

华中科技大学

课程实验报告

课程名称: 编译原理实验

专业班级: CS1802

学 号: U201814531

姓 名: 李响

指导教师: 杨茂林

报告日期: 2021 年 7 月 6 日

计算机科学与技术学院

目 录

0 概述.....	1
1 实验一 Defan 语言定义.....	2
1.1 实验目的与内容.....	2
1.2 Defan 语言文法设计.....	2
2 实验二 词法分析器设计与实现.....	5
2.1 实验目的与内容.....	5
2.2 Defan 语言单词文法描述.....	5
2.3 词法分析器构建.....	6
2.3 词法分析器调试与测试.....	10
3 实验三 自定义语言语法分析器设计与实现.....	11
3.1 实验目的与内容.....	11
3.2 语法分析器构建.....	11
3.3 抽象语法树建立.....	16
3.4 语法分析器调试与测试.....	17
4 实验四 符号表管理和属性计算.....	21
4.1 实验目的与内容.....	21
4.2 符号表设计与管理.....	21
4.3 语法树属性计算.....	24
3.4 符号表测试.....	24
5 实验五 静态语义分析.....	26
5.1 实验目的与内容.....	26
5.2 语义分析处理.....	27
5.3 语义分析测试.....	28
6 实验六 中间代码生成.....	31
6.1 实验目的与内容.....	31
6.2 中间语言的定义.....	31
6.3 翻译模式设计（ 6.3.4 为重点 ）.....	32

6.4 中间代码生成测试.....	38
7 实验七 代码优化.....	40
7.1 实验目的与内容.....	40
7.2 DAG 图数据结构定义.....	40
7.3 DAG 优化与全局优化.....	41
7.4 代码优化测试.....	43
8 实验八 目标代码生成.....	45
8.1 实验目的与内容.....	45
8.2 目标语言的指令定义.....	45
8.3 寄存器分配与目标代码生成.....	46
8.4 目标代码的生成测试.....	48
9.1 实验完成情况.....	50
9.2 实验感想.....	51
9.3 展望.....	51

0 概述

本次实验的主要目标是自定义一个高级程序设计语言的词法和语法规则，并构造该语言的编译器，将自定义的语言翻译成汇编语言。

通过分析与权衡后，本次将基于 Decaf 语言定义一个纯面向对象的高级语言并命名为“Defan”，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，提高学生系统软件开发技术。

Defan 语言的主要语法以及本次构造的编译器所支持的功能如下：

1. 数据类型：char 类型、int 类型、float 类型；
2. 数组与类：Defan 语言为纯面向对象的语言，支持**多个类定义以及多维数组**的使用（**支持类数组**），为简化编译器的实现，类不支持继承以及反射；
3. 所支持运算：基本逻辑运算、基本算数运算、自增自减运算、复合赋值运算以及比较运算；
4. 控制语句：if 语句、if-else 语句、while 语句、for 语句、break 语句以及 continue 语句，其中 if 语句、if-else 语句、while 语句、for 语句都支持内部定义变量，**且与 C 语言类似**，当使用简单语句时，可以不使用花括号进行包括；
5. 输入输出语句：使用 Print 语句进行输出，Scan 语句进行输入；
6. 函数调用语句：类似 C 语言，可以使用“**类对象.函数名**”的方式进行函数调用，**也支持 this 指针的使用**；
7. 注释：只支持行注释，不支持块注释；

1 实验一 Defan 语言定义

1.1 实验目的与内容

自定义一个高级程序设计语言的词法和语法规则，可以基于某种熟悉的高级程序设计语言，设计一个自定义的高级语言子语言，完成其词法和语法规则；也可以选用 Decaf 语言作为源语言，需要完成的要求如下：

1. 为语言命名；
2. 标识符、常数、字符串等单词的文法；
3. 符号集、保留字集、运算符、界符；
4. 说明语句文法；
5. 赋值语句文法（简单赋值）；
6. 表达式求值文法（简单算术运算，包括自加自减运算）；
7. 分支语句文法；
8. 循环语句文法；
9. 输入语句、输出语句文法
10. 过程或函数调用语句文法；

1.2 Defan 语言文法设计

本次实验所设计的语言 Defan 基于 Decaf 语言以及 C 语言设计。

与 Decaf 语言相同，本次实验定义的 Defan 语言为纯面向对象的语言，函数的定义实现以及变量的声明必须在类中进行；语言框架与 C 语言类似，如使用分号分隔语句，for 等控制语句的使用也与 C 类似等等。

本次实验定义的 Defan 语言的具体文法如下：

1. 标识符、常数等单词文法

(1) 标识符： $ID \rightarrow [A-Z a-z _][A-Z a-z 0-9 _]^*$

(2) 常数： $Constant \rightarrow INT \mid FLOAT \mid CHAR$

$INT \rightarrow ([1-9][0-9]^*)(0[xX][0-9a-fA-F]^+)(0[0-7]^*)$

$FLOAT \rightarrow [0-9]^*.[0-9]^+([eE][-+]?([1-9][0-9]^*[0]))?$

$CHAR \rightarrow '([^\backslash]|\backslash[\"'?\backslash abfnrtv]|\backslash[0-7]\{1,3\}|\backslash[Xx][0-9A-Fa-f]^+)'$

2. 符号集、保留字集、运算符、界符

(1) 符号集: ASCII 码集

(2) 保留字集: int float char void class this return break continue

if else for while null true false Print Scan

(3) 运算符: + - * / % < <= > >= == != && || ! ++ -- += -= *= /= %=

(4) 界符: ; , . [] () { }

3. 说明语句文法

(1) 类声明: $\text{Program} \rightarrow \text{ClassDefList}$

$\text{ClassDefList} \rightarrow \text{ClassDef ClassDefList} \mid \varepsilon$

(2) 类成员声明 (数据成员函数以及函数成员):

$\text{ClassDef} \rightarrow \text{class ID} \{ \text{FieldList} \}$

$\text{FieldList} \rightarrow \text{Field FieldList} \mid \varepsilon$

$\text{Field} \rightarrow \text{VariableDef} \mid \text{FunctionDef}$

$\text{VariableDef} \rightarrow \text{Type IDList};$

$\text{IDList} \rightarrow \text{ID} \mid \text{ID}, \text{IDList}$

$\text{Type} \rightarrow \text{int} \mid \text{char} \mid \text{void} \mid \text{class ID} \mid \text{Type}[\text{INT}]$

$\text{FunctionDef} \rightarrow \text{Type ID} (\text{Formals}) \text{ StmtBlock}$

$\text{Formals} \rightarrow \text{VariableList} \mid \varepsilon$

$\text{VariableList} \rightarrow \text{Variable} \mid \text{Variable}, \text{VariableList}$

$\text{Variable} \rightarrow \text{Type ID}$

(3) 程序块说明:

$\text{StmtBlock} \rightarrow \{ \text{StmtList} \}$

$\text{StmtList} \rightarrow \text{Stmt StmtList} \mid \varepsilon$

$\text{Stmt} \rightarrow \text{VariableDef} \mid \text{SimpleStmt}; \mid \text{IfStmt} \mid \text{WhileStmt}$

$\mid \text{ForStmt} \mid \text{BreakStmt}; \mid \text{ReturnStmt}; \mid \text{PrintStmt};$

$\mid \text{ScanStmt}; \mid \text{ContinueStmt}; \mid \text{StmtBlock}$

4. 赋值语句文法 (简单赋值)

$\text{SimpleStmt} \rightarrow \text{Expr} \mid \varepsilon$

$\text{Constant} \rightarrow \text{INT} \mid \text{FLOAT} \mid \text{CHAR} \mid \text{null} \mid \text{true} \mid \text{false}$

5. 表达式求值文法:

$\text{Expr} \rightarrow \text{Constant} \mid \text{Expr} = \text{Expr} \mid \text{this} \mid (\text{Expr})$
 $\mid \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \mid \text{Expr} * \text{Expr} \mid \text{Expr} / \text{Expr} \mid \text{Expr} \% \text{Expr}$
 $\mid \text{Expr} < \text{Expr} \mid \text{Expr} \leq \text{Expr} \mid \text{Expr} > \text{Expr} \mid \text{Expr} \geq \text{Expr}$
 $\mid \text{Expr} == \text{Expr} \mid \text{Expr} != \text{Expr} \mid \text{Expr} \&\& \text{Expr} \mid \text{Expr} \parallel \text{Expr} \mid ! \text{Expr}$
 $\mid \text{Expr} ++ \mid \text{Expr} -- \mid ++\text{Expr} \mid --\text{Expr} \mid - \text{Expr} \mid \text{ID} \mid \text{Expr}[\text{Expr}] \mid \text{Expr}.\text{ID}$

$\text{Actuals} \rightarrow \text{ExprList} \mid \varepsilon$

$\text{ExprList} \rightarrow \text{Expr}, \text{ExprList} \mid \text{Expr}$

6. 分支语句文法

$\text{IfStmt} \rightarrow \text{if} (\text{Expr}) \text{Stmt} \mid \text{if} (\text{Expr}) \text{Stmt} \text{ else } \text{Stmt}$

7. 循环语句文法;

$\text{WhileStmt} \rightarrow \text{while} (\text{Expr}) \text{Stmt}$

$\text{ForStmt} \rightarrow \text{for} (\text{SimpleStmt} ; \text{SimpleStmt} ; \text{SimpleStmt}) \text{Stmt}$

$\text{BreakStmt} \rightarrow \text{break}$

$\text{ContinueStmt} \rightarrow \text{continue}$

8. 输入语句、输出语句文法

$\text{PrintStmt} \rightarrow \text{Print} (\text{Expr})$

$\text{ScanStmt} \rightarrow \text{Scan} (\text{Expr})$

9. 过程或函数调用语句文法

$\text{Expr} \rightarrow \text{ID} (\text{Actuals}) \mid \text{Expr}.\text{ID} (\text{Actuals})$

2 实验二 词法分析器设计与实现

2.1 实验目的与内容

使用词法分析程序的自动生成工具 LEX 或 FLEX，将单词文法转换为正规式，设计并实现能够输出单词序列（二元式）的词法分析器，具体实验要求如下：

1. 理解单词的分类和形式化描述；
2. 掌握自动生成工具 flex 的使用技术；
3. 定义保留字和操作符、界符的内部码；
4. 实现一个完整的词法分析器；
5. 显示词法分析结果（二元组）列表；

2.2 Defan 语言单词文法描述

本次实验所设计的语言 Defan 基于 Decaf 语言以及 C 语言设计，单词文法与 C 语言的单词文法类似，共可以分为 6 类：标识符、关键字、运算符、界符、常量及注释（只支持行注释，不支持块注释）。

其中，常量与标识符使用正则表达式进行匹配，标识符的正则式命名为 ID，整型常量正则式命名为 INT，浮点常量正则式命名为 FLOAT，字符常量正则式命名为 CHAR，具体正则式参见 1.2 Defan 语言文法设计。

具体单词文法定义如表 2-1 所示，单词种类码用于标识单词类型，可以在后续实验用于构建语法树：

表 2-1 单词文法定义

符号说明	种类码	符号说明	种类码
{ID}	ID	"void"	TYPE
{INT}	INT	"int"	TYPE
{FLOAT}	FLOAT	"float"	TYPE
{CHAR}	CHAR	"char"	TYPE
"if"	IF	"class"	CLASS
"else"	ELSE	"null"	NULL_P
"for"	FOR	"true"	TRUE
"while"	WHILE	"false"	FALSE
"break"	BREAK	"Print"	PRINT

"continue"	CONTINUE	"Scan"	SCAN
"return"	RETURN	">"	RELOP
"="	ASSIGNOP	"<"	RELOP
"+"	PLUS	">="	RELOP
"-"	MINUS	"<="	RELOP
"*"	STAR	"=="	RELOP
"/"	DIV	"!="	RELOP
"%"	MOD	"&&"	AND
"++"	AUTOADD	" "	OR
"--"	AUTOMINUS	"!"	NOT
"+="	PLUSASSIGNOP	"."	DOT
"_="	MINUSASSIGNOP	“,”	SEMI
"*="	STARASSIGNOP	“,”	COMMA
"/=	DIVASSIGNOP	“(”	LP
"%="	MODASSIGNOP	“)”	RP
[\n]	换行符无种类码	"["	LB
[\r\t]	制表符无种类码	"]"	RB
\\[^\n]*	注释无种类码	“{”	LC
		“}”	RC

