动态分析之力：自适应$(Adaptive)LL(*)$解析算法

摘要

尽管如PEG，$LL(*)$，GLR和GLL等现代解析策略取得了不小的进步，但解析本身并不是一个完全被解决了的问题。现有的方式存在许多缺陷，如：难以支持增强式嵌入动作(action)，性能低下或者说不可预测的解析速度，以及有悖直觉的解析策略。本文介绍的$ALL(*)$解析策略结合了传统的自上而下$LL(k)$解析器的简单、高效、可预测能力，以及类GLR机制的强大解析决策功能。其关键的创新之处在于将语法分析推迟到了解析运行时，这使得$ALL(*)$能够处理任何非左递归的上下文无关文法。理论上$ALL(*)$的解析时间复杂度是$O(n^4)$，但在实际的文法分析中却始终呈线性复杂度，性能比GLL和GLR等一般的解析策略高出好几个数量级。ANTLR4生成的$ALL(*)$解析器，可以通过重写文法来支持直接左递归。ANTLR4的广泛使用（2013 年的下载量为 5000 次/月）表明，$ALL(*)$对各种应用程序都卓有成效。

1. 导论

尽管现代语法解析策略已经非常先进成熟，并且在学术研究上历史悠久，但在实践中，计算机语言的解析仍然未能被彻底解决。迫使程序员修改文法以匹配确定性（注：确定性一词，指的是确定性有限自动机DFA，与之对应的时非确定性有限自动机NFA）$LALR(k)$或$LL(k)$解析器生成器的规范来提升解析性能，在硬件资源稀缺时是无可厚非的。但随着硬件资源成本的不断降低，研究人员开发出了功能更加强大、但解析成本更高的非确定性解析策略，这些策略遵循"自下而上"（LR风格）或"自上而下"（LL风格）的解析方式。这些策略包括GLR，解析器表达式语法(PEG Parser Expression Grammar)，来自ANTLR 3的$LL(*)$以及最近提出的一种全通用自上而下的GLL解析策略。

虽然这些策略在使用上比$LALR(k)$和$LL(k)$解析器生成器更加便捷，但它们也存在各种短板。

首先，非确定性解析器有时会出现未预料的行为。由于GLL和GLR是专为处理自然语言语法设计的，而通常情况下，这种语法会潜在地包含二义性，故GLL和GLR会为存在二义性的语法返回多种解析树（森林）。对于计算机语言来说，二义性几乎就是一种异常。当然为处理这种不常见的情况，我们可以对构建出的解析森林进行遍历来消除此等二义性，但这往往意味着需要消耗额外的时间、空间和硬件资源。根据语法定义，PEG是一种无二义性的文法，但存在一种特殊情况，即规则$A \rightarrow a \mid ab$（表示非终结符A要么匹配a要么匹配ab）永远无法匹配ab，因为只要输入的前缀匹配a，PEG文法将会选择第一种产生式。存在嵌套的回溯时，调试PEG将变得举步维艰。

其次， 像打印语句这种由程序员提供的能够对解析过程产生影响的动作(actions)或者是修改器(mutators)都应当在持续预测（PEG）或支持多种解释（GLL和GLR）的解析策略中避免，因为这些动作或修改器可能并不应当执行（虽然DParser支持'final'动作，即如果程序员确定某次归约是一个二义性语法的最终产物之一，那么这个动作或者修改器将得到执行）。在没有副作用的情况下，可能触发的动作必须在某个不可变的数据结构被缓存下来，或者是解析本身提供撤销操作。前一种机制受到内存大小的限制，而后一种机制实现则相对复杂并且有可能无法实现。而避免上述副作用的一种典型解决方式是构建一个解析树，用于解析后（post-parse）处理，但这种刻意产物从根本上限制了对输入大小的限制，因为要将整个输入的解析树放入内存中（译者注：要将整个输入整成一个解析树放内存里，可不得要求输入尽量小点嘛）。构建解析树的解析器是无法为大文件输入或者是无边界流数据提供解析能力，除非可以将这些数据按照逻辑分块(logical chunks)进行处理。

