**简化版Pascal解释器**

**詹海鲸 2013/2/22**

**一、概述**

临近毕业，准备找实习、找工作。本科学的《编译原理》都还给老师了，为了把这些东西捡回来，做了一个简化版的Pascal解释器。

程序代码的编译执行需要经过以下几个步骤：

词法分析

语义分析

语法分析

中间代码生成

解释运行

图1 编译执行的步骤

词法分析是将程序代码转换成token流，比如语句sum = sum + i;，经过词法分析形成的token流为(”sum”, ID)，(”=”, OP)，(“sum”, ID)，(“sum”, ID)，(“+”, OP)，(“i”,ID)，其中，ID，OP表示token的类型。

语法分析的输入是token流，使用递归向下的方法从token流中构造出抽象语法树(AST)。

语义分析主要做的是类型的检查。例如检查某个操作符左右两边的操作数的类型是否符合该操作符的要求，检查函数调用和函数声明是否一致。

中间代码生成做的是将AST树转化成虚拟机可以运行的指令。

最后一步进程虚拟机解释运行上一步生成的指令，得到程序运行的结果。

二、词法分析

2.1 在内存中表示正则表达式

正则表达式包含两种基本要素，即：

1. 符号(Symbol)，例如字母(a-zA-Z)，数字(0-9)；
2. (Epsilon)，表示空字符

正则表达式包含三种基本操作，即：

1. &(Concatenation联结)(这个符号自己定义的)，比如正则表达式abc，则只有”abc”满足这个正则表达式；
2. |(Alternation可选)，比如正则表达式a|b|c，则字符串”a”，”b”，”c”都满足这个正则表达式；
3. \*(Repetition重复)，比如正则表达式(ab)\*，表示ab的0至多次Concatenation运算的结果，字符串”ab”，”abab”，”ababab”，…….都满足这个这个正则表达式。

我们使用树在内存中表达正则表达式。例如正则表达式[a-zA-Z]((a|b|c)|(ab)\*)生成的正则表达式树为：

联结

符号集

a-zA-Z

可选

符号集

a、b、c

重复

min = 0

max = ∞

联结

符号集a

符号集b

图2 正则表达式树

为了生成表达式树，我们先由正则表达式生成后缀表达式。什么是后缀表达式呢？举个四则运算的例子，比如a+b\*c-(a/c)，这个表达式的后缀表达式为abc\*+ac/-，得到这个后缀表达式后我们就可以用一个栈来运行这个表达式，原理很简单，遇到操作数直接入栈，遇到操作符从栈里pop两个操作数，进行运算后把结果push到栈里，最后栈里只剩下一个操作数就是表达式的运算结果。那么怎么生成后缀表达式呢？首先操作符有优先级，例如”\*”、”/”的优先级比”+”、”-“高，其次，要考虑左结合和右结合，比如a+b+c，左结合的运行顺序为(a+b)+c，右结合的运行顺序为a+(b+c)，一般情况下，都是左结合。转成后缀表达式，我们需要一个stack和vector，从左到右遍历表达式，当遇到操作数直接插入vector，遇到操作符时，首先会将stack里的优先级大于等于这个新的操作符的操作符一个一个pop并且插入到vector，然后再将新的操作符插入到stack里。当遍历到尾时，再将stack里剩下的操作符全部移到vector里面。其中”(“、”)”要做特殊处理，我们使得”(“的优先级最大，”)”的优先级最小，当遇到”)”时，将stack里的第一个”(“插入到vector后，”)”的处理就结束了，”)”不会使得”(“后面的其他操作符也被插入到vector里。按这个原理我们就可以生成后缀表达式了。

有了后缀表达式，我们就可以按照运算后缀表达式的过程来生成正则表达式树

2.2 生成-NFA

由正则表达式生成-NFA比较简单，我们只需要将正则表达式的几种基本操作对应成几种-NFA生成规则。

1. 联结

状态图

状态图

1. 可选

状态图

状态图

1. 重复(\*)

状态图

1. 重复(+)

状态图

1. 重复(?)

