北京邮电大学

《编译原理与技术课程设计》  
报 告

**指导教师 ： 李文生**

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **班级** | **学号** | **备注** |
| **徐永杰** | **2015211301** | **2015211135** |  |
| **黄勇康** | **2015211301** | **2015211137** |  |
| **贾东港** | **2015211301** | **2015211138** |  |
| **甘钊宇** | **2015211301** | **2015211143** | **组长** |
| **田哲元** | **2015211301** | **2015211146** |  |
| **田国梁** | **2015211302** | **2015211169** |  |

**计算机学院**

**2018 年 5 月**

目录

[1.课程设计题目 5](#_Toc513745754)

[2.课程设计目标和要求 5](#_Toc513745755)

[2.1设计目标 5](#_Toc513745756)

[2.2设计要求 5](#_Toc513745757)

[3.需求分析 5](#_Toc513745758)

[3.1数据流图 5](#_Toc513745759)

[3.2词法分析需求分析 5](#_Toc513745760)

[3.2.1功能分析 6](#_Toc513745761)

[3.2.2单词种类、单词模式与右线性文法 6](#_Toc513745762)

[3.2.3注释与分隔符 7](#_Toc513745763)

[3.2.4错误处理 7](#_Toc513745764)

[3.3语法分析需求分析 7](#_Toc513745765)

[3.3.1功能分析 7](#_Toc513745766)

[3.3.2语法结构与文法 7](#_Toc513745767)

[3.3.3语法错误类型 10](#_Toc513745768)

[3.3.4分析方法 10](#_Toc513745769)

[3.3.5改写文法 11](#_Toc513745770)

[3.4语义分析 11](#_Toc513745771)

[3.4.1功能分析 11](#_Toc513745772)

[3.4.2类型 11](#_Toc513745773)

[3.4.3类型检查 11](#_Toc513745774)

[3.4.4符号表 11](#_Toc513745775)

[3.4.5作用域的判定 11](#_Toc513745776)

[3.5代码生成 11](#_Toc513745777)

[3.5.1 Pascal-s — C语言的映射关系 12](#_Toc513745778)

[3.5.2 Pascal-S—C语言的语法映射关系 14](#_Toc513745779)

[4.开发环境 17](#_Toc513745780)

[5.总体设计说明 17](#_Toc513745781)

[5.1数据结构设计 17](#_Toc513745782)

[5.1.1符号表 17](#_Toc513745783)

[5.2总体结构设计 18](#_Toc513745784)

[5.2.1功能模块的划分 18](#_Toc513745785)

[5.2.2模块功能 18](#_Toc513745786)

[5.2.3模块之间的关系 19](#_Toc513745787)

[5.2.4模块之间的接口 19](#_Toc513745788)

[5.3用户接口设计 22](#_Toc513745789)

[6.详细设计说明 22](#_Toc513745790)

[6.1词法分析器的详细设计 22](#_Toc513745791)

[6.1.1接口描述 22](#_Toc513745792)

[6.1.2功能描述 23](#_Toc513745793)

[6.1.3所用数据结构说明 25](#_Toc513745794)

[6.1.4算法描述 25](#_Toc513745795)

[6.2语法分析器的详细设计 25](#_Toc513745796)

[6.2.1接口描述 25](#_Toc513745797)

[6.2.2功能描述 25](#_Toc513745798)

[6.2.3所用数据结构说明 31](#_Toc513745799)

[6.2.4算法描述 31](#_Toc513745800)

[6.3语义分析器的详细设计 31](#_Toc513745801)

[6.3.1接口描述 31](#_Toc513745802)

[6.3.2功能描述 32](#_Toc513745803)

[6.3.3所用数据结构说明 32](#_Toc513745804)

[6.3.4算法描述 32](#_Toc513745805)

[6.4代码生成器的详细设计 32](#_Toc513745806)

[6.4.1接口描述 33](#_Toc513745807)

[6.4.2功能描述 33](#_Toc513745808)

[6.4.3所用数据结构说明 33](#_Toc513745809)

[6.4.4算法描述 33](#_Toc513745810)

[6.5符号表的详细设计 33](#_Toc513745811)

[6.5.1接口描述 33](#_Toc513745812)

[6.5.2功能描述 36](#_Toc513745813)

[6.5.3所用数据结构说明 37](#_Toc513745814)

[6.5.4算法描述 37](#_Toc513745815)

[7.程序清单 38](#_Toc513745816)

[8.测试报告 38](#_Toc513745817)

[8.1测试环境 38](#_Toc513745818)

[8.2测试计划 38](#_Toc513745819)

[8.3词法分析测试报告 38](#_Toc513745820)

[8.3.1测试一 38](#_Toc513745821)

[8.3.2测试二 39](#_Toc513745822)

[8.4语法分析测试报告 40](#_Toc513745823)

[8.4.1测试一 40](#_Toc513745824)

[8.4.2测试二 41](#_Toc513745825)

[8.5语义分析测试报告 42](#_Toc513745826)

[8.5.1测试一 42](#_Toc513745827)

[8.5.2测试二 45](#_Toc513745828)

[8.5.3测试三 48](#_Toc513745829)

[8.6代码生成测试报告 49](#_Toc513745830)

[8.6.1测试一 49](#_Toc513745831)

[8.6.2测试二 49](#_Toc513745832)

[8.6.3测试三 50](#_Toc513745833)

[8.6.4测试四 51](#_Toc513745834)

[8.6.5测试五 53](#_Toc513745835)

[8.7汇总测试 56](#_Toc513745836)

[9.实验总结 58](#_Toc513745837)

[9.1词法分析部分 58](#_Toc513745838)

[9.2语法分析部分 59](#_Toc513745839)

[9.3语义分析部分 59](#_Toc513745840)

[9.4代码生成部分 60](#_Toc513745841)

1. **课程设计题目**

Pascal-S语言编译程序的设计与实现

1. **课程设计目标和要求**
   1. **设计目标**

按照所给Pascal-S语言的语法，参考Pascal语言的语义，设计并实现Pascal-S语言的编译程序。

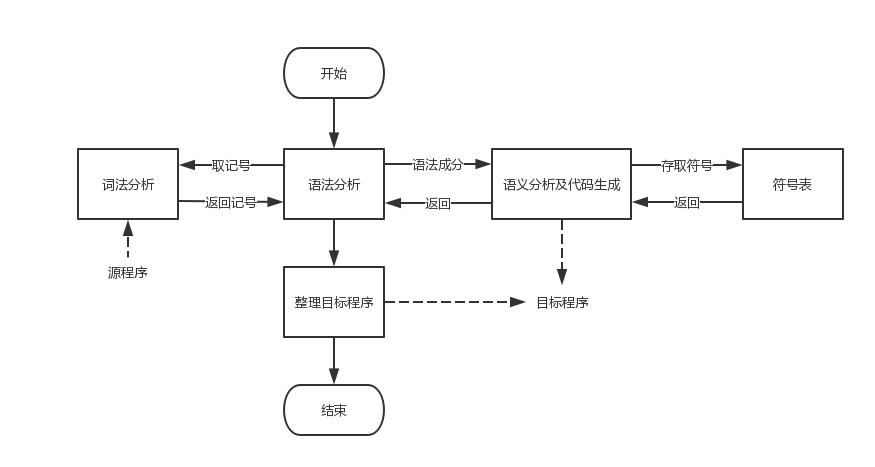
* 1. **设计要求**

要求给出各阶段的设计成果，如：

* 需求分析报告
* 总体设计报告（软件功能描述、功能模块划分、软件结构图、符号表结构设计、模块间接口定义等）
* 详细设计报告（模块功能、输入/输出、处理逻辑等）
* 编码实现（源程序、可执行程序）
* 测试报告（测试计划、测试用例、测试结果及分析等）

1. **需求分析**
   1. **数据流图**

根据一遍编译器的一般设计架构，本编译器按功能划分为词法分析、语法分析、语义分析、代码生成和符号表五个部分。各部分之间的数据流如下：



* 1. **词法分析需求分析**
     1. **功能分析**

词法分析主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal语言的词法规则识别出一个个单词符号，把识别出来的标识符存入符号表中，并产生用于语法分析的记号序列，在词法分析过程中还可以完成用户接口有关的一些任务，如跳过注释和空格，把来自编译程序的错误信息和源程序联系起来，如记住单词在源程序中的行/列位置，从而行号可以作为错误信息的一部分提示给用户。

* + 1. **单词种类、单词模式与右线性文法**

1. **关键字**

Pascal语言中共有三十五个关键字，关键字作为保留字,分为六类列出：

* 程序、函数和过程内的起始符号：program，function，procedure；
* 说明部分专用定义符： array，const，file，label，packed，var，record，set，type；
* 语句用符；case，of，do，for，while，repeat，until，if，then， else，to，downto，goto，with；
* 运算符：and，or，not，div，mod，in；
* 分隔符号：begin，end；
* 空指针常量：nil；

1. **标识符**

标识符：它是用来标识程序、函数、过程、类型、常量、变量等名字，Pascal中允许的有效长度为8个字符，可使用英文字母、数字、下划线，但必须以字母开头，后面可跟字母或数字，中间不能含空格。它有标准标识符与自定义标识符两种。

1. 标准标识符是系统预先定义好的标识符，它们有特定的含义，按规定共有四十个。

* 标准常量符：false，true，maxint；
* 标准类型名：boolean，char，integer，real，text；
* 标准文件名：input，output；
* 标准函数名：(算术函数)abs，sqr，sqrt，exp，ln，sin，cos，tan， arctan，random，frac，oddeof，eoln等；
* （转换函数）ord，pred，round，chr，succ，trunc；
* 标准过程名：read，readln，write，writeln，put，dispose，get，new， pack，page，reset，rewrite，unpack

标识符的记号id，匹配以字母开头的字母数字串。

右线性文法：

Letter->[a-zA-Z]

Digit->[0-9]

Id->letter(letter|digit)\*

1. 自定义标识符：它是由用户自己定义的标识符，它可以是变量名、常量名或过程名。但要注意，禁用关键字，不能数字开头，不含空格及非字符非数字的字符。
2. **算术运算符**

+、-、\*、/（除）、div（整除）、mod（求余）等6个.

1. **关系运算符**

relop -> =|<>|<|<=|>|>=

Addop-> +|- | or

Mulop-> \*| / |div|mod | and

Assignop-> :=

1. **常量与变量**

常量：指在程序运行过程中不能被修改的量。在Pascal中的常量有：整型、实型、布尔型和字符型、符号常量、字符串常量等六种。

变量:在程序运行过程中,其值可以改变的量为变量。变量有 变量名、变量类型和变量值等三个要素。

* + 1. **注释与分隔符**

源程序中的关键字（除开头的program和末尾的end之外）前、后必须有空格符或换行符，其它词汇间的空格符是可选的。源程序中的注释：用一对花括号括起来，可以出现在任何单词之后。编译程序应该可以处理注释。

* + 1. **错误处理**

如果词法分析程序从源程序中读入了一个不合法的字符即Pascal语言不包括此字符开头的单词符号。词法分析显示打印错误信息，并跳过这个字符，然后转开始状态继续识别和分析下一个单词符号。

* 1. **语法分析需求分析**
     1. **功能分析**

根据Pascal-S语言的语法规则，从词法分析程序产生的记号序列中识别出各种语法成分，同时进行语法检查，为语义分析和代码生成做准备。

输入：记号序列；输出：分析树。

* + 1. **语法结构与文法**
       1. **（主）程序结构**

program <程序名>(input, output);

<常量声明>

<变量声明>

<函数定义/过程定义>

<主程序块>

文法：

* + - 1. **常量定义**

const <常量名1>=<值1>; …; <常量名n>=<值n>;

值可以是数字也可以是字符串，不用声明数据类型（仅允许数字和字符）。

文法：

* + - 1. **变量定义**

var <变量列表1>:<类型1>; …; <变量列表n>:<类型n>;

变量列表中的变量用逗号隔开。 real / integer / char / boolean

文法：

* + - 1. **表达式/赋值语句**

<变量>:=<表达式>

表达式有算术表达式和逻辑表达式。

文法：

* + - 1. **输入/输出语句**

输入：read(<变量表>); readln(<变量表>)