最后，我们的实验结果（第 7 节）表明，GLL和GLR在时间和空间上都性能低下，并且无法预测。它们的复杂度分别为$O(n^3)$和$O(n^{p+1})$，其中p是语法中最长的产生式长度（GLR的经典复杂度为$O(n^3)$，那是因为Kipps给出的这个算法具有一个高得不合理的常数）。理论上，通用解析器应当在处理确定性语法时具有线性($O(n)$)的时间复杂度。但在实践中，我们发现GLL和GLR在12920个Java6的库源代码文件

（约123M）的解析上，要比$ALL(*)$慢大约135倍，而在一个3.2M大小的Java源文件解析上，甚至要比$ALL(*)$慢6个数量级。

$LL(*)$文法通过提供一种确定性的解析策略来解决上述的短板，该解析策略使用正则表达式（表现为一种确定性有限自动机DFA）来处理剩下的输入，而不是像$LL(k)$那样的固定k长度序列。即使前看序列（一个所有可能的剩余输入短语集）通常是上下文无关的，但使用正则的前看语法DFA用作前看策略便能够将$LL(*)$的决策进行限制，并区分出不同的产生式。但$LL(*)$语法存在一个主要的问题，它在静态阶段是无法进行文法判定的，并且有时候也无法找到能够区分不同产生式的正则表达式。ANTLR 3的静态分析能够检测并避免潜在的不可判定情况，并将这种情况回退到回溯解析策略。这使得$LL(*)$具有与PEG解析类似$A \rightarrow a \mid b$语法时相同的窘境。回溯解析策略也无法检测如$A \rightarrow \alpha \mid \alpha$这样明显的二义性错误，因为回溯会始终选择第一个产生式进行匹配，故其中的文法符号序列α就使得$\alpha \mid \alpha$不再是$LL(*)$规范的了。

1.1 动态语法分析

在本文中，我们将介绍自适应（Adaptive）$LL(*)$文法，简称$ALL(*)$文法，这种解析器将自顶向下解析器具有的简单易实现和类GLR解析器具有的一些强大机制进行结合，并在解析时做出决策。具体来说，LL解析会在每个预测决策点（非终结符）挂起，然后会在预测机制选择了合适的产生式去展开的时候恢复。而关键的创新在于将语法分析转移到解析时；不再需要静态的语法分析。这种方式可以使我们避免在静态$LL(*)$语法分析的不可判定性，并且让我们能够为任何非左递归的上下文无关文法(Context Free Grammar，CFG)生成正确的解析器（定理6.1）。相比于静态分析需要考虑所有可能得输入序列，动态分析只需要考虑当前实际输入序列的有限结合。

$ALL(*)$预测机制背后的理念是在每个备选产生式的决策点启动(launch)一个子解析器。子解析器以一种假并行的方式去展开所有可能的路径。当子解析器展开的某条路径和剩余输入序列不匹配时，子解析器就会被销毁。子解析器之间步调一致的向前读取输入序列，这样分析过程就能以最小的前看深度，识别出唯一匹配的产生式，这样就具备了唯一预测产生式的能力。当然，如果多个子解析器在最后碰面或者一起达到输入序列末尾，预测器就会告知一个二义性，并根据存活子解析器相关的产生式编号的最小值（即最高优先级）来解决这个二义性（产生式编号代表优先级，是类PEG解析器默认解决二义性的一种方式；Bison解析器还可以通过自选产生式来解决冲突）。当然，程序员可以嵌入语义谓词（semantic predicates）来在具有二义性的产生式之间做出选择。

$ALL(*)$解析器能够对分析结果做缓存，以增量和动态的方式为一个DFA做映射缓存，即前看短语序列到预测产生式的映射（在这里我们使用分析($analysis$)一词的原因在于$ALL(*)$的分析也能够像$LL(*)$分析($analysis$)时那样，产生前看DFA）。解析器可以通过查询缓存，快速地在同一个解析器决策和前看短语中做出后面的解析预测。而未命中缓存的输入短语将会触发语法分析机制，并同时预测一个备选产生式然后更新DFA到缓存中。尽管在给定决策下的前看语言通常是上下文无关的，但DFA仍然很适合用来记录预测结果。动态分析只需要考虑在解析时遇到的有限上下文无关文法子集，并且任意的有限子集都是有规律可循的。

为避免非确定性子解析器具有的指数复杂度这一性质，预测采用了一种叫图形化结构堆栈GSS($graph-structured\ stack$)的数据结构来避免冗余计算。GLR使用的策略与此基本相同，只不过$ALL(*)$只预测使用了这种策略的子解析器的产生式，而GLR直接使用这种方式进行解析。因此，GLR必须将非终结符压入GSS中，而$ALL(*)$则不需要。

