语法分析文档

设计目的

- 实现对 MiniC 语言的语法分析并输出语法分析树

实现方法

- yacc

代码设计

c 语言定义部分

```
%{
    #include "main.h"
    #include "utils.h"
    extern node* programNode;
    extern FILE* result;
    extern int yylineno;
    extern char* yytext;
    extern int yylex(void);
    extern "C"{
        void yyerror(const char *s);
        int yywrap(void);
}
%}
```

- main.h 头文件中包含了所需要的数据结构的定义

```
    typedef struct
    {
        std::string m_id; // string 类型存储id 的名字
        std::string m_reserved; // string 类型存储关键字
        int m_num; // int 类型 存储常量值
        std::string m_op; // string 类型 存储符号
        node *m_node; // node 存储语句
    }   ytype;
```

- utils.h 头文件中包含在生成语法树时所需要使用的新建节点、添加节点等功能函数

```
/**
 * @description: 新建一个 statement 类型的 node 并返回
 * @param {StmtKind}
 * @return: node*
 */
node *newStmtNode(StmtKind kind);

/**
 * @description: 新建一个 expression 类型的 node 并返回
 * @param {ExpKind}
 * @return: node*
 * @author: David.Huangjunlang
 */
node *newExpNode(ExpKind kind);

/**
 * @description: 在 nodeList 链表的末尾添加一个 statement 类型的 node
 * @param {node*, node*}
 * @return: void
 * @author: David.Huangjunlang
 */
void addNode(node *list, node *stmt);
```

- yylineno 为 token 的行号，yylex 为 scan 的入口函数，yytext 为 token 的值，yyerror()为解析错误时输出错误信息到目标文件 result，programNode 为程序的开始节点

```
/**
 * @description: 处理解析错误并打印到文件
 * @param {void}
 * @return: void
 * @author: David.Huangjunlang
 */
void yyerror(const char *s){
    fprintf(result ,"error: %s\n in line : %d\n unexpected token: %s\n", s,
yylineno, yytext);
    yywrap();
}
```

yacc 定义部分

- token 为终结符有 m_id, m_num, m_op, m_reserved 四种类型
- type 为非终结符,均为节点类型

```
%token <m_id> ID
%token <m_num> NUM
%token <m_op> PLUS MINUS MULTI DIVIDE
%token <m_op> LESS LESSEQUAL GREATER GREATEREQUAL EQUAL UNEQUAL
%token <m_op> ASSIGNMENT SEMICOLON COMMA
%token <m_op> LEFTBRACKET RIGHTBRACKET LEFTSQUAREBRACKET RIGHTSQUAREBRACKET
LEFTBRACE RIGHTBRACE
%token <m_reserved> ELSE IF INT VOID RETURN WHILE

%type <m_node> declaration fun_declaration var_declaration param expression
simple_expression additive_expression var term factor call program
%type <m_node> expression_stmt compound_stmt statement selection_stmt
iteration_stmt return_stmt
%type <m_node> params local_declarations args
%type <m_node> statement_list param_list declaration_list
```

- program 是程序的开始,只有一个列表的子节点，是一些列的定义，有可能只有一个定义，也有可能有多个，所以是一个列表
- declaration_list 是函数(fun_declaration)、变量(var_declaration)的定义列表，有可能为一个或多个

```
program : declaration_list {$$ = newStmtNode(ProgramK); $$->listChild[0] =
$1;programNode = $$; programNode->name = "helloworld";}
    ;

declaration_list : declaration_list declaration {addNode($1, $2);$$ = $1;}
    |   declaration {$$ = newStmtNode(DeclK); addNode($$, $1);}
    ;

declaration : var_declaration {$$ = $1;}
    | fun_declaration {$$ = $1;}
    ;
```

