**近期个人作品**

　　出于学习的目的，近期在尝试写一种新的计算机语言（编译器）。

　　这种语言并没有创造一种新的语法，也不使用固定的语法，而是提供一个语言框架，提供多种计算机语言协作编程的能力，使各种语言能够发挥各自的优点，共同构建一个软件系统。并提供了一系列的辅助工具，降低编写计算机语言的难度，使一种新的语言能够很容易添加到该语言框架中。

　　该设计理念基于一个简单的思想：语言只是提供了一种描述能力，描述要让计算机做什么事情，而在底层的运行逻辑都是一致的。

　　举个例子：C#语言提供了非常丰富的描述能力，例如为集合查询提供了linq语法：

// C#: 选择一个数组中所有的偶数：

int[] arr = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var items = from item in arr where item % 2 == 0 select item;

　　同样，在VB.net中，也提供了linq语法：

' VB.NET 选择一个数组中所有的偶数：

Dim arr() As Integer = { 0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9 }

Dim items = From item In arr Where item % 2 == 0 Select item;

　　随着C#语言描述能力的越来越丰富，其语法集合也越来越庞大，Linq，动态编程，异步编程等，使掌握一种语言的学习成本也随之增大。另一个问题是，这些好用的功能在VB.NET中也需要再实现一次，使语法符合该种语言的特征。

　　如果把linq独立出来作为一种新的语言，可以嵌入到其它语言中，那么每种语言都具有了linq查询的能力：

// C#

int[] arr = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var items = <`linq from item in arr where item % 2 == 0 select item `>;

' VB.NET

Dim arr() As Integer = { 0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9 }

Dim items = <`linq from item in arr where item % 2 == 0 select item `>;

　　类似的，正则表达式也属于一种语言，各种语言以函数库的形式提供正则表达式的功能。同样也可以把正则表达式也独立为一种语言，嵌入到其它宿主语言中。

string text = "I am a teacher.";

result = <`re match text with [a-zA-Z0-9]+ `>;

　　基于这种模式，我们写一种语言就不需要实现一个完整的编译器，而是只实现一种语法。也可以没有完整的功能，只实现一个语言片段（不能独立使用，例如linq），嵌入到宿主语言中，增强宿主语言的描述能力。编译器的优化可以作为另一个独立的课题来研究，语言的目的只是增强描述能力。

　　而April框架的首要目标，就是把现有流行的编程语言都融入到这个框架中。接下来，将融入更多的领域特定语言。

# ****语言的层次****

　　我们将论述一下为什么要使用语言层面的协作编程。

　　先从汇编语言说起，汇编语言是偏底层的语言，这种语言是给计算机安排好指令序列，让计算机完全按照我们的指令来运行。

　　汇编语言当然是不可取代的，不过，就应用场景而言，有时候我们只是希望计算机完成一项任务，而不关心计算机如何来做。就像上面的linq，可以用很清晰的语言来描述我们想要的结果，类似还有SQL，只需要描述希望得到什么样的结果，而如何来做可以交给数据库本身。

　　这至少有两方面的好处：1）降低编程的复杂性，提高编程效率，提高代码的可读性。2）底层的实现逻辑可以独立维护，可以由单核改为多核，可以由单点改为分布式，这些工作都可以对使用方透明。

　　我们都听说过命令式语言和声明式语言，命令式语言就是指导计算机如何来做事情（例如汇编语言），声明式语言就是只告诉计算机我们想要的结果（例如linq, SQL）。实际上大多数语言都不需要在汇编语言这种级别上来编程了，例如C/C++，已经脱离了指令序列，这就可以在编程的时候不需要考虑CPU兼容性了，只要有合适的编译器，这种语言就可以用在任何CPU上了。

　　大多数语言都是即有命令式编程的成分，也有声明式编程的成分，两种编程方式的区别主要在于抽象的层次，这种抽象层次决定了语言的层次。中级语言C/C++是对CPU的抽象；高级语言C#/Java是对内存的抽象；linq, SQL是对算法的抽象；而其它各种应用领域需要开发自己的领域特定语言（DSL）。

　　抽象层次越高，应用面就越小，所以汇编语言，C/C++不可能被完全取代。像SQL只能用在数据查询中，而计算机应用的领域越来越广，并且领域的边界也不一定很清晰，时常会遇到多个领域协作的情况。