输出：write(<输出表>); writeln(<输出表>)

Pascal-S暂不支持。

* + - 1. **选择结构**
* if 语句

if <条件> then

<语句1>

else

<语句2>

* case 语句

case <表达式> of

<值表1>: <语句1>

<值表2>: <语句2>

…

<值表n>: <语句n>

[else <语句>]

end

Pascal-S暂不支持。

* + - 1. **循环结构**
* for 语句

for <循环变量>:=<初值> to <终值> do <语句>

文法：

for <循环变量>:=<终止> downto <初值> do <语句>

Pascal-S暂不支持。

* while 语句

while <条件> do

<循环体>

Pascal-S暂不支持。

* repeat 语句

repeat

<循环体>

until <条件>

Pascal-S暂不支持。

* + - 1. **函数与过程结构**
* 函数

function <函数名> (<形式参数表>): <返回值类型>;

<常量声明>;

<变量声明>;

begin

<函数体>

end

形式参数表与声明变量的文法一致

文法：

* 过程

procedure <过程名> (<形式参数表>);

<常量声明>;

<变量声明>;

begin

<过程体>

end

文法同上。

* + - 1. **自定义数据类型**

type <类型名>=<类型>

* 子界类型

<上界> .. <下界>

文法：

* 枚举类型

(<枚举列表>)

Pascal-S暂不支持。

* 集合类型

set of <类型名>

Pascal-S暂不支持。

* 指针类型

^<类型名>

Pascal-S暂不支持。

* 数组类型

array [<子界1>,…,<子界n>] of <类型名>

* 记录类型

Record

<变量名1>:<类型1>;

…

<变量名n>:<类型n>;

End;

Pascal-S暂不支持。

* 文件类型

file of <类型名>

Pascal-S暂不支持。

* + 1. **语法错误类型**
       1. **缺少元素**

缺少“(”或“)”，算术表达式缺少运算对象，过程/函数定义没有过程体，缺少分号等，例如：

2+3)\*5、6/

program HelloWorld(input, output);

{没有过程体}

* + - 1. **多余的元素**

多余的“(”或“)”，if...else不匹配 等

if a<b then

c:=a

else

c:=b

else {多余的元素}

c:=0

* + - 1. **错误**

for i:=1 tooooooooooooooooo 9 do

readln(a[i])

* + 1. **分析方法**

借助YACC工具自动分析。

* + 1. **改写文法**

首要目标是实现上述所有基本文法，然后视工程进度进行文法扩展，实现其它Pascal-S未支持的语法。

* 1. **语义分析**
     1. **功能分析**

在语法分析的基础上，对抽象语法树进行分析，并生成报错信息。

* + 1. **类型**

显式声明，通过类型表达式进行判断。

* + 1. **类型检查**

静态类型检查。

**（1）需要检查程序内的类型信息是否一致；**

**（2）类型表达式分为不同类型：**

a). 基本类型；

b). 类型名；

c). 类型构造器作用于类型表达式的结果；

d). 类型表达式可以包含变量，变量的值是类型表达式；

**（3）类型等价的判定：**

a). 结构等价的判定；

b). 名字等价的判定；

* + 1. **符号表**
       1. **符号表内容**
       - 名字
       - 类型
       - 存储地址
       - 维数及参数个数
       - 声明行
       - 引用行
       - 链域
       1. **符号表组织**

采用栈式散列符号表；通过块索引表进行定位以及重定位操作。

* + 1. **作用域的判定**

通过定位以及重定位操作进行作用域的判定。

* 1. **代码生成**
     1. **Pascal-s — C语言的映射关系**
* programstruct → program\_head ；program\_body .

{

program\_head(转换为相应C头文件);

program\_body

}

* program\_head → program id ( idlist )

{与C语言不同，转换为C语言格式}

* program\_body →

const\_declarations{注意相应的变量作用域以及实现}

var\_declarations{注意相应的变量作用域以及实现}

subprogram\_declarations

compound\_statement

* idlist → idlist , id | id
* const\_declarations → const const\_declaration ; {相应作用域转换}| ∅
* const\_declaration → const\_declaration ; id = const\_value | id = const\_value
* const\_value → + id | - id | id | + num | - num | num | ′ letter ′ {转换为c语言相应格式}
* var\_declarations → var var\_declaration ; | ∅

{var\_declarations → var\_declaration ; | ∅}

* var\_declaration → var\_declaration ; idlist : type | idlist : type

{ idlist : type变换为 type idlist }

* type → simple\_type | array [ period ] of simple\_type

{变换为相应的C语言规范}

* simple\_type → integer | real | boolean | char
* period → period ， digits .. digits | digits .. digits
* subprogram\_declarations →subprogram\_declarations subprogram; | ∅

{subprogram\_declarations → subprogram; | ∅}

* subprogram → subprogram\_head ; subprogram\_body

subprogram → subprogram\_head

{

subprogram\_body

}

* subprogram\_head → procedure id formal\_parameter

| function id formal\_parameter: simple\_type

subprogram\_head →simple\_type function id formal\_parameter

* formal\_parameter → ( parameter\_list ) | ∅
* parameter\_list → parameter\_list ; parameter | parameter
* parameter → var var\_parameter | value\_parameter
* var\_parameter→ var value\_parameter

{

var.type value\_parameter

}

* value\_parameter → idlist : simple\_type
* subprogram\_body → const\_declarations

var\_declarations

compound\_statement

* compound\_statement→begin statement\_list end

{

statement\_list（并根据C语言相应规范进行对应）

}

* statement\_list → statement\_list ; statement | statement
* statement → variable assignop expression

| procedure\_call

| compound\_statement

| if expression then statement else\_part

{

If(expression){statement}

else\_part

}

| for id assignop expression to expression do statement

{

For(id=;expression to expression)

{ statement}

}

|∅ 

* variable→id id\_varpart
* id\_varpart →[ expression\_list ] | ∅
* procedure\_call → id | id ( expression\_list )
* else\_part → else statement | ∅
* expression\_list → expression\_list , expression | expression
* expression → simple\_expression relop simple\_expression

| simple\_expression

* simple\_expression → simple\_expression addop term | term
* term → term mulop factor | factor
* factor → num

| variable

| id ( expression\_list )

| ( expression )

| not factor

| uminus factor

* + 1. **Pascal-S—C语言的语法映射关系**

有 ~~删除线~~ 的项目是当前基本Pascal-S语言不支持的语法，需要扩展文法。

* + - 1. **主程序结构**

Pascal-S代码：

program <程序名>(input, output);

<常量声明>

<变量声明>

<函数定义/过程定义>

<主程序块>

C代码：

<全局常量声明>

<全局变量声明>

<函数定义>

int main(int argc, char\* argv[])

{

<主程序块>

return 0;

}

* + - 1. **常量定义**

Pascal-S代码：

const <常量名1>=<值1>; …; <常量名n>=<值n>;

C代码：

const <常量1类型> <常量名1>=<值1>;

…

const <常量n类型> <常量名n>=<值n>;

* + - 1. **变量定义**

Pascal-S代码：

var <变量列表1>:<类型1>; …; <变量列表n>:<类型n>;

C代码：

<类型1> <变量列表1>;

…

<类型n> <变量列表n>;

* + - 1. **表达式/赋值语句**

Pascal-S代码：<变量名>:=<表达式>

C代码：<变量名>=<表达式>

* + - 1. **if 语句**

Pascal-S代码：

if <条件> then

<语句1>

else

<语句2>

C代码：

if (<条件>)

<语句1>

else

<语句2>

* + - 1. **~~case 语句（分支判断）~~**

Pascal-S代码：

case <表达式> of

<值表1>: <语句1>

<值表2>: <语句2>

…

<值表n>: <语句n>

[else <语句>]

end

C代码：

switch (<表达式>)

{

/\* 值表1 \*/

case <值1-1>:

case <值1-2>:

…

case <值1-n>:

<语句1>

break;

…

/\* 值表n \*/

case <值n-1>:

…

case <值n-n>:

<语句n>

break;

[default: <语句>]

}

* + - 1. **for语句**

Pascal-S语言：

for <循环变量>:=<初值> to <终值> do <语句>

~~for <循环变量>:=<初值> downto <终值> do <语句>~~

C语言：

for (<循环变量>:=<初值>; <循环变量><=<终值>; <循环变量>++)

<语句>

for (<循环变量>:=<初值>; <循环变量>>=<终值>; <循环变量>--)

<语句>

* + - 1. **~~while句~~**

Pascal-S语言：

while <条件> do

<循环体>

C语言：

while (<条件>)

<循环体>

* + - 1. **~~repeat语句（do-while循环）~~**

Pascal-S语言：

repeat

<循环体>

until <条件>

C语言：

do

<循环体>

while (<条件>)

* + - 1. **函数**

Pascal-S语言：

function <函数名> (<形式参数表>): <返回值类型>;

<常量声明>;

<变量声明>;

begin

<函数体>

end

C语言：

<返回值类型> <函数名> (<形式参数表>)

{

<常量声明>;

<变量声明>;

<函数体>

}

* + - 1. **过程**

Pascal-S语言：

procedure <过程名> (<形式参数表>);

<常量声明>;

<变量声明>;

begin

<过程体>

end

C语言：

void <过程名> (<形式参数表>)

{

<常量声明>

<变量声明>

<过程体>

}

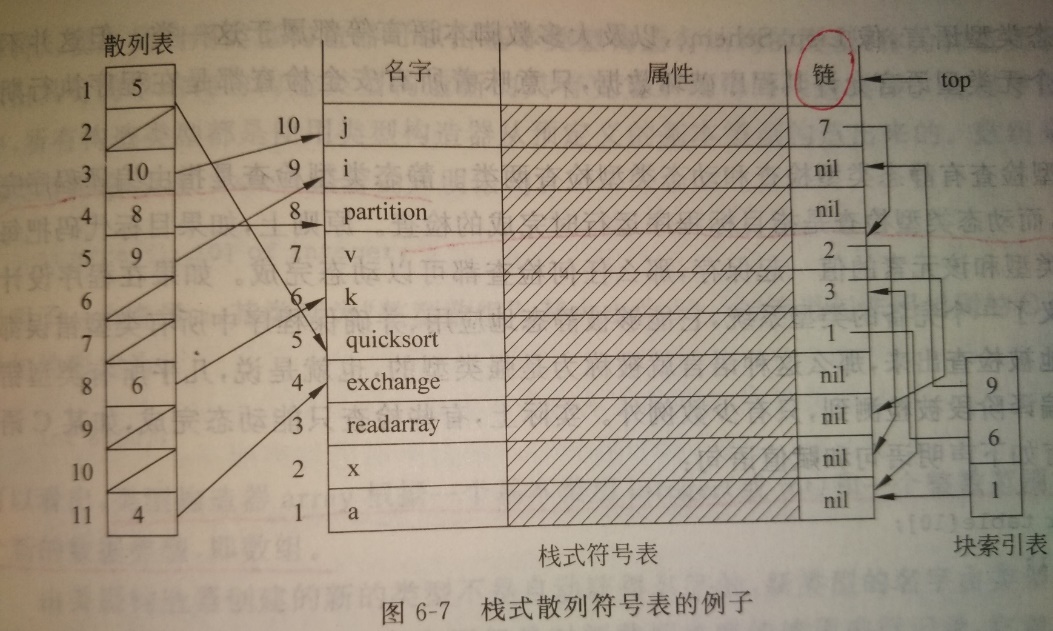
1. **开发环境**

* Windows 操作系统
* Flex / Bison (LEX/YACC)
* Microsoft Visual Studio Code 文本编辑器
* GNU Compiler Collection

1. **总体设计说明**
   1. **数据结构设计**
      1. **符号表**

符号表是编译程序使用的一个非常重要的数据结构。基于符号表中记录的信息，可以检查源程序上下文语义的正确性，同时在代码生成时可以辅助我们正确地生成代码。

本编译器使用了栈式散列符号表，它由散列表、栈式符号表、块索引表三个部分组成。



* + - 1. **散列表**

散列表是根据关键码值而直接进行访问的数据结构，它通过预先设置好的散列函数，把关键码值映射到表中一个位置来访问记录，以加快符号查找的速度。同时，我们在每一个符号的属性中设置一个链域，使其指向相邻的同散列值的变量的位置，将同散列值的符号组成了一个符号链，即使用链地址法解决了散列冲突。

