北京邮电大学

《编译原理与技术课程设计》
报　告

指导教师： **王雅文**

计算机学院
2023 年 5 月

目 录

Pascal-S语言编译程序的设计与实现

1．课程设计任务和目标

1.1. 目标1：针对课程设计的任务要求，对编译程序的设计与实现进行分解和细化，
培养学生的系统分析和设计能力，培养学生设计开发功能模块及计算机系统软件的能力。

1.2. 目标2：培养学生能够运用计算机开发环境和工具，对设计的编译程序原型系统进行功能仿真、测试，提高系统分析问题和解决问题的能力。

1.3. 目标3：培养学生合理分析和评价编译程序设计与实现相关的工程实践和复杂工程问题解决方案的可行性，及解决方案可能带来的安全、法律等方面的影响的能力，理解应承担的责任的能力。

1.4. 目标4：培养学生根据所承担的角色，组织、协调和带领团队开展工作的能力，在团队中完成自己承担的任务的能力。

1.5. 目标5：培养学生根据课程设计的任务要求，撰写设计文档和课程设计报告的能力，通过对课程设计成果进行陈述、展示和答辩，培养学生针对计算机及相关信息领域复杂工程问题与业界同行及社会公众进行有效沟通和交流的能力。

2．需求分析

2.1 系统概述

本编译器旨在将Pascal-S语言源代码编译成C语言目标代码，以便在C语言环境下进行编译和运行。编译器工作流程分为词法分析、语法分析、语义分析、及代码产生。由于不涉及汇编，省去IR产生、IR优化环节。

2.2 功能需求

2.2.1 词法分析

2.2.1.1 实现功能

识别Pascal语言源代码中的单词、符号和常量，并生成对应的词法单元序、列。词法分析作为语法分析的子程序。语法分析程序通过调用 yylex() 得到下一个 token 的类型以及相应的属性。

Pascal-S 识别的单词种类

- **关键字（Keyword）**：program, const, var, integer, boolean, real, char, array, procedure,

function, begin, end, of, if, then, else, while, for, to, do.

Note: Pascal 关键字不区分大小写

- **关键字单词模式**：以program为例,其模式表达为：[Pp][Rr][Oo][Gg][Rr][Aa][Mm]

注释（comment）

多行注释：{+注释内容}

单词模式：\{[\S\s]*?\}

由于需要记录token的 row 以及 column 来定位，所以直接采用匹配 { 进行相应的处理。

- **标识符（identifier）**

单词模式 [A-Za-z][A-Za-z0-9_]*

- 常数（constant）

单词模式：[0-9]+|[0-9]+.[0-9]+

- 赋值运算符（assign operator）: :=
- 关系运算符（relation operator）：>, <, >=, <=, <>, =
- 算数运算符（arithmetic operator）: +, -, or, *, /, mod, div, and
- 分隔符（delim）：:, ;, 逗号, 句号
- 空白（space）：" "|"\t"
- 其他（other）：[,], (,)

2.2.1.2 错误处理：

在进行词法分析时，可能会出现各种不同的错误，例如非法字符、缺失分隔符等。

我们采取以下几种错误处理方式

- 输出错误信息：如果在分析过程中出现错误，可以输出错误信息，告诉用户发生了什么错误。错误信息。

应该包括错误类型、错误行号和列号等相关信息，方便用户快速定位错误。

- 跳过错误：当遇到部分错误时，部分场景选择跳过错误并继续分析后续的代码。这种方法可以保证编译器的容错性。
- 恢复错误：在lex规则中，可以自定义规则处理错误。
- 终止分析：如果遇到无法处理的错误，可以调用yyterminate()函数终止分析。

2.2.2 语法分析

2.2.2.1 实现的语法结构

语法分析实现的额外功能：
包括begin end结构中的最后一条语句在pascal-s里面必须不能加上;但现在我们能加上分号，末尾可识别多个分号。拓展了语法结构，如repeat until，while …do，for downto 的实现，readln，writeln的实现，无参函数调用，无参过程调用，是否有括号（函数，过程定义）
compound_statement -> t_begin statement_list semicolon t_end

```
1. programstruct -> program_head ;  program_body .
2. program_head -> program id ( idlist ) | program id
3. program_body -> const_declarations var_declarations
subprogram_declarations compound_statement
4. idlist -> id | idlist , id
5. const_declarations -> 空 | const const_declaration ;
6. const_declaration -> id = const_value | const_declaration ;id =
   const_value

7. var_declarations -> 空 | var var_declaration ;
8. var_declaration -> idlist : type | var_declaration ; idlist
                              : type
9. type -> basic_type | array [ period ] of basic_type
10.basic_type -> integer | real | boolean | char
11.period -> digits .. digits | period ,  digits .. Digits
12.subprogram_declarations -> 空 | subprogram_declarations
                                                  subprogram ;
13.subprogram -> subprogram_head ; subprogram_body
14.subprogram_head -> procedure id formal_parameter
| function id formal_parameter : basic_type
15.formal_parameter -> 空 | ( parameter_list )
16.parameter_list -> parameter | parameter_list ; parameter

17.parameter -> var_parameter | value_parameter
18.var_parameter -> var value_parameter
19.value_parameter -> idlist : basic_type
20.subprogram_body -> const_declarations var_declarations
compound_statement
21.compound_statement -> begin statement_list end
22.statement_list -> statement | statement_list ; statement

23.statement -> 空
| variable assignop expression
| func_id assignop expression
| procedure_call
| compound_statement
| if expression then statement else_part
| for id assignop expression to expression do statement
| read ( variable_list )
| write ( expression_list )
24.variable_list -> variable | variable_list , variable
25.variable -> id id_varpart
26.id_varpart ->空 | [ expression_list ]
```

```
27.procedure_call -> id | id ( expression_list )
28.else_part -> 空 | else statement
29.expression_list -> expression | expression_list ,expression
30.expression -> simple_expression | simple_expression relop simple
   _expression
31.simple_expression -> term | simple_expression addop term
32.term -> factor | term mulop factor
33.factor -> num | variable | ( expression )
   | id ( expression_list ) | not factor| uminus factor
```

2.2.2.2 语法错误处理

1. 当产生式出现非预期token造成错误时，我们能够跳过这些错误避免对整体产生式的识别造成影响，从而能够归约出我们所想要的产生式，不受错误状态的影响。
2. 当产生式缺少一些必要token时，且只是缺少这些其他的归约必须的产生式都存在时，我们能够补全这些缺失，让编译器归约出当前这个产生式。

处理的语法包括了普通的语句，if … then …else，if … then … ，for … to … do，for … downto … do ，begin … end，赋值语句，repeat … until，while … do，函数参数，const 变量定义，var 变量定义等等。
详见语法分析部分。

2.2.3 语义分析

2.2.3.1 符号表

由于需要做到类型检查、作用域的判断（包括变量名是否出现重复或者作用域的冲突），在此过程中需要记录各标识符对应的类型和数据，故采用符号表的方式进行存储， 记录标识符和常量的声明和引用信息，并检查其类型是否匹配。对于程序块，需要建立新的符号表，并支持符号表的嵌套和作用域嵌套。

须在符号表中加入以下元素
- 标识符名称：表示程序中出现的标识符名称。
- 数据类型：表示标识符所对应的数据类型，包括整型、浮点型、字符型、数组等。
- 作用域：表示标识符所处的作用域，包括全局作用域、局部作用域等。
- 数值：针对整型、浮点型等
- 函数的参数
- 数组维度
- 其他属性，例如是否为引用变量，是否为常量

2.2.3.2 符号表

在这里类型检查只做静态检查即读入源程序但不执行源程序的情况下进行的检查

完成

- 检查操作的合法性和数据类型的相容性
  - 用户自定义函数的参数类型、返回值类型
  - 函数参数的数据类型和数量是否与函数声明或定义中的参数列表匹配
  - assign 赋值常量，condition and loop，for loop的条件判断Boolean
  - array下标，引用，procedure调用，参数等
  - 数值类型比较，运算检查
- 唯一性检查
  - 一个标识符在同一作用域中必须且只能被说明一次
  - CASE语句中用于匹配选择表达式的常量必须各不相同
  - 枚举类型定义中的各元素不允许重复

2.2.3.3 错误处理

- 显示出错信息。报告错误出现的位置和错误性质
- 终止分析：如果遇到错误，直接终止分析。

2.2.3.4 代码生成

代码生成将遍历AST树，结合符号表中存放如引用信息，类型信息，数组维度，函数调用等将pascal-s转化为C语言代码
包含以下内容：

- 头文件的引用
- 类型关键字的转化
- pascal主函数内的函数，进程的声明和定义转换为c全局函数
  例如，pascal程序

```
program main:
 function fun1(x:integer):integer;
 begin
        ...
    end;
 procedure fun2(y:integer);
 begin
        ...
    end;
begin
    ...
end.
```

函数和进程定义在主函数内，C语言无法这样定义，需将函数改为全局形式，如下：

```
int fun1(int x){
 ...
}
void fun2(int x){
 ...
}
```

```
int main(){
 ...
 return 0;
}
```

- pascal主函数局部变量转换为c全局变量

这和上一点同理。

- 引用参数在函数定义内变换为指针，调用时取地址

C语言无法使用C++的传引用，只能通过指针实现引用调用，例如，pascal进程

```
procedure swap(var x, y:integer);
var z:integer;
begin
    z:=x;
    x:=y;
    y:=z;
end;
```

需改写为如下C代码

```
void swap(int *x, int *y){
   int z;
   z=*x;
   *x=*y;
   *y=z;
}
```

- 数组维度的转换和偏移量的处理

pascal的数组定义如下：

```
name[l1..r1][l2..r2]...[ln..rn]
```

每一个l_i,r_i可以是任意值。而C数组下标从0开始，因此在保证数组空间完全一致的前提下，C代码对该数组定义为

```
name[r1-l1+1][r2-l2+1]...[rn-ln+1]
```

在调用数组时需要对表达式进行偏移，例如pascal代码

```
name[a1][a2]...[an]:=0;
```

改写的C代码为

```
name[a1-l1][a2-l2]...[an-ln]=0;
```

- 循环分支语句的转化，如repeat-until结构改为do-while结构
- pascal直接以函数名作为函数调用或函数返回值的处理

例如pascal函数

```
function func(x, y:integer):integer
begin
 if x<=0 or y<=0 then
  func:=1;
 else
  func:=func + func(x-1, y) + func(x, y-1);
end;
```

这里func:=0和func:=func+func(x-1, y)+func(x, y-1);中前两个func都是返回值，而func(x-1, y)+func(x, y-1)中两个func都是调用，正确的C代码应该是

```
int func(int x,int y){
 int _func;
    if(x<=0||y<=0){
     _func=1;
```

```
    }
    else{
     _func=_func+func(x-1,y)+func(x,y-1);
    }
    return _func;
}
```

2.3 非功能需求

- 可靠性 编译器应具有较高的可靠性，能够正确地识别、分析和生成Pascal语言源代码，不出现严重的错误。
- 性能 编译器应具有较高的性能，能够在合理的时间内完成编译任务。
- 可扩展性 编译器应具有较好的可扩展性，能够支持Pascal语言的不同版本和扩展，以及C语言的不同编译环境和平台。
- 易用性 编译器应具有良好的易用性，提供友好甚至实时的交互体验，以及详细的错误提示和调试信息。

2.4 开发环境

本项目采取CMake开发，CMake是一个跨平台的安装/编译工具，可以用简单的语句来描述所有平台的安装/编译过程，词法分析阶段采用flex编写，语法分析用Bison，语义分析，代码生成及各模块接口等均采取C++20标准编写

- cmake_minimum_required ——VERSION 3.16
- CMAKE_CXX_STANDARD 20
- Flex 2.6.4
- bison(GNU Bison) 3.5.1
- IDE:
  - VSCode 1.76.2
  - CLion 2022.3
- 测试工具: ctest version 3.16.3

3．总体设计

3.1 数据结构设计

3.1.1.AST树

TreeNode　定义了AST语法树节点，包含父节点id，token，所规约产生式，以及yyval所需的值。操作包括基本的获取text信息，获取产生式右部，获取类型，设置类型，获取父节点id，获取子节点迭代器，根据token获取子节点等。

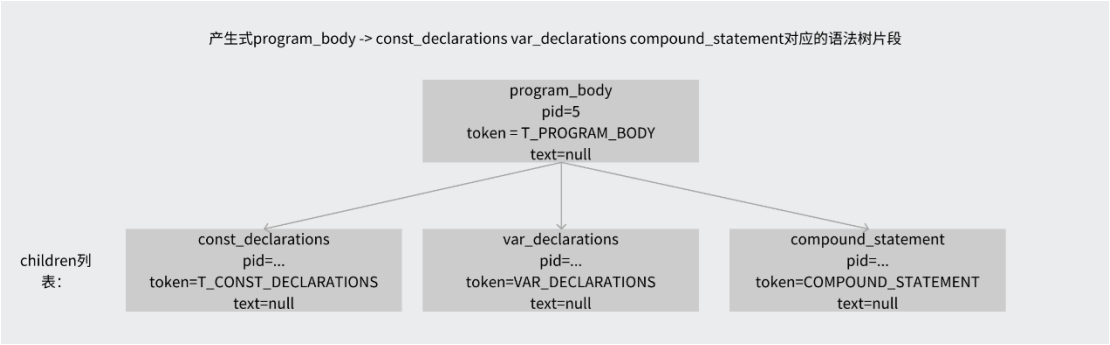
```
class TreeNode {
```

```
private:
    int pid{};
    Token token = T_ERROR;
    std::string text;  // for ID and Literal, empty for others
    std::vector<TreeNode*> children;
    symbol::BasicType type = symbol::TYPE_NULL;
public:
    TreeNode() = default;
    TreeNode(int pid, Token token, std::string text, std::vector<Tre
  eNode*> children = {}) :
             pid(pid), token(token), text(std::move(text)), children(
  std::move(children)) {}

    Token get_token() const { return token; }
    std::string get_text() const { return text; }
    TreeNode* get_child(int child_id) const { return children.at(chi
  ld_id); }
    std::vector<TreeNode*>& get_children() { return children; }
    symbol::BasicType get_type() const { return type; }
    void set_type(symbol::BasicType type) { this->type = type; }
    int get_pid() const { return pid; }
    auto children_begin() { return children.begin(); }
    auto children_end() { return children.end(); }
    void childrenPush(TreeNode* x) { children.push_back(x); }
    void set_pid(int x) { pid = x; }
    TreeNode* get_child_by_token(Token child_token) const;
};
```

语法树结构如下，

```
class Tree {
private:
    TreeNode* root;
public:
    Tree() = default;
    explicit Tree(TreeNode* root) :
         root(root) {}

    TreeNode* get_root() const { return root; }
};
```

这里给出一个示例



3.1.2.符号表

SymbolTableEntry定义了符号表的每一项，包括类型，是否是常量，是否是引用，是否是初始化，初始化值。其中ComplexType为enum，包含basic，array，record，function类型，extra_info包含诸如数组维度，record字段

名和类型，函数返回类型和参数等信息，采取struct展现，如

```
struct Param {
    BasicType type{};
    bool is_referred = false;
    Param() = default;
    Param(BasicType type, bool is_referred):
            type(type), is_referred(is_referred) {}
};
```

