

# 实验一 词法分析器的实现

软工 1403 2014010919 蔡嘉豪

## 一、 实验目的

使用 C++编写一词法分析器，使其可以对 C 语言的子集(ANSI C 标准，支持共 35 个关键字，详见附录三)进行词法分析，将其切割成不同的词素，并生成相应的词法单元流。在分析过程中，能够尽可能多地找到错误与警告，并将其按出现的顺序输出。

## 二、 词法分析器的设计

### 1) 符号表的设计

符号表的设计主要参照 GCC1.21 中的 symseg.h 文件，其符号表的结构经整理如下图所示：

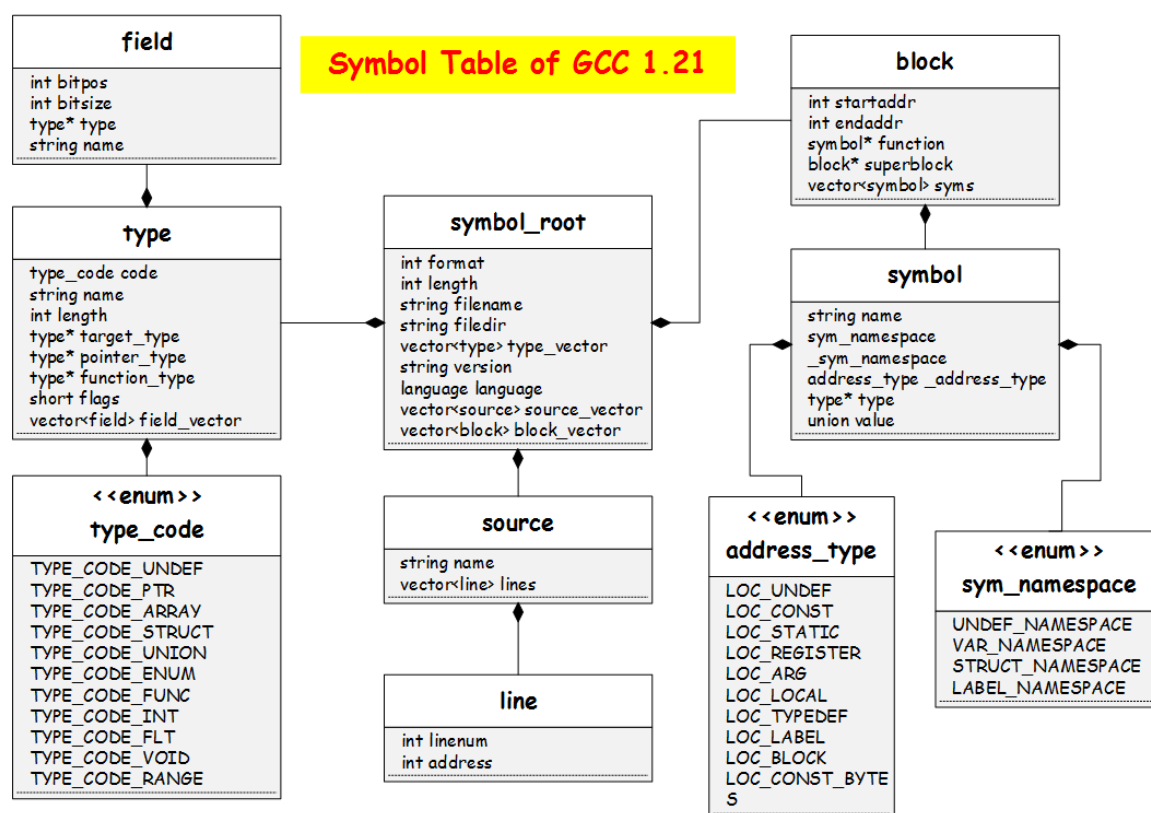


图 1 GCC 1.2.1 的符号表设计

由于 GCC 1.21 没有配置词法分析器，直接通过 bison 生成的语法分析器来进行语法分析，故其符号表没有为词法分析留下相应的空间。本词法分析器的符号表在此基础上稍作改动，增加了 2 个结构体，修改了多个结构体，使其能够支持词法分析。该符号表的基本结构如下图所示：