本编译器设计了如下散列函数：

取首字母作为散列值的结果。合法的ID将会产生52个散列值，由此产生了如下散列表，并存储在char[]类型的全局变量 hash\_table 中：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 键（key） | ‘a’ | ‘b’ | … | ‘Z’ |
| 位置（value） | <position1> | <position2> | … | <position52> |

* + - 1. **栈式符号表**

块程序结构语言常用栈式符号表，它通过符号栈来记录变量的属性。根据已有的设计经验以及实际情况，本编译器设计了如下变量属性：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 名字 | 类型 | 子类型 | 维数 | 是否  常量 | 是否  引用 | 函数  参数 | 链域 |
|  |  |  |  |  |  |  |  |
| …… | | | | | | | |
|  |  |  |  |  |  |  |  |

各属性的含义如下：

* 名字：标识符名称；
* 类型：标识符类型；
* 子类型：数组元素的类型/函数返回值类型；
* 维数：数组维度/函数参数个数；
* 是否常量：标记该标识符是否为常量；
* 是否引用：标记该标识符是否为引用；
* 函数参数：函数参数域，由以下字段构成：
* 参数的散列表；
* 参数的栈式符号表；
* 符号表头部指针；
* 链域：指向下一个拥有相同散列值的符号。
  + - 1. **块索引表**

栈结构，记录每个块的起始位置。

* 1. **总体结构设计**
     1. **功能模块的划分**

根据需求分析的数据流图，本编译器划分为5个功能模块：**词法分析器**、**语法分析器**、**语义分析器**、**代码生成器**以及**符号表**。

* + 1. **模块功能**
       1. **词法分析**

主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal 语言的词法规则识别出一个个单词符号，并产生用于语法分析的记号序列。

* + - 1. **语法分析**

根据Pascal-S 语言的语法规则，从词法分析程序产生的记号序列中识别出各种语法成分，同时进行语法检查，为语义分析和代码生成做准备。

* + - 1. **语义分析**

在语法分析的基础上，对抽象语法树进行分析，并生成报错信息。

* + - 1. **代码生成**

把经过分析阶段所获得的源程序的中间表示翻译成目标代码。

* + - 1. **符号表**

符号表是编译程序使用的一个非常重要的数据结构。基于符号表中记录的信息，可以检查源程序上下文语义的正确性，可以辅助正确地生成代码。

* + 1. **模块之间的关系**

以语法分析器为中心，各模块之间形成调用与被调用的关系，数据在模块之间传递。

* 语法分析器调用词法分析器的接口，词法分析器取出源文件中的一个单词并返回；
* 语法分析器调用语义分析器的接口，将语法成分发送给语义分析器，语义分析器检测语义的正确性，并返回结果；
* 语义分析器调用符号表的接口，请求读取或写入一个文法符号，符号表返回操作的结果；
* 语法分析器调用代码生成器的接口，将语法成分发送给代码生成器，代码生成器产生对应的C语言代码并写入文件，然后返回。
  + 1. **模块之间的接口**
       1. **语法分析器与词法分析器之间的接口**

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 描述 |
| yylval | char\* | 单词的值 |
| yylineno | int | 当前行号 |
| row | int | 当前列号 |
| len | int | 单词长度 |
| yylloc | struct YYLTYPE | 单词的位置信息 |

* + - 1. **语法分析器与语义分析器之间的接口**

翻译方案。语法分析器直接执行对应的语义分析过程。

* + - 1. **语义分析器与符号表之间的接口**

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| int get\_parameter(char \* a) | a为函数名或过程名 | 返回参数数组中该函数或过程名对应的第一个参数的位置 | 得到参数数组中该函数或过程名对应的第一个参数的位置 |
| unsigned int find\_element(const char\* name); | name为符号的名称 | 返回该元素在符号表中的位置 | 得到该元素在符号表中的位置 |
| STATUS add\_element(const char\* name, symbol\_type type, symbol\_type sub\_type,  unsigned int dimension, const int is\_const, const int is\_reference); | name为符号的名称  type为元素类型  subtype 为符号返回类型  dimension为数组元素的维数或过程的参数数量或函数的参数数量  Is\_const 为变量是否为常量  Is\_reference 为参数是否为引用 | 返回成功或失败 | 将元素加入符号表 |
| STATUS add\_parameter\_element(  const char\* function\_name,  const char\* name, symbol\_type type, symbol\_type sub\_type,  unsigned int dimension, const int is\_const, const int is\_reference); | function\_name为该元素对应的函数或过程名称  name为参数的名称  type为参数类型  subtype 为符号返回类型  dimension为数组元素的维数或过程的参数数量或函数的参数数量  Is\_const 为变量是否为常量  Is\_reference 为参数是否为引用 | 返回成功或失败 | 将参数加入参数表 |
| STATUS locate\_table(void); | 无 | 返回成功 | 定位 |
| STATUS relocate\_table(void); | 无 | 返回成功 | 重定位 |
| symbol\_type get\_type(  const char\* name); | 元素名 | typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | 获取元素的type |
| symbol\_type get\_return\_type(  const char\* name); | 函数或过程名 | typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | 获取元素的subtype |
| unsigned int get\_parameter\_number(  const char\* name); | 函数或过程名 | 该函数或过程对应的参数个数 | 获取该函数或过程对应的参数个数 |
| int is\_const(  const char\* name); | 符号名 | 是，否 | 判断该符号是否是常量 |
| int is\_reference(const char\* name); | 符号名 | 是，否 | 判断该符号是否引用 |
| int is\_reference\_parameter(  const char\* function\_name,  const char\* name); | 参数名 | 是，否 | 判断该参数是否引用 |
| STATUS check\_parameter\_types(  const char\* function\_name,  const int n, const symbol\_type  st[]); | function\_name为函数或过程名  n为表达式中的dimension  st[]为表达式中参数的类型数组 | 成功或失败 | 函数调用的类型检查 |

* + - 1. **语法分析器与代码生成器之间的接口**

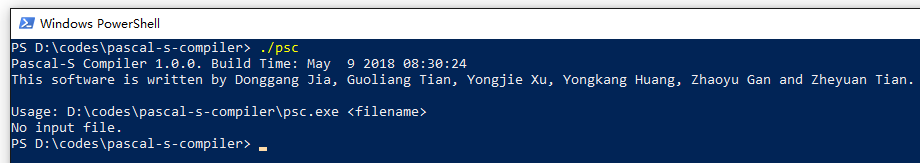
|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| void print(  const char\* a) | a 为目标代码 | 无 | 输出对应的目标代码 |
| void file\_clean() | 无 | 无 | 清空目标代码文件 |

* 1. **用户接口设计**

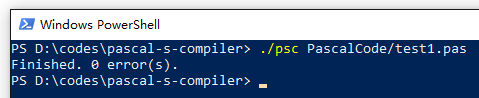
通过C语言main函数的argc与argv两个参数，我们可以接收用户传入的参数，进而确定用户的请求内容。

本编译程序是控制台应用程序，用户通过命令行参数传入需要编译的文件路径，编译得到的目标代码通过文件输出，编译过程中的错误提示通过控制台（stderr流）输出。

当用户没有提供命令行参数时，编译器会给出当前版本信息以及帮助信息，如图所示。



当用户提供需要编译的Pascal文件时，编译器会输出文件，并给出相应的错误信息（如果有），如下图。



1. **详细设计说明**
   1. **词法分析器的详细设计**
      1. **接口描述**

词法分析部分采用了GNU Flex工具来帮助完成词法分析器代码的生成，LEX有声明部分，翻译规则部分和辅助过程部分。在具体的执行动作中由yylval识别的字符的记号返回给语法分析，声明了len变量来记录识别的字符的长度给语法分析，声明的row变量来记录识别动作时的列号给语法分析，由return直接返回识别的字符的属性给后面的语法分析部分。

Flex工具提供了一个接口用于读取下一个单词符号。

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| yylex() | 无 | int  返回单词的类型 | 读取下一个单词符号 |

同时通过全局变量传递单词的属性信息。

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 描述 |
| yylval | char\* | 单词的值 |
| yylineno | int | 当前行号 |
| row | int | 当前列号 |
| len | int | 单词长度 |
| yylloc | struct YYLTYPE | 单词的位置信息 |

* + 1. **功能描述**

主要任务是从左到右逐个字符地对源程序进行扫描，按照Pascal 语言的词法规则识别出一个个单词符号，并产生用于语法分析的记号序列。

通过正则表达式，词法分析器可以识别以下模式，并通过yylex函数返回。

|  |  |  |
| --- | --- | --- |
| 记号 | 模式 | 描述 |
| digit | [0-9] | 数位 |
| letter | [A-Za-z] | 字母 |
| id | ({letter}|\_)+({letter}|{digit}|\_)\* | 标识符 |
| num | [0-9]\*(\.[0-9]+)? | 数字 |
| string | '[^']\*' | 字符串 |
| expl | \{[^}]\*\} | 注释 |
| SYM\_program | program | program 关键字 |
| SYM\_function | function | function 关键字 |
| SYM\_procedure | procedure | procedure关键字 |
| SYM\_string | string | string关键字 |
| SYM\_array | array | array关键字 |
| SYM\_const | const | const关键字 |
| SYM\_file | file | file关键字 |
| SYM\_label | label | label关键字 |
| SYM\_packed | packed | packed关键字 |
| SYM\_var | var | var关键字 |
| SYM\_record | record | record关键字 |
| SYM\_set | set | set关键字 |
| SYM\_type | type | type关键字 |
| SYM\_case | case | case关键字 |
| SYM\_of | of | of关键字 |
| SYM\_do | do | do关键字 |
| SYM\_for | for | for关键字 |
| SYM\_while | while | while关键字 |
| SYM\_repeat | repeat | repeat关键字 |
| SYM\_until | until | until关键字 |
| SYM\_if | if | if关键字 |
| SYM\_then | then | then关键字 |
| SYM\_else | else | else关键字 |
| SYM\_to | to | to关键字 |
| SYM\_downto | downto | downto关键字 |
| SYM\_goto | goto | goto关键字 |
| SYM\_with | with | with关键字 |
| SYM\_and | and | and关键字 |
| SYM\_or | or | or关键字 |
| SYM\_not | not | not关键字 |
| SYM\_div | div | div关键字 |
| SYM\_mod | mod | mod关键字 |
| SYM\_in | in | in关键字 |
| SYM\_begin | begin | begin关键字 |
| SYM\_end | end | end关键字 |
| SYM\_nil | nil | nil关键字 |
| SYM\_integer | integer | integer关键字 |
| SYM\_boolean | boolean | boolean关键字 |
| SYM\_char | char | char关键字 |
| SYM\_real | real | real关键字 |
| SYM\_string | string | string关键字 |
| OPR\_become | := | := 运算符 |
| OPR\_plus | + | + 运算符 |
| OPR\_minus | - | - 运算符 |
| OPR\_times | \* | \* 运算符 |
| OPR\_slash | / | / 运算符 |
| OPR\_lss | < | < 运算符 |
| OPR\_leq | <= | <= 运算符 |
| OPR\_gtr | > | > 运算符 |
| OPR\_geq | >= | >= 运算符 |
| OPR\_neq | <> | <> 运算符 |
| OPR\_eql | = | = 运算符 |
| OPR\_jin | # | # 运算符 |
| BDY\_lparen | ( | ( 符号 |
| BDY\_rparen | ) | ) 符号 |
| BDY\_comma | , | , 符号 |
| BDY\_semicolon | ; | ; 符号 |
| BDY\_lfang | [ | [ 符号 |
| BDY\_rfang | ] | ] 符号 |
| BDY\_lhua | { | { 符号 |
| BDY\_rhua | } | } 符号 |
| BDY\_colon | : | : 符号 |
| BDY\_dot | . | . 符号 |
| OPR\_eval | = | = 符号 |
| BDY\_squo | ‘ | ‘ 符号 |
| BDY\_ddot | .. | .. 符号 |

