目 录

章 节 目 录

图 目 录

示 例 代 码 目 录

摘 要

在很多情况下，由于受表达机制（语法构造，控制结构，组合手段等）的限制，我们无法用通用编程语言来构建一个清晰、自然和高效的解决方案，而这时针对特定问题的领域特定语言则是一个很好的选择。因此，如果开发人员能够方便地创造适合他们自己需求的领域特定语言将是极有意义的。

本文以动态语言 Lua 为例，设计并实现了一个支持静态元编程的可扩展式开放 Lua 编译器——OpenLua，它能够支持程序员在无需改动编译器的情况下扩展语言的语法设施，从而快速地构造出领域特定语言。

文章首先介绍了静态元编程和开放式编译器的基本概念，也简要介绍了标准 Lua 语言，同时还对 OpenLua 做了一个概览。

接着详细描述了 OpenLua 在语言层面对标准 Lua 所做的扩展，包括 OpenLua 为了支持静态元编程和自定义语法而为标准 Lua 语言引入的 3 个新的语法设施——用户自定义语法，源代码转换子，编译期模块导入。跟着介绍了 OpenLua 编译器为静态元程序提供的可编程接口和运行环境。

然后详细描述了 OpenLua 编译器的实现，包括作为 OpenLua 开放式架构基础的开放式词法分析算法、开放式 SLR 解析算法以及用户自定义语法的确定性递归下降解析算法等。接着解释了编译器如何处理与静态元编程相关的 3 个新的语法设施和转换子调用，即它们的语义。然后还深入探讨了编译器提供的可编程接口和元程序运行环境的实现。

随后作者通过给出 OpenLua 的若干有趣应用来展示开放式架构可以如何支持语言的扩展。也介绍了该领域的相关研究工作，并将 OpenLua 与它们做了比较，最后分析总结了 OpenLua 的不足，并提出了相应的改进计划。

关键词 元编程；开放式编译器；可扩展式语言；Lua

ABSTRACT

General purpose programming language sometimes can not be used to solve some problems naturally and effectively, because of the limitations of the language's expressive power(syntax constructs, control structures, combination means etc.). At that time, a domain-specific language will be a good choice.So it's significant that programmers can create a DSL conveniently for their own purposes.

I design and implement an extensible open compiler supporting static meta-programming named OpenLua in this paper.It allows programmer to create a DSL rapidly by intoducing new syntax constructs.

At first, there is a introduction to meta-programming, open compiler, and Lua language. An overview of OpenLua follows.

Then I describe the language extensions made to Lua by OpenLua, which include three new constructs: user-defined syntax, source transformer, and module importation, and how compiler deals with them ,namely their semantics. Then the concrete algorithms are illustrated, which includes open lexcial analysis, open SLR parsing, and predictive recursive-descent parsing, etc. . And also you can see the details of the exposed programmable interfaces of OpenLua and the execution environment of meta-program. After that ,it will be explained that how to perform a source transformation.

I use some interesting examples to demonstrate how OpenLua can be used to extend the programming language. And then readers can see some related work in this area. At last, I summarize some shorcommings of OpenLua and give a plan of improvements in the

future.

Keywords meta-programming; open compiler; extensible language; Lua

第 1 章 引言

1.1 基本概念介绍

1.1.1 静态元编程

元(meta)这个词是从希腊词汇中借来的，意为“after”或者“before”，用于表示级别的改变。而在计算机科学中，它主要表示“being about”(关于)的意思[CE2004]。因此元程序(meta-program)是关于程序本身的程序，正如元数学(meta-mathematics)是关于数学本身的数学一样。

元程序实际上可以在不同的语境(context)和不同的时间段运行，但如果元程序是在它们操纵的代码被载入(loaded)之前运行的话，那么就称其为静态元程序(static meta-program)，相应的设计活动称为静态元编程(static meta-programming)。包含静态元程序的系统最常见的例子就是编译器和预处理器。这些系统操纵表示输入的内部数据结构(比如抽象语法树)，并把它们转换为使用其它语言(例如汇编)或者同样语言但结构被修改了的程序[CE2004]。

在编译期运行的元程序只是静态元程序的一种，不过却是非常常见的一种。如果没有特别说明，本文剩下部分提到的元程序或静态元程序均特指编译期运行的静态元程序。如果要让元程序在编译期运行，那么编译器就有义务为它们提供一个完整的执行环境(execution environment)。

1.1.2 开放式编译器

开放式编译器(open compiler)的思路是向静态元程序提供某些定义良好的高层接口，使其能够改变默认的编译动作(比如按照新的语法来解析源代码)，获取并操纵编译过程中源代码的内部表示(比如语法解析树：syntax parse tree)。事实上每一个编译器或多或少都能称得上是开放式的，因为编译器后端(代码生成器)必然要调用前端提供的接口以取得代码的中间表示(通常是抽象语法树或 3 元组)，向元程序提供接口只不过是朝开放的道路上更前进了一步而已。

1.2 研究意义

从理论上来说，任何一种图灵完备(Turing-complete)语言的计算能力都是等价的，但实际上却没有一种语言(即使它号称通用语言)对解决所有问题都是适合的。因为语言内建的表达机制 (语法设施、数据类型，控制结构，组合手段)通常是固定的，而适合描述某类问题的语言，对其它类型问题可能就力不从心了。领域专用语言(DSL，Domain Specific Language)通过提供合适的机制可以优雅而高效地解决特定领域的问题，因此DSL越来越受到开发者的重视。然而从头构建一门新的针对特定领域的语言对许多开发者来说是一项非常困难的工作，因此程序员迫切需要一种允许使用者在基本框架的基础上不断发展出新机制的语言，即可扩展式编程语言（extensible programming language）。开放式架构的编译器提供了这种可能：用户可以在原始语言的基础上定义新的语法格式，可以把用新语法格式书写的源代码转换成语义相等的原始语言代码(这不是必需的，事实上你可以对它们做任何事)。这样程序员便可以在无需重写或扩展编译器的前提下，为一门语言引入适合描述他们所面对问题的新的语法设施、新的控制结构，新的关键字等等。这可不就是创造了一门DSL?

1.3 动态语言 Lua

1.3.1 简介

Lua([IFC2003],[Ier2004])是由巴西里约热内卢天主教大学(Pontifical Catholic University of Rio de Janeiro)研究人员创造的一门语法类Pascal而语义类Scheme[ASS2004]的过程型动态语言，最初主要用来扩展C程序，所以整个系统(包括解释器、编译器、虚拟机)全部采用ANSI C实现并以C程序库的方式提供，同时语言本身定义了完善的与C语言交互的接口。Lua的主要特点有：变量(variable)没有类型，只有值(value)才有类型，运行期可以赋给变量任何类型的值；函数是一阶值(first-class value)，并支持closure概念；采用词法定界(lexical scoping)；提供了协程(coroutine)机制；系统自带内存回收(garbage collection)功能；表(table)是唯一的数据结构，相当于关联数组(associative array)，任何类型的值都可以作为表的键来检索(index)数据；提供了较丰富的反射(reflection)功能；可以为

每一个表设定元表(metatable)和元方法(metamethod)，从而改变表操作的原始语义，此即所谓的动态元机制(dynamic meta-mechanism)。

选择 Lua 作为研究对象主要是因为它在许多领域尤其是电脑游戏开发中得到了越来越广泛的应用，是一门非常有活力的成长型语言，为它实现的开放式编译器将确实有可能帮助工业界的程序员提高开发效率。

1.3.2 基于原型的面向对象模型

由于 Lua 提供的独特的面向对象编程模型在本文的研究与实现中占有重要的位置，所以本节将对其做一个介绍。

Lua 的数据类型里并没有类(class)与对象的概念，但实际上却可以很容易用表来表示对象，程序 1—1 是一个简单示例。在本文的余下部分，表和对象这两个词将不加区分地混合使用。

```
Point2d = {x = 1} -- 创建一个对象(实际就是一个表)并赋给 Point2d，
                  -- 该对象有一个名为 x 值为 1 的属性
Point2d.y = 0      -- 为 Point2d 对象添加一个名为 y 值为 0 的属性
function Point2d:MoveX(dx)  -- 这是 Lua 为面向对象编程提供的语法糖衣，
                            -- 相当于写 function Point2d.MoveX(self,dx)
                            -- 隐藏参数 self 代表当前对象
  self.x = self.x + dx
end
print(Point2d.x,Point2d.y) -- 打印出 1 和 0
Point2d:MoveX(3) -- 面向对象调用，相当于写 Point2d.MoveX(Point2d,3)
print(Point2d.x,Point2d.y) -- 打印出 4 和 0
```

程序 1—1 用表来实现对象

与 C++、Java 这些基于类的面向对象语言不同，Lua 无法通过类的概念来描述继承关系，但通过语言的动态元机制则可以构建出一种基于原型(prototype-based)的继承模型。采用这种模型的典型语言有 JavaScript、SELF[US1987]。原型实际上就是普通对象，任何对象都可以成为其它对象(包括原型)的原型。如果一个对象以另一个对象为原型，

那么在它内部就会保存一个指向原型的引用(refrence)，当该对象碰到它不能理解的消息时便到原型中去寻找相应的处理方法，如果原型中没找到就再到原型的原型中找，直至找到或者不再有原型。由此可见，对象与原型之间是一种委托(delegate)与被委托的关系，它们之间的区别完全是由所担当角色的不同而产生的。通过指向同一个原型，许多对象可以共享该原型的行为与属性[Lie1986]，因此它们也就具有了某些共同性质，从而成为同一类对象。实例化某类对象很简单，仅仅是根据该类对象的原型克隆(clone)出一个新对象，并为新对象设置一个指向原型的引用。而为了方便，我们称该原型为对象的直接原型(direct prototype)。假设根据原型 B 实例化出对象 A，然后再为 A 添加新的或者重定义(override)B 原有的属性与方法，这样便可以利用 A 为原型实例化出与 B 相似但又有自己特色的对象。此时我们称原型 B 派生(derive)了原型 A，而 A 继承(inherit)了 B，B 是 A 的父原型(parent prototype)，A 是 B 的子原型(child prototype)。假如 B 又继承了 C，C 又继承了 D……，那么它们便构成了一条继承链(inheritance chain)，并且 B、C、D……位于 A 的上方，而所有在继承链中位于 A 上方的原型被统称为 A 的祖先原型(ancestor prototype，包括父原型)。

Lua 允许一个对象用另一个普通的表作为自己的元表(metatable)，元表中有若干特殊字段代表着元方法(metamethod)，__index 就是其中之一。一个元表的__index 字段的值应该要么是一个表，要么是一个型构(signature)为 function (table,key)的函数。如果以不存在的键检索某个对象时，Lua 或者会把在元表的__index 中检索到的值返回，或者会以对象和键作为参数执行__index 代表的函数，并把函数返回值作为检索值返回。假如元表自身还有元表，并且也有相应的__index 元方法，那么将重复上述过程。程序 1—2 演示了如何利用元表与元方法来构建基于原型的单继承对象模型。代码执行完后对象的布局如图 1—1 所示。New 是 Point2d 的成员函数，它返回的对象都以 Point2d 为元表。Point3d 正是由 Point2d:New 实例化的对象，由布局图可以看到 Point3d 本身并不真正包含 x 和 New 字段，但是当我们试图在 Point3d 中检索它们的值时，Lua 便会自动到元表的__index(也即 Point2d)中去找，这就使得 Point2d 成为了 Point3d 的原型。Point3d.y 是重定义的属性，Point3d.z 是新添加的属性，因为都位于 Point3d 中，所以检索它们无需到原型中寻找。同样地，Point3d 调用继承来的 New 函数实例化的对象 v 以 Point3d 为原型，它的 y、z 属性在 Point3d 中检索到，x 属性则最终在 Point2d 中找到。

```
function Point2d:New(o) -- 实例化函数
   o = o or {}   -- 如果 o 为 nil 则创建一个空表
   rawset(self,"__index",self) -- 设置当前对象的__index 字段值为对象本身
   setmetatable(o,self) -- 设置对象 o 的元表为当前对象，使得 o 以 self 为原型
   return o
end
Point3d = Point2d:New() -- 从 Point2d 派生出一个新的原型
Point3d.y = 2           -- 将 y 属性定义为新的值
Point3d.z = 3           -- 为 Point3d 添加一种新的属性
v = Point3d:New()       -- 实例化一个以 Point3d 为原型的对象
print(v.x,v.y,v.z)      -- 打印出 1、2 和 3
```

程序 1—2 实例化(instantiation)与派生(derivation)

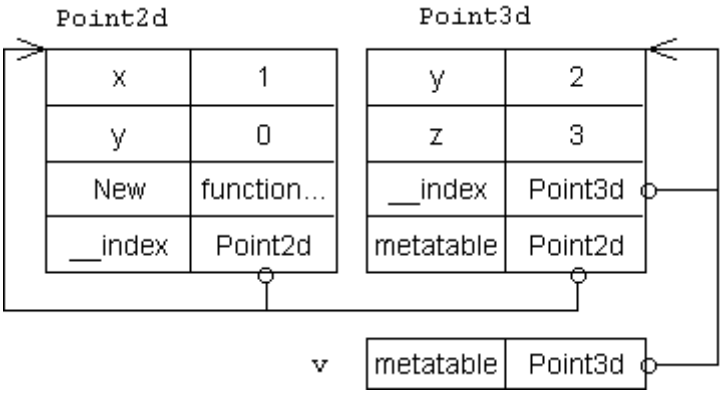

图 1—1 利用元表构建的单继承对象模型

1.3.3 标准 Lua

在本文中，作者将 Lua5.0 参考手册[IFC2003]定义的语言称为标准 Lua，而由 Lua 官方网站 http://wwww.lua.org 发行的完全遵循了[IFC2003]定义的 Lua 实现称为标准 Lua 实现或标准 Lua 系统。

1.4 本文的主要贡献

本文在可扩展式编程语言、开放式开发系统、元编程技术的研究上主要做出了以下几方面的贡献：

<1> 对标准 Lua 编程语言进行了扩展，引入了新的支持静态元编程的语法设施，为 Lua 同时作为运行期和编译期两个层次的编程语言奠定了基础；

<2> 将自定义语法的概念引入了 Lua 语言，并且设计了自定义语法的解析接口、语法树的操作及转换接口以及转换后生成的程序文本如何与原有的代码集成的机制，从而为用户自行扩展 Lua 构建了一个良好的框架；

<3> 针对扩展后的 Lua 语言开发了一个开放式编译器——OpenLua，该编译器除了为静态元程序提供一个完善的运行环境外，还实现了自定义语法解析接口、语法树操作及转换接口，它的完成为 Lua 语言使用者学习及实践可扩展式编程语言提供了一个良好的平台；

<4> 利用编译器的开放式能力为 Lua 语言实现了若干有趣的扩展，从而在实际应用中检验了此类开放式系统的有效性及易用性。

1.5 本文的组织结构

本文的具体组织结构安排如下：

- ✧ 第 2 章概括地介绍了 OpenLua 编译系统及其使用格式，并且描述了从用户角度看到的编译过程。
- ✧ 第 3 章详细地介绍了 OpenLua 的设计与实现，包括 OpenLua 为标准 Lua 语言引入的新的语法设施的格式及语义、OpenLua 提供的元编程环境及开放的编程接口的定义及实现，还包括为开放式架构定制的特别的词法分析器的实现和 OpenLua 内建语法及用户自定义语法的解析算法。
- ✧ 在第 4 章中，作者通过若干具体的应用实例展示了 OpenLua 的扩展能力，并检验了该系统的有效性与易用性。
- ✧ 第 5 章分析了在可扩展编程语言、开放式开发系统、元编程技术以及相关的领域内国际国内其他研究人员所做的主要工作、贡献，并与本文的工作进行了比较。
- ✧ 第 6 章对本文研究工作进行总结，着重分析了 OpenLua 系统在使用及实现上的缺陷，并提出了相应的改进计划。

