中国科学技术大学
University of Science and Technology of China

语法分析 III

《编译原理和技术》

张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

中国科学技术大学
University of Science and Technology of China

ANTLR 解析原理

自上而下分析器的自动生成

张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

解析器的生成器

中国科学技术大学
University of Science and Technology of China

- 生成器
  - 生成Lexer：Flex（for windows）、Jflex
  - 生成Parser
    - LALR: Bison（for windows）、Java CUP
    - LL: JavaCC、ANTLR – LL(*)[PLDI2011], ALL(*)[OOPSLA2014]
- 文法对Parser的影响
  - LR Parser的优势：速度快、表达能力强
  - LL Parser的优势：代码结构与文法对应，易理解，容易增加错误处理和错误恢复

ANTLR(ANother Tool for Language Recognition)

[http://www.antlr.org/] [https://github.com/antlr/antlr4/blob/master/doc/index.md]

- **Prof. Terence Parr，since 1989**
- **支持多种代码生成目标**

  Java、C++、C#、Python 2|3、Go、JavaScript、Swift、PHP、Dart

基于ANTLR开展的实验

- 开源：源码阅读，消化、理解生成器基于的原理
- 各种语言实现：31种模式

1. 从文法到递归下降识别器
2. LL(1)递归下降词法分析器
3. LL(1)递归下降语法解析器
4. LL(k)递归下降语法解析器
5. 回溯解析器
6. 记忆解析器
7. 谓词解析器
8. 解析树（分析树）
9. 同型抽象语法树
10. 规范化异型AST
11. 不规则的异型AST
12. 内嵌遍历器
13. 外部访问者
14. 文法访问者
15. 模式匹配者（子树）
16. 单作用域符号表
17. 嵌套作用域符号表
18. 数据聚集的符号表
19. 类的符号表型
20. 计算表达式的类型
21. 自动类型提升
22. 静态类型检查
23. 动态类型检查
24. 语法制导的解释器
25. 基于树的解释器
26. 字节码汇编器
27. 栈式解释器
28. 寄存器解释器
29. 语法制导的翻译器
30. 基于规则的翻译器
31. 模型驱动的转换

相关资源

ANTLR (ANother Tool for Language Recognition)

https://www.antlr.org/ v4.9.2 (Mar. 12, 2021), v4.11.1 (Sept. 4, 2022)

- doc, github, grammars
- Book

  The Definitive ANTLR 4 Reference (Source Code), P2.0 Sept. 16, 2014
- Video
  - The Definitive ANTLR 4 Reference Jan 3, 2013
  - ANTLR v4 with Terence Parr Feb 14, 2013
- Paper
  - LL(*)[PLDI2011]: ANTLR3的原理
  - ALL(*)[OOPSLA2014] : ANTLR4的原理

中国科学技术大学
University of Science and Technology of China

ANTLR4 简介

ANTLR 的文法文件 .g4

https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md

格式

```
grammar MyG;
options { … }
import … ;
tokens { … }
channels {…} //lexer only
@actionName { … }
ruleName : <stuff> ;
……
```

```
channels {
    WHILESPACE_CHANNEL,
    COMMENTS_CHANNEL
}
```

模式定义
词法状态

ruleName

- 词法：大写字母开头
- 语法：小写字母开头

纯词法分析器

```
lexer grammar MyG;
```

词法规则

```
INT : DIGIT+ ;
fragment DIGIT : [0-9] ;
LQUOTE : '"' -> more, mode(STR) ;
mode STR;
STRING : '"' -> mode(DEFAULT_MODE);
```

正规定义，
DIGIT不是记号

模式调用

用在词法规则中，设置当前分析的通道，
可跳过空白符或注释

```
WS: [\t\n\r]+ -> channel(WHITESPACE);
```

缺省的通道为Token.DEFAULT_CHANNEL

https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

ANTLR：Lexer命令

中国科学技术大学
University of Science and Technology of China

- 命令格式

  TokenName : <选项> -> 命令名 [(参数)]