2.3 词法分析器构建

构建词法分析器，既可以按教材中介绍的方法，自行编写程序来完成，也可以使用工具来实现，为简化实验，本次实验使用词法分析程序的自动生成工具Flex构建词法分析器。

首先，根据语言的词法规则，以正则表达式的形式给出词法规则；

第二，按照Flex要求的格式，编写词法文件（.l文件）；

第三，使用Flex编译词法文件（.l文件），并使用编译生成的词法分析源程序Lex.yy.c进行词法分析；最后，以二元式的形式输出词法分析结果。

具体词法分析器构建过程如下：

2.3.1 定义部分

定义部分主要包含两块内容，第一块内容包含在%{到}%}区间中，主要包含C语言的一些宏定义，如文件包含、宏名定义，以及变量和类型的定义和声明，该部分会直接被复制到词法分析器源程序lex.yy.c中。

考虑到本实验构建的词法分析器需要与实验3构建的语法分析器进行联合编译，因此需要引用一些文件，如图2-1所示，需要引入头文件“parser.tab.h”，此文件中包含实验3中定义的单词的种类码；引入头文件“def.h”和“string.h”，用于字符串的处理；定义联合体YYLVAL用于传递信息，如将标识符的名称以及常量值传递到语法分析器中，便于进行语法分析以及语法树的建立。

图 2-1 定义部分声明代码截图

```
ID [A-Za-z_][A-Za-z0-9_]*
INT ((1-9)[0-9]*)|([0xX][0-9a-fA-F]+)|([0-7]*)
FLOAT [0-9]*.[0-9]+|([eE][-+]?([1-9][0-9]*)|([0])?)
CHAR \'([^\\"\\|\\\'?\"\\abfnrt\\|\\[0-7]{1,3}|\\[Xx][0-9A-fA-f]+)|\'
ERRORID [0-9][A-Za-z0-9_]*
```

图 2-2 定义部分正则式宏定义代码截图

该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，当词法分析器识别出单词后，将该单词对应字符串保存在`vytext`中，长度为`vy leng`。

1. 常量匹配, 如图2-3所示, 使用正则表达式对常量进行匹配, 并输出对应的二元组, 将常量信息转化成对应类型的量使用yylval传递到语法分析器中。

```
{INT} {printf("(s, INT)\n", yytext);yylval.type_int=atoi(yytext); return INT;}
{FLOAT} {printf("(s, FLOAT)\n", yytext);yylval.type_float=atof(yytext); return FLOAT;}
{CHAR} {printf("(s, CHAR)\n",yytext);strncpy(yylval.type_string,yytext+1,strlen(yytext)-2);return CHAR;}
"null" {printf("(s, NULL)\n", yytext);return NULL_P;}
"true" {printf("(s, TRUE)\n", yytext);return TRUE;}
>false" {printf("(s, FALSE)\n", yytext);return FALSE;}
```

图 2-3 常量关键字匹配规则截图

2. 类型匹配, 如图2-4所示, 对类型说明关键字void、int、float、char以及类定义关键字class以及this关键字进行匹配, 并将void、int、float、char以字符串的形式传递到语法分析器中, 便于语法分析器辨别数据的类型。

```
"void" {printf("(s, TYPEVOID)\n", yytext);strcpy(yylval.type_id, yytext);return TYPE;}
"int" {printf("(s, TYPEINT)\n", yytext);strcpy(yylval.type_id, yytext);return TYPE;}
"float" {printf("(s, TYPEFLOAT)\n", yytext);strcpy(yylval.type_id, yytext);return TYPE;}
"char" {printf("(s, TYPESTRING)\n",yytext);strcpy(yylval.type_id, yytext);return TYPE;}
"class" {printf("(s, CLASS)\n",yytext); return CLASS;}
>this" {printf("(s, THIS)\n",yytext);return THIS;}
```

图 2-4 类型关键字匹配规则截图

3. 关键字匹配, 如图2-5所示, 对循环控制、分支控制等几类关键字进行匹配, 返回对应的种类码并输出二元式, Print关键字与Scan关键字为内置的输出与写入关键字, 用于输出和读入。

```
"if" {printf("(s, IF)\n", yytext);return IF;}
"else" {printf("(s, ELSE)\n", yytext);return ELSE;}
"for" {printf("(s, FOR)\n", yytext);return FOR;}
"while" {printf("(s, WHILE)\n", yytext);return WHILE;}
"break" {printf("(s, BREAK)\n", yytext);return BREAK;}
"continue" {printf("(s, CONTINUE)\n", yytext);return CONTINUE;}
"return" {printf("(s, RETURN)\n", yytext);return RETURN;}
"Print" {printf("(s, PRINT)\n", yytext);return PRINT;}
"Scan" {printf("(s, SCAN)\n", yytext);return SCAN;}
```

图 2-5 其他关键字匹配规则截图

4. 标识符匹配, 如图2-6所示, 使用标识符正则式匹配标识符, 由于标识符由字母(A-Z, a-z)、数字(0-9)、下划线“_”组成, 并且首字符不能是数字, 但可以是字母或者下划线, 因此该规则必须放置在关键字的匹配之后, 以防止将关键字错误匹配成标识符。在匹配成功后, 输出对应的二元式, 并将标识符以字符串的形式传递到语法分析器中, 以便于进行语法分析。

```
{ID} {printf("(s, IDENTIFIER)\n", yytext);strcpy(yylval.type_id, yytext); return ID;}
```

图 2-6 标识符匹配规则截图

5. 运算符匹配，如图2-7所示，对各类运算符进行匹配，包括比较运算符、逻辑运算符、算数运算符、赋值运算等，返回对应的种类码，以便于进行语法树的构建，同时输出对应的二元式。

```
">"|"<"|">="|"<="|"=="|!=" {printf("(s, RELOP)\n", yytext);strcpy(yylval.type_id, yytext);return RELOP;}
"=" {printf("(s, ASSIGNOP)\n", yytext);return ASSIGNOP;}
"+" {printf("(s, PLUS)\n", yytext);return PLUS;}
"-" {printf("(s, MINUS)\n", yytext);return MINUS;}
"*" {printf("(s, STAR)\n", yytext);return STAR;}
"/" {printf("(s, DIV)\n", yytext);return DIV;}
%" {printf("(s, MOD)\n", yytext);return MOD;}

"&&" {printf("(s, AND)\n", yytext);return AND;}
"||" {printf("(s, OR)\n", yytext);return OR;}
"!" {printf("(s, NOT)\n", yytext);return NOT;}

"++" {printf("(s, AUTOPLUS)\n", yytext); return AUTOPLUS;}//自增
"--" {printf("(s, AUTOMINUS)\n", yytext); return AUTOMINUS;}//自减
"+=" {printf("(s, PLUSASSIGNOP)\n", yytext); return PLUSASSIGNOP;}
"-=" {printf("(s, MINUSASSIGNOP)\n", yytext); return MINUSASSIGNOP;}
"*=" {printf("(s, STARASSIGNOP)\n", yytext);return STARASSIGNOP;}
"/=" {printf("(s, DIVASSIGNOP)\n", yytext);return DIVASSIGNOP;}
"%=" {printf("(s, MODASSIGNOP)\n", yytext);return MODASSIGNOP;}
```

图 2-7 运算符匹配规则截图

6. 界符、分隔符以及注释匹配，如图2-8所示，对各类界符进行匹配，其中“.”号作用为访问作用域，其他界符包括小括号“（）”、中括号“[]”、大括号“{}”以及逗号“,”和分号“;”；换行以及回车等符号仅作为分隔使用，无意义；ERRORID用于匹配非法的标识符。

```
". " {printf("(s, DOT)\n", yytext);return DOT;}
";" {printf("(s, SEMI)\n", yytext);return SEMI;}
"," {printf("(s, COMMA)\n", yytext);return COMMA;}
"(" {printf("(s, LP)\n", yytext);return LP;}
")" {printf("(s, RP)\n", yytext);return RP;}
"{" {printf("(s, LC)\n", yytext);return LC;}
"}" {printf("(s, RC)\n", yytext);return RC;}
"[" {printf("(s, LB)\n", yytext);return LB;}
"]" {printf("(s, RB)\n", yytext);return RB;}
"\n" {printf("(s, EOL)\n");yycolumn=1;}
["\t"] {}

. {printf("Error type A :Mysterious character \"s\"\\n\\t at Line %d\\n",yytext,yylineno);}
\\/[^\n]* {printf("(s, LINECOMMENT)\n", yytext);}//匹配注释
{ERRORID} {printf("(s, 禁止数字为开头的标识符)\n",yytext);strcpy(yylval.type_id,yytext);}
```

图 2-8 界符与注释匹配规则截图

2.3 词法分析器调试与测试

为简化实验流程，本实验实际上采用了将实验 2 与实验 3 进行联合编译与测试的方式进行实验的调试和测试，本节仅展示词法分析相关的调试与测试过程，具体的调试和测试流程如下：

首先，下载并安装 flex、bison 以及 gcc 编译器，准备进行调试；然后，使用命令“flex Defan.l”以及“bison -d -v parser.y”编译词法文件（.l 文件）和语法文件（.y 文件），然后，使用命令“gcc -o parser lex.yy.c parser.tab.c ast.c”生成语法分析器 parser；最后，使用 parser 进行测试，测试输出的二元式如图 2-9 所示，可以正确识别各类的符号，并以二元式的形式进行输出。

```
(class, CLASS)
(_My_test123, IDENTIFIER)
({, LC)
(}, RC)
(//测试类体为空&标识符, LINECOMMENT)
(\n, EOL)
(class, CLASS)
(My_test, IDENTIFIER)
(\n, EOL)
({, LC)
(\n, EOL)
(\n, EOL)
(int, TYPEINT)
(num, IDENTIFIER)
(,, SEMI)
(//测试int类型, LINECOMMENT)
(\n, EOL)
```

图 2-9 词法分析器测试结果图

本次实验的难点在于 flex 工具的使用，以及设计合理正则式匹配各类符号。

正则式设计和排列中，最需要注意的是，由于 flex 在进行匹配时遵循**最长匹配**以及**最先匹配**原则，即优先选择可匹配长度最大的规则进行匹配，如果匹配的长度相等，则选择排列次序靠前的规则进行匹配。因此，标识符 ID 不能排在关键字之前，否则会将关键字错误判断为标识符。

3 实验三 自定义语言语法分析器设计与实现

3.1 实验目的与内容

选择语法分析器实现方法，可以选择自上而下的方法，也可以使用自底向上的方法，并将语法分析器与实验 2 构建的词法分析器进行关联，实现一个自底向上的 LR 分析器，具体实验要求如下：

1. 调研语法生成工具 Bison；
2. 根据实验（一）定义的语法规则，编写 bison 源程序；
3. 用 bison 生成语法分析程序；
4. 显示语法分析结果-语法树

3.2 语法分析器构建

本次实验所设计的语言 Defan，详细语言文法详见 1.2 Defan 语言文法设计，根据设计的语法规则，按 Bison 要求的格式，编辑 Parser.y 文件。

3.2.1 声明部分

该部分内容包含语法分析中需要的头文件、宏定义和全局变量的定义等，如题 3-1 所示，除了基本的几个头文件外，还引用了头文件“def.h”，该文件中包含构建语法树所需要的数据结构的定义与函数的声明。

```
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "def.h"
extern int yylineno;
extern char *yytext;
extern FILE *yyin;
int yylex();
void yyerror(const char* fmt, ...);
void display(struct ASTNode *,int);
void displayRoot();
```

图 3-1 语法文件声明部分代码截图

3.2.2 辅助定义部分

辅助定义部分主要包含 3 个部分的内容，分别是终结符与语法树结点类型定义部分，语义值的类型定义部分以及优先级与结合性定义部分，这 3 个部分可以

帮助进行语法分析并为语法树的构建提供帮助，具体的定义如下：

1. 终结符与语法树结点类型定义

由于单独实现语法分析器是毫无意义的，只有将词法分析器和语法分析器进行结合，形成词法语法分析器才可以有效地扫描语言并实现分析，因此本次实验将联合使用 Flex 和 Bison 进行分析器的生成。

为实现两者的有效沟通，需要统一单词的种类编码。使用%token 并罗列单词种类码标识符后，使用 Bison 文件 parser.y 进行编译后，会生成枚举常量头文件“parser.tab.h”为定义的终结符提供一个编号。

