

设为首页 收藏一下又不会死! 新人公告 申请友链

0xaa55 · 技术宅的结界

帐号

UID/用户名/Email

☐ 自动登录

找回密码

密码

登录

立即注册 → 加入我们

用QQ帐号登录

只需一步, 快速开始

论坛广场 新人公告 搜索帖子 最近发表 淘专辑 帮助

快速导航

请输入搜索内容

帖子

Q

热搜: 下载 VB C 实现 编写

» 主页 » 计算机技术 » 技巧探讨 » lex与yacc简介

发帖

返回列表

查看: 5855 | 回复: 3

lex与yacc简介 [复制链接]

cyycuish

My Brain

85 260 3681

主题 帖子 积分

用户组: 管理员

No. 418

UID 418

精华 13

威望 52 点

宅币 1969 个

贡献 1281 次

宅之契约 0 份

在线时间 252 小时

注册时间 2014-8-9

收听TA 发消息

发表于 2016-11-15 01:12:59

只看该作者

只看大图

楼主 电梯直达

欢迎访问技术宅的结界, 请注册或者登录吧。

您需要 [登录](#) 才可以下载或查看, 没有帐号? [立即注册→加入我们](#) [用QQ帐号登录](#)

Lex与Yacc是用来构建编译器的工具。

在这篇帖子中, 我将会带大家简单认识lex与yacc。这篇文章的宗旨是“有C语言编程基础”的同学就可以同我一起来学lex与yacc。所以有一些专业性非常强的单词我会做简单解释, 有的一下子没法解释清楚我会将单词高“暗”标出, 大家可以利用搜索引擎查询这些单词, 或者购买相关书籍进一步学习。

首先我必须引进一些编译系统的知识。如果我们用C语言写一款程序, 写好后在IDE内选择 编译 / 执行。然后就可以得到运行结果。除了前期学习程序语法和编写程序的过程, 一切看起来都是那么简单容易。而简单容易的背后隐藏着计算机科学最复杂的分支之一“编译系统”的整个内容。有一句话讲: 编写程序, 就是将程序使用对象当傻子看。(桌面应用程序员, 将用户当傻子。编写程序库的程序员将使用库的程序员当傻子)而学习计算机科学, 却需要无比勤奋的头脑。所以我们勤奋地拆解一下“鼠标左键单击‘编译执行’”这一步简单动作背后发生的一系列事情。

第一步, 我们的源程序被保存到磁盘(外部存储设备)。然后由IDE调用编译器。接着在编译器内部, 词法分析器(Lexer / Lexical Analyzer)将源程序读入。编译器这时无法理解整个程序, 甚至连个void int float也认不得, 在词法分析器入口处它读到的是字符流。然后词法分析器将出现字母 'i'、'n'、't', 组合成 int, 一个C语言关键字 从而被识别。被识别出的 int 成为一个词素。如果 int 后紧跟着一个Tab或是一个空格 int 就被词法分析器成功吐出。这些若干个词素组成词素流, 进入语法分析器(Syntax Analyzer/Parser)的口中。在语法分析器中, int空格main被理解为“要定义一个主函数”, 而int空格mian分号被理解为“定义了一个叫做mian的整型变量”。这些意思被语法分析器理解后, 构成了一个叫做“语法树”的数据结构实例。大家可以想象, 在内存中长出一棵大树。这棵树是由你写的源程序构成的。在树上, 编译器继续对源程序做语义检查。在这一步 int a=0; int a=1; 会被报变量重定义、int a = 1+2+3; 会被理解为 int a = 6; 语法树进一步长大成为抽象语法树(Abstract Syntax Tree / AST)。

在这以后, 抽象语法树继续发生变化。但是在这个阶段我们把它称为“编译器后端(A compiler's back-end, 前端叫做 Front-end)”在后端内目标代码逐渐被生成。如果是通常意义上的C语言, 对应平台的汇编语言被生产出来。如果是C#语言, MSIL(微软中间语言 Microsoft Intermediate Language)被造出来。如果是java, 那么会出现Java虚拟机字节码(Java VM ByteCode)。当然, 抽象语法树AST的结果就如同人类生孩子, 也是相当不容易的。在最终的目标代码来到世界上之前, 编译器后端必须对目标代码进行优化, 以增加运行效率、减少程序体积。这里我举两个个后端优化的例子: 这第一种优化叫做“Stack Packing”(直译为栈打包)其中原理是通过对一些寄存器的复用来达到缩减目标代码体积的效果。第二种是优化失败的例子, 这个问题曾经导致了严重的IE漏洞! 它仅仅是一个微软C/C++编译器后端C2. DLL内错误处理“悬挂指针(dangling pointer)”导致的。攻击者在HTML内嵌的javascript中首先复制一个事件, 但是不在事件内添加关于任何对象的引用, 接着删除一个不存在的对象引用。此时编译器后端并未很好地优化发现事件指针指到了一处未被分配未加以管理的内存区域。接着攻击者便可将自己的代码任意地通过这个“悬挂指针”植入对方计算机。这个起源于C2. DLL内的“后门”最终被微软C++优化器架构师 Jim Radigan 修复关闭。

