编译框架

本框架用于对字符串解析然后生成语法分析树，然后根据指令集对语法树深度优先遍历生成二进制机器码。

当文法和词法结构发生改变的时候，修改lex和yacc重新生成代码。

Tree.h描述的是语法树结构。

CodeGenerateAPI.h声明代码生成接口，需要实现`std::string GenerateCode(std::string str)`和`std::string GetVarAddr(char var)`。此接口和Tree是耦合的，我也很无奈。

CodeGenerateAPIImpl.h声明一个实现，对应20160521版本的汇编指令集。

当汇编指令集发生改变时候，只需要修改此实现。

lex 和 yacc设置

先设定好文法规则和词法规则。

lex设置 lexya_a.l

```
%{
  #include<stdlib.h>
  #define YY_NO_UNISTD_H 1
  void yyerror(char *);
  #include "lexya_a.tab.h"
  #define YY_DECL int yylex (char* str)
  #define YY_USER_INIT if ( ! YY_CURRENT_BUFFER ) {\
        yyensure_buffer_stack ();\
        YY_CURRENT_BUFFER_LVALUE = yy_scan_string(str);\
        }
%}
%%
([1-9][0-9]+)|([0-9])       { yylval.Double = atof(yytext); return
INTEGER; }
[-+*/]       {return *yytext;}
[\t]         {}
[abcd]        {yylval.Char=*yytext;return VARA;}
[(]           {return *yytext;}
[)]           {return *yytext;}
.            {yyerror("��ч��");}
[\n]        {return *yytext;}
%%
int yywrap(void)
{
  return 1;
}
```

yacc设置 lexya_a.y

```
%{
#include <stdlib.h>
#ifndef TREE_H
#include "Tree.h"
#endif
#define YYLEX_PARAM str
int yylex(char* str);
void yyerror(char *);
static Tree* TreePtr;
%}
%union {
    Node* node;
    Tree* tree;
    char Char;
    double Double;
}
%token <Double> INTEGER
%token <Char> VARA
%left '+' '-'
%left '*' '/'
%right UMINUS

%type <node> expr
%%
program : program expr '\n' { TreePtr=new Tree($2);return 1;}
        |
        ;
expr : INTEGER { $$ = new Number($1); }
     | expr '*' expr { $$ = new BinaryOperator($1,$3,'*');}
     | expr '/' expr { $$ = new BinaryOperator($1,$3,'/');}
     | expr '+' expr { $$ = new BinaryOperator($1,$3,'+');}
     | expr '-' expr { $$ = new BinaryOperator($1,$3,'-');}
     | '('expr')'{$$=$2;}
     | '-' expr %prec UMINUS{$$ = new RightOperator($2,'-');}
     | VARA { $$ = new Var($1);}
     ;
%%
void yyerror(char *s)
{
  printf("%s\n", s);
}
Tree* LEGO_Parse(char* str)
{
  if(TreePtr!=NULL){
      TreePtr->Clear();
      TreePtr=NULL;
  }
  yyparse(str);
  return TreePtr;
}
```

生成代码

执行`win_flex.exe --wincompat lexya_a.l`得到`lex.yy.c`。

执行`win_bison.exe -d lexya_a.y`得到`lexya_a.tab.c`和`lexya_a.tab.h`。

由于c和c++语法相近，为了避免编译时候奇奇怪怪问题，直接修将.c修改成.cpp。

修改生成的代码

lexya_a.tab.h

首先检查候头文件`Tree.h`的引用是否规范。

```
#ifndef TREE_H
#include "Tree.h"
#endif
```

检查函数`yyparse()`的定义

```
int yyparse (char *str);
```

在文件最后添加函数声明`Tree* LEGO_Parse(char* str);`

修改生成文件`lexya_a.tab.c`
修改`yyparse()`定义

lexya_a.tab.cpp

在文件最下面找到如下代码

```
Tree* LEGO_Parse(char* str)
{
  if(TreePtr!=NULL){
      TreePtr->Clear();
      TreePtr=NULL;
  }
  yyparse(str);
  return TreePtr;
}
```

右键`yyparse(str);`找到定义，检查定义是否为

```
int
yyparse (char* str)
```

接口函数

再通过LEGOP.h里面的Tree* LEGOP(char* str)就可以获得语法树。失败的时候返回NULL。

使用

在需要进行语法分析的类里面，包含CodeGenerateAPIImpl.h。然后构造MemoryInfo实例，分别将

```
1       std::map<char, std::string> VarNameToAddr;
2       std::set<std::string> TempAddrs;
3       std::string ResultAddr;
```

设置好，用于构造CodeGenerateAPIImpl实例。

然后设置解析错误的回调函数SetErrorCallback(回调函数指针)

“注意回调函数是类函数的话要声明成static。”

调用std::string GenerateCode(std::string str)获得生成的二进制机器码。“str中\n表示末尾，所以请只在末尾加\n”。

相关说明

yacc里面的一些设置

设置符号值的类型

将数据结构定义到外部的头文件中，在 yacc 中 include 进去

在 yacc 中，所有的符号都会有一个对应的值与之相关联。这个值默认是int。

通过将类型声明添加到 union 中，我们可以添加其他类型进去。

数据结构需要在 action 中使用就必须在定义符号之前包含在 union 体中

```
%union {
    Node *node;
    NBlock *block;
    NExpression *expr;
    NStatement *stmt;
    NIdentifier *ident;
    NVariableDeclaration *var_decl;
    std::vector *varvec;
    std::vector *exprvec;
    std::string *string;
    int token;
}
```

即使是使用的类类型要明确写进去，如果用子类就写子类，用父类就写父类。

> *union 联合体很像 struct，其作用就是让这个联合体类型的变量拥有足够空间去容纳联合体内的任一类型。*

在定义符号的时候，就要告诉yacc，这个符号是什么类型的。对于终结符使用 `%token <类型> 符号` 来定义，对于非终结符使用 `type <类型> 符号` 。要注意的是，类型填的是 `union` 中的变量名，不是变量类型。（也就是填 node，不要填 Node*）。

如果不需要使用某个符号的值的时候，大可以不用定义类型，不过定义符号还是要的。当 yacc 中存在 union ，所有所有使用值的符号都需要类型定义（就不会默认是 int 了）。

action 部分

`$$` 表示产生式左边符号的值，`$x` 表示产生式右边第x个符号（可以是终结符也可以是非终结符）。所以打印结果的加法式子 `E->a+b` action应该这样写`{printf(%d =%d + %d);}`。