## LR分析法

此方法基于状态法。

## LR(k)分析法 （一般k<=1）

LR(0)

不需要下一个输入符号，根据当前状态即可自动转移。

转移函数GO(I,S)=CLOSURE({A->aS.b}),即S后面以.结尾的产生式的闭包。

## 概念

LR（0）项目，首先对文法进行**拓广**，然后产生文法的所有项目，将其分类为**核心项目**和**非核心项目**，求每一个项目的**闭包**

求闭包的方法，如果一个点在B的前面的产生式（形如A->a.Bb）被加入I的闭包中，则所有的B产生式都会被加入I的闭包中。

注意，求闭包的方法不包含递归（即不依赖于其他的闭包），但是新加入的表达式会引起新的非核心项目集引入。

**项目集规范族** 拓广语法G’所有的闭包

**LR(0)语法** 如果项目集规范族中含有多于1个归约项目，称为**归约-归约冲突**，如果既含有归约项目，又含有移进项目，就称为**移进-归约冲突**；不含有移进归约冲突的语法称为**LR（0）语法**

# 文法翻译

## 翻译器的结构

一个引用的文法,

一个规范的(没有冲突的)分析表.

一个语义动作表(在产生一个规约的时候进行什么动作)

翻译器定义通用的翻译过程,但是保留具体的翻译细节给使用者.所以提供通用方法:

onNoGoto -- 在Goto表中没有相应的项时调用

onNoAction -- 在需要使用action表中的一个项时没有找到调用

onReduce

onAccept

onShift -- 移入一个符号的时候调用

(一般来说,上面两种情况都属于错误)

翻译器—归约过程的结构:

symStack 压入产生的symbolAttributes,在符号表中引用开始的位置,以及在代码中代码开始的位置

genedCodes 产生的代码序列

symTableList 符号表集合

currentInput 指针类型,指向当前的输入流

terminatorIndexStack 在归约过程中,如果用到了stringval,number,char以及id等通用符号,需要用到具体的词法属性,使用这个栈

symTableSwitchMap 指示符号表切换的map

## 输入流类型

将T类型转换为标准的int类型输出,应当是一个支持随机移动, 读取的stream类型

类型转换

StreamConvertor结构: 继承iostream(或者不继承)

注意WordStream的输出还不是我们想要的int流,因为在那里int代表的前面词法符号的类型, 所以我们还需要对其进一步处理才能得到我们想要的词法流.

比如 int – ID 就直接转化成为 int对应的整数. 为了生成这样的表, 我们需要为词法分析器指定一个动作,用于将vector<pair>转化成LexicalStream.

* 转化方法

我们可以定义一个特化的转换流 LexicalToGrammarStream

为了转化,它必须与一个已经定义好的文法协同工作.

因为文法已经将输入符号类型定义完毕, 转换的工作就是给定一个 LexicalParser解析流stream, 和一个文法g, 生成一个词法流

注意,我们需要区分 值和类型, 如果 int a=0中的 0也返回int,显然是不可行的.应当返回intval

## 工作方式

需要先将冲突的分析表改造成没有冲突的分析表.

通过获取文法的GOTO表和ACTION表(总称分析表)来构建一个DFA.

* 翻译过程

(curState表示当前ss的栈顶)

设定初始状态 curState, 在状态栈ss上压入curState,在输入栈is上压入结束符号$

对输入流的每一个输入符号i,

如果i的类型是变量

如果存在goto,则将goto对应的状态压入ss,输入流保持不变

否则报错

如果i的类型是终结符

查找action[curState,i],如果动作为移入,则is上压入i,ss上压入相应状态

如果动作为归约,则is弹出归约式长度的符号(如果是空产生式,长度为0),压入归约符号,然后弹出栈顶状态,压入栈顶对应的GOTO, 输入流保持不变,调用语义动作

否则报错

* 出错处理

错误是否恢复?(简单情况,直接退出)

给定一个输入流,然后执行翻译动作, 输出翻译结果.

输出结果是三地址形式的结果.

## 翻译器的动作(自动机)

翻译器也是一个自动机, 在某些输入流上同样需要提前进入结束状态等.

对翻译器自动机来说, 状态由当前栈顶的状态决定.

如果一个翻译器可以并行翻译多个程序(我们有意这样设计),则所有提供的方法都应当是const的,不能在成员变量中记录与翻译相关的状态,所以每一个函数都是可以重入的.

我们调用一个动作,比如onReduce需要哪些参数呢?

* 产生式, 由i,j标明
* 当前翻译器所处的状态(即栈顶)

语义动作:

全局变量声明为例: int a=0;

其输入流为 : int id = number ;

归约顺序:

1. Reducing TYPE -> int
2. Reducing EXPRESSION -> number
3. Reducing VAR\_DECLARATION -> TYPE id = EXPRESSION

归约1时,需要一个临时存储,比如temptype.type=int

归约2时,需要一个临时存储,比如tempexpr.val=number.lexval,tempexpr.type=int

归约3时,查找a(id)在符号表中的位置p,此时为了确定TYPE和EXPRESSION的实际值,需要用到分析栈的特性.因为此时栈的格局是:

TYPE id = EXPRESSION

我们需要另一个存储临时符号的栈st,则分析栈对应的临时符号栈就是temptype tempexpr

我们用st[top],st[top-i]来指代对应的符号的临时变量,则归约3的时候,语义就是 p.type=st[top-1].type p.val=st[top].val且要求st[top-1].type==st[top].type.这种情况下语义可能发现错误.

