2024编译技术实验文法定义及相关说明

一、概要

SysY 语言是实验所完成的编译器的源语言，是 C 语言的一个子集。该文件中有且仅有一个名为 main 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。SysY 语言支持 **int** 和 **char 两种类型**和这两种数据类型的**一维数组类型**，其中 int 型整数为 32 位有符号数，char 为8位无符号整数；**const** 修饰符用于声明常量。

SysY 语言通过 getint，getchar 以及 printf 函数完成 I/O 交互，函数用法已在文法中给出，需要同学们自己实现。

- **函数**：函数可以带参数也可以不带参数，参数的类型可以是 int, char 或者两者的一维数组类型；函数可以返回 int 类型或者 char 类型的值，或者不返回值(即声明为 void 类型)。当参数为 int 或 char 时，按值传递；而参数为数组类型时，实际传递的是数组的起始地址。函数体由若干变量声明和语句组成。
- **变量/常量声明**：可以在一个变量/常量声明语句中声明多个变量或常量，声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量，在函数内声明的为局部变量/常量。
- **语句**：语句包括赋值语句、表达式语句(表达式可以为空)、语句块、if 语句、for 语句、break语句、continue 语句、return 语句。语句块中可以包含若干变量声明和语句。
- **表达式**：支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。**算符的优先级和结合性以及计算规则(含逻辑运算的"短路计算")与 C 语言一致。**

二、输入输出函数定义

```
<InStmt> → <LVal> = 'getint''('')'
<InStmt> → <LVal> = 'getchar''('')'
<OutStmt> → 'printf''('<StringConst>{,<Exp>}')'
```

其中，**StringConst** 为字符串常量终结符，其规范参考第三章**文法及测试程序覆盖要求**的第三小节第五部分**字符串常量。**

因此，语句`<Stmt>`的右侧需要添加上述输入输出语句，即：

```
<Stmt> → ...
    | <LVal> = 'getint''('')'';'
    | <LVal> = 'getchar''('')'';'
    |'printf''('<StringConst>{,<Exp>}')'';
```

如果希望在C语言中测试测试程序，只需要将 getint 以及 getchar 加入函数声明即可，示例如下：

```
int getchar(){
    char c;
    scanf("%c",&c);
    return (int)c;
}
int getint(){
    int t;
    scanf("%d",&t);
    while(getchar()!='\n');
    return t;
}
```

注意： getchar() 的返回值为 int ， 因此应该取返回值的低8位赋值给 char 类型的变量；getint() 会读取一整行。

三、文法及测试程序覆盖要求

1. 覆盖要求

测试程序是为了测试编译器而编写的符合文法规则的SysY语言程序，在实验的“文法解读作业”中需要同学们编写测试程序。测试程序需覆盖文法中**所有的语法规则**（即每一条推导规则的每一个候选项都要被覆盖），并**满足文法的语义约束**（换而言之，测试程序应该是正确的）。在下一节中，文法正文中以**注释形式**给出需要覆盖的情况。

2. 文法

SysY 语言的文法采用扩展的 Backus 范式（EBNF，Extended Backus-Naur Form）表示，其中：

- 符号[...]表示方括号内包含的为可选项
- 符号{...}表示花括号内包含的为可重复 0 次或多次的项
- 终结符或者是由单引号括起的串，或者是 Ident、IntConst、CharConst、StringConst 这样的记号
- **所有类似 'main' 这样的用单引号括起的字符串都是保留的关键字**

SysY 语言的文法表示如下，其中 CompUnit 为开始符号：

重要：建议同时对照文法第三部分的语义约束。

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef // 1.是否存在Decl 2.是否存在FuncDef

