北京交通大學

BEIJING JIAOTONG UNIVERSITY

《编译原理》实验报告

实验名称：专题二 递归下降语法分析设计原理与实现

学　　号：22281188

姓　　名：江家玮

学　　院：计算机科学与技术学院

日　　期：2024 年 12 月 05 日

目录

1.程序功能描述

本程序是一个基于递归下降分析法（专题一）的语法分析器，用来解析和验证输入的表达式是否符合特定的语法规则。递归下降分析是一种自顶向下的解析方法，通过使用一组递归函数来解析输入的表达式。程序的核心功能是根据预定义的文法规则，对输入的表达式进行解析并判断其是否合法。

程序从专题一的结果文件 D:/lexical_analysis_output.txt 中读取输入的二元式，做出相应转化，并使用递归下降的方式，通过多个递归函数(S, E, T, F, A, M 等)来验证输入的表达式是否符合文法规则。判断输入的表达式是否符合文法规则，输出结果为语句合法或语句不合法。

2.主要数据结构描述

1、字符数组 char s[500]

用于存储转换后的表达式的字符数组。程序会将从文件读取的内容解析并转换为一个没有冗余信息的简洁字符串，然后存储在 s 中。这是整个语法解析过程中被递归函数操作的主要字符串。

2、变量 i

索引变量，表示当前解析字符在字符数组 s 中的位置。解析器在分析表达式时，会根据这个索引变量逐步遍历字符数组 s。

3、变量 SIGN

用于表示语法分析过程中是否出现错误。SIGN 的值为 0 表示当前没有错误，而非 0 表示已经检测到错误。

3.程序结构描述

3.1 主程序

1、功能：

main() 函数是程序的入口，负责读取文件中的输入内容，将其转换为程序可以理解的格式，并调用递归函数 S() 开始解析输入的表达式。

