Front End

前三个assignment做的东西：lexical analysis ,syntactic analysis, semantic analysis, intermediate code/representation generation// ast就是representation，

Back End

Code generation //assignment 4

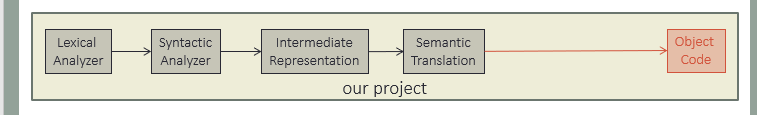
为什么要区分front end 与back end

front end 是machine-independent的，与机器本身无关，不管你是mac,windows都这么写，提高重复使用性,哪怕在不同的target machine

Back end 是machine-dependent的。 back end的步骤与target的machine language相关

在Sytactic analysis之后，我们有很多种选择

共同点在于，他们都基于syntactic-semantic analysis中的验证处理后的syntactic information



我们的方案

Lexical Analyzer与Syntactic analyzer每个方案都要做， 但是我们用intermediate representation也就是AST

Semantic Translation:在最终生成前check你的程序是semantic right的

Intermediate representation:

把所有syntactic information在parse过程中集合起来，最后形成一个树或者directed graph

Intermediate representation允许high-level code优化optimization

在我们的project中，我们使用AST作为intermediate representation

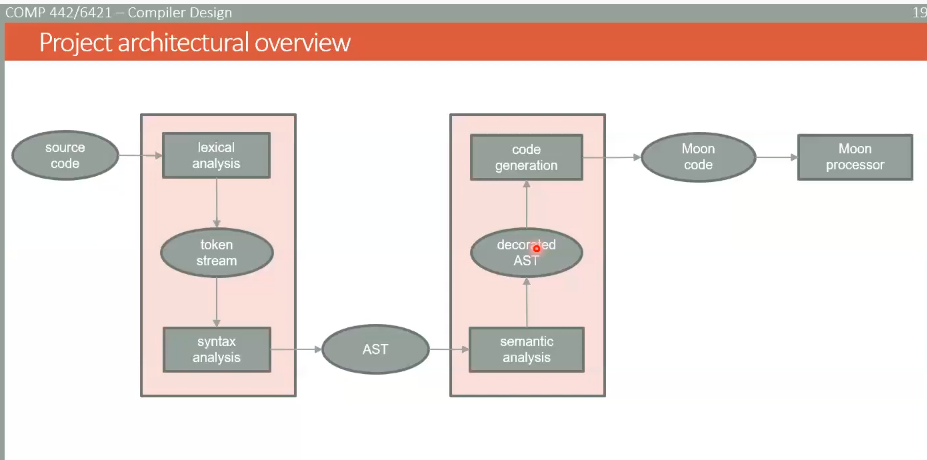
Intermediate code: 这个程序的low-level coded representastion(纯文字)，可以直接翻译成object code， 我们这个project中没有intermediate code

Intermediate representations

每个node代表着grammer中一个rule的应用

一个subtree仅在处理完右侧parsing完以后才能生成

。。。  
一长串



Decorated AST

你给ast的node加上了symboltable,这种，就叫对他decorated

我们接下来要对decorated AST再一次遍历，进行code generation

生成的是moon code，被Moon processor使用

Visitor mode:

随着我们遍历整棵树，对应的semantic action被trigger

因为有很多semantic action，所以我们使用不同的visitor，将他们分类成phases

这样我们就可以一组一组action 来check

下面那张图意思就是我们有个父类Abstract Visitor

具体模块分为Semantic Analyzer以及Code generator两个模块，code generator又会feed moon code

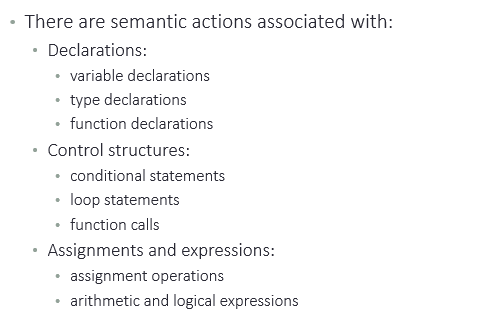
Semantic actions and code generation

Semantics 就是给我们的program正式赋予meaning的过程，因为我们写的代码本质上是High level(人能理解的代码)，我们需要给他meaning，让他通过code generation变成机器能理解的代码