声明 Decl → ConstDecl | VarDecl // 覆盖两种声明

常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // 1.花括号内重复0次 2.花括号内重复多次

基本类型 BType → 'int' | 'char' // 覆盖两种数据类型的定义

常量定义 ConstDef → Ident [ '[' ConstExp ']' ] '=' ConstInitVal // 包含普通变量、一维数组两种情况

常量初值 ConstInitVal → ConstExp | '{' [ ConstExp { ',' ConstExp } ] '}' | StringConst // 1.常表达式初值 2.一维数组初值

变量声明 VarDecl → BType VarDef { ',' VarDef } ';' // 1.花括号内重复0次 2.花括号内重复多次
```

变量定义 VarDef → Ident ['[' ConstExp ']'] | Ident ['[' ConstExp ']'] '=' InitVal // 包含普通常量、一维数组定义

变量初值 InitVal → Exp | '{' [Exp { ',' Exp }] '}' | StringConst // 1.表达式初值 2.一维数组初值

函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block // 1.无形参 2.有形参

主函数定义 MainFuncDef → 'int' 'main' '(' ')' Block // 存在main函数

函数类型 FuncType → 'void' | 'int' | 'char'// 覆盖三种类型的函数

函数形参表 FuncFParams → FuncFParam { ',' FuncFParam } // 1.花括号内重复0次 2.花括号内重复多次

函数形参 FuncFParam → BType Ident ['[' ']'] // 1.普通变量2.一维数组变量

语句块 Block → '{' { BlockItem } '}' // 1.花括号内重复0次 2.花括号内重复多次

语句块项 BlockItem → Decl | Stmt // 覆盖两种语句块项

语句 Stmt → LVal '=' Exp ';' // 每种类型的语句都要覆盖
| [Exp] ';' //有无Exp两种情况
| Block
| 'if' '(' Cond ')' Stmt ['else' Stmt] // 1.有else 2.无else
| 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt // 1. 无缺省，1种情况 2. ForStmt与Cond中缺省一个，3种情况 3. ForStmt与Cond中缺省两个，3种情况 4. ForStmt与Cond全部缺省，1种情况
| 'break' ';' | 'continue' ';'
| 'return' [Exp] ';' // 1.有Exp 2.无Exp
| LVal '=' 'getint''('')'';'
| LVal '=' 'getchar''('')'';'
| 'printf''('StringConst {','Exp}')'';' // 1.有Exp 2.无Exp

语句 ForStmt → LVal '=' Exp // 存在即可

表达式 Exp → AddExp // 存在即可

条件表达式 Cond → LOrExp // 存在即可

左值表达式 LVal → Ident ['[' Exp ']'] //1.普通变量、常量 2.一维数组

基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number | Character// 四种情况均需覆盖

数值 Number → IntConst // 存在即可，IntConst详细解释见下方 (3) 数值常量

字符 Character → CharConst // CharConst详细解释见下方 (4) 字符常量

一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp // 3种情况均需覆盖,函数调用也需要覆盖FuncRParams的不同情况

单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中 // 三种均需覆盖

函数实参表 FuncRParams → Exp { ',' Exp } // 1.花括号内重复0次 2.花括号内重复多次 3.Exp需要覆盖数组传参和部分数组传参

```
乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp // 1.UnaryExp 2.* 3./ 4.% 均需覆盖

加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp // 1.MulExp 2.+ 需覆盖 3.- 需覆盖

关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp // 1.AddExp 2.< 3.> 4.<= 5.>= 均需覆盖

相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp // 1.RelExp 2.== 3.!= 均需覆盖

逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp // 1.EqExp 2.&& 均需覆盖

逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp // 1.LAndExp 2.|| 均需覆盖

常量表达式 ConstExp → AddExp 注：使用的 Ident 必须是常量 // 存在即可
```

3. 终结符特征

(1) 标识符 Ident

SysY 语言中标识符 **Ident** 的规范如下 (identifier) :

