**高级语言源程序格式处理工具**

**一、任务要求：**

1. 选定一个高级语言，例如C语言的一个简单子集，只要求基本数据类型，数组、结构等构造类型不做硬性要求

2. 构造词法分析,依据DFA的状态转换图，实验时给出DFA，并解释如何在状态迁移中完成单词识别，最终生成词法分析子程序。

3. 语法分析

（1）外部变量的申明

（2）函数声明

（3）局部变量的申明

（4）语句及表达式

**（5）生成语法树并显示**

**（6）按缩进编排得到源程序文件**

**二、词法分析说明：**

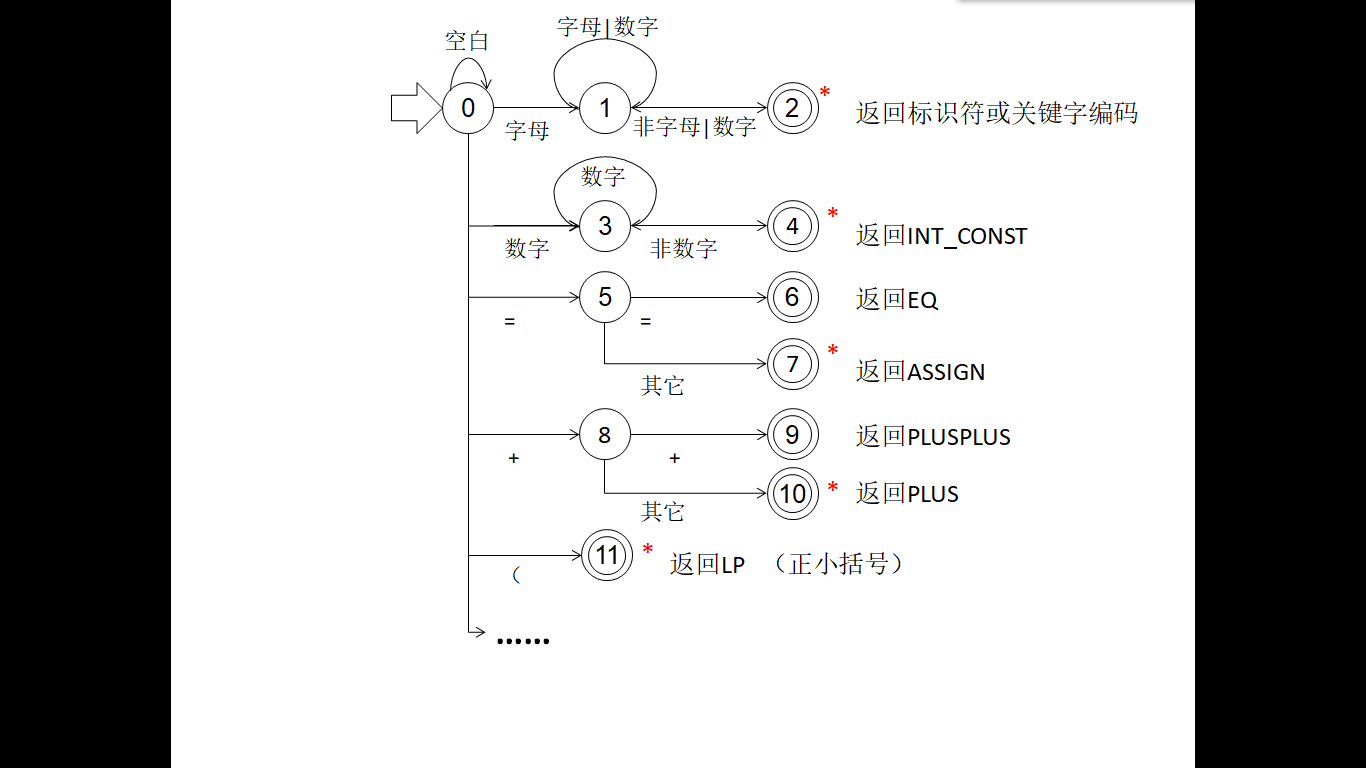
词法分析需要识别出五类单词，标识符、关键字、常量、运算符和定界符，返回单词的编码。为唯一确定各单词的种类编码，可通过枚举类型定义各类单词的种类编号：

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abc；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。

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。

表示词法分析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，可以设计出词法分析的函数：

**int gettoken(源文件指针：fp) {**

**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**

**}**

**}**

**三、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。设w为全局变量，存放当前读入的单词类别，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，实验时自行进行扩展，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> <局部变量序列> ；**

