浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	张佳瑶 韩耕诗 戴陈威
学　　院：	计算机科学与技术学院
系：	
专　　业：	软件工程
学　　号：	3170103240 3170103236 3170103641
指导教师：	李莹

目录

1 序言

1.1 编译器描述

该编译器是一个类 PASCAL 语言的编译器。实验语言为 C 和 C++。整体分为词法分析、语法分析、语义分析、中间代码生成、目标代码生成五个模块。

1.1.1 词法分析

实现方法：Lex

输入：类 pascal 语言源代码

输出：token 序列

可视化：token 序列和对应的行号打印在 tokens.txt 中。

实现功能：将类 pascal 语言源代码转换为 token 序列。识别不规范的字符串，打印错误信息。

1.1.2 语法分析

实现方法：Yacc

输入：token 序列

输出：语法树

可视化：将语法树的父子节点结构转换为目录递进结构，打印在文件 syntaxTree.txt 中。

实现功能：依据 token 序列构建语法树。识别类 pascal 语言源代码中的语法上不符合语言规范的错误。

1.1.3 语义分析

实现语言：C

输入：语法树

输出：编译信息

可视化：若编译错误，在终端打印错误详情，包含错误类型和错误位置。

实现功能：在语义分析阶段，我们通过构建符号表，对类 pascal 语言源代码做基本的语义分析，检查语义方面的逻辑错误，实现的功能有变量/函数重名检查、变量/函数未定义检查、类型检查，参数个数检查。在这里我们假设不存在自动类型转换，需要类型完全匹配。同时更改语法树中的变量名，加上作用域信息，辅助代码生成。

1.1.4 中间代码生成

实现语言：C

输入：语法树

输出：三地址码形式中间代码文件 ircode.txt

实现功能：在中间代码生成阶段，我们通过遍历更新后的语法树，翻译出三地址码形式的中间代码，基本实现语句的中间代码翻译。

1.1.4.1 目标代码生成

实现语言：C++

输入：中间代码文件

输出：MIPS32 指令序列（包含伪指令）

可视化：在 QtSpim 上模拟运行

实现功能：将三地址码中间代码转换为 MIPS32 指令序列（包含伪指令），支持整型简单变量的函数逻辑运算。

1.1.5 编译运行环境

Linux 环境下的编译和运行

（1）Linux 2.6 以上版本

（2）GCC3.4 以上版本

（3）Bison 2.2 以上版本

（4）Flex 2.5.33 以上版本

1.2 文件说明

实验报告 report_24.pdf

原代码文件：

词法分析-中间代码生成：pascal-complier

目标代码生成：genCode

可执行文件：

词法分析-中间代码生成：complier

目标代码生成：genCode

测试案例

1.3 组员分工

戴陈威：词法分析、语法分析

张佳瑶：语义分析、目标代码生成、测试

韩耕诗：中间代码生成、语法树可视化、测试

2 词法分析

编译器的第一个步骤称为词法分析或扫描。词法分析器读入组成源程序的字符流，并且将它们组织成为有意义的词素的序列。对于每个词素，词法分析器将词法单元（token）作为输出。这个词法单元被传送给下一个步骤，即语法分析。词法分析器通常还要和符号表进行交互。

2.1 扫描程序的实现

扫描程序的任务是从源代码中读取字符并形成由编译器的以后部分（通常是分析程序）处理的逻辑单元。由扫描程序生成的逻辑单元称作记号（token），将字符组合成记号与在一个英语句子中将字母构成单词并确定单词的含义很相像。我们的编译器采用 Lex 作为扫描程序的生成器。

Pascal 语言的记号分为三个典型类型：保留字（如 IF 和 THEN 等）、特殊符号（如算术符号加减号、比较符号、括号、分号、赋值符号等）和“其他”记号（即表示多字符串的记号，如 digit 和 identifier，分别表示数字和标识符）。

尽管扫描程序的任务是将整个源程序转换为记号序列，但扫描程序却很少一次性地完成它。实际上，扫描程序是在分析程序的控制下进行操作的，它通过函数从输入中返回有关命令的下一个记号。该函数在 scan.h 中声明如下，该函数在调用时消耗输入字符从输入中返回下一个记号。

```
TokenType getToken();
```

关于该函数的具体实现在 lexical.l 中实现，如下。

```
TokenType getToken(void)
{
    static int first = True;
    TokenType currentToken;
    if(first)
    {
        first = False;
        lineno++;
        yyin = source;
        yyout = listing;
    }
    currentToken = yylex();
    strncpy(tokenString,yytext,MAXTOKENLEN);
    if(TraceScan) {
        fprintf(listing,"\t%d:\t%s\n",lineno, tokenString);
    }
    return currentToken;
```

```
}
```

扫描程序使用了 3 个全程变量：输入与输出文件变量 source 和 listing（函数中分别对应 yyin 与 yyout），在 globals.h 中声明且在 main.c 中被分配和初始化的整型变量 lineno 行号。在扫描程序中，所要计算的唯一特性是词法或是被识别的记号的串值，它位于变量 tokenString 之中。这个变量同 getToken 一并是提供给编译器其他部分的唯一的两个服务，它们的定义被收集在头文件 scan.h 中。需要注意的是，scan.h 中同时固定了 tokenString 的长度为 41，因此那个标识符也就不能超过 40 个字符。

```
#define MAXTOKENLEN 40
extern char tokenString[MAXTOKENLEN+1];
```

currentToken 存储了调用 yylex()后返回的 TokenType，并利用 strncpy 函数将指定长度 MAXTOKENLEN 的存储在 yytext 中的当前匹配的字符串复制到 tokenString 中。

TraceScan 也是在 global.h 中一并定义，用以标识每一个 token 是否被扫描程序所辨认，如果辨认，则通过输出流 listing 输出扫描程序的结果，包括 lineno 与 tokenString。

2.2 Lex 中的正规表达式与规则行为

Lex 是一个将包含了正规表达式的文本文件作为其输入的程序，此外还包括每一个表达式被匹配时所采取的动作。

如上所示，表示多字符串的记号需要正规表达式来匹配，在 lexical.l 中定义如下。

```
digit [0-9]
letter [a-zA-Z]
intvalue {digit}+
realvalue {digit}+\.{digit}+
charvalue \'.\'
stringvalue \"[^\"]*\"
identifier ("_"|{letter})("_"|{letter}|{digit})*
newline \n
whitespace [ \t]+
```

依次匹配了数字、字母、整型数、实数、字符、字符串、变量名、换行以及空格。需要注意的是，intvalue 和 realvalue 的定义利用了前面定义的 digit，而 identifier 的定义利用了前面定义的 letter 和 digit。由于新行会导致增加 lineno，所以定义还区分新行和其他的空白格（空格和制表位）。

Lex 输入的行为部分由各种记号的列表和 return 语句组成，其中 return 语句返回特定的 toke 记号。在这个 Lex 定义中，在标识符规则之前列出了保留字规则。假若首先列出标识符规则，Lex 的二义性解决规则就会总将保留字识别为标识符。

保留字规则行为定义如下。

```
"abs"             {return ABS;}
```

```
"and"             {return AND;}
"array"           {return ARRAY;}
"begin"           {return TOKEN_BEGIN;}
"boolean"         {return BOOLEAN_TYPE;}
"case"            {return CASE;}
"char"            {return CHAR_TYPE;}
"chr"             {return CHR;}
"const"           {return CONST;}
"div"             {return DIV;}
"do"              {return DO;}
"downto"          {return DOWNTO;}
"else"            {return ELSE;}
"end"             {return END;}
"false"           {return FALSE;}
"for"             {return FOR;}
"function"        {return FUNCTION;}
"goto"            {return GOTO;}
"if"              {return IF;}
"integer"         {return INTEGER_TYPE;}
"maxint"          {return MAXINT;}
"not"             {return NOT;}
"odd"             {return ODD;}
"of"              {return OF;}
"or"              {return OR;}
"ord"             {return ORD;}
"pred"            {return PRED;}
"procedure"       {return PROCEDURE;}
"program"         {return PROGRAM;}
"read"            {return READ;}
"real"            {return REAL_TYPE;}
"read"            {return READ;}
"record"          {return RECORD;}
"repeat"          {return REPEAT;}
"sqr"             {return SQR;}
"sqrt"            {return SQRT;}
"succ"            {return SUCC;}
"then"            {return THEN;}
"to"              {return TO;}
"true"            {return TRUE;}
"type"            {return TYPE;}
"until"           {return UNTIL;}
"var"             {return VAR;}
"while"           {return WHILE;}
"write"           {return WRITE;}
```

```
"writeln"         {return WRITELN;}
```

同时特殊符号的规则行为定义如下。

```
"("         {return LP;}
")"         {return RP;}
"["         {return LB;}
"]"         {return RB;}
"."         {return DOT;}
".."        {return DOTDOT;}
","         {return COMMA;}
";"         {return SEMI;}
":"         {return COLON;}
"+"         {return PLUS;}
"-"         {return MINUS;}
"*"         {return MUL;}
"/"         {return DIV;}
"%"         {return MOD;}
">="        {return GE;}
">"         {return GT;}
"<="        {return LE;}
"<"         {return LT;}
":="        {return ASSIGN;}
"="         {return EQUAL;}
"<>"        {return UNEQUAL;}
```

最后是一些特殊符号，即标识符的规则行为。

```
{intvalue}       {return  TOKEN_INTEGER;}
{realvalue}      {return  REAL;}
{charvalue}      {return  CHAR;}
{stringvalue}    {return  STRING;}
{identifier}     {return  TOKEN_ID;}
{newline}        {lineno++;}
{whitespace}     {}
```

特别注意的是，lex 还需要匹配并处理识别注释以及其他非转义字符等其他输入，对于这些情况要进行合理的报错与忽略以确保正确地更新了 lineno。

```
'(\\.|[^'\n])*$                 {yyerror("lexical");}
\"(\\.|[^"\n])*$                {yyerror("lexical");}
"{"[^{}]*"}"                    {/* ignore comments */}
```

3 语法分析

语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”、“语句”、“表达式”等等，并构造出一棵语法树。语法分析程序判断源程序在结构上是否正确，源程序的结构由上下文无关文法描述。本编译器语法分析程序由 Yacc 工具自动生成。

3.1 分析程序的实现

分析程序的任务是从由扫描程序产生的记号中确定程序的语法结构，以及或隐式或显式地构造出表示该结构的分析树或语法树。因此，可将分析程序看作一个函数，该函数把由扫描程序生成的记号序列作为输入，并生成语法树作为它的输出。

