

[19] 中华人民共和国国家知识产权局

[51] Int. Cl.

G06F 9/45 (2006.01)

G06F 9/44 (2006.01)



[12] 发明专利说明书

专利号 ZL 200380110287.3

[45] 授权公告日 2009 年 3 月 18 日

[11] 授权公告号 CN 100470480C

[22] 申请日 2003. 10. 3

[21] 申请号 200380110287.3

[30] 优先权

[32] 2003. 2. 28 [33] US [31] 60/450,320

[86] 国际申请 PCT/US2003/031312 2003. 10. 3

[87] 国际公布 WO2004/079571 英 2004. 9. 16

[85] 进入国家阶段日期 2005. 10. 28

[73] 专利权人 洛克希德马丁公司

地址 美国马里兰州

[72] 发明人 迈克尔·C·达普 赛·伦·额

[56] 参考文献

CA2307529A1 2001. 9. 29

审查员 刘邵频

[74] 专利代理机构 中科专利商标代理有限责任公司

代理人 朱进桂

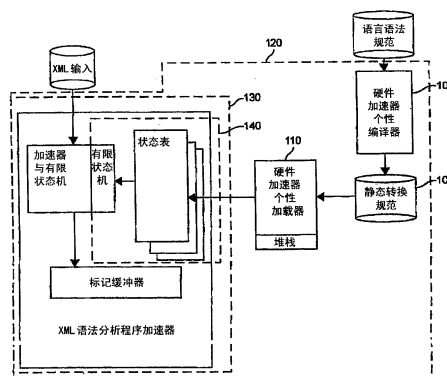
权利要求书 2 页 说明书 36 页 附图 10 页

[54] 发明名称

分析程序加速器装置以及更新其的方法

[57] 摘要

一种硬件加速器个性编译器，通过辨别与各个可执行功能、标识、变元、句法、语法规则、特殊符号等相对应的标记，从如在编程语言中以诸如 Backus - Naur 形式或其派生形式的形式记号提供的一组预期可执行功能的规范、自动地产生无差错状态表。标记可以是递归的(例如无限的)，在该情况下标记被变换为确定性有限自动机或非确定性有限自动机。非确定性有限自动机被变换为有限自动机，然后被变换为用于创建状态表的状态转换，然后该状态表能够被存储、或优选地被加载到硬件语法分析程序加速器的有限状态机中，以定义其个性。



- 1、一种动态地重新配置分析程序加速器的方法，包括：
提供语法规范；
提供具有有限状态机和状态表存储器的分析程序加速器；
对语法进行编译，以产生有限自动机；
以自描述格式从有限自动机创建有限状态机转换规范；以及
将有限状态机状态转换规范载入到状态表存储器中。
- 2、根据权利要求1所述的方法，其中自描述格式是标记语言。
- 3、根据权利要求2所述的方法，其中标记语言是可扩展标记语言XML。
- 4、根据权利要求3所述的方法，其中语法规范包括分析程序加速器的希望性能特性的规范。
- 5、根据权利要求1所述的方法，其中语法规范包括关于分析程序加速器的希望性能特性的规范。
- 6、根据权利要求1所述的方法，所述方法的特征在于：
电子地以包括标记语言的自描述格式提供规范，所述规范包括一组所希望的可执行功能；以及
通过根据加载到状态表存储器中的状态机状态转换规范而实时更新分析程序加速器，来随着时间适应输入流中遇到的样式。
- 7、根据权利要求6所述的方法，还包括从有限状态机向标记缓冲器发送数据。
- 8、根据权利要求6所述的方法，其中标记语言是可扩展标记语言XML。
- 9、根据权利要求6所述的方法，其中规范包括语法。
- 10、根据权利要求6所述的方法，其中从有限自动机创建有限状态机状态转换规范包括以可扩展标记语言XML创建有限状态机状态转换规范。
- 11、根据权利要求6所述的方法，其中规范包括与可执行功能、标识、变元、句法规则或语法规则中的至少一个相对应的一个或多个

标记。

12、一种具有自适应学习能力的分析程序加速器装置，包括：

有限状态机，被配置用于分析文档；

存储器，被配置用于存储至少一个状态表；

分析程序加速器编译器，被配置用于编译语法规则并以包括标记语言的自描述格式产生状态转换规范；以及

分析程序加速器加载器，被配置用于把与状态转换规范相对应的状态表加载到存储器中，

其中分析程序加速器编译器和分析程序加速器加载器被配置成，响应改变的条件而重新配置分析程序加速器。

13、根据权利要求12所述的装置，其中改变的条件包括文档中的样式。

14、根据权利要求12所述的装置，还包括：

标记缓冲器，被配置用于存储从有限状态机接收的至少一个标记；

其中分析程序加速器编译器和分析程序加速器加载器被配置成，实时地响应电子文档中检测的数据样式而重新配置分析程序加速器，从而随着时间自适应地提供学习能力。

15、根据权利要求12所述的装置，还包括：

标记缓冲器，被配置用于存储从有限状态机接收到的至少一个标记；

其中分析程序加速器编译器和分析程序加速器加载器被配置成，实时地响应该装置所遇到的改变的条件而重新配置分析程序加速器，从而随着时间自适应地提供学习能力。

16. 根据权利要求15所述的装置，其中改变的条件包括电子文档中检测的数据样式。

17. 根据权利要求12所述的装置，其中，分析程序加速器编译器和分析程序加速器加载器被配置成，实时地响应该装置所遇到的改变的条件而重新配置分析程序加速器，从而随着时间自适应地提供学习能力，其中改变的条件包括电子文档中检测的数据样式。