2、主要步骤：

打开输入文件，读取每一行的内容到字符数组 arr 中。

根据文件内容，将 arr 中的字符逐步添加到字符串 str 和 med 中，并根据变量标识符（例如 (2, a) 中的 2）将变量标记为 i。

将字符串 med 作为转换后的表达式，复制到字符数组 s 中，添加结束符 #。

检查输入是否为空（即第一个字符为 #），若为空则直接结束程序。

调用递归函数 S() 进行语法分析。

分析结束后，检查是否以 # 正常结束，并输出解析结果。

3.2 递归下降分析器子函数

根据规则实现了递归下降分析器，每个递归函数对应一个文法规则。

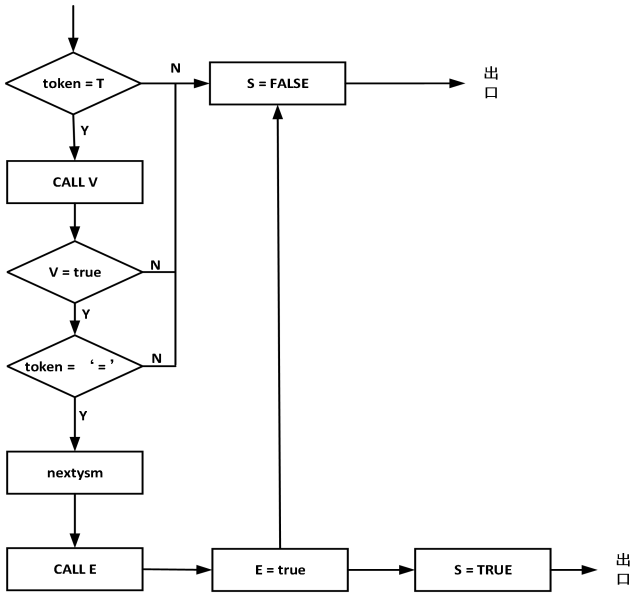
3.2.1 S() 函数

处理赋值语句，形式为 $S \rightarrow V = E$。

检查是否为变量 i。

确认后检查等号 =。

调用 E() 解析等号右侧的表达式。

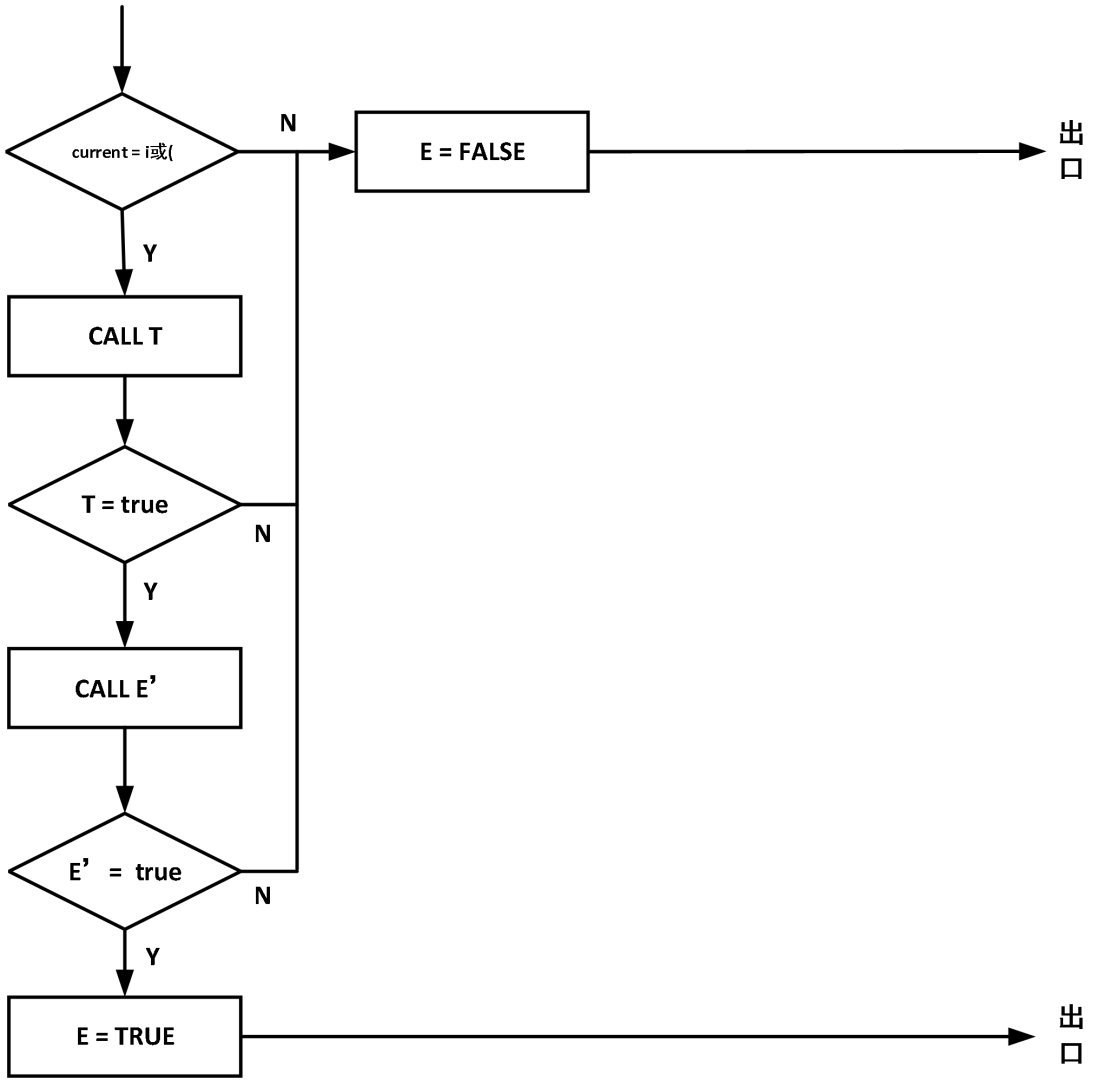


3.2.2 E() 函数

处理表达式，形式为 E → T E'。

检查是否为括号或变量 i，并调用 T() 解析项。

调用 E_() 处理剩余的部分（如果存在加减操作）。



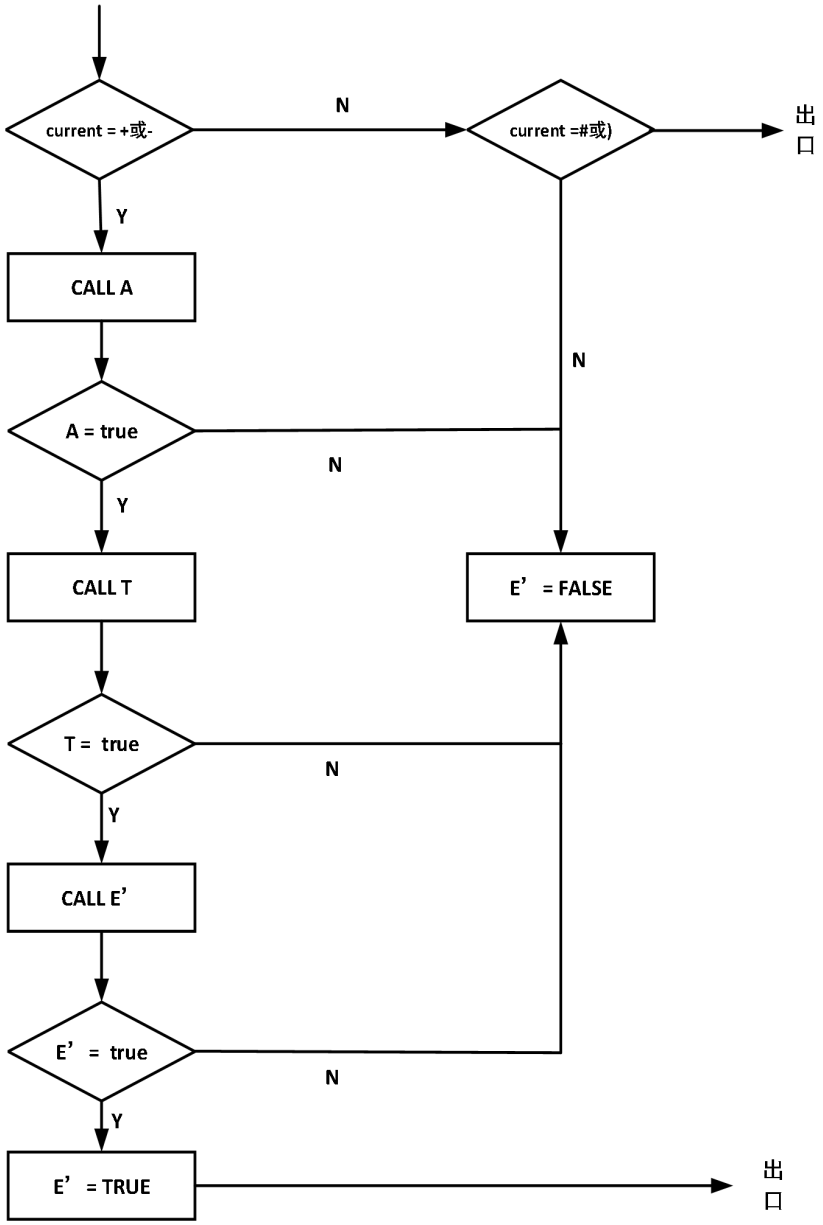
3.2.3 E_() 函数

处理表达式的后续部分，形式为 E' → + T E' | - T E' | ε。

判断是否为 + 或 - 操作符。

如果是，则调用 A() 解析加减操作。

之后调用 T() 解析接下来的项，继续递归调用 E_() 解析可能存在的后续表达式。