```
identifier → identifier-nondigit
    | identifier identifier-nondigit
    | identifier digit
```

其中，identifier-nondigit为下划线或大小写字母，digit为0到9的数字。

注：请参考 ISO/IEC 9899 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf第 51 页关于标识符的定义**同名标识符**的约定：

- 全局变量（常量）和局部变量（常量）的作用域可以重叠，重叠部分局部变量（常量）优先
- 同名局部变量的作用域不能重叠，即同一个 Block 内不能有同名的标识符（常量/变量）
- 在不同的作用域下，SysY 语言中变量（常量）名可以与函数名相同
- 保留关键字不能作为标识符

```c
// 合法样例
int main() {
    int a = 0;
    {
    int a = 1;
    printf("%d",a); // 输出 1
    }
    return 0;
}
```

```
// 非法样例
int main() {
    int a = 0;
    {
    int a = 1;
    int a = 2; // 非法!
    }
    return 0;
}
```

(2) 注释

SysY 语言中注释的规范与 C 语言一致，如下：

- 单行注释：以序列 '//' 开始，直到换行符结束，不包括换行符。
- 多行注释：以序列 '/*' 开始，直到第一次出现 '*/' 时结束，包括结束处'*/'。

注：请参考 ISO/IEC 9899 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf第 66 页关于注释的定义

(3) 数值常量

SysY 语言中数值常量可以是整型数 **IntConst**，其规范如下（对应 integer- const）：

```
    整型常量 integer-const → decimal-const | 0

•   decimal-const → nonzero-digit | decimal-const digit

•   nonzero-digit 为1至9的数字。
```

注：请参考 ISO/IEC 9899 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf第 54 页关于整型常量的定义，在此基础上忽略所有后缀。

(4) 字符常量

SysY 语言中数值常量可以是单个字符 **CharConst**，其规范如下

```
CharConst → '\''ASCII'\''   \\这里的 '\'' 表示单引号 '
```

- **ASCII** 包括以下内容
  - 包括 32-126 的所有 ascii字符，具体如下表（十进制表示ASCII码，也就是字符在计算机中的数值），其中 ", ', \需要转义。

十进制	图形	十进制	图形	十进制	图形	十进制	图形	十进制	图形
32	space	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o

十进制	图形	十进制	图形	十进制	图形	十进制	图形	十进制	图形
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

- 还包括转义字符，具体如下表

十进制	转义字符	十进制	转义字符	十进制	转义字符
007	\a	011	\v	092	\\
008	\b	012	\f	000	\0
009	\t	034	\"		
010	\n	039	\'		

注意：转义字符中不包括 \r，这是因为因换行符在windows系统和linux系统不同，为了便于评测，转义字符不包括\r。

(5) 字符串常量

SysY 语言中字符串常量 **StringConst** 组成如下：

```
StringConst → '"' {ASCII} '"'
```

当 **StringConst** 出现在 printf 后时，还将遵从以下约束：

- 转义字符中只会出现 **'\n'**。
- **单引号**、**双引号**不会作为转义字符出现，也不会直接出现。

- 反斜杠只会与 n 一起出现作为一个整体表示换行。

4. 难度分级

为了更好的帮助同学们完成编译器的设计，课程组今年将题目难度做了区分。难度等级分为三级：A、B、C，难度依次递减。下文将对难度等级进行详细介绍：

- C：最简单的等级，重点考察编译器的基础设计，不包括任何与数组有关的部分，if 与 for 中的条件表达式也只有一个表达式（即不出现 **&&** 和 **||** ）不考察短路求值的规则。
- B：在 C 级的基础上，**新增**数组，包括数组定义，数组元素的使用、数组传参等。
- A：在 B 级的基础上，**新增**复杂条件的运算和判断，要遵守短路求值的规则。

  短路求值样例：

