Санкт-Петербургский политехнический университет
Институт компьютерных наук и технологий
Кафедра «Компьютерные системы и программные технологии»

К У Р С О В О Й П Р О Е К Т

Разработка парсера JSON (DOM)

по дисциплине «Прикладное программирование»

Выполнила
студентка гр.23531/1 М.А. Кузина

Руководитель
М.А. Петров

«___» _____ 2018 г.

Санкт-Петербург
2018

Санкт-Петербургский политехнический университет

ЗАДАНИЕ
НА ВЫПОЛНЕНИЕ КУРСОВОГО ПРОЕКТА

студентке группы 23531/1 Кузиной Марии Александровне

1. Тема проекта (работы): разработка парсера JSON (DOM)

2. Срок сдачи законченного проекта (работы) 02.06.2018

3. Исходные данные к проекту (работе):
Парсер должен посимвольно считывать полученный на вход файл в формате json и формировать внутреннее представление в виде древовидной структуры
Для проверки работы должен использоваться вывод в cmd.

4. Содержание пояснительной записки: введение, основная часть (раскрывается структура основной части), заключение, список использованных источников, приложения.

Дата получения задания: «___». _____ 2018 г.

Руководитель _____ М.А. Петров

Задание принял к исполнению _____ М.А. Кузина

_____
(дата)

СОДЕРЖАНИЕ

Введение. . . . . .4.

Методика решения…….....5

Код программы, реализующей парсер……..…….……6

Пример работы программы……11

Заключение 12

Список использованных источников 12

ВВЕДЕНИЕ

В данном курсовом проекте мною будет написана программа, являющаяся парсером формата JSON, что соответствует моему индивидуальному заданию по прикладному программированию.

В ней будут реализовано чтение файла в формате JSON посимвольно, формирование древовидного представления информации в памяти компьютера, а также вывод полученной структуры в cmd интерпретатор командной строки для проверки правильности работы программы.

МЕТОДИКА РЕШЕНИЯ

Программа осуществляет формирование в памяти древовидной структуры по входному файлу в формате json. Все данные считаются корректными, осуществляется проверка только на длину строки и количество детей (не обрабатывается, если длина строки больше 100 и детей больше 100)

```
struct Node{
    char type[STRMAX];  \\ тип узла
    char name[STRMAX]; \\ имя узла (ключ узла, если он содержится в hash)
    char value[STRMAX]; \\ значение простого узла
    int children_count; \\ кол-во детей
    struct Node *parent;
    struct Node *children[MAXCHILDREN];
};
```

Основной файл библиотеки json.c, его заголовочный файл json.h.

Основная функция библиотеки build_dom, которая принимает в себя указатель на указатель на родителя и файл, по которому необходимо построить древовидную структуру в формате json.

Входная точка функция build_dom. Она занимается разбором входного файла, в ней происходит инициализация вспомогательной структуры (state), которая отражает состояние парсера.

```
struct State{
    char ch; /текущий символ
    FILE *input_file; /файл, из которого происходит чтение
    char name[STRMAX]; /имя состояния
    char previous[STRMAX]; /имя предыдущего состояния
    char key[STRMAX]; /предыдущее строковое значение
    char token[STRMAX]; /текущий токен
    struct Node *current; /указатель на текущий узел
    struct Node *parent; /указатель на текущего родителя
    struct Node *root; /корневой узел дерева
    int token_length; /текущая длина токена
};
```

Затем build_dom обрабатывает входные данные и возвращает указатель на корневой узел, после чего очищает память (удаляет из памяти вспомогательные структуры).

Функция process посимвольно передает данные в функцию process_char. Внутри функции process_char игнорируются пробельные символы, разбираются только значимые. Основная логика – «грамматика» парсера, записана в функции process_char. По сути это конечный автомат с тремя состояниями:

- number;
- string;
- пустое состояние

На основе работы функций process_char и state_change_handler происходит разбор входного файла. При считывании очередного символа мы проверяем, в каком состоянии находимся.

Например, если сейчас состояние string, то при считывании очередного символа мы просто добавляем его в текущую строку, а если встретили закрывающую кавычку, то мы переходим в другое состояние. Аналогично, если мы считываем число, и встречаем цифру, то она считается частью текущего числа, а если встречаем какой-нибудь другой символ, то меняем состояние.

Если мы находимся в пустом состоянии, то представление о характере информации сформируется по встреченному нами символу. Например, если встретим открывающую скобку, то это начало нового hash или list. Таким образом, мы добавляем новый узел типа hash или list в дерево. В зависимости от того, что мы встретили – «:» или «,», мы интерпретируем текущую строку как значение ключа или просто строковую константу.

Для контроля работы используются функции печати print_node и print_input. Вывод древовидной структуры в cmd интерпретатор командной строки в соответствующих форматах: для типа число (number) – число, для типа строка (string) – строка в кавычках, для типа список (array) – перечисление отформатированных текстовых представлений содержащихся в списке узлов, для типа набора пар ключ-значение (hash) так же перечисление ключей и текстовых представлений узлов.

