浙江大学

本科实验报告

课程名称：编译原理

陈健 3200102329

XXX 3XXXXXXXX

学　　院：计算机学院

专　　业：计算机科学与技术

指导老师：李莹

2023 年 5 月 28 日

浙江大学实验报告

0.序言

本次实验完成了一个C语言的编译器，能够分析部分C语言的语法，并将其编译至自己实现的IR代码，最后再编译至目标代码。我们实现的C编译器支持以下C语言特性：

- **所有C语言基本语句。**包括`if`, `for`, `while`, `do`, `switch`, `case`, `break`, `continue`, `return`。
- **所有C语言表达式。**包括括号`()`, 数组下标`[]`, `sizeof`, 函数调用, 结构体`->`, `.`, 一元`+`, 一元`-`, 强制类型转换, 前缀`++`, 前缀`--`, 后缀`++`, 后缀`--`, 取地址`&`, 取内存`*`, 位运算`&`, `|`, `~`, `^`, 逻辑运算`||`, `&&`, `!`, 比较运算`>`, `>=`, `<`, `<=`, `==`, `!=`, 算术运算`+`, `-`, `*`, `/`, `%`, 移位运算`<<`, `>>`, 赋值语句`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `|=`, `&=`, `^=`, 逗号表达式`,`, 三元运算符`?:`。
- **基础类型系统。**基本数据类型包括`bool`, `char`, `short`, `int`, `long`, `float`, `double`, `void`，并实现了数组与简单的指针类型。
- **类型转换。**本实验中默认实现了一部分的强制类型转换。
- **符号表作用域。**我们的编译器允许在`if`, `for`, `while`, `do`, `switch`以及语句块`{}`内定义变量，并且可以覆盖外层作用域的重名变量。变量的作用域只在语句块内。
- **编译优化。**可使用优化函数接口对编译结果进行一定程度上的优化。

0.1 依赖项

1. Flex & Bison

   我们使用flex和bison生成词法分析器和语法分析器。

2. C++开发环境

3. Python开发环境

0.2 使用说明

- 得到语法树与IR code：仅需编译代码中.cpp文件并使用编译后的可执行文件执行对应的test.c文件即可在输出中得到语法树与IR code。
- 得到汇编：修改IR code的输出为文件，并使用codeGen.py将其中内容转化为MIPS汇编。

0.3 代码规范

为了对变量加以区分，我们做如下代码规范：

- lex分析得到的token均由大写字母组成。
- 本实验中所有要用到的数据结构均在tree.h中进行定义。
- 节点类型与yacc语法均采用驼峰命名。

0.4 分工

- **lex & IR生成 & 目标代码生成 & 语法树搭建**

  陈健

- **yacc & Parser**

  舒晴

1、词法分析

编译器的词法分析（lexical analysis）阶段可将源程序读作字符文件并将其分为若干个记号。典型的记号有：关键词（key word），例如`if`和`while`，它们是字母的固定串；标识符（identifier）是由用户定义的串，它们通常由字母和数字组成并由一个字母开头；特殊符号（special symbol）如算数符号`+`和`*`、一些多字符符号，如`>=`和`<>`。

1.1 正则表达式

Lex是一个词法分析程序生成器，其输入为包含了正则表达式的`.l`文件和每个表达式被匹配时采取的动作。其中正则表达式的Lex约定如下：

格式	含义
a	字符a
"a"	即使a是一个元字符，它仍是字符a
\a	即使a是一个元字符，它仍是字符a
a*	a的零次或多次重复
a+	a的一次或多次重复
a?	一个可选的a
a|b	a或b
(a)	a本身
[abc]	字符a、b或c中的任一个
[a-d]	字符a、b、c或d中的任一个
[^ab]	除了a或b外的任一个字符
.	除了新行之外的任一个字符
{xxx}	名字xxx表示的正则表达式

1.2 Lex具体实现

1.2.1 定义

```
D           [0-9]
L           [a-zA-Z_]
H           [a-fA-F0-9]
E           ([Ee][+-]?{D}+)
P           ([Pp][+-]?{D}+)
FS          (f|F|l|L)
IS          ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))
```

