

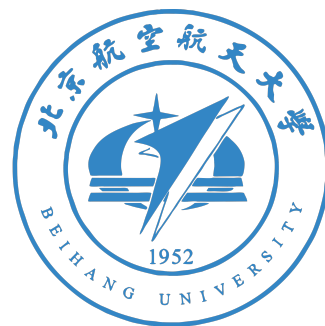


编译器课程设计实验参考（2019）

第一部分

时间：September 13, 2019

版本：0.6



Go!Go!Go!

目 录

序章	1
0.1 评测环境说明	1
0.2 评测时输入输出约定	1
0.3 文法的一些约定	1
0.4 一些建议	3
1 文法解读	4
1.1 文法解读的一些样例	4
1.2 提交测试程序	5
2 错误处理	6
2.1 错误处理的要求	6
2.2 评测时输出约定	6
3 词法分析	8
3.1 词法分析阶段需要完成的一些功能	8
3.2 一种建议的词法分析实现方式	9
3.3 评测时词法分析输出约定	9
4 语法分析	13
4.1 语法分析阶段需要完成的一些功能	13
4.2 评测时语法分析输出约定	16
5 语义分析	21
5.1 语义分析与语法分析的区别	21
5.2 语义分析阶段需要完成的一些功能	22

序章

0.1 评测环境说明

教学平台的评测环境使用 Clang8.0.0 编译环境，请自行测试在此环境下能否成功编译和运行。鉴于不同的编译环境下运行结果可能存在较大的差异，可在代码实现过程中在平台上多提交几次，观察能否在测试环境下成功编译，防止临近 DDL 才发现存在问题。

0.2 评测时输入输出约定

编译器应从 `testfile.txt` 中读取测试代码文件。在词法分析、语法分析与生成中间代码阶段，输出至 `output.txt` 文件中。在生成目标代码阶段，选择生成 MIPS 代码的同学将编译器生成的 MIPS 代码输出至 `mips.txt` 文件中；选择生成 PCODE 的同学将生成的 PCODE 输出至 `pcode.txt` 文件后用解释执行程序运行生成的 PCODE 代码通过命令窗口进行输入输出最终结果。在考察错误处理时，输出至 `error.txt` 中。

在课程设计生成目标代码阶段以及最终提交的文件中，仅输出目标代码生成阶段要求的内容即可，而之前在词法分析、语法分析和生成中间代码阶段要求的输出都可不必输出了。

0.3 文法的一些约定

如下是对于课程设计要求文法的一些补充约定。

1. 扩充 C0 文法中，不用考虑参数是整个数组的情况，不能传数组名，可以是一个数组元素作为实参传进去。
2. 数组的下标从 0 开始。
3. 无返回值的函数要么没有 `return` 语句，要么只有 `"return;"`，`"return (0)"` 是不可以出现在无返回值函数中的。
4. `scanf` 的参数在语法形式上是只写标识符，但处理时的语义跟 C 语言 `scanf` 的语义类似。其他函数的运算结果只能通过返回值传递。

5. 函数参数调用时实参的计算顺序的问题。对于如下样例：

```
int global;
int add1(){
    global = global + 1;
    return (global);
}
int add2(){
    global = global + 2;
    return (global);
}
int sum(int a, int b){
    printf("a = ", a);
    printf("b = ", b);
    return (a + b);
}
void main(){
    global = 1;
    printf(sum(add1(), add2()));
}
```

add1 和 add2 两个函数的执行顺序不同将会影响最终的结果，而不同的编译器的求值顺序可能存在差异，运行结果也会不同。最终要求生成的目标码运行结果与 Clang8.0.0 编译器运行的结果一致即可。


6. 标识符、保留字区分大小写。
7. 赋值语句左侧的标识符不能为常量。
8. char 类型的变量或常量，用字符的 ASCII 码对应的整数参加运算。
9. printf(字符串, 表达式)，应该按照顺序输出，先输出字符串内容，再输出表达式的内容
10. 写语句中，字符串原样输出（不存在转义），单个字符类型的变量或常量、字符类型数组变量、字符类型的函数调用输出字符，其他各种情况的表达式都按整型输出。
11. 变量没有初始化的情况下没有初值。引用没有初始化的变量可以报错，考核的测试程序都会进行初始化，不会考核这种情况。


