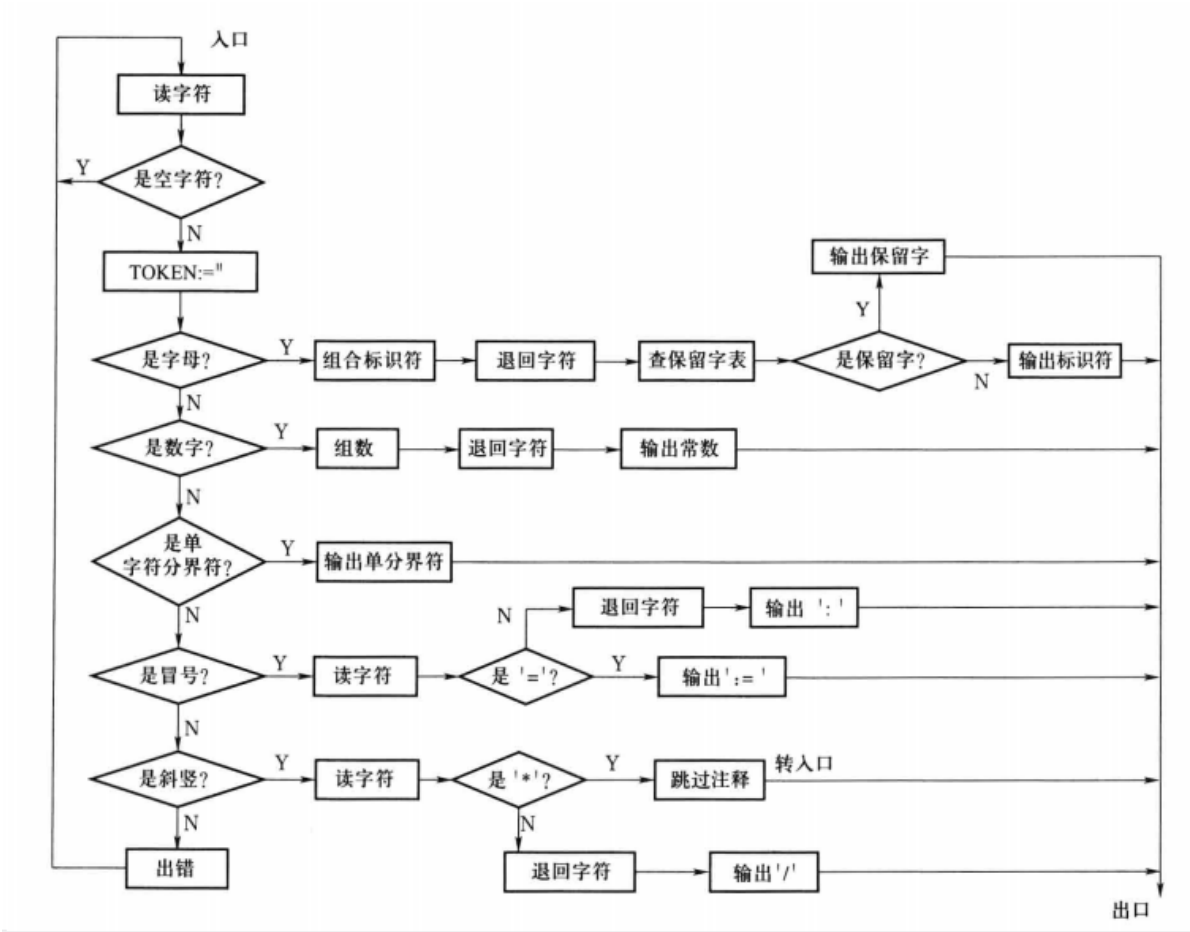
2021011587 计科21-2 吴维皓

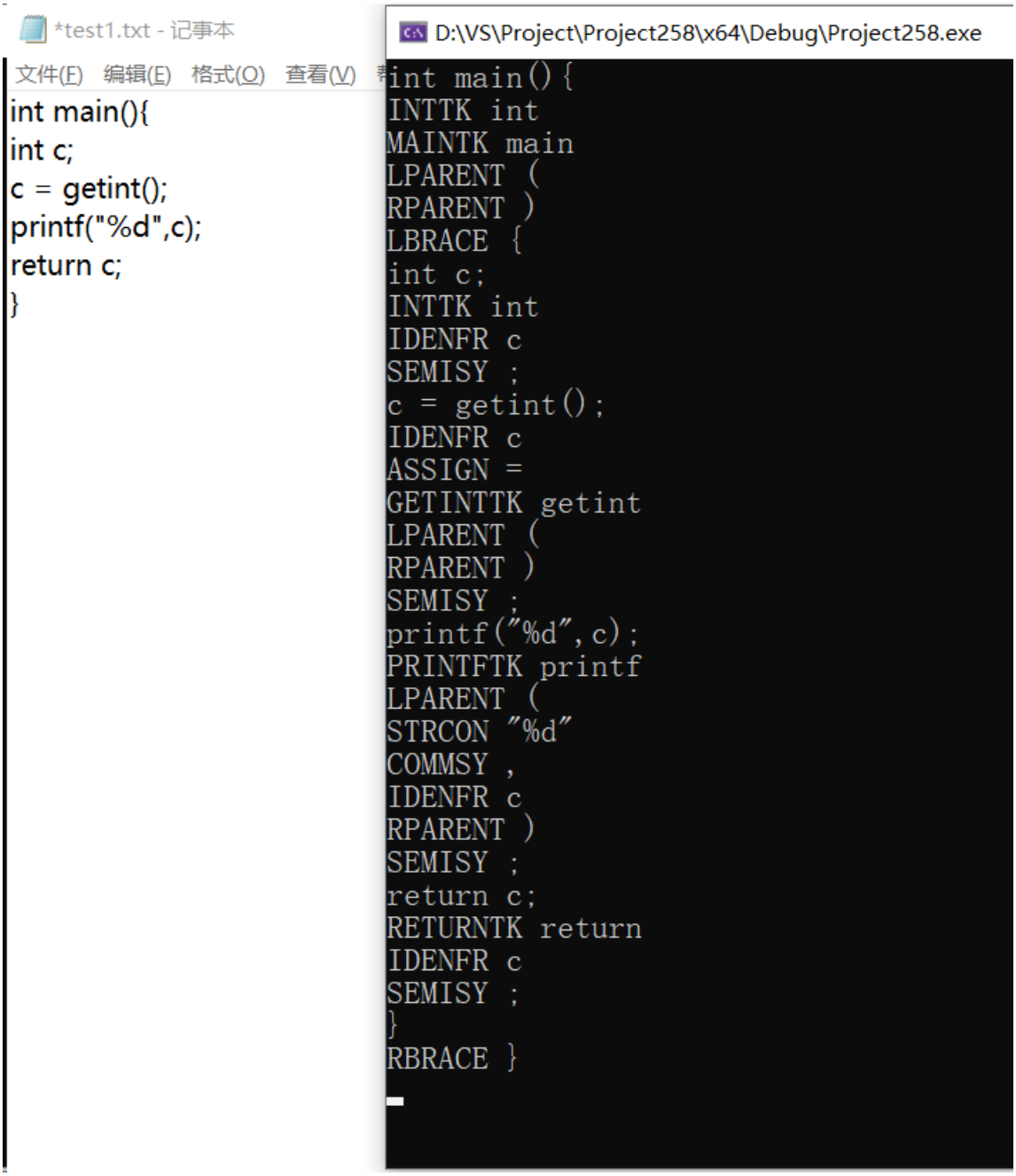
词法分析：

实现方式：定义一个`string`类型的`token`变量，一个`string`类型的`symbol`变量，它们分别表示单词符和类型。在`main`函数中，是一个无限的`while`循环，用来获取输入的字符串并进行词法分析，最后输出`token`和对应类型。在`getsym()`中，每次读取一个输入的字符，并对字符的类型进行判断后与`token`进行拼接。

代码流程图：



输入和结果：


*test1.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V)
int main(){
int c;
c = getint();
printf("%d",c);
return c;
}
D:\VS\Project\Project258\x64\Debug\Project258.exe
int main(){
INTTK int
MAINTK main
LPARENT (
RPARENT)
LBRACE {
int c;
INTTK int
IDENFR c
SEMISY ;
c = getint();
IDENFR c
ASSIGN =
GETINTTK getint
LPARENT (
RPARENT)
SEMISY ;
printf("%d",c);
PRINTFTK printf
LPARENT (
STRCON "%d"
COMMSY ,
IDENFR c
RPARENT)
SEMISY ;
return c;
RETURNTK return
IDENFR c
SEMISY ;
}
RBRACE }

test1.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
int main(){
int a ,b = 32, c = 33, d;
if(c>b) a = 0;
else a = 1;
d = getint();
printf("%d",a); //打印c
return c;
}
```

D:\VS\Project\Project258\x64\Debug\Project258.exe

```
int main(){
INTTK int
MAINTK main
LPARENT (
RPARENT )
LBRACE {
int a ,b = 32, c = 33, d;
INTTK int
IDENFR a
COMMSY ,
IDENFR b
ASSIGN =
INTCON 32
COMMSY ,
IDENFR c
ASSIGN =
INTCON 33
COMMSY ,
IDENFR d
SEMISY ;
if(c>b) a = 0;
IFTK if
LPARENT (
IDENFR c
GRE >
IDENFR b
RPARENT )
IDENFR a
ASSIGN =
INTCON 0
SEMISY ;
else a = 1;
ELSETK else
IDENFR a
ASSIGN =
INTCON 1
SEMISY ;
d = getint();
IDENFR d
ASSIGN =
GETINTTK getint
LPARENT (
RPARENT )
```

```
SEMISY ;
printf(”%d”,a); //打印c
PRINTFTK printf
LPARENT (
STRCON ”%d”
COMMSY ,
IDENFR a
RPARENT )
SEMISY ;
return c;
RETURNTK return
IDENFR c
SEMISY ;
}
RBRACE }
```

源码：

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

char ch, ch1;
bool lch;
string token;
string symbol;
int num;

unordered_map<string, string> mp = {
    {"main", "MAINTK"},
    {"const", "CONSTTK"},
    {"int", "INTTK"},
    {"break", "BREAKTK"},
    {"continue", "CONTINUETK"},
    {"if", "IFTK"},
    {"else", "ELSETK"},
    {"!", "NOT"},
    {"&&", "ADN"},
    {"||", "OR"},
    {"while", "WHILETK"},
    {"getint", "GETINTTK"},
    {"printf", "PRINTFTK"},
    {"return", "RETURNTK"},
    {"+", "PLUS"},
    {"-", "MINUS"},
    {"void", "VOIDTK"},
    {"*", "MULT"},
    {"/", "DIV"},
    {"%", "MOD"},
    {"<", "LSS"},
    {"<=", "LEQ"},
```

```
    {">", "GRE"},
    {">=", "GEQ"},
    {"==", "EQL"},
    {"!=", "NEQ"},
    {"=", "ASSIGN"},
    {";", "SEMICN"},
    {",", "COMMA"},
    {"(", "LPARENT"},
    {")", "RPARENT"},
    {"[", "LBRACK"},
    {"]", "RBRACK"},
    {"{", "LBRACE"},
    {"}", "RBRACE"},
};

bool isSpace() // 判断是否为空格
{
    return ch == ' ';
}

bool isNewline() // 判断是否为换行符
{
    return ch == '\n';
}

