**高级语言源程序格式处理工具实验指导**

**一、词法分析说明：**

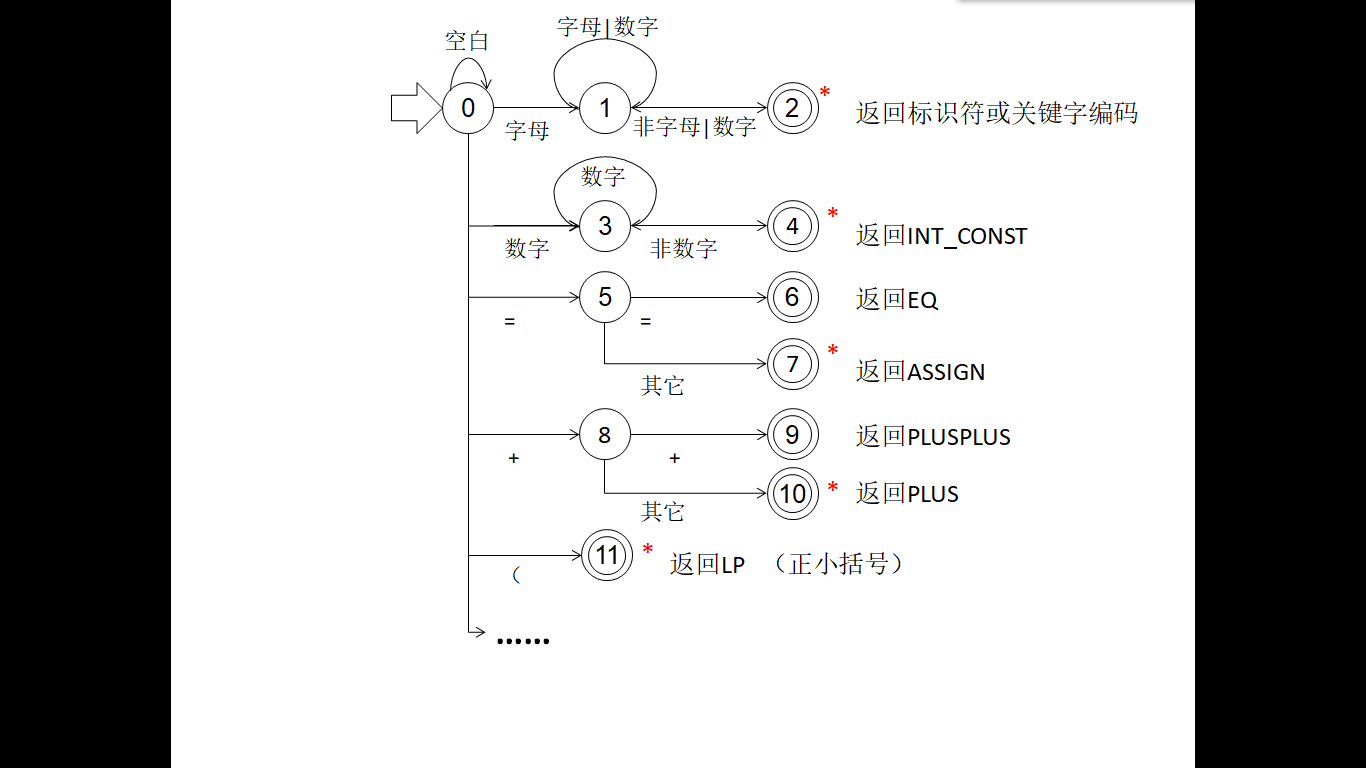
词法分析需要识别出五类单词，标识符、关键字、常量、运算符和定界符，词法分析每识别出一个单词，就可返回单词的编码。为唯一确定各单词的种类编码，可通过枚举类型定义各类单词的种类编号：

enum token\_kind {ERROR\_TOKEN, IDENT, INT\_CONST, FLOAT\_CONST, CHAR\_CONST, INT ,FLOAT,CHAR,IF,ELSE,..........EQ,ASSIGN, ......,LP,RP,...... SEMI，COMMA，......}