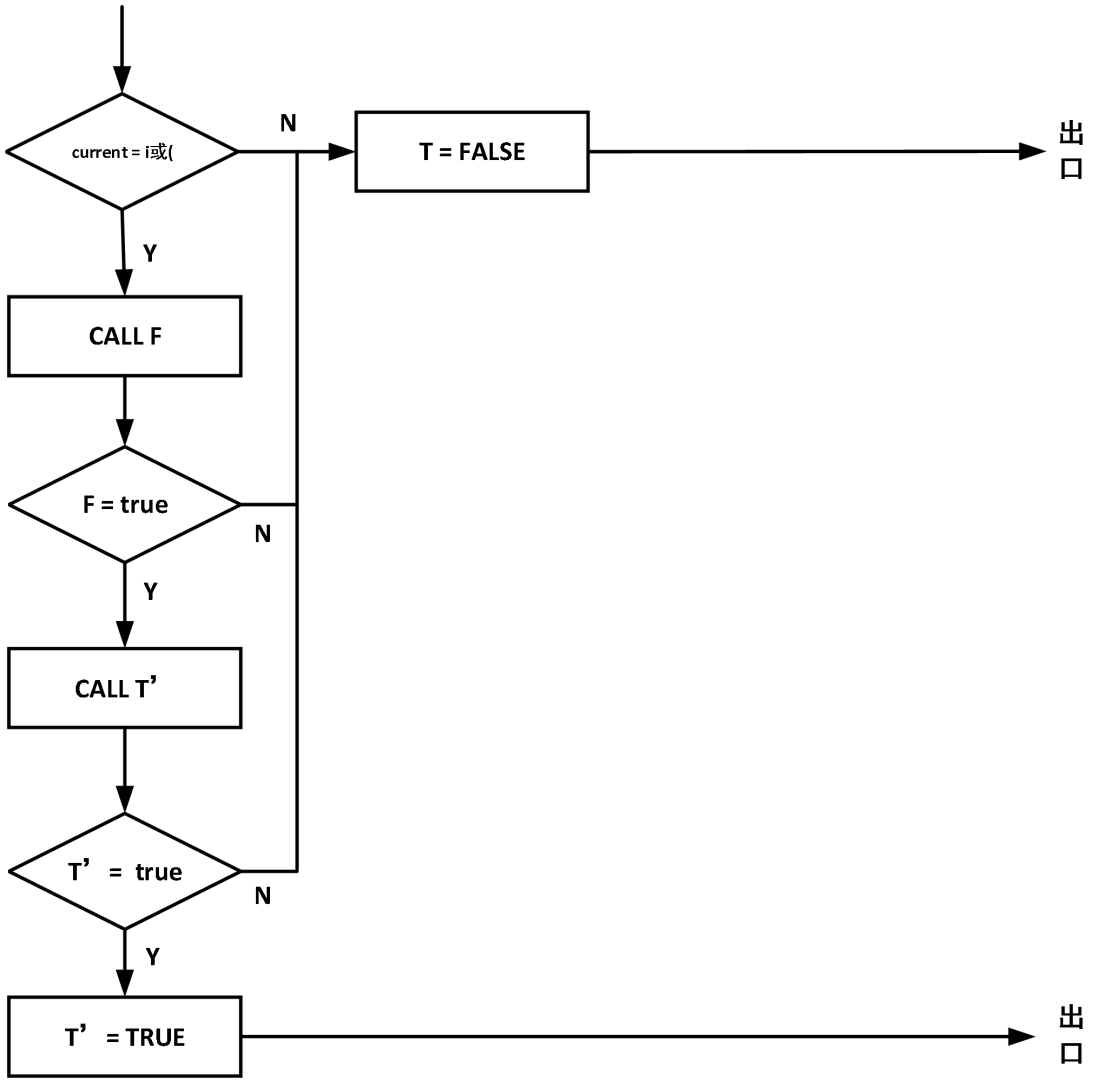


3.2.4 T() 函数

处理项，形式为 T → F T'。

检查是否为括号或变量 i，并调用 F() 解析因子。

调用 T_() 处理剩余的部分（如果存在乘除操作）。



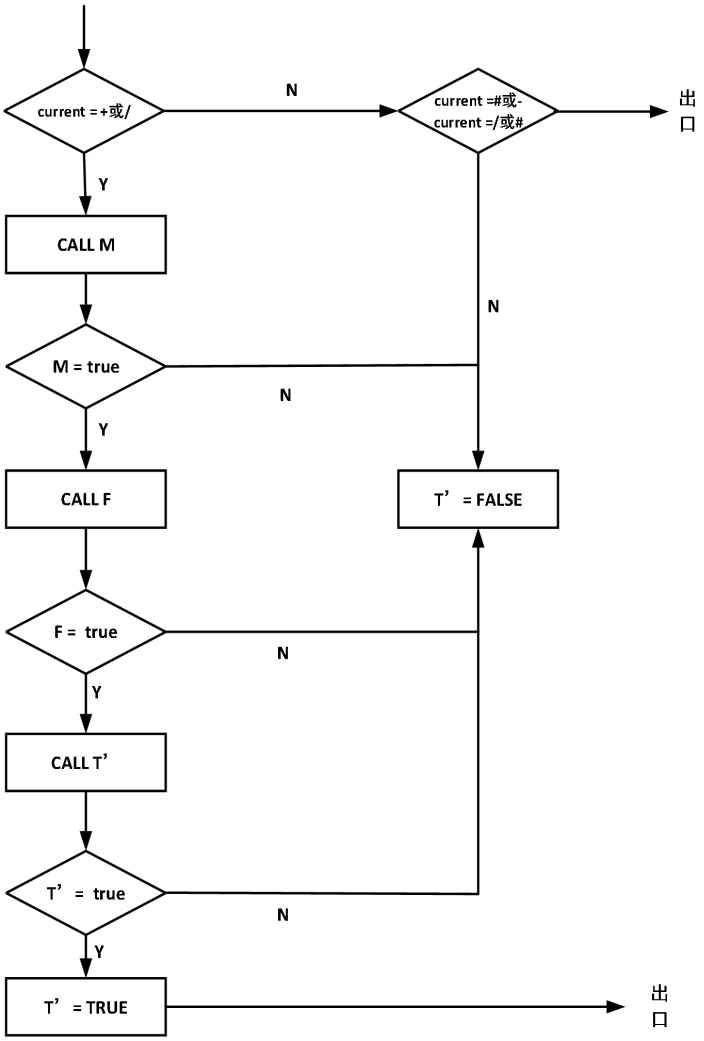
3.2.5 T_() 函数

处理项的后续部分，形式为 T' → * F T' | / F T' | ε。

判断是否为 * 或 / 操作符。

如果是，则调用 M() 解析乘除操作。

之后调用 F() 解析接下来的因子，并继续递归调用 T_() 解析可能存在的后续项。



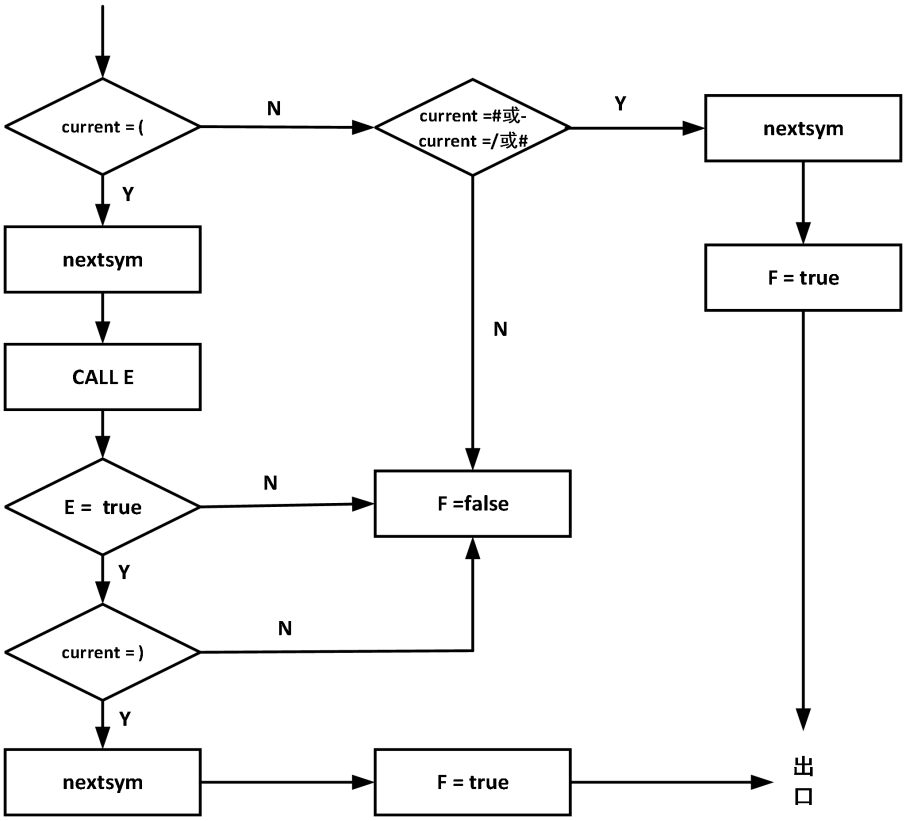
3.2.6 F() 函数

处理因子，形式为 $F \rightarrow (E) \mid i$。

检查是否为括号开头。

如果是，则跳过左括号 (，调用 E() 解析括号内的表达式，并检查右括号)。

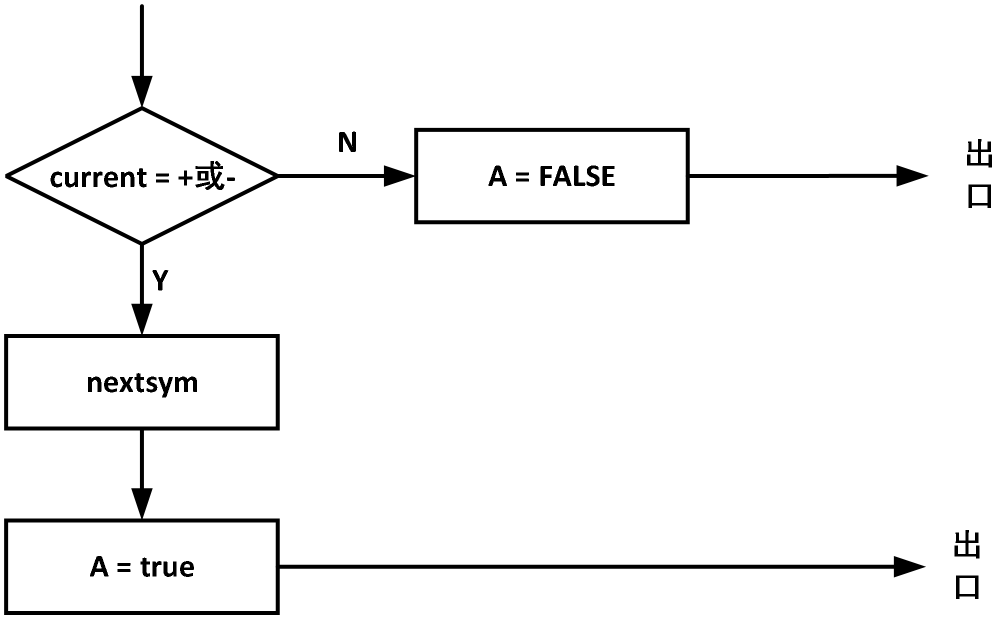
如果不是括号，则检查是否为变量 i。



3.2.7 A() 函数

处理加减运算符。

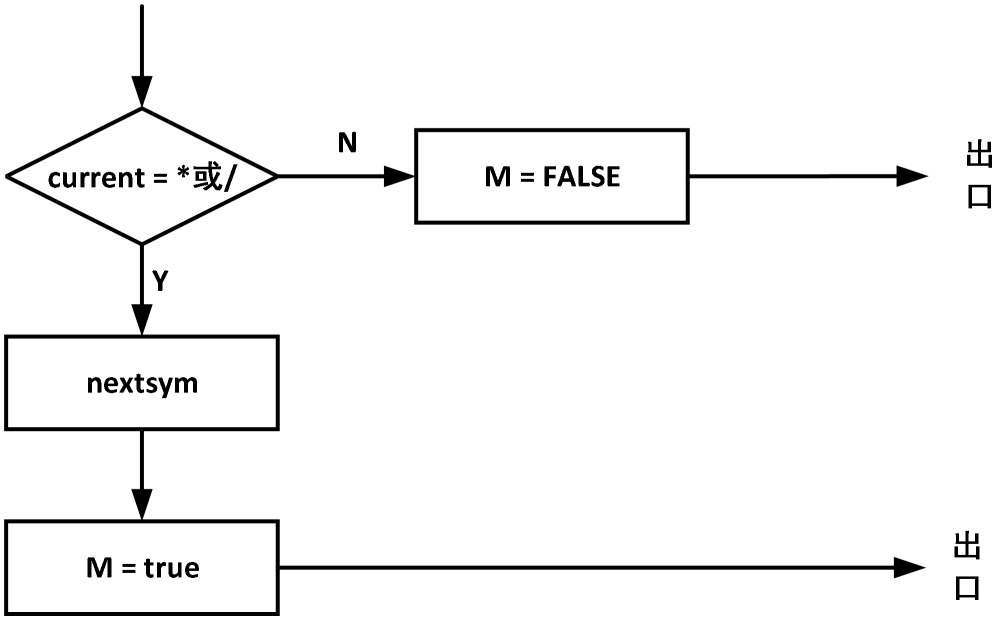
判断当前字符是否为 + 或 -，并跳过该字符。



3.2.8 M() 函数

处理乘除运算符。

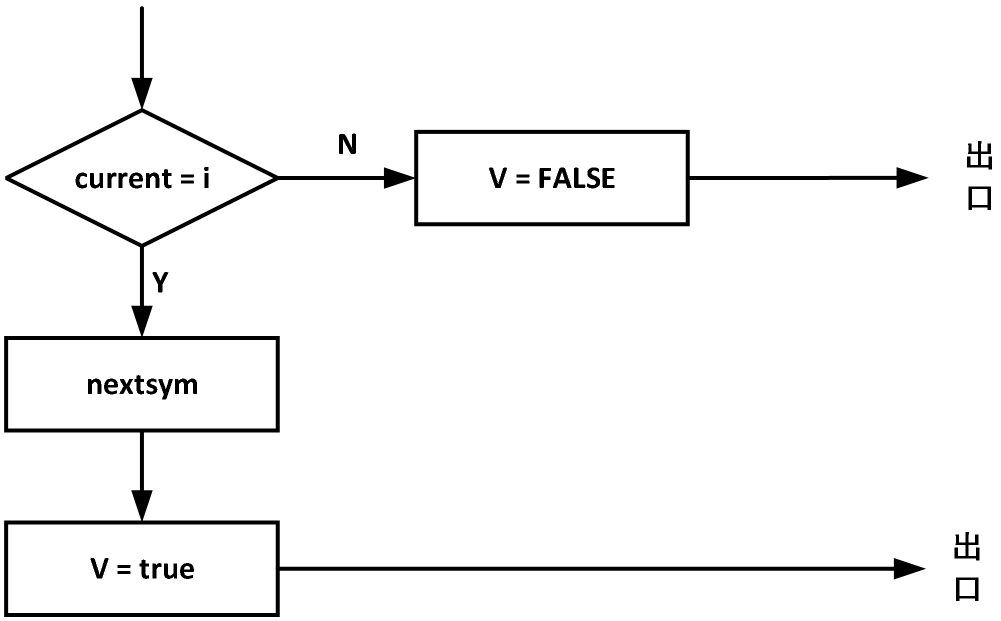
判断当前字符是否为 * 或 /，并跳过该字符。



3.2.9 V() 函数

处理变量。

判断当前字符是否为变量标识符 i，并跳过该字符。



4.程序测试

4.1 合法情况

