MiniC2RISC-V 编译实习报告

北京大学信息科学技术学院 吴箫 1600012807

简介

运行代码，请参照README.md.

本报告分为如下几部分，简介简要说明实现的编译器功能，接下来MiniC2Eeyore，Eeyore2Tigger，Tigger2RISC-V，g--分别介绍各部分的具体思考、探索和实现的过程，最后一部分简述我个人的经历、收获以及对课程的建议。

实现了将MiniC代码转化为RISC-V汇编代码的编译器。实现过程分为三部分，分别将MiniC代码转化为Eeyore代码（三地址码），将Eeyore代码转化为Tigger代码（寄存器分配），和将Tigger代码转化为对应的RISC-V汇编代码。最后，将三部分的可执行文件共同纳入一个使用方式类似gcc的可执行文件g--中。

MiniC2Eeyore部分，支持了类似gcc的报错和报warning的方式，扩展了MiniC的基础语法集。具体来说包括：

- 报错系统
  - 出错定位（精确到列）和原语句呈现
  - 变量和函数管理
    - 对未声明的函数报warning
    - 对未声明的变量使用报错
    - 对重复定义的函数和变量报错
    - 所有变量的作用域精确符合C的标准
    - 传参错误处理，对传入参数数量或类型和定义不符的针对性报错
  - 语法错误处理，对变量定义、缺少分号等情况针对性报错，并输出错误语句
  - 零除报warning
- 语法扩展
  - 变量的自增和自减
  - 表达式独立成句
  - 连续等式赋值
  - 对于函数末尾没有return语句的自动补全
  - 表达式参数传递
  - 表达式化简

该部分还可以完善的地方包括：

- 扩展类型（包括double，unsigned以及自定义数据类型struct等)
- 类型检查系统（目前只检查了函数传参类型）
- for循环，do-while循环，switch语句
- 条件表达式短路

Eeyore2Tigger部分，采用了简单的寄存器分配方式，尽量利用了所给的寄存器

- 能够生成通过评测的Tigger代码

该部分不足的地方：

- 寄存器分配的方式过于简单，可以用图染色或线性扫描的方式
- 该代码在非常复杂的情况下会出现正确性问题
- 可以进行活跃变量分析，公共表达式复用，死代码消除，常量传播，窥孔优化等优化

Tigger2RISC-V部分，较为简单，简单实现即可

g--部分，用一个可执行文件统一了以上的实现

- 能够用-o指定输出文件名
- 在有报错的情况下停止编译

该部分可以完善地方：

- 支持类似g++的更多的参数（--help, -i等）

MiniC2Eeyore

报错系统

出错定位

Bison的文档提供了生成用于定位的数据结构的方法，在`.y`文件中利用语句`%locations`和`%defines`可以生成头文件，包含了结构体定义和全局变量yylloc。

```
typedef struct YYLTYPE YYLTYPE
struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
};
extern YYLTYPE yylloc;
```

但是，yylloc并不会自动更新。Google一番后，在StackOverFlow上我找到了想要的实现，要实时追踪位置，需要与词法分析器结合起来，识别出一个token后，需要手动进行位置更新。

以下代码中，YY_USER_ACTION是一个识别出token后会执行的宏，此时yytext中存储着识别出的token。

```
#define YY_USER_ACTION update_loc();
void update_loc(){
    static int curr_line = 1;
    static int curr_col  = 1;

    yylloc.first_line   = curr_line;
    yylloc.first_column = curr_col;

    {char * s; for(s = yytext; *s != '\0'; s++){
    if(*s == '\n'){
        curr_line++;
        curr_col = 1;
    }else{
        curr_col++;
```

```
    }
    }}

    yylloc.last_line   = curr_line;
    yylloc.last_column = curr_col-1;
}
```

在`.y`文件中可以利用@来获得位置信息，例如：

```
Expression: Expression DIVIDE Expression
{
    ...
    if(((ExprNode*)$3)->valueID=="0"){
            printWarningInfo("division by zero", @2);
    }
}
```

原语句呈现

由于出错的地方可能在一个语句读了一半的时候，如果要一边扫描一边生成报错信息并呈现该语句，需要先将输入预先保存，这里处理的方式是将输入预先按行存储。

这样做可以解决另外一个小问题，源文件输入中的`tab`可能在shell中占8格，导致定位箭头不准，存储时可以全部将`\t`变成4个空格。

例子如下：

```
int main(){
    int a;
    int a;
    return 0;
}
```

```
: In function 'main':
:3:9: error: redefinition of 'a'
    int a;
        ^
```