其中枚举常量IDENT是标识符符的种类编码，表示识别出来的单词是标识符，如识别出单词abc，def等标识符，就会返回IDENT；枚举常量INT\_CONST表示识别出来的是各种形式的整型常数，如123；INT表示识别出来的是关键字int；.... ;LP,RP分别表示左右括号，SEMI表示分号，COMMA表示逗号等等。通过这个枚举类型的定义，使用不同的枚举常量，即使用一个整数值表唯一地表示一个单词。

另外对有些单词，仅有种类编码是不够的，如标识符abc，词法分析返回IDENT，但标识符的字符串值abc需要保存在一个字符数组中token\_text，称为单词的自身值，token\_text是一个全局变量；同样123，词法分析返回INT\_CONST，数字字符串123要保存在token\_text中，以备后续处理。

词法分析的单词识别流程的过程状态转换图如下：



每次从状态0开始，从源程序文件中读取一个字符，可以到达下一个状态，当到达环形的状态（结束状态）时，表示成功的读取到了一个单词，返回单词的编码，单词自身值保存在全局变量token\_text中。结束状态上标有星号的，表示从源程序文件中多读取了一个字符，这个字符可能是下一个单词的一部分，需要退回到文件的输入缓冲区中。根据这个状态转换图，可以设计出如下词法分析的函数。每调用一次得到一个单词的种类码和自身值。

**int gettoken(源文件指针：fp) { // fp也可以作为全局变量，该函数不使用参数**

**初始化单词自身值**token\_text**为空;**

**while ((c=fgetc(fp))为空白符); //过滤掉空白符号，如果考虑报错位置，对回车需要单独**

**//处理，每次回车，设置一个行数计数器加1**

**if (c是字母) {**

**do {** token\_text**+c🡺**token\_text**}while ((c=fgetc(fp))是字母或数字) //拼标识符串**

**ungetc(c,fp); 退回多读的字符到文件输入缓冲区**

**标识符可能是关键字，需要判定并返回对应种类码，符号串在**token\_text**中**

**可以将所有关键字做成一个查找表，当标识符和某个关键字相等时，返回**

**关键字的种类编码，否则返回IDENT**

**}**

**if (c是数字) {**

**do {**token\_text**+c🡺**token\_text**}while ((c=fgetc(fp))是数字) //拼数字串**

**ungetc(c,fp); 退回多读的字符**

**数字串在**token\_text**中，返回INT\_CONST。**

**}**

**switch （c）{**

**case ‘=‘: c=fgetc(fp));**

**if (c==‘=‘) 返回相等运算符编码EQ；**

**ungetc(c,fp);**

**返回赋值运算符编码ASSIGN；**

**……….**

**default: if (feof(fp)) return EOF;**

**else return ERROR\_TOKEN; \\报错；错误符号**

**}**

**}**

实验时，为验证词法分析的正确性，每调用一次该函数，显示一个单词的信息，直到返回值为EOF为止。例如给定代码段：

**int a；**

**a=10；**

词法分析可显示单词信息：

**单词类别 单词值**

**关键字 int**

**标识符 a**

**分号 ；**

**标识符 a**

**赋值号 =**

**整型常量 10**

**分号 ；**

**二、相关语法分析**

语法分析，需要根据高级语言的语法规则分析程序的语法是否正确，如果正确，生成源程序的抽象语法树AST。

语法分析过程中，需要逐个读取源程序中的单词，具体实现时，可以使用2种方式，一种是一次性的读取源程序文件的所有单词，得到一个单词的线性表，每个数据元素保存单词的种类码和自身值。单词的线性表即可采用顺序表，也可以用链表方式，这样语法分析需要单词时，直接从线性表中取单词；第二种方式是语法分析时，需要单词时，调用一次词法分析的函数，读取一个单词。后续的有关说明是按第二种方式进行介绍。

设w为全局变量，存放当前读入的单词种类编码，token\_text保存单词的自身值。errors 表示错误标记，一旦有错，释放抽象语法树全部结点的空间。

首先要清楚高级语言按巴克斯（BNF）范式定义的语法规则，下面定义了一个很简单的语法规则，实验时各位同学自行进行扩展语言定义，尽可能地接近C语言（或你感兴趣的某种高级语言）的语法规则：

