華中科技大學

课 程 实验 报 告

题目：c 语言编译器的设计与实现

课程名称：编译技术实验

专业班级：软工 1805

学　　号：U201817092

姓　　名：易可欣

指导教师：胡雯蔷 徐丽萍 祝建华

报告日期：2020.12.3

软件学院

目录

目录

1 概述

1.1 任务

本次实验是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的喜好选择实现。建议大家选用 decaf 语言或 C 语言的简单集合 SC 语言。

实验的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生系统软件研发技术。

1.2 目标

实现一个 C 语言的编译器，目标代码是汇编语言。按照词法分析，语法分析，语义分析，中间代码生成的顺序逐步实现，准确报出每个步骤出现的错误，对正确的源代码生成目标 MIPS32 代码，并在模拟程序中正确执行。

1.3 语言定义

本实验实现了一个 C 语言的编译器，命名为 miniC，是 C 语言的子集，语言语法与 C 语言十分类似.

1.4 工具描述

Flex

Flex 是一个生成词法分析器的工具，它可以利用正则表达式来生成匹配相应字符串的 C 语言代码，其语法格式基本同 Lex 相同。

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。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() 加入自己的工程文件中使用。

当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行

匹配。 它一次读入一个输入字符，直到找到一个匹配的模式。 如果能够找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。 另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

Bison

Bison 是属于 GNU 项目的一个语法分析器生成器。Bison 把一个关于“向前查看 从左到右 最右”(LALR) 上下文无关文法的描述转化成可以分析该文法的 C 或 C++ 程序。它也可以为二义文法生成 “通用的 从左到右 最右”(GLR)语法分析器。

1.4 开发环境

环境	版本
OS	Windows 10
flex	2.5.4
bison	2.4.1
gcc	4.4.1
MIPS 模拟器	Qtspim 9.1.19

2 系统描述

2.1 自定义语言概述

1.数据类型包括 char、int、float
2.基本运算包括算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算
3.控制语句包括 if、while、for、break、continue
4.包括一维数组、二维数组

2.2 单词文法与语言文法

单词文法：

单词是 MiniC 语言中具有独立意义的最小单位，可分为 5 大类：关键字（保留字）、运算符、界符、常量和标识符。

关键字包括：

基本的表示类型的关键字：int、float、char

分支循环控制字：if、else、while、break、continue、for

函数返回关键字：return

运算符包括：

“+ - * / < <= > >= = == != && || ! [] ()”，涉及到了算术运算和逻辑运算。

界符包括：

“; , . {}”。

常量包括：

空指针（null）、浮点数常量、整型常量（十进制）、字符常量。

语言文法：

程序结构：

一个 MiniC 程序是由函数、变量定义的序列组成。

一个 MiniC 程序中必须包含一个名为“main”的主函数，以 int 类型参数作为返回值，不带任何参数。

类型：

基本类型包括 int float char，数组类型可以通过任何基本类型建立起来。

函数：

函数的定义用于建立函数名字以及与这个名字相关联的类型签名，类型签名包括函数是否是静态的、返回值类型、形参表的大小以及各形参的类型。函数的定义提供类型签名以及组成函数体的语句。函数之间不允许嵌套。函数可以有零或者多个形参。形参的类型可以是 int、float、char，不允许有数组类型的形参。

用在形参列表中的标识符必须是唯一的（即形参不能重名）。函数的返回类型可以是任何的基本类型。一个函数只能被定义一次。函数中的任何 return 语句必须返回一个与该返回类型兼容的值。

所调用的函数必须是有定义的，无论其定义是否出现在调用处之前。函数调用中实参的个数必须与函数所需形参的个数相匹配。函数调用中每个实参的类型必须与对应形参的类型相匹配。函数调用时实参的求值顺序是从左至右。函数调用过程中执行到一个 return 语句或者到达函数在源程序中的结尾时把控制权交还给调用方。函数调用结果的类型是函数声明时候的返回值类型。

作用域：

MiniC 支持多层次的作用域。每个函数都有一个用户声明形参表的参数作用域和存放函数体的局部作用域。局部作用域的一对大括号建立了一个嵌套的局部作用域。内层作用域屏蔽外层作用域。需要注意的是：

1.局部作用域的变量必须先声明后使用，而不能在声明的同时赋初值。

2.在同一个作用域中，标识符不可互相冲突

3.在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，但不允许在局部作用域中声 明与外层的局部作用域或参数作用域中的变量同名的变量。

4.不可以再访问闭作用域中声明的标识符。

变量：

一旦被声明，则该变量保持可见直到该作用域关闭。需要注意的是：局部变量可以在语句序列的任意地方声明，而且在声明点到该声明所在的作用域末之间的区域可访问。

2.3 符号表结构定义

符号表是一个顺序栈，栈中的每一个元素为 symbol 类型的结构体，该结构体记录了符号的基本信息。symbol 结构体定义如下：

类型	名称	描述
char[]	name	符号名（临时变量无符号名）
int	level	层号，全局变量为 0，形参名为 1
int	type	变量类型或返回值类型
int	paramnum	若符号为函数名，记录其参数个数
char[]	alias	别名（实验三、四使用）
char[]	scope	作用域（打印符号表时标识符号作用域）
int	offset	偏移量（实验四使用）