首先，通过如上定义，我们将数字、字母等可能出现的符号进行划分，便于后续表达。

```
void Comment() {
    column = 0;
    char c, prev = 0;

    while (cin >> c)      /* (EOF maps to 0) */
    {
        if (c == '/' && prev == '*')
            return;
        prev = c;
    }
    printf("unterminated comment");
}

void Count(void)
{
    int i;

    for (i = 0; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n')
            column = 0;
        else if (yytext[i] == '\t')
            column += 4 - (column % 4);
        else
            column++;
    ECHO;
}
```

——Comment函数用以处理/*注释的后半段

——Count函数用来统计此时的代码列数

1.2.2 具体规则（仅展现部分进行举例）

```
"/*"            { Comment();/*注释,词法接收时仅接纳前半部分,通过Comment函数进行后半部分的处理
*/}
"//"[^\n]*      { /* 直接消除 //-comment */ }

"bool"          { Count(); yylval.nd =create_tree("BOOL",0,yylineno);
return(BOOL);   }
"int"           { Count(); yylval.nd =create_tree("INT",0,yylineno);
return(INT); }
"char"          { Count(); yylval.nd =create_tree("CHAR",0,yylineno);
return(CHAR); }
"double"        { Count(); yylval.nd =create_tree("DOUBLE",0,yylineno);
return(DOUBLE); }
……
```

——消除注释语句，并对基本的C语言标识符进行识别。

```
{L}({L}|{D})*   {Count(); yylval.nd =create_tree("IDENTIFIER",0,yylineno);
return(IDENTIFIER);/*IDENTIFIER*/ }

0[xX]{H}+{IS}?  { Count(); yylval.nd =create_tree("CONSTANT_INT",0,yylineno);
return(CONSTANT_INT); /*16进制*/}
0[0-7]*{IS}?    { Count(); yylval.nd =create_tree("CONSTANT_INT",0,yylineno);
return(CONSTANT_INT); /*8进制*/}
[1-9]{D}*{IS}?  { Count(); yylval.nd =create_tree("CONSTANT_INT",0,yylineno);
return(CONSTANT_INT); /*10进制*/}
L?'(\\.|[^\\'\n])+'     { Count(); return(CONSTANT); }

{D}+{E}{FS}?                { Count(); yylval.nd
=create_tree("CONSTANT_DOUBLE",0,yylineno);    return(CONSTANT_DOUBLE); /*浮点数*/}
{D}*"."{D}+{E}?{FS}?        { Count(); yylval.nd
=create_tree("CONSTANT_DOUBLE",0,yylineno);    return(CONSTANT_DOUBLE); /*浮点数*/}
{D}+"."{D}*{E}?{FS}?        { Count(); yylval.nd
=create_tree("CONSTANT_DOUBLE",0,yylineno);    return(CONSTANT_DOUBLE); /*浮点数*/}
0[xX]{H}+{P}{FS}?           { Count(); return(CONSTANT); }
0[xX]{H}*"."{H}+{P}?{FS}?   { Count(); return(CONSTANT); }
0[xX]{H}+"."{H}*{P}?{FS}?   { Count(); return(CONSTANT); }

L?\"(\\.|[^\\"\n])*\"   { Count();  yylval.nd
=create_tree("STRING_LITERAL",0,yylineno);    return(STRING_LITERAL); /*字符串常量*/}
```

——对变量/函数名、常量值进行识别，需要注意的是，在进行编译时对除十进制以外的值需要进行额外的0x(X)/0o(O)标识。

```
">>="    { Count(); yylval.nd =create_tree("RIGHT_ASSIGN",0,yylineno);
return(RIGHT_ASSIGN); }
"<<="    { Count(); yylval.nd =create_tree("LEFT_ASSIGN",0,yylineno);
return(LEFT_ASSIGN); }
"+="     { Count(); yylval.nd =create_tree("ADD_ASSIGN",0,yylineno);
return(ADD_ASSIGN); }
"-="     { Count(); yylval.nd =create_tree("SUB_ASSIGN",0,yylineno);
return(SUB_ASSIGN); }
"*="     { Count(); yylval.nd =create_tree("MUL_ASSIGN",0,yylineno);
……
```

——对运算符进行识别。

此外，在lex文件中我们需要使用yytext以传递文本信息，使用yylval存储建立语法树需要的节点数据`treeNode`，使用yylineno存储并传递行号。

2、语法分析

语法分析程序从扫描程序中获取记号形式的源代码，并完成定义程序结构的语法分析，这与自然语言中句子的语法分析类似。语法分析定义了程序的结构元素及其关系。语法分析的结果表示为抽象语法树（Abstract Syntax Tree）。

