浙江大學

C-Minus Compiler

课程名称：编译原理

2021-2022学年春夏学期

学院：竺可桢学院

专业：计算机科学与技术

指导教师：李莹

小组成员：汪辉 徐正韬 谢文想

Date:2022-05-21

概述

1 项目概述

本次实验我们小组基于Flex，bison以及LLVM，使用C++语言设计并实现了一个简单的C-语言的编译器，该编译器以符合C语言规范的代码文件文本为输入，输出为指定机器的目标代码。该编译器的实现涵盖词法分析、语法分析、语义分析、代码生成等阶段和环节，使用flex以及yacc实现对输入文件进行词法以及语法分析，并生成对应的抽象语法树，所使用的具体技术包括但不限于：

- Flex实现词法分析
- Yacc实现语法分析
- LLVM实现中间代码生成、目标代码生成
- 实现AST可视化

本项目目前支持的数据类型以及操作包括如下：

- 数据类型
  - void
  - int
  - float
  - double
  - char
  - 一维数组
  - 二维数组
- 操作
  - 变量的声明、初始化，包括本地变量以及全局变量
  - 函数声明、定义、调用
  - 循环语句、条件语句
  - 二元运算符、赋值、取地址、自增自减运算符等
  - 数组传参
  - ...

2 文件说明

```
|-- README.md                           <--项目说明文档
|-- AST_visualization
|   |-- example.jpg
|   |-- example.json
|   |-- visualization.sh
|-- doc
|   |-- Project 验收细则.md
|   |-- Project 验收细则.pdf
|   |-- report.md
|   |-- report.pdf                      <--项目报告
|   |-- image                           <--项目报告图片
|-- src
|   |-- ast.h                           <--AST头文件，定义节点类
|   |-- ast_builder.cpp                 <--
|   |-- ast_setget.cpp                  <--
|   |-- generator.cpp                   <--
|   |-- generator.h                     <--
|   |-- globals.cpp                     <--
```

```
|   |-- main.cpp                        <--项目主函数
|   |-- parser.yacc                     <--语法分析
|   |-- scanner.l                       <--词法分析
|   |-- type.h                          <--定义数据类型
|   |-- variable.cpp                    <--
|   |-- variable.h                      <--
|   |-- visualization.cpp               <--可视化
|-- test
|   |-- auto-advisor.c                  <--测试样例1 C-源代码
|   |-- matrix-multiplication.c         <--测试样例2 C-源代码
|   |-- quicksort.c                     <--测试样例3 C-源代码
```

3 开发环境

- 操作系统：Linux
- 编译环境：
  - Flex
  - Bison
  - LLVM

4 组员分工

组员	具体分工
汪辉	
徐正韬	
谢文想	

第一章 词法分析

1.1 flex介绍

Flex(快速词法分析器生成器)是lex的免费开源软件替代品，它是生成词法分析器的计算机程序，词法分析器是识别文本中的词汇模式的程序。flex程序读取给定的输入文件，生成一个C源文件作为输出。flex的输入文件称为lex源文件，它内含正规表达式和对相应模式处理的C语言代码。lex源文件的扩展名习惯上用.l表示。flex通过对源文件的扫描自动生成相应的词法分析函数int yylex()，并将之输出到名规定为lex.yy.c的文件中。实用时，可将其改名为lexyy.c。该文件即为lex的输出文件或输出的词法分析器。也可将int yylex()加入自已的工程文件中使用。

lex对源文件的格式要求非常严格，比如若将要求顶行书写的语句变成非顶行书写就会产生致命错误。而lex本身的查错能力很弱，所以书写时一定要注意。

lex的源文件由三个部份组成，每个部分之间用顶行的“%%”分割，其格式如下：

```
{definitions}
%%
{rules}
%%
{subroutines}
```

1.2 具体实现

词法分析是编译器实现的第一步，我们所需要做的就是将输入转化为一些列的token然后传给yacc。本项目中定义的token一共包括：加减乘除等运算符、关系运算符、取地址符号、括号、逗号、句号、C语言关键字、用户自定义的标识符ID、整型常量、浮点型常量、字符常量、字符串常量、换行空格等其它符号等。

我们需要在parser.yacc中先声明这些token，然后再scanner.l中定义相关token的操作。

token共分为以下几种情况：

- **关键字**

```
"char"            {yylval.node = new Node(yytext, "CHAR", 0); return CHAR;}
"int"             {yylval.node = new Node(yytext, "INT", 0); return INT;}
"float"           {yylval.node = new Node(yytext, "FLOAT", 0); return FLOAT;}
"double"          {yylval.node = new Node(yytext, "DOUBLE", 0); return DOUBLE;}
"void"            {yylval.node = new Node(yytext, "VOID", 0); return VOID;}
"if"              {yylval.node = new Node(yytext, "IF", 0); return IF;}
"else"            {yylval.node = new Node(yytext, "ELSE", 0); return ELSE;}
"for"             {yylval.node = new Node(yytext, "FOR", 0); return FOR;}
"while"           {yylval.node = new Node(yytext, "WHILE", 0); return WHILE;}
"continue"        {yylval.node = new Node(yytext, "CONTINUE", 0); return
CONTINUE;}
"break"           {yylval.node = new Node(yytext, "BREAK", 0); return BREAK;}
"return"          {yylval.node = new Node(yytext, "RETURN", 0); return RETURN;}
```

