Practical 5: Write a program in Lex /Yacc to create LR(1) Parser

Gahan Saraiya (18MCEC10)

18mcec10@nirmauni.ac.in

I. AIM

Write a program in Lex /Yacc to create LR(1) Parser

II. Implementation

I. C file

```
#include <stdio.h>
#include <stdlib.h>
#define t_shift 0
#define t_reduce 1
#define t_accept 2
#define t_blank 3

struct action{
        int type;
        int value;
};

struct goto_table{
        char *symbols;
        int **table;
};

struct action_table{
        char *symbols;
        struct action **table;
};

struct lr_table{
        int num_states;
        int num_nonterm;
        int num_term;
        struct action_table at;
        struct goto_table gt;
};

int char_to_col(char c, char* ca, int len){
        for(int i=0; i<len; ++i){
                if(ca[i]==c) return i;
        }
        return -1;
}

void PrintTable(struct lr_table* lrt){
        printf("action table:\n");
        for(int i=0; i<lrt->num_term; ++i){
                printf(" %c ", lrt->at.symbols[i]);
        }
        printf("\n");
        for(int i=0; i<lrt->num_states; ++i){
                for(int j=0; j<lrt->num_term; ++j){
                        int type = lrt->at.table[i][j].type;
                        if(type==t_shift) printf("s%d ",
                         ↪  lrt->at.table[i][j].value);
                        else if(type == t_reduce) printf("r%d ",
                         ↪  lrt->at.table[i][j].value);
                        else if(type == t_accept) printf(" a");
                        else printf("   ");
                }
                printf("\n");
        }
        printf("goto table:\n");
        for(int i=0; i<lrt->num_nonterm; ++i){
                printf("%c ", lrt->gt.symbols[i]);
        }
        printf("\n");
        for(int i=0; i<lrt->num_states; ++i){
                for(int j=0; j<lrt->num_nonterm; ++j){
                        int val = lrt->gt.table[i][j];
                        if(val==-1) printf("  ");
                        else printf("%d ", val);
                }
                printf("\n");
        }
```

```
}

void PrintTableNice(struct lr_table* lrt){
        printf("\nTable:\n");
        printf("|       |");
        for(int i=0; i<lrt->num_term; ++i){
                printf("%c\t", lrt->at.symbols[i]);
        }
        printf("|");
        for(int i=0; i<lrt->num_nonterm; ++i){
                printf(" %c\t", lrt->gt.symbols[i]);
        }
        printf("|\n");
        int type;
        for(int i=0; i<lrt->num_states; ++i){
                printf("|   %2d  |", i);
                for(int j=0; j<lrt->num_term; ++j){
                        type = lrt->at.table[i][j].type;
                        if(type==t_shift) printf("s%d\t",
                         ↪  lrt->at.table[i][j].value);
                        else if(type == t_reduce) printf("r%d\t",
                         ↪  lrt->at.table[i][j].value);
                        else if(type == t_accept) printf(" a\t");
                        else printf("\t");
                }
                printf("|");
                for(int j=0; j<lrt->num_nonterm; ++j){
                        int val = lrt->gt.table[i][j];
                        if(val==-1) printf("  \t");
                        else printf("%2d\t", val);
                }
                printf("|\n");
        }
        printf("\n");
}

char* appendToCharArray(char c, char* array, int len){
        if(array==NULL){
                char *cp = (char*)malloc(sizeof(char));
                *cp = c;
                return cp;
        }
        char *cp = (char *)malloc(len+1);
        for(int i=0; i<len; ++i){
                *(cp+i) + *(array+i);
        }
        *(cp+len) = c;
        free(array);
        return cp;
}

int discardable(char c){
        if(c=='\t' || c=='\n' || c==' ') return 1;
        return 0;
}

struct lr_table* CreateTable(){
        int k;
        struct lr_table* lrt = (struct lr_table*)malloc(sizeof(struct lr_table));
        printf("How many non-terms are there ?: ");
        scanf("\n%d", &lrt->num_nonterm);
        printf("How many terminals are there ?: ");
        scanf("\n%d", &lrt->num_term);
        printf("How many states are there ?: ");
        scanf("\n%d", &lrt->num_states);

        // Enter non terminals
        lrt->gt.symbols = (char*)malloc(lrt->num_nonterm * sizeof(char));
        printf("Enter non terminals: ");
        char c;
        for(int i=0; i<lrt->num_nonterm; ++i){
                scanf("%c", &c);
                if(discardable(c)){
                        i--;
                        continue;
                }
                lrt->gt.symbols[i] = c;
        }

        // Enter terminals
        lrt->at.symbols = (char*)malloc(lrt->num_term * sizeof(char));
        printf("Enter terminals: ");
        for(int i=0; i<lrt->num_term; ++i){
                scanf("%c", &c);
                if(discardable(c)){
                        i--;
                        continue;
                }
                lrt->at.symbols[i] = c;
        }

        // Enter action table
        printf("Enter action table in matrix form: 00=blank, si=shift i,
         ↪  ri=reduce i, a0=accept\n");
        lrt->at.table = (struct action**)malloc(lrt->num_states * sizeof(struct
         ↪  action*));
        int type;
        for(int i=0; i<lrt->num_states; ++i){
                lrt->at.table[i] = (struct action*)malloc(lrt->num_term *
                 ↪  sizeof(struct action));
                for(int j=0; j<lrt->num_term; ++j){
                        scanf(" %c%d", &c, &k);
                        if(c=='s') type = t_shift;
                        else if(c=='r') type = t_reduce;
                        else if(c=='a') type = t_accept;
                        else type = t_blank;
                        lrt->at.table[i][j].type = type;
                        lrt->at.table[i][j].value = k;
                }
        }

        // Enter goto table
        printf("Enter goto table in matrix form: -1=blank\n");
        lrt->gt.table = (int **)malloc(lrt->num_states * sizeof(int *));
        for(int i=0; i<lrt->num_states; ++i){
                lrt->gt.table[i] = (int*)malloc(lrt->num_nonterm * sizeof(int));
                for(int j=0; j<lrt->num_nonterm; ++j){
                        scanf(" %d", &k);
                        lrt->gt.table[i][j] = k;
                }
        }

        return lrt;
}
#define explen 100