那么Lex与Yacc是干什么的? Lex其实是Lexer的简写, 它不是一个词法分析器可以被嵌入到任意编译器的前端, 而它是一个“词法分析器”的生成器, 它用来帮助你写一个“词法分析器”。至于Yacc它是语法分析器的生成器。Yacc的全名叫做 Yet Another Compiler Compiler, 甚至是编译器的编译器呢——它名字就是这么叫的。顾名思义是用来编译编译器的编译器(首先它是一个编译器, 意味着他有自己的一整套系统, 词法分析、语法分析、目标代码生成……其次它被用来写其他编译器。如果你有的是空闲时间, 你可以用Lex+Yacc写个Yacc)。这个好怪啊, Lex与Yacc是一种编译器, 他们用来写其他编译器。见怪不怪, 我们稍加思索会想到: 我是人, 生我的是人, 我生的也是人。生我的人可以生人, 我生的人也

可以生人，生我的人生出来的是人，我生的人生出来也是人。晕~(@_@)~

既然刚才说了，Lex和Yacc都是独立的编译系统，那么Lex和Yacc接受的源代码是什么，目标代码又是什么？Lex接受扩展名为.l的语法分析文件，Yacc接收.y为扩展名的词法分析文件。而Lex和Yacc的目标代码都是.c结尾的纯种C语言源文件。过程是这样的.l文件被Lex编译，生成一个默认叫做 lex.yy.c 的C源文件。.y的文件被Yacc编译生成一个叫做某某.tab.c 的C源文件。这两个C源文件再被C编译器编译，生成可执行文件。这个可执行文件，就可以是一个BASIC编译器的词法分析器和语法分析器。它没有仔细处理语法，也没有生成任何目标代码，所以它——这个可执行文件仅仅是一个BASIC编译器前端的前端。为了形象化理解，我们先把一个编译器分成两半：前1/2叫做前端，后1/2叫做后端。我们把前1/2再剪成两半。这最前的1/4的工作由Lex与Yacc帮我们完成。那么编译器的前1/8叫做词法分析，第2/8节处叫做语法分析。3/8、4/8是我们自己写的用来构建语法树的东西，编译器的后1/2用来处理、优化、生成目标代码（学到后来我们发现这个步骤可以由LLVM（底层虚拟机 Low-level virtual machine）代劳）。

但是既然这是一篇有关 Lex和Yacc 的简介。我只会“简单”介绍整个编译器的前1/4。这1/4已经复杂到不能再复杂了，关于此1/4的理论，可以写一本500页左右的书。什么？你现在放弃学习了？我们在开头约定好的：编写程序，就是将程序使用对象当傻子看。而学习计算机科学，却需要无比勤奋的头脑。你是想临阵脱逃当个傻子，还是努力勤奋地当个计算机科学家？事先给你鼓点气：计算机之父图灵研究的大致就是编译器前端（确切地讲，叫做自动机理论(Automata Theory)，[同时还有可计算性：NP可解啊、NP难解啊等，引申到算法分析，AI……]）。



据说凡是谈到编译系统总得放只“龙”作为象征，所以我也找了条龙以表对传统的尊重。
至于为什么会是中国龙，因为文章作者是中国人嘛 2333.

我们先谈谈Lex与Yacc的那年那些事儿。首先说yacc，它与1975到1980年间被他爸：贝尔实验室的 Stephen C. Johnson 写成。当时基于对自动机理论的研究，不少人都在写自己的语法分析器。而史蒂芬 这伙计的yacc是语法分析器里边最出色的。flex和bison一书对其的解释是：首先因为 有D. E. Knuth 这个科学家的理论奠基同在贝尔实验室 Johnson 写了一款当时最可靠的语法分析器，yacc的优良基因最早可以追溯到这儿。后来一个叫做 Bob Corbett 的加州大学伯克利分校毕业生改进了yacc的内部算法。Bob就是加大伯克利版yacc的作者。加大伯克利yacc有着比Bell更灵活的授权协议，比贝尔yacc更快的速度，于是乎 GNU 的 Richard Stallman 就来了激情：哟！我也想把你的伯克利yacc搞到我们 FSF(Free software foundation, 自由软件基金会) 这儿来呀。FsF团也需要yacc呢！) 后来加入 FSF 的yacc就叫做Bison。读到这儿的读者注意了：哟怎么还改名了呢？对，下文我们遇到了Bison就是yacc啦。

1975年，俩好朋友 Mike Lesk 和 Eric Schmidt 写了 Lex。其实Lex大部分代码由 Schmidt 完成。(Lesk是攻，Schmidt是受，Lesk提供建议，Schmidt帮我造lex——开玩笑的！Schmidt后来是Google CEO) 当时 lex 和 yacc 都是独立的程序部件。因为Lex也火了，1987年 劳伦斯伯克利实验室的 Vern Paxson 看上了lex，之后他便将lex用C改写。并命名为flex——意为(Fast Lexical Analyzer Generator 快速词法分析器生成器) 从此lex就成了伯克利Flex了。

