1、实验目的与要求

一、实验目的：

①、理解 TINY 语言的词法及词法分析器的实现。

②、掌握词法分析器的设计与实现方法。

③、深入理解最长匹配原则，确保词法分析的准确性。

④、学会处理各种词法错误，保证词法分析的健壮性。

⑤、培养编程能力，完成对 TINY+ 语言的词法分析器的编写和测试。

二、实验要求：

①、完成 TINY+词法分析程序的编写及测试（使用提供的测试代码或自己编写测试文件）；

2、实验内容

- TINY 语言的词法由 TINY Syntax.ppt 描述；
- TINY 语言的词法分析器由 TINY Scanner.rar 的 C 语言代码实现；
- TINY+语言的词法由 TINY+ Syntax.doc 描述。

任务：理解 TINY 语言的词法及词法分析器的实现，并基于该词法分析器实现拓展语言 TINY+的词法分析器。

要求：

（1）、TINY+词法分析器以 TINY+源代码为输入，输出为识别出的 token 序列；

（2）、词法分析器以最长匹配为原则，例如‘:=’应识别为赋值符号而非单独的‘:’及‘=’；

（3）、Token 以（种别码，属性值）表示，包含以下类型的种别码：

a) KEY 为关键字；

b) SYM 为系统特殊字符；

c) ID 为变量；

d) NUM 为数值常量；

e) STR 为字符串常量。

（4）、识别词法错误。词法分析器可以给出词法错误的行号并打印出对应的出错消息，主要包含以下类型的词法错误：

a) 非法字符。即不属于 TINY+字母表的字符，比如$就是一个非法字符；

b) 字符串匹配错误，比如右部引号丢失，如‘scanner

c) 注释的右部括号丢失或匹配错误，如 {this is an example

3、实验过程步骤及说明

一、理解词法分析器原理并构造 TINY+的 DFA

首先根据 TINY Syntax.ppt，我了解了 TINY 语言中保留的关键字、特殊符号等。再查阅 TINY+ Synatx.doc，我了解了 TINY+语言相比 TINY 语言新增的关键字、特殊符号等。其中 TINY 语言的扫描器对应的 DFA 如下图所示：