1、简单赋值语句：a = b

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
 a = b
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(2,b)
```

递归下降分析运行结果：

```
输入的语句为： a=b#
转化为： i=i#
检查 S 中...
V   E   T   F
语句合法
```

2、带括号的表达式：a = (b + c) * d

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a = (b + c) * d
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(4,()
(2,b)
(4,+)
(2,c)
(4,))
(4,*)
(2,d)
```

递归下降分析运行结果：

```
输入的语句为: a=(b+c)*d#
转化为: i=(i+i)*i#
检查 S 中...
V   E   T   F   E   T   F   E'   A   T   F   T'   M   F
语句合法
```

3、连续加减运算：a = b + c - d + e

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a = b + c - d + e
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中:
(2,a)
(4,=)
(2,b)
(4,+)
(2,c)
(4,-)
(2,d)
(4,+)
(2,e)
```

递归下降分析运行结果：

```
输入的语句为: a=b+c-d+e#
转化为: i=i+i-i+i#
检查 S 中...
V   E   T   F   E'   A   T   F   A   T   F   A   T   F
语句合法
```

4.2 不合法情况

1、 缺少赋值运算符：a b + c

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a b + c
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中:
(2,a)
(2,b)
(4,+)
(2,c)
```

递归下降分析运行结果：

```
输入的语句为： ab+c#
转化为： ii+i#
检查 S 中...
V    错误：在 S 处缺少'='

语句不合法
```

2、 无效的起始字符：a = + b

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a = + b
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(4,+)
(2,b)
```

递归下降分析运行结果：

```
输入的语句为： a=+b#
转化为： i=+i#
检查 S 中...
V    E    E错误
```

3、 无效的结束字符：a = b + c +

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a = b + c +
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(2,b)
(4,+)
(2,c)
(4,+)
```

递归下降分析运行结果：

```
输入的语句为： a=b+c+#
转化为： i=i+i+#
检查 S 中...
V   E   T   F   E'   A   T   F   A   错误：在 E' 中缺少有效的项（T）
```

4、 不匹配的括号：a = (b + c)) * d

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
 a = (b + c)) * d
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(4,()
(2,b)
(4,+)
(2,c)
(4,))
(4,))
(4,*)
(2,d)
```

递归下降分析运行结果：

```
输入的语句为： a=(b+c))*d#
转化为： i=(i+i))*i#
检查 S 中...
V   E   T   F   E   T   F   E'   A   T   F
语句不合法
```

5、 缺少右括号：a = (b + c * d

专题一运行结果：

```
请输入测试用例代码（以单独一行的 #END 结束输入）：
a = (b + c * d
#END

以下是结果输出,并同时写入 lexical_analysis_output.txt 中：
(2,a)
(4,=)
(4,()
(2,b)
(4,+)
(2,c)
(4,*)
(2,d)
```

递归下降分析运行结果：

```
输入的语句为： a=(b+c*d#
转化为： i=(i+i*i#
检查 S 中...
V   E   T   F   E   T   F   E'   A   T   F   T'   M   F   错误：在 F 中缺少右括号 ')'