```
int global_var = 0;
int func() {
    global_var = global_var + 1;
    return 1;
}
int main() {
    if (0 && func()){
        ;
    }
    printf("%d",global_var); // 输出 0
    if (1 || func()) {
        ;
    }
    printf("%d",global_var); // 输出 0
    return 0;
}
```

四、 语义约束

符合上述文法的程序集合是合法的 SysY 语言程序集合的超集。下面进一步给出 SysY 语言的语义约束。

特别说明

- 在编写测试样例的过程中，对 char 类型进行赋值时，程序本身必须保证赋值为数值在 [0, 127] 这个区间内的整数。
- **使用 char 类型进行运算时编译器必须先将 char 扩展为 int 类型再使用扩展后的数进行计算。**
- 对数组的访问应保证不超过数组的长度。
- 对于字符数组的初始化，需要注意字符数组末尾会有一个默认的 ‘/0’ 会占据一个位置。

```
char s[5] = "abcd"; //合法
char s[5] = "abcde"; //不合法
```

I/O语句

- 输入语句以函数调用的形式出现，对应函数声明如下。虽然输入语句以函数调用形式出现，**getint 与 getchar 仍被识别为关键字而不是标识符**，在词法分析阶段和语法分析阶段需要注意。

```
int getint();
int getchar();
```

  - **getint 会跳过空格，制表符（`\t`），换行符（`\n`），回车符（`\r`），以及其他可能的空白字符。**
  - **getchar 不会跳过空格，制表符（`\t`），换行符（`\n`），回车符（`\r`），以及其他可能的空白字符。**
  - **getint 只能用于为 int 类型变量赋值，getchar 只能用于为 char 类型变量赋值。**

```
int x;
x = getint(); // 合法
x = getchar(); // 不合法
char c;
c = getint(); // 不合法
c = getchar(); // 合法
```

  - **在设计编译器时，应该先将 getchar 返回值从 int 类型截断为 char 类型再赋值给 char 类型变量。**

- 与 C 语言中的 printf 类似，输出语句中，格式字符将被替换为对应标识符，普通字符原样输出。其中格式字符只包含 **%d** 与 **%c**，其他 C 语言中的格式字符，如 %f 都当做普通字符原样输出。
- printf 默认不换行。

编译单元 CompUnit

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef

声明 Decl → ConstDecl | VarDecl
```

注意：

1. CompUnit 的**顶层**变量/常量声明语句（对应 Decl ）、函数定义（对应 FuncDef ）都不可以重复定义同名标识符（ Ident ），即便标识符的类型不同也不允许。
2. CompUnit 的变量/常量/函数声明的作用域从该声明处开始到文件结尾。

常量定义 ConstDef

```
常量定义 ConstDef → Ident [ '[' ConstExp ']' ] '=' ConstInitVal
```

1. ConstDef 用于定义符号常量。ConstDef 中的 Ident 为常量的标识符，在 Ident后、'=' 之前是可选的一维数组以及一维数组长度的定义部分，在 '=' 之后是初始值。**常量必须有对应的初始值。**
2. ConstDef 的一维数组以及一维数组长度的定义部分不存在时，表示定义单个变量。
3. ConstDef 的一维数组以及一维数组长度的定义部分存在时，表示定义数组。其语义和 C 语言一致，每维的下界从 0 编号。 ConstDef 中表示各维长度的 ConstExp 都必须能在编译时求值到**非负整数**，即只能使用常数、可以取得具体值常量以及由它们组成的算术表达式。

变量定义 VarDef

```
变量定义 VarDef → Ident [ '[' ConstExp ']' ] | Ident [ '[' ConstExp ']' ] '='
InitVal
```

1. 变量的定义与常量基本一致，但是变量可以没有初始值部分。

初值 ConstInitVal / InitVal

```
常量初值 ConstInitVal → ConstExp | '{' [ ConstExp { ',' ConstExp } ] '}' |
StringConst

变量初值 InitVal → Exp | '{' [ Exp { ',' Exp } ] '}' | StringConst
```