bool isTab() // 判断是否为Tab
{
    return ch == '\t';
}

bool isLetter() // 判断是否为字母
{
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
        return true;
    else
        return false;
}

bool isDigit() // 判断是否为数字
{
    if (ch >= '0' && ch <= '9')
        return true;
    else
        return false;
}

bool isColon() // 判断是否为冒号
{
    return ch == ':';
}

bool isComma() // 判断是否为逗号
{
    return ch == ',';
}
```

```
91
92  bool isSemi() // 判断是否为分号
93  {
94      return ch == ';';
95  }
96
97  bool isEqu() // 判断是否为等号
98  {
99      return ch == '=';
100 }
101
102 bool isPlus() // 判断是否为加号
103 {
104     return ch == '+';
105 }
106
107 bool isMinus() // 判断是否为减号
108 {
109     return ch == '-';
110 }
111
112 bool isDivi() // 判断是否为除号
113 {
114     return ch == '/';
115 }
116
117 bool isStar() // 判断是否为星号（乘*）
118 {
119     return ch == '*';
120 }
121
122 bool isDouble()
123 {
124     return ch == '"';
125 }
126
127 void getsym()
128 {
129     token = "";
130     symbol = "";
131
132     // 读入字符，通过lch判断上一次有没有多读的字符
133     if (!lch)
134         ch = getchar();
135     else
136     {
137         ch = chl;
138         lch = 0;
139     }
140
141     while (isSpace() || isNewline() || isTab())
142         ch = getchar();
143     if (isLetter())
144     {
145         while (isLetter() || isDigit())
```

```
        {
            token += ch;
            ch = getchar(); // 会多读一个字符
        }

        // 处理多读的字符
        chl = ch;
        lch = 1;

        if (mp.count(token))
            symbol = mp[token];
        else
            symbol = "IDENFR";
    }
    else if (isDigit())
    {
        num = 0;
        while (isDigit())
        {
            token += ch;
            ch = getchar(); // 多读一个字符
        }

        // 处理多读的字符
        chl = ch;
        lch = 1;

        num = stoi(token);
        symbol = "INTCON";
    }
    else if (isColon())
    {
        token += ch;
        symbol = "COLON";
    }
    else if (isPlus())
    {
        token += ch;
        symbol = "PLUS";
    }
    else if (isMinus())
    {
        token += ch;
        symbol = "MINUS";
    }
    else if (isStar())
    {
        token += ch;
        symbol = "MULT";
    }
    else if (isComma())
    {
        token += ch;
        symbol = "COMMMA";
    }
```

```
    else if (isSemi())
    {
        token += ch;
        symbol = "SEMICN";
    }
    else if (ch == '>')
    {
        char ch_ = ch;
        token += ch;
        ch = getchar();
        if (ch == '=')
        {
            token += ch;
            symbol = mp[token];
        }
        else
        {
            // 处理多读的字符
            chl = ch;
            lch = 1;
            symbol = mp[token];
        }
    }
    else if (ch == '<')
    {