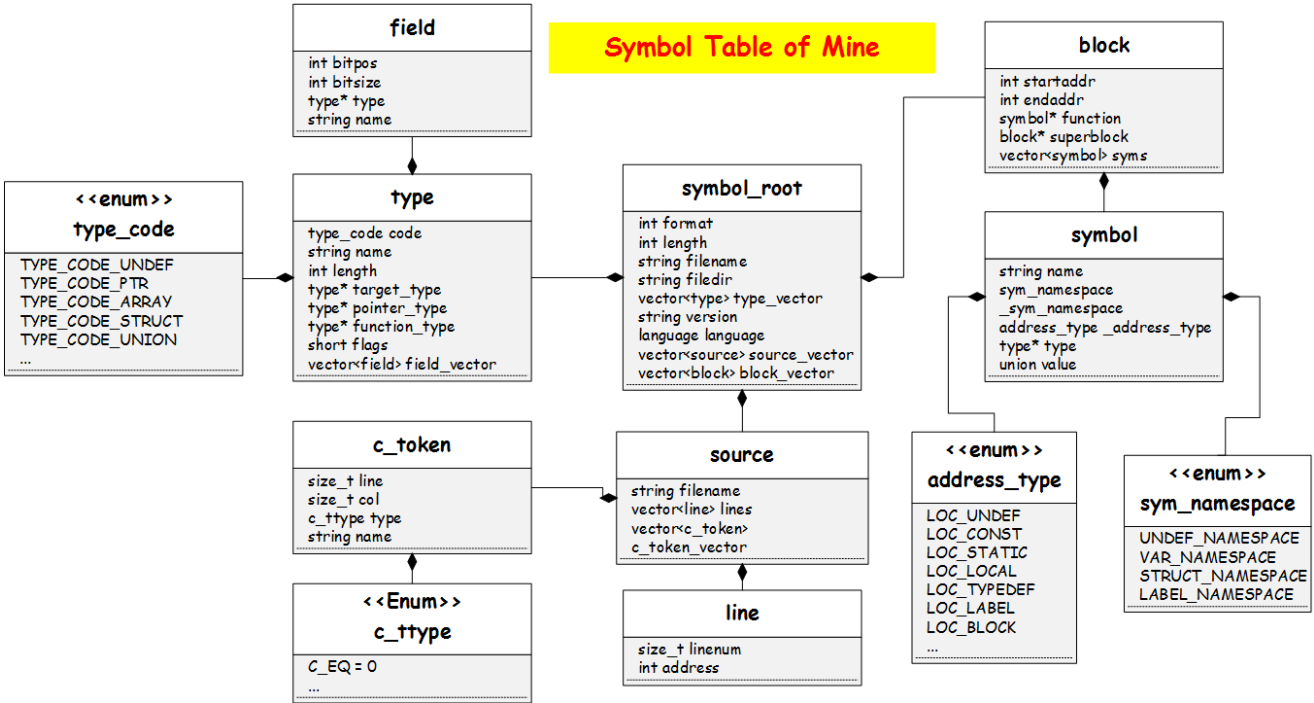


图 2 本词法分析器的符号表设计

本词法分析器主要用到了 source、line、c\_token、c\_ttype 这四个结构体，以及 symbol\_root 结构体中的几个属性，其具体含义如下表所示：

结构体	属性名	含义
Symbol_root	Filename	主文件名
	Filedir	主文件路径
	Version	当前版本号
	Language	当前要分析的语言
source	filename	源文件名
	lines	文件中每一行的信息的集合
	c_token_vector	所有token的集合
line	linenum	文件内行号
	address	行的地址
c_token	line	行号
	col	列号
	type	token的类型
	name	token的名称
c_ttype	所有属性	都表示该一种符号的类型 (该表太长，故在附录中给出，详见附录一)

图 3 部分符号表属性说明

## 2) 词法分析器运行机制

该词法分析器内置了简单的错误处理机制，当需要给出错误或警告时并不立即停止解析并退出，而是将当前错误或警告压入相应的队列，在分析完毕后按照先进先出的顺序将警告与错误先后输出到屏幕。

其运行机制大致如下：

- 词法分析器先判断 `lex_config.h` 中的 `#define TEST_ON` 宏是否存在，若存在则使用内置的测试数据来进行分析，若不存在则通过 `getline` 函数读取 `Sample.c`，`Sample.c` 是待分析的 C 文件(详见附录二)。
- 词法分析器使用 `lex()` 函数来分析读入的字符流，若分析过程中出现错误或警告，则分别 `push` 到 `Error queue` 与 `Warning queue` 中。
- 解析完成后，生成词法单元流，此时有两种选择，这两种选择通过 `lex_config.h` 中的 `#define DUMP_TOKEN_STREAM` 宏来控制（详见附录二）：
  1. 可以将词法单元流以及警告与错误输出到屏幕。
  2. 保留警告与错误，不将词法单元流输出到屏幕，而是将它们传给语法分析器继续解析。