0.4 一些建议


如之前一位学长所说的：“对于编译技术课程设计，本身课程内容上，实际并不困难，仅仅只是需要时间，来应对整体工程所需的复杂性，即需要时间调试、测试。不存在‘做不到’的情况”，编译器课设并不困难。要求的文法是在 C 语言的基础上简化了很多的类 C 文法，这就极大地降低了实现的难度。

建议尽早进行编译课程设计的工作，不要等到 DDL 的时候再来冲刺，防止出现意外情况，影响自己的成绩。

后续各章节内容除输出约定等评测时的要求外，只是笔者之前实现时的一些想法，完全可以不必阅读，自行设计实现。如果在实现中遇到一些问题，可再来看看，若能有所启发，也算是不枉费一番心血了。

 **注意** 强烈建议在开始编译器的开发之前仔细阅读和分析 PL/0 编译器源代码及 Pascal-S 编译器源代码，可以将编译器源代码在 Pascal 集成开发环境（比如 Lazarus）中编译运行，输入测试程序进行跟踪调试，以帮助理解。在阅读现有的编译器的源代码的过程中，能够对编译的各个过程有充分的了解，并能够在一开始就设计好自己的编译器，以减少之后不必要的改动带来的种种麻烦。

 **注意** 设计永远大于实现，没有做好设计之前千万不要着急去码代码。在正式开始编译器的实现之前，提前做好设计是很有必要的。建议考虑好总体架构和各部分之间的接口之后，再开始码代码。同时，代码中注意添加适当的注释。

 **注意** 在各个阶段都要考虑错误处理的问题，编译器在任何情况下都应能够正常退出，且在测试代码出错的情况下希望能找到尽可能多的错误。

在报告错误时，需包含行、列（或单词序号）等位置信息和错误提示信息，为了便于与后续“错误处理”考核中的要求一致，每类错误请预留一个以字符 a-z, A-Z 标识的类别码。

顺便提一句，鉴于 windows 平台和 linux 平台每行结尾的不同，实现时应当考虑到测试样例中可能存在“\r\n”的情况，建议判断空白符使用 isspace 函数。

第1章 文法解读

在正式开始完成编译器课程设计之前，需要先对文法进行分析。对文法中每条规则所定义的语法成分进行分析，考虑其作用、限定条件，思考语法成分中各种情况的组合。在测试自己编写的编译器时，除了满足评测所需的对每条规则的覆盖、对规则内不同组合情况的覆盖外，还要考虑对各种不同语法成分之间的常见组合进行测试。

1.1 文法解读的一些样例

下面列举一些分析文法作用、限定条件、思考各种语法成分的不同组合的示例。

1.1.1 作用

$\langle \text{关系运算符} \rangle ::= < \mid <= \mid > \mid >= \mid != \mid ==$

如上规则说明了由 $\langle \text{关系运算符} \rangle$ 可以推导出这些终结符。

1.1.2 限定条件

$\langle \text{复合语句} \rangle ::= [\langle \text{常量说明} \rangle] [\langle \text{变量说明} \rangle] \langle \text{语句列} \rangle$

$\langle \text{常量说明} \rangle ::= \text{const } \langle \text{常量定义} \rangle ; \text{const } \langle \text{常量定义} \rangle ;$

$\langle \text{常量定义} \rangle ::= \text{int } \langle \text{标识符} \rangle = \langle \text{整数} \rangle , \langle \text{标识符} \rangle = \langle \text{整数} \rangle \mid \text{char } \langle \text{标识符} \rangle = \langle \text{字符} \rangle , \langle \text{标识符} \rangle = \langle \text{字符} \rangle$
 $\langle \text{无符号整数} \rangle ::= \langle \text{非零数字} \rangle \{ \langle \text{数字} \rangle \}$
 $\mid 0$

$\langle \text{整数} \rangle ::= [+ \mid -] \langle \text{无符号整数} \rangle$

$\langle \text{标识符} \rangle ::= \langle \text{字母} \rangle \{ \langle \text{字母} \rangle \mid \langle \text{数字} \rangle \}$

$\langle \text{变量说明} \rangle ::= \langle \text{变量定义} \rangle ; \langle \text{变量定义} \rangle ;$

