## 语义分析与符号表

在语义分析与符号表构建阶段，所做的工作主要有三点：

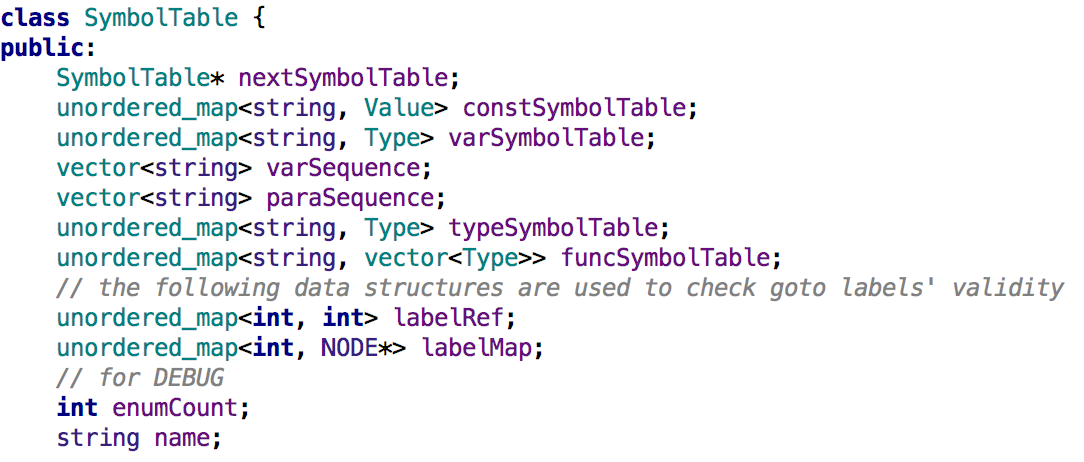
1. 检查每个表达式的类型，并在合适的时候自动提升类型
2. 对于每个作用域（对于Pascal而言是函数以及子函数）构建一张符号表，将符号（常量/类型定义/变量/函数/过程等）插入到其对应作用域的符号表中。对于符号的引用，给出一个对应符号表的链接。
3. 在类型不匹配或符号引用缺失的情况下提示错误，并中止后续任务（因为不可能再生成合法的中间代码）。

为简单起见，我们的Pascal 编译器使用的是静态作用域的设计。

语义分析主要基于符号表，因此下面给出符号表与符号表所使用的类型系统的设计：

### 符号表：

我们定义了SymbolTable作为符号表的类，其中包含若干个unordered\_map（hashmap），给出符号名即可在常数时间找到对应的符号，相比于手动实现的hash表更加方便快捷。



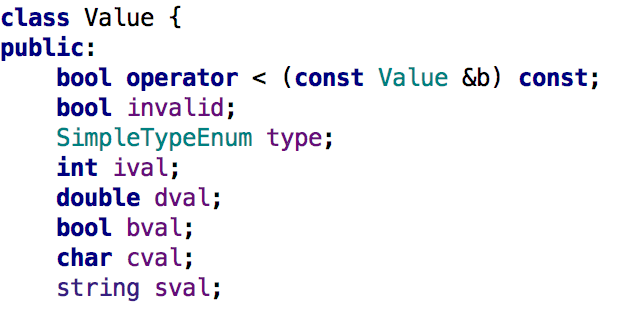
我们采用了每个作用域一张符号表的做法，使用一个list<SymbolTable>将其串接起来，每次使用list.front()获取到当前作用域的符号表，使用list.pop()在离开作用域时弹出当前符号表，符号表本身也有一个nextSymbolTable，指向它上一级作用域的符号表，这一过程可以在list.push时向SymbolTable的构造函数传入list.front()来完成。不难看出，整个SymbolTable依托于AST形成一个树状结构，每个SymbolTable保留指向父亲的指针，如图1所示。这样的做法可以保证在知道当前作用域的符号表后可以访问到当前作用域所能访问到的所有符号，而『屏蔽』其不应该访问到的符号。

/Users/orpine/Downloads/symboltable.png

图 1

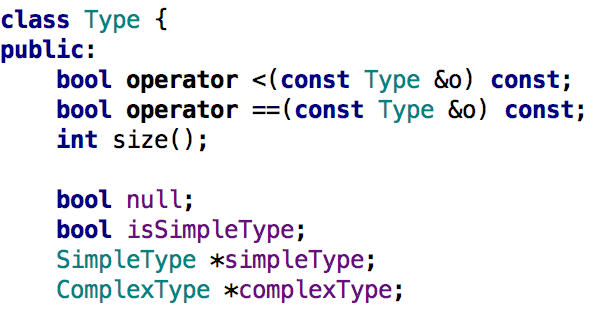
SymbolTable类中包含若干子表，其中包括constSymbolTable, varSymbolTable, typeSymbolTable, funcSymbolTable.