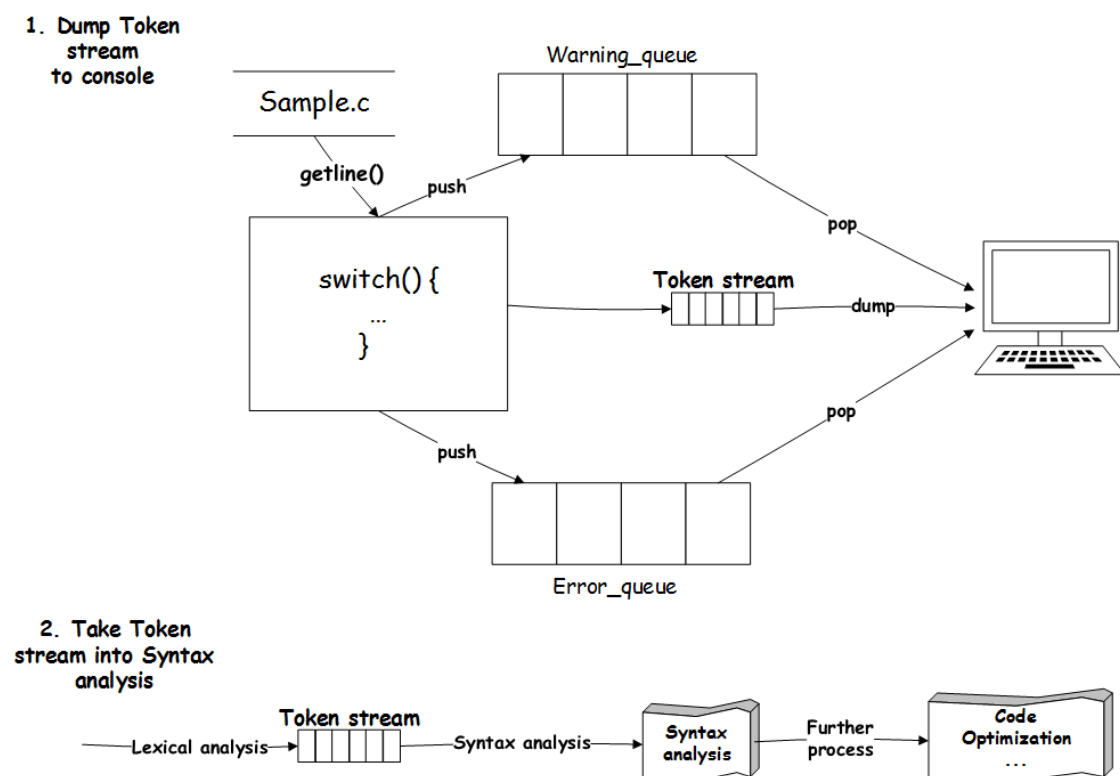


图 4 该词法分析器的运行机制

### 三、词法分析器的实现

#### 1) 准备工作

- 平台： Ubuntu 15.04
- 语言： C++ (C++11 标准)
- 编译工具： GNU GCC 4.9.2

#### 2) 词法分析的实现

因词法分析的符号种类过多而篇幅有限，且符号之间相似性较大，故只选取有代表性的数字、标识符以及字符串常量来说明

##### 1. 数字

数字的解析首先考虑数字的可能组成，主要考虑进制问题。

这里主要有三种情况：

- 以 0 开头的整数（八进制）
- 以 0x 或 0X 开头的整数（十六进制）
- 以 1-9 开头的整数（十进制）
- 小数

其 DFA 如下图所示：

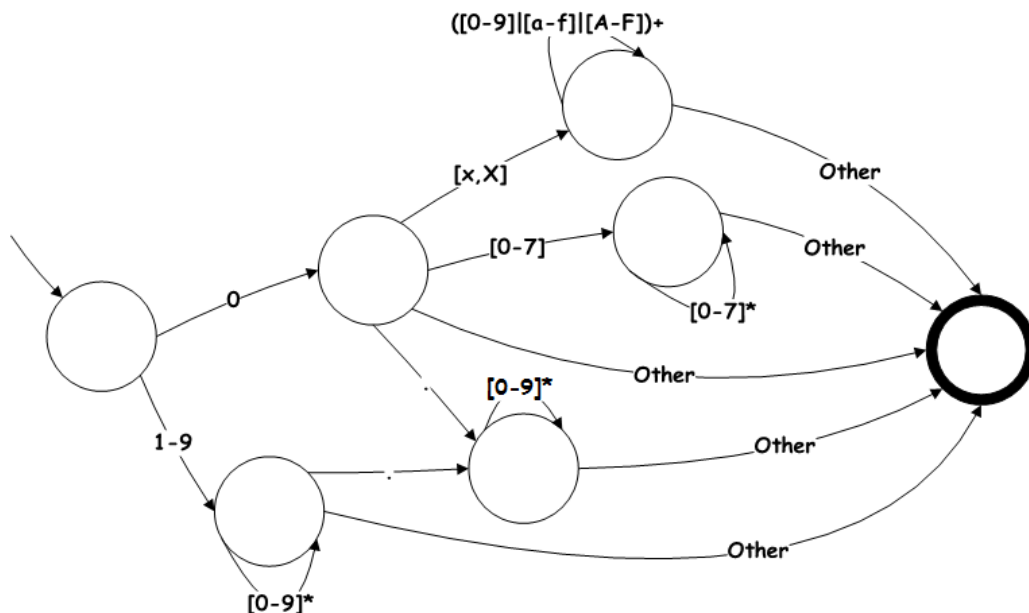


图 5 解析数字的 DFA

在程序实现时，使用相应的宏定义来替代图中简单的正则表达式：

```
#define ISDIGIT(ch) ((ch) >= '0' && (ch) <= '9')
#define ISDIGIT1T09(ch) ((ch) >= '1' && (ch) <= '9')
#define ISDIGIT0(ch) ((ch) == '0')
#define ISHEX(ch) (ISDIGIT((ch)) || ((ch) >= 'a' && (ch) <= 'f') || ((ch) >= 'A' && (ch) <= 'F'))
```