变量和函数管理

对于变量域的管理，简单而言，一个变量的生命周期，从它被声明开始，到包括它的一对花括号的右括号`}`处结束。函数参数可以被特殊处理。从而可以用一个栈维护目前还没有结束的`{``}`域，遇到右括号就弹栈，遇到左括号就压栈。

每个变量声明时，在表末尾插入表项，表项包括当前的域值和名字；变量使用时，在表中从后向前查找，如果有名字匹配且目前活跃的表项，则找到了对应的变量。否则，说明变量使用先于定义，报错。变量重定义的方式也类似解决。

具体代码也较为简洁：

```
Scope::Scope(){
```

```
    counter = 0;
    currentScope = 0;
    aliveScope = vector<int>();
    aliveScope.push_back(currentScope);
}
bool Scope::isScopeAlive(int x){
    for(auto& i:aliveScope)
        if(x==i)
            return true;
    return false;
}
void Scope::newScope(){
    counter++;
    currentScope = counter;
    aliveScope.push_back(currentScope);
}
void Scope::endScope(){
    aliveScope.pop_back();
    currentScope = aliveScope.back();
}
```

变量重定义的例子上面已经提到了，下面是一个变量使用先于定义的报错例子

```
int main(){
    a = a + 1;
    return 0;
}
```

```
```

函数的管理较为简单，但要注意的是，声明、定义不符（名字相同，参数列表不同），多次定义，未声明使用，调用参数传入数量偏多或偏少，情况较多。这里各自作了处理。此处仅举两例：

```
int f(int a, int b);
int f(int a){
    return a;
}
int main(){
    f();
    return 0;
}
```

```
: In function 'f':
:2:5: error: conflicting types for 'f'
int f(int a){
    ^
: In function 'main':
:6:5: error: too few arguments to function 'f'
    f();
    ^
```

```
int f(int a, int b){
    a+b;
}
int main(){
    int a ;
    f(a);
    return 0;
}
```

```
: In function 'main':
:6:5: error: too few arguments to function 'f'
    f(a);
    ^
```

语法错误处理

以上的情况都是语法没有错误的情况，对于语法本身出错（例如缺少分号等），这里也进行了处理。Google一番后知道，Bison提供了符号`error`，通过添加产生式`Statement: error ';'`，Bison能够生成错误处理的代码。工作机理是这样的，例如在移入了一些表达式之后，发现少了运算符而进入了panic模式，于是语法分析器不停弹栈，直到栈顶为一个可以推出`error`的产生式`Statement`，之后不停丢弃输入直到遇到`;`，规约到Statement，恢复正常并继续规约。

丢弃输入可能会引发新的错误，比如变量定义的语法错误可能会导致接下来的引用未定义的变量的错误。但这里没有作更多处理。

实现中，为了捕获所有的语法错误而不致使语法分析器崩溃，要分层设计`error`的产生式，这个过程很像catch exception。在较低层级上的（Expression，Statement）被捕获的`error`就会不停丢弃输入直到可以被规约。一般来说，这里就用`;`作为错误的分界线，而不去找到更具体的错误细节。下面举一个例子：

这是变量定义的SDT

```
VarDefn:
error ';'
{
    printErrorInfo("wrong variable definition method", @2);
    ...
}
```

对于以下输入文件，

```
int main(){
    int b[];
    return 0;
}
```

有输出：

```
syntax error: In function 'main':
:2:12: error: wrong variable definition method
    int b[];
           ^
```

尽管如此，还是会有一些问题，比如以下代码因为规约-规约冲突而始终无法被调用，也就是说，无法精确输出错误原因，要解决关于`error`的规约-规约冲突涉及了更复杂的细节，这里没有继续深究下去。

```
Statement:
error ';'
{
    string errMsg = "broken statement";
    printErrorInfo(errMsg, @2);
    ...
}
```

零除warning

加一个特判实现，较为简单

```
int main(){
    int a;
    a = 1;
    return a/0;
}
```

```
: In function 'main':
:4:13: warning: division by zero
    return a/0;
            ^
```

