# 摘要

目前，关于XML函数依赖的现有研究普遍沿用关系库理论中函数依赖的概念，虽然研究了各种XML数据项之间的函数依赖关系，且普遍考虑到了XML数据中的结构约束关系，进而讨论函数依赖的推理和规范化设计问题，但是都没有涉及到XML数据类型，所以就无法避免上述的数据冗余和更新异常问题。

Currently, the existing researches about XML functional dependency basically follow the concept of the functional dependency in Relatonal Database theory. Many researches study varisous function denpendency relations between XML data items, and they take into account the structural constraints relations and discuss about the reasoning for functional dependency and normalizing for databse design, but they do not involve the XML type in this area, and they cannot solve the problems of data redundancy and update anomalies.

为此，北京工业大学计算机软件与理论实验室为XML的数据依赖设计了一种通用XML依赖关系，其中包含数值相关的函数依赖关系及类型相关的依赖关系；且提出了基于DOM的XML类型相关的数据依赖检查算法并已经实现检查原型系统。虽然针对算法的检查原型系统已经实现，但尚未实现对XML完整性约束描述的解析器来将用户提出的约束描述转换为系统识别的内部数据对象。因此需要一个方便且强大的解释器。

The laboratory of Computer Software and Theory in Beijing University of Technology propose the XML polymorphic dependency relationship, in which the relationships include not the dependencies between values but also the dependencies between types and values. The dependencies checking prototype system based on XML DOM has been implemented for checking XML polymorphic denpendencies in XML document, but we still need a parser to parse the XML dependencies and corresponding dependency instances wrote in an .TXT file to be the inner data structures for checking.

本文在参考北京工业大学计算机软件与理论实验室提出的通用XML依赖关系的基础上，设计了用户编写的完整性约束描述使用的文法。根据该文法，本文又设计并实现了针对用户编写的完整性约束描述的解析器，分为词法分析Lexer及文法分析Parser两个部分，最终将用户编写的完整性约束描述解析为后续系统识别的内部数据结构，且对该解析器进行了功能测试。

In this paper, we introduce the syntax design for XML polymorphic dependency and dependency instances, and the design and implementations of the lexer and parser that used to convert text descriptions of the XML polymorphic dependency and instances into the inner data structrues. We also do some experiments to do functional test for our parser, and the result show our parser can work successfully.

关键字：完整性约束描述；XML依赖关系；词法分析器；文法分析器

# Abstract

Currently, the existing researches about XML functional dependency relations generally follow the concept of functional dependency in Relational Database theory. While they study the various functions of XML data dependencies between items, and also generally take into account the structure binding relationship of XML data, and then discuss the question about the reasoning and standardization of functional dependencies design issues, they do not the XML data type, so they cannot avoid the problem about data redundancy and update anomalies above.

To this end, Beijing University of Technology Laboratory of Computer Software and Theory design a generic XML dependency for the XML data dependencies, which contains the values associated with functional dependency and the type of the associated dependencies. And they also put up a DOM-based checking algorithm for XML type on the data dependency and have achieved the inspection system. Although the inspection system algorithm has been implemented, but it has not yet achieved the XML integrity constraint parser which is used to transform the description that the user proposed into internal data structure. So they need a convenient and powerful interpreter.

In reference to the generic XML dependency which are proposed by Beijing University of Technology Laboratory of Computer Software and Theory, this paper designs and implements the parser for the integrity constraint dependencies written by user which is divided into two parts: the lexical analysis part which is named Lexer and the grammar analysis part which is named Parser. Eventually, the parser transforms the integrity constraint dependencies written by user into internal data structure and then is tested.

Keywords: Integrity constraint dependencies; XML dependencies; Lexer; Parser;

目录

[摘要 I](#_Toc357439738)

[Abstract II](#_Toc357439739)

[1. 绪论 1](#_Toc357439740)

[1.1. 课题背景 1](#_Toc357439741)

[1.2. 研究现状 2](#_Toc357439742)

[1.2.1. 类型相关的依赖关系研究 2](#_Toc357439743)

[1.2.2. 文法解释器的实现技术 2](#_Toc357439744)

[1.3. 任务及主要工作 3](#_Toc357439745)

[1.4. 目的及意义 3](#_Toc357439746)

[2. 相关技术介绍 4](#_Toc357439747)

[2.1. XML语言 4](#_Toc357439748)

[2.2. XML正规式 5](#_Toc357439749)

[2.3. 递归下降分析法 5](#_Toc357439750)

[2.3.1. 思想及优缺点介绍 5](#_Toc357439751)

[2.3.2. 与本课题的适用性 7](#_Toc357439752)

[3. 文法设计 8](#_Toc357439753)

[3.1. 文法总体概述 8](#_Toc357439754)

[3.2. 类型绑定及类型声明文法释义 9](#_Toc357439755)

[3.3. 依赖声明及依赖实例文法释义 11](#_Toc357439756)

[4. 词法分析及文法分析的设计 16](#_Toc357439757)

[4.1. 输入输出设计 16](#_Toc357439758)

[4.1.1. 解释器输入输出概况 16](#_Toc357439759)

[4.1.2. 词法分析的输入输出 17](#_Toc357439760)

[4.1.3. 文法分析的输入输出 17](#_Toc357439761)

[4.2. 词法分析及文法分析的任务 18](#_Toc357439762)

[4.2.1. 词法分析 18](#_Toc357439763)

[4.2.2. getToken及nextToken操作 19](#_Toc357439764)

[4.2.3. 文法分析 19](#_Toc357439765)

[4.2.4. 词法分析和文法分析的关系 20](#_Toc357439766)

[4.3. 词法分析及文法分析的结构 20](#_Toc357439767)

[4.3.1. 词法分析结构 20](#_Toc357439768)

[4.3.2. 文法分析状态转换结构 21](#_Toc357439769)

[4.3.3. 错误处理机制 26](#_Toc357439770)

[5. 实现及实验 27](#_Toc357439771)

[5.1. 数据结构的设计及实现 27](#_Toc357439772)

[5.1.1. 多态的体现 27](#_Toc357439773)

[5.1.2. 类间关系及说明 28](#_Toc357439774)

[5.2. 词法分析器Lexer的实现 32](#_Toc357439775)

[5.2.1. token的划分及区分 33](#_Toc357439776)

[5.2.2. getToken总体流程 34](#_Toc357439777)

[5.2.3. nextToken总体流程 35](#_Toc357439778)

[5.3. 文法分析器Parser的实现 35](#_Toc357439779)

[5.3.1. 总体流程 35](#_Toc357439780)

[5.3.2. 依赖声明和依赖实例的特殊处理 36](#_Toc357439781)

[5.4. 实验及测试 37](#_Toc357439782)

[结论 39](#_Toc357439783)

[参考文献 40](#_Toc357439784)

[致谢 41](#_Toc357439785)

[附录 42](#_Toc357439786)

[多个测试用例 42](#_Toc357439787)

[通配符的获取 42](#_Toc357439788)

[多个QNAME（用户自定义类型）的获取 42](#_Toc357439789)

[前缀PATH的获取 43](#_Toc357439790)

[无 CONSPATH有CONSTYPOF的处理 43](#_Toc357439791)

[有 CONSPATH无CONSTYPEOF的处理 44](#_Toc357439792)

[有CONSPATH有CONSTYPEOF的处理 44](#_Toc357439793)

[无CONSSTR有CONSTYPE的处理 44](#_Toc357439794)

[有CONSSTR无CONSTYPE的处理 45](#_Toc357439795)

[有CONSSTR有CONSTYPE的处理 45](#_Toc357439796)

# 绪论

## 课题背景

随着Internet的不断发展，人们迫切寻求一种一致的通信接口，以便能够在网络上更方便地进行数据交换。XML（eXtensible Markup Language）即“可扩展标记语言”的出现，解决了这一问题。现如今XML的使用正在变得越来越广泛[1]。

在关系数据库设计中，不良的数据模式设计可能会导致数据更新过程出现异常，而良好的数据模式设计则需要根据数据之间的函数依赖关系来组织数据。由此可见，函数依赖等完整性约束是分析数据冗余，避免更新异常的重要手段。基于同样的理由，良好的XML数据模式设计也需要建立在良好的XML函数依赖等完整性约束理论之上。鉴于XML数据的广泛应用及其自身特点等，关于XML完整性约束问题的研究得到了不少关注，且取得了一些进展[2]。

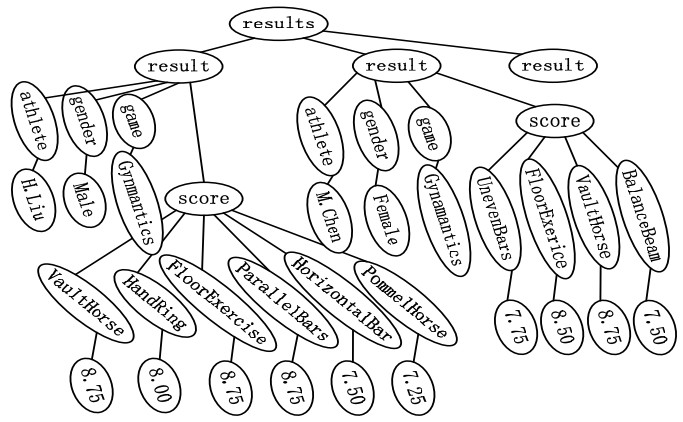
然而，分析表明，当XML数据集中同时出现XML函数依赖及类型相关的XML数据依赖时，XML数据模式的规范化理论遇到了新问题。北京工业大学计算机软件与理论实验室发现，对于XML数据，数据冗余问题不仅来自于不同数据项之间数据值与数据值之间的函数依赖关系，而且来自于多态数据的组织结构和其他数据的数值之间的依赖关系。XML数据的半结构化特征经常表现为具有相同标记的元素具有不同的组织结构，也就是具有多态性。

图 1‑1 保存比赛成绩的XML文档树

例如，图 1‑1给出了一个保存体操比赛结果的XML树。该树以results为树根，树中的各个result元素中保存的是一组体操比赛的成绩（即score元素）；在每个score元素又存有元素athlete、gender、game分别保存了某个运动员的名称、性别、比赛名称及其单项成绩。成绩元素score中为每个单项设置了一个元素，每个单项元素以项目名称为标记，保存了此项比赛的分数。由于针对不同性别的比赛可能具有不同的单项，导致score元素的结构有所不同。如图所示，男子体操比赛具有跳马、鞍马、单杠、双杠、自由体操和吊环等项目，而女子体操则具有跳马、平衡木、自由体操和高低杠等项目。不同的元素组织结构说明score元素具有多态性。而实际应用时的情况通常也是这样的，同样的比赛名和性别等数据会反复地出现在多个运动员的result元素中。此时如果某个比赛中设置的单项有所变化，则必须修改所有参加该比赛的运动员的score元素结构，否则将导致更新异常。这就是XML数据模式规范化理论遇到的新问题——类型相关的多态依赖问题。

在实际应用中，随处可见与上述例子类似的类型相关的多态依赖问题（即某些类型，它们的名称相同，但实际结构不同）。这个问题亟待解决。

## 研究现状

### 类型相关的依赖关系研究

目前，关于XML函数依赖的现有研究普遍沿用关系库理论中函数依赖的概念，虽然研究了各种XML数据项之间的函数依赖关系，且普遍考虑到了XML数据中的结构约束关系，进而讨论函数依赖的推理和规范化设计问题，但是都没有涉及到XML数据类型，所以就无法避免上述的数据冗余和更新异常问题（如只修改比赛单项时就需要修改所有运动员的score元素结构）[3][4]。北京工业大学计算机软件与理论实验室认为，良好的XML模式设计需要依靠一种新型的XML完整性约束来分析和解决这种XML数据特有的数据冗余问题，规范化的XML数据库设计理论的发展有必要研究此类类型相关的依赖关系。

为此，北京工业大学计算机软件与理论实验室针对上述各种类型相关的XML数据依赖关系给出了一种形式化定义，且将这些依赖关系与XML函数依赖汇总起来，提出了一组新型的XML数据依赖定义，用来描述XML数据项之间的这种数据值和数据类型的依赖关系。针对根据不同场景的依赖关系，分别定义XML元素与其成员元素之间的类型/数值依赖关系和数值/类型依赖关系，以及不同的XML元素之间的类型/数值依赖关系和数值/类型依赖关系；进而将这些类型相关的XML依赖关系与XML函数依赖汇总为一种通用XML依赖关系。北京工业大学计算机软件与理论实验室希望可以实现一种解释器，用于将这些类型相关的依赖关系转化成对应的内部数据结构，为后续实现基于DOM的XML类型相关的数据依赖检查算法做准备。

### 文法解释器的实现技术

上述解释器的实现首先需要对XML依赖文法构成的XML依赖描述进行解析，即需要实现一个解析器，这需用到编译原理技术的相关知识[5]。目前，编译原理的主要技术分为以下几种：