第 2 章 OpenLua 概述

OpenLua 这个名字代表两个意思：一是指为支持静态元编程而对标准 Lua 进行扩展得到的语言；二是指一款针对上述语言的，并且开放了若干内部可编程接口的编译器。读者一般根据上下文就能判断当前 OpenLua 这个词确指哪个概念，只有在可能引起混淆的情况下作者才会做额外说明。

OpenLua 提供给用户的静态元编程语言就是标准 Lua 本身，因此程序员无需学习另外一门语言即可轻松地利用开放式编译器的扩展能力。而有的系统提供的静态元编程语言与普通源代码使用的语言间的编程模型相差太大，最突出的例子便是 C++的 template。作为元编程语言的 template 本质上提供的是一种函数式编程范式(functional programming paradigm)：没有变量、不允许赋值(也就没有 side-effects)、用递归来实现各种控制结构，这对许多 C++程序员来说都是非常陌生的。

该系统主要用标准 Lua 语言并辅以少量的 C 来实现，所有功能(包括词法分析，语法分析)均为从头独立构建，并未采用任何外部辅助工具(比如 YACC 等)。做出这样的设计决定有这么几个原因：1、Lua 官方网站 http://www.lua.org 已经为我们提供了一个高效率的标准实现，这使得用 Lua 来编写 Lua 语言编译器(一个有趣的元环：meta-circle)成为可能；2、标准 Lua 本身非常简洁、易于使用并且功能足够强大，尤其是提供了比较完善的文本处理函数，这一切都有助于开发效率的提高；3、Lua 是作为一门嵌入式语言(embeded language)来设计的，并且以 C 程序库的形式提供完整系统，这使得我们很容易将标准 Lua 语言的解释器集成到 OpenLua 编译器中，从而为静态元程序提供一个完善的运行时环境；4、OpenLua 是一个开放式架构的编译器，它的很多编译行为(包括词法分析、语法分析)与非开放式系统有很大不同，而 YACC 等自动生成工具只能生成普通的封闭式系统，因此类似工具并不适合在本项目中使用。

因为 Lua 虚拟机及其指令集并非语言定义的一部分，它们有可能在未通知语言使用者的情况下变更，而且底层的虚拟机平台与 OpenLua 提供的静态元编程能力、开放式架构并无关联，所以 OpenLua 编译器没有做生成字节码(bytecode)的工作。

具体一点来说，OpenLua 的编译过程就是：首先解析源文件中的 OpenLua 源代码，并同时执行相应的静态元程序(如果有的话)以对源代码进行各种转换与处理，如果不出错(意味着语法正确、静态元程序运行无误)那么解析完毕后即生成一颗标准 Lua 语言的语法

解析树(syntax parse tree)，最后如果需要(见下文关于 OpenLua 使用格式的介绍)便将这颗语法解析树反解析(unparse)回标准 Lua 程序文本。图 2—1 形象地刻画了整个过程。反解析得到的标准 Lua 程序代码一定是能被标准 Lua 编译器编译通过的语法正确的代码，因为语法树的生成过程即保证了这一点。

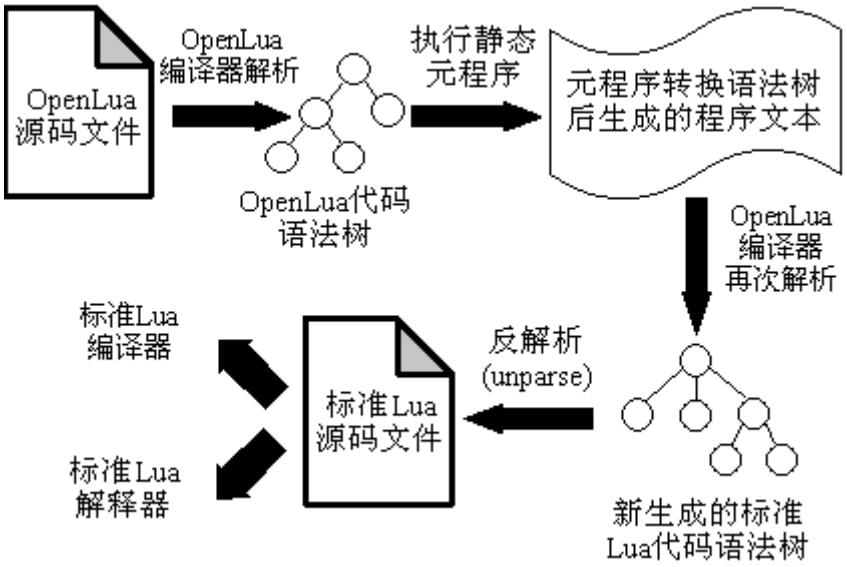

图 2—1 OpenLua 的编译过程

OpenLua 的项目主页是 http://www.luachina.org/projects/openlua/ ，2005 年 12 月 28 日 0.01 版发布，读者可访问以上地址下载软件包。

OpenLua 的使用格式是：

```
openlua sourcefile [outputfile]
```

sourcefile 是待编译的 openlua 源文件名，outputfile 是可选的输出文件名。上述命令会首先编译 sourcefile，如果出错便打印错误信息并退出；如果成功且指定了 outputfile 参数，openlua 就会把编译 sourcefile 得来的语法树反解析回标准 Lua 程序并输出到 outputfile 中，最后显示编译成功消息并退出。

第 3 章 OpenLua 的设计与实现

3.1 OpenLua 对标准 Lua 语言的扩展

3.1.1 语法的形式化定义及其书写规则

标准 Lua 语言语法的形式化定义采用的是 EBNF 格式，大概有 70 个左右的产生式[IFC2003]。基于实现上的考量，作者将它们改写成了标准 BNF 形式。

一个产生式形如：

```
field : '[' exp ']' '=' exp
```

冒号左边是非终结符(nonterminal)，也即产生式左部(left side)，而冒号右边则是产生式右部(right side)。产生式中出现的所有语法符号有两种书写形式：(1)没有被单引号引用的字符串，代表该语法符号的名字，它必须以下划线(_)或字母开头，其后紧随 0 或多个字母、数字或下划线，这与标准 Lua 中对变量名的词法规定是一致的，非终结符和关键字(keyword)一定要写成这种形式。(2)用单引号引用住的字符串，这种形式用于表示标点(punctuation)、操作符(operator)等语法符号，注意单引号本身不是符号的一部分。

具有相同左部的产生式可以把左部合并在一起，右部之间用|号分隔，不同的右部可以写在一行内，也可分几行写，但是同一个右部只能写在同一行内。冒号或者|号必须与它们引导的右部在同一行，并且右部不允许为空。比如

```
fieldsep : ',' | ';'
```

或者

```
fieldsep : ','
         | ';'
```

都是正确的格式，但是

```
fieldsep : ',' |
           ';'
```

却是错误的格式，因为|号引导的右部';'与它不在同一行内。

如果左部是新的非终结符，那么产生式一定要另启新的一行写。empty 和 eof 是两个

特殊的终结符，代表空符号和输入结尾符，用户不应该用它们做任何非终结符的名字。

OpenLua 在标准 Lua 的基础上引入了 3 个新的语法成分：用户自定义语法(user-defined syntax)、源代码转换子(source code transformer)以及编译期模块导入(import module)语句。附录 A 是 OpenLua 语法完整的形式化定义。

3.1.2 用户自定义语法

引入用户自定义语法的产生式如下：

```
syntaxdef : syntax Name ':' Literal
```

syntax 关键字后是自定义语法的名字，然后一个冒号，最后是包含了完整描述自定义语法产生式的字符串。程序 3—1 是个例子：

```
syntax contractfuncSyntax :
[[
cf : Name '(' optional_parlist ')' pre block post end
pre : require exp
    | empty
post : ensure exp
     | empty
]]
```

程序 3—1 一个用户自定义语法的例子

在 OpenLua(以及标准 Lua)中，被 [[和]] 包住的字符串是 raw Literal，即意味着它不会因换行而中止，也不解释任何转义字符，这使得书写包含程序文本之类内容的字符串非常方便。自定义语法的产生式书写规则与上一小节介绍的完全一致，用户可以在自定义语法中引入新的关键字(如本例中的 require 和 ensure)，也可以引入新的长度为 1 的标点或操作符(本例并无体现)。很多时候定义语法成分是一件很繁琐的事情，如果每次都要求用户从头开始，可想而知他们会多么沮丧。因此系统允许自定义语法引用 OpenLua 内建的所有语法符号(包括终结符与非终结符)，如本例的 Name 终结符和 end 关键字，以及 optional_parlist 和 block 非终结符，但不可重定义它们。用户无需关心内建非终结符如何解析，他只需明白解析器(parser)会在合适的时候根据内建非终结符的语法规则进

行解析并生成一个正确的语法变量，这就使得内建非终结符在自定义语法里表现得就象一个终结符一样。对自定义语法的最后一点要求是：它必须是个 LL(1)语法，否则编译器将不会接收它。

3.1.3 源代码转换子

源代码转换子定义的产生式为 ：

```
transformerdef : transformer Name block end
```

其中 transformer 是关键字，Name 是转换子的名字，block 是一个标准 Lua 代码块，在这里作为转换子的转换体用于执行具体的转换操作，它的返回值要么是一个字符串要么是 nil。一个名字被定义为 transformer 后，它在源代码中就具有特殊的意义了：编译器遇到一个已被定义为 transofrmer 的名字后，会把它从源代码输入流(input stream)中取出并抛弃，然后立刻执行它的转换体(即 block 代表的代码块)，如果第一个返回值是 nil，编译器报错并退出，如果第一个返回值是字符串，编译器就会把这一段字符串当作转换后得到的程序文本插入到当前源码输入流的头部，这整个过程称作转换子的调用。在 OpenLua 中，静态元编程能力正是由转换子的定义与调用提供的。

假设有这么一段程序：

```
transformer test
   return "print('test')"
end
a = 2006
test
```

那么它被编译后得到的输出便是：

```
a = 2006
print('test')
```

由此可见转换子调用的结果就好象程序员自己在调用处写下了一些不同的代码(这些代码正是转换体的返回值)一样。当然，本例定义的转换子并没有任何实际意义，但是它却演示了转换子的一个基本作用：代码替换。在介绍完 OpenLua 提供的编程接口后，读者会看到转换子的另一个基本作用：代码转换。

3.1.4 编译期模块导入

编译期模块导入语句的产生式为：

```
import_module : import Literal
```

import 关键字后是你要导入的模块文件名。当编译器遇到 import 语句时，它会立刻去找相应的文件，如果找到了即会编译它(是的，完全编译)，但编译产生的那些运行期代码被简单地抛弃，而只有文件中的语法定义、转换子定义得以保留。毫无疑问，编译期模块导入机制的最大作用就是为了复用(reuse)那些自定义语法和转换子。

3.2 OpenLua 提供的可编程接口及元程序运行环境

OpenLua 通过 transformer 提供了静态元编程能力，但是如果编译器没有开放出一些必需的内部接口，并为元程序运行提供一个完善的运行环境，那么元程序能够做的事情将会很有限。因此作为开放式架构编译器的 OpenLua 提供了一系列的关键接口。

让我们先来看一个例子：

```
syntax metablockSyntax :
[[
metablock : block endmeta
]]
transformer meta
   -- lock 用于禁止这段代码重入
   if not lock() then
      return nil,"meta transformers can't internest"
   end
   -- parse 是编译器开放的接口
   local tree,error = parse(metablockSyntax)
   if nil == tree then
      return nil,error
   end
   local f = loadstring(tree:get_child(1):emit())
```

```
   -- _METAENV 是用于保存编译期运行变量的全局环境
   setfenv(f,_METAENV)
   local succeed,msg = pcall(f)
   if not succeed then
      return nil,"metaprogram run error : "..msg
   end
   unlock()
   return ""
end
```

程序 3—2 meta 转换子的定义

请首先注意 meta 转换子定义用到的 parse 函数，这是编译器开放给用户的最重要的接口。它以某个自定义语法为参数，并有两个返回值：第一个代表生成的语法树，第二个代表可能的出错信息。该函数被调用时会按照指定语法对当前源码输入流进行解析，如果成功则返回生成的语法树和 nil，如果失败就返回 nil 和出错信息。

一颗语法树可以通过保存其根节点来保存，而使用语法树亦是通过引用其根节点来进行的。一颗语法树节点(syntax tree node)是一个 Symbol 类型对象，也即所谓的语法变量(syntactical variable)，在接下来的论述中作者对这三个概念将不加区分，混合使用。Symbol 类型对象提供了 type、name、value、has_children、children_count、get_child、emit 等一系列接口来取得相关信息。假设 s 是一个语法变量，那么 s:type()用来取得语法变量的类型名，对于非终结符来说，就是它们的名字，对于终结符来说，分 keymark(包括关键字、操作符、标点)、Name、Literal 和 Number 4 种。s:value()用于取得语法变量的值，keymark、Name 和 Literal 类的值就是表示相应终结符的字符串，而对于 Number 来说则是它代表的那个数的值，这个函数对非终结符没有意义。对于所有非 keymark 类语法变量来说，s:name()得到的就是 s:type()，而对于 keymark 来说，得到的却是 s:value()。s:has_children()返回一个标明该语法树节点是否拥有子节点的 bool 值，所有终结符变量都不可能有子节点，只有非终结符变量才可能有。children_count 和 get_child 分别用于计算语法树节点的子节点个数和取得某个指定的子节点。比如 OpenLua 语法中定义 syntaxdef 的产生式是这样的：

```
syntaxdef : syntax Name ':' Literal
```

那么当 s 为 syntaxdef 类型的语法变量时，s:children_count()就得到 4(因为产生式的右部有 4 个符号)，而 s:get_child(1)将得到一个类型为 keymark 值为 syntax 的语法变量(因为 syntax 关键字是右部的第一个符号)，同理 s:get_child(4)将得到一个类型为 Literal 的语法变量。OpenLua 只提供了按照它们在产生式右部所处的位置来取得子节点的方式，这是因为右部完全可能同时出现好几个名字相同的符号，而只有它们在右部所占的位置才是唯一的。如果 get_child 的位置参数超出了相应的范围，那么返回值将是 nil。s:emit()会将 s 所代表的语法树反解析回程序文本，并以字符串的形式返回。

每一段 Lua 代码都有一个与之相关的环境(environment)，供其在执行之时寻找代码中遇到的所有全局变量，这与 C++中的命名空间(namespace)非常类似。而且，一个环境本质上就是一个保存了所有全局变量 name-value 对的普通 table，因此我们可以象操纵普通 table 那样操纵环境。尽管在编译期运行，转换子的转换体仍然是一段不折不扣的标准 Lua 代码，它同样也该有自己的环境。OpenLua 为所有的转换体设置了一个统一的环境，我们称之为静态元环境(static meta-environment)或元环境(meta-environment)。已定义的语法名保存在这里，parse 等编译器接口也在这里，更重要的是，标准 Lua 提供的所有基本函数与库函数[IFC2003]都可以在这个环境中获得。最后一点非常重要，因为它使得静态元程序拥有了与运行期程序同样丰富的编程接口，而这当然大大方便了元编程工作。因为用户自定义语法是保存在元环境中，所以转换体代码可以通过它们的名字来引用这些语法，并且语法名可与运行期代码中的变量名相同，因为它们的生命期(life time)根本不会重叠。在元环境中有一个特殊的变量叫_METAENV，它的值就是元环境本身，很显然，_METAENV._METAENV(别忘了环境就是一个 table)等于_METAENV。需要特别指出，已定义的转换子并非保存在元环境中，而是另一个特殊的编译期环境，因此转换子和自定义语法、编译期函数都可以同名，并不会产生冲突。