在本编译器分析程序的实现中，记号序列不是显式输入参数，但是当分析过程需要下一个记号时，分析程序就调用 getToken 函数的扫描程序过程以从输入中获得它。分析函数在 parse.h 中声明如下，在 yacc 代码文件 grammar.y 中实现。

```
SyntaxTree parse(void)
{
    yyparse();
    return savedTree;
}
```

savedTree 被用来暂时储存由 yyparse 过程产生的语法树（yyparse 本身可以仅返回一个整型标志）。

3.2 Pascal 语言的上下文无关文法

根据《PASCAL Syntax_2020》，Pascal 语言的文法可以分为如下几部分，每一部分仅列出代表例子。

①程序和块

program -> program_head　routine　DOT

②声明和类型定义

const_part -> CONST　const_expr_list　|　ε

type_part -> TYPE type_decl_list　|　ε

③函数与过程声明

function_decl -> function_head　SEMI　sub_routine　SEMI

procedure_decl -> procedure_head　SEMI　sub_routine　SEMI

④形参和自变量

para_decl_list -> para_decl_list SEMI para_type_list | para_type_list

var_para_list -> VAR name_list

⑤类型

array_type_decl -> ARRAY LB simple_type_decl RB OF type_decl

record_type_decl -> RECORD field_decl_list END

⑥语句

if_stmt -> IF expression THEN stmt else_clause

for_stmt -> FOR ID ASSIGN expression direction expression DO stmt

case_stmt -> CASE expression OF case_expr_list END

⑦表达式和变量

expression_list -> expression_list COMMA expression | expression

factor -> NAME | NAME LP args_list RP | SYS_FUNCT |

SYS_FUNCT LP args_list RP | const_value | LP expression RP

| NOT factor | MINUS factor | ID LB expression RB

| ID DOT ID

3.3 语法树的相关数据结构及其实现

语法树的结构在很大程度上依赖于语言特定的语法结构。这种树通常被定义为动态数据结构，该结构中的每个节点都由一个记录组成，而这个记录的域包括了编译后面过程所需的特性（即：并不是那些由分析程序计算的特性）。节点结构通常是节省空间的各种记录。特性域还可以是在需要时动态分配的结构，就像一个更进一步节省空间的工具。

根据 Pascal 语言的结构类型和表达式类型，语法树节点首先按照它是语句还是表达式等来分类，接着根据语句或表达式的种类进行再次分类。语法树的数据结构类型定义在 global.h 中。

```
typedef enum {
    GeOp,
    GtOp,
    LeOp,
    LtOp,
    EqualOp,
    UnequalOp,
    PlusOp,
    MinusOp,
    OrOp,
```

```
    MulOp,
    DivOp,
    ModOp,
    AndOp,
    NotOp,
    AbsOp,
    ChrOp,
    OddOp,
    OrdOp,
    PredOp,
    SqrOp,
    SqrtOp,
    SuccOp,
    ReadOp,
    WriteOp,
    WritelnOp,
    ToOp,
    DowntoOp
} OpKind;

typedef enum {
    StatementNode,
    ExpressionNode,
    DeclareNode,
    TypeNode
} NodeKind; /* 节点属性：语句、表达式、定义 */

typedef enum {
    LabelStmt,
    AssignStmt,
    GotoStmt,
    IfStmt,
    RepeatStmt,
    WhileStmt,
    ForStmt,
    CaseStmt,
    ProcIdStmt,
    ProcSysStmt,
    FuncIdStmt,
    FuncSysStmt
} StmtKind; /* 语句类型 */

typedef enum {
```

```
    IdExp,
    ConstExp,
    OpExp,
    CaseExp
} ExpKind;

typedef enum {
    ProgramDecl,
    ProgramheadDecl,
    RoutineDecl,
    RoutineheadDecl,
    FunctionDecl,
    FunctionheadDecl,
    ProcedureDecl,
    ProcedureheadDecl,
    ConstDecl,
    VarDecl,
    TypeDecl,
    VarParaDecl
} DeclKind;

typedef enum {
    SimpleSysType,
    SimpleIdType,
    SimpleEnumType,
    simpleLimitType,
    ArrayType,
    RecordType
} TypeKind;

/* ExpType is used for type checking */
typedef enum {
    UnknowExpType,
    UserExpType,
    VoidExpType,
    IntExpType,
    RealExpType,
    PointerExpType,
    CharExpType,
    StringExpType,
    BoolExpType,
    ArrayExpType,
    RecordExpType,
    EnumExpType,
```

```
    LimitExpType,
    FuncExpType
} ExpType;
```

枚举类型 NodeKind 代表语法树节点的类型是声明（包括函数和变量的声明）、语句（包括各种语句，如循环语句、选择语句等）、表达式（包括各种表达式及各种变量、常量）、类型（包括各种数据结构类型）。

对于不同的语句，有枚举类型 StmtKind。对于不同的表达式，有枚举类型 ExpKind。对于不同的声明，有枚举类型 DeclKind。对于不同的类型，有枚举类型 TypeKind。

另外，还有枚举类型 ExpType，在后面的表达式类型检查中会用到它。

枚举类型 OpKind 代表各种操作符和操作函数类型。

```
#define MAXCHILDREN 4

typedef struct TreeNode {
    struct TreeNode* child[MAXCHILDREN];
    struct TreeNode* sibling;
    int lineno;
    NodeKind nodeKind;
    union {
        StmtKind stmt;
        ExpKind exp;
        DeclKind decl;
        TypeKind type;
    } kind;
    union {
        OpKind op;
        int intValue;
        char charValue;
        char* stringValue;
        float realValue;
        char* name;
    } attr;
    ExpType type;   /* for type checking of exps */
} * SyntaxTree;
```

树节点最大可有 4 个孩子的结构，在 for 语句中可能会涉及到 4 个孩子。

语法树节点中，还有指向孩子和兄弟的指针。lineno 代表节点在 source 中的行数。nodekind 记录节点的类型，根据 nodeKind 的类型选择正确的 kind 并获取值。

C union 类型 kind 代表节点的小类型，这样表示可以节省空间，因为节点不能既是一个声明节点，同时又是一个表达式节点。

C union 类型 attr 代表节点的值，op 记录相关的运算符或操作函数，intValue 代表 int 类型和 boolean 类型的值，charValue、stringValue 和 realValue 分别表示字符、字符串和实数类型的值。当然，如果是 ID 类型，则还有名字 name。

最后的 type 则是用来在后面的表达式类型检查。

下面是关于语法树分配新的节点的操作函数，它们定义在 util.h 中，具体实现在 util.c 中。

```
SyntaxTree newStmtNode(StmtKind s);
SyntaxTree newExpNode(ExpKind e);
SyntaxTree newDeclNode(DeclKind d);
SyntaxTree newTypeNode(TypeKind type);
```

①新增语句类型节点。

```
SyntaxTree newStmtNode(StmtKind stmtKind)
{
    SyntaxTree t = (SyntaxTree)malloc(sizeof(struct TreeNode));
    int i;
    if (t == NULL)
        fprintf(listing, "Out of memory error at line %d\n", lineno);
    else {
        for (i = 0; i < MAXCHILDREN; i++)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodeKind = StatementNode;
        t->lineno = lineno;
        t->kind.stmt = stmtKind;
    }
    return t;
}
```

nodeKind 记为 StatementNode，kind 中的 stmt 记为 stmtKind。

②新增表达式类型节点。

```
SyntaxTree newExpNode(ExpKind expKind)
{
    SyntaxTree t = (SyntaxTree)malloc(sizeof(struct TreeNode));
    if (t == NULL)
        fprintf(listing, "Out of memory error at line %d\n", lineno);
    else {
        int i;
        for (i = 0; i < MAXCHILDREN; i++)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodeKind = ExpressionNode;
```

```
        t->lineno = lineno;
        t->kind.exp = expKind;
        t->type = VoidExpType;
    }
    return t;
}
```

nodeKind 记为 ExpressionNode，kind 中的 exp 记为 expKind。需要注意的是，这里需要将 type 记为 VoidExpType 为后面的类型检查做准备。

③新增声明类型节点。

```
SyntaxTree newDeclNode(DeclKind declKind)
{
    SyntaxTree t = (SyntaxTree)malloc(sizeof(struct TreeNode));
    int i;
    if (t == NULL)
        fprintf(listing, "Out of memory error at line %d\n", lineno);
    else {
        for (i = 0; i < MAXCHILDREN; i++)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodeKind = DeclareNode;
        t->kind.decl = declKind;
        t->lineno = lineno;
    }
    return t;
}
```

nodeKind 记为 DeclareNode，kind 中的 decl 记为 declKind。

④新增类型节点。

```
SyntaxTree newTypeNode(TypeKind typeKind)
{
    SyntaxTree t = (SyntaxTree)malloc(sizeof(struct TreeNode));
    int i;
    if (t == NULL)
        fprintf(listing, "Out of memory error at line %d\n", lineno);
    else {
        for (i = 0; i < MAXCHILDREN; i++)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodeKind = TypeNode;
        t->lineno = lineno;
        t->kind.type = typeKind;
    }
```

```
    return t;
}
```

nodeKind 记为 TypeNode，kind 中的 decl 记为 typeKind。

3.4 Yacc 程序相关实现

首先讨论 Yacc 说明中的定义部分。

```
#define YYPARSER
```

YYPARSER 用以区分 Yacc 和其他代码文件的输出。

接下来包括了一系列头文件。

```
#define YYSTYPE SyntaxTree
```

YYSTYPE 的定义通过使 Yacc 分析过程为指向节点结构的指针返回的值（TreeNode 本身被定义在 global.h 中），这样就允许了 Yacc 分析程序构造出一棵语法树。

```
static SyntaxTree savedTree;    /* stores syntax tree for later return
*/
extern int yylineno;
extern int errFlag;
```

savedTree 被用来暂时储存由 yyparse 过程产生的语法树（yyparse 本身可以仅返回一个整型标志）。yylineno 和 errFlag 分别表示行号和错误信息。

需要特别注意的是，接下来还定义了 yylex 函数，返回的是 getToken 函数的返回值。这是因为 lex 中的输出函数为 getToken，而 Yacc 是调用 lex 的 yylex()来获得标志(token)的。

```
static int yylex(void)
{
    return getToken();
}
```

接下来%token 定义语义值类型。

```
%token PROGRAM FUNCTION PROCEDURE CONST TYPE VAR
%token IF THEN ELSE REPEAT UNTIL WHILE DO CASE TO DOWNTO FOR
%token EQUAL UNEQUAL GE GT LE LT ASSIGN PLUS MINUS MUL DIV OR AND NOT M
OD READ WRITE WRITELN
%token LB RB SEMI DOT DOTDOT LP RP COMMA COLON
%token INTEGER_TYPE BOOLEAN_TYPE CHAR_TYPE REAL_TYPE
%token TRUE FALSE MAXINT
%token ARRAY OF RECORD TOKEN_BEGIN END GOTO
%token TOKEN_ID TOKEN_INTEGER REAL CHAR STRING
%token ABS CHR ODD ORD PRED SQR SQRT SUCC
```