* 自顶向下语法分析：预测分析法、递归下降分析法等；
* 自底向上语法分析：算符优先分析法、LR分析法等；

## 任务及主要工作

北京工业大学计算机软件与理论实验室为XML的数据依赖设计了一种通用XML依赖关系，其中包含数值相关的函数依赖关系及类型相关的依赖关系；且提出了基于DOM的XML类型相关的数据依赖检查算法并已经实现检查原型系统。虽然针对算法的检查原型系统已经实现，但尚未实现对XML完整性约束描述的解析器来将用户提出的约束描述转换为系统识别的内部数据对象。因此需要一个方便且强大的解释器。

本课题的任务是，设计并实现该解析器中的词法分析及文法分析部分。主要工作如下：

* 第2章相关技术介绍中主要说明的是本课题中所有需要用到的技术名称、研究内容、以及一些术语的解释；
* 第3章文法设计中介绍了为实现本课题而设计的文法；
* 第4章词法分析及文法分析的设计中介绍了根据上述文法，为了实现本课题而进行的分析程序设计；
* 第5章实现及实验中介绍了本课题程序的具体实现，及实现后的实验和测试部分。
* 最后是总结和致谢部分。

## 目的及意义

本课题的目的在于能够正确地解析用户提出的XML完整性约束描述，该描述文法依据北京工业大学计算机软件与理论实验室设计的类型相关的依赖关系；将解析的结果转化成对应的内部数据结构，为后续实现基于DOM的XML类型相关的数据依赖检查算法及实验系统做准备。

# 相关技术介绍

## XML语言

XML文档是一种半结构化的、以数据为中心的、树形结构的文档。简单而言，它就是一种带有标签的文档。如果将它里面所有的标签都去掉，那它和我们日常所写的文章、表格就没有任何区别。

XML可以用一种符合用户需求的格式来存储和组织几乎任何信息。

* + XML使用Unicode作为标准字符集，支持大量的书写系统和符号。
  + XML提供了语法规则、内部链接检查、数据类型等多种方法来检验文档的质量。
  + XML具有清晰简单的语法和无歧义的结构，人和程序都能很轻松地阅读、解析。
  + XML易于和样式表组合，以便以任何用户想要的样式来创建格式化文档。

使用XML编写的文档，主要由“标签”和“内容”两部分组成，“标签”和“内容”合在一起包含了文档的所有信息。其中“标签”是一些添加到文档中的信息，用以标识文档的各个部分以及各部分之间的关系；而“内容”是读者关注的具体信息。表 2‑1是一个典型的XML文档：

表 2‑1 XML文档示例

|  |
| --- |
| <store>  <book year="2000">  <title>Data on the Web</title>  <publisher>Morgan Kaufmann Publishers</publisher>  <price>39.95</price>  </book>  <book year="1999">  <title>  The Economics of Technology and Content for Digital TV  </title>  <publisher>Kluwer Academic Publishers</publisher>  <price>129.95</price>  </book>  </store> |

XML中标签的作用就相当于这些注释，它以“〈*标签名*〉”开始，以“〈/*标签名*〉”结束，而“标签名*”*的作用就是告诉处理器，包含在这个标签里面的内容，究竟是什么含意，也就是说XML中标签表明了其存储内容的语义。

其实在我们日常的文字处理工作中，时时处处都在用这种“标签”：我们在行文中使用标点符号，划分段落，划分章节，都是一种无形的标签。因为XML文档主要是面向机器处理和信息定位的，所以文档里面能被添上标签的地方都被添上了标签。因此，计算机能够很容易地处理XML文档：通过这种格式编排数据，计算机能很方便地将字符流转换成树状结构，支持查询，数据交换，集成等功能。本课题的主要工作就是将字符罗转换成树状结构。同时，通过XSLT，计算机还可以将XML文档转换成人们比较习惯的格式供人们阅读[1]。

## XML正规式

北京工业大学计算机软件与理论实验室提出了一种通用的XML依赖关系，在该依赖关系中，人们不仅可以声明数值依赖，同样可以声明类型相关的依赖关系，这一点需要能够在XML数据类型描述中有效地解决XML中某些“标签”的多态性问题（即同一名称的“标签”，其具体结构不同）。而W3C提出的用于规范XML文档类型的XMLSchema是不能够解决此问题的。因此本文中使用XML正规式用于描述XML文档数据的类型。因此，文本中XML依赖关系的形式化定义采用的是基于路径式的XML数据定位方法，以及基于正规式类型的分类描述方法[6][7]。表 2‑2是一个XML正规式示例：

表 2‑2 XML正规式

|  |
| --- |
| 1. typedef **Male-Game**  := vauling\_horse[Double], hand\_ring[Double],   floor\_exercise[Double], parallel\_bars[Double],  hirozontal\_bar[Double], pommel\_horse[Double];  typedef **Female-Game** := uneven\_bars[Double], floor\_exercise[Double],  vaulting\_horse[Double], balance\_beam[Double];   1. results[ result[ team[String], gender[String], game[String] 🡪   score[ **Male-Game** | **Female-Game** ]]\*]; |

表 2‑2中所示的XML正规式与图 1‑1相对应。(1)部分的意思是说，标签“Male-Game”下分“vauling\_horse”、“hand\_ring”、“floor\_exercise”、“parallel\_bars”、“hirozontal\_bar”、“pommel\_horse”6个子标签；标签“Female-Game”下分“uneven\_bars”、“floor\_exercise”、“vaulting\_horse”、“balance\_beam”4个子标签；且规定这10个子标签的“内容”需以Double形式给出。(2)部分是说标签“results”下分多个子标签“result”，标签“result”下分“team”、“gender”、“game”、“score”4个子标签，且规定这4个子标签的“内容”需以String形式给出[8]。

## 递归下降分析法

### 思想及优缺点介绍

编译技术中的所谓递归下降分析法，是指根据各个候选式的结构，为文法的每个语法变量编写一个处理子程序，用来识别该语法变量所代表的语法成分。

设有正规式A-->X1X2…Xk…Xn，则与A对应的处理子程序遇到的Xk是终结符是直接进行匹配，耳语道Xk是语法变量时就掉用于Xk对应的处理子程序。由于正规式中的语法变量可能是递归定义的，所以这种实现方法要求处理子程序可以递归调用。另外，这种分析方法也是寻求输入串的一个推导过程（最左推到），所以称为递归下降分析法。

如果编译程序的实现语言允许进行过程的递归调用，则可以采用一般形式的递归下降分析法，称为递归子程序法。

对于词法分析程序的设计者来说，状态转换图是非常有用的设计工具。其实，状态转换图同样可以用来帮助建立文法分析程序。当然，文法分析程序的状态转换图和词法分析程序的状态转换图有着明显的区别，语法分析程序的状态转换图简称语法图，它具有如下特点。

* 每个语法变量对应每一个语法图，边上的标记是符号和语法变量。
* 符号上的转换意味着如果该符号是下一个输入符号，就进行相应的转换。
* 语法变量A上的转换是指调用A所对应的过程。
* 按照递归分析法，这种图同时也表达出了一个语法成分的处理过程。
* 语法图中的状态实际上可以被看成是对一个语法成分进行分析的进展情况。

基于语法图的语法分析器以语法图为基础来完成语法分析工作，其工作方式如下：

* 初始时，分析器进入语法图的开始状态，输入指针指向输入符号串的第一个符号。
* 经过一些动作后，它进入状态s，且从状态s到状态t有一条标记为X的边。如果X是一个（输入）符号a，此时下一个输入符号又正好是a，则分析器将输入指针向右移动一位，并进入状态；如果X是一个语法变量A，则分析其进入A的初始状态，但不移动输入指针。一旦到达A的终止状态，则立刻进入状态t，表明通过“调用”A的分析过程而完成了对A的分析。反映在当前的语法图上，相当于它已经从输入符号串中“读入”了A，所以分析器从状态s转移到状态t。
* 如果从s到t有一条标记为ɛ的边，那么分析器从状态s直接进入状态t而不移动输入指针。

如果给定文法的语法图是确定的，即一个状态对于某一个输入只有一个转换，则上述方法是有效的。

递归子程序法同样要求所处理语言的文法必须是LL(1)文法，这种方法的主要优点是易于实现。此外，由于分析器和文法的紧密对应性，这种方法还易于保证语法分析器的正确性。其缺点是频繁的递归调用会降低分析器的效率，而且与预测分析法相比，这种分析器的代码规模非常大。尽管如此，递归子程序法仍然是一种有效地语法分析方法，许多高级程序设计语言的编译系统都采用这种语法分析方法[8]。

### 与本课题的适用性

本课题的任务是实现一个能够根据XML中数据和类型相关的依赖关系，将用户提出的约束描述转换为系统识别的内部对象的方便且强大的解释器。因为用户提出约束描述的目的是说明一个XML语言下的结构，而XML语言又是由“标签”“内容”两部分组成，且存在许多“标签”和“标签”之间的嵌套关系，所以该解释器要想仅采用编译技术中的有限状态自动机技术来实现，不仅不容易实现，而且容易使得解释过程变得越来越混乱，出现许多意想不到甚至不知如何解决的问题。相比有限状态自动机技术来说，递归下降分析法由于是根据语法变量，将解释过程分解为多个处理子程序，每次解释完一个子程序即永久性解释完毕，解释器无须再返回来，这样不仅容易实现，且易于调试。

除此之外，递归子程序法中的每个语法变量或语法成分完全可以一一对应于XML中的每个“标签”，实现语法变量或语法成分与“标签”的一一映射。并且递归下降分析法的这种“分部解释性”使其在输出解释结果时很容易实现分部输出，可以非常方便的实现想输出哪个语法变量就输出哪个语法变量的效果，而这恰好对应于该解释器最终输出应为内部数据结构（所有的语法变量一起输出）的要求，不仅可以连起来输出，甚至可以仿照XML树状结构的样式输出。所以，本人使用递归下降分析法设计及实现本课题。

# 文法设计

文法是编译系统核心，是编译程序运行的基本依据。没有文法就无法进行系统的设计及实现，故在进行系统的具体设计及实现前，需要设计在本课题系统中使用的文法。有了文法，系统不但可以识别输入中的各个部分，更可以更确切地规范用户输入。整体来说，本文法包含四个部分：类型声明、类型定义、依赖声明、依赖实例，现展开说明。

## 文法总体概述

**类型绑定**

1. QNAME ::= STRING (:表示任意字符串:)
2. NAMETEST ::= STRING (:表示任意字符串:)
3. TYPEDEFINE ::= "*typedef"* QNAME ":=" CONSTYPE";" //type define

**类型声明**

1. TYPEDEC ::= TYPE "*;*" //type declaration
2. TYPE::=NAMETEST"*[*"SIMPLETYPE|CONSTYPE"*]*"WILDCARD?"*;*"//type
3. WILDCARD ::= "*?*"|"*+*"|"*\**" //wildcards
4. SEQTYPE ::= "*(*"CONSTYPE"*)*"WILDCARD //sequence type
5. **CONSTYPE ::=(SEQTYPE | UNIONTYPE)(** "***,***"**CONSTYPE)\***

**//concatenate type**

1. UNIONTYPE::=TYPE|QNAME ("*|*"UNIONTYPE)\* //union type
2. SIMPLETYPE ::= "*Double*" | "*Integer*" | "*String*" //simple type

**依赖声明**

1. PATH::=NAMETEST("*/*"NAMETEST)\* //path
2. TYPEOF ::= "*typeof*""*(*" PATH "*)*" /type of
3. CONSPATH ::= PATH（"*,*"PATH）\* //concatenate path
4. CONSTYPEOF::=TYPEOF ("*,*" TYPEOF)\* //concatenate typeof
5. DEPEND ::=

“*dependency*” ( PATH "*:*")? "*{*"CONSPATH| CONSTYPEOF | CONSPATH "*,*" CONSTYPEOF"*🡪*" CONSPATH| CONSTYPEOF | CONSPATH "*,*" CONSTYPEOF "*}*" "*;*" //depend

**依赖实例**

1. CONSSTR::= " *‘* "STRING" *’* " ("*,*"" *‘* "STRING" *’* ")\* //concatenate strings
2. DEPENDINS::=

"*instance*""*{*" CONSSTR | CONSTYPE | CONSSTR"*,*"CONSTYPE "*🡪*"CONSSTR | CONSTYPE | CONSSTR"*,*"CONSTYPE"*}*""*;*" //dependent instance

表 3‑1中所示文法示例是一个根据上述文法所写出的实际例子：

表 3‑1 总体文法示例

|  |
| --- |
| *typedef* ***Male-Game***  *:= vauling\_horse[Double], hand\_ring[Double],*  *floor\_exercise[Double], parallel\_bars[Double],*  *hirozontal\_bar[Double], pommel\_horse[Double];*  *typedef* ***Female-Game*** *:= uneven\_bars[Double], floor\_exercise[Double],*  *vaulting\_horse[Double], balance\_beam[Double];*  *results[ result[ team[String], gender[String], game[String] 🡪*  *score[* ***Male-Game*** *|* ***Female-Game*** *]]\*];*  *dependency results/result: {gender, game 🡪 typeof( score )};*  *dependency {results/result/gender,results/result/game 🡪 typeof(results/result/score)};*  *instance {‘Male’, ‘gymnastics’ 🡪 score [Male-Game]};*  *instance {‘Female’, ‘gymnastics’ 🡪 score [Female-Game]};* |