```

```
struct rule{
        char c;
        int n;
};

struct rule* rules;

int* append_int(int n, int *arr, int *p);
int getReduction(int k);
char getRedChar(int k);
void printStack(int* stack, int n);
struct rule* appendRule(struct rule r, struct rule* _rules, int p);
void printBuffer(int buffer[], int buf_pos);
void reduceBuffer(int buffer[], int* buf_pos, int rule);

int main(){

        int num_rules;
        printf("How many rules are there ?: ");
        scanf("\n%d", &num_rules);
        printf("Enter rules properties: left(symbol) right(count). Eg. {A->Aa}=>{A 2}");
        rules = (struct rule*)malloc(5*sizeof(struct rule));
        char lhs; int rhs;
        struct rule r;
        for(int i=0; i<num_rules; ++i){
                scanf("\n%c %d", &lhs, &rhs);
                r.c = lhs;
                r.n = rhs;
                rules = appendRule(r, rules, i);
        }
        struct lr_table* lrt = CreateTable();
        // PrintTable(lrt);
        PrintTableNice(lrt);

        // Scan expression
        char expr[explen];
        scanf("%s", expr);
        printf("Expression: %s\n", expr);

        char c;
        int i,j;
        int state = 0;
        struct action act;
        int* stack = (int*)malloc(5*sizeof(int));
        int stack_ptr = 1;
        int red;
        stack[0] = state;
        int buffer[100];
        int buf_pos=-1;
        printf("stack: ");
        printStack(stack, stack_ptr);
        for(i=0; expr[i]!='\0'; ++i){
                c = expr[i];
                j = char_to_col(c, lrt->at.symbols, lrt->num_term);
                act = lrt->at.table[state][j];
                switch (act.type)
                {
                        case t_accept:
                                reduceBuffer(buffer, &buf_pos, 1);
                                printf("Accepted\n");
                                return 0;
                                break;
                        case t_shift:
                                printf("shift: s%d\n", act.value);
                                state = act.value;
                                stack = append_int(c, stack, &stack_ptr);
                                stack = append_int(state, stack, &stack_ptr);
                                int k = c - '0';
                                if(k==0 || k==1){
                                        buf_pos++;
                                        buffer[buf_pos] = k;
                                }
                                printf("stack: ");
                                printStack(stack, stack_ptr);
                                printBuffer(buffer, buf_pos);
                                break;
                        case t_reduce:
                                printf("Rduce: r%d\n", act.value);
                                red = getReduction(act.value);
                                c = getRedChar(act.value);
                                stack_ptr -= red*2;
                                if(stack_ptr<0){
                                        printf("Error!\n");
                                        return 0;
                                }
                                reduceBuffer(buffer, &buf_pos, act.value);
                                printBuffer(buffer, buf_pos);
                                stack = append_int(c, stack, &stack_ptr);
                                j = char_to_col(c, lrt->gt.symbols, lrt->num_nonterm);
                                state = stack[stack_ptr-2];
                                state = lrt->gt.table[state][j];
                                stack = append_int(state, stack, &stack_ptr);
                                printf("stack: ");
                                printStack(stack, stack_ptr);
                                i--;
                                break;
                        default:
                                printf("Error!\n");
                                return 0;
                                break;
                }
        }
        return 0;
}

int* append_int(int n, int *arr, int *p){
        if(*p>0 && *p%5==0){
                int* a = (int*)malloc((*p+5)*sizeof(int));
                for(int i=0; i<*p; ++i){
                        a[i] = arr[i];
                }
                a[*p] = n;
                free(arr);
                *p = *p + 1;
                return a;
        }
        arr[*p] = n;
        *p = *p + 1;
        return arr;
}

int getReduction(int k){
        return rules[k-1].n;
}
```

```
char getRedChar(int k){
        return rules[k-1].c;
}

void printStack(int* stack, int n){
        for(int i=0; i<n; ++i){
                if((i&1)==0){
                        printf("%d ", stack[i]);
                }
                else printf("%c ", stack[i]);
        }
        printf("\n");
}

struct rule* appendRule(struct rule r, struct rule* _rules, int p){
        if(p>0 && p%5==0){
                struct rule* array = (struct rule*)malloc((p+5)*sizeof(struct
                 ↪  rule));
                for(int i=0; i<p; ++i){
                        array[i] = _rules[i];
                }
                array[p] = r;
                return array;
        }
        _rules[p] = r;
        return _rules;
}

void printBuffer(int buffer[], int buf_pos){
        printf("\t\t\t\tbuffer: ");
        for(int i=0; i<=buf_pos; ++i){
                printf("%d ", buffer[i]);
        }
        printf("\n");
}

void reduceBuffer(int buffer[], int* buf_pos, int rule){
        int pos = *buf_pos;
        switch (rule)
        {
                case 1:
                        printf("S = %d\n", buffer[pos]);
                        break;
                case 2:
                        buffer[pos-1] = buffer[pos-1] + buffer[pos];
                        *buf_pos = *buf_pos - 1;
                        printf("E = %d\n", buffer[pos-1]);
                        break;
                case 3:
                        buffer[pos-1] = buffer[pos-1] * buffer[pos];
                        *buf_pos = *buf_pos - 1;
                        printf("E = %d\n", buffer[pos-1]);
                        break;
                case 4:
                        printf("E = %d\n", buffer[pos]);
                        break;
                case 5:
                        printf("B = 0\n");
                        break;
                case 6:
                        printf("B = 1\n");
                        break;
                default:
                        break;
        }
}
```

