



PASCAL 编译器

设计及使用手册

谢宁宁 孔晗聪 单才华



目录

1	引言.....	3
1.1	简介.....	3
1.2	编写目的.....	3
1.3	背景.....	3
1.4	定义.....	4
2	系统设计.....	5
2.1	处理流程.....	5
2.2	全局数据结构.....	6
2.2.1	树的节点类型.....	6
2.2.2	声明的类型.....	6
2.2.3	语句的类型.....	7
2.2.4	表达式的类型.....	9
2.2.5	类型声明的类型.....	10
2.2.6	树节点的表达式类型.....	10
2.2.7	树节点的结构定义.....	11
2.3	词法分析.....	12
2.3.1	返回记号.....	12
2.3.2	字段记录.....	13
2.4	语法分析.....	14
2.4.1	便利函数.....	14
2.4.2	语法规则.....	16
2.4.3	lex 与 yacc 结合	17
2.5	符号表架构.....	17
2.6	数据类型结构.....	19
2.6.1	枚举类型.....	19
2.6.2	子界类型.....	19
2.6.3	数组类型.....	20
2.6.4	自定义结构体.....	20
2.7	符号表项结构.....	21
2.7.1	变(常)量表项结构	21
2.7.2	类型表项结构.....	22
2.7.3	函数表项结构.....	23
2.7.4	过程表项结构.....	24
2.8	符号表操作.....	24
2.8.1	插入接口.....	24
2.8.2	查找接口.....	25
2.9	代码生成.....	26
2.9.1	主要流程.....	26
2.9.2	语句处理.....	26
2.9.3	函数递归调用.....	26
2.9.4	函数嵌套定义函数时变量访问	27
2.9.5	类型处理 (int real bool char array record)	28

2.9.6	表达式计算.....	29
2.10	错误提示.....	31
3	代码使用方法:	31
4	测试.....	31
4.1	基本类型测试.....	31
4.1.1	数组.....	31
4.1.2	自定义结构体.....	32
4.2	浮点及四则运算测试.....	33
4.3	简单函数及过程.....	35
4.4	函数及过程递归测试.....	36
4.5	函数及过程嵌套测试.....	37
4.6	系统过程测试.....	39
5	总结.....	40
6	分工.....	40

1 引言

1.1 简介

近年来，计算机界的编程语言越来越多，从迄今已有 60 高龄的 Fortran，到近日苹果发布了 Swift。编程语言的学习是无止境的，然而，想要成为真正的计算机学者，学习编程语言背后的实现方式才是最为重要的。语言的语法是如何来的，想要解析语言语法，如何编写和生成语法树，这都是在设计和实现一门语言时应该考虑的。

1.2 编写目的

软件的设计要求软件开发人员能够设计并实现一个简单的 Pascal 编译器，要求能对基本语句进行分析和编译。通过对该编译器的设计与实现，提供学生的系统编程能力，加深对编译原理课程以及编译系统设计课程上所涉及的课程原理的认识。

因此，编写此文档的目的，是向读者介绍软件的功能，并从系统的结构、编译的步骤等角度，详细地阐述系统的主要架构和详细设计。同时，通过提供测试文件和测试结果，阐明系统的可用性。

预期读者：

1. Pascal 编译器设计人员
2. Pascal 编译器开发人员
3. 软件维护人员

1.3 背景

项目名称	Pascal 编译器
项目提出方	浙江大学编译系统设计课程任课老师——王强
开发人员	谢宁宁 孔晗聪 单才华
目标用户	
系统开发环境	PC: 4G RAM OS: Ubuntu

1.4 定义

■ Pascal

最早出现的结构化编程语言，具有丰富的数据类型和简洁灵活的操作语句，其主要特点有：严格的结构化形式；丰富完备的数据类型；运行效率高；查错能力强。

■ 编译器

编译器是将一种语言翻译为另一种语言的计算机程序。编译器将源程序编写的程序作为输入，而产生用目标语言编写的等价程序。

■ 词法分析

将源程序读作字符文件并将其分为若干个记号。

■ 语法分析

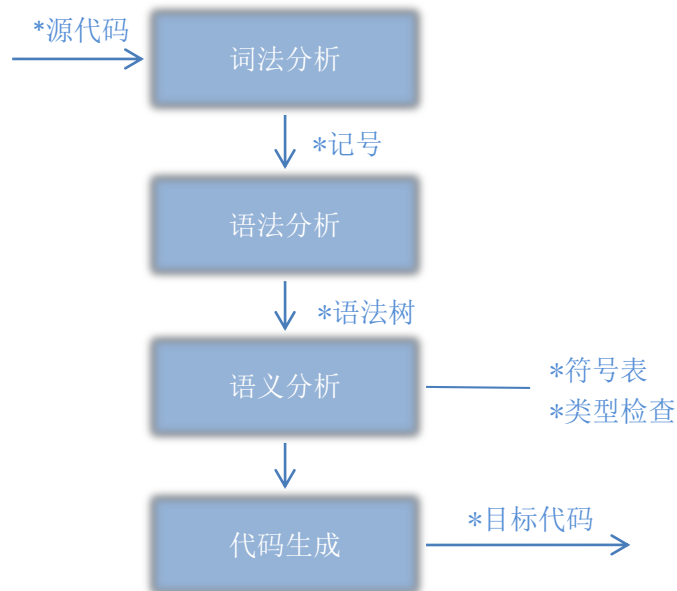
语法分析从扫描程序中获取记号形式的源代码，语法分析定义了程序的结构元素及其关系。将语法分析的结果表示为分析树（`parse tree`）或语法树（`syntax tree`）。

■ 符号表

符号表是一种用于编译器中的数据结构，在符号表中，程序源代码中的每个标识符都和它的声明或使用信息绑定在一起，比如其数据类型、作用域以及内存地址。

2 系统设计

2.1 处理流程



词法分析

主要采用 Lex 扫描程序生成器来生成一个扫描程序,此次在系统中使用的 Lex 版本为 flex (Fast Lex),它是 Gnu compiler package 的一部分。Lex 从作为正则表达式的 TINY 记号的描述中生成一个扫描程序,这个扫描程序用源代码作为输入,并进行扫描,生成记号。

语法分析

主要采用 Yacc 分析程序生成器来生成一个分析程序,此次在系统中使用的 Yacc 版本为 Gnu Bison,它是 Gnu 软件的一部分。Yacc 从定义的语法规则的描述中生成一个分析程序,这个分析程序可以与 Lex 相结合,根据 Lex 提供的记号,并根据语法规则来生成一棵语法树。

语义分析

代码生成

2.2 全局数据结构

2.2.1 树的节点类型

```
typedef enum{  
    NODE_STATEMENT, NODE_EXPRESSION, NODE_DECLARE, NODE_TYPE  
}NodeKind;
```

根据 Pascal 语法，树的节点总共有 4 种类型，分别为

■ NODE_DECLARE

代表声明。在 Pascal 的语法规则中，有各种不同的声明，如声明一个 routine，声明一个 function，声明一个变量 var，等等。

■ NODE_STATEMENT

代表语句。在 Pascal 的语法规则中，支持多种语句，如 if-else 语句，while 语句，switch-case 语句等等。

■ NODE_EXPRESSION

表达式语句。主要是进行各种运算，如加法减法等等，同时也包括常量表达式，如 1,-2 等。

■ NODE_TYPE

代表类型的声明。由于可以自定义类型，而类型的定义不同于其他的定义，需要更多的处理，因此类型的声明被单独提出。

2.2.2 声明的类型

在节点的类型为 NODE_DECLARE 时，需要进一步指定声明的类型。

```
typedef enum{
```

```
DECL_ROUTINEHEAD,  
  
DECL_FUNCTION,DECL_FUNCTIONHEAD,  
  
DECL_PROCEDURE,DECL_PROCEDUREHEAD,  
  
DECL_CONST,DECL_VAR,DECL_TYPE,  
  
DECL_VAR_PARA,DECL_VAL_PARA  
}DeclKind;
```