图 6 解析数字时所用到的宏

## 2. 标识符

首先查看 C 语言标识符的规则，如下图所示(该图片来自 MSDN):

**nondigit** : 以下项之一  
\_ a b c d e f g h i j k l m n o p q r s t u v w x y z  
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

**digit** : 以下项之一  
0 1 2 3 4 5 6 7 8 9

标识符名称的第一个字符必须是 **nondigit** (即, 第一个字符必须是下划线、大写字母或小写字母)。ANSI 允许外部标识符名称包含 6 个有效字符, 内部 (一个函数中) 标识符名称包含 31 个有效字符。外部标识符 (在全局范围内或使用存储类 **extern** 进行声明) 可能要受其他命名限制的约束, 因为这些标识符必须由其他软件 (如链接器) 处理。

图 7 C 语言标识符规则

标识符的构成情况相对简单，其 DFA 如下所示：

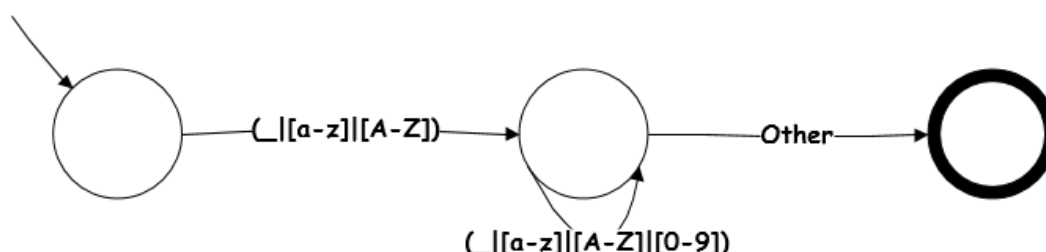


图 8 解析标识符的 DFA

在程序实现时，同样地，用宏来代替简单的正则表达式，需要注意的是，第一个字符必须单独处理。该宏的实现也非常简单，可以复用上一个例子中使用过的宏：

```
#define ISDIGIT(ch) ((ch) >= '0' && (ch) <= '9')
#define ISLETTER(ch) (((ch) >= 'a' && (ch) <= 'z') || ((ch) >= 'A' && (ch) <= 'Z'))
#define ISUNDERSCORE(ch) ((ch) == '_')
#define ISIDENTIFIER(ch) (ISDIGIT((ch)) || ISLETTER((ch)) || ISUNDERSCORE((ch)))
```

图 9 解析标识符所用到的宏

## 3. 字符串常量

C 语言规定，字符串常量是夹在两个双引号之间的内容；同时，C 语言也规定，符号可以被反斜杠(\)转义，改变其原有的含义。在字符串常量的解析过程中，要注意双引号被转义的情况。例如字符串“hello \”world”，不能被解析成 hello ，这样一来，后一个双引号会找到错误的双引号，或找不到一个可以匹配的双引号，导致词法分析器崩溃。该字符串应当被解析为 hello ”world。

明确了以上规则之后，就可以进行字符串常量的解析了。词法分析逐字符地判断是否找到正确的双引号，从而解析出字符串常量。同样地，这里还是用一个宏定义来判断：

```
#define ISREAL2QUOTE(ch1, ch2) ((ch1) != '\\' && (ch2) == '\"')
```

图 10 解析字符串常量所用到的宏

### 3) 词法分析的测试

词法分析器需求明确，是非常适合写测试的一种程序。在本程序中，使用了一个简单的测试框架，内置了一批测试数据，这些测试数据覆盖了该词法分析器的所有当前功能。在对该词法分析器进行修改后，只需重新运行一遍这些测试用例，即可确认当前修改是否影响到词法分析器的功能。

同样地，这个测试框架是由一个宏来实现的，原因是使用宏可以获取当前出错代码所在文件的文件名以及行号，便于定位错误代码：

```
/*
 * Simple Test Framework
 */
#define EXPECT_EQ_BASE(equality,expect,actual,format) \
do { \
    test_count++; \
    if (equality){ \
        test_pass++; \
    }else{ \
        main_ret = 1; \
        fprintf(stderr,"%s:%d: expect: " format " actual: " format "\n",__FILE__,__LINE__,expect,actual);\
    } \
}while (0)

#define EXPECT_EQ_INT(expect, actual) EXPECT_EQ_BASE((expect) == (actual), expect, actual, "%d")
#define EXPECT_EQ_STRING(expect, actual) \
    EXPECT_EQ_BASE(actual.compare(expect) == 0, expect, actual.c_str(), "%s")
#define EXPECT_EQ_SIZE(expect, actual) EXPECT_EQ_BASE((expect) == (actual), expect, actual, "%zu")
```

图 11 测试框架代码

测试用例主要分为 7 个部分，分别测试了以下功能：

1. 运算符的解析
2. 数字的解析
3. 标识符的解析
4. 注释的解析
5. 常量的解析(包括字符常量与字符串常量的识别)
6. 错误的解析
7. 警告的解析