符号表维护了一个栈结构，由全局结构体 symboltable 定义，该结构体有 2

个成员变量，一个为 symbol 数组类型的 symbols，用于存放符号，一个为 int 类型用于维护栈顶索引。

对于插入符号表，定义 2 个函数，fillSymbolTable 用于压入变量或函数，fill_Temp 用于压入临时变量，该函数在实验三、实验四中用到。

定义查询符号表的函数 searchSymbolTable，传入查询的变量或函数名，返回其在符号表中的索引，若不存在则返回-1，用于分析变量或函数的重名。

2.4 错误类型码定义

(1)函数重复定义
(2)没有返回语句
(3)break 在循环外测试
(4)continue 在循环外测试
(5)对非函数名采用函数调用的形式
(6)对函数名采用非函数形式访问
(7)使用未定义的变量
(8)调用未定义的函数
(9)形参和实参类型不匹配
(10)对非数组变量采用下标变量的形式访问
(11)数组变量的下标不是整型表达式
(12)对非结构变量采用成员选择运算符
(13)函数调用时参数个数不匹配(过少）
(14)函数调用时参数个数不匹配(过多）
(15)赋值号左边不是左值表达式
(16)对非左值表达式进行自增、自减运算；
(17)类型不匹配。
(18)函数返回值类型与函数定义的返回值类型不匹配；
(19)双目运算两边值不匹配

2.5 中间代码结构定义

通过前面对 AST 遍历，完成了语义分析后，如果没有语法语义错误，就可以再次对 AST 进行遍历，计算相关的属性值，建立符号表，并生成以三地址代码 TAC 作为中间语言的中间语言代码序列，具体格式为 Result := Opn1 Op Opn2，其中，Op 为操作名称， Result、Opn1、 Opn2 分别为目的操作数、源操作数 1、源操作数 2。

在程序中，定义结构体类型 opn，存储操作数的有关信息，opn 的定义如下：

类型	名称	描述
int	kind	标识联合体成员的属性
int	type	标识操作数的数据类型
union int **float** **char** **char[]**	const_int const_float const_char id	操作数为整型常量 操作数为浮点型常量 操作数为字符型常量 变量或临时变量的别名
int	level	变量的层号，0 表示全局变量，数据保存在静态数据区
int	offset	局部变量或临时变量对于函数头的偏移，全局变量对程序头的偏移

中间代码生成规则定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			X
FUNCTION f:	定义函数 f	FUNCTION			F
x := y	赋值操作	ASSIGN	X		X
x := y + z	加法操作	PLUS	Y	Z	X
x := y - z	减法操作	MINUS	Y	Z	X
x := y * z	乘法操作	STAR	Y	Z	X
x := y / z	除法操作	DIV	Y	Z	X
GOTO x	无条件转移	GOTO			X
IF x [relop] y GOTO z	条件转移	[relop]	X	Y	Z
RETURN x	返回语句	RETURN			X
ARG x	传实参 x	ARG			X
x:=CALL f	调用函数	CALL	F		X
PARAM x	函数形参	PARAM			X
READ x	读入	READ			X

WRITE x	打印	WRITE			X

中间代码的生成要符合上述表格的要求来实现。TAC 语句中的变量名字对应一个存储位置，变量名字对应的存储位置信息（相对于基地址的偏移量）总是可以从符号表中得到，可以认为变量的取值即为其名字对应的存储位置上存储单元的内容。对于运算语句，赋值号右边的值，用临时变量存储，再用赋值语句赋值给目标操作数。

2.6 目标代码指令集选择

目标语言选定 MIPS32 指令序列，可以在 SPIM Simulator 上运行。TAC 指令和 MIPS32 指令的对应关系如下表所示。其中 reg(x)表示变量 x 所分配的寄存器。

中间代码	MIPS32 指令
LABEL x	x：
x :=#k	li reg(x),k
x := y	move reg(x), reg(y)
x := y + z	add reg(x), reg(y) , reg(z)
x := y - z	sub reg(x), reg(y) , reg(z)
x := y * z	mul reg(x), reg(y) , reg(z)
x := y / z	div reg(y) , reg(z) mflo reg(x)
x := y % z	rem reg(x), reg(y) , reg(z)
GOTO x	j x
RETURN x	move $v0, reg(x) jr $ra
IF x==y GOTO z	beq reg(x),reg(y),z
IF x!=y GOTO z	bne reg(x),reg(y),z
IF x>y GOTO z	bgt reg(x),reg(y),z
IF x>=y GOTO z	bge reg(x),reg(y),z
IF x<y GOTO z	ble reg(x),reg(y),z

IF x<=y GOTO z	blt reg(x),reg(y),z
X:=CALL f	jal f move reg(x),$v0

3 系统设计与实现

3.1 词法分析器

词法分析器可采用词法生成器自动化生成工具 GNU Flex，该工具要求以正则表达式（正规式）的形式给出词法规则，遵循上述技术线路，Flex 自动生成给定的词法规则的词法分析程序。于是，设计能准确识别各类单词的正则表达式就是关键。

第一个部分为定义部分，其中可以有一个**%{ 到%}**的区间部分，主要包含c语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明。会直接被复制到词法分析器源程序lex.yy.c中。

%{ 到%}之外的部分是一些正则式宏名的定义，对于我设定的标识符和类型其正则表达式如下：

```
id    [A-Za-z][A-Za-z0-9]*
int     [0-9]+
float   ([0-9]*\.[0-9]+)|([0-9]+\.)
char    \'[A-Za-z0-9!!@#$%\^&\*()_\+-=?<>,.\n\r\t]\'
```