- 命令
  - skip：不返回记号给parser，如识别出空白符或注释
  - type(T)：设置当前记号的类型
  - channel(C)：设置当前记号的通道，缺省为Token.DEFAULT_CHANNEL(值为0)；Token.HIDDEN_CHANNEL(值为1)
  - mode(M)：匹配当前记号后，切换到模式M
  - pushMode(M)：与mode(M)类似，但将当前模式入栈
  - popMode：从模式栈弹出模式，使之成为当前模式

https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

Parser规则

格式

- 可以带标签(#标签名，后跟空格或换行)

```
e : e '*' e # Mult  | e '+' e # Add  | INT    # Int ;
```

ANTLR为每个标签产生规则上下文类 XXX Parser.MultContext

- 有何用处？

ANTLR会生成与该标签对应的语法结构的 enter和exit方法

```
public interface XXXListener extends ParseTreeListener {
    void enterMult(XXXParser.MultContext ctx);
    void exitMult(XXXParser.MultContext ctx);
    ……
}
```

XXX为用户设置的文法名称

https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md

Parser规则

带内嵌动作的规则：可以引用上下文对象

```
grammar Expr;

@header {
package tools;
import java.util.*;
}

@parser::members {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    int eval(int left, int op, int right) {
        switch ( op ) {
            case MUL : return left * right;
            case DIV : return left / right;
            case ADD : return left + right;
            case SUB : return left - right;
        }
        return 0;
    }
}
```

Java代码，直接复制到生成的分析器源码的头部

自定义解析器类的成员
- ✓ 数据成员memory
- ✓ 方法成员eval

Parser规则

带内嵌动作的规则：可以引用上下文对象

```
stat:   e NEWLINE          {System.out.println($e.v);}
    |   ID '=' e NEWLINE    {memory.put($ID.text, $e.v);}
    |   NEWLINE
    ;
e returns [int v]
    : a=e op=('*'|'/') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | a=e op=('+'|'-') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | INT                   {$v = $INT.int;}
    | ID
      {
      String id = $ID.text;
      $v = memory.containsKey(id) ? memory.get(id) : 0;
      }
    | '(' e ')'             {$v = $e.v;}
    ;
MUL : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

Parser规则

左递归

- ANTLR容许哪些左递归？直接左递归
- ANTLR对所支持的左递归如何处理？下例会怎样？

```
e : e '+' e # Add | e '*' e # Mult | INT # Int ;
```

规则的组成元素

- T、‘literal’：终结符、文本串 => 记号
- r：小写字母开头，代表非终结符
- r[参数]：传入一组逗号分隔的参数，相当于函数调用
- {动作}：在之前的元素之后、后继元素之前执行该动作
- {谓词}?：如果谓词为假，则不继续分析

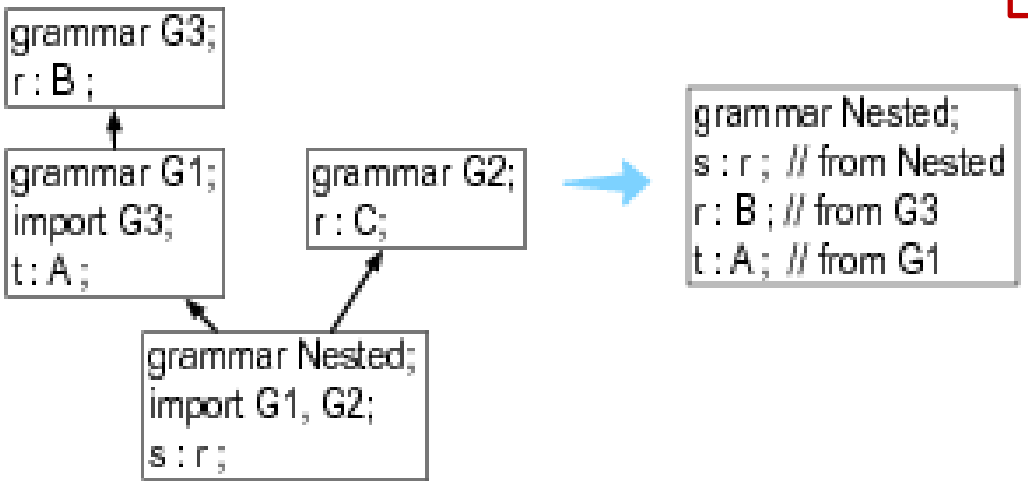
Grammar Imports

- 将文法分解成多个可复用的块
  - 通过import导入的文法类似于面向对象语言中的超类

