# 实验报告：构词规则识别程序设计

*王贤义*

*计算机基地班*

*320210931221*

## 实验目的

本实验旨在设计一个程序，用于根据选定源语言的构词规则，从输入文件中识别出所有合法的单词符号，并以二元组形式输出，对无法识别的子串进行相应处理，将结果信息写入输出文件。

## 源语言描述

源语言包括关键字、常量、分隔符、单字符运算符、双字符运算符等基本元素。其中，关键字包括`else`, `if`, `int`, `while`, `bool`, `do`, `read`, `write`, `then`等；常量包括`true`, `false`和数字；分隔符包括`(`, `)`, `{`, `}`, `;`, `,`；运算符包括`+`, `-`, `*`, `/`, `<`, `>`, `=`, `!`, `<=`, `>=`, `==`, `!=`, `&&`, `||`, `:=`等。

## 词法规则

- 标识符：由字母打头后跟字母、数字任意组合的字符串；长度不超过8；不区分大小写；把下划线看作第27个字母。
- 整常数：完全由数字组成的字符串；正数和0前面不加符号，负数在正数前面加-构成；长度不超过8；十进制。
- 布尔常量：true 和 false 。
- 关键字、运算符、分隔符仅包含在文法定义中出现过的终结符。关键字保留。
- 字母表定义为上述规则中出现的字符的集合；不在该集合中的符号都以非法字符看待。
- 源程序中可以出现单行注释和多行注释，其语法参考C/C++语言。

## 设计约束

本程序设计至少包含两个模块：驱动模块和工作模块。驱动模块负责输入、输出处理及调用工作模块（Driver函数）；工作模块负责清洗、分割、识别、编码等工作。此设计方式便于后续任务直接调用工作模块，减少代码修改需求（analyze函数）。

此外还设计了一个辅助模块用于定位错误发生的具体位置（get_line_column函数）。

## 程序设计与实现

### 工作流程

1. **输入处理**：读取源文件内容。
2. **词法分析**：根据构词规则分析源文件，识别合法的单词符号及处理错误。
3. **输出处理**：将分析结果和错误信息写入输出文件，中间用error进行分隔。

### 关键算法描述

词法分析主要通过循环读取源文件中的每个字符，根据字符的类型（如字母、数字、运算符等）和上下文关系，进行相应的处理，包括：

- 过滤空白字符（包括空格、制表符、换行符等）。

```
# 如果当前字符是需要过滤的空白字符，则跳过
    if source[i] in filter_chars:
        i += 1
        continue
```

- 跳过注释。

```

        # 如果当前字符是单行注释的开始，则跳过注释内容
        elif source[i] == "/" and i + 1 < len_source and source[i + 1] == "/":
            i += 2  # 跳过注释标志“//”
            while i < len_source and source[i] != "\n":  # 跳过注释内容直到行末
                i += 1
            continue
        # 如果当前字符是多行注释的开始，则跳过注释内容
        elif source[i] == "/" and i + 1 < len_source and source[i + 1] == "*":
            i += 2  # 跳过注释标志“/*”
            while i < len_source:
                if source[i] == "*" and i + 1 < len_source and source[i + 1] ==
"/":  # 遇到注释结束标志“*/”
                    i += 2
                    break
                i += 1
            continue
```