第二个部分为规则部分，一条规则的组成为：

正则表达式　　动作

这部分以正则表达式的形式，罗列出所有种类的单词，表示词法分析器一旦识别出该**正则表达式**所对应的单词，就执行**动作**所对应的操作，返回单词的种类码。

其中罗列自己定义的所能识别的单词和动作：

```
"int"         {strcpy(yylval.type_id,  yytext);return TYPE;}
"float"       {strcpy(yylval.type_id,  yytext);return TYPE;}
"void"       {strcpy(yylval.type_id,  yytext);return TYPE;}


"return"      {return RETURN;}
"if"          {return IF;}
"else"        {return ELSE;}
"while"       {return WHILE;}
"for"         {return FOR;}
"break"       {return BREAK;}
"continue"    {return CONTINUE;}


{id}        {strcpy(yylval.type_id,  yytext); return ID;/*由于关键字的形式
";"           {return SEMI;}
","           {return COMMA;}
">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type_id, yytext);;return RELOI
"="           {return ASSIGNOP;}
"++"          {return DPLUS;}
"--"          {return DMINUS;}
"+"           {return PLUS;}
"-"           {return MINUS;}
"*"           {return STAR;}
"/"           {return DIV;}
"&&"          {return AND;}
"||"          {return OR;}
"!"           {return NOT;}
"("           {return LP;}
")"           {return RP;}
"{"           {return LC;}
"}"           {return RC;}
"["           {return LB;}
```

```
"//".*        {}
"/*"(.|\n)*"*/"      {}
```

其中对于注释是这样直接屏蔽的，上面是//类型注释，下面是/* */型注释。

每当词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng，供后续使用。

第三个部分为用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序lex.yy.c中。

为了能在词法分析和语法分析报错时提供错误的详细位置信息，运用了 Flex 的部分高级特性，例如：开启 yylineno 选项，从而全局变量 yylineno 会记录当前正在分析的词法单元在源程序中的行号，并由 Flex 自行维护（初值设为 1）。

3.2 语法分析器

语法分析采用生成器自动化生成工具 GNU Bison（前身是 YACC），该工具采用了 LALR（1）的自底向上的分析技术，完成语法分析。在实验的语法分析阶段，当语法正确时，生成抽象语法树，作为后续语义分析的输入。在具体的转化过程中，有一个问题：由 MiniC 语法规范直接转化来的生成式存在大量的移进-规约冲突或规约-规约冲突，需要通过一定的方法来消除冲突。

parser.y 程序结构如下：

声明等部分略过。

辅助定义部分（%union），是语义值的类型定义，将所有符号属性值的类型用联合的方式统一起来后，某个符号的属性值就是该联合中的一个成员的值。

%type 定义非终结符的语义值类型，都归为 ptr。

在 parser.y 中的%token 后面罗列出所有终结符(单词)的种类码标识符，来使用 lex.l 中识别出的单词的种类码。

二义性与冲突处理，则是通过显示规定优先级和结合性来解决。为了解决二义性与冲突，在 Bison 中定义的结合性和优先级如下所示。

```
%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left DPLUS DMINUS
%left STAR DIV
%right UMINUS NOT
```

接下来是规则部分，规则后面{}中的是当完成归约时要执行的语义动作。规则左部的 Exp 的属性值用$$表示。节选部分如下：

```
program: ExtDefList    {display($1,0);}
         ;
ExtDefList: {$$=NULL;}
          | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);}   //每一个EXTDEFLIST的
          ;
ExtDef:   Specifier ExtDecList SEMI   {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}   //该结点对应
         |Specifier FuncDec CompSt    {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}          //该结
         | error SEMI   {$$=NULL;}//若出错跳过分号继续向后进行语法分析
         ;
Specifier:  TYPE    {$$=mknode(0,TYPE,yylineno);
            strcpy($$->type_id,$1);
              if(strcmp($1,"int")==0){
                $$->type = INT;
            } else if (strcmp($1,"char")==0){
                $$->type = CHAR;
            } else if (strcmp($1,"float")==0){
                $$->type = FLOAT;
            }else if (strcmp($1,"void")==0){
                $$->type = VOID;
            }
        }
          ;
ExtDecList:  VarDec      {$$=$1;}       /*每一个EXT_DECLIST的结点，其第一棵子树对应一个变量名(I
           | VarDec COMMA ExtDecList {$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}
           ;
VarDec:  ID          {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}   //ID结点，标识符符号串
          | ID LB Exp RB   {$$=mknode(2,ARRAY,yylineno,$1,$3);strcpy($$->type_id,$1);}
          |ID LB Exp RB LB Exp RB   {$$=mknode(3,ARRAY2,yylineno,$1,$3,$6);strcpy($$->type_id,$
         ;
FuncDec: ID LP VarList RP   {$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);}//函数名
        |ID LP  RP   {$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=NULL;}//
```

最后是用户函数部分。

设计语法分析器的第二步即构建 ast.c。构造语法分析树。考虑到语法分析树是一棵各节点度都不相等的树，所以决定采用多叉树表示语法分析树，由于 MiniC 较简单，多叉树可能有的分支最多为 3，所以使用 3 叉树足够。

