6 - Left Recursion & Left Factoring

Tuesday, 27 February 2024 10:03 AM

Ashima Fatima Seik Mugibur Raghman, 21BAI1830

AIM: Write a program to eliminate left recursion and left factoring from the given production.

Left Recursion.

If production is $A \rightarrow A\alpha / \beta$

Rule to eliminate left recursion is,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$

Sample I/p

$S \rightarrow SAb | d$

Sample O/p

$S' \rightarrow AbS' | \epsilon$

e.g. $S \rightarrow aA$

$A \rightarrow ABc | d$

$B \rightarrow d | \epsilon$

o/p $S \rightarrow AA$

$A \rightarrow dA'$

$A' \rightarrow BcA' | \epsilon$

$B \rightarrow d | \epsilon$

Left Factoring

If grammar production is, $A \rightarrow \alpha A_1 | \alpha A_2 | \alpha A_3 \ldots | \alpha A_n$

Rule to eliminate left factoring is,

$A \rightarrow \alpha A'$

$A' \rightarrow A_1 | A_2 | A_3 \cdots | A_n$

Sample I/p

$S \rightarrow aA | aB | aC$

Sample O/p

$S \rightarrow aS'$

$S' \rightarrow A | B | C$

e.g. $S \rightarrow A | C$

$A \rightarrow bA | bB | bC$

o/p $S \rightarrow A | C$

$A \rightarrow bA'$

$A' \rightarrow A | B | C.$

$C \rightarrow bc$

$B \rightarrow a$

Q. $R \rightarrow Right | Ring | Rim$ $\quad$ $R \rightarrow Ri R'$ $\quad$ $R \rightarrow ght | ng | m$

$S \rightarrow here | hear | he$ $\quad$ $S \rightarrow he S'$ $\quad$ $S' \rightarrow re | ar | \epsilon$

$S \rightarrow bat | ball | box$ $\quad$ $S \rightarrow b S'$ $\quad$ $S' \rightarrow at | all | ox$

$S \rightarrow the | there | three$ $\quad$ $S \rightarrow th S'$ $\quad$ $S' \rightarrow e | ere | ree$

PROCEDURE :

Left Recursion –

1. input the production string and store it in an array

Left Recursion –

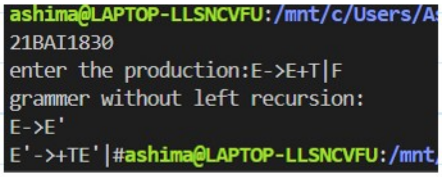
1. input the production string and store it in an array
2. extract the non-terminal symbol from the production string array and store it in the variable 'non_terminal'
3. check for left recursion by comparing the non-terminal symbol with the symbol at index i in the production string array.
    - a. If the non-terminal symbol matches, extract the alpha and beta parts of the production
    - b. If the beta part is not empty, print the modified grammer without left recursion
    - c. If the beta part is empty, print "grammer can't be reduced."
    - d. If the non-terminal symbol doesn't match, print "grammer is not left recursive."

Left Factoring:

1. enter the number of productions and the productions in a specific format.
2. Read each production and parse it to identify the non-terminal and its corresponding productions
3. Iterate through each non-terminal & check if left factoring can be applied. If there are multiple productions sharing a common prefix, left factor those productions by introducing a new non-terminal
4. Implement this left factoring algorithm by identifying common prefixes in the productions. If a common prefix is found, a new non-terminal is created for the common prefix and the existing productions are modified accordingly.
5. The left factoring process is iterated until no more left factoring can be applied.

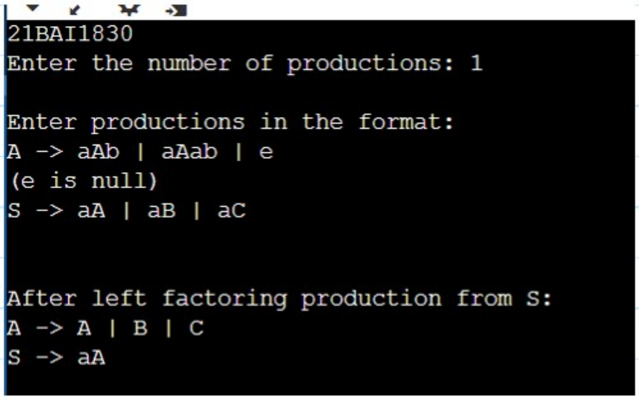
SAMPLE CODE OUTPUT

Left Recursion



```
ashima@LAPTOP-LLSNCVFU:/mnt/c/Users/A
21BAI1830
enter the production:E->E+T|F
grammer without left recursion:
E->E'
E'->+TE'|#ashima@LAPTOP-LLSNCVFU:/mnt/
```

Left Factory



```
21BAI1830
Enter the number of productions: 1

Enter productions in the format:
A -> aAb | aAab | e
(e is null)
S -> aA | aB | aC


After left factoring production from S:
A -> A | B | C
S -> aA
```