1. 任何常量的初值表达式必须是常量表达式 ConstExp。
2. 常量数组的每一个元素都必须给定初始值。
3. 全局变量的初值表达式也必须是常量表达式 ConstExp。但局部变量的初值表达式是 Exp，可以使用已经声明的变量。
4. 对于单个的常量或变量，初始值也必须是单个的表达式。（表达式包括单个数字，单个字符，单个变量，单个常量等情况）
5. 对于任何有初始值的字符数组，编译器应该在初始化时将未使用的部分置0，例子如下：

```
char s[10] = "abcde";
// 也就是说从 s[5] 至 s[9] 都应该是 '\0'，对应的 ascii 码值是 0
```

6. **对于全局变量，如果没有给出初始值，那么应该全部置 0，局部变量不需要。**

函数形参 FuncFParam 与实参 FuncRParams

```
函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }
函数实参表 FuncRParams → Exp { ',' Exp }
```

1. FuncFParam 定义一个函数的一个形式参数。当 Ident 后面的可选部分存在时， 表示数组定义。
2. 函数实参的语法是 Exp。对于普通变量，遵循按值传递；对于数组类型的参数，则形参接收的是实参数组的地址，并通过地址间接访问实参数组中的元素。
3. 普通常量可以作为函数参数，但那是常量数组不可以，如 const int arr[3] = {1,2,3}，常量数组 arr **不能**作为参数传入到函数中
4. 函数调用时要保证实参类型和形参类型一致，具体请看下面例子。

```
void f1(int x){
    return ;
}
void f2(int x[]){
    return ;
}
void f3(char c){
    return ;
}
void f4(char c[]){
```

```
    return ;
}

int main(){
    int x = 10;
    int t[5] = {1, 2, 3, 4, 5};
    char c = 'a';
    char s[5] = "abcd";
    f1(x); // 合法
    f1(c); // 合法，具体原因见下方第五章中的解释
    f1(t[0]); // 合法
    f1(s[0]); // 合法，具体原因见下方第五章中的解释
    f1(t); // 不合法
    f2(t); // 合法
    f2(s); // 不合法
    f3(x); // 合法，具体原因见下方第五章中的解释
    f3(c); // 合法
    f3(t[0]); // 合法，具体原因见下方第五章中的解释
    f3(s[0]); // 合法
    f4(t); // 不合法
    f4(s); // 合法
}
```

函数定义 FuncDef

```
函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
```

1. . FuncDef 表示函数定义。其中的 FuncType 指明返回类型。当返回类型为 int 或 char 时，函数内所有分支都应当含有带有 Exp 的 return 语句。不含有 return 语句的分支的返回值未定义。当返回值类型为 void 时，函数内只能出现不带返回值的 return 语句。
2. FuncFParams 的语义如前文。

语句块 Block

1. Block 表示语句块。语句块会创建作用域，语句块内声明的常量和变量的生存期在该语句块内。
2. 语句块内可以再次定义与语句块外同名的变量或常量（通过 Decl 语句），其作用域从定义处开始到该语句块尾结束，它隐藏语句块外的同名变量或常量。
3. 为了简化学生开发编译器的难度，我们保证评测的样例程序中：有返回值的函数 Block 的最后一句一定会显式的给出 return 语句，否则就当作“无返回语句”的错误处理。同时，同学们编写上传的样例程序时，也需要保证这一点。
4. main 函数的返回值只能为常数0。

语句 Stmt

1. Stmt 中的 if 类型语句遵循就近匹配。
2. 单个 Exp 可以作为 Stmt。这个 Exp 会被求值，但是所求的值会被丢弃。

左值 LVal

1. LVal 表示具有左值的表达式，可以为变量或者某个数组元素。
2. 当 LVal 表示数组时，方括号个数必须和数组变量的维数相同（即定位到元素）。
3. 当 LVal 表示单个变量时，不能出现后面的方括号。
4. 对 char 类型进行赋值时，需要根据你的选择进行截断，使得这个赋值时符合预期的。当然，因为我们的限制条件，赋给 char 类型的值只在 0-127 上，因此可以放心的直接截取低八位。