- 识别分隔符、关键字、常量、标识符、运算符等。

```
# 如果当前字符是开括号，则将其压入栈中，并记录为分隔符
        elif source[i] in brackets.keys():
            stack.append(source[i])
            output.append((source[i], "delimiter"))
            i += 1
        # 如果当前字符是闭括号
        elif source[i] in brackets.values():
            if (
                not stack or brackets[stack.pop()] != source[i]
            ):  # 如果栈为空或不匹配，则记录为非法括号
                error.append(
                    (source[i], f"illegal bracket{get_line_column(source, i)}")
                )
            else:  # 否则记录为分隔符
                output.append((source[i], "delimiter"))
            i += 1
        # 如果当前字符是分隔符，则记录为分隔符
        elif source[i] in delimiters:
            output.append((source[i], "delimiter"))
            i += 1
        # 如果当前字符是字母或下划线，则可能是关键字、常量或标识符
        elif source[i].isalpha() or source[i] == "_":
            token = ""  # 初始化词元字符串
            # 循环读取字符直到非字母、非数字、非下划线
            while i < len_source and (
                source[i].isalpha() or source[i].isdigit() or source[i] == "_"
            ):
```

```
                token += source[i]
                i += 1
            token = token.lower()  # 将词元转换为小写，以便匹配关键字和常量
            # 如果词元长度超过8个字符，则记录为非法标识符
            if len(token) > 8:
                error.append((token, f"illegal
identifier{get_line_column(source, i)}"))
            # 如果词元是关键字，则记录为关键字
            elif token in keywords:
                output.append((token, "keyword"))
            # 如果词元是常量，则记录为常量
            elif token in constants:
                output.append((token, "constant"))
            # 否则，词元被视为标识符
            else:
                tokens.setdefault(token, len(tokens) + 1)  # 为新标识符分配一个唯一
编号
                output.append((token, f"identifier {tokens[token]}"))  # 记录为标
识符
        # 如果当前字符是数字或负号且后面紧跟数字，则可能是常量
        elif source[i].isdigit() or (
            source[i] == "-" and i + 1 < len_source and source[i + 1].isdigit()
        ):
            token = source[i]  # 初始化词元字符串
            i += 1
            # 循环读取数字字符
            while i < len_source and source[i].isdigit():
                token += source[i]
                i += 1
            if token == "0":
                output.append((token, "constant"))
            # 如果词元长度超过8个字符或以0开头，则记录为非法常量
            elif len(token) > 8 or token[0] == "0":
                error.append((token, f"invalid constant{get_line_column(source,
i)}"))
            else:  # 否则，记录为常量
                output.append((token, "constant"))
        # 如果当前字符是感叹号，可能是逻辑非运算符或不等于运算符
        elif source[i] == "!":
            if (
                i + 1 < len_source and source[i + 1] == "="
            ):  # 如果下一个字符是等号，则是不等于运算符
                output.append(("!=", "operator"))
                i += 2  # 跳过“!=”
            else:  # 否则，是逻辑非运算符
                output.append(("!", "operator"))
                i += 1
        # 如果当前字符和下一个字符组成的字符串是运算符，则记录为运算符
        elif source[i : i + 2] in operators:
            output.append((source[i : i + 2], "operator"))
            i += 2
        # 如果当前字符是单字符运算符，则记录为运算符
        elif source[i] in operators:
            output.append((source[i], "operator"))
            i += 1
```

- 处理未知字符。

```
        # 如果当前字符不属于以上任何一种情况，则记录为未知错误
        else:
            error.append((source[i], f"illegal unknown{get_line_column(source,
i)}"))
            i += 1
```

## 出错处理

- 处理非法括号

```
        # 如果当前字符是闭括号
        elif source[i] in brackets.values():
            if not stack or brackets[stack.pop()] != source[i]:  # 如果栈为空或不匹
配，则记录为非法括号
                error.append(
                    (source[i], f"illegal bracket{get_line_column(source, i)}")
                )
            else:  # 否则记录为分隔符
                output.append((source[i], "delimiter"))
            i += 1
```

- 处理过长标识符

```
# 如果当前字符是字母或下划线，则可能是关键字、常量或标识符
        elif source[i].isalpha() or source[i] == "_":
            token = ""  # 初始化词元字符串
            # 循环读取字符直到非字母、非数字、非下划线
            while i < len_source and (
                source[i].isalpha() or source[i].isdigit() or source[i] == "_"
            ):
                token += source[i]
                i += 1
            token = token.lower()  # 将词元转换为小写，以便匹配关键字和常量
            # 如果词元长度超过8个字符，则记录为非法标识符
            if len(token) > 8:
                error.append((token, f"illegal
identifier{get_line_column(source, i)}"))
            # 如果词元是关键字，则记录为关键字
            elif token in keywords:
                output.append((token, "keyword"))
            # 如果词元是常量，则记录为常量
            elif token in constants:
                output.append((token, "constant"))
            # 否则，词元被视为标识符
            else:
                tokens.setdefault(token, len(tokens) + 1)  # 为新标识符分配一个唯一
编号
                output.append((token, f"identifier {tokens[token]}"))  # 记录为标
识符
```

- 处理非法常量

```
# 如果当前字符是数字或负号且后面紧跟数字，则可能是常量
        elif source[i].isdigit() or (
            source[i] == "-" and i + 1 < len_source and source[i + 1].isdigit()
```

```
        ):
            token = source[i]  # 初始化词元字符串
            i += 1
            # 循环读取数字字符
            while i < len_source and source[i].isdigit():
                token += source[i]
                i += 1
            if token == "0":
                output.append((token, "constant"))
            # 如果词元长度超过8个字符或以0开头，则记录为非法常量
            elif len(token) > 8 or token[0] == "0":
                error.append((token, f"invalid constant{get_line_column(source,
i)}"))
            else:  # 否则，记录为常量
                output.append((token, "constant"))
```