* + 1. **所用数据结构说明**

yylval定义了可以为char\*型，len为int型变量，row为int型变量。Keyword为二维字符数组。在id，num等执行动作中定义了char\*型变量temp来暂存识别的字符。

* + 1. **算法描述**

在声明部分，声明了YYSTYPE char\*，<stdio.h>等头文件，还列出了关键字数组，这些部分直接复制到lex.yy.c里。在正则定义式里，定义了digit(表示整数）,letter（表示单个字母）,id（表示自己定义的标识符）,num（表示所有数字）,str（表示字符串）,expl（表示翻译部分）。在翻译规则部分，左边是正则表达式，右边是执行的动作；翻译部分先识别关键字，然后是<,>等relop符号，最后是digit,letter,id,num,strhe exopl。在具体的执行动作中由yyval返回识别的字符的记号，声明了int型len变量来记录识别的字符的长度，声明的row变量来记录识别动作时的列号，由return直接返回识别的字符的属性给后面的语法分析部分。在辅助过程部分，定义了yywrap函数来达到在识别一个字符后，词法分析器识别动作不会停止。

* 1. **语法分析器的详细设计**
     1. **接口描述**

语法分析器借助 GNU Bison生成。Bison预留了一个接口，作为语法分析器的启动入口。

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| yyparse() | 无 | int  标识语法分析过程中是否有错误。 | 启动语法分析器。 |

* + 1. **功能描述**

根据Pascal-S 语言的语法规则，从词法分析程序产生的记号序列中识别出各种语法成分，同时进行语法检查，为语义分析和代码生成做准备。

为了调用对应的语义检查过程以及代码生成过程，我们在YACC文件中定义了72个标准翻译方案和 20个附加翻译方案。定义的翻译方案如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 非终结符 | 文法 | 描述 |
| 1 | programstruct | program\_head BDY\_semicolon program\_body BDY\_dot | 程序总结构 |
| 2 | program\_head | SYM\_program id BDY\_lparen idlist BDY\_rparen | 程序声明头部 |
| 3 | program\_body | const\_declarations  var\_declarations  subprogram\_declarations  program\_compound\_statement | 程序内容 |
| 4 | idlist | idlist BDY\_comma id | 标识符列表（多个） |
| 5 | idlist | id | 标识符列表（单个） |
| \*1 | function\_idlist | function\_idlist BDY\_comma id | 函数参数列表（多个） |
| \*2 | function\_idlist | id | 函数参数列表（单个） |
| 6 | const\_declarations | SYM\_const const\_declarations BDY\_semicolon | 常量声明部分（有） |
| 7 | const\_declarrations |  | 常量声明部分（无） |
| 8 | const\_declaration | const\_declaration BDY\_semicolon id OPR\_eql const\_value | 常量列表（多个） |
| 9 | const\_declaration | id OPR\_eql const\_value | 常量列表（单个） |
| 10 | const\_value | OPR\_plus id | +id赋值到常量 |
| 11 | const\_value | OPR\_minus id | -id 赋值到常量 |
| 12 | const\_value | id | id 赋值到常量 |
| 13 | const\_value | OPR\_plus num | 正数赋值到常量 |
| 14 | const\_value | OPR\_minus num | 负数赋值到常量 |
| 15 | const\_value | num | 数字赋值到常量 |
| 16 | const\_value | str | 字符串赋值到常量 |
| 17 | var\_declarations | SYM\_var var\_declaration BDY\_semicolon | 变量声明部分（有） |
| 18 | var\_declarations |  | 变量声明部分（无） |
| 19 | var\_declaration | var\_declaration BDY\_semicolon idlist BDY\_colon type | 变量列表（多个） |
| 20 | var\_declaration | idlist BDY\_colon type | 变量列表（单个） |
| 21 | type | simple\_type | 简单类型 |
| 22 | type | SYM\_array BDY\_lfang period BDY\_rfang SYM\_of simple\_type | 数组类型 |
| 23 | simple\_type | SYM\_integer | 整数类型 |
| 24 | simple\_type | SYM\_real | 实数类型 |
| 25 | simple\_type | SYM\_boolean | 布尔类型 |
| 26 | simple\_type | SYM\_char | 字符类型 |
| \*3 | simple\_type | SYM\_string | 字符串类型 |
| 27 | period | period BDY\_comma num BDY\_ddot num | 数组声明段（多） |
| 28 | period | num BDY\_ddot num | 数组声明段（单） |
| 29 | subprogram\_declarations | subprogram\_declarations  subprogram  BDY\_semicolon | 子程序声明部分（有，多） |
| 30 | subprogram\_declarations |  | 子程序声明部分（无） |
| 31 | subprogram | subprogram\_head BDY\_semicolon  subprogram\_bdoy | 单个子程序结构 |
| 32 | subprogram\_head | SYM\_procedure id formal\_parameter | 子程序头部（过程） |
| 33 | subprogram\_head | SYM\_function id formal\_parameter BDY\_colon simple\_type | 子程序头部（函数） |
| 34 | formal\_parameter | BDY\_lparen parameter\_list BDY\_rparen | 参数列表形式 |
| 35 | formal\_parameter |  | 参数列表形式（空） |
| 36 | parameter\_list | parameter\_list BDY\_semicolon parameter | 参数列表（多） |
| 37 | parameter\_list | parameter | 参数列表（单） |
| 38 | parameter | var\_parameter | 引用参数类型 |
| 39 | parameter | value\_parameter | 值参数类型 |
| 40 | var\_parameter | SYM\_var value\_parameter | 引用参数形式 |
| 41 | value\_parameter | function\_idlist BDY\_colon simple\_type | 值参数形式 |
| 42 | subprogram\_body | const\_declarations  var\_declarations  compound\_statement | 子函数结构 |
| 43 | compound\_statement | SYM\_begin statement\_list SYM\_end | 子函数过程部分 |
| \*4 | program\_compound\_statement | SYM\_begin statement\_list SYM\_end | 主程序过程部分 |
| 44 | statement\_list | statement\_list BDY\_semicolon statement | 语句列表（多） |
| 45 | statement\_list | statement | 语句列表（单） |
| 46 | statement | variable  assignop expression | 赋值语句 |
| 47 | statement | procedure\_call | 调用过程 |
| 48 | statement | compound\_statement | 语句块 |
| 49 | statement | SYM\_if expression SYM\_then statement else\_part | if语句 |
| 50 | statement | SYM\_for  id assignop expression  SYM\_to expression SYM\_do statement | for循环 |
| 51 | statement |  | 空语句 |
| 52 | variable | id id\_varpart | 变量 |
| 53 | id\_varpart | BDY\_lfang array\_expression\_list BDY\_rfang | 数组索引 |
| 54 | id\_varpart |  | 空 |
| 55 | procedure\_call | id | 无参数子程序调用 |
| 56 | procedure\_call | id BDY\_lparen expression\_list BDY\_rparen | 有参数子程序调用 |
| 57 | else\_part | SYM\_else statement | else部分语句 |
| 58 | else\_part |  | 无else部分语句 |
| 59 | expression\_list | expression\_list BDY\_comma expression | 表达式列表（通常用于数组，多） |
| 60 | expression\_list | expression | 表达式列表（单） |
| 61 | expression | simple\_expression relop simple\_expression | 关系表达式 |
| 62 | expression | simple\_expression | 简单表达式 |
| 63 | simple\_expression | simple\_expression addop term | 加减法表达式 |
| 64 | simple\_expression | term | 表达式 |
| 65 | term | term mulop factor | 乘除法表达式 |
| 66 | term | factor | 表达式 |
| 67 | factor | num | 数字表达式 |
| \*5 | factor | str | 字符串表达式 |
| 68 | factor | variable | 变量表达式 |
| 69 | factor | id BDY\_lparen expression\_list BDY\_rparen | 子程序调用表达式 |
| 70 | factor | BDY\_lparen expression BDY\_rparen | 括号表达式 |
| 71 | factor | SYM\_not | not表达式 |
| 72 | factor | uminus factor | 负数表达式 |
| \*6 | array\_expression\_list | array\_expression\_list BDY\_comma array\_expression | （数组）同59 |
| \*7 | array\_expression\_list | array\_expression | （数组）同60 |
| \*8 | array\_expression | array\_simple\_expression | （数组）同62 |
| \*9 | array\_simple\_expression | array\_simple\_expression addop array\_term | （数组）同63 |
| \*10 | array\_simple\_expression | array\_term | （数组）同64 |
| \*11 | array\_term | array\_term mulop array\_factor | （数组）同65 |
| \*12 | array\_term | array\_factor | （数组）同66 |
| \*13 | array\_factor | num | （数组）同67 |
| \*14 | array\_factor | variable | （数组）同68 |
| \*15 | array\_factor | id BDY\_lparen expression\_list BDY\_rparen | （数组）同69 |
| \*16 | array\_factor | BDY\_lparen expression BDY\_rparen | （数组）同70 |
| \*17 | array\_factor | SYM\_not array\_factor | （数组）同71 |
| \*18 | array\_factor | uminus array\_factor | （数组）同72 |
| \*19 | str | str string | 字符串 |
| \*20 | str | string | 字符串 |

* + 1. **所用数据结构说明**

GNU Bison使用自底向上的LR分析方法，此分析方法需要使用两个栈，分别作为符号栈和状态栈。在生成代码中，这两个栈分别被声明为yyvsa和yyssa。

* + 1. **算法描述**

**LR分析方法**

do {

令S是栈顶状态,a是ip所指向的符号;

if (action[S,a] = Shift S’)

把 a 和 S’分别压入符号栈和状态栈的栈顶;

推进ip,使它指向下一个输入符号;

};

else if (action[S,a] = reduce by A→B) {

从栈顶弹出 |β| 个符号;

令 S’是当前栈顶状态，把A和goto[S’,A]分别压入符号栈和状态栈的栈顶;

输出生成式 A→β;

};

else if (action[S,a] = accept) return;

else error();

} while(1);

* 1. **语义分析器的详细设计**
     1. **接口描述**

语义分析中用于暂存参数的结构体 p\_list：

|  |  |  |
| --- | --- | --- |
| 字段 | 类型 | 描述 |
| name | char\* | 参数名 |
| type | char\* | 参数类型 |
| is\_var | int | 是否引用型参数 |
| proc\_func\_name | char\* | 该参数对应的函数名或过程名 |

函数描述：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| symbol\_type return\_symtype(char\* a) | a为“int”“float”等字符串型类型 | typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | 将字符串类型转化成符号表中的类型表示方式 |
| char \* return\_chartype(symbol\_type a) | a为typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | “int”“float”等字符串型类型 | 将符号表中的类型表示方式转化成字符串类型 |
| unsigned int IS\_DEFINED(const char\* name); | 符号名 | 是，否 | 判断该符号是否已定义 |
| void into\_pf(const char\* name) | 函数或过程名或数组名 | 无 | 进入函数或过程调用时进行的保存现场操作 |
| void out\_pf() | 无 | 无 | 完成函数或者过程调用所需要的恢复现场操作 |

* + 1. **功能描述**

在语法分析的基础上，对抽象语法树进行分析，并生成报错信息。

* + 1. **所用数据结构说明**

/\*语义分析中用于暂存参数的结构体\*/

typedef struct p\_list

{

char \* name; //参数名

char \* type; //参数类型

int is\_var; //是否引用型参数

char \* proc\_func\_name;//该参数对应的函数名或过程名

}p\_list;

* + 1. **算法描述**

根据不同的语法成分，执行符号表操作，或使用判断语句检查类型是否符合要求等。

* 1. **代码生成器的详细设计**
     1. **接口描述**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口 | 参数 | 返回值 | 描述 |
| char\* char\_strcpy(  char\* a, char\* b) | a,b: 两个字符串 | 字符串 | 传入两个字符串，将两个字符串连接 |
| void file\_clean() | 无 | 无 | 将输出文件情况 |
| char\* code\_get\_type(  char\* a) | 字符串 | 字符串 | 传入一个字符串。返回一个类型 |
| void print(const char\* a) | 字符串 | 无 | 将字符串打印到输出文件中中 |