语句不合法
```

5.选做：识别 if 语句

要识别 if 语句，就要相应的设计 IF 函数，用于处理 if-else 语句，设计思路如下：

（1） 检测 if 关键字

通过 strncmp 检查当前输入是否以 if 开头。

如果是 if，则跳过该关键字，并继续解析条件表达式部分。

（2） 解析条件表达式

if 语句的条件表达式通常放在圆括号 () 中。

检查是否有左括号 (，然后调用表达式解析函数 E() 来处理括号中的逻辑表达式。

解析结束后，检查右括号) 是否存在，以确保条件表达式完整。

（3） 解析 if 语句块

条件表达式结束后，语句块应当包含在大括号 {} 内。

检查左大括号 {，并调用 S() 解析 if 语句块内部的内容。

解析完语句块后，检查右大括号 } 以确保语句块的结束。

（4） 处理 else 语句

在解析完 if 语句后，检查是否有 else 关键字。

如果存在 else，同样检查 {}，并递归调用 S() 解析 else 语句块的内容。

最后，检查 else 语句块的右大括号 } 是否存在。

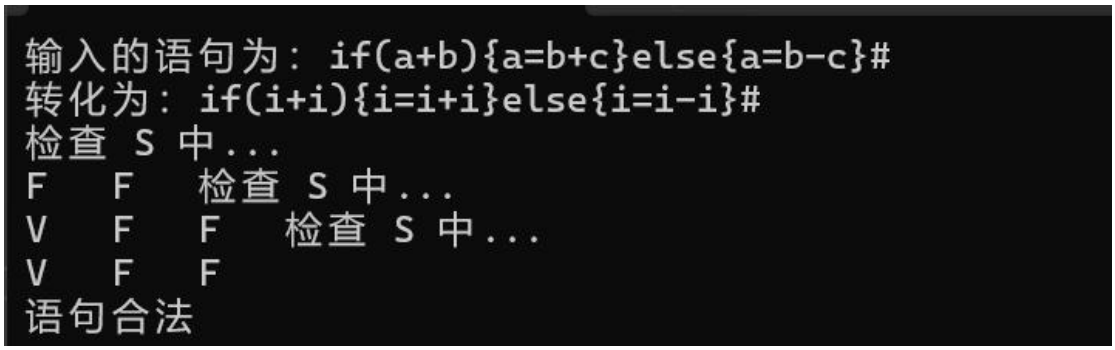
（5） 错误处理

在每个步骤（如括号匹配、关键字匹配、大括号匹配）都加入了错误检查。

如果解析过程中遇到缺少括号或无效字符，设置 SIGN = 1 表示错误，并输出相应的错误信息。

测试用例：if (a+b) { a=b+c } else { a=b-c }

程序运行截图：



6.附录：完整代码

两版完整代码都在文件夹的 cpp 文件中。

```
#define _CRT_SECURE_NO_WARNINGS 1
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<iostream>
#include <cassert>
#include<fstream>
#define MAX 1024
using namespace std;

char s[500];
int i;
int SIGN;// 用来标记语句是否正确

void S();
void E();
void E_();
void T();
void T_();
void F();
void A();
void M();
void V();

void S()
```

```
{
// 仅当未检测到错误时继续执行
if (SIGN != 0) return;

printf("检查 S 中...\n");

// 尝试识别一个变量
if (s[i] == 'i') {
    V(); // 处理变量的逻辑
} 
else {
    // 如果不是变量，标记错误并返回
    SIGN = 1;
    cout << "错误：在 S 处缺少变量" << endl;
    return;
}

// 检查变量后是否紧跟着 '='
if (SIGN == 0 && s[i] == '=') {
    i++; // 跳过 '='
    E(); // 解析等号后的表达式
}
else {
    // 如果 '=' 缺失，标记错误
    SIGN = 1;
    cout << "错误：在 S 处缺少'='" << endl;
}
}
```

```
54.
55.
56. void E()
57. {
58.  if (SIGN == 0) {
59.   printf("E   ");
60.   if (s[i] == '(' || s[i] == 'i') {
61.    T();
62.    if (SIGN == 0) {
63.     if (s[i] == '+' || s[i] == '-') {
64.      E_();
65.     }
66.     else if (s[i] == ')' || s[i] == '#') {
67.      return;
68.     }
69.     else {
70.      SIGN = 1;
71.      cout << "E 错误" << endl;
72.     }
73.    }
74.   }
75.   else {
76.    SIGN = 1;
77.    cout << "E 错误" << endl;
78.   }
79.  }
80. }
81.
82. void E_()
```

```
{
// 先检查是否有语法错误
if (SIGN != 0) return;
printf("E'   ");

// 判断当前字符是否为 '+' 或 '-'（表示有后续的表达式）
while (s[i] == '+' || s[i] == '-') {
    A(); // 解析操作符
    if (SIGN != 0) return;

    // 解析下一个项（T）
    if (s[i] == '(' || s[i] == 'i') {
        T(); // 解析项
    }
    else {
        SIGN = 1;
        cout << "错误：在 E' 中缺少有效的项（T）" << endl;
        return;
    }

    // 继续检查是否存在后续的运算符
    if (SIGN != 0) return;
}

// 检查 E' 的结束条件，应该为 ')' 或 '#'
if (s[i] != ')' && s[i] != '#') {
    SIGN = 1;
    cout << "错误：在 E' 中遇到无效字符" << endl;
```

```
}
}

void T()
{
// 先检查是否有语法错误
if (SIGN != 0) return;
printf("T    ");

// 判断当前字符是否为 '(' 或 'i'（表示有有效的因子）
if (s[i] == '(' || s[i] == 'i') {
    F(); // 解析因子
    if (SIGN != 0) return; // 如果解析失败则返回

    // 判断是否存在连续的 '*' 或 '/' 运算符
    while (s[i] == '*' || s[i] == '/') {
        T_(); // 解析乘除操作
        if (SIGN != 0) return; // 如果解析失败则返回
    }
}
else {
    // 起始字符无效，标记错误并返回
    SIGN = 1;
    cout << "错误：在 T 中缺少有效的因子（F）" << endl;
    return;
}