为此，设计并定义语法分析树的节点结构体：

```
struct ASTNode {
        //以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一
    int kind;
    union {
            char type_id[33];                   //由标识符生成的叶结点
            int type_int;                       //由整常数生成的叶结点
            float type_float;                     //由浮点常数生成的叶结点
            char type_char;
            };
    struct ASTNode *ptr[4];             //由kind确定有多少棵子树
    int place;                          //存放（临时）变量在符号表的位置序
    char Etrue[15],Efalse[15];           //对布尔表达式的翻译时，真假转移
    char Snext[15];                     //结点对应语句S执行后的下一条语句位
    char FathNext[15];
    struct codenode *code;              //该结点中间代码链表头指针
    int type;                           //用以标识表达式结点的类型
    int pos;                            //语法单位所在位置行号
    int offset;                          //偏移量
    int width;                          //占数据字节数
    int num;    //计数器，可以用来统计形参个数
    int hasReturn;
    int inWhile;
    };
```

在 ast.c 中，首先定义 mknode 函数，该函数是用于创建抽象语法树节点的，num 代表有几个参数，kind 代表节点类型，pos 代表行数

定义了语法分析树节点结合体，还需设计并实现语法分析树的生成函数和遍历函数：

```
void display(struct ASTNode *T, int indent)
{//对抽象语法树的先根遍历
```

完成了语法分析树相关结构体和函数的定义和实现后，需要将其与 Bison 中的生成式代码相结合。为此，需要完成两项工作：一是将所有终结符和非终结符的属性值类型声明为语法树节点的指针类型：

```
//  %type 定义非终结符的语义值类型，都归为ptr
%type  <ptr> program ExtDefList ExtDef  Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmLis

//% token 定义终结符的语义值类型
//终结符属性值的类型说明
%token <type_int> INT               //指定INT对应联合中的成员type_int
%token <type_id> ID  RELOP TYPE     //指定ID,TYPE对应联合中的成员type_id,即由词法分析得到的字符串
%token <type_float> FLOAT           //指定FLOAT对应联合中的成员type_float，即由词法分析得到的浮点类型
%token <type_char> CHAR             //指定CHAR对应联合中的成员type_char，即由词法分析得到的字符

%token VOID DPLUS DMINUS LP RP LC RC LB RB SEMI COMMA PLUSASSIGNOP MINUSASSIGNOP  /*用bison对该文件编译时，带参
%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR SWITCH CASE COLON DEFAULT BREAK CONTINU
/*以下为接在上述token后依次编码的枚举常量，作为AST结点类型标记*/
%token EXT_DEF_LIST EXT_VAR_DEF FUNC_DEF FUNC_DEC EXT_DEC_LIST PARAM_LIST PARAM_DEC VAR_DEF DEC_LIST DEF_LIST
%token FUNC_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO JLT JLE JGT JGE EQ NEQ ARRAY_CALL
```

二是为每条生成式添加语义动作，用来根据生成式构造语法树：

根节点显示如下：

```
program: ExtDefList     {prog=mknode(0,PROGRAM,0);display(prog,0);display($1,0);}
```

二维数组定义如下：

```
VarDec:  ID          {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}   //ID结点，标识符符号串存放结点的type_id
       | ID LB Exp RB   {$$=mknode(2,ARRAY,yylineno,$1,$3);strcpy($$->type_id,$1);}
       |ID LB Exp RB LB Exp RB   {$$=mknode(3,ARRAY2,yylineno,$1,$3,$6);strcpy($$->type_id,$1);}
```

3.3 符号表管理

符号表创建的思路：

本实验采用顺序表符号表。此时的符号表 symbolTable 是一个顺序栈，栈顶指针 index 初始值为 0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加 1。

```
//符号表
struct symboltable{
    struct symbol symbols[MAXLENGTH];
    int index;
    } symbolTable;
```

将 ast 的头结点传入，然后 AST 的遍历采用的是先根遍历，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。由此得到每个节点的继承属性和综合属性。以及顺便完成了地址偏移量的计算。

```
void semantic_Analysis(struct ASTNode *T)
{//对抽象语法树的先根遍历,按display的控制结构修改完成符号表管理和语义检查和TAC生成（语句部分）
```

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量 LEV 来管理，LEV 的初始值为 0。

```
struct symbol_scope_begin {
    //当前作用域的符号在符号表的起始位置序号,
    int TX[30];
    int top;
    } symbol_scope_TX;
```

每当遇到一个复合节点的开头，则将 LEV 加 1，代表着进入了一个新作用域。

每次要登记一个新的符号到符号表中时，首先在 symbolTable 中，从栈顶向栈底方向查层号为 LEV 的符号，是否有和当前待登记的符号重名，是则报重复定义错误，否则使用 LEV 作为层号将新的符号登记到符号表中。

```
int fillSymbolTable(char *name, char *alias, int level, int type, cha
    //首先根据name查符号表, 不能重复定义 重复定义返回-1
    int i;
    /*符号查重, 考虑外部变量声明前有函数定义,
    其形参名还在符号表中, 这时的外部变量与前函数的形参重名是允许的*/
```

当准备回到父节点的时候，这时复合语句语义分析完成，需要从符号表中删除该复合语句的变量，LEV 减一。

符号查找工作，当遇到变量的访问的时候，从栈顶向栈底方向查询是否有相同的符号定义，如果遍历完也没有查询到，则是没有。

数据保存结构：

在 def.h 中定义有 symbol，该结构体用于保存一个项的符号表属性，然后 symbolTable 是由 symbol 结构体构成的结构体数组以及包含了索引的结构体，符号表 symbolTable 是一个顺序栈，栈顶指针 index 初始值为 0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加 1。