假设每一步都没有错误,则第2步会生成赋值语句:

tempexpr.val = number.lexval -- (=,number.lexval,\_,tempexpr.val)

第3步生成:

p.val = tempexpre.val -- (=,tempexpr.val,\_,p.val)

## 分析表冲突解决办法

优先级和结合性.

看优先级,相同优先级看结合性

具体规则:

* 保留最高优先级的产生式
* 在同优先级的产生式组中,保留归约式中的一个

优先级表： map<int,int> 数值越小优先级越高

结合性表： map<int,int> LEFT,RIGHT

输入: p, m (优先级表，结合性表) 冲突类型的分析表C

输出：规范的分析表R

对C中的每一项i

当前下一个符号是s

如果 i含有多于一项的转移

对i中的每一个式子f，如果f是规约式,令fs为f的运算符

如果f是二元表达式,(只考虑二元表达式的情况),

## 消除二义性的一般方法

采用交互式的方法,问一个答一个.[输入流为y/n组成,其他符号报错,重新输入,可以从文件读取确定流]

分析表可以缓存,然后从文件构建,这样构建出来就是一个已知的没有冲突的分析表.

## 翻译过程中的符号表管理

基本概念:符号表集合

临时符号在符号表中可以没有名称,但是必须有地址.

* 符号表的内容

符号表是一个数组, 但是里面的内容是一个属性map(我更喜欢用属性来实现而不是链表,因为查找操作占用更多的时间), 属性单元由<属性名,属性值>构成.(使用hashmap来存储符号表,键为名称,那些临时符号将有一个临时的名称,由genTemp生成, genTemp针对全局空间生成一个未使用的符号名)

属性名是string类型的,属性值是int类型(假设)

* 四元组的表示

四元组有两个关键的地方: OperatorType 和 OperandType

OperatorType可以用int来实现(统一编码)

OperandType则复杂些, 因为在四元组中引用的类型可能有两种: 立即值和符号表决定的值

所以OperandType使用pair<int,int> 前者指明是立即值还是符号表决定的值,后者指明立即值或者符号表下标

* stringval的处理

stringval不能被当成立即数来处理,因为它是不定长的.

我们可以这样处理stringval.我们有输入流WordStream(当然,在翻译阶段用不着), 在将WordStream转换成标准流的过程中,保持一个stringval记录数组 strIndexes, 每次遇到一个stringval就将其在WordStream中的下标记录下来(这样保证stringval只保存一份副本).

在翻译过程中我们保持一个stringval计数器 strRefIndex ,其使用见下面的子处理过程.

* 与符号表相关项翻译过程

设程序的所有符号表为 SymbolTableLists ->(简写为)stls, 符号表列表的基本单元是 <int,SymbolTable>,前一个int指向父符号表.

设置一个栈SymbolTableStack->(简写为)sts,表示符号表对应的逻辑程序结构,

初始的时候, stls.pushnew( <0,{}> )

sts.push(0)

对每一个子处理程序:

如果进入了一个块

stls.pushnew( <sts.top(), {} > )

sts.push(stls.size() -1 )

否则如果离开了一个块

sts.pop()

如果遇到stringval, 在符号表中插入一项类型为string, val为strRefIndex的符号,返回其下标;

如果需要引用这个stringval,则使用这个返回的下标.

如果遇到number/charval, 为了后面类型检查,需要扩展四元组的OperandType的表示, 我们规定OperandType中的第一个值可以表示立即数的具体类型,因为立即数类型有有限种,所以这种方法是可行的.

* 如何与程序存储分配阶段协同工作

分配阶段主要依照符号表来分配空间.不同的符号表主要决定分配的策略

分配策略有

1. 静态分配,维持一个全局的下标(staticLast)
2. 栈分配
3. 堆分配(C语言没有这个特性,但是可以扩充,它的堆分配不是由编译程序决定的,而是调用系统函数,所以这个与编译没有关系)

我们应当记录符号表中每个符号的存储特性: 静态的还是基于栈的

我们可以添加默认规则:如果符号表是局部的,就是栈分配;如果符号表是全局的,就是静态分配.

其实符号表的主要作用是确定access权限问题, 所以需要添加额外的信息来记录分配方式.

当然可以定义另一种类型的符号表,以分配方式划分,但是那样做访问控制就难以实现.

1. 如果是静态的, 增加staticLast, 分配完成就返回结果
2. 如果是局部的, 则这个局部应当有一个localLast指针,分配完成也返回基于这个局部的偏移

所以翻译结果构成应当包含所有的四元式,全局下标,局部下标.如何存储呢?我们将符号表中的第0项设置为全局变量符号表即可.

* 如何与生成机器代码阶段协同工作

首先解决全局分配,只要一句staticLast的值保留空间即可;然后分配一个staticBase, 要求所有符号表中的static类型的地址偏移加上staticBase

然后是局部变量分配;这个只需要使用堆栈框架即可:

sub esp,localLast

引用的时候

mov ebp,eax

add offset of a, eax

所以我们可以看到其实在符号表中属性addr更加准确的叫法是offset.但是addr作为惯用用法已经不可更改.

* 翻译过程中使用的符号引用优化

每次翻译的时候,在栈中压入符号其实都是为了最后生成代码.同样做genedCodes中也会存在副本.

为此可以设置一个symbolAttributes Pool,可以在其中添加然后引用.

## 符号的类型:SymbolAttributes

立即数

字符串常量

引用符号表中的某一项

引用符号名称

主要是为了生成汇编代码的结构更加精确.

当前生成汇编代码,只需要生成符号即可, 偏移地址可以放到后面链接阶段再说.

