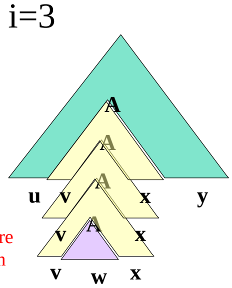


Pumping lemma for CFL

Pump Thrice i.e. uv^iwx^iy where i=3



We observed that uv^iwx^iy where i>=0 is in L hence 3rd condition is proved



Pumping lemma for CFL

- We have now shown all conditions of the pumping lemma for context free languages
- To show a language is not context free we
  - Pick a language L to show that it is not a CFL
  - Then some *n must exist, indicating the maximum yield and length* of the parse tree
  - We pick the string z, and may use *n as a parameter*
  - Break z into uvwxy subject to the pumping lemma constraints |vwx| < n, |vx| >=1
  - Pick *i* and show that uv^iwx^iy is not in L, therefore L is not context free.

Example

- Let L be the language $\{a^nb^nc^n \mid n >=1\}$. Show that this language is not a CFL.
  - Suppose that L is a CFL. Then some integer n exists and we pick $z = a^nb^nc^n$.
  - Since z=uvwxy and |vwx| < n
  - Let us pick n=4 hence w=aaaabbbbcccc.
  - Now break w into $\underbrace{aaaab}_{u}\underbrace{b}_{v}\underbrace{b}_{w}\underbrace{b}_{x}\underbrace{cccc}_{y}$ [|vwx|<4 and |vx|>=1]
  - Hence for i=2, uv^iwx^iy should belong to L. but for i=2 we see that the string aaaab bb b bb cccc does not belong to L.
  - Therefore L is not a CFL.

Closure Properties of CFL's

- CFL's are closed under union, concatenation, and Kleene closure.
- But not under intersection or difference.
- Closure of CFL's Under Union
  - Let L and M be CFL's with grammars G and H, respectively.
  - Assume G and H have no variables in common.
    - ✓Names of variables do not affect the language.
  - Let S_1 and S_2 be the start symbols of G and H.
  - Form a new grammar for $L \cup M$ by combining all the symbols and productions of G and H.
  - Then, add a new start symbol S.
  - Add productions $S \rightarrow S_1 \mid S_2$.

Closure Properties of CFL's

- Closure of CFL's Under Concatenation
  - Let L and M be CFL's with grammars G and H, respectively.
  - Assume G and H have no variables in common.
  - Let S_1 and S_2 be the start symbols of G and H.
  - Form a new grammar for LM by starting with all symbols and productions of G and H.
  - Add a new start symbol S.
  - Add production $S \rightarrow S_1S_2$.
  - Every derivation from S results in a string in L followed by one in M.

Closure Properties of CFL's

- Closure of CFL's Under Star
  - Let L have grammar G, with start symbol S_1.
  - Form a new grammar for L* by introducing to G a new start symbol S and the productions $S \rightarrow S_1S \mid \varepsilon$.
  - A rightmost derivation from S generates a sequence of zero or more S_1's, each of which generates some string in L.
- CFLs are not closed under intersection
  - Unlike the regular languages, the class of CFL's is not closed under $\cap$.
  - We know that $L_1 = \{0^n1^n2^n \mid n \geq 1\}$ is not a CFL (use the pumping lemma).

Closure Properties of CFL's

  - However, $L_2 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$ is.
    - ✓CFG: $S \rightarrow AB$, $A \rightarrow 0A1 \mid 01$, $B \rightarrow 2B \mid 2$.
  - So is $L_3 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$.
  - But $L_1 = L_2 \cap L_3$ and L_1 is not context-free so does $L_2 \cap L_3$.
- However the intersection of a CFL with a regular language is always a CFL. (Why?)
- CFLs are not closed under difference
  - Any class of languages that is closed under difference is closed under intersection.
  - Proof: $L \cap M = L - (L - M)$.
  - Thus, if CFL's were closed under difference, they would be closed under intersection, but they are not.