　　April尝试推动多种语言协作编程的能力，当然，这并不是一种新的思想，像Groovy语言也提供了类似的能力。April现在的做的只是一种实现上的尝试。

# ****April提供了什么？****

　　April是一个语言框架，提供了一系列的通用功能来构建一个新的编译器。并提供了一系列接口，可以让一种语言自定义自己的特性，使得一种语言可以很方便地接入到该语言框架中。

**通用词法分析器**：提供一个通用的方法，可以根据所设置的参数，将代码分隔为一系列的词。

**通用语法分析器**：根据语法的描述规则，将语法构造为Dom树。

**通用表达式解析器**：根据运算符的优先级，将表达式构建为二叉树。

**一些通用方法**：比如字符串向数字的转换方法。

**编译为中间语言**：将DOM树编译为中间语言。

**解释执行器**：直接将中间语言作为指令集来运行。

**内存管理器**：内存分配与回收。

**即时编译器**：在运行程序集之前将其编译为二进制机器语言。

**公共类型系统**：提供一系类型公共类型，每种语言都可以匹配自己的类型系统。

**公共类库**：各语言可以共用同一套类库。

**编译器语言**：可以使用这种语言来方便地构造一个新的语言。

　　另外，April需要先实现了一个完整的宿主语言，用于开发公共类库，这个语言选择了C#，并借鉴了许多.NET的设计理念，比如值类型，引用类型，接口，方法，属性，事件，特性等。只实现它最基础的部分，像动态编程，异步编程，linq都将另外实现为一种语言。

　　April语言框架是使用C++来编写的，C#语言的实现也使用C++来编写。

待C#语言成熟之后，将使用C#实现一种编译器语言，进一步降低实现一种新语言的难度，就像写几个正则表达式那么容易，像linq，内部会将其按照所定义的模式翻译为几个函数调用，然后再实现这几个函数调用即可实现一种新的语言。

# ****为什么要写一种语言？****

　　写这种语言的目的并不是为了成为XX语言之父，我相信世界上并不缺一种并不出色的语言。主要目的是练习算法，数据结构，架构设计这些最基本的编程素养。

　　一种编程语言所蕴含的技术非常丰富，对各种开发能力要求非常高，在开发过程中可以不断推动我去学习和实践，有些看起来简单的技术，只有在真正实践过程中才能深刻理解。

　　世上没有难做的技术，只有难练的基本功。

# ****核心算法****

这里是几个相对独立的算法

**数字转换**：一个相对简单的算法，实现了从各种字符串到数字的转换。

**语法解析器**：根据定义的模式对源代码进行匹配，构建成语法树。

**表达式解析器**：按照运算符的优先级，将表达式表示成二叉树的形式。

**其它算法**：综述一下April中的算法实现。

　　关于C++代码的内存分配与释放问题：虽然代码比较多，但通篇出现了不足10个new与delete。这是使用一个内存资源池的方案来实现的，会首先针对某个独立的任务建立一个资源池，在任务完成的时候释放该资源池，所有分配的内存都会释放。如果是该任务的目的是创建一个对象，那么该对象应该放在父资源池中，该父资源池会在任务开始的时候作为参数传递给该任务执行模块。由于内存的释放不需要显式进行，因此解决了内存泄漏的问题。

**数字转换**

　　这是一个相对简单的算法，实现了从各种字符串到数字的转换。

　　支持以下数字类型的转换：

123 // Integer

123.456 // Double

0123 // Octonary

0xF0 // Hexadecimal

0B010101 // Binary

123e+10 // Scientific notation

10F0H // Hexadecimal

123L // Long integer

123.456F // Float

123.456L // Long double

123U // unsigned int

123UL // unsigned long

123'456'789 // Numeric separator

　　由于有多种不同的表示方式，首先要根据字符串的特征确定其数字类型，进制。例如以0x开头表示为16进制，中间有e表示为科学计数法，有小数点表示为浮点数，有L表示为长整型，有U表示为无符号等，如果没有明确指定为长整型，整型的长度需要在转换时再确定。