当前:引入temp即可

类型指定各个字段不同的含义

规则1:尽量重用相同类型的字段

规则2:尽量不引入冗余字段

# 机器指令生成:以x86为例

0: (=, a, 0, \_)

1: (param, a, \_, \_)

2: (param, coming : %d, \_, \_)

3: (call, printf, \_, \_\_temp\_1)

4: ([]off, s, 0, \_\_temp\_2)

5: (=, 1, \_, \_\_temp\_2)

6: (>, a, 0, \_\_temp\_3)

7: (jmpfalse, \_\_temp\_3, 17, \_)

8: (-, 0, a, \_\_temp\_4)

9: (=, \_\_temp\_4, \_, a)

10: (<, a, 0, \_\_temp\_5)

11: (jmpfalse, \_\_temp\_5, 15, \_)

12: (+, a, 1, \_\_temp\_6)

13: (=, \_\_temp\_6, \_, a)

14: (jmp, 10, \_, \_)

15: (return, 0, \_, \_)

16: (jmp, 24, \_, \_)

17: (==, a, 0, \_\_temp\_7)

18: (jmpfalse, \_\_temp\_7, 22, \_)

19: (\*, a, a, \_\_temp\_8)

20: (return, \_\_temp\_8, \_, \_)

21: (jmp, 24, \_, \_)

22: (-, a, a, \_\_temp\_9)

23: (return, \_\_temp\_9, \_, \_)

# 附录：文法实用工具集

## 文法定义

于是文法G可以定义为文法符号的集合（即变量）和产生式的集合 (Gramma)

产生式定义为： 文法符号A, 一个句型

句型：一系列有序的文法符号

## 句型为空的检查

如果句型长度为0，返回真

如果句型长度为1，检查A的每一个产生式的右部句型是否为空；

如果句型长度不为0，则要求每一个语法变量都经过检查为空。

## 提取左因子算法

[1 2]

[1 3 4]

[2 4 5]

[1 2 6]

[3 4 6]

使用归类法：

从下标0开始

从第一个元组开始，将所有与此元组相同开头的元组归为一类

[1 2]

[1 3 4]

[1 2 6] 1类

[1 2]

[1 2 6] 2类

[1 3 4] 3类

[2 4 5] 2类

[3 4 6] 3类

下标递增，在每个子类中重复上述过程

直到列表中只有一个元素或者没有元素

编号每一个数组 0 1 2 3 4

void findLeftFactor(data,i,classMap)

while(true)

{

for( d in data)

{

if(i>=d.size)continue;

if classMap[d[i]]!={}

classsMap[d[i]].push\_back(d.index);

else

classMap[d[i]]={d.index}

}

for(m in classMap)

if(m.size > 1)

findLeftFactor(data,i+1,…)

}

形成的前缀树

1 2 3

2 3

[] [6] [4]

12

1

2

3

[1 2 3 4 5 7]

[1 2 3 4 5 6]

[1 2 3 3 4 5]

按照最短原则进行： 1 2 3 ([4 5 7]|[4 5 6]|[3 4 5])

再对 [4 5 7] [4 5 6] [3 4 5]进行

4 5 ( [7] | [6] )

进行提取的原则：查找公共前缀长度为0的所有串，如果全部相等，则查找长度为1的前缀，直到第一个不满足全部相等的串出现；

如果前缀长度大于0，提取；

在不满足全部相等的下标开始分类；

对每一个分类，如果分类数目大于1，则重复上面的过程

全部相等： 如果所有的串都具有相同的前缀，此标志为真

函数：

f(data, i, name )

{

while(data has common prefix)

{

reduce an instance of prefix

}

}

## 消除直接左递归算法

对 S -> Sa | b

变换成 S->b S\_’

S\_’ -> a S\_’ | e

对 S-> Sa1 | Sa2 | Sa3… | b1 | b2..

变换成 S-> b1 S\_’ | b2 S\_’ | b3 S\_’ …

S\_’ -> a1 S\_’ | a2 S\_’ … | e

## 消除所有左递归的算法

要求：不含有循环推导

定义顺序集合v

i = 0..v.size()循环

j=0 … i (not including i)

if(p[v[i]] startswith p[v[j]]) replaceFirstProduction(v[i],v[j])

reduceDirectLeftRecursive(v[i])

## 消除重复产生式

对于产生式集合U，验证第i个符号以后的所有符号是否相等

f(U,i)

如果|U| < 2 返回

对U中所有没有第i个符号的产生式进行合并

对U中的每一个产生式P,对P进行按第i个符号归类，如果存在一个2-分类，统计之，调用上面的过程

直到不存在任何2-分类

## FIRST集（返回一个set而非vector）

1. 对每个产生式的右部，如果第一个符号是空或者终结符，将其加入产生式右部符号的的FIRST集
2. 对每个产生式，依次从长度1递增检查其右部，如果当前的字串有空产生式，将相邻符号的FIRST集加入；如果遇到第一个非空句型，此符号就完成。
3. 重复2，直到发现没有任何符号的FIRST集变化

注意：上述算法对计算机程序而言是不友好的（效率低的）

高效率算法(没有无用步骤):