接下来，我们一起研究flex的基本原理及用法。词法分析技术所基于的理论叫做“确定型有穷自动机(DFA/Deterministic Finite automaton)”大致说来DFA就是一个这样的数学模型：一开始一个DFA存在在某种状态上，然后一些有限种类的符号被输入到机器内，机器接受符号，通过某种机制（这种机制统称为“转移函数”将自动机换成另一种状态，这些状态的数量都是可数的。当然，自动机不是永动机，这些状态里边还有一个停机 / 终结状态。）那么所谓DFA确定型有穷自动机，确定在每个转移函数都会返回给机器唯一的一种状态。自动机所有状态的总和可以被计数，就是有穷状态自动机。合起来叫做确定型有穷自动机，DFA。

哎~没办法，概念总是最枯燥，最难懂，最形式的。下面我们力求吧概念形象化：我们和主题词法分析结合起来。帖子第二段我们说过，一开始词法分析器只认得字符。源程序被看成字符流输入到编译器的词法分析器。这些字符就是一个一个输入信号。当C语言词法分析器扫描到void，会在内部标记出一个“我已经看到void啦”的状态。如果下一个字符是Tab或者空格，那么“我已经看到void啦”这个状态就会被送出，以便告诉语法分析器：“给你一个void词素”。那么我们假设：如果扫描到void，后面紧接着是个a，而不是空格或者Tab，那么这时后C语言词法分析器就会在“表”内查：“voida是不是C语言关键字呢？” 结果发现不是。然后再查：“voida 符合C语言变量函数命名规范吗？”。因为 voida 不是以数

字开头的，所以当voida后面跟着Tab、空格或者分号，小括号后，词法分析器又告诉语法分析器：“给你个voida，她是用户自定义的某种东西”。注意词法分析器不知道voida是变量还是函数名。如果有 `int voida;` 和 `int voida();`；则前者是变量后者是函数的判断由语法分析器构建语法树时负责处理。

词法分析的关键技术就在这里，大致来说是相当简单的。今天你会词法分析了么？So Easy！既然讲到了DFA，我们不能学啥东西单就事论事，不求扩展思维。大家除了Google一些生词术语外，还要进行照葫芦画瓢的步骤。想想，既然DFA属于“数学模型”，那就上升到一个更广的范畴。数学模型不是就事论事的，它是从一类事物中提取出的一种“模式”的集合。因此可以用来讨论所有具备这种模式的事物。好了 Mind Freeing Time: DFA在处理语言，确切地说是“形式语言”，更确切地说是“正则语言”。那么DFA的输入信号不仅仅只能为字符，甚至26个字母。在DFA中输入信号还可以是一部分字母的集合（可以叫单词）。只要输入集合有穷，输入状态有限，就符合DFA模型。这就是在离散数学中DFA的用法，用来处理各种有限的集合。那么怎么又跟正则语言这种“语言学”的东西拉呱上关系了呢？我们想象，当我们读到：“This is an apple. (这是个苹果)”。把我们的语言中枢想象成自动机：'t'、'h'、'i'、's'进来后我们大脑中立马从之前的状态形成一种新状态：“要说‘这个’东西了” “这个”是主语，至于‘这个’怎么了？没读后文之前，我们不知道。然后随着字符流入，字母被组建为单词，单词被组合称为语义。最终我们晓得了：这是个苹果。而不是个梨子——这也是终结状态。语言文字是媒介，到各个人的大脑中都会成为一种状态。有的时候各个人状态一至。而有时每个人理解不同——状态不一致。我跟不会中文的英国人说：“这是一个苹果。”他回答：“What'd hell are you talking about?” 这种现象的成因是该英国人大脑中并没有：“这是一个苹果。”这些中文字符 / 读音的词素表。换句话说人人都有个DFA，我们认识的符号都是有限的，通过勤奋学习可以扩充词汇表。那我们能不能用lex处理自然语言，做“智能机器女仆”的语言中枢呢？抱歉形式语言的处理与自然语言有别。关于语言学，AI，直至于“《自己动手制作机器女仆的语言中枢》”的更多方面，大家自行搜索了解一个人“诺姆 乔姆斯基”。如果还不能解决问题，你需要学习《认知神经科学》等，祝你成功！