在程序 3—2 里，parse 按照 metablockSyntax 解析成功后得到的语法树被保存在局部变量 tree 中，调用 tree:get_child(1)得到这颗树的第一颗子树(即 metablock : block endmeta 中的 block)。然后利用 emit 将 block 反解析回程序文本，并调用标准 Lua 基本函数 loadstring 把这段程序文本编译并生成一个新的函数，接着利用 setfenv(同样来自标准 Lua)将这个新生成的函数的环境设置为_METAENV(即统一的静态元环境)。最后在保护模式下执行它，如果执行成功则返回空字符串，否则返回 nil(这会导致编译器报错并退出，参见 3.1.3 对转换子的描述)和出错信息。

lock 和 unlock 是编译器提供的另外两个接口。 其中 lock 函数的功能是为 lock

的调用者“加一个锁”，如果之前该调用者还没被加锁，那么将其锁上并返回 true，如果已经加了锁，则什么也不做并返回 false。unlock 执行解锁功能，如果之前调用者确实被锁上，那么 unlock 将其解锁并返回 true，如果原本就没被锁上，那么什么也不做并返回 false。lock 和 unlock 的配对使用，可以很容易控制函数是否可重入，这正是程序 3—2 的做法(是的，转换子的转换体代码会被编译器悄悄地改造成一个函数)。

这个meta转换子起什么作用呢？其实它就是让用户无需定义transformer便能方便地运行元程序。假设程序 3—2 保存在一个名为 std.ol(ol 是推荐使用的 OpenLua 源代码文件后缀名)的文件中，并且另有一个名为 temp.ol 的文件：

```
-- temp.ol
import "std.ol" -- 象这样没指定路径的话，就要求 std.ol 在当前目录下
meta
   print("Hello world from metaprogram !")
endmeta
```

运行 openlua temp.ol 命令，那么你会看到这么两行输出信息 ：

```
Hello world from metaprogram !
Compile temp.ol file successfully !
```

这清楚地表明了介于 meta 和 endmeta 之间的标准 Lua 代码是不折不扣的元程序，在编译器解析完它们之后就会立刻运行，并且不会对编译器反解析出的程序文本产生任何影响(因为在 meta 的定义中，成功执行这些代码后返回的是一个空字符串)。不过特别要注意的是，meta、endmeta 之间不能再嵌套 meta、endmeta 了(这是通过 lock 与 unlock 实现的)，为什么？被第一层的 meta 和 endmeta 包含住的代码是有别于普通代码的元代码，假设里头又嵌套了一层，那么被这一层包含住的代码岂非成了元元代码(meta meta-program)？而它又该在什么时候执行呢？元代码的编译期？假设再多一层呢？由此可见，meta 嵌套带来的概念上的混乱远远大于其微乎其微的好处，所以作者坚定地禁止了这种写法。

meta 转换子的定义展示了 transformer 的另一个重要作用：代码转换。这里是将 meta、endmeta 内部的程序文本转换成了空字符串，当然(而且是通常)我们也可通过转换子将某种结构的源代码转换成另一种结构的源代码并插回到转换子的调用点。在第 4 章读者将会看到有关的应用示例。meta 是一个非常重要的转换子，下文还将多次使用。因此，

为了方便起见，作者始终假定它已保存在当前目录下的 std.ol 文件中，从而可以通过 import "std.ol" 语句来导入。

3.3 开放式词法分析器

3.3.1 词法分析器、输入流和数据源 3 层结构

OpenLua 要解析的代码不仅仅来自源程序文件，还来自编译过程中执行转换子调用后其返回的字符串，因此，词法分析程序必须要有能力应对这许多不同来源的输入流（input stream）。在 OpenLua 的实现中，词法分析器(lexer)是一个可以不断产生 token 的抽象机器，每一个词法输入器在创建初始到被销毁自始至终都只与一个输入流相关联（associate）。输入流也是一个抽象的机器，它的功能就是将它所关联的数据源中的数据传递出来。一个输入流在创建时必须设置某个数据源和它相关联，这个数据源要么是一个文件，要么就是一个字符串。图 3—1 是数据源、输入流及词法分析器相互关系的一个形象说明。从中可以看到，通过这种分层的结构我们得到了极大的灵活性：数据源到底是文件还是字符串对词法分析器来说并不重要，真正重要的是词法分析器知道它可以通过某个接口得到它所需要的数据，因此数据源的细节被输入流屏蔽在了底层而不与(也无需与)词法分析器发生任何直接关系。

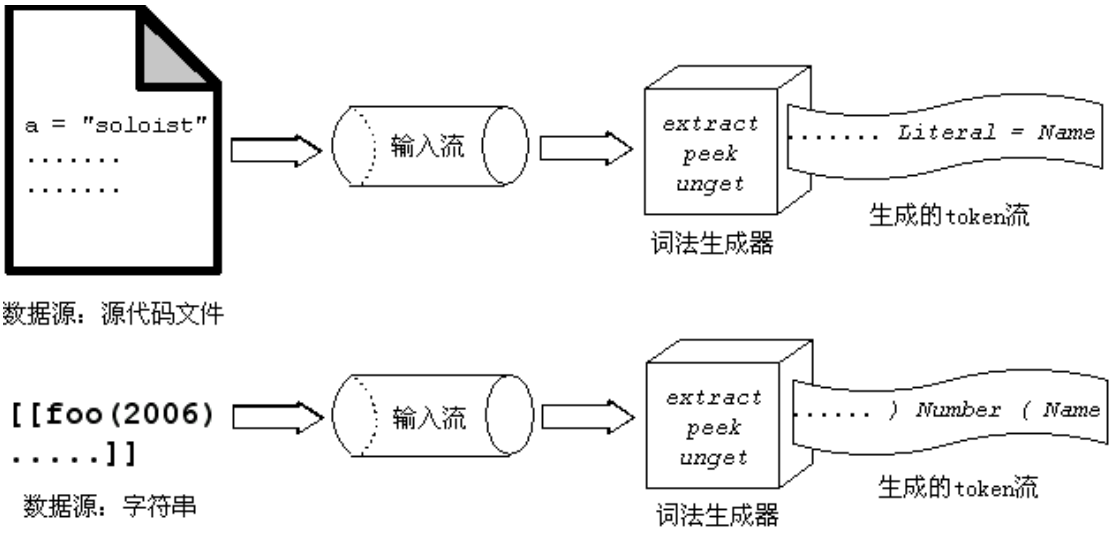

图 3—1 词法分析器与输入流示意图

3.3.2 词法分析器的 extract、peek 和 unget 接口

词法分析器最重要的接口有 3 个，分别是 extract（取出）、peek（窥探）和 unget（放回）。其中 extract 接口用于从词法分析器那里取得下一个 token，并且把相应的 lexeme 从输入流中拿走。unget 函数与 extract 恰好相反，它接收某个指定的 token 为参数，将其“放回”输入流的头部。被放回的 token 在输入流中的位置关系与它们被 unget 的顺序相反，即后 unget 的 token 在输入流中位于先 unget 的 token 之前，这与栈的后进先出性质非常类似。所以在词法分析器的实现代码中，程序并没有真的将数据插回到输入流的头部，而是另外开辟了一个栈空间来保存这些放回的 token，这个栈就被称为 unget 栈，这也是为什么要将上文的“放回”打上引号的原因。unget 操作在解析发生错误需要回溯(backtracking)时非常有用，它可以把那些不属于指定语法树的 token 统统放回去。有了 unget 栈以后，每一次调用 extract 就需要先查看 unget 栈顶是否有 token，如果有便弹出并返回该 token，如果没有那才到输入流中去取 lexeme 来生成 token。peek 接口与 extract 的行为非常类似，也是从词法分析器中读下一个 token，但是它与 extract 最重要的区别是，它既不会把 unget 栈顶的 token(如果有的话)弹出也不会从输入流中拿走任何一个 lexeme。也就是说如果两次 peek 操作之间没有 extract 和 unget 操作的话，那么它们得到的 token 一定是相同的。peek 这种能够窥探下一个 token 而又不改变输入流的性质可以避免许多无谓的 unget 操作，这对效率的提升极有好处。

3.3.3 用户自定义语法中的终结符问题

在 3.1.2 节已经提到过，用户在自定义语法中除了可以引用内建语法符号外，还可以引入新的关键字和长度为 1 的新标点或操作符，这就要求在按照用户语法进行解析时，词法分析器也能够将用户新引入的符号正确分析出来。比如程序 3—1 定义的语法引入了 require 和 ensure 这两个新关键字，而按照内建语法的定义它们却不是关键字，所以词法分析器如果仅仅根据 OpenLua 内建语法的终结符集合去分析的话，它们只能成为类型是 Name 的终结符。如此一来解析自定义语法时明明期待一个 require 关键字，得到的却是一个值为"require"的 Name，这必会导致解析失败，而用户却不明就里。因此要想办法让词法分析器能根据终结符集合来调整它的分析动作，为了达到此目的，词法分析器的 extract、peek 接口就需要做些变更。我们的方案是给 extract 和 peek 接口都添加一个代表终结符集合的参数，每次调用词法分析器的这两个接口时需同时提供一个终结符集

合，这样 extract 和 peek 便会根据传入的实际参数来调整它们的分析动作以返回正确类型的 token。

3.3.4 基于栈结构的开放式词法分析器

根据 3.1.3 节对转换子的描述可知，转换子调用返回的字符串相当于是被插入到输入流的头部的，但是我们是否必须得真的将数据插入到输入流中？输入流其实就是内存中的一段缓冲区，其大小一般是固定的，如果将数据插入到输入流的某个位置，那么这个位置后面的所有数据都要往后移，这就需要执行内存拷贝，更糟糕的情况是缓冲区大小不够了，那就需要重新分配一块大小合适的内存，然后再把所有数据拷贝过去。很明显，往输入流里插数据的操作是相当繁琐与低效的，我们应当坚决避免。那么用 unget 函数呢？根据上节的描述，unget 函数确实能把数据“放回”到输入流的头部，似乎应该是一个不错的选择。但是不要忘了，unget 操作是通过栈来实现的，而每一次 unget 只能将一个 token 放回，假如转换子调用返回的字符串是相当长的一段程序文本，那么采用 unget 方案将导致大量的压栈操作，这是非常低效的。而且由于栈是一种动态结构，极可能也会导致上文所述的内存重新分配、内存拷贝的糟糕情况。因此 unget 也不能承担起这个重任。那么转换子调用返回的字符串到底应该放到哪里去呢？

其实，既然当转换子调用返回时解析器需要的只是首先从词法分析器那得到由返回字符串生成的 token 流，那么我们何不再创建一个词法分析器，并把与之相关联的输入流的数据源设为对应字符串？有了多个词法分析器后，需要把它们很好地组织起来，否则混乱情况可想而知。因此为了方便使用，OpenLua 创建了一个抽象数据结构来统一管理多个词法分析器使它们能够协同工作，这个抽象数据结构就称之为开放式词法分析器(Openlexer)。在这里，开放指的是在编译过程中可以使用根据新的输入流创建的新词法分析器，而不仅仅局限于只读取源代码文件的那个。很明显，开放式词法分析能力是开放式编译器的基础之一。

由于转换子调用返回的字符串相当于插入到输入流的头部，所以很自然地便想到将所有的词法分析器组织成栈结构应该是一个合理的方案。这样一个词法分析器栈由 Openlexer 在内部维护，并且提供 push、pop 两个接口操作它。除此以外，它还通过 extract、peek 和 unget 这 3 个同名接口封装了根据词法分析器栈读取、窥探和放回 token 的操作。由于利用词法分析器扫描 token 时需同时提供一个终结符集合参数，因此 Openlexer 在内部还同时维护了一个终结符集合变量，并且提供 get_terminals 和 set_terminals 来

读取和设置这个内部变量。当需要给出终结符集合参数时，这个内部变量就会被自动传递。

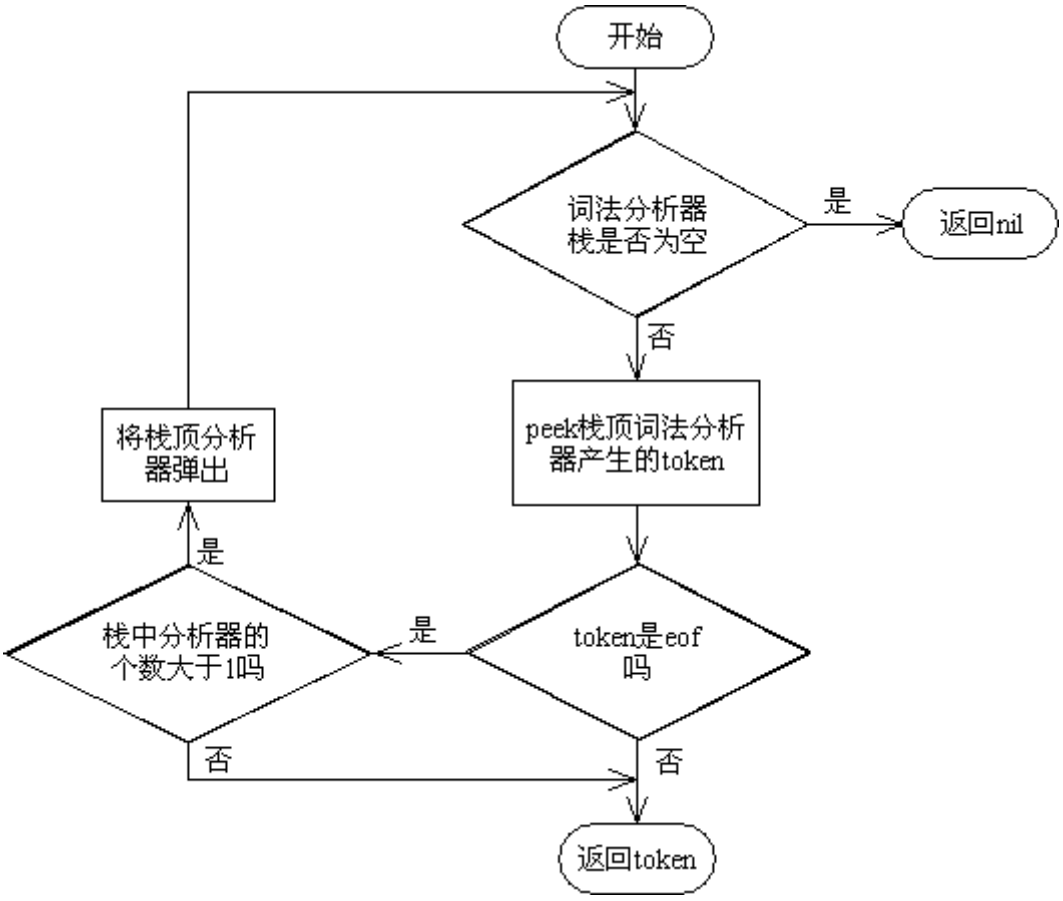

图 3—2 开放式词法分析器 peek 动作流程图

图 3—2 详细描述了开放式词法分析器 peek 动作的流程图。从中可以看出，从 Openlexer 得到的 token 都是调用位于栈顶的那个词法分析器的 peek 产生的，如果它已扫描到关联输入流的末尾，那么将被弹出栈外，相应的工作则由下一个栈顶分析器继续。extract 执行的流程与 peek 并无本质区别，不同的只是它的 token 由调用栈顶词法分析器的 extract 接口产生。Openlexer 的 unget 接口相对要简单许多，它仅仅是调用了栈顶元素的 unget 接口来完成工作。