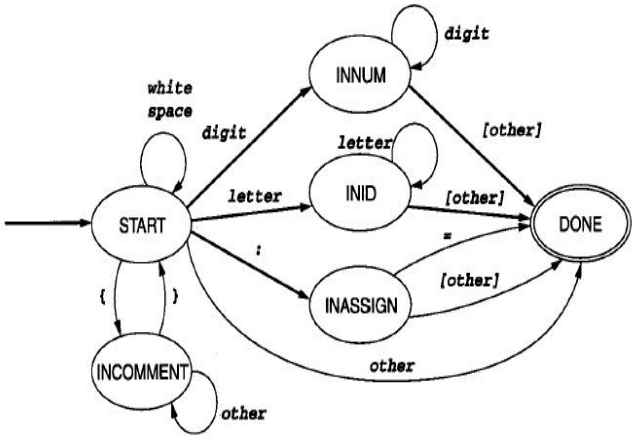

图 1 TINY 扫描器对应的 DFA

再查阅 TINY 的 C 语言源文件，可知各 C 文件的具体作用及词法分析器实现的流程，如下图所示。

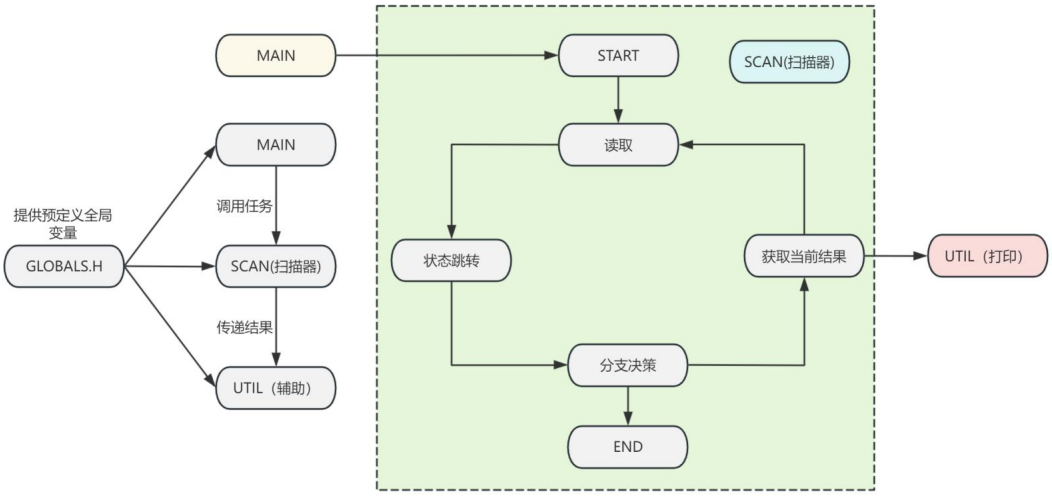

图 2 词法分析实现流程

globals.h 文件中定义了一些全局类型、变量以及一些标志。Main.c 文件是 TINY 编译器程序入口，负责调用编译器的各个模块完成编译任务，其中它首先检查命令行参数数量，确保只传入了一个参数（输入的源代码文件名），表示一次只能测试一个文件。

Scan.c 是实现 TINY 词法分析器的核心和关键，它逐个读取测试文件中的字符（token），并将其分类成不同的标记类型。Scan.c 实现了 TINY 扫描器的作用，它的实现逻辑对应图 1 中的 DFA。

Util.c 文件实现了一些辅助函数，用于支持 TINY 编译器的语法树构建和打印功能。其中 printToken 函数用于输出词法分析器识别的标记，便于调试和理解词法分析过程。