**<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。

**int i,j;**

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

**{**

**int m;**

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

**else m=b;**

**return m;**

**}**

**float x,y;**

语法分析的算法，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，每一个语法单位（尖括号括起的部分）对应一个子程序，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，由于有多种形式的语句，根据读入的单词w，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(){

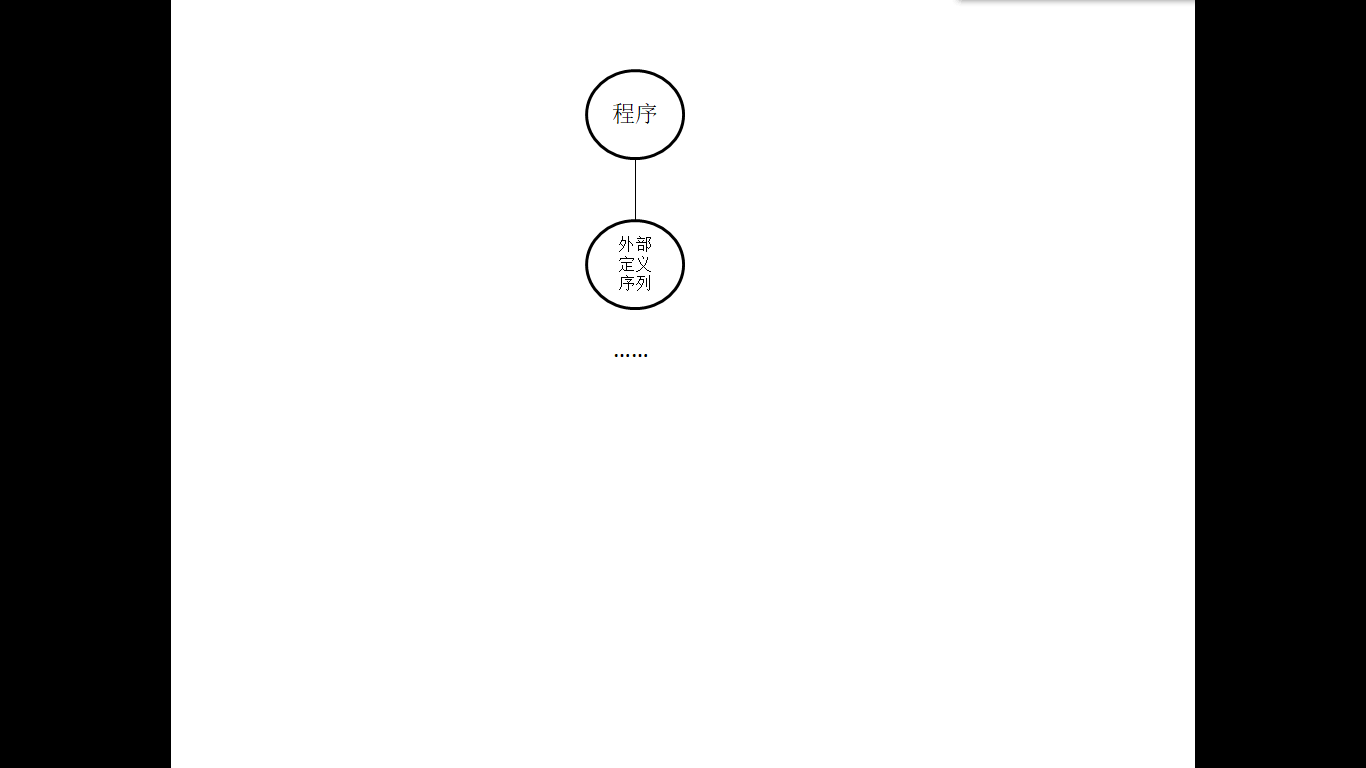
w=gettoken();

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。

ExtDefList(){

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

w=gettoken();

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**

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。

语法树的物理结构，可采用多叉树，也可以采用孩子兄弟表示法等物理结构，可自行确定。为统一所有结点类型，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，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：

**ASTTree ExtDef() { //处理外部定义序列**

**if (w==EOF) return NULL;**

**if (w不是类型关键字) 报错并返回**

**w=gettoken();**

**while (1){**

**if (w不是标识符) 报错并返回**

**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，生成一个外部定义序列**

**结点root，将p作为root的第一个子树**

**w=gettoken();**

**调用ExtDef()处理后续外部定义，正确时将得到的子树作为root的第二棵子树;**

**}**

**}**

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。

**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=='；'）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。

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=='；'）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tokenText中；**