在开始编译某个源文件时，OpenLua 会先创建一个属于这次编译过程的开放式词法分析器（此时内部的词法分析器栈为空），然后创建一个与该源文件(间接)相关联的一个词法分析器，并把它压入对应的 Openlexer（的内部词法分析器栈）中。在解析过程中每执行完一次转换子调用，便用其返回的字符串创建一个新的输入流，紧接着用这个输入流创建一个新的词法分析器，然后把它压入 Openlexer。解析器只与开放式词法分析器打交道，通过 Openlexer 的 peek、extract 接口来取得 token，而且它不需要显式地调用

Openlexer 的 pop 接口来弹出栈顶元素，因为 Openlexer 自己会自动地把已经不再有效(有效指还未扫描到输入流的末尾)的分析器弹出。需要注意的是，虽然每个词法分析器都能产生 eof，但是只有栈底的词法分析器产生的 eof 才会被返回。这是很自然的选择，因为在一次编译过程中，只应该遇到一个 eof，它标志着所有输入的结束。

3.4 OpenLua 内建语法的解析

3.4.1 消除语法的歧义

附录 A 定义的文法是一个准 SLR 文法。由它构造的 SLR 解析表(parsing table)绝大部分条目(entry)都不会有冲突(conflict)，但是却仍然有 36 个条目会包含 2 个动作(action)，其中既有规约(reduce)－规约冲突，又有规约－移入(shift)冲突。大多数冲突是由于在文法中并没有体现出运算符优先级(operator precedence)以及结合律(associative law)而导致的，这种冲突可以很容易根据语言在这方面的定义（参见附录 B）来解决。另外还有几种是由于 Lua 语句中的分隔符(分号)只是可选而非必需造成的。比如 foo(goo) (hoo)(1979)这条语句就有歧义，编译器既可以把 foo(goo)规约成一条语句(即 stat 语法变量)，这样它和后面的(hoo)(1979)一共是两条语句；又可以规约成前缀表达式(即 prefixexp 语法变量)，这样 foo(goo)(hoo)(1979)整个成为一条函数调用语句。类似的规约－规约冲突还会在另外三种情况下出现。请看：

```
-- prefixexp : functioncall 与 exp : functioncall 冲突。
-- 不知道该把 foo(goo)规约成 exp 还是 prefixexp
v = foo(goo) (hoo)(1979)
-- exp : var 与 prefixexp : var 冲突
-- 不知道变量 m 该被规约成 exp 还是 prefixexp
v = m (goo)(1979)
-- prefixexp : '(' exp ')' 与 exp -> '(' exp ')' 冲突
-- 不知道该把(foo)规约成 exp 还是 prefixexp
v = (foo) (goo)(1979)
```

程序 3—3 几种具有歧义的语句

所有的冲突都与 prefixexp 有关，而且前视(lookhead)token 都是左括号，此时编译

器在是否又该开启一个函数调用(functioncall 语法变量)上难以抉择。事实上，如果程序员肯在引发冲突的那部分代码后头显式地加一个分号的话，那么歧义将不复存在，但是标准 Lua 没有强加此要求，我们也不应该。因此最后选定的方案是：如果作为 lookhead 的左括号与引起规约冲突的那部分在同一行，那么我们优先选择将其(引发冲突的那部分)规约成 prefixexp；否则，优先选择另外一个产生式进行规约。这样的想法是很自然的。因为在编程实践中，如果程序员确实想进行函数调用，那么参数多半是紧跟函数并与之在同一行内的，而如果他们不是想调用函数的话，绝大多数程序员都会另启新的一行。

解决了歧义性后，就可以采用 SLR 解析法来解析 OpenLua 程序。在编译过程中，语法树既可以显式创建，又可以不这么做。通常基于规则的(rule based)翻译系统用不着真的创建一颗语法树，因为在每一次规约时都可根据相应的规则生成一小段代码，或者计算出一些信息供将来使用。但是，由于 OpenLua 能按照用户的自定义语法去解析代码片段，而如果要让用户通过直观、自然的手段去获取相关信息并操纵转换这些代码的话，那么在内存中实际构造出一颗语法树便是更好的选择了。OpenLua 的 SLR 解析所用的栈结构与[ASU1986]介绍的稍有不同，在那里符号与状态都保存在同一个栈内，而 OpenLua 是将两者分别保存在不同的栈上，即一个符号栈(symbol stack)和一个状态栈(state stack)。当然，在开始解析的时候，状态栈同样要首先压入一个初始状态(initial state)。

OpenLua 所用的 SLR 解析表也与[ASU1986]描述的标准 SLR 解析表有一些区别。其中最主要的不同是：我们的表中只有移入和规约两种动作(action)，没有表示解析成功的接收(accept)动作。这样做主要是因为 OpenLua 的自定义语法中可以使用内建语法中的所有符号(语法变量)，这其中当然包括起始符号(start symbol)——program(尽管可能很少人这么做)，而按照标准 SLR 解析法，当符号栈只有 chunk 并且 lookhead 是 eof(输入结束符)时，系统直接进入一个特殊的接收状态而不会再将 chunk 进一步规约成 program。用户从符号栈顶得到的语法树便也不是他们期待的 program 语法树，很明显这极容易引起混淆。所以我们把通常填入接收动作的那个条目改为按照 program : chunk 产生式继续规约成 program，并且在状态 1 下增加一个特殊的条目，使得解析表 parseTable(1,"program")等于 0。

3.4.2 开放式 SLR 解析

标准的 SLR 解析只适合把整个输入规约为起始符号，如果我们只想试着将输入的开头一部分解析成某个语法变量(包括但不限于起始非终结符)的话，那么标准 SLR 解析是无能为力的。我们姑且将这种解析称为开放式 SLR 解析吧。很明显，标准 SLR 解析是开放式解析的特例。因为自定义语法可以引用任意内建语法变量，所以开放式解析能力就显得至关重要了。

标准 SLR 解析在最开始要往状态栈压入一个初始状态，对于起始符号来说，它就是 1。但是如果要解析的是非起始符号，那它的初始状态是什么？通过仔细观察状态 1 和起始符号之间的联系我们能够得到某些重要启示。假设状态 n 是语法符号 s 的初始状态，当系统处于状态 n 时，如果输入流的头部确实能够由 s 导出(derive)的话，那么从状态 n 开始进行解析，经过一个有限的动作序列(action sequence)后必然会有向 s 规约这个动作发生。由进一步的分析可知，如果 parseTable(n,s)不为空，那么 n 就是这么一个满足上述条件的状态，即可以成为 s 的初始状态。按照我们对 parseTable(1,"program")的特殊定义(等于 0)，可知 program 确实能以 1 作为初始状态，而且因为对于其它所有不等于 1 的 n 来说，parseTable(n,"program")都没有定义，所以 1 也是 program 唯一的初始状态，这也是为什么标准 SLR 解析在一开始总要把状态 1 压栈的原因了。每个非终结符对应的初始状态会在计算 SLR 解析表时同时算出。

另外，有的时候我们并不在乎是否能把整个输入流解析为指定的语法树，而只关注是否能把输入流的前面一部分规约为指定语法变量。举一个简单的例子，有这么个自定义语法：syntax rootSyntax : " root : namelist"，并且假设此时的源代码输入流的开头一部分为" a,b,c foo()"，如果在这个时候用 parse(rootSyntax)来解析输入，那么"a,b,c"应该被解析为 root。因为尽管"a,b,c foo()"不是一个合法的 namelist，但是"a,b,c"是，因此这部分也就应该按照产生式 root : namelist 规约成 root。为了能够达成这个目的，判断是否解析成功的标准必需改变，即不能依靠判断下一步是否为接收动作来断定解析成功结束了(标准 SLR 解析表中只有唯一的一个条目是 accept 动作)。那么这个判断标准又应该是什么呢？根据上例可知，解析成功就意味着编译器根据（部分）输入构建出了一颗指定类型的语法树，因此更可靠更通用的判断标准应是看解析符号栈中会否出现指定的语法符号并且它是栈中唯一的元素。栈中出现相应语法符号表示已读入的输入确实能被正确规约，而要求它是栈中唯一元素则意味着所有已从输入流中读入的 token 都是

这颗语法树的一部分。

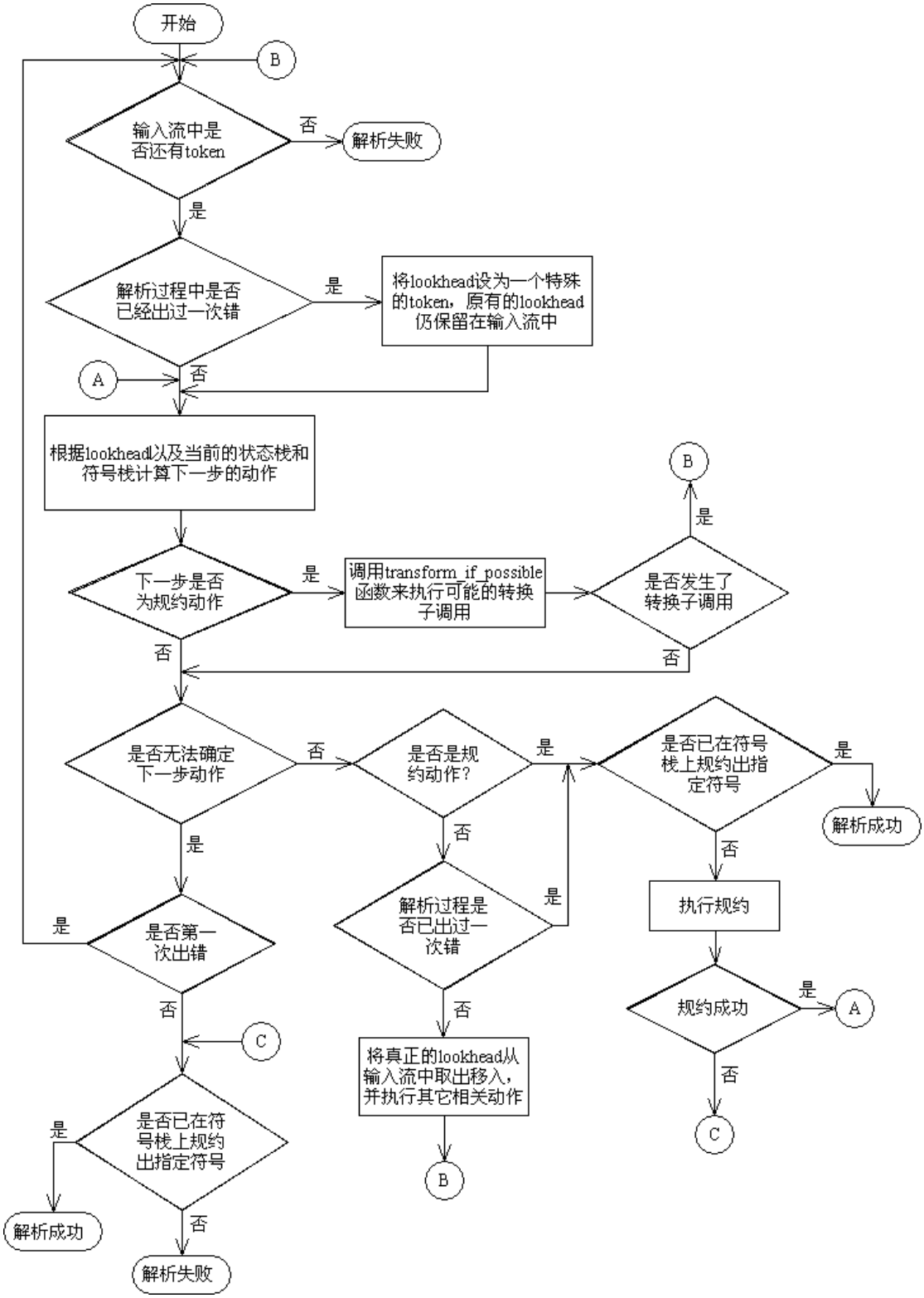

图 3—3 开放式 SLR 解析算法流程图

提出了新的判断标准后，OpenLua 的解析表不设 accept 条目就更好理解了，但是还有一个问题没有解决，那就是何时进行规约。前面已经说了，构造出指定类型语法树的输入

流可能是整个输入流的一部分(头部)，而剩下的输入中可以包含任何类型的 token 甚至对 OpenLua 内建语法来说是非法的字符（比如用户在自定义语法中引入的新的标点符号），它们未必就在指定非终结符的 followset 集合中，因此当它们被词法分析器读入成为 lookhead 时，下一步的动作就很可能不是期待的规约动作，也就无法向规约目标前进。为了解决此问题，我们决定允许在解析过程中出现一次错误(即解析表中未包含指定的下一步动作)，当第一次出现错误时程序设置一个标记用以表明接下来的解析动作都是带错进行的，并且从此用一个特定的 token 伪装成当前的 lookhead，来计算每一步的动作。那么这个特定的 token 到底是怎么选取的呢？其实只要 token 包含在本次解析的目标符号的 followset 集合中即满足条件。这么选取的目的是什么？因为目标符号 followset 中的所有符号都有可能引起一系列的规约动作链，最后导致期望的语法树被构造出来。换句话说，当解析过程第一次出错的时候，我们便用某个可以合法出现在目标符号后头的 token 来“迫使”解析器开始一些列朝向目标符号的规约动作，在这个过程中，如果符号栈的情况满足解析成功的判断标准的话，那么就退出，并返回栈顶的语法树。图 3—3 描述了开放式解析算法的流程。

3.4.3 “热切”原则

再回顾一下上文的"a,b,c foo()"例子。在这个输入的头部，可以成功解析出 3 种 namelist，分别是：（1）"a"；（2）"a,b"；（3）"a,b,c"，它们都是合法的 namelist。那最终生成的语法树是选择哪种情况呢？这里就有一个很重要的原则，即“热切”(eager)原则。它的意思是，编译器会尽可能地在输入流中读取输入，只要读进的这些 token 最终都会被规约成目标语法树的一部分。按照“热切”原则，编译器最终会把"a,b,c"都读入，然后把它们最终规约成一个长的 namelist。假设不采用 eager 原则，那么最后我们得到的常常是一颗语法正确但没有意义的语法树。考虑一下目标非终结符 s 可以为空的情况，一旦在符号栈中通过产生式 s : empty 进行规约生成了 s，没有采用 eager 原则的解析器就会认为此时解析已经成功，于是返回了这颗只能生成空串的语法树。所以，应用 eager 原则才能有效地避免上述没什么意义的行为。并且，根据作者的使用经验来看，“热切”原则更加符合程序员的直觉。就象在本例中，绝大多数人都会认为编译器会把"a,b,c"都读出来，而实际上 OpenLua 也的确会这么做。

3.4.4 与静态元程序相关的 3 个语法构造的特殊处理

OpenLua 中与静态元程序相关的 3 个语法构造是用户自定义语法、转换子定义、编译期模块导入，分别对应 OpenLua 语法定义中的 syntaxdef、transformerdef 和 import_module 3 个非终结符。更进一步，这 3 个非终结符通过如下产生式

```
metastat : syntaxdef stat_sep
         | transformerdef stat_sep
         | import_module stat_sep
```

被归类为 metastat，即元语句(meta statement)。因为 metastat 只与静态元程序相关，不对应任何运行时代码，因此在解析过程中对它们做完相应处理后就应该抛弃(discard)，就好象它们于输入流中所在的位置那儿只有一堆空白字符一样。为了与标准 Lua 的习惯保持一致，OpenLua 允许在上述 3 个语法构造后加上一个可选的分号，但是为什么要将 stat_sep 作为 metastat 的一部分呢？考虑下面这个程序：

```
a = 10;
transformer foo
   return "b = 10"
end;
```