建立依赖链:dependency={} //记录变量之间的依赖关系

计算出空符号映射: emptyMap //先计算出所有的变量是否能为空

对每个产生式的A->B

对右部B的每个符号语法符号a

如果a是终结符 将a加入A的FIRST集,

否则如果a是空符号 检查A的FIRST集中是否有

否则a是变量

如果FIRST(a)中没有空符号,

将a的FIRST集并入A的FIRST集,如果a中有空符号如果a包含在A中

+ 计算群体变量是否为空的算法

建立空依赖链 dependency={} alias dep dep[C]={A,B} --> C influences A,B

初始化空符号表映射: emptyMap

对每个产生式A->B

如果A尚未确定,则对右部B的每一个符号a

如果a是终结符,结束此次判断

如果a是空符号,进入下一个符号判断

如果a是变量

如果a已经确定

不能为空

如果a=A 结束此次判断

否则(应当有dep[a]={A}) 如果dep[A] 包含a,则结束此次判断

否则如果a已经确定且

## FOLLOW集

1. 将开始符号S的FOLLOW集设置为终结符 {#}
2. 对每个形如A -> aBb 的产生式，如果b是终结符，将b加入B的FOLLOW集
3. 对每个形如A->aBb的产生式，如果b是变量，将FIRST(b) – {EMPTY}加入B的FOLLOW集
4. 对每个形如A->aB, A->aBb,且b可以为空，将FOLLOW（A）加入B的FOLLOW集
5. 重复4直到每个符号的FOLLOW集不再变化

## 生成拓广文法和LR项目集

记录新的开始符号

项目集的记录要素

const GrammaSentence& g, dotIndex

以 S -> BB

B->b

B->aB为例

先生成拓广文法G’

S’ -> S

在拓广文法G’中，构建项目集

项目集中的元素是项目，一个项目由 首符号、产生式和dot下标组成, 即一个三元组

(int, int, int)

其中第二个int指定产生式，第三个int取值在0到产生式长度-1之间。

## 生成CLOSURE的算法（deprecated，using queue instead）

getClosure(I)

{

ClosureType C = { };

C += I;

std::map<int,bool> added = {false};//all false

hasnew=true;

while(hasnew)

hasnew=false

for ( c in C) //c是一个项

v = getFirstVariableAfterDot(c)

if( ! added[v] )

hasnew=true

for all item=(v,\*,0),add to C

added[v] = true;

}

## 生成GOTO的算法（deprecated，using queue instead）

getGoto(ClosureType C,int x)

{

GOTO = {}

hasnew=true;

while(hasnew)

{

hasnew=false;

for(item in C)

{

if item at dot is x:

I = <item[0],item[1],item[2]+1>

xClosure = getClosure( I )

if(!GOTO.contains xClosure)

GOTO += xClosure

hasnew=true;

}

}

}

## 如何高效地构造规范集项目族(待续)

所有的项目集均在 std::vector<ClosureType> vec中存储；初始时为空

第零类：以ItemType为key，记录其在vec中的位置 C0

第一类：以ClosureType作为key，ClosureType的index进行记录 C1

第二类 ： 以< ClosureType的index，x>作为key，ClosureType作为index进行记录 C2

vec 中加入 一项 {S’->S},存储在0

C0[<>]=0

求其闭包，存储在1

C1[C0[<>]] = 1

hasnew=true

while hasnew:

hasnew=false

对C1中的每个项集I

对I的每个.后面的X

如果 C2[I,x] 存在，Z=..

否则 Z = GOTO(I,X)且对C2进行修改

如果 Z 不在C1中，将其加入C1

# 附录：DFA，NFA和正则表达式

## FA的构成

由终结符号表、状态、开始状态、结束状态集合和转移函数。

终结符号表中可以含有空串，因此被认为是空输入。

符号和状态是分离的，因此管理它们的是不同的表。

符号可以通过正负数来区分是否特殊，在内部，我们仍然认为 字符串和相应的数字是等价的，但是数字更适合用来编写程序。

终结符号表的管理和状态的管理是同样的，即字符串数字等价性。我们可以建立一个T->字符串的映射，同时建立字符串->T的映射。

FA应当满足的性质：

1. 在处理输入流的过程中可以没有回退
2. 可以单独增加一个状态，记录要回退的情况；当那种情况产生的时候，就将输入流倒退一步。（这十分常见，并且十分有用，输入流回退时不处理输出流）

这是属于自动机需要处理的问题。

另外还有几个常用的控制： 比如说在什么状态应当直接停止

符号表带来的约束：

所有的符号查找通过str来实现，它们必然是唯一的。

## 自动机设计综述

最重要的是符号表，其次在真正处理的时候，还需要定义

1. 终止状态， 它是指提前将自动机在某个状态退出而不必遍历完整个输入流
2. 输入流回头 为了减少自动机的冗余，允许在某些状态之间进行转换时，保持输入流的指针

关于输出流

处理输出流是个复杂的问题，虽然从一般性上自动机要求输出流处理提提供一个process方法来处理，但是对于不同的问题输出流的处理是不一样的。

以词法分析器的输出流处理为例，通常词法的输出流采取懂动作都是有限的

## 映射关系实现 MutualMap

由两个map联合实现，增删改查同步。

## FA的实现

继承体系

FA(virtual base) = FAManager + staring + ending

NFA DFA ;share symbol table, states table, starting state, ending states, current state