2.1 支持语法

我们的CFG语法支持如序言所提的C语言特性，但是部分语法有所区别：

- 不支持宏定义。不支持`#include`和`#define`等宏定义。除print与read外的内置函数均需要自行定义。
- 所有代码应在一个源文件中。
- 指针类型不需声明。C语言中有各类方式识别，但在我们的实现中仅支持使用形如`*a`的表达进行指针运用，因此也无法将此类指针运用在复杂的数据结构中。

2.2 Yacc

我们使用Yacc作为我们的分析程序生成器，其输入是一个说明文件（`.y`后缀），并产生一个由分析程序的C源代码组成的输出文件，格式为：

```
{definations}
%%
{rules}
%%
{auxiliary routines}
```

2.2.1 符号声明部分

在yacc中，我们需要声明进行文法推导时所用到的符号，其中包含：

- token

```
%token <nd> IDENTIFIER CONSTANT STRING_LITERAL SIZEOF CONSTANT_INT CONSTANT_DOUBLE
%token <nd> PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token <nd> AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token <nd> SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token <nd> XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token <nd> CHAR INT DOUBLE VOID BOOL

%token <nd> CASE IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%token <nd> TRUE FALSE

%token <nd> ';' ',' ':' '=' '[' ']' '.' '&' '!' '~' '-' '+' '*' '/' '%' '<' '>' '^'
'|' '?' '{' '}' '(' ')'
```

——其中包含了lex所返回的所有值，并使用为数据格式。

```
%union{
    struct treeNode* nd;
}
```

- type

```
%type <nd> primary_expression postfix_expression argument_expression_list
unary_expression unary_operator
%type <nd> multiplicative_expression additive_expression shift_expression
relational_expression equality_expression
%type <nd> and_expression exclusive_or_expression inclusive_or_expression
logical_and_expression logical_or_expression
%type <nd> assignment_expression assignment_operator expression

%type <nd> declaration init_declarator_list init_declarator type_specifier

%type <nd> declarator

%type <nd> parameter_list parameter_declaration identifier_list
%type <nd> abstract_declarator initializer initializer_list designation
designator_list
%type <nd> designator statement labeled_statement compound_statement block_item_list
block_item expression_statement
%type <nd> selection_statement iteration_statement jump_statement translation_unit
external_declaration function_definition
%type <nd> declaration_list
```