        char ch_ = ch;
        token += ch;
        ch = getchar();
        if (ch == '=')
        {
            token += ch;
            symbol = mp[token];
        }
        else
        {
            // 处理多读的字符
            chl = ch;
            lch = 1;
            symbol = mp[token];
        }
    }
    else if (ch == '=')
    {
        char ch_ = ch;
        token += ch;
        ch = getchar();
        if (ch == '=')
        {
            token += ch;
            symbol = mp[token];
        }
        else
        {
            // 处理多读的字符
            chl = ch;
```

```
            lch = 1;
            symbol = mp[token];
        }
    }
    else if (ch == '&')
    {
        token += ch;
        ch = getchar();
        symbol = mp["&&"];
    }
    else if (ch == '|')
    {
        token += ch;
        ch = getchar();
        symbol = mp["||"];
    }
    else if (ch == '(')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == ')')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == '[' || ch == '{')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == ']' || ch == '}')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (isDivi())
    {
        char ch_ = ch;
        ch = getchar();
        if (isStar()) // 判断注释
        {
            while (ch != '\n')
                ch = getchar();
        }
        else if (isDivi())
        {
            while (ch != '\n')
                ch = getchar();
        }
        else
        {
            token += ch_;
            symbol = "DIVI";
        }
```

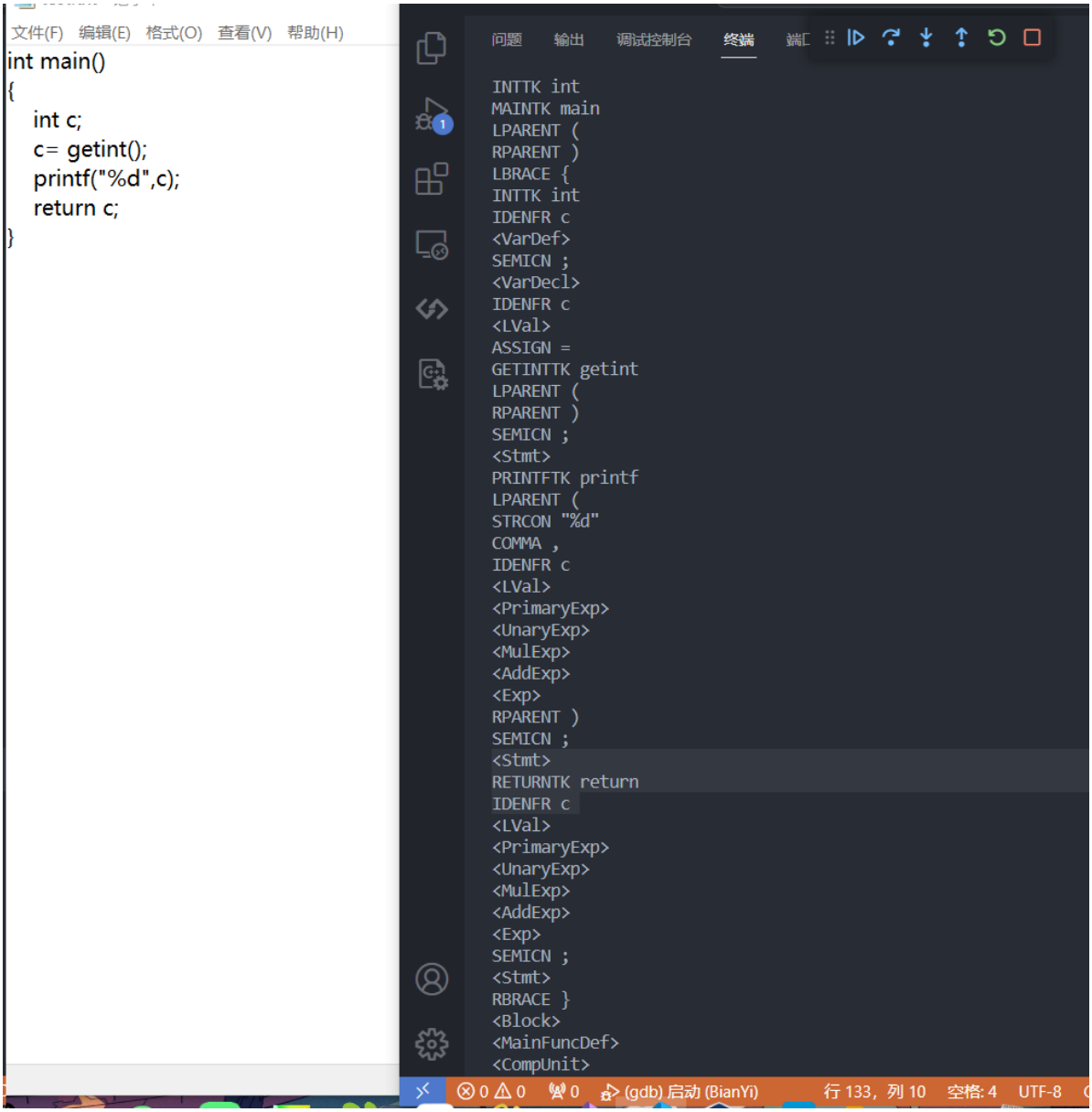
```
    }
    else if (isDouble())
    {
        do
        {
            token += ch;
            ch = getchar();
        } while (!isDouble());
        token += ch;
        symbol = "STRCON";
    }
}

int main()
{
    while (1)
    {
        getsym();
        if (symbol == "INTCON")
            cout << symbol << " " << num << endl;
        else if (ch != '\n') // 处理到注释时，ch最终读到的是\n，此时symbol和token都是空，所以要跳过
            cout << symbol << " " << token << endl;
    }
    return 0;
}
```

语法分析

实现方法：首先对整个输入程序进行词法分析，将得到的每个token等信息用链表连接。之后在语法分析时对链表从头到尾的顺序进行分析。

说明：语法分析中用到的词法分析代码和上面不同，因为一开始写词法分析的时候就每个token都getchar获取，但这种操作是不可逆的，不利于语法分析时的向前试探和预判，所以在语法分析程序中就不再使用上述代码，但大部分内容一样。

输入和结果：




```cpp
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <cctype>
#include <algorithm>
#include <unordered_map>
#include "head.h"
using namespace std;

struct Node
{
    string Tok;
    string sym;
    int tnum;
    struct Node *next;
    struct Node *pre;
};
struct Node *head = NULL;
```

```cpp
struct Node *nowtok;      // 用于语法分析
char ch, chl, chne = '@'; // ch是现在要分析的字符，chl是多读到的字符，nech是下一个待分析的字符
bool lch;
string fileContent;