Parsing

- ***Parsing or syntactic analysis*** is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar.
- A ***parser*** is a software component that takes input a string "*w*" and produces output either a parse tree for "*w*" or an error message indicating that "*w*" is not a sentence of grammar G.
- Two Basic type of parser for CFG
  - Top Down Parser
    - ✓ Top down parser starts with the root and work down to the leaves.
    - ✓ Example LL(1) parser
  - Bottom-up parser
    - ✓ Bottom-up parser build parse tree from bottom to the root.
    - ✓ Example :- LR(1), SLR(1) parser

Top-down Vs Bottom-up Parser

- *Top-down parser:*
  - starts at the root of derivation tree and fills in
  - picks a production and tries to match the input
  - may require backtracking
  - some grammars are backtrack-free (*predictive*)
- *Bottom-up parser:*
  - starts at the leaves and fills in
  - starts in a state valid for legal first tokens
  - as input is consumed, changes state to encode possibilities (*recognize valid prefixes*)
  - uses a *stack* to store both state and sentential forms

Top Down Parser

- A top down parser starts constructing the left most derivation from the start symbol
- In next step it finds a suitable production rule in such a way that by using that rule, it can move from a left most sentential form to its succeeding one.
- If the left most non-terminal has more than one production rule, a selection may be made depending on whether backtracking is permitted or not.
- If backtracking is allowed parser can make repeated scan of the input.
- If not allowed then parser has to select the correct alternate at each step.

Top Down Parser

- *A top-down parser starts with the root of the parse tree, labeled with the start or goal symbol of the grammar.*
- To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string
  - At a node labeled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
  - When a terminal is added to the fringe that doesn´t match the input string, backtrack
  - Find the next node to be expanded (must have a label in V_n)
- The key is selecting the right production in step 1
  ⇒ should be guided by input string

Top Down Parser

- Consider the following grammar

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> + <term>
3.		|	<expr> - <term>
4.		|	<term>
5.	<term>	::=	<term> * <factor>
6.		|	<term> / <factor>
7.		|	<factor>
8.	<factor>	::=	num
9.		|	id

Consider the input string x — 2 * y

Top Down Parser

Prod'n	Sentential form	Input
–	⟨goal⟩	↑x – 2 * y
1	⟨expr⟩	↑x – 2 * y
2	⟨expr⟩ + ⟨term⟩	↑x – 2 * y
4	⟨term⟩ + ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ + ⟨term⟩	↑x – 2 * y
9	id + ⟨term⟩	↑x – 2 * y
–	id + ⟨term⟩	x ↑– 2 * y
–	⟨expr⟩	↑x – 2 * y
3	⟨expr⟩ – ⟨term⟩	↑x – 2 * y
4	⟨term⟩ – ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ – ⟨term⟩	↑x – 2 * y
9	id – ⟨term⟩	↑x – 2 * y
–	id – ⟨term⟩	x ↑– 2 * y
–	id – ⟨term⟩	x – ↑2 * y
7	id – ⟨factor⟩	x – ↑2 * y
8	id – num	x – ↑2 * y
–	id – num	x – 2 ↑* y
–	id – ⟨term⟩	x – ↑2 * y
5	id – ⟨term⟩ * ⟨factor⟩	x – ↑2 * y
7	id – ⟨factor⟩ * ⟨factor⟩	x – ↑2 * y
8	id – num * ⟨factor⟩	x – ↑2 * y
–	id – num * ⟨factor⟩	x – 2 ↑* y
–	id – num * ⟨factor⟩	x – 2 * ↑y
9	id – num * id	x – 2 * ↑y
–	id – num * id	x – 2 * y ↑

LL(1) Parser

- Properties of LL(1) grammar
  - No ambiguous or left recursive grammar
  - All entries in the parsing table are unique for an LL(1) grammar
  - A grammar G is LL(1) if and only if the following conditions hold for any production $A \rightarrow \alpha \mid \beta$
    - ✓ $First(\alpha) \cap First(\beta) = \varphi$
    - ✓ $First(\alpha) \cap First(\beta) \neq \lambda$
    - ✓ $First(\alpha) \cap Follow(A) = \varphi$ if (λ belongs to $First(\beta)$)

Finding the First(α)