Semantic Checking,有可能syntactic 没有error，但是semantic上有error,所以我们要进行semantic checking，看看这个程序是否可以有meaning:既identifier被正确的declare与use, operator左右变量合适

//就算有semantic error，也能够generating code，只是在于无法executable

Semantic transition: 没有code generation error



code generation 所需要的semantic action一共有这么几种，

他们应用于强类型过程编程

Processing declarations//

这章讲的是如何通过declaration generating code

在处理declaration的时候，唯一需要做的type checking就是每个identifier被declared且只被declared过一次（assignment3已经做了）

限制也是通过symbol table来test的

Semantic translation

Type declaration中的code generation 关键点之一就是计算需要多少memory size，例如Integer（variable）,我们需要给他派多少byte的内存，但是对于array , object这种内存并不固定的，比较复杂，接下来会讨论

怎么知道我们需要多少的MEMORY

知道Symboltable中描述的variable，自然就知道每个element的size相加起来，也就是这个function/class需要的memory

有两种 code generation 方式：

tag based也就是这节课

stack based也就是下节课

Tag/lable based:

给具体的variable分配memory并且生成一个独一无二的label，这个label将在Moon machine cide中被使用，用来指代这个variable

Stack-based:用stack-based这个方案， 所有的variable的reverence必须使用offset-calculation 偏移量计算， 这个calculation假定function的所有variable被存在一个function call stack frame

Offset指的就是以一个function为例，假设他的stack起始点为原点，然后加减几为偏移量Offset，就能得到对应variable

现实生活中用的更多的是stack-based

但是我们这里可以用tag-based,而且如果你用stack-based, 也有一部分是tag-based

如果你要允许recursion，那么stack-based是必须的

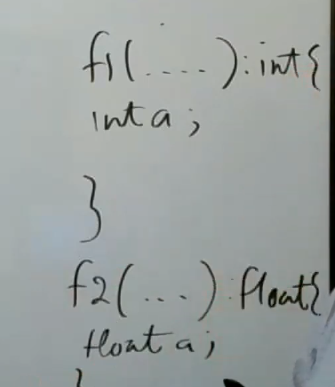
Semantic Checking:

检查一个entry确定已经进入symbol table了

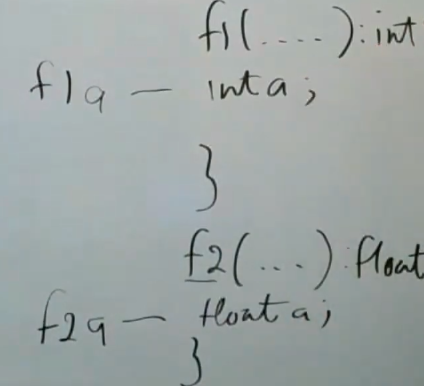
Semantic translation

如果你是用了label mode,你必须给variable生成一个lable。

label不能单纯的等于variable的名字



因为所有的tag都将被运往同一个moon machine，有着同样的tag，就会冲突



最简单的方式在前面加上前缀

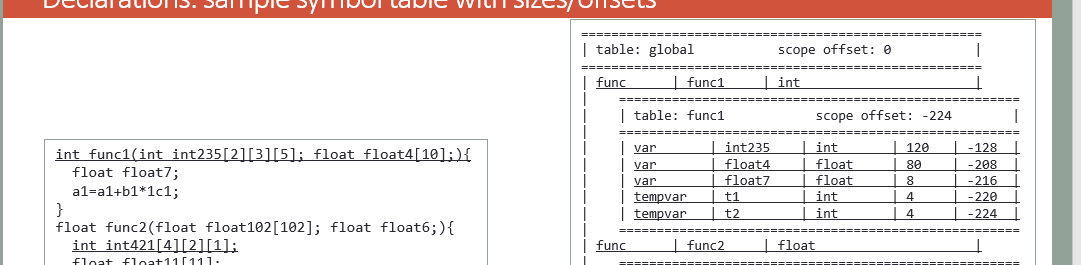
最好有个label generator,来generate unique name