分析程序加速器装置以及更新其的方法

技术领域

本发明一般涉及用于控制通用计算机操作的应用和文档处理，并且尤其涉及对给定但任意的语言或格式的应用程序、文档和/或其它逻辑符号序列执行语法分析操作。

背景技术

近些年来，计算机之间的数字通信以及将计算机连接到网络中的领域得到了迅速发展，它在许多方面都类似于前些年个人计算机的激增。远程处理互连性和可能性的这种增加大大提高了这种网络化系统中个体计算机的有效能力和功能性。然而，当计算机投入使用时，个体计算机和系统的使用多样性、它们用户的定位以及目前技术水平造成了单机和它们操作系统的能力和配置的高度多样性，单机和它们操作系统共同被称为“平台”，这些平台在某种程度上、尤其是在操作系统和编程语言级一般互不兼容。

平台特征的这种不兼容性，以及同时对通信及远程处理能力和用于支持它的足够兼容度的要求，导致了面向对象编程（面向对象编程提供一种通过实体、属性和关系的参照系统将应用及数据编译为一组不同程度的一般化模块的概念）以及用于实施面向对象编程的许多编程语言的发展。可扩展标记语言（XML™）就是这样一种语言，XML 已得到广泛使用，并且可以作为文档、在任意组成和体系结构的网络上传输。

在这种语言中，某些字符串对应于某些命令或标识，包括某些专用字符和其它重要数据（共同被称为控制字），这些专用字符和重要数据允许数据或操作实际上识别它们自己，使得此后它们可以被处理为“对象”，以致关联的数据和命令可以被翻译成不同语言不同应用的适

当格式和命令，以便产生足以支持给定机器上预期处理的各个连接平台兼容度。这些字符串的检测是通过一种被称为语法分析的操作来执行的，语法分析操作类似于更常规的把诸如句子的表达式的语法分解为其组成部分，并在语法上描述它们的用法。即使在其它可以被计算机搜索或相反被计算机处理的计算机编程语言和文档中，控制字也将限于有限的但可能很多，从而允许的符号序列将类似地被限制为内容的事件和语言的语法。此外，用于识别文档内容的文档语法分析已经证明是，一种通过检测可能代表攻击、未授权访问或其它可能安全性缺口的控制字来提供处理器和网络中安全性的重要工具。另外，或多或少具有复杂功能序列的其它许多设备如电话和/或诊断设备，响应取决于先前功能序列的类似刺激或输入，采用有限状态机来实现不同功能，而实际上许多这种设备的响应定制变得越来越需要，但是受产生与输入的预期响应序列相对应的状态表的困难的限制。

例如，当对 XML™ 文档进行语法分析时，中央处理器（CPU）执行时间的一大部分，并且可能主要部分，都花费在遍历文档、以便搜索如相对于正在处理的特殊 XML™ 标准而定义的控制字、专用字符和其它重要数据上。典型地这是通过软件来执行的，该软件查询每个字符，并确定每个字符是否属于所关心的一组预定义串，例如包括以下“<command>”、“<data=dataword>”、“<endcommand>”等的一组字符串。如果检测到任何一个目标串，就将标记和指向文档中标记开始位置和标记长度的指针一起保存。这些标记被累积，直到整个文档都被进行了语法分析为止。

对文档进行语法分析的常规方法是，用软件来实施基于表的有限状态机（FSM），以搜索所关心的这些串。状态表驻留在存储器中，并且被设计用于搜索文档中所关心的特定模式。当前状态用作状态表的基地址，并且输入字符的 ASCII 表示是表的索引。例如，假定状态机处于状态 0（0）、并且第一个输入字符是 ASCII 值 02，则状态项的绝对地址将是基地址（状态 0）与索引/ASCII 字符（02）的和/连接。FSM 以 CPU 从存储器中取出输入文档的第一字符而开始。然后，CPU 将绝对地址构造到存储器中与初始化/当前状态和输入字符相对应的状态

表中，然后从该状态表取出状态数据。基于所返回的状态数据，如果不同（表示字符与所关心的串的第一个字符相对应），则 CPU 将当前状态更新为新值，并执行状态数据中指示的其它任何行动（例如，如果单一字符是专用字符，或者如果一旦进一步重复上述操作，就发现当前字符是所关心的串的最后一个字符，则发出标记或中断）。