如果 stat_sep 不是 metastat 的一部分，那么解析器将 metastat(此处是一个 transformerdef)抛弃后，得到的等价程序就应该是这样的：

```
a = 10;;
```

注意该语句后面有 2 个分号，而这是非法的。由此我们可以看出，将语句分隔符作为 metastat 的一部分并使得它随着 metastat 的废弃而被废弃可以保证元语句对运行期代码的形式没有任何干扰。

3.4.4.1 自定义语法的处理

当一个自定义语法出现在程序中，那么其后的静态元程序便可以通过它的名字来引用它，因此解析器一旦遇到一个语法定义后，就要把它备妥并保存起来。具体来说，就是分析语法的所有产生式，把产生式信息转换成内部形式，并计算每个语法符号的 firstset 和 followset 集合（此过程中如果发现用户重定义了 OpenLua 内建的关键字或非终结符的

话，那么编译器就报错退出)。完成这些以后，还需要依次遍历每一个非终结符，判断以相同符号为左部的所有产生式的选择(select)集合是否有交集，也即判断该语法是否是 LL(1)语法。如果通过了 LL(1)检查后，便可在静态元环境中以语法名保存所有这些信息。

需要特别注意的是，由于用户自定义语法中可以引用 OpenLua 所有的内建非终结符，而它们在用户语法中绝不会有对应的产生式（否则就出错了），所以被当作一个 keyword 型的终结符，这样为该语法计算出的 firstset 和 followset 就不是我们想要的结果了。有鉴于此，计算 firstset 和 followset 的方法便需要稍做修改：在计算 firstset、folloset 之前，把所有被引用的内建语法符号相应的 firstset 和 followset 作为已经计算好的数据传递给分析程序，让它在这个基础上继续计算，这样最后出来的结果才会符合语法定义者的预期。

3.4.4.2 转换子定义的处理

转换子最重要的功能便是定义了一段静态元程序，更确切的说是静态元函数。因此，当在符号栈顶规约出 transformerdef 时，解析器会立刻根据产生式 transformerdef : transformer Name block end 将其 block（也即该转换子的转换体部分）那一块提取出来，并反解析为标准 Lua 代码，然后利用 loadstring 函数将其编译为一个参数个数为 0 的函数，最后这个编译好的函数连同转换子名将被解析器储存在一个特殊的表(转换子表，与静态元环境不同)中。将来解析器如果遇到类型为 Name 的 token，并且这个名字在转换子表中存在，那么它就立刻调用相应的函数。

3.4.4.3 编译期模块导入语句的处理

import 语句的实际动作是编译用户指定的导入模块，但抛弃所有的运行期代码，用户可以利用它来引入模块文件中的语法定义、转换子定义和执行相应的静态元程序。如果指定导入的模块文件是之前已经被导入过的，那么编译器会忽略该条语句，而决不会重复导入动作。这个性质可以使用户放心地在他们自己的代码文件中导入任何想引用的模块，而无需顾忌模块内部是否指定了重复的导入模块。在处理 import 语句时还需注意有关 parse 接口的问题，3.6.3 小节将对此进行描述。

3.5 用户自定义语法的解析

3.5.1 为什么选择 LL(1)与递归下降算法？

解析按照用户自定义语法格式书写的代码采用的是预测型递归下降(predictive recursive-descent)算法，按照这种算法构建的语法树是从上至下进行的。为什么 OpenLua 对用户语法不采用与内建语法一致的 SLR 解析算法而选择了自顶向下(Top-Down)的方法呢？其实主要的原因有 2 条。

首先，SLR 解析需要计算语法的 SLR 解析表，这个计算过程相当费时。而且即使是比较简单的语法，其最终生成的解析表也会很庞大。比如 OpenLua 内建语法的 SLR 解析表就有 200 多个状态、3000 多个条目，而这些数据在解析代码时将全部读入内存，假如所有的用户语法都必需生成与之对应的解析表的话，那么可想而知内存的消耗将是多么庞大。自顶向下解析不需要这么一张解析表，因此编译器在解析时为了分析用户语法所需的计算量和内存就大大减少了，这将显著地加快编译速度。

其次，因为用户自定义语法可以引用任何内建语法符号，而内建非终结符在用户语法中是不允许被重定义的，因此在用户语法中它们就好象是终结符一样，而这样计算出来的 closure 集合[ASU1986]肯定是不完全的，由此生成的 SLR 解析表也就没有保留有关内建非终结符产生式的信息。按照它来计算下一步动作的时候，明明输入流的头部可以规约为指定的内建语法符号，它却会报错。OpenLua 在解析自定义语法时需要随时判断是否应该开始解析一个内建非终结符，这就需要明确地知道此时此刻采用了哪一个产生式进行导出(derive)，而预测型递归下降算法恰好提供了这种能力。

综上所述，OpenLua 强制要求用户自定义语法必须得是 LL(1)文法也就不难理解了。或许有人担心 LL(1)文法的表达能力不够强大，但用来表达用户自定义的扩展语法时，已经能够描述许多不同的语法格式了。而且，自定义语法还能够引用内建的语法变量，这更增强了它的表达能力。就作者自己的使用经验来看，对于许多复杂的语法格式问题，OpenLua 的方案均能有效地解决。 图 3—4 是自定义语法解析算法的详细流程图。

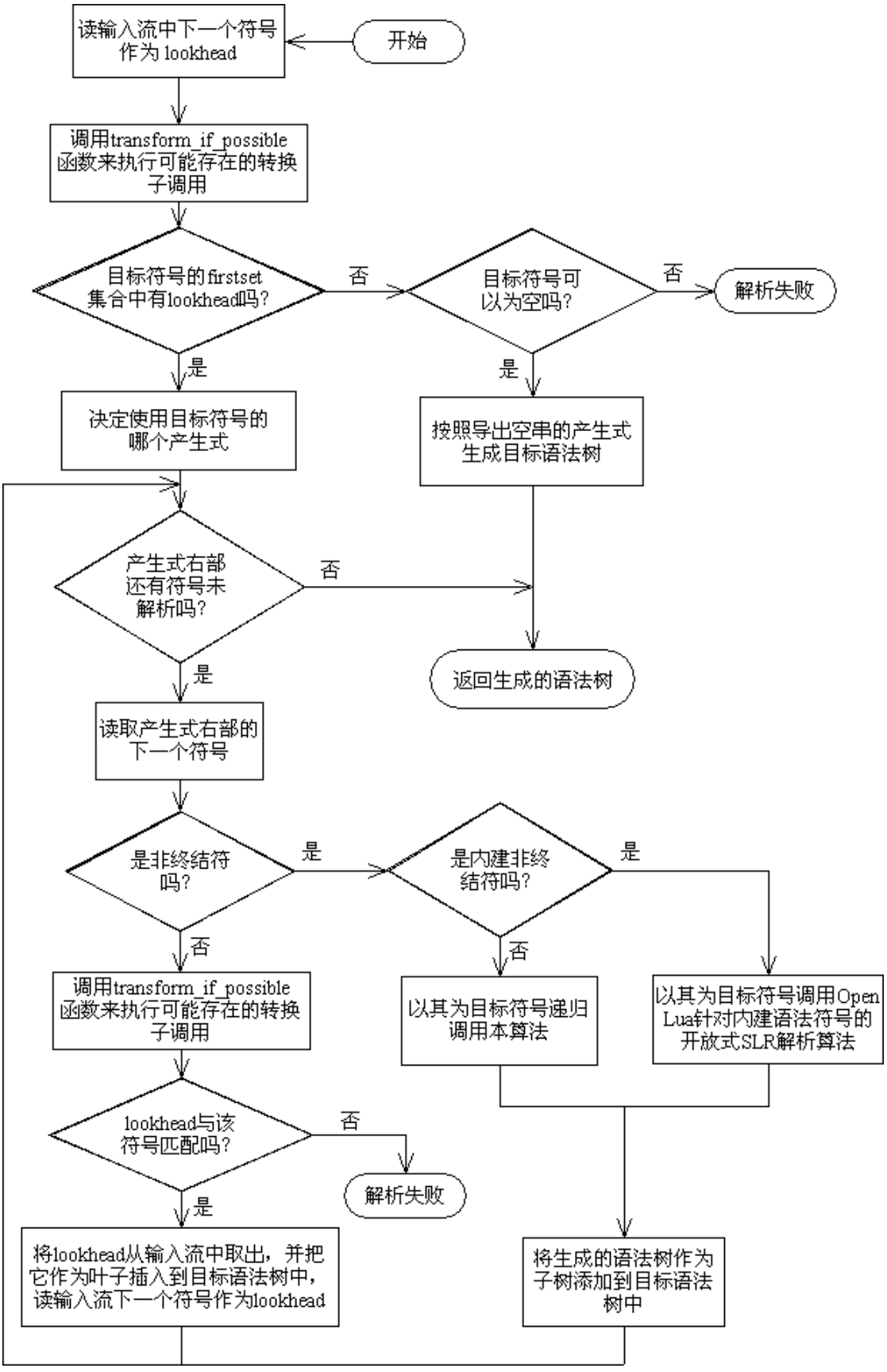

图 3—4 用户自定义语法解析算法流程图

3.6 parse 接口的实现

parse 接口的调用非常简单，只要将一个用户自定义语法传递给它作为参数就可以了，这也许会让读者奇怪，它是如何知道从哪里读入 token 的呢？这个问题牵涉到标准 Lua 语言的一些有趣特性，让作者做一个简单介绍。

3.6.1 闭包和 upvalue 简介

Lua 中的函数是一阶值(first-class value)，定义函数就象创建普通类型值一样(只不过函数类型值的数据主要是一条条指令而已)，所以在函数体中仍然可以定义函数。假设函数 f2 定义在函数 f1 中，那么就称 f2 为 f1 的内嵌(inner)函数，f1 为 f2 的外包(enclosing)函数，外包和内嵌都具有传递性，即 f2 的内嵌必然是 f1 的内嵌，而 f1 的外包也一定是 f2 的外包。内嵌函数可以访问外包函数已经创建的所有局部变量，这种特性便是所谓的词法定界(lexical scoping)，而这些局部变量则称为该内嵌函数的外部局部变量(external local variable)或者 upvalue(这个词多少会让人产生误解，因为 upvalue 实际指的是变量而不是值)。请看程序 3—4：

```
function f1(n)
   -- 函数参数也是局部变量
   local function f2()
      print(n) -- 引用外包函数的局部变量
   end
   return f2
end
g1 = f1(1979)
g1() -- 打印出 1979
g2 = f1(500)
g2() -- 打印出 500
```

程序 3—4 创建闭包的示例

当执行完 g1 = f1(1979)后，局部变量 n 的生命本该结束，但因为它已经成了内嵌函数 f2(它又被赋给了变量 g1)的 upvalue，所以它仍然能以某种形式继续“存活”下来，从

而令 g1()打印出正确的值。

可为什么 g2 与 g1 的函数体一样(都是 f1 的内嵌函数 f2 的函数体),但打印值不同?这就涉及到一个相当重要的概念——闭包(closure)。事实上,标准 Lua 编译一个函数时,会为它生成一个原型(prototype,与 1.4.2 节的原型非同一个概念),其中包含了函数体对应的虚拟机指令、函数用到的常量值(数,文本字符串等等)和一些调试信息。在运行时,每当 Lua 执行一个形如 function...end 这样的表达式时,它就会创建一个新的数据对象,其中包含了相应函数原型的引用、环境(environment,用来查找全局变量的表)的引用以及一个由所有 upvalue 引用组成的数组,而这个数据对象就称为闭包。由此可见,函数是编译期概念,是静态的,而闭包是运行期概念,是动态的。g1 和 g2 的值严格来说不是函数而是闭包,并且是两个不相同的闭包,而每个闭包可以保有自己的 upvalue 值,所以 g1 和 g2 打印出的结果当然就不一样了。

对面向对象编程熟悉的读者可能会觉得闭包与 object 非常类似,事实上我们确实可以这样认为:对象是和某些行为(behaviors)相关联的状态数据(state data),而闭包则是和某些状态数据相关联的行为[Nor1996],二者是一枚硬币的两个面。虽然闭包和函数是本质不同的概念,但为了方便,且在不引起混淆的情况下,我们对它们不做区分。

3.6.2 创建作为闭包的 parse 接口

parse 既需要保持接口的简单,又要能引用若干外部数据,而既然闭包可以保存属于自己的私有数据,那么将 parse 接口实现为带有若干 upvalue 的闭包就是一个自然的选择了。OpenLua 利用一个辅助函数来创建 parse 接口,这个函数的型构(signature)为 create_meta_parser(openlexer, basicTerminals),其中 openlexer 参数传递的是 parse 要使用的开放式词法分析器,basicTerminals 参数指的是供开放式词法分析器使用的一个基本的终结符集合,下文将会有更详细的解释。

根据 3.6.1 小节对闭包结构的描述,调用 create_meta_parser 后返回的闭包的内存布局如图 3—5 所示(闭包对环境的引用图中并未画出)。该闭包执行时(需传递一个自定义语法作为参数,为方便叙述暂假定其名为 llSyntax)的具体行为是:首先取得 openlexer 这个 upvalue 的内部终结符集合并保存在一个临时变量中,然后从 llSyntax 中取得自定义语法的终结符集合,接着计算它与 basicTerminals 的并集并将结果设定为 openlexer 的内部终结符集合,下一步就是调用由图 3—4 描述的解析算法并将结果保存

在临时变量，最后把 opelexer 的内部终结符集合恢复为旧值并将解析算法的结果返回。当一次编译开始，并且 OpenLua 已经创建好了属于此次编译过程的开放式词法分析器时，它会用这个已创建的开放式词法分析器和 OpenLua 内建语法的终结符集合作为参数调用 create_meta_parser 创建一个 parse 接口，并且将其保存到静态元环境中。由此可见，parse 引用的 basicTerminals 这个 upvalue 将始终是内建语法的终结符集合。这是很合乎逻辑的表现，因为自定义语法中的终结符要么是 OpenLua 内建的，要么是用户自己新引入的，所以在解析自定义语法时，要保证词法分析器既正确扫描内建终结符，也能扫描用户语法新引入的终结符，而这正是每次开始自定义语法解析前将 openlexer 的终结符集合设定为二者并集的根本原因。

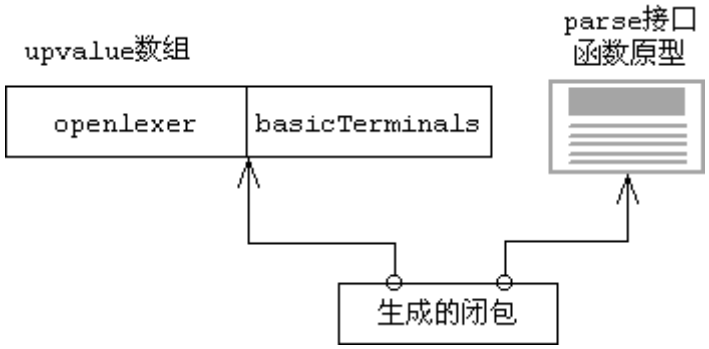

图 3—5 parse 接口闭包的内存布局

3.6.3 parse 与 import 语句的关联

根据 3.4.4.3 小节的描述，import 会引发另一个文件的编译，而这也意味着编译过程的嵌套，那么编译一个模块时该模块调用的 parse 接口应该与导入该模块的文件使用的那个一样吗？要回答这个问题，必须明白一点，即一个正在被编译的文件它调用 parse 接口时期待的一定是由该文件本身及其中的静态元程序生成的字符串产生的 token 流。而又由于 parse 接口需要的那个开放式词法分析器是作为它的 upvalue 保存起来的，所以每开始导入一个新文件时，都应当将已有的 parse 接口(用局部变量)保存起来，然后在元环境中创建一个与之适应的新 parse 接口，待导入动作完成后再将其恢复为原值。

