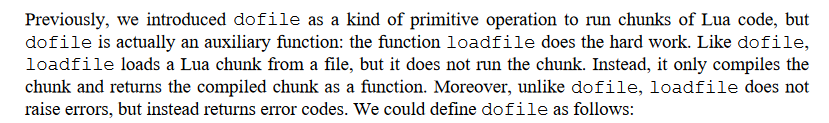
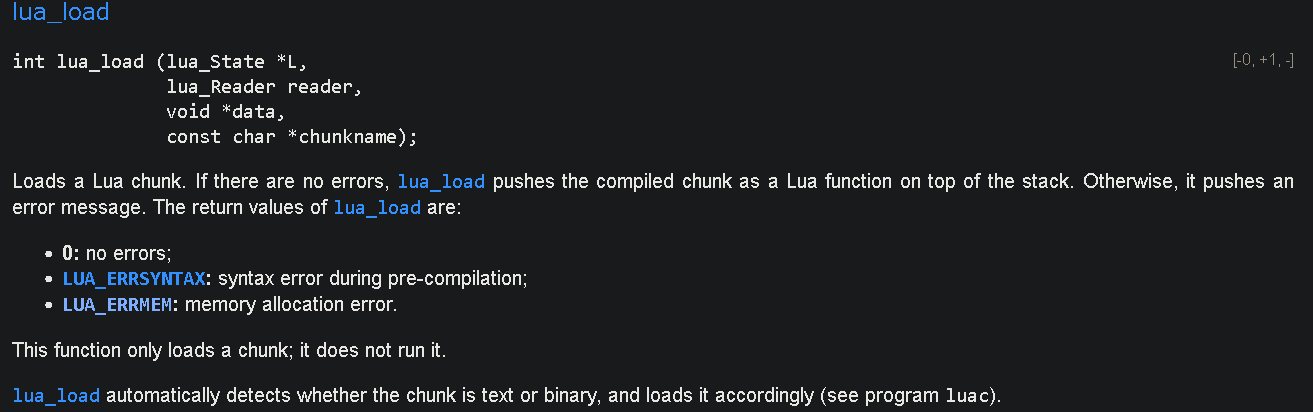
**Lua编译器以及解释器分析**

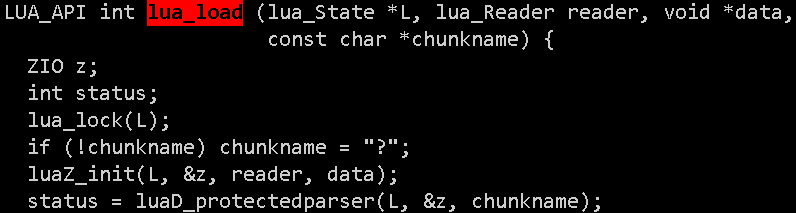
**Author: MG193.7 程为民 20240613**

由于小米路由器特有的Lua编译器会在编译时隐藏调试细节, 并混淆部分寄存器操作, 故本档意在分析luac的编译流程以辅助Unluac优化器的开发. 当luac开始执行时, luac.c源文件内的pmain函数将在早期被调用, 接着, Luac将携带全局lua\_State结构以调用解释器的关键入口**luaL\_loadfile**. 关于该全局结构, 根据Pil文档(Programming in Lua)描述可知, 其内部存储了Lua的所有运行状态, 负责记录解释/编译时产生的各类元数据, 并且维护LuaVM使用到的全局虚拟栈(Lua的大部分数据交换甚至函数调用都基于虚拟栈, 该部分详看Pil文档), 且此结构全程以单例模式运行.