constSymbolTable所存放的是利用const关键字所声明的常量符号，在引用了对应的标识符时（例如数组声明时利用了之前定义的常量标识符）需要查找这张表，由于常量在编译期即可确定，这部分存放的是具体的值，所以constSymbolTable的类型是string->Value。我们使用Value类将所有简单类型的值统一装箱，在需要使用时再进行拆箱，Value类定义如下：

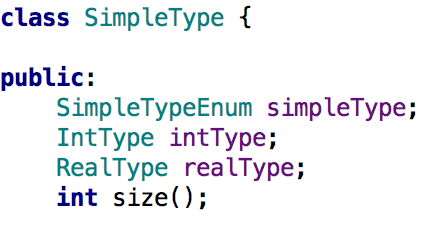


其中利用type指明了Value的具体类型，ival，dval，bval，cval，sval分别存放integer，real，boolean，char，string类型的数据。

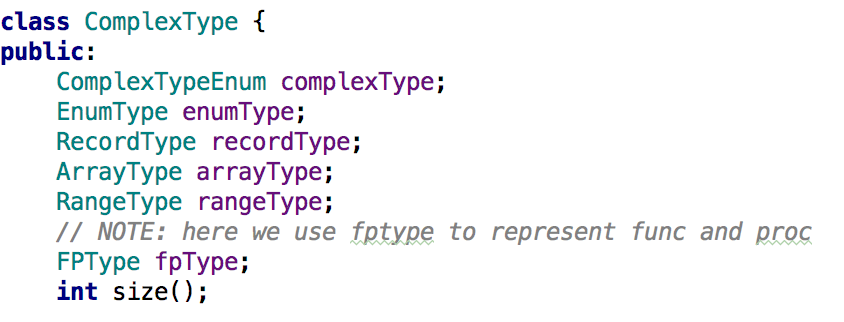
在介绍后面的符号表之前，我们需要先介绍Type类，Type类是我们所定义的类型系统，在经过合适的设计之后，一个Type类的变量即可表达任意本编译器可处理的类型。



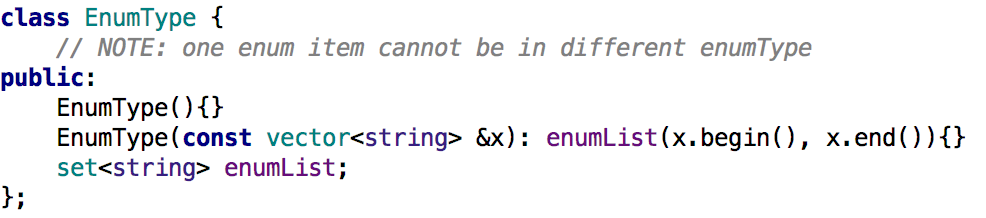
Type类实际上是对SimpleType类与ComplexType类的简单封装，分别代表系统中的标准类型与构造类型，由于ComplexType类中某些部分又引用了Type类型，为了避免无限循环引用，我将SimpleType与complexType类定义为指针类型。下面介绍SimpleType与ComplexType类



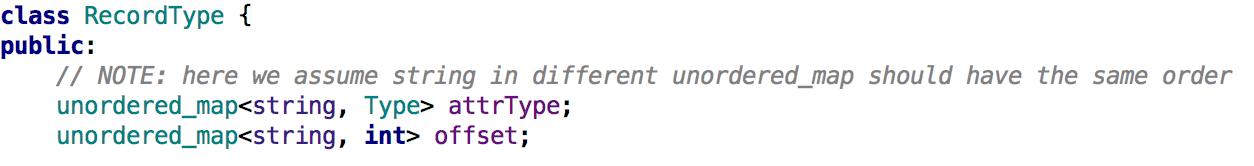
SimpleType类中包括了所有的系统简单类型，包括integer，real，boolean，char，string五种，利用simpleType指示出具体是哪一种类型，当是int或者real时，再通过intType，realType来细分。