然后是文法规则，文法规则根据《PASCAL Syntax_2020》的条目书写，每个文法规则都有其相结合的动作。

在绝大多数情况下，这些动作表示与该点上的分析树相对应的语法树的构造，需要从 util.h 中调用相关分配节点的函数来分配新的节点，而且也需要指派新树节点的合适的子节点。

以 program 文法规则为例，其相结合的动作如下。

```
program             :   program_head routine DOT
                        {
                            $$ = newDeclNode(ProgramDecl);
                            $$->attr.name = copyString($1->attr.name);
                            $$->child[0] = $2->child[0];
                            $$->child[1] = $2->child[1];
                            savedTree = $$;
                            freeNode($1);
                            free($2);
                        }
                    ;
```

第一个指令调用 newDeclNode 并指派返回的值为 ProgramDecl 的值。接着调用 copyString 函数将$1 也就是 routine 的 ID 复制给该结点。copyString 函数在 util.h 中定义，通过这个函数确保了没有共享存储。

```
char* copyString(char* s)
{
    int n;
    char* t;
    if (s == NULL)
        return NULL;
    n = (int)strlen(s) + 1;
    t = (char*)malloc(sizeof(char) * n);
    if (t == NULL)
        fprintf(listing, "Out of memory error at line %d\n", lineno);
    else
        strcpy(t, s);
    return t;
}
```

接着 routine 节点的两个孩子为 program 语句的树节点的两个孩子。

然后将该树节点赋值给 savedTree，使得其可由 parse 过程之后返回。

最后释放相关存储，freeNode 函数也在 util.h 中定义。

```
void freeNode(SyntaxTree node)
{
    int i;
    for (i = 0; i < MAXCHILDREN; i++)
        free(node->child[i]);
```

```
    free(node->sibling);
    free(node);
}
```

当文法规则中遇到左递归的情况时，则统一用下面的代码处理。注意，左递归的节点存储在 sibiling 中，而不是 child 中。

```
expression_list     :   expression_list COMMA expression
                        {
                            YYSTYPE t = $1;
                            if(t != NULL)
                            {
                                while(t->sibling != NULL)
                                    t = t->sibling;
                                t->sibling = $3;
                                $$ = $1;
                            }
                            else
                                $$ = $3;
                        }
                    |   expression
                        {$$ = $1;}
                    ;
```

文法规则相结合的操作构造完后，最后是辅助过程部分，包括 yyerror、yylex 和 parse 的定义。

yyerror 是出错处理函数。msg 是 Yacc 在遇到错误时产生的错误信息。

```
int yyerror(const char *msg)
{
    extern char *yytext;
    if(msg = NULL)
    {
        msg = "grammar";
    }
    fprintf(stderr, "[Syntax error]: Error in line %d error near '%s'\n
", yylineno, yytext);
    errFlag = 1;
    return 0;
}
```

yylex 在上面已经提到已经被定义为 getToken。

parse 过程是由主程序调用，它将调用 Yacc 定义的分析过程 yyparse 并且返回保存的语法树。

4 语义分析

4.1 语义分析的目的

语义分析阶段计算编译过程中所需的附加信息，包括构造符号表、记录声明中建立的名字的含义、在表达式和语句中进行类型推断和类型检查以及在语言的类型规则作用域内判断它们的正确性。语义分析是为了分析一段语法上正确的源代码可能包括的逻辑错误。

4.2 符号表的数据结构

在编译过程中，编译器使用符号表记录源程序中各种名字的特性信息。名字包括常量、变量、函数、过程等的标识符，特性信息包括该名字对应的类型、地址、字面量等。

4.2.1 散列表

在我们的编译器中，符号表的核心数据结构采用散列表。散列表的优势在于可以让插入、查找和删除的平均时间复杂度达到 O(1)。

对于符号表，我们将散列表的大小设定为 211（SIZE），我们采用一个简单的 hash 函数，相加所有字符的值，并且重复使用上一次相加后得到的常数，用取模的方式来防止溢出。对于哈希冲突，散列表中的每一个 bucket 采用链表的数据结构。

hash 函数：

```
static int hash(char* key)
{
    int temp = 0;
    int i = 0;
    while (key[i] != '\0') {
        temp = ((temp << SHIFT) + key[i]) % SIZE;
        ++i;
    }
    return temp;
}
```

散列表的 key 对应源程序中的一系列名字，散列表的 value（即 bucket）对应该名字的特性信息。我们将 bucket 分为两类：①一类是过程名字，例如 `function a` 中的 a，`procedure b` 中的 b；②一类是常量、变量、参数的名字，例如 `var a` 中的 a，`const b` 中的 b。

对于非过程名字的 bucket，存储的特性信息有名字标识符、类型、是否可以修改值、数值。对于特殊类型参数需要额外存放信息，例如对于数组，需要额外存放上下界信息。

Parameter：

```
typedef struct ParameterNode {
    char* name;
    ExpType type; /* 类型 */
    int modification;
    union {
        int intValue;
        char charValue;
        char* stringValue;
        float realValue;
    } value[NUMBER];
    struct ParameterNode* next;
} * Parameter;
```

对于过程名字的 bucket，存储的特性信息包括名字标识符、参数个数、参数说明、返回类型说明、函数体语法树入口地址。参数说明采用上述数据结构，并按照声明顺序生成链表结构，存放在过程的 bucket 中。

Function：

```
typedef struct FunctionNode {
    int paramCount; /* 参数个数 */
    Parameter paramAddr; /* 参数开始地址，以链表的形式存储 */
    ExpType returnType; /* 返回值类型 */
    SyntaxTree funcOpAddr; /* routine-body */
} * Function;
```

根据 type 区分 bucket 的种类，在此基础上加上名字对应在源代码中的行数，整体散列表的 bucket 设计如下：

```
typedef struct Entry {
    char* name; /* id name or function name */
    int lineno; /* line num in the code */
    BucketType type;  /* 区分是变量定义还是函数定义 */
    Function function;  /* 函数结构 */
    Parameter parameter; /* 参数结构 */
    int baseAddr; /* base address */
    int offset; /* offset */
    struct Entry* next;
} * Bucket;
```

4.2.2 支持多层作用域

pascal 编译器需要支持多层作用域。Program 是一层作用域。当源代码中出现了 function 或者 procedure 的定义，就意味着进入了一个新的语句块，也就是一个新的作用域。不同作用域之间需要区分变量的定义，解决变量同名的冲突。

我们的编译器采用为每一层作用域创建一个独立的符号表。每一个符号表有对应的作用域名、该作用域内参数的散列表。不同作用域的符号表之间通过指针相连，parent 指向直接上一层作用域，child 指向直接下一层作用域，next 指向同层作用域。

符号表数据结构：

```
typedef struct SymtabNode {
    char* nameSpace; /* 作用域名 */
    Bucket table[SIZE]; /* 散列表 */
    int baseAddr; /* 基地址 */
    int offset; /* 偏移 */
    struct SymtabNode* parent;
    struct SymtabNode* child;
    struct SymtabNode* next;
} * Symtab;
```

4.2.3 符号表操作

在词法分析、语法分析生成的语法树的基础上，遍历语法树的节点，构造符号表。

- 为当前作用域创建空符号表：Symtab SymtabCreate(**char*** nameSpace, Symtab parent);
- 从当前符号表出发，自底层向上层，搜索名字：Bucket SymtabSearch(Symtab symtab, **char*** name);
- 在当前符号表中搜索名字：Bucket CurrentSymtabSearch(Symtab symtab, **char*** name);
- 将一个 bucket 插入符号表：**void** SymtabInsert(Symtab symtab, Bucket bucket);

- 初始化一个参数桶：Parameter createParamNode(SyntaxTree keyNode, SyntaxTree valueNode);
- 创建一个参数桶：Bucket createParamBucket(SyntaxTree keyNode, SyntaxTree valueNode);
- 创建一个函数桶：Bucket createFuncBucket(SyntaxTree keyNode, SyntaxTree resNode, SyntaxTree opNode);
- 初始化函数参数：**void** setFunctionParameter(Bucket funcBucket, SyntaxTree argsKeyNode, SyntaxTree argsValueNode);

4.3 属性分析和语义分析

在对源程序有无逻辑错误的属性分析和语义分析时，自底向上遍历语法树，因此整体函数采用递归逻辑。通过 switch-case 判断语法树节点类型，递归调用语义分析驱动程序，触发逻辑进行具体的语义分析。语义分析检查的错误类型有：

1) 变量在使用时未经定义。

2) 函数在调用时未经定义。

3) 一个作用域内变量出现重复定义。

4) 函数出现重复定义或者函数与变量重名。

5) 赋值号两边的表达式类型不匹配。

6) 操作数类型不匹配或者操作数类型与操作符不匹配。

7) 函数调用实参和形参数目不匹配或者类型不匹配。

……

语义分析例子如下：

①在处理变量定义时，需要判断是否出现变量重名的错误。根据变量名在本层作用域内搜索，如果搜索到一致的变量名，则报错；如果变量未出错，则为其创建 bucket，加入到本层符号表的散列表中。通过同层语法树节点获得变量名与变量类型。定义为相同类型的变量在语法树中是 sibling 的关系，需要遍历变量名节点的 sibling。

```
case VarDecl:
            nodeTemp = treeNode->child[0];
            do {
                temp = CurrentSymtabSearch(symtab, nodeTemp->attr.name)
;
                if (temp != NULL) {
                    printError(nodeTemp, temp, Redefinition);
                    break;
                }
                bucket = createParamBucket(nodeTemp, treeNode->child[1]
);
                SymtabInsert(symtab, bucket);
                nodeTemp = nodeTemp->sibling;
            } while (nodeTemp != NULL);
            buildSymtab(treeNode->sibling, symtab);
            break;
```