II. Input

```
6
S 1
E 3
E 3
E 1
B 1
B 1
3
5
9
S E B
+ * 0 1 $
00 00 s3 s4 00
```

```
s5 s6 00 00 a0
r4 r4 00 00 r4
r5 r5 00 00 r5
r6 r6 00 00 r6
00 00 s3 s4 00
00 00 s3 s4 00
r2 r2 00 00 r2
r3 r3 00 00 r3
-1 1 2
-1 -1 -1
-1 -1 -1
-1 -1 -1
-1 -1 -1
-1 -1 7
-1 -1 8
-1 -1 -1
-1 -1 -1
1*1+1+1$
```

II.1 Output

```
How many rules are there ?: Enter rules properties: left(symbol) right(count).
 ↪  Eg. {A->Aa}=>{A 2}How many non-terms are there ?: How many terminals are
 ↪  there ?: How many states are there ?: Enter non terminals: Enter terminals:
 ↪  Enter action table in matrix form: 00=blank, si=shift i, ri=reduce i,
 ↪  a0=accept
Enter goto table in matrix form: -1=blank

Table:
|       |+        *        0        1        $        | S         E
 ↪  B        |
|    0  |                s3        s4                 |           1
 ↪  2        |
|    1  |s5        s6                         a        |
 ↪          |
|    2  |r4        r4                        r4        |
 ↪          |
|    3  |r5        r5                        r5        |
 ↪          |
|    4  |r6        r6                        r6        |
 ↪          |
|    5  |                s3        s4                |
 ↪  7        |
|    6  |                s3        s4                |
 ↪  8        |
|    7  |r2        r2                        r2        |
 ↪          |
|    8  |r3        r3                        r3        |
 ↪          |

Expression: 1*1+1+1$
stack: 0
shift: s4
stack: 0 1 4
                                buffer: 1
Rduce: r6
B = 1
                                buffer: 1
stack: 0 B 2
Rduce: r4
E = 1
                                buffer: 1
stack: 0 E 1
shift: s6
stack: 0 E 1 * 6
                                buffer: 1
shift: s4
stack: 0 E 1 * 6 1 4
                                buffer: 1 1
Rduce: r6
B = 1
                                buffer: 1 1
stack: 0 E 1 * 6 B 8
Rduce: r3
E = 1
                                buffer: 1
stack: 0 E 1
shift: s5
stack: 0 E 1 + 5
                                buffer: 1
```

```
shift: s4
stack: 0 E 1 + 5 1 4
                                buffer: 1 1
Rduce: r6
B = 1
                                buffer: 1 1
stack: 0 E 1 + 5 B 7
Rduce: r2
E = 2
                                buffer: 2
stack: 0 E 1
shift: s5
stack: 0 E 1 + 5
                                buffer: 2
shift: s4
stack: 0 E 1 + 5 1 4
                                buffer: 2 1
Rduce: r6
B = 1
                                buffer: 2 1
stack: 0 E 1 + 5 B 7
Rduce: r2
E = 3
                                buffer: 3
stack: 0 E 1
S = 3
Accepted
```