状态图

运用上面的规则就可以由正则表达式构造出-NFA。NFA是一个图，我们可以用邻接表来表示，主要包括NFAModel、NFANode、NFAEdge，NFANode包含有由这个点出去的NFAEdge，NFAEdge包含有两个成员变量，一个是边上的字符集，一个是目标点。NFAModel包含有两个点，分别表示NFA的开始和结束点。我们使用visitor模式来生成NFA，visitor模式请参考设计模式的书: )，所有的正则表达式类实现了一个accept函数

NFAModel \*accept(AbstractNFAConverter \*converter)。

类NFAConverter的接口如下

class AbstractNFAConverter {

public:

virtual NFAModel \*convertSymbolExp(SymbolRegularExp \*exp) = 0;

virtual NFAModel \*convertConcateExp(ConcatenationExp \*exp) = 0;

virtual NFAModel \*convertAlterExp(AlternationExp \*exp) = 0;

virtual NFAModel \*convertRepeatExp(RepeatationExp \*exp) = 0;

virtual ~AbstractNFAConverter();

};

class NFAConverter : public AbstractNFAConverter {

public:

NFAConverter();

~NFAConverter();

NFAModel \*convert(RegularExp \*exp);

NFAModel \*convertSymbolExp(SymbolRegularExp \*exp);

NFAModel \*convertConcateExp(ConcatenationExp \*exp);

NFAModel \*convertAlterExp(AlternationExp \*exp);

NFAModel \*convertRepeatExp(RepeatationExp \*exp);

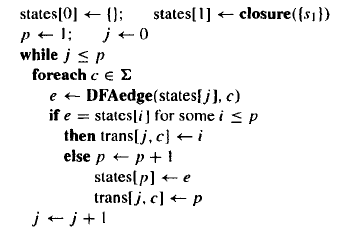
};

2.3 NFA生成DFA

由于从NFA的一个点吃进字符可能达到的状态有多个，这不利于我们进行词法分析。因此我们需要将NFA转成DFA。

首先我们要明确一个点集的闭包的概念，假如S是一个点集，则closure(S)为S里的点吃进可以到达的点的集合和S的并集。

NFA转DFA的算法：



其中DFAedge(states[j],c)表示states[j]里的点吃进字符c能到的点的集合的closure。

进行词法分析时，一般我们都有好多条正则表达式，通过选择操作将这些正则表达式组在一起，但我们必须记下NFA的结束状态跟正则表达式的对应关系，再得到DFA后，如果DFA里的一个点(DFA的点为NFA状态的集合)包含有其中一个NFA的结束状态，则当词法分析走到这个DFA点时，我们就匹配了该NFA结束状态对应的正则表达式。另外，进行词法分析时要注意最长匹配的原则，当匹配到一个正则表达式，我们还需要继续，直到出错或者找到一个更长的匹配，如果出错，我们就采用一开始匹配，如果找到一个更长的匹配，我们就采用那个更长的匹配。

三、语法分析

使用递归向下的方法直接构造抽象语法树。

考虑下面几种产生式：

1. FunctionList -> Function FunctionList |

其实这种产生式我们可以改写为FunctionList -> Function\*，表示Function的0到多个重复。对于这种产生式，如果下一个token在Function的FIRST集里，我们选用产生式FunctionList-> Function FunctionList；如果下一个token在FunctionList的FOLLOW集里，我们选用产生式FunctionList->；否则出错。

1. Arguments -> Arguments;Argument|Argument|

其实这种产生式我们可以改写为Arguments -> Argument(;Argument)\*|

这表示参数列表可以为空，可以只有一个参数(不带semicolon)，可以有很多个参数(参数间由semicolon隔开)。如果下一个token在Argument的FIRST集里，并且现在已经处理的Argument数为0，则将Argument加入到Arguments的childs里；如果下一个token是;(semicolon)，并且现在Arguments的child数>0，则将Argument加入到Arguments的childs里；如果下一个token在Arguments的Follow集里，则使用产生式Arguments->，表示Arguments解析结束；否则出错。

1. SimpleExpression->SimpleExpression Op Term | Term

其实这种产生式我们可以改写为SimpleExpression -> Term(Op Term)\*

由于肯定有Term，所以我们先parse一个Term，然后，如果下一个token是Op，则解析Op后继续解析Term；如果下一个token在SimpleExpression的Follow集里，表示SimpleExpression解析结束。

假如采用一般的办法生成语法树，例如E->E + T | T，解决左递归后产生式变为

E ->TE’

E’-> +TE’|

这时，a+b+c会被解析成

E

a

E’

+

b

E’

+

c

这不利于进行语义检查，并且它破坏了左结合，因此，一般我们还需要将这种语法树转成抽象语法树。我采用的方法是直接转成比较容易进行语义分析的形式。上面的例子会被转成

E

a

+

b

+

c

四、语义分析

语义分析主要做的是类型检查，以及检查变量是否定义，是否有重复定义等工作，具体如下：

表达式：

1. 最简单的表达式，获取表达式的类型（id，相应类型的常数）
2. 由简单表达式组成的表达式，判断操作符左右两边的表达式类型并获取类型
3. 变量是否在使用前已经定义，变量定义的唯一性
4. 表达式为函数调用，检测实参与形参的类型是否一致，获取返回值的类型为该表达式的类型