②在处理函数定义时，判断函数是否出现同名错误。如果函数定义未出错，创建好函数定义的 bucket，并将函数参数信息加入到函数 bucket 中。将这个 bucket 加入到本层符号表的散列表中。因为函数定义意味着出现了一个新的作用域。为这个新作用域创建一个空的符号表，将函数参数加入到新符号表中。然后在新符号表上遍历语法树节点，进行之后的语义分析。旧作用域与新作用域的关系时父子关系，新作用域同时会加在旧作用域的子作用域的链表之中，具体体现是符号表数据结构中的 parent\child\next 指针。

```
case ProcedureDecl:
            temp = SymtabSearch(symtab, treeNode->attr.name);
            if (temp != NULL)
            {
                printError(treeNode, temp, Redefinition);
                break;
            }
            bucket = createFuncBucket(treeNode, treeNode->child[1], tre
eNode->child[3]);
            nodeTemp = treeNode->child[0];
            while (nodeTemp != NULL)
            {
                setFunctionParameter(bucket, nodeTemp->child[0], nodeTe
mp->child[1]);
                nodeTemp = nodeTemp->sibling;
            }
            SymtabInsert(symtab, bucket);
            symtab = SymtabCreate(treeNode->attr.name, symtab);
            buildSymtab(treeNode->child[0], symtab);
            buildSymtab(treeNode->child[2], symtab);
            buildSymtab(treeNode->child[3], symtab);
            break;
```

③在处理赋值语句时，如果是 ID，判断 ID 是否存在；判断等号两边类型是否匹配。

```
case AssignStmt:
            type1 = semanticAnalysis(treeNode->child[0], symtab, 1);
            if (treeNode->child[2] != NULL) {
                /* array type */
                type2 = semanticAnalysis(treeNode->child[1], symtab, 0)
;
                if (type2 != IntExpType) {
                    printError(treeNode, NULL, TypeMismatch);
                }
                type2 = semanticAnalysis(treeNode->child[2], symtab, 0)
;
            } else {
                type2 = semanticAnalysis(treeNode->child[1], symtab, 0)
;
            }
            if (type1 != type2) {
                printError(treeNode, NULL, TypeMismatch);
            }
            type = type1;
```

```
            break;
```

④在处理函数调用时，判断函数是否存在，判断实参形参是否匹配。

```
void paramCheck(Bucket funcBucket, SyntaxTree paramNode)
{
    SyntaxTree paramTemp = paramNode;
    if (funcBucket == NULL)
        return;
    int count = 0;
    while (paramTemp != NULL) {
        count++;
        paramTemp = paramTemp->sibling;
    }
    if (count != funcBucket->function->paramCount) {
        printError(NULL, funcBucket, ArgumentNumber);
        return;
    }
    paramTemp = paramNode;
    while (paramTemp != NULL) {
        paramTemp = paramTemp->sibling;
    }
}
```

5 优化考虑

在语义分析阶段，分析运算左右两边是否是常量，如果两边都是常量，直接完成常量的运算，用新的节点替换旧节点。例如：

```
write(100+200);
```

100+200 的运算会在语义分析时完成。生成的中间代码是

```
var0 = #300
BEGIN_ARGS
ARG var0
CALL WRITE
```

6 中间代码生成

因为代码的生成是一个比较复杂的过程，虽然我们使用的 Pascal 语言的定义相对来说有所简化，但还是有比较多需要处理的部分，想要直接生成目标代码的难度比较大。所以一般编译器都会将代码生成分成若干个子阶段，其中就包含了中间代码。

在编译原理课程学习过程中，我们主要了解到的两种比较经典的中间代码，分别是三地址码(three address code)和 P-code，我们这次实现的编译器采用的中间代码就是三地址码的形式。

6.1 数据结构：四元组链表

一条三地址码包含至多三个地址和一个操作方法，也就是说，可以用四元组的数据结构来表示一条三地址码，定义如下数据结构，QuadOpKind 表示操作方法，AddrKind 表示地址的类型，Address 结构体表示地址，其中包含种类和内容。Quad 结构体表示一个四元组，也就是代表一条三地址码的存储结构，其中包含操作方法和三个地址，以及用于链接的 next 指针。

```
typedef enum {
    constassign, assign, plus, sub, minus, mul, divide, mod,
    andi, ori, noti,
    lt, eq, gt, le, ge, ne,
    array1, array2,
    if_goto, if_f_goto, lab, gotolab,
    funcf, entryf, ret, retwithvalue, ret0, beginargs, argparam, call,
    param, valuefromcall,
    wt, wtln, rd
} QuadOpKind;

typedef enum { Empty, IntConst, RealConst, String }AddrKind;

typedef struct {
    AddrKind kind;
    union {
        int intVal;
        float realVal;
        char* name;
    } contents;

} Address;

typedef struct Quad{
    QuadOpKind op;
```

```
    Address addr1, addr2, addr3;
    struct Quad * next;
} * pQuad;

extern pQuad QuadListHead;
extern pQuad QuadListEnd;
```

最后，为了组织所有三地址码，采取了链表来连接所有四元组，因此定义了指向链表头的指针 QuadListHead 和指向链表尾部的指针 QuadListEnd，在 initQuadList 函数中完成链表的初始化。

6.2 四元组链表的初始化和插入操作

定义好了四元组的数据结构，并且定义了指向链表头的指针 QuadListHead 和指向链表尾部的指针 QuadListEnd，可以看到四元组以链表的形式连接和存储，这样做的好处是不用像数组那样固定一个很大的空间，并且插入到链表尾部的时间复杂度是 O(1)常数级，在实现上虽然相比数组多了指针，但是可实现性还是非常好的。

在 quadruple.h 中，声明了 initQuadList 函数，这个函数是用来对四元组链表进行初始化的。初始化的操作可以描述为：为链表头结点申请一块内存，初始化头结点的 next 指针为 NULL，令尾指针指向头结点。如下为 initQuadList 函数的定义。

```
void initQuadList(){
    QuadListHead = (pQuad)malloc(sizeof(struct Quad));
    QuadListHead->next = NULL;
    QuadListEnd = QuadListHead;
}
```

初始化过的链表就可以通过对尾指针的操作来进行插入操作，比如现在要申请一个节点 q 的内存，进行赋值后将其插入到链表尾部，就可以通过如下语句来实现：

```
pQuad q = (pQuad)malloc(sizeof(struct Quad));
/* 对 q 的一些赋值操作 */
q->next = NULL;
QuadListEnd->next = q;
QuadListEnd = QuadListEnd->next;
```

在生成四元组链表的过程中，插入节点的操作是经常用到的。

6.3 遍历语法树生成四元组链表

经历了前面的词法文法分析，已经得到了一个分析的语法树，语法树的结构定义在 global.h 头文件中。定义好了存储结构，即四元组链表，就可以开始遍历语法树生成存储中间代码的四元组链表。

首先，可以从课件中看到遍历语法树生成中间代码的伪代码如下：

```
Procedure gencode (T: treenode);
    Begin
      If T is not nil then
        Generate code to prepare for code of left child of T;
        Gencode(left child of T);
        Generate code to prepare for code of right child of T;
        Gencode(right child of T);
        Generate code to implement the action of T;
    End;
```

我们遍历语法树生成四元组链表的递归函数为 genCode 函数，声明如下：

```
void genCode(SyntaxTree node, char* resName);
```

其中 node 为当前语法树节点，resName 为在某些时候需要传递的结果变量名。

在 genCode 函数中，根据当前节点的类型，有不同的处理方式，这是根据我们前面词法文法分析的规则定义来决定的，可以在我们的.y 文件中看到具体的文法和语法树的生成规律。

首先，判断当前节点的种类 switch(node->nodeKind)，根据 global.h 中对 nodeKind 的定义，有四种 case: StatementNode, ExpressionNode, DeclareNode, TypeNode。

根据这四种 case，每种 case 内部都有进一步划分种类，比如在语法树中，StatementNode 类型的节点一般代表一个语句，往往是一行或者是像 if, while, for, switch 语句这样视为一个整体的多行语句。而 ExpressionNode 则是更小的单元，往往代表 StatementNode 语句的一个部分，比如 IdExp 代表一个 ID 节点。DeclareNode 类型的节点是声明类型的，比如函数声明、变量声明等等，TypeNode 类型则对应了声明的类型，在生成中间代码的过程中几乎可以忽略。

因此 genCode 函数最外圈是一个如下的对节点类型进行判断的结构，如下为 genCode 函数的实现，因为函数的代码比较长，所以只展示了外层的选择以及 IfStmt 的具体插入链表的操作，以 IfStmt 节点的具体操作为例来描述 genCode 的实现过程，其他类型节点的具体操作省略，具体实现细节在 quadruple.c 的 genCode 函数定义中可以看到：

```
void genCode(SyntaxTree node, char* resName) {
    if (node == NULL) {
        return;
    }
    switch (node->nodeKind) {
        case StatementNode: {
            switch (node->kind.stmt) {
                case LabelStmt: {
                    // 省略
                    break;
                }
```

```
                case AssignStmt: {
                    // 省略
                    break;
                }
                case GotoStmt: {
                    // 省略
                    break;
                }
                case IfStmt: {
                    //IF expression THEN stmt else_clause, child0:expre
ssion, child1:stmt, child2:else_clause
                    //IF expression THEN stmt
                    //生成 expression 的 code
                    char* tempName1;
                    tempName1 = getTempName();
                    genCode(node->child[0], tempName1);
                    //生成 label
                    char* tempLabel1;
                    tempLabel1 = getTempLabel();
                    // if_false tempName1 goto L1: if_false addr1 goto
addr2
                    pQuad q1 = (pQuad)malloc(sizeof(struct Quad));
                    q1->op = if_f_goto;
                    q1->addr1.kind = String;
                    q1->addr1.contents.name = tempName1;
                    q1->addr2.kind = String;
                    q1->addr2.contents.name = tempLabel1;
                    q1->next = NULL;
                    QuadListEnd->next = q1;
                    QuadListEnd = QuadListEnd->next;
                    //生成 stmt 的 code
                    genCode(node->child[1], resName);
                    //label tempLabel1: label addr3
                    pQuad q2 = (pQuad)malloc(sizeof(struct Quad));
                    q2->op = lab;
                    q2->addr3.kind = String;
                    q2->addr3.contents.name = tempLabel1;
                    q2->next = NULL;
                    QuadListEnd->next = q2;
                    QuadListEnd = QuadListEnd->next;
                    if (node->child[2] != NULL) {
                        //有 else，相当于 if t1 goto L2; stmt2; L2
                        // if t1 goto label: if addr1 goto addr2
                        char* tempLabel2;
```

```
                tempLabel2 = getTempLabel();
                pQuad q3 = (pQuad)malloc(sizeof(struct Quad));
                q3->op = if_goto;
                q3->addr1.kind = String;
                q3->addr1.contents.name = tempName1;
                q3->addr2.kind = String;
                q3->addr2.contents.name = tempLabel2;
                q3->next = NULL;
                QuadListEnd->next = q3;
                QuadListEnd = QuadListEnd->next;
                //生成 stmt2 的 code
                genCode(node->child[2], resName);
                //label tempLabel2: label addr3
                pQuad q4 = (pQuad)malloc(sizeof(struct Quad));
                q4->op = lab;
                q4->addr3.kind = String;
                q4->addr3.contents.name = tempLabel2;
                q4->next = NULL;
                QuadListEnd->next = q4;
                QuadListEnd = QuadListEnd->next;
            }
            break;
        }
        case RepeatStmt: {
            // 省略
            break;
        }
        case WhileStmt: {
            // 省略
            break;
        }
        case ForStmt: {
            // 省略
            break;
        }
        case CaseStmt: {
            // 省略
            break;
        }
        case ProcIdStmt: {
            // 省略
            break;
        }
        case ProcSysStmt: {
```

```
                // 省略
                break;
            }
            case FuncIdStmt: {
                // 省略
                break;
            }
            case FuncSysStmt: {
                // 省略
                break;
            }
            default:
                break;
        }
        //stmt 通过 sibling 连接
        genCode(node->sibling, resName);
        break;
    }
    case ExpressionNode: {
        switch (node->kind.exp) {
            case IdExp: {
                // 省略
                break;
            }
            case ConstExp: {
                // 省略
                break;
            }
            case OpExp: {
                // 省略
                break;
            }
            case CaseExp: {
                // 省略
                break;
            }
            default:
                break;
        }
        genCode(node->sibling, resName);
        break;
    }
    case DeclareNode: {
        switch (node->kind.decl) {
```

```
                case ProgramDecl:{
                    // 省略
                    break;
                }
                case RoutineDecl:{
                    // 省略
                    break;
                }
                case RoutineheadDecl:{
                    // 省略
                    break;
                }
                case FunctionDecl:{
                    // 省略
                    break;
                }
                case ProcedureDecl:{
                    // 省略
                    break;
                }
                case ConstDecl:{
                    // 省略
                    break;
                }
                default:{
                    // 省略
                    break;
                }
            }
            break;
        }
        case TypeNode: {
            // 省略
            break;
        }
        default:
            break;
    }
}
```