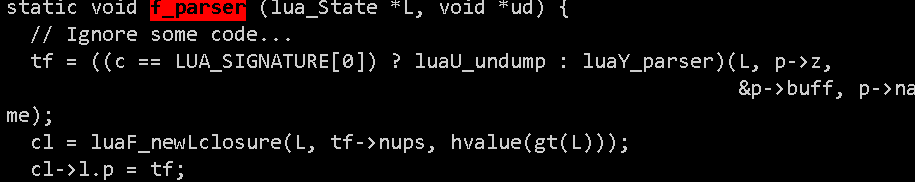


根据Pil文档所述, luaL\_loadfile接口会将欲解释的Lua源代码文件加载为Lua chunk, 但该过程只负责编译Lua chunk, 而不对其进行执行操作. 需要注意的是, 在Lua中, chunk是一种概念, 任何合法的代码片段都可以称作chunk, 该部分详看Pil文档的**Chunks**章节. 接着, loadfile将通过lua\_load接口以编译提取出的chunk片段. 根据lua.org操作手册的描述可知, 若load接口的编译过程没有发生错误, 则代码会将编译后的chunk作为Lua函数并存储在虚拟栈顶部, 最终返回状态”0”. 当然, 此虚拟栈由全局lua\_State维护.

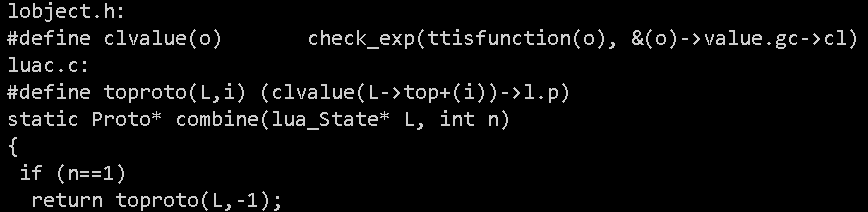
 https://www.lua.org/manual/5.1/manual.html



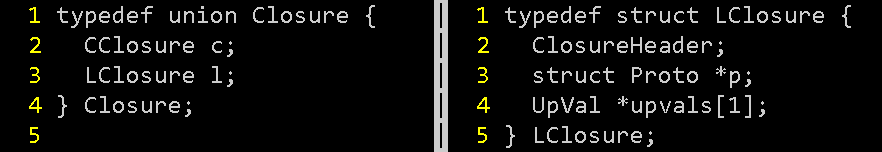
观察lua\_load接口的具体实现, 可见代码通过Z\_init接口初始化了流操作结构ZIO, 该结构用以读入/写出编译时产生的数据, 本档不关心ZIO的具体实现, 接下来, 代码携带ZIO并调用了luaD\_protectedparser函数. 函数内, 代码随即调用f\_parser接口以预处理待编译chunk.



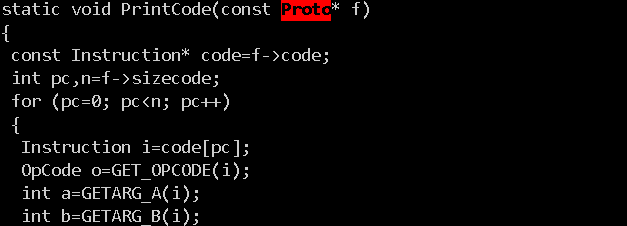
忽略f\_parser内的ZIO操作, 代码会首先判断Lua chunk是否带携带头部魔数(编译后的二进制chunk首部将携带签名魔数”\0x33Lua”), 若存在, 则使用luaU\_undump接口直接抽取编译后chunk内的字节码, 并以此构造Proto结构体, 该结构体内携带具体的指令对象(Instruction)指针code成员以及其它字节码元数据, 若chunk为非编译的源码形式, 则使用luaY\_parser接口对源码进行编译/解释. 而本档假设的场景为使用Luac编译源码文件, 故此, 后续将跟入分析Y\_parser接口.



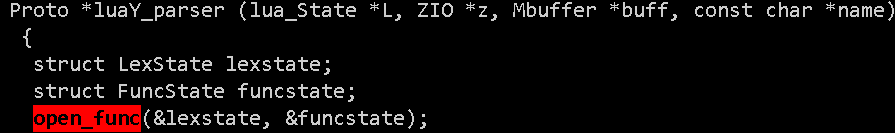
在分析Y\_parser接口实现之前, 本档首先需要分析关键结构体**Proto**在编译前后的作用. 可以发现, luac.c源文件中的pmain函数引用了该结构体. 在所有解释与编译环节结束后, pmain将调用combine接口将所有Lua源代码文件编译后产生的二进制chunk进行合并, 以输出连续的指令集方便后续解释, 如果只有一个源代码文件参与编译, combine将直接调用toproto宏, 该宏将访问全局lua\_State维护的虚拟栈中最顶层的StkId结构, 并从中取出成员名为”gc”的可回收对象(GCObject), 以访问Closure联合体结构.