　　由于有这么多判定准则，如何写一个高效的算法就变得有些复杂。本算法大量使用类模板来定义各自的转换算法，将不同的类型交与不同的类模板实例来处理，对于整型长度不确定的情况，使用几种类型模板实例接力来处理的方式，避免了重复尝试。

　　源码地址：

<https://github.com/xuyouchun/april/blob/master/algorithm/numeric.cpp>

**通用语法解析器**

　　这是一个复杂的算法。

　　通用语法解析器可以根据定义的模式来匹配源代码，将源代码构建成语法树。

　　模式定义的语法如下：

$: $class; # 由$定义根节点，此处定义了该语言有一个$class分支。

$class: ($method | $field)\* ;

# $class分支由$method或$field分支组成，用竖线来

# 表示“或”，括号后面的\*表示括号里的内容可以重复0次

# 或多次，+表示可重复1次或多次，?表示可重复0次或多次。

$field: int name ( = value)? \; ;

# 其中的name和value都是词法分析器返回的token的

# 名字，\表示转义。

　　上述只是简单的语法，而实际情况则复杂得多，比如下面是C#表达式的语法定义：（依然是简化了一下）

$\_single\_expression: ($\_\_item:

# $\_\_item子分支直接定义在了$\_single\_expression分支内

name | $cvalue | base | this

| $index | $function | $new | $new\_array | $type\_of

# 这些分支都另外有单独的定义

| $type\_cast\_exp| $default\_value | $type\_name\_exp

# 这些分支都另外有单独的定义

| \( $\_expression (, $\_expression)\* \)

) (. $\_\_item)\* # 表示$\_\_item子分支可以以.分隔的形式重复若干次

;

$\_expression: # $\_expression是多个$\_single\_expression的组合

$\_single\_expression

(

(

\+ | - | \\* | / | %

| \+= | -= | \\*= | /= | %=

| = | == | != | `> | >= | `< | <=

| << | <<= | >> | >>=

| && | \|\|

| \| | ^ | & | \|= | &= | ^=

| as | is

) $\_expression # binary operator： x + y

| \? $\_expression `\: $\_expression # xxx? xxx : xxx ? :

| (\+\+ | --) # right unary operator x--

)\*

| (`\+ | `- | `\+\+ | `-- | ! | ~ ) $\_expression

# left unary operator: ++x, -x

;

　　上述包含了递归定义，即$\_expression与$\_simple\_expression相互嵌套，通用语法分析器需要处理这种嵌套的情况。

　　可以看出，这是一个非常复杂的算法，不仅仅是正则表达式那样简单的模式匹配，也不是一种简单的状态机，当然也是使用了状态机的思想，但还需要处理错踪复杂的的引用和循环引用的情况。另外，如此复杂的定义，难免会出现多种匹配结果，需要决定哪一种匹配结果才是正确的。

　　有一些词法上的语义，也是在语法分析器的环节才得出结论，比如 “<” 在词法分析中只能分析出可能是小于号或左括号，在语法分析中便可根据所定义的模式进一步确认结论。

　　性能也需要尽量高，需要首先对分析树进行去冗余等优化处理，这也是一个复杂的算法。由于这是通用语法分析器，所以总体性能比专用语法分析器略差一些（相对于go-lang），但考虑到语法分析只是编译过程中的一个环节，所以这种代价还是可以接受的。在Mac Air上进行单核测试，每秒钟可以匹配6万行代码（不包括其它编译环节）。

　　C#语言的完整语法定义：

<https://github.com/xuyouchun/april/blob/master/modules/lang_cs/text.src>

　　源码地址：(代码量上万行)

<https://github.com/xuyouchun/april/blob/master/modules/compile/__analyze_tree.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/analyze_tree.cpp>

**表达式解析器**

　　按照运算符的优先级，将表达式表示成二叉树的形式。例如 a + b \* c，需要解析为 a + (b \* c)。

　　《编译原理》中有表达式解析的算法，不过在这里用了一个更易理解的算法：使用双栈的方式，将变量与运算符分别入不同的栈，当然运算符的优先级递增的时候，便继续入栈，否则将依次提取出运算符与变量合并为表达式，并将新合成的表达式入变量栈。