```
grammar MyLang;
import Lang;
expr: INT | ID
INT: [0-9]+
```

```
grammar Lang;
stmt: (expr ';')+;
expr: ID
WS:[\r\n\t]+ ->skip
ID: [a-z]+
```

```
grammar MyLang;
stmt: (expr ';')+;
expr: INT | ID
INT: [0-9]+
WS:[\r\n\t]+ ->skip
ID: [a-z]+
```



https://github.com/antlr/antlr4/blob/master/doc/grammars.md

中国科学技术大学
University of Science and Technology of China

ANTLR 的原理

- ANTLR3：LL(*)[PLDI2011]
- ANTLR4：
  Adaptive LL(*)[OOPSLA2014]

Terence Parr
Univ. of San Fran.

Sam Harwell
Microsoft

Kathleen Fisher
Tufts University

确定的分析技术

确定的分析(deterministic parsing)

- LL(k)、LR(k)、LALR(k)
- LR分析对左递归的处理能力
  - 直接左递归：✓
  - 隐藏的左递归（hidden left recursion）：可能不终止
    - $A \Rightarrow^* \beta A \mu$ 其中$\beta \Rightarrow^+ \varepsilon$

例如，$S \to C\,a \mid d$　　$B \to \varepsilon \mid a$　　$C \to b \mid B\,C\,b \mid b\,b$

不确定的分析技术

- **Generalized LR(GLR)**[book1986]
  - 面向自然语言分析，其中的句子可能有二义性
  - 并行地处理（BFS）分析表中的多重条目
    - Stack List => Tree-Structured Stack => Graph-Structured Stack (GSS，4.2.2节)
    - Parse forest, next actions
  - 复杂，但在LR确定性文法上有线性性能

  如何降低栈的数量、状态的数量？
- **GLL(Generalised LL)**[entcs2010]
  - 所有上下文无关文法(含左递归文法)，最坏为立方时间
  - 将分析栈组合成一个GSS，用GSS中的环处理左递归

LL分析技术的发展

PEGs (Parser Expression Grammars)[popl2004]

- 自上而下分析，线性分析器
- 引入Prioritized choice operator '/'来提供非二义的选择

  e_1/e_2 先尝试e_1，如果e_1失败则再从同一起点尝试e_2

  - 在上下文无关文法CFG中, $A \rightarrow ab|a$与$A \rightarrow a|ab$等价
  - 但PEG规则$A \leftarrow ab/a$与$A \leftarrow a/ab$不等, 后者的分支2永不成功

  $\&e$ 尝试匹配e，然后无条件回溯到起点，只是返回是否匹配成功

  $!e$ 如果匹配e，则失败；如果不匹配，则成功
- 可以表示所有LR(k)以及其他文法，包括某些非CFG

  例如，可以表示非上下文无关语言 $a^n b^n c^n$

  $G = \{\{A, B, D\}, \{a, b, c\}, R, D\}$, 其中 R 包括

  $A \leftarrow aAb/\varepsilon$　　$B \leftarrow bBc/\varepsilon$　　$D \leftarrow \&(A\,!\,b)a^* B!.$

前述分析方法的不足

- **GLR和PEG分析器不总按设计意图来执行**
  - GLR默认接受二义的文法，不得不动态检测二义性
  - PEG不会有文法二义性，因为它总选第一个匹配的分支
- **不确定的分析器难以调试**
  - 自底向上：状态表示文法中的多个位置，难预测下一步
  - 自顶向下：易于理解，但难跟踪嵌套的回溯
- **在不确定的分析器中难以产生高质量的错误消息**
  - 自顶向下：在二义的上下文下预测
  - 自底向上：归约的不确定性
- **不确定的分析策略不易支持任意、内嵌的文法动作**
  - 预测分析器不能执行有副作用的动作
  - GLR可以以多种方式匹配相同的规则，如何处理这多个结果

ANTLR的引入

- LL(*) – ANTLR3
  - 通过句法谓词支持任意的lookahead
  - 使用正规式区分不同产生式分支，提供近似确定性分析
  - 文法不能是左递归的（可以自动转换直接左递归）
  - **不足之处**：LL(*)文法条件静态不可判定，故有时不能找到正规式来区分不同产生式分支
- Adaptive LL(*) – ANTLR4
  - 在决策点发起多个子分析器：并发分析
  - 记忆分析结果，增量动态构建DFA
  - 使用GSS(graph-structured stack)避免冗余计算

ANTLR3：LL(*)

- 谓词文法 $G = (N, T, P, S, \Pi, \mathcal{M})$
  - N:非终结符集合
  - T :终结符集合
  - P :产生式集合
  - $S \in N$:开始符
  - Π:无副作用的语义谓词
  - $\mathcal{M}$:动作集合 mutators

$A \in N$	Nonterminal
$a \in T$	Terminal
