参考：<https://segmentfault.com/a/1190000006875885>

摘要

DSL的核心在于受限的表达能力，容易理解的语义以及清晰的语义模型，最后一点虽然并非是必要的，但在绝大多数时候往往是成立的。其中受限的表达能力是我最想强调的一点，我们码农在写代码的时候往往对于程序的可扩展性和API支持的泛化能力有一种执着的追求，但对于以表达而非功能的DSL，却并非是一件好事。扩展和泛化的职责应该交由语义模型完成，而非DSL自身，在大多数情况下采用DSL级联比实现一个快要接近于通用语言的DSL要好，你可以把它和程序设计中的单一职责原则联系起来，虽然不那么恰当，我想说的就是让你的DSL保持小巧精悍的样子。··

DSL探讨的问题

DSL本身只是一层为了表达能力而做的浅浅封装，在你考虑DSL的时候你应该将绝大部分的精力花在构建语义模型上，但反过来说DSL的设计从某种程度上来说帮你理清了语义模型的需求，这让我们看起来开始像一条产品狗了（笑），接下来我们大致过一下DSL设计和实现主要考虑的问题吧。

* 首要的就是如何设计DSL表达形式和你的语义模型，这是领域相关的技术
* 如何解析DSL文件并根据情况生成IR（中间数据）
* 如何在解析过程中或者在解析之后通过DSL 来构建语义模型
* 有时候仅仅得到语义模型是不够的，我们需要考察一些代码生成之类的输出相关的问题。

接下来我们将分别详细介绍这些主题。

解析DSL

这里的解析DSL显然指的是外部DSL，那么我们接下来讨论的你可能在编译原理前端都已经听到过了，但我还是需要在这里再简要地讲述一遍，请原谅我的啰嗦。

解析的过程主要可以分为词法解析、语法解析，视你的DSL的复杂程度你可能还需要在中间生成符号表，使用语境变量和维护作用域上下文。词法解析过程将DSL文本解析成一个Token Stream，token包含词的原始内容和位置，同时这个过程将会丢弃一些无用的字符比如空格符等。语法解析从Token Stream中生成语法解析树，在生成的过程中你可以采取一系列的动作来组装你的语义模型，这一点我们到下一节再讲。

接下来让我们考虑一下最简单的情形，比如读入一个CSV文件，我们并不需要定义语法，通过使用','分隔每一行就足够进行解析了，这就是所谓的分隔符指导翻译，由于语法过于简单，你只是通过分隔符做了一下词法解析。

我们可以考虑一个稍微复杂的场景，你需要从一个日志文件中的日志记录快速读取若干信息，比如错误类型、相关用户，这时候你可以考虑使用正则词法分析，你快速获得了你的词条，并且并没有什么语法解析的过程就可以组装出你的语义模型（可能是一个键值对或者一行记录）。

上面所列举的例子都没有涉及到语法分析，对于简单的情形，你可以根据自己的逻辑编写LL和LR解析器进行解析而不需要一个显式的文法定义。但往往我们首先需要定义我们的文法并通过文法来指导我们做语义解析。关于BNF文法和EBNF文法的内容这里就不再提及了，你可以自行查阅。当你不采用语法树来帮助你进行解析的情况下，上面方法几乎都是在组合解析器，欲匹配A规则，A由B和C组成，则先匹配B再匹配C，整个解析过程基于这种嵌套递归，有人将这种解析过程为解析器组合子。这里面有一些有趣的情形，当两条文法规则拥有相同的Token前缀的时候，我们怎么解决？可以通过向前看K步直到发现不一致的地方为止，扩展到极端情况下就是回溯法。我们还可以再做一点优化，在回溯的时候记录下已经匹配上的子规则和其在标记流中的位置，切换到其他候选规则时就可以不必重复匹配相同前缀了，有些人将这种方法称为是记忆解析器。**需要我们注意的是否使用IR如语法树对于语法解析的过程并无本质区别，不同点在于使用IR可以多次遍历语法结构更方便构建语义模型，同时AST等可以通过许多已有工具快速生成**。

对于上下文相关文法，对于C++比如T(x)这种，既可能是函数调用也可能是类型转换，你必须结合上下文来判断，看有没有T的函数定义。这时候你可能需要构建符号表和作用域上下文来帮助你做谓词解析（语义预测）。