——其中包含了文法推导时用到的各级表达式，如赋值、声明等。

2.2.2 规则部分

每一条规约首行为规则，花括号内为C语言操作。`$$`为规约后压入栈的值，`$1`，...，`$n`为规约前栈中的值。如下为范例：

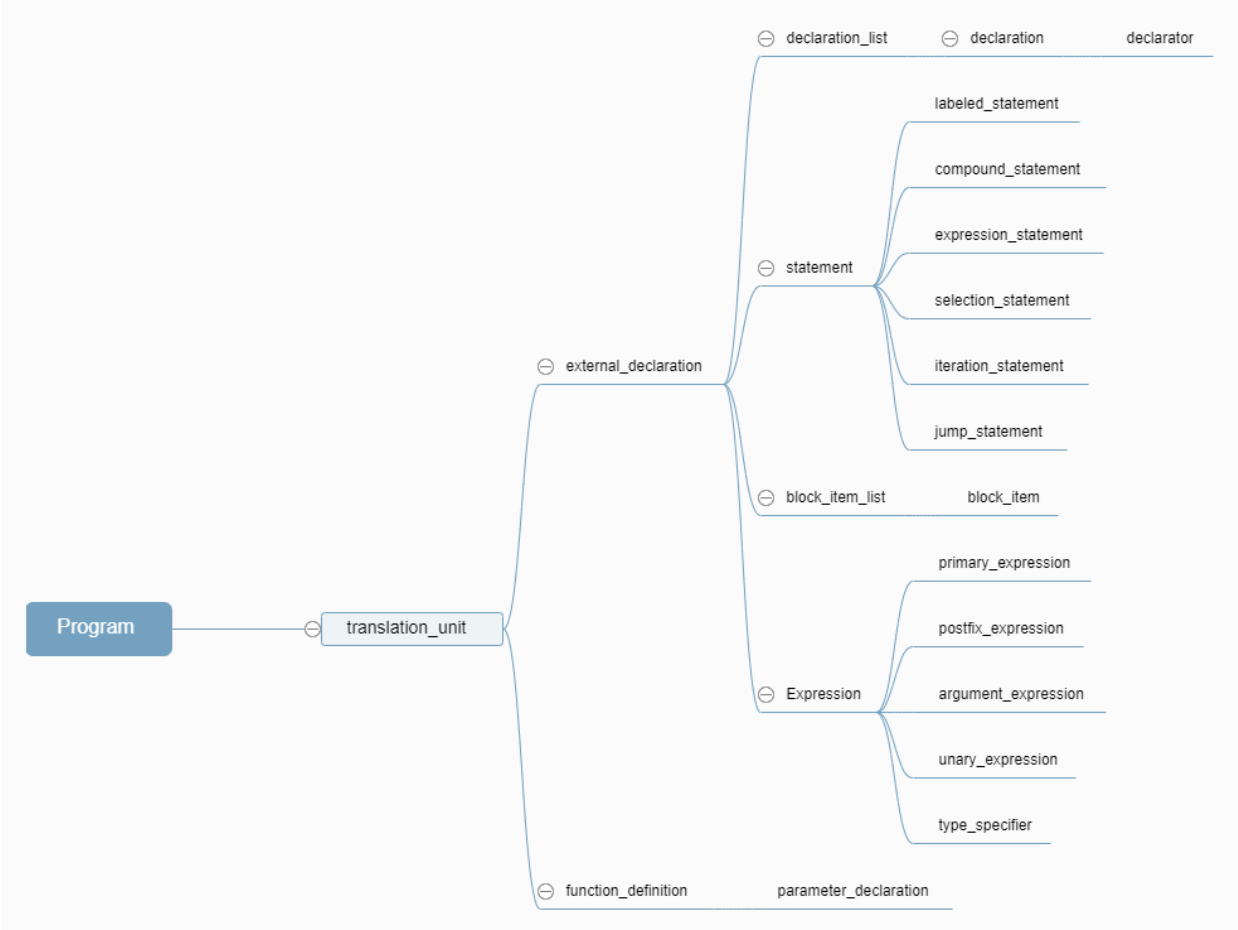
```
Program:
    translation_unit {
        root = create_tree("Program",1,$1);
    }
    ;

primary_expression:
    IDENTIFIER {
        $$ = create_tree("primary_expression",1,$1);
    }
... ...
```

由于在lex词法分析中已完成了相对完善的识别工作，因此在语法分析时使用单独的左递归已经足够完成相应的语法构建。

2.3 抽象语法树

语法分析程序的输出是抽象语法树，即称作抽象语法的快速计数法的树形表示。抽象语法树的每一个节点表示一种类型，我们定义的类型之间的继承关系如下（篇幅原因无法完全绘制导图，因此进行了可视化调整）：



2.3.1 数据结构描述

```
struct treeNode {
    string content;
    string name;
    int line;       //所在代码行数
    vector<struct treeNode *> sibs;
    struct treeNode *parent;
};
```

`treeNode`结构体中包含该节点名称（name）、内容（content）、所在代码行数（line）、子节点容器（sibs）、父节点（parent）。

2.3.2 函数解析

```
treeNode* createTree(string name, int num,...) {
    va_list valist;
    treeNode* head = new treeNode();
    if(!head) {
        //内存不足
        printf("空间不足 \n");
        exit(0);
    }
    head->content = "";
    head->name = name;
    va_start(valist,num);
    //对于多节点情况，如果仅为 name-content 形式，则直接修改内容，否则使其与后续节点一同进入子节点的vector
    if(num > 0) {
        treeNode* temp = NULL;
        temp = va_arg(valist,treeNode*);
        temp->parent = head;
        head->sibs.push_back(temp);
        head->line = temp->line;
        if(num == 1) {
            if(temp->content.size() > 0)
                head->content = temp->content;
        }
        else {
            for(int i = 1; i < num; ++i ) {
                treeNode* tempForRight = NULL;
                tempForRight = va_arg(valist, treeNode *);
                tempForRight->parent = head;
                head->sibs.push_back(tempForRight);
            }
        }
    }
    //对于token节点，直接对其进行类型判断，将常数直接进行处理
    //值得注意的是，尽管此处没有单独列出，identifier也是特殊单独处理，只是其内容与其余内容的yytext并无区分
    else {
```

```
        int line = va_arg(valist,int);
        head->line = line;
        if(head->name == "CONSTANT_INT") {
           int value;
           if(strlen(yytext) > 1 && yytext[0] == '0' && yytext[1] != 'x') {
               sscanf(yytext,"%o",&value); //8进制整数
           }
           ……
        }
        else if(head->name == "CONSTANT_DOUBLE") {
           head->content = yytext;
        }
        ……
        }
    }
    return head;
}
```