通过上述分析，要完成 TINY+词法分析器的设计，首先需要设计一个与 TINY+扫描器匹配的 DFA，其中需要注意新增的关键字和特殊字符等。

我设计的 DFA 如下图所示，**其符合最长匹配原则**，其中各个状态的含义如下：

INNUM：表示当前进行数字的判断

INASSIGN：表示进行赋值符号的判断，**用于区分：和:=**

INID：表示进行变量标识符的判断

INLE：表示进行<=（小于等于）符号的判断，**用于区分<和<=**

INME：表示进行>=（大于等于）符号的判断，**用于区分>和>=**

INUPDOX：表示进行字符串（‘’）的判断

INPOWR：表示进行**（幂运算）符号的判断，**用于区分*和****

INCOMMENT：表示进行注释内容（{}）的判断，**需要特别注意嵌套的判断**

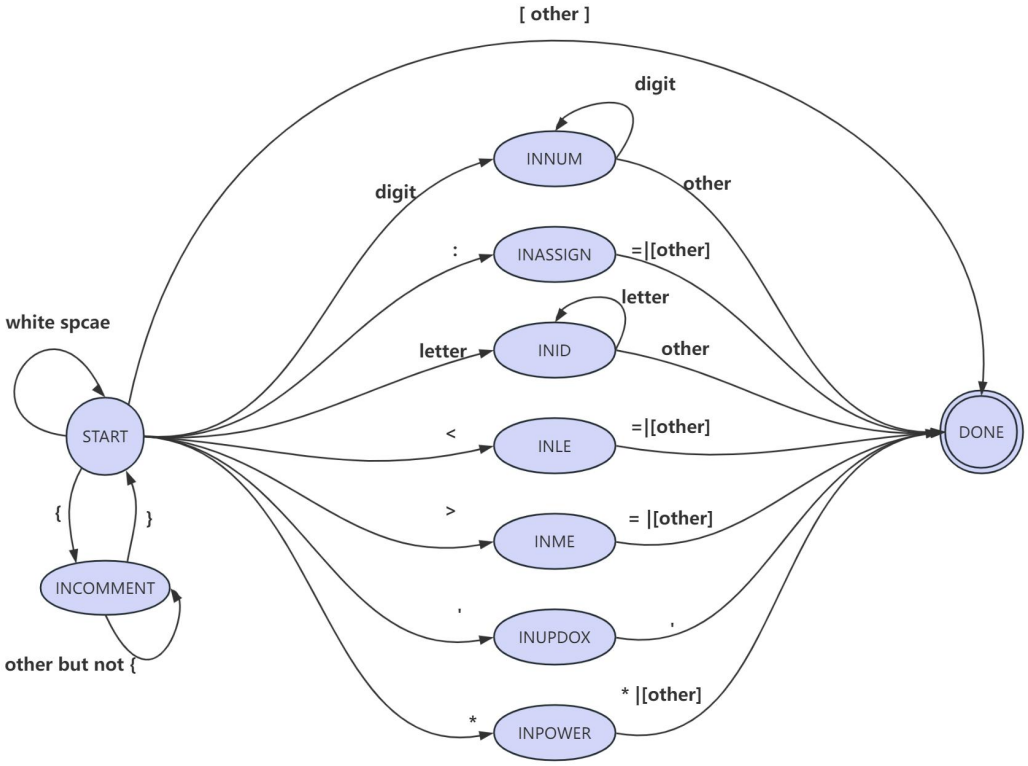

图 3 TINY+扫描器对应的 DFA

二、添加新增类型和 DFA 中新增状态。

1、在 globals.h 添加新增类型

首先在 globals.h 文件中添加 TINY+相比 TINY 新增的枚举类型，即在 globals 文件中对应的 tokenType（TINY+Synatx.doc 文件中获得），如下表所示。

枚举类型值	含义	代码	符号
TRUE1	布尔类型的真值(关键字)	true	
FALSE1	布尔类型的假值(关键字)	false	
OR	逻辑或运算符(关键字)		or
AND	逻辑与运算符(关键字)		and
NOT	逻辑非运算符(关键字)		not
INT	整数类型关键字	int	
BOOL1	布尔类型关键字		
STRING	字符串类型关键字		

表 1 新增 tokenType 及其含义（1）

枚举类型值	含义	代码	符号
FLOAT	单精度浮点数关键字	float	
DOUBLE	双精度浮点数关键字	double	
DO	循环语句关键字	do	
WHILE	循环语句关键字	while	
INCLUDE	包含文件关键字		
BREAK	中断循环关键字	break	
CONTINUE	继续循环关键字	continue	
STR	字符串常量符号		
MT	大于符号		>
ME	大于等于符号		>=
LE	小于等于符号		<=
COMMA	逗号		,
UPDOX	单引号		'
PERCENT	百分号		%

表 2 新增 tokenType 及其含义（2）

在代码中添加如下：

```
typedef enum
    /*  book-keeping tokens */
   {ENDFILE,ERROR,
     /* 关键字 */
    IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,
    /* 标识符和数字 */
    ID,NUM,

    /* 特殊字符 */
    ASSIGN,EQ,LT,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI,
    /* :=   =  <   +    -     *     /     (      )     ;    */
    /* 新增关键字 */
    TRUE1,FALSE1,OR,AND,NOT,INT,BOOL1,STRING,FLOAT,DOUBLE,DO,WHILE,INCLUDE,BREAK,CONTINUE,
    /* true,false,or,and,not,int,bool,string,float,double,do,while,include,break,continue */
    /* 新增变量符号，字符串*/
    STR,
    /* 新增特殊字符 */
    MT,ME,LE,COMMA,UPDOX,PERCENT,POWER,
/* > >= <=    ,      '     %      **    */
   } TokenType;
```

图 4 新增枚举类型

2、在 scan.c 添加新增保留字的键值对

这里需要额外注意的是 globals 中定义了 MAXRESERVED 表示保留字的数量，原来定义了是 8 个，现在需要修改为 23，如果不修改后续新增的关键字无法正确判断。

```
/* MAXRESERVED = the number of reserved words */
#define MAXRESERVED 23
```

图 5 修改保留字的数量

在 scan.c 中添加关键字的键值对，例如 TRUE1 对应“true”。如下图所示，原来只有 8 个，TINY+增加了 15 个。

```
/* 保留字查找表 */
static struct{
    char* str;
    TokenType tok;
} reservedWords[MAXRESERVED]
= { {"if",IF},{"then",THEN},{"else",ELSE},{"end",END},
  {"repeat",REPEAT},{"until",UNTIL},{"read",READ},
  {"write",WRITE},{"true",TRUE1},{"false",FALSE1},
  {"or",OR},{"and",AND},{"not",NOT},{"int",INT},
  {"bool",BOOL1},{"string",STRING},{"float",FLOAT},
  {"double",DOUBLE},{"do",DO},{"while",WHILE},
  {"include",INCLUDE},{"break",BREAK},{"continue",CONTINUE}};
```