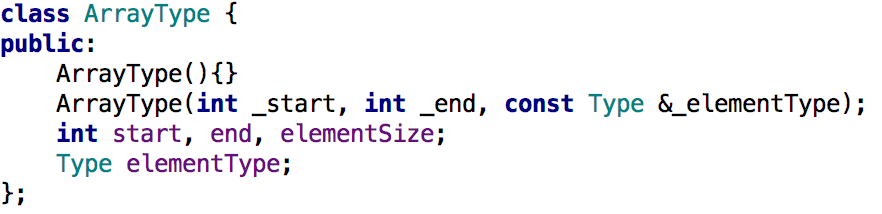
ComplexType类包含了一些较为复杂的类型，例如枚举类型，record类型，数组类型，range类型，函数/过程类型等



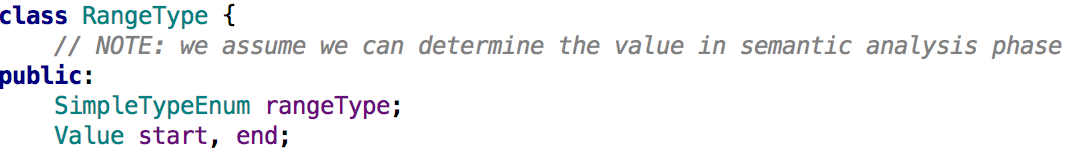
EnumType类包含了该枚举类型所有的标识符，需要注意的是，将这些值存下来是基于最大化保留信息量的考虑。实际上除了语义分析阶段（检查当时定义的枚举类型中是否有重复的标识符）以外这个类型并不会被用到，因为我们实际对枚举类型的实现是为其分配连续的常量值，将标识符插入到常量表中来实现的。为了不排除进一步改进的可能性，还是将所有的信息都保留下来了。



RecordType类实现了record类型的定义，利用一个unordered\_map<string, Type>，来记录每个域的类型，注意到由于这里递归的使用了Type类型，我们的类型系统是支持多重嵌套的record类型定义的，虽然实际上由于文法的限制我们的编译器实际只能支持一层record访问。



ArrayType类通过记录开始，结束的下标以及对应元素的类型来定义一个数组，类型系统支持定义一个元素类型为复杂类型的数组，但受限于文法，我们实际上不支持数组元素为record类型下record类型的域的访问。对于a[i].b这种写法在语法分析阶段就会报错。



RangeType记录了start，end的值，在我们的编译器设计中该类型实际上只用于进行数组声明，因此目前的实现中后续阶段也不会用到。



FPType通过记录参数和返回值的类型定义一个函数和过程类型，其中argVarList记录的是对应位置的参数是否要求为左值，当retType为一个空Type时FPType表示一个过程，否则为一个函数

Type定义的示意图如图2所示：



图 2

以上就是我们的类型系统的设计，实际上还有一些类型如文件/指针等没有包含，但是由于我们的编译器本身并不支持，所以也就没有考虑了。下面让我们回到符号表本身的介绍。

typeSymbolTable中存放的是利用type关键字所声明的类型，在变量声明部分如果使用了非预定义的标识符就需要查找此表，显然此表的类型应是string->Type，因为type关键字所声明的类型一定能用一个Type变量来表达。只需要查找这张表就可以知道标识符所对应的类型。

varSymbolTable中存放的是当前作用域中声明的变量的类型，用于类型检查以及后续的代码生成。

funcSymbolTable 中记录了当前作用域声明的函数与过程，用于函数调用的时候查找对应的符号。由于需要支持函数的重载，此表的类型是string->vector<Type>，因为一个标识符可以对应多个函数，我们需要进一步根据参数的类型确定具体是哪一个函数，注意此处并不考虑返回值类型，即不能同时插入两个名称一样，参数类型一样，但返回值不一样的函数/过程。

由于本编译器的文法不支持goto Label的统一声明，而是使用了C-style的方式。因此在语义分析时有可能发生引用了尚未『看到』的label。因此我们需要使用两个unordered\_map来记录goto Label所在的<label number，对应语句的AST节点>和引用goto Label所在的<label Number，对应语句的行号（方便生成错误信息）>，在离开一个作用域时统一进行检查，是否存在引用了无效label的情况。

varSequence/paraSequence这两个vector主要方便后续代码生成，记录了变量/参数的声明/传递顺序，以便于计算栈上的offset，与本阶段关系不大

符号表的大致组成部分由图3所示，该图隐藏了varSequence等非关键成员，其中typeSymbolTable与labelMap由虚线所示，表示这两部分不会被后续代码生成阶段所使用



图 3

以上就是符号表的定义部分，事实上，完成了符号表与类型系统的定义，本阶段的工作就完成了一大半，语义分析的工作将基于以上部分以及AST生成规则，在AST上进行遍历而完成。