Lex的.l文件。其实就是一条条正则表达式。能被该条正规式（就是正则表达式 / Regular expressions / 简称: Regexp）规则所识别的字符集合，变成一条条的词素信号。现在我们不讲怎样写.l文件。我们马上来说说话法分析技术。词法分析处理的语言叫做正则语言，使用的是DFA。正则语言的证明可以使用“泵引理”。语法分析处理的语言叫做上下文无关语言(context free language / CFL)，使用下推自动机(push-down automaton / PDA)处理。并且可以证明CFG与PDA等价。为什么又要换一种自动机模型了呢？因为DFA处理能力有限。最经典的一个例子来源于《自动机理论、语言和计算导论》一书。给你个不可能完成的任务写一个描述“回文”的正则表达式——所谓“回文”不是回族的文字，Madam im Adam. (亚当见到了夏娃时说的)、黄山落叶松叶落山黄、上海自来水来自海上、等。什么还没发觉什么是回文？把这些句子逐字倒过来读即可。（等。这句也是哦）利用泵引理立马可以证明“回文”形式不属于正则语言。这时我们要引入一种新的自动机模型——NFA（不确定型有穷自动机 / undeterministic Finite automaton）。我们来形式化定义一个NFA：一个有穷状态集合。一个有穷输入符号集合。一个属于有穷状态集合的初始状态。一个属于有穷状态集合的终结状态的集合。终结状态集合是状态集合的子集。一个转移函数。转移函数接受一个输入符号，返回“一个状态集合”，一个状态集合意味着可以是多个状态，也可以是一个状态。对于输入了空集（没有输入就可以返回状态）的转移函数构成的NFA，我们叫做带有空转移的NFA，简称 epsilon-NFA。（空的英文：Empty取单词首字母E对应希腊字母Epsilon‘就是那个极其张扬的e’）。在概念上DFA与NFA的区别在于DFA的转移函数返回确定的单一状态，NFA的转移函数返回状态集合。形象地解释DFA与NFA的区别：NFA可以同时处在若干个状态上，脚踏n条船。因此NFA具有对下一个状态进行“猜测”的能力。一个NFA读到“黄山落叶松”可以先假设这句话已经是回文了但是不巧在接下来读出“叶落山秃。”，进而假设被推翻，自动机不能到达：“这句话是回文”这样一种状态。NFA的所谓不确定性，不确定在转移函数返回的状态数量上。（注意不是状态类型总量。）

那么PDA又是什么鬼？PDA简而言之就是带了一个栈结构的epsilon——NFA。所谓带了一个栈结构的就是指在自动机运行过程中附加一个能够收纳状态的栈。允许进栈的状态们像手枪子弹一样被压入弹夹“栈”。取出时会依照从后到的往先到的子弹（状态）这样的顺序一一取出。好了不要以为你带了个栈我就认不出你是NFA了。你出去，把你的栈也带走，你就是个带栈的epsilon-NFA。关于PDA和上下文无关语言(CFL)等价的证明，我不想在这里提了，有兴趣的读者可以自行找资料了解。接下来是一个新概念上下文无关文法(CFG / context free grammar)。上下文无关文法的概念正是由“诺姆 乔姆斯基(Norm. Chomsky)”提出。起初乔姆斯基想用CFG描述自然语言，但是至今未得实现。但是呢这种文法是构建编译器语法分析器的关键理论。我们的yacc也就是bison使用类似巴克斯范式(BNF / Backus Normal Form)的东西来描述上下文无关文法。好了我们看，yacc编译器的源文件是“简化的巴克斯范式”，目标文件是一个c源文件。lex/flex的源文件是一堆正则表达式，目标文件也是一个c源文件。那么下面的重点是：BNF是怎么写的咯？

下面，我会通过一个表达式计算器的例子来告诉大家flex / bison的基本用法。

首先确定表达式计算器的功能：输入带有double数据类型的四则运算表达式计算出结果，四则运算必须带有优先级。附加要求能使用括号改变表达式的计算顺序。

接着我们分析带括号的四则运算表达式的形式：

0. *aa*

1. (*aa*)

2. *aa+bb*

3. *aa-bb*

4. *aa*bb*

5. *aa/bb*

6. 将上述规则按照四则运算优先级组合起来得到：

*aa *或者/ bb*，其中*aa*可以代表一个数字，*aa*可以等于*bb*，*aa*或者*bb*又可以拆解为

aa=cc +或者- dd，其中*cc*可以代表一个数字，*cc*可以等于*dd*，*cc*和*dd*又同时可以为上述规则构

成的子表达式。

所以，对于第2，3条的加减法表达式，因为加减优先级相同，则使用yacc语法写为：

```
exp : factor
    | exp ADD factor
    | exp SUB factor
    ;
```

那么使用自然语言描述上述内容则为：(exp) 表达式这个“非终结符 (nonterminal / nonterminal symbol)”由 (factor, 产生式) 推出。或者 (用管道符 “|” 表示) 由一个产生式加上 (ADD) 一个产生式推出。又或者由一个产生式减去 (SUB) 一个产生式推出。

这条BNF可以写作一行：exp : factor | exp ADD factor | exp SUB factor;

真正的BNF在冒号 “:” 处是一个 “:=” 符号读作“推出自……”，只不过yacc语法将 推出自 符号简化为冒号。

以此类推，乘法可以这么写：

```
factor : term
        | factor MUL term
        | factor DIV term
        ;
```

读作：产生式自 term 推出。产生式也可以由 产生式 乘以 (MUL) 项 (term) 推出。产生式还可以由 产生式 除以 (DIV) 项 (term) 推出。

这里说明几个问题：产生式 “factor” 解释为由一些项目推导出的一个式子。在形如 aa::=bb OP cc的BNF中推出符号左侧 (left-hand side / LHS) 叫做非终结符。而右侧 (RHS) 每一项叫做一个终结符。非终结符与终结符不能相同。设想这样 aa:=aa; aa由aa推出？那么aa是什么？无法想象，yacc报语法错误。