■ DECL_ROUTINEHEAD

程序的头部声明，也就是对应语法规则中的

routine_head : label_part const_part type_part var_part routine_part

■ DECL_FUNCTION / DECL_FUNCTIONHEAD

函数声明 / 函数的头部声明

■ DECL_PROCEDURE / DECL_PROCEDUREHEAD

过程声明 / 过程头部声明

■ DECL_CONST

在函数头部的常量声明

■ DECL_VAR

在函数头部的变量声明

■ DECL_TYPE

在函数头部的类型声明

■ DECL_VAR_PARA / DECL_VAL_PARA

过程 VAR 参数声明 / 过程参数声明

2.2.3 语句的类型

在节点的类型为 NODE_STMT 时，需要进一步指定语句的类型。

```
typedef enum{
```



```
    STMT_LABEL,  
    STMT_ASSIGN,STMT_GOTO,STMT_IF,STMT_REPEAT,STMT_WHILE,STMT_FOR,STMT_CASE,  
    STMT_PROC_ID,STMT_PROC_SYS  
}StmtKind;
```

■ STMT_LABEL

带标号的语句，此时需要额外存储标号。

■ STMT_ASSIGN

赋值语句，对应：

```
assign_stmt : ID ASSIGN expression  
            | ID LB expression RB ASSIGN expression  
            | ID DOT ID ASSIGN expression
```

■ STMT_GOTO

GOTO 语句，对应于带标号的语句：

```
goto_stmt : GOTO INTEGER
```

■ STMT_IF

if-else 语句，对应规则：

```
if_stmt : IF expression THEN start else_clause  
else_clause : ELSE stmt |  $\epsilon$ 
```

■ STMT_REPEAT

repeat 语句，对应规则：

```
repeat_stmt : REPEAT stmt_list UNTIL expression
```

■ STMT_WHILE

while 语句，对应规则：

```
while_stmt : WHILE expression DO stmt
```

■ STMT_FOR

for 语句，对应规则：

```
for_stmt : FOR ID ASSIGN expression direction expression DO stmt
```

■ STMT_CASE

case 语句，对应规则：

case_stmt : CASE expression OF case_expr_list END

- STMT_PROC_ID

过程调用，调用的名称作为 ID，代表是自定义过程

- STMT_PROC_SYS

系统过程调用，有 read 和 write 等。

2.2.4 表达式的类型

在节点的类型为 NODE_EXP 时，需要进一步指定表达式的类型。

```
typedef enum{  
    EXP_ID,  
    EXP_CONST,  
    EXP_OP,EXP_CASE,EXP_FUNC_ID,EXP_FUNC_SYS  
}ExpKind;
```

- EXP_ID

ID 表达式，ID 为之前已定义过的变量或是常量

- EXP_CONST

常量表达式，例如值 1，-2 等。

- EXP_OP

运算表达式，例如加法，减法，与，等等。

- EXP_CASE

在 switch-case 语句中，存储 case 的值以及符合 case 情况下对应的语句序列。

- EXP_FUNC_ID

函数调用，调用的名称作为 ID，还需要保存参数

- EXP_FUNC_SYS

系统函数调用，有 abs 和 succ 等。

2.2.5 类型声明的类型

在节点的类型为 `NODE_TYPE` 时，需要进一步指定声明类型是属于哪一种已有类型。

```
typedef enum{  
    TYPE_SIMPLE_SYS,TYPE_SIMPLE_ID, TYPE_SIMPLE_ENUM, TYPE_SIMPLE_LIMIT,  
    TYPE_ARRAY, TYPE_RECORD  
}TypeKind;
```

■ TYPE_SIMPLE_SYS

简单类型中的系统类型，也就是 `INT,BOOLEAN` 等。

■ TYPE_SIMPLE_ID

简单类型中的 ID 类型，也就是声明这个类型与已知类型 ID 相同

■ TYPE_SIMPLE_LIMIT

简单类型，主要是出现在数组中的写法，也就是 `1..2` 等。

■ TYPE_SIMPLE_ENUM

简单类型中的枚举类型。

■ TYPE_ARRAY

数组类型

■ TYPE_RECORD

结构类型

2.2.6 树节点的表达式类型

标注该节点的类型，只有在声明常量和常量表达式两种情况下会在语法分析中写入。

例如，声明 `const a=2;b=3;c=2.1;`，那么 `a,b,c` 的类型都会存为 `EXPTYPE_INT`。

包含所有需要支持的内嵌类型。

```
typedef enum{  
  
    EXPTYPE_VOID,EXPTYPE_INT,EXPTYPE_REAL,  
  
    EXPTYPE_CHAR,EXPTYPE_STRING,EXPTYPE_BOOL,  
  
    EXPTYPE_ARRAY,EXPTYPE_RECORD,  
  
    EXPTYPE_SIMPLE_ID,EXPTYPE_SIMPLE_ENUM,EXPTYPE_SIMPLE_LIMIT  
  
}ExpType;
```

2.2.7 树节点的结构定义

有了前面的类型定义后，来定义树的节点类型就简单了很多。

在语法树中，所有的节点都具有下面的结构：

```
typedef struct treeNode{  
  
    struct treeNode * child[MAXCHILDREN];  
  
    struct treeNode * sibling;  
  
    NodeKind nodekind;  
  
    int lineno;  
  
    union{  
  
        StmtKind stmt;  
  
        ExpKind exp;  
  
        DeclKind decl;  
  
        TypeKind type;  
  
    }kind;  
  
    union{  
  
        TokenType op;  
  
        char * name;  
  
        int val;  
  
        char char_val;  
  
        char * string_val;
```

```
double real_val;

}attr;

ExpType type;

}TreeNode;
```

1. child

节点的孩子。在程序中定义 `MACHILDREN=4`，这个数字可以根据需要调整。

2. sibling

节点的兄弟。

3. lineno

记录该节点出现的位置在源代码中的行号

4. NodeKind nodekind

属于 `NodeKind` 枚举类型中的一种，标识节点的类型

5. union kind

具体地标识节点类型，与前几节的枚举类型相对应。例如，如果 `nodekind=NODE_STMT`，那么在 `kind` 中，唯一有效的为 `StmtKind stmt`，它的值为 `StmtKind` 枚举值中的一个。例如，如果是 `if-else` 语句，那么对应 `nodekind=NODE_STMT`，`kind.stmt=STMT_IF`。

6. union attr

节点的额外属性记录。例如如果节点为 `ID`，那么 `attr.name` 将会保存节点的名称，如果节点为 `EXP_CONST` 且 `EXPTYPE_INT`，那么该节点的值将会被存在 `attr.val`。