具体这些结构定义在详细设计报告中具体展现。

```
struct SymbolTableEntry {
    ComplexType type{};
    ExtraInfo extra_info;



    SymbolTableEntry() = default;
    SymbolTableEntry(ComplexType type, ExtraInfo extra_info):
            type(type), extra_info(std::move(extra_info)) {}
    explicit SymbolTableEntry(BasicType type, bool is_const = false,
   bool is_referred = false):
            type(ComplexType::TYPE_BASIC),
   extra_info(BasicInfo(type, is_const, is_referred)) {}
    SymbolTableEntry(BasicType type,
   std::vector<std::pair<size_t,size_t>> dims, bool is_const =
   false):
            type(ComplexType::TYPE_ARRAY),
   extra_info(ArrayInfo(type, std::move(dims), is_const)) {}
    explicit SymbolTableEntry(std::map<std::string, BasicType>
   fields):
            type(ComplexType::TYPE_RECORD),
   extra_info(RecordInfo(std::move(fields))) {}
    SymbolTableEntry(BasicType ret_type, std::vector<Param> params):
            type(ComplexType::TYPE_FUNCTION),
   extra_info(FunctionInfo(ret_type, std::move(params))) {}
};
```

SymbolTableNode定义了符号表的每一层，包括父节点，子节点，符号表。其中，符号与其表项采取map方式映射。接口包括获取父节点，获取子节点，判断是否有符号，获取符号，添加符号，添加子节点，添加父节点

```
class SymbolTableNode {
private:
    std::shared_ptr<SymbolTableNode> parent;
    std::vector<std::shared_ptr<SymbolTableNode>> children;
    std::map<std::string, std::shared_ptr<SymbolTableEntry>> entries
;
public:
    SymbolTableNode() = default;
```

```
    explicit SymbolTableNode(std::shared_ptr<SymbolTableNode> parent
) : parent(std::move(parent)) {}

    std::shared_ptr<SymbolTableNode> get_parent() const { return par
ent; }
    std::vector<std::shared_ptr<SymbolTableNode>> get_children() con
st { return children; }
    bool has_entry(const std::string& name) const { return entries.f
ind(name) != entries.end(); }
    std::shared_ptr<SymbolTableEntry> get_entry(const std::string& n
ame) const { return entries.at(name); }
    void add_entry(const std::string& name, const std::shared_ptr<Sy
mbolTableEntry>& entry) { entries.emplace(name, entry); }
};
```

SymbolTableTree则为当前作用域对应语法树的符号表表项，定义接口如进入新的作用域，退出当前作用域，查找符号，获取符号，添加符号

```
class SymbolTableTree {
private:
    std::shared_ptr<SymbolTableNode> root;
    std::shared_ptr<SymbolTableNode> current_node;
public:
    SymbolTableTree() = default;

    // Initialize the symbol table tree
    void initialize();

    // Enter a new scope
    void push_scope();

    // Exit the current scope
    void pop_scope();

    enum SearchResult {
        FOUND,
        NOT_FOUND,
        FOUND_IN_PARENT,
    };

    // Search the entry with the given name, from the current scope
  to the root scope
    SearchResult search_entry(const std::string& name);

    // Get the entry with the given name, from the current scope to
  the root scope
    std::shared_ptr<SymbolTableEntry> get_entry(const std::string& n
  ame);

    // Add an entry to the current scope
    void add_entry(const std::string& name, const std::shared_ptr<Sy
  mbolTableEntry>& entry);
};
```

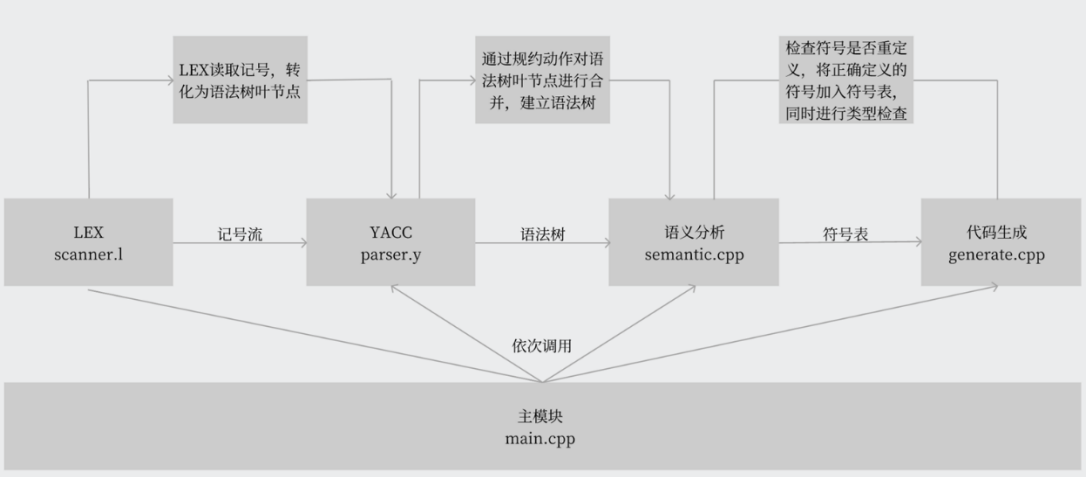
3.2 总体结构设计

3.2.1功能模块的划分与模块功能

本程序程序主要分为 Lex, Yacc 和 语义分析模块与C代码生成4个模块

- **词法分析：**编写lex——parser.y　识别Pascal语言源代码中的单词、符号和常量，并生成对应的词法单元序列。词法分析作为语法分析的子程序。语法分析程序通过调用 yylex() 得到下一个 token 的类型以及相应的属性。
- 语法分析：编写yacc——scannner.I　根据产生式设计规约，具体实现的产生式详见详细设计报告，在这里加入所规约的产生式pid以方便后续语义分析
- **语义分析：**在这个模块，我们采用符号表的方式进行存储，　记录标识符和常量的声明和引用信息，并检查其类型是否匹配。对于程序块，需要建立新的符号表，并支持符号表的嵌套和作用域嵌套。并且类型检查、作用域的判断也可与符号表的建立一同在dfs遍历AST语法树时完成
- **C代码生成：**代码生成将遍历AST树，结合符号表中存放如引用信息，类型信息，数组维度等将pascal-s转化为C语言代码

3.2.2模块关系



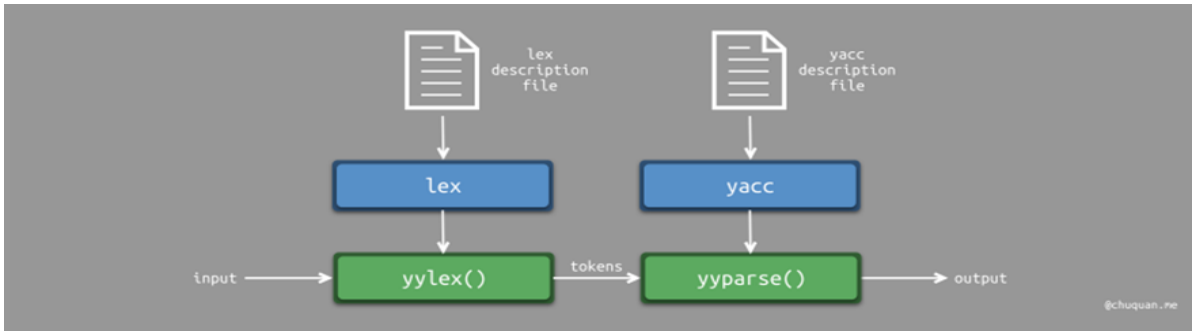
本程序的执行逻辑主要如下：

Pascal-S 源代码作为程序输入，由 Lex 模块对其作扫描以将其分析为符号流传递给 Yacc 模块，由 Yacc 模块对符号流进行 LR 语法分析，识别符号流中存在的语法错误并在部分情况下尝试作错误恢复，同时在语法分析过程中建立分析树。Lex与Yacc模块同步工作，工作时两模块共同维护一节点栈，用于辅助分析树的构建工作，以及用于将词法分析阶段所获取的必要信息如标识符名、常量名等记录到符号对应的分析树节点上，以便后续阶段使用。Lex 模块在识别出符号后将该符号对应的分析树节点入栈，Yacc模块在使用生成式进行规约时将生成式右边对应节点出栈，构造生成式左部符号对应节点并将其入栈。Lex与Yacc模块分析完成时，节点栈中唯一节点即保存了输入程序对应的分析树。语义分析模块先将词法分析与语法分析生成的分析树转化为抽象语法树(AST)，再利用该语法树进行语义检查，同时在检查过程中生成并维护符号表结构。最后C代码生成模块对语义分析阶段生

成的语法树进行遍历，同时结合符号表生成源程序对应的目标C代码。

3.2.3.模块之间接口

Lex与yacc协同工作原理如图



lex和yacc分别使用各自的描述文件生成词法分析器yylex()和语法/语义分析器yyparse() 。

yyparse()自身并不进行词法分析，而是调用yylex()进行词法分析。yylex()返回一个 token 号，表示 token 的类型。token 值则存储在 yylval 变量中。比如：token 的类型为算术运算符，token 的值为+。yyparse()则通过读取yylex()的返回值以及yylval变量分别获取token类型和token值。

yacc可根据所给文法产生式，构建AST树，定义结构如下：

节点定义如下：

语义分析环节需将每个符号对应type填入，并检查类型是否匹配，标识符和常量的声明和引用信息等。

代码生成需用到所生成的AST树及语义分析阶段所产生的符号表，如下所示：

Node节点结构定义在上文数据结构部分已经展示。

3.3 用户接口设计

本程序默认接受源代码 Pascal-S 代码作为标准输入，将生成目标代码 C 代码输出到标准输出，同时将编译生成过程中产生的调试信息输出到标准错误，三个输入/输出流均支持重定向到文件。

本程序为命令行工具，项目构建后可执行文件为 pascals-to-c，命令语法为：

pascals-to-c [-h] [-i <file>] [-o <file>] [-l <file>]

各命令选项：

- -h,--help：查看帮助信息
- -i,--input <file>：重定向输入代码为指定文件
- -o,--output <file>：重定向输出目标代码到指定文件
- -l,--log <file>：重定向调试信息到指定文件

3.4 错误处理

Lex和C代码生成部分只做错误检测，若识别到错误则输出错误信息并中止。Yacc部分采用错误检测+错误恢复，采用panic策略进行错误的恢复，即持续删除输入串当前符号，直到可以继续进行分析。具体如下：

- 找到一个可以作为恢复点的语法单元

在遇到语法错误时，Panic-Mode Recovery会尝试跳过一些语法单元，直到找到一个可以作为恢复点的语法单元。可以作为恢复点的语法单元通常是具有较强的同步性质的语法单元，例如语句之间的分号、代码块的结束符号等。

- 跳过语法单元

一旦找到了恢复点，Panic-Mode Recovery会跳过一些语法单元，直到达到恢复点。在跳过语法单元的过程中，可以根据需要向用户发出一些警告信息，以便用户能够尽早地发现并修复语法错误。

- 恢复语法分析

一旦达到了恢复点，Panic-Mode Recovery会尝试从该语法单元开始重新进行语法分析。由于跳过了一些语法单元，可能会导致后面的语法分析过程出现更多的错误，但是这种方法仍然可以使得语法分析能够继续进行下去，从而让用户能够看到更多的错误信息。

4．详细设计

4.1.词法分析

词法分析部分我们采用lex处理， 当 Lex接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。 它一次读入一个输入字符，直到找到一个匹配的模式。 如果能够找到一个匹配的模式，Lex 就执行相关的动作，返回一个Token。 另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

定义Token如下：

```
    enum Token {
1.          T_ERROR = -1,
2.          T_ID,
3.          T_NUM,
4.          T_LITERAL_INT,
5.          T_DOUBLE_VALUE,
6.          T_DOUBLE,
7.          T_LITERAL_CHAR,
8.          T_LITERAL_BOOL,
9.          T_LITERAL_STRING,
10.         T_RELOP,
11.         T_SEPERATOR,
12.         T_OR_OP,
13.         T_QUATEOP,
14.         T_ADDOP,
15.          T_MULOP,
16.         T_ASSIGNOP,
17.         T_KEYWORD,
18.         T_PROGRAM_STRUCT,
19.         T_PROGRAM_HEAD,
20.         T_PROGRAM_BODY,
21.         T_CONST_DECLARATIONS,
22.         T_CONST_DECLARATION,
23.         T_CONST_VALUE,
24.         T_VAR_DECLARATIONS,
25.         T_VAR_DECLARATION,
26.         T_COMPOUND_STATEMENT,
```

```
27.         T_TYPE,
28.         T_BASIC_TYPE,
29.         T_PERIOD,
30.         T_FORMAL_PARAMETER,
31.         T_PARAMETER_LIST,
32.         T_PARAMETER,
33.         T_VALUE_PARAMETER,
34.         T_VARIABLE_LIST,
35.         T_ID_VARPART,
36.         T_EXPRESSION_LIST,
37.         T_PROCEDURE_CALL,
38.         T_ELSE_PART,
39.         T_SUBPROGRAM,
40.         T_SUBPROGRAM_DECLARATIONS,
41.         T_SUBPROGRAM_DECLARATION,
42.         T_SUBPROGRAM_HEAD,
43.         T_SUBPROGRAM_BODY,
44.         T_ARGUMENTS,
45.         T_STATEMENT_LIST,
46.         T_STATEMENT,
47.         T_VARIABLE,
48.         T_EXPRESSION,
49.         T_SIMPLE_EXPRESSION,
50.         T_TERM,
51.         T_EQUALOP,
52.         T_FACTOR,
53.         T_NOTOP,
54.         T_PROGRAM,
55.         T_CONST,
56.         T_VAR,
57.         T_PROCEDURE,
58.         T_FUNCTION,
59.         T_BEGIN,
60.         T_END,
61.         T_IF,
62.         T_THEN,
63.         T_ELSE,
64.         T_WHILE,
65.         T_DO,
66.         T_FOR,
67.         T_TO,
68.         T_DOT,
69.         T_DOWNTO,
70.         T_REPEAT,
71.         T_UNTIL,
72.         T_CASE,
73.         T_READ,
74.         T_WRITE,
75.         T_BOOLEAN,
76.         T_INTEGER,
77.         T_CHAR,
78.         T_REAL,
79.         T_OF,
80.         T_ARRAY,
81.         T_IDLIST,
82.         T_VAR_PARAMETER,
83.         T_LEFTPAREN,
84.         T_RIGHTPAREN,
```

```
        T_LEFTBRACKET,
        T_RIGHTBRACKET,
        T_SEMICOLON,
        T_COMMA,
        T_COLON,
        DOT,
        T_SUBOP,
        T_WRITELN,
        T_READLN,
        T_LONGINT,
        T_SHORTINT,
        T_BYTE,
        T_SINGLE,
        T_STRING,
    T_TRUE,
    T_FALSE,
};
```

我们计划采用std::vector<std::string> comment_vector存储注释，每个终结符Token存储在之前的{[^}]*}形式的注释。同时，由于我们希望记录word_count，line_count，char_count等信息，我们在YY_USER_ACTION 中定义这些操作

```
#define YY_USER_ACTION \
yylloc.first_line   = curr_line;\
yylloc.first_column = curr_col;\
{\
    char * s; \
    for(s = yytext; *s != '\0'; s++)\
    {\
        if(*s == '\n'){\
        curr_line++;\
        curr_col = 1;\
        }\
        else{\
        curr_col++;\
        }\
    }\
}\
yylloc.last_line   = curr_line;\
yylloc.last_column = curr_col-1;\
```

Lex 部分正则匹配如下：

```
    letter [a-zA-Z]
    digit [0-9]
    digits ({digit})+
    optional_fraction "."{digits}
    num {digits}
    float_num {digits}{optional_fraction}
    literal {quateop}({letter}|{digit})*{quateop}
    id {letter}({letter}|{digit})*
    comment \{[^}]*\}
    literal_char {quateop}({letter}|{digit}|{others}){quateop}
    literal_string {quateop}({letter}|{digit}|{others})*{quateop}
```