`createTree`函数为建立语法树的主要函数，接收参数后迭代式地根据参数从`root`处向下构建子节点，并最终完成整体的树构建。具体流程为：

根节点构建->参数list中含下一个节点->构造该节点并将其纳入子节点容器->构造时子节点参数包含下一层的参数->重复以上过程直至所有分析达到叶子节点

```
void Eval(treeNode *head,int leavel) {
    if(head!=NULL) {
        string Name = head->name;
        if(head->line!=-1) {
            for(int i=0;i<leavel;++i) {
                cout << ". ";
            }
            cout << head->name;

            if(head->name == "IDENTIFIER"||head->name == "BOOL"|| head->name ==
"INT" ||
            head->name == "CHAR" || head->name == "DOUBLE") {
                cout << ":" << head->content;
            }
            ……
        }
        //逐个子节点进行打印
        for (int i = 0; i < head->sibs.size(); i++){
            Eval(head->sibs[i],leavel+1);
        }
    }
}
```

Eval函数从传入节点开始迭代式地进行打印，并使用和树高参数（leavel)成正比的"."来进行输出流中树支架的区分。

```
void freeGramTree(treeNode* node) {
    if (node == NULL)
        return;
    auto treeIt = node->sibs.begin();
        for (; treeIt <= node->sibs.end() - 1;treeIt++){
        freeGramTree(*treeIt);
        }
    delete node;
}
```

`freeGramTree`是我们进行空间管理的函数。由于C++没有自动的垃圾回收机制，我们应当实现空间管理以确保代码的健康性与空间可用性。

3、Praser模块

4、IR code生成模块

本次实验中，我们将IR code的生成工作集成在Praser模块中调用函数接口完成，以下为该代码生成模块的具体实现。

4.1 数据结构

```
class Node{
public:
    Node(void) { ···
    Node(varNode * node1) { ···
    Node(arrayNode * node1) { ···
    Node(funcNode * node1) { ···

    varNode *retVar(void) { return (Type == VAR) ? var : NULL; }
    arrayNode *retArray(void) { return (Type == ARRAY) ? array : NULL; }
    funcNode *retFunc(void) { return (Type == FUNC) ? func : NULL; }
    string retType(void) {···
    string retName(void) {···
    bool retUseAdd(void){···
    int retNum(void){···
private:
    varNode * var;
    arrayNode * array;
    funcNode * func;
    enum {
            FUNC,
            ARRAY,
            VAR,
            EMPTY
        } Type;
};
```

```
//变量节点
struct varNode {
```

```
    string name;
    string type;
    int num = -1;
    bool useAddress = false;    //判断指针类型，使其可以判断无struct等语法情况下的指针
    string boolString;          //将bool变量囊括进普通变量中
};

//函数节点
struct funcNode {
    bool isdefinied = false;
    string name;                //函数名
    string rtype;               //函数返回类型
    vector<varNode> paralist;   //记录形参列表
};

//数组节点
struct arrayNode {
    string name;
    string type;
    int num = -1;
};
```

我们将Praser需要的节点分为变量、函数、数组三类，并将这三者集成在`Node`类中完成封装，通过接口返回需要的名称、类型、是否使用地址（指针）、函数返回类型等。

```
class InnerCode {
private:
    vector<string> codeList;

public:
    int tempNum = 0;
    int varNum = 0;
    int labelNum = 0;
    int arrayNum = 0;

    InnerCode();
    void addCode(string);
    void printCode();
    //面向var的代码生成重载
    string createCode(string tempname, string op, Node node1, Node node2);
    //面向assign的代码生成重载
    string createCode(Node node1,Node node2);
    //面向Parameter的代码生成重载
    string createCode(Node node);
    //面向return与arguments的生成重载，使用时form参数填入“RETURN”与“ARG”
    string createCode(string form, Node node);

    //获取节点名称,form 填入“VAR”与“ARRAY”进行选择
    string getNodeName(string form , Node node);
};
```