- 处理未知符号

```
        # 如果当前字符不属于以上任何一种情况，则记录为未知错误
        else:
            error.append((source[i], f"illegal unknown{get_line_column(source,
i)}"))
            i += 1
```

## 数据结构设计

- **tokens**：字典类型，用于存储标识符及其对应的编号。
- **brackets**：字典类型，用于记录括号配对关系信息。
- **output**：列表类型，用于收集识别出的合法单词符号。
- **error**：列表类型，用于记录无法识别的子串及错误信息。
- **keywords**,**constants**,**filter_chars**,**operators**:列表类型，用于记录关键字，常量等

## 程序结构

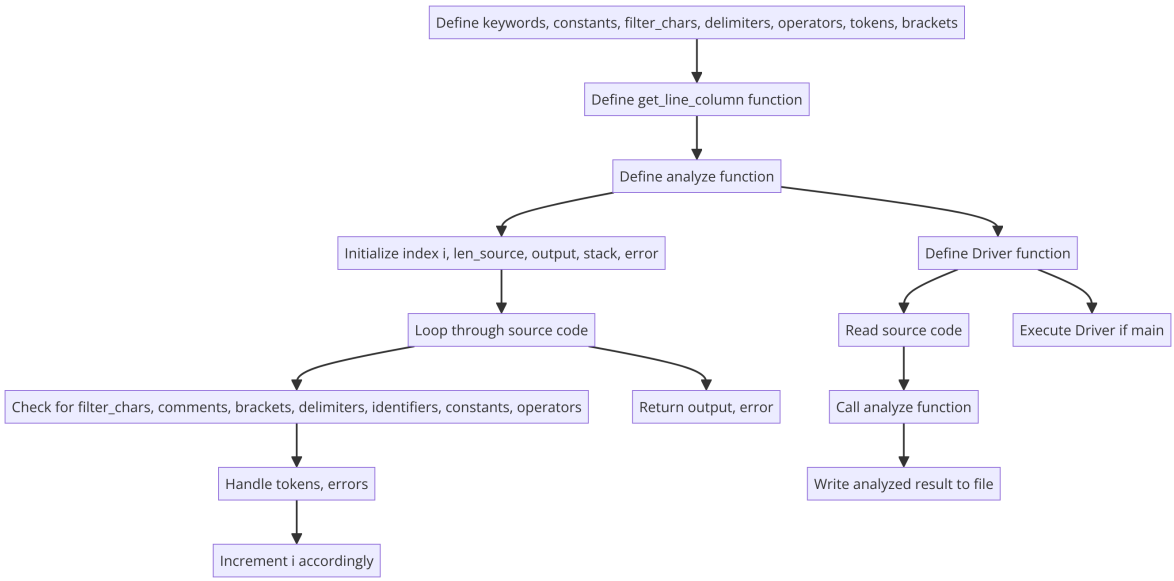
```
keywords = [...]
constants = [...]
filter_chars = [...]
delimiters = [...]
operators = [...]
tokens = {}
brackets = {...}

def get_line_column(source, i): ...

def analyze(source): ...

def Driver(): ...

if __name__ == "__main__":
    Driver()
```




# 测试用例及结果

程序经过5组测试用例进行了充分测试，测试结果显示程序能正确识别合法的单词符号，并对不合法的子串进行了适当处理。

**测试1**

输入 source1.txt：

source1.txt

```
1   //program 1: add two numbers.
2   {
3       int a, b, c  ;
4       a = 1;
5       b = 2;
6       c = a + b ;
7   }
8
```

输出 lex_out1.txt：

lex_out1.txt

```
<{, delimiter>
<int, keyword>
<a, identifier 1>
<,, delimiter>
<b, identifier 2>
<,, delimiter>
<c, identifier 3>
<;, delimiter>
<a, identifier 1>
<=, operator>
<1, constant>
<;, delimiter>
<b, identifier 2>
<=, operator>
<2, constant>
<;, delimiter>
<c, identifier 3>
<=, operator>
<a, identifier 1>
<+, operator>
<b, identifier 2>
<;, delimiter>
<}, delimiter>

```

**测试2**

输入 source2.txt:

source2.txt

```
//program 2: do some calculation.
{
    int a, b, c  ;
    a = 5;      //positive number
    b = -3;     //negative number
    c = (a+b)*(a-b);//calculation
    write c  ;  //output:16.
}