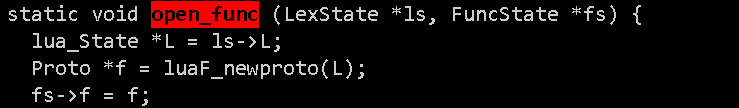


观察Pil文档以及编译器代码上下文, 可以将Closure视为用于存储函数实体的联合结构, 其内部进一步分为CClosure与LClosure结构, 显然, 二者用以存储不同类型的函数实体. 由toproto宏访问虚拟栈的实现代码”value.gc->cl->l.p”可知, 宏通过成员名为”l”的LClosure结构以访问其内部存储的Proto结构体, 显然, LClosure用于存储Lua函数实体. 至于为什么要访问lua\_State虚拟栈的最顶层, 上文在分析load接口及其操作手册时已经解释过, 所有编译后产生的chunk都会被推入虚拟栈顶, 所以, 当只有一个源代码文件参与编译时, 宏仅需要访问栈顶结构就可以获取Proto.

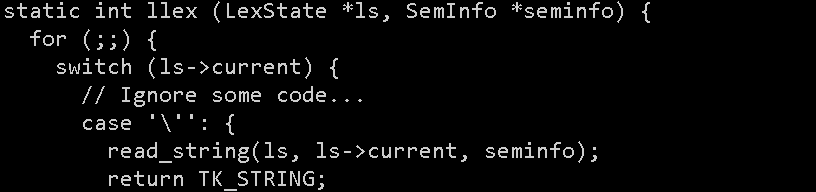


假设当前场景下, Luac使用了”-l”参数, 那么在pmain获取到Proto后, Luac将携带Proto指针调用luaU\_print函数, 最终, 函数会将指针传递至PrintCode接口. 通过观察代码可以发现, 接口会提取Proto内的指令对象指针, 并模拟pc指针对字节码串内的所有指令进行解释和输出, 在访问指令的同时, 代码还将解析指令的操作数. 显然, Proto在编译时参与了字节码的存储, 在编译后参与了指令集的解释, 该结构体贯穿了整个编译与解释流程, 故此, 在后续的分析中, 将随时关注代码针对Proto的操作.

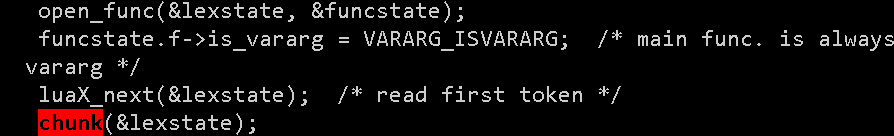


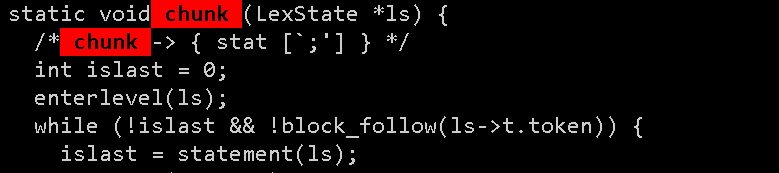


luaY\_parser是将源码编译为二进制chunk的入口, 也是词法与语法解析前的预处理阶段, 其首先会为LexState与FuncState结构分配内存, 根据代码上下文可知, LexState用于维护当前词法解析产生的元数据与各类状态, FuncState则负责管理当前函数的编译状态, 其结构内包含的成员名为”f”的Proto结构指针指向当前正在编译的chunk. 随后, parser将二者结构的引用传递至open\_func接口并处理其它初始化操作. 其关键操作包括: 利用luaF\_newproto接口为当前词法解析分配Proto结构指针, 并将该指针转储至LexState内的f成员, 随后, FuncState引用被转储至LexState内的fs成员.