新增

图 6 新增保留字查找表键值对

3、在 util.c 添加新增保留字的输出

在 util.c 的 printToken 中可以看到保留字是直接输出的对应的 tokenString 的，因此需要添加后续新增的保留字，如下图所示，他们用 KEY 标识：

```
void printToken( TokenType token, const char* tokenString )
{ switch (token)
  { case IF:
    case THEN:
    case ELSE:
    case END:
    case REPEAT:
    case UNTIL:
    case READ:
    case WRITE:
    case TRUE1:
    case FALSE1:
    case OR:
    case AND:
    case NOT:
    case INT:
    case BOOL1:
    case STRING:
    case FLOAT:
    case DOUBLE:
    case DO:
    case WHILE:
    case INCLUDE:
    case BREAK:
    case CONTINUE:
      fprintf(listing,
        "KEY: %s\n",tokenString);
      break;
```

新增保留字

图 7 在 printToken 中添加新增保留字

4、在 util.c 添加新增特殊符号的输出

同理，还需要添加后续新增的特殊符号，其中“{”“}”这两个符号不需要添加，因为他们用来标识注释，在 util 中不需要处理。这些特殊符号用 SYM 标识，添加后总共有 17 个特殊符号，如下图所示：

```
case ASSIGN: fprintf(listing,"SYM: :=\n"); break;
case LT: fprintf(listing,"SYM: <\n"); break;
case MT: fprintf(listing,"SYM: >\n"); break;
case LE: fprintf(listing,"SYM: <=\n"); break;
case ME: fprintf(listing,"SYM: >=\n"); break;
case EQ: fprintf(listing,"SYM: =\n"); break;
case COMMA: fprintf(listing,"SYM: ,\n"); break;
case UPDOX: fprintf(listing,"SYM: \'\n"); break;
case PERCENT: fprintf(listing,"SYM: %\n"); break;
case LPAREN: fprintf(listing,"SYM: (\n"); break;
case RPAREN: fprintf(listing,"SYM: )\n"); break;
case SEMI: fprintf(listing,"SYM: ;\n"); break;
case PLUS: fprintf(listing,"SYM: +\n"); break;
case MINUS: fprintf(listing,"SYM: -\n"); break;
case TIMES: fprintf(listing,"SYM: *\n"); break;
case POWER: fprintf(listing,"SYM: **\n"); break;
case OVER: fprintf(listing,"SYM: /\n"); break;
```

图 8 在 printToken 中添加新增特殊符号

5、在 util.c 添加字符串常量的输出

字符串常量是 TINY+新增的，用 STR 标识。如下图所示，将其添加在 ID 之后。

```
case STR:
  fprintf(listing,
      "STR, val= %s\n",tokenString);
  break;
```

图 9 在 printToken 中添加字符串常量的输出

6、添加新增 DFA 状态

相比 TINY，TINY+新增了四个状态，INLE, INME, INUPDOX,INPOWER。DFA 状态的定义在 scan.c 文件中，用 StateType 枚举类型存储。如下图所示：

```
/* scanner DFA 中的状态 */
//  开始     赋值     注释     数字 标识符 结束 小于等于 大于等于  字符     幂
typedef enum                                                        新增
{
    START, INASSIGN, INCOMMENT, INNUM, INID, DONE, INLE, INME, INUPDOX,INPOWER
}
StateType;
```

图 10 添加新增状态

三、完善扫描器 scan。

1、保留字和标识符的判断

在词法分析器中，保留字和标识符的判断逻辑如下：

- 当开始状态扫描到一个字母时，进入变量标识符状态（INID）。
- 在标识符状态下，继续扫描后续字符，**直到遇到不是字母或数字的字符为止。**
- 如果扫描到的字符序列长度超过了最大标识符长度（MAXTOKENLEN），则不再保存后续字符。

- **如果扫描到的字符序列匹配了保留字表中的某个字符串，则识别为相应的保留字；否则，识别为标识符。**

首先需要完善对标识符的判断，初始代码只允许标识符中存在字母，而不允许数字，直接修改原来的 if 判断即可，如下图所示：

```
case INID:
    if (!(isalpha(c) || isdigit(c))) // 如果不是字母或数字
        ungetNextChar();
        save = FALSE;
        state = DONE;
        currentToken = ID;
    }
    break;
```