string token;
string symbol;
int num, idx;

unordered_map<string, string> mp = {
    {"main", "MAINTK"},
    {"const", "CONSTTK"},
    {"int", "INTTK"},
    {"break", "BREAKTK"},
    {"continue", "CONTINUETK"},
    {"if", "IFTK"},
    {"else", "ELSETK"},
    {"!", "NOT"},
    {"&&", "ADN"},
    {"||", "OR"},
    {"while", "WHILETK"},
    {"getint", "GETINTTK"},
    {"printf", "PRINTFTK"},
    {"return", "RETURNTK"},
    {"+", "PLUS"},
    {"-", "MINUS"},
    {"void", "VOIDTK"},
    {"*", "MULT"},
    {"/", "DIV"},
    {"%", "MOD"},
    {"<", "LSS"},
    {"<=", "LEQ"},
    {">", "GRE"},
    {">=", "GEQ"},
    {"==", "EQL"},
    {"!=", "NEQ"},
    {"=", "ASSIGN"},
    {";", "SEMICN"},
    {",", "COMMA"},
    {"(", "LPARENT"},
    {")", "RPARENT"},
    {"[", "LBRACK"},
    {"]", "RBRACK"},
    {"{", "LBRACE"},
    {"}", "RBRACE"},
};

void go()
{
    nowtok = nowtok->next;
    if (nowtok->sym == "INTCON")
        cout << nowtok->sym << " " << nowtok->tnum << endl;
    else
        cout << nowtok->sym << " " << nowtok->Tok << endl;
}
```

```
75
76  void appendNode(struct Node **head)
77  {
78      // 创建新节点
79      // struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
80      struct Node *newNode = new struct Node;
81      if (newNode == NULL)
82          return;
83      newNode->Tok = token;
84      newNode->sym = symbol;
85      newNode->tnum = num;
86      newNode->next = NULL;
87      newNode->pre = NULL;
88
89      // 如果链表为空，新节点即为头节点
90      if (*head == NULL)
91      {
92          *head = newNode;
93      }
94      else
95      {
96          // 否则，遍历链表找到末尾，将新节点连接到末尾
97          struct Node *current = *head;
98          while (current->next != NULL)
99          {
100             current = current->next;
101         }
102         current->next = newNode;
103         newNode->pre = current;
104     }
105 }
106
107 void freeList(struct Node *head)
108 {
109     struct Node *current = head;
110     struct Node *next;
111
112     while (current != NULL)
113     {
114         next = current->next;
115         delete (current);
116         current = next;
117     }
118 }
119
120 void CompUnit()
121 {
122     nowtok = head;
123     if (nowtok->sym == "INTCON")
124         cout << nowtok->sym << " " << nowtok->tnum << endl;
125     else
126         cout << nowtok->sym << " " << nowtok->Tok << endl;
127
128     while (1)
129     {
```

```
        if (nowtok->sym == "CONSTTK")
        {
            Decl();
        }
        else if (nowtok->sym == "VOIDTK") // 变量定义
        {
            FuncDef();
        }
        else // INTTK
        {
            go();
            if (nowtok->sym == "MAINTK") // 主函数
            {
                MainFuncDef();