## 类型绑定及类型声明文法释义

在本文法中为了方便用户使用，需要允许将类型的名称与类型绑定在一起（就像在使用一个对象之前需要先声明它名称、类型一样），这一部分是通过“类型绑定”输入部分实现的，它允许声明多个类型绑定。而“类型声明”部分主要负责声明各种可能出现在输入流中的类型、它的路径及其数量等，其中数量使用通配符WILDCARD表示。由于类型声明中会使用到类型绑定中的新类型，故将这两个部分一起在表 3‑2中说明。

下面图 3‑1、图 3‑2及图 3‑3给出根据类型绑定及类型声明部分的文法示例，并给出该示例根据本文法生成的文法生成树结构。

**类型绑定**：

*typedef Male-Game := vauling\_horse*[*Double*]*,hand\_ring*[*Double*]*,*

*floor\_exercise*[*Double*]*, parallel\_bars*[*Double*]*,*

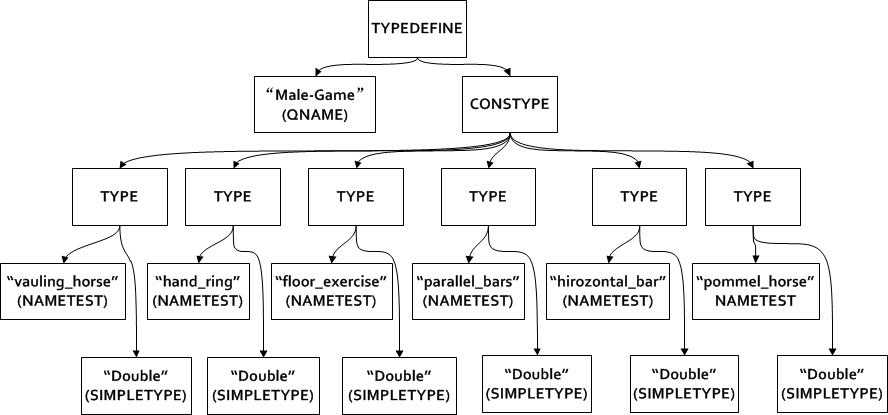
*hirozontal\_bar*[*Double*],*pommel\_horse*[*Double*];

图 3‑1 “类型绑定”Male-Game文法生成树

*typedef Female-Game :=uneven\_bars*[*Double*]*, floor\_exercise*[*Double*]*,*

*vaulting\_horse*[*Double*]*, balance\_beam*[*Double*];

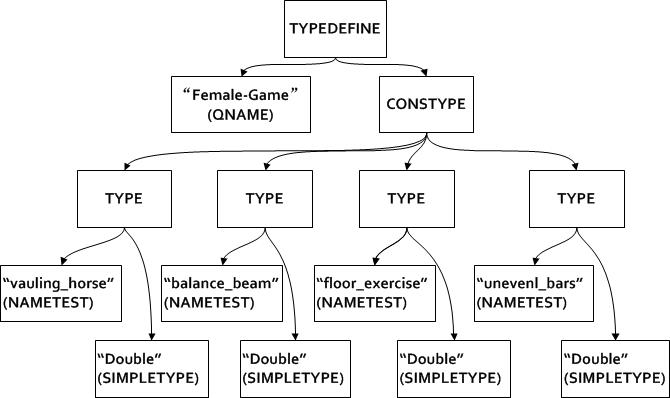


图 3‑2 “类型绑定”Female-Game文法生成树

**解释及说明**：

在图 3‑1及图 3‑2中，根据文法第(3)条，系统将处在关键字“*typedef*”之后的字符串判断为“类型绑定”中新定义的类型的名称，在图 3‑2中为“Male-Game”，在图 3‑2中为“Female-Game”。系统将关键字“*:=*”后至关键字“*;*”前的部分判断为高级复合类型CONSTYPE型。

根据文法第(8)条，判定图 3‑1中CONSTYPE型应被解释为6个UNIONTYPE型，图 3‑2中CONSTYPE型应被解释为4个UNIONTYPE型。

根据文法第(9)条，系统判定图 3‑1及图 3‑2中每个UNIONTYPE型应被解释为1个TYPE型。而由于UNIONTYPE中仅持有个TYPE型，为了简化该文法生成树的结构，故将图 3‑1中6个UNIONTYPE型直接移除，其解释成的共6个TYPE型，直接作为CONSTYPE型的直属下级，接在CONSTYPE型下，形成CONSTYPE型被解释为6个TYPE型的效果，将图 3‑2中的4个UNIONTYPE型变成4个TYPE型。

根据文法第(5)条，每个TYPE型被解释为图 3‑1或图 3‑2中所示样式。

**类型声明**：

*results*[ *result*[ *team*[*String*]*, gender*[*String*]*, game*[*String*]*,*

*🡪 score*[ *Male-Game | Female-Game* ]*\** ];

**解释及说明：**

在图 3‑3中，根据文法第(4)条，关键字*“;*”前的部分被解释为TYPE型。

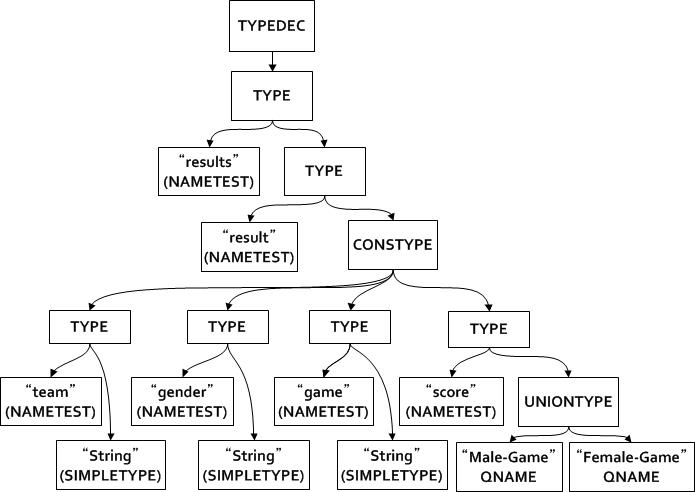
根据文法第(5)条，关键字“results”后“*[*”及最后一个“*]*”中间的部分被解释为CONSTYPE型。根据文法第(8)条及第(9)条，CONSTYPE型被解释为UNIONTYPE型，UNIONTYPE型被解释为TYPE型。其中由于CONSTYPE型及UNIONTYPE型均只持有1个子类型，故将该处的CONSTYPE及UNIONTYPE型移除，即将“*[*”及最后一个“*]*”中间的部分解释为TYPE型。关键字“result”后“*[*”及倒数第二个“*]*”中间除最后一个TYPE型外其余的部分同样使用此方法解释。在最后一个TYPE型中，CONSTYPE型被解释为UNIONTYPE型，UNIONTYPE型被解释为两个QNAME型，即在图 3‑1及图 3‑2中新定义的类型绑定。CONSTYPE型由于只持有一个UNIONTYPE型，故被移除，UNIONTYPE型位置上移。

图 3‑3 “类型声明”文法生成树

根据文法第(5)条，每一个TYPE型均被解释为图 3‑3中所示样式。

## 依赖声明及依赖实例文法释义

在“依赖声明”部分，用户可以自定义多种依赖关系，依赖关系是后面所写依赖实例的基础，意在表述出一种“根据A与B……可确定C”的关系。“依赖实例”是指将上述依赖声明实例化，即给出相应依赖关系的实际例子，对应关系为：第一个依赖声明对应第一个依赖实例，第二个依赖声明对应第二个依赖实例……以此类推。由于依赖关系是后面所写依赖实例的基础，所以依赖关系必须与依赖实例在类型上严格一一对应，故将这两个部分一起在表 3‑3中说明。

表 3‑2 类型绑定及类型声明文法逐条释义

|  |  |
| --- | --- |
| 类型绑定及类型声明 | |
| 文法标号 | 文法释义 |
| (1) | QNAME是由任意字符串定义的，在TYPEDEFINE中该字符串将作为新类型的名称使用。 |
| (2) | 同QNAME一样，NAMETEST也是由任意字符串定义的，在TYPE中该字符串将作为TYPE类型的名称使用。 |
| (3) | 该行文法用于处理用户提出的“类型绑定”部分，该行文法意为：将名称为QNAME的类型绑定为CONSTYPE类型，CONSTYPE类型在“类型声明”部分定义。 |
| (4) | 该行文法用于处理用户提出的“类型声明”，其主要部分为简单TYPE类型，*";"*代表用户输入应以*";"*结尾。 |
| (5) | 简单TYPE类型的定义，NAMETEST为该对象的名称，*"["*和*"]"*中间所夹部分为这个对象的类型，SIMPLETYPE代表原始类型，CONSTYPE代表高级复合类型。 |
| (6) | 通配符共有三种可能：  *"\*"*：该符号所跟类型有0-N个；  *"?"：*代表该符号所跟类型有0-1个；  *"+"：*代表该符号所跟类型有N（N不可为0）个； |
| (7) | **CONSTYPE代表高级复合类型，高级复合类型是由多种初级复合类型组成的，在每两个初级复合类型之间"*,"*连接。** |
| (8) | SEQTYPE代表其中一种初级复合类型，它是由一个高级复合类型和通配符组成的，其中高级复合类型需要用*"("")"*括起来。 |
| (9) | UNIONTYPE代表另一种初级复合类型，它是由多个简单TYPE类型或“类型绑定“中声明的新类型组合而成的，在每两个TYPE或新类型之间用*"|"*连接。 |
| (10) | SIMPLETYPE代表原始类型，其下分为三种子类型：  Double型：对应基本数据类型里的double型；  Integer型：对应基本数据类型里的int型；  String型：对应基本数据类型里的String型； |

下面给出根据依赖声明及依赖实例部分的文法示例，并给出该示例根据本文法生成的文法生成树结构。

**依赖声明1**：

*dependency results/result: {gender, game* 🡪 *typeof( score )};*

**依赖声明2**：

*dependency {results/result/gender,results/result/game*🡪

*typeof(results/result/score)};*

表 3‑3 依赖声明及依赖实例文法逐条释义

|  |  |
| --- | --- |
| 依赖声明及依赖实例 | |
| 文法标号 | 文法释义 |
| (11) | 普通类型PATH代表路径，由于依赖关系中的每个部分都必须以路径的形式给出，故PATH部分用于表示该路径。它由一个或多个NAMETEST组成，每两个NAMETEST间用"*/*"相连，NAMETEST中的内容可自定义。 |
| (12) | 普通类型TYPEOF只由PATH组成，意为该PATH所指向的类型。它由"*typeof*"后跟PATH组成。 |
| (13) | CONSPATH是一种过渡性类型，由一个或多个PATH组成，是一个复合型结构，目的是方便系统处理。每两个PATH之间用“*,*”相连。 |
| (14) | CONSTYPEOF是一种过渡性类型，由一个或多个TYPEOF组成，是一个复合型结构，目的是方便系统处理。每两个TYPEOF之间用“*,*”相连。 |
| (15) | DEPEND定义的是依赖关系的整体结构，它集合了PATH、CONSPATH、CONSTYPEOF等结构，主要目的是规范依赖关系的格式。需要说明的是，"(PATH "*:*")?"部分为"前缀PATH"，它的出现说明该PATH部分是后续所有CONSPATH及CONSTYPEOF的公共前缀，若无公共前缀存在，则不出现"前缀PATH"部分；且"*🡪*"前后均不可以为空，但可以只有CONSPATH或CONSTYPEOF任意一种结构。 |
| (16) | CONSSTR是一种过渡性类型，其由一个或多个字符串组成（STRING代表任意字符串，且每个字符串需以" *‘* "开头，以" *’* "结尾），每两个字符串之间用"*,*"相连，是一种复合型结构。 |
| (17) | DEPNDINS定义的是依赖实例的整体框架，它集合了CONSSTR、CONSTYPE等结构，主要目的是规范依赖实例的格式。值得注意的是，“*🡪*”前后均不可以为空，但可以只有CONSSTR或CONSTYPE任意一种结构。 |

**解释及说明：**

在图 3‑4及图 3‑5中，根据文法第(15)条，图 3‑4中关键字“*:*”前的部分被解释为PATH型，关键字“*{*”关键字“*🡪*”前的部分被解释为CONSPATH型，关键字“*🡪*”后关键字“*}*”前的部分被解释为CONSTYPEOF型；图 3‑5中关键字“*{*”后关键字“*🡪*”前的部分被解释为CONSPATH型，关键字“*🡪*”后关键字“*}*”前的部分被解释为CONSTYPEOF型。