修改if判断

图 11 完善对标识符的判断

其中判断扫描到的字符序列是否匹配保留字表中的某个字符串的函数是 reservedLookup，该函数遍历保留字数组 reservedWords 来查找输入字符串 s 是否与保留字表中的某个字符串相匹配，如果找到了匹配的保留字，则返回相应的标记类型 reservedWords[i]，否则返回标识符类型。

```
/* 查找标识符以确定是否为保留字，使用线性查找 */
static TokenType reservedLookup(char* s)
{
    int i;
    for (i = 0; i < MAXRESERVED; i++)          // 遍历保留字数组
        if (!strcmp(s, reservedWords[i].str)) // 比较字符串是否相等
            return reservedWords[i].tok;       // 如果相等，返回相应的标记类型
    return ID;                                 // 否则返回标识符类型
}
```

图 12 reservedLookup 函数

因此只需要在原来的结束状态添加一个判断即可，如果当前类型是标识符，则调用 resevedLoopup 函数进行判断（这里原来的扫描器已实现）。

```
if (state == DONE)                                    // 如果状态为结束状态
{
    tokenString[tokenStringIndex] = '\0';
    if (currentToken == ID)
        currentToken = reservedLookup(tokenString);
}
```

图 13 调用函数进一步区分保留字和标识符

2、特殊字符的判断

2.1、常规特殊单字符

对于一些单字符——, % ; + - / ( ) = 可以直接识别。即在开始状态如果读取到这些单字符，将状态设置为 DONE，并设置 currentToken，其中这一步还可以直接判断非法符号和文件末尾。

注意，这里还需要删掉原来对“*” “<” 这两个符号的判断，因为 TINY+新增了以这些字符为首的特殊多字符。

```
switch (c){
case EOF:
    save = FALSE;
    currentToken = ENDFILE;
    break;
case '=':
    currentToken = EQ; break;
case '+': currentToken = PLUS; break;
case '-': currentToken = MINUS; break;
case '/': currentToken = OVER; break;
case '(': currentToken = LPAREN; break;
case ')': currentToken = RPAREN; break;
case ';': currentToken = SEMI; break;
case ',': currentToken = COMMA; break;
case '%': currentToken = PERCENT;break;
default:
    currentToken = ERROR;
    break;
}
```

特殊单字符直接识别

图 14 常规特殊单字符直接识别

2.2、多字符及其相关的单字符的判断。

对于 <= 、>=、**、:=及其的初始单字符<、>、:、*需要额外判断，他们判断逻辑相似，如下：

- 在开始状态中，**如果读到第一个单字符**，则将状态切换到对应的多字符状态。
- 在对应的多字符状态下，**如果读到对应的第二个单字符**，则将状态切换为 DONE，**并将 currentToken 设置为对应的多字符。**
- 在对应的多字符状态下，**如果读到其他字符**，则回退到上一个字符，并将 **currentToken 设置为读到的第一个单字符（:=例外，返回错误标志）。**

以**为例子，在开始状态 START 下，如果读取到 :，则将状态切换为 INPOWER（幂运算符号状态）。在状态 INPOWER 下，如果读取到 *，则表示 **，状态切换为 DONE，并返回 POWER（幂运算符号）。如果后续读取到其他字符，则单独返回 TIMES。

```
else if (c == ':')          // 如果是冒号，切换到赋值状态
    state = INASSIGN;
else if (c == '>')          // 如果是大于号，切换到大于等于状态
    state = INME;
else if (c == '<')          // 如果是小于号，切换到小于等于状态
    state = INLE;
else if (c == '*')          // 如果是星号，切换到 INPOWER 状态
    state = INPOWER;
```

图 15 读到第一个单字符，切换到对应的多字符状态

```
case INLE:
    if (c == '='){          // 如果是等于号,表示小于等于
        state = DONE;
        currentToken = LE;
    }
    else{                   // 否则表示小于
        ungetNextChar();    // 回退一个字符，表示等于号
        save = FALSE;
        state = DONE;
        currentToken = LT;
    }
    break;
```

图 16 在多字符状态进一步判断（以<=为例，其他类似，但:=不同）

这样，根据当前字符和后续字符的组合，可以正确判断出 <=、>=、**、:= 及其的初始单字符 <、>、:、*。

3、字符串常量的判断

根据 TINY+的词法规则：

STRING=' any character except ' '