同时，为方便后续进行语法树的生成，需要为每种树的结点定义一个唯一的标识符，树类型的标识符也需要使用%token 的方式进行定义（部分结点种类可以由终结符类型进行标识，因此，此部分与终结符定义存在部分重叠，可以对终结符定义进行部分复用以减少符号的定义）。

在本次设计的语言 Defan 中，终结符主要分为标识符、常量、运算符、界符以及保留字。其中，由于终结符与常量需要使用其语义值，因此需要定义语义类型，标识符与常量定义详见 2，其他各个部分的定义如下：

```
/* 运算符 */
%token ASSIGNOP PLUS MINUS STAR DIV MOD AND OR NOT
%token AUTOPLUS AUTOMINUS PLUSASSIGNOP MINUSASSIGNOP
%token STARASSIGNOP DIVASSIGNOP MODASSIGNOP

/* 保留字 */
%token NULL_P TRUE FALSE VOID BOOL
%token IF ELSE WHILE FOR BREAK CONTINUE RETURN
%token PRINT SCAN CLASS THIS

/* 界符 */
%token DOT COMMA SEMI LP RP LB RB LC RC EOL
```

在运算符中，ASSIGNOP 表示赋值符号“=”，运算符 PLUS、MINUS、STAR、DIV、MOD 分别表示运算符“+”“-”“*”“/”以及“%”，运算符 AND、OR、NOT 分别表示逻辑运算符“&&”“||”“!”，运算符 AUTOPLUS、AUTOMINUS 分别表示运算符“++”以及“--”，运算符 PLUSASSIGNOP、MINUSASSIGNOP、STARASSIGNOP、DIVASSIGNOP、MODASSIGNOP 分别表示“+=”“-=”“*=”“/=”“%=”。

保留字标识使用保留字原文的大写形式，而在界符中，DOT 表示点号 “.”，COMMA 表示逗号 “,” SEMI 表示分号 “;”，LP 与 RP 表示小括号 “(” 与 “)”，LB 与 RB 表示中括号 “[” 与 “]”，LC 与 RC 表示大括号 “{” 与 “}”，符号 EOL 则表示文件尾。

语法树结点的类型标识符定义如下，具体表示的结点的类型详见规则部分：

```
%token CLASS_DEF_LIST CLASS_DEF
%token FIELD_LIST VAR_DEF_FUNC_DEF TYPE_COL
%token VAR_DEF_LIST ID_DEF_LIST
%token FUNC_DEF_ACT FORMALS_DEF VAR_DEF_S
%token STMT_BLOCK STMT_LIST PRINT_STMT SCAN_STMT RETURN_EXP
%token IF_STMT IF_ELSE_STMT ID_ACTUALS
%token FOR_STMT WHILE_STMT BREAK_STMT CONTINUE_STMT
%token EXP_EXP EXP_ID EXP_ID_ACTUALS ACTUALS EXP_LIST
```

2. 语义值的类型定义

终结符和非终结符都会有一个属性值，这个值的类型默认为**整型**。为方便后续进行语法树构建以及语义分析等实验，需要为特定的终结符与非终结符设置特定的类型，如非终结符的属性类型是树结点（**struct ASTNode**）的指针，标识符 ID 的属性类型为字符串，常量的属性类型为常量的类型。

由于不同的符号对应不同的属性类型，因此使用联合将这多种类型统一起来，联合体的定义如下所示，前3个分别存储INT、FLOAT以及CHAR类型的常量属性值，type_id用于存储标识符等需要传递字符串的类型的属性值，ptr为语法树的结点类型，用于传递树的指针数值以便于语法树的构建。

```
%union {
    int      type_int;
    float    type_float;
    char      type_char;
    char      type_id[128];
    struct    ASTNode *ptr;
};
```

终结符（标识符、常数）以及非终结符属性值的类型说明如下，非终结符由于都是语法树的结点，因此都需要设置为指针属性，常量和标识符等设置为对应的联合体中的属性即可。

```
/* 终结符的语义值类型 */
%token <type_int> INT
%token <type_float> FLOAT
```



```

%token <type_string> CHAR
%token <type_id> ID RELOP TYPE ERRORID

/* 非终结符的语义值类型 */
%type <ptr> Program ClassDefList ClassDef Field FieldList
%type <ptr> VariableDef IDList FunctionDef Formals Variable VariableList
%type <ptr> StmtBlock Stmt StmtList SimpleStmt PrintStmt ScanStmt
%type <ptr> IfStmt WhileStmt ForStmt BreakStmt ContinueStmt ReturnStmt
%type <ptr> Expr ExprList Actuals Constant Type

```

3. 优先级与结合性定义

为了使得所设计的语言文法简单易懂，设计语言文法通常会导致文法产生二义性，无法使用标准的 LR 分析对语言进行分析（会产生大量 shift/reduce 移进/归约冲突、reduce/reduce 归约/归约冲突），为解决这种问题，可以通过设定单词优先级和结合性解决二义性的问题。

本实验设计的语言 Defan 的运算规则类似于 C 语言，符号之间的优先级关系与符号的结合性的定义与 C 语言类似。此外，除了基本的运算符的优先级和结合性的定义外，还需要处理“-”前置负号以及 if 语句和 else 语句之间匹配导致的“else”悬空的问题，具体的优先级从低到高以及结合性的定义如下：

```

%left ASSIGNOP PLUSASSIGNOP MINUSASSIGNOP
%left STARASSIGNOP DIVASSIGNOP MODASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV MOD
%left AUTOPLUS AUTOMINUS
%right UMINUS NOT
%left LB RB LP RP DOT
%nonassoc ELSE

```

其中，赋值符号“=”以及“+=”等复合赋值运算符的优先级最低，然后依次是或运算符“||”以及与运算符“&&”，“RELOP”表示比较运算符“>”“>=”“<”“<=”“==”“!=”。之后与C语言类似为算数运算符以及前置运算符，最后是界符包括作用域运算符点号“.”、小括号“()”与中括号“[]”，优先级最高的为ELSE，这样设置后else将匹配最邻近的那个else。

3.2.3 规则部分

Bison采用的是LR分析法，规则之间使用分号隔开，Bison在进行LR分析时可以在每条规则后给出相应的语义动作（类似S-属性文法），本次实验使用这些语义动作生成语法树。

由于在辅助定义部分对终结符以及非终结符进行了语义属性类型的定义，因此可以使用“\$\$”访问规则左部的变量（右部的非终结符都被定义为语法树指针类型struct ASTNode*），使用“\$1”等“\$+数字”的形式访问对应的右部符号。

根据实验1中定义的文法书写规则，并设计相应的语义动作生成语法树，以下方的语句为例：

```
ClassDefList: {$$=NULL;}  
             | ClassDef ClassDefList {$$=mknnode(2,CLASS_DEF_LIST,yylineno,$1,$2);}  
             ;
```

规则分为左部与右部，两者使用冒号“:”分隔，左部为非终结符，右部为终结符与非终结符闭包，不同右部使用符号“|”分隔，并可使用大括号给出语义动作，最后使用分号“;”作为规则结束符，详细语法规则设计见附录。

为了在进行LR分析过程中构建语法树，需要使用结点生成函数mknnode生成语法树结点，该函数参数个数可变，第一个参数为子结点的个数，第二个参数为语法树结点的类型，第三个为语言的行号，之后为子结点的序号。

3.2.4 用户定义部分

这部分的代码会被原封不动的拷贝到parser.tab.c中，以方便用户自定义所需要的函数。在这次实验中，提供了报错函数yyerror，该函数在语言出现词法和语法错误时可准确、及时地进行报错，并给出错误位置以及错误性质。

主程序以及报错函数yyerror用户函数部分代码如下所示：

```
int main(int argc, char *argv[])  
{  
    yyin=fopen(argv[1],"r");  
    if (!yyin) return -1;  
    yylineno=1;  
    yyparse();  
    return 0;  
}
```

```

#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    fprintf(stderr, "Grammar Error at Line %d Column %d: ",
        yylloc.first_line, yylloc.first_column);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, ".\n");
}

```

3.3 抽象语法树建立

为方便后续的语义分析以及中间代码生成，在语法分析阶段可以生成抽象语法树AST，为实现功能，在原实验样板中提供的结构的基础上新增部分量。

由于该抽象语法树的建立是为了进行语义分析以及中间代码的生成，因此，还包含符号表相关、语义分析相关以及中间代码生成相关的变量，具体的树结点结构如下：

```

struct ASTNode {
    struct ASTNode *ptr[4];
    int kind;
    int pos;
    union {
        char type_id[33];
        int type_int;
        float type_float;
        char type_char;
        char type_string[33];
    };

    /* 符号表相关 */
    int place;          /* 符号表的地址 */
    int type;           /* 表达式的类型 */
    char flag;          /* 表达式的类别，与符号表类别标识相同 */
    int offset;         /* 偏移量 */
    int width;          /* 占数据字节数 */
    int num;            /* 函数参数个数 */
    int col_num;        /* 数组维数 */
    int index_class;    /* 类变量的类定义符号表地址 */
    int return_flag, return_pos; /* 标识函数中是否存在返回语句 */
}

```

```

/* 中间代码生成相关 */
char Etrue[15],Efalse[15]; /* 布尔表达式真假转移目标的标号 */
char Snext[15];           /* 下一条语句位置标号 */
char Sbreak[15], Scontinue[15]; /* 中断与继续位置标号 */
struct codenode *code;     /* 中间代码链表头指针 */
};

```

抽象语法树的结点的基础主体包括三部分，第一是结点种类标记变量kind；第二是结点语义属性值联合体，该联合体中保存树结点的语义属性值；第三是子结点指针ptr[4]，通过文法设计可知每个语法树的结点最多存在4个子结点。

除了基本的主体结构外，为进行语义分析和中间代码的生成，该树结点中还包含了符号表相关以及中间代码生成相关到变量，具体作用参见注释以及实验4、实验5以及实验6。

在使用Bison进行语法分析的同时，使用语义动作建立语法树。然后将语法树打印出来，具体打印语法树代码见源码。

3.4 语法分析器调试与测试

为简化实验流程，本实验实际上采用了将实验2与实验3进行联合编译与测试的方式进行实验的调试和测试，具体的调试和测试流程如下：

使用命令“flex Defan.l”以及“bison -d -v parser.y”编译词法文件（.l文件）和语法文件（.y文件）；然后，使用命令“gcc -o parser lex.yy.c parser.tab.c ast.c”生成语法分析器 parser；最后，使用 parser 进行测试，具体调试过程如下：

3.4.1 调试问题一

在语法分析器的调试中，最重要的部分是使用 Bison 编译语法文件(.y文件)，如果定义的文法存在问题，或者优先级和结合性定义存在问题会导致大量的移进/归约（shift/reduce）冲突以及归约/归约（reduce/reduce）冲突。在本次实验调试过程中，我也遇到了类似的问题。

```

PrintStmt: PRINT LP Expr ExprList RP {$$=mknode(1,PRINT_STMT,yylineno,$3);};
ScanStmt: SCAN LP Expr ExprList RP {$$=mknode(1,SCAN_STMT,yylineno,$3);};
State 41 conflicts: 1 shift/reduce
State 136 conflicts: 1 shift/reduce
State 144 conflicts: 8 shift/reduce
State 145 conflicts: 8 shift/reduce
State 154 conflicts: 11 reduce/reduce

```

图 3-2 文法定义错误代码以及冲突结果图

首先，如图 3-2 所示，由于文法定义存在问题表达式未使用分号或逗号进行分隔，导致语言出现歧义现象，进而造成了 LR 分析过程中的冲突现象。

如图 3-3 所示，以状态 41 产生的冲突为例，当状态 41 接收到小括号 “(” 时，出现了移进/归约（shift/reduce）冲突也即存在某种情况下，标识符 ID 遇到小括号时不应该进行移进操作，这不符合语言的定义，当 ID 后直接出现小括号 “(” 时一定表示函数调用，必须要进行移进。

```
state 41

80 Expr: ID . LP Actuals RP
82   | ID .

LP | shift, and go to state 75
LP [reduce using rule 82 (Expr)]
$default reduce using rule 82 (Expr)
```

图 3-3 状态41语法冲突结果图

找到问题后，分析出现此问题的原因，该问题可以通过优先级解决，例如将 ID 的优先级设定低于 LP 即可。但是，本次实验设计的文法 Defan 不应出现除了函数调用之外的 ID 后紧跟 LP（小括号 “(” ）的情况，因此应寻找文法设计的问题。在寻找过后发现是 Print 状态的定义存在问题，具体问题如图 3-2 所示。