- For a string of grammar symbols α, define FIRST(α) as:
  - the set of terminal symbols that begin strings derived from α:
    $\{ a \in V_t \mid \alpha \Rightarrow^* a\beta \}$
  - If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in FIRST(\alpha)$
- To build FIRST(X):
  1. If $X \in V_t$, then FIRST(X) is { X }
  2. If $X \rightarrow \varepsilon$ then add ε to FIRST(X)
  3. If $X \rightarrow Y_1 Y_2 \ldots Y_k$
     a) Put $FIRST(Y_1) - \{\varepsilon\}$ in FIRST(X)
     b) $\forall i: 1 < i \le k$, if $\varepsilon \in FIRST(Y_1) \cap \ldots \cap FIRST(Y_{i-1})$
        (i.e., $Y_1 Y_2 \ldots Y_{i-1} \Rightarrow^* \varepsilon$)
        then put $FIRST(Y_i) - \{\varepsilon\}$ in FIRST(X)
     c) If $\varepsilon \in FIRST(Y_1) \cap \ldots \cap FIRST(Y_k)$
        then put ε in FIRST(X)

Repeat until no more additions can be made.

Finding the Follow(A)

- For a non-terminal A, define FOLLOW(A) as:
  - the set of terminals that can appear immediately to the right of A in some sentential form
  - I.e., a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.
  - A terminal symbol has no FOLLOW set.
- To build FOLLOW(A):
  1. Put $ in FOLLOW(<goal>)
  2. If $A \rightarrow \alpha B \beta$:
     1. Put $FIRST(\beta) - \{\varepsilon\}$ in FOLLOW(B)
     2. If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in FIRST(\beta)$ (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW(A) in FOLLOW(B)

Repeat until no more additions can be made

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. ∀ production $A \to \alpha$:
   a) $\forall a \in FIRST(\alpha)$, add $A \to \alpha$ to $M[A,a]$
   b) If $\varepsilon \in FIRST(\alpha)$:
      I. $\forall b \in FOLLOW(A)$, add $A \to \alpha$ to $M[A,b]$
      II. If $\$ \in FOLLOW(A)$, add $A \to \alpha$ to $M[A,\$]$
2. Set each undefined entry of M to error

If $\exists M[A,a]$ with multiple entries then G is not LL(1).

LL(1) parse table construction

Let us take a grammar:

$S \to E$
$E \to TE$
$E \to +E \mid -E \mid \varepsilon$
$T \to FT$
$T \to *T \mid /T \mid \varepsilon$
$F \to num \mid id$

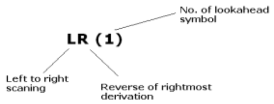
	FIRST	FOLLOW
S	{num, id}	{\$}
E	{num, id}	{\$}
E'	{ε, +, −}	{\$}
T	{num, id}	{+, −, \$}
T'	{ε, *, /}	{+, −, \$}
F	{num, id}	{+, −, *, /, \$}
id	{id}	−
num	{num}	−
*	{*}	−
/	{/}	−
+	{+}	−
−	{−}	−

	id	num	+	−	*	/	\$
S	$S \to E$	$S \to E$	−	−	−	−	−
E	$E \to TE'$	$E \to TE'$	−	−	−	−	−
E'	−	−	$E' \to +E$	$E' \to -E$	−	−	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	−	−	−	−	−
T'	−	−	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to$ id	$F \to$ num	−	−	−	−	−

Bottom-up Parser

- A parser can start with the input and attempt to rewrite it to the start symbol.
- Bottom-up parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.
- Also known as Shift-Reduce parser
  - Shift -> Move terminal symbol to left string
  - Reduce -> Immediately on the left of "." identify a string same as RHS of a production and replaced by LHS

Bottom-up Parser



- SLR(1) Parser :
  - Simple LR(1) parser
  - All SLR grammars are unambiguous grammar
  - A grammar for which SLR parser can be constructed is said to be an SLR grammar.
  - SLR parsing is based on LR(0) items.
  - LR(0) items can be made by inserting "." (dot) symbol at right position in the RHS of the rule
    - ✓Example :- D -> type list; => D->.type list;

SLR(1) Parser

  - For A -> ε simply we can write A- > .
- LR(0) items mark which we have already seen at a given point of time.
- Making of Parse tree of SLR(1)
  - Add an augmented production in the grammar
  - Find the closure of variables followed by "." (dot)
  - Construct the DFA for possible inputs and make the states.
  - From the DFA make the parsing table
    - ✓Parsing table has two parts one section is called "Action" in which only transition with terminals are kept and second part is "Goto" part in which transition with variables or non-terminals are kept.