### 语义分析：

得益于Pascal的良好结构，每一部分有其对应的特点，在Pascal上进行语义分析是一件相对比较好完成的工作。根据其AST生成规则，我们可以将其大致分为如图4所示的几部分：



图 4

我们需要做的，就是定义一个RoutineAnalysis函数，分别解析其const part/type part/var part/func|proc part还有statement list部分。对于func|proc part内部的解析，递归调用RoutineAnalysis函数即可，因为两者结构是类似的。

让我们按照图4的顺序一步步来说明：

semanticAnalysis函数作为整个Semantic Analysis阶段的主函数，这里需要对最开始的根符号表做一些初始化的工作，例如，处理一些原生函数（由于MIPS模拟器功能所限，并不支持Pascal的诸多原生函数如chr, ord等，而这些函数的实现非本次大程的重点，所以我们只支持read/readln, write/writeln两组IO函数，其中read在符号表中插入对应元素，write进行特殊判断（因为write支持任意多参数）来通过语义分析的检查阶段）。在准备好最开始的根符号表后调用routineAnalysis函数，该函数主要包含以下部分：

const Analysis:

首先是constAnalysis部分，该部分假定常量声明一定是以a = b的形式一条条给出，其中a是标识符，b已经作为Value的实例被存放到AST的节点中，因此只需要从AST中取出a对应的字符串与b对应的Value，将其插入到当前符号表即可。在插入时，需要进行常量标识符与变量标识符重名的判断，虽然此时还未进入当前作用于的变量定义阶段，但考虑到如果此时在函数内部，就存在虽然还未定义变量，但已经存在变量符号的情况，那就是函数的参数。

type Analysis:

typeAnalysis与constAnalysis类似，类型定义也是以a = b的形式一条条给出，其中a是标识符，b是一个类型，可以是简单类型也可以是复杂的数组或者record类型定义，因此我们需要使用一个parseType函数来解析出b的类型，最后将其插入当前符号表的typeSymbolTable。此处我们需要考虑a有可能与当前符号表中的常量重名，因此在插入时需要查找当前的constSymbolTable中是否有同名常量。

Var Analysis:

varAnalysis与前面类似，区别在于变量定义是以namelist: type的形式给出，我们调用parseType解析出type的具体类型后，遍历namelist将其插入varSymbolTable，此处需要判断namelist中的标识符是否与当前作用域的const与type重名。

Function/Procedure Analysis:

如果存在函数/过程的定义，我们首先需要解析函数/过程头部，取得函数名，参数类型与返回类型（如果有的话），插入一个FPType到当前作用域的funcSymbolTable中，然后，我们新建一张符号表，插入到symbolTableList的头部作为当前符号表，将函数的参数插入到新的符号表中的varSymbolTable，如果是函数，还需插入一个与函数同名的变量。然后递归调用routineAnalysis函数进行分析，分析完成后，弹出symbolTableList的头部，恢复到原来的作用域。

Statement Analysis:

前面提到的都是定义部分，主要完成了符号表的构建工作，接下来的部分是对程序主体进行分析，基于之前定义的符号表进行符号和类型检查。

程序主体是一个包含了多个Statement的Statement List，我们遍历这个Statement List，一句句的对Statement作分析。

对于Statement的分析类似于递归下降，在去掉了goto Label后，statement可分为如下几部分：

赋值，函数调用，条件，循环，case，goto语句。

赋值部分：

我们认为以下三种值为左值，可以进行赋值，以identifier形式表示的变量（类型为简单类型或者record及enum类型），以identifer[expression]形式表示的数组元素（类型同上），以identifier.identifier形式表示的record成员（类型同上），如前文所述，受限于文法，我们不支持record的多层访问。

对于赋值号的右边，我们调用expressionAnalysis解析出其类型，expressionAnalysis是一个递归下降的解析过程，其结构如图5所示：

../../../Downloads/expression.png

图 5

其中每一层处理按照优先级处理其对应的符号，并完成类型的定义以及自动转换（如expression，如果存在<,>,<=,>=,=,<>等符号，首先判断符号两边类型是否一致，如果一致，设置当前节点的类型为boolean，否则报错）。