接着，将括号元素融入语法分析文件：

```
term : NUMBER
      | LB exp RB
      ;
```

读作：项 (term) 由 数字 (NUMBER) 终结符构成。项也可以由一个左括号 (LB) 加一个表达式，再加一个右括号 (RB) 构成。

接下来我们使用一个递归技术的BNF术将整个BNF首尾串联起来：

```
calclist : EOL
          | calclist exp EOL
          ;
```

以上BNF读作：计算列表 (calculation-list) 由空项 (什么表达式也不输入) 构成。也可以由计算列表和表达式加上一个行尾结束标记 (EOL / End of line) 构成。

进一步深入yacc的BNF。有了上下文无关文法的描述，yacc已经可以根据这种CFG生成语法分析器了。但是要让yacc计算表达式，我们还需要学会一个技巧：终结符，非终结符在yacc内部的表示方法。这个技巧大致是这样的：如果有 nt:tm1 ADD tm3; 这样的BNF。我们把该条BNF式子左侧的非终结符写做 \$\$, tm1从终结符得出的结果写做\$1。那么ADD终结符应该写做\$2, 但是ADD只代表了 + 这个记号，不会得出任何结果，所以我们不书写\$2。tm3写做\$3。(也许可以这样记忆：从左往右，最有钱的\$\$等于第一个有钱人\$1的资本加上第二个有钱人\$2的资本……加上第n个有钱人的资本；笑~) 接着我们需要在每条产生式的后边跟上非终结符是如何得出自己的值的计算表达式，这些表达式的语法就是C语言表达式语法，\$\$,\$1,\$3可以看作变量：

```
exp : factor
    | exp ADD factor { $$ = $1 + $3; }
    | exp SUB factor { $$ = $1 - $3; }
    ;
```

要把这些表达式C语言程序放在每条产生式后的花括号中。注意上边 factor 是没有计算过程的，因为不写表达式相当于让yacc默认了计算表达式为 \$\$ = \$1; 把factor的值传递给exp。

就像C语言一样，每个yacc的终结符都需要事先定义和声明它们的数据类型。

在yacc / bison中我们这样声明一个终结符标记：

```
%token EOL
```

还可以如此这般声明一堆终结符标记：

```
%token ADD SUB MUL DIV LB RB
```

至于每个终结符的数据类型，前面我们要求可以计算双精度浮点数 (就是精度很高的小数，double)。所以我们还要使用yacc独有的 %union语句 进一步声明一个double类型：

```
%union
{
    double Decimal;
}
```

然后这样声明 NUMBER，因为 NUMBER 就代表了数字：

```
%token <Decimal> NUMBER
```

将 calclist exp factor term subterm 的数据类型统称为 Decimal 的浮点数类型：

```
%type <Decimal> exp factor term subterm
```

关于BNF我们还得引进一个新术语：“二义性文法（Ambiguous Grammars）”，也许读者会想到我们为什么不把加减乘除一气呵成写成这样？：

```
exp : exp ADD exp
    / exp SUB exp
    / exp MUL exp
    / exp DIV exp
    / NUMBER
    ;
```

答案是，因为这样就会存在不能区分优先级以及文法二义性冲突的问题。yacc / bison使用一种名为：移进 / 规约（shift / reduce）的方法处理表达式。移进 / 规约的方式利用PDA，将当前输入符号压入栈的同时也改变自动机的状态。压入符号的过程叫做移进。当yacc的PDA遇到BNF右侧产生式描述出的所有规则时，弹出栈中所有符号，同时加以计算处理。这个过程就是“规约”。然后再以BNF左侧非终结符的方式找到另一条BNF右侧对应的终结符，重新压入栈中。yacc可以使用LALR(1)与GLR两种方式进行语法分析，LALR(1)速度快，效率高。但是对于有些冲突则鞭长莫及，因为LALR(1)意味着只能向后多读一个记号。关于yacc / bison的二义性文法问题以及什么是LALR(1)（Look Ahead Left to Right with a one-token lookahead，自底向上从左至右分析，超读一个记号）GLR（Generalized Left to Right，常规从左到右的分析方法），包括yacc的移进 / 规约冲突请检索参阅相关资料。

最后关于yacc的语法分析文件我还要说明一下整个文件的结构
语法分析文件.y文件通常如下：

/* 可选的老式的C语言注释作为的开头，说明文件信息。bison只认得这种注释而不认c语言的行注释“//，因为bison历史悠久啊 */

```
%{
#include <stdio.h>
#include <stdlib.h>
// 紧接着是以一对 %{, %} 符号所包裹的声明区域，用来包含C语言头文件和存放函数声明。
// 这些东西都会被yacc原封不动地放进生成的 xxx.tab.c 文件中。
}%
%union
{
//可选的%union语句声明非终结符 / 终结符所用到的数据类型，如果不声明yacc默认为int
}
%%
/* 上面是两个“百分号”标记，用来区分y文件的“声明段”结束了，接下来是bnf定义段。 */
/* 此处应有BNF。 */
%%
/* 上面又是两个“百分号”标记，指示：“BNF段”结束了。 */
/* 下面是一些bnf段内用到的C函数的定义，以及main函数的定义。 */
/* 完 */
```