根据文法第(13)条及第(14)条，图 3‑4及图 3‑5中的CONSPATH型均被解释为两个PATH型，CONSTYPEOF型被解释为1个PATH型。

根据为您发第(11)条，PATH型被解释为多个NAMETEST型，NAMETEST的具体内容如图 3‑4或图 3‑5中所示。

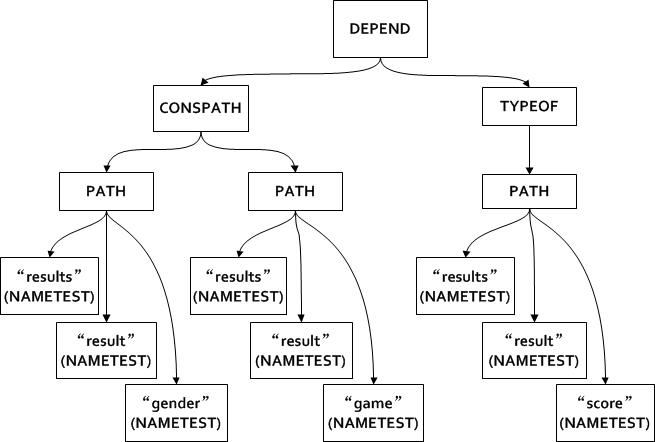
**依赖实例：***instance {‘Male’, ‘gymnastics’* 🡪 *score* [*Male-Game*]};

图 3‑5 “依赖声明”2文法生成树

**解释及说明：**

图 3‑6 “依赖实例”文法生成树

在图 3‑6中，根据文法第(17)条，关键字“*{*”后关键字“*🡪*”前的部分被解释为CONSSTR型，关键字“*🡪*”后关键字“*}*”前的部分被解释为CONSTYPE型。

根据文法第(16)条，CONSSTR型被解释为两个STRING型，STRING型的具体内容如图 3‑6中所示。

根据文法第(8)条，CONSTYPE型被解释为1个UNIONTYPE型，且根据文法第(9)条，UNIONTYPE型被解释为1个TYPE型，由于CONSTYPE和UNIONTYPE里都只持有1个类型，故将CONSTYPE和UNIONTYPE移除，关键字“*🡪*”后的部分被直接解释为TYPE型。

根据文法第(5)条，TYPE型被解释为图 3‑6中所示样式。

# 词法分析及文法分析的设计

本解析器主要分为两部分实现：词法分析部分、文法分析部分。

词法分析部分（即词法分析器Lexer）负责将用户输入流划分为不同的token，并传递给文法分析部分。

文法分析部分（即文法分析器Parser）负责根据Lexer划分出的不同token判断用户提出的完整性约束是否正确，是否符合文法。若符合文法，则生成期望的内部数据结构，若不符合则提示错误。文法分析部分工作时需要不断转换至不同的状态，这些状态是根据文法确立的。

## 输入输出设计

词法分析器Lexer和文法分析器Parser的输入输出有部分是在彼此间传递的，也有部分是从解释器外部得到，或向解释器外部传递。

### 解释器输入输出概况

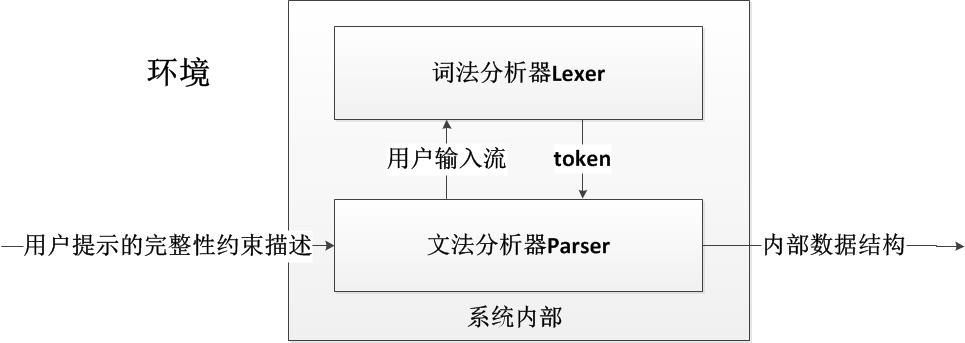
下图以整体的角度说明了本解释器整体与外部、两个部分（词法分析部分和文法分析部分）之间的输入输出关系。

图 4‑1 系统输入输出概况示意图

本解释器从外部环境获取用户编写的完整性约束描述，在文法分析器Parser中进行处理，形成用户输入字符流，该输入字符流随后被传递至词法分析器Lexer中。Lexer分析该输入流，获取不同的token，并传递给Parser供其进行状态转换使用。在状态转化过程中形成内部数据结构。当词法分析器、文法分析器均分析结束时，本解释器向外部环境传递该内部数据结构。

### 词法分析的输入输出

词法分析器Lexer的主要输入源就是由文法分析器Parser处理完成后得到的用户输入流，其格式要求为：

* 单词与单词之间必须有空格符“ ”，其余地方不存在空格符；
* 不存在换行符；
* 流中其余字符与用户输入字符一致。

词法分析器Lexer的输出为不同的数字，这些数字代表不同的token，且作为文法分析器Parser的输入，供状态转换时使用。

### 文法分析的输入输出

文法分析器Parser的主要输入源为用户编写的完整性约束描述。Parser需要通过处理用户编写的完整性约束描述来产生用户输入流，其中主要操作包括消去空格符、换行符等。

**用户编写的完整性约束描述格式要求：**

* 全部字符均需为半角字符输入；
* 需以类型绑定、类型声明、依赖声明、依赖实例四个部分按顺序组成；
* 类型绑定：以“*typedef*”单词开头；与类型名称间有空格符；类型名称后跟符号“*:=*”；以“*;*”结束；以上内容可重复多次。

示例：*typedef* ***Male-Game*** *:= vauling\_horse*[*Double*]*, hand\_ring*[*Double*]*,*

*floor\_exercise*[*Double*]*, parallel\_bars*[*Double*]*,*

*hirozontal\_bar*[*Double*], *pommel\_horse*[*Double*];

*typedef* ***Female-Game*** *:= uneven\_bars*[*Double*]*, floor\_exercise*[*Double*]*,*

*vaulting\_horse*[*Double*]*, balance\_beam*[*Double*];

* 类型声明：每种类型名称需后跟符号“*[*”及“*]*”（“*[*”“*]*”需个数对应）；“*]*”后可跟符号“*\**”或“*?*”或“*+*”；最后以“*;*”结束；以上内容不可重复。

示例： *results[ result[ team[String], gender[String], game[String],*

*score[ Male-Game | Female-Game ]\* ]];*

* 依赖声明：以“*dependency*”单词开头；不存在“*:*”时“*dependency*”后需跟符号“*{*”；若存在“*:*”时则“*:*”后需跟符号“*{*”；类型间需以“*/*”分隔；多个部分以“*,*”分隔；需要有“*🡪*”；每种类型需以“*typeof*”单词开头；以“*}*”、“*;*”结束，以上内容可重复多次。

示例： *dependency results/result:{ gender, game* 🡪 *typeof( score )};*

*dependency {results/result/gender,results/result/game*

🡪*typeof(results/result/score)};*

* 依赖实例：以“*instance*”单词开头；后跟“*{*”；各“值”需前跟“*‘*”后跟“*’*”；多个部分以“*,*”分隔；需要有“*🡪*”；每种类型名称需后跟“*[*”及“*]*”（“*[*”“*]*”需个数对应）；以“*}*”、“*;*”结束；以上内容可重复多次。

示例： *instance { ‘Male’, ‘gymnastics’* 🡪 *score* [*Male-Game*]};

*instance {‘Female’, ‘gymnastics’* 🡪 *score*[*Female-Game*]};

文法分析器Parser的输出有两种：供Lexer使用的用户输入流，以及供系统中后续程序使用的内部数据结构。其中内部数据结构中主要包括以下四项：

* 类型绑定数组：将用户输入的一个或多个基本声明（“*typedef*”……）拆分成相应的数据结构，并存储其中；
* 类型声明变量：存放根据用户输入的一个类型相关合并而成的数据结构；
* 依赖声明数组：将用户输入的一个或多个依赖声明（“*dependency*”……）拆分成相应的数据结构，并存储其中；
* 依赖实例数组：将用户输入的一个或多个依赖实例（“*instance*”……）拆分成相应的数据结构，并存储其中。

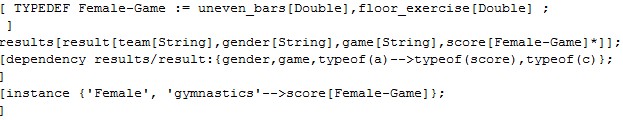
由于毕业设计时间有限，故仅将将该内部数据结构以打印的形式展示出来，以供检验是否正确生成内部数据结构，后继的XML依赖验证算法实现交由他人负责。输出示例如图 4‑2所示：

图 4‑2 文法分析器Parser输出示例

## 词法分析及文法分析的任务

### 词法分析

词法分析部分，又称词法分析器Lexer，其主要任务是将用户输入字符流划分为不同的token，并传递给文法分析部分使用。词法分析器的每次调用只识别一个token，且每次传递只传递一个token，共能识别28种token（含输入结束符end）。

词法分析器划分token是通过进行逐个字符的识别完成的。其中，单字符形的符号，如“*{*”及“*}*”等，识别为单独token；单词或字符串的划分是通过多个单一字符的拼接完成的，不同的token保存为不同的数字，以供区分。

Lexer作为词法分析器，其工作仅负责将用户的输入划分出不同的token，不考虑该输入是否正确，是否符合文法等问题。而判断用户输入的正确与否，以及将用户的输入进行逻辑拆分，进而生成后续课题中所用的内部数据结构等工作均由文法分析部分负责。这样就不可避免地需要词法分析部分和文法分析部分相互“通信”，该“通信”传递的信息即为由词法分析器划分出的每个token。表 4‑1详细给出了词法分析器Lexer可识别的所有关键字。

### getToken及nextToken操作

在词法分析器Lexer中，有两个重要操作：getToken操作及nextToken操作。其中，getToken操作的主要工作是获取当前token，而nextToken操作的主要工作是探测下一个token（还未获取，即将获取）。

表 4‑1 Lexer可识别的所有关键字

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 单字符token | : | / | , | ‘ |
| [ | ; | ] | } |
| ( | | | ) | + |
| - | \* | { | ？ |
| 多字符token | := | 🡪 | typedef | Double |
| String | Integer | typeof | dependency |
| instance | nametest | end |  |

若使用getToken操作，则获取的token即为分析器当前token，若使用nextToken操作，仅探测出下一个准备获取的token，如要识别为当前token需要调用getToken操作。对此，本人设计了一个数值型指针cur\_ptr，来实现这种区别。

由于词法分析器Lexer对用户输入流的操作是以字符为单位的，故本人对指针cur\_ptr进行这样的设计：每一次获取字符都从指针cur\_ptr所指位置开始；每获取一个字符后，指针cur\_ptr都前移一位；如果cur\_ptr不做前移，那么下一次获取的字符还是当前已获取的这个字符。当使用getToken操作时，指针cur\_ptr前移，当使用nextToken操作时，指针cur\_ptr不移动。每次使用getToken操作或nextToken操作时，在输入的字符流中获取token的起点都是该指针所指位置。

### 文法分析

文法分析部分（即文法分析器Parser）主要任务是根据Lexer划分出的不同token判断用户提出的完整性约束是否正确，是否符合文法。若符合文法，则生成期望的内部数据结构，若不符合则提示错误。

文法分析器Parser工作时需要不断转换至不同的状态，这些状态是根据文法确立的。Parser会在状态间转换的同时，根据文法，完成对token的匹配，判断用户编写的完整性约束描述是否正确，从而决定是否生成期望的内部数据结构。Parser的状态转换主要有以下两种方式：

* Parser根据从Lexer中获取到的token确定目标状态；
* Parser根据从Lexer中探测到的token判断是否符合转入唯一目标状态的条件，若符合即转入。

在某些状态完成后，已获得的某些token将被组合在一起形成已声明的内部数据结构。当所有token均被组合后，文法分析部分结束，解释器输出与用户编写的完整性约束描述对应的内部数据结构。

### 词法分析和文法分析的关系

除了上述中分别说明的词法分析和文法分析的主要特点，还有一点需要特别说明，那就是词法分析和文法分析的关系问题：词法分析和文法分析是同步且相辅相成的，一次词法分析会产生一次文法分析，而一次文法分析结束后又会自动进行下一次词法分析。当用户输入流中的所有token都被Lexer获取完毕，则词法分析部分不再使用。此时除去子程序结束时解释器因向上递归而产生的状态转换后，若还存在有多次文法分析（状态转换），则提示用户编写的完整性约束描述存在错误，否则视为成功解析。