其中 getTempName 函数可以得到一个新生成的临时变量名 var<number>，getTempLabel 函数可以得到一个新生成的 Label 名 L<number>。

因为 genCode 整体代码过长，因此这里整理了对于各个不同类型节点的处理方法列表，

具体代码实现在 quadruple.c 中查看。

node->nodeKind	node->kind.stmt / node->kind.exp / node->kind.decl / node->kind.type	对应语法	生成三地址码
StatementNode	LabelStmt	INTEGER COLON non_label_stmt	genCode(node->child[0], resName)
	AssignStmt	ID ASSIGN expression	genCode(node->child[1], t1) ID = t1;
		ID LB expression RB ASSIGN expression	genCode(node->child[1], t1) genCode(node->child[2], t2) ID[t1] = t2
		ID DOT ID ASSIGN expression	略
	GotoStmt	GOTO INTEGER	GOTO node->attr.intValue
	IfStmt	IF expression THEN stmt	genCode(node->child[0], t1) IF_FALSE t1 GOTO L1 genCode(node->child[1], resName) LABEL L1
		IF expression THEN stmt else_clause	

			genCode(node->child[0], t1) IF_FALSE t1 GOTO L1 genCode(node->child[1], resName) LABEL L1 IF t1 GOTO L2 genCode(node->child[2], resName) LABEL L2
	RepeatStmt	略	略
	WhileStmt	WHILE expression DO stmt	LABEL L1； genCode(node->child[0], t1) IF_FALSE t1 GOTO L2 genCode(node->child[1], resName) GOTO L1; LABEL L2;
	ForStmt	FOR ID ASSIGN expression TO expression DO stmt	genCode(node->child[1], t1) ID = t1; LABEL L1;

			genCode(node->child [2], t2) t3 = ID > t2 IF t3 GOTO L2; genCode(node->child [3], resName) ID = ID + 1 GOTO L1; LABEL L2;
		FOR ID ASSIGN expression DOWNTO expression DO stmt	genCode(node->child [1], t1) ID = t1; LABEL L1; genCode(node->child [2], t2) t3 = ID < t2 IF t3 GOTO L2; genCode(node->child [3], resName) ID = ID - 1 GOTO L1; LABEL L2;

	CaseStmt	CASE expression OF case_expr_list END	genCode(node->child [0], t0) t1 = t0== case1.value; IF_FALSE t1 GOTO L1; genCode(stmt1, resName) LABEL L1 t2 = t0== case2.value; IF_FALSE t2 GOTO L2; genCode(stmt2, resName) LABEL L2 ... IF_FALSE tN GOTO LN; genCode(stmtN, resName) LABEL LN

	ProcIdStmt,	略	略
	ProcSysStmt	READ LP factor RP	BEGIN_ARGS ARG node->child[0]->attr. name CALL READ
		WRITE	CALL WRITE
		WRITE LP expression_list RP	genCode(node->child [0],t1); BEGIN_ARGS ARG t1 CALL WRITE
		WRITELN	CALL WRITELN
		WRITELN LP expression_list RP	genCode(node->child [0],t1); BEGIN_ARGS ARG t1 CALL WRITELN
	FuncIdStmt	ID	BEGIN_ARGS resName = CALL ID

		ID LP args_list RP	BEGIN_ARGS genCode(arg1, t1) ARG t1 ... genCode(argN, tN) ARG tN resName = CALL ID
	FuncSysStmt	Func LP args_list RP	BEGIN_ARGS genCode(arg1, t1) ARG t1 ... genCode(argN, tN) ARG tN resName = CALL ID
	IdExp	TOKEN_ID	resName = ID

ExpressionNode	ConstExp	INTEGER/REAL/CHAR/STRING/TRUE/FALSE/MAXINT	resName = constValue
	OpExp	expression op expr / expr op term / term op factor /	genCode(node->child[0], t1) genCode(node->child[1], t2) resName = t1 op t2
		op factor	genCode(node->child[0], t1) resName = op t1
	CaseExp	略	略
Declaration Node	ProgramDecl	program_head routine DOT	genCode(node->child[0], resName) FUNCTION main : genCode(node->child[1], resName) RETURN #0 genCode(node->sibling, resName)
	Programhead Decl	略	略
	RoutineDecl	略	略
	RoutineheadDecl	const_part type_part var_part routine_part	genCode(node->child[0], resName) genCode(node->child[3], resName) genCode(node->sibling, resName)

	FunctionDecl	function_head SEMI sub_routine SEMI	genCode(node->child [2], resName); //routine_head FUNCTION f : PARAM id1 ... PARAM idN genCode(node->child [3], resName); //routine_body RETURN tempf
	Functionhead Decl	略	略
	ProcedureDecl	procedure_head SEMI sub_routine SEMI	genCode(node->child [2], resName); //routine_head FUNCTION f : PARAM id1 ... PARAM idN genCode(node->child [3], resName); //routine_body RETURN #0

	ProcedureheadDecl	略	略
	ConstDecl	ID EQUAL const_value SEMI	ID = const_value genCode(node->sibling, resName);
	VarDecl	略	略
	TypeDecl	略	略
	VarParaDecl	略	略
TypeNode	略	略	略

以上表格就是整理得到的对于各个类型的节点的操作方法。

最后，在 main.c 里面 initQuadList 后，调用 genCode 函数，传入语法树的根节点，就可以完成遍历语法树生成四元组链表的过程，最后生成的链表，QuadListHead 指向链表头，头结点是一个空节点，从头结点的 next 开始是第一个四元组元素。

6.4 将四元组打印成中间代码

遍历语法树生成了四元组链表后，需要将存储在链表中的四元组打印成可视化的中间代码。

因此定义了 printQuad 函数和 printIRcode 函数。

printQuad 函数将一个四元组打印成对应的三地址码。首先根据地址的 kind 将地址的值转换成字符串 str1, str2, str3，然后是根据操作方法 op 来打印成三地址码。

printQuad 函数中根据 op 打印三地址码的代码部分如下：

```
//print code
    //fprintf(ircode, "%d: ", index);
    switch (q->op)
    {
        case assign:
            fprintf(ircode, "%s = %s", str3, str1);
            break;
        case constassign:
            fprintf(ircode, "CONST %s = %s", str3, str1);
            break;
        case plus:
            fprintf(ircode, "%s = %s + %s", str3, str1, str2);
            break;
        case sub:
```

```
            fprintf(ircode, "%s = %s - %s", str3, str1, str2);
            break;
        case minus:
            fprintf(ircode, "%s = minus %s", str3, str1);
            break;
        case mul:
            fprintf(ircode, "%s = %s * %s", str3, str1, str2);
            break;
        case divide:
            fprintf(ircode, "%s = %s / %s", str3, str1, str2);
            break;
        case mod:
            fprintf(ircode, "%s = %s mod %s", str3, str1, str2);
            break;
        case andi:
            fprintf(ircode, "%s = %s & %s", str3, str1, str2);
            break;
        case ori:
            fprintf(ircode, "%s = %s | %s", str3, str1, str2);
            break;
        case noti:
            fprintf(ircode, "%s = not %s", str3, str1);
            break;
        case lt:
            fprintf(ircode, "%s =  %s < %s", str3, str1, str2);
            break;
        case eq:
            fprintf(ircode, "%s = %s == %s", str3, str1, str2);
            break;
        case gt:
            fprintf(ircode, "%s = %s > %s", str3, str1, str2);
            break;
        case le:
            fprintf(ircode, "%s = %s <= %s", str3, str1, str2);
            break;
        case ge:
            fprintf(ircode, "%s = %s >= %s", str3, str1, str2);
            break;
        case ne:
            fprintf(ircode, "%s = %s != %s", str3, str1, str2);
            break;
        case array1:
            fprintf(ircode, "%s[%s] = %s", str3, str1, str2);
            break;
```

```
        case array2:
            fprintf(ircode, "%s = %s[%s]", str3, str1, str2);
            break;
        case if_goto:
            fprintf(ircode, "IF %s GOTO %s", str1, str2);
            break;
        case if_f_goto:
            fprintf(ircode, "IF_FALSE %s GOTO %s", str1, str2);
            break;
        case lab:
            fprintf(ircode, "LABEL %s", str3);
            break;
        case gotolab:
            fprintf(ircode, "GOTO %s", str3);
            break;
        case entryf:
            fprintf(ircode, "ENTRY %s", str3);
            break;
        case funcf:
            fprintf(ircode, "FUNCTION %s :", str3);
            break;
        case ret:
            fprintf(ircode, "RETURN");
            break;
        case ret0:
            fprintf(ircode, "RETURN #0");
            break;
        case retwithvalue:
            fprintf(ircode, "RETURN %s", str3);
            break;
        case beginargs:
            fprintf(ircode, "BEGIN_ARGS");
            break;
        case param:
            fprintf(ircode, "PARAM %s", str3);
            break;
        case argparam:
            fprintf(ircode, "ARG %s", str3);
            break;
        case call:
            fprintf(ircode, "CALL %s", str3);
            break;
        case valuefromcall:
            fprintf(ircode, "%s = CALL %s", str3, str1);
```

```
            break;
        case wt:
            fprintf(ircode, "BEGIN_ARGS\n");
            fprintf(ircode, "ARG %s\n", str3);
            fprintf(ircode, "CALL WRITE");
            //fprintf(ircode, "WRITE %s", str3);
            break;
        case wtln:
            fprintf(ircode, "BEGIN_ARGS\n");
            fprintf(ircode, "ARG %s\n", str3);
            fprintf(ircode, "CALL WRITELN");
            //fprintf(ircode, "WRITELN %s", str3);
            break;
        case rd:
            fprintf(ircode, "BEGIN_ARGS\n");
            fprintf(ircode, "ARG %s\n", str3);
            fprintf(ircode, "CALL READ");
            //fprintf(ircode, "READ %s", str3);
            break;
        default:
            break;
    }
    fprintf(ircode, "\n");
}
```

printQuad 函数实现了将一个四元组元素转换成三地址码的功能，那么将四元组链表转换成三地址码就是遍历链表，对于每个四元组都调用一次 printQuad 函数，就能将四元组链表全部转换成三地址码写入 ircode 文件。