SLR(1) Parser

- Closure function Closure(I)
  - I is a set of items for a grammar G
  - Every item in I is in Closure(I)
  - If $A \rightarrow \alpha \bullet B \beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G Then add $B \rightarrow \bullet \gamma$ to Closure(I)
    - ✓If it is not already there
    - ✓Meaning
      - When α is in the stack and B is expected next
      - One of the B-production rules may be used to reduce the input to B
        - May not be one-step reduction though
  - Apply the rule until no more new items can be added

Building an automata

- Given a grammar G
- Step 1: augment G
- Step 2: initial state
  - Construct the valid item set "I" of State 0 (the initial state)
  - Add S' -> S into I
    - ✓ All expansions have to start from here
  - Compute Closure(I) as the complete valid item set of state 0
    - ✓ All possible expansions S can lead into
- Step 3:
  - From state I, for all grammar symbol X

    Construct J = Goto(I, X)

    Compute Closure(J)
  - Create the new state with the corresponding Goto transition
    - ✓ Only if the valid item set is non-empty and does not exist yet
- Repeat Step 3 till no new states can be derived

Building an automata

- Grammar G:

  $S \to E$

  $E \to E + T \mid T$

  $T \to id \mid (E)$
  - Step 1: Augment G

    $S' \to S \quad S \to E \quad E \to E + T \mid T \quad T \to id \mid (E)$
  - Step 2:
    - ✓ Construct Closure(I_0) for State 0
    - ✓ First add into I_0: $S' \to \bullet S$
    - ✓ Compute Closure(I_0)

      $S' \to \bullet S \quad S \to \bullet E$

      $E \to \bullet E + T \quad E \to \bullet T$

      $T \to \bullet id \quad T \to \bullet (E)$

Expect to see S next

S won't just appear
May have to see E first and reduce it to S using this rule

Building an automata

- Step 3
  - I_1
    - ✓ Add into I_1: Goto(I_0, S) = $S' \to S \bullet$
    - ✓ No new items to be added to Closure (I_1)
  - I_2
    - ✓ Add into I_2: Goto(I_0, E) = $S \to E \bullet \quad E \to E \bullet + T$
    - ✓ No new items to be added to Closure (I_2)
  - I_3
    - ✓ Add into I_3: Goto(I_0, T
    - ✓ No new items to be a
  - I_4
    - ✓ Add into I_4: Goto(I_0, id) = $T \to id \bullet$
    - ✓ No new items to be added to Closure (I_4)

I_0:

$S' \to \bullet S \quad S \to \bullet E$

$E \to \bullet E + T \quad E \to \bullet T$

$T \to \bullet id \quad T \to \bullet (E)$

When E is moved to the stack (after a reduction), these two are the possible handles

$S \to E \bullet$ implies a reduction is to be done

- **should be done if seeing Follow(S)**

$E \to E \bullet + T$ implies + is expected to be the next input

Building an automata

- Step 3
  - I_5
    - ✓Add into I_5: Goto(I_0, "(") = $T \rightarrow (\bullet E)$
    - ✓Closure(I_5)

      $E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

      $T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$
  - No more moves from I_0
  - No possible moves from I_1
  - I_6
    - ✓Add into I_6: Goto(I_2, +) = $E \rightarrow E + \bullet T$
    - ✓Closure(I_5)

      $T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$
  - No possible moves from I_3 and I_4

I_0:

$S' \rightarrow \bullet S \quad S \rightarrow \bullet E$

$E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

$T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$

After seeing (, we expect E next
E could be reduced from other
E-production rules
So, put E-productions in the set