3.4.2 调试问题二

第二个问题是经典的 “dangling esle” 悬空 ELSE 问题，也就是当存在多个 if 时 else 的匹配问题，通常 else 将匹配最邻近的那个 if。为解决此问题，需要为 ELSE 及 if 规则设置优先级，可将 if 字句的优先级设置低于 ELSE 字符优先级，这样当遇到 else 时会优先进行移进操作，使得 else 匹配上最邻近的 if 子句。

传统的解决方案是使用 “%prec LOWER_THEN_ELSE” 为 if 子句设置优先级，并规定 LOWER_THEN_ELSE 优先级低于 ELSE。但是，若 if 子句不设置优先级，规则优先级由最右端含优先级定义的字符代表，而 if 子句中 RP 优先级低于 ELSE，因此不额外设置优先级也可以实现此功能，解决方案如图 3-4 所示。

```
%left LB RB LP RP DOT
%nonassoc ELSE

IfStmt: IF LP Expr RP Stmt {$$=mknnode(2,IF_STMT,yylineno,$3,$5);}
      | IF LP Expr RP Stmt ELSE Stmt {$$=mknnode(3,IF_ELSE_STMT,yylineno,$3,$5,$7);}
```

图 3-4 悬空else问题解决方案图

3.4.3 语法分析器测试

当测试程序中存在的语法问题时，如图 3-5 所示，当第 15 行漏写分号时，语法分析会报错，期望分号 SEMI 错误接收到 FOR，出现语法错误。

```
13      int main()
14      {
15          Scan(num)
16          for(i=1;i<num;i++)
17              Print(f.fibo(i));
18          return 0;
19      }
```

```
(num, IDENTIFIER)
(, RP)
(\n, EOL)
(for, FOR)
Grammar Error at Line 16 Column 9: syntax error, unexpected FOR, expecting SEMI.
```

图 3-5 语法分析异常情况测试结果图

当测试程序不存在语法错误时，如图 3-6 所示，会显示当前程序的语法树，语法树使用缩进作为层次区分的标识，每一层的语法树会显示当前层语法树结点的类型以及语义属性值。

```
1      class Fibo
2      {
3          int fibo(int a)
4          {
5              if(a==1||a==2) return 1;
6              return fibo(a-1)+fibo(a-2);
7          }
8      }
```

```
类定义: Fibo (8)
  类成员定义: (8)
    类型: int
    函数名: fibo
    函数形式参数: (3)
      类型: int
      参数名: a
    复合语句: (7)
      复合语句的语句部分:
        条件语句(IF): (6)
          条件:
            OR
            ==
              ID: a
              INT: 1
            ==
              ID: a
              INT: 2
          IF子句: (6)
            返回语句: (5)
              INT: 1
```

图 3-6 测试代码以及部分语法树截图

3.4.4 实验小结

本次实验的难点在于使用 Bison 构建语法分析器，深入了解 Bison 分析器的对于各种情况的处理，尤其是优先级和结合性的处理是本次实验的重点。

本次实验中最容易出现的问题就是移进/归约（shift/reduce）冲突以及归约/归约（reduce/reduce）冲突这两种冲突。对于这两种问题既可以通过重写文法以及定义优先级、结合性实现。

对于归约/归约（reduce/reduce）冲突通常是通过重写文法的方式实现的。当出现归约/归约冲突时，表示当前文法存在冲突的规则，即同一段语句可能被解释成不同的含义，这对于正常的高级语言都是不可接受的，健壮的语言不应该对同样的语句给出不同的解释。

对于移进/归约冲突理论上都可以使用优先级定义的方式消除，但是一种可读性良好的语言，除了正常的运算符以及 ELSE 等少数情况，其他符号都不应该设置优先级。大量的优先级设置虽然可以避免冲突，但是会给语言的使用者产生不必要的负担。

4 实验四 符号表管理和属性计算

4.1 实验目的与内容

参考实验教程，设计符号表数据结构和关键管理功能，同时通过扫描抽象语法树完成相关属性计算，具体的实验要求如下：

1. 选择合适的数据结构实现符号表；
2. 实现符号表上的操作，包括创建符号表、插入表项、查询表项、修改表项、删除表项、释放符号表空间等等；
3. 扫描语法树并计算符号的属性值；
4. 输出符号表和符号表动态变化过程。

4.2 符号表设计与管理

符号表是语义分析、中间代码生成以及目标代码生成的基础，符号表中记录了编译过程中各类“符号”的特征信息，这些信息包括：符号名称、符号的种类、符号的类型、符号的偏移等等信息，具体的符号表结构与管理如下。

4.2.1 符号表设计

符号表可以采用不同的数据结构实现，实验中可采用不同的数据结构来实现，本次实验采用教程中推荐的顺序表实现符号表，符号表的结构如下：

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

符号表 `symbolTable` 包含两个变量，第一个是符号表项组成的数组 `symbols[MAXLENGTH]`，其最大长度使用宏定义调整；第二个是栈顶指针 `index`，其初始值为 0，每次填入新的符号后栈顶指针加 1。

符号表的表项中包含的属性较多，有些用于语义分析，有些则是用于中间代码生成以及目标代码地生成，各个属性详细作用见附录的注释，在此仅做简述。


```

/* 符号表的表项 */
struct symbol {
    char flag, name[128], alias[10];
    int level, type, paramnum, offset, is_pointer, index_class;
    int index_array[20];
};

```

在各属性中，`flag` 为符号的种类标记，‘C’表示类定义，‘F’表示函数定义，‘P’表示函数参数定义，‘Q’表示函数数组参数定义，‘V’表示变量定义，‘T’表示临时变量，‘A’表示数组变量定义；`name` 为符号的名称，记录通过词法分析获得的语义值；`alias` 为符号的别名，是符号在符号表中的唯一标识，用于解决嵌套层次中标识符重名的问题。

在本语言中，允许不同作用域的变量重名，但是一个作用域内不允许名字重复，因此使用 `level` 标识符号的层数，用于符号的作用域管理。特别地，本语言规定类定义为 0 层，类数据成员以及函数成员定义为 1 层，函数参数定义为 2 层，其他嵌套层数依次递增。

`type` 为变量的类型，包括 `int`、`float`、`char`、`void` 以及 `class`，当 `flag` 为 ‘C’ 时本项无意义，当 `flag` 为 ‘F’ 时表示返回值的类型；同时，若 `type` 的值为 `class` 则需要配合 `index_class` 指向类的定义在符号表的下标。

当 `flag` 为 ‘Q’ 或 ‘A’ 时，表示当前量为数组变量或参数，需要使用 `paramnum` 保存数组的维数，数组 `index_array[]` 保存各维度的数组的上限，保存维数的顺序与数组定义的顺序相同；同时，当 `flag` 为 ‘F’ 是 `paramnum` 表示参数个数（规定函数返回值不得是数组类型）。

对于类数据成员，`offset` 表示成员相对于类基址的偏移；对于函数成员来说，`offset` 表示函数的活动记录大小；对于局部变量和函数参数来说，`offset` 表示变量相对于函数栈基址的偏移。

变量 `is_pointer` 标识当前量是否为指针量，如果为指针量则在中间代码生成中存在特殊处理，用于中间代码的生成。

4.2.2 符号表管理

首先，需要实现符号表上的操作，包括创建符号表、插入表项、查询表项、修改表项、删除表项、释放符号表空间等操作；由于使用数组模拟顺序表的方式

建立符号表，因此不需要进行符号表的创建与释放操作；且符号表的表项为顺序存放，如果要删除原有的表项，只需要使用新的表项将原有的表项覆盖即可；因此只需要定义插入表项、查询表项以及修改表项的操作即可。

其次，由于本语言支持使用作用域管理变量的访问，为此，使用全局变量 LEV 来管理作用域，LEV 的初始值为 0。并设计结构 `symbol_scope_begin` 存储各个作用域的基地址，使用栈的形式管理基地址，当进入一个新的作用域后，将新作用域压栈；当退出当前作用域后，将栈顶的作用域出栈，并修改符号表的指针下标；同时，设计结构 `class_index` 保存各个类的基地址，用于类的快速查找。

```
struct symbol_scope_begin {  
    int TX[100];  
    int top;  
} symbol_scope_TX;
```

```
struct class_index {  
    int TX[100];  
    int index;  
} class_index;
```

最后，当通过遍历抽象语法树进行语义分析时，需要进行两大类检查操作，第一类是**控制流检查**，该类检查与符号表无关；第二类是**唯一性检查、名字的上下文相关性检查以及类型检查**，该类检查与符号表相关，需要将定义的符号填入符号表中，然后读取符号表进行以上类型的三种检查，因此本节主要就第二类检查中的符号表管理流程进行描述，具体的描述如下：

（1）符号的声明与定义

在抽象语法树的扫描过程中，当读取到类的声明、函数的声明、函数参数的声明以及变量的声明语句相关的结点时，需要将声明的符号以一定的形式填入符号表中。本语言中存在 3 种不同类型的声明，分别是类声明、函数声明（包含参数声明）、以及变量声明（包含类成员以及局部变量的声明），对于这 3 类声明需要进行不同的操作。

变量存在作用域限制，每次遇到一个嵌套语句结点 `STMT_BLOCK`，首先对 LEV 加 1，表示准备进入一个新的作用域；当登记新符号时，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。

(2) 唯一性、上下文相关性以及类型检查

首先，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；其次，名字的出现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等；最后，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。

这些操作都需要在 `symbolTable` 中，从栈顶向栈底方向查询对应的符号，并就查询情况进一步进行语义分析。

4.3 语法树属性计算

使用先根遍历遍历抽象语法树，在遍历过程中，需要根据结点的类型计算其属性值。

如说明部分结点需要填写符号表，执行部分结点需要读取符号表进行语义分析；当进行代码片段的结点访问时，需要根据当前偏移（`offset`）修改子结点的地址偏移值，实现继承属性的计算；在子结点计算完成，得到该结点的宽度值（`width`）后修改当前结点的宽度，完成综合属性的计算，具体的操作非常多，详见代码。

3.4 符号表测试

为体现符号表动态变化过程，本次实验将在进入某个作用域前和退出某个作用域前将符号表输出，以观察符号表的动态变化过程，测试代码如图 4-1 所示，测试进入 `main` 函数之前以及退出 `main` 函数前的符号表。

```
1  class Main
2  {
3      int m,n;
4      int fibo(int a)
5      {
6          if(a==1||a==2) return 1;
7          return fibo(a-1)+fibo(a-2);
8      }
9      int main()
10     {
11         int a,b,c;
12         m=5;
13         a=fibo(5);
14         return 0;
15     }
16 }
```

图 4-1 符号表测试代码截图

如图 4-2 所示，在进入 main 函数之前，符号表中仅包含 main 函数的定义，不包含参数的定义与变量的定义；当进入 main 函数后符号表，依次出现 a、b、c 这 3 个表项，并统一设置别名，各种属性设置正确，测试成功。

Symbol Table							
变量名	别名	层号	类型	类索引	维数	标 记	偏移量
Main	v1	0				C	0
m	v2	1	int			V	0
n	v3	1	int			V	4
fibonacci		1	int		1	F	20
a	v4	2	int			P	12
main		1	int			F	0

Symbol Table							
变量名	别名	层号	类型	类索引	维数	标 记	偏移量
Main	v1	0				C	0
m	v2	1	int			V	0
n	v3	1	int			V	4
fibonacci		1	int		1	F	20
a	v4	2	int			P	12
main		1	int			F	0
a	v5	2	int			V	12
b	v6	2	int			V	16
c	v7	2	int			V	20

图 4-2 符号表测试结果图

5 实验五 静态语义分析

5.1 实验目的与内容

定义错误类型，熟悉语义规则的表达形式，完成类型检查；借助于符号表以及一些相关的数据结构，完成静态语义检查，可以根据实验定义的语言，检查出如下所述类型的静态语义错误：