2.3 词法分析

lex 版本: flex

在 `lex` 中，需要定义 `getToken()` 函数，它主要做两种工作：返回记号；字段记录。

2.3.1 返回记号

词法分析主要是使用 `flex`，并根据语法规则制定记号的正则表达式规则，根据规则进行匹配，并返回匹配到的记号。

所有的记号定义：

```
%token TOKEN_PROGRAM TOKEN_FUNCTION TOKEN_PROCEDURE TOKEN_CONST TOKEN_TYPE
TOKEN_VAR

%token  TOKEN_IF  TOKEN_THEN  TOKEN_ELSE  TOKEN_REPEAT  TOKEN_UNTIL  TOKEN_WHILE
TOKEN_DO TOKEN_CASE TOKEN_TO TOKEN_DOWNTO TOKEN_FOR

%token  TOKEN_EQUAL  TOKEN_UNEQUAL  TOKEN_GE  TOKEN_GT  TOKEN_LE  TOKEN_LT
TOKEN_ASSIGN TOKEN_PLUS TOKEN_MINUS TOKEN_MUL TOKEN_DIV TOKEN_OR TOKEN_AND
TOKEN_NOT TOKEN_MOD TOKEN_READ TOKEN_WRITE TOKEN_WRITELN

%token TOKEN_LB TOKEN_RB TOKEN_SEMI TOKEN_DOT TOKEN_DOTDOT TOKEN_LP TOKEN_RP
TOKEN_COMMA TOKEN_COLON

%token TOKEN_INTEGER_TYPE TOKEN_BOOLEAN_TYPE TOKEN_CHAR_TYPE TOKEN_REAL_TYPE

%token TOKEN_TRUE TOKEN_FALSE TOKEN_MAXINT

%token TOKEN_ARRAY TOKEN_OF TOKEN_RECORD TOKEN_BEGIN TOKEN_END TOKEN_GOTO

%token TOKEN_ID TOKEN_INT TOKEN_REAL TOKEN_CHAR TOKEN_STRING

%token ERROR

%token  TOKEN_ABS  TOKEN_CHR  TOKEN_ODD  TOKEN_ORD  TOKEN_PRED  TOKEN_SQR
TOKEN_SQRT TOKEN_SUCC

%%
```

2.3.2 字段记录

有两个全局的变量需要 `lex` 进行记录。

- **tokenString**

除了返回记号之外，有时候还需要这个记号背后的字符串，例如在返回 `ID` 的时候，需要记录 `ID` 的名字以保存到 `attr.name` 中。因此，在 `lex` 返回记号的同时，需要使用 `tokenString` 记录此时的字符串，也就是复制 `yytext`，以便取用。

- **lineno**

`lineno` 记录当下解析到源文件的行号。

在匹配规则中，有一条定义

```
newline \n
```

对应规则

```
{newline} {lineno++;}
```

用以保持在遇到新行时更新 `lineno`。

在生成新的树节点时，树节点中 `lineno` 就会被赋予全局变量 `lineno` 的值。

2.4 语法分析

yyac 版本: bison

yyac 主要是根据指定的语法规则，利用 `util.c` 中提供的便利函数生成新的树节点，并确认树节点之间兄弟和孩子之间的关系，同时赋值节点中所对应的各种类型。

2.4.1 便利函数

在 `util.c` 中，提供不少的便利函数，在此以 `newDeclNode()` 为例，`newDeclNode()` 函数能够生成一个新的树节点，树节点的 `NodeKind=NODE_DECLARE`，并做一系列的初始化操作。

代码如下：

```
TreeNode * newDeclNode(DeclKind kind){
    TreeNode * t = (TreeNode *)malloc(sizeof(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=NODE_DECLARE;
```

```

        t->kind.decl = kind;

        t->lineno=lineno;

    }

    return t;
}

```

完成一系列的动作：

1. 申请空间
2. 初始化孩子指针和兄弟指针为 NULL
3. 将 nodekind 赋值为 NODE_DECLARE
4. 传输的参数为 DeclKind 的一个对象，由此值来初始化 kind.decl
5. 记录行号

完成这一系列的动作后，返回该节点的指针。

其余的便利函数

TreeNode * newStmtNode(StmtKind s);	//新的 NODE_STATEMENT 节点
TreeNode * newExpNode(ExpKind e);	//新的 NODE_EXPRESSION 节点
TreeNode * newDeclNode(DeclKind d);	//新的 NODE_DECLARE 节点
TreeNode * newTypeNode(TypeKind type);	//新的 NODE_TYPE 节点
TreeNode * newOpExpNode(TreeNode*, TreeNode*, TokenType);	
//newExpNode 的进一步包装的便利函数，针对二元表达式	
TreeNode * newFuncSysExpNode(TokenType op, TreeNode* args);	
//newExpNode 的进一步包装的遍历函数，针对系统函数调用	
void freeNode(TreeNode*);	//释放节点，适用于一些临时节点
char * copyString(char*);	//复制字符串
void printTree(TreeNode *);	//打印语法树

2.4.2 语法规则

语法规则决定了树节点之间的关系，例如，对于语法规则

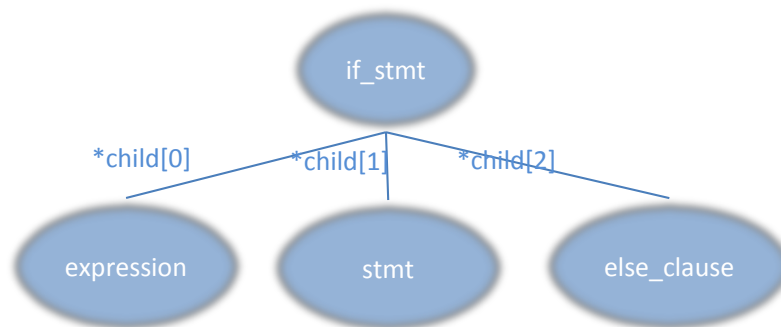
```
if_stmt : IF expression THEN start else_clause
```

对应于在 yacc 中的规则

```
if_stmt          :  TOKEN_IF expression TOKEN_THEN stmt  else_clause
                    {
                        $$=newStmtNode(STMT_IF);
                        $$->child[0]=$2;
                        $$->child[1]=$4;
                        $$->child[2]=$5;
                    }
```

首先，结点申请了一个新的 `NODE_STATEMENT` 类型的节点，其 `kind.stmt = STMT_IF`，并且该节点有 3 个孩子，第一个孩子对应于 if 语句的判断表达式，第二个孩子对应于 if 语句的 then-部分，也就是表达式为 `True` 时执行的语句，第三个孩子对应于 else-部分，也就是表达式为 `False` 时执行的语句。

对应语法树：



需要注意在语法为空时的处理。

因为在语法规则中语法有可能为空，因此需要赋节点为 `NULL` 而不能不管它，例如在上述的 `if_stmt` 语法中，`else_clause` 有可能为空，因此在 `else_clause` 中，规则制定为：

```
else_clause      :  {$|=NULL;}
                  |  TOKEN_ELSE stmt  {$|= $2;}
```

最终生成的整棵语法树的根节点保存在全局变量 `savedTree` 中。

```
static TreeNode* savedTree;
```

在解析过程后最终会返回 `savedTree`

```
TreeNode * parse(){  
    yyparse();  
    return savedTree;  
}
```

2.4.3 lex 与 yacc 结合

- Token

Token 原本作为枚举值在 `.l` 文件中定义，然而由于要结合 `lex` 与 `yacc`，因此需要删除 `.l` 文件中的 `Token` 定义，转而在 `.y` 文件中以 `%token` 的形式定义 `Token`。例如 `%token TOKEN_PROGRAM TOKEN_FUNCTION TOKEN_PROCEDURE` 等等。

- `yylex()`

`yacc` 默认通过调用 `yylex()` 函数来获取记号，由于我们在 `lex` 中定义了函数 `getToken()` 来获取记号，因此需要在 `yacc` 的第一部分定义 `yylex()`，来调用 `getToken()`

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

2.5 符号表架构

符号表是一个典型的目录数据结构，提供插入，查找删除这 3 中基本操作。为了保证查找变量的效率，符号表的结构采用杂凑表，并且使用分离链表的方式来处理冲突。

`Pascal` 中的说明包括常量及变量声明，类型声明，过程及函数声明三种，为了方便处理我们采用四张哈希表分别称为变(常)量表，类型表，过程表和函数表。

```
/*变(常)量的哈希表*/  
static VariableList variableHashTable[SIZE];  
  
/*类型的哈希表*/  
static TypeList typeHashTable[SIZE];  
  
/*函数的哈希表*/  
static FuncList funcHashTable[SIZE];  
  
/*过程的哈希表*/  
static ProcList procHashTable[SIZE];
```

