Lua源码剖析（二）：词法分析、语法分析、代码生成

词法分析

lua对与每一个文件(chunk)建立一个 `LexState` 来做词法分析的context数据，此结构定义在llex.h中。词法分析根据语法分析的需求有当前token，有lookahead token， `LexState` 结构如图：

其中 `token` 结构中用 `int` 存储实际 `token` 值，此 `token` 值对于单字符 `token` (+ - * /之类)就表示自身，对于多字符(关键字等) `token` 是起始值为257的枚举值，在llex.h文件中定义：

```
#define FIRST_RESERVED	257


/*
* WARNING: if you change the order of this enumeration,
* grep "ORDER RESERVED"
*/
```

```
enum RESERVED {
  /* terminal symbols denoted by reserved words */
  TK_AND = FIRST_RESERVED, TK_BREAK,
  TK_DO, TK_ELSE, TK_ELSEIF, TK_END, TK_FALSE, TK_FOR, TK_FUNCTION,
  TK_GOTO, TK_IF, TK_IN, TK_LOCAL, TK_NIL, TK_NOT, TK_OR, TK_REPEAT,
  TK_RETURN, TK_THEN, TK_TRUE, TK_UNTIL, TK_WHILE,
  /* other terminal symbols */
  TK_CONCAT, TK_DOTS, TK_EQ, TK_GE, TK_LE, TK_NE, TK_DBCOLON, TK_EOS,
  TK_NUMBER, TK_NAME, TK_STRING
};
```

`token`结构中还有一个成员`seminfo`，这个表示语义信息，根据`token`的类型，可以表示数值或者字符串。

lex提供函数`luaX_next`和`luaX_lookahead`分别lex下一个`token`和lookahead token，在内部是通过llex函数来完成词法分析。

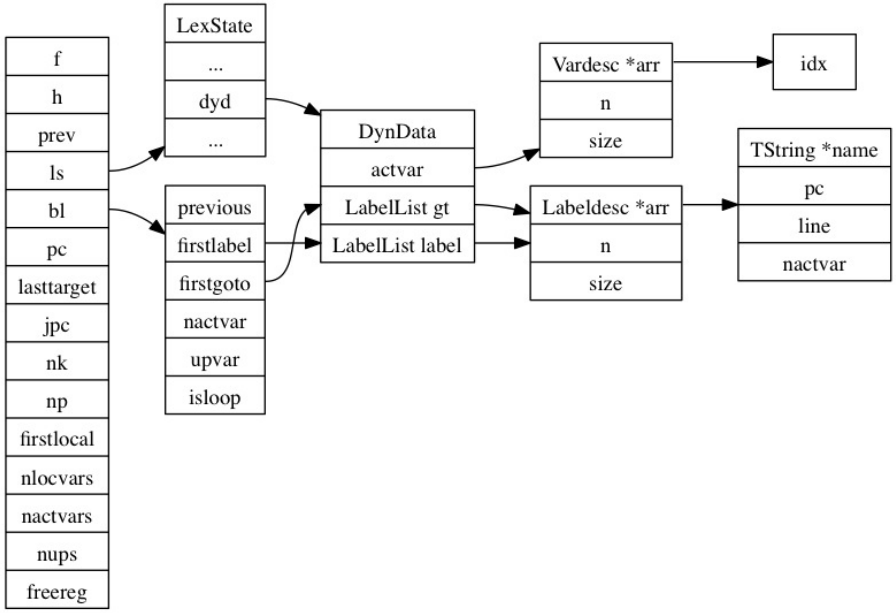
语法分析

lua语法分析是从lparser.c中的`luaY_parser`开始：

```
Closure *luaY_parser (lua_State *L, ZIO *z, Mbuffer *buff,
                      Dyndata *dyd, const char *name, int firstchar) {
  LexState lexstate;
  FuncState funcstate;
  Closure *cl = luaF_newLclosure(L, 1);  /* create main closure */
```

```
  /* anchor closure (to avoid being collected) */
  setclLvalue(L, L->top, cl);
  incr_top(L);
  funcstate.f = cl->l.p = luaF_newproto(L);
  funcstate.f->source = luaS_new(L, name);  /* create and anchor TString */
  lexstate.buff = buff;
  lexstate.dyd = dyd;
  dyd->actvar.n = dyd->gt.n = dyd->label.n = 0;
  luaX_setinput(L, &lexstate, z, funcstate.f->source, firstchar);
  mainfunc(&lexstate, &funcstate);
  lua_assert(!funcstate.prev && funcstate.nups == 1 && !lexstate.fs);
  /* all scopes should be correctly finished */
  lua_assert(dyd->actvar.n == 0 && dyd->gt.n == 0 && dyd->label.n == 0);
  return cl;  /* it's on the stack too */
}
```

此函数创建一个closure并把 `LexState` 和 `FuncState` 初始化后调用 `mainfunc` 开始parse，其中 `FuncState` 表示parse时函数状态信息的，如图：