Exp 与 Cond

1. **Exp 在 SysY 中代表 int 型表达式，这是因为在前文中提到，如果 char 类型参与计算，必须首先转化为 int，因此 Exp 定义为 AddExp；Cond 代表条件表达式，故它定义为 LOrExp。前者的单目运算符中不出现'!'，后者可以出现。**
2. LVal 必须是当前作用域内、该 Exp 语句之前有定义的变量或常量；对于赋值号左边的 LVal 必须是变量。
3. 函数调用形式是 Ident '(' FuncRParams ')'，其中的 FuncRParams 表示实际参数。实际参数的类型和个数必须与 Ident 对应的函数定义的形参完全匹配。
4. SysY 中算符的优先级与结合性与 C 语言一致，在上面的 SysY 文法中已体现出优先级与结合性的定义。

一元表达式 UnaryExp

1. 相邻两个 UnaryOp 不能相同，如 int a = ++-i; ，但是 int a = +-+i; 是可行的。
2. UnaryOp 为 '!' 只能出现在条件表达式中。

常量表达式 ConstExp

1. **ConstEXp 在 SysY 中代表 int 型表达式。**
2. ConstExp -> AddExp 中使用的 ident 必须是普通常量，**不包括数组元素、函数返回值**；当然 ConstExp 中还可以使用数值常量，字符常量。

五、对 int 与 char 的补充说明

当你完整阅读文法至此时，相信你对本次编译实验的文法已经有了充分的认知，但也许仍有许多疑问，尤其是关于 int 与 char 之间的类型转换问题。因此，本部分将通过例子对 int 和 char进行详细说明。

1. Exp 与 ConstExp

对于下面这样几个例子

```
int x1 = 'a' + 10; // 首先将 'a' 转化为对应的 ascii 码97，并将97转为一个 int 型整数再与10相加，最后赋值给 x1
int x2 = 'a' + 'a' + 'a'; // 同上面，先将两个 'a' 都转为 ascii 码对应的 int 整型，再相加赋值给 x2，因此 x2 的值应该是291，而不是41
char a1 = '0' - 7; // 同上，右边的 ConstExp 表达式计算结果应该是一个值为41的 int 型32位整数，取41的低8位赋值给 a1，因此 a1的值是41
int x3 = a1; // 即便只有一个变量，但等号右边本质依然是一个表达式，因此应该先将 a1 的值符号扩展为一个 int 类型的32位整数，再赋值给 x3
int x4 = a1 + a1; // 同上，先将 a1 的值转为 int 类型再相加，赋值给 x4
char a2 = a1; // 先将 a1 转为 int 类型，再取其低8位赋值给 a2
char a3 = a1 + a1; // 先将 a1 的值转为 int 类型再相加，再取低8位赋值给 a3
```

也就是说，对于任意一个 Exp 表达式或者 ConstExp 表达式，都应该先将其中的 char 类型变量，常量转为 int 类型再进行计算，在计算完成之后，再根据被赋值的对象的类型决定是否进行截断，如果被赋值的对象是 int 类型就直接进行赋值，如果是 char 那么应该取结果二进制的低8位进行赋值。

这些例子对于常量，变量的声明，以及 Lval = Exp 都是适用的，因为它们的本质都是将一个 Exp 表达式赋值给一个变量，或者一个 ConstExp 表达式赋值给一个常量。

2. getint 与 getchar

getint 只能用于为 int 类型变量赋值，getchar 只能用于为 char 类型变量赋值。

对于 char c = getchar(); 这样一个例子来说，应该是取 getchar() 返回的 int 类型的结果的低8位赋值给 char 类型变量 c，当然在样例中 getchar() 获取的值也应该在 [0, 128) 这个范围内。