**<程序> ：：=<外部定义序列>**

**<外部定义序列>：：=<外部定义> <外部定义序列> | <外部定义>**

**<外部定义>：：=<外部变量定义>| <函数定义>**

**<外部变量定义>：：=<类型说明符> <变量序列> ；**

**<类型说明符>：：= int | float | char**

**<变量序列>：：=<变量> ， <变量序列> | <变量>**

**<函数定义>：：=<类型说明符> <函数名>（<形式参数序列>）<复合语句>**

**<形式参数序列>：：=<形式参数> ， <形式参数序列> | <空>**

**<形式参数>：：=<类型说明符> 标识符**

**<复合语句>：：={ <局部变量定义序列> <语句序列> }**

**<局部变量定义序列>：：=<局部变量定义> <局部变量定义序列> | <空>**

**<局部变量定义>：：=<类型说明符> <变量序列> ；**

**<语句序列>：：=<语句><语句序列> | <空>**

**<语句>：：= <表达式>； | return <表达式>；**

**| if （<表达式>）<语句>**

**| if （<表达式>）<语句> else <语句>**

**<表达式>：：=<表达式> + <表达式> | <表达式> - <表达式> |<表达式> \* <表达式>**

**|<表达式> / <表达式> | INT\_CONST | IDENT | IDENT(<实参序列>)**

**|<表达式> == <表达式> |<表达式> != <表达式> |<表达式> > <表达式>**

**|<表达式> > <表达式> |<表达式> >= <表达式> |<表达式> < <表达式>**

|<表达式> <= <表达式> | 标识符=<表达式>

<实参序列>：：=<表达式> <实参序列> | <空>

根据这个语言的语法规则定义，给出了下面的测试程序样例,通过这个例子解析语法器的处理流程的设计,理解语法分析器所要完成的任务。

**int i,j;**

**int fun(int a, float b)**

**{**

**int m;**

**if (a>b) m=a;**

**else m=b;**

**return m;**

**}**

**float x,y;**

实验中使用的语法分析算法，在编译技术中称为递归下降分析法，大量的使用了递归算法的设计，一般情况下，每一个语法单位（尖括号括起的部分）对应一个子程序，实验时可灵活处理。例如<程序>、<外部定义序列>、<外部定义>、<外部变量说明>，<函数定义>，<语句>，<表达式>等等。

每个子程序根据其语法规则完成相应处理，例如对语句的语法规则：

<语句>：：= <表达式>； | return <表达式>；

| if （<表达式>）<语句>

| if （<表达式>）<语句> else <语句>

语法单位<语句>对应的子程序，调用此子程序时，语句的第一个单词已经读入到了w中。由于有多种形式的语句，根据读入单词w的值，确定下一步处理什么样的语句，当读入的单词是RETURN时，处理返回语句，读入的单词是IF时，处理条件语句语句，其它情况表达式语句。