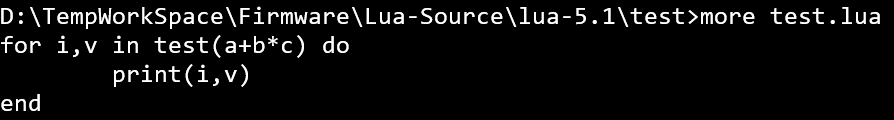


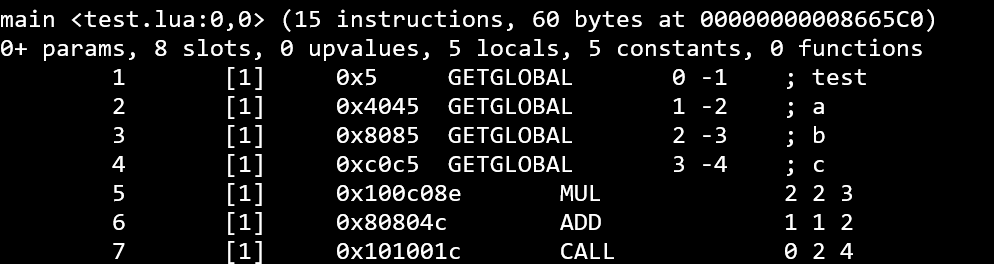
一旦初始化完成所有必要参数, luaY\_parser会将当前的LexState引用传递至**luaX\_next**函数. 通常, 在无需预读词法符号的情况下, 函数会调用llex.c源文件内的llex接口, 并传递由parser输入的LexState指针. llex接口负责提取源码内的词法符号(即Token), 需要注意的是, 接口并不会一次性解析出全部的词法符号, 而是单次执行仅解析一个符号, 当成功解析出任意符号时, 接口会立即停止后续解析, 并返回当前的符号类型. 同时, llex接口在词法解析过程中, 会同步更新LexState指针内类型为Token结构的t成员为当前解析出的符号, 此外, 指针内的其它元数据, 例如当前字符与源码行号也会被相应调整.



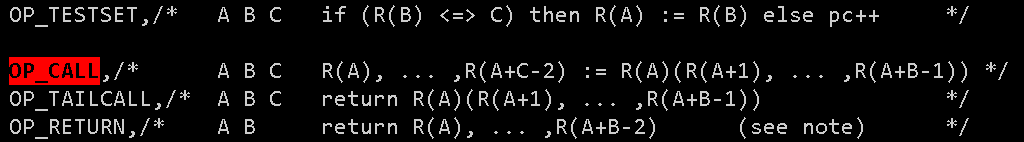


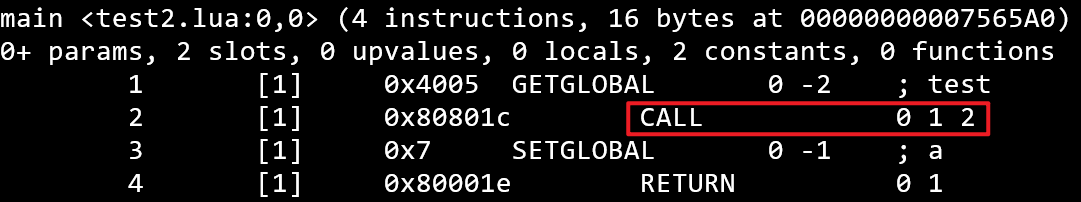
在X\_next解析出首个符号后, Y\_parser会立即调用chunk接口并将LexState引用传入. 可见chunk接口的具体实现, 其在while代码块内调用了statement函数, 并将返回值将赋予局部变量islast, while循环的判断条件则与该局部变量有关, 结合代码上下文不难推断出, statement是编译器进行语法解析的具体实现, 而islast标志着解析是否结束. 观察chunk接口内的首行注释, 这是拓展巴科斯范式(EBNF)的标准表述形式, 其通过描述上下文无关文法(CFG)来解释编程语言中的语法定义, 关于CFG与BNF, 详看<编译原理>第4.2章节的解释. 根据注释可知, 编译器定义chunk由多个stat组成, 因此, 接口才利用while循环以及statement函数来依次解释源码(chunk)内的所有stat.