如下为实现了遍历四元组链表打印出三地址码的 printIRcode 函数的实现代码：

```
void printIRcode(){
    int index = 0;
    pQuad tmp = NULL;
    tmp = QuadListHead->next;

    while(tmp!=NULL){
        printQuad(index, tmp);
        tmp = tmp->next;
        index++;
    }
    return;
}
```

在 main.c 中完成四元组链表的生成后，调用 printIRcode 函数，即可将四元组链表打印出的三地址码写入 ircode 文件。

7 目标代码生成

7.1 实现原理

读取中间代码文件，将中间代码翻译为 MIPS 指令序列，并在 SPIM Simulator 上运行。中间代码在上一章节已说明，我们采用三地址码的格式。一条中间代码对应一条或者多条目标代码。

7.2 指令选择

我们采用线形 IR，因此我们使用模式匹配的翻译方法。中间代码与 MIPS32 指令对应的一个示例。

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 + #k	addi reg(x),reg(y),k
x = y + z	add reg(x),reg(y),reg(z)
x = y - #k	addi reg(x),reg(y),-k
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(x),reg(y),reg(z) mflo reg(x)
x = y % z	div reg(x),reg(y),reg(z) mfhi reg(x)
x = y == z	seq reg(x),reg(y),reg(z)
x = y != z	sne reg(x),reg(y),reg(z)
x = y >= z	sge reg(x),reg(y),reg(z)
x = y <= z	sle reg(x),reg(y),reg(z)
IF t GOTO L0	beq reg(t),$zero,L0
IF_FALSE t0 GOTO L0	bne reg(t),$zero,L0

CALL WRITELN(READ/WRITE)	addi $sp,$sp,-8 sw $a0,0($sp) sw $ra,4($sp) jal WRITELN lw $a0,0($sp) lw $ra,4($sp) move reg(0),$v0 addi $sp,$sp,8"
CALL x	寄存器压栈 jal x 寄存器读栈
GOTO L0	j L0
FUNCTION x :	x:
BEGIN_ARGS	无
PARAM t	move reg(t),argument register
ARG t	move getTempRegister(),reg(argument) move reg(argument),reg(t)
RETURN #0	move $v0,$zero jr $ra
RETURN x	move $v0,reg(x) jr $ra

代码示例：

```
    if (line[0] == "IF_FALSE") {
        return "\tbeq " + getRegister(line[1]) + ",$zero," + line[3];
    }
    if (line[0] == "IF") {
        return "\tbne " + getRegister(line[1]) + ",$zero," + line[3];
    }
```

```
if (line[3] == "+") {
```

```
            if (line[4][0] == '#') {
                return "\taddi " + getRegister(line[0]) + "," +
getRegister(line[2]) + "," + line[4].substr(1);
            } else {
                return "\tadd " + getRegister(line[0]) + "," +
getRegister(line[2]) + "," + getRegister(line[4]);
            }
        }
```

7.3 寄存器分配

寄存器的使用和指派没有完全遵循 MIPS32 的约定。$a0-$a3 存放函数的四个参数，$v0 存放函数结果，$t0-$t9 $s0-s7 存放变量，统一由函数调用者负责保存。$ra 存放返回地址。$sp 存放栈顶置针。$zero 存放立即数 0。

使用一个数组来记录寄存器是否闲置，如果寄存器在当前状态下不是闲置，而且属于需要保护的寄存器阵营就需要保存在内存中。

```
string emitCallBeforeCode()
{
    string s = "";
    int memory = 0;
    for (int i = 0; i < tempreg.size(); i++) {
        s += "\tsw $" + regs[tempreg[i]] + "," + to_string(memory) +
"($sp)\n";
        memory += 4;
    }
    s += "\tsw $ra," + to_string(memory) + "($sp)\n";
    memory += 4;
    for (int i = 0; i < REG_SIZE; i++) {
        if (regValid[i] == 0 && find(tempreg.begin(), tempreg.end(),
i) == tempreg.end() && find(variable2.begin(), variable2.end(), i)
== variable2.end()) {
            s += "\tsw $" + regs[i] + "," + to_string(memory) +
"($sp)\n";
            memory += 4;
        }
    }
    s = "\taddi $sp,$sp,-" + to_string(memory) + "\n" + s;
    return s;
}
```

7.3.1 支持作用域

整段中间代码按照不同的函数被分为一个个基本块。在语义分析阶段，遍历语法树的时候，更改每一个变量的名字，在名字上加上后缀“-”+作用域名字。因此得到一个变量名对应一个唯一的作用域。在中间代码生成阶段，在变量名前面加上“temp”的标识符。在此阶段，根据前缀是否有“temp”标识符获取判断是否是变量。如果是变量，考虑两种情况。一种是跨多个作用域的变量，一种是只在一个作用域内的变量。只在一个作用域内的变量就是函数的参数，会出现在中间代码“PARAM x”的 x 位置。排除只在一个作用域内的变量，为跨多个作用域的变量分配好空闲的寄存器。在后续的函数调用阶段，这些寄存器不参与栈存取，以达到跨域的效果。

```
void assignRegister()
{
    for (int i = 0; i < line.size(); i++) {
        if (line[i][0] == "PARAM") {
            params.push_back(line[i][1]);
            continue;
        }
        for (int j = 0; j < line[i].size(); j++) {
            if (line[i][j].substr(0, 4) == "temp" &&
find(params.begin(), params.end(), line[i][j]) == params.end()) {
                if (table.find(line[i][j]) == table.end()) {
                    for (int t = 0; t < REG_SIZE; t++) {
                        if (regValid[t] == 1) {
                            variable2.push_back(t);
                            table.insert(make_pair(line[i][j], t));
                            regValid[t] = 0;
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

7.3.2 寄存器换入换出

在每一个基本块内，为新出现的临时变量分配好寄存器。根据中间代码的特性，当新出现一个临时变量时，前面使用过的临时变量的寄存器就可以被释放。

```
for (int i = 0; i < keys.size(); i++) {
        string key = keys[i];
```

```
        if (key.substr(0, 3) == "var" && find(variable.begin(),
variable.end(), key) == variable.end()) {
            map<string, int>::iterator it = table.find(key);
            regValid[it->second] = 1;
            table.erase(it);
        }
    }
```

在函数调用阶段，将当前使用的寄存器保存到内存中，就可以释放这个寄存器。让下一个作用域可以使用这些闲置的寄存器。

```
void releaseTempReg()
{
    for (int i = 0; i < REG_SIZE; i++) {
        if (regValid[i] == 0 && (find(tempreg.begin(), tempreg.end(),
i) != tempreg.end() || find(variable2.begin(), variable2.end(), i)
== variable2.end())) {
            regValid[1] = 0;
        }
    }
    tempreg.clear();
}
```

8 测试案例

8.1 错误案例

```
program proc;
var a, b, c : integer; e : real;

function max(num1, num2: integer): integer;
var result: integer;
begin
     if (num1 > num2) then
     begin
        result := num1;
        result1 := num2;
     end
     else
     begin
        result := num2;
     end
     ;
     max := result + c;
end;

begin
   a := 100;
   b := 200;
   c := max(a, b);
   d := 100;
   c := max(a);
   c := maxi(a, b);
   e := "123";
   write(c);
end.
```

```
[Unknown identifier]: Use of undeclared identifier 'result1' in line 10.
[Type mismatch]: Type mismatch in line 10.
[Unknown identifier]: Use of undeclared identifier 'd' in line 24.
[Type mismatch]: Type mismatch in line 24.
[Argument unmatch]: Illegal arguments to function call 'max' in line 18, expected 2.
[Unknown symbol]: Use of undeclared function 'maxi' in line 26.
[Type mismatch]: Type mismatch in line 26.
[Type mismatch]: Type mismatch in line 27.
-----------------------------Compile Error-------------------------------
```

```
program proc;
var a, b, c;
```

```
begin
   a := 100;
   b := 200;
   write(c);
end.
```

```
[Syntax error]: Error in line 2 error near ';'
------------------------------Compile Error-------------------------------
```

```
program proc;
var a, b, c
begin
   a := 100;
   b := 200;
   write(c)
end.
```

```
[Syntax error]: Error in line 3 error near 'begin'
[Syntax error]: Error in line 6 error near 'write'
[Syntax error]: Error in line 7 error near 'end'
------------------------------Compile Error-------------------------------
```

8.2 正确案例

8.2.1 IF.pas

源代码	语法树
program proc; var a, b, c : integer; function max(num1, num2: integer): integer; var result: integer; begin if (num1 > num2) then begin result := num1; end else begin result := num2; end ; max := result + c; end;	ProgramDecl:proc RoutineheadDecl VarDecl Id:a Id:b Id:c SimpleSysType:IntExpType FunctionDecl:max VarParaDecl Id:num1 Id:num2 SimpleSysType:IntExpType SimpleSysType:IntExpType RoutineheadDecl VarDecl