3.4 语义检查

第一步是构建符号表，见上。

第二步是语义分析，这部分完成的是静态语义分析，主要包括：

（1）控制流检查。控制流语句必须使得程序跳转到合法的地方。例如一个跳转语句会使控制转移到一个由标号指明的后续语句。如果标号没有对应到语句，那么就出现一个语义错误。再者，break、continue 语句必须出现在循环语句当中。

（2）唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

（3）名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

（4）类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

3.5 报错功能

对 AST 树采用先根遍历，判断各个 AST 树结点的类型并通过各种类型对应的函数检查是否满足语义要求，若不满足则调用 semantic_error 函数进行报错打印。

```
void semantic_error(int line, char *msg1, char *msg2) {
    //这里可以只收集错误信息，最后一次显示
    printf("在%d行,%s %s\n", line, msg1, msg2);
```

3.6 中间代码生成

中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式。一方面，中间代码将编译器自然地分为前段和后端两个部分；另一方面，采用中间代码有利于进行机器无关的优化。

中间代码生成是通过遍历语法树并维护作用域来实现的。生成中间代码是采用了静态语义分析的部分内容，采用自顶向下的方式递归地进行语法分析树的遍历。

为了避免可能存在的重名带来的问题，中间代码里的变量没有使用原变量名，而是使用的一个唯一的别名。对于跳转语句的目标语句，也为其定义了一个唯一的标号。

实验 3 在实验 2 的基础上，在对语法树进行遍历以完成语义分析的同时产生了中间代码，并对抽象语法树的结点类型进行了判断，对相应结点按一定的规则建立了中间代码语句，并加入到中间代码的双向链表中。

3.7 代码优化

基本块的划分：

一个入口语句和一个出口语句构成了一个基本块。

入口语句包括：

1.程序的第一个语句

2.转移语句的目标语句

3.条件转移语句后第一个语句

出口语句包括

1.转移语句

2.停止语句

3.入口语句之前的语句

根据以上信息，可以在得到中间代码的同时，对基本块进行划分。

具体到现实代码：

在遇到 FUNCTION 和 LABEL 时，代表进入块。

在遇到 GOTO、RETURN 时，代表离开块

在遇到 JLE、JLT、JGE、JGT、EQ、NEQ、CALL 时，代表先离开块，后进入块。

3.8 汇编代码生成

采用朴素寄存器分配法，把所有的变量或临时变量都放在内存里，每次翻译一条中间代码，都需要将用到的变量加载到寄存器中，得到计算结果后又需要将结果写回内存。

由于选择朴素的寄存器分配，只会用到几个寄存器，为简单起见，每句目标代码的目的操作数存于 t3，0pn1 存于 t1，0pn2 存于 t2，函数执行返回值存于 v0。每次执行前从内存读取数据存入 t1，t2，执行完后，将 t3 的值写回内存。

表 朴素寄存器分配的翻译

中间代码	MIPS32 指令
x :=#k	li $t3,k sw $t3, x 的偏移量($sp)

x := y	lw $t1, y的偏移量($sp)
	move $t3,$t1
	sw $t3, x的偏移量($sp)

x := y + z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	add $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y - z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	sub $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y * z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul $t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y / z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul $t3,$t1,$t2
	div $t1,$t2
	mflo $t3
	sw $t3, x 的偏移量($sp)

RETURN x	move $v0, x 的偏移量($sp) jr $ra

IF x==y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	beq $t1,$t2,z

IF x!=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bne $t1,$t2,z
IF x>y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bgt $t1,$t2,z
IF x>=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bge $t1,$t2,z
IF x<y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) ble $t1,$t2,z
IF x<=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) blt $t1,$t2,z
X:=CALL f	

在 prnIR 函数中输出中间代码：

```
//输出中间代码
void prnIR(struct codenode *head)
```

将生成的目标代码放在 output.asm 文件里，采用 MARS 来运行 MIPS 代码，需要增加入口函数 main0，在入口函数中调用 main 函数，对应的代码段为：

```
fprintf(fp, ".globl main0\n");
fprintf(fp, ".data\n");
fprintf(fp, "_Prompt: .asciiz \"Please enter an integer:  \"\n");
fprintf(fp, "_ret: .asciiz \"\\n\"\n");
fprintf(fp, ".globl main\n");
fprintf(fp, ".text\n");
fprintf(fp, "read:\n");
```

接下来对每一句中间代码都用寄存器分配算法翻译。

```
switch (h->op) {
case ASSIGNOP:
    if (h->opn1.kind == INT)
        fprintf(fp, "  li $t3, %d\n", h->opn1.const_int);
    else {
        fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);
        fprintf(fp, "  move $t3, $t1\n");
    }
    fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);
    break;
case PLUS:
case MINUS:
case STAR:
case DIV:
    fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);
    fprintf(fp, "  lw $t2, %d($sp)\n", h->opn2.offset);
    if (h->op == PLUS)       fprintf(fp, "  add $t3,$t1,$t2\n");
    else if (h->op == MINUS) fprintf(fp, "  sub $t3,$t1,$t2\n");
    else if (h->op == STAR)  fprintf(fp, "  mul $t3,$t1,$t2\n");
    else  {
        fprintf(fp, "  div $t1, $t2\n");
        fprintf(fp, "  mflo $t3\n");
    }
    fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);
    break;
case FUNCTION:
    fprintf(fp, "\n%s:\n", h->result.id);
    if (!strcmp(h->result.id, "main"))
        fprintf(fp, "  addi $sp, $sp, -%d\n", symbolTable.symbols[h->result.
    break;
case PARAM:
    break;
case LABEL: fprintf(fp, "%s:\n", h->result.id);
    break;
case GOTO:  fprintf(fp, "  j %s\n", h->result.id);
    break;
```

