实验报告

姓名: 潘文博　　学号: 7203610819

一. 程序内容

本实验实现了一个 C- 语言的语法检查器，通过语法检查器，我们可以检查输入的 C- 代码是否符合预定的语法规则。程序主要实现了以下功能：

1. 词法分析：通过 Flex 工具和 lexical.l 文件定义了各种词法规则，将输入的代码字符串分解为一系列的 token。
2. 语法分析：通过 Bison 工具和 syntax.y 文件定义了各种语法规则，结合词法分析得到的 token，构建抽象语法树（AST）。
3. 抽象语法树（AST）的构建和打印：在 ast.h 和 syntax.y 文件中定义了 AST 的数据结构，并实现了构建和打印树的功能。
4. 语法错误处理：在 lexical.y 和 syntax.y 文件中定义了错误处理规则，当遇到语法错误时，程序会给出相应的错误提示。

具体实现如下：

- 在 lexical.l 文件中，定义了一系列的正则表达式规则，用于识别不同的 token，如整数、浮点数、标识符等。当识别到这些 token 时，会调用 create_terminal_node 函数创建一个终结符节点，并将其添加到 yylval 中以供语法分析使用。

```
0|[1-9][0-9]*|0x[0-9a-fA-F]+|0[1-7][0-7]+      { TNODE("INT",
yytext); return INT; }
([0-9]+\\.[0-9]*|\\.[0-9]+)([Ee][-+]?[0-9]+)|[0-9]+\\.[0-9]+ {
TNODE("FLOAT", yytext); return FLOAT; }
[a-zA-Z_][a-zA-Z0-9_]* { TNODE("ID", yytext); return ID; }
```

- 在 syntax.y 文件中，定义了一系列的语法规则，结合词法分析得到的 token，逐步构建抽象语法树。每当匹配到一个语法规则时，会调用 create_node 函数创建一个非终结符节点，并将其子节点添加到该节点中。

```
ExtDef: Specifier ExtDecList SEMI { $$=cnode("ExtDef", $1, 3, $1,
$2, $3); }
    | Specifier SEMI { $$=cnode("ExtDef", $1, 2, $1, $2); }
    | Specifier FunDec CompSt { $$=cnode("ExtDef", $1, 3, $1, $2,
$3); }
    | error SEMI { $$ = $2; }
    ;
```

- 在 ast.h 文件中，定义了 AST 的数据结构 Node，包括节点名称、行号、值字符串、是否为终结符、第一个子节点以及下一个兄弟节点等属性。同时提供了创建节点、创建终结符节点以及打印树的函数。

```
typedef struct Node {
    char *name;
    int lineno;
    char *value_string;
    int is_terminal;
    struct Node *first_child;
    struct Node *next_sibling;
} Node;

Node* create_node(char *name, int lineno, int num_children, Node
**children);

Node* create_terminal_node(char * name, int lineno, char* value);

void print_tree(Node *node, int level);
```

- 在 lexical.y 和 syntax.y 文件中，通过调用 yyerror 函数处理错误，根据错误类型给出相应的提示信息。例如，当出现语法错误时，会输出 "Error type B"，而当出现词法错误时，会输出 "Error type A"。

```
// lexical.y
.           { yyerror("lexical error"); }

// syntax.y
void yyerror(const char *s) {
    error_count++;
    root = NULL;
    if (strcmp(s, "syntax error") == 0) {
        fprintf(stderr, "Error type B at Line %d: syntax
error\\n", yylineno);
    } else if (strcmp (s, "lexical error") == 0) {
        fprintf(stderr, "Error type A at Line %d: Invalid
Character \\"%s\\".\\n", yylineno, yytext);
    }
}
```

二. 编译方法

程序的编译可以通过以下命令完成：

```
bison -d -t syntax.y && flex lexical.l && gcc -o grammar
syntax.tab.c lex.yy.c  && ./grammar testfile.c
```

这条命令包含了以下几个步骤：

1. 使用 bison 工具编译 syntax.y 文件，生成语法分析器文件 syntax.tab.c 和 syntax.tab.h 。
2. 使用 flex 工具编译 lexical.l 文件，生成词法分析器文件 lex.yy.c。
3. 使用 gcc 编译器将生成的词法分析器和语法分析器文件一起编译，生成可执行文件 grammar。
4. 运行 grammar 程序，以 testfile.c 为输入文件进行语法检查。

三. 程序特点

1. 优化的抽象语法树结构：程序中定义的抽象语法树结构包含了节点名称、行号、值字符串、是否为终结符等属性，这样的设计使得抽象语法树具有较高的可读性和可扩展性，便于后续对语法树进行进一步处理，如代码优化、目标代码生成等。
2. 简洁的编译命令：本程序提供了一条简洁的编译命令，将词法分析、语法分析和编译链接等多个步骤集成在一起，便于用户一次性完成整个编译过程，提高了使用体验。 精细化的优先级和结合性设置：在 syntax.y 文件中，程序通过 %left
3. 对浮点数和注释的处理: 本程序可以正确的处理科学记数法的浮点数，十六进制和八进制表示的整数以及单行/多行注释.

总结：

本实验实现了一个 C- 语言的语法检查器，具有良好的错误处理能力和清晰的模块化设计。通过实验，我们熟悉了 Flex 和 Bison 工具的使用，掌握了词法分析、语法分析和抽象语法树的构建等技术.