苏航：词法分析、语法分析、语义分析、中间代码生成、micode指令集设计、编译器前端测试、文档

叶梓辉：虚拟机设计与维护、micode指令集设计、中间代码检测与测试、文档

概要：依次进行词法分析、语法分析、语义分析、中间代码生成步骤，每一步骤检查出错误之后，予以报错，并不再进行接下来的步骤。

使用了预测分析表法。

翻译中间代码时，使用了后缀表达式法。

然后翻译成micode指令之后，调用虚拟机解释执行。

词法分析：

1、根据源代码长度动态分配缓冲区空间，将源代码一口气读入缓冲区中，增加读入效率，其中遇到了linux系统与windows系统不同换行符的问题。

因为我需要手动在缓冲区的结尾添加字符串终止符（’\0’），但linux 与 windows换行符分别是:\n and \r\n导致长度不一致。

2、词法分析读入的分析结果可分为如下四类：

操作符：+、-、\*、/、

界符：（、）、[、]、｛、｝、空格、换行符、制表符

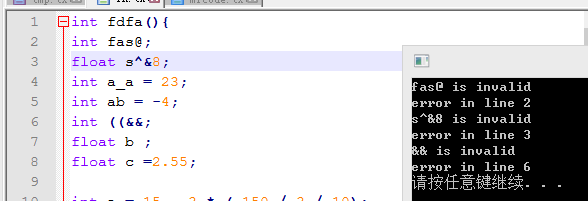
数字串：整数（形如123456）、浮点数（形如：123.456）

字符串：关键字（形如for、while、int、float）

用户自定义标识符：abc等等

词法分析的过程如下：不停读入字符，直到遇到操作符或者界符，将读入的一串字符根据各自的特质，识别为不同的Token。当读入用户自定义标识符时，将其加入ID表（用stl的map实现）。

3、对于读入的非法字符串，形如 ab@、 213￥、则予以提醒，精确到每一行。



4、实现了/\* this is a test\*/

以及//两种类型的注释语句，并且注释语句，不会对报错的行号产生影响。

语法分析、语义分析、中间代码生成：

将这三个模块混在一起讲，是因为我将这三个模块混在一起写的。

语法分析：

1. 使用了预测分析表法，使用LL（1）文法，但我将文法完全改写成LL（1）型文法之后，发现在中间代码生成阶段，函数声明以及变量声明变得特别麻烦，于是将产生式中涉及到函数声明以及变量声明的部分，重新改写为LL（2）型文法，在遇到函数声明以及变量声明，使用特殊判断来进行处理。

使用的SCI语言产生式如下：

Program -> Block Program | DclrFun Program | EOF

DclrFun -> Type ID '(' DclrVarList ')' 'FunBody

FunBody -> '{' SubBlock 'return' (Expression ';' | Empty) '}'

DclrVarList -> Type ID DclrVarList1 | @

DclrVarList1 -> DclrVarList2 | @

DclrVarList2 -> ',' Type ID DclrVarList2 | @

Block -> '{' SubBlock '}' | Stat

SubBlock -> Stat SubBlock | @

VarList -> (ID | num) VarList1 | @

VarList1 -> VarList2 | @

VarList2 -> ',' (ID | num) VarList2 | @

NumList -> Num NumList1 | @

NumList1 -> NumList2 | @

NumList2 -> ',' Num NumList2 | @

Stat -> Read | Write | Empty | While | For | If | DclrVar | FunCall | Assign

DclrVar -> Type ID DclrVar2

DclrVar2 -> ';' | '=' Expression ';' | '[' INTEGER | ID ']' DclrVar3

DclrVar3 -> ';' | '=' {' NumList '}' ';'

Read -> 'read' SupID ';'

Write -> 'write' Expression ';'

Empty -> ';'

Assign -> SupID '=' Expression

Expression -> Item Expression\_

Expression\_ -> ('+' | '-') Item Expression\_ | @

Item -> Factor Item\_

Item\_ -> ('\*' | '/') Factor Item\_ | @

Factor -> Num | SupID | '(' Expression ')' | '-' Factor\_

Factor\_ -> '(' Expression ')' | SupID | Num

While -> 'while' '(' BooleanExp ')' Block

For -> 'for' '(' (Assign ';' | Empty) BooleanExp ';' Assign ')' Block

If -> 'if' '(' BooleanExp ')' Block If\_

If\_ -> 'else' Blcok | @

SupID -> ID SupID1

SupID1 -> '[' INT | ID ']' | '(' VarList ')' | @

BooleanExp -> Expression RltOp Expression

FunCall -> ID '(' varList ')' ';'

RltOp -> '<' | '>' | '==' | '<=' | '>=' | '!='

Type -> 'int' | 'float' | 'void'