重复上述过程，并且当找到所关心的串的后续字符时，改变状态。即，如果初始字符被认为是所关心的串的初始字符，则 FSM 的状态可以前进到新状态（例如，从初始状态 0 到状态 1）。如果字符不是所关心的，则状态机将（一般）通过在从状态表地址返回的状态表项目中指定相同的状态（例如状态 0）（或者通过不命令状态更新），来保持相同状态。可能的行动包括但不限于，设置中断、存储标记以及更新指针。然后，对后面的字符重复该过程。应该注意，当正在跟踪所关心的串、并且 FSM 处于非 0 状态（表示还没有找到所关心的串或当前正在跟随所关心的串的其它状态），可以找到与当前串不一致，但是是另一个关心的串的初始字符的字符。在这种情况下，状态表项目将指示适当的行动，以便指出和识别先前跟踪的串片段或部分，并跟踪可能的新的所关心串，直到完全识别新串，或发现新串不是所关心的串为止。换句话说，所关心的串可能被嵌套，并且状态机必须能够在另一个关心的串内检测到所关心的串，等等。这可能要求 CPU 遍历许多次 XML™ 文档的各部分，以便对 XML™ 文档进行彻底的语法分析。

然而，可以容易理解，FSM 的状态表必定是给定的计算机语言及其控制字和/或语法及句法所特有的。也可以理解，随着控制字和格式规则数的增大，状态表的尺寸必定变得非常大。此外，目前通常的做法是，产生制定完善，且使用日益频繁的工业标准语言的增强或扩展版本，并且任何计算机语言的任何修订或扩展都必定需要用于对那种语言文档进行语法分析的 FSM 状态表的相应修订。换句话说，由控制字给出的所有允许符号组合都必定反映在状态表中，并且表面上控制字组和/或语言语法的少量修订或扩展可能需要 FSM 状态表尺寸的大大修正或增加。

较实际的做法是，手动地产生这些状态表、并将它们装载到 FSM

可存取的存储器中,以便在避免改变 FSM 硬件的同时适应语言的变化。FSM 所针对的语言以及 FSM 对那种语言文档进行语法分析的能力,有时被称为 FSM 的“个性(personality)”。即使状态表的开发可能包括计算机语言或采用那种语言的应用程序的大部分开发费用,也不存在切实可行的备选方案来替换用于改变 FSM 个性的手动状态表产生过程。进一步,关于所有手动过程,手动产生状态表常遭受错误,必须在可以可靠使用 FSM 之前检测并校正这些错误。实际的效果是,在需要文档语法分析的情况下,开发状态表所需的时间造成了软件应用和修改及其扩展和升级的实施的延迟,即使在现代处理器和网络环境中这种语言修改、扩展和升级正变得越来越频繁。而且,在文档语法分析用作检测可能安全缺口的工具的情况下,当照这样识别出指示这种可能安全缺口的串时,应该尽可能及时地将所关心的串添加到状态表中,即使这种添加可能需要对用于这种用途的状态表进行大幅度修订。更一般的是,可能需要修改 FSM 个性,以改变包括 FSM 的设备的功能的任何情况,都可能受益于产生相应状态表的困难度、成本和错误灵敏度的减小。

发明内容

因此,本发明的目的是提供一种用于简单且无差错地改变有限状态机状态表的技术和设备。

本发明的另一个目的是,提供一种技术和设备在不进行硬件修改的情况下重新配置有限状态机,以及诸如包括有限状态机的硬件语法分析程序加速器的装置,以便尤其适应计算机语言和应用修改与扩展、或全新计算机语言和/或应用规范。

本发明的又一目的是,提供一种用于产生状态转换表,并以诸如 XML™ 的自描述数据格式记录它们的方法和设备。

为实现本发明这些及其它目的,本发明提供一种用于执行方法和加载器的方法学和编译器,该方法和加载器优选地在诸如硬件语法分析程序加速器的装备内用软件来实施,该硬件语法分析程序加速器能够读取规范或概括预期可执行功能的规范,以产生输出,该输出能够

被加载到可以由诸如语法分析加速器的包括有限状态机(FSM)的设备访问的存储器中,以便定制FSM的个性,而该设备又包括FSM。优选地,用形式记号如Backus-Naur形式(BNF)或其派生物、或其它正规表达式,来写语言或其它规范。基于这种输入,根据本发明的编译器产生相应的状态转换,来形成包括一个或多个状态表的状态转换规范。

附图说明

由以下参考附图的本发明优选实施例详细说明,将可以更好地理解本发明上述及其它目的、特征及优点,其中:

图1是本发明的高级示意框图,

图2A是代表对理解本发明有用处的状态表的图,

图2B是本发明一般化形式的基本操作的高级流程图,

图3是本发明优选实施例的操作的高级流程图,

图4是本发明优选实施例的高级上下文图,

图5A、5B、5C、5D、5E、5F、5G、5H和5I显示了分组和识别语法规则定义中的子表达式,以及

包括图6A和6B的图6显示了完全用自描述数据格式表示的输出状态表规范文件的例子。

具体实施方式

参考附图,尤其是参考图1,图1显示了根据本发明的、且被连接以便向优选地为硬件语法分析加速器的设备中的有限状态机(FSM)提供状态表的个性编译器的基本形式的高级示意框图。最初,应该注意,可以把个性编译器100实施为可连接到存储器105的单独设备(例如在硬件语法分析程序加速器离线的情况下),然后当基于请求方式而需要时,可以访问存储器105以获得状态转换规范,以便由加载器110将状态转换规范加载到FSM状态表中、或者使状态转换规范与任意设备(由虚线120指示)中的FSM140相结合,以部分地或完全控制该状态转换规范,由此允许实时地或基本上实时地更新设备的个性。应该理解,在后一种情况下,基本上是实时的本发明操作,尤其是通过