在这四张表中，类型表，函数表和过程表主要用于存储所声明的完整的数据结构，而变(常)量表需要指明对应变量的数据结构并返回存储的基地址和偏移地址。

为实现作用域的嵌套，我们使用类似堆栈的方式处理，每一个变量需要采用一个 `NestLevel` 进行标记作用域层级，这样符号表插入操作时不必改写之前的说明且可以临时隐藏父过程中同名的变量。为此我们需要维护一个标记当前 `NestLevel` 的全局变量。

```
static int currentNestLevel=0;
```

然后给出三个接口，分别处理初始化，进入新的作用域，退出作用域的情况。

```
void initScope();  
int enterNewScope(TreeNode* t, int new_base);  
void leaveScope();
```

`initScope` 实现清空符号表，将全局 `currentNestLevel` 归零的功能。

`enterNewScope` 在进入新作用域时调用，首先将全局 `currentNestLevel` 加一，并通过以给出的进入新作用域的语法树节点为入口遍历树，生成符号表，存储对应的基地址和偏移地址信息，返回符号表总的偏移量。

`leaveScope` 在离开作用域时调用，将 `currentNestLevel` 减一，遍历符号表，将其中的 `NestLevel` 大于当前 `currentNestLevel` 的变量或声明全部清除。

2.6 数据类型结构

Pascal 提供了包括整型，浮点型，布尔型，枚举型，子界型等简单数据类型，还有数组和自定义结构等复杂数据类型。对于较复杂不同的数据类型我们设计不同的结构进行存储。

2.6.1 枚举类型

枚举类型以链表的方式实现，以字符串的方式存储枚举项。其结构定义如下。

```
typedef struct EnumDefRec {  
    char* mark; //指向语法树种的常量字符串  
    struct EnumDefRec* next; //指向下一个结构体  
}* EnumDef;
```

为了构建一个枚举类型我们给出如下创建接口。

```
EnumDef newEnumDef(char* mark);  
EnumDef insertEnumDef(EnumDef enumList, char* mark);
```

前者用于创建一个枚举类型，后者用于在给定的枚举链表中插入枚举项。

2.6.2 子界类型

子界类型中的子界可以分为整数型，字符型或者枚举型，所以我们设计如下的结构定义。

```
typedef union {  
    int i; //整型  
    char c; //字符型  
    char* m; //枚举型  
} Bound; //界的结构定义  
  
typedef struct SubBoundDefRec {
```

```
ExpType boundType; //指明子界的类型

Bound LowerBound; //指向下界的 Bound 结构体

Bound UpperBound; //指向上界的 Bound 结构体

}* SubBoundDef;
```

为了构建一个子界类型我们给出如下创建接口

```
SubBoundDef newSubBoundDef(ExpType type, void* upper, void* lower)
```

参数为子界的类型，上界值，和下界值，以指针的方式传递。

2.6.3 数组类型

枚举类型以链表的方式实现，以字符串的方式存储枚举项。其结构定义如下。

```
typedef struct EnumDefRec {

    char* mark; //指向语法树种的常量字符串

    struct EnumDefRec* next; //指向下一个结构体

}* EnumDef;
```

为了构建一个枚举类型我们给出如下创建接口。

```
EnumDef newEnumDef(char* mark);

EnumDef insertEnumDef(EnumDef enumList, char* mark);
```

前者用于创建一个枚举类型，后者用于在给定的枚举链表中插入枚举项。

2.6.4 自定义结构体

自定义结构体分为两种类型，一种是匿名类，一种是已经在 Pascal 的 type 字段定义过的结构体。若是已定义过的结构体，给出指向定义过的类型结构的指针，若为匿名类，新建一个类型结构并给出其指针。

```
typedef enum {ANONYMOUS, DEFINED} RecordType; //自定义结构分匿名和一定义两类

typedef struct RecordNodeRec {
```

```

RecordType type; //指明自定义结构类型

union {

    struct TypeListRec* pDef; //pDef 指向 TypeList 中的自定义结构体

    struct TypeListRec* pAnony; //pAnony 指向一个新建的类型结构

} ptr;

}* RecordDef;

```

为了创建一个自定义类型我们给出如下接口。

```

RecordDef newDefinedRecord(TypeList ptr); //新建已定义类

RecordDef newAnonyRecord(TypeList ptr); //新建匿名类

```

前者用于创建已定义类，后者用于创建匿名类。

2.7 符号表项结构

2.7.1 变(常)量表项结构

在变(常)量表中以如下数据结构作为一个“项”，用以存储变量的类型以及地址。

```

typedef struct VariableListRec {

    char* name; //记录变(常)量名称

    ExpType type; //根据 global.h 的定义指明变量类型

    int isConst; //该数据是否是常量

    int nestLevel; //标记数据作用域

    void* pAttr; //当数据类型为数组，枚举，子界或者自定义类型时指向数据的详细定义

                //若为简单类型时则为 NULL

    LineList lines; //记录变(常)量出现行号

    MemLoc memloc; //记录变(常)量存储地址

    struct VariableListRec* next; //指向下一个符号表表项

}* VariableList;

```

其中记录存储地址的 MemLoc 结构如下。

```
typedef struct MemLocRec {  
    int baseLoc; //作用域的基地址  
    int offset; //作用域的偏移地址  
} MemLoc;
```

记录行号的 LineList 结构如下。

```
typedef struct LineListRec {  
    int lineno; //行号  
    struct LineListRec* next; //指向下一个结构的指针  
}* LineList;
```

2.7.2 类型表项结构

在类型表中以如下数据结构作为一个“项”，存储 **type** 字段中定义的类型结构，所占空间大小及别名等信息。

```
typedef struct TypeListRec {  
    char* name; //根类型名  
    AliasList aliasSet; //别名  
    ExpType type; //指明是简单类型，数组类型及其他  
    int nestLevel; //嵌套层  
    int size; //该类型所占空间大小  
    void* pAttr; //当数据类型为数组，枚举，子界或者自定义类型时指向数据的详细定义  
                //若为简单类型时则为 NULL  
    struct TypeListRec* next;  
}* TypeList;
```

其中对于类型的别名，因为 **Pascal** 采用说明等价，所以需要记录同一类型的别名。以如下结构存储。

```
typedef struct AliasListRec {  
    char* alias; //别名
```

```
struct AliaseListRec* next; //指向下一个结构的指针  
}* AliaseList;
```

当判断类型等价有时需要根据别名获得根类型名。

2.7.3 函数表项结构

在函数表中以如下数据结构作为一个“项”，存储函数的参数表和返回值类型等信息。

```
typedef struct FuncListRec {  
    char* name; //函数名  
    SimpleTypeList paraList; //参数表  
    ExpType retType; //返回类型  
    int nestLevel; //嵌套层  
    struct FuncListRec* next; //指向下一个结构的指针  
}* FuncList;
```

其中的参数表用如下一个简单类型的链表实现，依序存储函数的参数类型及名称。对于 Pascal 而言，函数的参数表用关键词 **Var** 来指明是调用值还是调用变量，所以需要有一个变量 **isVar** 来记录这一点。

```
typedef struct SimpleTypeListRec {  
    char* name; //参数名  
    ExpType type; //参数类型  
    int isVar; //指明是 Value 还是 Variable  
    struct SimpleTypeListRec* next; //指向下一个结构体  
}* SimpleTypeList;
```

构建参数表提供了如下两个接口函数。

```
SimpleTypeList newSimpleTypeList(char* name, ExpType type, int isVar); //新建参数表  
SimpleTypeList insertSimpleTypeList(SimpleTypeList simpleList, char* name, ExpType type, int
```



```
isVar); //向已经建立的参数表中加入新的参数项
```

2.7.4 过程表项结构

在过程表中以如下数据结构作为一个“项”，存储过程的参数表等信息。

```
typedef struct ProcListRec {  
    char* name; //过程名  
    SimpleTypeList paraList; //参数表  
    int nestLevel; //嵌套层  
    struct ProcListRec* next; //指向下一结构体  
}* ProcList;
```

