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

语法分析 III-ANTLR

《编译原理和技术(H)》、《编译原理(H)》

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

3.4 ANTLR4 简介

ANTLR(ANother Tool for Language Recognition)

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

- **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)递归下降词法解析器
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) , v4.13.1 (Sept. 4, 2023)

- **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**
  - **AdaptiveLL(*)[OOPSLA2014] : ANTLR4的原理**

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

ANTLR的引入

使用向前看正规式区分不同产生式选择，提供近似确定性分析

- LL(*) – ANTLR 3.3 [PLDI 2011]
  - 引入语法谓词、语义谓词支持任意的向前看(lookahead)
  - 文法不能是左递归的（但可以自动消除直接左递归）
  - **不足**：LL(*)文法条件静态不可判定，故有时找不到向前看正规式来区分
- Adaptive LL(*) – ANTLR 4 [OOPSLA 2014]
  - 引入语义谓词和动作产生式，支持所有的LL(*)文法
  - 在决策点发起多个子解析器：并发解析
  - 记忆解析结果，增量动态构建DFA
  - 使用GSS(graph-structured stack)避免冗余计算

ANTLR 3：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 状态修改器

$A \in N$	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:	
$A \to \alpha_i$	i^{th} context-free production of A
$A \to (A'_i)\text{=>}\ \alpha_i$	i^{th} production predicated on syntax A'_i
$A \to \{\pi_i\}?\ \alpha_i$	i^{th} production predicated on semantics
$A \to \{\mu_i\}$	i^{th} production with mutator

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

LL(*)的产生式形式以及二义性消除

□ 产生式的形式

- 标准的产生式： $A \longrightarrow \alpha_i$
- 含语法谓词的产生式： $A \longrightarrow (A_i') \Rightarrow \alpha_i$
  仅当当前输入也匹配由A_i'描述的语法时，A展开成α_i
  - □ 语法谓词$(A_i') \Rightarrow$可以替换为语义谓词{synpred(A_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

□ 二义性的消除

- 指定语义谓词来消除歧义
- 冲突时选择位于文法中靠前的产生式规则

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

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

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

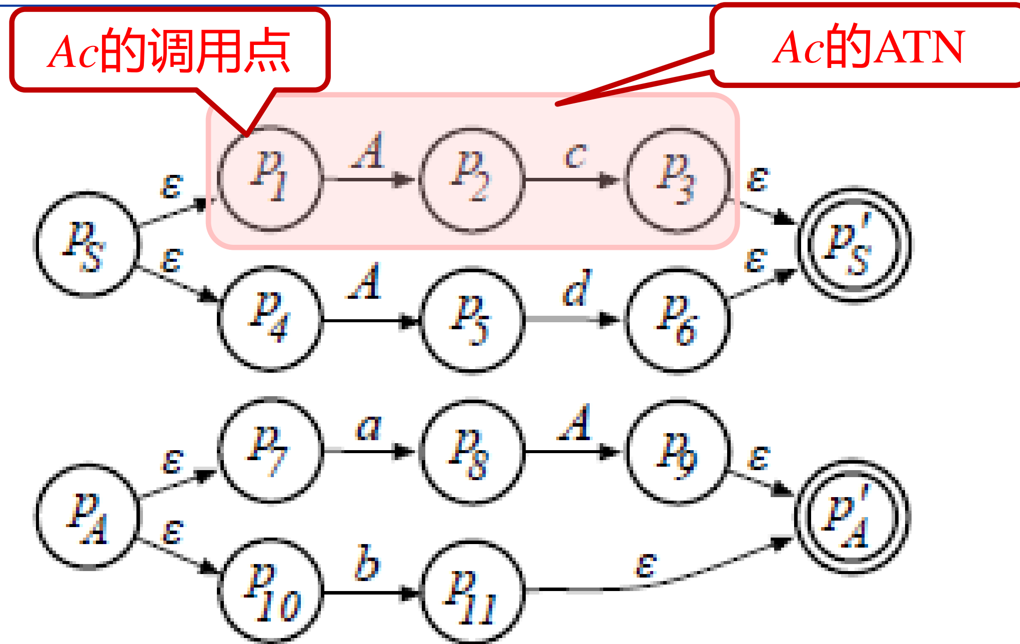
在当前状态仅当由A_i'推导出的串w是剩余输入串w_r的前缀

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

LL(*)向前看DFA

文法到增强转换网络ATN的构造

输入文法的元素	对应的 ATN 状态转换
$A \to \alpha_i$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\varepsilon} \boxed{\alpha_i} \xrightarrow{\varepsilon} p'_A$
$A \to \{\pi_i\}?\alpha_i$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\varepsilon} p'_A$
$A \to \{\mu_i\}$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \to \varepsilon$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\varepsilon} p'_A$
$\boxed{\alpha_i} = X_1X_2\ldots X_m$	$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \ldots \xrightarrow{X_m} p_m$