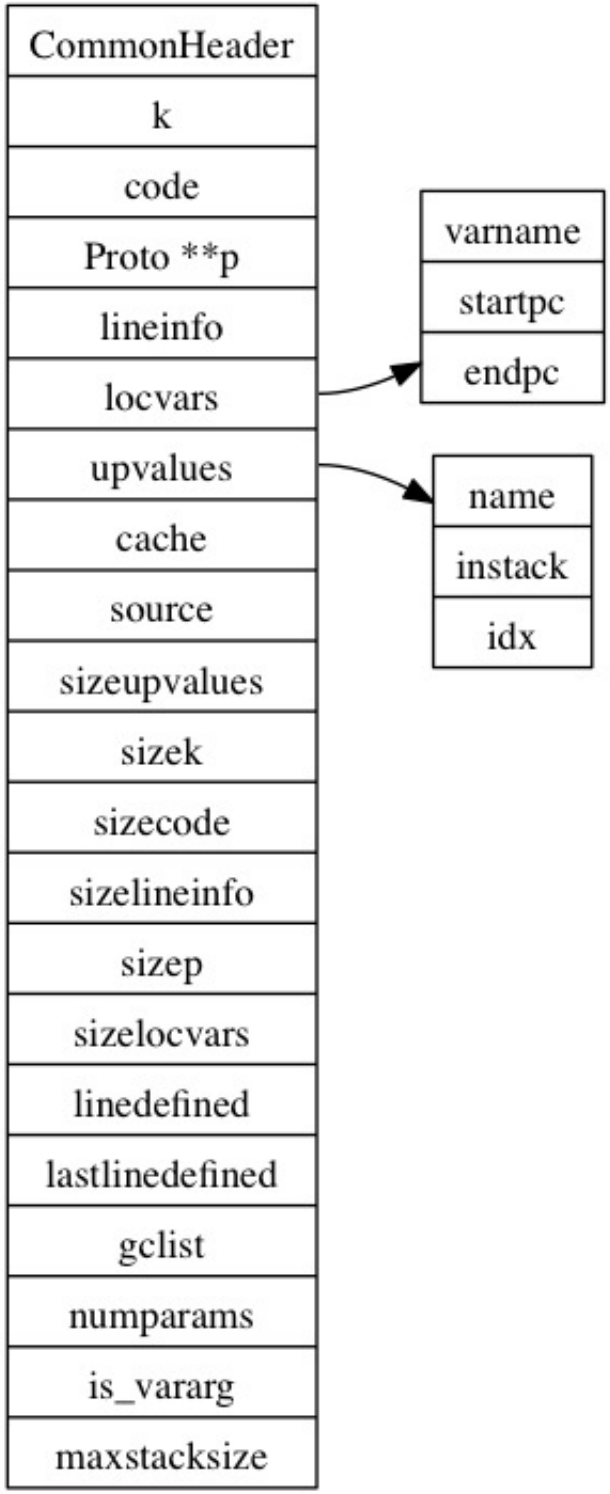


每当parse到一个function的时候都会建立一个 `FuncState` 结构，并将它与所嵌套的函数通过 `prev` 指针串联起来， `body` 函数就是完成嵌套函数parse。

```
static void body (LexState *ls, exposed *e, int ismethod, int line) {
  /* body ->  `(' parlist `)' block END */
  FuncState new_fs;
  BlockCnt bl;
  new_fs.f = addprototype(ls);
  new_fs.f->linedefined = line;
  open_func(ls, &new_fs, &bl);
  checknext(ls, '(');
  if (ismethod) {
    new_localvarliteral(ls, "self");  /* create 'self' parameter */
    adjustlocalvars(ls, 1);
  }
```

```
  parlist(ls);
  checknext(ls, ')');
  statlist(ls);
  new_fs.f->lastlinedefined = ls->linenumber;
  check_match(ls, TK_END, TK_FUNCTION, line);
  codeclosure(ls, e);
  close_func(ls);
}
```

`FuncState` 中的 `f` 指向这个函数的 `Proto`， `Proto` 中保存着函数的指令、变量信息、upvalue信息等其它信息， `Proto` 的结构如图：


CommonHeader
k
code
Proto **p
lineinfo
locvars
upvalues
cache
source
sizeupvalues
sizek
sizecode
sizelineinfo
sizep
sizelocvars
linedefined
lastlinedefined
gclist
numparams
is_vararg
maxstacksize
varname
startpc
endpc
name
instack
idx

`k` 指向一个这个 `Proto` 中使用到的常量， `code` 指向这个 `Proto` 的指令数组， `Proto **p` 指向这个 `Proto` 内部的 `Proto` 列表， `locvars` 存储local变量信息， `upvalues` 存储upvalue的信息， `cache` 指向最后创建的closure， `source` 指向这个 `Proto` 所属的文件名，后面的 `size*` 分别表示前面各个指针指向的数组的大小， `numparams` 表示固定的参数的个数， `is_vararg` 表示这个 `Proto` 是否是一个变参函数， `maxstacksize` 表示最大stack大小。