编译语言语法规则的替换版本加速实时操作而实现的基本上实时的本发明操作，允许本发明始终适于在输入流中遇到的模式和状态；由此在个性编译器以及包括 FSM 的设备中提供基本学习能力。通过相同的标记，应该理解，将在下面描述的产生中间结果的处理的一部分，如语法规则预处理(例如直到图 2B 的步骤 250 的处理或用于提供被归档存储的预产生状态表的处理)，可以以单独的形式操作，并且当需要时处理从存储的数据(例如有限自动机或状态表)起开始操作。本发明的优选应用和环境连同如虚线 130 所示的硬件加速器一起被配置为集成的形式、或完全或部分单独的形式。

与本发明的实施无关，回顾 FSM 状态表的性质对于理解本发明是有用的，尤其是就优选的硬件语法分析程序加速器环境而论。在全部在 2002 年 12 月 31 日提交并且被委派给本发明代理人的美国专利申请 10/331,315、10/331,879 和 10/334,086 (事务所编号 FS-00766、FS-00767 和 FS-00768) 中，分别公开了三种不同的硬件语法分析程序加速器实施，它们在此整个被引入作为参考。图 2A 显示了其中公开的示范性状态表的一部分。

应该理解，图 2A 所示的状态表潜在地只是用于对文档进行语法分析的状态表的很小一部分，并且其本质上意图作为示例。虽然至少在所的形式上、完整的状态表通常在物理上不存在，并且图 2A 也可用于方便理解公知软件语法分析程序的操作，但是图 2A 中没有一个部分被认为是关于本发明的先有技术。

应该注意，XML™ 文档在此用作可以利用根据本发明的加速器处理的一种逻辑数据序列的例子。也可以根据意图被共享服务器计算机执行的网络数据分组内容、如用户终端命令串，来构造其它逻辑数据序列。(这种命令串经常由恶意用户产生，并且被发送给共享计算机作为长期入侵企图的一部分。)根据本发明的加速器适合于处理多种这样的逻辑数据序列。注意到图 2A 所示状态表的一部分是复制的也将是有用的。

方便且优选的是，将符号的十六进制表示用作状态表索引，并据此将状态表的垂直列标定为“00”至“FF”。对行进行编号，以反映

FSM 可以呈现的各种状态。从而，将多行基地址分成与可以用于代表文档中要被执行语法分析的字符的代码的数量相对应的许多列；在该例子中，分成与字符的基本 8 位十六进制字节相对应的 256 列。可以以这种形式提供和可能需要的、可打印或不可打印的字符一样多的字符。

注意到所示状态表项目的几个方面将是有益的，尤其是在理解图 2A 所示示范性状态表的多小部分支持许多字检测的方面：

1. 在所示的状态表中，在状态为 0 的行中只有两项包括不同于“保持在状态 0”的项，当正在测试的字符不和任何关心的串的初始字符匹配时，“保持在状态 0”项维持初始状态。为前进到状态 1 作准备的单项对应于所有关心的串都以相同字符开始的特殊情况。将为前进到另一种状态作准备的其它任何字符一般将、但不一定前进到不同于状态 1 的状态，但是对可以通过另一个字符到达的相同状态的进一步参考可能对例如检测嵌套串有用处。把 {状态 0, FD} 所示的具有“保持在状态 0”的命令（例如“特殊中断”）包括进来，将用于检测和操作特殊单字符。

2. 在状态 0 以上的状态中，“保持在状态 n”项为通过例如可能在命令数值变元中遇到的一个或多个字符的潜在长行程来维持状态作准备。本发明提供对这种类型字符串的特殊处理，以便提供增强的加速，如将在下面详细讨论的。

3. 在状态 0 以上的状态中，“转到状态 0”项表示检测到把串和任何关心的串区分开的字符，而与先前已检测到多少匹配的字符无关，并且“转到状态 0”项使语法分析过程返回到初始/默认状态，以便开始搜索另一个关心的串。（为此，到目前为止，“转到状态 0”项一般将是状态表中出现最频繁或最多的项。）返回到状态 0 可能需要语法分析操作返回到文档中在检测到区别字符时跟踪的字符串的开始字符之后的字符。

4. 包括具有“转到状态 0”的命令的项指示对所关心的完整串的检测的完成。一般来说，命令将要存储此后允许串被处理为对象的标记（和标记的地址和长度）。然而，具有“转到状态 n”的命令为起动

中间点操作、同时继续跟踪可能潜在地和所关心的串匹配的串，作准备。

5. 为避免搜索在两个关心的串之间发生分支的任何点处的模糊性（例如具有 $n-1$ 个相同初始字符、但具有不同的第 n 个字符的两个串，或具有不同初始字符的两个串），一般需要进行到不同（例如不连贯）状态，如 {状态 1, 01} 和 {状态 1, FD} 所示。除特殊字符所包括的串和所关心的串具有共同初始字符的特殊情况以外，完全识别任意长度 n 的串将需要 $n-1$ 种状态。为此，即使对于较适度的所关心串的数量，状态表的状态和行的数量通常也必定很大。