3. FuncFParam 和 FuncRParams

在上面的例子已经说明了 exp，在调用函数时，我们传递的其实也是一个一个的 exp 表达式的计算结果，那么现在来看**语义约束**部分的这个例子：

```
void f1(int x){
    return ;
}
void f3(char c){
    return ;
}

int main(){
    int x = 10;
    int t[5] = {1, 2, 3, 4, 5};
    char c = 'a';
    char s[5] = "abcd";
    f1(c); // 合法，因为是将 c 的值转为 int 再传递给 f1，作为 f1 中形参 x 的值
    f1(s[0]); // 合法，理由同上
    f3(x); // 合法，将 x 这个 exp 表达式的值取低8位传递给 f3，作为 f3 中形参 c 的值
    f3(t[0]); // 合法，理由同上
}
```

六、错误处理

注意：以下内容是为了词法分析及以后的实验作业准备的，在完成文法解读作业时请编写符合上面文法的正确的源程序！

从词法分析作业开始，我们每次作业都要求同学们开发的编译能够处理正确的程序与错误的程序，对于正确的程序按照每次作业要求输出正确结果，对于错误的程序输出所有的错误的行号和错误类别码。

错误主要分为三类，词法分析部分错误，语法分析部分错误，语义分析部分错误。同学们在涉及编译器时，应该完成这三个部分的所有错误处理之后再进行错误的输出。

注意：后续实验中，对于错误的源程序完成语义分析后不进行中间代码生成。

错误类型列表

错误类别 a 为词法分析中会出现的错误。

错误类别 i, j, k 为语法分析中会出现的错误。

剩余错误类别均为语义分析中会出现的错误。

错误类型	**错误类别码**	**解释**	**对应文法及出错符号，...表示省略该条规则后续部分**
非法符号	a	出现了 '&' 和 '|' 这两个符号，应该将其当做 '&&' 与 '||' 进行处理，但是在记录单词名称的时候仍记录 '&' 和 '|'， 报错行号为 **'&' 或 '|' 所在的行号。**	LAndExp → LAndExp '&&' EqExp LOrExp → LOrExp '||' LAndExp
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号 **Ident** 所在行数。	ConstDef → Ident ... VarDef → Ident ... Ident ... FuncDef → FuncType Ident ... FuncFParam → BType Ident ...
未定义的名字	c	使用了未定义的标识符报错行号为 **Ident** 所在行数。	LVal → Ident ... UnaryExp → Ident ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。	UnaryExp → Ident '(' [FuncRParams] ')'
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。**关于参数类型不匹配会在下方特别说明进行详细说明。**	UnaryExp → Ident '(' [FuncRParams] ')'
无返回值的函数存在不匹配的return语句	f	报错行号为**'return'**所在行号。	Stmt → 'return' { '[' Exp ']' } ';'

错误类型	错误类别码	解释	对应文法及出错符号，...表示省略该条规则后续部分
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的'}'**所在行号。	FuncDef → FuncType Ident '(' [FuncFParams] ')' Block MainFuncDef → 'int' 'main' '(' ')' Block
不能改变常量的值	h	**LVal** 为常量时，不能对其修改。报错行号为 **LVal** 所在行号。	Stmt → LVal '=' Exp ';' Stmt → LVal '=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	Stmt, ConstDecl 及 VarDecl 中的 ';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用 (UnaryExp)、函数定义 (FuncDef, MainFuncDef) 、 Stmt 及 PrimaryExp 中的 ')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义 (ConstDef, VarDef, FuncFParam) 和使用 (LVal) 中的 ']'
printf中格式字符与表达式个数不匹配	l	报错行号为**'printf'**所在行号。	Stmt → 'printf' '(' FormatString {, Exp}')' ';'
在非循环块中使用break和continue语句	m	报错行号为 **'break'** 与 **'continue'** 所在行号。	Stmt → 'break';'|'continue';'