$ALL(*)$解析器以LL的简易性来处理终结符的匹配和非终结符的展开任务，但其理论的时间复杂度为$O(n^4)$，因为在最坏情况下，解析器必须对每个输入符号进行预测，而每次预测都必须检查整个剩余输入。检查一个 输入符号的时间复杂度为$O(n^2)$。$O(n^4)$的时间复杂度和GLR一致。不过在第7节中，我们通过实测证明，常见语言的$ALL(*)$解析器是十分高效的，并在实践中表现出线性的时间复杂度。

$ALL(*)$的优势在于将语法分析推迟到解析时，但这一选择给语法的功能测试带来了额外的负担。与所有动态解析方式一样，程序员必须覆盖尽可能多的语法定义和输入序列组合，以找到语法歧义（二义性）。标准的源代码覆盖工具可以帮助程序员测量$ALL(*)$解析器的语法检查覆盖情况。生成代码中的语法覆盖率有多高语法检查覆盖率就有多高。

$ALL(*)$算法是ANTLR 4解析器生成器的基础（ANTLR 3基于$LL(*)$）。ANTLR 4于2013年1月发布，每月下载量约为5000次（从web服务日志中过滤出人为下载痕迹，并使用唯一IP地址来计算包括源代码、二进制文件或ANTLRworks2 开发环境的下载次数）。这些活跃的数据恰恰证明$ALL(*)$的用途广发和可用性。

本文的其余部分安排如下。首先，我们简要介绍ANTLR 4解析器生成器（第 2 节），并对$ALL(*)$解析策略（第 3 节）进行详细讨论。接下来，我们将定义谓词文法$(predicated\ grammar)$、以及其增强转换网络$(ATN\ augmented\ transition\ network)$表示法和前看DFA（第 4 节）。然后，我们将描述$ALL(*)$语法分析并介绍其解析算法本身（第 5 节）。最后，我们将证明$ALL(*)$的正确性（第 6 节）和性能（第 7 节），以及其相关检测工作（第 8 节）。附录A提供了$ALL(*)$定理的关键证明，附录B讨论了其语法算法，附录C将提供消除左递归的细节。

2. ANTLR 4

ANTLR 4能接收任何不包含直接或间接左递归（间接左递归通过另一条规则调用自身，如：$A \to B, B \to A$，另外，但一个空产生式出现时，也会出现左递归，如：$A \to BA, \to \epsilon$）的上下文无关文法输入序列。ANTLR 4会根据编写的语法，生成一个使用$ALL(*)$产生式预测函数的递归下降解析器。ANTLR目前具有Java或者C#解析生成器。ANTLR 4的语法类似$yacc$，使用带BNF(EBNF)范式操作符，如克林闭包(Kleene start(*))和单引号中的词法单元字面量。为方便起见，ANTLR 4的语法同时包含了词法规则和语法规则这样的组合规范。通过使用独立的字符作为输入符号，ANTLR 4的语法可以是弱扫描并且是可组合的，因为$ALL(*)$语法在联合的情况下是封闭的（定理6.2），这种特性有助于Grimm所描述的模块特性（此后，我们将ANTLR 4简称为ANTLR，早期版本我们会额外标注）。

程序员们可以在语法中嵌入用解析器宿主语言（译者注：如使用的是Java的ANTLR 4，就用Java语法）编写的动作（actions，或者成为修改器(mutators)）。这些动作可以获取到当前解析器的状态。不过解析器会在预测的过程中忽略这些动作（修改器），从而尽力避免"踩雷"。通常来说，这些动作的主要目的是从输入流中提取信息然后构建一些数据结构。

ANTLR还支持语义谓词$(semantic\ predicates)$，这些谓词使用解析器宿主语法编写的不具备副作用的产生布尔值的表达式，用于确定某个特定产生式的语义可达性。如果在评估过程中，一个语义谓词产出为假。那么解析器就会标记这些产生式不可达，从而在解析时改变由编写语法生成的语言[1]。谓词可以检测解析栈以及相关的上下文，来提供一种非标准的上下文有关文法的解析能力，从而大大增强了解析策略的能力。基于先前检测的信息，语义动作和谓词共同作用于解析器，改变整个解析。举个例子，一个C语言语法可以嵌入动作，以便在复杂构造中定义类型符号，如`typedef int i32;`可以在随后的定义中，将类型名称和其他的标识符区分开来，如`i32 x;`。