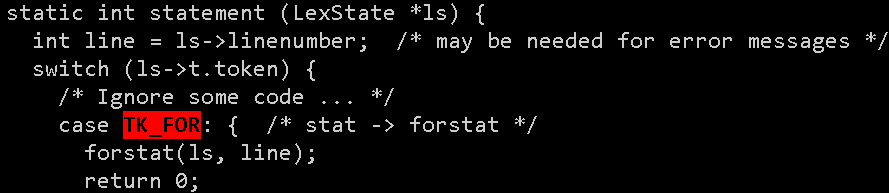


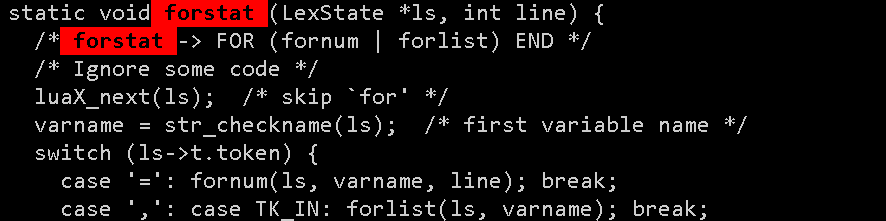
在正式分析statement之前, 本档提供了一段示例代码”test.lua”作为编译器的测试用例. 用例中, 关键字”in”后方的条件表达式”test(a+b\*c)”显然是函数调用操作, 利用”-l”参数Dump出代码对应的指令串, 可以看到首七条指令即为该条件表达式的具体实现, 忽略寄存器(虚拟栈)操作指令以及运算指令, 末尾CALL指令的第三个操作数为4, 显然这与常规情况不符. (注意, 本档修改了PrintCode接口的部分代码, 编译器在输出指令串的同时会输出指令对应的字节码)



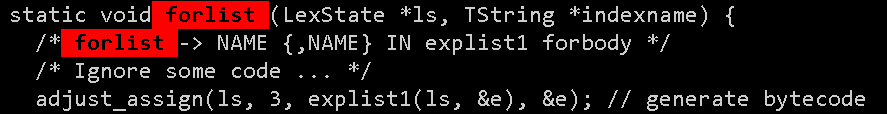


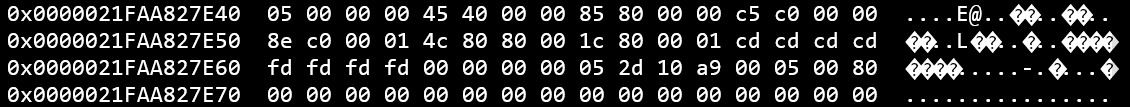
本档约定, 将任意指令的操作数按序以别名”A,B,C…”进行定义, 此外, 下文使用符号”R(X)”将操作数X解释为”寄存器X”. 那么, 根据编译器对CALL指令的描述, 以及<[A No Frills Intro To Lua VM Inst](http://www.mcours.net/cours/pdf/hasclic3/hasssclic818.pdf)>文档的指令解释列表可知, 该指令的操作数C定义了CALL操作的返回结果数量, 所有返回结果按序存储在寄存器R(A)至R(A+C-2)中, 当操作数C为2时, 表示该指令要求存储一个返回值, 理论上, 为了引用循环条件, Luac应当为条件表达式分配单个寄存器以存储返回结果, 但编译器却为表达式分配了操作数C为”4”的CALL指令. 显然, 此时可以猜测, 编译器在对”for”句柄进行语法解析时, 隐含地为关键字”in”后方的条件表达式分配了三位寄存器, 但问题是, 该解析过程是如何进行的? 本档将保留该疑问与猜测, 并在下文对statement中的语法解析流程进行详细分析.