$P = \{S \to Ac \mid Ad, A \to aA \mid b\}$

- **ATN中非终结符边(边标记为A):**
  - 对A分析函数的调用，控制权转移到A的子自动机

向前看DFA的状态变化

- **DFA状态：一组ATN格局**

$$\frac{p \xrightarrow{a} q}{(\mathbb{S}, p, aw) \mapsto (\mathbb{S}, q, w)} \qquad \frac{\pi_i(\mathbb{S}) \quad p \xrightarrow{\pi_i} f_i}{(\mathbb{S}, p, w) \xrightarrow{\pi_i} (\mathbb{S}, f_i, w)}$$

$$\frac{(\mathbb{S}, f_i, w)}{\text{Accept, 预测产生式}i}$$

ANTLR4的Adaptive LL(*)

- **LL(*)的主要问题**
  - 静态不可判定
  - 文法分析有时找不到能区分不同产生式选择的正规式
  - 回溯决策也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解析器

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

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

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

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

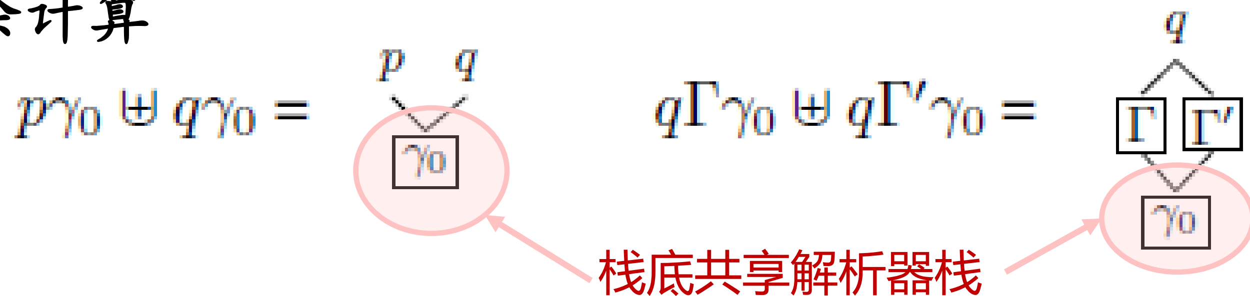
ALL(*)

预测机制

- 在决策点，为每个产生式选择发起一个子解析器，各子解析器可并发探索
- 调用栈敏感的预测

  $S \rightarrow xB \mid yC \qquad B \rightarrow Aa \qquad C \rightarrow Aba \qquad A \rightarrow b \mid \varepsilon$

  - 无解析器栈（强LL解析器）：在A的决策点面临ba时，A的两个选择都可以—预测冲突
  - 有解析器栈：B 调用 A 时预测使用 $A \rightarrow b$ 展开，C 调用 A 时预测使用 $A \rightarrow \varepsilon$ 展开
- 使用**图结构栈**(GSS)避免冗余计算



记忆分析结果

- **增量动态**构建DFA

  建立向前看短语到预测的产生式编号的映射

ALL(*) 解析器分析的复杂度 $O(n^4)$

ANTLR的使用

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
}
```

- 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

模式定义 词法状态

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

□ 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

http://lab.antlr.org/

词法语法切换

Welcome to the ANTLR lab, where you can learn about ANTLR or experiment with and test grammars! Just hit the Run button to try out the sample grammar.

To start developing with ANTLR, see getting started.

Feedback/issues welcome. Brought to you by Terence Parr, the maniac behind ANTLR.

Disclaimer: *This website and related functionality are not meant to be used for private code, data, or other intellectual property. Assume everything you enter could become public! Grammars and input you enter are submitted to a unix box for execution and possibly persisted on disk or other mechanism. Please run antlr4-lab locally to avoid privacy concerns.*

Lexer Parser | Sample

```
1  parser grammar ExprParser;
2  options { tokenVocab=ExprLexer; }
3
4  program
5      : stat EOF
6      | def EOF
7      ;
8
9  stat: ID '=' expr ';'
10     | expr ';'
11     ;
12
13 def : ID '(' ID (',' ID)* ')' '{' stat* '}
14
15 expr: ID
16     | INT
17     | func
18     | 'not' expr
19     | expr 'and' expr
20     | expr 'or' expr
21     ;
22
23 func : ID '(' expr (',' expr)* ')' ;
```

文法编写区

Input | sample.expr

```
1  f(x,y) {
2      a = 3+foo;
3      x and y;
4  }
```

文法测试输入区

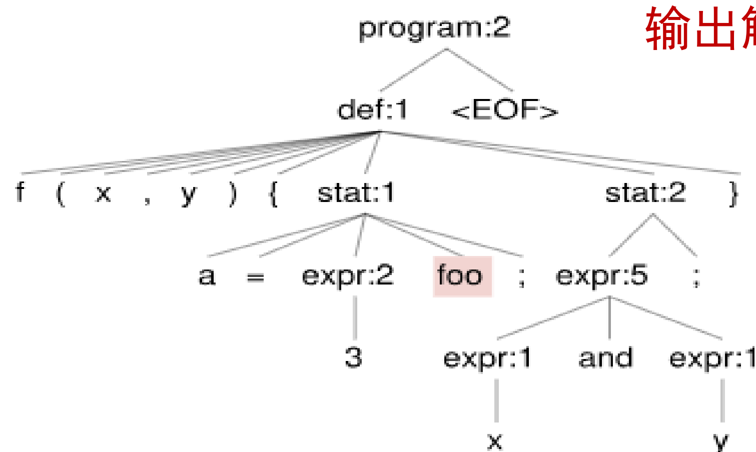
Start rule: program | Run | Show profiler

设置开始符号进行测试

Parser console