具体测试数据详见附录四。

以下是该测试运行的一个例子，此时该程序刚对部分代码做了一些改动，运行测试得到如下结果：

```
Time:10:15:29      User:james      Dir:/mnt/hgfs/For-Linux/LambCompiler/bin
$./lexer
src/test.cpp:142: expect: hello, \"world actual: hello, \"world";
70/71 (98.59%) passed
```

图 12 测试结果 1

可以看到 test.cpp 中 142 行的测试用例出现了问题，该程序期望输出 hello,\"world，但是实际却输出了 hello,\"world";，多了一个双引号与分号。根据这条出错信息，直接快速定位到解析字符串的代码，修正 bug。

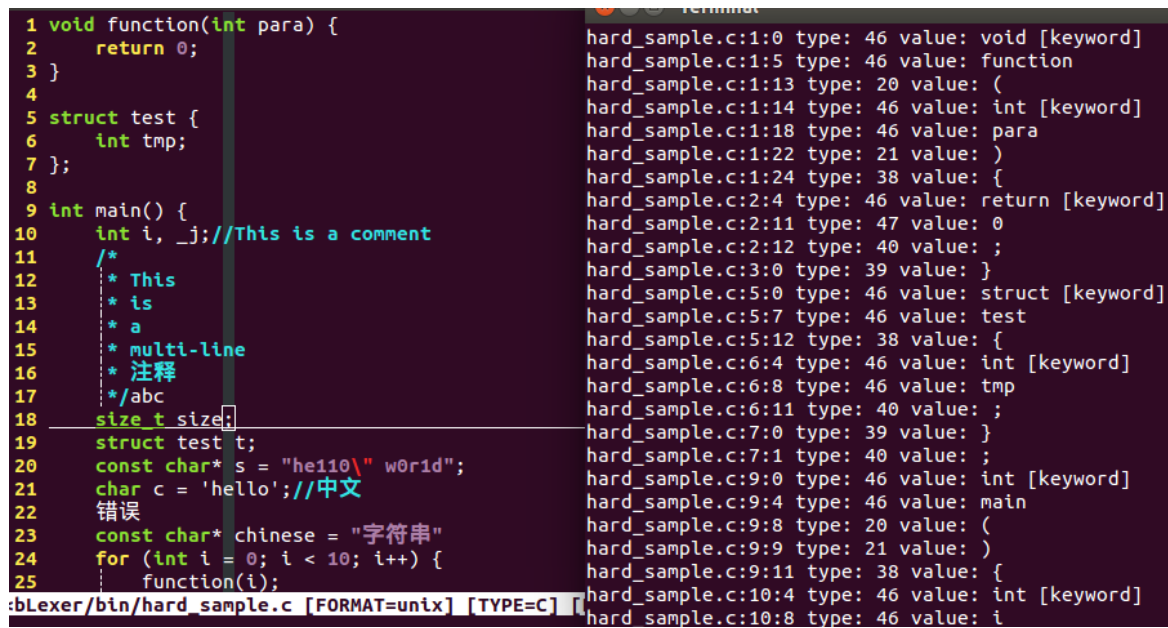
若程序通过了所有测试用例，则会显示如下结果：

```
Time:10:19:12      User:james      Dir:/mnt/hgfs/For-Linux/LambCompiler/bin
$./lexer
71/71 (100.00%) passed
```

图 13 测试结果 2

#### 4. 运行结果

当关闭 TEST\_ON 宏时，该词法分析器将从外部读取文件，并可以将分析的结果输出，示例输出如下所示(左边是待分析文件 hard\_sample.c，右边是分析结果)：



```
1 void function(int para) {
2     return 0;
3 }
4
5 struct test {
6     int tmp;
7 };
8
9 int main() {
10     int i, _j; // This is a comment
11     /*
12      * This
13      * is
14      * a
15      * multi-line
16      * 注释
17      */
18     size_t size;
19     struct test t;
20     const char* s = "he110\" w0rld";
21     char c = 'hello'; // 中文
22     错误
23     const char* chinese = "字符串"
24     for (int i = 0; i < 10; i++) {
25         function(i);
26     }
27 }
```

```
hard_sample.c:1:0 type: 46 value: void [keyword]
hard_sample.c:1:5 type: 46 value: function
hard_sample.c:1:13 type: 20 value: (
hard_sample.c:1:14 type: 46 value: int [keyword]
hard_sample.c:1:18 type: 46 value: para
hard_sample.c:1:22 type: 21 value: )
hard_sample.c:1:24 type: 38 value: {
hard_sample.c:2:4 type: 46 value: return [keyword]
hard_sample.c:2:11 type: 47 value: 0
hard_sample.c:2:12 type: 40 value: ;
hard_sample.c:3:0 type: 39 value: }
hard_sample.c:5:0 type: 46 value: struct [keyword]
hard_sample.c:5:7 type: 46 value: test
hard_sample.c:5:12 type: 38 value: {
hard_sample.c:6:4 type: 46 value: int [keyword]
hard_sample.c:6:8 type: 46 value: tmp
hard_sample.c:6:11 type: 40 value: ;
hard_sample.c:7:0 type: 39 value: }
hard_sample.c:7:1 type: 40 value: ;
hard_sample.c:9:0 type: 46 value: int [keyword]
hard_sample.c:9:4 type: 46 value: main
hard_sample.c:9:8 type: 20 value: (
hard_sample.c:9:9 type: 21 value: )
hard_sample.c:9:11 type: 38 value: {
hard_sample.c:10:4 type: 46 value: int [keyword]
hard_sample.c:10:8 type: 46 value: i
```