## 词法分析及文法分析的结构

本解析器的结构设计是分两部分进行的：词法分析结构和文法分析结构。其中，词法分析结构主要采用自动机的结构模式设计，而文法分析结构主要依据设计的文法进行设计。

### 词法分析结构

词法分析的结构参照有限状态自动机结构设计，Lexer通过读取单一字符从而实现对用户输入流的操作。

有限状态自动机根据每次读取的单一字符确定目标状态，实现自动状态转换。该自动机的状态数量有限，共有30个状态（含初始状态），每个状态的转入条件均与其他状态不相同，从而排除了出现冲突或文法二义性的可能性。

图 4‑3中详细给出了Lexer的所有状态及其状态间转换的条件。值得注意的是，本有限状态自动机的状态虽然较多，但转换条件相对简单，所以几乎每个状态都可以是终结状态（即图中黄色粗线圆圈），但其中的第0、2、21状态为非终结状态（即图中黑色细线圆圈），即词法分析器Lexer不会在处于这3个状态时进行token的获取，否则即提示输入错误。

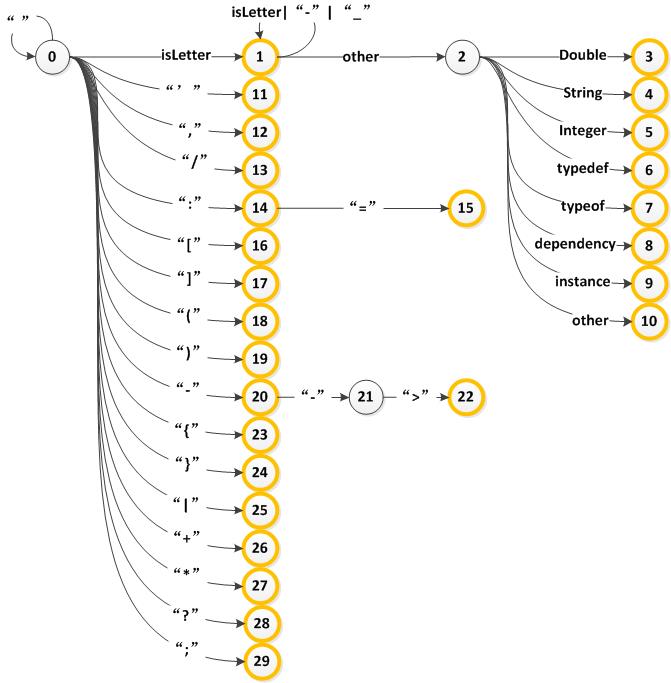


图 4‑3 Lexer结构设计

### 文法分析状态转换结构

文法分析的整体结构采用递归下降分析法设计，而每个子程序中采用有限状态自动机设计，并且与文法具有较强的对应关系，即将文法中的每个语法变量设计成该文法分析器中的一个子程序。当文法提示需要使用该结构时，Parser即进入该结构对应的子程序，Parser状态进入该子程序状态，子程序结束，Parser状态返回至调用该子程序的状态之中。Parser每次进行状态转换，都需要指示Lexer再获取一个token作为当前token。

图 4‑4至图 4‑17是本文法分析器Parser的状态转换图，简称语法图。其中

* 文法中的每个语法变量对应一个一个语法图，边上的标记是符号和语法变量。
* 符号上的转换意味着如果该符号是下一个输入符号，就进行相应的转换。
* 某一语法变量X上的转换是指调用X所对应的过程。
* 按照递归分析法，这种图同时也表达出了一个语法成分的处理过程。
* 语法图中的状态实际上可以被看成是对一个语法成分进行分析的进展情况。

例如：当其他子程序A调用某个子程序B时，Parser的状态由A中的某个状态转换到到B的初始状态，若B中还需调用子程序C，则调用时Parser的状态转换到C的初始状态，若B中不发生调用，则Parser的状态始终处在B状态中。当子程序B结束，Parser的状态转换至A中的后续状态。图 4‑4至图 4‑17中所用格式说明如下：

* 大写字母代表相应语法成分的处理过程，如“TYPEDEF”代表语法成分TYPEDEF的处理过程，即TYPEDEF子程序；
* 相邻状态间连线上的字代表进入目标状态的条件；
* 标明Nexttoken的地方表示探测下一个token（还未获取为当前token），未标明的表示已经获取为当前token；
* 紫色粗线圆圈代表文法分析结束，黄色粗线圆圈代表该子程序的可以在此状态时正常结束。

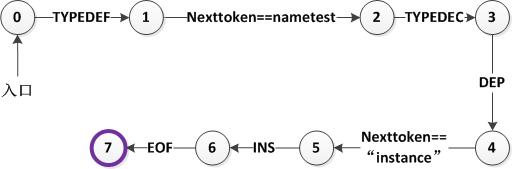


图 4‑4 Parser整体状态结构

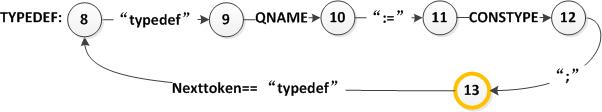
为使文法分析能够自动衔接不间断，则需有一个关于文法分析整体结构的设计，上图就是该文法分析的整体结构。TYPEDEF、TYPEDEC、DEP、INS四个子程序代表文法分析的主要语法成分的处理过程——相应的子程序。TYPEDEF代表“类型绑定”的处理过程，由于该过程的分析内容可以不是一个，而是多个类型绑定，故在TYPEDEF与TYPEDEC的衔接处需要进行nextToken操作。DEP与INS的衔接处与此含义相同。TYPEDEC代表“类型声明”的处理过程、DEP代表“依赖声明”的处理过程、INS代表“依赖实例”的处理过程。

图 4‑5 “类型绑定”整体结构

图 4‑5表示“类型绑定”部分的整体结构。Parser若处在黄色圆圈状态时nextToken操作所探测到的下一个token是“typedef”，则提示用户输入的类型绑定不止一个，需要循环处理，故Parser状态返回至TYPEDEF子程序的初始状态8。CONSTYPE表示系统此时应调用CONSTYPE自称需，Parser状态应转换到CONSTYPE子程序的初始状态。

图 4‑6 “类型声明”整体结构

图 4‑6表示“类型声明”部分的整体结构。由于类型声明部分只允许有一个类型声明，故该结构中不存在循环结构。TYPE表示Parser此时应调用TYPE子程序，Parser状态应转换到TYPE子程序的初始状态。

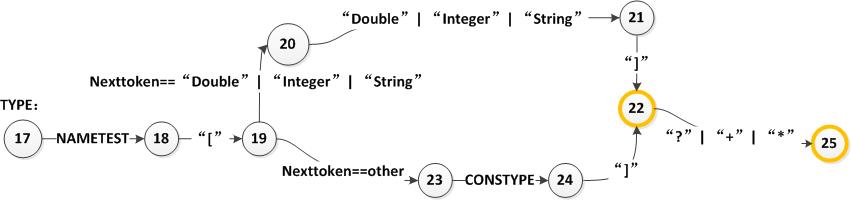


图 4‑7语法成分TYPE的整体结构

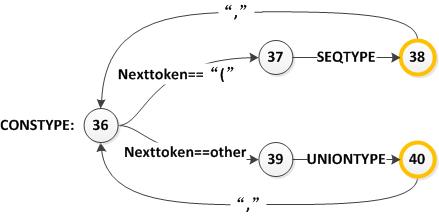
图 4‑7表示语法成分TYPE的整体结构，其中状态19处需要进行nextToken操作，所探测到的token可以用于辅助状态选择。CONSTYPE表示解释器此时应调用CONSTYPE子程序，Parser状态应转换到CONSTYPE子程序的初始状态。状态22及状态25提示均可以在该状态时结束TYPE子程序，区别在于若在状态22时遇到当前token为“?”|“+”|“\*”时，Parser转换至状态25结束TYPE子程序，若未遇到则在状态22结束TYPE子程序。图 4‑8表示语法成分CONSTYPE的整体结构。在该结构初始状态时需要进行nextToken操作，所探测到的token可以用于辅助状态选择，SEQTYPE表示Parser此时应调用SEQTYPE子程序，Parser状态应转换到SEQTYPE子程序的初始状态，UNIONTYPE表示Parser此时应调用UNIONTYPE子程序，Parser状态应转换到UNIONTYPE子程序的初始状态。

图 4‑8语法成分CONSTYPE的整体结构



图 4‑9语法成分SEQTYPE整体结构

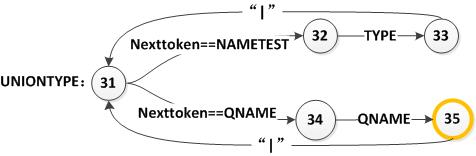
图 4‑9表示语法成分SEQTYPE的整体结构，CONSTYPE表示Parser此时应调用CONSTYPE子程序，Parser状态应转换到CONSTYPE子程序的初始状态。Parser需在状态30时结束SEQTYPE子程序。

图 4‑10 语法成分UNIONTYPE整体结构

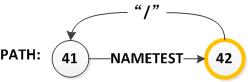
图 4‑10表示语法成分UNIONTYPE的整体结构。在该结构初始状态时需要进行nextToken操作，所探测到的token可以用于辅助状态选择。TYPE表示Parser此时应调用TYPE子程序，系统解释器应转换到TYPE子程序的初始状态。

图 4‑11语法成分PATH整体结构

图 4‑11表示语法成分PATH的整体结构，Parser若处在黄色圆圈状态时nextToken操作所探测到的下一个token是“/”，则提示用户输入的PATH尚未分析完毕，需要循环处理，故系Parser状态返回至PATH子程序的初始状态。

图 4‑12语法成分TYPEOF整体结构

图 4‑12表示语法成分TYPEOF的整体结构。PATH表示Parser此时应调用PATH子程序，Parser状态应转换到PATH子程序的初始状态。

图 4‑13语法成分ONSPATH整体结构



图 4‑13表示语法成分CONSPATH的整体结构，Parser若处在黄色圆圈状态时当前token为“,”且nextToken操作所探测到的下一个token不是“typedef”，则提示该处的“,”不是“依赖声明”中语法成分CONSPATH与CONSTYPEOF之间的“,”，而是CONSPATH内部的“,”，即PATH与PATH之间，故需要循环处理，故Parser状态返回至CONSPATH子程序的初始状态。

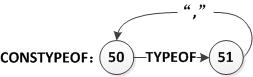


图 4‑14语法成分CONSTYPEOF整体结构

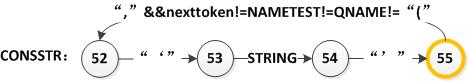
图 4‑14表示语法成分CONSTYPEOF的整体结构，Parser若处在黄色圆圈状态当前token为“,”，则提示尚有未分析的TYPEOF类型，需要循环处理，故Parser状态返回至CONSTYPEOF子程序的初始状态。

图 4‑15语法成分CONSSTR整体结构

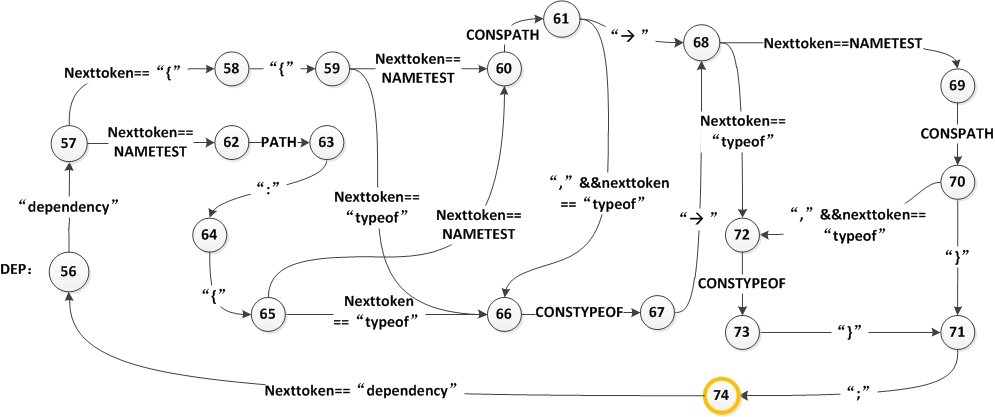
图 4‑15表示语法成分CONSSTR的整体结构，Parser若处在黄色圆圈状态时当前token为“,”且nextToken操作所探测到的下一个token不是NAMETEST也不是QNAME，则表明该处的“,””不是“依赖实例”中语法成分CONSSTR与CONSTYPE之间的“,”，而是CONSSTR内部的“,”，即“ ’ ”STRING“ ’ ”与“ ’ ”STRING“ ’ ”之间的，故需要循环处理，故Parser状态返回至CONSSTR子程序的初始状态。

图 4‑16 “依赖声明”整体结构

图 4‑16表示“依赖声明”部分的整体结构，在状态57、59、61、65、68、70处需要进行nextToken操作，所探测到的token可以用于辅助状态选择CONSPATH表示Parser此时应调用CONSPATH子程序，Parser状态应转换到CONSPATH子程序的初始状态，CONSTYPEOF表示Parser此时应调用CONSTYPEOF子程序，Parser状态应转换到CONSTYPEOF子程序的初始状态。