语句：没出错的话返回的类型为void，出错返回error\_type

1. 检查if，while，do while里的Expression的类型是否为Boolean，内部的stmt返回的类型是否为void
2. Assignstmt，左右两边的类型是否一样
3. ProcedureStmt，实参和实参的类型是否一致
4. CompoundStmt，检测各个子stmt是否返回void
5. break, continue只能出现在循环里
6. return 语句，有个变量储存现在所在的函数，检测return语句的类型时，检测表达式类型与返回值类型是否一致。

函数：没出错的话返回的类型为void，并将该函数放入符号表里，出错返回error\_type

1. 不支持函数重载
2. 所有函数里有且只有一个main函数
3. Function必须有返回值，包含return Exp，且Exp的类型跟函数返回的类型一致，procedure包含return，且后面没有返回表达式。

要完成这些工作需要用到符号表，符号表使用stack实现，stack里面放的是map，当分析一个新的函数时，会新建一个map，然后push到stack里，这个解释器不支持全局变量，除了函数能到最底层查找外，其他符号只能在stack的栈顶的map里查找。由于stl的stack不支持遍历，所以基于vector实现了一个支持遍历的stack类。

五、虚拟机指令生成

虚拟机指令的生成也是使用递归向下的方法处理AST树

5.1 表达式的中间代码生成

指令1：Type operand1

指令2：Type operand2

指令3：Operator（+ - \* /。。。。。） （这种指令会自动把结果push到栈里）

5.2 语句的中间代码生成

**CompundStmt**

为每条语句生成指令

**AssignStmt**

指令1：ID lvalue

指令2：右边表达式的指令

指令3：Assign （符号表的lvalue的值会改变）

**IfStmt**

指令1：判断表达式生成的指令

指令2：JmpFalse 指令序号（如果else有语句，序号为else语句的起始指令序列，否则为IfStmt后面的第一条指令的序列）

指令3：if里面的statement生成的指令（如果else有语句，GoTo IfStmt后面的第一条指令的序列号）

指令4：else里面的statement生成的指令，如果有的话

**WhileStmt**

指令1：判断表达式生成的指令

指令2：JmpFalse 指令序号，whileStmt后面的第一条语句序列

指令3：whilestmt里面的语句生成的指令以及GoTo 判断表达式第一条指令序列号

**DoWhileStmt**

指令1：DoWhileStmt里面的语句生成的指令

指令2：判断表达式生成的指令

指令3：JmpTrue DoWhileStmt里面的语句的第一条指令的序列号

**ReturnStmt**

指令1：如果有Exp，Exp的指令

指令2：FuncReturn 或者 ProcReturn

**breakStmt**

指令：GoTo 循环后的第一条指令的序列号

**continueStmt**

指令：GoTo 循环里的语句的第一条指令的序列号

**函数调用的指令序列**

前几条指令，参数的对应指令

指令I：FuncCall 函数第一条指令序列号（该指令的处理函数需要记录函数返回后的下一条指令序列号，FuncReturn指令将使用到该指令序列号）

5.3 函数的中间代码生成

指令1：参数对应的指令，每一个形参对应Paramenter symbol Type这样一条指令

指令2：局部变量对应的指令，每一个变量对应Declaration symbol Type 这样一条指令

指令3：CompoundStmt对应的指令

六、虚拟机

虚拟机由两部分组成，运行时栈以及符号表栈。运行时栈用于存放函数的参数、函数的局部变量，以及语句运行的中间结果；符号表栈用于储存某个符号在运行时栈中的位置以及函数的返回地址。

假如有个函数function max(a,b:integer) : integer，调用该函数的语句为c = max(a, b)，这个语句转换成虚拟机的代码会变成

ID a

ID b

FuncCall 36 (36为函数max第一条指令的序列号)

Assign c

函数max的指令大概为

Parameter INT a

Parameter INT b

。。。。。。。。

FuncReturn

ID a和ID b将a和b的值push到运行时栈里，FuncCall 36将下一条指令定为序号为36的指令，并在符号表栈记录下函数返回的地址，及Assign c的序号。Paramenter INT a和Parameter INT b会获得上面a和b的值(通过top和pop)，并将这些值push到运行时栈，符号表栈会记录a、b在运行栈的位置。继续运行函数里的语句，最后函数的结果会放在运行栈里。FuncReturn会取得函数的结果，然后把函数的局部变量和参数从运行时栈里删掉，然后从符号表栈获得函数返回地址，然后把函数的结果再次放到运行时栈里，然后Assign c会把函数的结果赋给变量c。

七、总结与展望

语法分析时，为了避免要时刻记得delete来释放内存，使用智能指针来自动化管理内存的释放，以避免内存泄露。下来要实现可以动态申请内存，会支持struct，支持new申请内存，实现gabage collector自动回收内存，降低程序员的负担。