和函数表项的差别是过程表项不包含返回值。

2.8 符号表操作

对于符号表的操作,查找和插入两类,删除的操作已在前文进出嵌套域的函数中涵盖了。

2.8.1 插入接口

对于四个符号表的操作我们分别提供一个接口,输入为待查找符号的名字,输出为指向对应结构体的指针。

函数和过程表的插入接口如下,所给的参数是语法树中 `FunctionHead` 和 `ProccessHead` 的树节点,接口函数会遍历这个节点及其子节点在函数表中新建自身的表项,并在变量表中插入参数表变量,若是函数的话还包括与函数同名的返回值,用以在代码生成阶段计算偏移量。

```
int procListInsert(TreeNode* procHead);  
int funcListInsert(TreeNode* funcHead);  
void typeListInsert(char* name, ExpType type, int nestLevel, void* pAttr, int size);
```

```
void varListInsert(char* name, ExpType type, int isConst, int nestLevel, void* pAttr, int lineno, int baseLoc, int offset);
```

特别的，为实现 Pascal 的说明等价，类型表插入还另有一个插入别名的函数用以为同一类型插入等价类型别名。

```
void typeListAliasInsert(char* name, char* aliase);
```

2.8.2 查找接口

对于四个符号表的操作我们分别提供一个接口，输入为待查找符号的名字，输出为指向对应结构体的指针。

```
VariableList varListLookup(char* name);  
FuncList funcListLookup(char* name);  
ProcList procListLookup(char* name);  
TypeList typeListLookup(char* name);
```

其他，因为我们对于单层嵌套的数组或者自定义结构提供了专门的查询函数以提高效率，在这中情况下数组查询输入为数组名和数组下标，自定义结构体输入结构名和对应结构内的变量名，返回值为一个包括嵌套层信息和偏移量的结构体。

```
LookupRet arrayLookup(char* a, int i);  
LookupRet recordLookup(char* rec, char* a);
```

返回值的结构体定义如下。

```
typedef struct LookupRetRec {  
    int totalOff; //总偏移量  
    int jumpLevel; //跳转嵌套层数  
} LookupRet;
```

2.9 代码生成

2.9.1 主要流程

- step1: 根据运行环境（linux/windows）生成 x86 汇编的头部代码
- step2: 获得一个语法树节点，根据节点类型转到相应的子程序处理
- step3: 根据符号表和节点类型，在子程序中生成相应 x86 汇编代码
- step4: 执行当前节点的下一个兄弟节点，返回 step2
- step5: 执行完所有节点后，根据运行环境（linux/windows）生成 x86 汇编的尾部代码

2.9.2 语句处理

我们实现了所有语句处理，包括 assign/if-else/for(to/downto)/while/repeat/switch-case/label-goto，以及基本的系统函数调用(read 和 write，可以输入输出 int 和 real)

```
void CGStmtAssign(TreeNode* pnode);
void CGStmtFor(TreeNode* pnode);
void CGStmtIf(TreeNode* pnode);
void CGStmtRepeat(TreeNode* pnode);
void CGStmtWhile(TreeNode* pnode);
void CGStmtLabel(TreeNode* pnode);
void CGStmtCase(TreeNode* pnode);
void GStmtOutput(TreeNode* pnode);
void GStmtInput(TreeNode* pnode);
```

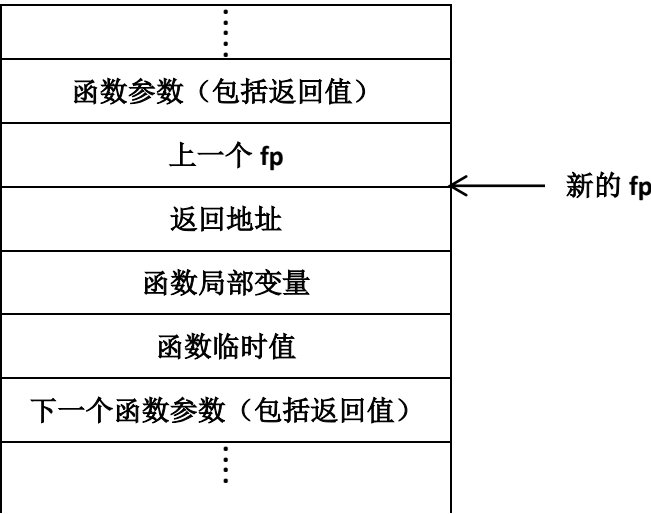
2.9.3 函数递归调用

我们使用了访问控制链（access control link）来处理函数调用。当要进行函数调用时，在主函数中，我们存入子函数的所有参数。如果是 function 我们需要存返回值，如果是 procedure 则不需存。之后存入主函数的 fp 指针。然后使用 x86 中的 call 调用子函数（x86

中的 `call` 指令会自动将返回地址存入栈中，不需要显示存储）。最后，在子函数中，存入子函数的局部变量，执行子函数每一句指令。

在执行完子函数后，首先需要将函数的局部变量出栈。之后使用 `x86` 的 `ret` 指令返回主函数（自动将返回地址出栈）。然后替换 `fp`。最后处理函数的参数，根据有无 `var` 关键词判断参数是形参还是实参，如果是实参的话需要将这个变量原来的值替换为子函数运行后的参数值。

这样实现的函数调用自然而然就可以实现函数递归调用。



2.9.4 函数嵌套定义函数时变量访问

当有多个函数嵌套定义时，内部函数可能会访问之前函数的变量。我们的实现方式是首先通过符号表来得到该变量所在函数离当前函数的层数差与变量在所在函数中离函数 `fp` 之间的偏移。然后在栈内回溯地寻找到该变量所在函数的 `fp`。使用那层函数的 `fp` 加上偏移就是变量的实际位置。

```
// level 中存有变量所在函数离当前函数的层数差

//offset 中存有变量在所在函数中离函数 fp 之间的偏移

sprintf(tmp, "mov edi, %d\n", offset);

CG_OUTPUT("mov esi, ecx\n");

while (level){    //找到变量所在函数的 fp

    CG_OUTPUT("mov eax, [esi]\n");
```

```

CG_OUTPUT("mov esi, eax\n");

level=level-1;

}

CG_OUTPUT("add esi, edi\n"); //fp+offset 为变量实际位置

//运行此段程序后，esi 中存有变量的实际位置

```

2.9.5 类型处理（int real bool char array record）

我们实现了 int 和 real 类型，bool 和 char 类型被直接当作 int 型处理，实现了简单的 array 型和 record 型（不支持嵌套定义）。

我们实现的 array 的下标可以是表达式，且数组范围可以不从 1 开始。实现方式如下：查找 array 中某个元素的位置，我们首先得到 array 的首元素的位置及范围，然后通过计算得到元素位置。

```

VariableList ssvar=varListLookup(pnode->attr.name);

lower=((ArrayDef)ssvar->pAttr->subBound->LowerBound.i; //得到 array 范围下界

level=ssvar->memloc.baseLoc; //首元素的位置

offset=ssvar->memloc.offset;


CGNodeExpression(pnode->child[0]); //计算 array 的下标值(可能是表达式)

sprintf(tmp,"mov ebx, %d\n",lower); //数据范围下界

CG_OUTPUT(tmp);

CG_OUTPUT("sub eax, ebx\n"); //得到与首元素的差值

CG_OUTPUT("mov ebx, 4\n");

CG_OUTPUT("imul eax,ebx\n"); //地址差值，默认每个元素 4byte

sprintf(tmp,"mov edi, %d\n",offset);

CG_OUTPUT(tmp);

CG_OUTPUT("add edi, eax\n"); //a[i]的实际位置