FAManager应当提供的方法

对符号/状态的增删改查

对空符号，未定义符号的管理

FA应当提供的方法

当前状态,

设置开始状态

添加/删除结束状态

添加转移函数

转移（即执行一次匹配,next函数）

匹配并输出结果（也有相应的语义）（执行输入流然后获取输出流，匹配就是一个将输入流按顺序处理的过程）

DFA应当提供的方法

转移函数<q,a>最多有一个

NFA应当提供的方法

转移函数允许多个

提供一个到DFA的转换

## 如何处理输入流和生成输出流

输入流就是一个std::vector<int>, 通过将输入流连接到一个FA上，获取一个输出流;输出流是预定义的，由FA本身所确定的。

输出流在每一个状态上被处理，最后串联起来。

看起来处理输出流的动作是与状态和输入相关的。例如：

对于正则表达式 (\w+\d)\w+\d

我们知道需要捕获输入流中的第一个\w+\d； 再例如：对于程序语言，我们希望返回词法流， 我们需要知道何时开始捕获，何时终止捕获。捕获的输入流将会被存储到哪一个输出流中。为了确定这些信息，添加一些关于输出流的状态：

{

状态集合：{开始，捕获，忽略}

动作集合：{基于选择的流，开辟新的流}

开始状态

转移函数：

}

我们定义一种处理输出流的规范：

* 输出流状态只在FA状态转换时发生转换
* 输出流不能被清空，只能创建一个新的输出流或者附加在一个已经存在的输出流之上
* 输出流的相对移动只能 +1,-1,0, 可以向前随机引用

输出流必须在每个状态下的每个输入进行改变；所以处理函数是 <faqCur,inCur> --> action

## 输出流的动作

一般输出流：状态转移函数， 输出流缓存数组

输出流缓存数组通常就是vector<vector<T>>类型的，T是对应与输入流的类型。但是我们还不能这么假定，因为有的输出流通常还需要附加类型信息；甚至输出流还有可能是汇编代码。

所以我们应当假设输出流类型为泛型V, 从而使得输出缓存数组为vector<vector<V>>.

所以输出流的动作细节大部分被留给了使用者决定，我们可以假定一些十分普通的动作：

1. 附加到输入流i上
2. 新建新的输入流i来容纳

而对于语法制导的翻译，还有可能是某些赋值。所以这一部分工作将会留给具体的使用者实现。而输出流类通过接口来约束这种动作。

譬如，当输出流被告知需要进行一次处理的时候，这时，它调用自己的 process方法来处理这个字符，然后返回。

至于process方法的接口，可能是像这样的： void process(faqCur, inCur); 或者如果通过action的方式，则action原型也是这样的。由于在实现细节中faqCur需要涉及对FA这个类的访问，所以通常process和FA的关系是紧密相关的，所以process可以确定是由使用它的FA所确定的。通常，我们会为专门的FA编写特定的process函数。将这个类传递给我们希望获得输出流的FA，从而获取一个输出流。

## 输出流的动作

一个词法输出流可以将动作类型定义为 <state,in> --> <action,type, position> 的类型

假设输出流的类型为V, action的原型如下：

(state,in,position)

{

}

我们假设还有某种映射函数 将state,in映射为某种输入类型 ins

(ins,position)

我们知道对于词法流，它的输出类型就是 V=pair<int,T>类型的

它的action类型为 {附加一个<int,类型>到位置position上 } 变化的只有两个参量: 类型 和 position 的取值决定动作类型 : POSITION\_NEW 开辟新空间但不存储 POSITION\_APPEND 追加 POSTION\_NEWAPPEND 开辟新空间并且存储

POSITION\_NEW

POSITION\_APPEND

POSITION\_NEWAPPEND

POSITION\_NOTHING

## 带有分组处理的FA

在正则表达式中我们可以使用 [a-z]来表达式a~z的字符组；对于一个组，FA总是采用相同的动作，所以组对于提升FA的能力十分关键。

对于字符型输入流，因为字符不可以为负数，所以可以使用负数来表达组的概念。但是对于任意类型T，我们无法在编译期确定它的组的表达形式。

从实现上来说，我们可以在FA的内部增加新的组转移函数，对于某些状态，某些组，我们都可以采取相同的动作。这样一来，状态转移函数变成了

map<set<int>,set<int>,int> ；当我们在原来的组上找不到动作时，我们执行组查找，undefined分组（这是一个特殊分组，与已经定义的符号有关）,然后是其它的自定义分组。但是为了提高效率，我们决定还是使用原来的转移函数，只不过在添加分组时做相应的处理。

分组信息 : map<string,std::set<int>>

分组定义： addGroup(“name”,std::set<T>);//提供复制和移动两种构造函数

分组查询： inGroup(“name”,const T&)

ingroup (“name”,int)

添加转移函数的新处理： addTransitionByGroup(state,”groupName”,..)

addTransitionUndefined(state,”g1”,”g2”,) 添加分组s3相对于分组s1(或者s1,s2…)未定义的分组

addTransitionUndefined(state,”g1”,{“g2”,”g3”,…}

addTransitionUndefinedAll(state,”g1”,) //相对于已经定义的所有输入

addTransitionUndefinedAll(state,{“g1”,”g2”,…},)

## 词法分析器到DFA的转换

输入单个字符，DFA决定如何将词输出；默认情况下DFA将选择保留每一个形成的单词

输入： int main(const char \*argv)