CODE

Left Recursion

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 20
void main(){
    printf("21BAI1830\n");
    char pro[SIZE], alpha[SIZE],beta[SIZE];
    char nont_terminal;
    int i,j,index=3;

    printf("enter the production:");
    scanf("%s",pro);

    nont_terminal = pro[0];
    if(nont_terminal==pro[index]){
        for(i=++index,j=0;pro[i]!='|';i++,j++){
            alpha[j]=pro[i];

            if(pro[i+1]==0){
                printf("this grammar can't be reduced");
                exit(0);
            }
        }
        alpha[j]='\0';
        if(pro[++i]!=0){
            for(j=i,i=0;pro[j]='\0';i++,j++){
                beta[i]=pro[j];

            }
            beta[i]='\0';
            printf("grammer without left recursion: \n");
            printf("%c->%s%c\'\n",nont_terminal,beta,nont_terminal);
            printf("%c\'->%s%c\'|#",nont_terminal,alpha,nont_terminal);
        }
        else{
            printf("this grammer can't be reduced");
        }
    }
    else{
        printf("this grammar is not left recursive");
    }
}
```

Left Factoring

```c
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
struct production {
    int length;
    char prod[10][10];
} typedef production;
struct grammer {
    int number;
    production nonT[26];
} typedef grammer;
grammer gram;
void printProd(production *);
void printGram();
int getF(char *str) {
    int i = 1;
    while (i < strlen(str) && str[i - 1] != '>') i++;
    while (i < strlen(str) && str[i] == ' ') i++;
    return i;
}
void setProd(char *str) {
    production *produce = &gram.nonT[str[0] - 'A'];
    int prod = 0, j;
    for (int i = getF(str); i < strlen(str); i++) {
        for (j = 0; i < strlen(str) && str[i] != '|'; i++)
            if (str[i] != ' ')
                produce->prod[prod][j++] = str[i];
        produce->prod[prod++][j++] = '\0';
    }
    produce->length = prod;
}
void copy(char *src, char *dest, int i) {
    int j = i + 1;
    do {
        dest[j - i - 1] = src[j];
    } while (src[j++] != '\0');
    if (dest[0] == '\0') {
        dest[0] = 'e';
        dest[1] = '\0';
    }
}
bool factorProd(production *prod) {
    bool prev[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    bool next[10] = {0};
    char c[128] = {0}, ch;
    int j = 0, maxx;
    while (true) {
        maxx = 0;
        for (int i = 0; i < 10; i++)
            if (prev[i] && strlen(prod->prod[i]) > j)
                c[prod->prod[i][j]]++;
        for (int i = 0; i < 128; i++) {
            if (c[i] > maxx) maxx = c[i], ch = i;
            c[i] = 0;
        }
        if (maxx < 2) break;
        for (int i = 0; i < 10; i++)
            prev[i] = j < strlen(prod->prod[i]) && prod->prod[i][j] == ch;
        j++;
    }
    if (j == 0) return false;
    int newNonT = 0, p_no = 0;
    while (gram.nonT[newNonT].length != 0) newNonT++;
    production *newP = &gram.nonT[newNonT];
    for (int i = 0; i < 10; i++)
        if (prev[i] && prod->prod[i][0] != '\0') {
            copy(prod->prod[i], newP->prod[p_no], j - 1);
            prod->prod[i][j] = newNonT + 'A';
            prod->prod[i][j + 1] = '\0';
            p_no++;
        }
    gram.number++;
    newP->length = p_no;
    int i = 0, num = 0;
    while (prev[i] == false) i++;
    for (i++; i < 10; i++)
        if (prev[i] && prod->prod[i][0] != '\0')
            prod->prod[i][0] = '\0', num++;
    prod->length -= num;
    return true;
}
void leftFactor() {
    for (int i = 0; i < 26; i++) {
        if (gram.number == 26) {
            printf("No new production can be formed.\n");
            return;
        }
        if (gram.nonT[i].length >= 2)
            while (factorProd(&gram.nonT[i])) {
                printf("\n\nAfter left factoring production from %c:", i + 'A');
                printGram();
            }
    }
}
int main() {
```

```c
    char str[100];
    printf("21BAI1830\n");
    printf("Enter the number of productions: ");
    scanf("%d", &(gram.number));
    printf("\nEnter productions in the format:\n");
    printf("A -> aAb | aAab | e\n");
    printf("(e is null)\n");
    char c;
    for (int i = 0; i < gram.number; i++) {
        scanf("%c", &c);
        scanf("%[^\n]s", str);
        setProd(str);
    }
    leftFactor();
    return 0;
}
void printProd(production *prod) {
    printf("%s ", prod->prod[0]);
    for (int i = 1; i < 10; i++)
        if (prod->prod[i][0] != '\0')
            printf("| %s ", prod->prod[i]);
}
void printGram() {
    for (int i = 0; i < 26; i++)
        if (gram.nonT[i].length) {
            printf("\n%c -> ", i + 'A');
            printProd(&gram.nonT[i]);
        }
}
```