$\langle \text{变量定义} \rangle ::= \langle \text{类型标识符} \rangle (\langle \text{标识符} \rangle \mid \langle \text{标识符} \rangle ' [\langle \text{无符号整数} \rangle ']) , (\langle \text{标识符} \rangle \mid \langle \text{标识符} \rangle ' [\langle \text{无符号整数} \rangle '])$

根据以上文法规则可知，函数定义的复合语句中，常量和变量的声明顺序已经被限定好了，不能随便更改声明顺序，如“`int b, c[10]; const int a = 6;`”此种顺序是不允许的。

1.1.3 各种语法成分的不同组合

$\langle \text{程序} \rangle ::= [\langle \text{常量说明} \rangle] [\langle \text{变量说明} \rangle] \langle \text{有返回值函数定义} \rangle \mid \langle \text{无返回值函数定义} \rangle \langle \text{主函数} \rangle$

如上这条规则说明了程序可以存在常量说明、变量说明，也可以不存在。那么就意味着在设计编译器时就需要考虑这两种成分存在与否对编译器的影响了。特别是"int"、"char"都同时属于 $\langle \text{变量说明} \rangle$ 和 $\langle \text{有返回值函数定义} \rangle$ 这两种语法成分的 FIRST 集，就更需要对类似这种情况进行测试了。

$\langle \text{条件语句} \rangle ::= \text{if '}' \langle \text{条件} \rangle \text{'}' } \langle \text{语句} \rangle [\text{else } \langle \text{语句} \rangle]$

同样，如上这条规则说明了条件语句可能存在 else 分支也可能不存在，对这两种情况也需要进行测试。

这里只是列举了一些例子，其他情况还请自己仔细思考。评测时主要根据对每条规则的覆盖、对规则内不同组合情况的覆盖进行打分，但建议同学们自己考察各种语法成分的不同组合覆盖情况。

1.2 提交测试程序

为保证最后实现的编译器的正确性，尽可能多的设计各种情况测试程序是很有必要的。而我们课程设计的第一个任务也正是阅读和分析文法并提交测试程序。评测时通过检查提交的测试程序的语法覆盖情况来给出测试程序部分的成绩。需要注意的是，提交的测试程序需要符合文法语法和语义的要求。



注意 建议对每一条规则都进行详细解读，在此基础上再开始编写测试程序。建议同学们每个测试程序都尽可能多地覆盖文法的所有规则和各种情况，如果一个测试程序未能覆盖全，可以编写多个测试程序进行覆盖。如果提交的任何一个测试程序有任何语法或语义错误，都不能得到分数。

$\langle \text{字符} \rangle$ 中的单引号是为了界定字符，源程序中单引号会出现，如 `char aa='a'`，跟界定字符串的双引号类似。而其他出现在 $()[]\{\}$ 这几个符号左右的单引号，是表示该处需要一个作为终结符号的 $($ 或 $[$ 或 $\{$ 或 $)$ 或 $]$ 或 $\}$ ，是为了跟扩充的 BNF 中的 $()[]\{\}$ 这几个符号相区分的，因为文法的定义中用这几个扩充的 BNF 范式的元符号。比如 $\langle \text{主函数} \rangle ::= \text{void main'(')'} \{ \langle \text{复合语句} \rangle \}$ 表示符合该规则的句子应该形如 `void main() { ... }`

第 2 章 错误处理

编译程序对于语法和语义上正确的源程序要能正确地进行编译，生成等价的目标程序。而对于有错误的源程序，不能一发现错误就停止编译，而是要对错误做适当的处理，从而使编译工作能继续往下进行。也就是说，编译程序必须具有能发现错误和对错误进行处理的能力。

2.1 错误处理的要求

从编译程序的角度，一般将源程序中的错误分为语法错误和语义错误，相关的概念见课本中的描述。

发现第一个错误后要继续分析，不能马上退出，需要用错误局部化处理的方法进行处理。发现错误就不用生成中间代码和目标代码。为了便于进行自动评测，考核时会指定特定类型的错误作为考核范围，其他错误可自行设计，合理即可。

2.2 评测时输出约定

请提交能够处理指定错误类型的编译器源代码，要求将测试代码中的错误信息输出到命名为"error.txt"的结果文件中。结果文件中每行按如下方式组织(中间仅用一个空格间隔)：