图 4‑17表示“依赖实例”部分的整体结构，在状态77、79、82、84处需要进行nextToken操作，所探测到的token可以用于辅助状态选择，CONSSTR表示Parser此时应调用CONSSTR子程序，Parser状态应转换到CONSSTR子程序的初始状态，CONSTYPE表示Parser此时应调用CONSTYPE子程序，Parser状态应转换到CONSTYPE子程序的初始状态。

### 错误处理机制

图 4‑17 “依赖实例”整体结构

在错误处理方面，由于本科毕设时间所限，我们令Parser仅简单的处理一种类型的错误，即token匹配错误。

token匹配错误，就是说当文法分析器Parser在解释文法，进行状态转换的过程中，发现由Lexer提供的当前token（或下一个token）不同于此时所处的处理子程序中期望的token，则Parser提示用户编写的完整性约束描述有错。提示方式为，输出错误提示语句，且输出当前token及下一个token的对应int型数值。

# 实现及实验

## 数据结构的设计及实现

由于文法分析器Parser的输出形式是供后续课题使用的内部数据结构，所以本人采用面向对象方法对该数据结构进行了设计，实现语言为Java程序设计语言。

该内部数据结构的总体设计思路是，为文法中出现的每种语法变量设计一种数据结构，这种数据结构的持有属性依据文法中对应部分的定义设计，例如：CONSTYPE结构在文法中是一个高级复合类型，它可持有多个UNIONTYPE类型、SEQTYPE类型，针对CONSTYPE的该种结构特征，表 5‑1是本人本人实现的语法变量CONSTYPE的数据结构：

表 5‑1 语法变量CONSTYPE的数据结构

|  |
| --- |
| **public** **class** CONSTYPE **implements** ITYPE{  **public** ArrayList<ITYPE> types = **new** ArrayList<ITYPE>();  **int** len;  } |

可以看到CONSTYPE类中持有两个属性，一个是数组types，另一个是该数组的长度len。数组types用于存放语法变量CONSTYPE中持有的UNIONTYPE或SEQTYPE等语法变量。

### 多态的体现

**ITYPE类型**

在表 5‑1所示的例子中，数组types持有的对象可以是UNIONTYPE类型或是SEQTYPE类型，不同的类型导致在设计数组types时，在已有的类型中找不到一个合适的类型，使其可以既覆盖UNIONTYPE又覆盖SEQTYPE的范围，故为此需要在UNIONTYPE及SEQTYPE等类型的基础上，再抽象出一个类型作为数组types的类型定义，这就是ITYPE类型。

**IPATH类型及ITYPEOF类型**

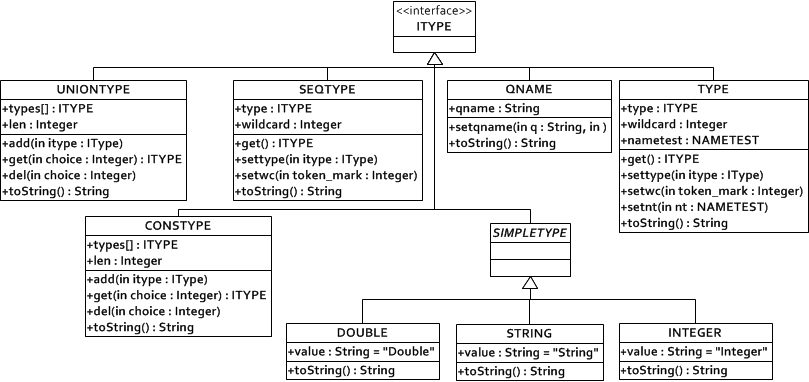
与ITYPE类型类似，由于语法变量DEP对应的类型中可以持有CONSPATH类型，也可以持有PATH类型（当CONSPATH中支持有一个PATH时，Parser使用PATH类型而不是CONSPATH类型），所以为了将该两种类型统一起来，也为了便于设计，本人特抽象出一个IPATH类型来被CONSPATH及PATH实现。同理，由于语法变量DEP对应的类型中还可以持有CONSTYPOEF类型，也可以持有TYPEOF类型（当CONSTYPEOF中支持有一个TYPEOF时，Parser使用TYPEOF类型而不是CONSTYPEOF类型），所以为了将该两种类型统一起来，也为了便于设计，本人特抽象出一个ITYPEOF类型来被CONSTYPEOF及TYPEOF继承。

**SIMPLETYPE类型**

在文法中出现的SIMPLETYPE代表原始类型，其下分为三种子类型，分别代表基本数据类型里的Double型、Int型及String型。由于这三种类型均可用在简单TYPE类型中的类型定义部分（即“*[*”“*]*”中的部分），故为了便于实现，本人将这三种基本数据类型抽象为SIMPLETYPE类型，进而能够与其他可以用在TYPE类型中类型定义部分的类型共同管理、操作。

### 类间关系及说明

**ITYPE类型及其派生类**

图 5‑1是ITYPE类型及其派生类型的详细设计。本人将ITYPE类型设计为一个接口，它共被5个类型的类所实现，分别是UNIONTYPE类型、SEQTYPE类型、CONSTYPE类型、TYPE类型及SIMPLETYPE类型。其中由于在文法中UNIONTYPE类型及CONSTYPE类型均可以持有多个子类型，故本人为它们设计了能够持有多个ITYPE类型的数组types。SEQTYPE类型及TYPE类型均可带有通配符。需要说明的是，QNAME类型及SIMPLETYPE类型也实现自ITYPE接口，原因是QNAME类型及SIMPLETYPE类型派生出的DOUBLE、STRING、INTEGER类型均可为用户自定义类型时所用，故为了它们可以被其他类型正常使用（其他类型的type均定义为ITYPE类型），需使该类型同样为ITYPE类型。

**IPATH类及其派生类**

图 5‑1 ITYPE类及其派生类间的关系

图 5‑2及图 5‑3是IPATH类、ITYPEOF类及它们派生类的详细设计。当CONSPATH或CONSTYPEOF类型中只持有一个子类型时，为了能够简化数据结构及文法分析树的结构，本人特别将PATH、CONSPATH、TYPEOF及CONSTYPEOF再度进行抽象，将PATH和CONSPATH的共同父类抽象为IPATH类，将TYPEOF和CONSTYPEOF的共同父类抽象为ITYPEOF。本人将IPATH类和ITYPEOF类设计为一个接口。由于在文法中CONSPATH类型及CONSTYPEOF类型均可以持有多个子类型，故本人分别为它们设计了能够持有多个PATH类型的数组pt及持有多个TYPEOF类型的数组pt。图 5‑2及图 5‑3给出了这些类间的关系：

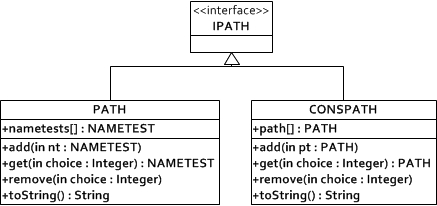
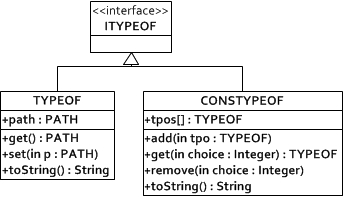


图 5‑2 IPATH及其派生类间的关系

图 5‑3 ITYPEOF及其派生类间的关系

**PATH类及与PATH类有关的数据结构**

图 5‑4是普通类型NAMETEST、TYPEOF、PATH及过渡性类型CONSSTR、CONSPATH、CONSTYPEOF之间的关系。其中有四处复合关系：

* PATH与NAMETEST是整体与部分的关系，且PATH依赖于NAMETEST存在；
* CONSPATH与PATH是整体与部分的关系，且CONSPATH依赖于PATH存在；
* TYPEOF与PATH是整体与部分的关系，且TYPEOF依赖于PATH存在；
* CONSTYPEOF与TYPEOF是整体与部分的关系，且CONSTYPEOF依赖于TYPEOF存在。

需要说明的是，CONSSTR数据结构与其他类无任何关系，它单独存在。

**与“类型绑定”相关的数据结构**

图 5‑5表示与“类型绑定”相关的数据结构间的关系。其中，Binding类存放用户输入的“类型绑定”，即QNAME与其对应的类型（为ITYPE接口的实现类）之间的绑定关系。ITYPE接口可以被TYPE、SEQTYPE、UNIONTYPE及CONSTYPE类实现。Binding与QNAME及ITYPE接口是整体与部分的关系，Binding依赖QNAME及ITYPE接口存在；

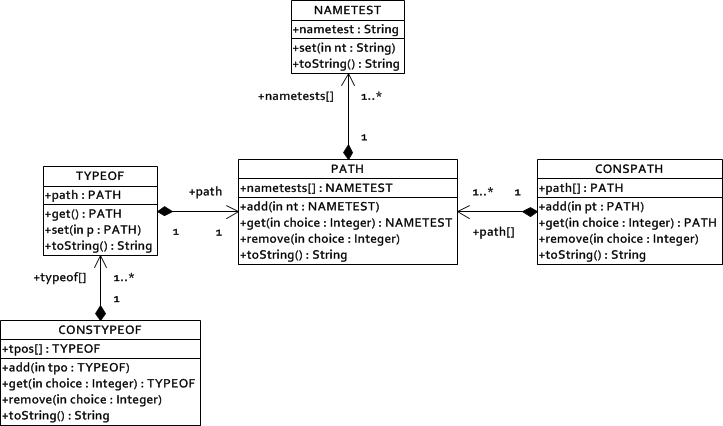


图 5‑4 PATH类及与PATH类有关的数据结构

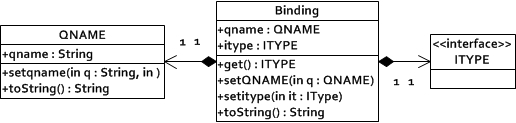


图 5‑5 “类型绑定”部分对应的数据结构

**与“依赖声明”相关的数据结构**

图 5‑7表示与“依赖声明”相关的数据结构间的关系。其中，DEP类持有两个接口：IPATH接口及ITYPEOF接口。IPATH接口可以被PATH及CONSPATH类实现，ITYPEOF接口可以被TYPEOF及CONSTYPEOF类实现。除此之外，针对用户在“依赖声明”部分提出的“前缀PATH”，DEP类还持有一个PATH类。

**与“依赖实例”相关的数据结构**

图 5‑6表示与“依赖实例”相关的数据结构间的关系。其中，INSTANCE类持有一个ITYPE接口，IPATH接口可以被TYPE、SEQTYPE、UNIONTYPE及CONSTYPE实现。

**Parser的返回类及其与其它类间的关**

图 5‑8说明了文法分析器Parser返回类型PASTYPE分别和“类型绑定”、“类型声明”、“依赖声明”及“依赖实例”之间的关系：

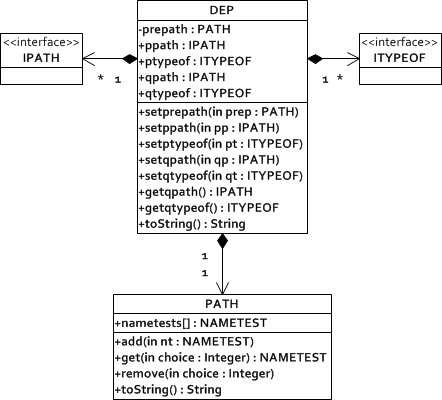


图 5‑7 “依赖声明”部分对应的数据结构

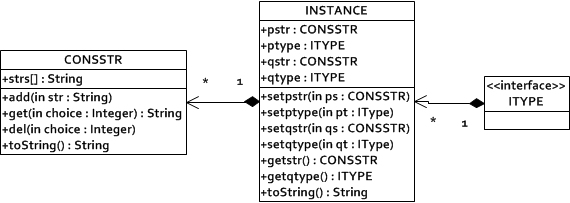
* BindingList类持有一个数组binding，存放用户输入的所有“类型绑定”，BindingList可持有多个Binding。BindingList与Binding是整体与部分的关系，BindingList依赖Binding存在。PASTYPE与BindingList是整体与部分的关系，且PASTYPE依赖于BindingList存在；
* PASTYPE与TYPE是整体与部分的关系，且PASTYPE依赖于TYPE存在；
* PASTYPE与DepList是整体与部分的关系，且PASTYPE依赖于DepList存在；

图 5‑6 “依赖实例”部分对应的数据结构

* PASTYPE与InstanceList是整体与部分的关系，且PASTYPE依赖于InstanceList存在；
* DepList与DEP是整体与部分的关系，且DepList依赖于DEP存在，DepList可持有多个DEP；
* InstanceList与INSTANCE是整体与部分的关系，且InstanceList依赖于INSTANCE存在，InsatanceList可持有多个INSTANCE。