2.1 语法示例

以下用了一个简单的语法片段展示ANTLR的类yacc元语言语法：一个以分号结尾的赋值表达式语句

> 译者注：小写开头的表示语法规则，大写的表示词法规则

```
gramma Ex;  // generates class ExParser
// action defines ExParser member: enum_is_keyworkd
@members {boolean enum_is_keyword = true;}
stat : expr '=' expr ';'    // production 1     产生式1
     | expr ';'             // production 2     产生式2
     ;
```

```
expr : expr '*' expr
     | expr '+' expr
     | expr '(' expr ')'    // f(x)  函数
     | id
     ;

id   : ID | {!enum_is_keyword}? 'enum' ;
ID   : [A-Za-z]+ ;  // match id with upper, lowercase
WS   : [ \t\r\n]+ -> skip ; // ignore whitespace
```

代码1. 一个左递归的ANTLR4谓词语法Ex示例

在上面的示例中，有两个特征使得这个语法示例不符合$LL(*)$文法（因此ANTLR3无法识别上述语法定义）。首先，语法规则`expr`是左递归的，ANTLR4会自动重写该语法规则，使其成为非左递归和非二义性的，详见2.2节；其次，`stat`的两个备选产生式都具有一个共同的递归前缀（`expr`），这足以让`stat`从$LL(*)$的角度变得不可判断。ANTLR3会在运行时检测到产生式左侧的递归，然后回退到回溯法进行决策。

词法规则`id`中的谓词`{!enum_is_keyword}?`能够在解析预测的时候根据此谓词允许或禁止`enum`是否能作为一个标识符（译者注：在某些语言中enum是保留关键字，上述的语法定义就表示词法规则id不能匹配enum这样的词法单元）。当谓词为false时，解析器只会将`id : ID ;`视为`id`而不会匹配`enum`输入序列，词法会将`enum`作为`ID`以外的单独词法单元进行匹配。当然，这个例子仅仅用来展示谓词是怎样作用于语法，从而能够描述同一语言的子集或变体。

2.2 删除左递归

$ALL(*)$解析策略本身不支持左递归，但是ANTLR在生成解析器之前通过语法重写来支持直接左递归。直接左递归涵盖了大多数语法情况，例如我们上面提到的`id`这样的语法规则以及C语言里的声明符。我们从工程设计的角度不支持间接左递归或者隐藏左递归，因为这些形式并不常见，并且消除这样的左递归后，会导致转换的语法呈指数级增长。例如，C11的语法规范包含了大量的直接左递归但却没有间接或隐藏左递归，详见附录2.2。

2.3 使用$ALL(*)$进行词法分析

ANTLR使用$ALL(*)$的一种变体来进行词法分析，并且能够对词法单元进行完整匹配，而不是像$ALL(*)$语法解析器那样去预测产生式。在预热之后，词法解析器将会静态地构建一个基于正则表达式的类似DFA。而关键的区别在于$ALL(*)$词法是谓词化的上下文无关文法，而不仅仅是正则表达式，因此可以识别上下文无关的词法单元比如嵌套的注释语法，并且能够根据语义的上下文来控制词法单元的吞吐。这种设计之所以可行，是因为$ALL(*)$在处理词法和语法时足够快。

$ALL(*)$也适用于弱扫描的解析方式，因为它具有强大的识别能力，这在处理对上下文敏感的词法问题（如合并C语言和SQL语言）时非常有用。这种合并没有明确的词法哨兵(界限)来划分词法区域：

```
int next = select ID from users where name='Raj'+1;
int from = 1, select = 2;
int x = select * from;
```

概念证明请参见 [19] 中的语法代码`/extras/CSQL`。

3. $ALL(*)$简介

在本节中，我们将解释$ALL(*)$解析背后的思想和直观感受。然后在第5节中将更加正式地介绍该算法。

1. ANTLR以前的版本支持语法谓词($syntactic\ predicates$)，以消除能够导致静态语法分析失败的二义性；但由于$ALL(*)$具有动态解析的能力，故ANTLR4不需要这一功能 ↩