```

我们实现的 record 内可以包含有 int,real 等类型，实际实现如下：

```
st_var=recordLookup(pnode->attr.name,(pnode->child[0])->attr.name);  
level=st_var.jumpLevel;  
offset=st_var.totalOff;
```

之后的实现与变量地址查找一致。

2.9.6 表达式计算

在确定表达式符号两边节点的类型时，我们使用了一个 `runningType` 来记录节点类型。如果两边节点类型一致则该表达式节点的 `runningType` 也是此类型，否则报错。表达式两边的值一开始在栈中，最后计算得到的值在 `eax` 中。

```
if ((pnode->child[0])->RuningType==EXPTYPE_REAL &&  
    (pnode->child[1])->RuningType==EXPTYPE_REAL)  
    pnode->RuningType=EXPTYPE_REAL;  
if ((pnode->child[0])->RuningType==EXPTYPE_INT &&  
    (pnode->child[1])->RuningType==EXPTYPE_INT)  
    pnode->RuningType=EXPTYPE_INT;  
if (pnode->RuningType==EXPTYPE_INT){ //两边都是 int 型，结果保存在 eax 中  
    CG_OUTPUT("pop ebx\n");  
    CG_OUTPUT("pop eax\n");  
    switch(pnode->attr.op)  
    {  
    case TOKEN_PLUS:  
        CG_OUTPUT("add eax, ebx\n");  
        break;  
    case TOKEN_MINUS:  
        CG_OUTPUT("sub eax, ebx\n");  
        break;  
    case TOKEN_MUL:
```

```

        CG_OUTPUT("xor edx, edx\nimul ebx\n");

        break;

    case TOKEN_DIV:

        CG_OUTPUT("xor edx, edx\nidiv ebx\n");

        .....

    }

else { //两边都是 real 型，结果保存在 eax 中

    CG_OUTPUT("fld dword ptr [esp+4]\n");

    CG_OUTPUT("fld dword ptr [esp]\n");

    CG_OUTPUT("pop eax\n");

    CG_OUTPUT("pop eax\n");

    switch(pnode->attr.op){

    case TOKEN_PLUS:

        CG_OUTPUT("fadd\n");

        break;

    case TOKEN_MINUS:

        CG_OUTPUT("fsub\n");

        break;

    case TOKEN_MUL:

        CG_OUTPUT("fmul\n");

        break;

    case TOKEN_DIV:

        CG_OUTPUT("fdiv\n");

        break;

        .....

    CG_OUTPUT("sub esp,4\n");

    CG_OUTPUT("fstp dword ptr [esp]\n");

    CG_OUTPUT("pop eax\n");

}

```

2.10 错误提示

我们共有如下的错误提示:

赋值语句的左值不能为常理。

表达式符号两边类型一致。

变量不存在或变量重定义。

变量不是数组或记录，变量用法出错。

3 代码使用方法:

- 1、编译：在目录下使用 `make`（如果 `a.out` 已经存在则需先将它删除）
- 2、在 `test2` 中输出 `pascal` 代码，使用 `./a.out` 翻译
- 3、翻译结果在 `out.asm` 中，使用 `masm` 来运行 `out.asm`
- 4、windows 下我们使用 `visual studio` 来运行 `out.asm`。

具体教程见 <http://blog.csdn.net/natepan/article/details/6781439>

4 测试

4.1 基本类型测试

4.1.1 数组

Pascal 源程序

```
program a;
var
i:integer;
a:array[1..10] of integer;
begin
for i:=1 to 10 do
    a[i]:=i;
for i:=1 to 10 do
```

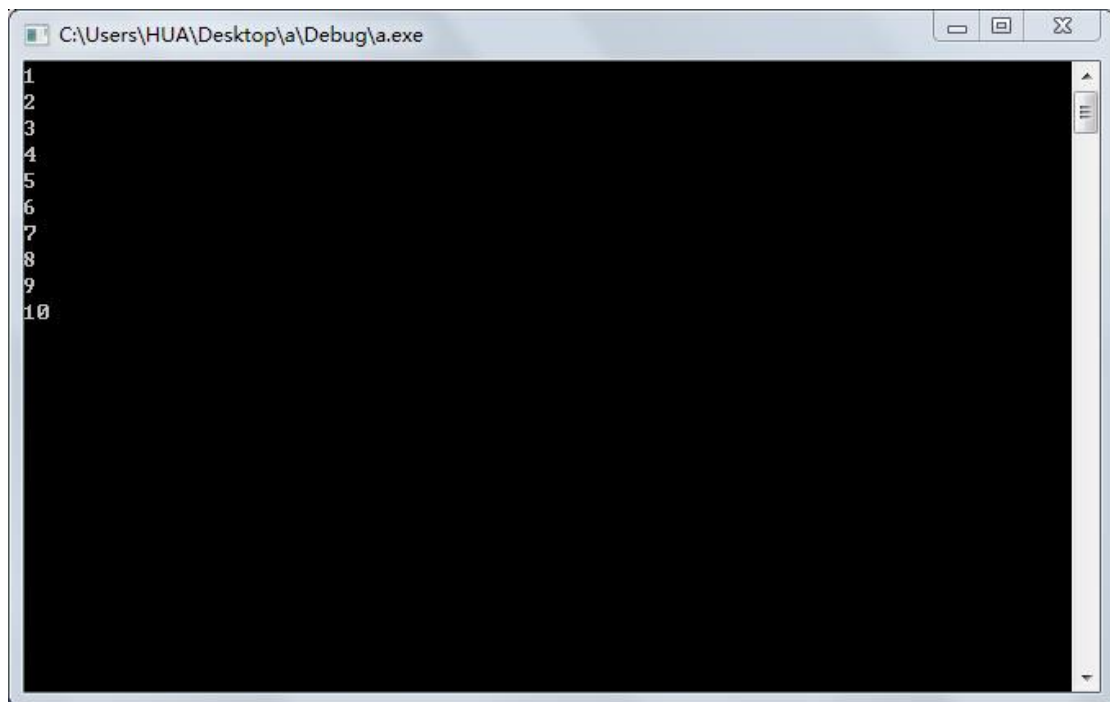


```
        writeln(a[i]);
end.
```

符号表

```
Symbol table:
Variable Name      NestLevel      Location      Line Number
-----
a                  0              -48           4
i                  0              -8            3
Function Name      NestLevel      Return Type   Parameter
-----
cong@cong-Lenovo-IdeaPad-Y400:~/project_final$
```

X86 汇编码执行结果



4.1.2 自定义结构体

Pascal 源程序

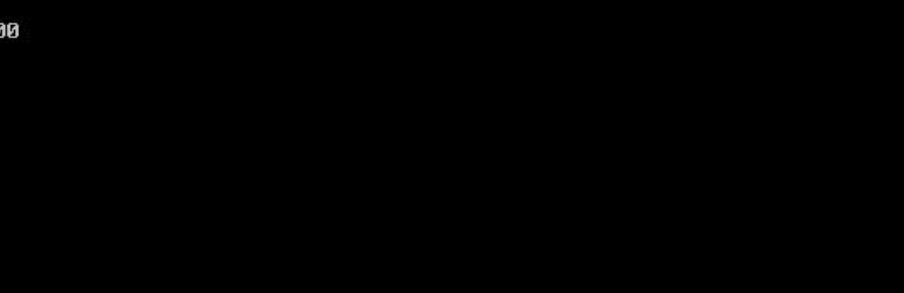
```
program a;  
type  
    stu=record  
        num:integer;  
        price:real;  
    end;  
var  
    student:stu;
```

```
begin
student.num:=5;
student.price:=1.2;
writeln(student.num);
writeln(student.price);
end.
```

符号表

```
Symbol table:
Variable Name      NestLevel      Location      Line Number
-----
student           0             -12           8
Function Name      NestLevel      Return Type   Parameter
-----
cong@cong-Lenovo-IdeaPad-Y400:~/project_final$
```

x86 汇编码执行结果



C:\Users\HUA\Desktop\Debug\1.exe

```
5
1.200000
```

4.2 浮点及四则运算测试

Pascal 源程序

```
program a;  
var  
i,j,z:integer;
```

```

a,b,c:real;
begin
a:=0.2;
b:=-0.3;
c:=a+b-0.001;
writeln(c);
c:=a*b;
writeln(c);

if c<0.1 then
  writeln(1)
else
  writeln(0);

i:=2;
j:=-3;
z:=i*j;
writeln(z);
z:=i mod j;
writeln(z);
end.