它的输出流DFA如下：

开始状态 catch

## 简单正则表达式到DFA的转换

## 语法制导的翻译

## 统一化的语法设计

涉及软件升级的问题。

最初开发的时候可能采用的是无架构模式，后来找到某种架构，发现这种架构可以适用于原来的代码，此时就需要进行架构升级。

升级主要的工作：

等价变换

功能升级

过时标记

# 附录：无错开发方法

在编写程序的较小的变动内就将编译错误和警告修正。

实现这一目标的简单方法是在修改一个文件之后进行一次build。

# 附录：C++构造函数如何高效？

## 一般方法

我们不假定任何依赖于编译器的优化。

构造函数的主要构造方式有：bitwise copy， 指针构造

即使向vector这种类，它的动态空间依然是基于动态内存的；除非它自定义了自身

## 开放成员变量

另外，我们可以不使用任何构造函数做有效的事情，相反，开放所有属性。

## default move-ctor

### Implicitly-defined move constructor

If the implicitly-declared move constructor is neither deleted nor trivial, it is defined (that is, a function body is generated and compiled) by the compiler if [odr-used](http://en.cppreference.com/w/cpp/language/definition" \l "ODR-use" \o "cpp/language/definition). For union types, the implicitly-defined move constructor copies the object representation (as by [std::memmove](http://en.cppreference.com/w/cpp/string/byte/memmove" \o "cpp/string/byte/memmove)). For non-union class types (class and struct), the move constructor performs full member-wise move of the object's bases and non-static members, in their initialization order, using direct initialization with an [xvalue](http://en.cppreference.com/w/cpp/language/value_category" \l "xvalue" \o "cpp/language/value category) argument.

# 附录：C++设计原则：如何设计具有延迟生成结果的类？

## 缓存法

如果类需要使用某个结果，但是在类的初始化过程中、以及其他一些使用例程并不需要这些结果，则这些结果可以使用缓存。

通用缓存的设计：1.某些方法会使用某些结果（GET）

2.某些方法会生成某些结果（SET）

如果一个GET/SET方法的参数是A， 则将A作为缓存的键，记录一个boolean键值对的map，get时先查看缓存中是否有该值，如果没有就调用SET方法；

这是某个运行程序多次对同一结果求值的情况，这里可以看到，对于S -> BBB这种情况，算法将会进行多次重复求值，然而只要不改变类的成员，这些值是不会变的。

symbol is S

sentence is B B

symbol is B

sentence is a B

symbol is a

sentence is b

symbol is b

sentence is EMPTY

symbol is EMPTY

symbol is B

sentence is a B

symbol is a

sentence is b

symbol is b

sentence is EMPTY

symbol is EMPTY

1

## 导出法

如果类延迟产生某些结果，可以用一个包装器将结果和产生结果的上下文保存起来；但是如果两个导出结果之间需要相互调用，实现就比较困难。

注：延迟生成结果是为了效率提升而设计的。

# 附录：C++设计原则：如何在保证类完成自己职能的情况下不引入冗余信息

在设计一个类时，可能需要组合数个子类型的以简化设计。但是子类型间又将共享某个主类中的变量，如何在不引入多余的成员变量的情况下完成在子类内部完成所需功能？

class Mixed{

public:

Util util;

PartOne p1;

PartTwo p2;

};

util将会被p1,和p2用到。

1. 在调用时传参
2. 引入一个新的成员变量（对于许多需要存储的小类来说并不实用）
3. 引用一个静态变量（没有扩展性）
4. 在Mixed中定义PartXXX的功能（这样显得逻辑就会混乱，这种方法叫做代理）

# 附录：C++设计原则：类型统一和类型混合

一个错误的设计是，一种类型具有几种子类型，而每个子类型的结构都是不同的；试图通过最小公倍数方法来构造一个兼容每个子类型的类，而又不提供统一的方法来存储每种类型的代表，反而通过结构直接提供。

良好的设计是，在最小公倍数的类中加入类型信息，每个实例通过这个类型信息来标识自己所属的子类型。由于多数情况下只有父类型会被用到，因此存储这些实例各自的类型即可。

# 附录：C++设计原则：基于enum如何实现继承

# 附录：C++几种构造函数

struct V{

};

V() 调用构造函数

V(V&) 使用在定义时使用=,或者明确调用此函数，都会调用此函数； 使用=时调用可以认为是一种语义相等上的优化。因为比起赋值函数，构造函数更加高效。

V(V&&) 使用move语义时调用此函数

operator= 调用方法: v.V::operator=()。如果是静态方法，参数为V& v1,V& v2; 如果是成员函数，则为 V\* pv1,V& v2。 如果是在定义之外使用 = ,则调用的是operator=函数。在语义上，operator=应当和复制函数保持一致。

# 附录：C++右值引用

标准：C++11之后

与move语义相关

左值：一个可以被赋值的内存

右值：一个可以赋值给内存的值

析构函数会阻止隐式的move构造函数，也就是 A(A&&)在~A()的情况下会被删除。

///========以下面的程序为例=========///

#include <iostream>

using namespace std;

vector<int> doubleValues (const vector<int>& v)

{

    vector<int> new\_values;

    new\_values.reserve(v.size());

    for (auto itr = v.begin(), end\_itr = v.end(); itr != end\_itr; ++itr )

    {

        new\_values.push\_back( 2 \* \*itr );

    }

    return new\_values; //编译器也许会自动优化这里

}

int main()

{

    vector<int> v;

    for ( int i = 0; i < 100; i++ )

    {

        v.push\_back( i );