**调用ExtVarList，得到的子树作为root的第二棵字数，但会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**

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。

**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**

**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。

**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；

while(1) {

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1条语句，可能是错误，或语句序列已结束

返回NULL

else { 生成语句序列的结点root（第1孩子对应1条语句，第2个对应语句序列）

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。

**8 语法单位<语句>的子程序**

switch （w）{

case IF：//条件语句

匹配(

处理表达式，以）结束，正确时得到条件表达式子树结点指针

处理一条语句

w=gettoken();

if （w==ELSE）处理一条语句，生成IF-ELSE结点

else生成IF结点

case { ：调用处理复合语句子程序

case WHILE：......

default：//各种表达式语句，含赋值等，形式为表达式以分号结束

处理表达式；

if （w！='；'） 报错，语句应该以分号结束

返回表达式语句子树的根结点指针

}

测试用例中条件语句的抽象语法树形式如下图所示。

**9. 语法单位<表达式>子程序**

考虑的结束形式，分号、），逗号，或遇到错误时结束

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优先关系和结合性。运算符的优先关系表如下,其中既体现了运算符的优先级，也体现了结合性。剩下没有列出的运算符，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，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) //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，这里简单就用字符**

**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++;**

**根据退栈出来的运算符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 (运算符栈只有一个结点指针&& gettop(op)==# && 无错误)**

**return运算符栈结点指针; //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**

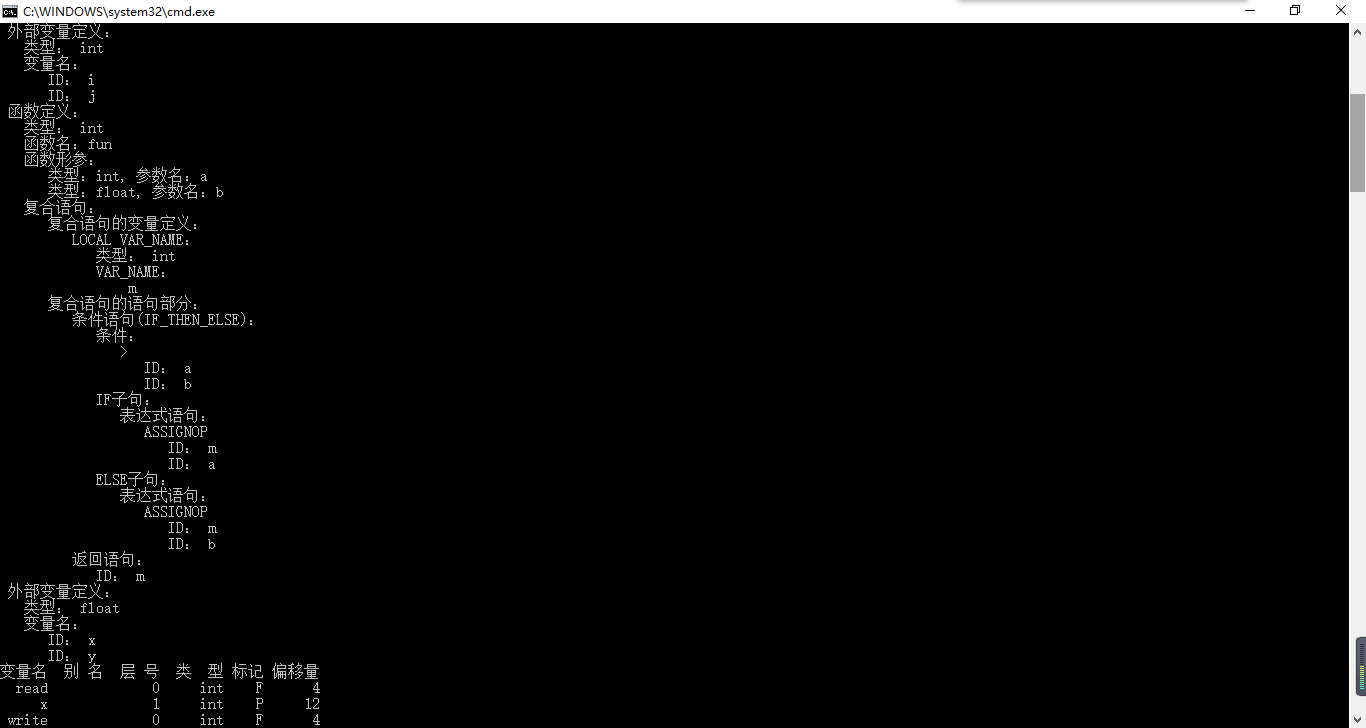
**}**

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的结点，其孩子指针指向该子树根结点。

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。

**四、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，具体格式可网上查阅相关资料，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;**