编译原理实验五：编译器自动生成工具

原创

[裕东方](#) 于 2019-04-12 22:11:54 发布 5112 收藏 23
分类专栏： 编译原理实验 文章标签： 编译原理
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/yyd19981117/article/details/89259964
版权

编译原理实验 专栏收录该内容
7 篇文章 9 订阅
订阅专栏

一、词法分析程序自动生成工具的使用（4小时）

实验目的

学习使用词法分析自动工具LEX。

实验任务

使用LEX工具实现编译器的词法分析程序。

实验内容

(a) 学习文档“LEX的用法.pdf”。

(b) 准备一个LEX工具，如这里提供的“FLEX251.ZIP”，可上网搜索下载更新的版本。

(c) 以文档中提供的4个输入文件为例，测试LEX工具。有些版本的FLEX需要在辅助程序部分增加yywrap()函数：
int yywrap() {return 1;}

1. 生成LEX版本的TINY词法分析器，与其它部分组合成一个完整的TINY语言编译器，并完成测试验证。（参见tiny编译器的使用.ppt）
2. 编写某语言（如：C-语言）的词法描述文件，生成其词法分析器，与其它部分组合成一个完整的TINY语言编译器，并完成测试验证。（提示：可利用增量编程，修改TINY语言的词法描述文件tiny.l，为C-语言编写词法描述文件。）

（二）语法分析程序自动生成工具的使用（4小时）

学习使用语法分析程序自动生成工具YACC；使用YACC工具实现编译器的词法分析程序。

实验内容

(a) 学习文档“YACC的用法.pdf”。

(b) 准备一个YACC工具，如这里提供的“bison.zip”，可上网搜索下载更新的版本。（有源程序可供参考）

(c) 以文档中提供的输入文件为例，测试YACC工具。需要将两个文件拷贝到特殊目录，详情请阅readme.txt。

1. 生成YACC版本的TINY语法分析器，与其它部分组合成一个完整的TINY语言编译器，并完成测试验证。（提示：全局头文件GLOBALS.H需要替换为YACC目录下的那个。）

（3）编写某语言（如：C-语言）的语法描述文件，生成其语法分析器，与其它部分组合成一个完整的TINY语言编译器，并完成测试验证。（提示：可利用增量编程，修改TINY语言的语法描述文件tiny.y，为C-语言编写语法描述文件，全局头文件GLOBALS.H在替换为YACC目录下的那个后需相应修改。）

实验工具及源代码下载

https://pan.baidu.com/s/14nbZ3Xu5nsaGCRPcycJUHw

密码：msry

实验原理讲解

1、LEX工具的使用

（1）以文档中提供的4个输入文件为例，测试LEX工具。

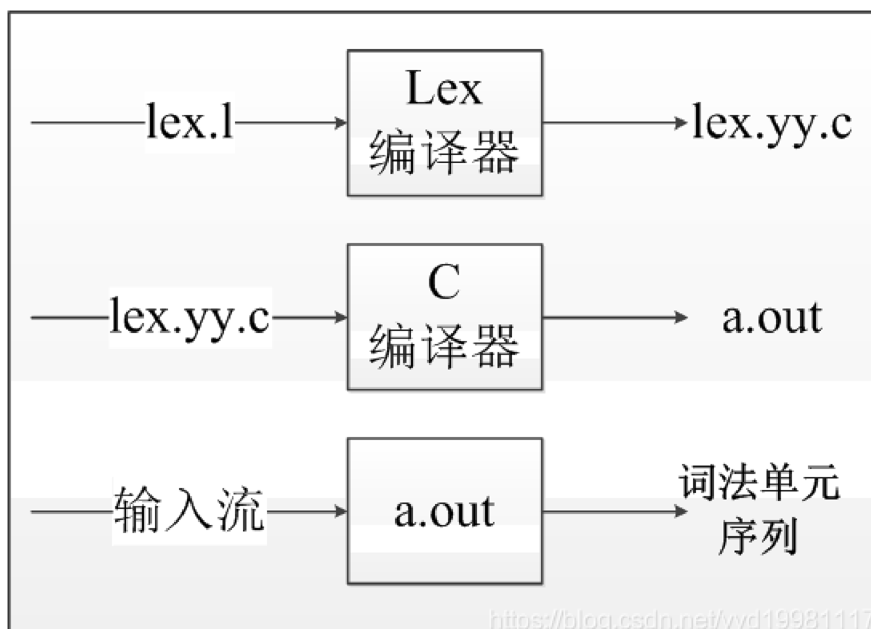
步骤：

1.将flex.exe复制到tiny.l的目录中

2.输入命令“FLEX tiny.l”生成lex.yy.c

3.将main.c lex.yy.c util.c globals.h util.h scan.h放入一个工程中，编译生成.exe文件

FLEX语言的结构如下：



https://blog.csdn.net/yyd19981117

其中，FLEX文件是使用一种特殊的语言——LEX，编写的，它描述了一种语言的词法结构：其中，第一部分为定义，FLEX工具在处理这一部分的时候，会将definitions部分的所有代码原封不动的插入到lex.yy.c（词法分析代码）中，我们可以在这一部分中写程序需要用到的头文件等内容，通常情况下，这些代码被放在放在“%{”和“%}”之内，直接插入lex.yy.c。