　　实际算法要复杂得多，因为要考虑多种场景，比如要考虑左结合与右结合，要考虑一元还是二元，还要处理三元运算符?:的情况。

　　April提供了接口，允许每种语言对表达式的解析过程中的某些步骤进行自定义。

　　源码地址：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__analyze_stack.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/analyze_stack.cpp>

其它算法

　　April中还有若干算法，大多都比较复杂，难以一一展开论述，以下是几个源码地址：

　　字符串各种编码的转换：

<https://github.com/xuyouchun/april/blob/master/common/__string.h>

　　若干集合类的定义：

<https://github.com/xuyouchun/april/blob/master/algorithm/__collections.h>

　　有关字符串的一些简单算法：

<https://github.com/xuyouchun/april/blob/master/algorithm/__string.h>

<https://github.com/xuyouchun/april/blob/master/algorithm/string.cpp>

　　对线程，并发，同步的封装：

<https://github.com/xuyouchun/april/blob/master/algorithm/__thread.h>

　　Assembly的读写：

<https://github.com/xuyouchun/april/blob/master/modules/core/__assembly_layout.h>

<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_layout.cpp>

<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_reader.cpp>

<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_writer.cpp>

　　语法树的构建：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__ast.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/ast.cpp>

　　表达式转换为中间语言指令集：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__expression.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/expression.cpp>

　　语句转换为中间语言指令集：（例如把for语句转换为跳转指令）

<https://github.com/xuyouchun/april/blob/master/modules/compile/__statements.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/statements.cpp>

　　局部变量，字段布局：

<https://github.com/xuyouchun/april/blob/master/modules/rt/__layout.h>

<https://github.com/xuyouchun/april/blob/master/modules/rt/layout.cpp>

　　解释运行器：

<https://github.com/xuyouchun/april/blob/master/modules/exec/commands.cpp>

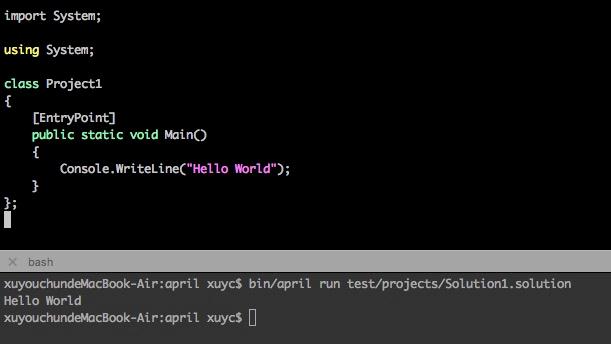
# 功能演示

　　实现一个C#编译器是一个庞大的工程，目前只有业余的时间可以用于编码和研究，个人的时间与能力有限，目前只实现了一个可以解释运行的雏形（和Python性能相当），一些重要的模块还没有实现，比如内存回收器，目前正在研究这方面的内容。

　　代码优化的技术也在研究中，借鉴业界成熟的经验，里面的算法相当复杂。

　　编译成机器代码的工作，打算使用业界成熟的编译器后端LLVM，目前还没有开始研究这方面的技术。

* **从一个Hello World开始**



　　打印出Hello World的那一刻，实际上已经走过了很长的路，词法分析器，通用语法分析器，已经开发完成，语义分析器，中间代码生成器，解析执行器已见雏形，然后的就是逐渐扩充其语法。

　　和C#的语法尽量保持兼容，但也有一些差异，比如可以用import来导入程序集，用标有[EntryPoint]的静态方法来表示运行入口。

**源码地址：**

　　词法分析器：

<https://github.com/xuyouchun/april/blob/master/modules/lang_cs/cs_token_reader.cpp>

　　语法分析器：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__analyze_tree.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/analyze_tree.cpp>

　　语法树的构建：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__ast.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/ast.cpp>

　　指令集定义：

<https://github.com/xuyouchun/april/blob/master/modules/core/__xil.h>

<https://github.com/xuyouchun/april/blob/master/modules/core/xil.cpp>

　　语句生成伪指令集：（伪指令集是从语句到指令集的过渡）

<https://github.com/xuyouchun/april/blob/master/modules/compile/__statements.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/statements.cpp>

　　伪指令集生成指令集：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__xilx.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/xilx.cpp>

　　表达式生成指令集：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__expression.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/expression.cpp>

　　程序集的读写与分析：

<https://github.com/xuyouchun/april/blob/master/modules/core/__assembly_layout.h>