- **各种运算符**

```
"+"               {yylval.node = new Node(yytext, "PLUS", 0); return PLUS;}
"-"               {yylval.node = new Node(yytext, "MINUS", 0); return MINUS;}
"*"               {yylval.node = new Node(yytext, "MUL", 0); return MUL;}
"/"               {yylval.node = new Node(yytext, "DIV", 0); return DIV;}
"%"               {yylval.node = new Node(yytext, "MOD", 0); return MOD;}
"="               {yylval.node = new Node(yytext, "ASSIGN", 0); return ASSIGN;}
"++"              {yylval.node = new Node(yytext, "INCR_P", 0); return INCR_P;}
"--"              {yylval.node = new Node(yytext, "INCR_M", 0); return INCR_M;}
"&&"              {yylval.node = new Node(yytext, "AND", 0); return AND;}
"||"              {yylval.node = new Node(yytext, "OR", 0); return OR;}
"!"               {yylval.node = new Node(yytext, "NOT", 0); return NOT;}
"=="              {yylval.node = new Node(yytext, "EQUAL", 0); return EQUAL;}
"!="              {yylval.node = new Node(yytext, "NOTEQUAL", 0); return
NOTEQUAL;}
">"               {yylval.node = new Node(yytext, "GT", 0); return GT;}
">="              {yylval.node = new Node(yytext, "GE", 0); return GE;}
"<"               {yylval.node = new Node(yytext, "LT", 0); return LT;}
"<="              {yylval.node = new Node(yytext, "LE", 0); return LE;}
"&"               {yylval.node = new Node(yytext, "ADDRESS", 0); return
ADDRESS;}
```

- **括号及标点**

```
"("               {yylval.node = new Node(yytext, "OPENPAREN", 0); return
OPENPAREN;}
")"               {yylval.node = new Node(yytext, "CLOSEPAREN", 0); return
CLOSEPAREN;}
"["               {yylval.node = new Node(yytext, "OPENBRACKET", 0); return
OPENBRACKET;}
"]"               {yylval.node = new Node(yytext, "CLOSEBRACKET", 0); return
CLOSEBRACKET;}
"{"               {yylval.node = new Node(yytext, "OPENCURLY", 0); return
OPENCURLY;}
"}"               {yylval.node = new Node(yytext, "CLOSECURLY", 0); return
CLOSECURLY;}
","               {yylval.node = new Node(yytext, "COMMA", 0); return COMMA;}
";"               {yylval.node = new Node(yytext, "SEMI", 0); return SEMI;}
```

- 用户自定义变量以及常量

```
digit   [0-9]
letter   [a-zA-Z]
ID      {letter}({digit}|{letter}|"_")*

ICONST   "0"|[1-9]{digit}*
FCONST   "0"|[1-9]{digit}*"."{digit}+
CCONST   \'.\'|\'\\.\'
SCONST   \"(\\.|[^"\\])*\"
%%
{ID}              {yylval.node = new Node(yytext, "ID", 0); return ID;}
{ICONST}          {yylval.node = new Node(yytext, "Integer", 0); return
Integer;}
{FCONST}          {yylval.node = new Node(yytext, "Realnumber", 0); return
Realnumber;}
{CCONST}          {yylval.node = new Node(yytext, "Character", 0); return
Character;}
{SCONST}          {yylval.node = new Node(yytext, "String", 0); return String;}
```

- 注释以及其它转义符

```
COMMENT ("//"[^\n]*)
%%
{COMMENT}         { ;}
"\n"              {lineno = lineno + 1;}
"\t"              { ;}
" "               { }
.                 {yyerror("Unrecognized character!\n");}
```

第二章 语法分析

2.1 Yacc介绍

Yacc(Yet Another Compiler Compiler)是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器），它使用巴克斯范式(BNF)定义语法，能处理上下文无关文法(context-free)。Yacc生成的编译器主要是用C语言写成的语法解析器（Parser），需要与词法解析器Lex一起使用，再把两部份产生出来的C程序一并编译。
与Lex相似，Yacc的输入文件由以%%分割的三部分组成，分别是声明区、规则区和程序区。三部分的功能与Lex相似，不同的是规则区的正则表达式替换为CFG，在声明区要提前声明好使用到的终结符以及非终结符的类型。

```
/* definitions */
//....
%%
/* rules */
//....
%%
/* auxiliary routines */
//....
```

定义部分：定义部分包括有关语法定义中使用的标记的信息，还可以包括解析器和变量声明之外的C代码，位于第一列的%{和%}中。

规则部分：规则部分包含修改后的BNF格式的语法定义。动作是{}中的C代码，可以嵌入其中。

辅助例程部分：辅助例程部分仅是C代码，它包括规则部分中所需的每个函数的函数定义。如果解析器要作为程序运行，它也可以包含main()函数定义，main()函数必须调用函数yyparse()。

2.2 抽象语法树

在计算机科学中，抽象语法树（Abstract Syntax Tree，AST），或简称语法树（Syntax tree），是源代码语法结构的一一种抽象表示。它以树状的形式表现编程语言言的语法结构，树上的每个节点都表示源代码中的一一种结构。

之所以说语法是"抽象"的，是因为这里里里的语法并不不会表示出真实语法中出现的每个细节。比比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而而类似于if-condition-then这样的条件跳转语句句，可以使用用带有两个分支支的节点来表示。