3.7 编译器其它接口及静态元环境的实现

3.7.1 lock 和 unlock

回顾一下程序 3—2 定义的 meta 转换子，它不允许 meta...endmeta 中再嵌套 meta 转换子，这是通过 lock 和 unlock 的配套使用来实现的。试想如果用语法规则该如何表达这样的约束条件？就作者的使用经验看来，这比较困难或至少是很不直观。其实这种约束条件已经是一种语义上的规则了，既如此就应该提供一种语义层的操作而不是将所有工作都压在语法层上。lock 和 unlock 正是这样一对可在语义上防止代码重入的接口。

这两个函数的实现思想非常简单，只需知道是哪个函数(严格说是指哪个闭包)调用了它们，然后在一个特殊的表里设置与该函数相对应的标志即可。既然要知道哪个函数调用它们，那就需要取得当前函数调用栈(function call stack)信息，即 lock 和 unlock 要能找到调用栈中它们的上一层函数。由于标准 Lua 提供了较完善的反射(reflection，或称内省：introspection)机制，这个任务也不难完成，debug 库函数中的 getinfo 便用于取得相应信息。我们可以按 debug.getinfo(level,info_str)这种方式调用它，level 指的是调用栈的层次，当前函数位于第 1 层，而当前函数的调用者(caller)位于第 2 层，依此类推。getinfo 执行完后返回的是一个表，这个表中具体包含哪些信息项由 info_str 字符串指定，如果该字符串中有 f 这个字符，那么在返回的信息表中就会有个 func 字段，它的值就是位于调用栈中相应层次的函数。

3.7.2 静态元环境

静态元环境是静态元程序用于寻找全局变量的地方，也是 OpenLua 在内部维护的一张表。由于静态元程序可以使用标准 Lua 的所有基本函数与库函数，所以一定要能够从这张表访问到它们。为了避免做大量的设置工作，OpenLua 利用了标准 Lua 提供的动态元机制来完成任务。具体来说就是创建一个新的并且__index 字段设为_G（它是个特殊的全局变量，值就是全局环境，即_G._G 等于_G），然后将这个新创建的表设为静态元环境的元表（metatable）。完成上述操作后，所有在元环境中找不到的字段都会到它的元表的__index 字段（即_G）中去找，这样静态元程序引用的所有基本函数和库函数最终都能通过元环境的元表到_G 中找到。

3.8 转换子的调用

转换子被定义后，它的名字就成为了一个”trigger word”，这意味着当编译器遇到这个名字后就会执行转换子的转换体代码。如果调用转换子后得到的输入流的首部仍然是一个转换子，那么就继续调用该转换子。图 3—6 描述了转换子调用的流程，它由 transform_if_possible 函数实现。

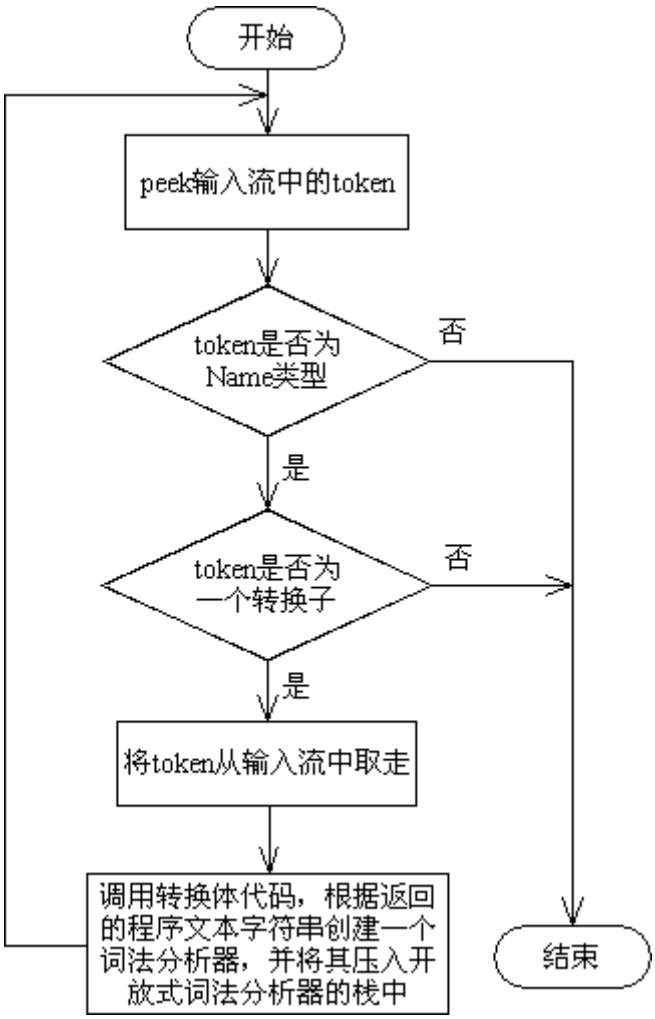

图 3—6 转换子调用动作流程图

根据流程图可以看出，在进行转换子调用时会有可能进入无限循环。比如有一个 foo 转换子，它的调用返回结果是以 foo 为首的程序文本字符串："foo ..."，很显然编译器将会不停地调用 foo 的转换体代码。这种情况很象在递归下降解析算法中遇到了直接左递归，所以程序员在定义转换子时要千万注意避免这种情况的发生。

3.9 小结

本章详细地描述了 OpenLua 系统的设计与实现，其中包括 OpenLua 对标准 Lua 语言的扩展、开放式词法分析器的结构、静态元程序的运行环境、自定义语法的解析、编译器提供的可编程接口等。从中可以看出我们是如何通过引入编译期可编程能力以及开放具体语法树这样一个核心数据结构来为 OpenLua 构造出一个可扩展式框架的。

第 4 章 应用实例

在开始本章之前作者需要特别强调，**以下介绍的所有特性都不是直接实现在 OpenLua 编译器内部的，而是在不改变编译器的前提下，通过静态元编程及 OpenLua 开放的接口扩展得到的**，读者从中可以很直观地体会到开放式架构所拥有的强大扩展能力。如无特别说明，所有扩展的实现均在软件包 import 目录下的 std.ol 文件中。另外，本节大部分示例代码均省略了相应的 import 语句。

4.1 宏系统

C 和 C++的程序员对宏（macro）一定不会陌生，如果使用得当的话，宏可以大大提高程序的可读性及可维护性。标准 Lua 并不支持宏系统，而且尽管不少使用者建议加入这项功能，标准 Lua 的开发者也还是没有引入宏的计划。OpenLua 也一样没有在编译器层支持宏，但是通过它的开放机制，该功能将非常容易实现。

定义一个宏的格式如下：

defmacro Name (parameters)$_{opt}$ Literal

defmacro 开启一个宏定义，Name 是宏的名字，随后可以跟可选的的参数列表，最后的字符串包含的则是宏的定义体(body)。这里要注意的是，如果写上了小括号的话，那么它内部的参数列表不能为空。不带参数的宏使用时只需给出名字，而带参数的则可以象调用函数那样使用，这与 C 语言的习惯非常类似。让我们来看看具体的例子，下面这段代码

```
defmacro PI "3.1415926"
defmacro SQRT(x) "math.sqrt(x)"
local a = math.sin(PI / 2)
local b = SQRT(100)
```

被 OpenLua 编译后将变成

```
local a = math.sin(3.1415926 / 2)
local b = math.sqrt(100)
```

正如预期的那样，所有出现宏的地方最终都被转换成了宏定义体，并且相应的形参都被正确地替换成了实参。和 C 语言一样，这个宏系统工作在词法层，也就是说在宏定义体的任何

位置出现的形参名字都将被替换。

上文的宏系统工作得很好，可是它们到底是如何实现的呢？事实上，defmacro 是一个 transformer，它会将宏定义自动转换成一个同名的转换子定义，而该转换子对应的转换体代码则具体执行在宏调用时(实际是转换子调用)根据传入的实参进行形实替换的动作并返回处理后的宏定义体字符串。

4.2 编译期求值

有了编译期运行静态元程序的能力，很自然便会提出这样的问题：能否把那些在编译期就能决定的值尽可能地预先求出，从而为运行期代码减轻负担呢？这个问题的答案毫无疑问是肯定的。实际上，编译期求值正是静态元程序的一个主要应用。

为了能将编译期求出的值很好地嵌入源程序文本中，需要一个辅助的转换子——text，它接收一个表达式作为参数，当编译器调用 text 转换子时，就会计算这个表达式的值，如果成功则返回值的字符串形式，否则报错。由于计算表达式的动作是在调用 text 转换体代码时进行的，这就要求表达式一定要是可以在编译期的元环境中求值的。看一个例子，假设有

```
defmacro COS(angle)
[[ text(math.cos(angle)) ]]
```

这样的宏，然后在代码中调用它：

```
local value = COS(3.1415926 / 6)
```

上面这条语句将首先（在内部）被转换成

```
local value = text(math.cos(3.1415926 / 6))
```

然后传递给 text 的表达式被编译器求值（注意在该表达式中使用了标准 Lua 库函数 math.cos，这是允许的，因为求值动作位于元环境内，而在元环境中可以引用所有的标准 Lua 函数），最后计算得到的结果被转换成字符串形式插入到 text 的调用点，即

```
local value = 0.86602540825025
```

而 0.86602540825025 正是 30 度角余弦的近似值。

由上述代码可以看出，每当用户书写 text(exp)时，就相当于在该位置写下了 exp 的值的文本形式，明白这一点便不会导致概念上的混乱了。当然，本例的情况非常简单，用户完全可以做许多更复杂更有意义的计算，系统并没有特殊的限制，唯一需要牢记的是：待计

算的表达式必须是一个编译期合法的表达式。

4.3 条件编译与循环编译

C 语言的预处理器（preprocessor）提供了若干根据某些条件对源代码进行处理的指令，比如条件编译#if，这是个很有用的特性，尤其在需要对源代码进行复杂的配置与管理时。与标准 Lua 一样，OpenLua 也未直接提供这样的特性，但是通过静态元程序可以很容易实现出类似的功能。更重要的是，利用图灵完备的静态元语言，我们还能进一步实现大多数普通编译器（包括 C 预处理器）都没有提供的循环编译功能。

4.3.1 条件编译

作者实现的条件编译扩展与 C 语言预处理器的#if 非常类似，用户可以这样使用：

```
meta
  debug = true
endmeta
when (true == debug)
   print("debug version")
endwhen
```

程序 4—1 简单条件编译结构

开启条件编译的“关键字”（实际也是一个 transformer）是 when，随后的括号内写上相应的条件表达式(再次强调,该表达式要能在编译期求值),接着是待编译的代码块(block)，最后的 endwhen 关键字用于标志条件编译结构的结束。象语言内建的 if 语法结构一样，条件编译也允许嵌入 0 或多个分支选择块，它们分别由 elsewhen 和 else 引出。其中 elsewhen 块的格式与 when 部分的完全一样，只不过 when 关键字由 elsewhen 代替，而 else 块只能出现在整个结构的尾部,且没有条件表达式部分。下面是一个多分支的例子：

```
meta
  ver = 2
endmeta
when (1 == ver)
```

```
   print("ver 1.0")
elsewhen (2 == ver)
   print("ver 2.0")
elsewhen (3 == ver)
   print("ver 3.0")
else
   print("unkonwn version")
endwhen
```

程序 4—2 多分支条件编译结构

OpenLua 在编译这段代码时会从上到下依次判断每个条件表达式是否为真，如果为真就选择编译相应分支的代码块并跳出该条件编译结构，如果所有条件都不满足那么最后才会选择 else 分支（如果有的话）的代码块。因此，上述代码编译后的输出将是

```
print("ver 2.0")
```

4.3.2 循环编译

程序员在写程序时经常需要从某个位置开始重复键入大体相同但细节有差别的代码片段，并且这些差别是有规律的。显然这是一项非常机械且繁琐的工作，而凡是机械性的工作都应当尽量将其自动化，循环编译正是为了完成这个任务引入的。它的使用格式请参看下例：

```
meta
   i = 0
endmeta
local cubicTable = {}
loop (i < 4)
" cubicTable[text(i)] = text(i * i * i) "
with
   i = i + 1
endloop
```

程序 4—3 循环编译结构示例

loop 转换子用于开启一个循环编译结构，其后括号内是循环条件，然后是一个包含待编译程序片段的字符串，接着是一个 with 关键字，引出一段执行步进动作的元程序，最后是标志结束的 endloop 关键字。该结构与语言内建的 while 循环非常相似，执行流程如下：

<1> 首先判断循环条件是否为真，如果不为真退出该过程，否则进入下一步；

<2> 编译指定的程序文本，然后运行表示步进动作的静态元程序；

<3> 跳转到<1>步骤。

请注意上例中待循环编译的程序文本中使用了 text 转换子以取得（静态）循环变量的值，这是一种很常用的技巧，正是通过它才实现了大致相同但细节有差别的代码片段的自动生成。程序 4—3 被 OpenLua 编译后输出如下标准 Lua 代码：

```
local cubicTable = { }
cubicTable [ 0 ] = 0
cubicTable [ 1 ] = 1
cubicTable [ 2 ] = 8
cubicTable [ 3 ] = 27
```

如果将循环条件 i < 4 改成 i < 100，那么最终得到的结果就相当于程序员手工书写了 100 条语句，它们依次往 cubicTable 中存入从 0 到 99 的立方值。上述代码或许没有什么实际意义，但是却示范了循环编译的一个重要用途，即通过与编译期求值的搭配使用来自动化地构造某种形式的表（通常是查询表）。

4.4 面向方面的编程与静态代码织入

良好编程的一个主要原则是把与正在开发的系统有关的重要问题以一种清晰的、局部化的方式表达成代码[CE2004]，这种想法早在很多年前就被 Dijkstra 归纳为关切点分离原则（principle seperation of concerns）[Dij1976]。它的好处有许多，首先可以更容易理解在程序中是如何处理问题的，因为不需要在不同的地方寻找它和把它从其它问题分辨出来。此外，可以用定义良好的、局部化模块化的单元（如对象）表示重要的领域概念（domain concepts），从而使开发者可以类比问题领域的结构来组织程序的结构，而这通常会产生比较自然的解决方案。

已有的编程模型已经提供了许多将问题局部化的手段，比如过程、对象、组件等等。但是也确实存在许多问题，它们不能用现有的机制清晰地、局部化地表达，例如并发中的同步、

安全控制等等，它们通常由散落在各处的小代码片段表示。这些难以用传统模块化结构局部化的问题被称为方面（aspect），它们对系统结构的影响可以形象地表示为横切（crosscut）了若干模块。面向方面的编程（Aspect-Oriented Programming，也即AOP）便是为了解决这类问题的一种新的程序设计方法学，它首先由 Xerox PARC 的研究人员提出[KLM+1997]。AOP 的目标是提供方法和技术，用于把问题分解成一系列功能组件（functional component）和一系列贯穿多个功能组件的方面（aspect），然后组合这些组件和方面得到整个系统。AOP 的实现方法有许多种，静态代码织入(static code weaving)便是其中之一。它的做法是根据方面与功能组件之间的关系自动生成表示方面的代码片段并插入到功能组件的内部。由于开放式编译器允许静态元程序操纵源代码的语法树，因此静态代码织入的实现是一件非常轻松的事，接下来的例子便展示了这种便利性。