错误所在的行号 错误的类别码

表 2.1: 错误的类别码

错误类型	类别码
不符合语法规则	a
重定义	b
未定义符号	c
参数个数不匹配	d
类型不匹配	e
缺少分号	f
缺少右括号']'	g
缺少右中括号']'	h

行号由 1 开始计数（评测时样例输出的行号一定是明确的，不必过多的考虑某些存在争议的情况）。错误的类别码请统一按表 2.1 形式定义。自动评测时仅会考核作业要求中列出的错误情况（以作业题目为准），一个文件中会存在 1-3 个错误。



注意 需要说明的是，这里对于输出格式的规定，仅是为了方便评测，而不代表在实际的编译器中也必须是按照这样的输出格式来设计编译器的。同样，这里约定的这些错误情况仅是为了方便评测，实际的编译器中还应考虑更多的错误情况。

例如对如下样例

```
const int const1 = 1, const2 = -100;
const char const3 = '?';
int change1;
char change3;
int gets1(int var1,int var2){
    change1 = var1 + var2    return (change1);
}
void main(){
    change1 = 10;
    printf("Hello World");
    printf(gets1(10, 20));
}
```

我们可以看出其中第 2 行的字符常量不符合词法规则；第 6 行中缺少了一个分号，不符合语法规则，故 error.txt 中输出应为

```
2 a
6 f
```

第 3 章 词法分析

词法分析程序的功能是扫描源程序字符，按语言的词法规则将输入的源程序分解成一系列的单词符号。对于一门编程语言来说，这些单词包括保留字、标识符、常数、分界符或操作符等。例如，赋值语句 `const char lowera = 'a'` 在词法分析中被分解为以下一系列符号：

- 保留字 `const`
- 保留字 `char`
- 标识符 `lowera`
- 操作符 `=`
- 常数 `a`

其中分隔这些单词符号的空格将被忽略。

3.1 词法分析阶段需要完成的一些功能

3.1.1 剔除空白符

在词法分析阶段，应剔除源代码中的空白符，使语法分析器不必考虑空白符，可以简化语法分析的实现。

3.1.2 常数转换

词法分析应将源代码中的常数转换为对应的值。例如，将 `i=2019` 中的 2019 作为一个单词，并计算出其值为 2019。

令 `num` 是表示整数的单词类别。当一个数字序列出现在输入的源程序中时，词法分析器将把 `num` 传递给语法分析器。并将该整数的值作为单词的属性值传递给语法分析器。如果我们把单词类别和它的属性值用 `<>` 括起来作为一个元组，那么输入 `31+28+59` 就可以写成 `< num, 31 > < + > < num, 28 > < + > < num, 59 >`。

3.1.3 对一些易混淆单词的区分

保留字 (reserved word), 指在高级语言中已经定义过的字, 使用者不能再将这些字作为变量名或过程名使用。许多程序设计语言使用固定的字符串 (如 `const`, `while` 等) 作为某种结构的标识, 而这些字符串通常也满足组成标识符的规则。因此, 在词法分析器中也需要实现对标识符和保留字的区分。

同时, 在词法分析器中还需要实现对具有相同前缀的单字符分界符和双字符分界符的处理, 例如区分 “<” 和 “<=” 等符号。可以通过从输入串中超前的读入一些字符来实现。

3.1.4 考虑错误处理

每个阶段都可能遇到错误。在词法分析阶段检测到错误后, 也需要以适当的方式进行错误处理, 使得编译器能够继续运行。可以在词法分析阶段就设计好一种错误处理的机制。



注意 在任何情况下, 编译程序都应能够正常的退出。

3.2 一种建议的词法分析实现方式

词法分析器介于语法分析器和输入流之间, 并与这两者进行交互。词法分析器从输入串中读取字符并按词法规则识别出各类单词或符号, 然后将单词的类别及其属性值传递给编译器的下一个阶段。在某些情况下, 词法分析器在把单词传给语法分析器前, 需要从输入串中超前地读入一些字符, 以确定需要传递给语法分析器的正确单词。

将词法分析和语法分析安排在同一遍中是一种比较好的实现方式。可以将词法分析编成一个子程序, 该子程序由语法分析器调用, 返回读入的单词的类别和其具有的属性值。关于词法分析的具体实现可以参考 Pascal-S 编译器源代码中的 `insymbol` 过程。