```
begin
   a := 100;
   b := 200;
   c := max(a, b);
   write(c);
end.
```

```
          Id:result

SimpleSysType:IntExpType
     If
       Oper:>
         Id:num1
         Id:num2
       Assign
         Id:result
         Id:num1
       Assign
         Id:result
         Id:num2
     Assign
       Id:max
       Oper:+
         Id:result
         Id:c
  Assign
    Id:a
    Const:100
  Assign
    Id:b
    Const:200
  Assign
    Id:c
    FuncId
      Id:a
      Id:b
  ProcSys
    Id:c
```

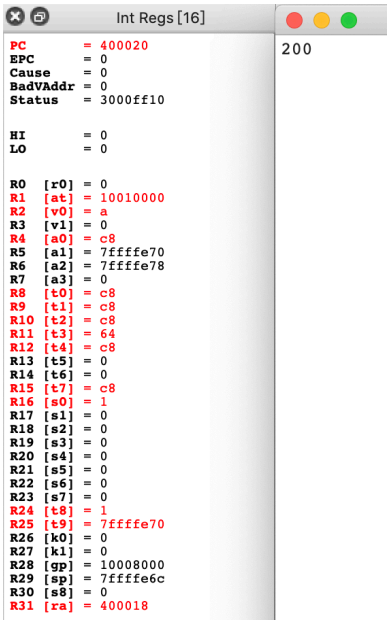
中间代码	目标代码
FUNCTION max : PARAM tempnum1-max PARAM tempnum2-max var0 = tempnum1-max > tempnum2-max IF_FALSE var0 GOTO L0 tempresult-max = tempnum1-max	.data _prompt: .asciiz "Enter an integer:" _ret: .asciiz "\n" .globl main .text READ: li $v0,4

```
LABEL L0
IF var0 GOTO L1
tempresult-max = tempnum2-
max
LABEL L1
var1 = tempresult-max +
tempc-proc
tempmax = var1
RETURN tempmax
FUNCTION main :
tempa-proc = #100
tempb-proc = #200
BEGIN_ARGS
var3 = tempa-proc
ARG var3
var4 = tempb-proc
ARG var4
var2 = CALL max
tempc-proc = var2
var5 = tempc-proc
BEGIN_ARGS
ARG var5
CALL WRITE
RETURN #0
```

```
    la $a0,_prompt
    syscall
    li $v0,5
    syscall
    jr $ra
WRITE:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
WRITELN:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
max:
    move $t5,$a0
    move $t6,$a1
    sgt $t7,$t5,$t6
    beq $t7,$zero,L0
    move $t0,$t5
L0:
    bne $t7,$zero,L1
    move $t0,$t6
L1:
    add $t7,$t0,$t1
    move $t2,$t7
    move $v0,$t2
    jr $ra
main:
    li $t3,100
    li $t4,200
    move $t7,$t3
    move $t8,$a0
    move $a0,$t7
    move $t7,$t4
    move $t9,$a1
```

```
    move $a1,$t7
    addi $sp,$sp,-24
    sw $t8,0($sp)
    sw $t9,4($sp)
    sw $ra,8($sp)
    sw $t5,12($sp)
    sw $t6,16($sp)
    sw $t7,20($sp)
    jal max
    lw $a0,0($sp)
    lw $a1,4($sp)
    lw $ra,8($sp)
    lw $t5,12($sp)
    lw $t6,16($sp)
    lw $t7,20($sp)
    addi $sp,$sp,24
    move $t7 $v0
    move $t1,$t7
    move $t7,$t1
    move $s0,$a0
    move $a0,$t7
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITE
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    move $v0,$zero
    jr $ra
```

运行截图：



8.2.2 FOR.pas

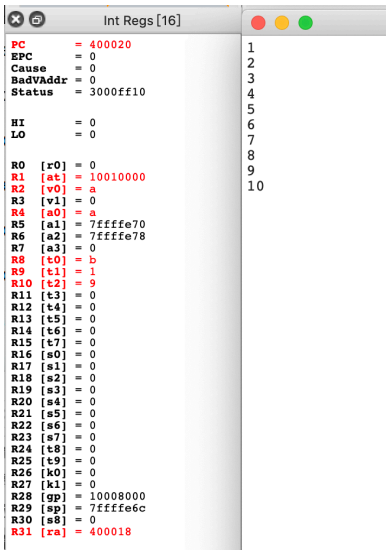
源代码	语法树
program hello; var a : integer; begin for a := 1 to 10 do begin writeln(a); end; end .	ProgramDecl:hello RoutineheadDecl VarDecl Id:a SimpleSysType:IntExpType For Id:a Const:1 Const:10 ProcSys Id:a

中间代码	目标代码
FUNCTION main : tempa-hello = #1 LABEL L0 var0 = #10 var1 = tempa-hello > var0 IF var1 GOTO L1 var2 = tempa-hello BEGIN_ARGS ARG var2 CALL WRITELN	.data _prompt: .asciiz "Enter an integer:" _ret: .asciiz "\n" .globl main .text READ: li $v0,4 la $a0,_prompt syscall

```
tempa-hello = tempa-hello +
#1
GOTO L0
LABEL L1
RETURN #0
```

```
    li $v0,5
    syscall
    jr $ra
WRITE:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
WRITELN:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
main:
    li $t0,1
L0:
    li $t1,10
    sgt $t1,$t0,$t1
    bne $t1,$zero,L1
    move $t1,$t0
    move $t2,$a0
    move $a0,$t1
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITELN
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    addi $t0,$t0,1
    j L0
L1:
    move $v0,$zero
    jr $ra
```

运行截图：



8.2.3 gcd.pas

源代码	语法树
program hello; var ans : integer; function gcd(a, b : integer) : integer; begin if b = 0 then begin gcd := a; end else begin gcd := gcd(b , a % b); end ; end ; begin ans := gcd(9 , 36) * gcd(3 , 6); writeln(ans); end .	ProgramDecl:hello RoutineheadDecl VarDecl Id:ans SimpleSysType:IntExpType FunctionDecl:gcd VarParaDecl Id:a Id:b SimpleSysType:IntExpType SimpleSysType:IntExpType RoutineheadDecl If Oper:= Id:b Const:0 Assign Id:gcd Id:a Assign Id:gcd FuncId Id:b

	Oper:mod Id:a Id:b Assign Id:ans Oper:* FuncId Const:9 Const:36 FuncId Const:3 Const:6 ProcSys Id:ans

中间代码	目标代码
FUNCTION gcd : PARAM tempa-gcd PARAM tempb-gcd var1 = #0 var0 = tempb-gcd == var1 IF_FALSE var0 GOTO L0 tempgcd = tempa-gcd LABEL L0 IF var0 GOTO L1 BEGIN_ARGS var3 = tempb-gcd ARG var3 var4 = tempa-gcd % tempb-gcd ARG var4 var2 = CALL gcd tempgcd = var2 LABEL L1 RETURN tempgcd FUNCTION main : BEGIN_ARGS var7 = #9 ARG var7 var8 = #36 ARG var8 var6 = CALL gcd	.data _prompt: .asciiz "Enter an integer:" _ret: .asciiz "\n" .globl main .text READ: li $v0,4 la $a0,_prompt syscall li $v0,5 syscall jr $ra WRITE: li $v0,1 syscall li $v0,4 la $a0,_ret syscall move $v0,$0 jr $ra WRITELN: li $v0,1 syscall li $v0,4 la $a0,_ret

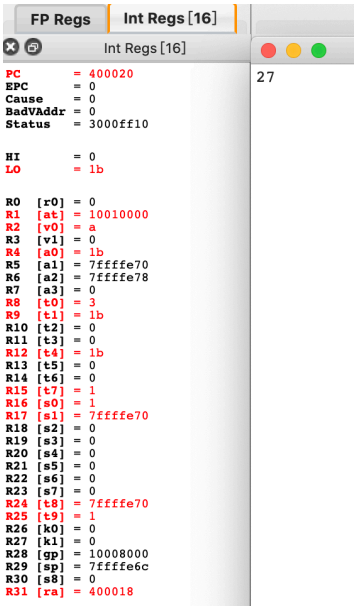
```
BEGIN_ARGS
var10 = #3
ARG var10
var11 = #6
ARG var11
var9 = CALL gcd
var5 = var6 * var9
tempans-hello = var5
var12 = tempans-hello
BEGIN_ARGS
ARG var12
CALL WRITELN
RETURN #0
```

```
    syscall
    move $v0,$0
    jr $ra
gcd:
    move $t2,$a0
    move $t3,$a1
    li $t4,0
    seq $t4,$t3,$t4
    beq $t4,$zero,L0
    move $t0,$t2
L0:
    bne $t4,$zero,L1
    move $t4,$t3
    move $t5,$a0
    move $a0,$t4
    div $t2,$t3
    mfhi $t4
    move $t6,$a1
    move $a1,$t4
    addi $sp,$sp,-24
    sw $t5,0($sp)
    sw $t6,4($sp)
    sw $ra,8($sp)
    sw $t2,12($sp)
    sw $t3,16($sp)
    sw $t4,20($sp)
    jal gcd
    lw $a0,0($sp)
    lw $a1,4($sp)
    lw $ra,8($sp)
    lw $t2,12($sp)
    lw $t3,16($sp)
    lw $t4,20($sp)
    addi $sp,$sp,24
    move $t4 $v0
    move $t0,$t4
L1:
    move $v0,$t0
    jr $ra
main:
    li $t4,9
    move $t7,$a0
    move $a0,$t4
```

```
    li $t4,36
    move $t8,$a1
    move $a1,$t4
    addi $sp,$sp,-32
    sw $t7,0($sp)
    sw $t8,4($sp)
    sw $ra,8($sp)
    sw $t2,12($sp)
    sw $t3,16($sp)
    sw $t4,20($sp)
    sw $t5,24($sp)
    sw $t6,28($sp)
    jal gcd
    lw $a0,0($sp)
    lw $a1,4($sp)
    lw $ra,8($sp)
    lw $t2,12($sp)
    lw $t3,16($sp)
    lw $t4,20($sp)
    lw $t5,24($sp)
    lw $t6,28($sp)
    addi $sp,$sp,32
    move $t4 $v0
    li $t9,3
    move $s0,$a0
    move $a0,$t9
    li $t9,6
    move $s1,$a1
    move $a1,$t9
    addi $sp,$sp,-44
    sw $s0,0($sp)
    sw $s1,4($sp)
    sw $ra,8($sp)
    sw $t2,12($sp)
    sw $t3,16($sp)
    sw $t4,20($sp)
    sw $t5,24($sp)
    sw $t6,28($sp)
    sw $t7,32($sp)
    sw $t8,36($sp)
    sw $t9,40($sp)
    jal gcd
    lw $a0,0($sp)
```