2.2.1 Node类

我们将抽象语法树的所有节点抽象为一个Node类，Node有结点名、结点类型、儿子数、行数等属性，也实现了了初始化、添加结点、设置类型、获取类型等相关函数。

对于结点类型，我们定义如下：

```
/**
* @brief
* the type of the node, valid values for nodeType
*  "Program":                  the root node of the AST,
*  "GlobalDefinitionList":     node of list of all the function and global var's
definition,
*  "GlobalDefinition":         node of function or global var's definition,
*  "Typer":                    node refers 4 valid data type identifier,
*  "GlobalVariableList":       node of list of global var's definition for
current data type's definition
*  "GlobalVariable":           node of global var's definition
```

```
*  "Function":                node of function's definition
*  "ParameterList":           node of list of function's parameters
*  "Parameter":               node of single parameter
*  "FunctionCode":            node of all the instructions in a function
*  "Instruction":             node of single instruction
*  "Definition":              node of a definition instruction
*  "LocalVariableList":       node of list of local vars' definition
*  "LocalVariable":           node of local var's definition
*  "Statement":               node of a instruction but definition
*  "Expression":              node of a expression
*  "Arguments":               node of arguments
*/
string node_Type;
```

其中对于每个结点类型都有相应的描述，在此不再阐述。

2.3 语法分析的具体实现

2.3.1 定义部分

定义AST根节点指针：

```
extern Node *ASTroot;
```

定义token：

```
%token <node> CHAR INT FLOAT DOUBLE VOID
%token <node> Integer Realnumber Character
%token <node> IF ELSE FOR WHILE CONTINUE BREAK RETURN
%token <node> ID
%token <node> COMMA SEMI
%token <node> PLUS MINUS MUL DIV MOD
%token <node> ASSIGN
%token <node> INCR
%token <node> ADDRESS
%token <node> AND OR NOT
%token <node> EQUAL NOTEQUAL GT GE LT LE
%token <node> OPENPAREN CLOSEPAREN OPENBRACKET CLOSEBRACKET OPENCURLY CLOSECURLY
```

定义非终结符：

```
%type <node> Program GlobalDefinitionList GlobalDefinition Typer
%type <node> GlobalVariableList GlobalVariable Function ParameterList Parameter
%type <node> FunctionVariable FunctionCode Instruction Definition
LocalVariableList
%type <node> LocalVariable Statement Expression Arguments
```

定义优先级和结合性：

```
%right ASSIGN
%left OR
%left AND
%left EQUAL NOTEQUAL GT GE LT LE
%left PLUS MINUS
%left MUL DIV MOD
%right NOT
%left OPENPAREN CLOSEPAREN OPENBRACKET CLOSEBRACKET
```

2.3.2 规则部分

按自底向上的顺序构造语法树，部分文法如下：

```
Program
    :           GlobalDefinitionList {
                    $$ = new Node("", "Program", 1, $1);
                    ASTroot = $$;
                }
    ;

GlobalDefinitionList
    :           GlobalDefinition GlobalDefinitionList {
                    $$ = new Node("", "GlobalDefinitionList", 2, $1, $2);
                }
    |           %empty {
                    $$ = nullptr;
                }
    ;

GlobalDefinition
    :           Typer GlobalVariableList SEMI {
                    $$ = new Node("", "GlobalDefinition", 3, $1, $2, $3);
                }
    |           Typer Function {
                    $$ = new Node("", "GlobalDefinition", 2, $1, $2);
                }
    ;

Typer
    : //...
    ;

GlobalVariableList
    :           GlobalVariable COMMA GlobalVariableList {
                    $$ = new Node("", "GlobalVariableList", 3, $1, $2, $3);
                }
    |           GlobalVariable {
                    $$ = new Node("", "GlobalVariableList", 1, $1);
                }
    ;

GlobalVariable
    : //...
    ;
```

```
Function
    :           ID OPENPAREN ParameterList CLOSEPAREN OPENCURLY FunctionCode
CLOSECURLY {
                    $$ = new Node("", "Function", 7, $1, $2, $3, $4, $5, $6,
$7);
                }
    ;

ParameterList
    : //...
    ;

Parameter
    :           Typer FunctionVariable {
                    $$ = new Node("", "Parameter", 2, $1, $2);
                }
    ;

FunctionVariable
    : //...
    ;

FunctionCode
    : //...
    ;

Instruction
    :           Definition {
                    $$ = new Node("", "Instruction", 1, $1);
                }
    |           Statement {
                    $$ = new Node("", "Instruction", 1, $1);
                }
    ;

Definition
    :           Typer LocalVariableList SEMI{
                    $$ = new Node("", "Definition", 3, $1, $2, $3);
                }
    ;

LocalVariableList
    :           LocalVariable COMMA LocalVariableList {
                    $$ = new Node("", "LocalVariableList", 3, $1, $2, $3);
                }
    |           LocalVariable {
                    $$ = new Node("", "LocalVariableList", 1, $1);
                }
    ;

LocalVariable
    : //...
    ;

Statement
```

```
    : //...
    ;

Expression
    : //...
    ;

Arguments
    :           Expression COMMA Arguments {
                    $$ = new Node("", "Arguments", 3, $1, $2, $3);
                }
    |           Expression {
                    $$ = new Node("", "Arguments", 1, $1);
                }
    ;

%%
```