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

https://blog.csdn.net/yyd19981117

第二部分是词法规则部分：格式为【模式 { 动作 }】，其中模式为正则表达式，动作是代码片段。当匹配相对应的正则表达式时，这些代码片段就会被执行。

```
/****************************************************/
/* File: tiny.l                                     */
/* Lex specification for TINY                       */
/* Compiler Construction: Principles and Practice   */
/* Kenneth C. Louden                                */
/****************************************************/

%{
#include "globals.h"
#include "util.h"
#include "scan.h"
/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];
%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+

%%

"if"            {return IF;}
"then"          {return THEN;}
"else"          {return ELSE;}
"end"           {return END;}
"repeat"        {return REPEAT;}
"until"         {return UNTIL;}
"read"          {return READ;}
```

https://blog.csdn.net/yyd19981117

表达式	匹配	例子
c	单个非运算符字符 c	a
\c	字符 c 的字面值	*
"s"	串 s 的字面值	"**"
.	除换行符以外的任何字符	a.*b
^	一行的开始	^abc
$	行的结尾	abc$
[s]	字符串 s 中的任何一个字符	[abc]
[^s]	不在串 s 中的任何一个字符	[^abc]
r*	和 r 匹配的零个或多个串连接成的串	a*
r+	和 r 匹配的一个或多个串连接成的串	a+
r?	零个或一个 r	a?
$r\{m,n\}$	最少 m 个，最多 n 个 r 的重复出现	a{1,5}
r_1r_2	r_1 后加上 r_2	ab
$r_1 \mid r_2$	r_1 或 r_2	a|b
(r)	与 r 相同	(a|b)
r_1/r_2	后面跟有 r_2 时的 r_1	abc/123

图 3-8 Lex 的正则表达式

最后一个部分包括着一些C代码，这些函数被称为辅助函数，它们用于在第2个部分被调用且不在任何地方被定义的辅助程序。如果要将Lex输出作为独立程序来编译，则这一部分还会有一个主程序。LEX对此部份不作任何处理，仅仅将之直接拷贝到输出文件lex.yy.c的尾部。在些部份，可定义对模式进行处理的C语言函数、主函数和yylex要调用的函数yywrap()等。如果用户在其它C模块中提供这些函数，用户代码部份可以省略。

```
%%

TokenType getToken(void)
{ static int firstTime = TRUE;
  TokenType currentToken;
  if (firstTime)
  { firstTime = FALSE;
    lineno++;
    yyin = source;
    yyout = listing;
  }
  currentToken = yylex();
  strncpy(tokenString,yytext,MAXTOKENLEN);
  if (TraceScan) {
    fprintf(listing,"\t%d: ",lineno);
    printToken(currentToken,tokenString);
  }
  return currentToken;
}
```

https://blog.csdn.net/yyd19981117

通过执行以上步骤，可以顺利生成对应词法分析代码lex.yy.c。如图是文件夹内部的情况。如果需要正式编译程序进行完整的词法分析，还需要加上主函数部分。

名称

- Debug
- flex
- lex.yy
- lex.yy.dsp
- lex.yy.dsw
- lex.yy
- lex.yy.ncb
- lex.yy.opt
- lex.yy.plg
- test1.L

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
%{
/* a Lex program that adds line numbers
to lines of text, printing the new text
to the standard output
*/
#include<stdio.h>
int lineno=1;
%}
line *.\n
%%
"line" {printf("%d %s",lineno++,yytext);}
%%
int yywrap()
  {
    return 1;
  }

main()
{
  yylex(); return 0;
}
```

https://blog.csdn.net/yyd19981117

（2）使用增量编程，将tiny词法分析器改为C-词法分析器。

这一部分的代码其实在上面的压缩包里面有。。。。

C-的词法规则结构要比TINY复杂，同时，TINY词法中也含有一些C-没有的关键字，比如repeat这些。由于C-的定义与tiny有所区别，只需要根据C-的规则，对TINY.L进行增删操作即可。

以下是改进后的C-.L源程序：

```
%{
#include "globals.h"
#include "util.h"
#include "scan.h"
/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];
%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+
newline     \n
whitespace  [ \t]+