    }

    v = doubleValues( v ); //但是这里return一个临时对象，然后临时对象被赋值给v，这里编译器也许不会优化。

}

在过去的标准中（c++03），没有办法告诉编译器一个对象是临时对象。在C++11中，可以通过调用一个具有右值引用的构造函数、复制函数来简单地执行构造。但是右值必须是可以修改的，否则，当原来的右值到达析构阶段时，它的各个值就会被依次设置为无效；因而move语义的构造函数、复制函数也需要完成那些在析构函数中将数据置为无效的操作。这会修改右值，所以右值不能为const属性。

move构造函数：如果域是指针，则复制指针之后将原指针置为无效

如果域是对象，调用std::move将一个可能是作值的引用转换成右值引用。从本质上来说，std::move就是一个static\_cast。

**注意一点，右值引用的域不是右值引用，而是左值。**

**现在，标准库中的所有函数都可以被高效地返回。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | class ArrayWrapper  {  public:      // default constructor produces a moderately sized array      ArrayWrapper ()          : \_p\_vals( new int[ 64 ] )          , \_size( 64 )      {}        ArrayWrapper (int n)          : \_p\_vals( new int[ n ] )          , \_size( n )      {}        // move constructor      ArrayWrapper (ArrayWrapper&& other)          : \_p\_vals( other.\_p\_vals  )          , \_size( other.\_size )      {          other.\_p\_vals = NULL;          other.\_size = 0;      }        // copy constructor      ArrayWrapper (const ArrayWrapper& other)          : \_p\_vals( new int[ other.\_size  ] )          , \_size( other.\_size )      {          for ( int i = 0; i < \_size; ++i )          {              \_p\_vals[ i ] = other.\_p\_vals[ i ];          }      }      ~ArrayWrapper ()      {          delete [] \_p\_vals;      }    private:      int \*\_p\_vals;      int \_size;  }; |

# 附录：C++初始化列表 – initializer\_list

std::initializer\_list<T>

## 定义

其中的类型为const T，这种类型的值由编译器分配空间。

注意，如果复制一个list，则复制的list仍然指向原来的元素，不是真正的deepcopy。

关于初始化列表只提供const access的讨论，见<http://stackoverflow.com/questions/8193102/initializer-list-and-move-semantics>

另外，关于聚合类型初始化和列表初始化，参照

<http://zh.cppreference.com/w/cpp/language/list_initialization>

http://zh.cppreference.com/w/cpp/language/aggregate\_initialization

## 使用

struct myclass {

myclass (int,int);

myclass (initializer\_list<int>);

/\* definitions ... \*/

};

myclass foo {10,20}; // calls initializer\_list ctor

myclass bar (10,20); // calls first constructor

## 遍历

全局函数begin，end在其中重载。

// begin/end (initializer\_list)

#include <iostream> // std::cout

#include <initializer\_list> // std::initializer\_list

template<class T> void print\_list (std::initializer\_list<T> il) {

for (const T\* it=begin(il); it!=end(il); ++it) std::cout << ' ' << \*it;

std::cout << '\n';

}

int main ()

{

print\_list ({10,20,30});

return 0;

}

## 自动转换

initializer\_list 并不是 { }语法的全部，我们可以看到initializer\_list要求所含对象为同一类型，而我们却可以在多种情况下写出下面这样的语句：

pair<int,string> p={9,”ssss”};

这会调用具有相同参数的构造函数。

## 几个问题

对initializer\_list<pair<int,string>> list;

为什么std::move(it.second)的返回值并非右值类型，而是const左值？

因为list中的值全部都是const类型的，仅仅使用std::string&&类型是不足够的，需要const std::string &&,不过这样就不能对原字符串进行修改。

这样，就不能阻止当一个临时对象被析构时，它共享给被移动对象的资源被释放。

# 附录：C++标准库set的使用

链接：<http://en.cppreference.com/w/cpp/algorithm>

algorithm库提供的函数：

|  |  |
| --- | --- |
| [**merge**](http://en.cppreference.com/w/cpp/algorithm/merge) | merges two sorted ranges  (function template) |
| [**inplace\_merge**](http://en.cppreference.com/w/cpp/algorithm/inplace_merge) | merges two ordered ranges in-place  (function template) |
| [**includes**](http://en.cppreference.com/w/cpp/algorithm/includes) | returns true if one set is a subset of another  (function template) |
| [**set\_difference**](http://en.cppreference.com/w/cpp/algorithm/set_difference) | computes the difference between two sets  (function template) |
| [**set\_intersection**](http://en.cppreference.com/w/cpp/algorithm/set_intersection) | computes the intersection of two sets  (function template) |
| [**set\_symmetric\_difference**](http://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference) | computes the symmetric difference between two sets  (function template) |
| [**set\_union**](http://en.cppreference.com/w/cpp/algorithm/set_union) | computes the union of two sets  (function template) |

#include <iostream>

#include <algorithm>

#include <vector>

#include <iterator>

int main() {

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<int> v1 {1, 2, 5, 5, 5, 9};

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<int> v2 {2, 5, 7};

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<int> diff;

std::set\_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),

[std::inserter](http://en.cppreference.com/w/cpp/iterator/inserter)(diff, diff.begin()));

for (auto i : v1) [std::cout](http://en.cppreference.com/w/cpp/io/cout) << i << ' ';

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "minus ";

for (auto i : v2) [std::cout](http://en.cppreference.com/w/cpp/io/cout) << i << ' ';

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "is: ";

for (auto i : diff) [std::cout](http://en.cppreference.com/w/cpp/io/cout) << i << ' ';

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << '**\n**';

}

# 附录：C++关于类型的符号

decltype(t) 获取t的类型声明

typedef decltype(t) functor\_type;

using

using alias = what;

相当于

typedef what alias;

如果希望类型可以带上template参数，需要

template <typename T> using alias=…;

# 附录：C++ std::tuple 和 std::tie(…)

#include <tuple>

auto t=std::make\_tuple(10,20,…)

[std::tuple](http://en.cppreference.com/w/cpp/utility/tuple)<int, [std::string](http://en.cppreference.com/w/cpp/string/basic_string), float> t1(10, "Test", 3.14);

std::get<0>(t);

std::get<1>(t);

int a,b;

std::tie(a,b)=t;