7. 与前一段相反，大多数状态都可以完全由一个或两个唯一的、且默认值为“转到状态 0”的项来表征。本发明利用图 2A 状态表的该特征，以便相对于所关心的串的一般情况、获得硬件高度节约及语法分析过程的大幅度加速。

如常规地执行的语法分析操作以处于给定默认/初始状态如图 2A 中状态 0 的系统开始，然后一旦重复过程，当找到所关心的字符串的匹配字符时，语法分析操作就前进到编号更高的状态。当所关心的串被完全识别、或者当在潜在地是匹配串的串中的中间位置指定了特殊操作时，执行诸如存储标记或发出中断的操作。然而，每当对文档的每个字符重复操作，都必须从 CPU 存储器中取出字符，必须取出状态表项目（再次从 CPU 存储器中），并且必须在顺序操作中更新各种指针（例如指向文档字符和状态表基地址的指针）和寄存器（例如寄存初始匹配字符地址和累积串长度的寄存器）。以上引入的应用中所公开的硬件语法分析程序加速器通过为并行执行这些操作中的许多操作作准备、同时通过其中的有限状态机评定文档的后续字符，来加速语法分析过程。

总之，语法分析程序的基本功能是，唯一识别所关心的输入字符（例如符号或二进制信号序列）串，并且一旦实现这种识别就发出唯一标记和其它信息。在某些情况下为了某些目的，也必须检测和验证所关心的嵌套串的识别。因此，重要的是认识到，能够导致标记发出的所有字符串都是被执行语法分析的文档的语言的、如通过那种语言

的控制字和特征句法定义的事件。相反，就语言规范而论，由控制字和/或它们的顺序排列表示的语言事件也可以被认为是标记。从而，语言规范包含足够的信息，用于语法分析程序为给定语言或一组关心的字符串定义能够导致标记发出的所有关心的字符串，从而足以产生要识别所有关心的字符串的状态表。

参考图 2B，图 2B 显示了本发明一般化形式的操作流程图。一旦调用过程，“下一个标记”就被调用，如 210 所示。假定，只有在按照表示语言规范的数据的连续顺序的语言规范中，才存在某种顺序。在存在顺序的意义上，实际顺序可以是任意的，并且在任何情况下都不影响将被开发的状态转换规范的可用性，因为语法分析程序被配置成识别任何顺序的所关心串。标记的顺序可以影响所分配的状态号，但是那些状态号没有实际意义。即，任何关心的串都将造成通过状态表状态序列前进，以达到所关心的串将被唯一识别的终结状态，但是状态和状态序列的数量对结果没有影响。

从而，“下一个标记”的调用用于通过使整个过程循环直到所有标记都被考虑为止，来提供一种促使考虑整个语言规范的机制。优选地，通过读取语法输入文件 215、识别语法实体如字符/符号的控制字和句法要求（例如分支语句、字符定界域等）、并通过将唯一标记分配给每个被识别的实体来标记化它们，以执行该操作。在该过程中也可以考虑和应用特殊匹配规则或准则（例如指定任意字符的数量）。在图 2B 的 220 集体指出这些功能。

该过程将导致如 230 所示的用于某些语法实体（如代表语言中提供的命令的控制器）的一组转换图或有限自动机（以下可以通过该术语来参考这种转换图），而其它语法实体如递归分支语句和定界符符号将需要附加处理和变换，来获得可以在状态表中表示的字符串。具体地说，在 240，对还没有被变换为字符串的剩余语法规则进行测试，以确定它们是递归的、还是表示其它性质如“排除”操作。如果需要，根据该测试，在 245 简化语法规则，以便将语法规则表示为字符串、或者将语法规则扩展为扩展语法规则。在这一点上，执行 246 的用于复制如循环 249 所示的步骤的嵌套子过程，以便为递归符号产生一组

新有限自动机。该递归符号变为这组新有限状态机的起始状态，并且嵌套子过程内遇到的任何附加递归符号将被处理为好像是文字符号。文字符号是能够直接用作状态转换输入的符号。在返回到 230 的主处理步骤之前，为递归符号产生的一组新有限自动机被保存在存储器中，以便稍后进行处理，并且递归符号被标示为语法规则中的文字符号，使得当处理返回到步骤 230 时，它中断递归。然后，通过循环到 210 来重复过程，如以上提到的循环 249 所示，直到所有语法实体都被考虑到，并被处理以形成完整的有限自动机序列或状态转换图为止。

现在，在获得被表示为有限自动机序列的完整语言语法之后，处理继续以 250 的起始状态开始。状态转换图由状态节点和转换标签边缘组成。标签边缘识别两种信息：输入（例如转换条件）和下一状态。如果相同输入（例如字符）可以造成转换到不同状态的多种转换，则有限自动机被称为非确定性的。230 的变换处理既产生非确定性有限自动机（NFA）、又产生确定性有限自动机（DFA）。NFA 不适于构造硬件加速器 FSM 的状态表。在 260 执行检查，以挑出 NFA。然后，在 265 通过使具有确定性质的状态退缩为闭集，来将 NFA 变换为 DFA。