Building an automata

- Step 3
  - I_7
    - ✓Add into I_7: Goto(I_5, E) =

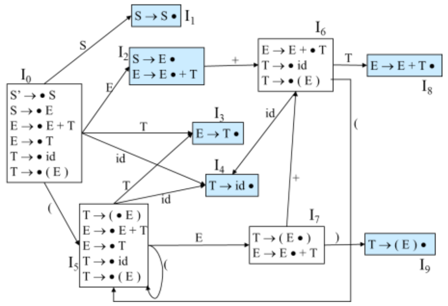
      $T \rightarrow (E \bullet) \quad E \rightarrow E \bullet + T$
    - ✓No new items to be added to Closure (I_7)
  - Goto(I_5, T) = I_3
  - Goto(I_5, id) = I_4
  - Goto(I_5, "(") = I_5
  - No more moves from I_5
  - I_8
    - ✓Add into I_8: Goto(I_6, T) = $E \rightarrow E + T \bullet$
    - ✓No new items to be added to Closure (I_8)
  - Goto(I_6, id) = I_4
  - Goto(I_6, "(") = I_5

Building an automata

- Step 3
  - I_9
    - ✓Add into I_9: Goto(I_7, ")") =

      $T \rightarrow (E) \bullet$
    - ✓No new items to be added to Closure (I_9)
  - Goto(I_7, +) = I_6
  - No possible moves from I_8 and I_9

Building an automata

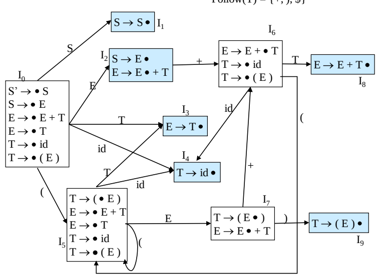


Building an automata

Stack	Input	Action
0	id + id $	S4
0 id 4	+ id $	T→id, Goto[0,T]=3
0 T 3	+ id $	E→T, Goto[0,E]=2
0 E 2	+ id $	s6
0 E 2 + 6	id $	S4
0 E 2 + 6 id 4	$	T→id, Goto[6,T]=8
0 E 2 + 6 T 8	$	E→E+T, Goto[0,E]=2
0 E 2	$	S→E, Goto[0,S]=1
0 S 1	$	accept

See how parsing works directly on the automata

Follow(S) = {$}
Follow(E) = {+,), $}
Follow(T) = {+,), $}



Building Parsing Table

- If [A->α.aβ] ϵ I_i and goto [I_i, a] = I_j then action[I ,a]= S_j
- If [A -> α] ϵ I_i then action [I, a] = r_j
- If [S' -> S] ϵ I_n then action [I,$] =Accept
- For all non-terminals A
  - If goto [I_i, A] =I_j then goto [I,A]=j
- All remaining entry are marked as error.

Building Parsing Table

Follow(S) = {$}
Follow(E) = {+,), $}
Follow(T) = {+,), $}

Action Table | **Goto Table**

	+	id	(	)	$	S	E	T
0		4	5			1	2	3
1					Acc			
2	6				S→E			
3	E→T			E→T	E→T			
4	T→id			T→id	T→id			
5		4	5				7	3
6		4	5					8
7	6			9				
8	E→E+T			E→E+T	E→E+T			
9	T→(E)			T→(E)	T→(E)			

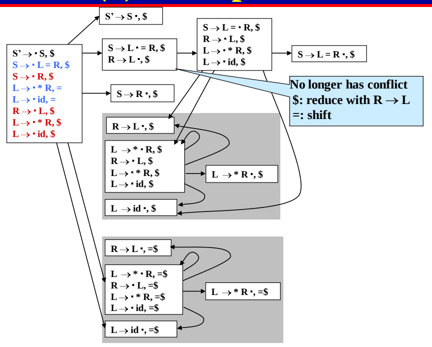
Building Parsing Table

- Problems in SLR(1)
  - Shift Reduce conflict
  - Reduce reduce conflict

LR(1)

- Canonical LR(1) parser
- It contains second component which is lookahead symbol so that wrong reduction by A->α will be ruled out.
- The general form of item becomes [A -> $\alpha\beta$, a] where "a" is follow of A.
- Construction of parsing table is similar to SLR(1).

LR(1) Example



© Bharati Vidyapeeth's Institute of Computer Applications and Management,, New Delhi-63, by Manish Kumar U2. 85