```
    lw $a1,4($sp)
    lw $ra,8($sp)
    lw $t2,12($sp)
    lw $t3,16($sp)
    lw $t4,20($sp)
    lw $t5,24($sp)
    lw $t6,28($sp)
    lw $t7,32($sp)
    lw $t8,36($sp)
    lw $t9,40($sp)
    addi $sp,$sp,44
    move $t9 $v0
    mul $t4,$t4,$t9
    move $t1,$t4
    move $t4,$t1
    move $t9,$a0
    move $a0,$t4
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITELN
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    move $v0,$zero
    jr $ra
```

运行截图：



8.2.4 same.pas

源代码	语法树
program hello; var f : integer; k : integer; function go(var b : integer; a : integer): integer; var fk : integer; t : real; begin if a = 0 then begin go := a * go(b , a - 1); end else begin go := 1; end ; b := b + go; k := k + go; end	ProgramDecl:hello RoutineheadDecl VarDecl Id:f SimpleSysType:IntExpType VarDecl Id:k SimpleSysType:IntExpType FunctionDecl:go VarParaDecl Id:b SimpleSysType:IntExpType VarParaDecl Id:a SimpleSysType:IntExpType SimpleSysType:IntExpType RoutineheadDecl VarDecl Id:fk

```
;

begin
    k := 0;
    f := go(k , 5);
    writeln(f);
    writeln(k);
end
.
```

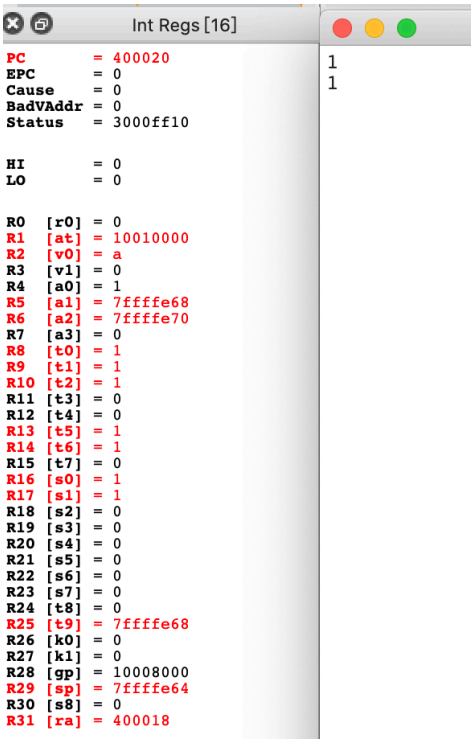
```
SimpleSysType:IntExpType
      VarDecl
        Id:t

SimpleSysType:RealExpType
     If
       Oper:=
         Id:a
         Const:0
       Assign
         Id:go
         Oper:*
           Id:a
           FuncId
             Id:b
             Oper:-
               Id:a
               Const:1
       Assign
         Id:go
         Const:1
     Assign
       Id:b
       Oper:+
         Id:b
         Id:go
     Assign
       Id:k
       Oper:+
         Id:k
         Id:go
  Assign
    Id:k
    Const:0
  Assign
    Id:f
    FuncId
      Id:k
      Const:5
  ProcSys
    Id:f
  ProcSys
    Id:k
```

中间代码	目标代码
FUNCTION go : PARAM tempb-go PARAM tempa-go var1 = #0 var0 = tempa-go == var1 IF_FALSE var0 GOTO L0 BEGIN_ARGS var4 = tempb-go ARG var4 var6 = #1 var5 = tempa-go - var6 ARG var5 var3 = CALL go var2 = tempa-go * var3 tempgo = var2 LABEL L0 IF var0 GOTO L1 tempgo = #1 LABEL L1 var7 = tempb-go + tempgo tempb-go = var7 var8 = tempk-hello + tempgo tempk-hello = var8 RETURN tempgo FUNCTION main : tempk-hello = #0 BEGIN_ARGS var10 = tempk-hello ARG var10 var11 = #5 ARG var11 var9 = CALL go tempf-hello = var9 var12 = tempf-hello BEGIN_ARGS ARG var12 CALL WRITELN var13 = tempk-hello BEGIN_ARGS ARG var13 CALL WRITELN	.data _prompt: .asciiz "Enter an integer:" _ret: .asciiz "\n" .globl main .text READ: li $v0,4 la $a0,_prompt syscall li $v0,5 syscall jr $ra WRITE: li $v0,1 syscall li $v0,4 la $a0,_ret syscall move $v0,$0 jr $ra WRITELN: li $v0,1 syscall li $v0,4 la $a0,_ret syscall move $v0,$0 jr $ra go: move $t3,$a0 move $t4,$a1 li $t5,0 seq $t5,$t4,$t5 beq $t5,$zero,L0 move $t6,$t3 move $t7,$a0 move $a0,$t6 li $t6,1 sub $t6,$t4,$t6 move $t8,$a1

RETURN #0	move $a1,$t6 addi $sp,$sp,-28 sw $t7,0($sp) sw $t8,4($sp) sw $ra,8($sp) sw $t3,12($sp) sw $t4,16($sp) sw $t5,20($sp) sw $t6,24($sp) jal go lw $a0,0($sp) lw $a1,4($sp) lw $ra,8($sp) lw $t3,12($sp) lw $t4,16($sp) lw $t5,20($sp) lw $t6,24($sp) addi $sp,$sp,28 move $t6 $v0 mul $t6,$t4,$t6 move $t0,$t6 L0: bne $t5,$zero,L1 li $t0,1 L1: add $t5,$t3,$t0 move $t3,$t5 add $t5,$t1,$t0 move $t1,$t5 move $v0,$t0 jr $ra main: li $t1,0 move $t5,$t1 move $t6,$a0 move $a0,$t5 li $t5,5 move $t9,$a1 move $a1,$t5 addi $sp,$sp,-32 sw $t6,0($sp) sw $t9,4($sp) sw $ra,8($sp)

```
    sw $t3,12($sp)
    sw $t4,16($sp)
    sw $t5,20($sp)
    sw $t7,24($sp)
    sw $t8,28($sp)
    jal go
    lw $a0,0($sp)
    lw $a1,4($sp)
    lw $ra,8($sp)
    lw $t3,12($sp)
    lw $t4,16($sp)
    lw $t5,20($sp)
    lw $t7,24($sp)
    lw $t8,28($sp)
    addi $sp,$sp,32
    move $t5 $v0
    move $t2,$t5
    move $t5,$t2
    move $s0,$a0
    move $a0,$t5
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITELN
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    move $t5,$t1
    move $s1,$a0
    move $a0,$t5
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITELN
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    move $v0,$zero
    jr $ra
```

运行截图：



8.2.5 recursion.pas

源代码	语法树
program hello; var i : integer; function go(a : integer): integer; begin if a = 1 then begin go := 1; end else begin if a = 2 then begin go := 1; end else begin go := go(a - 1) + go(a - 2); end	ProgramDecl:hello RoutineheadDecl VarDecl Id:i SimpleSysType:IntExpType FunctionDecl:go VarParaDecl Id:a SimpleSysType:IntExpType SimpleSysType:IntExpType RoutineheadDecl If Oper:= Id:a Const:1 Assign Id:go Const:1 If

`` ; end ; end ; begin i := go(10); writeln(i); end . ```	``` Oper:= Id:a Const:2 Assign Id:go Const:1 Assign Id:go Oper:+ FuncId Oper:- Id:a Const:1 FuncId Oper:- Id:a Const:2 Assign Id:i FuncId Const:10 ProcSys Id:i ```

中间代码	目标代码
`` FUNCTION go : PARAM tempa-go var1 = #1 var0 = tempa-go == var1 IF_FALSE var0 GOTO L0 tempgo = #1 LABEL L0 IF var0 GOTO L1 var3 = #2 var2 = tempa-go == var3 IF_FALSE var2 GOTO L2 tempgo = #1 LABEL L2 IF var2 GOTO L3 BEGIN_ARGS var7 = #1 var6 = tempa-go - var7 ```	``` .data _prompt: .asciiz "Enter an integer:" _ret: .asciiz "\n" .globl main .text READ: li $v0,4 la $a0,_prompt syscall li $v0,5 syscall jr $ra WRITE: li $v0,1 syscall li $v0,4 ```

```
ARG var6
var5 = CALL go
BEGIN_ARGS
var10 = #2
var9 = tempa-go - var10
ARG var9
var8 = CALL go
var4 = var5 + var8
tempgo = var4
LABEL L3
LABEL L1
RETURN tempgo
FUNCTION main :
BEGIN_ARGS
var12 = #10
ARG var12
var11 = CALL go
tempi-hello = var11
var13 = tempi-hello
BEGIN_ARGS
ARG var13
CALL WRITELN
RETURN #0
```

```
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
WRITELN:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
go:
    move $t2,$a0
    li $t3,1
    seq $t3,$t2,$t3
    beq $t3,$zero,L0
    li $t0,1
L0:
    bne $t3,$zero,L1
    li $t3,2
    seq $t3,$t2,$t3
    beq $t3,$zero,L2
    li $t0,1
L2:
    bne $t3,$zero,L3
    li $t3,1
    sub $t3,$t2,$t3
    move $t4,$a0
    move $a0,$t3
    addi $sp,$sp,-16
    sw $t4,0($sp)
    sw $ra,4($sp)
    sw $t2,8($sp)
    sw $t3,12($sp)
    jal go
    lw $a0,0($sp)
    lw $ra,4($sp)
    lw $t2,8($sp)
    lw $t3,12($sp)
    addi $sp,$sp,16
    move $t3 $v0
    li $t5,2
```

```
    sub $t5,$t2,$t5
    move $t6,$a0
    move $a0,$t5
    addi $sp,$sp,-24
    sw $t6,0($sp)
    sw $ra,4($sp)
    sw $t2,8($sp)
    sw $t3,12($sp)
    sw $t4,16($sp)
    sw $t5,20($sp)
    jal go
    lw $a0,0($sp)
    lw $ra,4($sp)
    lw $t2,8($sp)
    lw $t3,12($sp)
    lw $t4,16($sp)
    lw $t5,20($sp)
    addi $sp,$sp,24
    move $t5 $v0
    add $t3,$t3,$t5
    move $t0,$t3
L3:
L1:
    move $v0,$t0
    jr $ra
main:
    li $t3,10
    move $t5,$a0
    move $a0,$t3
    addi $sp,$sp,-24
    sw $t5,0($sp)
    sw $ra,4($sp)
    sw $t2,8($sp)
    sw $t3,12($sp)
    sw $t4,16($sp)
    sw $t6,20($sp)
    jal go
    lw $a0,0($sp)
    lw $ra,4($sp)
    lw $t2,8($sp)
    lw $t3,12($sp)
    lw $t4,16($sp)
    lw $t6,20($sp)
```

```
    addi $sp,$sp,24
    move $t3 $v0
    move $t1,$t3
    move $t3,$t1
    move $t7,$a0
    move $a0,$t3
    addi $sp,$sp,-8
    sw $a0,0($sp)
    sw $ra,4($sp)
    jal WRITELN
    lw $a0,0($sp)
    lw $ra,4($sp)
    addi $sp,$sp,8
    move $v0,$zero
    jr $ra
```

运行截图：