- (1) 使用未定义的变量；
- (2) 调用未定义或未声明的函数；
- (3) 在同一作用域，名称的重复定义（如变量名、函数名、结构类型名以及结构体成员名等），可拆分成几种类型的错误，如变量重复定义等；
- (4) 对非函数名采用函数调用形式；
- (5) 对函数名采用非函数调用形式访问；
- (6) 函数调用时参数个数不匹配，如实参表达式数太多、或实参表达式数太少；
- (7) 函数调用时实参和形参类型不匹配；
- (8) 对非数组变量采用下标变量的形式访问；
- (9) 数组变量的下标不是整型表达式；
- (10) 对非结构变量采用成员选择运算符“.”；
- (11) 结构成员不存在；
- (12) 赋值号左边不是左值表达式；
- (13) 对非左值表达式进行自增、自减运算；
- (14) 对结构体变量进行自增、自减运算；
- (15) 类型不匹配，如数组名与结构变量名间运算，需要指出类型不匹配错误；
有些需要根据定义的语言的语义自行进行界定。
- (16) 函数返回值类型与函数定义的返回值类型不匹配；
- (17) 函数没有返回语句（当函数返回值类型不是 `void` 时）；
- (18) `break` 语句不在循环语句中；
- (19) `continue` 语句不在循环语句中；

5.2 语义分析处理

静态语义分析主要包含 2 方面的内容，分别是**控制流检查**与**符号表相关检查**，其中，符号表相关检查又可细分为 3 类，包括唯一性检查、名字上下文相关性检查以及类型检查，各类检查的要求如下：

（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，如一个跳转语句会使控制转移到由标号指明的后续语句，若标号没有对应到语句，那么就出现语义错误。

其次，**break**、**continue** 语句必须出现在循环语句当中，需要进行判断；函数必须存在返回语句，且函数必须能正常返回（可以在函数最后显式添加返回语句，以防止函数无法正常返回）。

对于判断函数是否存在返回语句，计划在 AST 结点中新增变量 `return_flag`，该变量为结点的**综合属性**，作用是向上传递某个代码块是否存在返回语句。

对于类型为 `STMT_LIST` 的抽象语法树的结点，其子结点为 `Stmt` 以及 `StmtList` 类型。当 `StmtList` 不为空时，当前结点的 `return_flag` 为 `StmtList` 子结点的 `return_flag` 数值（对于相邻的代码片段，只要排列靠后的代码片段存在返回语句即可，排在前面的代码会顺序执行或跳转到靠后的片段）。

对于非 `STMT_LIST` 类型的结点，如果为表达式类结点则 `return_flag` 置为 0，其他语句类的结点（如 `while` 结点、`for` 结点等）则直接向上传递，对于 `return` 语句则将 `return_flag` 置为 1。

判断 **break**、**continue** 语句是否出现在循环语句当中，设立全局变量 `flag_loop` 用于标识是否在循环中，`flag_loop` 在进入新循环中进行自增，退出循环后进行自减，当扫描到 `break` 类型以及 `continue` 类型的结点时，使用 `flag_loop` 进行判断，若 `flag_loop` 为 0 则报错，否则不报错。

（2）唯一性检查

检查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，同一个类中不得出现相同的成员变量等。该检查在进行**类定义**、**变量定义**以及**函数定义**时进行，具体处理方式如下：

对于类定义，不得定义名称相同的类。在填充符号表时，当定义的符号为类即 `flag` 的数值为“C”时，如图 5-1 所示，使用结构体 `class_index`（保存类定义

的基址)进行类查找,查找是否存在同名的类。若存在同名类则返回-1,并报第3种错误“error(3): 类重复定义”,若不存在则将符号填入符号表继续分析。

```
if(flag == 'C')
{
    for(i=0; i<class_index.index; i++)
        if (!strcmp(symbolTable.symbols[class_index.TX[i]].name, name))
            return -1;
}
```

图 5-1 类定义唯一性检查代码图

对于变量定义,唯一性检查较为复杂。对于类数据成员定义,成员之间不得重名,但是与成员函数参数可以重名;对于局部变量,可以与类的数据成员以及外层作用域的局部变量同名(内层覆盖外层),但是同一个作用域中不得重名。

对于函数定义,函数名不得相同,由于本语言不支持重载,因此严格要求函数之间不得重名。

(3) 名字的上下文相关性检查

名字的出现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等。该检查通常在变量进行引用计算时进行,需要检查的类别很多,具体如下:

使用未定义的变量;调用未定义或未声明的函数;对非函数名采用函数调用形式,对函数名采用非函数调用形式访问;对非数组变量采用下标变量的形式访问;对非结构变量采用成员选择运算符“.”以及结构成员不存在。

(4) 类型检查

类型检查包括表达式计算与赋值过程中类型匹配问题,也包括函数参数类型不匹配以及参数数目不匹配的问题,本次定义的问题中,以下需要在类型检查过程中进行检查:

包括但不限于,函数调用时参数个数不匹配,如实参表达式数太多、或实参表达式数太少;函数调用时实参和形参类型不匹配;函数返回值类型与函数定义的返回值类型不匹配;数组变量的下标不是整型表达式;赋值号左边不是左值表达式;对非左值表达式进行自增、自减运算;对结构体变量进行自增、自减运算等等。

5.3 语义分析测试

编写测试程序将所有错误类型检测一遍,测试代码如图 5-2 所示。

```

37      d; // (1) 使用未定义的变量
38      func(1); // (2) 调用未定义或未声明的函数
39      int a; // (3) 在同一作用域, 名称的重复定义: 变量重复定义
40
41      a(); // (4) 对非函数名采用函数调用形式
42      fibo; // (5) 对函数名采用非函数调用形式访问
43      fibo(1,4); // (6) 函数调用时参数个数不匹配, 实参表达式个数太多
44      fibo(); // (6) 函数调用时参数个数不匹配, 实参表达式个数太少
45      fibo('1'); // (7) 函数调用时实参和形参类型不匹配
46      a[4]; // (8) 对非数组变量采用下标变量的形式访问
47      int_array_1[1.1][3]; // (9) 数组变量的下标不是整型表达式
48      a.b; // (10) 对非结构变量采用成员选择运算符“.”
49      class_var_1.int_num_1=5;
50      this.class_var_1.int_num_1=5;
51      class_var_1.int_num_5; // (11) 结构成员不存在
52      a+b=c; // (12) 赋值号左边不是左值表达式
53      (a+b)++; fibo(5)++; // (13) 对非左值表达式进行自增、自减运算
54      char_1++; // (14) 对非整型变量进行自增、自减运算
55
56      a='a'; // (15) 类型不匹配
57      a=a+char_1; // (15) 类型不匹配
58
59      break; // (18) break语句不在循环语句中
60      continue; // (19) continue语句不在循环语句中
61
62      void t; // (20) 不得声明void类型的参数或变量
63      class test_class_5 templ; // (21) 使用未定义的类
64
65      a>=4; // (23) 逻辑运算不可出现在除if/while/for的函数体中
66      -'a'; // (24) 负号只能对int/float进行操作
67      array+5; // (25) 不可对数组量以及类类型量进行运算"

```

图 5-2 静态语义分析测试代码图

使用语义分析程序进行分析, 得到如下分析结果:

在 37 行,error(1): 使用未定义的变量 d

在 38 行,error(2): 调用未定义或未声明的函数 func

在 39 行,error(3): 变量重复定义 a

在 41 行,error(4): 对非函数名采用函数调用形式 a

在 42 行,error(5): 对函数名采用非函数调用形式访问 fibo

在 43 行, error(6): 函数调用时参数个数不匹配, 实参个数太多

在 44 行, error(6): 函数调用时参数个数不匹配, 实参个数太少

在 45 行, error(7): 函数调用时实参和形参类型不匹配

在 46 行, error(8): 对非数组变量采用下标变量的形式访问

在 47 行, error(9): 数组变量的下标不是整型表达式

在 47 行, error(8): 对非数组变量采用下标变量的形式访问

在 48 行, error(10): 对非结构变量采用成员选择运算符 ‘.’

在 51 行, error(11): 结构成员不存在

在 52 行, error(12): 赋值号左边不是左值表达式

在 53 行, error(13): 对非左值表达式进行自增、自减运算

在 53 行, error(13): 对非左值表达式进行自增、自减运算

在 54 行, error(14): 对非 int 型变量进行自增、自减运算

在 56 行, error(15): 赋值语句两端类型不匹配

在 57 行, error(15): +-*/%运算的运算数类型不匹配

在 57 行, error(15): 赋值语句两端类型不匹配

在 59 行, error(18): break 语句不在循环语句中

在 60 行, error(19): continue 语句不在循环语句中

在 62 行, error(20): 不得声明 void 类型的参数或变量

在 63 行, error(21): 使用未定义类 test_class_5

在 65 行, error(23): 逻辑运算不可出现在除 if/while/for 的函数体中

在 66 行, error(24): 负号只能对 int/float 进行操作

在 67 行, error(25): 不可对数组量以及类类型量进行运算

在 103 行, error(16): 函数返回值类型与函数定义的返回值类型不匹配

6 实验六 中间代码生成

6.1 实验目的与内容

遍历 AST，计算相关的属性值，利用符号表，生成以三地址代码 TAC 作为中间语言的中间语言代码序列并显示，具体要求如下：

- (1) 掌握中间代码形式设计技术；
- (2) 熟悉中间代码生成规则以及逻辑表达式的生成技术；
- (3) 完成各种可执行语句的中间代码生成。

6.2 中间语言的定义

采用三地址代码 TAC 作为中间语言，中间语言代码的定义如表 6-1 所示。

表 6-1 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
$x := y$	赋值操作	ASSIGN	y		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
$x := * y$	右部取地址运算	EXP_POINT	y		x
$* x := y$	左部取地址运算	POINT_EXP	y		x
FUNCTION f	定义函数 f	FUNCTION			f
PARAM x	函数形参	PARAM			x
LABEL x	定义标号 x	LABEL			x
GOTO x	无条件转移	GOTO			x
IF x [relop] y GOTO z	条件转移	[relop]	x	y	z
ARG x	传实参 x	ARG			x
$x := \text{CALL } f$	调用函数(有返回值)	CALL	f		x
RETURN x	返回语句	RETURN			x

6.3 翻译模式设计

6.3.1 函数定义

翻译模式采用 L-翻译模式，通过遍历语法树 AST 实现中间代码的生成，为了生成中间代码序列，定义了几个函数：

(1) newtemp 函数，生成一临时变量，登记到符号表中，以“temp+序号”的形式组成的符号串作为别名，符号名称用空串的形式登记到符号表中；

(2) newLabel 函数，以“LABEL+序号”形式生成一个标号；

(3) genLabel 函数，生成标号语 TAC 的中间代码语句；

(4) genIR 函数，生成一条 TAC 的中间代码语句，为方便最后阶段的目标代码生成，需要将其偏移量（offset）这个属性和数据类型同时带上；

(5) merge 函数，将多个 TAC 语句序列顺序连接在一起。

6.3.2 数据结构定义

同时，为了完成中间代码的生成，对于抽象语法树 AST 中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式的计算方法完成属性计算，具体树结点结构如下（部分内容隐去，详见代码）：

```
struct ASTNode {
    struct ASTNode *ptr[4];
    int kind;
    int pos;
    union {
        char type_id[33];
        int type_int;
        float type_float;
        char type_char;
        char type_string[33];
    };
    /* 符号表相关 */
    int place;          /* 符号表的地址 */
    int type;           /* 表达式的类型 */
    char flag;          /* 表达式的类别，与符号表类别标识相同 */
    int offset;         /* 偏移量 */
    int width;          /* 占数据字节数 */
    int num;            /* 函数参数个数 */
};
```

```

int col_num;      /* 数组维数 */
int index_class;  /* 类变量的类定义符号表地址 */
int return_flag, return_pos; /* 标识函数中是否存在返回语句 */
/* 中间代码生成相关 */
char Etrue[15], Efalse[15]; /* 布尔表达式真假转移目标的标号 */
char Snext[15];             /* 下一条语句位置标号 */
char Sbreak[15], Scontinue[15]; /* 中断与继续位置标号 */
struct codenode *code;      /* 中间代码链表头指针 */
};

```

.place: 符号在符号表中的序号（index 下标）；

.type: 数据的类型（int、float、char、class 以及 void），用于表达式计算；

.flag: 数据的类别，如 ‘C’ 表示类名，‘F’ 表示函数名，‘P’ 表示函数参数名，‘Q’ 表示函数数组参数名，‘V’ 表示变量名，‘T’ 表示临时变量名，‘A’ 表示数组变量名；

.offset: 记录类数据成员与类基址的偏移，或局部变量和临时变量在活动记录中的偏移；另外对函数，利用该数据项保存活动记录的大小；

.width: 记录定义的变量和临时单元所需要占用的字节数，借此能方便地计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小；