错误输出示例

输出结构

```
错误的行号 错误的类别码（中间仅用一个空格间隔）
```

样例输入

```
const int const1 = 1, const2 = -100;
int change1;
int gets1(int var1,int var2){
   const1 = 999;
   change1 = var1 + var2          return (change1);
}
int main(){
   change1 = 10;
   printf("Hello World$");
   return 0;
}
```

样例输出

```
4 h
5 i
```

文法符号与错误类型对应

为方便对照, 下文给出了文法符号与可能存在的错误的对应关系:

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef

声明 Decl → ConstDecl | VarDecl

常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // i

基本类型 BType → 'int' | 'char'

常量定义 ConstDef → Ident [ '[' ConstExp ']' ] '=' ConstInitVal // b k

常量初值 ConstInitVal → ConstExp | '{' [ ConstExp { ',' ConstExp } ] '}' |
StringConst

变量声明 VarDecl → BType VarDef { ',' VarDef } ';' // i

变量定义 VarDef → Ident [ '[' ConstExp ']' ] | Ident [ '[' ConstExp ']' ] '='
InitVal // b k

变量初值 InitVal → Exp | '{' [ Exp { ',' Exp } ] '}' | StringConst

函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block // b g j

主函数定义 MainFuncDef → 'int' 'main' '(' ')' Block // g j

函数类型 FuncType → 'void' | 'int' | 'char'

函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }

函数形参 FuncFParam → BType Ident ['[' ']'] // b k

语句块 Block → '{' { BlockItem } '}'

语句块项 BlockItem → Decl | Stmt

语句 Stmt → LVal '=' Exp ';' // h i
| [Exp] ';' // i
| Block
| 'if' '(' Cond ')' Stmt [ 'else' Stmt ] // j
| 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt // h
| 'break' ';' | 'continue' ';' // i m
| 'return' [Exp] ';' // f i
| LVal '=' 'getint''('')'';' // h i j
| LVal '=' 'getchar''('')'';' // h i j
| 'printf''('StringConst {','Exp}')'';' // i j l
```

```
语句 ForStmt → LVal '=' Exp // h

表达式 Exp → AddExp

条件表达式 Cond → LOrExp

左值表达式 LVal → Ident ['[' Exp ']'] // c k

基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number | Character// j

数值 Number → IntConst

字符 Character → CharConst

一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
// c d e j

单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中

函数实参表 FuncRParams → Exp { ',' Exp }

乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp

加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp

关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp

相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp

逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp // a

逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp // a

常量表达式 ConstExp → AddExp 注：使用的 Ident 必须是常量
```

特别说明

1. 错误 i，j，k 类型中的“前一个非终结符”强调的是在文法规则里出现在 `; ) ]` 之前的非终结符号，在分析中处理的是该非终结符产生的终结符号串的最后一个符号，也就是 `; ) ]` 本应该正常出现的位置的上一个单词。
2. 所有错误都不会出现恶意换行的情况，包括字符、字符串中的换行符、函数调用等等。所谓恶意换行是指**会使得错误处理难以正确进行的换行。**
3. 其他类型的错误，错误的行号以能够断定发现出错的第一个符号的行号为准。例如有返回值的函数缺少返回语句的错误，只有当识别到函数末尾的}时仍未出现返回语句，才可以断定出错，报错行号即为}的行号。
4. 为了避免因为测试程序中的错误导致出现语法二义性，使得语法树以错误的方式建立，我们保证：**for 语句不会出现除了 h 类型以外的任何错误。进一步的，我们保证，所有会导致语法树不能正确建立的错误都不会出现。**
5. 每一行中最多只有一个错误。
6. 对于一个名字重定义的函数，也应该完整分析函数内部是否具有其它错误。
7. 对于调用函数时，参数类型不匹配一共有以下几种情况的不匹配：

- 传递数组给变量。
- 传递变量给数组。
- 传递 char 型数组给 int 型数组。
- 传递 int 型数组给 char 型数组。