```

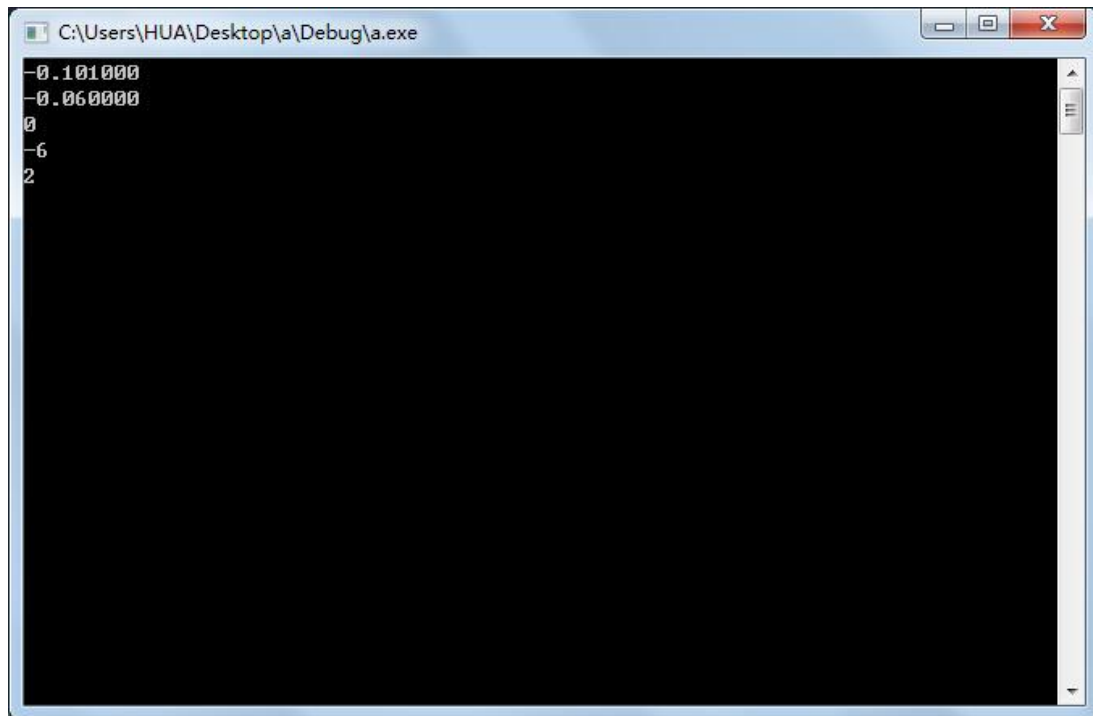
符号表

Symbol table:

Variable Name		NestLevel	Location	Line Number
a	0	-20	4	
b	0	-24	4	
c	0	-28	4	
i	0	-8	3	
j	0	-12	3	
z	0	-16	3	
Function Name		NestLevel	Return Type	Parameter

cong@cong-Lenovo-IdeaPad-Y400:~/project_final\$

X86 汇编码执行结果



4.3 简单函数及过程

Pascal 源程序

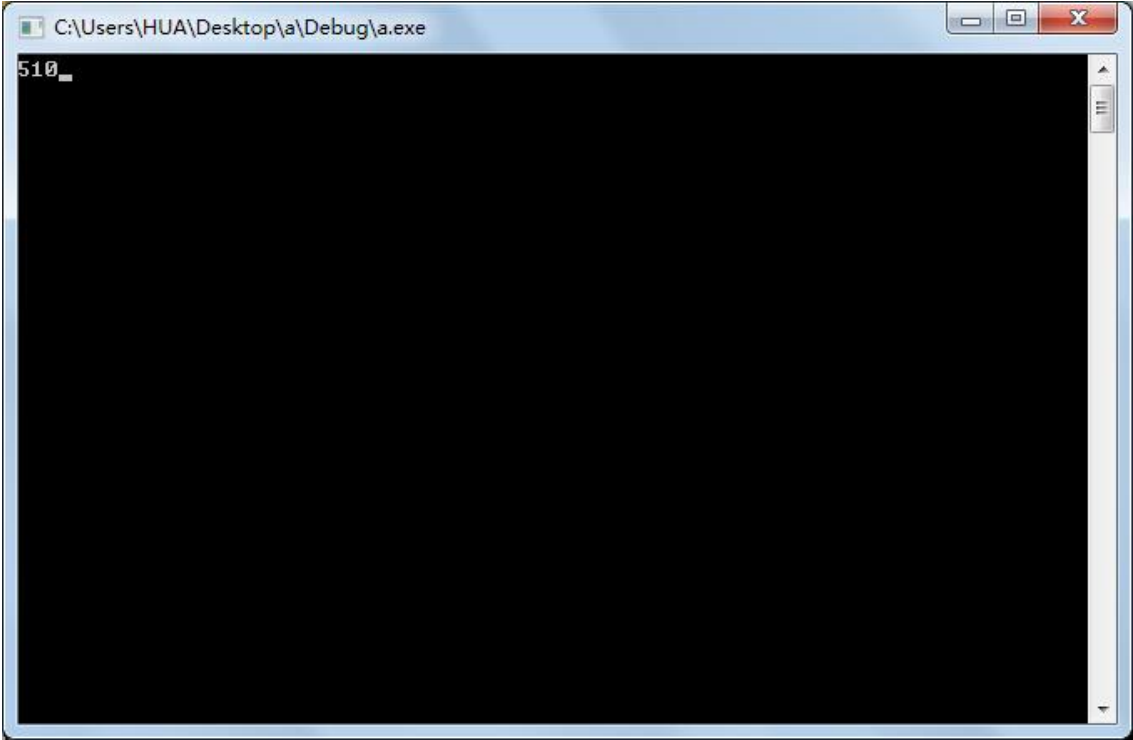
```
program a;  
var  
i,j:integer;  
procedure swap(var a:integer;var b:integer);  
var t:integer;  
begin  
    if a>b then  
        begin  
            t:=a;  
            a:=b;  
            b:=t;  
        end;  
end;  
  
begin  
i:=10;  
j:=5;  
swap(i,j);  
write(i);  
write(j);  
end.
```

符号表

Symbol table:

Variable Name		NestLevel	Location	Line Number
a	1	4	4	
b	1	8	4	
i	0	-8	3	
j	0	-12	3	
t	1	-8	5	

X86 汇编码执行结果



4.4 函数及过程递归测试

Pascal 源程序

```
program a;
var
i,j:integer;
function gcd(a:integer;b:integer):integer;
begin
    if b=0 then
        gcd:=a
    else
        gcd:=gcd(b,a mod b);
end;
begin
```

```
j:=gcd(36,24);  
write(j);  
end.
```