.num: 用于统计和传递函数参数的个数，以便于后续进行函数名登记；

.col_num: 用于统计和传递数组变量的维数，以便于后续进行数组名登记；

.index_class: 记录类定义的符号表地址，若当前变量为类类型则需要使用此量保存当前类在符号表中的位置；

.return_flag: 记录语法单位是否存在返回语句，用于语义分析；

.code: 记录中间代码序列的起始位置，该属性就是一个链表的头指针；

.Etrue&.Efalse: 布尔表达式值为真、假时要转移的程序位置；

.Sbreak&.Scontinue: break 语句以及 continue 语句要转移的程序位置；

.Snext: 该结点的语句序列执行完后，要转移到的程序位置；

定义完这些属性和函数后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。

6.3.3 结点翻译方法

可以将抽象语法树结点分为若干类：执行语句、基本表达式、布尔表达式和其它结点等，详细的中间代码翻译方法参见源代码，此处仅展示部分具有代表性的翻译方法。

1. 执行语句类结点翻译方法

执行语句包括 while 循环语句、for 循环语句、if 以及 if-else 分支语句以及涉及到代码块的语句，如 Stmt、StmtList 以及其衍生语句 VariableDef、SimpleStmt、BreakStmt、ContinueStmt、ReturnStmt、PrintStmt、ScanStmt 以及 StmtBlock 等，此处仅展示 if 语句以及 if-else 语句的翻译方法。

表 6-2 语句类结点的中间代码生成

当前结点类型	翻译动作
IF_STMT T1 条件子树 T2 if 子句子树	访问到 T: T1.Etrue=newLabel, T1.Efalse= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code T1.Etrue T2.code
IF_ELSE_STMT T1 条件子树 T2 if 子句子树 T3 else 子句子树	访问到 T: T1.Etrue=newLabel, T1.Efalse=newLabel T1.Snext= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code T1.Etrue T2.code goto T.Snext T1.Efalse T3.code

2. 基本表达式类结点翻译方法

表达式的计算，分为两种类型，第一种是基本表达式的计算，第二种是布尔表达式的计算。通常第一种可以构造表达式语句，第二种是在控制语句（条件语句和循环语句）中作为控制条件。

基本表达式的翻译，需要按计算次序，完成表达式中的所有计算操作步骤，最后得到表达式的值，翻译表 6-3 给出了部分基本表达式类结点的中间代码翻译方法。

基本的翻译方法与实验指导书给出的方法类似，与之不同的是，由于本语言支持类以及多维数组的定义与使用（类数组同样也支持），因此表达式的翻译异常复杂，具体参见源代码。

表6-3 基本表达式类结点的中间代码生成

当前结点类型	翻译动作
ASSIGNOP	访问到 T: T.place=T1.place
T1 左值表达式子树	访问 T 的所有子树后: T.code=T1.code T2.code T1.alias=
T2 左值表达式子树	T2.alias
OP 算术运算符。	t _i =newtemp, t _i 在符号表的入口赋值给 T.place
T1 第一操作数子树	访问 T 的所有子树后:
T2 第二操作数子树	T.code=T1.code T2.code t _i =T1.alias OP T2.alias
UMINUS	t _i =newtemp, t _i 在符号表的入口赋值给 T.place
T1 操作数子树	访问 T 的所有子树后: T.code=T1.code t _i =- T1.alias
RELOP 关系运算符	t _i =newtemp, t _i 在符号表的入口赋值给 T.place,
T1 第一操作数子树	Label1=newLabel, Label2=newLabel。
T2 第二操作数子树	访问 T 的所有子树后:
	T.code=T1.code T2.code
	if T1.alias RELOP T2.alias goto label1
	t _i =0 goto label2 label1: t _i =1 label2:
AND	t _i =newtemp, t _i 在符号表的入口赋值给 T.place,
T1 第一操作数子树	Label1=newLabel。
T2 第二操作数子树	访问 T 的所有子树后:
	T.code=T1.code T2.code t _i =T1.alias * T2.alias
	if t _i ==0 goto label1 t _i =1 label1:

6.3.4 数组与类翻译思想（重点）

本班设计的 Defan 语言为纯面向对象的语言，细节的语法与 C 语言类似，因此控制流语句的翻译、定义声明类语句的翻译以及基本的表达式的翻译与实验指导教程类似，不想花过多的篇幅去描述，本次将对类以及数组的翻译与实现进行叙述，这是本语言实现过程的关键，也是本次实验中的亮点。

数组和类的定义与访问使用语法生成式进行定义，详见实验 1 的定义部分，数组和类支持的难点在于存储空间的分配以及如何用中间代码对数组和类进行

通用化的访问，针对这两个问题，我使用如下解决方案：

1. 数组与类的空间分配问题

计划使用**堆分配**的方式为类和数组进行空间分配，但是考虑到本次实现的语言的简易性，堆分配的回收与再利用过于复杂，因此使用“伪堆分配”的方式进行分配，也就是**只分配不回收**。使用约定的指针作为堆指针，当需要进行分配时直接自增对应的偏移即可。

设计 New 语句进行显式的空间分配，New 语句会返回空间地址，并根据偏移移动堆指针，数组变量与类变量中保存的是地址值（类似于 Java 中），指向堆中的某片区域。

当进行数组与类的定义，借助符号表中定义的**数组内情向量**以及**类的偏移量**，使用 New 语句进行空间分配。以数组为例，如图 6-1 所示，将数组的维数以及内情向量（每一维度的上限）填入符号表，并计算数组长度 length，最后使用 New 语句进行空间的分配。

```
/* 如果变量为数组变量，则将内情向量填入符号表 */
symbolTable.symbols[rtn].paramnum=T->col_num;
if(rtn!=-1 && T->col_num > 0)
{
    length=1;
    fill_array(T->col_num, rtn);
    for(i=0; i<T->col_num; i++)
        length*=index_array[i];
    opn1.kind=INT;
    opn1.const_int = length*4;
    result.kind = ID;
    strcpy(result.id, symbolTable.symbols[rtn].alias);
    result.offset = T->offset;
    T->code=merge(2, T->code, genIR(NEW, opn1, opn2, result));
}
```

图 6-1 数组定义中间代码生成图

数组元素除了支持基本元素（int、char、float），同时也支持类变量作为数组元素，在进行类数组的创建时，先创建数组，然后创建类变量，将类变量的地址依次填充到数组元素中即可完成定义。

2. 数组与类的访问

首先，对数组 $a[n_1, n_2, \dots, n_n]$ 的访问，假设数组采用行优先的连续布局，数组首元素地址保存在 a 中，则数组元素 $a[i_1, i_2, \dots, i_n]$ 的地址 D 可以如下计算：

$$D = (\dots((i_1 * n_2 + i_2) * n_3 + i_3) * n_4 + \dots + i_{n-1}) * n_n + i_n$$

特别的，对于一维数组可以使用 i_1 直接访问，多维数组可以使用上述公式进行计算即可，进行具体的访问使用 $*(a + D * 4)$ 的方式取出对应的数值，本次为了实现数组的访问，定义 “ $x := *y$ ” 以及 “ $*x := y$ ” 进行取地址的指针操作，详细的实现过程见源码。

类的访问与数组类似，需要使用类成员相对于类基址的偏移计算访问的具体地址，假设类变量为 x 则采用 $*(x + \text{offset})$ 的形式访问。

类数组的访问形式更为复杂，是数组访问与类访问的组合， $*(a + D * 4)$ 取出的是类的基址，具体的访问使用 $*(*(a + D * 4) + \text{offset})$ 的形式。

3. 类成员函数以及 this 指针访问

由于本次设计的语言为面向对象的语言，存在 “**this.变量名**” 以及 “**变量名**” 访问类数据成员的访问方式，以及使用 “**类变量.方法**” 的形式访问类的方法成员，本次使用**隐式添加 this 指针**的方式解决这个问题。

在进行函数声明与定义时，在所有参数之前**隐式定义 this 指针参数**，在进行函数访问时，隐式地为 this 指针进行赋值。

对于 “**类变量.方法**” 的访问形式，将类变量的内容直接为 this 指针进行赋值即可，若是直接使用 “**方法**” 或 “**this.方法**” 形式进行访问，则将当前函数的 this 指针为 this 指针进行赋值。

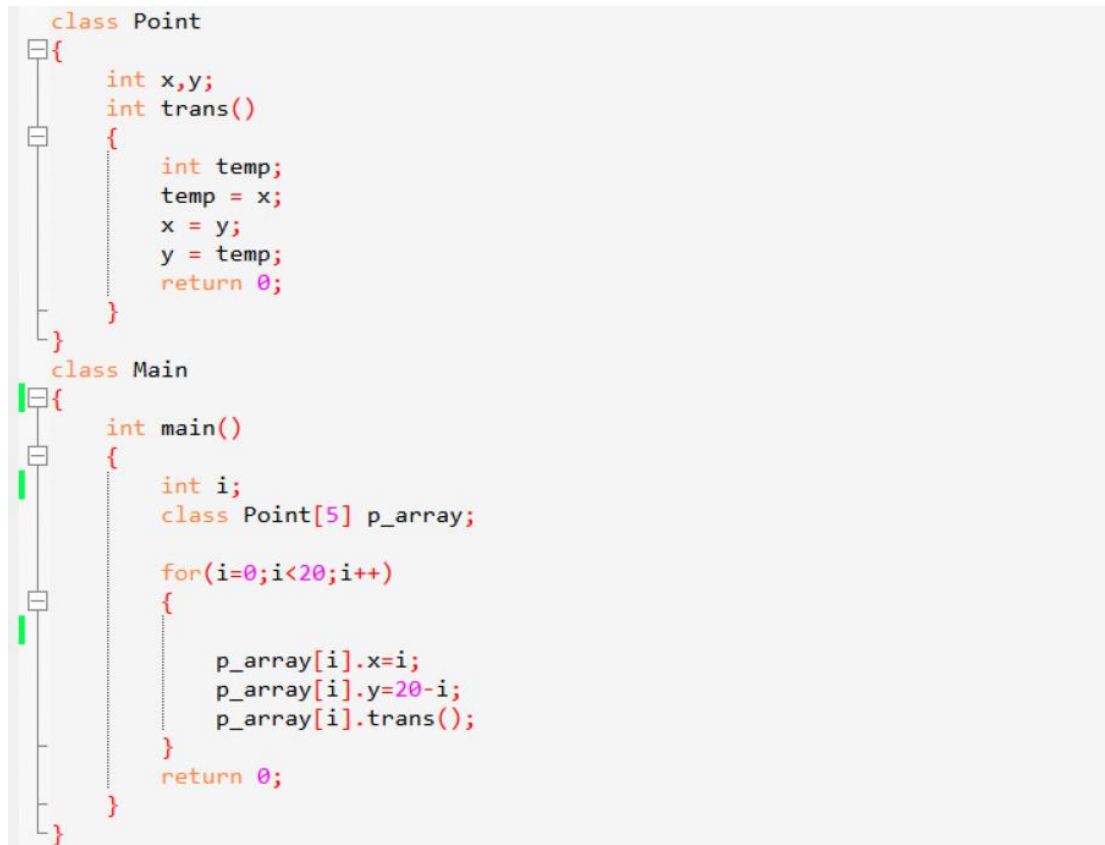
在进行这样的操作后，对于类数据成员就可以使用 $*(this + \text{offset})$ 的形式进行访问了。需要注意的是，如图 6-2 所示，main 函数的 this 指针需要在翻译的时候显式地赋值，main 函数所在的类需要在所有代码前先行创建空间。

```
/* 如果为main函数，则生成this变量 */
if(strcmp(T->type_id, "main")==0)
{
    opn1.kind=INT;
    /* 变量大小等待最后填入见semantic_Analysis0 */
    opn1.const_int = 0;
    result.kind = ID;
    strcpy(result.id, symbolTable.symbols[rtn].alias);
    result.offset = T->offset;
    main_p = genIR(NEW, opn1, opn2, result);
    T->code=merge(2, T->code, main_p);
    main_num = class_index.TX[class_index.index-1];
}
```

图 6-2 main函数this指针生成图

6.4 中间代码生成测试

使用命令“gcc -o parser lex.yy.c parser.tab.c ast.c support.c semantic_analysis.c”生成中间代码生成程序，使用命令“parser test.c”对测试程序进行分析并生成中间代码，测试程序的代码如图 6-3 所示。



```
class Point
{
    int x,y;
    int trans()
    {
        int temp;
        temp = x;
        x = y;
        y = temp;
        return 0;
    }
}

class Main
{
    int main()
    {
        int i;
        class Point[5] p_array;

        for(i=0;i<20;i++)
        {
            p_array[i].x=i;
            p_array[i].y=20-i;
            p_array[i].trans();
        }
        return 0;
    }
}
```