- fun_declartion,有三个子节点，第一个是 id 是函数的名字，第二个是 params 是参数列表，第三个是组成语句 compound_stmt,函数的返回类型有可能是 int 也有可能是 void
- compound_stmt 有两个子列表节点，local_declaration 是局部定义声明，以及 statment_list 是语句列表，均有可能为空

```
fun_declaration : VOID ID LEFTBRACKET params RIGHTBRACKET compound_stmt{$$ =
newStmtNode(VoidFundecK);
                                                                       $$->name =
$2; $$->listChild[0] = $4;
                                                                       $$-
>nodeChild[0] = $6;}
    |   INT ID LEFTBRACKET params RIGHTBRACKET compound_stmt{$$ =
newStmtNode(IntFundecK);
                                                                       $$->name =
$2; $$->listChild[0] = $4;
                                                                       $$-
>nodeChild[0] = $6;}


    ;

params : param_list {$$ = $1;}
    |   VOID {$$ = nullptr;}
    ;

param_list : param_list COMMA param {addNode($1, $3); $$ = $1;}
    |   param {$$ = newStmtNode(ParamlK); addNode($$, $1);}
    ;

param : INT ID {$$ = newExpNode(IdK); $$->name = $2;}
    |   INT ID LEFTSQUAREBRACKET RIGHTSQUAREBRACKET {$$ = newExpNode(ArrayptrK);
$$->name = $2;}
    ;

compound_stmt : LEFTBRACE local_declarations statement_list RIGHTBRACE {$$ =
newStmtNode(CompK);
                                                                         $$-
>listChild[0] = $2;
                                                                         $$-
>listChild[1] = $3;}
    |   LEFTBRACE local_declarations RIGHTBRACE {$$ = newStmtNode(CompK); $$-
>listChild[0] = $2;}
    |   LEFTBRACE statement_list RIGHTBRACE {$$ = newStmtNode(CompK); $$-
>listChild[1] = $2;}
    ;
```

```
local_declarations : local_declarations var_declaration SEMICOLON {addNode($1, $2); $$ = $1;}
    |   var_declaration COMMA {$$ = newStmtNode(LocdeclK); addNode($$, $1);}
    |   {}
    ;

statement_list : statement_list statement {addNode($1, $2); $$ = $1;}
    |   statement {$$ = newStmtNode(StmtlK); addNode($$, $1);}
    ;
```

- statement 有表达式语句，if 语句，while 语句以及返回语句,还有复合语句
- 表达式语句 由简单语句以及分号组成
- if 语句有可能有两个或存在 else 时有三个子节点，第一个是表达式类型，第二个和第三个为语句类型的子节点
- while 语句有两个子节点，第一个为表达式类型，第二个为语句类型
- return 语句 只有一个表达式节点，并有可能为空

```
statement : expression_stmt {$$ = $1;}
    |   selection_stmt {$$ = $1;}
    |   compound_stmt {$$ = $1;}
    |   iteration_stmt {$$ = $1;}
    |   return_stmt {$$ = $1;}
    ;

expression_stmt: expression SEMICOLON {$$ = newStmtNode(ExpressionK); $$->nodeChild[0] = $1;}
    |   SEMICOLON {$$ = newStmtNode(ExpressionK);}
    ;

selection_stmt : IF LEFTBRACKET expression RIGHTBRACKET statement {$$ = newStmtNode(SelectK); $$->nodeChild[0] = $3; $$->nodeChild[1] = $5;}
    |   IF LEFTBRACKET expression RIGHTBRACKET statement ELSE statement {$$ = newStmtNode(SelectK); $$->nodeChild[0] = $3; $$->nodeChild[1] = $5; $$->nodeChild[2] = $7;}
    ;

iteration_stmt : WHILE LEFTBRACKET expression RIGHTBRACKET statement {$$ = newStmtNode(IteraK); $$->nodeChild[0] = $3; $$->nodeChild[1] = $5;}
    ;

return_stmt : RETURN SEMICOLON {$$ = newStmtNode(ReturnK);}
    |   RETURN expression SEMICOLON {$$ = newStmtNode(ReturnK); $$->nodeChild[0] = $2;}
    ;
```

- expression 有两种，一种赋值语句，一种简单表达式
- 赋值语句有两个子节点，第一个是变量子节点，第二个是表达式节点

```
expression : var ASSIGNMENT expression {$$=newExpNode(AssignK); $$->nodeChild[0] = $1; $$->nodeChild[1] = $3;}
    |  simple_expression {$$ = $1;}
```