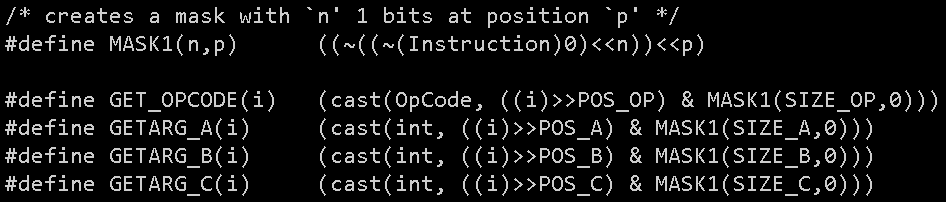


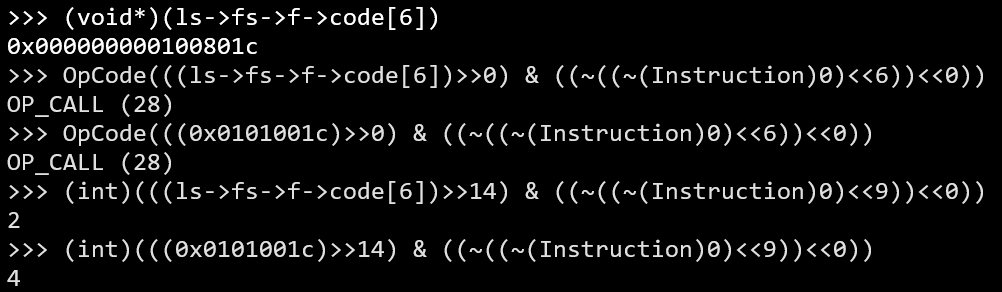
statement函数首先从LexState指针中获取当前Token结构, 并利用switch句柄为结构内的token成员分配合适的case分支, 在本测试用例中, X\_next的首次调用将解析出”for”符号, 该符号及对应的结构将被编译器引导至TK\_FOR分支内. 随后, forstat函数被调用以进一步解析语句. 此函数会继续解析出下个符号, 并分配相应的处理分支, 通常情况下, 编译器将开始解析”for”关键字后的循环参数, 但本档不关心参数的处理, 故忽略该解析过程, 直接进入TK\_IN分支, 在forlist函数内, 编译器将处理条件表达式.



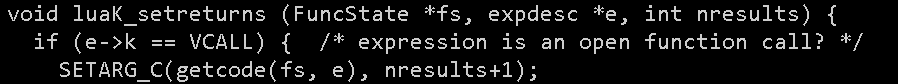


注释中EBNF显示, 编译器以别名”explist1”来定义条件表达式. 忽略无关代码, 不难发现Luac处理条件表达式的函数同样名为”explist1”. 本档暂时不对表达式的解析细节进行分析, 相反, 假设表达式已经解析完毕, 此时Luac会将编译后得到的指令存储在LexState指针中的FuncState结构内. 获取首条指令的地址并Dump该处内存, 可以访问内存中总共28Byte长的字节码, 其内容与前文使用”-l”参数获取的首7位指令字节码基本一致, 但该段内存中的最后4字节与前文CALL指令对应的字节码有略微区别: 内存中, 从首地址偏移6个指针长度后可以得到”0x0100801c”, 而前文CALL指令对应的字节码为”0x0101001c”.