2.4 抽象语法树可视化

在本项目中，通过visulization.cpp中定义的Json::Value Node::jsonGen()可以对抽象语法树的根节点进行前序遍历并输出到ast_tree.json中，随后利用pytm-cli树图渲染命令行工具将Json数据生成对应的HTML格式的树图即可可视化抽象语法树。

jsonGen()函数：

```
Json::Value Node::jsonGen(){
    Json::Value root;
    string addstr = "";
    if(this->node_Type == "Typer" || this->node_Type == "Exp"){
        switch (this->getValueType())
        {
        case TYPE_VOID:
            addstr = "void"; break;
        case VAR:
            addstr = "var"; break;
        case ARRAY:
            addstr = "array"; break;
        case FUN:
            addstr = "function"; break;
        case TYPE_INT:
            addstr = "int"; break;
        case TYPE_INT_ARRAY:
            addstr = "int_Array"; break;
        case TYPE_FLOAT:
            addstr = "float"; break;
        case TYPE_FLOAT_ARRAY:
            addstr = "float_Array"; break;
        case TYPE_CHAR:
            addstr = "char"; break;
        case TYPE_CHAR_ARRAY:
            addstr = "char_Array"; break;
        case TYPE_DOUBLE:
            addstr = "double"; break;
        case TYPE_DOUBLE_ARRAY:
```

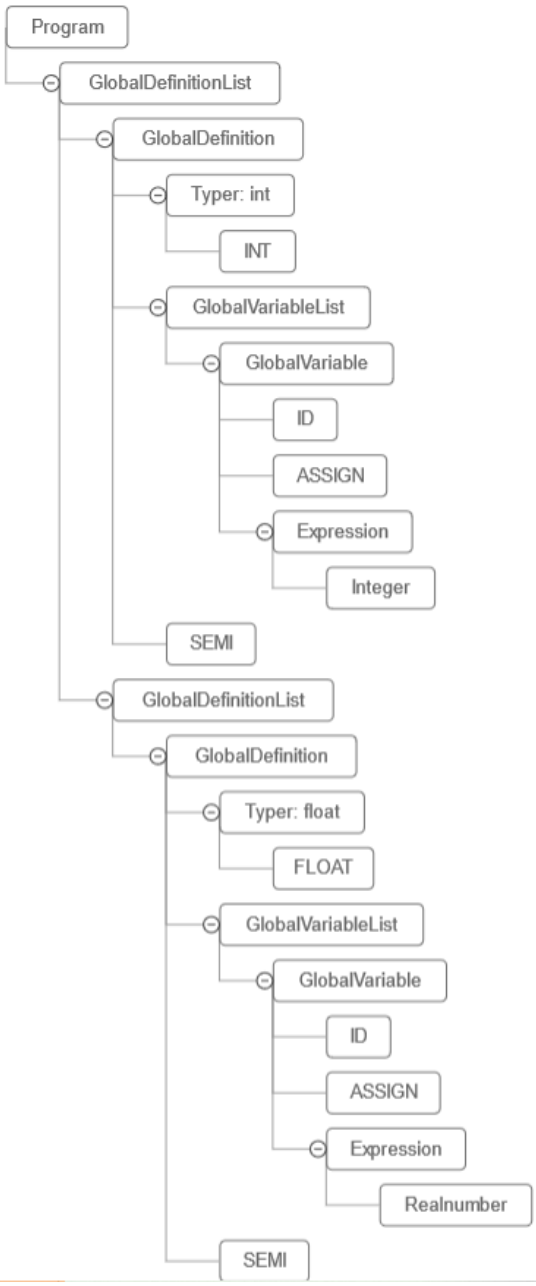
```
            addstr = "double_Array"; break;
        case POINTER:
            addstr = "pointer"; break;
        default:
            break;
        }
    }

    root["name"] = this->node_Type + (addstr == "" ? "" : ": " + addstr);

    for(int i = this->child_Num - 1; i >= 0; i--){
        if(this->child_Node[i]){
            root["children"].append(this->child_Node[i]->jsonGen());
        }
    }
    return root;
}
```

生成的抽象语法树示例：

```
int a=1;
float b=2.0;
```



第三章 语义分析

3.1 谈到语义

有一些问题需要在我们讲解语义的生成之前做一些说明。考虑到生成底层各类架构的汇编代码所需要的巨大的工作量，我起初试图完成将C语言编译成class文件这一工作，然而这至少有以下几点困难：

1. 我无法处理class文件格式中关于常量池的约束。我仍然可以在语义分析之前遍历所有的叶子结点并把const属性的字面量收集起来，但是对于这些常量的引用关系和对各个数据类型的变量的引用是不明确的，至少有一个问题导致了这个现象：C程序中的内存分为了堆和栈两部分，而运行的JVM则会将class文件中的所有的字面量放在常量池中。

```
    char* str = "Hello,world!" ;
    char tem[] = "Hello,world!" ;
    printf("%p %s\n",str,str) ;
    printf("%p %s\n",tem,tem) ;
//  这段代码实际运行一次的结果：
//  0000000000404000 Hello,world!
//  000000000061FE0B Hello,world!
```

`str`指向了和`tem`不同的区域，尽管他们都有一样的内容。

2. 我无法处理class文件对自身所代表的类的含义的解释。C并不是一个面向对象的语言，至少他不具备面向对象的语言要求。对于编译的源文件，我也不需要像Java语言那样将它当作一个对象处理，那么对于class文件格式中的类访问标识、父类索引、接口索引等等内容都无从谈起，编译器的设计者或许可以想当然地把这些部分都置零或者忽略，但JVM可能就不会这样认为了。
3. 在第二点的基础上，为了适应class的格式要求，我不得不将C中的全局变量实现成class文件中类中的静态变量，然而这与Java语言中的static语义根本就是相违背的。进而的，我无法实现C中static的语义。