<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_layout.cpp>

<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_reader.cpp>

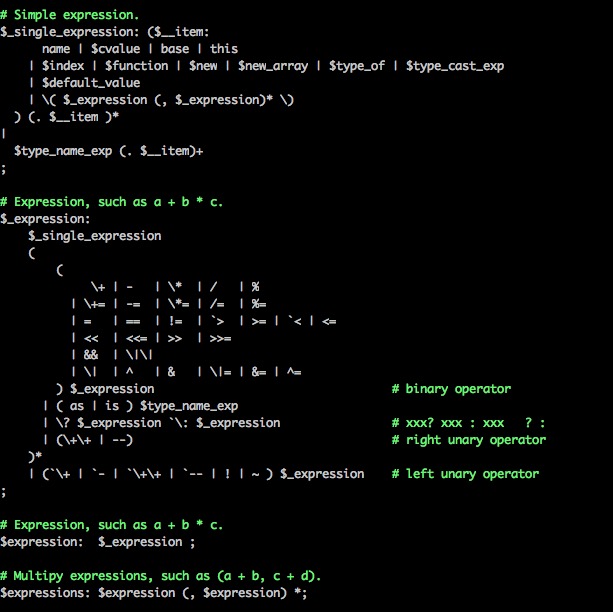
<https://github.com/xuyouchun/april/blob/master/modules/core/assembly_writer.cpp>

　　解释运行器：

<https://github.com/xuyouchun/april/blob/master/modules/exec/commands.h>

<https://github.com/xuyouchun/april/blob/master/modules/exec/commands.cpp>

* **通用语法分析器的模式匹配文法**



　　这是一个C#表达式的文法表示，使用这种文法定义一种语言，通用语法分析器根据该文法匹配出代码的各个部分。

　　这种文法由几个简单的语法，?表示前面的内容可有可无，\*表示可以重复0次或多次，+表示可以重复1次或多次，例如 name+ 表示name至少出现一次，以竖线"|"连接各个可选部分，整个C#的由近百条文法定义组成，文法之间可以相互嵌套，甚至可以包含自己。

　　使用语法制导翻译技术，通用语法分析器首先读入文法，先进行文法的优化，去掉冗余部分，在进行代码匹配的过程中动态生成自动机，这是一种图的结构，在匹配过程中需要处理的情况极多，防止分支蔓延，选择最佳匹配路径，处理死循环（消除左递归），还需要尽量保证其性能。

　　在Mac Air上进行单核性能测试，每秒能够达到匹配6万行代码的性能。实际上这种CPU密集型计算可以进行多核并行处理，这个步骤的资源争抢甚微。

　　C#的完整文法定义：

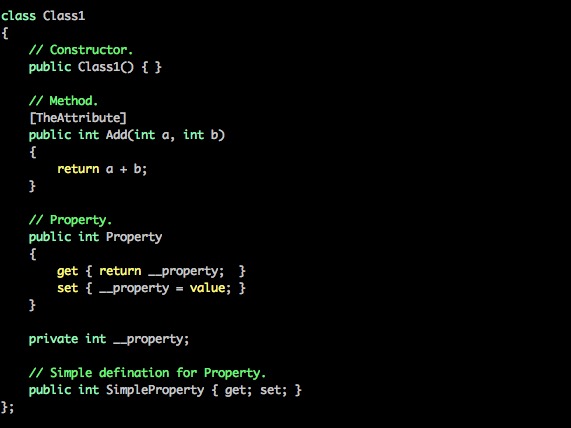
<https://github.com/xuyouchun/april/blob/master/modules/lang_cs/text.src>

　　通用语法分析器源码地址：(代码量近万行)

<https://github.com/xuyouchun/april/blob/master/modules/compile/__analyze_tree.h>

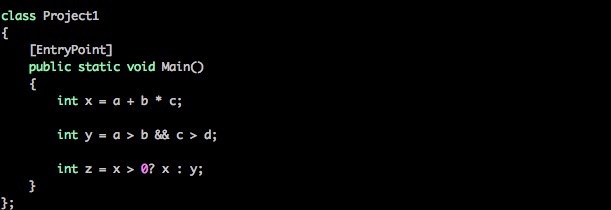
<https://github.com/xuyouchun/april/blob/master/modules/compile/analyze_tree.cpp>