图 6-3 中间代码生成测试代码图

生成的中间代码如下：

```
FUNCTION trans :
    PARAM v4
    temp1 := v4 + #0
    temp1 := * temp1
    v5 := temp1
    temp2 := v4 + #0
    temp3 := v4 + #4
    temp3 := * temp3
    * temp2 := temp3
    temp4 := v4 + #4
    * temp4 := v5
    temp5 := #0
    RETURN temp5
FUNCTION main :
```

```

PARAM v7
temp6 := #0
v8 := temp6
LABEL label5 :
    temp7 := #20
    IF v8 < temp7 GOTO label4
    GOTO label3
LABEL label4 :
    temp8 := v8 * #4
    temp9 := temp8 + v9
    temp9 := * temp9
    temp10 := temp9 + #0
    * temp10 := v8
    temp11 := v8 * #4
    temp12 := temp11 + v9
    temp12 := * temp12
    temp13 := temp12 + #4
    temp14 := #20
    temp15 := temp14 - v8
    * temp13 := temp15
    temp16 := v8 * #4
    temp17 := temp16 + v9
    temp17 := * temp17
    ARG temp17
    temp18 := CALL trans
LABEL label6 :
    v8 := v8 + #1
    GOTO label5
LABEL label3 :
    temp19 := #0
    RETURN temp19

```

7 实验七 代码优化

7.1 实验目的与内容

使用 DAG 图对中间代码进行局部优化，本节要求以合适的数据结构构建 DAG 图，全局优化为选做要求，具体要求如下：

- (1) 掌握局部优化技术，能够将中间代码分为基本块的集合；
- (2) 生成 DAG 图并实现局部优化
- (3) 计算优化率。

7.2 DAG 图数据结构定义

为实现 DAG 优化以及后续全局删除未引用变量，需要定义合适的数据结构存储 DAG 图以及标识符，具体的定义如下：

首先，由于 DAG 优化是以基本块为单位，因此需要设计一张局部标识符表，用于存储单个基本块内部的标识符。

定义结构体 ID_table，具体结构如下，结构体成员包括 name 表示标识符名称，node_index 指向 DAG 图结点的下标，表示当前标识符所附加的 DAG 图结点，all_num 指向全局标识符表的下标，该量主要配合全局标识符表进行全局优化，用于未引用变量的删除。

```
struct ID_table{
    char name[33];
    int node_index, all_num;
}id_table[40];
```

其次，设计全局标识符表，用于存储整个程序的标识符，该结构体的主要作用是进行全局的变量引用检查，以便于删除未引用变量。

定义结构体 all_table，具体结构如下，成员包括 name 表示标识符名称，is_save 表示是否被引用，用于未引用变量的删除。

```
struct all_table{
    char name[33];
    int is_save;
}all_table[200];
```

最后，也是最重要的需要定义 DAG 图的结点结构，参照 DAG 图的建构过程，为简化设计与实现，将使用数组模拟图结构。DAG 需要分析 3 种类型的四元式，分别是 0 型、1 型以及 2 型四元式，其子结点个数不超过 3 个，且结点可以分为常数类结点以及标识符类的结点。

根据以上分析，设计如下结构作为 DAG 图的结点，其中 kind 表示结点的类型与四元式的类型是相同的；is_leaf 与 is_valid 分别表示是否为叶节点以及是否有效（可能在分析过程中删除某些结点，因此需要表示是否有效）；联合体保存常数量；id_num 与 id_index[] 为结点附加标识符的个数以及指向的 id_table 的下标；child_num 与 child_index[] 为子结点个数以及子结点 DAG 下标。

```
struct DAG_Node{
    /* 结点类型 & 是否叶节点 & 是否有效*/
    int kind, is_leaf, is_valid;
    union {
        int      const_int;
        float    const_float;
        char     const_char;
    };
    /* 附加标识符个数以及下标数组 */
    int id_num;
    int id_index[MAX_ID_NUM];

    /* 子节点个数以及下标数组 */
    int child_num;
    int child_index[3];
}DAG[200];
```

7.3 DAG 优化与全局优化

7.3.1 DAG 优化流程

DAG 优化的基本流程主要分为 3 步，首先划分基本块，然后构造基本块四元式的 DAG 图，最后按照 DAG 结点顺序，重写基本块四元式。

对于不同类型的四元式，构造 DAG 图的流程与方法存在不同，根据四元式的类型以及使用的变量数可以将四元式分为三大类，0 型、1 型以及 2 型四元式。

为便于理解本处对后续可能使用的操作进行叙述：

- (1) **NODE (X)** 表示标识符 **X** 当前指向的 **DAG** 图的结点的下标序号；
 - (2) 定义 **NODE (X)** 表示将标识符 **X** 将填入标识符表 **id_table** 中，创建 **DAG** 图结点并将 **B** 在标识符表中的下标填入 **id_index[]** 中，**id_num** 进行自增；
 - (3) **X** 附加到结点 **n** 表示将标识符 **X** 在标识符表中的下标填入 **id_index[]** 中，**id_num** 进行自增；
 - (4) 把 **X** 从 **NODE (X)** 结点上附加标识符集中删除，该流程需要先检查 **id_num** 是否为 1，若是则表示删除后此节点将没有标识符可以标识，不删除；若大于 1 则将 **X** 在标识符表中的下标从 **id_index[]** 移除，并将 **id_num** 进行自减；
- 各类四元式 **DAG** 图的具体处理过程如下：

1. 0 型四元式 **A:=B** 处理流程

- (1) 如果 **NODE (B)** 无定义，则将标识符 **B** 填入标识符表 **id_table** 中，并构造标记为 **B** 的 **DAG** 图结点（将 **B** 在标识符表中的下标填入 **id_index[]** 中，**id_num** 进行自增），**B** 标识符表项的 **node_index** 指向刚定义的 **DAG** 图结点，假设图结点为标号为 **n**；
- (2) 如果 **NODE (A)** 无定义，则把 **A** 附加在结点 **n** 上并令 **NODE(A)=n**；否则先把 **A** 从 **NODE(A)** 结点上附加标识符集中删除，把 **A** 附加到新结点 **n** 上并令 **NODE(A)=n**。

2. 1 型四元式 **A:=op B** 处理流程

- (1) 如果 **NODE (B)** 无定义，则定义 **NODE (B)**；
- (2) 如果 **NODE (B)** 是标记为常数的叶结点，则转 (3)，否则转 (4)。
- (3) 执行 **op B**（合并已知量），得到新常数 **P**。若 **NODE (B)** 是当前四元式时新构造的结点，则删除；若 **NODE (P)** 无定义，则定义 **NODE (P)**，转 (5)。
- (4) 检查 **DAG** 中是否已有一结点，其唯一后继为 **NODE (B)**，且标记为 **op**（找公共子表达式）。如果没有则构造该结点 **n**，否则就把已有的结点作为它的结点并设该结点为 **n**。
- (5) 如果 **NODE (A)** 无定义，则定义 **NODE (A)**；否则先把 **A** 从 **NODE(A)** 结点上附加标识符集中删除，并把 **A** 附加到新结点 **n** 上并令 **NODE(A)=n**。

3. 2 型四元式 $A:=B \text{ op } C$ 处理流程

- (1) 如果 $\text{NODE}(B)$ 无定义, 则定义 $\text{NODE}(B)$;
- (2) 如果 $\text{NODE}(C)$ 无定义, 则定义 $\text{NODE}(C)$;
- (3) 如果 $\text{NODE}(B)$ 和 $\text{NODE}(C)$ 都是常数叶结点转 (4), 否则转 (5);
- (4) 执行 $B \text{ op } C$ (合并已知量), 得新常数 P 。若 $\text{NODE}(B)$ 或 $\text{NODE}(C)$ 是当前新构造结点, 删除; 若 $\text{NODE}(P)$ 无定义, 则定义 $\text{NODE}(P)$, 转 (6);
- (5) 检查中 DAG 中是否已有一结点, 其左后继为 $\text{NODE}(B)$, 其右后继为 $\text{NODE}(C)$, 且标记为 op (找公共子表达式)。如果没有, 则构造该结点 n , 否则就把已有的结点作为它的结点并设该结点为 n ;
- (6) 若 $\text{NODE}(A)$ 无定义, 则定义 $\text{NODE}(A)$; 否则先把 A 从 $\text{NODE}(A)$ 结点上附加标识符集中删除, 把 A 附加到新结点 n 上并令 $\text{NODE}(A)=n$ 。

按照如上步骤进行 DAG 图的建构, DAG 图构建完成后, 使用 DAG 图重写四元式, 由于使用数组模拟图, 数组的下标从小到大为图结点的构造顺序, 因此按照数组下标从小到大重写即可, 具体的重写流程如下:

- (1) 从左至右写右部有标记的叶结点的四元式;
- (2) 写内结点的四元式, 当某内结点存在附加标记, 生成对应四元式。如果有多个附加标记, 从第二个开始生成赋值四元式。

7.3.2 全局优化流程

本次全局优化仅对为引用量进行删除, 在 DAG 重写结束后, 对重写的四元式进行扫描, 对每个引用 (在右部出现即为被引用) 新的变量更新全局标识符表, 将对应的标识符的 is_save 量置为 1 表示其曾经被引用。

当 DAG 优化完成后, 再对完成的四元式进行二次扫描, 删除那些未引用量, 即当处理一条四元式时, 扫描全局标识符表, 若对应的标识符 is_save 为 0, 表示其未被引用, 将其删除即可。

7.4 代码优化测试

使用构建完成的代码优化程序进行优化效果的测试, 测试代码如图 7-1 所示, 该测试程序使用了类数组, 会产生大量有关地址计算的代码, 这些代码会出现大量公共子表达式, 可以通过优化删除这些语句。

```

int main()
{
    int a,i;
    class Point[5] p_array;

    for(i=0;i<a;i++)
    {
        p_array[i].x=i;
        p_array[i].y=20-i;
        p_array[i].trans();
    }
    return 0;
}

```

图 7-1 代码优化测试代码图

优化前后的中间代码如下，最左侧是未经过优化的版本，中间是经过 DAG 优化的版本，最右侧是经过 DAG 优化以及全局为引用变量删除版本，原中间代码数量为 32 条，最终优化版本的中间代码数为 23，优化率为 28.13%。

<p>FUNCTION main :</p> <p>PARAM v11</p> <p>temp13 := #0</p> <p>v13 := temp13</p> <p>LABEL label6 :</p> <p>IF v13 < v12 GOTO label5</p> <p>GOTO label4</p> <p>LABEL label5 :</p> <p>temp14 := v13 * #4</p> <p>temp15 := temp14 + v14</p> <p>temp15 := * temp15</p> <p>temp16 := temp15 + #0</p> <p>* temp16 := v13</p> <p>temp17 := v13 * #4</p> <p>temp18 := temp17 + v14</p> <p>temp18 := * temp18</p> <p>temp19 := temp18 + #4</p> <p>temp20 := #20</p> <p>temp21 := temp20 - v13</p> <p>* temp19 := temp21</p> <p>temp22 := v13 * #4</p> <p>temp23 := temp22 + v14</p> <p>temp23 := * temp23</p> <p>ARG temp23</p> <p>temp24 := CALL trans</p> <p>LABEL label7 :</p> <p>v13 := v13 + #1</p> <p>GOTO label6</p> <p>LABEL label4 :</p> <p>temp25 := #0</p> <p>RETURN temp25</p> <p>LABEL label3 :</p>	<p>FUNCTION main :</p> <p>PARAM v11</p> <p>temp13 := #0</p> <p>v13 := #0</p> <p>LABEL label6 :</p> <p>IF v13 < v12 GOTO label5</p> <p>GOTO label4</p> <p>LABEL label5 :</p> <p>temp14 := v13 * #4</p> <p>temp17 := temp14</p> <p>temp22 := temp14</p> <p>temp15 := temp14 + v14</p> <p>temp15 := * temp15</p> <p>temp18 := temp15</p> <p>temp23 := temp15</p> <p>temp16 := temp15 + #0</p> <p>* temp16 := v13</p> <p>temp19 := temp15 + #4</p> <p>temp20 := #20</p> <p>temp21 := #20 - v13</p> <p>* temp19 := temp21</p> <p>ARG temp15</p> <p>temp24 := CALL trans</p> <p>LABEL label7 :</p> <p>v13 := v13 + #1</p> <p>GOTO label6</p> <p>LABEL label4 :</p> <p>temp25 := #0</p> <p>RETURN #0</p> <p>LABEL label3 :</p>	<p>FUNCTION main :</p> <p>PARAM v11</p> <p>v13 := #0</p> <p>LABEL label6 :</p> <p>IF v13 < v12 GOTO label5</p> <p>GOTO label4</p> <p>LABEL label5 :</p> <p>temp14 := v13 * #4</p> <p>temp15 := temp14 + v14</p> <p>temp15 := * temp15</p> <p>temp16 := temp15 + #0</p> <p>* temp16 := v13</p> <p>temp19 := temp15 + #4</p> <p>temp21 := #20 - v13</p> <p>* temp19 := temp21</p> <p>ARG temp15</p> <p>temp24 := CALL trans</p> <p>LABEL label7 :</p> <p>v13 := v13 + #1</p> <p>GOTO label6</p> <p>LABEL label4 :</p> <p>RETURN #0</p> <p>LABEL label3 :</p>
---	--	---