* + 1. **功能描述**

把经过分析阶段所获得的源程序的中间表示翻译成目标代码。本编译器翻译的目标代码为C++。

* + 1. **所用数据结构说明**

无数据结构。

* + 1. **算法描述**

根据Pascal-S语句产生对应的C++代码。

* 1. **符号表的详细设计**
     1. **接口描述**

符号表提供以下接口，实现符号表的添加、查询、定位以及重定位功能。

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名 | 参数 | 返回值 | 功能 |
| int get\_parameter(char \* a) | a为函数名或过程名 | 返回参数数组中该函数或过程名对应的第一个参数的位置 | 得到参数数组中该函数或过程名对应的第一个参数的位置 |
| unsigned int find\_element(const char\* name); | name为符号的名称 | 返回该元素在符号表中的位置 | 得到该元素在符号表中的位置 |
| STATUS add\_element(const char\* name, symbol\_type type, symbol\_type sub\_type,  unsigned int dimension, const int is\_const, const int is\_reference); | name为符号的名称  type为元素类型  subtype 为符号返回类型  dimension为数组元素的维数或过程的参数数量或函数的参数数量  Is\_const 为变量是否为常量  Is\_reference 为参数是否为引用 | 返回成功或失败 | 将元素加入符号表 |
| STATUS add\_parameter\_element(  const char\* function\_name,  const char\* name, symbol\_type type, symbol\_type sub\_type,  unsigned int dimension, const int is\_const, const int is\_reference); | function\_name为该元素对应的函数或过程名称  name为参数的名称  type为参数类型  subtype 为符号返回类型  dimension为数组元素的维数或过程的参数数量或函数的参数数量  Is\_const 为变量是否为常量  Is\_reference 为参数是否为引用 | 返回成功或失败 | 将参数加入参数表 |
| STATUS locate\_table(void); | 无 | 返回成功 | 定位 |
| STATUS relocate\_table(void); | 无 | 返回成功 | 重定位 |
| symbol\_type get\_type(  const char\* name); | 元素名 | typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | 获取元素的type |
| symbol\_type get\_return\_type(  const char\* name); | 函数或过程名 | typedef enum {TYPE\_ERROR, INT, CHAR, FLOAT, BOOL, ARRAY, FUNCTION, STRING, PROCEDURE} symbol\_type;中的值 | 获取元素的subtype |
| unsigned int get\_parameter\_number(  const char\* name); | 函数或过程名 | 该函数或过程对应的参数个数 | 获取该函数或过程对应的参数个数 |
| int is\_const(  const char\* name); | 符号名 | 是，否 | 判断该符号是否是常量 |
| int is\_reference(const char\* name); | 符号名 | 是，否 | 判断该符号是否引用 |
| int is\_reference\_parameter(  const char\* function\_name,  const char\* name); | 参数名 | 是，否 | 判断该参数是否引用 |
| STATUS check\_parameter\_types(  const char\* function\_name,  const int n, const symbol\_type  st[]); | function\_name为函数或过程名  n为表达式中的dimension  st[]为表达式中参数的类型数组 | 成功或失败 | 函数调用的类型检查 |

定义了以下宏，用于简化函数调用：

|  |  |  |
| --- | --- | --- |
| 宏 | 调用的函数 | 描述 |
| add\_variable(name, type) | add\_element(  name, type,  TYPE\_ERROR, 0, 0, 0) | 添加（基本）变量 |
| add\_const(name, type) | add\_element(name, type,  TYPE\_ERROR, 0, 1, 0) | 添加常量 |
| add\_array(name, element\_type, dimension) | add\_element(name, ARRAY, element\_type,  dimension, 0, 0) | 添加数组 |
| add\_procedure(name, parameter\_number) | add\_element(name, PROCEDURE, TYPE\_ERROR,  parameter\_number, 0, 0) | 添加过程 |
| add\_function(name, return\_type, parameter\_number) | add\_element(name, FUNCTION, return\_type,parameter\_number, 0, 0) | 添加函数 |
| add\_parameter(function\_name,  name, type) | add\_parameter\_element(function\_name, name, type,  TYPE\_ERROR, 0, 0, 0) | 添加函数参数 |
| add\_reference\_parameter(function\_name, name, type) | add\_parameter\_element(function\_name, name, type,  TYPE\_ERROR, 0, 0, 1) | 添加函数引用参数 |

* + 1. **功能描述**

在符号表上最常执行的操作是插入和检索，Pascal-S语言要求变量显式声明，然后才能引用。

对于本次要求变量显示声明的Pascal-S语言，编译程序在处理声明语句时，首先，需要插入操作，因为声明语句是变量属性的初始描述，从中收集到的变量属性值应该记入符号表。其次，还需要检索操作，因为在同一个作用域不允许有变量重名，所以在插入之前要先对符号表进行检索，完成“查重”任务。另外，如果符号表是根据变量名排了序的有序表的话，则在插入操作之前也要先进行检索操作，以找到该变量及其属性应该存放的位置。

同时，Pascal-S语言也是块结构的语言，除了插入和检索外，在符号表上还要进行两种附加的操作，即定位和重定位。当编译程序识别出块的开始时，需要执行定位操作，当遇到块的结束时，则需要执行重定位操作。

* + 1. **所用数据结构说明**

本编译程序使用栈式散列符号表。

* 在符号表模块中存在一个散列表，声明如下：

HASH\_TYPE hash\_table[HASH\_SIZE];

* 然后声明一个栈式符号表，栈的声明及其结构如下：

/\* 函数参数结构体 \*/

typedef struct \_parameter\_info

{

HASH\_TYPE \*hash\_table; // 参数的散列表

struct \_symbol \*para\_stack; // 参数的栈式散列符号表

unsigned int para\_top;

} parameter\_info;

/\* 符号元素结构体 \*/

typedef struct \_symbol

{

const char\* name; // 符号名

symbol\_type type; // 符号类型

symbol\_type sub\_type; // 子类型

unsigned int dimension; // 维数

int is\_const; // 是否常量

int is\_reference; // 是否引用

parameter\_info parameter\_list; // 函数参数

unsigned int next; // 链域

} symbol;

/\* 符号表 \*/

symbol symbol\_stack[STACK\_SIZE];

/\* 栈顶指针 \*/

unsigned int ss\_top = 0;

* 最后声明一个栈式结构的块索引表，用于保存每个块的边界。声明如下：

/\* 块索引表 \*/

unsigned int table\_of\_block[BLOCK\_SIZE];

/\* 栈顶指针 \*/

unsigned int tob\_top = 0;

* + 1. **算法描述**
* **散列函数算法：**取每个字符串的首字母的ASCII值作为散列值
* **插入算法：**将符号元素复制到栈顶

symbol\_stack[++ss\_top] = symbol;

* **检索算法：**计算符号的散列值，通过查询散列表找到符号在栈中的位置；如果该位置不是此符号，则顺着链域继续往下寻找，直到找到为止。

for (unsigned int pos = hash\_table[hash(name)]; pos;

pos = symbol\_stack[pos].next)

if(!strcmp(symbol\_stack[pos].name, name))

break;

* **定位算法：**将当前符号表栈顶指针复制到块索引表的栈顶

table\_of\_block[++tob\_top] = ss\_top+1;

* **重定位算法：**将当前符号表栈顶指针设置为块索引表的栈顶值

ss\_top = table\_of\_block[tob\_top--]-1;

1. **程序清单**

* lex.l：词法分析器模块文件。
* parser.y：语法分析器模块文件，包括翻译方案（即语义分析和代码生成部分的代码）。
* parsererror.c/parsererror.h：错误处理模块文件。
* code.h：代码生成模块辅助文件。
* symtable.c/symtable.h：符号表模块文件。

1. **测试报告**
   1. **测试环境**

* Microsoft Windows 10 操作系统
* Microsoft PowerShell / CMD
  1. **测试计划**
* 单元模块内的一个功能开发完成后，构造相应的测试用例，进行功能测试；
* 单元模块开发完成后，编写一些Pascal程序作为测试用例，对单元模块进行单元测试，记录测试结果，并进行分析；
* 所有单元测试通过后，将编译器进行整合，进行汇总测试。

在测试过程中，我们会编写正确的和错误的Pascal程序，分别对编译器进行测试，以检查编译器是否能正确生成代码和检测错误。

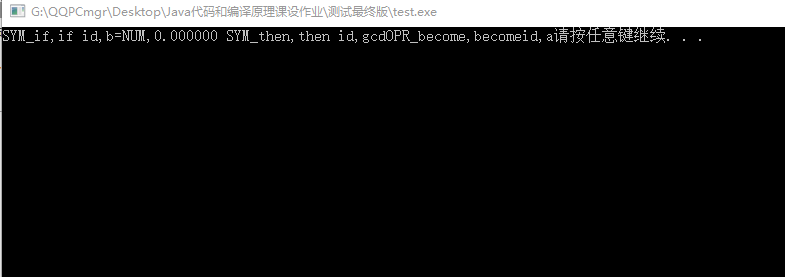
* 1. **词法分析测试报告**
     1. **测试一**
* 测试用例

if b=0 then gcd:=a

* 预期结果

逐个识别出if，b等字符的记号和属性。字符If的记号为“if”,属性为SYM\_if; 字符b的记号为“b”,属性为id。字符0的记号为“0.000000”属性为‘num’等等。

* 测试结果与分析

测试结果符合测试预计。

* + 1. **测试二**
* 测试用例

program program\_sum(input,output);

var sum, i: integer;

begin

sum:=0

for i:=1 to 100 do

sum:=sum+I;

write\_int(sum);

end.

* 预期结果

见下图。

* 测试结果与分析



与预测结果相符。

* 1. **语法分析测试报告**
     1. **测试一**
* 测试用例：快速排序

该测试用例包含了常量、变量、过程/函数声明，数组结构，赋值，关系表达式，运算式，判断结构，if结构，for/while结构，过程/函数调用等语法，适合全面测试语法器的性能。

program sort (input, output);

const c = 9;

var x: integer;

a: array[0..10] of integer;

procedure readarray;

var s: integer;

begin

for s:=1 to c do read(a[s])

end;

function partition (y,z: integer): integer;

var i,j: integer;

pivot: integer;

begin

pivot:=a[y];

i:=y;

j:=z;

while i<j do

begin

while (i<j) and (a[j] >= pivot) do j:=j-1;

a[i]:=a[j];

while (i<j) and (a[i] <= pivot) do i:=i+1;

a[j]:=a[i];

end;

a[i]:=pivot;

partition:=i;

end;

procedure quicksort (m,n:integer);

var k: integer;

begin

if (m < n) then

begin

k:=partition(m, n);

quicksort(m, k-1);

quicksort(k+1, n)

end;

end;

begin

readarray;

quicksort(1, 9);

for x:=1 to c do

begin

write(a[x]);

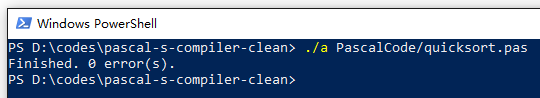
end;

end.