                break;
            }
            else if (nowtok->sym == "IDENFR") // 变量定义
            {
                Decl();
            }
        }
    }
    cout << "<CompUnit>" << endl;
}

void Decl()
{
    if (nowtok->sym == "CONSTTK")
    {
        ConstDecl();
    }
    else
    {
        VarDecl();
    }
    // cout << "<Decl>" << endl;
}

void MainFuncDef()
{
    go(); // 取(
    go(); // 取）
    Block();
    cout << "<MainFuncDef>" << endl;
}

void ConstDecl()
{
    BType();
    ConstDef();
    // 判断是否定义多变量
    while (nowtok->next->sym == "COMMA")
    {
        ConstDef();
        go(); // 最后一次读到分号;
```

```
    }
    cout << "<ConstDecl>" << endl;
}

void VarDecl()
{
    BType();
    VarDef();
    while (nowtok->next->sym == "COMMA")
    {
        VarDef();
    }
    go(); // 取;
    cout << "<VarDecl>" << endl;
}

void BType()
{
    if (nowtok->sym != "INTTK")
        go(); // 获取int
    // cout << "<BType>" << endl;
}

void ConstDef()
{
    go(); // 获取Ident
    go(); // 获取等号，假设不是数组
    ConstInitVal();
    cout << "<ConstDef>" << endl;
}
void Ident() {}
void ConstExp()
{
    AddExp(); // CONST
    cout << "<ConstExp>" << endl;
}
void ConstInitVal()
{
    ConstExp();
    cout << "<ConstInitVal>" << endl;
}
void VarDef()
{
    go(); // 取变量名
    if (nowtok->next->sym == "ASSIGN")
        InitVal();
    cout << "<VarDef>" << endl;
}
void InitVal()
{
    Exp();
    cout << "<InitVal>" << endl;
}
void Exp()
{
```

```
    AddExp();
    cout << "<Exp>" << endl;
}
void FuncDef()
{
    // FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
    FuncType();
    go(); // 获取函数名Ident
    go(); // 获取左括号(

    if (nowtok->next->sym != "RPARENT") // 有参数
    {
        FuncFParams();
    }
    else
    {
        go(); // 取)
        Block();
    }
    cout << "<FuncDef>" << endl;
}

void FuncType()
{
    // FuncType → 'void' | 'int'
    go(); // 取函数类型
    cout << "<FuncType>" << endl;
}

void FuncFParams()
{
    FuncFParam();
    cout << "<FuncFParams>" << endl;
}

void FuncFParam()
{
    BType();
    cout << "<FuncFParam>" << endl;
}
void Block()
{
    go(); // 取左花括号{
    do
    {
        BlockItem();
    } while (nowtok->next->sym != "RBRACE");
    go(); // 取右花括号}
    cout << "<Block>" << endl;
}
void BlockItem()
{
    if (nowtok->next->sym == "CONSTTK" || nowtok->next->sym == "INTTK")
    {
        if (nowtok->next->sym == "CONSTTK")
```

```
            ConstDecl();
        else
            VarDecl();
    }
    else
    {
        Stmt();
    }
    // cout << "<BlockItem>" << endl;
}
void Stmt()
{
    if (nowtok->next->sym == "IDENFR")
    {
        // LVal '=' Exp ';'
        LVal();
        go(); // 取=
        if (nowtok->next->sym == "GETINTTK")
        {