最后无论是offset还是label都需要直接插入symbol table或者是直接成为node本身的一个属性//最好还是存在table entry里，因为放入AST node会使Node过分复杂·

简单的来说我们需要一个label generator，

我们还需要一个visitor，如果遇到了table，计算每个table entry所需size，计算整个table size//loop tree，如果遇到table, 算出size

如果你是用了label mode,就是要用哪个variable就用他的label  
如果是stack mode,我们要使用address+offset，.//stackFrame +address



这张图的解释：

第四个是size，单位为byte

第五个是offset, 为什么是负的

假设

stack顶--------------------

variableX

variableY

这样一直累加，就能算出最终的总和224

那么我们要call这个function的时候，

直接pointer往下移224，这就是我们要分配的memory.

temp var就像这种



（（1+2）+3）

我们可以通过添加temp var，来给他们运算的空间

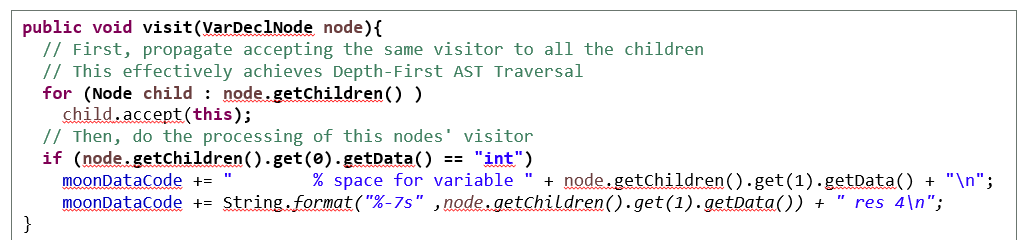
要注意temp var 不是指的123这些node

123本质上是一颗tree

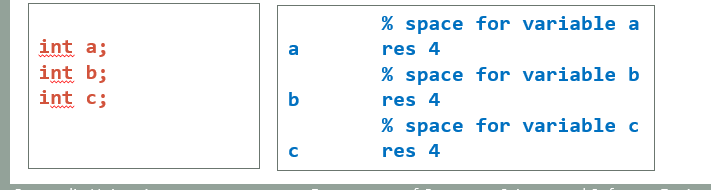
我们先要1+2,记录tempVar

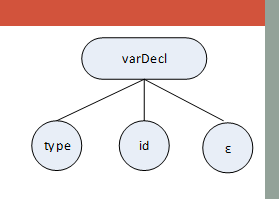
再让这个tempVar+3，记录tempVar,把这个值给y，结束

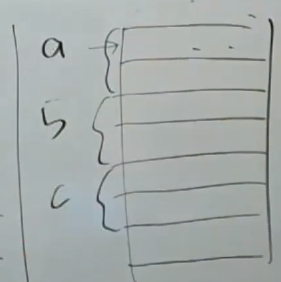
怎么转换成moon code



例如普通的varDecl node，看看他是不是int，那么就按指定格式给他生成code，指定格式下一课会讲//右边蓝字就是generate出来的moon instruction，a是label，a是他所reserve的memory cell的第**一**格，res 4代表reserve预定 4bytes







是第一格

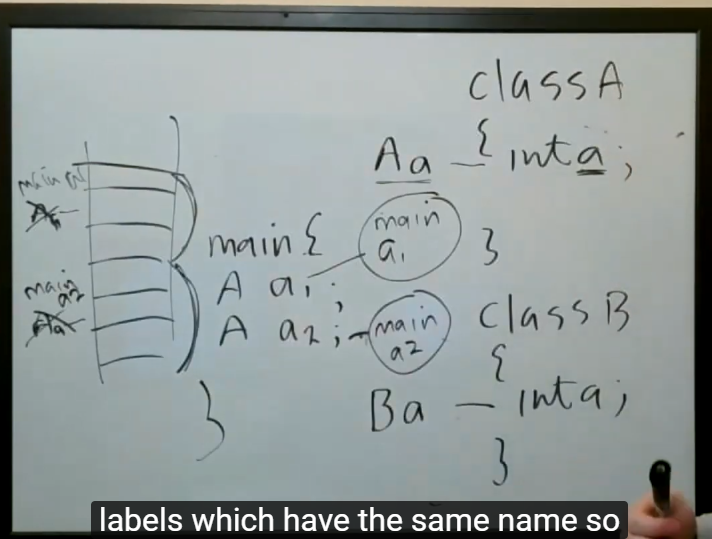
这只是atomic data

what if array data, object data

Array:

我们的array declare的时候必须是固定size的

object



完全可以有两个同样的object，

可是这样里面的变量就重复名字了

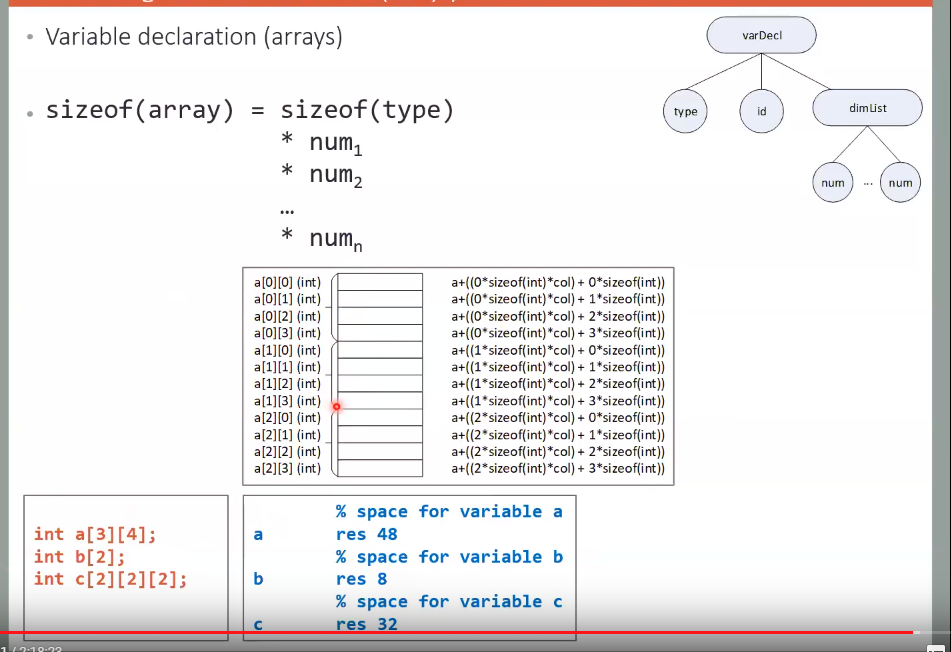
第一个是Aa(classA inta)