从而，形成闭集的这些状态被组合，然后被代表闭集的新状态替代。然后，在标签边缘进入和离开新状态的情况下，调节状态转换。适于该变换的合适技术对于编译器设计领域的技术人员是周知的，在“Principles of Compiler Design (编译器设计原理)” by Aho and Ullman, Addison-Wesley Publishing Co., 1977, pp.91-93 中，给出了教科书例子。通过 268 的循环，对附加状态重复变换。在所有 NFA 都被变换为 DFA 之后，则可以在 270 优化 DFA，并且在 280，在把优化的 DFA 加载到 FSM 之前将其变换为状态表数据存储在大容量存储器中，或者将优化的 DFA 直接加载到 FSM 中。

既然状态以及语言主要部分的状态转换完成了，则在循环 292 对在 245 识别的每个递归符号都重复将有限自动机变换为状态表的过程。在 290，识别出具有还未被变换为状态表的有限自动机的递归符号表中的每个递归符号。在 295，专门为递归符号初始化新状态表。该新状态表不一定是物理上分开的表。可以将该新状态表附加到早先

产生的语言主要部分的状态表上。在此为简化描述，在逻辑上把该新状态表看作是分开的新状态表。在 296，把先前为递归符号产生的有限自动机收集在一起，使得再次从步骤 260 开始执行将有限自动机变换为状态表的相同过程。重复 292 的循环，直到所有递归符号都被变换为状态表为止。

上述描述作为本发明一般形式的概要，现在将参考图 3 至图 6 来描述本发明的优选实施例。优选实施例针对于产生针对特殊 XMLTM 形式的状态表。然而，应该理解，可以以各种形式、在各种实施例中，以及为不同目的、如检测潜在安全缺口企图（潜在安全缺口企图可能使用多种计算机语言中任一种语言的某些命令）或仅仅辨别特殊命令、句法等，来使用本发明。

本领域技术人员应该理解，图 3 所示的本发明优选实施例的操作基本上是图 2B 一般化流程图的扩展。另外，图 3 的操作被显示为顺序的、而没有分支操作，这对于快速执行是优选的，同时足以适应 XMLTM。为进一步加速处理，优选地通过在产生表中提供中间和临时存储器来避免某些分支，使得只有需要进一步处理的语法实体才保持在处理流中。

一旦起动过程，就读取语法文件，并识别和标记化语法实体，如 310 所示。然后，将标记化的语法规则存储在产生表中，如 320 所示。然后，将语法规则操作尽可能地变换为字符串（字符集），如 330 所示。

如以上提到的，优选地将语法文件表示为形式记号，如 Backus-Naur 形式（BNF）或其派生形式，如扩展 Backus-Naur 形式（EBNF）。环球网联盟以这种形式来使 XMLTM 文档化，并且普遍地可以以电子形式得到它。EBNF 记号的概要描述如下：

语言由符号组成，该符号具有一组控制符号怎样能够被正确组合在一起的规则（语法）。每条 EBNF 语法规则都被规定如下：

符号 ::= 表达式

语言以起始符号开始，并且用右手边表达式来定义符号，如以上使用附加符号、描述符、属性和算子的记号所示。在后续规则中定义新的符号，直到为语言定义了所有符号为止。

可以出现在右手边表达式中的符号描述符、属性和算子被定义如下：

`#xN`

其中 N 是十六进制整数，表达式匹配 ISO/IEC 10646 中的，当被解释为无符号二进制数时、其规范（UCS-4）码值具有所示值的字符。`#xN` 形式中的前导零数目是无意义的；相应码值中的前导零数目由使用中的字符编码决定，并且不重要。

`[a-zA-Z]`, `[\#xN-\#xN]`

和具有有所指示的包括范围中的值的任何字符匹配。

`[abc]`, `[\#xN\#xN\#xN]`

和具有有所枚举的字符中的值的任何字符匹配。可以在一组括号中混合枚举和范围。

`[^a-z]`, `[^\#xN-\#xN]`

和具有不在所给出的字符中的值的任何字符匹配。可以在一组括号中混合枚举和禁值范围。

`"string"`

和双引号内的文字串匹配。

`'string'`

和单引号内的文字串匹配。

可以对这些符号进行组合，以匹配如下的更复杂模式，其中 A 和 B 代表简单表达式：

(表达式)

表达式被处理为单元，并且可以如该列表所述那样组合表达式。

`A?`

匹配 A 或什么都不匹配；A 是任选的。

`A B`

匹配后面接着 B 的 A。该算子具有比“择一 (alternation)”高的优先权；从而 `A B | C D` 和 `(A B) | (C D)` 是相同的。

`A|B`

匹配 A 或 B，但是不匹配 A 和 B 两者；也被称为“择一 (或)”。

$A - B$

和匹配 A 但不匹配 B 的任何串匹配；（从 A 中排除 B）。

A^+

匹配 A 的一次或多次出现。连接具有比“择一”高的优先权；
从而 $A^+ \mid B^+$ 和 $(A^+) \mid (B^+)$ 是相同的。

A^*

匹配 A 的零次或多次出现。连接具有比“择一”高的优先权；
从而 $A^* \mid B^*$ 和 $(A^*) \mid (B^*)$ 是相同的。

产生过程中使用的其它记号（或规则组）：

$/\dots/$

表示注释。

使用以上记号来定义 XML™ “Name”的例子如下：

$\text{Namechar} ::= \text{Letter} \mid \text{Digit} \mid '.' \mid '-' \mid '_' \mid ':'$