这些类是c++函数编程的重要组成部分

std::apply(add,std::make\_tuple(1,2)) 可以调用函数 （c++17）

[std::tuple](http://en.cppreference.com/w/cpp/utility/tuple)<int, [std::string](http://en.cppreference.com/w/cpp/string/basic_string), float> t1(10, "Test", 3.14);

int n = 7;

auto t2 = std::tuple\_cat(t1, [std::make\_pair](http://en.cppreference.com/w/cpp/utility/pair/make_pair)("Foo", "bar"), t1, [std::tie](http://en.cppreference.com/w/cpp/utility/tuple/tie)(n));

n = 10;

print(t2);

## 总结

tuple pair make\_tuple make\_pair tie tuple\_cat

tie函数

Creates a tuple of lvalue references to its arguments or instances of [std::ignore](http://en.cppreference.com/w/cpp/utility/tuple/ignore" \o "cpp/utility/tuple/ignore).

创建左值引用，如果是std::ignore就忽略。

这样就可以使用tuple之间的对等赋值。

std::tuple<**int**,**int**,std::string> mt={**0**,**0**,**"1"**};

**int** n=**9**;

**auto** amt=std::tuple\_cat(mt,std::tie(n));

std::get<**0**>(amt)=**2**;

std::get<**3**>(amt)=**10**;

std::cout << **"mt<0>="**<<std::get<**0**>(mt)<<std::endl;

std::cout << **"n="**<<n<<std::endl;

输出：

mt<0>=0

n=10

我们可以看到 tuple\_cat 属于复制而非引用。如果使用std::move，可以看到调用move构造函数之后 mt<2> = “” 为空值

# 附录：C++ lambda表达式

常用形式：

[捕捉列表](参数列表){表达式;}

比如 [](auto a){return &a;}

## auto的使用注意事项

因为auto很有可能在不经意间调用到复制函数，如下面的语句

A& getA(){}

auto a=getA(); //=======这一句实际上调用了构造函数

合适的方法是auto & a=getA();

## 关于捕捉列表

以epsilon(一个本地的变量)

* [&epsilon] capture by reference
* [&, epsilon] specify that the default way of capturing is by reference and what we want to capture
* [=, &epsilon] capture by value by default, but for epsilon use reference instead

# 附录：C++标准库algorithm的使用

C++的标准库的通用性和Java标准库的通用性有本质区别。因为Java不可能产生同一个模板的多个实例化，因此它不存在方法特化，也就是说无法针对具体的容器进行更快速的同名算法操作。然而C++天生自带这种特性。

参考：<http://en.cppreference.com/w/cpp/algorithm>

## 非修改性函数

std::for\_each

std::for\_each\_n for\_each on the first n elements

std::transform

std::find std::find\_if std::find\_if\_not 使用顺序查找

使用：auto result1 = std::find([std::begin](http://en.cppreference.com/w/cpp/iterator/begin)(v), [std::end](http://en.cppreference.com/w/cpp/iterator/end)(v), n1);

count count\_if

search

使用二分查找

upper\_bound -- 5 5 6, 查找 5的upper\_bound,就返回6 x>5

lower\_bound -- 在一个有序的序列中查找第一个大于等于指定数，返回迭代器 x>=5

for(int I : b )… for each循环

search 针对一段范围进行搜索

find 针对一个特定的值进行搜索

includes 相当与java中的contains，但是这个函数只能针对有序集合。

## 修改性函数

在一个序列中指定位置插入另一个序列： 使用std::inserter(容器，迭代器位置)

#include<iterator>

back\_inserter 在末尾

front\_inserter 在首部

使用 std::copy(start,end,to)

# 附录：使用gdb调试c++

# 附录：使用gdb server

# 附录：使用eclipse配合gdb调试

首先不要使用任何从 Makefile构建个工程，请选择使用eclipse自动生成makefile

# 附录：语法分析基础概念

语法树：含有

1. 子树：以A为根的语法子树

语法树的结果：所有的叶子结点按序写下

短语：所有至少具有父子两代的语法子树的结果

直接短语：只包含父子两代的语法子树的结果

句柄：语法树的最左直接短语

FIRST集

FOLLOW集

## 一个问题：求FIRST集或者FOLLOW集是否需要消除左递归（以及提取左因子）

# 附录：Git—合并一个现有提交并保留当前状态

git merge –abort 撤销一个不成功的合并（即有冲突的合并）

如果merge出现

双方添加： z.txt

这种情况，则z.txt已经被修改。这是因为了两个commit中的同路径同名文件中的内容不一致所致。

这种情况下，z.txt已经被修改了。而可能我们想要的结果是保留当前提交中的z.txt(因为它更新)。

无论使用 –ff 或者是 –no-ff，结果都是同样的。

在这种情况下，z.txt已经被移除当前合并的暂存区域（即所有的冲突暂存文件都已经被移除，除非手动添加那些冲突文件）。这时，如果使用 git add z.txt ，则就会将冲突 “象征性地解决“，但其实z.txt保留的是diff格式。

保留自己改变的情况下，使用命令：

git checkout –ours/--theirs 文件

来确定要保留的更改。

使用 git checkout –ours \*

来保留所有己方的改变。

但是另外一些文件，是不希望从另外的分支获取的，他们将被添加至暂存区域。这种情况下可以先通过本地创建一个空的commit，然后与远程合并之后，将合并的commit清空，然后与本地合并。

清空可以使用 git rm \* -f