$X \in (N \cup T)$	Grammar symbol
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, x, y, w \in T^*$	Sequence of terminals
$w_r \in T^*$	Remaining input terminals
ϵ	Empty string
$\pi \in \Pi$	Predicate in host language
$\mu \in \mathcal{M}$	Action in host language
$\lambda \in (N \cup \Pi \cup \mathcal{M})$	Reduction label
$\vec{\lambda} = \lambda_1..\lambda_n$	Sequence of reduction labels
Production Rules:	
$A \rightarrow \alpha_i$	i^{th} context-free production of A
$A \rightarrow (A'_i)\Rightarrow \alpha_i$	i^{th} production predicated on syntax A'_i
$A \rightarrow \{\pi_i\}?\ \alpha_i$	i^{th} production predicated on semantics
$A \rightarrow \{\mu_i\}$	i^{th} production with mutator

[PLDI2011]Terence Parr, Kathleen Fisher. LL(*): The Foundation of the ANTLR Parser Generator.

LL(*)的产生式形式

产生式的形式

- $A \longrightarrow \alpha_i$
- 含句法谓词的产生式： $A \longrightarrow (A_i') \Rightarrow \alpha_i$
  仅当当前输入也匹配由A_i'描述的语法时， A展开成α_i
- 含语义谓词的产生式： $A \longrightarrow \{\pi_i\}?\,\alpha_i$
  仅当到目前所构造的状态满足谓词π_i时， A展开成α_i
- 动作： $A \longrightarrow \{\mu_i\}$
  根据动作μ_i更新状态

LL(*)谓词文法的最左推导

中国科学技术大学 University of Science and Technology of China

谓词文法的最左推导规则

- 判断形式(judgement form)

The judgment form $(\mathbb{S}, \alpha) \stackrel{\lambda}{\Rightarrow} (\mathbb{S}', \beta)$, may be read: "In machine state $\mathbb{S}$, grammar sequence α reduces in one step to modified state $\mathbb{S}'$ and grammar sequence β while emitting trace λ."

- 推导规则

$$Prod\ \frac{A \to \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \qquad Action\ \frac{A \to \{\mu\}}{(\mathbb{S}, uA\delta) \stackrel{\mu}{\Rightarrow} (\mu(\mathbb{S}), u\delta)}$$

$$Sem\ \frac{\pi_i(\mathbb{S}) \quad A \to \{\pi_i\}?\,\alpha_i}{(\mathbb{S}, uA\delta) \stackrel{\pi_i}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)} \qquad Syn\ \frac{(\mathbb{S}, A_i') \Rightarrow^* (\mathbb{S}', w) \quad w \preceq w_r \quad A \to (A_i')\text{=>}\alpha_i}{(\mathbb{S}, uA\delta) \stackrel{A_i'}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)}$$

在当前状态仅当由A_i'推导出的串w是剩余输入串w_r的前缀

$$Closure\ \frac{(\mathbb{S}, \alpha) \stackrel{\lambda}{\Longrightarrow} (\mathbb{S}, \alpha'),\ (\mathbb{S}, \alpha') \stackrel{\vec{\lambda}}{\Longrightarrow}{}^*(\mathbb{S}, \beta)}{(\mathbb{S}, \alpha) \stackrel{\lambda\vec{\lambda}}{\Longrightarrow}{}^*(\mathbb{S}, \beta)}$$

LL(*)

- 二义性的消除
  - 指定语义谓词来消除歧义
  - 按产生式在文法中出现的先后次序来解决歧义，冲突时选择前面的产生式规则
- 谓词LL正规文法Predicated LL-regular grammars
  - LL正规文法与LL(k)的区别

    分析器使用整个剩余输入来区分可选的产生式，而不只是k个符号

文法举例

- 下述文法是LL(*)，但不是LR(k) [PLDI2011]