当处理返回语句时，接着调用语法单位<表达式>处理子程序，返回后，如当前读到的单词w是分号，则完成了返回语句的处理。其它条件语句，表达式语句类似处理。

下面就上述定义的语言的各个语法单位的处理程序进行说明。

**1. 语法单位<程序>的子程序**

对于一个程序，按其语法定义： <程序> ：：=<外部定义序列>

语法单位<程序>的子程序如下，完成的功能是生成一棵语法树，根指针指向的是一个外部定义序列的结点。

program(){

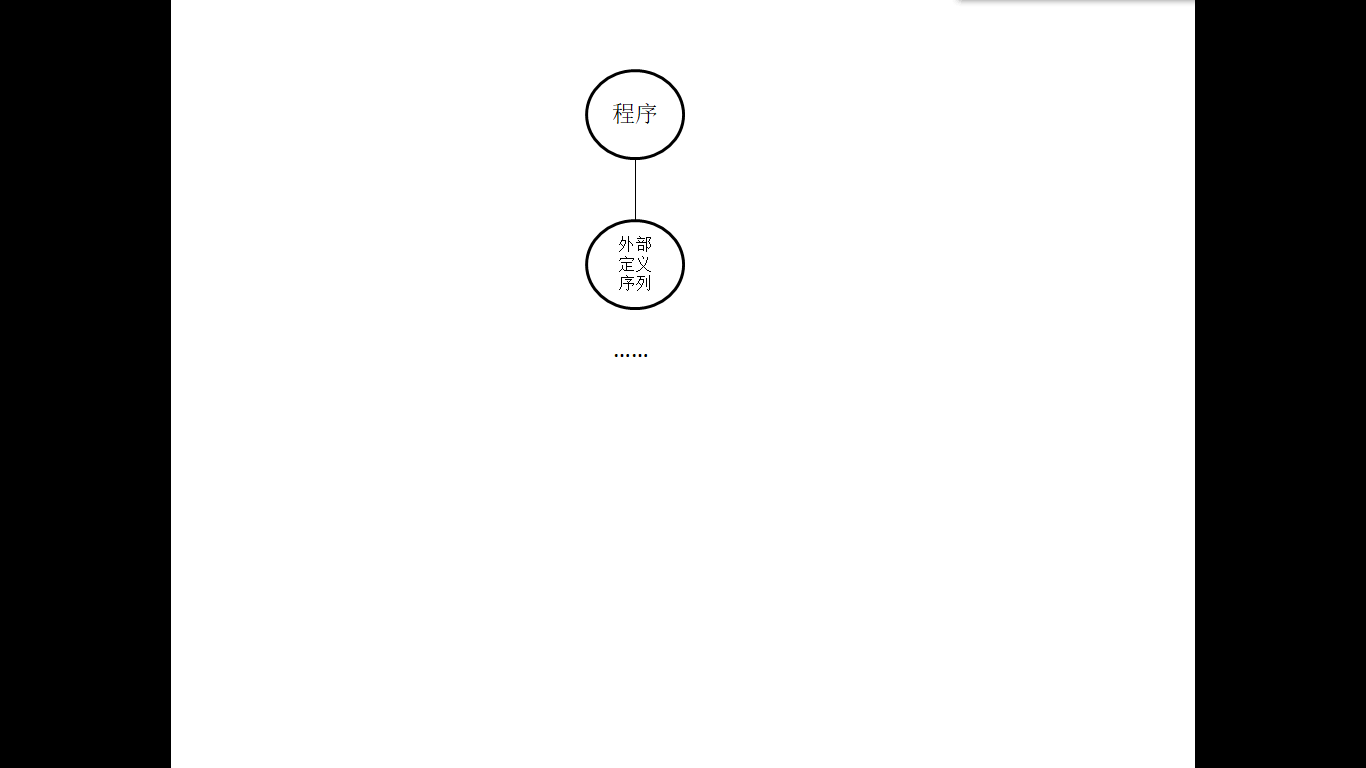
w=gettoken();

if ( ExtDefList() ) {程序语法正确，返回的语法树根结点指针，可遍历显示}

else 有语法错误

}

该子程序执行后，得到的AST如下，程序结点的子树是一个外部定义序列。



**2. 语法单位<外部定义序列>的子程序**

语法单位<外部定义序列>的定义：

**<外部定义序列>：：=<外部定义> <外部定义序列> | <外部定义>**

这是一个递归定义，该子程序处理一系列的外部定义，每个外部定义序列的结点，其第一个子树对应一个外部定义，第二棵子树对应后续的外部定义。

在一个源程序中，每次成功处理完一个外部定义后，如果遇到文件结束标记，则语法分析结束。调用此子程序，已经读入了一个外部定义的第一个单词到w中。

ExtDefList(){//处理外部定义序列，正确时，返回子树根结点指针，否则错误处理

if (w==EOF) return NULL；

生成一个外部定义序列结点root

ExtDef(); //处理一个外部定义，得到一棵子树，作为root的第一棵子树

ExtDefList(); //得到的子树，作为root的第二棵子树

return root；

}

**3. 语法单位<外部定义>的子程序**

此子程序完成一个外部定义的处理，调用此子程序时，已经读入了一个外部定义的第一个单词到w中。该子程序处理完后，刚好处理到外部定义的最后一个符号，后续单词还没读入。

不管是外部变量定义，还是函数定义，第一个单词必须是类型关键字，第二个一定是标识符，只有读入第三个才可能区分，如果是小括号，就是函数的定义与声明，调用函数定义子程序，否则按外部变量的形式来处理，调用外部变量定义子程序。