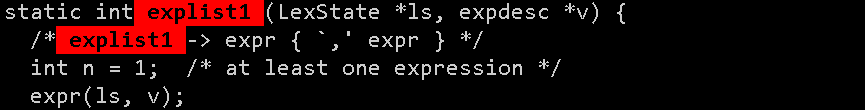




在opcodes.h源文件内, 编译器定义了数个用以从指令中提取操作码与操作数的宏, 通过宏内的运算操作, 可以轻易解析任意字节码. 在explist1执行完毕时通知调试器挂起进程, 并于当前代码上下文执行表达式”ls->fs->f->code[6]”, 检查执行结果, 确保访问的内存与欲分析内容一致, 随后提取二者的操作码, 可以发现表达式执行结果均为28, 即OpCode枚举内的OP\_CALL元素, 这证明访问的内存以及解析方法没有出错. 校准内存后, 提取目标中的操作数C, 并以之对比字节码”0x0101001c”内的操作数C, 可见二者在此处存在偏差. explist1在解析与编译条件表达式时, 为CALL指令分配了值为2的操作数C, 显然这符合预想中仅为条件表达式分配单个寄存器的情况. 而在adjust\_assign函数执行完毕后, 操作数C才被更新为4.

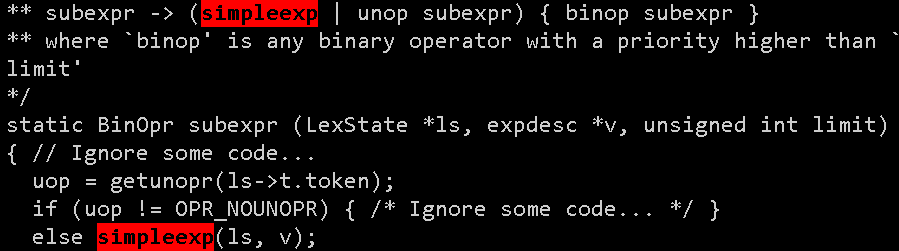


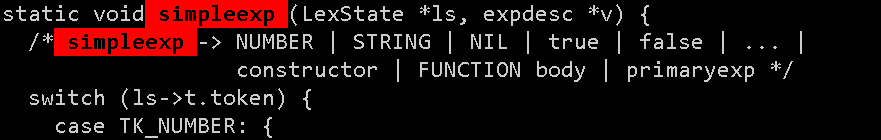
**expdesc**是编译器用以记录表达式描述信息的结构, 该结构中的k成员是描述表达式类型的expkind枚举. 前文中, forlist函数在入口处对该结构进行了申明, 并为其分配内存. 随后, 编译器调用explist1并传入该结构引用, 以此在语法解析时记录条件表达式的描述信息. 由于被解析表达式属于函数调用操作, 编译器自然会为描述结构的k成员分配VCALL元素(有关expkind枚举, 详看Luac源码). 最终, explist1执行完毕后, 代码将进入adjust\_assign函数以处理多赋值语句, 并根据表达式的左右值管理与分配合适的虚拟栈空间, 当描述信息中的表达式类型符合VCALL元素时, 函数将调用luaK\_setreturns接口, 并传入需要分配的寄存器个数, 接口会通过SETARG\_C宏更新CALL指令中操作数C的值为传入的寄存器数量+1, 由于forlist为adjust\_assign提供的寄存器分配数量为3, 故此, 最终条件表达式的CALL指令将得到值为4的操作数C, 这恰好验证了前文中的推测.



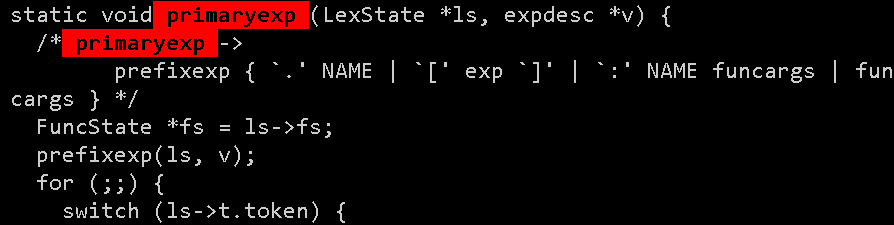


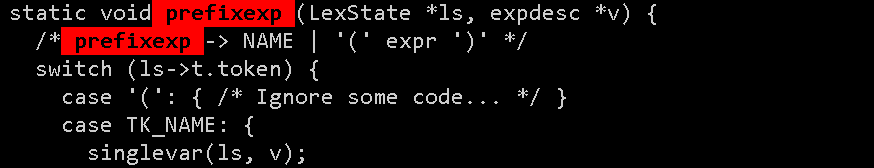
下文, 本档开始分析explist1中的语法解析细节, 且仍以”test.lua”测试用例中的条件表达式作为被解析对象. 由注释中的EBNF可知, 编译器定义explist1由多个子表达式”expr”组成, 在expr函数内, 代码仅执行一个动作: 调用subexpr接口并传入三个参数, 前二者是维护解析状态的指针”ls”与记录表达式描述的指针”v”, 前文已有分析, 不再赘述, 需要注意的是第三个参数, 它是初始值”0”.