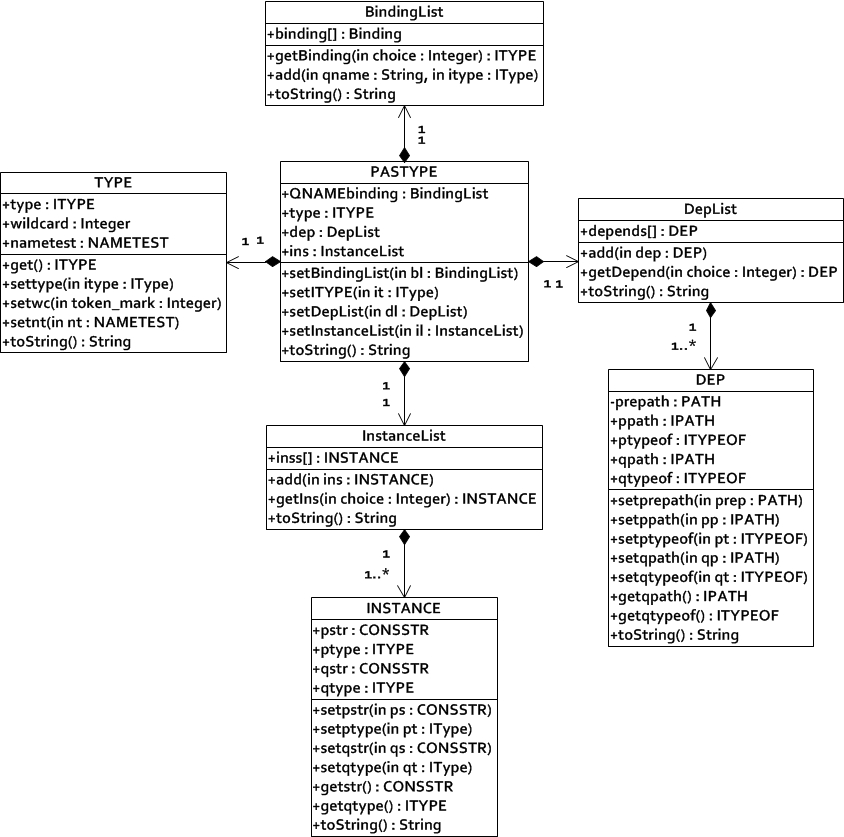


图 5‑8用户输入的四部分对应的数据结构及Parser返回类型

## 词法分析器Lexer的实现

词法分析器Lexer的主要任务是将用户输入流划分为不同的token，并传递给文法分析部分使用。词法分析器的每次调用只识别一个token，且每次传递只传递一个token，它共能识别28种token（含输入结束符end），不同的token保存为不同的int值，以供区分，具体int值划分在本节中将以表格给出。

词法分析器Lexer从用户输入流中一个一个地接收字符。此处用到两个中间变量：int cur\_ptr及char cur\_char。cur\_ptr为当前Lexer在用户输入流中已获取到的位数，cur\_char为当前cur\_ptr所指向的用户输入流中的字符，如：cur\_ptr初始状态为0，即读入用户输入流中第1个字符，传递给cur\_char。

在Lexer能够识别的28种token中，有16种是单字符token，即该token只由一个字符组成，一般是符号，如“*(*”“*)*”等。剩下12中都是非单字符token，即该token是一个字符串。本人在实现非单字符的token时采用了循环结构来接收用户输入流中的字符，若发现已接收但未形成token的多个字符可以形成某个非单字符token，即将这些字符合并成该token，并返回供文法分析器Parser使用。

词法分析器Lexer中有两个重要操作：getToken()和nextToken()。getToken方法和nextToken方法的作业已在设计部分进行介绍，这里不再赘述。

### token的划分及区分

词法分析器Lexer所能识别的28种token在文法分析器Parser中有不同的作用，为了能够正确地、有效地对用户输入进行文法分析，需要对这28种token进行特殊标识，将它们唯一地区分开来。为提高效率，我们通常不直接使用token的字符形式，而是为每种token分配一个不同的数值加以区分（本课题中使用基本数据类型int型），即每种token对应唯一的一个数值，而每个数值对应唯一的一种token。表 5‑2中详细给出了具体的token与数值对应关系：

表 5‑2 token的划分及区分方式

|  |  |  |  |
| --- | --- | --- | --- |
| token | token对应数值 | token | token对应数值 |
| : | 1 | := | 15 |
| , | 2 | / | 25 |
| [ | 3 | ‘ | 26 |
| ] | 4 | ; | 28 |
| ( | 5 | typedef | 16 |
| ) | 6 | Double | 17 |
| - | 7 | String | 18 |
| 🡪 | 8 | Integer | 19 |
| { | 9 | typeof | 20 |
| } | 10 | dependency | 21 |
| | | 11 | instance | 22 |
| + | 12 | nametest | 23 |
| \* | 13 | 结束 | 27 |
| ? | 14 |  |  |

### getToken总体流程

图 5‑9为getToken操作的总体流程图，它清晰展示了单字符token、非单字符token的判断方法。

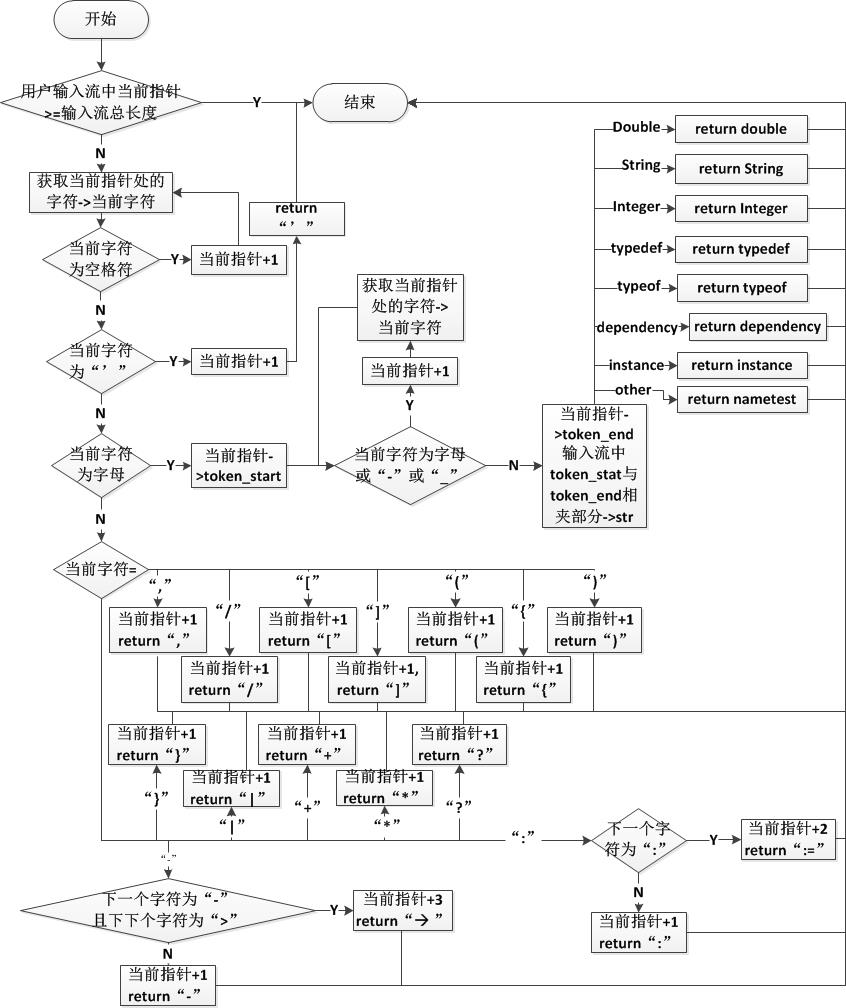


图 5‑9 getToken操作流程

此处以获取nametest为例。系统开始后，判断cur\_char是否为字母，如果为字母，则提示可能为nametest，系统保存当前的cur\_ptr作为nametest字符串起始的位置token\_start，cur\_ptr前移一位，然后系统进入循环。进入循环后，系统继续判定cur\_char是否为字母或“-”或“\_”，若是则继续循环，若发现不是，则提示该nametest已获取结束，系统即退出循环，且保存当前的cur\_ptr作为字符串结束的位置token\_end。

退出循环后，系统根据token\_start和token\_end的值获取该nametest的具体内容。前面本人提到该词法分析器Lexer是根据不同的int值区分token的，所以在此处系统需要判断该nametest是否为Double、String、Integer、typedef、typeof、dependency或instance中的某一个token。若是，系统则return该token对应的int值，若不是，系统则return nametest对应的int值。

### nextToken总体流程

nextToken操作是以getToken操作为基础的，没有getToken操作就无法进行nextToken操作。nextToken的具体内容为：首先进行getToken操作；由于nextToken操作结束后，指针cur\_ptr不能移动，所以在getToken操作结束后，需要将已经发生移动的指针cur\_ptr再移回到原位置，实现cur\_ptr未发生移动的效果，进而实现提前预知下一个token，但该token实际尚未获取的效果。该流程以图 5‑11的形式在中给出。

## 文法分析器Parser的实现

### 总体流程

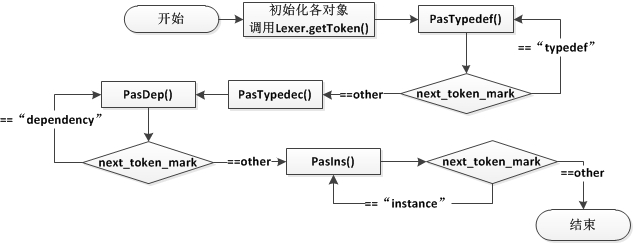
图 5‑10展示了文法分析器Parser的整体流程，说明了文法分析器Parser在用户输入的四个部分（“类型绑定”、“类型声明”、“依赖声明”及“依赖实例”）之间的转换关系。PasTypedef()对应于处理“类型绑定”的子程序，PasTypedec()对应于处理“类型声明”的子程序，PasDep()对应于处理“依赖声明”的子程序，PasIns()对应于处理“依赖实例”的子程序。特别是在PasTypedef()、PasDep()、PasIns()结束处需要进行nextToken操作，以判断“类型绑定”（或“依赖声明”或“依赖定义”）是否不止一个，若不止一个，即nextToken操作提示下一个token为“*typedef*”（或“*dependency*”或“*instance*”），则解析器需要进入循环结构。

图 5‑10 Parser总体流程

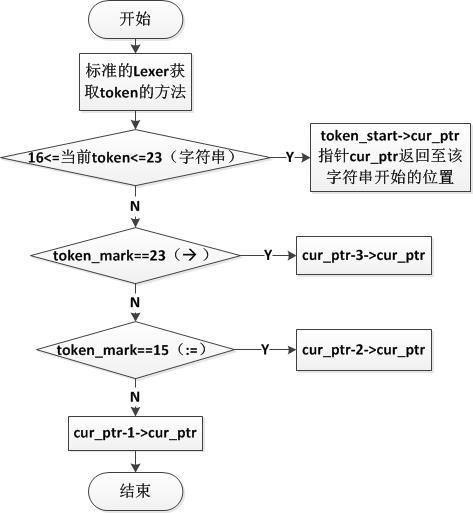


图 5‑11 nextToken操作流程

### 依赖声明和依赖实例的特殊处理

从图 4‑16及图 4‑17中可看出，子程序DEP()及子程序INS()的处理相对繁琐，状态转换相对频繁，条件相对复杂，这一点在实际实现时会严重影响运行效率，同时会增加该解释器的复杂程度。为了解决这一问题，本人在实现时将子程序DEP()及子程序INS()均分拆成两个部分，均以符号“🡪”为分界点。“🡪”前的状态转换过程仍处在子程序DEP()及子程序INS()中，而“🡪”后的状态转换过程将分拆至新的子程序中，即子程序DEPBACK()和子程序INSBACK()中。

在子程序DEP()及子程序INS()中，解释器状态转换至“🡪”时，下一步即使进入到子程序DEPBACK()或子程序INSBACK()中，当新的两个子程序结束时，解释器状态再次转换至子程序DEP()及子程序INS()中，即实现如图 5‑12新的状态转换：

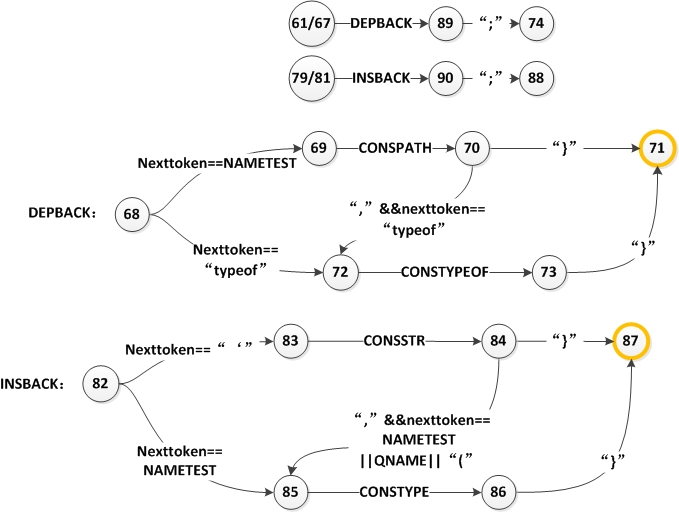


图 5‑12依赖声明和依赖实例的特殊处理

## 实验及测试

经过之前所有的设计和实现，本课题实现的词法分析器Lexer及文法分析器Parser均能够顺利运行。为保证运行结果正确、有效，本人选取了9个测试案例，对依赖描述的各个方面进行测试（每个测试案例所需时间按照处理10次后的平均时间计算），在表 5‑3中详细说明。