当匹配到时，我们在log中debug级别输出匹配和定位信息，并创建对应Token的Tree_node

```
{num}
```

```
{log( std::string( "num:") + yytext , yylineno, DEBUG);yylval.token_
Tree = new tree::Tree(new tree::TreeNode(tree::leaf_pid,tree::T_LITE
RAL_INT,yytext ,{yylloc.first_line, yylloc.first_column, yylloc.last
_line, yylloc.last_column}, comment_vector)); comment_vector.clear()
;
return num;}
```

其中，treenode记录pid为叶节点，所属类型，text信息，定位信息，以及前面的注释信息。num变量用于将与标记关联的语义值从词法分析器传递到语法分析。符号的语义值在 yacc 操作中作为$1, $2等访问
特别的，为处理多行注释，我们定义了BLOCK_COMMENT状态，若INITIAL遇到{进入BLOCK_COMMENT，在注释结束即}退出。

```
<INITIAL>{
    "{" {BEGIN BLOCK_COMMENT; char_count += yyleng;}
}

<BLOCK_COMMENT>{
    <<EOF>> {log("Unfinished block comment", yylineno, ERROR); retur
n 0;}
    "{" {BEGIN INITIAL; char_count += yyleng;}
}
```

在词法分析环节，我们定义的错误恢复包含unexpected_char 和注释未完整unwrapped_string_error。对于前者，我们输出Unexpected character: 然后跳过继续分析，对于后者，我们直接匹配到行尾，然后补全后}输出错误位置，然后同注释一致新建叶节点，跳过该行继续分析。

4.2 语法分析

4.2.1语法树节点

语法树节点类，包括4个私有变量。其中 pid 用于标记该节点对应的产生式； token 表示其记号名称， text 表示所有终结符在pascal代码中对应的字符串，对于非终结符为空， children 表示其通过产 生式生成的token序列对应的节点。
该类还包括一系列函数，其具体含义见如下代码注释

```
class TreeNode {
private:
    PID pid{};                                  // 产生式编号，因一个
token由多个产生式规约需要
    Token token = T_ERROR;                      // 节点对应的词法单元的类
型
    std::string text;                           // 对于标识符和字面量，存
储其文本值；对于其他类型的节点，该值为空字符串
    std::vector<TreeNode*> children;            // 子节点的指针向量
    symbol::BasicType type = symbol::TYPE_NULL; // 存储节点的数据类型
    Position position;                          // 节点在源代码中的位置信
息
    std::vector<std::string> comments;          // 节点前面的注释信息
```

```
public:
    TreeNode() = default;
    // 构造函数
    TreeNode(PID pid, Token token, std::string text, Position
position, std::vector<std::string> comments={},
std::vector<TreeNode*> children = {}) :
            pid(pid), token(token), text(std::move(text)),
position(position), comments(std::move(comments)),
children(std::move(children)) {}

    // 返回节点对应的词法单元类型
    Token get_token() const { return token; }

    // 返回节点所在行号
    int get_line() const { return position.first_line; }

    // 返回节点位置信息
    Position get_position() const { return position; }

    // 返回节点值
    std::string get_text() const { return text; }

    // 根据子节点编号获取对应的子节点指针
    TreeNode* get_child(int child_id) const { return
children.at(child_id); }

    // 返回子节点向量
    std::vector<TreeNode*>& get_children() { return children; }

    // 返回节点的数据类型
    symbol::BasicType get_type() const { return type; }

    // 设置节点的数据类型
    void set_type(symbol::BasicType type) { this->type = type; }

    // 返回产生式编号
    PID get_pid() const { return pid; }

    //用于迭代child
    auto children_begin() { return children.begin(); }
    auto children_end() { return children.end(); }

    // 向子节点指针向量末尾添加一个节点
    void childrenPush(TreeNode* x) { children.push_back(x); }
```

```
    // 设置产生式编号
    void set_pid(PID x) { pid = x; }

    // 根据词法单元类型查找对应的子节点指针
    TreeNode* get_child_by_token(Token child_token) const;

    // 返回节点前面的注释信息
    std::vector<std::string> get_comments() const { return comments;
}
};
```

4.2.2 语法分析主体部分（parser.y）

由若干形如下列表示记号流产生式及其动作的代码组成：

由于产生式繁多，以下仅围绕其中一个产生式介绍：

```
program_body :| const_declarations var_declarations
compound_statement{
    tree::Position pos = {$1->get_root()->get_position().first_line,
$1->get_root()->get_position().first_column,
$3->get_root()->get_position().last_line,
$3->get_root()->get_position().last_column };
    log( "Use production: subprogram_body -> const_declarations
var_declarations compound_statement", pos, DEBUG);
    $$ = tools::reduce({$1, $2, $3}, pos,
tree::subprogram_body__T__const_declarations__var_declarations__comp
ound_statement
    , tree::T_SUBPROGRAM_BODY);
}
```

在这段代码中：
第一行 program_body : const_declarations var_declarations compound_statement 表示一个 产生式 program_body -> const_declarations var_declarations compound_statement,这是由 pascal_S语言语法决定的定式。
注释中pid=5表示以该产生式规约得到的program_body记号在语法树节点中的pid是5(见1.2)。
std::cerr 用于输出调试信息。
$$ = tools::reduce({$1, $2, $3}, 5, tree::T_PROGRAM_BODY); 表示一个规约动作，{$1,$2,$3} 表示产生式右侧符号对应的语法树，5表示pid ,对应2.2; T_PROGRAM_BODY表示记号类型，整个 reduce操作将使产生式右侧符号对应的语法树节点成为产生式左侧符号对应语法树节点的根。以下是reduce的代码段：

```
tree::Tree* reduce(std::initializer_list<tree::Tree*> list, int pid,
 tree::Token token){
    //使list中的每个树成为新生成的token所在的语法树根节点的子节点
```

```
    auto* tnode = new tree::TreeNode(pid, token, turn_token_text(tok
en));//新生成token所在的语法树根节点
    tnode->set_pid(pid);//设置该根节点的pid
    for (auto it = list.begin(); it != list.end(); ++it) { //枚举
list中所有树
        tnode->childrenPush((*it)->get_root());    //将其树根变成
tnode的孩子节点
    }
    return new tree::Tree(tnode);        //返回以tnode为根的语法树
}
```

4.2.3 生成语法树

主要维护一个作为树根的语法树节点：
示例如下：

```
class Tree {
private:
    TreeNode* root;
public:
    Tree() = default;
    explicit Tree(TreeNode* root) :
         root(root) {}

    TreeNode* get_root() const { return root; }
};
```

4.2.4 错误处理及恢复

在查阅了yacc的文档后，以及对实际的情况进行合理分析，我们语法分析环节采用Panic-Mode Recovery模式进行错误处理和错误恢复。通过实现了错误处理和错误恢复，我们能够避免一出现错误就停止继续分析，而是进行适当的修复，从而能够继续进行分析。而错误恢复的，具体策略如下

- 找到一个可以作为恢复点的语法单元
  在遇到语法错误时，Panic-Mode Recovery会尝试跳过一些语法单元，直到找到一个可以作为恢复点的语法单元。可以作为恢复点的语法单元通常是具有较强的同步性质的语法单元，例如语句之间的分号、代码块的结束符号等。
- 跳过语法单元
  一旦找到了恢复点，Panic-Mode Recovery会跳过一些语法单元，直到达到恢复点。在跳过语法单元的过程中，可以根据需要向用户发出一些警告信息，以便用户能够尽早地发现并修复语法错误。
- 恢复语法分析
  一旦达到了恢复点，Panic-Mode Recovery会尝试从该语法单元开始重新进行语法分析。由于跳过了一些语法单元，可能会导致后面的语法分析过程出现更多的错误，但是这种方法仍然可以使得语法分析能够继续进行下去，从而让用户能够看到更多的错误信息。

总的来说，我们实现了

1. 当产生式出现非预期token造成错误时，我们能够跳过这些错误避免对整体产生式的识别造成影响，从而能够归约出我们所想要的产生式，不受

错误状态的影响。

2. 当产生式缺少一些必要token时，且只是缺少这些其他的归约必须的产生式都存在时，我们能够补全这些缺失，让编译器归约出当前这个产生式。

加入了这些功能后，便能充分提高编译器的鲁棒性，尽可能地去检测出更多的错误，尽可能让编译器的分析深入，而不是在浅层就停止运作，满足用户一次编译看到尽量多的错误的需求。

具体的实现过程如下：
以const_declaration-> id equalop const_value这个产生式为例，针对这个产生式，有可能在识别到id后发现id后面跟随的语法符号无法匹配到const_declartion的所有产生式。假设此时是在等待归约出const_declaration这个token，如果所有的产生式都无法归约，那么此时会进入error状态。在这个状态，按照yacc文档，在程序中，可以通过error这个token来表示错误状态。据此，我们设计含有 error的产生式如下：

```
const_declaration -> id error equalop const_value
const_declaration -> id error const_value
```

第一个产生式的作用是当id和’=’之间出现了非预期的错误token时，我们能够及时跳过，不将这些错误的token加入到我们的归约项中，直到遇到equalop和const_value后，才将error前出现的token和后面的token进行归约。
注：我们在具体实现的时候，在错误处理后，需要加入yyerrok;这条语句，这条语句告诉yacc，我们当前能够退出错误状态，能够继续正常处理其他产生式了。
第二个产生式的作用处理当id后面出现错误的字段且遗漏了equalop的情况，error在产生式中的作用和第一个产生式基本一致，我们设计这个产生式就能够避免缺少equalop而导致语法分析提前终止。“适当补上缺失”这个策略能够让编译器进一步拥有更强的鲁棒性。

我们举几个例子进一步说明
比如当待分析的语句为const i test ****=2;
程序在识别到test这个token时无法继续匹配产生式，从而语法分析报错，此时进入error状态，继续接收错误如****，直到遇到了=和2即equalop和num，匹配到我们预设的error产生式，从而归约，归约后一切恢复正常，程序能够继续分析。

```
DEBUG t_const:const: line 2
DEBUG blank: : line 2
DEBUG id:i: line 2
DEBUG blank: : line 2
DEBUG id:test: line 2
ERROR Error: syntax error at line: 2, encountering unexpected word test: line 2
DEBUG blank: : line 2
DEBUG mulop:*: line 2
DEBUG mulop:*: line 2
DEBUG mulop:*: line 2
DEBUG mulop:*: line 2
DEBUG equalop:=: line 2
DEBUG num:2: line 2
DEBUG Use production: const_value -> num: line 2, column 19
ERROR error that lack of equalop is fixed and use production: const_declaration -> id = const_value: line 2, column 7
DEBUG semicolon:;: line 2
```

类似的我们还做了以下产生式：

```
var_declaration ->  idlist colon error type
const_value ->  addop error num
```

```
const_value ->  subop error num
const_value ->   addop error double_value
const_value ->   subop error double_value
idlist ->  idlist error id
program_head ->  t_program error id
var_declaration -> idlist colon error type
var_declaration -> idlist error type
type ->  t_array leftbracket error period rightbracket t_of basic_type
type ->  t_array error leftbracket period rightbracket t_of basic_type
type ->  t_array leftbracket period error rightbracket t_of basic_type
type -> t_array leftbracket period rightbracket error t_of basic_type
period ->  num error num
period ->  num t_dot error num
subprogram_declarations -> subprogram error semicolon
subprogram_head -> t_function id formal_parameter error basic_type
subprogram_head -> t_function id formal_parameter error colon
basic_type
subprogram_head ->  t_procedure id error formal_parameter
formal_parameter ->  leftparen parameter_list error rightparen
value_parameter -> idlist error basic_type
value_parameter ->  idlist error colon basic_type
subprogram_body ->  const_declarations error compound_statement
subprogram_body ->  var_declarations error compound_statement
subprogram_body ->  const_declarations var_declarations error
compound_statement

subprogram_body ->  const_declarations error var_declarations
compound_statement
statement ->  variable error expression
statement ->  t_if expression error statement
statement ->  t_if expression error statement else_part
statement ->  t_if expression error t_then statement else_part
statement ->  t_while expression error t_do statement
statement ->  t_while expression error statement
statement ->  t_repeat statement_list error t_until expression
statement ->  t_repeat statement_list error expression
statement ->  t_readln error leftparen variable_list rightparen
statement ->  t_readln error variable_list rightparen
variable_list ->  variable_list error comma variable
variable_list ->  variable_list error variable
variable ->  id error id_varpart
id_varpart ->  leftbracket expression_list error rightbracket
```

4.3.语义分析

由于语义分析需要进行类型推断，对作用域进行判断，变量是否存在作用域的冲突以及变量是否重复定义，故在语义分析模块，我们主要设计符号表的方式进行存储相关，包括标识符和常量的声明，引用等，并利用符号表来进行相关错误检测。

总的来说，我们围绕各个功能需求来对符号表的结构进行相关设计。

4.3.1 基本数据结构和定义：

我们将类型分为基础类型（BasicType）和复杂类型（ComplexType），在实现中，我们使用两个枚举类型（enum）来分别表示。BasicType包括int，single，double，shortint，longint，byte，bool，string，char。ComplexType包括basictype，array，record，function。（具体参见代码symbol.h）。同时为了在类型检查中方便得出表达式的类型类别，我们引入类型的大类（TYPE_CATEGORY_INT, TYPE_CATEGORY_FLOAT, TYPE_CATEGORY_CHAR, TYPE_CATEGORY_BOOL, TYPE_CATEGORY_STRING）使用TypeCategory枚举类型来表示。

由于实现过程中，需要存储各个变量的类型信息、引用信息、是否为常量等，且实际上不同的变量类型应当存储的信息又有所不同，如array数组类型需要存储维度信息，普通变量又不需要，以及function函数过程变量需要存储它自身的参数信息等等。故我们设计了四类info来存储表示各类型信息，分别为BasicInfo，ArrayInfo，RecordInfo，FunctionInfo。另外还特别使用Param来存储函数形参信息，因为形参涉及引用，又和其他变量有所区别。其中BasicInfo存储了如是否常量(is_const)，是否是引用(is_refered)，以及变量的类型basic。

其他的这里仅展示ArrayInfo，剩余的参见代码symbol.h

```
struct ArrayInfo {
    BasicType basic{};
    std::vector<std::pair<int,int>> dims;
    bool is_const = false;
    ArrayInfo() = default;
    ArrayInfo(BasicType basic, std::vector<std::pair<int,int>> dims,
bool is_const):
            basic(basic), dims(std::move(dims)), is_const(is_const) {}
};
```

另外ExtraInfo是类型安全的union，由其他4种类型联合。

```
typedef std::variant<std::monostate, BasicInfo, ArrayInfo, RecordInfo,
FunctionInfo> ExtraInfo;
```

4.3.2 符号表

为了实现符号表，我们设计了符号表表项（SymbolTableEntry），符号表节点（SymbolTableNode）和符号表树（SymbolTableTree）。符号表树以树的形式存储符号表节点，主要存储树的根节点 ，每个符号表节点代表一个作用域内的所有变量信息，每个变量信息通过向符号表节点插入一个符号表表项来表示。

4.3.2.1 符号表表项（SymbolTableEntry）

```
struct SymbolTableEntry {
    ComplexType type{};       // 存储该表项的类型信息
    ExtraInfo extra_info;       // 存储该表项的额外信息
}
```

4.3.2.2 符号表节点（SymbolTableNode）

符号表节点类（这里省略代码，参见symbol.h）记录了该节点的父亲（parent），该节点的子节点（children），该节点的表项（entries），该节点的名字（scope_name，在分析函数时需要使用），以及traverse_sequence（遍历顺序）

可使用的外部接口包括

- 获取父节点get_parent()，
- 加入一个子节点add_child(child)，
- 判断是否有一个名为name的表项记录has_entry(name)，
- 加入一个表项记录add_entry(name)，
- 获取一个表项记录get_entry(name)，
- 获取当前域的名字get_scope_name()，
- 获取所有的表项get_entries()
- 获取下一个儿子next_child()
- 初始化遍历顺序initialize_traverse_sequence()