语法扩展

- 变量的自增和自减
- 表达式独立成句
- 连续等式赋值
- 表达式参数传递

以上四条，只需要修改文法和添加相应语句即可，这里仔细设计文法避免了冲突情况，其它的过程都是trivial的。

- return自动添加

这一条是在解析函数完毕时，查找是否有return语句并额外添加来实现的。

- 表达式化简

如果一个表达式中全是整数，那么直接计算得到结果，通过在表达式节点中增添是否为整数字段实现，具体效果如下，对于C代码片段

```
...
a[0] = getint();
a[1] = getint();
a[2] = getint();
a[3] = getint();
a[4] = getint();
...
```

生成了

```
...
var t0
t0 = call f_getint
T0[0] = t0
var t1
t1 = call f_getint
T0[4] = t1
var t2
t2 = call f_getint
T0[8] = t2
var t3
t3 = call f_getint
T0[12] = t3
var t4
t4 = call f_getint
T0[16] = t4
...
```

这里我将讨论一下文法的设计和冲突情况的处理，下面是Bison生成的文法。这个文法是有冲突的，主要是`error`的问题没有得到特别好的处理。但对于没有语法错误的MiniC，这个文法不会遇到移入-规约或规约-规约冲突。

```
    0 $accept: Program $end

    1 Program: GlobalList

    2 GlobalList: VarDefn GlobalList

    3 $@1: %empty

    4 GlobalList: FuncDefn $@1 GlobalList
    5           | FuncDecl GlobalList
    6           | %empty
    7           | error

    8 VarDefn: TYPE ID ';'
    9        | TYPE ID '[' INTEGER ']' ';'
   10        | error ';'

   11 ParaDecl: TYPE ID
   12         | TYPE ID '[' INTEGER ']'
```

```
13         | TYPE ID '[' ']'
14         | TYPE
15         | TYPE '[' ']'

16 ParaDecls: ParaDecl
17          | ParaDecl ',' ParaDecls

18 ParaList: ParaDecls
19         | %empty

20 FuncDecl: TYPE ID '(' ParaList ')' ';'
21         | error ';'

22 $@2: %empty

23 FuncDefn: TYPE ID '(' ParaList ')' '{' $@2 Blocks '}'

24 Block: Statement

25 $@3: %empty

26 Block: '{' $@3 Blocks '}'
27      | IF '(' Expression ')' Block
28      | IF '(' Expression ')' Block ELSE Block
29      | WHILE '(' Expression ')' Block

30 Blocks: Block Blocks
31       | %empty

32 Statement: Expression ';'
33          | VarDefn
34          | RETURN Expression ';'
35          | error ';'

36 ExprList: Expressions
37         | %empty

38 Expressions: Expression ',' Expressions
39            | Expression

40 Expression: INTEGER
41           | ID
42           | '(' Expression ')'
43           | Expression PLUS Expression
44           | Expression MINUS Expression
45           | MINUS Expression
46           | Expression TIME Expression
47           | Expression DIVIDE Expression
48           | Expression MOD Expression
49           | Expression AND Expression
50           | Expression OR Expression
51           | NOT Expression
52           | Expression LESS Expression
```

```
53          | Expression GREATER Expression
54          | Expression EQUAL Expression
55          | Expression NOTEQUAL Expression
56          | ID '(' ExprList ')'
57          | ID '[' Expression ']'
58          | ID ASSIGN Expression
59          | ID '[' Expression ']' ASSIGN Expression
60          | DOUBLEPLUS ID
61          | ID DOUBLEPLUS
62          | DOUBLEMINUS ID
63          | ID DOUBLEMINUS
```

值得注意的有这么几点：

IF-ELSE悬空二义性问题

基于*优先级本身解决了二义性*的想法，最简单的方式是规定不匹配的产生式的优先级低于IF-ELSE匹配的产生式。

```
%nonassoc IF
%nonassoc ELSE
Block:
    IF '(' Expression ')' Block
    {
        ...
    } %prec IF
    |
```

Block、Statement、Expression的区别

出于以下目的：

- 避免引入冲突，方便地解决花括号`{}`括起一个语句块的问题
- 连等式的实现

规定*Expression*为一个有值的表达式，无分号`;`结尾，这样，a = b可以被视作一个值为b的表达式；规定*Statement*为一个以分号`;`结尾的字面意义单句，但不含花括号`{}`，*Statement*没有值；规定*Block*为可被视为一个整体的单句，可以含花括号`{}`，这样，以上代码块的Block的使用就显得简洁而合理，避免了IF语句后跟一个单句和用花括号组织的片段的讨论。