生成语义模型

生成语义模型几乎是我们今天最重要的问题了，我个人认为这也几乎是DSL开发的核心部分，我们从内部DSL和外部DSL分别介绍这个问题的解决方案。

外部DSL

从外部DSL生成语义模型，我们可以根据是否采用IR分开来考察，当不使用IR的时候，我们是当匹配上某一规则执行对应的操作，比如向模型里填充数据。但是当你有回溯动作的时候，一旦某个规则匹配失败，你可能不得不清除其子规则做出的更改。所以好的做法并不是直接做出对最终模型修改的动作，而是生成中间对象或者借助生成器，如果想象为树的构建，那么动作一定是在从下至上出子树时执行。

当借助IR进行构建时，一方面可以在构建AST时执行相应操作，另外一方面可以在构建AST完毕后再次遍历构建语义模型，大多数情况下这两种方法可以结合起来使用。下面我们以ANTLR为例介绍借助于AST的种种构建方法。

* 嵌入式语法翻译和嵌入助手：嵌入式语法翻译支持你在进入子树根节点和退出子树根节点时执行相应操作，它支持你在进入节点时定义需要使用的变量来帮助你更好地编写处理逻辑。如果将绝大部分的action逻辑都放置到外部类中，然后在文法文件中只需要调用外部类的方法，这样可以避免在文法中混入太多的通用代码以至于破坏了可读性和表达的清晰程度，这就是嵌入助手。

grammar Rows;

@parser::members { // add members to generated RowsParser

**int** col;

public RowsParser(TokenStream input, **int** col) { // custom constructor

this(input);

this.col = col;

}

}

**file**: (row NL)+ ;

row

locals [**int** i=0]

: ( STUFF

{

$i++;

**if** ( $i == col ) System.out.println($STUFF.**text**);

}

)+

;

TAB : '\t' -> skip ; // match but don't pass to the parser

NL : '\r'? '\n' ; // match and pass to the parser

STUFF: ~[\t\r\n]+ ; // match any chars except tab, newline

* 内嵌解释器：在树的构建过程中指明每个规则节点返回的类型，以及在文法文件中嵌入代码生成返回对象，这种只适合于非常简单的情况。

a returns [**string** **expr**]

: b {$expr = $b.**expr**;}

| c {$expr = $c.**expr**;}

;

* 外加代码：和内嵌解释器不同，代码并非加在文法定义文件中而是实际的DSL脚步中，与其说它是一种构建手段，称它为DSL表达能力的一种扩展可能更为合适。你可以使用通用语言在DSL中表达复杂的逻辑，在构建过程中可以将代码块当做闭包传入。

scott handles floor\_wax **in** MA **when** {/^Baker/.test(lead.name)};

* 通过Listener和Visitor重新遍历语法树，Listener在每次进入（top-down）和退出（bottom-up）节点的过程中允许你自定义动作，你可以在此组装你的语义模型。Visitor则让你在每次访问一个子树后生成一个新的对象。比较这两种模式可以看出Listener是嵌入助手的一个变体而Visitor是内嵌解释器的加强。对于ANTLR4已经不再支持自定义树的构建过程了，我们可以通过实现Visitor遍历已有的语法树来构建满足我们自己需求的新树。

grammar VecMath;

statList: stat+;

stat: VAR '=' expression # assignStat

| 'print' expression # printStat

;

primary: VAR | INT | list;

expression: multiExpr ( op=('+' | '-') multiExpr )\*;

parExpr: '(' expression ')' ;

unaryExpr: '-'? factorExpr;

multiExpr: unaryExpr (op=('\*' | '.') unaryExpr)\*;

factorExpr: parExpr | primary;

list: '[' expression (',' expression)\* ']' ;

**VAR**: ('a'..'z' | 'A'..'Z')+;

INT: ('0'..'9')+;

WS: ('\r' | '\n' | '\r' | ' ')+ {skip();};

**public** **class** **ParseTreeVisitor** **extends** **VecMathBaseVisitor**<**Node**> {

**@Override**

**public** Node **visitStatList**(VecMathParser.StatListContext ctx) {

List<Node> children = ctx.stat().stream().map(c->visit(c)).collect(Collectors.toList());

RuleNode node = **new** RuleNode("statList");

node.setChildren(children);