* **基本语法**



　　类型定义，属性，索引，事件，方法，特性，与C#的实现保持一致。

* **表达式分析器**



　　表达式的解析，支持运算符优先级，将表达式解析成为二叉树的形式，该表达式分析器提供了一个算法框架，便于新的语言定义新的运算符。

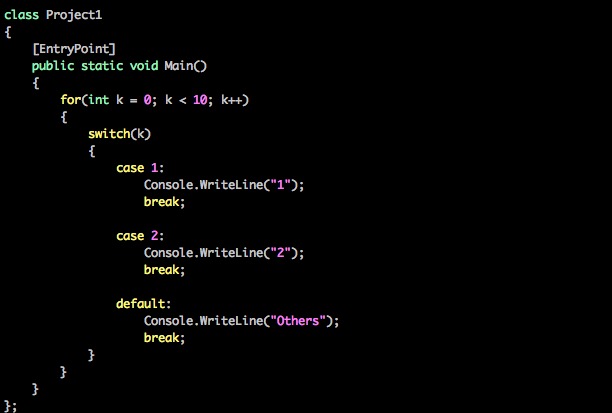
　　C#中的三元运算符"?:"的解析算法要复杂一些，把它们视为紧密相连的两个二元运算符来处理，依然将其实现在算法框架中了。

　　源码地址：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__analyze_stack.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/analyze_stack.cpp>

* **控制语句**



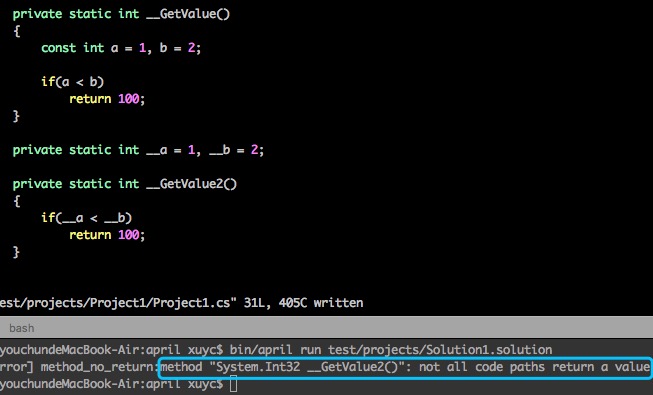
　　实现了C#中所有控制语句，包括for, while, do...while, switch，这些控制语句都是先生成一系列的伪指令集，便于分析与优化，再进一步生成指令集。

　　相关源码：

<https://github.com/xuyouchun/april/blob/master/modules/compile/__statements.h>

<https://github.com/xuyouchun/april/blob/master/modules/compile/statements.cpp>

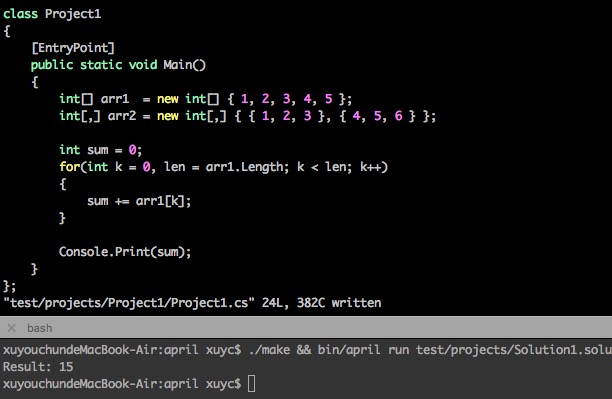
* **语义分析**



　　目前语义分析可以去掉一些执行不到的多余代码，检查函数的执行路径，优化一些始终为真或始终为假的判断条件等，为代码的进一步优化做好铺垫。

　　如上图所示：\_\_GetValue中的 a < b 的条件始终为真，return语句一定会执行到，而\_\_GetValue2中的两个静态字段是不确定的值，所以会报出编译错误：Not all paths return a value。