语法树逻辑上是一棵多叉树，其物理结构，既可简单直观地采用孩子表示法，也可以采用孩子兄弟表示法等，实验时自行确定。一棵语法树中有各种类型的结点，为统一管理所有结点，定义结点类型时，需要使用共用体的概念，通过一个标记说明该结点包含的信息（类似广义表的结点类型），明确每个孩子或子树的含义。处理外部定义（函数或外部变量）的处理流程可参考如下，这里ASTTree为抽象语法树结点指针类型：

ExtDef() { //处理外部定义序列，正确时，返回子树根结点指针，否则返回NULL

if (w不是类型关键字) 报错并返回NULL

保存类型说明符

w=gettoken();

if (w!=IDENT ) 报错并返回NULL

strcpy(tokenText0,tokenText); // 保存第一个变量名或函数名到tokenText0

w=gettoken();

if (w!=LP) p=ExtVar(); //调用外部变量定义子程序

else p=FuncDef(); //调用函数定义子程序

如果返回值p非空，表示成功完成一个外部定义的处理，返回p

}

对测试用例进行语法分析，当外部定义序列，外部定义(包括外部变量、函数定义)都处理完后，得到的抽象语法树的逻辑结构如下图显示，可见包含一系列的外部定义序列的结点，每个结点的第一棵子树分别对应一个外部变量的定义或函数定义。外部变量的定义或函数定义的处理在后面介绍。