STRING 用 '...' 括起来，除 ' 之外的任何字符都可以出现在 STRING 中。**一个 STRING 不能超过一行**，并且小写字母和大写字母是不同的。

根据以上规则设计字符串常量的判断逻辑如下：

①、在初始状态 START 下，如果读取到 '，则将状态切换为 INUPDOX（字符状态）

②、在状态 INUPDOX 下：

- 如果后续读取到 '，则表示字符串结束，状态切换为 DONE，并返回 STR（字符串标记）。
- 如果行缓冲区耗尽而未读取到 '，则表示字符串未闭合，状态切换为 DONE，并返回 ERROR（错误标记），同时在 tokenString 中记录错误信息（**实现对字符串不能换行的判断**）。
- 如果读取到其他字符，则继续保持在 INUPDOX 状态，表示字符串内容。

```
case INUPDOX:                       // 字符状态
    if (c == '\''){                 // 如果是'
        save = FALSE;
        state = DONE;
        currentToken = STR;
    }
    else if (!(linepos < bufsize)) { // 如果行缓冲区耗尽，输出错误信息
        save = FALSE;
        state = DONE;
        currentToken = ERROR;
        strcpy(tokenString, "Missing \" \' \" !");
        tokenStringIndex += 15;
    }
    break;
```

图 17 判断字符串常量

4、完善对注释的使用

根据 TINY+的词法规则：

①、成对出现：注释以 { 开始，以 } 结束。实验中对仅有‘}’情况才视为不合

法；并且如果到文件末尾都没找到}也视为不合法。

②、不能嵌套，即不能连续出现两个‘{’而中间没有‘}’；

③、允许嵌套多行

实现逻辑：

①、在开始状态 START 中，如果读取到注释开始符 {，则切换到注释状态 INCOMMENT。如果读取到注释结束符 },输出错误提示，如下图所示：

```
else if (c == '{'){  // 如果是注释开始符
    save = FALSE;
    state = INCOMMENT;
}
else if (c == '}'){  // 如果是注释结束符
    state = DONE;
    currentToken = ERROR;
    strcpy(tokenString, "Unexpected '}' outside of comment!");
    tokenStringIndex += 34;
    break;
}
```

图 18 开始状态对注释符号进行判断

②、当处于注释状态时：

- 首先会将 save 设为 FALSE，这表示注释内容不会被保存到 tokenString 中。
- 如果读取到文件结尾 EOF，则会切换到结束状态 DONE，返回错误标记，并设置错误信息为 "Missing " } " !"。
- 如果读取到注释结束符 },则会将状态切换回开始状态 START，表示注释结束。
- 如果读取到注释开始符 {，则会切换到结束状态 DONE，返回错误标记，并设置错误信息为 "Nested comments are not allowed!"，这表示不允许嵌套注释。

如下图所示

```
case INCOMMENT:  // 注释状态
  save = FALSE;              // 不保存到 tokenString
  if (c == EOF)  {           // 如果是文件尾
    state = DONE;            // 切换到结束状态
    currentToken = ERROR;    // 返回错误标记
    strcpy(tokenString, "Missing \" } \" !");
    tokenStringIndex += 15; // 更新 tokenString 索引
  }
  else if (c == '}')         // 如果是注释结束符
    state = START;           // 切换到开始状态
  else if (c == '{'){        // 如果是注释开始符
    state = DONE;            // 切换到结束状态
    currentToken = ERROR;    // 返回错误标记
    strcpy(tokenString, "Nested comments are not allowed!");
    tokenStringIndex += 31; // 更新 tokenString 索引
  }
  break;
```

图 19 注释状态分支决策

四、结果测试：

在 Linux 系统中进行测试：

- gcc main.c scan.c util.c -o main 指令编译程序；
- ./main 文件名 指令测试指定文件；

输出格式：

- KEY 为关键字;
- SYM 为系统特殊字符;
- ID 为变量;
- NUM 为数值常量;
- STR 为字符串常量。

1、测试样例

测试一个样例 tiny+2.txt，如下三图所示，可以正确进行词法分析，并以（种别码，属性值）表示。

```
zhangweijing_2021152011@ubuntu-2204:~/tiny$ ./main tiny+2.txt

TINY COMPILATION: tiny+2.txt

1: {this is an example}
2: int A,B;
        2: KEY: int
        2: ID, name= A
        2: SYM: ,
        2: ID, name= B
        2: SYM: ;
3: bool C;
        3: KEY: bool
        3: ID, name= C
        3: SYM: ;
4: string D;
        4: KEY: string
        4: ID, name= D
        4: SYM: ;
```