3.3 评测时词法分析输出约定


请提交词法分析程序的源代码, 要求将词法分析的结果输出到命名为 “`output.txt`” 的结果文件中。结果文件中每行按如下方式组织 (中间仅用一个空格间隔):

单词类别码 单词的字符/字符串形式

表 3.1: 单词的类别码

单词名称	类别码	单词名称	类别码	单词名称	类别码
标识符	IDENFR	while	WHILETK	>=	GEQ
整型常量	INTCON	for	FORTK	==	EQL
字符常量	CHARCON	scanf	SCANFTK	!=	NEQ
字符串	STRCON	printf	PRINTFTK	=	ASSIGN
const	CONSTTK	return	RETURNTK	;	SEMICN
int	INTTK	+	PLUS	,	COMMA
char	CHARTK	-	MINU	(LPARENT
void	VOIDTK	*	MULT)	RPARENT
main	MAINTK	/	DIV	[LBRACK
if	IFTK	<	LSS]	RBRACK
else	ELSETK	<=	LEQ	{	LBRACE
do	DOTK	>	GRE	}	RBRACE

单词的类别码请统一按表3.1形式定义。对于字符串和字符常量，输出时其“单词的字符/字符串形式”即为相应的字符串和字符常量值，对于字符串和字符常量两侧的单、双引号都不必输出，可参考下面给出的样例。

 **注意** 需要说明的是，这里对于输出格式的规定，仅是为了方便评测，而不代表在实际的编译器中也是按照这样的输出格式来设计编译器的

例如有如下程序段：

```
const int const1 = 1, const2 = -100;
const char const3 = '_';
int change1;
char change3;
int gets1(int var1,int var2){
    change1 = var1 + var2;
    return (change1);
}
void main(){
    printf("Hello World");
    printf(gets1(10, 20));
}
```

则输出的结果文件具有如下内容：

```
CONSTTK const
INTTK int
```

```
IDENFR const1
ASSIGN =
INTCON 1
COMMA ,
IDENFR const2
ASSIGN =
MINU -
INTCON 100
SEMICN ;
CONSTTK const
CHARTK char
IDENFR const3
ASSIGN =
CHARCON _
SEMICN ;
INTTK int
IDENFR change1
SEMICN ;
CHARTK char
IDENFR change3
SEMICN ;
INTTK int
IDENFR gets1
LPARENT (
INTTK int
IDENFR var1
COMMA ,
INTTK int
IDENFR var2
RPARENT )
LBRACE {
IDENFR change1
ASSIGN =
IDENFR var1
```



```
PLUS +
IDENFR var2
SEMICN ;
RETURNTK return
LPARENT (
IDENFR change1
RPARENT )
SEMICN ;
RBRACE }
VOIDTK void
MAINTK main
LPARENT (
RPARENT )
LBRACE {
PRINTF TK printf
LPARENT (
STRCON Hello World
RPARENT )
SEMICN ;
PRINTF TK printf
LPARENT (
IDENFR gets1
LPARENT (
INTCON 10
COMMA ,
INTCON 20
RPARENT )
RPARENT )
SEMICN ;
RBRACE }
```

第 4 章 语法分析

语法分析程序是编译过程中的核心部分，这一阶段的任务就是检查源程序是否符合规定的文法，并将源程序识别为不同的语法成分，为后续的语义分析和生成中间代码做准备。在实践课中，推荐使用递归下降分析法（一种自顶向下的分析方法）来实现语法分析部分。

4.1 语法分析阶段需要完成的一些功能

4.1.1 检查源程序是否符合文法

语法分析最主要的功能之一，就是检查要编译的源程序是否符合文法，只有符合文法，才能够进行下面的中间代码生成以及目标代码生成。

而递归下降分析法对每一个非终结符号都编写出一个子程序，以完成该非终结符号所对应的语法成分的分析与识别任务，若正确识别，则可以退出该非终结符号的子程序，返回到上一级的子程序继续分析；若发生错误，即源程序不符合文法，则要进行相应的错误信息报告以及错误处理。

由于并不一定所有的文法都是 LL(1) 文法，因此往往只预读一位并没法判断改进入到哪一种语法成分的分支中去。

如：

`< 变量定义 > ::= < 类型标识符 >(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']') , < 标识符 > | < 标识符 > '[' < 无符号整数 > ']')`