```
2:9 token recognition error at: '+'
2:10 extraneous input 'foo' expecting ';'
```

Tree Hierarchy



输出解析树

使用ANTLR构造解析器

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

解析器的生成、编译和运行

```
$ antlr4 ExprParser.g4
$ javac ExprParser*.java
$ grun Expr program -gui sample.expr
```

```
antlr4='java org.antlr.v4.Tool'
grun='java org.antlr.v4.gui.TestRig'
```

-gui ：以图形界面（GUI）形式展示解析树。它会生成一个窗口，直观显示解析树结构。

-tree：在终端以文本的层次结构表示解析树的树形结构，在没有图形界面环境时很有用。

-trace：详细输出解析步骤，包括进入和退出每个规则时的状态，有助于调试和分析。

```
import org.antlr.v4.runtime.*;
import java.nio.charset.Charset;
import java.nio.file.Paths;
import java.io.IOException;
public class Main {
    public static void main(String[] args) throws Exception {
        Charset charset = Charset.defaultCharset ();
        CharStream input = CharStreams.fromPath(Paths.get(args[0]), charset);
        ExprLexer lexer = new ExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExprParser parser = new ExprParser(tokens);
        ParserRuleContext tree = parser.program();  // 开始解析
        System.out.println(tree.toStringTree(parser));  // 打印语法树
    }
}
```

Parser规则

语法规则

规则名 : 选择1 | . . . | 选择n;

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

Parser规则

□ 給规则选择加标签

■ 格式： #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，如 XXXParser.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，

作为在遍历解析树时要监听触发的事件方法

```
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

给规则选择加标签

- **可以复用标签**

e : e '*' e # BinaryOp | e '+' e # BinaryOp | INT # Int ;

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

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 enterBinaryOp (XXXParser. BinaryOpContext ctx);
    void exitBinaryOp (XXXParser. BinaryOpContext 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

□ 规则属性定义

规则名[参数args] returns [返回值rvals] locals [局部变量lvars] : ... ;

例如：

```
add[int x] returns [int ret] : '+' INT { $ret = $x + $INT.int; } ;
```

□ 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.*;
}
```

Java代码，直接复制到生成的解析器源码的头部

```
@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 ADD : return left + right;
            ......
        }
        return 0;
    }
}
```

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

```
program: stat+ ;
stat:   e NEWLINE           {System.out.println($e.v);}
    |   ID '=' e NEWLINE    {memory.put($ID.text, $e.v); System.out.println($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
```

输入程序及其解析结果

```
a=3 +4 ;
b=a+4 ;
```

```
a=7
b=11
(program (stat a = (e (e 3) + (e 4)) ; \n) (stat b = (e (e a) + (e 4)) ;
```

Grammar Imports

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

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

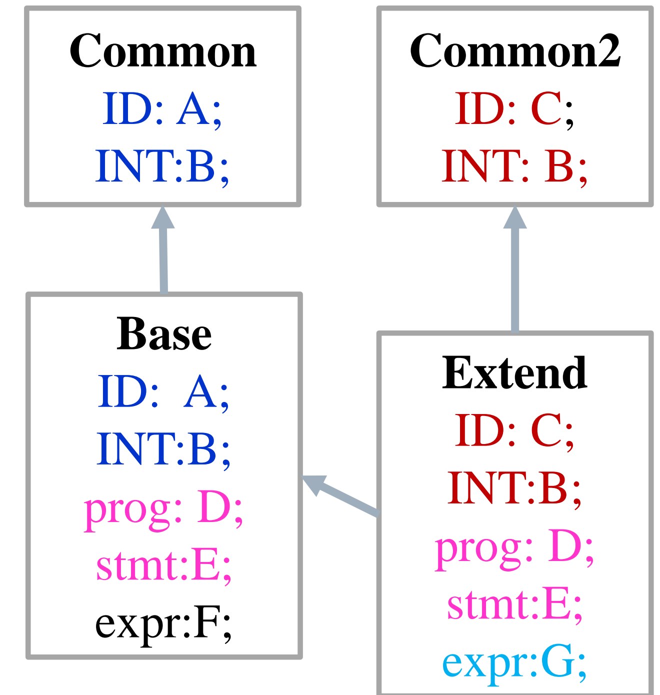
```
lexer grammar Common;
ID  : [a-zA-Z_][a-zA-Z_0-9]* ;
INT : [0-9]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

```
lexer grammar Common2;
ID  : [a-zA-Z][a-zA-Z_0-9]* ;
INT : [0-9]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

```
grammar Base;
import Common;
prog : stmt* ;
stmt : expr ';' ;
expr : ID ;
```

```
grammar Extend;
import Common2,Base;

// 扩展表达式规则
expr : ID | INT ;
```



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

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**