            go(); // 取 getint
            go(); // 取(
            go(); // 取)
            go(); // 取;
        }
        else
        {
            Exp();
            go(); // 取;
        }
    }
    else if (nowtok->next->sym == "IFTK")
    {
        go(); // 取 if
        go(); // 取(
        Cond();
        go();                                   // 取)
        if (nowtok->next->sym == "ELSETK") // 判断有没有else
        {
            go(); // 取 else
            Stmt();
        }
        else
            Stmt();
    }
    else if (nowtok->next->sym == "WHILETK")
    {
        go(); // 取 while
        go(); // 取(
        Cond();
        go(); // 取)
        Stmt();
    }
    else if (nowtok->next->sym == "BREAKTK")
    {
        go(); // 取 break
```

```
350         go(); // 取;
351     }
352     else if (nowtok->next->sym == "CONTINUETK")
353     {
354         go(); // 取 continue
355         go(); // 取;
356     }
357     else if (nowtok->next->sym == "RETURNTK")
358     {
359         //'return' [Exp] ';'
360         go(); // 取 return
361         if (nowtok->next->sym != "SEMICN")
362             Exp();
363         go(); // 取;
364     }
365     else if (nowtok->next->sym == "PRINTFTK")
366     {
367         go();                                   // 取 printf
368         go();                                   // 取(
369         go();                                   // 取一个字符串
370         while (nowtok->next->sym == "COMMA") // 有参数
371         {
372             go(); // 取 ,
373             Exp();
374         }
375         go(); // 取 )
376         go(); // 取 ;
377     }
378     else if (nowtok->next->sym == "LBRACE") // BLOCK情况
379     {
380         Block();
381     }
382     else //[Exp] ';'
383     {
384         if (nowtok->next->sym != "SEMICN")
385             Exp();
386         go(); // 取;
387     }
388     cout << "<Stmt>" << endl;
389 }
390 void LVal()
391 {
392     if (nowtok->sym != "IDENFR") // 看是否已经取到了变量名
393         go();
394     cout << "<LVal>" << endl;
395 }
396 void Cond()
397 {
398 }
399 void FormatString() {}
400 void AddExp()
401 {
402     MulExp();
403     cout << "<AddExp>" << endl;
404 }
```

```
void LOrExp() {}
void PrimaryExp()
{
    if (nowtok->next->sym == "IDENFR")
        LVal();
    else
        Number();
    cout << "<PrimaryExp>" << endl;
}
void Number()
{
    IntConst();
    cout << "<Number>" << endl;
}
void IntConst()
{
    go(); // 获取一个整数
    cout << "<IntConst>" << endl;
}
void UnaryExp()
{
    PrimaryExp();
    cout << "<UnaryExp>" << endl;
}

void UnaryOp() {}
void FuncRParams() {}
void MulExp()
{
    UnaryExp();
    cout << "<MulExp>" << endl;
}
void RelExp() {}
void EqExp() {}
void LAndExp() {}
void identifier() {}