%%

"if"            {return IF;}
"while"          {return WHILE;}
"else"          {return ELSE;}
"return"         {return RETURN;}
"int"          {return INT;}
"void"         {return VOID;}
"="            {return ASSIGN;}
"=="             {return EQ;}
"<"             {return LT;}
"<="             {return LTEQ;}
```

```
">"             {return RT;}
">="             {return RTEQ;}
"!="             {return UNEQ;}
"+"             {return PLUS;}
"-"             {return MINUS;}
"*"             {return TIMES;}
","             {return NOB;}
"/"             {return OVER;}
"("             {return LPAREN;}
")"             {return RPAREN;}
";"             {return SEMI;}
"["             {return MLPAREN;}
"]"             {return MRPAREN;}
"{"             {return LLPAREN;}
"}"             {return LRPAREN;}
{number}        {return NUM;}
{identifier}    {return ID;}
{newline}       {lineno++;}
{whitespace}    {/* skip whitespace */}
"/*"            {     char c;
                      char d='f';
                      do
                      {     c = input();
                            if (c == EOF) break;
                            if (c == '\n') lineno++;
                            if (c == '*'){
                                c = input();
                                if (c == '/') d = 'a';
                            }
                      } while (d == 'f');
                }
.               {return ERROR;}

%%
TokenType getToken(void)
{    static int firstTime = TRUE;
     TokenType currentToken;
     if (firstTime)
     {     firstTime = FALSE;
           lineno++;
           yyin = source;
           yyout = listing;
     }
     currentToken = yylex();
     strncpy(tokenString,yytext,MAXTOKENLEN);
     if (TraceScan) {
        fprintf(listing,"\t%d: ",lineno);
        printToken(currentToken,tokenString);
     }
     return currentToken;
}
```

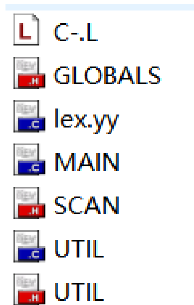
首先是第一部分：宏定义和词法正则表达式描述，这些根据文档中的定义就可以完成，且C-的词法分析器与TINY一样，需要包含globals.h中的宏定义、SCAN.h中的扫描文件方法，这些头文件都需要包含进去。

第二部分是匹配规则部分：这里将TINY的关键字全部修改为C-语言的关键字，同时结合实验二中所做的词法分析程序，将字符的返回标记全部更改好。通过增量编程的思想，在TINY.L中对相关语句进行修改即可。

第三部分是用户自定义函数。这一部分与TINY没有差别。

最后，我们还需要对TINY编译器中的其他组件——utils.c、globals.h等文件进行修改，使其符合C-语言的关键字标准。主要是将utils.c中的printtoken函数，更改为适应C-词法分析的关键字代码即可，做出符合增量编程的一些调整，就可以组合成完整的C-词法分析器。

最终的C-词法分析器包含以下这些组件：



2、YACC工具的使用

（1）以文档中提供的输入文件为例，测试YACC工具。

步骤：（与LEX词法分析类似）

1、将语法分析YACC工具BISON和语法描述文件.Y放入同一个文件夹内。

2、命令行中输入“bison .y类型的文件名”就可以生成语法分析代码.Y.tab.C。

3、将生成的语法分析代码与其他部件组合在一起，就可以生成完整的语法分析器。

类似于词法分析文件lex，语法描述语言YACC文件的格式也由三部分组成：定义、规则、函数。

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

首先是定义部分：定义部分包括Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息。它还包括了必须在它的开始时直接进入输出文件的任何C代码(主要是其他源代码文件的#include指示)。说明文件的这个部分可以是空的。

```
%{
#define YYPARSER /* distinguishes Yacc output from other code files */

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

#define YYSTYPE TreeNode *
static char * savedName; /* for use in assignments */
static int savedLineNo;  /* ditto */
static TreeNode * savedTree; /* stores syntax tree for later return */

%}
```

https://blog.csdn.net/yyd19981117

规则部分包括修改的BNF格式中的文法规则以及将在识别出相关的文法规则时被执行的C代码中的动作。文法规则中使用的元符号惯例如下：通常，竖线被用作替换(也可分别写出替换项)。

```
stmt        : if_stmt { $$ = $1; }
            | repeat_stmt { $$ = $1; }
            | assign_stmt { $$ = $1; }
            | read_stmt { $$ = $1; }
            | write_stmt { $$ = $1; }
            | error  { $$ = NULL; }
            ;
if_stmt     : IF exp THEN stmt_seq END
                { $$ = newStmtNode(IfK);
                  $$->child[0] = $2;
                  $$->child[1] = $4;
                }
            | IF exp THEN stmt_seq ELSE stmt_seq END
                { $$ = newStmtNode(IfK);
                  $$->child[0] = $2;
                  $$->child[1] = $4;
                  $$->child[2] = $6;
                }
```

https://blog.csdn.net/yyd19981117

最后是用户自定义函数部分，辅助程序部分包括了过程和函数声明，这个部分也可为空。此部分与词法分析中无太大差别。