外部定义序列

外部变量定义

函数定义

外部定义序列

外部变量定义

外部定义序列

……

i，j的定义i部分

……

x，y的定义i部分

……

函数定义部分

**4. 语法单位<外部变量定义>子程序**

调用此子程序时，**外部变量类型**和**第一个变量名**的单词已经读入，变量名保存在tokenText0中，这时外部变量定义的处理流程可参考如下。

**status ExtVarDef() {**

**root=生成外部变量定义结点；**

**根据已读入的外部变量的类型，生成外部变量类型结点，作为root的第一个孩子**

**p=生成外部变量序列结点，p作为root的第二个孩子，每个外部变量序列结点会对应一个变量名**

**由保存在tokenText0的第一个变量名生成一个变量名结点，作为p的第一个孩子；**

**w=gettoken(); //开始识别后续的变量名**

**while（1） { //每个外部变量序列结点的第一个孩子对应一个变量**

**if （w！='，' || w！='；'） 报错，释放root为根的全部结点，返回空指针**

**if （ w=='；'）{**

**w=gettoken();**

**返回根结点root；**

**}**

**w=gettoken();**

**if （ w不是标识符）报错，释放root为根的全部结点，返回空指针**

**生成外部变量序列结点，根指针为q，作为p的第二个孩子，插入到树中。**

**p=q；**

**根据tokenText的变量名生成一个变量结点，作为p的第一个孩子；**

**w=gettoken();**

**}**

**}**

对测试用例的第一行，得到的子树如下图。

外部变量定义

类型

int

i

外部变量序列

j

外部变量序列

实现时也可以按照语法规则的定义，将<变量序列>这个语法成分单独编制一个子程序ExtVarList。这时算法流程如下：

**ExtVarList(){ // 初始时，tokenText0保存了第一个变量名**

**root=生成外部变量序列结点**

**由保存在tokenText0的第一个变量名生成一个变量名结点，作为root的第一个孩子；**

**w=gettoken(); //开始识别后续的变量名**

**if （w！='，' || w！='；'） 报错，释放root为根的全部结点，返回空指针**

**if （ w=='；'）{**

**w=gettoken();**

**返回根结点root；**

**}**

**w=gettoken();**

**if （ w不是标识符）报错，释放root为根的全部结点，返回空指针**

**将变量名w保存在tokenText0中；**

**调用ExtVarList，得到的子树作为root的第二棵字数，返回root；**

**}**

ExtVarDef() {

root=生成一个外部变量定义结点；

**根据已读入的外部变量的类型，生成外部变量类型结点，作为root的第一个孩子**

**调用ExtVarList，得到的子树根作为root的第二个子树**

**返回root；**

**}**

**5. 语法单位<函数定义>子程序**

调用此子程序时，函数返回值类型和函数名，正小括号的单词已经读入，函数名保存在tokenText0中，这时函数定义的处理流程可参考如下：

**funcDef（） {**

**生成函数定义结点root；**

**生成返回值类型结点，作为root的第一个孩子**

**处理参数部分到反小括号结束，调用形参子程序，得到参数部分的子树根指针，无参函数得到NULL，该子树作为root的第二棵子树；**

**读入符号，如果分号，就是函数原型声明，函数体子树为空；正大括号，则调用函数体（复合语句）子程序，得到函数体子树根指针，其它符号表示有错。得到的函数体子树作为root的第三棵子树**

**返回root；**

**}**

对形参的处理，可参照外部变量的定义子程序。对测试用例处理完函数后，得到抽象语法树的子树如下图。

函数定义

int

函数体

fun

形参

形参序列

形参

形参序列

int

a

float

b

**6. 语法单位<复合语句>子程序**

调用此子程序时，已经读入了单词{，继续处理时，遇到遇到}，结束复合语句，算法流程如下：

**root=生成复合语句结点；注意其中局部变量说明和语句序列都可以为空**

**w=gettoken();**

**if （w是类型关键字）{ 调用处理局部变量说明序列子程序，**

**得到返回的子树根结点作为root的第一个孩子}**

**else { 无局部变量说明，root的第一个孩子设置为空指针}**

**调用处理语句序列子程序，返回子树根结点指针，作为root的第2个孩子**

**if （w不是反大括号} ）返回空指针，报错并释放结点**

**w=gettoken();**

**返回复合语句的子树根指针。**

对测试用例处理完函数体后，得到抽象语法树的子树如下图。

局部变量定义序列

复合语句

If-else语句

语句序列

局部变量定义

语句序列

Return

语句

……

表达式

……

int

变量名

序列

m

**7. 语法单位<语句序列>子程序**

要考虑语句序列为空的情况

**初始化子树，根指针root=NULL；**

**调用处理一条语句的子程序；返回其子树根指针r1；**

**if （r1==NULL）//没有分析到1条语句，errors>0时处理错误，**

**否则表示语句序列已结束**

**返回NULL**

**else { 生成语句序列的结点root**

**root->第1孩子=r1;**

**root->第2孩子=递归调用处理语句序列子程序后的返回值；**

**返回root；**

**}**

对测试用例处理完函数体复合语句的语句序列，得到的抽象语法树的子树如下图。

If-else语句

语句序列

语句序列

Return

语句

……

表达式

……

**8 语法单位<语句>的子程序**

调用此子程序时，语句的第一个单词已经读入，处理一条语句时，根据这条语句的第一个单词，确定处理什么类型的语句。如遇到关键字if，则处理条件语句，首先处理if （表达式） 语句1，完成后，再读入下一个单词，如果是else，则表示是if-then-else语句，否则就是if-then语句，注意体会是否能正确处理条件语句嵌套的二义性问题。

此子程序调用结束时，会读入下一条语句的第一个单词到w中，以便后续处理。

statement（）{

switch （w）{

case IF：//分析条件语句

w=gettoken();

if (w不是左小括号） 报错并返回空

调用处理表达式的子程序（结束符号为反小括号），正确时得到条件表达式子树结点指针

调用处理一条语句的子程序，得到IF子句的子树根指针。

if （w==ELSE）{调用处理一条语句的子程序，得到IF子句的子树根指针。

生成IF-ELSE结点，下挂条件、IF子句、ELSE子句3棵子树}

else 生成IF结点，下挂条件、IF子句2棵子树

case { ：调用处理复合语句子程序,返回得到的子树根指针

case WHILE：......

case (： //各种表达式语句，含赋值等，形式为表达式以分号结束

case 标识符：

case 常数：

调用表达式处理子程序（结束符号为分号）；

正确时，w=gettoken(); 返回表达式语句子树的根结点指针

case }: //语句序列结束符号，如果语言支持switch语句，

结束符号还会有case 和default

w=gettoken();

return NULL；

default: errors+=1，报错并返回NULL；

}

}

测试用例中条件语句的抽象语法树形式如下图所示。

If-else语句

m

赋值语句

>

b

a

b

m

赋值语句

a

其中赋值语句部分也可以采用如下形式，具体采用哪一种，可自行选择。

表达式语句

m

=

a

**9. 语法单位<表达式>子程序**

考虑表达式结束符号，处理表达式语句时以分号结束、，作为条件表达式时，反小括号）结束。该子程序处理完后，有2种情况，一是表达式有语法错误，这是，可终止语法分析程序的运行并报错，二是正确时刚好处理到表达式结束符号，后续单词还没读入。