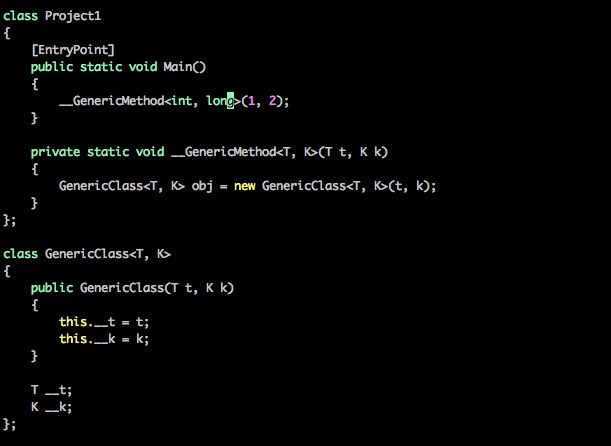
* **数组 & 多维数组**



　　数组，多维数组的实现。出于性能方面的考虑，数组在创建时的初始化有专门的指令来完成。

　　源码分布在若干位置，通常都带有array字样。

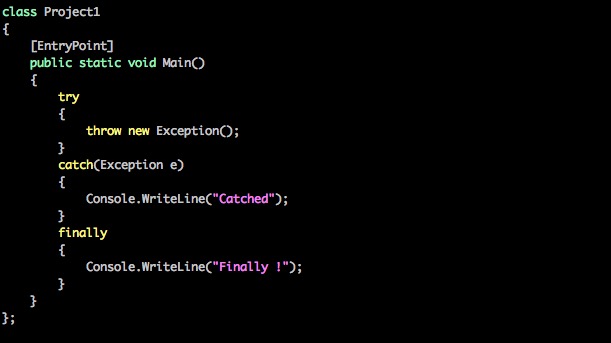
* **泛型**



　　泛型方法，泛型类型皆已支持。泛型的支持增加了不少的工作量，包括指令集的设计，程序集中相关的泛型描述，各种类型检查，以及运行时的动态编译等。

　　源码分布在若干位置，通常都带有generic字样。

* **异常处理**

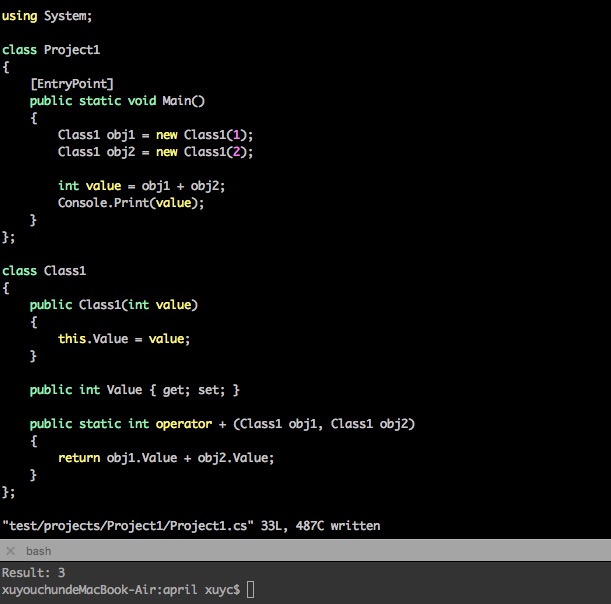


　　实现了异常处理之后，才明白原来C#对异常的处理需要有这么多地方需要考量，包括查找异常处理代码块，查找finally块，函数调用栈的回退，还要处理相互嵌套的情况，还有finally中再次抛出异常的情况。