Eeyore2Tigger

这是我写的时间最久，debug时间也最久的一段代码，但要说到这段代码完成了什么，似乎没什么能说的了。期间还重构了一次，但种种原因，现在的tigger还是最基础的版本，用线性扫描法寄存器分配的部分一直没有debug成功通过测评。此外，也没有进行更进一步的代码优化。

MiniC2Eeyore的时候，代码生成比较直接，直接on-the-fly代码生成。但Eeyore2Tigger较为复杂，就先建树再生成代码。

寄存器分配有许多细节问题，包括何时第一次将变量载入寄存器，何时将变量换出寄存器，在函数调用时要保存现场，在函数调用结束时要恢复现场，以及要管理栈空间。

以下是大致思路，更多细节参照代码

- 建树后在每个函数内进行活跃变量分析，找到变量的活跃区间
- 从上到下逐语句给活跃的变量分配寄存器，如果用完则分配栈
- 如果变量不再活跃则退还寄存器
- 留出额外的调用者保存寄存器作为临时变量，考虑到类似将全局变量保存没有直接的指令，还是需要留出临时寄存器的
- 对于每一个语句，考虑左值和右值有交集的情况，考虑右值中有整数可以进行化简

活跃变量分析

我之后试图重构源代码的代码中，按照课件上的方法，在每一个函数里将每一条语句视作基本块，从后向前地扫描，不断更新每条语句的OUT活跃变量集。这一部分代码大致如下。但重构的代码没有完成，现在使用的策略还是更简单的策略。

```
// 计算每个孩子Expr的活跃变量
// 调用calSuccForExprs和calAliveVarsForExprs
void FuncNode::analyzeLiveness(){
    calSuccForExprs();  //得到后继语句
    calAliveVarsForExprs(); //得到OUT活跃变量集
}
// 为所有孩子ExprNode计算后继语句集
void FuncNode::calSuccForExprs(){
    for(int i = 0;i<children.size();i++){
        ExprNode* child = (ExprNode*)children[i];
        ExprType childType = child->getExprType();
        //  条件语句的后继语句为后一条和跳转到的语句
        if(childType==IfBranchType){
            if(i!=children.size()-1){
                ExprNode* nextChild = (ExprNode*)children[i+1];
                child->addSuccExpr(nextChild);
            }
            string label = child->getLabel();
            ExprNode* jumpToNode = searchLabel(label);
            child->addSuccExpr(jumpToNode);
        }
        //  goto语句的后继语句为跳转到的语句
        else if(childType==GotoType){
            string label = child->getLabel();
            ExprNode* jumpToNode = searchLabel(label);
            child->addSuccExpr(jumpToNode);
        }
        //  其它语句的后继语句是后一条语句
        else{
            if(i!=children.size()-1){
                ExprNode* nextChild = (ExprNode*)children[i+1];
                child->addSuccExpr(nextChild);
            }
        }
    }
}
// 迭代地从后向前扫描更新每个Expr的活跃变量集
void FuncNode::calAliveVarsForExprs(){
    bool hasChanged = false;
```

```
    while(true){
        for(auto it = children.rbegin();it!=children.rend();++it){
            ExprNode* child = (ExprNode*)(*it);

            // newSet = (join - left) U right
            set<IdNode*> oldAliveVarSet = child->getAliveVarSet();
            set<IdNode*> joinAliveVarSet = child->getJoinAliveVarSet();
            set<IdNode*> leftValueVarSet = child->getLeftValueVarSet();
            set<IdNode*> rightValueVarSet = child->getRightValueVarSet();

            set<IdNode*> tmp;
            set<IdNode*> newAliveVarSet;
            set_difference(oldAliveVarSet.begin(), oldAliveVarSet.end(),
            leftValueVarSet.begin(), leftValueVarSet.end(),tmp);
            set_union(tmp.begin(),tmp.end(),rightValueVarSet.begin(),
            rightValueVarSet.end(), newAliveVarSet);

            child->setAliveVarSet(newAliveVarSet);

            if(!hasChanged){
                hasChanged = !equal(newAliveVarSet.begin(), newAliveVarSet.end(),
                oldAliveVarSet.begin(), oldAliveVarSet.end());
            }
        }
        //  未发生改变则终止循环
        if(!hasChanged) break;
    }
}
```