一个正确的表达式连接上一个结束符号肯定不是正确的表达式，如：1+2）或1+2；。这样当分析一个表达式遇到错误时，如果刚读入的符号正好和结束符号相同，且前面分析的部分正好也是一个完整表达式，就表示表达式语法正确，否则表达式是真正意义上的语法错误。

表达式的语法处理，可借鉴与数据结构第3章的表达式求值，注意体会运算符优先关系表中的运算符的优先关系和结合性。增加了赋值、关系运算符后的运算符的优先关系表如下。剩下没有列出的运算符，如逻辑与，逻辑或，可以在理解这张表的基础上，自行增加上去。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **+** | **-** | **\*** | **/** | **（** | **）** | **=赋值** | **大小于** | **==和！=** | **#** |
| **+** | **>** | **>** | **<** | **<** | **<** | **>** |  | **>** | **>** | **>** |
| **-** | **>** | **>** | **<** | **<** | **<** | **>** |  | **>** | **>** | **>** |
| **\*** | **>** | **>** | **>** | **>** | **<** | **>** |  | **>** | **>** | **>** |
| **/** | **>** | **>** | **>** | **>** | **<** | **>** |  | **>** | **>** | **>** |
| **（** | **<** | **<** | **<** | **<** | **<** | **=** |  | **>** | **>** | **>** |
| **）** | **>** | **>** | **>** | **>** | **>** |  |  | **>** | **>** | **>** |
| **=赋值** | **<** | **<** | **<** | **<** | **<** |  | **<** | **<** | **<** | **>** |
| **大小于** | **<** | **<** | **<** | **<** | **<** | **>** |  | **>** | **>** | **>** |
| **==和！=** | **<** | **<** | **<** | **<** | **<** | **>** |  | **<** | **>** | **>** |
| **#** | **<** | **<** | **<** | **<** | **<** |  | **<** | **<** | **<** | **=** |

每当遇到一个操作数，生成一个结点，将结点指针进操作数栈，每当需要处理一个运算符，生成运算符的结点，把操作数的结点作为该结点的孩子。算法流程可参考如下：

**结点指针类型 exp(int endsym) //表达式结束符号endsym可以是分号，如表达式语句，**

**//可以是反小括号，作为条件时使用**

**{ //调用该算法时，在调用者已经读入了第一个单词在w中**

**定义运算符栈 op;并初始化，将起止符#入栈**

**定义操作数栈opn，元素是结点的指针**

**错误标记error设置为0**

**while ( (w!=# || gettop(op)!=#) && !error ) //当运算符栈栈顶不是起止符号，并没有错误时**

**{**

**if (w是标识符或常数等操作数时)**

**{ 根据w生成一个结点，结点指针进栈opn, w=gettoken();}**

**else if (w是运算符)**

**switch (precede[gettop(op)][w]) {**

**case '<': push(op,w);w=gettoken();break;**

**case '=':if (!pop(op,t)) error++; w=gettoken();break; //去括号**

**case '>':if (!pop(opn,t2)) error++;**

**if (!pop(opn,t1)) error++;**

**if (!pop(op,t)) error++;**

**根据运算符栈退栈得到的运算符t和操作数的结点指针t1和t2，**

**完成建立生成一个运算符的结点，结点指针进栈opn**

**break;**

**default: if (w==endsym) w=BEGIN\_END; //遇到结束标记），w被替换成#**

**else error=1;**

**}**

**else if (w==endsym) w=BEGIN\_END；//遇到结束标记分号，w被替换成#**

**else error=1;**

**}**

**if (操作数栈只有一个结点指针&& gettop(op)==# && 无错误)**

**return操作数栈唯一的这个结点指针; //成功返回表达式语法树的根结点指针**

**else return NULL; //表达式分析有错**

**}**

例如表达式语句a+b\*10；其表达式部分的抽象语法树如下图所示。作为表达式语句，可以考虑再增加一个类型为表达式语句的结点，其孩子指针指向该子树根结点。如前面**语法单位<语句>的子程序**部分所述。