图 14 运行结果

## 四、实验总结

### 1) 心得

该词法分析器是对编译原理的一次实践，运用了自动机理论来将源文件切割为词法单元流。通过对 GCC 1.21 源代码的参考，为该词法分析器设计的符号表实质上是一个完整编译器的符号表，具有良好的可拓展性。在 main.cpp 文件中，只有一个 main 函数，其函数体如下所示：

```
int main(int argc, char** argv) {  
    return lex_main(argc, argv);  
}
```

图 15 词法分析器的 main 函数

从中可以看到 main 函数的实现细节全都放在了 lex\_main 函数中，lex\_main 函数作为该词法分析器对外的接口，如果直接被 main 函数调用，则该程序是一个独立可运行的词法分析器；如果该函数被语法分析器调用，则该词法分析器是作为编译器整体的一个部件，编译器可以直接获取该词法分析器所生成的词法单元流。

在编写过程中，也进行了多次重构，很多重复的地方都采用了宏定义来取代。宏定义的好处很多，比如修改更方便，比如可以通过相应的预编译指令在 config 文件中对程序进行相应的配置，具体在附录二中的项目配置小节中有详细的阐述。

### 2) 缺陷

在开发过程中，由于时间有限，很多地方直接使用了 STL 容器，而实际上，STL 容器的适用性是有限的。比如在这个词法分析器中有一个关键字表，需要使用字符串作为键值存取，我为了达到  $O(1)$  的存取，采用了 C++11 的标准哈希表 unordered\_map，但是该哈希表在这个场景下并不适用。因为我只定义了一个 C 语言关键字的子集，共 35 个关键字，然而 unordered\_map 初始化时，会初始化上万甚至上十万个对象。使用工具检测后发现该程序运行时申请了 15MB 的内存，其主要负载都在这个哈希表中。所以，为了效率，应当自行选取哈希函数，维护一个相对较小的哈希表。这样一来可以提高程序的效率，减少内存占用。

### 3) 扩展

该词法分析器只是一个完整编译器的一小部分。由于考虑到该编译器生成代码需要具有跨平台性，直接生成对应机器的目标代码过于繁琐，故该编译器生成的目标代码将运行在自行设计的虚拟机上。该虚拟机可以具有其他高级语言虚拟机所具有的特性，例如垃圾回收机制，使 C 语言的使用者可以无须考虑内存释放的问题。该虚拟机的设计及实现将在本学期内完成。



附录一 c\_ttype 表

Type	Value	Type	Value
C_EQ	=	C_MULT_EQ	*=
C_NOT	!	C_DIV_EQ	/=
C_GREATER	>	C_MOD_EQ	%=
C_LESS	<	C_AND_EQ	&=
C_PLUS	+	C_OR_EQ	=
C_MINUS	-	C_XOR_EQ	^=
C_MULT	*	C_RSHIFT_EQ	>>=
C_DIV	/	C_LSHIFT_EQ	<<=
C_MOD	%	C_OPEN_SQUARE	[
C_AND	&	C_CLOSE_SQUARE	]
C_OR		C_OPEN_BRACE	{
C_XOR	^	C_CLOSE_BRACE	}
C_RSHIFT	>>	C_SEMICOLON	;
C_LSHIFT	<<	C_PLUS_PLUS	++
C_COMPL	~	C_MINUS_MINUS	--
C_AND_AND	&&	C_DEREF	->
C_OR_OR		C_DOT	.
C_QUERY	?	C_SCOPE	::
C_COLON	:	C_NAME	SPELL_INDENT
C_COMMA	,	C_NUMBER	SPELL_NUMBER
C_OPEN_PAREN	(	C_CHAR	SPELL_STRING
C_CLOSE_PAREN	)	C_OTHER	SPELL_CHAR
C_EQ_EQ	==	C_STRING	SPELL_STRING
C_NOT_EQ	!=	C_HEADER_NAME	SPELL_STRING
C_GREATER_EQ	>=	C_COMMENT	SPELL_STRING
C_LESS_EQ	<=	C_PADDING	SPELL_NONE
C_PLUS_EQ	+=	C_EOF	SPELL_NONE
C_MINUS_EQ	-=		

图 16 词法分析器支持的符号种类

## 附录二 使用说明

### 1. 使用介绍

该词法分析器接受三种参数：

- 1) -v  
-v 参数显示词法分析器当前的版本，该版本号为 0.0.1。
- 2) -h  
-h 参数显示词法分析的帮助菜单。
- 3) 文件名  
输入文件名可以对该文件进行词法分析。