4.3.2.3 符号表树（SymbolTableTree）

主要存放的是树的根节点，和记录当前节点。提供的接口包括面向语义分析和代码生成。

```
class SymbolTableTree {
private:
    std::shared_ptr<SymbolTableNode> root; // 树的根节点
    std::shared_ptr<SymbolTableNode> current_node;  // 当前节点
public:
    SymbolTableTree() = default;

    //  初始化符号表树
    void initialize();

    // 创建并进入下一个子节点（语义 分析使用）
    void push_scope(BasicType return_type, const std::string&
scope_name);

    // 从当前节点退出
    void pop_scope();

    //进入下一个子节点（提供给代码生成的接口 ）
    void next_scope();

    // 搜寻表项的结果表示
    enum SearchResult {
        FOUND,
        NOT_FOUND,
        FOUND_IN_ANCESTOR,
    };
```

```
    // 获取当前域名
    std::string get_scope_name() const { return
current_node->get_scope_name(); }

    // Search the entry with the given name, from the current scope to
the root scope
    // 根据给定的变量名name从当前域遍历至根域搜寻表项，此时结果未知，需要返回
搜寻结果
    SearchResult search_entry(const std::string& name, bool
ignore_scope_name = false);

    //根据给定的变量名name从当前域遍历至根域获取表项（注：和上面不同，此时已知
一定有该表项）
    std::shared_ptr<SymbolTableEntry> get_entry(const std::string&
name, bool ignore_scope_name = false);

    // 将一个表项加入 到当前作用域中
    void add_entry(const std::string& name, const
std::shared_ptr<SymbolTableEntry>& entry);

    std::shared_ptr<SymbolTableNode> get_current_node() const { return
current_node; }
};
```

4.3.3 实现语义分析

主要实现逻辑在dfs_analyze_node函数内，主要完成两个任务：**建立符号表和进行类型推断**。通过自上往下遍历语法分析得到的ast，建立符号表树。具体即不断新建符号表节点（即开辟新作用域），并将相关的变量信息以表项的形式插入到对应的符号表内，**再遍历子节点**。其次是自底向上进行类型推断，先遍历完子节点，再结合子节点的信息进行类型推断。

4.3.3.1 建立符号表

即在合适的时机开辟新的子作用域，根据pascal-s产生式可知即在program_head-> program id，或者program_head-> program id (idlist) 以及新进入一个函数或过程，如subprogram_head-> function id formal_parameter : basic_type，需要新开辟一个符号表节点，并连接上。
且除了需要判断函数或者过程名字是否有重复外还需要判断函数/过程的形参和局部变量是否有冲突，一旦冲突即进行报错。插入表项使用add_entry接口。加入新的作用域使用push_scope接口。当遇到const的声明语句时，还需要针对声明的变量的is_const信息进行修改。
在实现时，采用switch case 结构，对当前ast节点的产生式编号（我们在实现时使用枚举类型来存储和表示，直观同时避免修改时的混乱）
以伪代码形式展示我们的具体实现设计：
symbol_table_tree是当前维护的符号表树，

node是当前的ast节点，node->get_pid()获知当前的产生式编号。

```
switch (node->get_pid()) {
        case
tree::programstruct__T__programhead_semicolon__programbody_dot:  此时是
刚进入程序，初始化符号表
        case tree::program_head__T__t_program__id:
            symbol_table_tree.add_entry(id,null)符号表加入表项，内容是
program的id，类型为空
        case
tree::program_head__T__t_program__id_leftparen__idlist__rightparen: {
            symbol_table_tree.add_entry(id,null)符号表加入表项，内容是
program的id，类型为空
            循环判断当前ast节点的子节点的id是否已经出现，如果没有则加入到符号
表内
            }
        }
        case tree::const_declaration__T__id__equalop__const_value:
        case
tree::const_declaration__T__const_declaration__semicolon__id__equalop__
const_value: {
            const声明，如果有重复定义的变量则报错，否则加入const变量到当前符
号表，并设置is_const
        }
        case
tree::subprogram_head__T__t_function__id__formal_parameter__colon__basi
c_type:
        case
tree::subprogram_head__T__t_procedure__id__formal_parameter:
        case
tree::subprogram_head__T__t_function__id__colon__basic_type:
        case tree::subprogram_head__T__t_procedure__id: {
            函数过程产生式，需要对函数，过程的id进行重复检查，同时设置返回值
类型，对函数形参进行检查，如果没问题则加入到符号表内。
        }
        default:
            break;
```

4.3.3.2 遍历子节点

node->get_children为当前ast的各个子节点，遍历每个子节点。

```
for (auto child: node->get_children()) {
        dfs_analyze_node(child);
}
```

4.3.3.3 类型推断

首先需要针对不同的语句进行不同的判断，如赋值语句，左值和右值的考虑，有些量不能当做左值；另外类型不匹配也不能进行赋值。就举赋值语句为例，先判断statement的左部是否是可赋值的量（通过调用check_variable_assignable），比如是否是const，是否是函数调用，然后再获取表达式的左值和右值的类型left_type，right_type，最后通过调用check_type_assignable(left_type, right_type)的接口来判断是否能够赋值。check_type_assignable完成对两个类型能否相容的判断，比如byte，shortint，longint之间相容，single和double之间的相容。我们在实现时使用的是比较左右值类型的大类(即type_category)。

其余分析见下面**伪代码**，我们针对每种需要做类型推断的产生式进行如下讨论：

以下是我们对产生式进行分类并且对此的处理

我们用Switch case来实现我们的主体框架，根据node->get_pid()来分析

```
switch (node->get_pid())
```

例1：

```
        case
tree::subprogram__T__subprogram_head__semicolon__subprogram_body:
```

过程/函数的结束，该产生式说明了可以退出当前作用域，故直接退出。symbol_table_tree.pop_scope();

例2：

```
        case tree::statement__T__variable__assignop__expression:
```

该产生式是一个赋值语句，先判断statement的左部是否是可赋值的量（通过调用check_variable_assignable），比如是否是const，是否是函数调用，然后再获取表达式的左值和右值的类型left_type，right_type，最后通过调用check_type_assignable(left_type, right_type)的接口来判断是否能够赋值。

例3：

```
case
tree::statement__T__t_if__expression__t_then__statement__else_part:
        case tree::statement__T__t_if__expression__t_then__statement:
        case
tree::statement__T__t_while__T__expression__t_do__statement:
        case
tree::statement__T__t_repeat__statement_list__t_until__expression:
```

该语句是一个条件判断或while、repeat循环语句，需要检查这些语句的expression部分是否是bool类型。

例4：

```
 case
tree::statement__T__t_for__id__assignop__expression__t_to__expression__
t_do__statement:
        case
tree::statement__T__t_for__id__assignop__expression__t_downto__expressi
on__t_do__statement:
```

该语句是一个for语句，则需要判断该for语句是否合法，具体包括3个部分，首先循环变量的大类（type_category）是否是int类型，其次初始值的大类（type_category）是否是int类型，以及终止值的大类（type_category）是否是int类型。只有这三点同时满足才通过检测。

例5：

```
case tree::statement__T__t_read__leftparen__variable_list__rightparen:
        case
tree::statement__T__t_readln__leftparen__variable_list__rightparen:
```

read或readln语句，只需判断判断待读入的id，idlist是否是可赋值的。

例6：

```
case tree::variable__T__id
```

id转换为variable产生式。有一个特别点即函数返回值的赋值，由于存在无参数的函数调用，故需要分辨无参数函数调用和在函数内部的返回值赋值。如果是返回值的赋值，则设置当前节点的类型，使用node->set_type()。事实上进行类型推断的目的就是检查类型并且确定当前节点的类型。否则进行id判断，即调用check_id接口（具体参见函数定义），若确认该id可被赋值，且id被定义，则继续判断，根据是否是函数类型分别进行当前节点的类型确定。

例7：

```
        case tree::variable__T__id__id_varpart:
```

当前产生式是将一个表示数组元素的式子归约成一个variable，需要判断id是否合法，即id对应的类型是否为TYPE_ARRAY，以及varpart是否合法：包括维数是否和id对应数组维数相符，id_varpart各维度表示的量的大类(type_category)是否为int。包括搜寻表项查询该id是否存在（check_id）。

例8：

```
case
tree::procedure_call__T__id__leftparen__expression_list__rightparen:
```

当前产生式是将一个表示数组元素的式子归约成一个variable，需要判断id是否合法，即id对应的类型是否为TYPE_ARRAY，以及varpart是否合法：包括维数是否和id对应数组维数相符，id_varpart各维度表示的量的大类(type_category)是否为int。包括搜寻表项查询该id是否存在（check_id）。

例9：

```
case tree::procedure_call__T__id:
        case tree::procedure_call__T__id__leftparen__rightparen:
```

过程调用的产生式，但参数列表为空，需要先判断该id是否在符号表里存在，然后判断id 的类型是否是函数或者过程，如果是，是否是无返回值，即procedure，这些都成立再检查过程调用的参数部分是否匹配，这里是无参调用，故需要检查该过程是否为无参过程。设置当前节点类型。

例10：

```
        case tree::expression__T__simple_expression:
        case tree::simple_expression__T__term:
        case tree::term__T__factor:
        case tree::factor__T__variable
```

将子节点类型传递到当前节点类型

例11：

```
        case tree::simple_expression__T__literal_char:
```

将当前节点类型设置为char

例12：

```
        case tree::simple_expression__T__literal_string
```

将当前节点类型设置为string

例13：

```
 case tree::expression__T__simple_expression__relop__simple_expression:
        case
tree::expression__T__simple_expression__equalop__simple_expression:
```

该产生式为关系运算产生式，则需要确保两边的表达式的类型要匹配，即type_category要相等，同时这些类型需要可以比较，比如string就不能比较。若都合法，则将当前节点的类型设置为bool。

例14：

```
case tree::simple_expression__T__term__addop__term:
       case tree::simple_expression__T__term__subop__term
```

当前产生式为将term+/-term归约为简单表达式，需要判断两项的计算结果的大类(type_category)是否能够计算，即需要是int或者float类型，然后进一步得到计算结果的类型，设置为当前节点的类型。

例15：

```
case tree::term__T__term__mulop__factor
```

当前产生式为将term和term之间的乘法，除法，mod，and运算归约为简单表达式，当进行乘除法时需要判断两项的计算结果的大类(type_category)是否能够计算，即需要是int或者float类型，然后进一步得到计算结果的类型，设置为当前节点的类型。如果为and运算，需要确保两项计算结果为bool类型。如果为mod运算，需要确保两项计算结果的大类(type_category)均为int类型。检查完后，设置当前节点类型。

例16：

```
case tree::simple_expression__T__term__or_op__term:
```

当前产生式为将term和term之间的or运算，和and运算进行的处理基本一致。

例17：

```
case tree::factor__T__leftparen__expression__rightparen
```

当前产生式为将表达式外套括号规约为一个factor，则无需进行类型推断，只需要设置当前节点的类型。

例18：

```
case tree::factor__T__id__leftparen__rightparen:
       case
tree::factor__T__id__leftparen__expression_list__rightparen
```

函数调用的产生式，参数列表至少含有一个项。需要先判断该id是否在符号表里存在，然后判断id 的类型是否是函数或者过程，这些都成立再检查过程调用的参数部分是否匹配，包括类型和原过程要求的各个位置的类型是否匹配，数量是否匹配，以及若该位置的参数要求是var，即引用变量，则expression_list里面对应位置不能是const变量。设置当前节点类型。

例19：

```
case tree::factor__T__num:
```

将num归约为factor，只需将当前节点的类型设置为int即可

例20：

```
case tree::factor__T__double_value:
```

将浮点值归约为factor，只需将当前节点的类型设置为double即可

例21：

```
      case tree::factor__T__notop__factor
```

当前产生式是对factor取not，需要判断factor的类型是否能够取not，即是否为int，bool类型，若是则设置当前节点类型，否则报错。

例22：

```
case tree::factor__T__subop__factor
```

该情况可以视为两个项进行加减运算的特例，即变为一个项进行这样的运算，进行的操作和两个项的类似，这里省略。

例23：

```
case tree::factor__T__bool_value
```

将布尔值归约为factor，只需将当前节点的类型设置为布尔类型即可

4.3.3.4 其他函数接口

如下是实现语义分析所需要调用的函数接口，也是我们针对语义分析实现的各个工具函数，具体功能和实现参见源代码。

```
void error_detected()
bool check_type_assignable(BasicType type_a, BasicType type_b)
bool check_id(TreeNode* node, bool expect_basic = true, bool
expect_not_constant = false, bool ignore_scope_name = false)
bool check_variable_assignable(TreeNode* node, bool suppress_log =
false)
```

```
bool check_variable_list_assignable(TreeNode* node)
BasicType get_const_type(TreeNode* node)
BasicType get_basic_type(TreeNode* node)


std::vector<TreeNode*> get_id_node_list(TreeNode* node)
std::vector<std::pair<int, int>> get_dims(TreeNode* node)
std::pair<std::vector<TreeNode*>, symbol::Param>
get_single_param(TreeNode* node)
std::vector<std::pair<TreeNode*, symbol::Param>> get_params(TreeNode*
node)
std::shared_ptr<SymbolTableEntry> get_type(TreeNode* node)
std::vector<TreeNode*> get_expression_list(TreeNode* node)
bool check_expression_variable(TreeNode* node)
std::vector<bool> check_expression_list_variable(TreeNode* node)
```

4.4 代码生成

4.4.1 主要函数和接口

1. generate_code(tree::TreeNode* node, std::shared_ptr<symbol::SymbolTableNode> snode); node表示当前所在的语法树节点，snode表示当前的符号表。该函数是调用代码生成模块的接口。
2. id_process(tree::TreeNode* node,ID_TYPE type),其中node表示当前所在的语法树节点，type表示id的类型。用于处理各类id的输出。
3. idlist_process(tree::TreeNode* node, ID_LIST_TYPE type),其中node表示当前所在的语法树节点，type表示id列表的类型，用于处理不同位置id列表输出。
4. input_tab(bool enter)用于处理换行和缩进。enter表示在缩进前是否换行。
5. varpart_process(tree::TreeNode* node)处理表达式中数组输出格式的转换和下标偏移量，node表示当前所在的语法树节点。
6. function_call_para(tree::TreeNode* node, tree::TreeNode* func_node)用于处理函数调用的输出，区分传值调用和引用调用，node表示调用参数列表节点，func_node表示函数id在语法树上的节点。

4.4.2 功能描述

1. generate_code(int indent, tree::TreeNode* node)针对当前的产生式符号进行代码生成，若该符号不是终结符，则递归调用函数生成代码，否则直接输出相应符号。
2. id_process(tree::TreeNode* node,ID_TYPE type)用于处理标识符，包括数组变量名称、一般变量名称、函数名称、变量引用调用等各类情况。具体包括：一般变量名称只输出变量名本身；数组变量在输出数组名称后继续输出各个维度的偏移量；对于函数名称，需判断其为函数调用还是对函数返回值进行运算，根据情况分别处理；对于变量引用调用，需要在变量名之前加入`*`,即指针符号。
3. idlist_process(tree::TreeNode* node, ID_LIST_TYPE type)用于处理标识符列表，包括变量定义、参数列表、输入列表等一系列情况。具体包括：变量定义一般只在开头输出类型，然后以逗号分隔每个变量；函数定义处的参数列表一般在每个变量前都要输出类型，然后输出变量名，同时如果是引用调用需在变量名前输出*；函数调用处的参数列表则不输出类型，直接输出变量名，同时如果是非数组变量引用调用需在变