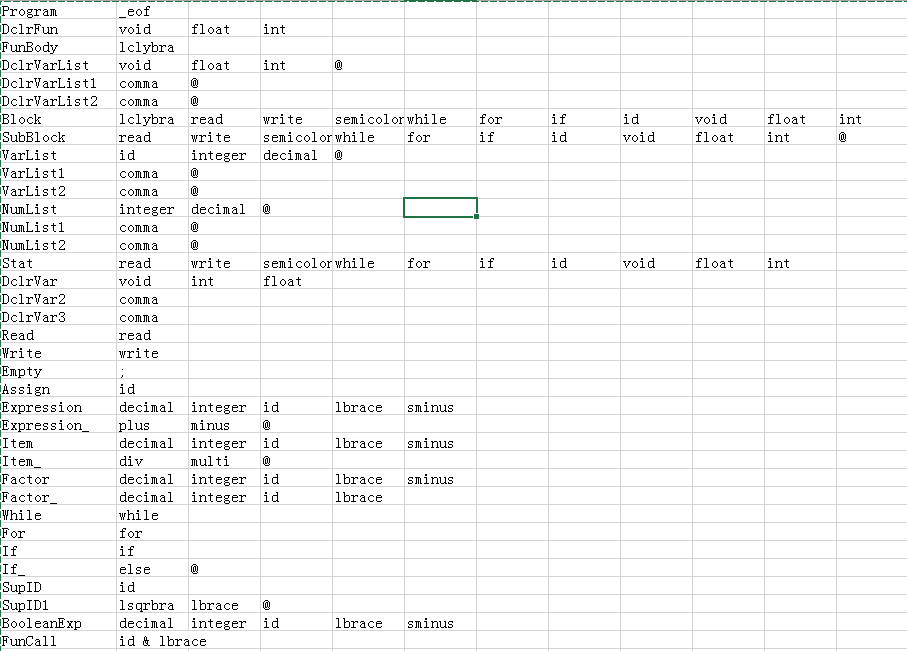
ID -> 标识符

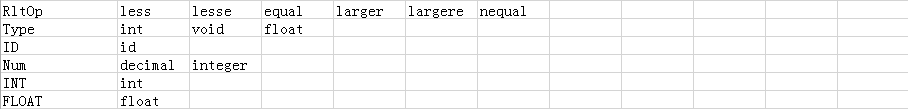
Num-> INT | FLOAT

INT-> 整型数

FLOAT -> 浮点数

SCI语言对于的预测分析表如下：





具体到代码实现细节，我本人模仿龙书中的做法，使用指针提前指向下一个Token，根据该Token的值来判断应该调用哪一条产生式，并对该产生式调用match函数，吞入已处理Token。

范例如下：

void Match(int i)

{

if(arrToken[p].m\_type == i)

++p;

else

{

hasError = true;

cout<<"line "<<arrToken[p].m\_lineNum<<" has error, "<<symbTable[i]<<" is expected,"<<" but get "<<symbTable[arrToken[p].m\_type]<<endl;

}

}

Read args;

void Read()

{

int record = p;

Match(READ);

NormId();

expStk.push\_back(arrToken[record]);

HandleExp();

Match(SEMICOLON);

}

1. 支持单目减法（负号）

形如：int a = - - - + a +(- ( + (- - b)));

的奇葩语句。

中间代码生成：

1. 表达式（形如： a \* (b + 32) / 23.423 + -fun（a, b） / arr[index];）

处理表达式时，使用了中缀转后缀的方法，将上述表达式转换成：

a b 32 + \* 23.423 / a b fun -（单目减） + index arr /

从前向后扫描操作符号，每扫到一个操作符，就先对操作数进行翻译，然后对操作符进行翻译。

支持表达式中多次调用函数。

1. for、if-else跳转语句翻译
2. 支持不同类型之间赋值、支持在同一个表达式中，不同类型变量 或 数字 进行混合运算，而且运算结果，严格与C语言一致。

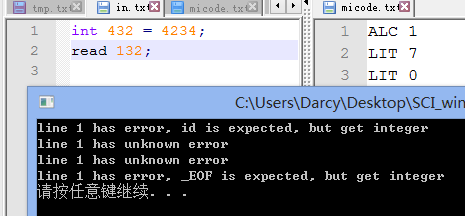
例如 1/2 == 0; 1.0 / 2 == 0.5

Int a = 1 / 2.0; a == 0;

Float a = 1 / 2; a == 0.0;

1. 变量的生命周期为一个函数体，变量可以在任意位置声明。
2. 语法分析报错定位，精确到行，可以精确到错误的是什么，正确的应该是什么，例如下列语句。

read 132;



1. 小细节：

支持c语言风格数组越界、

支持数组下标为标识符、

if-else、for、while支持无限嵌套

函数调用，只要内存空间足够，可以无限递归。

支持空语句、

支持函数返回值，支持函数返回值直接参与表达式计算。

支持void类型函数。

三、micode指令集、虚拟机

LIT： 取常数于栈顶，此操作会使栈顶指针++；

CAL：调用函数；

ALC：为函数划分内存空间；

JMP：无条件跳转；

JPC： 有条件跳转；

RET：返回函数，return。此操作进行后指向代码的指针会指向下一行代码；

WRT：输出，0为整形，1为浮点 栈顶指针—

IPT： 输入，0为整形，1为浮点，栈顶指针++

FTI： 操作数默认为零，操作栈顶元素

ITF： 操作数默认为零，操作栈顶元素

OPR：操作符

1. 栈顶整形取反

1-4 + - \* / (整数)

5-10 < <= > >= == != （整形）

11 栈顶浮点取反

12-15 + - \* / （浮点运算）

16-21 < <= > >= == != （浮点）

22 根据栈顶值取值到栈顶。栈顶指针不动 （取）

23： 将次栈顶的值存到栈顶所指向的空间。栈顶指针-- 两次 （存）