**return** node;

}

**@Override**

**public** Node **visitAssignStat**(VecMathParser.AssignStatContext ctx) {

RuleNode node = **new** RuleNode("stat");

node.setChildren(listChildren(ctx));

**return** node;

}

**@Override**

**public** Node **visitPrintStat**(VecMathParser.PrintStatContext ctx) {

RuleNode node = **new** RuleNode("stat");

node.setChildren(listChildren(ctx));

**return** node;

}

**@Override**

**public** Node **visitList**(VecMathParser.ListContext ctx) {

RuleNode node = **new** RuleNode("list");

node.setChildren(listChildren(ctx));

**return** node;

}

**private** List<Node> **listChildren**(ParserRuleContext ctx){

**return** ctx.children.stream().map(c->{

**if**(c **instanceof** TerminalNode)

**return** **new** TokenNode(((TerminalNode) c).getSymbol());

**else** **if**(c **instanceof** ParserRuleContext)

**return** visit(c);

**else**

**return** **null**;

}).filter(c->c!=**null**).collect(Collectors.toList());

}

}

* 重写输入流：有时候我们需要对子树进行模式匹配并对它做一些形式变换来更好地执行构建逻辑，我们可以采用重写输入流的方式来达到这一目的。

**public** **class** **RewriteListener** **extends** **IDataBaseListener** {

TokenStreamRewriter rewriter;

**public** **RewriteListener**(TokenStream tokens) {

rewriter = **new** TokenStreamRewriter(tokens);

}

**@Override**

**public** **void** **enterGroup**(IDataParser.GroupContext ctx) {

rewriter.insertAfter(ctx.stop, '\n');

}

}

内部DSL

对于内部DSL主题，可能更多的是一些代码技巧和语言特性的使用，作为一名程序猿你可能本身已经对此很熟悉了，但我还是要在此献丑一番了。

对于通用语言，生成模式的代码形式主要是链式调用（方法级联）、方法序列、嵌套方法；这些名词对你来说可能有些陌生，让我换一些更通俗的说法吧，上面分别对应于建造者模式、大量的连续指令也就是我们最常用的编程形式、将一个函数的返回值作为另一个函数的参数使用。

接下来我要提一些在使用这些形式时需要注意的问题和可能会用到的技巧。当你使用方法级联时你可能最需要注意的就是作用范围了。一个生成器（Builder）是最简单的情况，我们不如考虑这样一种情形，A的生成需要B、C、D，B又依赖于E和F，然后他们每一个都有大量的构造参数，这种时候你希望通过使用一次链式调用来完成A的创建，恐怕就需要限制级联方法的范围了，你可以在生成器中记录它的父生成器，父生成器的方法可以调用子生成器的同名方法，子生成器方法结尾再次返回父生成器。这是一种解决思路，但并不够优美，当然你也可以选择方法序列和链式调用结合。

让我们再考虑一种情形，如果A依赖于两个B对象进行构建，那链式调用可能是这样的ABuilder.B().E(XX).F(XX).B().E(XX).F(XX)，在实现过程中你必须能够分清楚两次不同的E方法分别作用于哪个B，所以你可能需要一个应用来表示当前正在进行的阶段和生成的对象，这就是语境变量。

当你使用嵌套函数的时候，最大的麻烦在于在你调用上层方法时，其嵌套方法都已经执行完毕了，数据也准备好了，你没法做一些更灵活的扩展。解决的方法一方面你可以传入生成器而非方法；另外一方面你可以使用闭包传入代码块。闭包是一种极其优秀的语言特性，也是实现修饰模式的极佳选择。使用闭包你可以将对象的new操作的时机掌控在自己手中，并通过执行闭包中的代码块来实际初始化对象。更有意思的是闭包不仅仅可以用来直接执行，你还可以将它当做是输入的DSL（不过这里DSL的语法恰好是通用语言）进行解析来生成语义模型，比如当闭包中是一个bool表达式组合，你就可以将这个表达式解析成语法树并最终构建出一个组合模式的语义模型（详见DSL 41章，这一部分很有意思）。