跟踪程序运行期函数的调用流程是开发过程中很常见的需求，完成这项任务的方法也有很多种。比如在集成开发环境（IDE）中用调试器（debugger）对程序进行单步跟踪便可观察到整个执行流程，但是这种方式需要人工的介入，并不完全自动化。另一种是程序员用手工的方式在函数入口处添加一段记录函数进入的代码，然后在每个出口处又加上一段记录函数退出的代码。很容易想见当函数非常复杂出口点不止一个时，采用这种方法该有多么费时费力！实际上在函数的入口处和每个出口处记录信息的行为是一个典型的横切（crosscut）了多个模块（即函数）的问题，即它是一个方面，并且这个方面对应的代码完全可以用静态织入的办法与函数体集成在一起。

import 目录下的 log.ol 文件正是依据此思路实现的，它为用户提供了一个相当方便的函数调用跟踪记录机制。程序 4—4(假设文件名是 testlog.ol)演示了它的使用。

```
import "log.ol" -- log.ol 定义了 LOG 转换子，在想跟踪的函数前加上即可
LOG function foo(flag)
   if 0 > flag then
      return -1
   elseif 0 < flag then
      return 1
   end
   return 0
end
```

程序 4—4 LOG 转换子使用示例

用 openlua testlog.ol testlog.lua 编译之后得到的 testlog.lua 将是这样的:

```
function foo ( flag )
   print ( "enter foo" )
   if 0 > flag then
      print ( "leave foo" )
      return - 1
   elseif 0 < flag then
      print ( "leave foo" )
      return 1
   end
   print ( "leave foo" )
   return 0
end
```

程序 4—5 LOG 修饰的函数被编译后生成的标准 Lua 代码

注意到了吗？在函数 foo 的入口处和每个出口处都自动插入了 print 函数以打印进入与退出信息，而且该信息中自动包含了函数的真正名字，而程序员所要做的不过是在想跟踪的函数前加上 LOG 而已。如果用户希望执行一些更复杂的记录动作，那么他们完全可以更改 LOG 的转换子定义从而使得 OpenLua 编译器自动插入更复杂的记录代码。

4.5 为 Lua 引入契约式开发机制

4.5.1 契约式开发简介

程序的正确性是软件质量的基础，为了能够开发出无错的软件，开发者需要某种形式化手段的帮助。契约（contract）机制正是面向对象语言 Eiffel[Mey1997]为了保证程序正确性而提供的一套形式化机制。它由类不变式(class invariant)、循环不变式(loop invariant)、例程(routine)的前后置断言(precondition、postcondition)组成，

这些不变式和断言用布尔表达式描述，程序运行期间将按照某种规则在合适的地方检查它们的值以判断程序的状态是否满足契约，如果不满足，说明程序有逻辑错误，此时程序报错并停止执行。契约的检查规则主要有：(1)对象在创建之时以及每次例程调用完毕后其状态都要满足相应类的不变式；(2)例程调用者在每次调用例程时必须保证它的前置断言为真；(3)在对象的类不变式以及前置断言为真时，例程的提供者必须保证例程执行完毕后后置断言为真，并且对象仍满足类不变式。继承体系下的契约稍复杂一点，需对上述规则做某些扩充。首先，子类对象不仅要满足本身的不变式还要满足所有祖先类的不变式；再者，如果存在重定义(redefined，或叫覆写：overriden)例程，那么根据著名的 Liskov 替换原则(即程序中所有的父类对象可以用它的子类对象透明地替换)可知，对祖先类对象是合法的例程调用，对子类对象也一定合法，而且子类(重定义)例程至少能完成祖先类例程能完成的任务。换句话说，子类例程对调用者的要求不应该比祖先类例程高，而它对调用者的保证却不应该比祖先类例程低。这样，子类例程调用者需满足的前置断言实际是例程在子类及所有祖先类中前置断言的逻辑或，例程执行完后需满足的后置断言是它在子类及所有祖先类中后置断言的逻辑与。契约提供了一种精确刻画程序语义的方法，使得开发者能够避免大量的逻辑错误，并且在错误发生时能够迅速而准确地找到根源。在此基础上，Eiffel 的作者进一步发展了一套以软件质量为中心目标的名为“契约式开发（Design by Contract）”的软件开发方法学([Mey1997]，[MM2003])，对程序设计领域产生了深远的影响。

4.5.2 支持契约机制的运行时标准 Lua 扩展库

[DZ2006]利用 Lua 提供的动态元机制设计了一个支持 Design by Contract 的运行时扩展库。该库在 1.4.2 节描述的单继承体系下基本实现了类似 Eiffel 语言的契约检查机制，但是其使用非常麻烦，不仅前置断言、后置断言的表达不如 Eiffel 中那么自然，而且程序员还需撰写大量的辅助代码，这在相当程度上阻碍了它的使用。其实仔细观察便不难发现，大量的辅助代码均是格式基本固定的非常机械的代码，而这正是诸如开放式编译器之类的自动代码产生器(code generator)的用武之地。而且，前置断言、后置断言等概念在表达上的不自然完全可以通过引入自定义语法来解决。因此我们很自然便想到可以利用 OpenLua 来改善契约在 Lua 中的实现。

4.5.3 利用 OpenLua 改善后的契约机制的实现

与契约机制相关的语法、转换子均定义在 contract.ol 中，这个文件可在 OpenLua 软件包中的 import 目录下找到。新引入的两个语法定义如程序 4—6 所示。

```
syntax contractfuncSyntax : [[
cf : Name ':' Name '(' optional_parlist ')' pre block post end
pre : require ':' exp end | empty
post : ensure ':' exp end | empty
]]
syntax invSyntax : [[
inv : Name ':' exp end
]]
```

程序 4—6 书写不变式、前置断言和后置断言的新语法

其中 invSyntax 用于定义不变式的语法格式，而 contractfuncSyntax 则指定了具备前置断言和后置断言函数的定义格式。contract.ol 中还定义了 contractfunc 和 invariant 两个 transformer，用于转换相应的代码。程序 4—7 演示了用户如何实现改进后的契约机制。

```
import "contract.ol"
require("contract.lua")-- contract.lua 是运行时支持库
Point = {x = 1,y = 1}
-- 按照新语法书写的原型不变式
invariant Point : self.x > 0 and self.y > 0 end
function Point:New(o)
   o = o or {}
   setmetatable(o,self)
   rawset(self,"__index",self)
   o = Contract.EquipInv(o)
   Contract.JustCheckInv(o)
   return  o
```

```
end
contractfunc Point:Zoom(times)
   require : times > 10 end -- 按新语法写的前置断言
   self.x = self.x * times
   self.y = self.y * times
   ensure : self.x == old(self.x) * times and
     self.y == old(self.y) * times  end -- 按新语法写的后置断言
end
Point3D = Point:New({z = 1})
invariant Point3D : self.z > 0 end
contractfunc Point3D:Zoom(times)
   require : times > 100 end -- 重定义函数的前置断言
   self.x = self.x * times
   self.y = self.y * times
   self.z = self.z * times
   ensure : self.z == old(self.z) * times end -- 重定义函数的后置断言
end
r1 = Point:New() --OK
r1:Zoom(0.5) -- 前置断言检查失败
r1:Zoom(15) -- OK
s1 = Point3D:New({x = -3,y = -3}) --Point 不变式检查失败
s2 = Point3D:New({z = -5}) --Point3D 不变式检查失败
s3 = Point3D:New({x = 3,y = 4,z = 5}) --OK
s3:Zoom(-2) -- 前置断言检查失败
s3:Zoom(12); -- OK
s3:Zoom(15) -- OK
```

程序 4—7 改进后的契约机制的使用

读者或许很好奇，由 contractfunc 和 invariant 修饰的代码到底会被转变成何种样式？或者说它们的语义是什么？拿 Point 不变式和 Point:Zoom 函数来说，它们实际上被

转换成这样的代码：

```
-- 由 invariant Point : ... 转换而来
function Point:__invariant()
   return self.x > 0 and self.y > 0
end
-- 由 contractfunc Point:Zoom... 转换而来
local function temp(self,times)
   self.x = self.x * times
   self.y = self.y * times
   send(self,times)
end
Point.Zoom   =   Contract.EquipAssert(temp,"self,times","times   >
10","self.x  ==  old(self.x)  *  times  and  self.y  ==  old(self.y)  *
times",true)
temp = nil
```

对比[DZ2006]中的用户代码，可以发现 invariant 和 contractfunc 并没有做什么神奇的事情，但是毫无疑问，它们的确为程序员提供了一种更自然、更有效的表达手段，并且把他们从书写机械性重复性代码的沉重负担中解脱了出来。只有用法上如此便捷的扩展功能，程序员才会乐于尝试吧。

第 5 章 相关研究工作

由于没有哪一种编程语言适合解决所有问题，因此在语言设计领域很早就产生了可扩展式编程语言的概念，并随之提出了用于支持这种扩展能力的元编程技术及开放式编程系统。本文实现的 OpenLua 系统参考了众多研究者在这方面的工作，因此下面几个小节将介绍该领域已有的成果以及与本文工作的联系和比较。

5.1 Lisp 的可编程宏系统

世界上最古老的编程语言 Lisp 语言家族（包括 Common Lisp[Ste1990]、Scheme[Sit2004]、Logo[Har1997]等）很早就引入了一种非常强大的语言扩展机制——可编程宏系统（programmable macro system）。用户可以通过定义一个宏来创建特殊形式的表达式（expression，Lisp 是一个以表达式而非语句为基础的语言）。一个宏是一个符号（symbol），它对应一个转换过程（transform procedure），当 Lisp 遇到一个宏表达式（即以一个宏开头的表达式）时会将宏对应的转换过程施用（apply）在宏表达式的子表达式（subforms）上以将其转换成标准形式的 Lisp 表达式，最后计算该标准形式表达式的值，其结果也就是整个宏表达式的值。不难看出，Lisp 的宏也就是定义了一种从某种形式的代码文本到另一种形式的代码文本的转换。可编程宏系统应用的一个典型例子便是著名的支持面向对象编程的 Lisp——CLOS（Common Lisp Object System），它便是完全用宏来实现的。

Lisp 语法格式非常独特，而且只支持统一且唯一的数据结构——列表（list，这也是 Lisp 名字的由来），尤其重要的是由于以上两点的支持，整个 Lisp 程序本身就是一个不折不扣的列表，再加上符号数据类型（Symbolic Datatype）的引入，这使得 Lisp 可以象操纵普通列表那样操纵自己的源程序。这正是可编程宏系统的基石，Lisp 也由此成为了一门表达能力异常强大的元语言(meta-language)。

就作者看来，Lisp 的可编程宏系统是一个非常卓越的框架，它的思想对后来相关领域的研究产生了深远的影响。但由于 Lisp 始终强调程序与数据对象的统一表示，即使是使用宏也无法自定义新的语法格式，这使得那些畏惧重重括号的程序员对 Lisp 一直敬而远之，尤其在工业界这种情况更为突出。

本文实现的 OpenLua 系统针对的是一个语法丰富（syntactically rich）的编程语言，并且系统允许用户为语言引入新的语法格式，这使得 OpenLua 能够为程序员提供比 Lisp 宏系统大得多的自由度。

5.2 基于抽象语法树数据类型和语法模板置换的宏系统

MS^2（Meta Syntactic Macro System）[WC1993]是一个针对 C 语言实现的语法宏系统。该系统中的宏使用一种扩展的 C 语言来定义，它提供了特殊的抽象语法树数据类型（AST datatypes）。宏的参数是一个由用户指定的语法模板（syntax template，它定义了新的语法），宏的功能体则利用模板置换操作（substitution operations）对模板进行转换。解析器在进行宏调用时会按照模板指定的语法格式进行解析，然后执行宏的功能体以将按照新语法书写的代码转换为相应的普通 C 代码，并将其插入到调用点。

MS^2 系统的语法模板及相应的置换操作在描述代码格式以及代码转换上具有很强的表达能力，但是也确实存在着某些很难用模板置换操作刻画的转换形式。与 MS^2 相比，OpenLua 的源代码转换系统则是一个提供了具体语法树操作接口的完全可编程系统，因此能够完成各种类型的转换操作。

5.3 C++模板元编程技术

ISO C++ [Str1997]包含了一种精巧的模板机制，用于定义参数化类型。C++的模板允许使用类型和整型参数以及部分和完全模板特化，这些语言设施与其它相关特征一起构成了一种图灵完备的编译时子语言。这就使得 C++成为了一种二级语言：一个 C++程序可能同时包含静态代码（它在编译期被执行）和动态代码（编译后在运行时执行）。模板是一种图灵完备语言意味着它同时包含一个条件和一个循环结构，因此它可以用来实现任意的算法，而相关的代码就由编译器在编译时解释。主要的编译时条件结构是模板特化：编译器必须在几个选项中选择一种最精确匹配的模板。编译时循环结构是模板递归，例如类模板的一个成员被用于它自己的定义中。C++对于一个程序的哪一个部分将在编译时求值，哪一个部分在运行时执行都有严格的规则，这些要素使得 C++模板成为具有完全可编程能力的二级语言，而该语言的执行则通过模板实例化（instantiation）和常量压缩（constant folding）来进行。C++模板不但提供了一种编译期编程语言，而且还可以利用它对源代码进行某些操作与转换从而在一定程度上实现领域特定语言。

尽管从理论上来说任何其它图灵完备语言能够完成的工作都可以由 C++模板完成，但是由于模板程序语法的的繁琐和语义的晦涩（与常规 C++程序有很大的区别），使得编写、调试模板程序比撰写常规 C++程序要复杂和困难许多，这无疑阻碍了它的应用。OpenLua 的静态元程序则与普通 Lua 程序具有完全一样的语法格式和几乎一样的语义，这使得任何一个 Lua 程序员都能很快地掌握编译期编程技术，因此该系统具有很强的可用性和易用性。

5.4 利用自省机制及元对象协议的开放系统

OpenC++（[Chi1995]，[Chi1998]）和 OpenJava[TCK+2000]是由同一个开发团队开发的分别针对 C++和 Java 的开放式开发系统。这两个系统都是通过定义比较完善的编译期元对象协议（Metaobject Protocol）和相应的编译期自省机制来实现开放式的架构。与上述其它开放式环境不同，在这种系统中用户取得的不是源程序的语法结构信息而是逻辑结构（logical structure）信息，因此程序员可以站在一个比较高的抽象层次上对源程序进行转换。

以 OpenJava 为例，它对源程序的处理过程如下：

1、分析源程序并为每一个 class 产生一个 class 元对象（metaobject）；

2、调用 class 元对象的成员函数执行宏扩展（macro expansion）；

3、生成常规 Java 代码，这些代码是通过 class 元对象对原始源程序进行转换的结果；

4、运行常规 Java 编译器编译转换后的程序并生成字节码。

通过 OpenC++和 OpenJava 定义的编译期元对象协议和扩展的编译期自省机制能在编译时方便地取得源程序的逻辑结构，这种方式相比语法树能够提供更多的高层语义信息，从而使得程序员能更加自然和高效地编写那些操纵和转换源代码的静态元程序。然而，由于元对象协议比较适合面向对象语言，所以这种类型的系统具有很大的局限性。由于 Lua 是一门过程型语言，源程序的基本组成单位不是 class，所以 OpenLua 选择了基于语法树的方式来提供开放式的框架。

5.5 意图编程

意图编程（Intentional Programming，[Sim1995]，[Sim1996]，[CE2004]）是一种基于主动源（active source）的扩展编程和元编程环境，用户可以扩展环境的任何部分，包括编译器、调试器、编辑器和版本控制系统等等。意图编程提供元程序设计能力，