除此之外还有很多没有考虑到的以及实际实现中可能会发生的问题，好在后来我们了解到更适合开发C语言的编译器的架构——LLVM，我们可以把它理解成实现在C语言的基础上的虚拟机，当然之前考虑class格式时的一些问题都不存在了。

我们采用了LLVM作为中间代码生成的工具，基于此以及其他未能说明的各种各样的问题的阻挠，我们实现的C语言的编译器终究只是一个部分成品，我们并不寄希望于实现能够分别编译出在Linux、Windows乃至MacOs上可继续链接和运行的中间代码，但我们希望作为“中间商”的LLVM能帮我们做到这一点。作为在如今编译器架构领域占有领先地位的编译器架构工具链技术，LLVM成为了很多开发语言编译工具的选择，包括LLVM自研的C编译器Clang，Rust语言以及苹果生态的swift语言。它极大地方便了我们在已经构造好的抽象语法树的基础上做后续的工作，并且通过LLVM库的api生成的LLVM IR也是一种平台兼容的可移植的中间代码，这一点与我起初企图构造Java编译器的目标class文件的初衷是一致的。

选择了LLVM工具后，为方便我们从根到叶子构建起代码的实际语义和中间代码，我们仍然做了很多基于我们有限的时间和精力范畴的对C语言的“阉割”，其中部分我认为虽然是“阉割”，但其实似乎更适合现在编程语言的规范。毕竟作为高级语言中“最不高级”的编程语言，C在很多方面赋予程序员的权限和自由现在看来在某些场合下似乎是有些多余了。

3.2 对语法的修改

3.2.1 关于变量

```
int a = 0 ;
char b = 10 ;
{
    int a = 1 ;
    {
        int b = 2 ;
        printf("%d %d\n",a,b) ;
    }
    printf("%d\n",b) ;
}
printf("%d %d\n",a,b) ;
```

不同于各种动态语言和脚本语言在变量名字上的自由，对于需要经过编译这一步骤的语言而言，通过嵌套代码段来递归查找变量名字的意义已经不大了，从实例中可以看到，重复定义同一名字的变量是允许的，他们甚至可以是不同类型的变量，包括Java语言在内的很多语言已经完全摒弃掉了这一不符合直觉的语义。上面的代码将会正常运行并且输出：

```
1 2
10
0 10
```

我们实现的C语言将不允许这样的同名变量的定义，**在一个函数的生命周期内，每一个名字只允许对应一个变量。**

3.2.2 关于函数声明

由于C语言顺序编译并且执行的特性，在使用编译器不认识的标识符时将会报错，对于函数调用而言，总是要求在出现函数调用之前必须至少编译过对应的函数声明或者函数定义的代码。在面向过程的语言的认知内这是符合直觉的，因为机器并不理解未曾见过的函数，但是在我们的编译器中我暂时删去了函数声明这一部分，原因主要有两点：

1. 作为编译器而言，我暂时并不考虑链接器和以后的工具才涉及到的多文件的编译过程，在单文件中，把函数定义放在使用到的地方之前似乎仅仅是复制粘贴的成本。作为程序员而言，任何2个及以上的函数循环调用的情况都是值得被避免的，将他们修改成递归的调用和辅助的函数应该更符合代码的规范。
2. 在中间代码的生成过程中，将所有的全局标识符进行一个预编译的过程能完全避免函数未声明的问题，我认为这是值得去做的工作尽管在目前的编译器当中我们还未实现。

3.2.3 关于输入和输出

同很多高级语言一样，我们实现了两个函数关键字`print`和`input`，尽管没有`lua`、`python`的原生函数那般完备，但这两个函数已足够进行一些简单的变量的读入和打印。

3.2.4 删除指针

我们没有实现指针这一数据类型，学过操作系统的我们知道C程序看到的指针的地址并不是真实的物理地址而是经过操作系统映射的虚拟地址，这个地址在我们需要用到具体的、较大的空间时是有意义的。对于我们只编译当前文件的编译器而言，摒弃这一类型是无关紧要的，以我在3.1中举例时使用的代码为例，`char* str`和`char tem[]`指向的是不同的地址，因为他们一个是指向内容为`“Hello,world!"`的指针，另一个是在栈中开辟了一块空间存放`"Hello,world!"`这一字符串内容。既然我们不考虑链接器的工作，只实现对当前文件的编译，甚至未考虑标准库中的`malloc`函数的使用，我们的程序实际上基本都是在栈区进行内存的开销，我们不允许`int* var`这类变量并不会对代码实现造成多大的开发障碍。

3.2.5 其他未完成

在当前的编译条件下，仍然有很多C的简洁的语法我们未能实现，比如定义数组时直接使用`{}`进行初始化等等，这些间接的语法可以通过先定义具体长度的数组再赋值等代码手段来代替，我们期待在后续的编译器版本中继续进行这一部分工作的完善。