// 检查 T 的结束条件，应该为 '+'、'-'、')' 或 '#'
```

```
139. if (s[i] != '+' && s[i] != '-' && s[i] != ')' && s[i] != '#') {
140.  SIGN = 1;
141.  cout << "错误：在 T 中遇到无效字符" << endl;
142. }
143. }
144.
145.
146. void T_()
147. {
148. // 先检查是否有语法错误
149. if (SIGN != 0) return;
150. printf("T'   ");
151.
152. // 判断当前字符是否为 '*' 或 '/'（表示有后续的乘除操作）
153. while (s[i] == '*' || s[i] == '/') {
154.  M(); // 解析乘除操作符
155.  if (SIGN != 0) return; // 如果解析失败则返回
156.
157.  // 解析接下来的因子（F）
158.  if (s[i] == '(' || s[i] == 'i') {
159.   F(); // 解析因子
160.  }
161.  else {
162.   SIGN = 1;
163.   cout << "错误：在 T' 中缺少有效的因子（F）" << endl;
164.   return;
165.  }
166.
```

```
167.  // 检查是否继续处理连续的 '*' 或 '/' 运算符
168.  if (SIGN != 0) return;
169. }
170.
171. // 检查 T' 的结束条件，应该为 '+'、'-'、')' 或 '#'
172. if (s[i] != '+' && s[i] != '-' && s[i] != ')' && s[i] != '#') {
173.  SIGN = 1;
174.  cout << "错误：在 T' 中遇到无效字符" << endl;
175. }
176. }
177.
178.
179. void F()
180. {
181. // 先检查是否有语法错误
182. if (SIGN != 0) return;
183. printf("F   ");
184.
185. // 判断当前字符是否为 '('，表示可能是一个括号内的表达式
186. if (s[i] == '(') {
187.  i++; // 跳过 '('
188.
189. // 解析括号内的表达式
190.  if (s[i] == '(' || s[i] == 'i') {
191.   E(); // 调用 E 解析括号内的内容
192.  }
193.  else {
194.   SIGN = 1;
```

```
195.    cout << "错误：在 F 中缺少有效的表达式" << endl;
196.    return;
197.  }
198.
199.  // 检查是否有匹配的 ')'
200.  if (SIGN == 0 && s[i] == ')') {
201.    i++; // 跳过 ')'
202.  }
203.  else {
204.    SIGN = 1;
205.    cout << "错误：在 F 中缺少右括号 ')'" << endl;
206.    return;
207.  }
208. }
209. // 如果当前字符是 'i'，表示一个变量
210. else if (s[i] == 'i') {
211.  i++; // 跳过变量 'i'
212. }
213. else {
214.  // 当前字符既不是 '(' 也不是 'i'，标记错误
215.  SIGN = 1;
216.  cout << "错误：在 F 中遇到无效字符" << endl;
217. }
218. }
219.
220.
221. void A()
222. {
223.  if (SIGN == 0) {
```

```
224.  printf("A   ");
225.  if (s[i] == '+' || s[i] == '-') {
226.   i++;
227.  }
228.  else {
229.   SIGN = 1;
230.   cout << "A 错误" << endl;
231.  }
232. }
233. }
234.
235. void M()
236. {
237.  if (SIGN == 0) {
238.  printf("M   ");
239.  if (s[i] == '*' || s[i] == '/') {
240.   i++;
241.  }
242.  else {
243.   SIGN = 1;
244.   cout << "M 错误" << endl;
245.  }
246. }
247. }
248.
249. void V()
250. {
251.  if (SIGN == 0) {
252.  printf("V   ");
```

```
253.  if (s[i] == 'i') {
254.   i++;
255.  }
256.  else {
257.   SIGN = 1;
258.   cout << "V 错误" << endl;
259.  }
260. }
261. }
262.
263. int main()
264. {
265.  FILE* fp;
266.  string str, med;
267.  char arr[MAX];
268.
269.  //printf("输入语句为：\n");
270.
271.  if ((fp = fopen("D:/lexical_analysis_output.txt", "r")) != NULL) {
272.   while (fgets(arr, MAX, fp) != NULL)
273.   {
274.    int len = strlen(arr);
275.    arr[len - 1] = '\0';
276.    //printf("%s \n", arr);
277.    int flag = 0;//标记用于引号检测
278.    if (arr[1] == '2') //读到变量时
279.    {
280.     med += 'i';
```

```
281.
282.    }
283.
284.    for (int i = 0; i < len; i++)
285.    {
286.     if (arr[i] == ',' && flag == 0) //找到逗号
287.     {
288.      i++;
289.      while (arr[i] != '\0') {
290.       str += arr[i];
291.       if (arr[1] != '2') {
292.        med += arr[i];
293.       }
294.       if (arr[i + 2] == '\0')//最后一个字符
295.        break;
296.       i++;
297.      }
298.     }
299.    }
300.   }
301.  }
302.  str += '#';
303.  med += '#';
304.  fclose(fp);
305.
306.  cout << "输入的语句为：" << str << endl;
307.  cout << "转化为：" << med << endl;
308.
```

```
SIGN = 0;
i = 0;

// 将 like 转换为字符数组 s
strncpy(s, med.c_str(), sizeof(s) - 1); // 使用 strncpy 进行安全复制
s[sizeof(s) - 1] = '\0'; // 确保字符串以 '\0' 结尾

// 检查输入是否只有结束符 '#'
if (s[0] == '#') {
 return 0;
}

// 调用解析函数 S 进行语法分析
S();

// 检查分析结果，确定语句是否合法
if (s[i] == '#' && SIGN == 0) {
 printf("\n 语句合法\n");
}
else {
 printf("\n 语句不合法\n");
}

return 1;
}
```