Чтобы использовать библиотеку, необходимо подключить json.h. Использование состоит в вызове одной функции build_dom, в которую необходимо передать указатель на корневой узел и входной файл для разбора). Функция возвращает значение типа int, которое интерпретируется стандартным образом: 0 – ошибок не было, 1 – произошла ошибка в процессе разбора данных, если ошибок не было, возвратит так же указатель на древовидную структуру. Пример использования (печать в cmd) находится в файле print_node.c. После использования необходимо очистить память функцией free_node.

Код программы, реализующей парсер, приложен (Приложение 1).

ПРИМЕР РАБОТЫ ПРОГРАММЫ

```
maria@maria-vb:~$ cd work/json/
maria@maria-vb:~/work/json$ make run
rm -rf *.o json
gcc -c json.c
gcc -c json_print.c
gcc json.o json_print.o -o json
./json
{"a":[{"a":3},{"b":"c"}]}

hash, children: 1
        list, a:, children: 2
                hash, children: 1
                        number, a: 3
                hash, children: 1
                        string, b: c
```

Рис1. Печать древовидной структуры в cmd

ЗАКЛЮЧЕНИЕ

В данном курсовом проекте мною была написана программа, реализующая парсер формата JSON.

Она поддерживает базовый функционал, необходимый для корректной работы программы, а также полностью соответствует заданным мне в индивидуальном задании требованиям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Герберт Шилдт. Полный справочник по C, 4-е издание. — Москва: Вильямс, 2004. — 800 с.
2. Интернет-ресурсы по программированию на C/C++, такие как stackoverflow (https://stackoverflow.com/)

ПРИЛОЖЕНИЕ 1

КОД ПРОГРАММЫ, РЕАЛИЗУЮЩЕЙ ПАРСЕР

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STRMAX 100
#define MAXCHILDREN 100

struct Node{
	char type[STRMAX];
	char name[STRMAX];
	char value[STRMAX];
	int children_count;
	struct Node *parent;
	struct Node *children[MAXCHILDREN];
};

struct State{
	char ch;
	FILE *input_file;
	char name[STRMAX];
	char previous[STRMAX];
	char key[STRMAX];
	char token[STRMAX];
	struct Node *current;
	struct Node *parent;
	struct Node *root;
	char last_key[STRMAX];
	int token_length;
	bool escape;
};

void print_node( char indent[STRMAX], struct Node *node );
void free_node( struct Node *node );
void print_self_node( struct Node *node );

void print_input( struct State *machine );
void init_state( struct State *machine );
void end_state( struct State *machine );
void process( struct State *machine );
void process_char( struct State *machine );

typedef void (*StateHandler)(struct State *);
typedef char String[STRMAX];
```

```
typedef struct State *Machine;
typedef struct Node *NodePtr;
typedef const char *StringPtr;

int main(){

	struct State *machine;
	machine =(struct State *) malloc (sizeof(struct State));

	print_input( machine );

	printf("\n\n");

	init_state( machine );
	process( machine );
	end_state( machine );

	return 0;
}

bool strings_equal( StringPtr str1, StringPtr str2 ){
	return strcmp( str1, str2 ) ==0; }

void print_node( char indent[STRMAX], struct Node *node ){
	printf( "%s%s", indent, node->type );
	if( !strings_equal( node->name, "" ) ) printf( ", %s:", node->name );
	if( !strings_equal( node->value, "" ) ) printf( " %s", node->value );
	if( node->children_count >0 ) printf( ", children: %d", node->children_count );
	printf( "\n" );
	char indent2[STRMAX];
	strncpy( indent2, indent, STRMAX );
	strcat( indent2, "  " );
	for( int i =0; i < node->children_count; i++ ){
		print_node( indent2, node->children[ i ] );
	}
}

void free_node( struct Node *node ){
	for( int i =0; i < node->children_count; i++ ){
		free_node( node->children[ i ] );
	}
	free( node );
}

void print_input( struct State *machine ){
	machine->input_file = fopen("input.json", "r"); // read mode
	while((machine->ch = fgetc(machine->input_file)) != EOF){
```

```
                printf("%c", machine->ch);
        }
        fclose(machine->input_file);
}

void init_state( struct State *machine ){
        machine->input_file = fopen("input.json", "r"); // read mode

        strncpy( machine->name, "", STRMAX );
        strncpy( machine->key, "", STRMAX );
        strncpy( machine->token, "", STRMAX );
        strncpy( machine->last_key, "", STRMAX );
        machine->token_length =0;

        machine->current =0;
        machine->parent =0;
        machine->escape =false;
}

void end_state( struct State *machine ){
        char indent[STRMAX] ="";
        print_node( indent, machine->root );

        free_node( machine->root );
        fclose(machine->input_file);
        free( machine );
}

void process( struct State *machine ){
        while((machine->ch = fgetc(machine->input_file)) != EOF){
                //printf("%c\n", ch);
                process_char( machine );
        }
}

bool check_state( Machine machine, StringPtr state ){
        return strings_equal( machine->name, state ); }

void string_copy( char *dest, StringPtr src ){
        strncpy( dest, src, STRMAX ); }

bool check_char( Machine machine, char ch ){
        return machine->ch ==ch; }

bool is_number_char( Machine machine ){
        return machine->ch =='.' || ( machine->ch >= '0' && machine->ch <= '9' ); }
```