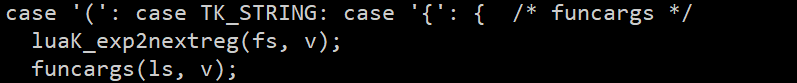


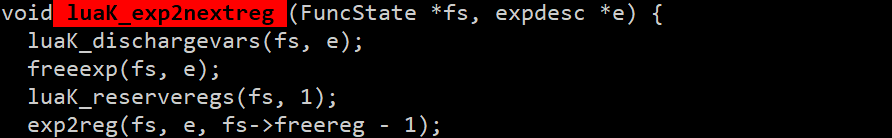
观察EBNF可知, subexpr由简单表达式或者单目表达式, 以及多个二元表达式组成, 并且定义单目表达式的处理优先级最高, 代码首先会通过getunopr接口尝试获取当前符号内的单目运算符, 但由于测试用例不涉及单目运算, 故此接口的返回结果为OPR\_NOUNOPR, 这将使得执行进入另一个分支, 即simpleexp接口. 顾名思义, 此接口用以处理简单表达式, 函数入口的EBNF定义, 简单表达式由数字, 字符串, 函数定义, 构造器以及若干保留字组成, 它们的共同特征是结构单一, 无复杂运算与代码嵌套, 故称之为”简单表达式”. 与statement函数类似, 代码通过switch句柄以及解析状态指针内的token成员分配合适的解析接口. 已知当前用例代码为表达式”**test(a+b\*c)**”, 为了方便后续分析, 假设使用llex接口对其进行完整的词法解析, 通过调试, 用例可以被拆分为如下符号: “(”, “)”以及运算符”+”, “\*”, 函数与变量名均被解释为符号”TK\_NAME”, 那么最终的解析结果即TK\_NAME(TK\_NAME+TK\_NAME\*TK\_NAME). 回到simpleexp接口, 此刻, 解析状态指针内的token成员指向表达式中首个TK\_NAME符号, 即函数名”test”, 由于该符号与表达式内的后续所有符号均不属于simpleexp的处理范围, 因此代码将分配primaryexp接口进行后续解析.





基本表达式(primaryexp)由单个表达式前缀(prefixexp)与若干表达式修饰部分组成, 修饰包括点运算符访问, 表访问与函数调用操作, primaryexp同样通过解析状态指针内的token成员来决定处理方式, 但与simpleexp不同的是, 由于表达式修饰部分的数量不是恒定的, 因此, 代码将switch处理句柄嵌入循环条件恒为真的for句柄内, 以便逐个解析基本表达式内的所有符号, 直至将修饰部分全部处理完毕. 首先观察表达式前缀的EBNF, 前缀由TK\_NAME或括号表达式组成, 由于当前token成员指向条件表达式的首个TK\_NAME符号, 所以编译器将调用singlevar函数以处理此符号. 函数会通过当前解析状态内FuncState以及全局lua\_State的局部与全局变量表查找符号索引, 并通过搜索结果标记此时的表达式描述类型为全局(VGLOBAL), 局部(VLOCAL)或上值(VUPVALUE). 在编译器处理该符号的同时, 代码将调用X\_next以将token成员移动到条件表达式内的下一位符号.





回到基本表达式处理函数, 此时编译器开始处理表达式修饰部分, 同时, token成员指向符号”(”, 根据处理分支下的代码, 可以判断编译器将表达式的后续修饰作为函数调用操作, 并通过funcargs接口进行处理.