3.3 LLVM的分析过程

3.3.1 LLVM IR

LLVM IR是LLVM虚拟机工作的中间代码，LLVM将其转化为其他架构的汇编代码以及可执行文件，使用LLVM库中的API我们可以快捷地实现LLVM IR的生成。每个IR文件称为一个Module，它是其他所有IR对象的顶级容器，包含目标信息、全局符号和所依赖的其他模块以及符号表等对象的列表，其中全局符号又包括了全局变量、函数声明和函数定义。函数由参数和多个基本块组成，其中第一个基本块称为entry基本块，这是函数开始执行的起点，另外LLVM的函数拥有独立的符号表，可以对标识符进行查询和搜索。每一个基本块包含了标签和各种指令的集合，标签作为指令的索引用于实现指令间的跳转，指令包含Phi指令、一般指令以及终止指令等。

3.3.2 IR生成过程

不同于语法分析时自底向上的建立抽象语法树的过程，进行语义分析时我们采取自顶向下的顺序。在语法分析步骤中我们已经建立起以`Program`为根的抽象语法树，我们遍历树从而建立每一部分的具体语义。具体的我们实现了下列这些方法来供构造过程调用，他们中有些可以从语法分析的生成规则中理解，有些则需要结合C语言的表达式规则作出解释。

```
    // 遍历的起点的调用
    llvm::Value *irBuild();
    // 生成变量的过程
    llvm::Value *irBuildVariable();
    // 生成函数的过程
    llvm::Value *irBuildFunction();
    // 生成C表达式的过程
    llvm::Value *irBuildExpression();
    // 生成以分号结尾的C的命令
    llvm::Value *irBuildStatement();
    // while循环结构
    llvm::Value *irBuildWhile();
    // if分支结构
    llvm::Value *irBuildIf();
    // return命令
    llvm::Value *irBuildReturn();
    // 构造完整的函数内代码的过程
    llvm::Value *irBuildCode();
    // 用于生成逻辑运算的结果
    llvm::Value *irBuildComparer();

    // 调用原生函数print和input时发生
    llvm::Value* irBuildPrint();
    llvm::Value* irBuildInput();
    // 即使在不引用标准头文件的代码中依然希望能正常使用标准格式的打印和输入
    llvm::Value* irBuildPrintf() ;
    llvm::Value* irBuildScanf() ;

    // 为了构造赋值运算的表达式我们需要构造左值和右值的概念
    llvm::Value* irBuildLeftValue() ;
    llvm::Value* irBuildRightValue() ;

    // 单条指令的构造函数，又可以分别生成变量定义和statement两种语句，在构造过程中他们将分别调
用irBuildStatement和         irBuildVariable两个方法
    llvm::Value* irBuildInstruction() ;

    // 处理代码中出现的字面量，包括Integer，Float和String
    llvm::Value* irBuildConst() ;
```

```
    // 分别处理如负号、取反等的一元运算符和加减乘除的二元运算符
    llvm::Value* irBuildUnaryOperator() ;
    llvm::Value* irBuildBinaryOperator() ;

    // 单独处理函数调用的命令
    llvm::Value* irBuildCallFunction() ;
```

此外，IR中间代码的生成依赖LLVM提供的一些运行时环境的搭建，为了使我们的项目结构更加清晰，我们把编译器的全局变量都定义在文件`globals.cpp`中：

```
// 语法分析的根节点
Node* ASTroot ;
// 上下文环境
llvm::LLVMContext context ;
llvm::IRBuilder<> builder(context) ;
// 全局Generator，实际上我们以单例模式设计了Generator，但是规避了static这样的声明
Generator generator() ;
// 用于存储分支结构的跳出地址的栈
stack<llvm::BasicBlock *> GlobalAfterBB ;
```

参考一些开源的项目结构，我们类似地构造了类Generator来帮助我们实现这一过程，在IR的生成过程中，Generator主要提供了这样一些方法帮助我们获得当前运行环境的函数及变量表：

```
    // 得到当前代码环境位于的函数
    llvm::Function* getCurFunction() ;
    // 发生函数调用时的入栈
    void pushFunction(llvm::Function* func) ;
    // 发生函数返回时的出栈
    void popFunction() ;
    // 在当前环境寻找变量名
    llvm::Value* findValue(const string & name) ;
    // 为打印和输入服务
    llvm::Function* createPrintf() ;
    llvm::Function* createScanf() ;
    // 语法分析完成后为生成IR的起点函数
    void generate(Node *root) ;
    //  得到IR对应的Module
    llvm::Module* getModule() ;
```

对于LLVM的开发环境而言，它事实上提供了`llvm::getGlobalVariable(const string name)`等等方法来帮助完成一些变量的插入和查找，我们实现的`findValue`希望能在当前环境查找不到时再去全局中寻找，这反映了我们对于C变量环境语义的修改。而局部变量的定义则依赖于当前`function`的符号表，我们以LLVM提供的一些方法去完成这些工作。

3.3.3 类型设计

关于数组的类型设计源于一些开源代码的灵感，我们设计了三类数据结构：`int,float,char`，尽管设计`bool`类型的变量对于现代很多编程语言而言已经是现状，但我们保留了C的代码习惯，选择`while(1)`而不是`while(true)`这样的表达式来完成`bool`变量的功能，因为无论是哪种编程语言`bool`类型的变量在内部实现时都实际上放在了32位的`int`大小的变量里。

```
#define TYPE_VOID (-1)
#define VAR 0
#define ARRAY 1
#define FUN 2
#define TYPE_INT 10
#define TYPE_INT_ARRAY 11
#define TYPE_INT_ARRAY_ARRAY 12
#define TYPE_FLOAT 20
#define TYPE_FLOAT_ARRAY 21
#define TYPE_FLOAT_ARRAY_ARRAY 22
#define TYPE_CHAR 30
#define TYPE_CHAR_ARRAY 31
#define TYPE_CHAR_ARRAY_ARRAY 32
```