量名前输出取地址符&；对于输入列表，一般不输出类型，直接输出取地址符和变量名，以逗号分隔。

4. input_tab(bool enter)用于输出换行和缩进。其中缩进数量为全局变量indent的值，generate_code会根据产生式维护indent的值。
5. varpart_process(tree::TreeNode* node)处理表达式中数组输出格式的转换和下标偏移量。
6. function_call_para(tree::TreeNode* node, tree::TreeNode* func_node)用于处理函数调用的输出，区分传值调用和引用调用。

4.4.3 算法描述

在代码生成部分，我们已经得到AST树，和符号表，上述结构已在前文详细介绍，另外，由于一个Token可以由多个产生式规约产生，故需要在前文语法分析时记录每个Token对应的产生式id。代码生成主函数为generate_code(tree::TreeNode* node)，调用时传进AST的root节点，并初始化符号表表项为最外层。generate_code需要根据每个Token利用switch-case结构分别处理，若当前节点为终结符，如FOR,REPEAT, UNTIL,WHILE等，则输出注释的同时需要输出节点text，如

```
case tree::T_RIGHTPAREN:
    outputexplaination(node);
    logger::output( ")");
    break;
```

若为非终结符，则需要根据pid判读产生式，递归调用函数生成代码。如非终结符expression有三个产生式对应编号用pid表示：

```
simple_expression -> term
   | simple_expression -> term addop term
   | simple_expression -> term subop term
   | simple_expression -> term orop term
   | simple_expression -> term subop term
   | simple_expression -> literal_string
   | simple_expression -> literal_char
```

需要根据不同pid设计递归

```
case tree::T_SIMPLE_EXPRESSION:{
    switch (node->get_pid()){
        case tree::simple_expression__T__term:  //fall!
        case tree::simple_expression__T__literal_char:
            generate_by_pid(node->get_child(0));
            break;
        case tree::simple_expression__T__term__addop__term:    //fall t
hrough!
        case tree::simple_expression__T__term__subop__term:
            generate_by_pid(node->get_child(0));
            generate_by_pid(node->get_child(1));
            generate_by_pid(node->get_child(2));
            break;
        case tree::simple_expression__T__term__or_op__term:
            logger::output("(");
            generate_by_pid(node->get_child(0));
            generate_by_pid(node->get_child(1));
            generate_by_pid(node->get_child(2));
            logger::output(")");
            break;
        case tree::simple_expression__T__literal_string:
            logger::output( "\"" + node->get_child(0)->get_text().subst
r(1, node->get_child(0)->get_text().size()-2) + "\"");
            break;
```

```
        default:
            logger::log("ERROR SIMPLE EXPRESSION:"+ node->get_pid());
    }
    break;
```

在这里，simple_expression__T__term__or_op__term和tree::simple_expression__T__term__addop__term没有选择合并是因为考虑到pascal与C 对OR的优先级不同，需要加括号。

4.4.4 遇到的问题

4.4.4.1 函数的调用与返回值

Pascal 中返回值是以函数名中间处理，最后无需return。但C语言中需要根据pascal 函数返回值类型定义返回变量。最后return结果。并且，由于pascal调用函数无需加（），不仅要根据产生式进行特判，还会导致难以区分是返回值类型还是递归调用。我们处理逻辑为认为遇到id时，需要首先判断当前符号表scope_name是否和id同名，若同名，带括号认为函数调用。否则返回值。若当前符号表scope_name是否和id不同名，则认为是函数调用。根据产生式适当加括号。
另外，符号表的部分结构特性会增加代码生成部分分类讨论的情况数量。当函数调用是递归时，递归调用的函数名称id是基础类型变量（因函数返回值以函数名作为的变量体现），而在非递归时，如其他函数调用了这个函数，则函数名称id是函数类型标识符。
返回值处理case如下

```
case symbol::TYPE_BASIC:
    if(type == NON_BRACKET){
        if(symbol_table_tree.get_scope_name() == node->get_text()){
            logger::output( "_");
        }
        if(std::get<symbol::BasicInfo>(symbol_table_tree.get_entry(node
->get_text())->extra_info).is_referred){
            logger::output( "*");
        }
        logger::output( node->get_text());
    }
    else{

        if(std::get<symbol::BasicInfo>(symbol_table_tree.get_entry(node
->get_text())->extra_info).is_referred)
            logger::output( "*");
        logger::output( node->get_text());
    }
    break;
```

4.4.4.2 函数引用传参问题

在pascal 语言存在类似C++的引用调用，我们需要在C语言函数定义时换成指针，并在引用时在相应变量前加&。因此在定义部分添加特判

```
if(std::get<symbol::BasicInfo>(symbol_table_tree.get_entry(node->get_te
xt())->extra_info).is_referred){
```

```
        logger::output( "*");
 }
```

调用部分代码如下

```
void function_call_para(tree::TreeNode* node, tree::TreeNode* func_node
){
    std::vector<tree::TreeNode*> expr;
    while(node->get_pid() == tree::expression_list__T__expression_list_
_comma__expression){
        expr.push_back(node->get_child(2));
        node = node->get_child(0);
    }
    expr.push_back(node->get_child(0));
    std::reverse(expr.begin(), expr.end());
    std::vector<symbol::Param> *paralist;
    if(symbol_table_tree.get_scope_name() != func_node->get_text())
        paralist = &(std::get<symbol::FunctionInfo>(symbol_table_tree.g
et_current_node()->get_entry(func_node->get_text())->extra_info).params
);
    else
        paralist = &(std::get<symbol::FunctionInfo>(symbol_table_tree.g
et_current_node()->get_parent()->get_entry(func_node->get_text())->extr
a_info).params);
    assert(paralist->size() == expr.size());
    for(int i=0;i<expr.size();++i){
        if((*paralist)[i].is_referred){
            logger::output( "&");
        }
        generate_by_pid(expr[i]);
        if(i!=expr.size()-1)logger::output(", ");
    }
}
```

4.4.4.3 string 拓展的处理

由于C语言无String类型，我们在此用char*　　代替，在此我们需要遇到定义时malloc一个空间（在此简单起见设置为固定长度），在最后需要free结果。下面先以free为例，在pop当前scope前我们遍历定义的每个变量，如果为string，且不为形参（传入的参数），则需free掉。

```
void free_string(){
    for(auto variable: symbol_table_tree.get_current_node()->get_entrie
s()){
        if(variable.second->type == symbol::TYPE_BASIC){
            if(std::get<symbol::BasicInfo>(variable.second->extra_info)
.basic == symbol::TYPE_STRING){
                if(symbol_table_tree.get_current_node()->has_entry(vari
able.first)&&
                        symbol_table_tree.get_current_node()->get_entry
(variable.first)->type == symbol::TYPE_BASIC
                        && !std::get<symbol::BasicInfo>(symbol_table_tr
ee.get_current_node()->get_entry(variable.first)->extra_info).is_referr
ed)
                {
                    logger::output( "free(" + variable.first + ");");
                    input_tab(true);
                }
```

```
            }
        }
    }
}
```

4.4.4.4 数组的处理

在Pascal语言中，数组的定义标注了数组起始与结束，这导致数组下标与C语言从0开始索引的不同，在数组定义中，我们需要使用end-begin+1定义维度，在数组引用时需要在原有下标-bias。数组的定义部分：

```
void varpart_process(tree::TreeNode* node){
    std::string array_name = node->get_child(0)->get_text();
    std::vector<int> bias;
    std::vector<tree::TreeNode*> expr;
    for(auto per_dim: std::get<symbol::ArrayInfo>(symbol_table_tree.get
_entry(array_name)->extra_info).dims){
        bias.push_back(per_dim.first);
    }
    node = node->get_child(1)->get_child(1);    //expression_list
    while(node->get_pid() != tree::expression_list__T__expression){
        expr.push_back(node->get_child(2));
        node = node->get_child(0);
    }
    expr.push_back(node->get_child(0));
    std::reverse(expr.begin(), expr.end());
    assert(bias.size() == expr.size());
    for(int i = 0; i < bias.size(); ++i){
        logger::output( "[");
        generate_by_pid(expr[i]);
        logger::output( "-(" +std::to_string(bias[i]) + ")]");
    }
}
```

数组的引用部分

```
void varpart_process(tree::TreeNode* node){//call a[x+y]
    std::string array_name = node->get_child(0)->get_text();
    std::vector<int> bias;
    std::vector<tree::TreeNode*> expr;
    for(auto per_dim: std::get<symbol::ArrayInfo>(symbol_table_tree.get
_entry(array_name)->extra_info).dims){
        bias.push_back(per_dim.first);
    }
    node = node->get_child(1)->get_child(1);    //expression_list
    while(node->get_pid() != tree::expression_list__T__expression){
        expr.push_back(node->get_child(2));
        node = node->get_child(0);
    }
    expr.push_back(node->get_child(0));
    std::reverse(expr.begin(), expr.end());
    assert(bias.size() == expr.size());
    for(int i = 0; i < bias.size(); ++i){
        logger::output( "[");
        generate_by_pid(expr[i]);
        logger::output( "-(" +std::to_string(bias[i]) + ")]");
    }
}
```

在代码生成环节还遇到如C代码缩进问题，新增补充功能又与先前的产生式冲突等在此不一一罗列。

5. 源程序清单

```
.
├── CMakeLists.txt
├── build
├── include
│   ├── generate.h
│   ├── logger.h
│   ├── position.h
│   ├── semantic.h
│   ├── symbol.h
│   ├── tools.h
│   └── tree.h
├── src
│   ├── generate.cpp
│   ├── logger.cpp
│   ├── main.cpp
│   ├── parser.y
│   ├── scanner.l
│   ├── semantic.cpp
│   ├── symbol.cpp
│   ├── tools.cpp
│   └── tree.cpp
```

6. 程序测试

6.1 单元测试

6.1.1 词法分析

6.1.1.1 测试1

```
program recovery;
const i =2;
var a,gcd,b: integer;
```

```
  d,num1,num2,d:double;
  invalid:string;
begin
  if b=0 then begin
    d:= $$s;
  end
  else b := 1;
  num1 := 1.234e5;
  num2 := 6.78e-3;
  invalid^ := k;
  writeln('num1 = ', num1);
  writeln('num2 = ', num2);
end.
{ unwrapped block comment
```

测试用例报告情况

```
ERROR Unexpected character: $ and now it is patched: line 8
ERROR Unexpected character: $ and now it is patched: line 8
ERROR Unexpected character: ^ and now it is patched: line 13
ERROR Unfinished block comment, auto discarded: line 17
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
ERROR Redefinition of 'd': line 4, column 15
ERROR Undefined identifier 's': line 8, column 11
ERROR Undefined identifier 'k': line 13, column 15
FATAL Failed at semantic analysis, abort.
FATAL Compilation failed.
```

6.1.1.2 测试2

```
program recovery;
var a,b: integer;
begin
  if b=0 then begin
    d:= $$s;
  end
  else b := 1;
  num1 := 1.234e5``;
         @@
  writeln('num1 = ', num1);
  writeln('num2 = ', num2);
end.
'unwrapped string
```

测试用例报告情况

```
ERROR Unexpected character: $ and now it is patched: line 5
ERROR Unexpected character: $ and now it is patched: line 5
ERROR Unexpected character: ` and now it is patched: line 8
ERROR Unexpected character: ` and now it is patched: line 8
ERROR Unexpected character: @ and now it is patched: line 9
ERROR Unexpected character: @ and now it is patched: line 9
ERROR Unwrapped string: 'unwrapped string' and now it is patched.: line
13, column 1
ERROR syntax error, encountering unexpected word 'unwrapped string:
line 13
FATAL Failed at lexical or syntactic analysis, abort.
FATAL Compilation failed.
```

程序分别对非法字符$,^等位置进行恢复，并能够继续分析执行。对未补全的行注释也正确识别。测试还补充了一些拓展功能如科学计数，string的样例

6.1.2 语法分析

6.1.2.1 用例1

```
program Hello;
const i =2;
var a,gcd,b: integer;
c:array test [1..10]  of integer;
d:array [1..10] test of integer;
f:array [1 test ..10]  of integer;
e:array  [1 test 10]  of integer;
begin
    if b=0 then begin
        a:=1
    end
    else b := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word test: line 4
ERROR error is fixed and use production: type -> array [ period ] of
basic_type: line 4, column 3
ERROR syntax error, encountering unexpected word test: line 5
ERROR error is fixed and use production: type -> array [ period ] of
basic_type: line 5, column 3
ERROR syntax error, encountering unexpected word test: line 6
ERROR error is fixed and Use production: period -> num .. num: line 6,
```

```
column 10
ERROR syntax error, encountering unexpected word test: line 7
ERROR error is fixed and Use production: period -> num .. num: line 7,
column 11
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=2;
int a, gcd, b;
int c[10];
int d[10];
int f[10];
int e[10];
int main(){
    if (b==0){
        a=1;
    }
    else {
        b=1;
    }
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针对数组定义进行测试，在此处我们不仅能判别array下标所对应产生式不正确，同时能根据ERROR产生式错误恢复策略，进行用户意图分析，判别...（适当添加）并能向下继续执行以生成C代码。

6.1.2.2 用例2

```
program Hello;
const i test ****=2;
var a,gcd,b:integer;
begin
    if b=0 then begin
        a:=1
    end
```

```
    else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word test: line 2
ERROR error that lack of equalop is fixed and use production:
const_declaration -> id = const_value: line 2, column 7
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=2;
int a, gcd, b;
int main(){
    if (b==0){
        a=1;
    }
    else {
        gcd=1;
    }
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针对const定义进行测试，在此处我们故意添加 ****一些无效字符以测试错误恢复策略，进行用户意图分析，判别 ****无效并能向下继续执行以生成C代码。

6.1.2.3 用例3

```
program Hello;
const i test ****=+ test again 2;
j = - last test * 2;
k = - last test / 2.3e-2;
var a,gcd,b:integer;
begin
    if b=0 then begin
        a:=1
    end
```

```
    else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word test: line 2
ERROR error is fixed and use production: const_value -> +  num: line 2,
column 19
ERROR error that lack of equalop is fixed and use production:
const_declaration -> id = const_value: line 2, column 7
ERROR syntax error, encountering unexpected word last: line 3
ERROR error is fixed and use production: const_value -> -  num: line 3,
column 5
ERROR syntax error, encountering unexpected word last: line 4
ERROR error is fixed and use production: const_value -> - double_value:
line 4, column 5
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=+2;
const int j=-(2);
const float k=-(2.3e-2);
int a, gcd, b;
int main(){
    if (b==0){
        a=1;
    }
    else {
        gcd=1;
    }
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针对const和var 赋值，初始化初始化进行测试，在此处我们故意添加了 **** last test一些无效字符以测试错误恢复策略，进行用户意图分析，程序正确报错，并判别 ****，last test无效并能向下继续执行以生成C代码。

6.1.2.4 用例4

```
program 234 Hello;
const i =2;
var a,gcd b c d e f test, p,q r s,t:integer;
begin
    if b=0 then begin
        a:=1
    end
    else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word 234: line 1
ERROR error fixed and Use production: program_head -> program id: line
1, column 1
ERROR syntax error, encountering unexpected word b: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word c: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word d: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word e: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word f: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word test: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word r: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word s: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
```

```
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=2;
int a, gcd, b, c, d, e, f, test, p, q, r, s, t;
int main(){
    if (b==0){
        a=1;
    }
    else {
        gcd=1;
    }
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针对idlist 这一重要产生式进行测试，在此处我们故意删除了‘，’以测试错误恢复策略，进行用户意图分析，程序正确报错，并输出**error that lack of comma is fixed and use production: idlist -> idlist**
能向下继续执行以生成C代码。