### 2. 项目结构与配置

该项目主要通过 `lex_config.h` 文件中的宏来进行配置。

#### 1) 项目结构

该项目的结构如下图所示：

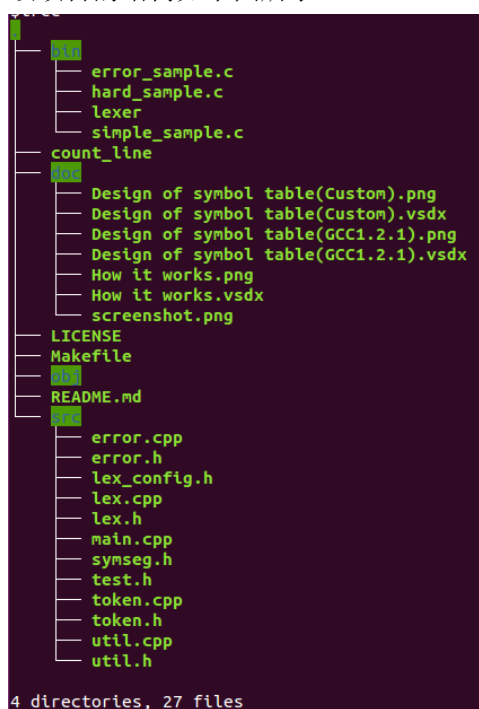


图 17 项目结构

其中：

- bin 目录存放生成的可执行二进制文件以及相应的测试数据
- doc 目录存放该项目对应的文档
- obj 目录存放生成的目标文件
- src 目录存放源文件
- Makefile 用于 GNU GCC
- LICENSE 采用 MIT LICENSE
- count\_line 是一个 shell 脚本，用于统计当前目录下 `cpp` 文件与 `h` 文件的代码行数

## 2) 项目配置

该项目主要通过 `lex_config.h` 文件来进行配置，该文件中共定义了 5 个宏。

- `#define TEST_ON`: 该宏可以控制词法分析器的测试行为。当该宏被定义时，该词法分析的运行无需参数，词法分析器将会执行测试，并将测试结果输出；当该宏未被定义时，该词法分析器不会执行测试，其运行需要相应参数，分析文件将从外部输入。
- `#define DUMP_TOKEN_STREAM`: 该宏可以控制是否将词法分析的结果 `dump` 到屏幕上。之所以设置这个控制宏，是因为在将来的语法分析阶段，很可能不再需要查看词法分析的结果，那么就可以通过对该宏的设置来关闭词法分析结果的输出。
- `#define PRINT_LOG`: 该宏用于在 `debug` 阶段控制是否输出自定义调试信息。在该项目的调试过程中，所有自定义调试信息均通过 `PRINT()` 宏来输出，若不希望输出调试信息，则直接关闭该宏即可。
- `#define SHOW_KEYWORD_FILTER`: 该宏可以控制是否进行关键字分析。本词法分析器在词法分析阶段默认识别关键字，若不希望在此阶段就识别关键字，则直接关闭该宏即可。
- `#define LONGEST_IDENTIFIER_LENGTH 32`: 该宏可以调整该词法中标识符的最大长度，默认为 32 个字节。

## 3) Makefile

该项目的主要 Makefile 内容如下：

```
1 # LamblExer
2 # by James 2017/04/06
3
4 CXX = g++
5 CXXFLAGS = -g -Wall -O2 -std=c++11
6 BIN = bin
7 SRC = src
8 OBJ = obj
9 EXE = $(BIN)/lexer
10
11 LEX_CONFIG_H = $(SRC)/lex_config.h
12 LEX_H = $(SRC)/lex.h
13 UTIL_H = $(SRC)/util.h
14 TOKEN_H = $(SRC)/token.h
15 TEST_H = $(SRC)/test.h
16 ERROR_H = $(SRC)/error.h
17 SYMSEG_H = $(SRC)/symseg.h
18
19 LEX_CPP = $(SRC)/lex.cpp
20 UTIL_CPP = $(SRC)/util.cpp
21 TOKEN_CPP = $(SRC)/token.cpp
22 MAIN_CPP = $(SRC)/main.cpp
23 ERROR_CPP = $(SRC)/error.cpp
24
25 OBJS = lex.o token.o error.o util.o main.o
26
27 all: $(OBJS)
28     $(CXX) $(CXXFLAGS) -o $(EXE) $(OBJ)/lex.o $(OBJ)/token.o $(OBJ)/util.o $(OBJ)/error.o $(OBJ)/main.o
29 main.o:
30     $(CXX) $(CXXFLAGS) -o $(OBJ)/main.o -c $(MAIN_CPP)
31 lex.o:
32     $(CXX) $(CXXFLAGS) -o $(OBJ)/lex.o -c $(LEX_CPP)
33 util.o:
34     $(CXX) $(CXXFLAGS) -o $(OBJ)/util.o -c $(UTIL_CPP)
35 token.o:
36     $(CXX) $(CXXFLAGS) -o $(OBJ)/token.o -c $(TOKEN_CPP)
37 error.o:
38     $(CXX) $(CXXFLAGS) -o $(OBJ)/error.o -c $(ERROR_CPP)
```

/mnt/hgfs/For-Linux/LamblExer/Makefile [FORMAT=unix] [TYPE=MAKE] [POS=1,1][2%] 11/04/17 - 21:13