void identifier_nondigit() {}
void digit() {}
void integer_const() {}
void decimal_const() {}
void nonzero_digit() {}

bool isSpace() // 判断是否为空格
{
    return ch == ' ';
}

bool isNewline() // 判断是否为换行符
{
    return ch == '\n';
}

bool isTab() // 判断是否为Tab
{
    return ch == '\t';
```

```
}

bool isLetter() // 判断是否为字母
{
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
        return true;
    else
        return false;
}

bool isDigit() // 判断是否为数字
{
    if (ch >= '1' && ch <= '9')
        return true;
    else
        return false;
}

bool isColon() // 判断是否为冒号
{
    return ch == ':';
}

bool isComma() // 判断是否为逗号
{
    return ch == ',';
}

bool isSemi() // 判断是否为分号
{
    return ch == ';';
}

bool isEqu() // 判断是否为等号
{
    return ch == '=';
}

bool isPlus() // 判断是否为加号
{
    return ch == '+';
}

bool isMinus() // 判断是否为减号
{
    return ch == '-';
}

bool isDivi() // 判断是否为除号
{
    return ch == '/';
}

bool isStar() // 判断是否为星号（乘*）
{
```

```
    return ch == '*';
}

bool isDouble()
{
    return ch == '"';
}

void Lexical()
{

    token = "";
    symbol = "";
    num = 0;

    ch = fileContent[idx];
    while (isSpace() || isNewline() || isTab())
        ch = fileContent[++idx];
    if (isLetter())
    {
        while (isLetter() || isDigit())
        {
            token += ch;
            ch = fileContent[++idx]; // 会多读一个字符
        }
        idx--; // 减去多读的一个
        if (mp.count(token))
            symbol = mp[token];
        else
            symbol = "IDENFR";
    }
    else if (isDigit())
    {
        num = 0;
        while (isDigit())
        {
            token += ch;
            ch = fileContent[++idx]; // 多读一个字符
        }
        idx--; // 减去多读的一个
        num = stoi(token);
        symbol = "INTCON";
    }
    else if (isColon())
    {
        token += ch;
        symbol = "COLON";
    }
    else if (isPlus())
    {
        token += ch;
        symbol = "PLUS";
    }
    else if (isMinus())
    {
```

```
570         token += ch;
571         symbol = "MINUS";
572     }
573     else if (isStar())
574     {
575         token += ch;
576         symbol = "MULT";
577     }
578     else if (isComma())
579     {
580         token += ch;
581         symbol = "COMMA";
582     }
583     else if (isSemi())
584     {
585         token += ch;
586         symbol = "SEMICN";
587     }
588     else if (ch == '>')
589     {
590         char ch_ = ch;
591         token += ch;
592 
593         if (ch == '=')
594         {
595             token += ch;
596             symbol = mp[token];
597         }
598         else
599         {
600             // 处理多读的字符
601             chl = ch;
602             lch = 1;
603             symbol = mp[token];
604         }
605     }