```
void set_type( Machine machine, StringPtr type ){
	string_copy( machine->current->type, type ); }

bool is_token_state( StringPtr state ){
	return strings_equal( state, "string" ) || strings_equal( state, "number" ); }

void set_key( NodePtr node, StringPtr key ){
	string_copy( node->name, key ); }

void empty_token( Machine machine ){
	string_copy( machine->token, "" );
	machine->token_length =-1; }


void new_node( Machine machine, StringPtr type ){
	machine->current=(NodePtr) malloc (sizeof(struct Node));
	machine->current->children_count =0;
	set_type( machine, type );
	if( machine->parent !=0 ){
		machine->parent->children[ machine->parent->children_count ]
=machine->current;
		machine->parent->children_count++;
		machine->current->parent =machine->parent;
		set_key( machine->current, machine->key );
	}else{
		machine->parent = machine->current;
		machine->root = machine->current;
	}
	string_copy( machine->key, "" );
}

void state_change_handler( Machine machine, StringPtr old_state, StringPtr new_state
){
	if( machine->parent != 0 ){
		if( strings_equal( machine->parent->type, "list" )
			&& is_token_state( old_state ) ){
				string_copy( machine->key, "" );
				new_node( machine, old_state ); }
		if( strings_equal( machine->parent->type, "hash" )
			&& strings_equal( old_state, "string" ) ){
				string_copy( machine->key, machine->token ); }
		if( is_token_state( new_state ) ){
			if( machine->current != 0 ) set_type( machine, new_state ); } }
	if( machine->current != 0 ){
		if( is_token_state( old_state ) ){
			if( strings_equal( machine->current->type, "string" )
				|| strings_equal( machine->current->type, "number" ) ){
```

```
                        string_copy( machine->current->value, machine->token );
                        } } }
        empty_token( machine );
        if( strings_equal( new_state, "number" ) ){
                machine->token[0] =machine->ch;
                machine->token[1] =0;
                machine->token_length =1; }
        if( strings_equal( old_state, "number" ) ){
                process_char( machine ); } }

void set_state( Machine machine, StringPtr state ){
        String old_state ="";
        string_copy( old_state, machine->name );
        if( !check_state( machine, state ) ){
                string_copy( machine->name, state );
                state_change_handler( machine, old_state, state ); } }

void append_char( Machine machine ){
        if( machine->token_length <0 ) machine->token_length =0;
        machine->token[ machine->token_length ] =machine->ch;
        machine->token[ machine->token_length +1 ] =0;
        machine->token_length++; }

void process_char( Machine machine ){
        if( machine->ch == '\n' || machine->ch == ' ' )
                return;

        if( check_state( machine, "string" ) ){
                if( machine->escape ){
                        append_char( machine );
                        machine->escape =false; }
                else if( check_char( machine, '\\' ) ){
                        machine->escape =true; }
                else if( check_char( machine, '"' ) ){
                        set_state( machine, "" ); }
                else{
                        append_char( machine ); } }
        else if( check_state( machine, "number" ) ){
                if( is_number_char( machine ) ){
                        append_char( machine ); }
                else{
                        set_state( machine, "" ); } }
        else{
                if( check_char( machine, '{' ) || check_char( machine, '[' ) ){
                        String type ="hash";
                        if( check_char( machine, '[' ) ) string_copy( type, "list" );
                        if( machine->root ==0 ){
```

```
                    new_node( machine, type ); }
                else{
                    if( strings_equal( machine->parent->type, "list" ) ){
                        new_node( machine, type ); }
                    else{
                        set_type( machine, type );
                     }
                    machine->parent =machine->current; }
                machine->current =0; }
            else if( check_char( machine, '"' ) ){
                set_state( machine, "string" ); }
            else if( is_number_char( machine ) ){
                set_state( machine, "number" ); }
            else if( check_char( machine, ':' ) ){
                new_node( machine, "" ); }
            else if( check_char( machine, ',' ) ){
                machine->current =0;
                if( !strings_equal( machine->parent->type, "list" ) ){
                    machine->current =0; } }
            else if( check_char( machine, '}' ) || check_char( machine, ']' ) ){
                if( strings_equal( machine->parent->type, "list" )
                     && machine->token_length >= 0 ){
                     }
                machine->parent =machine->parent->parent;
                machine->current =0; }
        }
        strncpy( machine->previous, machine->name, STRMAX );
}
```