关于为什么这些定义的数值是这么大，这就和class文件中为什么文件标识符是“CAFEBABE”一样任性，尽管我们目前只做到了二维数组的定义和使用，但是以长远的目光来看，将类型和对应的数组类型以ARRAY=1为单位增加也能让我们使用到这些类型宏的代码更加清晰，并且保留了足够高的维度来定义，当我试图定义3维、4维甚至9维数组时我仍然可以正常使用这些类型，若有10维及以上的数组则需要修改这些宏了。

设计这些宏的初衷仅仅是为了完善树的结点的构造时写得更加方便，LLVM并不认识这些类的宏，当我们要获得LLVM中代表类型的对象时我们需要根据这些类型来生成对应的LLVM中的对象，函数`llvm::Type* getLlvmType()`实现了这一目标，在最多生成二维数组的情况下，我设计重载该函数来分别生成一维数组和二维数组的`Type`对象，在完善生成更高维的数组的时候，我们可以采取可变个数的参数来实现他们。

3.3.4 全局的定义

我们可以简单的把C的代码看作是由很多个全局的定义组成，有的是变量，有的是函数，编译链接后他们都在当前程序的全局可被访问和调用。定义变量和定义函数一样都从一个类型名开始，所以我们把一个定义的单元作为语法生成的一个规则，不同的是由于代码的自由，我们可以在一条变量定义里定义多个变量，甚至定义不同类型的变量（比如`int a,b[10];`这样既有变量又有数组的定义），但是一个函数定义就只对应了一个函数。对于全局变量，通过LLVM的全局变量的类`llvm::GlobalVariable`的构造方法构造，对于函数则通过静态方法`llvm::Function::Create()`生成。

3.3.5 函数内代码的结构

函数生成规则`Function --> ID ( ParameterList ) { FunctionCode }`中的`FunctionCode`通过生成函数`irBuildCode()`来实现IR的生成。`FunctionCode`由若干条`Instruction`组成，`Instruction`又可以分为变量定义命令和其他的命令。函数内变量的定义和全局变量的定义调用的实际上都是`irBuildVariable()`函数，生成变量时我会判断当前的代码运行时环境是全局还是某个函数内，这一点有赖于上文提及到的`llvm::LLVMContext context`以及对应的由`context`构造的`llvm::IRBuilder<> builder(context)`这两个全局变量。除了变量定义以外的语句都通过`Statement`生成，每一个Statement以分号结尾，并且又可以生成表达式、while循环、if判断分支和返回这些语句。

构造表达式的过程是较为复杂的，通过多种多样的表达式我们能够写出功能复杂的C代码，为了使代码结构更加简洁明了我们又大致地对这些表达式做了一些区分来分别生成：

```
llvm::Value *Node::irBuildExpression(){
    // Expression --> %empty
    if ( this == nullptr )
        return NULL ;
```

```
    // Expression --> Integer | Realnumber | Character
    if ( this->child_Num == 1 )
        return this->irBuildConst() ;
    /**
     * Expression --> NOT Expression
     * Expression --> MINUS Expression
     * Expression --> INCR Expression
     * Expression --> ADDRESS Expression
     * Expression --> Expression INCR
     */
    if ( this->child_Num == 2 )
       return this->irBuildUnaryOperator() ;
    /**
     *  Expression --> Expression EQUAL Expression
     *  Expression --> Expression NOTEQUAL Expression
     *  Expression --> Expression GT Expression
     *  Expression --> Expression GE Expression
     *  Expression --> Expression LT Expression
     *  Expression --> Expression LE Expression
     */
    if ( this->child_Num == 3 &&
            ( this->child_Node[1]->node_Type == "EQUAL" ||
              this->child_Node[1]->node_Type == "NOTEQUAL" ||
              this->child_Node[1]->node_Type == "GT" ||
              this->child_Node[1]->node_Type == "GE" ||
              this->child_Node[1]->node_Type == "LT" ||
              this->child_Node[1]->node_Type == "LE" ) )
        return this->irBuildComparer() ;
    /**
     * Expression --> Expression ASSIGN Expression
     * Expression --> Expression AND Expression
     * Expression --> Expression OR Expression
     * Expression --> Expression PLUS Expression
     * Expression --> Expression MINUS Expression
     * Expression --> Expression MUL Expression
     * Expression --> Expression DIV Expression
     * Expression --> Expression MOD Expression
     */
    if ( this->child_Num == 3 && this->child_Node[2]->node_Type == "Expression"
&& this->child_Node[0]->node_Type == "Expression" )
        return this->irBuildBinaryOperator() ;
    // Expression --> OPENPAREN Expression CLOSEPAREN
    if ( this->child_Num == 3 && this->child_Node[1]->node_Type == "Expression"
)
        return child_Node[1]->irBuildExpression() ;
    /**
     * Expression --> ID OPENBRACKET Expression CLOSEPAREN OPENBRACKET Expression
CLOSEPAREN
     * Expression --> ID OPENBRACKET Expression CLOSEPAREN
     * Expression --> ID
     */
    if ( this->child_Node[0]->node_Type == "ID" && ( this->child_Num == 1 ||
this->child_Node[1]->node_Type == "OPENBRACKET" ) )
        return this->irBuildRightValue() ;
    /**
```

```
     * Expression --> ID OPENPAREN Arguments CLOSEPAREN
     * Expression --> ID OPENPAREN CLOSEPAREN
     */
    if ( this->child_Node[0]->node_Type == "ID" && this->child_Node[1]-
>node_Type == "OPENPAREN" )
        return this->irBuildCallFunction() ;
}
```