* 预期结果

没有语法错误。

* 测试结果与分析



符合预期结果。

* + 1. **测试二**
* 测试用例：错误的GCD程序

program abc(input,output);

const a = function; c = 00a; z = 15;

var x,y:integer m,n:const; c,d: array[5..10 of boolean; c: integer;

function abcd(a,b:integer):integer;

begin

gcd:=a;

else gcd:=gcd(b, a mod b)

end;

begin

read(x, y);

a:=b\*;

b:=666+++++;

c:=a/

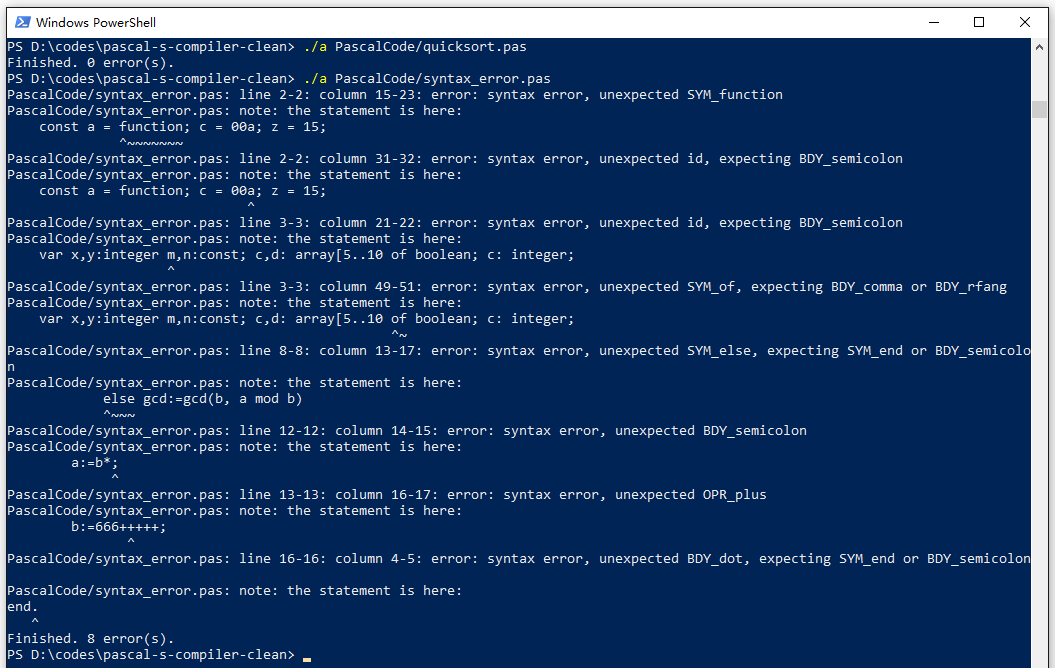
write(gcd(x, y666))

end.

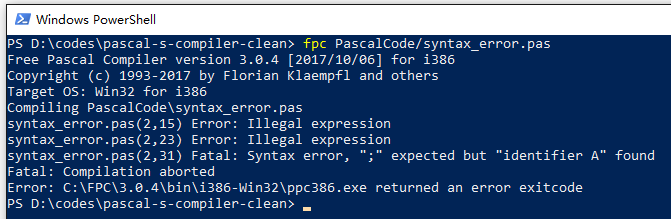
* 预期结果

9个语法错误，已在上述测试用例中以红色标出。

* 测试结果与分析



编译器找到了前8个语法错误并终止了分析。使用FreePascal 官方编译器编译同样的文件进行对比：



FreePascal Compiler只找到了3个错误并终止了分析。说明本编译器有着优秀的错误恢复能力。

* 1. **语义分析测试报告**
     1. **测试一**
* 测试用例：错误的快速排序

program sort (input, output);

var x: integer;

a: array[0..10] of integer;

procedure readarray;

var a,b:char;

c: array[0..3] of integer;

begin

for a:=b to 9 do read\_int(c[b,a]) //预计在a处出现类型错误，b处出现类型错误，在c处出现维数不匹配错误

end;

function partition (y,z: integer): integer;

var i,j: integer;

pivot: integer;

d:char;

begin

pivot:=c[3]; //出现未定义错误

i :=d; //出现类型不匹配错误

pivot:=a[y];

i:=y+3;

j:=x+d; //出现“+”前后不匹配错误

while i<j do

begin

while (d) do j:=j-1; //while中不能出现char

a[i]:=a[j];

while (i<j) and (a[i] <= pivot) do i:=i+1;

a[j]:=a[i];

end;

a[i]:=pivot;

partition:=i;

end;

procedure quicksort (m,n:integer);

var k: integer;

begin

if (m) then

begin

k:=partition(m, n,k); //partition维数错误

quicksort(m, partition(m,n));

quicksort(partition(m), n)//parttion维数错误

end;

end;

begin

write\_string('Please input a sequence:');

readarray;

quicksort(1,9);

for x:=1 to 9 do

begin

write\_int(a[x]);

write\_char(' ')

end;

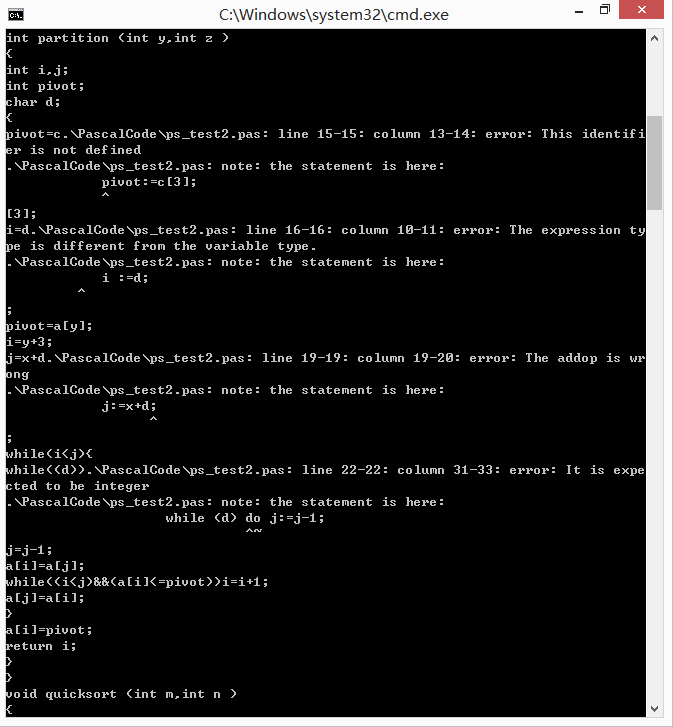
end.

* 预期结果

9个语义错误，已在上述测试用例用红色标出。

* 测试结果与分析





分析器发现了9个错误，符合实验预期。

* + 1. **测试二**
* 测试用例：正确的快速排序

program sort (input, output);

var x: integer;

a: array[0..10] of integer;

procedure readarray;

var s: integer;b:char;

c: array[0..3] of integer;

begin

for s:=1 to 9 do read\_int(a[x])

end;

function partition (y,z: integer): integer;

var i,j: integer;

pivot: integer;

begin

pivot:=a[y];

i:=y;

j:=z;

while i<j do

begin

while (i>y) and (a[j] >= pivot) do j:=j-1;

a[i]:=a[j];

while (i<j) and (a[i] <= pivot) do i:=i+1;

a[j]:=a[i];

end;

a[i]:=pivot;

partition:=i;

end;

procedure quicksort (m,n:integer);

var k: integer;

begin

if (m < n) then

begin

k:=partition(m, n);

quicksort(m, k-1);

quicksort(k+1, n)

end;

end;

begin

write\_string('Please input a sequence:');

readarray;

quicksort(1,9);

for x:=1 to 9 do

begin

write\_int(a[x]);

write\_char(' ')

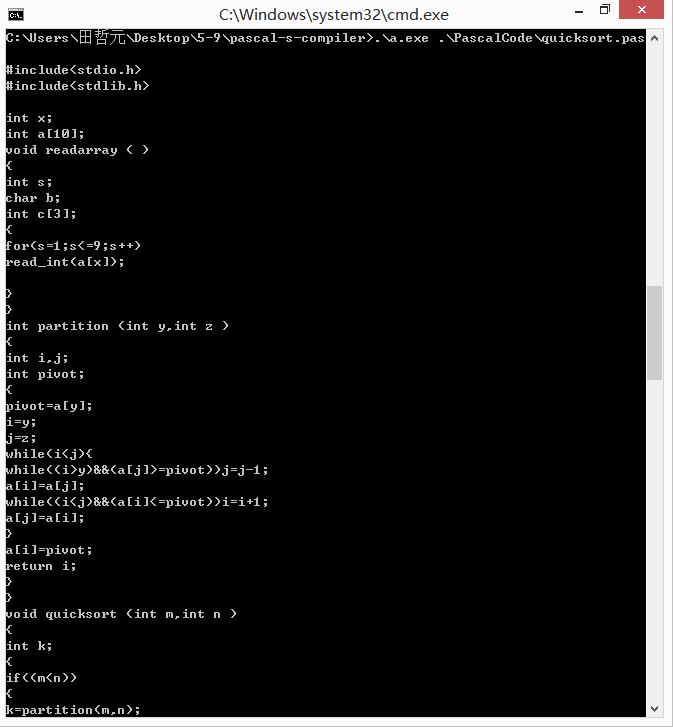
end;

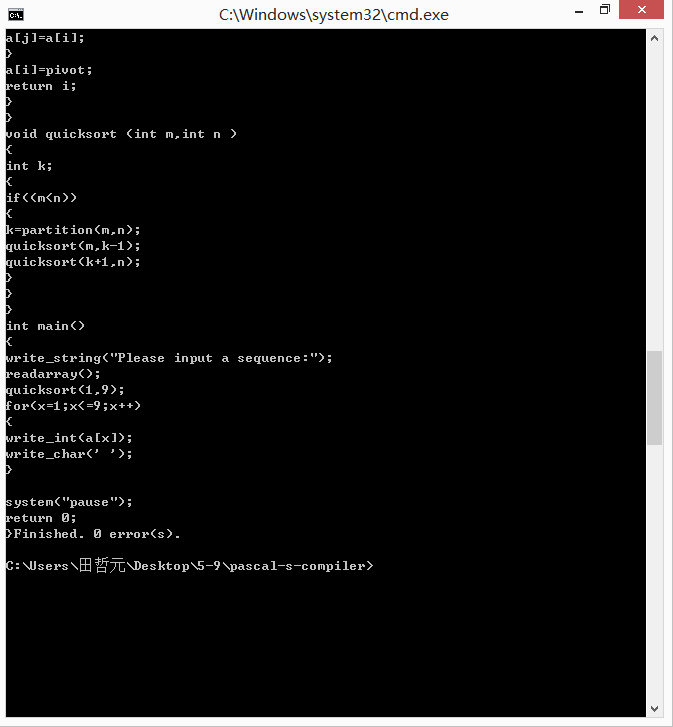
end.

* 预期结果

通过语法和语义检查，没有编译错误。

* 测试结果与分析





测试结果符合预期。

* + 1. **测试三**
* 测试用例：正确的求最大公约数代码

program gcd\_program(input,output);

var x,y:integer;

function gcd(a,b:integer):integer;

begin

if b=0 then gcd:=a

else gcd:=gcd(b, a mod b)

end;

begin

write\_string('Input a, b:');

read\_int(x);

read\_int(y);

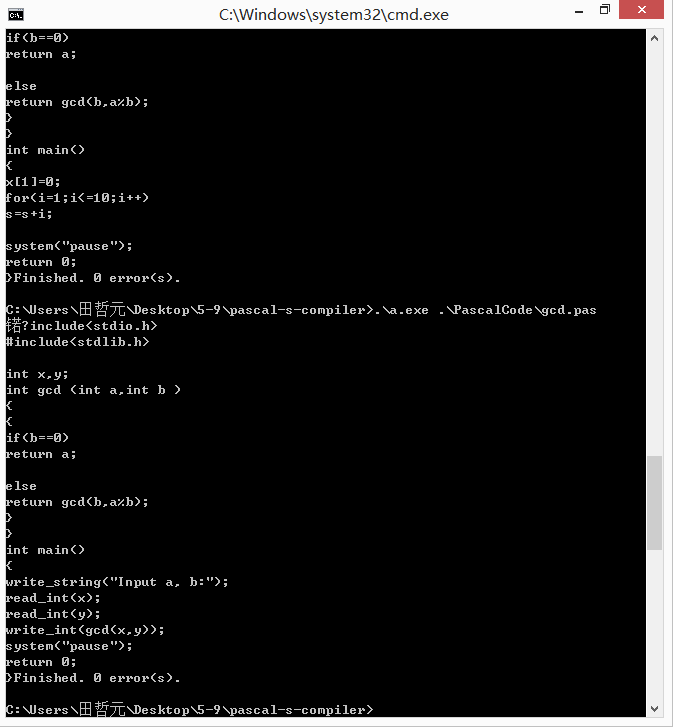
write\_int(gcd(x, y));

end.