```
a : b A+ X   // V_T={A, X, Y}
  | c A+ Y
b : ;
C : ;
```

Adaptive LL(*)

- LL(*)的主要问题
  - 静态不可判定
  - 文法分析有时会找不到能区分不同产生式分支的正规式
  - 回溯决策不能检测如下的二义性：$A \longrightarrow \alpha|\alpha$
    如果 α 是使得 $\alpha|\alpha$ 非LL(*)的文法符号序列
- Adaptive LL(*)，即 ALL(*)
  - 动态分析：将文法分析移到parse-time，避免LL(*)静态文法分析的不可判定性，可以为任何非左递归上下文无关文法产生正确的分析器

ALL(*)

- 预测机制
  - 在决策点，为每个候选产生式分支发起一个子分析器
  - 各子分析器可以并行地探索所有可能路径
  - 使用graph-structured stack(GSS)避免冗余计算
- 记忆分析结果
  - 增量动态构建DFA，将向前看短语映射到预测产生式
- ALL(*) parser分析的复杂度 $O(n^4)$

ANTLR4：ALL(*)

- 支持的文法
  - 支持ANTLR3 不支持的情况
    - 左递归文法，但是可以自动重写成非左递归且无二义的
    - 公共递归前缀
- 词法分析
  - 支持上下文无关的记号识别，如括号匹配、嵌套注释
  - ALL(*)适合用于scannerless parsing
- 语法分析
  - 使用类似于GLR-like机制来探索所有可能的决策路径
  - 增量动态构建lookahead DFA

ALL(*)谓词文法

- 文法$G = (N, T, P, S, \Pi, \mathcal{M})$
  - N:非终结符集合
  - T :终结符集合
  - P :产生式集合
  - $S \in N$:开始符
  - Π:无副作用语义谓词
  - $\mathcal{M}$:动作集合
  - 最左推导规则

$A \in N$	Nonterminal
$a, b, c, d \in T$	Terminal
$X \in (N \cup T)$	Production element
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, v, w, x, y \in T^*$	Sequence of terminals
ϵ	Empty string
\$	End of file "symbol"
$\pi \in \Pi$	Predicate in host language
$\mu \in \mathcal{M}$	Action in host language
$\lambda \in (N \cup \Pi \cup \mathcal{M})$	Reduction label
$\vec{\lambda} = \lambda_1..\lambda_n$	Sequence of reduction labels
Production Rules:	
$A \rightarrow \alpha_i$	i^{th} context-free production of A
$A \rightarrow \{\pi_i\}?\ \alpha_i$	i^{th} production predicated on semantics
$A \rightarrow \{\mu_i\}$	i^{th} production with mutator

$$Prod\ \frac{A \rightarrow \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)}$$

$$Sem\ \frac{\pi(\mathbb{S})}{A \rightarrow \{\pi\}?\ \alpha}\ \frac{}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \qquad Action\ \frac{A \rightarrow \{\mu\}}{(\mathbb{S}, uA\delta) \Rightarrow (\mu(\mathbb{S}), u\delta)}$$

$$Closure\ \frac{(\mathbb{S}, \alpha) \Rightarrow (\mathbb{S}', \alpha'),\ (\mathbb{S}', \alpha') \Rightarrow^* (\mathbb{S}'', \beta)}{(\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}'', \beta)}$$

思考题

- **ANTLR4 容许哪些类型的左递归？**
- **ANTLR4 对所支持的左递归如何处理？例如，对下面两种情况分别会怎样解析？**

```
Exp : Exp '*' Exp | Exp '+' Exp | IntConst;

Exp : Exp '+' Exp | Exp '*' Exp | IntConst;
```

  - ANTLR 能为上面哪种情况构造出符号‘*’的优先级比‘+’高的表达式解析器？这是基于ANTLR 采用的何种二义性消除规则？
  - 如果将下面的第1行改写成第2行，那么生成的解析器源码有什么样的变化？请理解和说明 '# Mult' 的作用和意义。

```
Exp : Exp '*' Exp | Exp '+' Exp | IntConst;

Exp : Exp '*' Exp # Mult | Exp '+' Exp # Add | IntConst # Int ;
```

- **给出ANTLR 不支持的左递归文法的例子并分析原因**