包括一个代码转换框架、用于协调使用独立的开发语言扩展的代码编译的协议、用于调试元代码的特殊调试机制和用于扩展环境的各个部分的标准 API 集合。

意图编程的一个主要思想是源程序不表示为简单的 ASCII 文本，而是作为主动源，就是一个带有编程时（programming time）行为的图形数据结构。主动源的行为是由相应的方法来实现的，方法定义源的不同方面，包括它的可视化、入口、浏览、编译、调试和版本控制行为。编程语言的各种设施被称为意图（intention），任何通用目的和领域特定语言的意图都可以被载入并在用户程序中使用。在最简单的情况下，可以载入一组 C 语言的意图，它将允许用户编写 C 程序。意图编程系统允许程序员声明新的意图并实现相应的方法从而实现新的意图，那些方法被编译成扩展库。当想使用某种意图时，就把对应的扩展库载入到开发环境中，根据扩展库中的代码，开发系统知道如何显示这些意图，并且知道如何支持它的入口、浏览、编译、调试和版本控制等等。

意图与对象类不同，一个意图的方法是在编程时被调用的，并且它们支持与使用该意图相关的各个方面。意图编程系统可以将领域特定抽象作为意图，这样就可以实现领域特定记法，而且记法除了可以是文本形式外，还能采用图形形式，因为每个意图都有相关联的方法来显示自己。比如，程序员可以在源程序中嵌入数学公式或者 GUI 控件等等，而且系统还可以应用复杂的应用程序代码优化来生成非常高效的代码。由于可以引入新的领域特定记法，程序员便能在源代码中表示更多的分析和设计信息，这使得原本许多需要通过注释来说明的问题变得不再必要了。

意图编程系统中的源程序在编辑时就作为一个带有相应链接关系的语法树（更确切地说是一个图，被称之为源图：source graph）而不是文本，因此意图编程系统不需要一个解析器。系统带有一个规约引擎（reduction engine），它在检查了源图的结构是否正确后便应用一系列转换产生规约代码（或称 R-code，只包括基本意图）。然后一个合适的后台根据给定的 R-code 产生特定的目标代码，最后标准链接器将目标文件链接成一个可执行文件或者库。系统的调试器、版本控制系统的工作均是基于源图而进行的。

意图编程是一个相当令人激动的想法，虽然其中的许多思想早有人提出，但是意图编程却将这些好思想紧密地集成在一起，并构建了一个具有无限扩展性的卓越的开放框架。OpenLua 与之相比，更象是该系统中一个只针对 Lua 语言的扩展库实例。毫无疑问，意图编程系统必然会将开放式开发系统的发展推向一个崭新的高度。

5.6 其它工作

除了上面介绍的系统外，还有一些其它相关工作：

1、[LKR+1992]介绍了一种利用编译期元对象协议实现的开放式 Scheme 语言编译器；

2、IBM Watson 和 Toronto 研究中心的研究人员为 C++语言定义了一种统一的名为 CodeStore 的内部表示，并且在此基础上开发了一个具有高度可扩展性的 C++开发环境——Montana（[SKB+1997]）；

3、由 REBOL Technologies 开发的 REBOL 项目[REB]包括两部分：一是名为 REBOL 的动态语言，另一个是基于 REBOL 语言的 Internet 操作系统。REBOL 语言提出了方言（dialect）的概念，允许通过指定 BNF 形式的产生式来引入具有新语法格式的标记法，从而使得用户可以很方便地按需定制领域特定语言。

5.7 小结

本章较全面地介绍了可扩展式开发系统研究领域的相关工作，这些成果中的许多思想都为 OpenLua 所借鉴。与这些系统相比，OpenLua 实现了一个以具体语法树为扩展操作的核心数据结构，针对一门语法丰富的过程型动态语言的开放式编译器，但它并不是一个完整的可扩展式的开发环境，因此相对于意图编程、Montana 这样的系统，它的可扩展能力仅仅局限在语言层面上。

第 6 章 总结与改进计划

6.1 编译终止问题

由于 OpenLua 的静态元程序是在编译期由编译器执行的，并且与常规 Lua 语言一样是一种图灵完备的编程语言，所以静态元程序的运行是否可终止就属于一个不可判定的问题。这意味着 OpenLua 程序的编译过程可能不会停止而一直进行下去。

在提供完全的静态元编程能力和保证编译终止这两者中，OpenLua 选择了前者，而把撰写正确的元程序以使编译过程一定结束的责任留给了使用者。因此，开发者在编写静态元程序时要相当细心与谨慎，他们必须明确地知道这些程序会导致编译器有什么样的行为。

6.2 编译时间

虽然静态元程序不会带来运行时代价，但是复杂的元程序的运行必然要耗费大量的编译时间，因此毫无节制与非理智地使用元程序会显著地降低开发效率，尤其在开发大型项目时会更加明显。程序员必须在开发时效率和运行时效率之间仔细权衡，明智地使用静态元程序，这样才能达到最大的综合效率。

OpenLua 编译器本身是用标准 Lua 语言开发的，因此它的运行速度相对来说比较慢。另外，OpenLua 是将源程序解析、转换后再反解析输出标准 Lua 源程序，然后由标准 Lua 编译器编译反解析得到的标准 Lua 程序，最后生成字节码。很明显，标准 Lua 编译器重复了一遍源程序的解析工作。如果将生成字节码的翻译器集成到 OpenLua 编译器中，那么就可以由 OpenLua 源程序直接编译得到字节码文件，这也会大大缩短整个编译过程。

作者的下一步改进计划的重点便是用 C 语言重新实现 OpenLua，并将后端翻译器集成到其中，以提高系统的编译速度。

6.3 用户自定义语法的描述

本文实现的 OpenLua 系统提供的描述自定义语法的手段是用标准 BNF 格式定义产生式，并且要求新引入的文法一定是 LL(1)文法。但是根据作者的经验，一些语法格式都不能或者难于用 LL(1)文法描述，并且由于采用标准 BNF，需撰写的产生式个数也较多，这

不但增加了用户定义领域特定标记法的难度，也会使得源程序的解析时间增加。EBNF 作为一种扩展了的 BNF，具有直观、自然的优点，有许多用 BNF 描述不清的语法换作 EBNF 后便显得简洁明了。因此在这方面的改进重点是允许用户用 EBNF 代替 BNF 来定义产生式，并且改进递归下降算法以去除自定义语法必须是 LL(1)文法的限制。

6.4 具体语法树的操作

OpenLua 为了实现扩展功能，开放了一系列操作具体语法树这一核心数据结构的接口，通过这些接口可以取得源程序的信息并执行各种各样的转换。但是因为解析树本身是源程序的一个完整镜像，包括了所有的低层次的细节，所以针对具体语法树的操作是相当低级与繁琐的。为了使用户可以不用或者少关心源程序完整的书写格式，编译器提供一套能操纵程序逻辑结构的接口将是一个非常好的选择。作者计划为 OpenLua 定义类似于编译期元对象协议这样的机制，然后通过它来提供工作在更高层次的可编程接口。

6.5 静态元程序的调试

由于编译器在编译过程中同时还要执行静态元程序，并且它所编译的是经过静态元程序转换后的源代码，所以如果这期间静态元程序执行错误或者转换后得到的源代码有语法错误都会导致编译过程出错退出。遇到这种情况时，用户很难根据编译器给出的简单的出错信息找到错误的真正引发点，因此就极需能够象调试动态程序那样调试静态元程序的能力，并且能够在任何一个执行点观察静态元程序产生的中间结果（即转换得到的源代码片段）。

目前的 OpenLua 编译器只为静态元程序提供了一个执行平台，并没有提供相应的调试能力，这使得系统无法为用户编写正确的静态元程序提供任何帮助。因此，为 OpenLua 加入元程序调试能力是改进计划中的优先任务，但对于如何实现该功能作者却还没有什么好的想法。在这方面最值得参考的当属意图编程系统了，作者打算对其做进一步的研究以借鉴相关思想来为 OpenLua 实现一种完善的元程序调试机制。

6.6 编译器的开放程度

OpenLua 是一个开放式的编译器，但是确切地说，它的可编程接口只开放给了静态元程序，也即用户只有通过编写元程序才能调用这些接口来实现对系统的扩展。相比于普通的

封闭的系统，OpenLua 的开放度当然很大，但如果更进一步思考就会提出这样一个问题：为什么不把这些接口开放给独立于编译器的外部的工具？要做到这一点，就需要一种更加开放的框架结构，以使外部独立的软件也能够通过某种机制调用编译器的可编程接口。

增加编译器的开放程度，将使得围绕着编译器构造一个完整的可扩展式开发环境成为可能，这也是作者所认为的 OpenLua 最有意义的发展方向。

附录 A OpenLua 语法(BNF 格式)

```
program : chunk

chunk : optional_compound_stat

optional_compound_stat : compound_stat | empty

compound_stat : compound_stat one_stat | one_stat

one_stat : metastat | stat stat_sep

metastat : syntaxdef stat_sep
         | transformerdef stat_sep
         | import_module stat_sep

syntaxdef : syntax Name ':' Literal

transformerdef : transformer Name block end

import_module : import Literal

stat_sep : ';' | empty

optional_stat_list : stat_list stat_sep | empty

stat_list : stat_list stat_sep stat | stat

block : optional_stat_list

stat : varlist '=' explist
     | functioncall
     | do block end
     | while exp do block end
     | repeat block until exp
     | if exp then block optional_elseif_part optional_else_part
end
     | return result_part
     | break
     | for Name '=' exp ',' exp do block end
     | for Name '=' exp ',' exp ',' exp do block end
     | for namelist in explist do block end
```

```
     | function funcname funcbody
     | local function Name funcbody
     | local localvar_list init_part
optional_elseif_part : elseif_part | empty
elseif_part : elseif_part elseif exp then block
            | elseif exp then block
optional_else_part : else block | empty
result_part : explist | empty
localvar_list : localvar_list ',' Name | Name
init_part : init | empty
funcname : Name dotname_list colone_name
dotname_list : dotname_list '.' Name | empty
colone_name : ':' Name | empty
varlist : varlist ',' var | var
var : Name
    | prefixexp '[' exp ']'
    | prefixexp '.' Name
namelist : namelist ',' Name | Name
init : '=' explist
explist : explist ',' exp | exp
exp : nil | false | true | Number | Literal
    | functiondef  | var | functioncall
    | '(' exp ')' | tableconstructor
    | exp binop exp | unop exp
prefixexp : var | functioncall | '(' exp ')'
functioncall : prefixexp args
             | prefixexp ':' Name args
args : '(' explist ')' | '(' ')'
     | tableconstructor | Literal
functiondef : function funcbody
```

```
funcbody : '(' optional_parlist ')' block end

optional_parlist : parlist | empty

parlist : parname_list | '...'
        | parname_list ',' '...'

parname_list : parname_list ',' Name | Name

tableconstructor : '{' optional_fieldlist '}'

optional_fieldlist : fieldlist | empty
                   | fieldlist fieldsep

fieldlist : fieldlist fieldsep field
          | field

field : exp | '[' exp ']' '=' exp
      | keyname '=' exp

keyname : Name

fieldsep : ',' | ';'

binop : '+' | '-' | '*' | '/' | '^' | '..'
      | '<' | '<=' | '>' | '>=' | '==' | '~='
      | and | or

unop : '-' | not
```

附录B OpenLua操作符优先级定义表

优先级由高到底：

操作符	类型	优先级	结合律
^	二元	8	右结合
Not	一元	7	左结合
-	一元	7	左结合
*	二元	6	左结合
/	二元	6	左结合
+	二元	5	左结合
-	二元	5	左结合
..	二元	4	右结合
<	二元	3	左结合
>	二元	3	左结合
<=	二元	3	左结合
>=	二元	3	左结合
~=	二元	3	左结合
==	二元	3	左结合
And	二元	2	左结合
Or	二元	1	左结合

参考文献

[ASS2004] Harold Abelson, Gerald Jay Sussman, Julie Sussman. **计算机程序的构造和解释**. 第二版. 裘宗燕译. 机械工业出版社，2004.

[ASU1986]Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman. **Compilers: Principles,Techniques,and Tools**. Addison Wesley, 1986.

[CE2004] Krzysztof Czarnecki, Ulrich W. Eisenecker. **产生式编程——方法、工具与应用**. 梁海华译. 中国电力出版社，2004.

[Chi1995] Shigeru Chiba. **A Metaobject Protocol for C++**. In Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95). ACM Press, 1995. 285-299.

[Chi1998] Shigeru Chiba. **Macro Processing in Object-Oriented Languages**. In Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98). IEEE Press, 1998.

[Dij1976] Edsger W. Dijkstra. **A Discipline of Programming**. Prentice Hall, 1976.

[DZ2006] 邓际锋，张桂戌. **在动态语言 Lua 中支持契约式开发**. 计算机工程与应用，2006 年 8 月（预计）.

[Har1997] Brian Harvey. **Computer Science Logo Style, volume 2: Advanced Techniques**. Second Edition. MIT Press, 1997.

[Ier2004] Roberto Ierusalimschy. **Programming in Lua**. http://www.lua.org/pil/, 2004.

[IFC2003] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes. **Reference Manual for Lua5.0**. http://www.lua.org/manual/5.0/, 2003.

[KLM+1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. **Aspect-Oriented Programming**. In Proceedings of the European

Conference on Object-Oriented Programming (ECOOP'97). Springer-Verlag, 1997. 220-242.

[Lie1986] Henry Lieberman. **Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems**. In Proceedings of OOPSLA'86. ACM Press, 1986. 214-223.

[LKR+1992] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., Erik Ruf. **An Architecture for An Open Compiler**. Workshop on Reflection and Meta-level Architectures, Tokyo, 1992.

[Mey1997] Bertrand Meyer. **Object-Oriented Software Construction**. Second Edition. Prentice Hall PTR, 1997.

[MM2003] Richard Mitchell，Jim Mckim. **Design by Contract 原则与实践**. 孟岩译. 人民邮电出版社，2003.

[Nor1996] Peter Norvig. **Design Patterns in Dynamic Programming**. http://norvig.com/design-patterns/ppframe.htm, 1996.

[REB] Homepage of REBOL Technologies. http://www.rebol.com, 2005.

[Sim1995] Charles Simoniy. **The Death of Computer Languages, The Birth of Intentional Programming**. Technical Report MSR-TR-95-52. Microsoft Research, 1995.

[Sim1996] Charles Simoniy. **Intentional programming: Innovation in the legacy age**. IFIP Working group 2.1 meeting, 1996.

[Sit2004] Dorai Sitaram. **Teach Yourself Scheme in Fixnum Days**. http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html, 2004.

[SKB+1997] Danny Soroker, Michael Karasick, John Barton, and David Streeter. **Extension Mechanisms in Montana**. In Proceedings of the 8th IEEE Israeli Conference on Computer Systems and Software Engineering. IEEE Computer Society Press, 1997.

[Ste1990] Guy L. Steele. **Common Lisp: The Language**. Second Edition. Digital Press, 1990.

[Str1997] Bjarne Stroustrup. **The C++ Programming Language**. Third

Edition. Addison Wesley, 1997.

[TCK+2000] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, Kozo Itano. **OpenJava: A Class-Based Macro System for Java**. In Lecture Notes in Computer Science 1826, Reflection and Software Engineering. Springer-Verlag, 2000. 117-133.

[US1987] David Ungar, Randall B. Smith. **SELF: The Power of Simplicity**. In Proceedings of OOPSLA'87. ACM Press, 1987. 227-241.

[WC1993] Daniel Weise, Roger Crew. **Programmable syntax macros**. In Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation. ACM Press, 1993. 156-165.