8 实验八 目标代码生成

8.1 实验目的与内容

将 TAC 的指令序列转换成目标代码，目标语言为汇编语言，具体要求如下：

- (1) 选择目标代码指令集（可以选用在组原课设 CPU 的指令集）；
- (2) 掌握寄存器分配的基本算法；
- (3) 编写目标代码生成程序；
- (4) 运行目标代码程序，正确输出结果。

8.2 目标语言的指令定义

目标语言选定 MIPS32 指令序列，但是如果要在组原课设 CPU 上运行，则不能包含 mul 以及 div 等乘除指令，本次主要在 MARS 进行运行，因此不考虑这个问题，TAC 指令和 MIPS32 指令的对应关系如表 8-1 所示。其中 reg(x)表示变量 x 所分配的寄存器。

表 8-1 中间代码与 MIPS32 指令对应关系

中间代码	操作类型	MIPS32 指令
$x := \#k$	ASSIGN (常数)	li reg(x), k
$x := y$	ASSIGN	move reg(x), reg(y)
$x := y + z$	PLUS	add reg(x), reg(y), reg(z)
$x := y - z$	MINUS	sub reg(x), reg(y), reg(z)
$x := y * z$	STAR	mul reg(x), reg(y), reg(z)
$x := y / z$	DIV	div reg(y), reg(z) mflo reg(x)
$x := *y$	EXP_POINT	lw reg(temp), (reg(y)) move reg(x), reg(temp)
$*x := \#k$	POINT_EXP	li reg(temp), k sw reg(temp), (reg(x))
$*x := y$	POINT_EXP	lw reg(temp), (reg(y)) sw reg(temp), (reg(x))

LABEL x	LABEL	x:
GOTO x	GOTO	j x
IF x==y GOTO z	EQ	beq reg(x), reg(y), z
IF x!=y GOTO z	NEQ	bne reg(x), reg(y), z
IF x > y GOTO z	JGT	bgt reg(x), reg(y), z
IF x >= y GOTO z	JGE	bge reg(x), reg(y), z
IF x < y GOTO z	JLT	blt reg(x), reg(y), z
IF x <= y GOTO z	JLE	ble reg(x), reg(y), z
x:=CALL f	CALL	jal f （参数入栈未写入详见代码） move reg(x), \$v0
RETURN x	RETURN	move \$v0, reg(x) jr \$ra

8.3 寄存器分配与目标代码生成

寄存器的分配方法使用朴素分配法，在进行目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。

由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用\$t1和\$t2，运算结果使用\$t3，翻译的方法如表 8-2 所示。

表 8-2 朴素寄存器分配的翻译

中间代码	操作类型	MIPS32 指令
x := #k	ASSIGN（常数）	li \$t3, k sw \$t3, x 的偏移量(\$sp)
x := y	ASSIGN	lw \$t1, y 的偏移量(\$sp) move \$t3, \$t1 sw \$t3, x 的偏移量(\$sp)
x := y + z	PLUS	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	MINUS	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)

$x := y * z$	STAR	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)
$x := y / z$	DIV	lw \$t1, y 的偏移量(\$sp) lw \$t2, z 的偏移量(\$sp) div \$t1,\$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
$x := * y$	EXP_POINT	lw \$t1, y 的偏移量(\$sp) lw \$t3, (\$t1) sw \$t3, x 的偏移量(\$sp)
$* x := \#k$	POINT_EXP	li \$t1, k lw \$t3, x 的偏移量(\$sp) sw \$t1, (\$t3)
$* x := y$	POINT_EXP	lw \$t1, y 的偏移量(\$sp) lw \$t3, x 的偏移量(\$sp) sw \$t1, (\$t3)
IF $x=y$ GOTO z	EQ	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) beq \$t1,\$t2,z
IF $x \neq y$ GOTO z	NEQ	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	JGT	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) bgt \$t1,\$t2,z
IF $x \geq y$ GOTO z	JGE	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	JLT	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) blt \$t1,\$t2,z
IF $x \leq y$ GOTO z	JLE	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) blt \$t1,\$t2,z
$x := \text{CALL } f$	CALL	
RETURN x	RETURN	move \$v0, x 的偏移量(\$sp) jr \$ra

对于函数调用，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间等，为简化实现，本次所有的参数都使用堆栈传递，具体步骤如下：

(1) 首先，根据符号表中函数定义项得到该函数活动记录的大小，开辟活动记录空间和保存返回地址；根据函数定义中的参数个数 `paramnum`，获得各个实参值所存放的单元，取出后送到形式参数的单元中；

(2) 使用 `jal f` 转到函数 `f` 处，并在函数执行完成后，释放活动记录空间和恢复返回地址；使用 “`sw $v0, X 的偏移量($sp)`” 获取返回值送到 `X` 中。

8.4 目标代码的生成测试

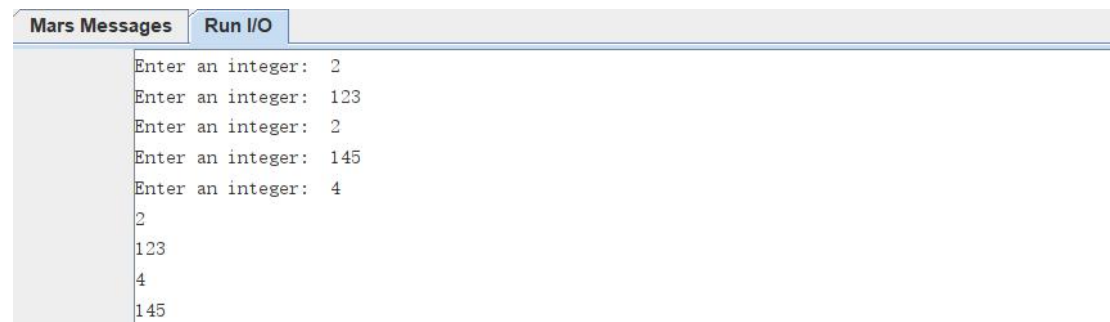
编写测试程序测试编译器的效果，测试程序如图 8-1 所示，定义了两个类分别是 `Point` 坐标点类以及 `Main` 入口类，其中 `Point` 类中包含 `x` 与 `y` 两个成员分别表示 `x` 坐标与 `y` 坐标，以及方法 `trans` 将 `x` 坐标与 `y` 坐标进行交换。从控制台读取点的个数以及 `x`、`y` 坐标后进行交换，并输出。

本测试程序使用了类数组进行相关数据的保存与运算，类数组使用伪堆分配法进行空间分配。

```
1  class Point
2  {
3      int x,y;
4      int trans()
5      {
6          int temp;
7          temp = x; x = y; y = temp;
8          return 0;
9      }
10 }
11
12 class Main
13 {
14     int main()
15     {
16         int i, num_of_point;
17         class Point[5] p_array;
18         Scan(num_of_point);
19         for(i=0;i<num_of_point;i++)
20         {
21             Scan(p_array[i].x);
22             Scan(p_array[i].y);
23             p_array[i].trans();
24         }
25         for(i=0;i<num_of_point;i++){
26             Print(p_array[i].x); Print(p_array[i].y);
27         }
28         return 0;
29     }
30 }
```

图 8-1 目标代码测试代码图

使用编译器处理测试程序，生成汇编文件“object.s”，使用 MARS 打开编译文件，进行编译并运行，运行结果如图 8-2 所示，第一个为点的个数，后面每两行为一个点的 x 坐标与 y 坐标，通过读取并交换后输出，可以看到正常交换，证明该编译器运行良好。



The screenshot shows the 'Run I/O' window of the MARS MIPS simulator. It displays a sequence of input and output operations. The input consists of five integers: 2, 123, 2, 145, and 4. The output consists of the same five integers: 2, 123, 4, and 145. The output sequence shows that the second and third inputs (123 and 2) have been swapped.

Operation	Value
Enter an integer:	2
Enter an integer:	123
Enter an integer:	2
Enter an integer:	145
Enter an integer:	4
Output	2
Output	123
Output	4
Output	145

图 8-2 测试代码运行结果图

9 总结

9.1 实验完成情况

本次编译器的设计与实现中，基于 Decaf 语言以及 C 语言的，综合设计了语言纯面向对象语言 Defan，该语言的语法细节与 C 语言类似，总体框架继承自 Decaf 语言类似于 Java，数据成员和方法必须定义在类中，且必须存在一个启动类 Main 以及 main 函数，完成了此语言的编译器，具体完成的工作如下：

（1）实验一：基于 Decaf 语言以及 C 语言定义语言纯面向对象语言 Defan，包括词法规则、语法规则等方面，该语言不支持继承、反射以及权限的控制，支持多维数组以及类数组的使用，函数的返回值必须为基本类型或 void；

（2）实验二：使用 Flex 实现词法分析器；

（3）实验三：使用 bison 实现语法分析器，并与实验二的 Flex 联合使用，生成词法语法分析器，能识别词法和语法问题并输出提示；

（4）实验四：实现了顺序表形式的符号表，可以进行作用域的管理与控制，可以实现包括数组、类以及函数、变量等不同类别符号的登记，并为之后的中间代码和目标代码生成做支持；

（5）实验五：能识别基本的 19 种异常，同时额外定义了 6 种异常，如“不得声明 void 类型的参数或变量”等等；

（6）实验六：能正常进行中间代码的翻译，使用“伪堆分配”的方式进行空间分配，使用指针操作实现类以及数组的访问；

（7）实验七：使用 DAG 进行局部优化，然后全局进行未引用变量的删除操作，优化率平均为 20%；

（8）实验八：生成 MIPS 汇编形式的目标代码，能在 MARS 软件上正常运行，支持类变量、数组变量以及类数组变量的访问，且可从控制台进行输入和输出；此外也支持部分程序（不包含 mul 以及 div 指令）在组原设计 CPU 上运行；

本次实验我完成了基本的 8 个实验，在实验指导书的基础上，额外增加了类、多维数组以及类数组的支持，此外实验七除了使用 DAG 完成局部优化外，还在全局删除未引用量（全局优化），达到较好的优化效果。

此外目标代码既可以在 MARS 上也可以在组原设计的 CPU 上运行。

9.2 实验感想

编译原理的实验是大学 3 年中难度极高的一门实验，与其他实验不同的是，本实验设计和实现的是整个编译器，处于计算机世界较为底层的部分。

通过本次实验，我对设计的语言掌握程度达到了如指掌的程度，对于类似与 C 语言的语法也有了更深的了解，理解了语法如此规定的原因，可以说，通过编译原理实验这门实验的磨练，我把握住了高级语言语法设计的内生的逻辑。

通过这八次实验，我最终完成了 Defan 语言的编译器，虽然这个语言相较于成熟的高级语言显得十分简陋，编译器也存在各式各样的问题，但实验带给我的成就感是无法抹煞的。

9.3 展望

编译器作为现代计算机世界的重要基石，编译原理这门课的重要性不言而喻，通过这次实验，我认识到要完成一个真正的编译器，需要付出的时间和精力都不是这次实验所能比的。

以后，如果有机会，我会更加深入地了解编译相关的知识，尤其是代码优化方面的知识，将其与实际生产进行结合，写出效率更高的代码。