　　异常处理的实现在解释运行器中：

<https://github.com/xuyouchun/april/blob/master/modules/exec/commands.cpp>

* **运算符重载**

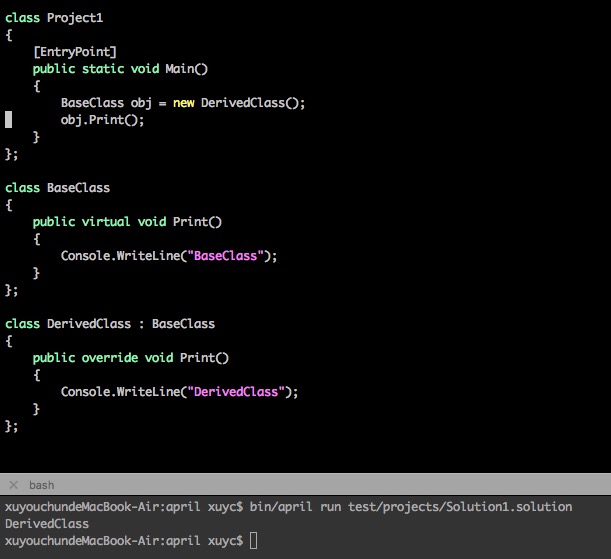


　　运算符重载的实现相对简单一些，将该函数转换为一个普通的方法即可，但需要检查参数类型及个数是否符合该运算符的要求。

　　相关源码包含在以下文件中，会遍历表达式中运算符作用于普通对象的情况，将其转换为相应的函数调用。

<https://github.com/xuyouchun/april/blob/master/modules/compile/ast_utils.cpp>

* **派生 & 虚函数**

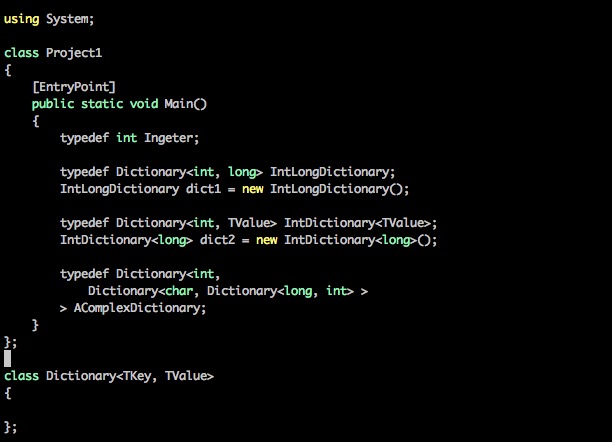


　　虚函数的实现，和C++中的思路一致，就是为虚函数建立虚函数表，保存在该对象的类型数据中，在运行的时候从该对象的类型数据中寻找真实的函数地址并执行。

　　相关源码包含在以下文件中，build\_vtable函数。

<https://github.com/xuyouchun/april/blob/master/modules/rt/rt.cpp>

* **扩展语法 typedef**



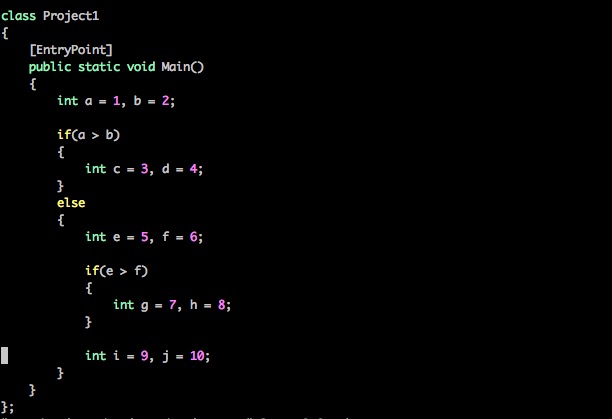
　　typedef是C/C++的特性，C#中并没有提供，个人感觉这个功能非常有用，尤其用在泛型定义中，于是将其实现在了C#中。

　　可以将一个简单类型定义成另一个名字，也可以将一个泛型定义成另一个名字，还可以在泛型定义中只指定一部分泛型参数，定义一个偏特化的类型。

　　相关源码包含在以下文件中，主要是对typedef所定义类型的解析：

<https://github.com/xuyouchun/april/blob/master/modules/compile/ast_utils.cpp>

* **局部变量的布局**



　　由于局部变量使用特征，有的局部变量不会同时使用，它们可以共用栈的空间，如何布局这些局部变量是一个非常有趣的算法。

　　比如上图中，c,d 与 e,f 不会同时使用到，它们可以共占栈空间，g,h 与 i,j 也可以共占栈空间。

　　相关源码：

<https://github.com/xuyouchun/april/blob/master/modules/rt/__layout.h>

<https://github.com/xuyouchun/april/blob/master/modules/rt/layout.cpp>

* **项目工程管理**

　　和.NET一样，使用project与solution的概念来管理项目与工程，也是使用XML格式的描述文件来描述一个项目或解决方案。

　　相关源码皆在以下目录：

<https://github.com/xuyouchun/april/tree/master/modules/april>