接下来，我将calc.y文件整个展示在这里：

```
/* calc.y */
%{
#include <stdio.h>
}%
%union
{
    double Decimal;
}
%token <Decimal> NUMBER
%token ADD SUB MUL DIV
%token LB RB
%token EOL
%type <Decimal> exp factor term subterm
%%
calclist : EOL /* Do nothing . */
          | calclist exp EOL { printf("= %lf\n", $2); }
```

```

;

exp : factor { $$ = $1; }
    | exp ADD factor { $$ = $1 + $3; }
    | exp SUB factor { $$ = $1 - $3; }
;

factor : term { $$ = $1; /* Actually, $$ = $1; is the default way. So we don't need to
write. */ }
    | factor MUL term { $$ = $1 * $3; }
    | factor DIV term { $$ = $1 / $3; }
;

term : NUMBER { $$ = $1; }
    | LB exp RB { $$ = $2; }
;

%%

main(int argc, char ** argv)
{
    yyparse();
    return 0;
}

yyerror(char * s)
{
    fprintf(stderr, "error: %s\n", s);
    return 0;
}
```

main函数调用yacc默认函数yyparser开始语法分析过程。yyerror函数属于yacc的错误处理，就像C语言编译器在遇到没有分号结尾的表达式时会报：xx错误：xx行缺少分号。yyerror函数的实现过程，是将yacc生成的默认错误信息打印到标准错误流中（通常是显示到屏幕上）。

将文件保存为calc.y后，我们使用bison编译该文件：`bison -d calc.y`
bison命令给参 -d是因为我们需要在生成 calc.tab.c 的同时生成一个 calc.tab.h 这个头文件包含了y文件内所有token的值。有点类似于Win32编程中资源文件都有一个声明的头文件，这个头文件描述了所有资源的ID，最后被MS资源编译器rc所编译。而yacc生成的标记（token）ID用来与lex交互。预知如何交互，请看下文。

至此我们得到了两个c文件：calc.tab.h 和 calc.tab.c。这两个文件就是yacc帮我们写好的语法分析器啦。有兴趣读者可以打开这两个文件，看看内容和行数。不过，至于它们的内容基本原理我已经在上文叙述，至于yacc的实现不在我们的研究范畴。此时，我们注意。我们仅仅实现了一个语法分析器。而根据上文所讲，一个编译器前端至少还需要一个词法分析器。下面我就带大家写lex / flex的.l文件。

首先，因为词法分析文件基本上就是若干条正规式（也叫正则表达式）我将会简单介绍Unix类型的正规式语法。

.	代替任意单一字符，除了换行符“\n”。
[]	代替任意出现在方括号内的单一字符，比如 [0123456789]代替0-9所有字符新版本的bison可以进一步简写为[0-9]注意0-9意味着0的ASCII码一定小于9的ASCII码，flex也只能处理ASCII，ANSI编码的文件。flex无法处理中文汉字。
^	代替行首出现的第一个字符。
\$	代替行尾出现的最后一个字符。
{}	如果花括号内出现一到两个数字，就表示匹配{前一个字符最小几次最多几次。比如A{1,3}匹配最小出现一次A最多出现3次A。再比如B{5}匹配BBBBB。
\	反斜杠就是C语言的escape符号：\n \r \10。
*	匹配之前的字符零次或多次。
+	匹配之前的字符一次或多次（一次以上）。

?	匹配之前出现的模式零次或一次。
	选择运算 A B 表示出现A就匹配A出现B就匹配B，除此以外啥都不匹配。
"....."	任何使用引号标记的东西都会被lex当成一个词素直接返回。
()	正则表达式的优先运算。(AB)C D 匹配 “ABC” 和 “ABD”

还有一个“\”反斜杠运算符叫做“（匹配尾巴）Trailing context” 因为使用较少比较特殊这里不做介绍，请自行查阅资料。

作为表达式计算器的词法分析器，我们只要识别出数字 +-*/ 括号 即可。lex词法分析文件的结构如下：

```
/* 和bison文件结构相同可选的老式的C语言注释作为的开头，说明文件信息。flex只认得这种注释而不认c语言的行注释
//，因为flex也历史悠久啊 */
%{
#include "xxx.tab.h"
// 紧接着是以一对 %{, %} 符号所包裹的声明区域，用来包含C语言头文件和存放函数声明。
// 通常，这里存放的就是 bison 使用-d参数同时生成出来的头文件。这些头文件里都是词素记号ID的宏。
// %} 后是两个百分号，说明声明段结束。后面紧跟的正规式段定义返回各个词素的正规式。
}%
%%
正规式 { return 对应词素ID; }
以上模式出现若干条.....
%%
正规式定义结束，后面可以是一些C语言函数。
```