第二个也是Aa(classA inta),

我们需要用offset来指代 element of Class, //OBJECT

a1a2label还是不同的

里面的Aa不应该，应该用offset指代



array的size很简单，只是type\*num\*num

重点是怎么找到某个array具体的位置，

通过a+公式

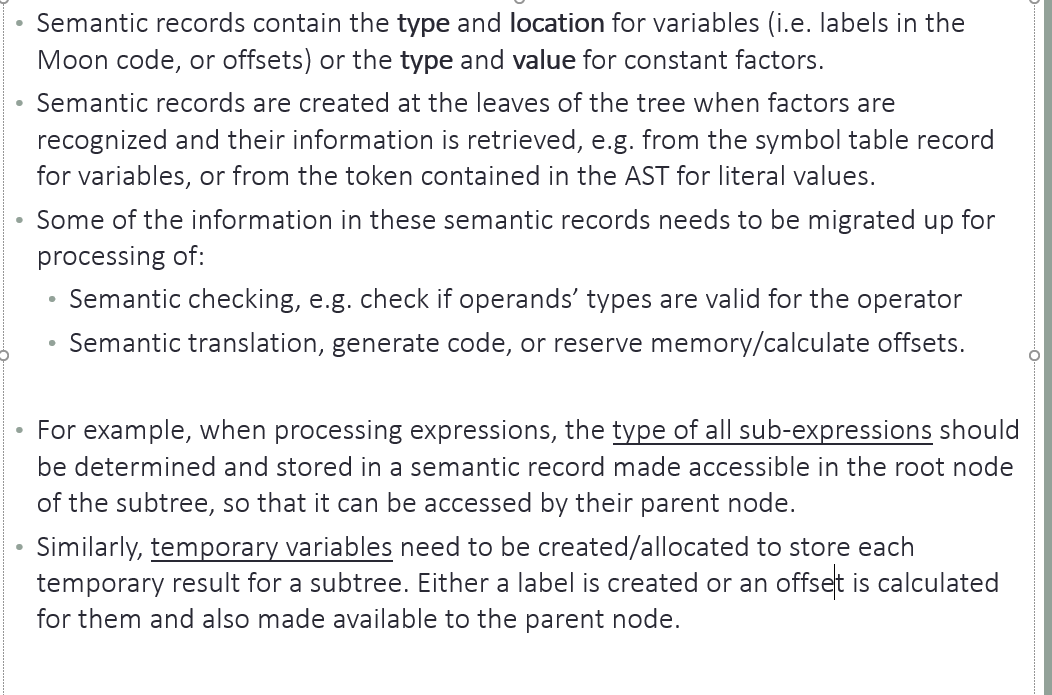
a指的是起始点

下一张图指的是一个object具体容量大小，首先

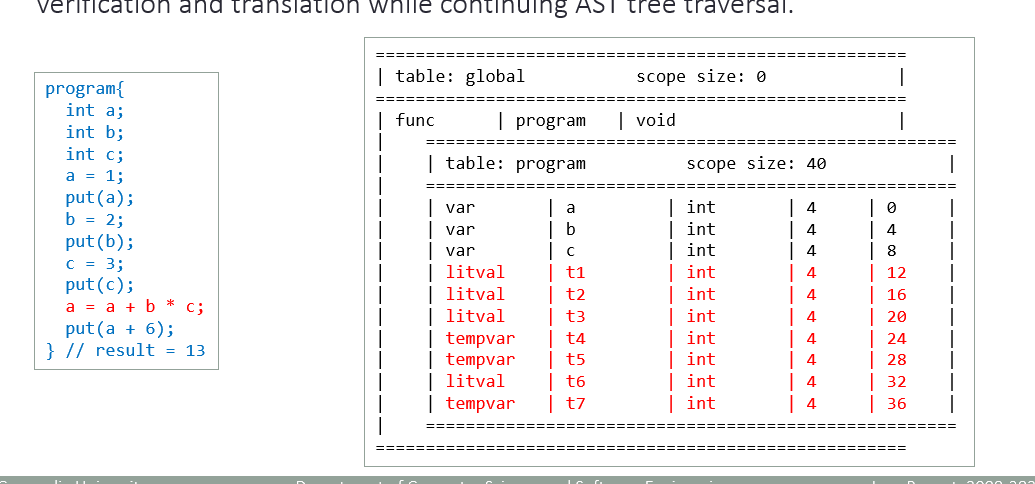
要把每个inherit class Size加上

其次要把member size加上

//我们这里不用，因为我的symbolTable 隐藏集成了所有条目，只需要加上所有条目size



重点只有一个，就是每一个subtree都要一个temporary结果，这样我们才能继续计算，这也是我们为什么要往table里面插入tempVar

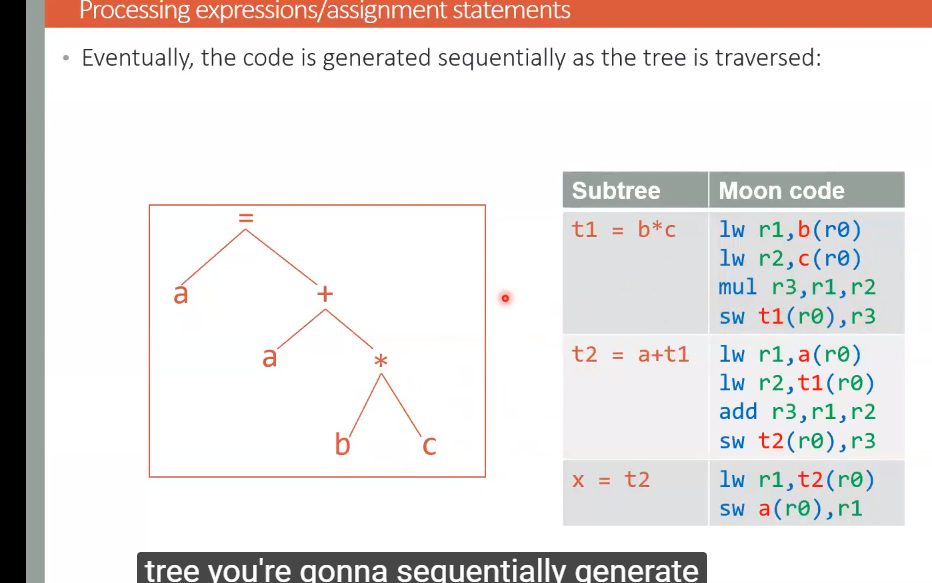


b\*c，tempVar

a+b\*c,tempVar

a+6, tempVar

+-\*的顺序不用管，只要traverse tree你就能generating code



例如这个，

我先traverse到右下角

lw r1,b(r0) //load value of b to register 1 //b指的是alias of第一个index, r0就是offset of 0