表 5‑3 九个测试案例

|  |  |  |
| --- | --- | --- |
| 案例编号 | 处理类型 | 所需时间 |
| 1 | 通配符的获取 | 35.38ms |
| 2 | 多个QNAME（用户自定义类型）的获取 | 31.25ms |
| 3 | 前缀PATH的获取 | 25.25ms |
| 4 | 无CONSPATH有CONSTYPEOF的处理 | 29.38ms |
| 5 | 有CONSPATH无CONSTYPEOF的处理 | 25.63ms |
| 6 | 有CONSPATH有CONSTYPEOF的处理 | 27.63ms |
| 7 | 无CONSSTR有CONSTYPE的处理 | 27.38ms |
| 8 | 有CONSSTR无CONSTYPE的处理 | 29.25ms |
| 9 | 有CONSSTR有CONSTYPE的处理 | 29.25ms |

**实验结果分析：**

实验对本解析器的几个主要功能进行了功能测试，共涵盖九个方面，分别是：

* 通配符的获取；
* 多个QNAME（用户自定义类型）的获取；
* 前缀PATH的获取；
* 无CONSPATH有CONSTYEPOF时依赖声明的获取；
* 有CONSPATH无CONSTYPEOF时依赖声明的获取；
* 有CONSPATH有CONSTYPEOF时依赖声明的获取；
* 无CONSSTR有CONSTYPE时依赖实例的获取；
* 有CONSSTR无CONSTYPE时依赖实例的获取；
* 有CONSSTR有CONSTYPE时依赖实例的获取；

从表 5‑3中可以看出，本解析器在处理通配符的获取时用时最长，为35.38ms，在处理前缀PATH的获取时用时最短，为25.25ms。最长时间和最短时间相差10.13ms，即0.01s，差距很小，可判定本解析器所需解析时间基本符合用户需求。值得注意的是，由于通配符主要出现在“类型绑定”及“类型声明”部分，由表 5‑3可以看出案例1的测试结果与其他8个测试案例的测试结果差距较大，可以认为通配符个数的多少对本解析器的解析时间有较大影响。

# 结论

北京工业大学计算机软件与理论实验室为XML的数据依赖设计了一种通用XML依赖关系，其中包含数值相关的函数依赖关系及类型相关的依赖关系；且提出了基于DOM的XML类型相关的数据依赖检查算法并已经实现检查原型系统。虽然针对算法的检查原型系统已经实现，但尚未实现对XML完整性约束描述的解析器来将用户提出的约束描述转换为系统识别的内部数据对象。因此需要一个方便且强大的解释器。本文在参考北京工业大学计算机软件与理论实验室提出的通用XML依赖关系的基础上，研究了如何将用户编写的完整性约束描述解析为内部数据结构，并对研究结果进行了实现。

本人参考通用XML依赖关系，设计了用户编写的完整性约束描述使用的文法。根据该文法，用户可提出数值相关、类型相关、数值/类型相关及类型/数值相关这四种依赖关系。文法是设计及实现解析器的基础。根据该文法，本人设计并实现了该解析器。该解析器主要包括两个部分：词法分析部分Lexer及文法分析部分Parser。Lexer主要用于将用户编写的完整性约束描述识别为不同的token，且将这些token逐个获取后提供给Parser；Parser主要负责根据Lexer获取的token，将用户编写的完整性约束描述解析为系统能够识别的内部数据对象。Lexer和Parser合并在一起成为用于解析用户编写的完整性约束描述的解析器，不可单独分开。在实现结束后，本人对该解析器进行了功能测试。测试结果显示该解析器可以解析数值相关、类型相关、数值/类型相关及类型/数值相关这四种依赖关系，能够满足北京工业大学计算机软件与理论实验室提出的需求。

由于毕业设计时间的关系，本解析器中还有许多有待完善的地方，如错误处理机制目前只是单纯地提示出出错时cur\_token及next\_token的值，而不能明确指出出错的位置等。

# 参考文献

1. Erik T.Ray. XML入门. 北京：中国电力出版社，2001年11月
2. 类型相关的XML数据依赖, 廖湖声, 苏航
3. 谈子敬、庞引明、施伯乐，XML上的函数依赖推理，软件学报，2003,14(9):1564-1570.
4. 吕腾、阎萍，XML函数依赖及其推理规则，计算机研究与发展，2005,42(5):792-796.
5. 编译原理，蒋宗礼，蒋守旭，高等教育出版社，2010年2月
6. 丁璇. XML正规式类型类体系设计，2011年2月
7. Polymorphism Dependencies for XML, Husheng Liao, Hang Su, Xueyun Jin
8. A Flexible XML-based Regular Compiler for Creation and Conversion of Linguistic Resources, Jakub Piskorski, Witold Drozdzynski, Oliver Scherf, Feiyu Xu

# 致谢

首先，我要感谢我的毕业设计指导老师苏航老师。苏老师总能在我遇到困难，寻求帮助的时候及时解答我的问题。在毕设开始阶段，当我还对本课题的背景知识不太了解时，苏老师耐心细致地给我讲解，在讲解的过程中，我有许多问题，苏老师都一一认真解答，最终使我对本课题的背景有了一个全面的掌握。在进行文法设计时，苏老师不断地对我设计的文法进行修改，力求做到严谨、科学、完整。虽然文法整体较多，但苏老师能够一条一条地检查我的设计并且加以修改，这令我非常钦佩。在设计词法及文法分析部分结构时，苏老师同样对我所做的设计逐步检查、逐步修改，一步步指导我设计出了较为成熟、合理的解析器结构，这使我在后期实现时未遇到较大的问题，且最终的实现结果几乎和所做设计一模一样。在后期实现解析器时，我遇到了一些不太熟悉的问题，我的每一次提问，苏老师都能很快地作出非常详细的解答，这令我毕业设计实现阶段进行的十分顺利。

其次，我还要感谢黄异蕖同学。在我论文撰写期间，她曾帮助我解决了许多论文撰写方面的问题，使我能够顺利的、规范的完成我的论文撰写工作。

除此之外，我更要感谢我的父母，在我毕业设计期间对我学习上、生活上的支持与帮助，使我能够潜心进行毕业设计，不受外界的干扰。

最后，感谢论文评审委员会的老师对我论文的指正。

# 附录

## 多个测试用例

### 通配符的获取

**用户输入文件.txt内容**

*typedef Male-Game := (a[String],d[Integer])+ , (vauling\_horse[Double]\*, hand\_ring[Double]?)\*;*

*typedef Female-Game := uneven\_bars[Double]?, floor\_exercise[Double]+;*

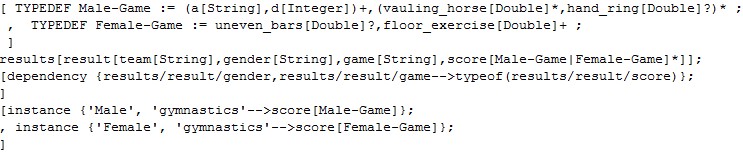
results[ result[ team[String], gender[String], game[String], score[ Male-Game | Female-Game ]\* ]];

dependency {results/result/gender,results/result/game -->

typeof(results/result/score)};

instance {'Male','gymnastics' --> score [Male-Game]};

instance {'Female','gymnastics' --> score[Female-Game]};

**输出结果**

### 多个QNAME（用户自定义类型）的获取

**用户输入文件.txt内容**

typedef Male-Game:=(a[String],d[Integer])+,vauling\_horse[Double]\*

,hand\_ring[Double]?;

typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

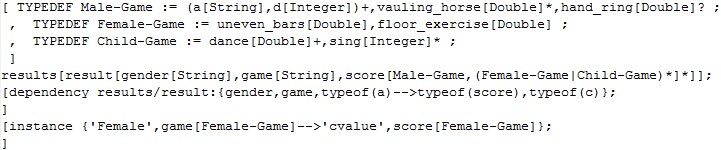
typedef Child-Game := dance[Double]+,sing[Integer]\*;

*results[result[gender[String],game[String]*

*,score[Male-Game,(Female-Game|Child-Game)\* ]\* ]];*

dependency results/result : {gender,game,typeof(a) --> typeof(score),typeof(c)};

instance {'Female',game[Female-Game] -->'cvalue',score[Female-Game]};

**输出结果**

### 前缀PATH的获取

**用户输入文件.txt内容**

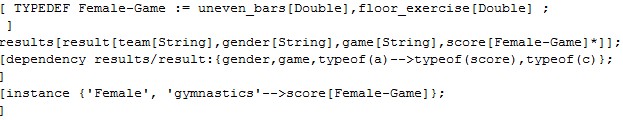
typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

*dependency results/result : {gender,game,typeof(a) --> typeof(score),typeof(c)};*

instance {'Female','gymnastics' --> score[Female-Game]};

**输出结果**



### 无 CONSPATH有CONSTYPOF的处理

**用户输入文件.txt内容**

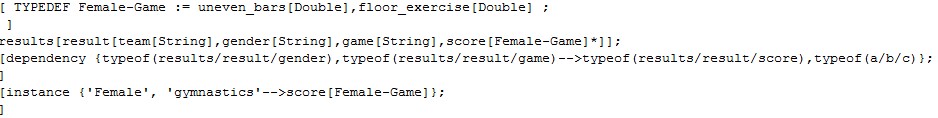
typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

*dependency {typeof(results/result/gender),typeof(results/result/game)-->*

*typeof(results/result/score),typeof(a/b/c)};*

instance {'Female','gymnastics' --> score[Female-Game]};

**输出结果**

### 有 CONSPATH无CONSTYPEOF的处理

**用户输入文件.txt内容**

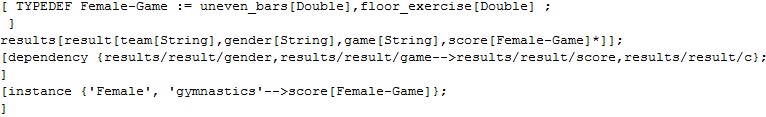
typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

*dependency{results/result/gender,results/result/game-->*

*results/result/score,results/result/c};*

instance {'Female','gymnastics' --> score[Female-Game]};

**输出结果**

### 有CONSPATH有CONSTYPEOF的处理

**用户输入文件.txt内容**

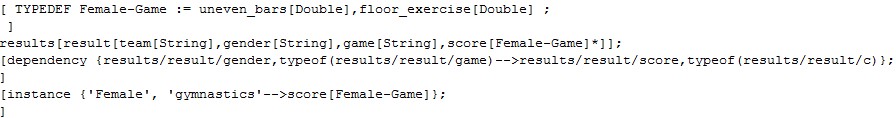
typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

*dependency{results/result/gender,typeof(results/result/game)-->*

*results/result/score,typeof(results/result/c)};*

instance {'Female','gymnastics' --> score[Female-Game]};

**输出结果**

### 无CONSSTR有CONSTYPE的处理

**用户输入文件.txt内容**

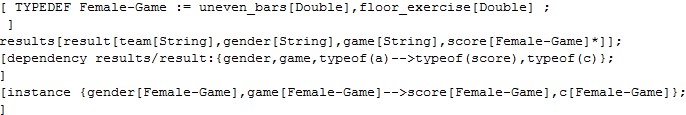
typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

dependency results/result : {gender,game,typeof(a) --> typeof(score),typeof(c)};

*instance{gender[Female-Game],game[Female-Game]-->score[Female-Game]*

*,c[Female-Game]};*

**输出结果**

### 有CONSSTR无CONSTYPE的处理

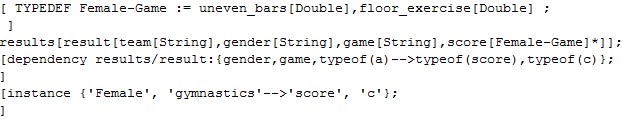
**用户输入文件.txt内容**

typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

dependency results/result : {gender,game,typeof(a) --> typeof(score),typeof(c)};

*instance {'Female','gymnastics' --> 'score','c'};*

**输出结果**

### 有CONSSTR有CONSTYPE的处理

**用户输入文件.txt内容**

typedef Female-Game := uneven\_bars[Double], floor\_exercise[Double];

results[result[team[String],gender[String],game[String],score[ Female-Game ]\*] ];

dependency results/result : {gender,game,typeof(a) --> typeof(score),typeof(c)};

*instance {'Female',game[Female-Game] -->'cvalue',score[Female-Game]};*

**输出结果**