- 简单语句有比较语句，加减语句，乘除语句，函数调用语句，以及常量，变量的调用

```
var : ID {$$ = newExpNode(IdK); $$->name = $1;}
    |   ID LEFTSQUAREBRACKET expression RIGHTSQUAREBRACKET {$$ = newExpNode(IndexK); $$->nodeChild[0] = $3;}
    ;

simple_expression : additive_expression LESSEQUAL additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0] = $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression LESS additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0]= $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression GREATER additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0]= $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression GREATEREQUAL additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0]= $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression EQUAL additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0]= $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression UNEQUAL additive_expression {$$ = newExpNode(OpK); $$->nodeChild[0]= $1; $$->nodeChild[1] = $3; $$->op = $2;}
    |   additive_expression {$$ = $1;}
    ;


additive_expression : additive_expression PLUS term  { $$ = newExpNode(AddK); $$->op = $2; $$->nodeChild[0] = $1; $$->nodeChild[1] = $3;}
    |   additive_expression MINUS term  { $$ = newExpNode(AddK); $$->op = $2; $$->nodeChild[0]= $1; $$->nodeChild[1] = $3;}
    |   term                     { $$ = $1;}
    ;

term : term MULTI factor {$$ = newExpNode(MulK); $$->op = $2; $$->nodeChild[0] = $1; $$->nodeChild[1] = $3;}
    |   term DIVIDE factor {$$ = newExpNode(MulK); $$->op = $2; $$->nodeChild[0] = $1; $$->nodeChild[1] = $3;}
    |   factor {$$ = $1;}
    ;

factor : LEFTBRACKET expression RIGHTBRACKET {$$ = $2;}
    |   var {$$ = $1;}
    |   call {$$ = $1;}
    |   NUM {$$ = newExpNode(ConstK); $$->val = $1;}
    ;

call : ID LEFTBRACKET args RIGHTBRACKET {$$ = newExpNode(CallK); $$->listChild[0] = $3; $$->name = $1;}
    |   ID LEFTBRACKET RIGHTBRACKET {$$ = newExpNode(CallK); $$->name = $1;}
    ;

args : args COMMA expression {addNode($1, $3); $$ = $1;}
    |   expression {$$ = newStmtNode(ArgsK); addNode($$, $1);}
    ;
%%
```

测试数据

```
void hello(void){return;}
/* hello*/
```

```
int gcd(int u, int v){
    if (v == 0)return u;
    else return gcd(v, u-u/v*v);
    /* test */
}

void main(void){
    int x, int y;
    x = input();
    y = input();
    output(gcd(x, y));
}
```

测试结果

```
fun-declaration:
    type: void
    name: hello
    params: void
    compound:
        local-declarations: null
        statement-list:
            return-stmt:
                expression: null
fun-declaration:
    type: int
    name: gcd
    params:
        var:
            name: u
        var:
            name: v
    compound:
        local-declarations: null
        statement-list:
            selection-stmt:
                expression:
                    simple-expression:
                        first:
                            var:
                                name: v
                        op: ==
                        second:
                            const:
                                val: 0
                statement:
                    return-stmt:
                        expression:
                            var:
                                name: u
                else-statement:
                    return-stmt:
                        expression:
                            functionCall:
                                name: gcd
                                args:
                                    var:
                                        name: v
                                    additive-expression:
                                        first:
                                            var:
                                                name: u
                                        op: -
                                        second:
                                            term:
                                                first:
                                                    term:
                                                        first:
                                                            var:
                                                                name: u
                                                        op: /
                                                        second:
                                                            var:
                                                                name: v
                                                op: *
                                                second:
                                                    var:
                                                        name: v
fun-declaration:
    type: void
    name: main
    params: void
    compound:
        local-declarations:
            var:
                name: x
            var:
                name: y
        statement-list:
            assignment:
                varName: x
                expression:
                functionCall:
                    name: input
                    args: void
            assignment:
                varName: y
                expression:
                functionCall:
                    name: input
                    args: void
            functionCall:
                name: output
                args:
                    functionCall:
                        name: gcd
                        args:
                            var:
                                name: x
                            var:
                                name: y
```