接下来我们所提到的可能更多的和语言特性相关，下面一一列举做个简要介绍：

* 动态接收：ruby等动态语言的特性，当类中的某些方法未被定义但是却被调用的时候允许你在类中自定义方法来处理这种情况。Ruby的Getter/Setter可以通过这种方法实现，同样，我们还可以联想到DAO中的findByXXX之类的方法，也可以采用类似的技术来实现。动态接收虽然java不支持但我们可以借用里面的思路，比如提供通用的API但是对于方法找不到的异常进行拦截并处理，最终调用通用API去完成其功能。当然实际上的Spring-Data-JPA应该不是这么做的，我猜他们是使用了动态代理技术并反射填充了method。动态接收可以帮助我们拥有许多更富有语义的方法却不用一一实现它而只用实现一个泛化的API和一个接收方法。
* 渐进式接口：为了显示规范链式调用中的操作顺序，你可以让链中的方法返回不同的接口，每个接口只包含后续操作的方法，但实际的生成器实现了所有的接口，这样当你不进行显式的类型转换时就无法按非法的顺序执行调用链了，这种方法适合构造动作有强顺序关系的场景。
* 字面量扩展：这个就是典型的语言特性了，比如对于已有的类Integer，我们在特定的命名空间里可以为它动态地扩展新方法，很可惜Java并不支持这种形式。字面量扩展对于内部DSl的表达能力是一个提升，但是我奉劝你谨慎的使用它，因为绝大多数时候不需要使用它你可以编写出出色的代码。
* Literal Map：有些语言比如Python它们支持关键字参数和默认参数，这样看起来语义更明晰而且在调用的时候更难以犯错。我们可以通过为方法传入列表或Map来实现类似的功能，只是需要你在方法实现中首先解析这些。但我同样不推荐这种做法，现代IDE的智能提示程度已经很大程度上为我们规避了调用时候出错的可能，编写同名方法也并非多么劳累的事情。
* 注解：注解的核心在于把定义和处理分离，其定义应当是声明式的，不应该和任何处理的逻辑流程相关。我们可以这样理解，注解是一种携带信息的索引，方便你在任何时刻任何阶段对被标注的实体进行处理。每一个Java程序猿对注解都不可能不熟悉，注解经常被用于取代复杂的配置文件并允许用户通过很少的代码进行配置（Spring Boot），同时注解也可以用来做Validation，关于注解的多种使用方式这里就不再赘述了。
* 类符号表：为了使用IDE的提示功能，我们可以在生成器里先声明所有需要的对象，但不进行初始化操作，在生成器额初始化过程中采用反射为它们赋予标识符并开辟内存空间，最终在实际的初始化代码中我们就可以直接使用这些对象进行操作了，这时你就可以使用IDE为它们的方法进行提示了。

经典的语义模型

在这里我主要想提三个比较常见和有代表性的语义模型：

依赖网络

依赖网络本质上是一个有向无环图，包含了节点和它们之间的依赖关系，根据节点的类型我们大致可以分为以任务为核心的依赖网络比如ANT和以输出为核心的依赖网络比如Makefile，这里面有些微妙的区别；前者可能更关心任务不会被重复执行也就是构建合理的任务执行序列，后者可能更关心当某个中间输出结果变动时需要重新生成部分的输出而并非全盘重新输出。

构建依赖网络的主要问题是：

1. 快速从任意节点出发遍历网络找出依赖关系。
2. 检测环的问题。
3. 出现事件时（中间输出变化或者依赖不满足）能够快速生成解决方案。

产生式规则系统

产生式规则系统的核心是一条一条的规则，当规则满足的时候会触发相应的动作，所以核心在于如何快速找出候选的规则集，毕竟每次遍历所有规则既不现实也不高效。这里需要重视的是规则和规则之间的关系，当某个规则满足并触发动作时可能会造成有一系列新的规则满足，这被称为规则的前向式交互。但我们依然需要谨慎地防止环的出现，一种办法是在加入规则时进行检测，另一种办法则是在检查规则的时候维护一个激活集，一个规则只能被激活一次。我认为提前构建好规则之间的关系是个不错的方法，这样有助于我们在实际判断规则条件是否满足的时候能够快速检索到那些被上一条规则激活的规则。

状态机

状态机的使用十分广泛，相信在座的程序猿没有几个木有用过状态机的，所以在这里就不多言了，但是它又是如此的重要以至于我必须把它作为一个醒目的标题列出来，忽视它是不可饶恕的。