目前的策略是，按照变量出现的范围，直接将该范围的闭包指定为这个变量的活跃区间，这样做简单安全不容易出错，且在给定的寄存器数量情况下，效率也并不低。

代码实现如下

```
// 遇到某个变量时，根据行数更新它的周期
void setNameRange(node* p){
    string idName = p->name;
    int i = 0;
    int l = ancestor->idTable.size();
    bool isArray_ = p->isArray;

    for(i = 0; i < l; i++){
        if(ancestor->idTable[i].name == idName){
            ancestor->idTable[i].start = min(ancestor->idTable[i].start, exprCnt);
            ancestor->idTable[i].end = max(ancestor->idTable[i].end, exprCnt);
            // ancestor->idTable[i].spillTime = ancestor->idTable[i].end;
            return;
        }
    }
    if(i==l){//没找到，可能是第一次，也可能是全局变量
        if(globalTable.find(idName)!=globalTable.end()){
            //全局变量
            idType in = globalTable[idName];
```

```
            idType pushed_in = in;
            pushed_in.start = pushed_in.end = exprCnt;
            pushed_in.spillTime = INF;
            pushed_in.name = idName;
            ancestor->idTable.push_back(pushed_in);
        }
        else{
            //第一次遇到的局部变量
            ancestor->idTable.push_back(idType(idName, exprCnt,false, isArray_));
        }
    }
}
```

线性扫描分配寄存器

在重构的代码中，我也实现了线性扫描分配寄存器的方法，但是时间紧张，没法debug了，所以还是采取了更为trivial的方法。下面是线性扫描分配寄存器的部分代码。

```
    for(auto& it:idTable){

        // 释放周期已经结束的变量对应的寄存器
        for(auto& jt:liveQueue){
            if(jt->end < it.start){
                r.release(jt->regsiter);
            }
        }
        // 并从队列中移除
        remove_liveQueue(it.start);

        // 给从start开始的变量it分配寄存器

        // 寄存器未满直接分配寄存器
        if(!r.full()){
            it.regsiter = r.acquire();
            insert_liveQueue(&it);
        }
        // 寄存器已满
        else{
            idType* last = liveQueue.back();
            // 队尾变量截止时间晚，将队尾变量入栈，
            // 给it分配寄存器并入队
            if(last->end > it.end){
                last->spillTime = it.start - 1;
                r.release(last->regsiter);
                liveQueue.pop_back();
                it.regsiter = r.acquire();
                insert_liveQueue(&it);
            }
            // 队尾变量截止时间早，将it入栈
            else {
                it.regsiter = -1;
            }
```

```
        }
    }
```

实现中采用的方法是将变量按活跃起始顺序排序，依次给变量指派寄存器。不能放入寄存器的指派栈空间。

Tigger2RISC-V

这一部分比较简单，但我也花了非常多时间来debug。具体情况是，我的Eeyore2Tigger代码通过了测评，但RISC-V无法通过测评，但这一步其实是相当简单的直接翻译。我最终克服心理障碍找了同学要了一份他的代码，做了交叉实验后得到以下结果。

eeyore	tigger	riscv	result
His	His	Mine	Pass
His	His	His	Pass
Mine	Mine	His	Fail
Mine	Mine	Mine	Fail

于是我推理是我RISC-V的翻译没有写错，既然它可以正确地翻译别人的Tigger代码（这一步让我花了很久时间），但是，我的Eeyore2Tigger可能在什么地方出了错，以至于它通过了测评，但留有错误，到下一个检查点才被发现。于是我回头重构了Tigger的代码，碍于期末季学业繁重，代码还有待进一步完善。

这一段本身没有特别之处，见代码即可。

g--

之前的三个部分生成的可执行文件只能流式地处理，这里将它们打包成一个发布文件`g--`，用`./g-- [filename] -o [filename]`或者`./g-- [filename]`即可以运行。但是由于缺乏软件打包发布的知识，这里只是用类似`exec`的方法来依次执行三个可执行文件。

更好的实现，应该是在三个文件夹下定义不同的命名空间，然后在外部统一地调用接口来发布。

总结

追赶进度

我这学期才开始上的编译原理课，在期中的时候我都还不知道SDT和属性文法，也不知道LALR的定义。刚开始写eeyore我都不知道常规的方法是建好树再去遍历生成代码，直接脑补出了on-the-fly的方式。当时也不知道如何解决冲突，再加上对C++的陌生，写出来的代码不伦不类。现在的MiniC2Eeyore是重写后的。