6.1.2.5 用例5

```
program 234 Hello;
const i =2;
var a,gcd,b:integer;
begin
    if b=0 then begin
        a:=1
    end
    else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word 234: line 1
ERROR error fixed and Use production: program_head -> program id: line
1, column 1
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
```

```
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=2;
int a, gcd, b;
int main(){
    if (b==0){
        a=1;
    }
    else {
        gcd=1;
    }
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针对program声明进行测试，在此处我们故意删除了添加无效字符以测试错误恢复策略，进行用户意图分析，程序正确报错，并输**error fixed and Use production: program_head -> program id: line 1, column 1**能向下继续执行以生成C代码。

6.1.2.6 用例6

```
program 234 Hello;
const i =2;
var a,gcd,b:integer;
check:boolean;
begin
    if b=0 then begin
        a:=1
    end
    else gcd := 1;
    b := 23;
    if a=0 %% begin end;
    if a=0 then b:=3
    else b:=2;

    while b<1  begin
    end;

    while a<>3 & do begin
    end;
```

```
    repeat
        writeln('Hello, world!');
    until a<>b;

    repeat
        writeln('Hello, world!');
    1>a;

    readln & (a, + b);

    writeln ('Hello, world!');
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word 234: line 1
ERROR error fixed and Use production: program_head -> program id: line
1, column 1
ERROR Unexpected character: % and now it is patched: line 11
ERROR Unexpected character: % and now it is patched: line 11
ERROR syntax error, encountering unexpected word begin: line 11
ERROR error is fixed and Use production: statement -> if expression
then statement: line 11, column 5
ERROR syntax error, encountering unexpected word begin: line 15
ERROR error is fixed and Use production: statement -> while expression
do statement: line 15, column 5
ERROR Unexpected character: & and now it is patched: line 18
ERROR syntax error, encountering unexpected word until: line 23
ERROR error is fixed and Use production: statement -> repeat
statement_list until expression: line 21, column 5
ERROR syntax error, encountering unexpected word 1: line 27
ERROR error is fixed and Use production: statement -> repeat
statement_list until expression: line 25, column 5
ERROR Unexpected character: & and now it is patched: line 29
ERROR syntax error, encountering unexpected word +: line 29
ERROR error is fixed and Use production: variable_list -> variable_list
, variable: line 29, column 15
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

测试结果

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int i=2;
int a, gcd, b;
bool check;
int main(){
    if (b==0){
        a=1;
    }
    else {
        gcd=1;
    }
    b=23;
    if (a==0){

    }
    if (a==0){
        b=3;
    }
    else {
        b=2;
    }
    while (b<1){
        }
    while (a!=3){
        }
    do{
        printf("%s\n", "Hello, world!");
    }while(!(a!=b));
    do{
        printf("%s\n", "Hello, world!");
    }while(!(1>a));
    scanf("%d%d", &a, &b);
    printf("%s\n", "Hello, world!");
    return 0;
}
```

该测试用例针重要的statement涉及的产生式声明进行测试，在此处我们故意增加了字符& %%等以测试错误恢复策略，进行用户意图分析，程序正确报错位置，并输出所用错误恢复产生式类别，能向下继续执行以生成C代码。

6.1.3 语义分析

6.1.3.1 用例1

```
program Hello;
const i =2;
var a,gcd,b: integer;
c:array test [1..10]  of integer;
d:array [1..10] test of integer;
f:array [1 test ..10]  of integer;
e:array  [1 test 10]  of integer;
begin
 if b=0 then begin
  a:=1
 end
 else b := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
ERROR Dimension mismatched for array 'a': line 7, column 3
ERROR Dimension mismatched for array 'a': line 9, column 3
ERROR Expected integer type for index 1 of array 'a', found others:
line 10, column 5
ERROR Expected integer type for index 2 of array 'a', found others:
line 10, column 8
ERROR Invalid usage of array 'a' as a basic type: line 13, column 3
FATAL Failed at semantic analysis, abort.
FATAL Compilation failed.
```

该测试用例针对数组定义，数组索引进行测试。测试结果正确反映了数组使用情况及错误位置

6.1.3.2 用例2

```
program Hello;
const i test ****=2;
var a,gcd,b:integer;
begin
 if b=0 then begin
  a:=1
 end
 else gcd := 1;
  writeln ('Hello, world!')
```

```
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word test: line 2
ERROR error that lack of equalop is fixed and use production:
const_declaration -> id = const_value: line 2, column 7
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

该测试用例针对常量定义和初始化测试，程序正确识别了const定义语句不符合规范。

6.1.3.3 用例3

```
program Hello;
const i test ****=+ test again 2;
j = - last test * 2;
k = - last test / 2.3e-2;
var a,gcd,b:integer;
begin
 if b=0 then begin
  a:=1
 end
 else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word test: line 2
ERROR error is fixed and use production: const_value -> +  num: line 2,
column 19
ERROR error that lack of equalop is fixed and use production:
const_declaration -> id = const_value: line 2, column 7
ERROR syntax error, encountering unexpected word last: line 3
ERROR error is fixed and use production: const_value -> -  num: line 3,
column 5
ERROR syntax error, encountering unexpected word last: line 4
ERROR error is fixed and use production: const_value -> - double_value:
line 4, column 5
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
```

```
INFO Compilation passed.
```

该测试用例针对常量进行测试，在program中，我们不允许使用表达式对const赋值，同时不允许对const重新赋值。

6.1.3.4 用例4

```
program 234 Hello;
const i =2;
var a,gcd b c d e f test, p,q r s,t:integer;
begin
 if b=0 then begin
  a:=1
 end
 else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例报告情况

```
ERROR syntax error, encountering unexpected word 234: line 1
ERROR syntax error, encountering unexpected word b: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word c: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word d: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word e: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word f: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word test: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word r: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
ERROR syntax error, encountering unexpected word s: line 3
ERROR error that lack of comma is fixed and use production: idlist ->
idlist id: line 3, column 5
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
```

```
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

该测试用例针对idlist进行测试，程序正确识别idlist产生式的错误位置并报错

6.1.3.5 用例5

```
program 234 Hello;
const i =2;
var a,gcd,b:integer;
check:boolean;
begin
 if b=0 then begin
  a:=1
 end
 else gcd := 1;
    b := 23;
    if a=0 %% begin end;
    if a=0 then b:=3
    else b:=2;

    while b<1  begin
    end;

    while a<>3 & do begin
    end;

    repeat
        writeln('Hello, world!');
    until a<>b;

    repeat
        writeln('Hello, world!');
    1>a;

    readln & (a, + b);

    writeln ('Hello, world!');
end.
```

测试用例结果

```
ERROR syntax error, encountering unexpected word 234: line 1
ERROR Unexpected character: % and now it is patched: line 11
ERROR Unexpected character: % and now it is patched: line 11
ERROR syntax error, encountering unexpected word begin: line 11
ERROR error is fixed and Use production: statement -> if expression
```

```
then statement: line 11, column 5
ERROR syntax error, encountering unexpected word begin: line 15
ERROR error is fixed and Use production: statement -> while expression
do statement: line 15, column 5
ERROR Unexpected character: & and now it is patched: line 18
ERROR syntax error, encountering unexpected word until: line 23
ERROR error is fixed and Use production: statement -> repeat
statement_list until expression: line 21, column 5
ERROR syntax error, encountering unexpected word 1: line 27
ERROR error is fixed and Use production: statement -> repeat
statement_list until expression: line 25, column 5
ERROR Unexpected character: & and now it is patched: line 29
ERROR syntax error, encountering unexpected word +: line 29
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

该测试用例针对statement进行测试，程序正确判断if else 等语义错误并进行了错误恢复。

6.1.3.6 用例6

```
program Hello;
const i =2;
var a,gcd,b:integer;

procedure MyProcedure test (var a:integer test);
begin
    writeln(i);
end
test ;

function MyFunction(var params:integer test) test:integer;
const t=1; 233&*^
var j:integer;
92
begin
    writeln(params);
end;;
function Anther(var params:integer test) test integer;
const i = 'a'; *&0.32
begin
```

```
    writeln(params);
end;

begin
 if b=0 then begin
  a:=1;
 end
 else gcd := 1;
  writeln ('Hello, world!');
end.
```

测试用例结果

```
ERROR syntax error, encountering unexpected word test: line 5
ERROR syntax error, encountering unexpected word test: line 5
ERROR syntax error, encountering unexpected word test: line 9
ERROR syntax error, encountering unexpected word test: line 11
ERROR syntax error, encountering unexpected word test: line 11
ERROR syntax error, encountering unexpected word 233: line 12
ERROR Unexpected character: & and now it is patched: line 12
ERROR Unexpected character: ^ and now it is patched: line 12
ERROR syntax error, encountering unexpected word 92: line 14
ERROR error is fixed and Use production: subprogram_body ->
const_declaration compound_statement: line 12, column 1
ERROR syntax error, encountering unexpected word test: line 18
ERROR syntax error, encountering unexpected word test: line 18
ERROR syntax error, encountering unexpected word *: line 19
ERROR Unexpected character: & and now it is patched: line 19
ERROR error is fixed and Use production: subprogram_body ->
const_declaration compound_statement: line 19, column 1
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

该用例针对subprogram进行测试，程序正确判读了function和procedure调用并比对实际传入参数。在错误位置进行报错。

6.1.3.7 用例7

```
program Hello;
const i =2;
var a,gcd,b: test * mod integer;
begin
 if b=0 then begin
  a:=1
 end
```

```
 else gcd := 1;
  writeln ('Hello, world!')
end.
```

测试用例结果

```
ERROR syntax error, encountering unexpected word test: line 3
ERROR error is fixed and use production: var_declaration -> id_list :
type: line 3, column 5
INFO Lexical and syntactic analysis passed.
INFO Semantic analysis...
INFO Semantic analysis passed.
INFO Code generation passed.
INFO Compilation passed.
```

该用例针对变量定义进行测试，识别到不符合变量定义的语义规则。

6.1.4 代码生成

6.1.4.1 测试一

测试用例及说明

```
program main;
 var x, y:longint;
 procedure exgcd(a, b, c:longint; var x, y:longint);
 var x1, y1:longint;
 begin
  if b=0 then
  begin
   if c mod a <> 0 then
   begin
    x:=0;
    y:=0;
   end
   else begin
    x:=c div a;
    y:=0;
   end;
  end
  else begin
   exgcd(b, a mod b, c, x, y);
   if x<>0 or y<>0 then
   begin
    x1:=x;
                        y1:=y;
    x:=y1;
    y:=x1-(a div b)*y1;
   end;
```

```
 end;
end;
begin
 exgcd(23, 74-37, 17+2, x, y);
 writeln(x, ' ', y);
end.
```

这是一个利用扩展欧几里得算法求解整数不定方程的代码。即求解一组x， y满足ax+by=c,其中a,b,c是给定值。在gcd(a,b)|c的条件下有解。这个代码中的主要测试点包括：

1.新增类型longint；

2.注释的输出；

3.进程同时进行传值引用调用；

4.writeln进行的输出和各种运算。

检测代码生成对这些情况处理的正确性。根据代码内容，这里要求计算方程

23x + (74-37)y = 17+2 = 19。

运行结果及说明

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
long x, y;
void exgcd(long  a, long  b, long  c,long  *x, long  *y){
    long x1, y1;
    if (b==0){
        if (c%a!=0){
            *x=-(1);
            *y=-(1);
        }
        else {
            *x=c/a;
            *y=0;
        }
    }
    else {
        exgcd(b, a%b, c, &*x, &*y);
        if (*x!=-(1)){
            x1=*x;
            y1=*y;
            /*b*x1 + (a-(a/b)*b)*y1 = c*/*x=y1;
            *y=x1-(a/b)*y1;
```

```
        }
    }
}
int main(){
    exgcd(23, 74-(37), 17+2, &x, &y);
    printf("%ld%c%ld\n", x, ' ', y);
    return 0;
}
```

生成结果正确处理了

1.新增类型longint及相关变量的声明和定义，即第5行long　x，y;和函数定义中的形参列表。

2.在和pascal代码相同的位置输出了注释。

3.正确处理了函数的传值和引用调用，在函数形参列表和函数体内引用变量前均有额外的指针符号，在调用函数时被引用变量前均有额外的取地址符号

4.正确使用printf对结果进行输出，控制符正确，换行符正确。

经检验，生成代码可以通过C编译。输出结果为-152，95。

经计算，　-23*152+95*37=19,运行结果正确。

6.1.4.2 测试二

测试用例及说明

```
program main;
const a1=5;
var a:array [2..100, 3..4, 0..9] of integer;
 i, j, k:integer;
```

```
begin
 for i:=2 to 100 do begin
  for j:=3 to 3 do begin
   for k:=2 downto 0 do begin
    a[i,j,k] := i*j*k;
   end
  end
 end;
 write(a[0, 3, 1])
end.
```

本代码主要测试了：

1.多维数组的定义和调用；

2.for循环的to和downto；

3.write输出；

4.const声明定义部分。

运行结果及说明

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
const int a1=5;
int a[99][2][10];
int i, j, k;
int main(){
    for (i=2; i<=100; ++i){
        for (j=3; j<=3; ++j){
            for (k=2; k>=0; --k){
                a[i-(2)][j-(3)][k-(0)]=i*j*k;
            }
        }
    }
    printf("%d", a[0-(2)][3-(3)][1-(0)]);
    return 0;
}
```

代码生成正确处理了

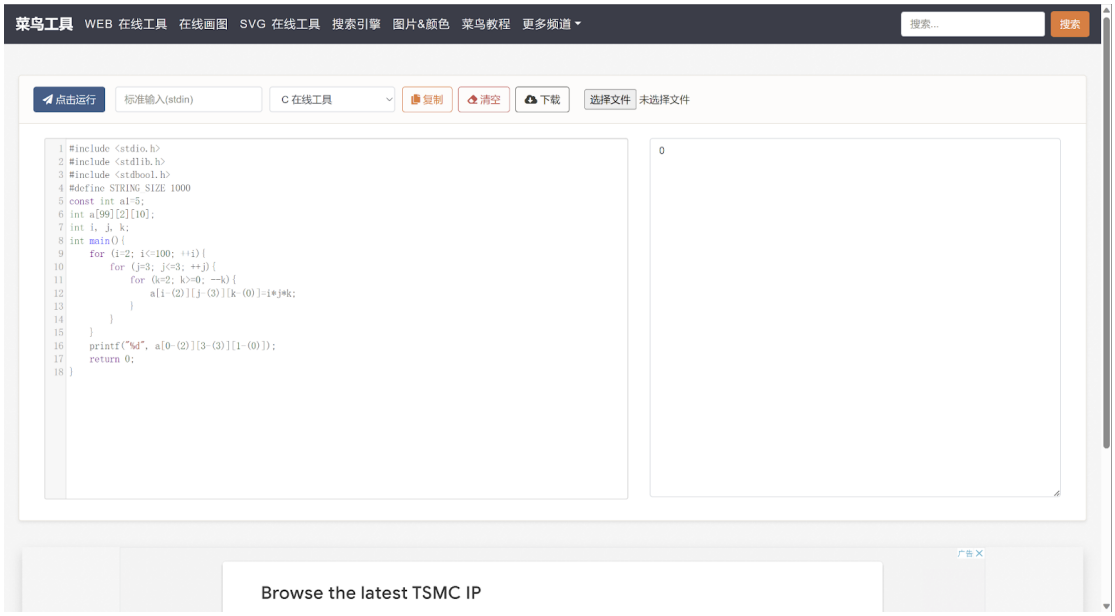
1.多维数组的定义和调用：对数组每个维度，将pascal的范围格式改为C的数字格式，该数字是范围的长度，下标从0开始；调用时对每一维减去范围起点。

2.for循环to和downto：程序正确处理了for循环的to和downto的情况，在to时使用<=和++，在downto时使用>=和--.