```

输出 lex_out2.txt：

lex_out2.txt

```
<{, delimiter>
<int, keyword>
<a, identifier 1>
<,, delimiter>
<b, identifier 2>
<,, delimiter>
<c, identifier 3>
<;, delimiter>
<a, identifier 1>
<=, operator>
<5, constant>
<;, delimiter>
<b, identifier 2>
<=, operator>
<-3, constant>
<;, delimiter>
<c, identifier 3>
<=, operator>
<(, delimiter>
<a, identifier 1>
<+, operator>
<b, identifier 2>
<), delimiter>
<*, operator>
<(, delimiter>
<a, identifier 1>
<-, operator>
<b, identifier 2>
<), delimiter>
<;, delimiter>
<write, keyword>
<c, identifier 3>
<;, delimiter>
<}, delimiter>

```

**测试3**

输入 source3.txt:

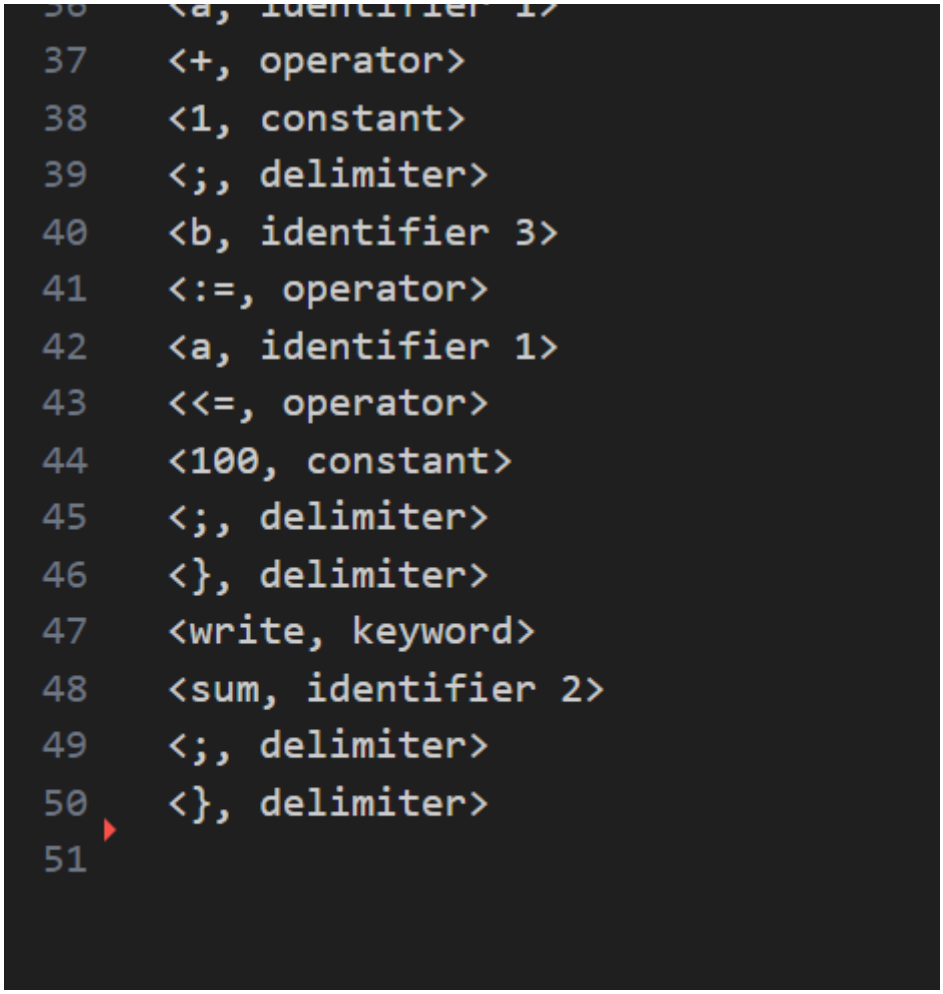
source3.txt

```
/*program 3: add numbers from 1 to 100
 *and print the result.
 */
{
    int a , sum ;
    bool  b ;
    a = 1 ;
    sum = 0 ;
    b := a <= 100 ;
    while b do
    {
        sum = sum + a ;
        a = a + 1 ;
        b := a <= 100 ;
    }
    write  sum  ;
}
```