* 预期结果

通过语法和语义分析，没有编译错误。

* 测试结果与分析



* 1. **代码生成测试报告**
     1. **测试一**
* 测试用例：验证变量声明与赋值、程序基本结构及简单的功能调用

program HelloWorld(xxx);

var x:char;y:array [0..100,0..100] of integer;

begin

x := 'a';

write\_char(x);

write\_string('Hello World!');

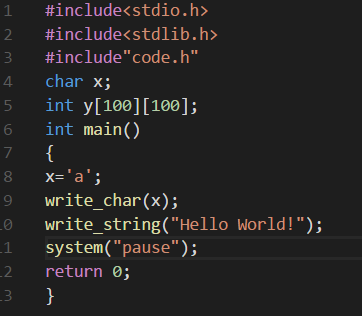
end.

* 预期结果

成功生成目标代码，没有编译错误。

运行时输出 a 和 Hello World!

* 测试结果与分析



C:\Users\18801\AppData\Roaming\Tencent\Users\1417472234\QQ\WinTemp\RichOle\$55AOPMH_0XO)RU0RJ0UW_F.png

运行结果符合预期。

* + 1. **测试二**
* 测试用例：验证运算的正确性以及for结构的使用

program program\_sum(input,output);

var sum,i:integer;

begin

sum:=0;

for i:=1 to 100 do

sum:=sum+i;

write\_int(sum);

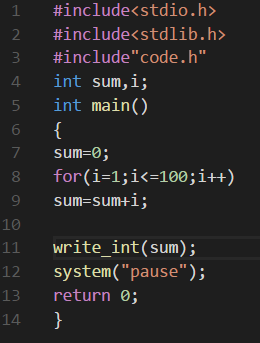
end.

* 预期结果

成功生成目标代码，没有编译错误。

运行时输出5050。

* 测试结果与分析



C:\Users\18801\AppData\Roaming\Tencent\Users\1417472234\QQ\WinTemp\RichOle\GCF[5}J6`W$$%J~78}WP6UQ.png

运行结果符合预期。

* + 1. **测试三**
* 测试用例：验证函数调用以及expression的正确性

program gcd\_program(input,output);

var x,y:integer;

function gcd(a,b:integer):integer;

begin

if b=0 then gcd:=a

else gcd:=gcd(b, a mod b)

end;

begin

write\_string('Input a, b:');

read\_int(x);

read\_int(y);

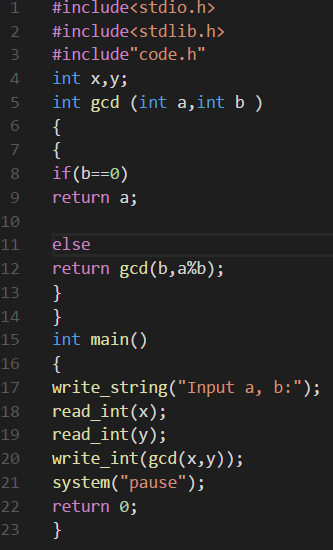
write\_int(gcd(x, y));

end.

* 预期结果

成功生成目标代码，运行目标代码能求出两个数的最大公约数。

* 测试结果与分析



![D:\docs\QQ用户数据\573038351\Image\C2C\Image1\]A~3I94JUVMO_834](UA[LB.png](data:image/png;base64,)

运行结果符合预期，3和9的最大公约数是3，8和9的最大公约数是1。

* + 1. **测试四**
* 测试用例：验证if结构使用

program which\_day(input,output);

var wd:integer;

weekday:string;

begin

write\_string('Which day is today in a week?');

read\_int(wd);

if wd = 0 then

weekday := 'Sunday'

else if wd = 1 then

weekday := 'Monday'

else if wd = 2 then

weekday := 'Tuesday'

else if wd = 3 then

weekday := 'Wednesday'

else if wd = 4 then

weekday := 'Thursday'

else if wd = 5 then

weekday := 'Friday'

else if wd = 6 then

weekday := 'Saturday'

else if wd = 0 then

weekday := 'Sunday'

else weekday := 'WROOOOOOOONG!';

write\_string('Today is ');

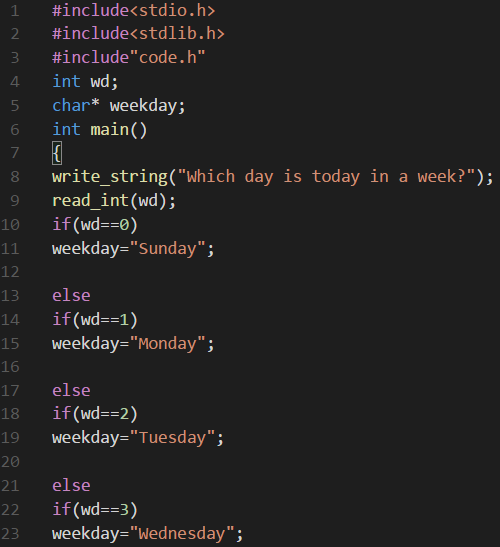
write\_string(weekday);

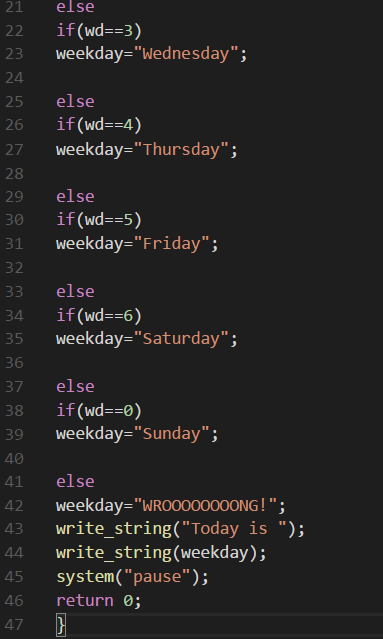
end.

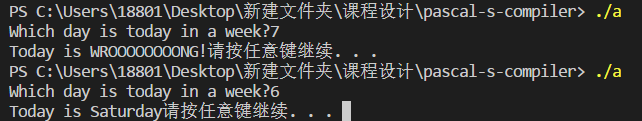
* 预期结果

成功生成目标代码，并能根据输入的数字推测出今天是星期几。

* 测试结果与分析







运行结果符合预期。

* + 1. **测试五**
* 测试用例：快速排序

program sort (input, output);

var x: integer;

a: array[0..10] of integer;

procedure readarray;

var s: integer;

begin

for s:=1 to 9 do read\_int(a[s])

end;

function partition (y,z: integer): integer;

var i,j: integer;

pivot: integer;

begin

pivot:=a[y];

i:=y;

j:=z;

while i<j do

begin

while (i<j) and (a[j] >= pivot) do j:=j-1;

a[i]:=a[j];

while (i<j) and (a[i] <= pivot) do i:=i+1;

a[j]:=a[i];

end;

a[i]:=pivot;

partition:=i;

end;

procedure quicksort (m,n:integer);

var k: integer;

begin

if (m < n) then

begin

k:=partition(m, n);

quicksort(m, k-1);

quicksort(k+1, n)

end;

end;

begin

write\_string('Please input a sequence:');

readarray;

quicksort(1, 9);

for x:=1 to 9 do

begin

write\_int(a[x]);

write\_char(' ')

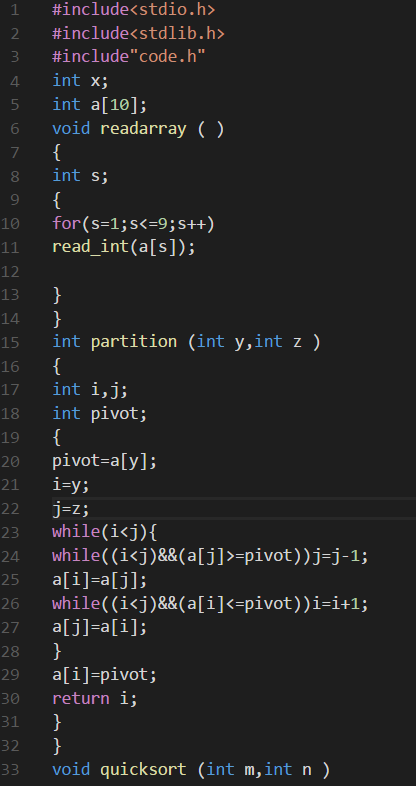
end;

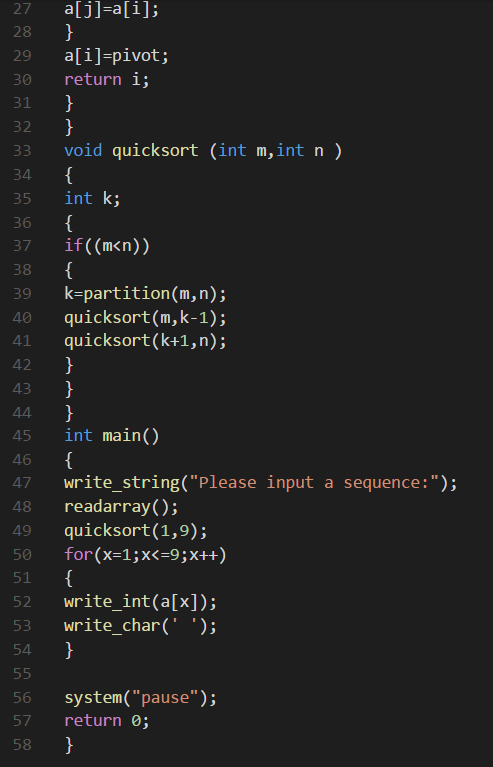
end.

* 预期结果

成功生成目标代码，并能通过快速排序算法从小到大排序输入的9个数字。

* 测试结果与分析





C:\Users\18801\AppData\Roaming\Tencent\Users\1417472234\QQ\WinTemp\RichOle\([SIZO%ZD2`DGN}$XL$QY7K.png

运行结果符合预期。数字序列 0 0 25 49 86 1003 9 30 8 被排序成 0 0 8 9 25 30 49 86 1003。

* 1. **汇总测试**

当所有的模块单元都开发完毕后，就可以组装成一个完整的编译器了，组装完毕后需要进行汇总测试。

对于正确的Pascal-S语言代码，汇总测试的结果与代码生成的测试结果一致，在此不在赘述。我们设计一个同时含有语法错误和语义错误的Pascal-S程序用于测试编译器的错误检查性能。

* 测试用例：同时含有语法错误和语义错误的程序

program abc(input,output);

const a = function; c = 00a; z = 15;

var x,y:integer m,n:const; c,d: array[5..10 of boolean; f: integer;

function abcd(a,b:integer):integer;

begin

gcd:=a;

else gcd:=gcd(b, a mod b)

end;

begin

read(x, y);

a:=b\*;

b:=666+++++;

c:=a/

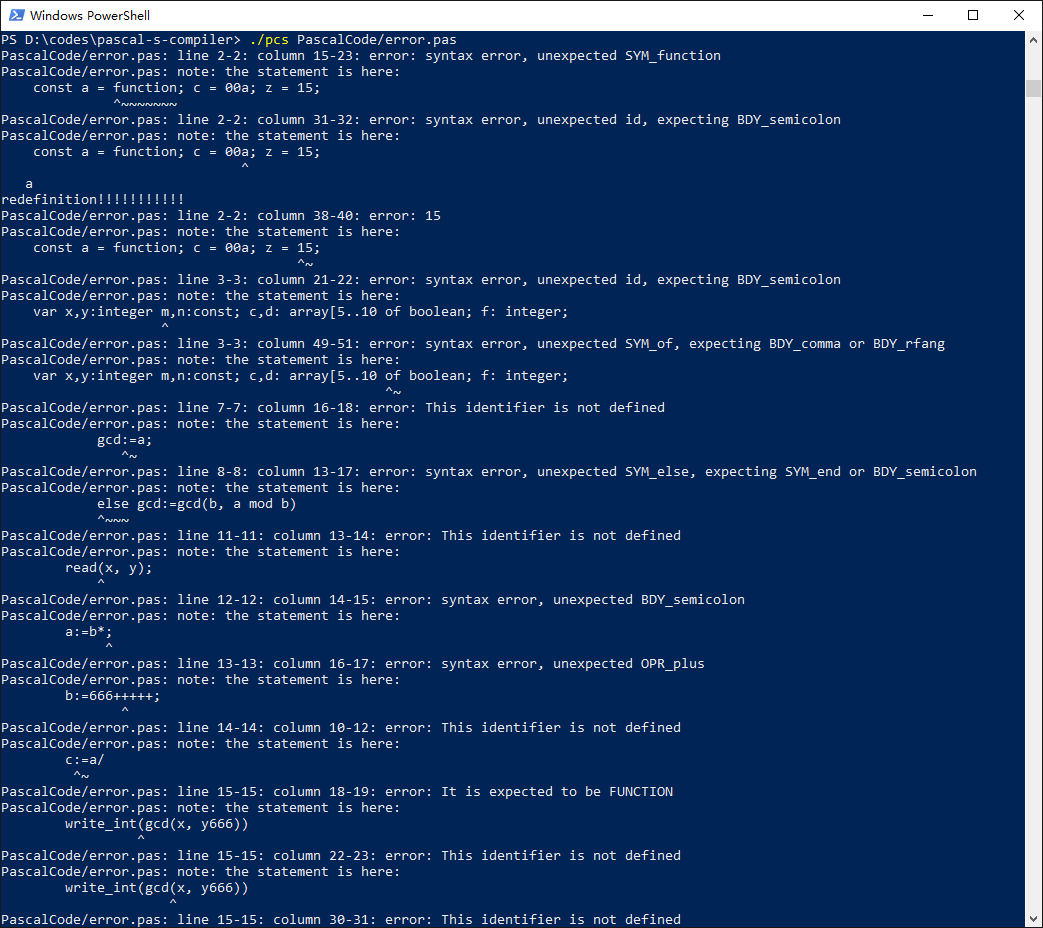
write\_int(gcd(x, y666))