`< 类型标识符 > ::= int | char`

`< 有返回值函数定义 > ::= < 声明头部 > '(' < 参数表 > ') ' < 复合语句 > ' '`

`< 声明头部 > ::= int < 标识符 > | char < 标识符 >`

对于以上文法，当出现 `int cnt;` 和 `int init().....` 的时候，读完 `int` 和 `< 标识符 >` 后，都无法判断应该继续进入 `< 变量定义 >` 子程序还是进入 `< 有返回值函数定义 >` 子程序。

对于这种非 LL(1) 文法，有两种解决方案：

(1) 将文法改写为 LL(1) 文法

这种方法书中有介绍，需要注意的是，要保证改写前后文法是等价的，即要保证改写前后的文法所确定的语言是相同的。

(2) 进行预读多位

另一种方法就是继续预读，直到读到那个能够区分该进入哪个分支的单词为止。比如说上文中提到的，当出现 `int cnt;` 和 `int init().....` 的时候，该如何判断进入 `< 变量定义 >` 子程序还是进入 `< 有返回值函数定义 >` 子程序。当程序预读到第三位，即读到 `;` 或 `(` 的时候，就可以判断应当继续进入哪一种子程序了。这种预读来解决非 LL(1) 文法的方法要注意对预读的多位单词的处理，无论是 (1) 将已经预读的各个单词保存下来再在子程序中进行处理还是 (2) 在此处直接处理完成后再进入子程序，只要考虑清楚如何处理预读的多位单词即可。

4.1.2 报告错误信息及错误处理

当语法分析遇到了不符合文法的部分的时候，要返回合适的错误信息以提醒用户是什么地方出了错。这也就意味着，不同的错误要抛出不同的错误信息提示，这里提供一种实现方法，即写一个错误信息处理的函数，传入不同的错误类别码就输出不同类型的错误信息，如：

```
void ErrorMessage(char ErrorType){
    //linenum为全局变量，记录源程序的当前所在行数
    switch(ErrorType){
        case 'a':
            printf("第%d行，不符合词法规则\n", linenum);
            .....
            break;
        case 'b':
            printf("第%d行，重定义\n", linenum);
            .....
            break;
        case 'c':
            printf("第%d行，未定义符号\n", linenum);
            .....
            break;
        //.....
        default: printf("第%d行，未知错误\n", linenum); break;
```

```
}  
}
```

与上述代码不同的是，在错误处理考核时，不用输出具体信息，而只输出类别码。可以考虑引入一个宏定义或条件编译的开关调节具体输出格式，而不必多次修改代码。

在需要进行报错的时候，只需要调用 `ErrorMessage()` 函数并传入相应的错误类别码即可，不过这也就要求同学要自己建立类别码和错误类型之间的映射关系。

仅进行报错还是不够的，发现错误后报错然后直接结束编译固然没错，但这会使得编译效率大大降低。设想一个有 100 个 bug 的程序，则最理想估计也需要编译 101 次才能够正确编译（每次发现错误后都可以改对），这样的低效情况是我们不愿意看到的。我们身为北航的优秀学子，写出的编译器肯定是编译一次就要将尽可能多的错误都报出来，因此我们可以进行错误局部化处理，使得检查到一个错误之后还可以继续向后处理之后的内容，可以尽可能多的检查出程序中存在的错误。

比如：

当出现句末缺少分号时，可以输出报错信息，然后忽略这个错误正常返回到上一级子程序；当非法字符出现时，要进行适当的跳读，然后继续分析等等……当然，由于跳读以后会进入到不可预知的状态，后续报告的错误可能有偏差，但至少要保证第一个错误一定要被发现并准确报出，并减少错误的影响范围。

4.1.3 符号表的建立

语法分析阶段还有一项工作就是建立符号表，其实符号表的建立并不是一定要放在语法分析阶段来做，但在语法分析阶段建立符号表可以更有助于接下来的语义分析。

符号表，顾名思义，就是记录符号相关信息的表格，那这个符号都包含哪些呢？常量、变量、函数名和在生成中间代码时会用到的临时变量等等，他们都有一些“属性”需要记录下来，方便之后查找、检验。符号表既然是个表格，就肯定有“增改删查”操作，当新的变量（为叙述简洁，下文中将常量、变量、函数名、临时变量等简称为变量，此处变量指的是需要在符号表中记录的符号）出现时，就要先查一下这个变量是否之前被声明过，也就是“查”；如果没有，则要把它的相关信息记录到符号表中，也就是“增”；当某个符号的信息发生变化时，比如某个变量被重新赋值，它的信息也会相应的变化，就要修改符号表，也就是“改”。（删好像确实没有不过不要在意这些……）