输出 lex_out3.txt：

lex_out3.txt

```
1    <{, delimiter>
2    <int, keyword>
3    <a, identifier 1>
4    <,, delimiter>
5    <sum, identifier 2>
6    <;, delimiter>
7    <bool, keyword>
8    <b, identifier 3>
9    <;, delimiter>
10   <a, identifier 1>
11   <=, operator>
12   <1, constant>
13   <;, delimiter>
14   <sum, identifier 2>
15   <=, operator>
16   <0, constant>
17   <;, delimiter>
18   <b, identifier 3>
19   <:=, operator>
20   <a, identifier 1>
21   <<=, operator>
22   <100, constant>
23   <;, delimiter>
24   <while, keyword>
25   <b, identifier 3>
26   <do, keyword>
27   <{, delimiter>
28   <sum, identifier 2>
29   <=, operator>
30   <sum, identifier 2>
31   <+, operator>
32   <a, identifier 1>
33   <;, delimiter>
34   <a, identifier 1>
35   <=, operator>
36   <a, identifier 1>
```




**测试4**

输入 source4.txt:

source4.txt

```
/* program8: input three numbers ,output the largest one.
 * Test Statement : If_then and If_Then_Else .
 */
{
    int a,b,c ;
    bool cond1,cond2,cond3;

    read  a;  read  b;  read  c;

    cond1 := a >= b ;
    cond2 := a >= c ;
    cond3 := b >= c ;

    if cond1 then
        if cond2 then write a ;
        else write c ;
    cond1 := a < b ;
    if cond1 then
        if cond3 then write b ;
        else write c ;
}

```

输出 lex_out4.txt：

lex_out4.txt

```
<{, delimiter>
<int, keyword>
<a, identifier 1>
<,, delimiter>
<b, identifier 2>
<,, delimiter>
<c, identifier 3>
<;, delimiter>
<bool, keyword>
<cond1, identifier 4>
<,, delimiter>
<cond2, identifier 5>
<,, delimiter>
<cond3, identifier 6>
<;, delimiter>
<read, keyword>
<a, identifier 1>
<;, delimiter>
<read, keyword>
<b, identifier 2>
<;, delimiter>
<read, keyword>
<c, identifier 3>
<;, delimiter>
<cond1, identifier 4>
<:=, operator>
<a, identifier 1>
<>=, operator>
<b, identifier 2>
<;, delimiter>
<cond2, identifier 5>
<:=, operator>
<a, identifier 1>
<>=, operator>
<c, identifier 3>
```

```
<;, delimiter>
<cond3, identifier 6>
<:=, operator>
<b, identifier 2>
<>=, operator>
<c, identifier 3>
<;, delimiter>
<if, keyword>
<cond1, identifier 4>
<then, keyword>
<if, keyword>
<cond2, identifier 5>
<then, keyword>
<write, keyword>
<a, identifier 1>
<;, delimiter>
<else, keyword>
<write, keyword>
<c, identifier 3>
<;, delimiter>
<cond1, identifier 4>
<:=, operator>
<a, identifier 1>
<<, operator>
<b, identifier 2>
<;, delimiter>
<if, keyword>
<cond1, identifier 4>
<then, keyword>
<if, keyword>
<cond3, identifier 6>
<then, keyword>
<write, keyword>
<b, identifier 2>
<;, delimiter>
<else, keyword>
<write, keyword>
```

```
73    <c, identifier 3>
74    <;, delimiter>
75    <}, delimiter>
76
77
```

**测试5**

**测试5包含错误处理结果和特殊处理，包含括号匹配错误，未知字符，超长标识符，0开头的正数，超长整型数等错误处理和由关键字拼接成的标识符等特殊处理**

输入 source5.txt:

source5.txt

```
1   {
2       ({)(}
3       @#!
4       dsafsf_asdfaw
5       1a4
6       012
7       0-2
8       a4 := !a4
9       123456789
10      1.2
11      true
12      boolbool
13      |||
14  }
```

输出 lex_out5.txt：

lex_out5.txt

```
<{, delimiter>
<(, delimiter>
<{, delimiter>
<(, delimiter>
<!, operator>
<1, constant>
<a4, identifier 1>
<0, constant>
<-2, constant>
<a4, identifier 1>
<:=, operator>
<!, operator>
<a4, identifier 1>
<1, constant>
<2, constant>
<true, constant>
<boolbool, identifier 2>
<||, operator>
----------------------------------error----------------------------------
<), illegal bracket(2, 7)>
<}, illegal bracket(2, 9)>
<@, illegal unknown(3, 5)>
<#, illegal unknown(3, 6)>
<dsafsf_asdfaw, illegal identifier(4, 18)>
<012, invalid constant(6, 8)>
<123456789, invalid constant(9, 14)>
<., illegal unknown(10, 6)>
<|, illegal unknown(13, 7)>
<}, illegal bracket(14, 1)>
```

## 实验总结

本实验通过设计一个能够根据构词规则识别合法单词符号的程序，加深了对编译原理中词法分析阶段的理解。通过模块化设计，提高了程序的复用性和可维护性。