$\text{Name} ::= (\text{Letter} \mid '_' \mid ':') (\text{Namechar})^*$

假定‘Letter’表示字母字符、‘Digit’表示数字字符 0—9，则 XML™ ‘Name’是以字母、下划线或冒号开始、接着是零个或多个‘Namechar’的字符序列。‘Namechar’是字母字符、数字字符、句号、破折号、下划线或冒号。

应该理解，上述一些记号指定了“排除”操作（例如 $A-B$ ）。在 332 辨别这些记号，并在 334 将这些记号变换为能够被表示为字符集字符串的简单规则。接着，在 340 识别递归语法规则。例如，考虑以下两条 XML™ 语法规则：

$\text{cp} ::= (\text{Name} \mid \text{choice} \mid \text{seq}) ('?' \mid '*' \mid '+')$

$\text{choice} ::= ('S?cp(S?' \mid 'S?cp)+S?')$

“cp”和“choice”两者的扩展相互参考。将符号“cp”或“choice”的定义代入语法规则表达式的右手边将导致无限长度的表达式，这是由于 cp 和 choice 相互参考的语法规则所引起的递归造成的。优选地，在 342，在将语法变换为一组有限自动机之后可以丢弃的临时存储器中，从起始符号起、从语法产生来扩展这些规则，将此刻的递归符号处理为特殊文字符号。文字符号是一种被它自己用作状态转换输入的符号。

这将导致整个语言的完整连续语法规则。在此被临时处理为文字符号的递归符号将在 344 被处理。

在 344, 先前识别的每个递归符号都用作新扩展的起始符号, 新扩展将以递归符号的完整连续语法规则结束。它使得能够专门为每个递归符号产生一组新有限自动机。稍后, 将在过程中根据该步骤所产生的有限自动机来产生这些递归符号所关联的一组状态。为进一步说明在将把递归符号变换为状态之后怎样处理递归符号, 在此我们将简要描述加载器(图 1 中的 110)内的功能。加载器根据由硬件加速器个性编译器(HAPC)产生的状态信息, 来填充硬件加速器 FSM 内的状态表。除状态识别和状态转换以外, HAPC 还识别去往加载器的所有递归符号, 如图 6 所示。当加载器处理涉及递归符号的状态转换时, 加载器识别出递归符号。加载器不是使 FSM 立即转到下一状态, 而是把作为该特殊转换动作的命令加载到 FSM 中, 以便将下一状态信息推进硬件加速器内的堆栈中, 并分支转到递归符号语法规则的起始状态。对于递归符号语法中的每个终结状态, 加载器都把作为终结状态动作的命令加载到 FSM 中, 以便从堆栈中托出状态信息, 并转到从堆栈中托出的下一状态。如果遇到了作为输入被嵌入递归符号语法规则的状态内的递归符号, 则加载器执行和刚才描述的操作相同的操作。作为取得语法规则中递归定义的结果, 硬件加速器内的堆栈使得能够处理这些嵌套的状态转换。

然后, 根据扩展语法规则产生 NFA, 并将所产生的 NFA 变换为 DFA, 如上述 355 所示。然后, 可以优化 DFA (360), 并将优化的 DFA 变换为状态表项目 (370), 然后存储该状态表项目, 如上所述。

优选地, 把以上操作提供为根据面向对象编程概念的软件对象。如本技术领域容易理解的, 对象实质上是把它们的操作(与程序整体功能和对象自身之间交互作用的功能有关的操作)封装和隐藏起来的较大程序, 同时如果需要, 对象能够调用其它对象来执行程序。也可以将对象装配为具有形成图 4 所示上下文的关系的类。在以下对软件对象类以及其中的对象的描述中, 对象以及所提供的对象功能的描述足以成功实行本发明, 并且对象所封装的对象进一步细节对于本发

明的成功实行是不重要的。

如图 4 所示,根据本发明的 HAPC 包括主 HAPC 类和十二个附加类:

1. InputMgr
2. Token
3. RuleMgr
4. ExpandedRule
5. CharSet
6. RecursiveSymbolMgr
7. RSEntry
8. NFAMgr
9. StateMgr
10. StateEntry
11. TransitionEntry
12. DFAMgr

以下将按顺序对它们进行讨论。

HAPC 类包含用于命令从读取输入、执行编译处理、直到写输出的执行的主程序和方法。InputMgr 类对象负责对来自语法规则规范文件的输入进行标记化。Token 类对象定义所支持的标记种类,并对访问、设置和更新标记提供支持。RuleMgr 类对象把标记化的语法产生规则组织在散列表中,以允许软件能够快速访问语法规则。CharSet 类对象对语法规则中的字符集实体提供专门支持。ExpandedRule 类对象提供一种用于从特定标记开始将语法规则改进为连续语言规则的工具。RecursiveSymbolMgr 类对象提供一种用于识别在语法规则定义中递归使用的符号的仓库。RSEntry 类对象定义递归符号仓库项目格式。NFAMgr 类对象对从语法规则创建非确定性有限自动机提供支持。StateMgr 类对象管理一种包含用于创建状态表的状态转换信息的仓库。StateEntry 类对象定义用于状态仓库中项目的格式。TransitionEntry 类对象提供一种用于存储状态转换信息的工具。DFAMgr 类对象对把非确定性有限自动机转换为适于产生状态表的确定性有限自动机提供支持。