图 20 样例测试结果-1

```
5: D:= 'scanner';
        5: ID, name= D
        5: SYM: :=
        5: STR, val= scanner
        5: SYM: ;
6: C:=A and not B;
        6: ID, name= C
        6: SYM: :=
        6: ID, name= A
        6: KEY: and
        6: KEY: not
        6: ID, name= B
        6: SYM: ;
7: do
        7: KEY: do
```

图 21 样例测试结果-2

```
8: A:=A*2
        8: ID, name= A
        8: SYM: :=
        8: ID, name= A
        8: SYM: *
        8: NUM, val= 2
9: while A<=D
        9: KEY: while
        9: ID, name= A
        9: SYM: <=
        9: ID, name= D
        10: EOF
sh: 1: pause: not found
```

图 22 样例测试结果-3

2、测试保留字和特殊字符

测试保留关键字：

如下三图所示，可以正确识别所有的 KEY 并输出对应的符号。

```
1: { 1、测试关键字 }
2: true false or and not int bool string float
        2: KEY: true
        2: KEY: false
        2: KEY: or
        2: KEY: and
        2: KEY: not
        2: KEY: int
        2: KEY: bool
        2: KEY: string
        2: KEY: float
```

图 23 保留字测试-1

```
3: double do while include break continue
        3: KEY: double
        3: KEY: do
        3: KEY: while
        3: KEY: include
        3: KEY: break
        3: KEY: continue
```

图 24 保留字测试-2

```
4: if then else end repeat until read write
        4: KEY: if
        4: KEY: then
        4: KEY: else
        4: KEY: end
        4: KEY: repeat
        4: KEY: until
        4: KEY: read
        4: KEY: write
```

图 25 保留字测试-3

测试特殊符号：

其中有三个特殊符号不需要单独测试，单引号和大括号，因为单引号用于在字符串的判断中，而大括号用于在注释的判断中。如下两图所示，可以正确识别所有的 SYM 并输出对应的符号。

```
6: { 2、测试特殊符号 }
7: > <= >= , % **
        7: SYM: >
        7: SYM: <=
        7: SYM: >=
        7: SYM: ,
        7: SYM: %
        7: SYM: **
```

图 26 特殊符号测试-1

```
8: ; := + - * / ( ) < =
        8: SYM: ;
        8: SYM: :=
        8: SYM: +
        8: SYM: -
        8: SYM: *
        8: SYM: /
        8: SYM: (
        8: SYM: )
        8: SYM: <
        8: SYM: =
```

图 27 特殊符号测试-2

3、测试特殊情况

3.1、变量标识符不能以数字开头

变量标识符只允许包含数字和字母，并且不可以以数字开头，其中字母有大小写区分。如下图所示，可以正确识别标识符，并且对于数字开头的情况，会先将前面的数字识别为 NUM，后面部分识别为 ID。

```
10: { 3、测试ID，只包含数字和字母，并且不可以以数字开头 }
11: int identifier1
        11: KEY: int
        11: ID, name= identifier1
12: string IDENTIFIER2
        12: KEY: string
        12: ID, name= IDENTIFIER2
13: double YUsvx2gz
        13: KEY: double
        13: ID, name= YUsvx2gz
14: float 3identifier {数字不能作为标识符开头}
        14: KEY: float
        14: NUM, val= 3
        14: ID, name= identifier
15: bool _underscore {_是无效字符}
        15: KEY: bool
        15: ERROR: _
        15: ID, name= underscore
```

图 28 标识符测试结果

3.2、测试 NUM

如下图所示，可以正确测试 NUM。

```
17: { 4、测试NUM}
18: 123 4567 0 987654321
        18: NUM, val= 123
        18: NUM, val= 4567
        18: NUM, val= 0
        18: NUM, val= 987654321
```