4 系统测试与评价

5.1 测试用例

实验一：

```
int m,n;
char w,i;
float a,b;
int array[5];        //一维数组
/*
    这是注释
*/
int arrays[2][1];
int main(){
char o;
int k,l;
m=print();
n=5;
k=1;
l=n*2+k;
while(k<=n)
    {
    l=fibo(k);
    write(l);
    k=k+1;
    }
for(i=1;i<5;i++)
{
    char i;
    k--;
    k++;
    ++k;
    --k;
    continue;
    break;
}
return 1;
}
int fibo(int f){
if(f==1||f==2)return 1;
return fibo(f-1)+fibo(f-2);
}
```

实验二：

```
int a,b,c;
char e,g;
float m,n;
int fibo(int a){
if(a==1||a==2)return m;//（1）函数返回值类型与函数定义的返回值类型不匹配；
//(2)没有返回语句测试
}
int w[7];  //注释1
/*
    注释2
*/
int t[1][2];

int fibo(int a){
    //（3）函数重复定义
}
int main(){
int m,n,i,p;
int m;//（4）变量重复定义
i=3+2*5;
m=read();
i=1;
p=1;
n=i+p*5;
v=3;//（5)使用未定义的变量
n();//  (6)对非函数名采用函数调用的形式fibo();//(6)函数调用时参数个数不匹配
fibo(1,3);//(7)函数调用时参数个数不匹配
fibo(r);//(8)形参和实参类型不匹配
fibo;//(9)对函数名采用非函数形式访问
p[5];//(10)对非数组变量采用下标变量的形式访问
w[e];//（11)数组变量的下标不是整型表达式
//m.1;//(12)对非结构变量采用成员选择运算符
i='a';//（13）类型不匹配。

while(i<=m)
    {
    n=fibo(i);
    write(n);
    i=i+1;
    continue;
    break;
    }
for(i=1;i<9;i++)
{
    char i;
    p--;
    p++;
    ++p;
}
5=5;//（14）赋值号左边不是左值表达式
5++;//（15）对非左值表达式进行自增、自减运算；
5--;
break; //（16）break在循环外测试
continue;//（17）continue在循环外测试
test();//(18)调用未定义的函数
return 1;
}
```

实验三：

```
int a,b,c;
float m,n;
char x,y;
int w[7];  //asdasdsadsa
/*
    asdasdfaf删除注释测试
*/
int h[1][2];
int fibo(int a){
if(a==1||a==2)return 0;
return 1;
}
int main(){
int m,n,i,k;
i=3+2*5;
m=read();
i=1;
k=1;
n=i+k*5;
while(i<=m)
    {
    n=fibo(i);
    write(n);
    i=i+1;
    continue;
    break;
    }
for(i=1;i<9;i++)
{
    char i;
    k--;
    k++;
    ++k;
}
return 1;
}
```

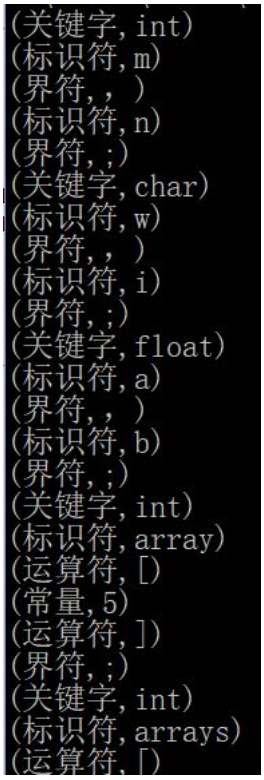
实验四：

```
int a,b,c;
char e,g;
float m,n;
int fibo(int a){
if(a==1||a==2)return 0;
return 1;
}
int t[3];  //注释1
/*
    注释2
*/
int u[1][2];

int main(){
int m,n,i,p;
i=1;
p=fibo(i);
i=7-8*3;
m=read();
return 1;
}
```

5.2 正确性测试

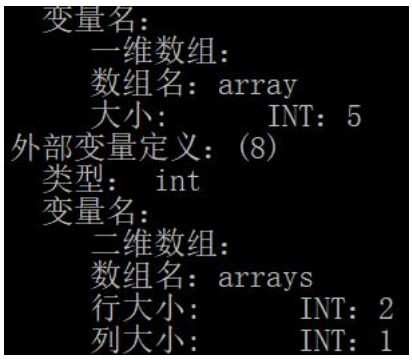
实验一：

二元式展示：



抽象语法树展示：

一维数组和二维数组：



实验二：

符号表生成和报错显示如下：

```
在14行,fibo 函数重复定义
变量名   别名   层号    类型    标记  偏移量
  read            0     int    F      4
 write            0     int    F      4
     x            1     int    P     12
     a     v1     0     int    V      0
     b     v2     0     int    V      4
     c     v3     0     int    V      8
     e     v4     0   float    V     12
     g     v5     0   float    V     20
     m     v6     0   float    V     28
     n     v7     0   float    V     36
  fibo     v8     0     int    F     20
     a     v9     1     int    P     12
     w    v10     0     int    A     16
在16行, 函数没有返回语句
在19行,m 变量重复定义
在25行,v 变量未定义
在25行, 赋值类型不匹配
在26行,n 不是一个函数
在27行, 函数调用参数太多!
在28行,r 变量未定义
在28行, 参数类型不匹配
在29行,fibo 是函数名, 类型不匹配
在30行,  对非数组变量采用下标变量的形式访问
在31行,  数组变量的下标不是整型表达式
在33行, 赋值类型不匹配
变量名   别名   层号    类型    标记  偏移量
  read            0     int    F     12
```