图 18 Makefile 内容

附录三 关键字列表

关键字	关键字
static	char
unsigned	float
long	double
const	void
extern	enum
register	struct
typedef	union
short	if
inline	else
volatile	while
signed	do
auto	for
int	switch
continue	case
return	default
goto	break
sizeof	

图 19 支持的关键字列表

## 附录四 测试数据

### 1. 综述

测试用例主要分为 7 个部分，分别测试了以下功能：

- [1] 运算符的解析
- [2] 数字的解析
- [3] 标识符的解析
- [4] 注释的解析
- [5] 常量的解析(包括字符常量与字符串常量的识别)
- [6] 错误的解析
- [7] 警告的解析

### 2. 测试数据

#### 2.1 运算符的解析

该词法分析一共可以识别 46 种运算符，在测试时，将这些运算符都按照一定顺序放在一起形成一个字符串，若该词法分析器能将这些运算符从中解析出来，则认为通过了测试。具体测试数据如下所示：

```
/* Test operator */
const std::vector<std::string> test_operator = {
    "      =!><+~* /%&|^>><<~&&| |?;,      ",
    "      ()==!=>=<=+=-=-*= /=%=&|=      ",
    "      ^=>>=<=[ ]{}; ++--> . : :      ",
};
```

图 20 运算符的测试数据

#### 2.2 数字的解析

该词法分析一共可以解析 4 种类型的数字，包括十进制整数、八进制整数、十六进制整数以及十进制实数。在测试时，为每种类型的数字创建一个合法实例，若该词法分析器能将这些数字解析出来，则认为通过了测试。具体测试数据如下所示：

```
/* Test number */
const std::vector<std::string> test_number = {
    "      int a = 123;      ",
    "      int b = 123.456;  ",
    "      int c = 0xff00;    ",
    "      int d = 0123;      ",
};
```

图 21 数字的测试数据

#### 2.3 标识符的解析

标识符主要由数字、字母以及下划线组成，但只能由下划线或字母作为首字母。在测试时，为标识符创建三个合法实例，包括字符开头与下划线开头的标识符以及包含字母、数字和下划线的标识符。若该词法分析器能将这些数字解析出来，则认为通过了测试。具体测试数据如下所示：

```
/* Test identifier */
const std::vector<std::string> test_identifier = {
    "      int identifier;      ",
    "      int _identifier;      ",
    "      int _identifier_1_plus;      ",
};
```

图 22 标识符的测试数据

## 2.4 注释的解析

注释主要包括单行注释与多行注释，其规范与 C 语言相同：单行注释以//开头，多行注释以/\*作为开头，以\*/作为结束。该测试中为注释创建两个合法实例，一个代表单行注释，一个代表多行注释。若该词法分析器成功忽略这些注释，则认为通过了测试。具体测试数据如下所示：

```
/* Test comment */
const std::vector<std::string> test_comment = {
    " //This is a single line comment ",
    " /* This is a ",
    " * multi-line ",
    " * 注释 ",
    " */ "
};
```

图 23 注释的测试数据

## 2.5 常量的解析

常量主要包括字符常量与字符串常量，字符常量以非转义的单引号开头并以非转义的单引号结尾，字符串常量以非转义的双引号开头并以非转义的双引号结尾。该测试中为常量创建两个合法实例，分别代表字符常量与字符串常量。若该词法分析器成功解析这些常量，则认为通过了测试。具体测试数据如下所示：

```
/* Test literal */
const std::vector<std::string> test_literal = {
    " string s = \"hello, \\\"world\"; ",
    " char c = 'b'; "
};
```

图 24 常量的测试数据

## 2.6 错误的解析

该词法分析器在词法分析阶段已经能找出一些错误，包括以下 5 个：

- [1] 数字中出现多于一个小数点
- [2] 在八进制数中出现 8 或 9
- [3] 字符串常量中找不到结尾的双引号
- [4] 字符常量中找不到结尾的单引号
- [5] 无法识别的类型，通常为中文

在测试中，为这 5 个错误构造了相应的错误实例。若该词法分析器成功分析出这些错误，则认为通过了测试。具体测试数据如下所示：

```
/* Test error */
const std::vector<std::string> test_error = {
    " int b = 123.456.789; ", /* too many decimal points in number */
    " int c = 09; ", /* invalid digit in octal constant */
    " string s = \"hello; ", /* missing terminating \" character */
    " char cc = 'h; ", /* missing terminating \' character */
    " 始 ", /* unknown type name */
};
```

图 25 错误的测试数据

## 2.7 警告的解析

该词法分析器在词法分析阶段已经能找出一个警告，主要是字符常量溢出的问题：字符常量的两个单引号之间只能有一个字符，若超过一个，则认为是字符常量溢出，词法分析器只取第一个字符并给出溢出警告。若该词法分析器成功给出了警告，则认为通过了测试。具体测试数据如下所示：

```
/* Test warning */
const std::vector<std::string> test_warning = {
    " cc = 'hello world'; ", /* character constant too long for its type */
};
```

图 26 警告的测试数据