同样，我放出整个表达式计算器的词法分析文件内容：

```
/* calc.l */
%{
#include <stdlib.h>
#include "calc.tab.h"
}%
%%
"+"      { return ADD; }
"-"      { return SUB; }
"*"      { return MUL; }
"/"      { return DIV; }
"("      { return LB; }
")"      { return RB; }
[0-9]+ "." [0-9]* |
"." [0-9]+ |
[0-9]+E[+-]?[0-9]+ |
[0-9]+ "." [0-9]*E[+-]?[0-9]+ |
"." [0-9]+E[+-]?[0-9]+ { yylval.Decimal = atof(yytext); return NUMBER; }
\n       { return EOL; }
[ \t]    { /* Ignore spaces and tabs. */ }
.        { printf("Mystery charater %c\n", *yytext); }
%%
```

合并解释一下flex与bison关于词法分析文件calc.l与语法分析文件calc.y的处理：函数main被调用后，用户输入表达式并按下回车，表达式进入C语言的标准输入流。表达式的每个字母进入lex生成的词法分析器，遇到+*/ 对应的词素会被当成记号返回给语法分析器。当处理到一个浮点数的时候，lex将之前读到的字符串形式的浮点数通过atof函数转

换为double类型，并且存储在yacc结构体的双精度浮点数变量Decimal中。如果遇到abc这样没设计好的词素，lex则会打印“Mystery character (蜜汁字符) xx”。然后各个词素以记号的形式送入yacc。yacc通过移进 / 规约的手段解逐步析表达式，遇到 ADD SUB 等照BNF分析。遇到不能分析的例如 +*64 通过yyerror函数打印“syntax error (语法错误)”信息并且yyerror返回，主函数返回0。程序结束。正常分析时，遇到NUMBER终结符，或者具有同Decimal一致类型的非终结符，移进处理时将存放于Decimal中的值压栈。规约处理时将栈中的Decimal类型相同的值弹出。这就是flex与bison协同工作的大致原理。

好了，我们在Unix环境下使用 `flex calc.l` 命令编译词法分析文件。

如果没有错误，flex生成一个名为lex.yy.c的C语言源文件。它就是我们的语法分析器啦。

最后使用C语言编译器编译bison与yacc生成的所有文件：`cc -o calc calc.tab.c lex.yy.c -ll`

注意：flex / yacc自带一个库，用C编译器编译yacc与lex生成的C语言源文件时必须加上这个库。在OSX / macOS系统下使用gcc的-l参数，在其他类Unix / Linux环境中请使用-lfl参数。

如果在类Unix平台下，现在你得到了一个可执行的目标文件名叫calc。使用 `./calc` 命令运行之，输入 `2*(3.14159+3)`回车，看看效果吧！

这个简单的计算器制作完成之后，大家可以对它加以改进。比如可以让它支持负数，给它添加更多功能的运算，幂、三角函数……。有能力的还可以给它加上流程控制语句（IF THEN ELSE DO WHILE……）。通过不断地改进表达式计算器，可以进一步了解编译原理相关知识哟！

扩展阅读：

参考书籍1.《Flex & Bison》作者 John R. Levine。OREILLY出版。有中文译版，名叫《Flex与Bison》。

参考书籍2.《Automata Theory Languages, and Computation (3rd Edition)》作者 John E. Hopcroft, Rajeev Motwani, Jeffery D. Ullman。机械工业出版社出版了中文版 名为：《自动机理论，语言和计算导论》。还有一本印度人写的书名相同的书，值得一读。但是我忘了作者的具体姓名了。个人感觉前者思维严谨，条理清晰，适合作为教材。印度人的书题材新颖一点，文字活泼，思维更活跃，同时思维严谨知识介绍全面（除了巴克斯范式还提到了格雷巴赫范式）。当然还有一本日本人写的名为《自己动手写编译器》？日本人写的在思维、用词方面更没话说，更适宜东方学习者的思维理解，但是内容不够严谨。知识介绍不够全面。


参考书籍3. 编译器设计三大经典：龙、虎、鲸书。这个太有名了，搜索一下就有一大堆。本站就有电子版资源。

Windows平台下配置flex与bison参考：

众所周知GNU开发的flex与bison在类Unix / GNU Linux平台上使用起来如鱼得水，但是在Windows平台下就不那么友好了。Windows下使用flex与bison方法大致有二：

一、安装Cygwin。Cygwin是Windows上的Linux环境模拟。安装Cygwin时可以选择添加gcc和flex/bison到Cygwin环境中。Cygwin官网：<http://cygwin.com>

二、在VisualStudio中使用Win Flex-Bison。Win Flex-Bison的下载地址：
<https://sourceforge.net/projects/winflexbison/>这里是Win Flex-Bison详细的安装配置说明：
<https://sourceforge.net/p/winflexbison/wiki/Visual%20Studio%20custom%20build%20rules/>

 编译原理, lex, yacc, flex, bison

分享到： QQ好友和群



相关帖子

- 【编译原理】调用约定及其意义
- 【破解生物密码】—解密基因及生物学工具Glab的原理
- 【VB6】使用VB6写“标准DLL”
- 【C】认识GCC的链接时间优化
- 【汇编】汇编写的SpVoice的Demo，COM类的调用例子
- 【C】++
- 【嵌入式】搭建一个基于gcc编译器的STM32F10x程序编译环境

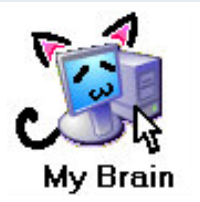
SIGNATURE

In the beginning I was not the best.
And the world was also not the best.
But I still know that I am who I am.
Because I think that it is good.
I have been working hard.
I have been keeping growth with the world.
And it was so.