符号表的具体设计因人而异，不过核心思想就是记录下的信息足够编译器使用即可，具体实现方式因人而异，每个人的数据结构（即要保存的信息不尽相同），设计多

少种表不同的同学也有不同的想法。有的同学一张表就可以搞定，有的可能对不同类型的符号所记录的信息不同，就会分成不同的多张表，比如说：全局变量/常量表、局部变量/常量表、函数名表等等。这里就不给出具体如何设计符号表了，希望大家可以根据自己的编译器后续需求出发，考虑如何设计符号表既可以满足编译器要查找的信息，又可以提高查找插入效率。比如：适当地引入标志位及多张符号表，进行分类管理；用时间复杂度较低的查找插入算法等等。

有关符号表的数据结构，大家也可以参考 Pascal-S 编译器源代码中符号表的数据结构，也可以自行设计。但希望同学们在开始写程序之前可以想好符号表到底如何设计、要记录哪些信息、这些信息是不是足以支持编译器使用，如果考虑不充分，将面临着无休止地修改……

4.2 评测时语法分析输出约定

请提交语法分析程序的源代码，要求将语法分析结果输出到命名为"output.txt"的结果文件中。结果文件中每行按如下方式组织（中间仅用一个空格间隔）：

1. 按照之前的词法分析作业要求，按行输出每个单词的信息；
2. 在规定的语法分析成分分析结束前，另起一行输出当前语法成分的信息，形如“< 常量说明 >”。

注意：

1、请注意语法成分信息与单词信息的前后顺序，尤其是在分界处。如“int a;”，当读完“int a”后应当输出< 变量定义 >；继续读完“;”后，如果后面不再有变量定义语句，应当输出< 变量说明 >。要着重注意< 变量定义 >和“;”的前后输出顺序，在实现程序中，我们往往在退出< 变量定义 >的子程序时，当前单词 token 中存的是下一个单词，即“;”，如果将单词输出放在词法分析阶段，就可能会导致先输出“;”的单词信息，后输出< 变量定义 >的语法成分信息的情况，从而导致错误。

2、当某个单词结束后，若同时完成了两个语法成分，应当与子程序返回顺序相同，由内至外输出语法成分。如“-1”，当读完“1”的时候，首先“1”是一个无符号整数，因此要先输出“< 无符号整数 >”，而后“-1”又是一个整数，因此要在“< 无符号整数 >”后面输出“< 整数 >”。



注意 需要说明的是，这里对于输出格式的规定，仅是为了方便评测，而不代表在实际的编译器中也是按照这样的输出格式来设计编译器的。

例如有如下程序段：




```
const int const1 = 1, const2 = -100;
const char const3 = '_';
int change1;
char change3;
int gets1(int var1,int var2){
    change1 = var1 + var2;
    return (change1);
}
void main(){
    printf("Hello World");
    printf(gets1(10, 20));
}
```

则输出的结果文件具有如下内容:

```
CONSTTK const
INTTK int
IDENFR const1
ASSIGN =
INTCON 1
<无符号整数>
<整数>
COMMA ,
IDENFR const2
ASSIGN =
MINU -
INTCON 100
<无符号整数>
<整数>
<常量定义>
SEMICN ;
CONSTTK const
CHARTK char
IDENFR const3
ASSIGN =
```

```
CHARCON _  
<常量定义>  
SEMICN ;  
<常量说明>  
INTTK int  
IDENFR change1  
<变量定义>  
SEMICN ;  
CHARTK char  
IDENFR change3  
<变量定义>  
SEMICN ;  
<变量说明>  
INTTK int  
IDENFR gets1  
<声明头部>  
LPARENT (  
INTTK int  
IDENFR var1  
COMMA ,  
INTTK int  
IDENFR var2  
<参数表>  
RPARENT )  
LBRACE {  
IDENFR change1  
ASSIGN =  
IDENFR var1  
<因子>  
<项>  
PLUS +  
IDENFR var2  
<因子>  
<项>
```