DSL的输出技术

在这一节里我们主要探讨代码翻译和代码生成的技术，根据是否需要显式借助语义模型、是否需要借助于外部信息等可以把它们分为三大类：

* 语法制导的翻译：不需要借助语义模型，不需要输出模型，以直接输出外部语句为主，在语法解析的同时就插入动作进行翻译，对当前的翻译不需要输入流后面的信息。
* 基于规则的翻译系统：虽然不需要语义模型，但是需要显式的外部规则的指导，一般规则是由输入语句的文法和执行的转换组成。对于基于规则的翻译系统，你既可以在解析语法结构的同时匹配规则；也可以先生成IR比如AST再遍历语法树通过树文法（在ANTLR4中已经废弃，你可以通过Visitor生成符合你遍历规则的新树来完成这件事）做子模式匹配来匹配规则，当然一般情况
* 基于语义模型的翻译：这种翻译和语法解析过程就几乎没有关系了，你的目标转变为一个个语义对象生成对应的输出。你可能需要定制特定目标的生成类，比如对于生成SQL语句，你可以为Table类单独编写一个生成输出的类，当然你也可以不借助于输出生成器，从Table组装它的每一列最终直接输出。但我认为输出的任务最好不要和语义模型混杂在一起，这违反了单一职责原则。

上面讲的都是代码生成的思路，对于如何操纵输出内容所提甚少，在实际开发中，输出模板和模板引擎往往是必不可少的。可以为每一个语义对象创建输出模板，然后将实际的语义对象传入模板引擎，在模板中填入动态变化的数据。这种方式对于AJAX时代之前的WEB程序猿简直是驾轻就熟，大量的模板被用来生成HTML中的动态内容，本质上并无任何不同。只不过当我们借助于语义模型并且有大量的嵌套操作时，我们可能得做些模板的嵌套和拼接了。

当不借助于语义模型时我们也可以在树的构建中使用模板以生成输出，不过在ANTLR4中你可能得手动编写Listener来调用模板了。说到这里我就不得不吐槽一下ANTLR4了，虽然我承认将文法和各种逻辑操作解耦是一个正确的方向，但是突然感觉一切没有那么灵活了，尽管去掉的这些功能几乎都可以很快地通过Listener和Visitor实现，依然有一种蛋疼的感觉。最后放一个StringTemplate的模板定义文件吧。

group Cymbol;

// START: file

file(defs) ::= <<

<defs; separator="\n">

>>

// END: file

// START: class

class(name, sup, members) ::= <<

class <name> <if(sup)>: <sup><endif> {

<members>

};

>>

// END: class

// START: method

method(name, retType, args, block) ::= <<

<retType> <name>(<args; separator=", ">) <block>

>>

// END: method

// START: block

block(stats) ::= <<

{

<stats; separator="\n">

}

>>

// END: block

// START: if

if(cond, stat1, stat2) ::= <<

if ( <cond> ) <stat1>

<if(stat2)>else <stat2><endif>

>>

// END: if

// START: assign

assign(a,b) ::= "<a> = <b>;"

// END: assign

return(v) ::= "return <v>;"

// START: callstat

callstat(name, args) ::= "<call(...)>;" // call() inherits name,args

// END: callstat

// START: decl

decl(name, type, init, ptr) ::=

"<type> <if(ptr)>\*<endif><name><if(init)> = <init><endif>"

var(name, type, init, ptr) ::= "<decl(...)>;"

arg(name, type, init, ptr) ::= "<decl(...)>"

// END: decl

// START: ops

operation(op, x, y) ::= "(<x> <op> <y>)"

// END: ops

// START: operator

operator(o) ::= "<o>"

// END: operator

unary\_minus(v) ::= "-(<v>)"

unary\_not(v) ::= "!(<v>)"

addr(v) ::= "&(<v>)"

deref(v) ::= "\*(<v>)"

index(array, i) ::= "<array>[<i>]"

member(obj, name) ::= "(<obj>).<name>"

// START: call

call(name, args) ::= <<

<name>(<args; separator=", ">)

>>

// END: call

小结

这篇文章的主要目的是为了整理和概括一下DSL的主要相关知识，可能内容也有些杂乱，有些地方也没有说的很清楚，或者是说到一半就戛然而止了，希望大家多多包涵，也希望大家有什么想法可以和我讨论。