 回复

使用道具  举报

cyyc0ish



85

260

3681

主题

帖子

积分

用户组: 管理员

No. 418

UID

418

精华

13

威望

52 点

宅币

1969 个

贡献

1281 次

宅之契约


0 份


在线时间

252 小时

注册时间

2014-8-9

 收听TA

 发消息

 楼主

发表于 2016-11-15 01:36:18

 只看该作者

关于语法分析除了构建AST还有其他手段：比如处理逆波兰式（后缀表达式）：<https://www.0xaa55.com/forum.php...>表达式计算>

或者直接用递归处理：

```
[mw_shl_code=c, true]
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#define MAXLEN (100)

float CatchFirstNumber(void);
float CalcBrackets(void);
float CalcAdditional(void);
float CalcAdvanced(void);
float CalcPrimary(void);

char* expr;

float CalcPrimary(void)
{
    float x, y;
    char c;
    x = CalcAdvanced();
    c = expr[0];
    while (c == '+' || c == '-')
    {
        expr++;
        y = CalcAdvanced();
        if (c == '+')
            x += y;
        else if (c == '-')
            x -= y;
        c = expr[0];
    }
    if (expr[0] == ')')
        expr++;
    return x;
}

float CalcAdvanced(void)
{
    float x, y;
    char c;
    x = CalcAdditional();
    c = expr[0];
    while (c == '*' || c == '/')
    {
        expr++;
        y = CalcAdvanced();
        if (c == '*')
            x *= y;
        else if (c == '/')
            x /= y;
        c = expr[0];
    }
    return x;
}
```

```
float CalcAddtional(void)
{
    float x, y;
    x = CalcBrackets();
    if (expr[0] == '^')
    {
        expr++;
        y = CalcAddtional();
        x = powf(x, y);
    }
    return x;
}

float CalcBrackets(void)
{
    if (expr[0] == '(')
    {
        expr++;
        return CalcPrimary();
    }
    else
    {
        return CatchFirstNumber();
    }
}

float CatchFirstNumber(void)
{
    char szTar[MAXLEN] = {0};
    char szT[2] = {0};
    for (;expr[0] != 0; expr++)
    {
        if (expr[0] >= 48 && expr[0] <= 57 || expr[0] == 46)
        {
            szT[0] = expr[0];
            strcat(szTar, szT);
        }
        else
            return atof(szTar);
    }
    return atof(szTar);
}

int main(int argc, char** argv)
{
    char szExpr[MAXLEN] = {0};
    expr = szExpr;
    while (scanf("%s", expr) != EOF)
    {
        printf("%f\n", CalcPrimary());
    }
    return 0;
}[/mw_shl_code]
```

SIGNATURE

In the beginning I was not the best.
And the world was also not the best.
But I still know that I am who I am.
Because I think that it is good.
I have been working hard.
I have been keeping growth with the world.
And it was so.

板凳



题外话，Windows下实用工具相当丰富，除了众所周知的 `debug.exe` 还有 `ntsd.exe` (NT符号调试器) 这俩后来在win64下都被砍去了。和 `findstr` 类似的还有 `find.exe` 后者不支持正规式。还有一个工具叫做 `fc.exe` 用来比较文件。`help fc` 命令可以查看 `fc` 的用法。`fc`不仅可以执行文本比较，还能执行二进制比较。大爱`fc`！

In the beginning I was not the best.
And the world was also not the best.
But I still know that I am who I am.
Because I think that it is good.
I have been working hard.
I have been keeping growth with the world.
And it was so.

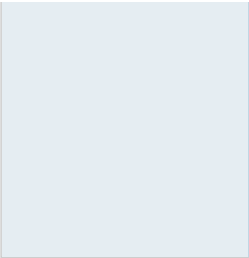
地板



在关系规范化的过程中 满足2NF需要实现没有部分函数依赖，达到3NF需要关系中没有传递条件。而BCNF除了满足2NF，3NF的条件外只要使得关系 $X \rightarrow Y$ ，X里边全都是码就可以了。因为全都是码的关系好找啊！所以我兴奋不已！称BCNF是“白痴NF”。

In the beginning I was not the best.
And the world was also not the best.
But I still know that I am who I am.
Because I think that it is good.
I have been working hard.
I have been keeping growth with the world.
And it was so.

[◀ 返回列表](#)



您需要登录后才可以回帖 登录 | 立即注册→加入我们  用QQ帐号登录

[发表回复](#) ☐ 回帖并转播 ☐ 回帖后跳转到最后一页

[本版积分规则](#)