这里具体实现的函数不再一一赘述，对于`irBuildRightValue()`做一些简要的说明，因为与之相应的我也实现了左值的生成函数用来实现赋值运算。从字面意义理解，左值便是可以出现在赋值符号左侧的表达式，右值即可以出现在赋值符号右侧的表达式。对于C语言而言，除了变量名以外，基础运算的表达式和函数调用都只能出现在赋值符号右侧，而左值则只能为变量或者一些取地址取值之类的表达式。考虑到指针的运算没有涉及到，我们处理的左值便只包含`ID`、`ID[Expression]`和`ID[Expression][Expression]`，乍一看似乎和右值的情况无异，但是注意到我们的`Expression`并不包含直接生成左值，但是包含直接生成右值，这是因为右值运算生成的结果可以被直接作为结果返回，而左值生成的结果并不是表达式的计算结果，而是`ID`、`ID[Expression]`和`ID[Expression][Expression]`的存储单元的"地址"，他们只会在赋值运算中使用到，将右值的结果`store`到左值的地址中。

3.3.6 类型匹配和转换

在语义分析的部分，我们希望能够完成对所有只含常数的表达式的直接计算，这样的优化可以减少部分表达式的编译代码。在表达式计算的过程中，有以下这些情况需要我们进行类型的自动转换：

1. 表达式中有常数类型不一致的计算；
2. 赋值运算左值和右值类型不一致。

对于表达式中类型不一致的常数，考虑到我们只生成整型和浮点型的字面量，所以可以直接转换为`float`类型的数据进行计算，这个过程可以在语法分析的过程中完成，生成的结果也被当作浮点型字面量以继续递归地调用来完成整个常数字面量表达式的计算。对于赋值运算的左值类型和右值类型不一致的情况，则把右值转换成左值进行赋值。赋值的类型转换不同于表达式中的类型转换，因为左值可以向`float`转换，也可以反过来由`float`转换成`int`，LLVM提供了部分定义良好的操作符来实现转换：

```
// Cast operators ...
// NOTE: The order matters here because CastInst::isEliminableCastPair
// NOTE: (see Instructions.cpp) encodes a table based on this ordering.
 FIRST_CAST_INST(38)
HANDLE_CAST_INST(38, Trunc   , TruncInst   )  // Truncate integers
HANDLE_CAST_INST(39, ZExt    , ZExtInst    )  // Zero extend integers
HANDLE_CAST_INST(40, SExt    , SExtInst    )  // Sign extend integers
HANDLE_CAST_INST(41, FPToUI  , FPToUIInst  )  // floating point -> UInt
HANDLE_CAST_INST(42, FPToSI  , FPToSIInst  )  // floating point -> SInt
HANDLE_CAST_INST(43, UIToFP  , UIToFPInst  )  // UInt -> floating point
HANDLE_CAST_INST(44, SIToFP  , SIToFPInst  )  // SInt -> floating point
HANDLE_CAST_INST(45, FPTrunc , FPTruncInst )  // Truncate floating point
HANDLE_CAST_INST(46, FPExt   , FPExtInst   )  // Extend floating point
HANDLE_CAST_INST(47, PtrToInt, PtrToIntInst)  // Pointer -> Integer
HANDLE_CAST_INST(48, IntToPtr, IntToPtrInst)  // Integer -> Pointer
HANDLE_CAST_INST(49, BitCast , BitCastInst )  // Type cast
HANDLE_CAST_INST(50, AddrSpaceCast, AddrSpaceCastInst)  // addrspace cast
  LAST_CAST_INST(50)
```

考虑到`Trunc`运算符直接截断操作数再赋值的特点，在需要转换成较短的结果时最好通过特别的代码来实现而不是截断。利用这些转换的操作符，在`Node`类中实现了静态方法来完成类型转换的工作：

```
llvm::Value* Node::typeCast(llvm::Value* src, llvm::Type* dst){
    llvm::Instruction::CastOps op = getCastOperator(src->getType(), dst);
    return builder.CreateCast(op, src, dst, "tmptypecast");
}
```

第四章 代码生成

第五章 测试结果

5.1 数据类型测试

普通类型测试

测试代码：

AST：

IR：

汇编指令：

结果：

数组测试

测试代码：

AST：

IR：

汇编指令：

结果：

5.2 数据运算测试

测试代码：

AST：

IR：

汇编指令：

结果：

5.3 控制流测试

循环语句测试

测试代码：

AST：

IR：

汇编指令：

结果：

条件语句测试

测试代码：

AST：

IR：

汇编指令：

结果：

5.4 函数测试

简单函数测试

测试代码：

AST：

IR：

汇编指令：

结果：

递归函数测试

测试代码：

AST：

IR：

汇编指令：

结果：

5.5 综合测试

测试样例1——quicksort

测试代码：

AST：

IR：

汇编指令：

结果：

测试样例2——matrix-multiplication

测试代码：

AST：

IR：

汇编指令：

结果：

测试样例3——auto-advisor

测试代码：

AST：

IR：

汇编指令：

结果：

第六章 总结