Институт приборостроения, автоматизации и информационных технологий

Кафедра информационных систем и цифровых технологий

Дисциплина «Теория языков программирования и методы трансляции»

Отчет к лабораторной работе № 6

«Построение дерева разбора SLR анализатора»

Выполнил:

Василения Иван Валерьевич

Мельниченко Артём Олегович

Принял:

Гордиенко А. П.

Орёл, 2024г

Листинг анализатора  
#ifndef ANALYZER\_H

#define ANALYZER\_H

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include "tokens.h"

#include "stack.h"

#include "tree.h"

#include "parse\_tables.h"

#define DEBUG\_MODE true

#define \_EOF 0

extern int yylex();

extern char\* yytext;

pair\_stack \*parse\_stack;

void reduce(); void shift();

void drop\_parse\_error() {

    printf("\033[31mParse error!!!!!!\033[0m\n");

    exit(EXIT\_FAILURE);

}

void show\_pair\_stack(pair\_stack\* s) {

    if (!DEBUG\_MODE) return;

    printf("Stack content: ");

    for (int i = 0; i <= s->head\_num; i++) {

        printf("\033[35m[St: %d, El: %d], \033[0m", s->pair[i].state, s->pair[i].token);

    }

    printf("\n");

}

bool is\_terminal(int tok) { return tok < NON\_TERMINAL\_TYPE\_BASE; }

bool is\_non\_terminal(int tok) { return tok >= NON\_TERMINAL\_TYPE\_BASE && tok < STATES\_TYPE\_BASE; }

bool is\_states\_type(int tok) { return tok >= STATES\_TYPE\_BASE && tok < REDUCE\_TYPE\_BASE; }

bool is\_reduce\_type(int tok) { return tok >= REDUCE\_TYPE\_BASE && tok < ACC; }

bool is\_accept(int tok) { return tok == ACC; }

void init() {

    parse\_stack = pair\_stack\_alloc();

}

void update\_curr\_state() {

}

void get\_string\_token(int term, char\* buffer){

    switch(term){

        case ATTR\_LIST:

            sprintf(buffer, "attr\_list");

            break;

        case WHERE\_CASE:

            sprintf(buffer, "where\_case");

            break;

        case CONDITION\_LIST:

            sprintf(buffer, "condition\_list");

            break;

        case CONDITION:

            sprintf(buffer, "condition");

            break;

        case GOAL:

            sprintf(buffer, "goal");

            break;

        case SELECT:

            sprintf(buffer, "SELECT");

            break;

        case \_STRING:

            sprintf(buffer, "STRING");

            break;

        case FROM:

            sprintf(buffer, "FROM");

            break;

        case WHERE:

            sprintf(buffer, "WHERE");

            break;

        case COMPARISON:

            sprintf(buffer, "COMPARISON");

            break;

        case NUM:

            sprintf(buffer, "NUM");

            break;

        case COMMA:

            sprintf(buffer, "COMMA");

            break;

        case OR:

            sprintf(buffer, "OR");

            break;

        case AND:

            sprintf(buffer, "AND");

            break;

        case LBRACKET:

            sprintf(buffer, "LBRACKET");

            break;

        case RBRACKET:

            sprintf(buffer, "RBRACKET");

            break;

        default:

            printf("unknown token %d!!\n", term);

            break;

    }

}

int curr\_token;

void parse\_start() {

    init();

    pair\_stack\_push\_pair(parse\_stack, S0, \_EOF);

    curr\_token = yylex();

    char token\_name[20];

    int token\_buffer[10];

    Node\* previous\_nodes[30];

    int total\_prev\_nodes = 0;

    while (1) {

        int curr\_state = pair\_stack\_get\_top\_state(parse\_stack);

        int action = action\_table[curr\_state - STATES\_TYPE\_BASE][curr\_token];

        if (is\_accept(action)) {

            printf("\033[32mВыражение принадлежит языку!!!!!!!!!!!!!!!!!!!!!!!\033[0m\n");

            for(int i = 0; i < total\_prev\_nodes; i++){

                Node\* child\_node = previous\_nodes[i];

                for(int j = i+1; j < total\_prev\_nodes; j++){

                    int child\_index = has\_child\_with\_token(previous\_nodes[j], child\_node->token);

                    if(child\_index != -1){

                        previous\_nodes[j]->children[child\_index] = child\_node;

                    }

                }

            }

            print\_tree(previous\_nodes[total\_prev\_nodes-1], 0);

            break;

        }

        else if (is\_states\_type(action)) {

            pair\_stack\_push\_pair(parse\_stack, action, curr\_token);

            curr\_token = yylex();

        }

        else if (is\_reduce\_type(action)) {

            int rule = action - REDUCE\_TYPE\_BASE;

            for (int i = 0; i < rules[rule][RULE\_LEN]; i++) {

                parse\_pair popped\_pair = pair\_stack\_pop\_pair(parse\_stack);

                token\_buffer[i] = popped\_pair.token;

            }

            get\_string\_token(rules[rule][RULE\_RES], token\_name); // rule result token

            Node\* node = create\_node(token\_name, rules[rule][RULE\_RES]);

            for (int i = rules[rule][RULE\_LEN] - 1; i >= 0; i--) {

                get\_string\_token(token\_buffer[i], token\_name);

                add\_child(node, create\_node(token\_name, token\_buffer[i]));

            }

            previous\_nodes[total\_prev\_nodes++] = node;

            int curr\_state = pair\_stack\_get\_top\_state(parse\_stack);

            int new\_state = goto\_table[curr\_state - STATES\_TYPE\_BASE][rules[rule][RULE\_RES] - NON\_TERMINAL\_TYPE\_BASE];

            get\_string\_token(rules[rule][RULE\_RES], token\_name);

            pair\_stack\_push\_pair(parse\_stack, new\_state, rules[rule][RULE\_RES]);

        }

        else {

            drop\_parse\_error();

        }

    }

}

void shift() {

    curr\_token = yylex();

}

Листинг файла tree.h

#ifndef TREE\_NODE\_H

#define TREE\_NODE\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_CHILDREN 10

typedef struct Node {

    char \*label;

    int token;

    struct Node \*children[MAX\_CHILDREN];

    int num\_children;

} Node;

void print\_tree(Node \*root, int depth) {

    if (root == NULL) return;

    for (int i = 0; i < depth; ++i) {

        if(depth - 1 == i) printf("- ");

        else printf("| ");

    }

    printf("%s\n", root->label);

    for (int i = 0; i < root->num\_children; ++i) {

        print\_tree(root->children[i], depth + 1);

    }

}

Node \*create\_node(char \*label, int token) {

    Node \*node = malloc(sizeof(Node));

    char\* node\_label = (char\*)malloc(strlen(label));

    strcpy(node\_label, label);

    node->token = token;

    node->label = node\_label;

    node->num\_children = 0;

    for (int i = 0; i < MAX\_CHILDREN; ++i) {

        node->children[i] = NULL;

    }

    return node;

}

int has\_child\_with\_token(Node\* node, int child\_token){

    if(node == NULL) return -1;

    for(int i = 0; i < node->num\_children; i++){

        if(node->children[i]->token == child\_token){

            return i;

        }

    }

    return -1;

}

void add\_child(Node \*parent, Node \*child) {

    if (parent->num\_children < MAX\_CHILDREN) {

        parent->children[parent->num\_children++] = child;

    } else {

        fprintf(stderr, "Maximum number of children reached.\n");

    }

}

#endif