图 29 测试 NUM

3.3、字符串不能换行

如下图所示，能够正确识别一般的字符串，并且如果 STRING 中包含了单引号，会把前面部分识别成一个字符串，字符串不允许跨行，会输出错误提示。

```
20: { 5、测试STRING，STRING不能多行，不能包含' }
21: str := ' string with valid characters '
        21: ID, name= str
        21: SYM: :=
        21: STR, val=  string with valid characters
22: 'string with invalid character (') '
        22: STR, val= string with invalid character (
        22: SYM: )
        22: ERROR: Missing " ' " !
23: 'string with
        23: ERROR: Missing " ' " !
24: multiple lines'
        24: ID, name= multiple
        24: ID, name= lines
        24: ERROR: Missing " ' " !
```

图 30 测试字符串常量

3.4、测试空白

用于测试空白的样例如下图所示，TINY+允许 ID、NUM 和关键字等之间存在空白。

```
{6、 测试空白 }
int    var1
do
    if (var1 <= 10)
        write('This is a test');

while (var1 > 0)
```

图 31 测试空白的样例

如下图所示，可以正确识别被空白隔开的 ID、NUM 和关键字等。

```
26: {6、 测试空白 }
27: int    var1
        27: KEY: int
        27: ID, name= var1
28: do
        28: KEY: do
29:     if (var1 <= 10)
        29: KEY: if
        29: SYM: (
        29: ID, name= var1
        29: SYM: <=
        29: NUM, val= 10
        29: SYM: )
```

图 32 空白测试结果-1

```
30:         write('This is a test');
        30: KEY: write
        30: SYM: (
        30: STR, val= This is a test
        30: SYM: )
        30: SYM: ;
31:
32: while (var1 > 0)
        32: KEY: while
        32: SYM: (
        32: ID, name= var1
        32: SYM: >
        32: NUM, val= 0
        32: SYM: )
```

图 33 空白测试结果-2

3.5、测试注释

如下图所示，允许执行多行注释，即注释跨越多行。还测试了注释前有其他词句的情况，可以正确分析注释前的词。

```
35: { 7、测试注释  }
36: {   测试多行注释    }
37: {
38:     这是 TINY+ 中的多行注释。它可以跨越多行。
39: }
40: {   测试注释前有其他符号    }
41: int sum; {  总和  }
        41: KEY: int
        41: ID, name= sum
        41: SYM: ;
```

图 34 注释测试-1

根据 TINY+词法规则，不允许注释嵌套，即不允许两个连续的“{”而中间没有“}”。如下图所示，可以输出错误信息，“Nested comments are not allowed”即不允许嵌套注释。

```
42: {   测试注释嵌套           }
43: {   注释    {       Nested  }
        43: ERROR: Nested comments are not allowed
        43: ID, name= Nested
        43: ERROR: }
44:
```

图 35 注释测试-2

测试只有测试开始符号或只有测试结束符号，如下图所示，会输出错误信息。

```
48: {   测试只有注释结束符号    }
49: test }
        49: ID, name= test
        49: ERROR: Unexpected '}' outside of comment!}
50: {   测试只有注释开始符号    }
51: {   test
        52: ERROR: Missing " } " !
        53: EOF
sh: 1: pause: not found
```

图 36 注释测试-3

3.5、测试最长匹配原则

如下图所示，符合最长匹配原则，可以正常区分特殊多字符及其相关特殊单字符。

```
45: {8、测试最长匹配原则 }
46: *** <=< >=> :=:
        46: SYM: **
        46: SYM: *
        46: SYM: <=
        46: SYM: <
        46: SYM: >=
        46: SYM: >
        46: SYM: :=
        46: ERROR: :
        47: EOF
```

图 37 测试最长匹配原则

4、实验总结：

本次编译原理实验的主要目标是理解 TINY+ 语言的词法及词法分析器的实现，并基于 TINY 扫描器的设计，完成对 TINY+ 的词法分析器的编写与测试。

在实验过程中，我首先通过学习 TINY 语言的词法规则以及已有的词法分析器实现原理，掌握了词法分析器的设计与实现方法。然后，根据 TINY+ 的词法描述，我进行了相应的扩展，设计了与 TINY+ 扫描器匹配的 DFA，根据这个 DFA 完善代码。

实际完成后，我还进行了充分的测试，确保词法分析器的准确性和健壮性。最终，我成功完成了对 TINY+ 的词法分析器的设计与实现。