同样的，写Eeyore2Tigger的时候，我也完全落在进度后面，不知道什么是活跃变量，以至于勉力通过测评之后就很难对源代码进行进一步优化。当时代码的正确性也未能完全保证。

当然，我最后还是做出了让我自己觉得很有成就感的一个小编译器。期间几乎完全依赖Google，学会了很多东西。因此，作为对课程的建议，我觉得这门课应该还是不适合和编译原理一起上的，进度上不太匹配，自己学会花很多时间，非常辛苦。

关于自学

在写代码过程中我自己学了很多东西，比如Bison和Flex的语法，如何像样地组织许多`.cc`和`.h`文件，如何写`makefile`来快捷地编译运行它们，如何在父文件夹调用子文件夹的`makefile`。我甚至去看了*Effective C++*来学习如何更好地写C++代码。

这是我目前写过的工程量最大的课程作业，我在其中学到了非常多。现在我看到*Segmentation Fault*都会有心理阴影。我觉得内存问题是debug中最困难的部分，因为错误的现场和原因往往相隔甚远。因此对于C++这样的语言，必须要格外小心。

我也总结了我学习习惯上的问题，我过去总是喜欢一个人学，不喜欢提问和交流，总认为是打扰，也觉得上网搜代码是近似抄袭的习惯。但我觉得假如把目标设定为完成工程并学到知识，自己单打独斗绝非最优解。自己写代码而缺乏交流，只会让coding和debug的时间变长，所以就少了很多用来学习和创造性思考的时间。比如在交换了代码做了试验之后我才确定了错误的范围，比如在Google很多之后才知道了一些惯例和方法。现在我后悔没有充分利用课堂和助教、老师进行交流。有的问题其实一说就可以知道，但自己想需要很久，这并非高效的方式。

课堂资源

我在Google、StackOverFlow上找到了很多帮助，但我觉得如果刘老师能够在教学网上提供一些**不止于文档的示例**就更好了。可以是比如，另一个语法翻译（比如MiniJava）的实现，或者往届同学的作品。可以将这个作为第一次作业，让大家对自己将要完成什么更有底气一些。相比之下，写一个计算器只是让我熟悉了Flex和Bison，并没有对我完成这个编译器起到更大的帮助。此外，这样的实例可以让我们学习如何用C++组织程序简便高效地完成工程，如何利用Bison更多的特性。

另外，我觉得龙书其实是侧重原理的、更为繁琐的一本书，非常不适于作为参考书，如果刘老师能够提供**更简明的工具书或手册**，我觉得会让我这样没有事先学过编译原理的同学更容易克服障碍一些。

困境与收获

我所遇到的困境有三类，在面对它们的时候我提高了工程能力

1. 从对自己要完成什么完全没有概念，到开始设计文件、类、函数，是困难的过程，这一部分让我感觉到了计算机学科作为艺术而不只是作为科学的一方面。怎样组织文件，能够让实现清晰有效，让接下来无论是debug还是更新代码都更为简单，是一个复杂但需要仔细考虑的问题，优秀的组织形式可以节约大量时间。我在这个过程中，大致有了如下感受
    - 在coding的时候就要考虑debug的事，设计错误处理的类，如果不用*catch-throw exception*的方法，就利用*assert*和定点输出的方法进行检查。打印源代码，打印log都是行之有效的方式。但逐步跟踪不适合较大的工程。
    - 对于几千行的可控代码来说，没有必要引入getter和setter，会让代码变丑陋，全部public就可以。
    - 一定要在指针类型转换的时候显式地写明，以避免潜在的*Segmentation Fault*
2. 从写完代码到debug完毕，意识到自己犯了这样或那样的错误，也是十分挣扎的困境，因为没有什么能保证自己可以在多少小时以内完成debug。用打印信息的方式对调试很有帮助。
3. 从debug完毕到迭代更新和改进代码，是很难的。既难在更好的方法的复杂性上，又难在在已有的代码上进行大规模改动。主要还是要在coding的时候就把框架搭好，不要引入过于复杂的特判，功能过于复杂具体的函数，尽量抽象和简洁。

总体来说，在付出了很多构思、coding和debug的时间后，最后能够完成一个编译器并提交这样一份报告，我学到了很多，这是一份让我有成就感的大作业。