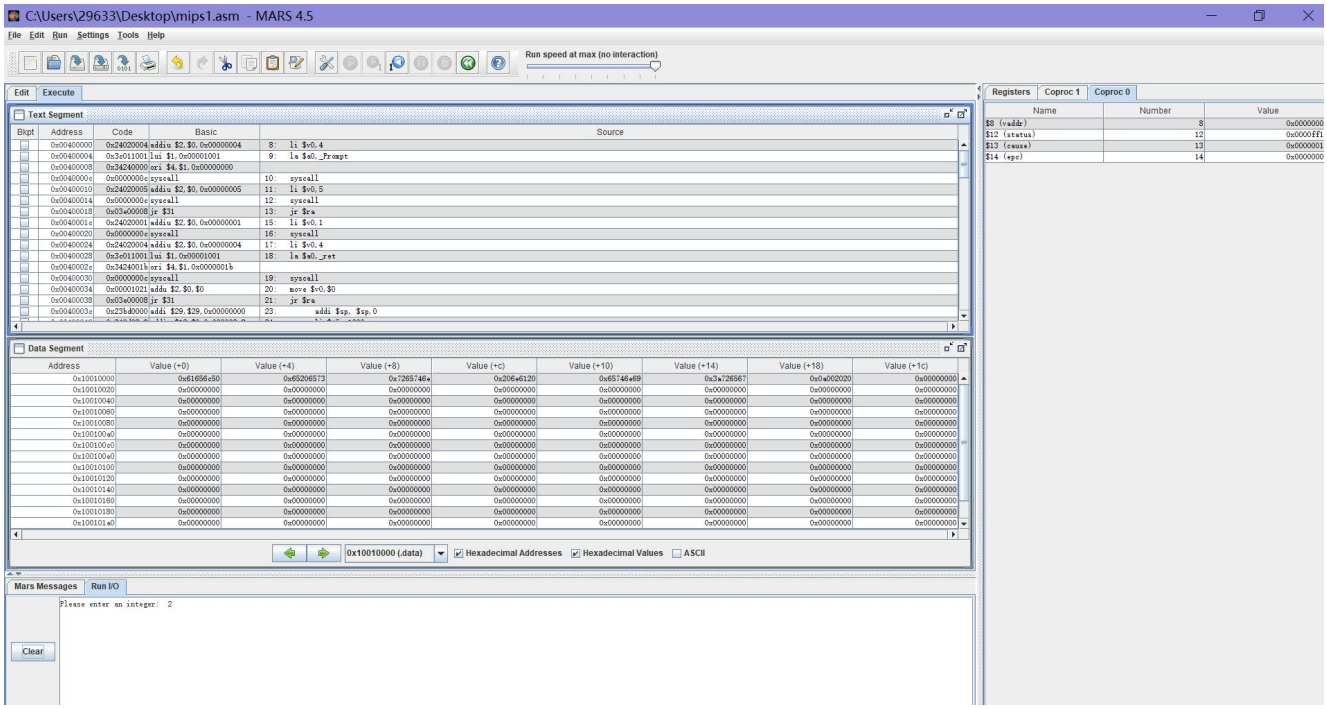
```
在49行, 赋值语句需要左值
在50行, 自增操作对象不是变量
在51行, 自减操作对象不是变量
在52行, break 语句不在循环语句中
在53行, continue 语句不在循环语句中
在54行,test 函数未定义
变量名   别名   层号    类型    标记  偏移
  read            0     int    F     12
 write            0     int    F      4
     x            1     int    P     12
     a     v1     0     int    V      0
     b     v2     0     int    V      4
     c     v3     0     int    V      8
     e     v4     0   float    V     12
     g     v5     0   float    V     20
     m     v6     0   float    V     28
     n     v7     0   float    V     36
  fibo     v8     0     int    F     20
     a     v9     1     int    P     12
     w    v10     0     int    A     16
  main    v12     0     int    F      0
     m    v13     1     int    V     12
     n    v14     1     int    V     16
     i    v15     1     int    V     20
     p    v16     1     int    V     24
         temp3     1     int    T     32
         temp4     1     int    T     36
         temp5     1     int    T     40
         temp6     1     int    T     44
```

实验三：

中间代码及基本块的划分（节选）：

```
进入基本块,块号为: (1)

FUNCTION fibo :
  PARAM v10
  temp1 := #1
  IF v10 == temp1 GOTO label3
离开基本块,块号为: (1)

进入基本块,块号为: (2)
  GOTO label4
离开基本块,块号为: (2)

进入基本块,块号为: (3)
LABEL label4 :
  temp2 := #2
  IF v10 == temp2 GOTO label3
离开基本块,块号为: (3)

进入基本块,块号为: (4)
  GOTO label2
离开基本块,块号为: (4)

进入基本块,块号为: (5)
LABEL label3 :
  temp3 := #0
  RETURN temp3
离开基本块,块号为: (5)
```