3.write输出：格式控制符正确，且没有输出换行。

4.const定义部分输出正确。

生成的代码可以通过C编译，输出结果正确，为0，即0*3*1。



6.2 综合测试

综合测试是结合前面各种单元测试进行的极端化测试，所谓极端化，即让测试覆盖尽量多的产生式，尽量涉及各种特殊情况，尽量使得测试用例复杂，使用的结构复杂，从外部角度尽量增大测试的难度，挑战编译器对极端情况的处理。且综合测试仅放让编译器最终能够运行成功的例子，即在未出错误的情况下尽可能复杂的例子。分析过程和单元测试类似，故这里不再针对结果进行分析，而是仅展示测试运行结果。

最终检查发现所有综合测试均通过。

6.2.1 综合测试一

测试用例

```
program finaltest1(input,output);
var
    x,y:integer;
    z,w:double;
    str:string;
    c:char;
begin
    read(x,y,z,w);
    str := '12345$#@&';
    c := 'c';
    x :=  not not not(------5); {multiple not and multiple minus}
```

```
    y := x and (-123);
    z := -----3.23344;
    if x < 0.3*0.4-(2*(2+3-2.3))*0.3 then begin
        w := (0.2-x)*2+(x mod 10)div
2+0.4*((2+3-24.3123)*((31.324-234.12)/(23.1-23.4)))
    end;
    writeln((x-y)/z*w/2.0+1.334-3.231);
    writeln(x,'temp',y,w,z,'temp');
    writeln('hello world!');
    writeln('for test 123 $%*&*@#!');
end.
```

运行结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
int x, y;
double z, w;
char* str;
char c;
int main(){
    str = (char *)malloc(sizeof(char) * STRING_SIZE)
    scanf("%d%d%lf%lf", &x, &y, &z, &w);
    str="12345$#@&";
    c='c';
    x=!!!(-(-(-(-(-(-(5)))))));
    /*multiple not and multiple minus*/y=(x&(-(123)));
    z=-(-(-(-(-(3.23344)))));
    if (x<0.3*0.4-(2*(2+3-2.3))*0.3){

w=(0.2-x)*2+(x%10)/2+0.4*((2+3-24.3123)*((31.324-234.12)/(23.1-23.4)));
    }
    printf("%f\n", (x-y)/z*w/2.0+1.334-3.231);
    printf("%d%s%d%lf%lf%s\n", x, "temp", y, w, z, "temp");
    printf("%s\n", "hello world!");
    printf("%s\n", "for test 123 $%*&*@#!");
    free(str);
    return 0;
}
```

6.2.2 综合测试二

测试用例

运行结果

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
char a[10];
char b[10];
long int c[12][29][2];
double d[20];
char i;
int j, x, y, z;
long int k;
double l;
char m;
double s;
int main(){
    i=a[1-(1)];
    printf("%s\n", "hello world!");
    scanf("%d%d%d", &x, &y, &z);
    if (a[(x+y)+x*y-z/x-(1)]<10.2){
        a[(x-y)/2-i+k*k*k/k%i-(1)]=214748;
        printf("%d\n", a[(x*y-2)/2+k*i%(i*2-3)-(1)]);
    }
    printf("%c\n", b[3-(3)]);
    scanf("%ld", &c[3-(1)][4-(2)][5-(34)]);
    a[a[0-(1)]-(1)]=a[1-(1)];
    b[3-(3)]='c';
    d[a[1-(1)]+1-(10)]=l+s*3/3.145;

c[-(2)-(1)][4-(2)][34-(34)]=((((12345&23)|15))*(34-23*71)&((231|4895)))
;
    printf("%d\n", ((((319-78*7)&2)|(3-2+17*8*z-x*y))|(x-y+z)));
    if (a[c[-(10)-(1)][15-(2)][35-(34)]*10-(1)]!=10){
        i=a[c[-(10)-(1)][15-(2)][35-(34)]*10-(1)];
        m=b[10-(3)];
        s=d[23-(10)];
    }
    return 0;
}
```

6.2.3 综合测试三

测试用例

```
program finaltest3(input,output);
var
    a : array[1..10] of byte;
    b : array[3..12] of char;
    c : array[1..12,2..30,34..55] of longint;
    d : array[10..29] of double;
    i : byte;
    j, x, y, z: integer;
    k : longint;
    m : char;
    s, l: double;
    t : single;
begin
 writeln('begin to test');
    i := a[1];
    read(x,y,z);
    k := c[x,y,z] + a[4]*a[3]-a[2]+i mod a[2]+k*c[10,23,35];
    s := t* c[10,2,a[3]] / d[10] + d[a[c[-2,4,34]]] * c[10,2,39] / t +
a[2]/x+a[3]/5*2.3-3.1+2.3*4.5;
    j := (x div 2)*y*z+i-2*32-j*(a[4]-a[3]) mod 2;
    l := s + 2.3*4.5;
    if ((x = c[a[0],a[1],a[3]]) or ((y-x+z*z mod x*(y-x*(y-321) div z))
<> ((12-321)*(y-x+z) div (3-12*z))) and ((y-x*z) < (x div y+z*z div
2))) then begin
        writeln(l,i,j,k)
    end;
    if (((y-x)<>(12*34-34*x+a[a[a[0]]])) and ((x-89+x div 2 mod 4)>
a[2]) or (a[i] > a[x]+a[y])) then begin
        if (a[i-j+x]>a[x+y-z]+2) then begin
            if a[i] = a[j] then write(i,j)
        end
    end
end.
```

运行结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
char a[10];
char b[10];
long int c[12][29][22];
```

```
double d[20];
char i;
int j, x, y, z;
long int k;
char m;
double s, l;
float t;
int main(){
    printf("%s\n", "begin to test");
    i=a[1-(1)];
    scanf("%d%d%d", &x, &y, &z);

k=c[x-(1)][y-(2)][z-(34)]+a[4-(1)]*a[3-(1)]-a[2-(1)]+i%a[2-(1)]+k*c[10-
(1)][23-(2)][35-(34)];

s=t*c[10-(1)][2-(2)][a[3-(1)]-(34)]/d[10-(10)]+d[a[c[-(2)-(1)][4-(2)][3
4-(34)]-(1)]-(10)]*c[10-(1)][2-(2)][39-(34)]/t+a[2-(1)]/x+a[3-(1)]/5*2.
3-3.1+2.3*4.5;
    j=(x/2)*y*z+i-2*32-j*(a[4-(1)]-a[3-(1)])%2;
    l=s+2.3*4.5;
    if
((((x==c[a[0-(1)]-(1)][a[1-(1)]-(2)][a[3-(1)]-(34)])|(((y-x+z*z%x*(y-x*
(y-321)/z))!=((12-321)*(y-x+z)/(3-12*z)))&((y-x*z)<(x/y+z*z/2))))))){
        printf("%lf%d%d%ld\n", l, i, j, k);
    }
    if
((((((y-x)!=(12*34-34*x+a[a[a[0-(1)]-(1)]-(1)]))&((x-89+x/2%4)>a[2-(1)]
))|(a[i-(1)]>a[x-(1)]+a[y-(1)])))){
        if ((a[i-j+x-(1)]>a[x+y-z-(1)]+2)){
            if (a[i-(1)]==a[j-(1)]){
                printf("%d%d", i, j);
            }
        }
    }
    return 0;
}
```

6.2.4 综合测试四

测试用例

```
program finaltest4(input,output);
var
    a : array[1..10] of byte;
    b : array[3..12] of char;
```

```
    c : array[1..12,2..30,34..350] of longint;
    d : array[10..29] of double;
    i : byte;
    j, x, y, z: integer;
    k : longint;
    l : double;
    m : char;
    s : double;
    str : string;
{here is simple test for function}
function MySmallFunction:char;
const ch = 'a';
begin
    MySmallFunction := ch
end;
{here is complex test for function}
function MyFunction(i,j:integer;var a,b:char;var s:string;var
d:double):integer;
const
    pi = 3.1415926;
    phi = 2.718281828;
var p,q:integer;
    r,g:char;
    t:string;
    temp_d:double;
begin
    writeln('This is a function');
    writeln(i,j,a,b);
    {here is test for global variable}
    writeln(d,c[1,3,34]);
    a := 'a';
    b := a;
    if (b='c') then begin
        i := (p-q)*(i+j);
        d := d*i/23-1.1415926;
        writeln(a,b,i,j);
        {here is test for seting of return value }
        MyFunction := (q-p*2)*(i-j*p div q)
    end;
    {here is test for internal function call}
    MyFunction := i+j + MyFunction(i,j,a,b,s,d);
    s := 'This is a string';
    m := 'm';
    k := 3924525;
    for i := 1 to 10 do begin
```

```
        k := k*i + MyFunction(i,j,r,g,t,temp_d);
        writeln(k)
    end;
    i := 2;
    {here is the test for const}
    writeln(pi, phi);
    temp_d := pi*phi/3
end;
begin
    {here is the basical test for array}
 writeln('begin to test');
    i := a[1];
    read(x,y,z);
    if a[(x+y)mod z+x*y-z div x] < 10.2 then begin
        a[(x-y)div 2-i+k*k*k div k mod i] := 214748;
        writeln(a[(x*y-2)div 2+k*i mod (i*2-3)])
    end;
    writeln(b[3]);
    read(c[3,4,5]);

    {below are the test for function}
    i := (MyFunction(x,y,b[4],b[5],str,d[28])+a[0]) div 10;
    if (MyFunction(x,z,b[2],b[8],str,s) = 1+(a[129]*a[a[3]])) then
begin
        writeln(MyFunction(y,z,b[8],b[6],str,1));
        b[3] := MySmallFunction();
        b[4] := MySmallFunction();
        b[9] := MySmallFunction();
        if MySmallFunction = 'a' then writeln('correct')
    end
end.
```

运行结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
char a[10];
char b[10];
long int c[12][29][317];
double d[20];
char i;
int j, x, y, z;
long int k;
```

```
double l;
char m;
double s;
char* str;
/*here is simple test for function*/char  MySmallFunction(){
    char  _MySmallFunction;
    const char ch='a';
    _MySmallFunction=ch;
    return _MySmallFunction;
}
/*here is complex test for function*/t = (char *)malloc(sizeof(char) *
STRING_SIZE)
int MyFunction(int i, int j,char  *a, char  *b,char * *s,double  *d){
    int _MyFunction;
    const float pi=3.1415926;
    const float phi=2.718281828;
    int p, q;
    char r, g;
    char* t;
    double temp_d;
    printf("%s\n", "This is a function");
    printf("%d%d%c%c\n", i, j, *a, *b);
    /*here is test for global variable*/printf("%lf%ld\n", *d,
c[1-(1)][3-(2)][34-(34)]);
    *a='a';
    *b=*a;
    if ((*b=='c')){
        i=(p-q)*(i+j);
        *d=*d*i/23-1.1415926;
        printf("%c%c%d%d\n", *a, *b, i, j);
        /*here is test for seting of return value
*/_MyFunction=(q-p*2)*(i-j*p/q);
    }
    /*here is test for internal function
call*/_MyFunction=i+j+MyFunction(i, j, &*a, &*b, &*s, &*d);
    *s="This is a string";
    m='m';
    k=3924525;
    for (i=1; i<=10; ++i){
        k=k*i+MyFunction(i, j, &r, &g, &t, &temp_d);
        printf("%ld\n", k);
    }
    i=2;
    /*here is the test for const*/printf("%f%f\n", pi, phi);
    temp_d=pi*phi/3;
```

```
    free(t);
    return _MyFunction;
}
int main(){
    str = (char *)malloc(sizeof(char) * STRING_SIZE)
    /*here is the basical test for array*/printf("%s\n", "begin to
test");
    i=a[1-(1)];
    scanf("%d%d%d", &x, &y, &z);
    if (a[(x+y)%z+x*y-z/x-(1)]<10.2){
        a[(x-y)/2-i+k*k*k/k%i-(1)]=214748;
        printf("%d\n", a[(x*y-2)/2+k*i%(i*2-3)-(1)]);
    }
    printf("%c\n", b[3-(3)]);
    scanf("%ld", &c[3-(1)][4-(2)][5-(34)]);
    /*below are the test for function*/i=(MyFunction(x, y, &b[4-(3)],
&b[5-(3)], &str, &d[28-(10)])+a[0-(1)])/10;
    if ((MyFunction(x, z, &b[2-(3)], &b[8-(3)], &str,
&s)==1+(a[129-(1)]*a[a[3-(1)]-(1)])){
        printf("%d\n", MyFunction(y, z, &b[8-(3)], &b[6-(3)], &str,
&l));
        b[3-(3)]=MySmallFunction();
        b[4-(3)]=MySmallFunction();
        b[9-(3)]=MySmallFunction();
        if (MySmallFunction()=='a'){
            printf("%s\n", "correct");
        }
    }
    free(str);
    return 0;
}
```

6.2.5 综合测试五

测试用例

```
program finaltest5(input,output);
var
    a : array[1..10] of byte;
    b : array[3..12] of char;
    c : array[1..12,2..30,34..350] of longint;
    d : array[10..29] of double;
    i : byte;
    j, x, y, z: integer;
    k : longint;
```

```
    l : double;
    m : char;
    s : double;
    str : string;
 char_array : array[0..10] of char;
{here is simple test for function}
procedure MySmallProcedure();
const ch = 'a';
begin
    {here is test for global variable}
    writeln(l,m,s,str)
end;
{here is complex test for function}
procedure MyProcedure(i,j:integer;var a,b:char;var s:string;var
d:double);
const
    pi = 3.1415926;
    phi = 2.718281828;
var p,q:integer;
    r,g:char;
    t:string;
    temp_d:double;
begin
    writeln('This is a function');
    writeln(i,j,a,b);
    {here is test for global variable}
    writeln(d,c[-1,3,34]);
    a := 'a';
 b := a;
    if (b='c') then begin
        i := (p-q+2)*(i+j);
        d := d*i/23-1.1415926;
        writeln(a,b,i,j);
        MyProcedure(i,j,a,b,s,d)
    end;
    {here is test for procedure call}
    MyProcedure(p,q,char_array[0],char_array[1],t,temp_d);
    {here is test for setting for the var parameter}
    s := 'This is a string';
    m := 'm';
    k := 3924525;
    {here is test for procedure call}
    for i := 1 to 10 do begin
        MyProcedure(i,j,r,g,t,temp_d)
    end;
```

```
    i := 2;
    {here is the test for const}
    writeln(pi, phi);
    temp_d := pi*phi/3
end;
begin
    {below are the test for function}
    writeln('begin to test');
    MySmallProcedure();
    MyProcedure(i,j,m,b[3],str,s);
    MySmallProcedure()
end.
```

运行结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
char a[10];
char b[10];
long int c[12][29][317];
double d[20];
char i;
int j, x, y, z;
long int k;
double l;
char m;
double s;
char* str;
char char_array[11];
void MySmallProcedure(){
    const char ch='a';
    /*here is test for global variable*/printf("%lf%c%lf%s\n", l, m, s,
str);
}
void MyProcedure(int i, int j,char  *a, char  *b,char * *s,double  *d){
    t = (char *)malloc(sizeof(char) * STRING_SIZE)
    const float pi=3.1415926;
    const float phi=2.718281828;
    int p, q;
    char r, g;
    char* t;
    double temp_d;
```

```
    printf("%s\n", "This is a function");
    printf("%d%d%c%c\n", i, j, *a, *b);
    /*here is test for global variable*/printf("%lf%ld\n", *d,
c[-(1)-(1)][3-(2)][34-(34)]);
    *a='a';
    *b=*a;
    if ((*b=='c')){
        i=(p-q+2)*(i+j);
        *d=*d*i/23-1.1415926;
        printf("%c%c%d%d\n", *a, *b, i, j);
        MyProcedure(i, j, &*a, &*b, &*s, &*d);
    }
    /*here is test for procedure call*/MyProcedure(p, q,
&char_array[0-(0)], &char_array[1-(0)], &t, &temp_d);
    /*here is test for setting for the var parameter*/*s="This is a
string";
    m='m';
    k=3924525;
    /*here is test for procedure call*/for (i=1; i<=10; ++i){
        MyProcedure(i, j, &r, &g, &t, &temp_d);
    }
    i=2;
    /*here is the test for const*/printf("%f%f\n", pi, phi);
    temp_d=pi*phi/3;free(t);