`FuncState` 中的 `ls` 指向 `LexState` ，在 `LexState` 中有一个 `Dyndata` 的结构，这个结构用于保存在parse一个chunk的时候所存储的 `gt label list` 和 `label list` 以及所有 `active` 变量列表，其中 `gt label list` 存储的是未匹配的 `goto` 语句和 `break` 语句的label信息，而 `label list` 存储的是已声明的label。待出现一个 `gt label` 的时候就在 `label list` 中查找是否有匹配的label，若出现一个label也将在 `gt label list` 中查找是否有匹配的 `gt` 。

`LuaY_parser` 调用 `mainfunc` 开始parse一个chunk：

```
static void mainfunc (LexState *ls, FuncState *fs) {
  BlockCnt bl;
  expdesc v;
  open_func(ls, fs, &bl);
  fs->f->is_vararg = 1;  /* main function is always vararg */
  init_exp(&v, VLOCAL, 0);  /* create and */
  newupvalue(fs, ls->envn, &v);  /* set environment upvalue */
  luaX_next(ls);  /* read first token */
  statlist(ls);  /* parse main body */
```

```
  check(ls, TK_EOS);
  close_func(ls);
}
```

在 `mainfunc` 中通过 `open_func` 函数完成对进入某个函数进行parse之前的初始化操作，每parse进一个block的时候，将建立一个 `BlockCnt` 的结构并与上一个 `BlockCnt` 连接起来，当parse完一个block的时候就回弹出最后一个 `BlockCnt` 结构。 `BlockCnt` 结构中的其它变量的意思是： `nactvar` 表示这个block之前的active var的个数， `upval` 表示这个block是否有upvalue被其它block访问， `isloop` 表示这个block是否是循环block。 `mainfunc` 中调用 `statlist` ， `statlist` 调用 `statement` 开始parse语句和表达式。

`statement` 分析语句采用的是LL(2)的递归下降语法分析法。在 `statement` 里面通过 `case` 语句处理各个带关键字的语句，在 `default` 语句中处理赋值和函数调用的分析。语句中的表达式通过 `expr` 函数处理，其处理的BNF如下：

```
exp ::=  nil | false | true | Number | String | '...' | functiondef |
         prefixexp | tableconstructor | exp binop exp | unop exp
```

`expr` 函数调用 `subexpr` 函数完成处理。

```
static BinOpr subexpr (LexState *ls, expdesc *v, int limit) {
  BinOpr op;
  UnOpr uop;
  enterlevel(ls);
```

```
  uop = getunopr(ls->t.token);
  if (uop != OPR_NOUNOPR) {
    int line = ls->linenumber;
    luaX_next(ls);
    subexpr(ls, v, UNARY_PRIORITY);
    luaK_prefix(ls->fs, uop, v, line);
  }
  else simpleexp(ls, v);
  /* expand while operators have priorities higher than `limit' */
  op = getbinopr(ls->t.token);
  while (op != OPR_NOBINOPR && priority[op].left > limit) {
    expdesc v2;
    BinOpr nextop;
    int line = ls->linenumber;
    luaX_next(ls);
    luaK_infix(ls->fs, op, v);
    /* read sub-expression with higher priority */
    nextop = subexpr(ls, &v2, priority[op].right);
    luaK_posfix(ls->fs, op, v, &v2, line);
    op = nextop;
  }
  leavelevel(ls);
  return op;  /* return first untreated operator */
}
```

当分析 `exp binop exp | unop exp` 的时候lua采用的是算符优先分析，其各个运算符的优先级定义如下：

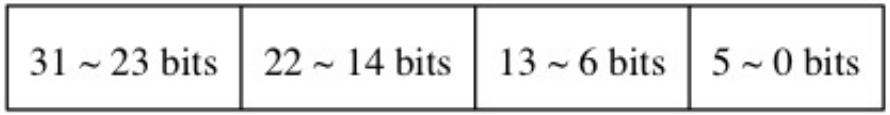
```
static const struct {
  lu_byte left;  /* left priority for each binary operator */
  lu_byte right; /* right priority */
} priority[] = {  /* ORDER OPR */
   {6, 6}, {6, 6}, {7, 7}, {7, 7}, {7, 7},  /* `+' `-' `*' `/' `%' */
   {10, 9}, {5, 4},                 /* ^, .. (right associative) */
   {3, 3}, {3, 3}, {3, 3},          /* ==, <, <= */
   {3, 3}, {3, 3}, {3, 3},          /* ~=, >, >= */
   {2, 2}, {1, 1}                   /* and, or */
};

#define UNARY_PRIORITY     8  /* priority for unary operators */
```

代码生成

lua代码生成是伴随着语法分析进行的，指令类型 `Instruction` 定义在llimits.h中：

```
/*
** type for virtual-machine instructions
** must be an unsigned with (at least) 4 bytes (see details in lopcodes.h)
*/
typedef lu_int32 Instruction;
```



`Ins`