实验四：



5.3 报错功能测试

实验一：

如图所示，将一个标识符的开头改成数字开头，另一行代码结尾的分号去掉。

```
int main(){
char 1o;
int k,l;
m=print();
n=5;
k=1
l=n*2+k;
```

结果如下：

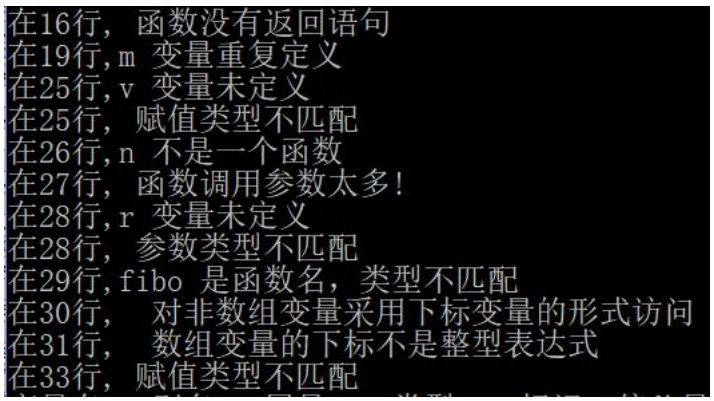
```
Grammar Error at Line 10 Column 6: syntax error, unexpected INT, expecting ID.
(标识符,o)
(界符,;)
(关键字,int)
(标识符,k)
(界符,,)
(标识符,l)
(界符,;)
(标识符,m)
(运算符,=)
(标识符,print)
(运算符,()
(运算符,))
(界符,;)
(标识符,n)
(运算符,=)
(常量,5)
(界符,;)
(标识符,k)
(运算符,=)
(常量,1)
(标识符,l)
Grammar Error at Line 15 Column 1: syntax error, unexpected ID.
```

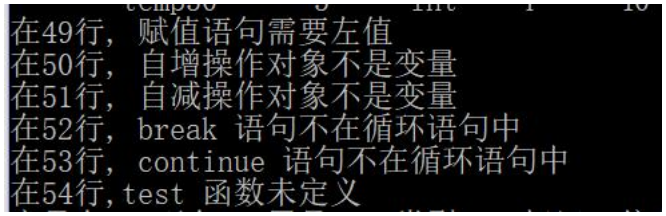
出现连续报错，则报错功能测试成功。

实验二：

如图所示：

```
在5行,,  返回值类型错误
变量名   别名   层号    类型    标记  偏移
  read            0     int    F     4
 write            0     int    F     4
     x            1     int    P    12
     a     v1     0     int    V     0
     b     v2     0     int    V     4
     c     v3     0     int    V     8
     e     v4     0   float    V    12
     g     v5     0   float    V    20
     m     v6     0   float    V    28
     n     v7     0   float    V    36
  fibo     v8     0     int    F     0
     a     v9     1     int    P    12
         temp1     1     int    T    16
         temp2     1     int    T    16
在7行, 函数没有返回语句
在14行,fibo 函数重复定义
```





5.4 系统的优点

可以完整的将一段 miniC 语言进行词法分析，语法分析，语义分析，中间代码生成，代码优化，目标代码生成，体现了完整的编译过程。并且有自己的增加项如识别二维数组，continue 和 break，自增自减等。

5.5 系统的缺点

由于时间的关系，实验三的 DAG 构造和代码优化算法没有来得及思考，希望在空闲时间可以好好思考下这几个问题。

5 实验小结或体会

通过编译原理实验，我了解了编译器的实现原理与过程，实现了词法分析，语法分析，语义分析，中间代码生成和目标代码生成的全过程。

这次实验和课设让我感触最深的是，有许多课上学到的内容，仅仅通过干瘪的背记，是无法理解到其意义与作用的。这次实验，提供了一次良好的机会，以练促学。虽然程序的主体上借用了老师给定的框架，省略了一些设计步骤，但不管怎样，在最后看到一个编译器能从自己手中运行，读入的程序顺利地生成了目标代码，并能正确运行的时候，还是蛮有成就感的。只有自己亲身实现从源代码到目标代码，再到实现，才能真正有效地帮助自己理解整个编译过程。

在本次实验中，我学会了如何编写词法分析器的正规式和使用自动化生成工具 Flex 得到词法分析程序；学会了如何使用自动化生成工具 Bison 得到语法分析程序，以及如何在 Bison 环境中使用属性文法、语义动作增强文法的表达能力； 学会了如何利用源程序的语法分析树，并对其进行语义分析；学会了如何对源程序进行分析而能得到其符号表，以及如何维护和检测源程序的作用域；学会了中间代码的常用形式三地址码，以及如何将源代码转化为中间代码；学会了如何将中间代码翻译成目标代码。

在今后学习中，我将继续保持积极的学习态度，认真思考遇到的问题，及时反思总结，尽快将学到的理论知识应用到实践中来，在实践中发现真知。

参考文献

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016
[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005
[3] 王元珍等. 80X86 汇编语言程序设计. 武汉:华中科技大学出版社,2005
[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005
[5] 曹计昌等. C 语言程序设计. 北京：科学出版社，2008

附件：源代码