```
/* yylex calls getToken to make Yacc/Bison output
 * compatible with ealier versions of the TINY scanner
 */
static int yylex(void)
{ return getToken(); }

TreeNode * parse(void)
{ yyparse();
  return savedTree;
}
```

（2）编写某语言（如：C-语言）的语法描述文件，生成其语法分析器。

由于C-的语法与TINY十分不同，根据实验指导书上C-的29条语法规则，参照TINY.Y中的格式，进行编写：

```
%start program
%%

program
	: declaration_list { exeNode($1, 0); freeNode($1); }
	;

declaration_list
	: declaration_list declaration { $$ = opr(';', 2, $1, $2); }
	| declaration { $$ = $1; }
	;

declaration
	: var_declaration
		{
			$$  = opr(GLOBAL_VAR, 1, $1);
		}
	| fun_declaration { $$ = $1; }
	;

var_declaration
	: type_specifier IDENTIFIER ';'
		{
			Node *tmp1;
			/* here to insert new var declaration */
			tmp1 = set_index($2);
			$$ = opr(VAR, 2, $1, tmp1);
		}
	| type_specifier IDENTIFIER '[' CONSTANT ']' ';'
		{
			Node *tmp1, *tmp2;
			tmp1 = set_index($2);
			tmp2 = set_content($4);
			$$ = opr(VAR, 3, $1, tmp1, tmp2);
		}
	;

type_specifier
	: INT { $$ = set_content(INT); }
	| VOID { $$ = set_content(VOID); }
	;

fun_declaration
	: type_specifier IDENTIFIER '(' params ')'
		{
			Node *tmp1;
			tmp1 = set_index($2);
			$$ = opr(FUNC, 3, $1, tmp1, $4);
		}

	| compound_stmt
		{ $$ = opr(FUNC, 1, $1); }
	;
```

https://blog.csdn.net/yyd19981117

上图展示的是C-语法分析文件的其中一部分，完整代码太长，没办法全部贴出，具体实现部分在文件夹的【Cminus增量语法】文件夹下。

实现完上面的C-语法描述文件后，我们通过更改一系列TINY语言中使用过的辅助文件如utils.c、globals.h等，让它们适应C-的语法特点（主要是关键字的实现有差别）之后生成一个工程，即可完成C-的语法分析器构造。

上面这个代码，压缩包里面也有，下载一下吧，emmm。。。

3、生成完整编译器

这是这个实验指导书上规定的最后一步，但是我很不理解，为什么到这里只实现了词法和语法的时候，就要去要求，能生成整个编译器呢？？

明显，是实验的安排出现了问题。

整体的编译器还需要语义分析、代码生成，甚至如果需要一个能够执行C-代码的虚拟机环境！！这个地方当时助教验收，很多人都是懵的，如果之后这个实验还有这个需求，那我也没有办法，直接用TINY语言的去验收吧。

【下面展示的是有源代码的TINY编译器的生成过程】

使用tiny编译器

- 1.运行tiny编译器（tiny.exe）
  - 在命令行键入：**TINY SAMPLE**

```
C:\WINDOWS\system32\cmd.exe
TINY COMPILATION: sample.tny
Building Symbol Table...
Symbol table:
Variable Name  Location   Line Numbers
-------------  --------   ------------
x              0            5    6    9   10   10   11
fact           1            7    9    9   12
Checking Types...
Type Checking Finished
D:\??\2010??\??\TINY>
```

TINY目录中生成可以在TM虚拟机上运行的目标代码文件sample.TM

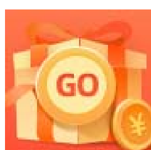
https://blog.csdn.net/wyd1098f117

使用tiny编译器

- 2.在目标机(TM.exe)上运行目标代码
  - 在命令行键入：**TM SAMPLE**，**TM**虚拟机开始运行如下图所示：

```
C:\WINDOWS\system32\cmd.exe - tm sample
TINY COMPILATION: sample.tny
Building Symbol Table...
Symbol table:
Variable Name  Location   Line Numbers
-------------  --------   ------------
x              0            5    6    9   10   10   11
fact           1            7    9    9   12
Checking Types...
Type Checking Finished
D:\??\2010??\??\TINY>tm sample
TM  simulation (enter h for help)...
Enter command:
```

https://blog.csdn.net/wyd1098f117



[创作打卡挑战赛](#)
[赢取流量/现金/CSDN周边激励大奖](#)