<表达式>
<赋值语句>
SEMICN ;
<语句>
RETURNTK return
LPARENT (
IDENFR change1
<因子>
<项>
<表达式>
RPARENT)
<返回语句>
SEMICN ;
<语句>
<语句列>
<复合语句>
RBRACE }
<有返回值函数定义>
VOIDTK void
MAINTK main
LPARENT (
RPARENT)
LBRACE {
PRINTF TK printf
LPARENT (
STRCON Hello World
<字符串>
RPARENT)
<写语句>
SEMICN ;
<语句>
PRINTF TK printf
LPARENT (
IDENFR gets1

LPARENT (
INTCON 10
<无符号整数>
<整数>
<因子>
<项>
<表达式>
COMMA ,
INTCON 20
<无符号整数>
<整数>
<因子>
<项>
<表达式>
<值参数表>
RPARENT)
<有返回值函数调用语句>
<因子>
<项>
<表达式>
RPARENT)
<写语句>
SEMICN ;
<语句>
<语句列>
<复合语句>
RBRACE }
<主函数>
<程序>

第 5 章 语义分析

如果说语法分析是让编译器能够正确地读取源程序中的各个语法成分，那么语义分析则是让编译器能够读懂源程序、将源程序之于编译器从“无意义的字符串”转化为“有逻辑的代码”。语义分析和语法分析虽说是两个阶段，但其实语义分析和语法分析是同时进行的，但两者的主要任务还是有一些区别的。

5.1 语义分析与语法分析的区别

语法分析检查的是读到的单词符号串是否符合规定的文法，只要符合即可通过语法分析。而语义分析关注的则更加深刻一些，它所检察的是语法分析所读到的各个语法成分之间，是否存在逻辑错误。有的字符串是可以通过语法分析的，但逻辑上是存在错误的。

例如有如下程序段：

```
const int a = 1706;
int b;

void main(){
    b = 1606;
    a = b;
}
```

“a = b;”从语法分析来说，它是符合“< 赋值语句 > ::= < 标识符 > = < 表达式 >”的文法的，因为 a 是一个符号表中可以找得到的标识符，而 b 是一个 < 表达式 >，所以语法分析会认为他是正确的并不会报错。然而，给一个常量赋值，肯定是不正确的，这时就需要语义分析来检查错误了，在赋值之前要检查被赋值的标识符是否可以赋值的，以及等号两边的类型是否匹配。

当然，这种语义错误只是举一个例子，真实编译中不符合语义的情况有很多，相信聪明又细心的你一定可以处理好这些错误检查！

5.2 语义分析阶段需要完成的一些功能

5.2.1 符号表的建立

在语法分析阶段，我们说过推荐在语法分析过程中完成符号表的建立，而严格来说，符号表的建立应该属于语义分析的任务，这也体现了语法分析和语义分析实际上是同时进行的。有关符号表的建立我们就不再赘述了，具体数据结构和方法可以参考上一章语法分析和 **Pascal-S** 的源代码。不过有一点还是想强调一下，就是在着手写代码之前，一定一定一定要设计好符号表，考虑清楚哪些信息是需要存下来的，否则语义分析的语义正确性检查以及后面的生成中间代码都将举步维艰。

5.2.2 语义正确性检查

正如“语义分析与语法分析的区别”中所讲到的，语义分析要检查输入的源程序是否符合逻辑，是否存在比如：给常量赋值、类型不匹配、参数个数不匹配等等的错误。希望同学们在进行语法分析和语义分析的时候，多考虑一下当前语法成分可能会出现哪些类型错误，多从“非法”的角度去思考。

5.2.3 生成中间代码

在第九章中，我们学习了在文法中插入动作符号，将文法改写成一个翻译文法，而动作符号就对应着一些子程序（某些操作），放在编译器中就可以理解为生成中间代码。语义分析和生成中间代码是密不可分的（然而我们还是将它们分开了），在编译器读懂了源程序以后，自然要将源程序中某一小段字符串，翻译成和它意义相同的、“等价的”中间代码。而中间代码也只是半成品，它作为编译器的一个中间环节，将会成为生成目标代码程序的输入，即编译器根据生成的中间代码生成目标代码，所以中间代码的重要性可想而知。在下一章，我们将会从一些语句切入，介绍一下如何正确生成中间代码，相信聪明的你一定可以举一反三，顺利完成源程序到中间代码的转化！