如上即为集成createCode函数重载并存储、打印IR code的Innercode模块，其中具体逻辑即为调用Node接口获取具体数据信息，而后将其进行字符串封装，打包成类似教材中的IR code字符串并通过`addCode`接口压入代码列表，以供后续打印。以Parameter的重载为例：

```
string InnerCode::createCode(Node node) {
    string result = "PARAM ";
    result += "var" + to_string(node.retNum());
    return result;
}
```

5、Git工作流

由于暂时没能搞懂git blame的使用与描述，因此本部分展示git log后的日志显示。

```
commit 4743f7cc71c5eef3a5c192d6a1b13069cab491c5 (HEAD -> master, origin/master,
origin/HEAD)
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Sat May 27 14:08:39 2023 +0800

    规范命名

commit 90cf274daa40499fd9ae235a8a3a79ce60281f63
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 23 17:32:36 2023 +0800

    name change

commit ad1c3c931b0d2587cd6f10c1185397c8278652b6
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 23 17:31:28 2023 +0800

    name change

commit 62613eae0665e3233aadb64f56948edbe2831d57
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 23 16:09:54 2023 +0800

    ir-gen update

commit 38a089311fb0e2578b73f8afc203317a23af0e46
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 23 15:37:28 2023 +0800

    tree and ir-gen update

commit 59254408cc7a80fb707f7c8257d68b5025a8147f
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 23 14:55:42 2023 +0800

    tree update 5.23
```

```
commit fc4704005284b8c4aea09e43db16843327f543d3
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Mon May 22 23:22:28 2023 +0800

    new tree

commit 98d0f789655afcd2c636a5896aa11def9b043256
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Mon May 22 21:12:18 2023 +0800

    new tree

commit 34b3ba8b9891c468d80a46fc9d4514906c85444c
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Sat May 20 16:50:55 2023 +0800

    edit

commit f60a5d613e3c43708cb5c7b2579e93b713db9095
Author: SingletonShu <SingletonShu@gmail.com>
Date:   Fri May 19 22:47:12 2023 +0800

    基本写完了词法分析的部分，接下来便是构建AST的过程，脑子里已经有思路了。
    之前画的饼没有完成QAQ，因为有很多我没有预料到的小问题。
    这周末可能有点忙，不过进度应该是ok的。
    fighting!

commit ee740e922186467ea7670a520c741747ca8e8b14
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Wed May 17 17:43:08 2023 +0800

    info update

commit fdeaa3b643449dc7d490242fafcd79b15a3a9d55
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 23:40:56 2023 +0800

    something new

commit 7d774ff4fd2e5257b12aa52e95edb4d52dd92df5
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 22:33:31 2023 +0800

    update tree.h

commit 9f78d71b19fe3d2a4cd8a21a59c09473e534070a
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 21:35:50 2023 +0800

    info update

commit dffec1a0aa9bb3c8f3dd79bf9abd50cebf2a5f73
```

```
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 16:32:55 2023 +0800

    info update

commit 231447e7a663d7cf9bcd350cdf0b0360228ea68d
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 14:41:47 2023 +0800

    update tree.cpp

commit bf825be6549d9ae42d9225af09d9919d44303819
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 13:56:32 2023 +0800

    5/16 tree update

commit 1053a79a6665e372d90c608489c9731960888e5c
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 11:54:09 2023 +0800

    info update

commit 1bc2051d62c2c396a075b49167716e4e8cc4b388
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Tue May 16 11:36:01 2023 +0800

    some change

commit 463ef1c34bf1a6f9c294c4bbe7ed4e4528c447e5
Author: SingletonShu <SingletonShu@gmail.com>
Date:   Sun May 14 17:11:12 2023 +0800

    做了一下词法分析，还不是完整品（union, enum，goto等还未实现），但大部分比较常见的语法是ok的。争取周二做完语法分析。

commit 9202f717199f0e5bb6401a9c29008caaeb8aed33
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Mon May 8 21:38:05 2023 +0800

    code for part1
commit f5ea93a829c16be1d9f1b2f4b0e55b95d0c2d41c
Author: Windhxer <3200102329@zju..edu.cn>
Date:   Mon May 8 20:57:46 2023 +0800

    commnet

commit 8d226e0c2cff78eade400550dbe83eca691210e6 (origin/main)
Author: WanShan <3200102329@zju.edu.cn>
Date:   Mon May 8 20:56:03 2023 +0800

    Initial commit
```