lw r2,c(r0) //load value of c to register r2

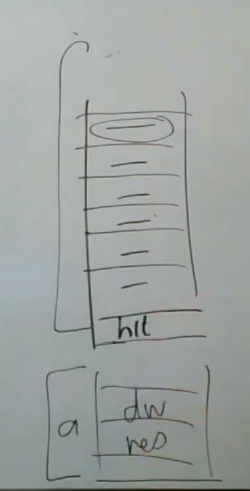
mul r3,r1,r2 //multiply r1 and r2 ,to r3

然后我们把r3的值存储到我们的t1//tempVar预定的位置

然后我们就能使用t1了

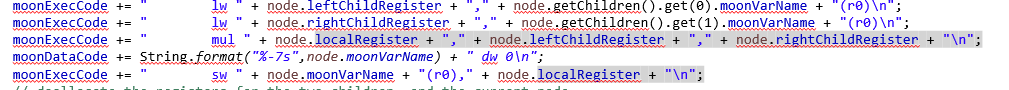
......

r1r2r0随便用



然后data存储与程序语句应该分开放，exe运行我们的variable data会崩溃

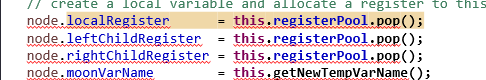
我们需要halt来终止



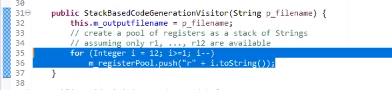
所以我们有两个stream，一个是execCode

一个是Data Code

注意



localRegister也就是r1r2r3那些



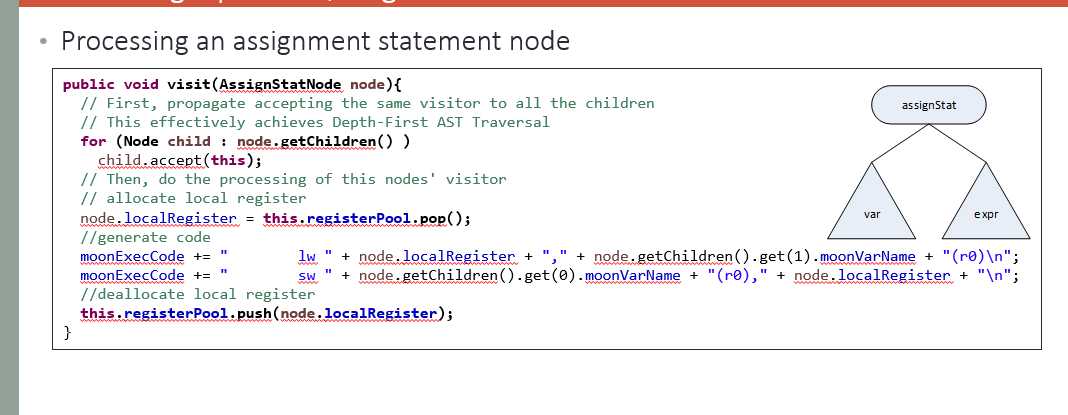
他会创造一个Stack of String,

里面有12个String, r1r2r3r4..r12

要用String了，pop出来用，

用完了push回去

这里直接使用moonVarName，作为一个simple case,实际上更复杂



Assign stat Node

把暂时的register提取出来，等于我们的localRegister

然后进行moonExecCode,

用完以后就可以push回去了