符号表

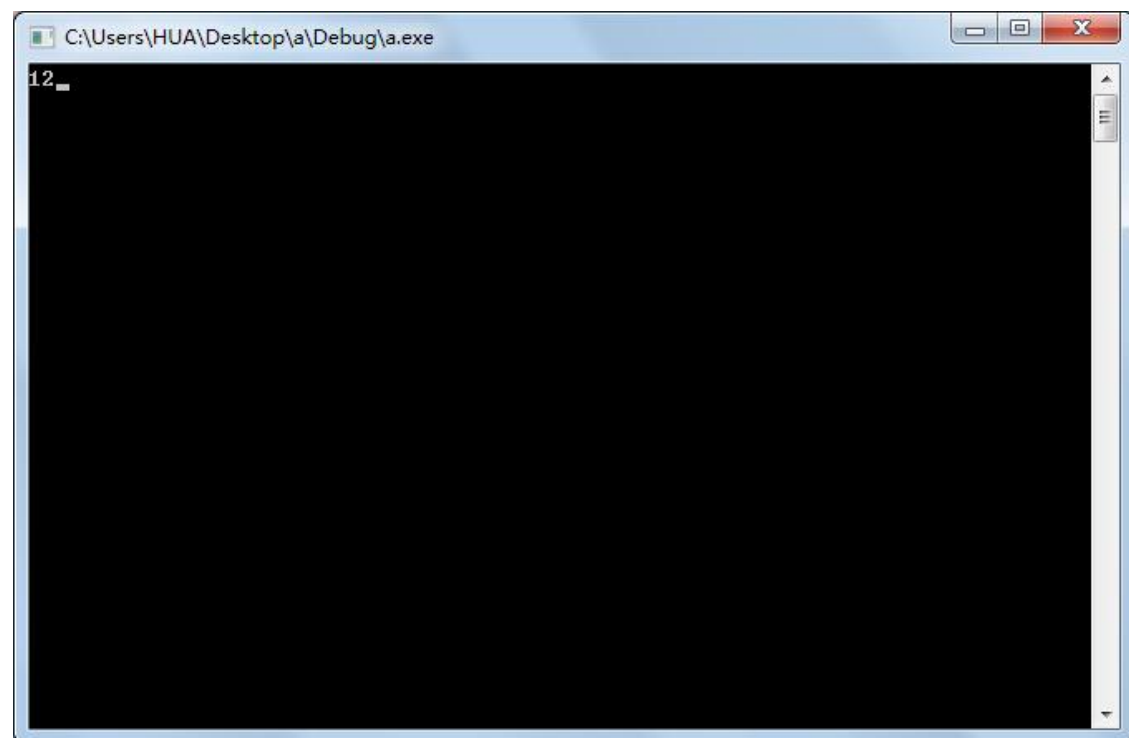
Symbol table:

Variable Name		NestLevel	Location	Line Number
a	1	4	4	4
b	1	8	4	4
i	0	-8	3	
j	0	-12	3	
gcd	1	12	4	4

Function Name		NestLevel	Return Type	Parameter
gcd	0	1	a	b

cong@cong-Lenovo-IdeaPad-Y400:~/project_final\$

X86 汇编码执行结果



4.5 函数及过程嵌套测试

Pascal 源程序

```
program a;  
var  
i:integer;  
function ff(u:integer):integer;
```

```

var x:integer;
  function gg(v:integer):integer;
    var y:integer;
      function hh(w:integer):integer;
        var z:integer;
        begin
          z:=10;
          hh:=w;
          x:=11;
          y:=12;
        end;
      begin
        gg:=hh(v+1);
        write(y);
      end;
    begin
      ff:=gg(u+1);
      write(x);
    end;

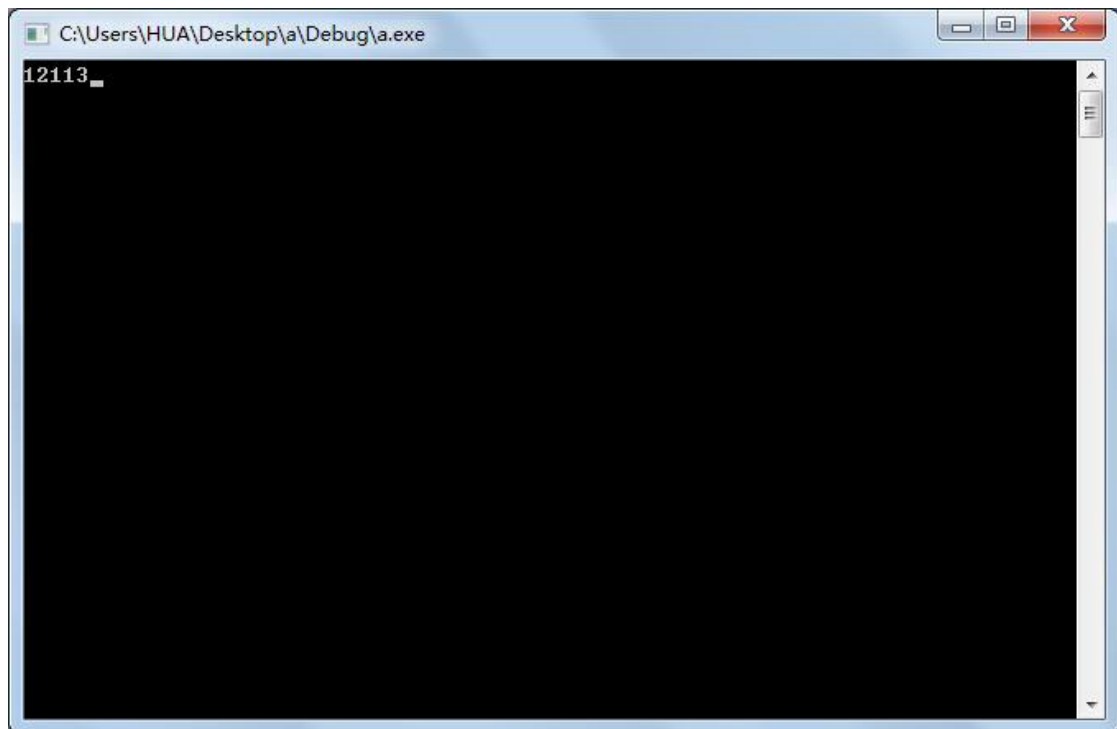
  begin
i:=ff(1);
write(i);
end.

```

符号表

Symbol table:				
Variable Name		NestLevel	Location	Line Number
ff	1	8	4	4
gg	2	8	6	6
hh	3	8	8	8
i	0	-8	3	
u	1	4	4	4
v	2	4	6	6
w	3	4	8	8
x	1	-8	5	
y	2	-8	7	
z	3	-8	9	
Function Name		NestLevel	Return Type	Parameter
ff	0	1	u	
gg	1	1	v	
hh	2	1	w	

X86 汇编码执行结果



4.6 系统过程测试

Pascal 源程序

```
program a;  
var  
i,j:integer;  
begin  
read(i);  
read(j);  
write(i+j);  
writeln(i+j);  
end.
```

符号表

```
Symbol table:  


| Variable Name |   | NestLevel | Location | Line Number |
|---------------|---|-----------|----------|-------------|
| i             | 0 | -8        | 2        |             |
| j             | 0 | -12       | 2        |             |



| Function Name | NestLevel | Return Type | Parameter |
|---------------|-----------|-------------|-----------|
|---------------|-----------|-------------|-----------|

  
cong@cong-Lenovo-IdeaPad-Y400:~/project_final$
```

x86 汇编码执行结果



5 总结

经过验证, 本编译系统实现的功能如下:

- ◆实现基本类型的常量定义
- ◆实现所有基本类型，以及枚举，数组，自定义结构体的类型定义
- ◆实现所有基本类型，以及枚举，数组，自定义结构体的变量定义
- ◆实现带基本类型参数的函数和过程
- ◆实现函数及过程的嵌套定义
- ◆实现子函数或子过程访问父函数或父过程变量
- ◆实现函数及过程的嵌套调用
- ◆实现 read, write, writeln 三个系统函数

6 分工

语法分析：谢宁宁

符号表：孔晗聪

代码生成：单才华

测试：单才华、孔晗聪