函数调用：

由于函数重载的原因，我们需要获取到所有参数的类型，然后在符号表中进行查找。这样一来对参数的匹配过程已经由符号表自动完成，如果符号表查找失败，则提示函数未定义。为简单起见，我们可以自动对传入参数进行upcast并且在有多个函数可以匹配成功时认为第一个匹配到的函数即为程序试图调用的函数而不是报错。特别需要注意的地方在于，如果函数声明时使用了var关键字，或者类似read等函数，函数的参数将会要求是一个左值，这种情况需要额外进行判断。我是用了一个checklvalue函数。

条件与循环：

If/repeat/while的部分都很简单，分析条件部分的Type是否为boolean，然后再调用statementAnalysis分析条件/循环的主体部分即可。稍复杂一点的是for循环，需要判断循环所使用的标识符代表一个变量以及变量类型为整数，同时判断起始值与终止值是integer类型（注意，枚举类型在我们的实现中也为integer类型）。然后再进行循环主体的分析。

Case语句 ：

只需匹配好case中expression与分支里常量的类型即可。

Goto语句：

需要将引用到的label放入当前的符号表中，在退出当前作用域时进行label合法性的检查。

#### 错误提示信息：

在语义分析阶段，我们有可能提示如下错误：

Const Analysis/Type Analysis/Var Analysis阶段：

Duplicate identifier: 重复的标识符

Undefined type: 未定义的类型

Invalid enum item identifier: 非法的枚举标识符（该标识符已被类型/常量/变量所使用）

Duplicate enum item: 重复的枚举标识符

Range data type mismatch: range类型左右两边类型不一致

Illegal range: 非法的范围（左边值大于右边值）

Supported range data type: 不支持的范围类型（左右两边的值不能为real等类型）

Undefined const identifier: 未定义的常量标识符

Duplicate field name: 重复的record 域名称

Duplicate function definition: 重复的函数声明

Duplicate procedure definition: 重复的过程声明

Duplicate parameter name: 重复的参数名

Routine Analysis阶段:

Duplicate label: 重复的goto label

Undefined variable: 未定义的变量引用

Unsupport assignment operator:变量不支持赋值语句

Type mismatch between assignment operator: 赋值号两边类型不一致

Cannot downcast data type automatically: 无法自动向下转换数据类型

Xx is not an array: xx不是一个数组

Index must be integer: 数组的下标必须为整数

Xx is not a record: xx不是一个record

Invalid attribute xx: record不存在域xx

Undefined function or precedure: 未定义的函数或过程

xx needs a lvalue: xx需要一个左值作为参数

the type of condition clause mys be boolean: 条件部分的类型需要是boolean

xx must be a variable: xx必须是一个变量

xx musb be integer: xx必须是一个整型变量

for loop require integer type: for循环的边界需要是整数类型

xx must be integer or char: xx必须是一个整型或者字符变量

label type mismatch: case语句的label类型不匹配

type mismatch for cmp operator: 比较操作符两边类型不匹配

type mismatch for + and -: +/-操作符两边类型不匹配

type mismatch for or: or 操作符两边类型不匹配

type mismatch for \*: \*操作符两边类型不匹配

type mismatch for /: /操作符两边类型不匹配

type mismatch for div: div操作符两边类型不匹配

type mismatch for mod: mod操作符两边类型不匹配

type mismatch for and: and操作符两边类型不匹配

type mismatch for not: not操作符两边类型不匹配

type mismatch for -: -（负号）操作符两边类型不匹配

undefined variable or const or function: 未定义的常量或变量或函数

undefined function: 未定义的函数

use a invalid label: 引用了非法的goto label

#### 开发体会与小结

在此次编写编译器系统设计程序的过程中，我所负责的部分是语义分析与符号表的设计与实现。我发现小组成员之间的沟通是十分重要的，由于不同模块之间需要进行协作，必须要对自己的程序有可能会对之前或之后部分造成的影响进行描述，尽量避免『牵一发而动全身』的情况，这样会降低整个小组的开发效率。符号表部分，符号表本身的设计是重中之重，如何设计一个简单易用同时通用性强的符号表数据结构是一个比较大的挑战，目前的设计虽然可堪使用，但我对其还不是特别满意，如果再花一些时间对其进行重构可能会有更加优美合理的结构，但由于该部分牵扯范围较大，此次项目并没有足够的时间了。语义分析部分的编写主要考察程序员的细心，需要考虑各种各样语义不合法的情况，相当于提前将bug扼杀在摇篮中，在写的时候经常需要想，『如果这个地方用户不这样写会怎么办？』，稍不注意便有可能漏掉某种corner case，例如之前我们就没有考虑到某些情况下函数的参数必须是左值。因此这也是一个不小的任务。