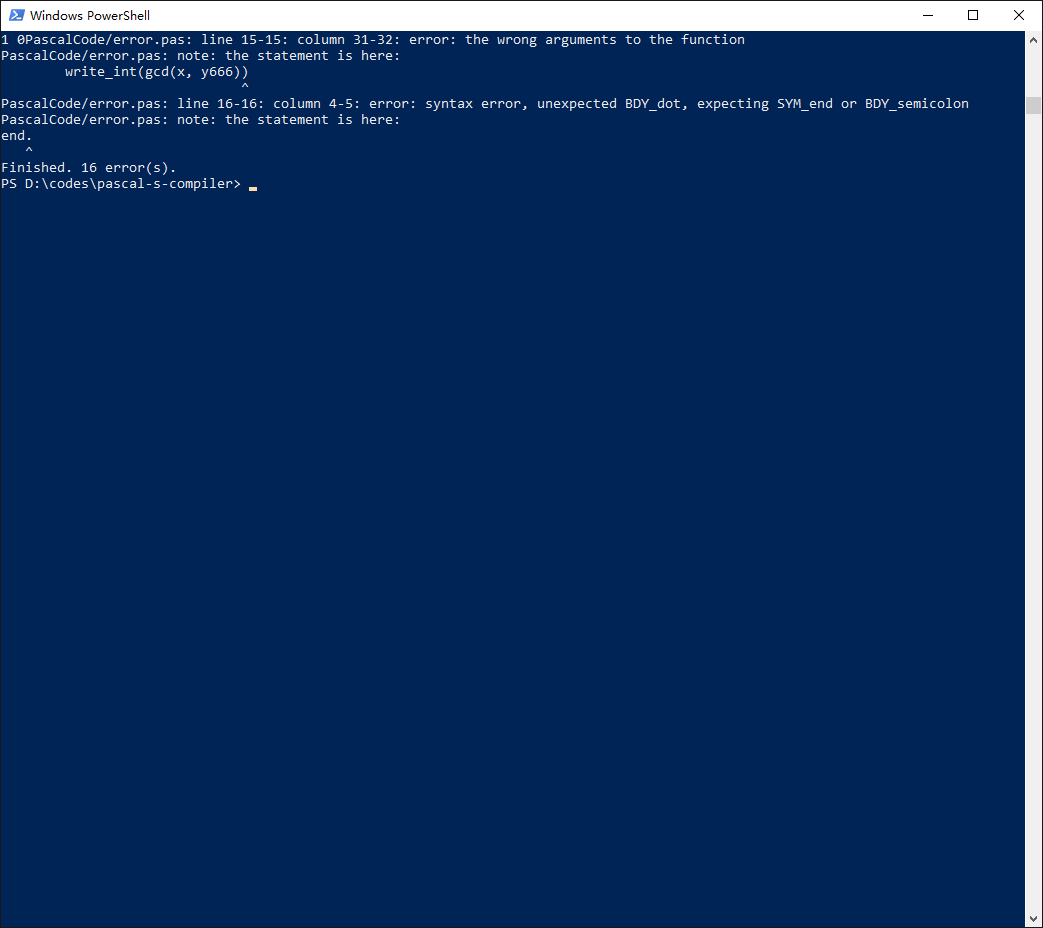
end.

* 预期结果

同时找到语法错误和语义错误

* 测试结果与分析





结果符合预期。编译器不仅找到了语法错误，同时还找到了语义错误，包括write\_int是没有返回值的过程不能用作除数，gcd、y666变量未定义等。

1. **实验总结**

通过半个学期的编译原理与技术课程设计，我们团队共同完成了一个运行优良的Pascal-S编译器，同学们也有着不同的收获和总结。

* 1. **词法分析部分**

徐永杰：经过本次编译原理的课程设计，我对整个如何编写一个简单Pascal语言的编译器的过程有了充分的了解，在过程中也学习了很多知识。同时对上个学期学习的编译原理的整体知识架构有了更进一步的加深学习。自己负责词法分析较多，所以对如何用lex工具来编写词法分析器也有了充分的掌握。

* 1. **语法分析部分**

甘钊宇：在语法分析器的编写过程中，通过实际的测试，自己对自底向上的LR分析方法有了更形象、更深入的理解，同时在研究与补充Pascal语言文法的过程中，也学会了Pascal语言。此外，语法分析器是一遍编译的编译器中的核心模块，它通过调用编译器的其它模块的接口完成源代码到目标代码的转换，这也使自己对编译器的整体结构有了更清楚的理解。

在开发语法分析器的过程中，我们遇到了如何使语法器在遇到错误时能忽略错误成分并恢复分析的问题。通过与周围同学交流与查阅GNU Bison再加上不断实验，最终我们找到了良好的解决方案，增强了编译器的功能，也收获了解决问题的方法。

* 1. **语义分析部分**

黄勇康：经过这次编译器的实验，我受益颇多。我和田哲元负责语义分析，我负责符号表的建立，田哲元负责调用，但甘钊宇已经将符号表的接口做好了，我只需要把符号存进去就好，任务不是特别重，但由于之前编译原理学的不怎么好，对移进规约过程不是很熟悉，所以语义动作的执行顺序也不清楚，于是花了很长的时间来学习了解移进规约过程，才开始写代码。知道了语义动作的执行顺序后，有点小膨胀，一股脑地把自己想好的东西全写上去了，导致根本无法调试，很是难受。之后开始将想好的语义动作一点一点地加到语法分析代码上，挺顺畅的，加了很多全局变量，刚弄完就傻傻地以为自己真的建好了符号表，当开始符号表调用后，bug渐渐显露，又是很长时间的debug，经过一段时间的努力后，语义分析基本完成，在我们俩这，测试都通过了，但是将代码发出来后，不到半小时，又是疯狂报错，于是又开始了苦逼的debug生活，原来是定位和重定位操作出现了问题，还是对语法分析了解不够全面，直到最后一周，才把已知bug调完，bug肯定还存在，暂时未找出。输入输出函数还存在一些问题，我们直接将这类函数在最开始就添加进符号表中，调用时就直接是调用符号表中的已定义函数了。通过这次实验，我更深刻地理解了团队合作的意义，虽然有问题，但可以通过和别人沟通来找到解决方案，事半功倍。这次实验，我们用到了软件工程的知识，通过需求分析，概要设计，详细设计，编码，调试这些过程，程序实现变得简单，有条理。

田哲元：此次编译原理实验我负责的是语句的类型检查部分。语句检查可以分为2个部分，一部分是语句归约中的类型检查，一部分是expression归约中的类型获取。下面我总结了自己在此次编译中遇到的问题，以及此次实验对自己的提升。

从实验开始我便遇到了编译问题，同样的文件在自己机器上不能运行，然后开始逐个排查，发现是gcc编译器的版本过低导致了编译不能通过。其次便是归约中如何进行类型的存储及释放，以及如何在对应翻译方案中如何进行对应的动作。在此次试验中，我采取的是使用数组来充当分析栈的角色，在分析栈中，每步归约都会有对应的动作以确保当前的分析栈顶元素是expression的类型，此次试验中，分析栈的难点在于函数调用中嵌套函数或者数组这种express\_list类型的类型保存，为解决这个问题，我使用了一个char\*数组来存储当前及之前嵌套的函数的名称，以确保能够在函数调用中参数为数组或函数时保护现场，不会丢失正在进行的类型检查。最后是对于实验测试示例的设计，要确保尽量能够涉及到每个翻译方案中的类型错误。

在此次试验中，通过对于分析栈的归约分析，对于语句以及expression的归约过程有了更加深刻的见解，了解了语义分析中的重要性，在每一步语义分析中要分析当前能够归约至此的每种情况，以保证在该种情况下，分析栈的归约不会失去在此之前的步骤。

* 1. **代码生成部分**

贾东港：实验中遇到或存在的主要问题：

（1）函数定义部分

由于pascal的函数定义部分的返回值是定义在函数头的最后部分，而且每一个参数的定义先定义名字再定义类型，这就给代码生成带来了很大困难。经过思考，打算采取先保存下来函数定义的参数，保存下来每个参数的姓名、类型、以及按什么方式调用。思考如何保存函数的参数类型是遇到的一个难点，最后的解决方案是通过一个链表来保存函数的参数。

（2）同样的问题也存在于变量、常量声明中，Pascal的这些声明也是先声明名字，再声明类型，所以采取的也是先保存这些变量常量的名字，然后在获取类型之后统一输出。

（3）函数返回部分：pascal的函数的返回值不靠return进行，这就给判断什么时候返回值造成了一定的困难，最后通过一些标志变量以及一些逻辑判断成功的判断了什么时候输出return。

体会/收获：加深了对编译原理的理解，以及对flex和yacc的运行过程更加了解了。加深了对pascal和c语言的理解，对pascal语言的特性和c语言的特性了解更加深刻了。体会到了不同的语言的差别还是蛮大的，不同语言之间的转换不是一个容易的工作。

田国梁：一个可行的实施方案，必然是众多方案中的最优方案。然而，最优方案并不全是技术最先进、实现方法最高效或者算法复杂度最低的那种方案。而是基于团队成员的执行能力、项目的时间节点以及项目的潜在风险等一系列影响项目成功因素，并以实现项目指标为目的的最优最可靠方案。

就如同本次编译原理课程设计中的代码实现一样，实现目的的方法可能是多种多样的，但其中必然有算法复杂度最低的那一种。此时，就必须要全面分析任务的目标和团队成员的具体情况，结合时间节点与实际执行能力，制定出一套最可靠的实施方案。然后在合理分工的基础上，共同协作完成这项任务。

由此可见，设计与实施方案必须要以实现项目目标为准则。合乎团队的具体情况，才能对项目的有效开展提供稳妥的保证。

对于团队中的个人来讲，必须要不断提高自己的能力，要努力适应并按计划完成团队中属于自己的子任务，不能没有责任感，没有大局观。团队中个人对任务的执行能力，直接影响着项目的整体推进。就如木桶效应一般，不能因为自己的失误，进而拖了整个项目团队的后腿。

另外，各个成员之间要有足够的默契程度，要清楚成员之间的工作习惯，各个成员的优缺点，并且队友之间要相互体谅，互相帮助，这样才能很好的完成项目的既定任务。

最后，我想说明的是，每一个项目都有其的共通性与独特性。我们要善于总结各个项目中的成功点与失败点，并将成功点不断优化，失败点不断改正，才能使团队能够更好的成长。