606     else if (ch == '<')
607     {
608         char ch_ = ch;
609         token += ch;
610         ch = fileContent[idx + 1];
611         if (ch = fileContent[idx + 1] == '=')
612         {
613             token += ch;
614             symbol = mp[token];
615         }
616         else
617         {
618             symbol = mp[token];
619         }
620     }
621     else if (ch == '=')
622     {
623         char ch_ = ch;
624         token += ch;
```

```
        ch = fileContent[idx + 1];
        if (ch = fileContent[idx + 1] == '=')
        {
            token += ch;
            symbol = mp[token];
        }
        else
        {
            symbol = mp[token];
        }
    }
    else if (ch == '&')
    {
        token += ch;
        idx++;
        symbol = mp["&&"];
    }
    else if (ch == '|')
    {
        token += ch;
        idx++;
        symbol = mp["||"];
    }
    else if (ch == '(')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == ')')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == '[' || ch == '{')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (ch == ']' || ch == '}')
    {
        token += ch;
        symbol = mp[token];
    }
    else if (isDivi())
    {
        char ch_ = ch;
        ch = fileContent[idx + 1];
        if (isStar()) // 判断注释
        {
            while (ch != '\n')
            {
                ch = fileContent[++idx];
            }
            idx--;
        }
```

```
        else if (isDivi())
        {
            while (ch != '\n')
            {
                ch = fileContent[++idx];
            }
            idx--;
        }
        else
        {
            token += ch_;
            symbol = "DIV";
        }
    }
    else if (isDouble())
    {
        do
        {
            token += ch;
            ch = fileContent[++idx];
        } while (!isDouble());
        token += ch;
        symbol = "STRCON";
    }
    // if (symbol == "INTCON")
    //     cout << symbol << " " << num << endl;
    // else if (ch != '\n') //处理到注释时，ch最终读到的是\n，此时symbol和token都是空，所以要跳过
    //     cout << symbol << " " << token << endl;
}

int main()
{
    std::ifstream inputFile("test.txt");

    if (inputFile.is_open())
    {
        string line;

        // 逐行读取文件内容
        while (getline(inputFile, line))
        {
            // 去除每一行的空格和制表符
            // line.erase(remove_if(line.begin(), line.end(), ::isspace), line.end());
            fileContent += line;
        }

        inputFile.close();
    }
    // 词法分析
    for (; idx < fileContent.size(); idx++)
    {
        Lexical();
        appendNode(&head); // 将token添加到链表末尾
```

```
733     }
734     CompUnit();
735     freeList(head);
736     return 0;
737 }
738 
```