+

b

\*

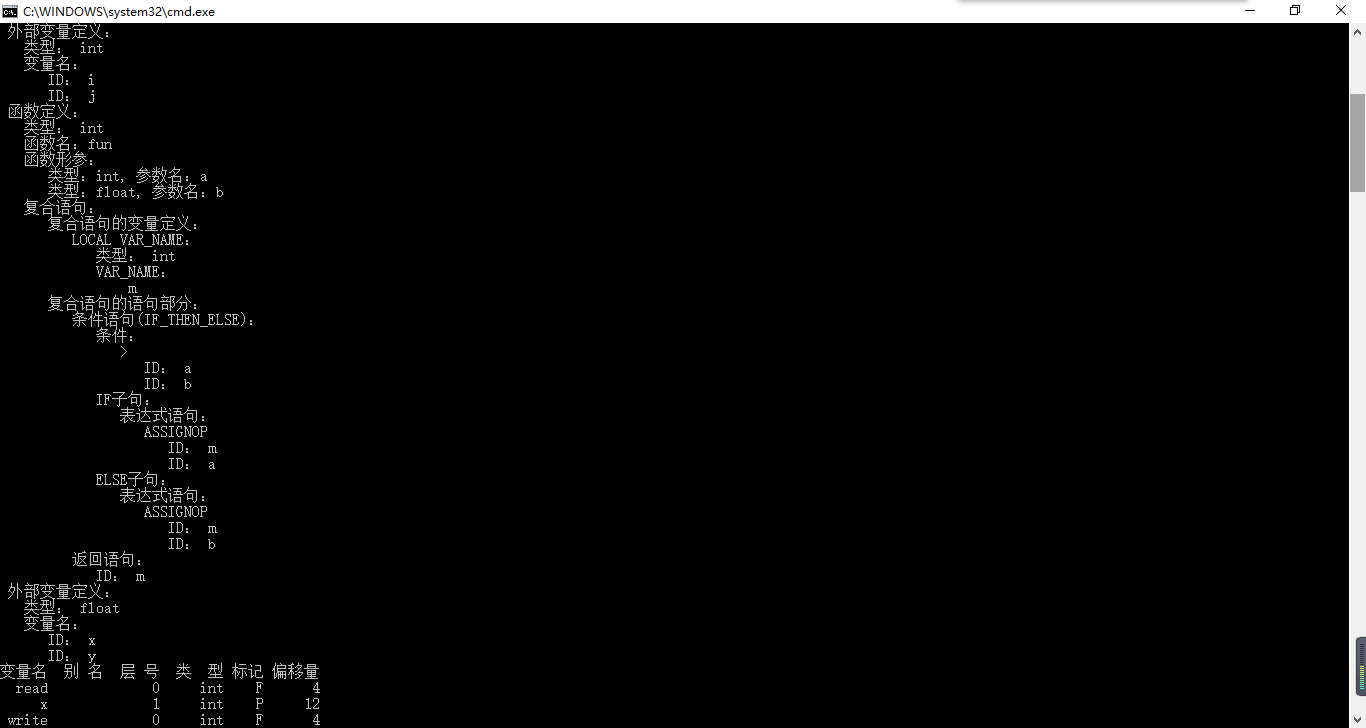
a

10

抽象语法树逻辑上是一棵多叉树，树中各种类型的结点混合在一起，为了区分各结点类型，正确访问各结点的属性，在定义树结点时，需要同时采用共用体与结构类型，来定义树结点的类型。

**三、语法树的显示与程序的格式化处理**

综上所述，可构造出一个源程序的抽象语法树，采用先根遍历的次序，显示抽象语法树，要求能很清晰的体现源程序个语法单位和抽象语法树的对应关系，能由抽象语法树方便地还原出源程序，抽象语法树的显示可参考下图。



同时通过对抽象语法树的遍历，生成风格统一的格式化缩进编排的源程序文件，具体格式可上网查阅相关资料，自行认定一种规范，参考如下。

**int i,j;**

**int fun(int a, float b)**

**{**

**int m;**

**if (a>b)**

**m=a;**

**else**

**m=b;**

**return m;**

**}**

**float x,y;**

**参考文献**

[1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第3版）. 北京：清华大学出版社. 前4章

[2] 严蔚敏等. 数据结构(C语言版). 清华大学出版社