}
int main(){
    str = (char *)malloc(sizeof(char) * STRING_SIZE)
    /*below are the test for function*/printf("%s\n", "begin to test");
    MySmallProcedure();
    MyProcedure(i, j, &m, &b[3-(3)], &str, &s);
    MySmallProcedure();
    free(str);
    return 0;
}
```

6.2.6 综合测试六

测试用例

```
program finaltest6(input,output);
var
    x,y,i,j:integer;
    z,w:double;
    str:string;
```

```
    c:char;
begin
    {here is test for readln read write writeln}
    writeln('begin to test');
    read(x,y,z,w);
    readln(str);
    readln(c);
    write(x,y,'test for write',z);
    writeln('123');
    {here is test for if else}
    if x>y then
        writeln(x)
    else if x >z then
        writeln(y)
    else if x> w then
        writeln(z)
    else begin
        writeln(w);
        writeln(x)
    end;

    {here is test for for ... to do and for... downto ... do}
    for i:=1 to 10 do begin
        writeln(i);
        z := z+i*x;
        if x> w then
            writeln(z)
        else begin
            writeln(w);
            writeln(x)
        end
    end;
    for i:=(x*y-2)div 3 downto -(x*y) do begin
        writeln(j);
        z:=z*i-x
    end;

    repeat
        for i:=x mod y to x*y do begin
            z:=z+8*i;
            writeln(i)
        end;
        writeln(i)
    until (i<>(j-8)div 2)
end.
```

运行结果

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
int x, y, i, j;
double z, w;
char* str;
char c;
int main(){
    str = (char *)malloc(sizeof(char) * STRING_SIZE)
    /*here is test for readln read write writeln*/printf("%s\n", "begin
to test");
    scanf("%d%d%lf%lf", &x, &y, &z, &w);
    scanf("%s", str);
    scanf("%c", &c);
    printf("%d%d%s%lf", x, y, "test for write", z);
    printf("%s\n", "123");
    /*here is test for if else*/if (x>y){
        printf("%d\n", x);
    }
    else {
        if (x>z){
            printf("%d\n", y);
        }
        else {
            if (x>w){
                printf("%lf\n", z);
            }
            else {
                printf("%lf\n", w);
                printf("%d\n", x);
            }
        }
    }
    /*here is test for for ... to do and for... downto ... do*/for
(i=1; i<=10; ++i){
        printf("%d\n", i);
        z=z+i*x;
        if (x>w){
            printf("%lf\n", z);
        }
        else {
            printf("%lf\n", w);
```

```
            printf("%d\n", x);
        }
    }
    for(i=(x*y-2)/3; i>=-((x*y)); --i){
        printf("%d\n", j);
        z=z*i-x;}
    do{
        for (i=x%y; i<=x*y; ++i){
            z=z+8*i;
            printf("%d\n", i);
        }
        printf("%d\n", i);
    }while(!((i!=(j-8)/2)));
    free(str);
    return 0;
}
```

6.3 综合测试二

在综合测试二模块，我们将测试一些需要输入的pascal样例，并用GCC编译所生成的C代码并用脚本测试对应C语言输出与预期输出。

6.3.1 测试用例1

```
program main;
var a, b, c, x, y:longint;
procedure exgcd(a, b, c:longint; var x, y:longint);
var x1, y1:longint;
begin
  if b=0 then
  begin
    if c mod a <> 0   then
    begin
      x:=0;
      y:=0;
    end
    else begin
      x:=c div a;
      y:=0;
    end;
  end
  else begin
    exgcd(b, a mod b, c, x, y);
    if (x<>0) or (y<>0) then
```

```
    begin
      x1:=x;
      y1:=y;
      x:=y1;
      y:=x1-(a div b)*y1;
    end;
  end;
end;
begin
  {please enter a,b,c which are all non-zero and not too big.}
  read(a, b, c);
  exgcd(a, b, c, x, y);
  if (x=0) and (y=0) then begin
    writeln('No solution');
  end
  else
  begin
    writeln('x=', x, ', y=', y);
    writeln(a,' * ',x,' + ',b,' * ',y,' = ',c);
  end
end.
```

所对应生成的C代码

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
long int a, b, c, x, y;
void exgcd(long  a, long  b, long  c,long  *x, long  *y){
    long int x1, y1;
    if (b==0){
        if (c%a!=0){
            *x=0;
            *y=0;
        }
        else {
            *x=c/a;
            *y=0;
        }
    }
    else {
        exgcd(b, a%b, c, &*x, &*y);
        if (((*x!=0)|(*y!=0))){
            x1=*x;
            y1=*y;
            *x=y1;
            *y=x1-(a/b)*y1;
```

```
        }
    }
}
int main(){
    /*please enter a,b,c which are all non-zero and not too
big.*/scanf("%ld%ld%ld", &a, &b, &c);
    exgcd(a, b, c, &x, &y);
    if (((x==0)&(y==0))){
        printf("%s\n", "No solution");
    }
    else {
        printf("%s%ld%s%ld\n", "x=", x, ", y=", y);
        printf("%ld%s%ld%s%ld%s%ld%s%ld\n", a, " * ", x, " + ", b, " *
", y, " = ", c);
    }
    return 0;
}
```

程序目的是辗转相除法来计算两个数的最大公约数，不仅涉及到long类型，也涉及到函数引用传参和传值传参，设计输入37 49 1784，正确输出
x=7136, y=-5352
37 * 7136 + 49 * -5352 = 1784

6.3.2 测试用例2

```
program main;
var a, b, c:longint;
function fpow(x, y, m:longint):longint;
var tmp:integer;
begin
  if y=0 then
  begin
    fpow:=1;
  end
  else begin
    tmp:=fpow(x, y div 2, m);
    fpow:=(tmp*tmp) mod m;
    if y mod 2=1 then
      fpow:=fpow*x mod m;
  end;
end;
begin
  read(a,b,c);
  writeln(a,' ^ ',b,' mod ',c,' = ',fpow(a,b,c));
end.
```

所对应生成的C代码

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
long int a, b, c;
long  fpow(long  x, long  y, long  m){
    long  _fpow;
    int tmp;
    if (y==0){
        _fpow=1;
    }
    else {
        tmp=fpow(x, y/2, m);
        _fpow=(tmp*tmp)%m;
        if (y%2==1){
            _fpow=_fpow*x%m;
        }
    }
    return _fpow;
}
int main(){
    scanf("%ld%ld%ld", &a, &b, &c);
    printf("%ld%s%ld%s%ld%s%ld\n", a, " ^ ", b, " mod ", c, " = ",
fpow(a, b, c));
    return 0;
}
```

该程序通过二分实现快速计算幂指数，输入43 79 589432正确输出
43 ^ 79 mod 589432 = 45227

6.3.3 测试用例3

```
program main;
var
  n,i:longint;
  a:array[0..10000] of longint;
procedure quick_sort(l,r:longint);
var
  i,j,mid:longint;
begin
  if l<r then begin
    i:=l;j:=r;mid:=a[(l+r) div 2];
    repeat
      while a[i]<mid do i:=i+1;
      while a[j]>mid do j:=j-1;
```

```
      if i<=j then
      begin
        a[0]:=a[i];
        a[i]:=a[j];
        a[j]:=a[0];
        i:=i+1;
        j:=j-1;
      end;
    until i>j;
    quick_sort(l,j);
    quick_sort(i,r);
  end;
end;
begin
  readln(n);
  for i:=1 to n do
    read(a[i]);
  quick_sort(1,n);
  for i:=1 to n do
    write(a[i], ' ');
end.
```

所对应生成的C代码

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define STRING_SIZE 1000
long int n, i;
long int a[10001];
void quick_sort(long  l, long  r){
    long int i, j, mid;
    if (l<r){
        i=l;
        j=r;
        mid=a[(l+r)/2-(0)];
        do{
            while (a[i-(0)]<mid){
                i=i+1;}
            while (a[j-(0)]>mid){
                j=j-1;}
            if (i<=j){
                a[0-(0)]=a[i-(0)];
                a[i-(0)]=a[j-(0)];
                a[j-(0)]=a[0-(0)];
                i=i+1;
                j=j-1;
            }
```

```
        }while(!(i>j));
        quick_sort(l, j);
        quick_sort(i, r);
    }
}
int main(){
    scanf("%ld", &n);
    for (i=1; i<=n; ++i){
        scanf("%ld", &a[i-(0)]);
    }
    quick_sort(1, n);
    for (i=1; i<=n; ++i){
        printf("%ld%c", a[i-(0)], ' ');
    }
    return 0;
}
```

程序实现了一个递归调用的快排程序，程序输入
10
77 32 142 43 -35 29 33 20 20 4
程序正常输出
-35 4 20 20 29 32 33 43 77 142

7. 所做拓展内容总结

这里罗列的拓展上文在详细设计已详细展开，这里稍作总结

- stament -> if...，这里拓展statement为statement_list
- 且添加了while语句的实现，repeat的实现，for downto 的实现，
- writeln，readln的实现
- expression加入生成式literal_string和literal_char，不能在factor里面加，因为factor可以运算
- factor产生式加入real_num和bool_value
-
- 类型拓展：
- byte，shortint，integer，longint，double，single（认为整型之间可以直接运算）
-
- 无参函数调用，无参过程调用，是否有括号（函数，过程定义）。
- id可以包含下划线
- idlist->idlist id（错误处理）
- 末尾可识别多个分号
- 错误恢复

8．课程设计总结

8.1 体会与收获

高畅：
本次课程设计我主要负责的工作一部分是整体框架的设计，整体结构、构建以及测试系统的编写。通过这部分的工作，我对《编译原理》这门课中讲到的编译器的词法分析，语法分析，语义分析和代码生成这几部分工作有了更全面的把握。同时在规划、构建项目上积累了经验。

我负责的另一部分工作是符号表的设计和语义分析部分代码的编写。符号表部分我们采用入栈、出栈两个操作建立了树形的作用域逻辑结构，在每个作用域的符号表中维护标识符的类型信息。通过这部分代码编写，我对符号表的逻辑结构有了更深入的认识，对编译过程中符号表的建立及维护有了更清晰的理解。同时，在语义分析的编写中，我们主要参考 pascal 而非 pascal-S 进行逻辑编写，以实现更通用的类型系统。通过对 pascal 及 c 这两种语言在类型系统中的共同点和差异进行深入分析，针对这两种语言中隐式类型转换的不同点（如 pascal 不支持浮点型向整形赋值）进行了特别处理，这使我对这两门语言的类型系统有了更全面的了解。语义分析中的一个关键环节是在表达式中自底向上地进行类型推断，从而建立一个表达式的类型，这部分的编写考虑了 pascal 中各种操作符的操作数类型以及结果类型，是语义分析中较为繁琐的部分。

最后，通过本项目的编写，我锻炼了团队合作开发的能力，获得了团队协调、组织工作的经验，在资料检索，自主学习，阅读、编写文档等方面也积累了丰富的经验。

谢国富：
通过这次课程设计，我动手实现了pascal-s到c语言的编译器。在这个过程中，我深入了解了编译器的各个模块所完成的事情，各个模块进行的工作和功能，包括词法分析，语法分析，语义分析和代码生成，复习了上学期所学的《编译原理》课程知识，并通过手动编写编译器巩固加深了我对《编译原理》这门课的理解和认识。

我主要实现的是词法分析和语法分析，在实现的过程中，我大量查阅资料，了解了各个编译器的实现方案，以及yacc和flex的使用方法，并根据yacc提供的接口完成了错误恢复和错误处理。在语法分析和词法分析中均做出了相应的拓展功能。在词法分析中，我进行了错误处理，了解了pascal-s的词法结构。在语法分析中，我学习了pascal-s的语法结构，并完成代码编写。整体上，通过这个项目，了解了编译器各个模块的功能和它们之间的协作关系。通过这次课程设计，我相对独立地进行了我的代码模块的编写，调试。在这个过程中，我遇到了大量的问题，例如 如何调用yyerrok、yyloc，如何使用yacc自带的error来进行错误处理等，最后通过查阅资料，具体分析，一一解决。这个过程中遇到的最大的问题在于学习yacc本身较为特殊的代码框架，而通过深入阅读文档后，我能够通过yacc来实现需求。并且在合作的过程中锻炼了团队沟通、协作能力。

整个项目极大锻炼了我开发能力，能够开发逻辑较为复杂的代码，并针对整个项目进行模块debug，模块测试。同时锻炼了我解决未知问题的能力，培养了我阅读源文档的习惯。总的来说，通过这次课程设计，我受益匪浅！

张天泽：
在此次课程设计中，我主要负责代码生成部分。在代码生成部分的编码初期，本人为了更快完成任务，直接使用parser.y中定义的pid作为区分产生式的标准，之后遇到词法分析调整产生式导致pid更改的问题，造成代码的大范围修改，同时部分switch-case结构没有写default部分，导致无法快速找到没有处理的产生式，这些都增加了调试的时间。在编码后期我们将对pid的switch改为对产生式构成的枚举类型的switch，同时新增default，降低了代码调试的时间成本。这次课程设计锻炼了我的编码和调试能力，培养了我阅读文档和团队协作的能力，也让我体会到编码和其他事情一样不能急于求成，而是应当思考清楚以后再开始行动。

马英珏：
在本次项目中，我主要完成了代码生成模块以及对应但单元测试和部分总体测试环节。在与他人协作进行开发中，我更加熟悉了git协同开发的使用，并对C++ 面向对象设计有了更深刻的理解。在不断更新增加新功能的情况下，体会到封装以及接口设计的重要性。同时，通过对AST，符号表到C代码的转化，更加深入理解了编译各个环节功能以及原理。在代码生成部分的编写过程中，我还遇到了诸如函数参数的传值和引用、高维数组、pascal-s内嵌函数在C代码中外调、字符串的malloc和free等问题。经过思考我们设计出合适的方案解决这些问题。此外，熟悉出现频率高的产生式及其推导，有助于快速定位代码出错的位置。

8.2 设计过程中遇到或存在的主要问题及解决方案

在初始代码编写环节，我们首先讨论所需完成的产生式，以及识别的Token，为其建立enum，AST树的结构和符号表结构建立可谓至关重要。词法分析部分问题有如何通过flex来进行字符串开闭错误，注释开闭问题；后面经过查阅资料，通过flex自带的一些实现工具，当判断是注释开头时进入一个状态，然后在状态内处理的方式来解决。

语法分析部分的最大问题有如何在yacc的框架下进行错误恢复和错误处理，我们通过利用yacc里面自带的error状态来实现错误恢复和错误处理。以及通过增加新的产生式来实现附加功能。

语义分析上的问题如类型推断、检查和符号表的建立，这些我们通过树型符号表解决，同时细节问题如多个本质相同的类型如integer，byte，longint的处理，无参函数的调用问题等。前者，我们设计了一个type_category来处理，后者我们特殊判断，记录scope_name。事实上有非常多的问题，类型相关的和符号表相关的。具体参见该章节的具体实现的伪代码部分。

代码生成部分的问题有函数调用和返回值问题，函数引用传参问题，拓展string问题，数组维度问题等。解决方案如针对第一个问题，我们通过判断scope_name和id是否相同来处理，函数引用传参问题，我们则在C语言函数定义时换成指针，并在引用时在相应变量前加&。拓展string问题，解决方案很多，我们采取先设定一个固定的值，作为初始化char*的长度，至于数组维度问题，我们使用bias来维护最低值。

总的来说，遇到问题是不可避免的，但我们依然设计了合理有效的解决方案来解决。具体参见各个章节。