HAPC

HAPC 类包含用于开始整个编译过程的主程序。除主方法以外，HAPC 类还包含以下方法：

```
genStates  
writeStateTransitions  
timestampToString
```

genStates 方法是编译过程的主驱动程序。genStates 方法创建其它类对象、并与所创建的其它类对象对接，以便读取输入语法规范、将语法规范信息处理为有限状态、并将状态转换信息写出到文件。

writeStateTransition 方法为 HAPC 所产生的状态转换规范创建输出流，并将信息写出到输出文件。

timestampToString 方法是一种支持 writeStateTransition 方法、以便将 timestamp（时间戳）信息格式化为可打印串的实用方法。

InputMgr

硬件加速器个性编译器输入管理程序 InputMgr 负责读取包含语言语法规则的输入文件、并将输入规则数据编码为标记。输入文件中的信息被分解为标记，使得能够通过它们的种类来容易地识别它们。

InputMgr 类支持以下构造程序和方法：

```
InputMgr  
next_token  
startNewSection  
next_line  
parseCharLiteral
```

InputMgr 构造程序设置 Java 缓冲区头，以便读入输入语法规则文件。输入语法规则文件由以下三个部分组成：用户指令，产生规则，以及产生规则重载。这三个部分通过始于且只包含两个字符%%的行，而相互分开。用户指令部分首先出现在文件的开头。所有用户指令关键字都以“%”为前缀。当前，唯一支持的用户指令是具有一个变元的

%StartSymbol。该变元指定在产生规则部分中定义的语言的起始符号。符号集内围起的注释：/*和*/可以出现在输入文件中的任何地方。产生规则部分包含要处理的语言的语法规则。当前，假定用 EBNF 格式来表示语法规则。产生规则的所有左手边符号都必须始于第 1 列。产生规则可以跨越许多行。所有续行都必须始于第 1 列的至少空白字符。产生规则重载部分是最后一部分，并且是任选的部分。产生规则重载部分允许用户重新规定早先出现在产生规则部分中的某些产生规则。当所有语法规则都由语言创建者定义时，这允许用户规定所有语法规则、而不对产生规则部分进行任何改变。如果某些规则具有一些不能被该软件自动处理的记号，用户可以仅仅利用产生规则重载部分中该软件所支持的记号，来重新规定那些规则。

在调用 InputMgr 构造程序之后，HAPC 软件可以开始通过重复调用 next_token 方法，每次一个标记从输入文件中提取整个输入语法产生规则。最初，通过识别从输入文件创建的输入字符流中的定界符字符，来形成每个标记。然后，将标记分类为不同的标记种类。在 Token 部分中进一步详细描述这些标记种类。InputMgr 透明地处理格式化信息，并跳过输入文件中的所有注释。对于输入文件中被指定为数值的字符文字，在对它们进行标记化之前，通过 parseCharLiteral 方法在内部将它们转换为字符值。

startNewSection 是一种允许调用程序使 InputMgr 从“规则部分结束”状态复位、由此允许软件读入附加产生规则来重载某些先前语法规则规范的简单方法。

构造程序、startNewSection 和 next_token 方法是 InputMgr 类对象的主要外部接口。InputMgr 类中实施的其它私有方法有：next_line 和 parseCharLiteral。私有方法 next_line 从输入文件得到一行字符，并将输入行的剪切型式返回给调用程序。next_line 方法保持输入文件的行计数，并剪掉输入文件开始和结尾处的空格。另一私有方法是 parseCharLiteral。parseCharLiteral 方法把被表示为十六进制数的字符文字转换为内部 ASCII 字符。这允许以和可打印字符相同的方式，来在软件内处理不可打印的字符。

Token

Token 类提供一种创建和维护标记的工具。通过将输入字符流分解为标记，软件可以容易地对输入文件中的每个逻辑字符序列进行分类，并据此处理信息。有 7 种主要标记种类：控制；符号；算子；属性；组；杂项 (Misc)；及未知。

控制种类内的最重要标记是文件结束 (EOF)，EOF 向软件指示到达了输入文件结尾。控制种类中也定义了其它少数标记，然而，它们仅供软件内短暂使用。因为这些少数标记对根据本发明基本原理的本发明实行不重要，所以在此将不详细描述它们。

属于符号种类的标记包括：StrProd (开始产生)、Symbol (正规语法符号)、RecursiveSymbol、Literal、Set 和 CharSet。StrProd 标记被创建用于存储新语法规则的名称。Symbol 标记表示一般语法规则符号。RecursiveSymbol 是一种在软件确定在语法规则中递归使用符号之后、从一般 Symbol 标记重新分类的标记。当对单字符、字符的数字表示以及字符串进行标记化时，将它们标示为文字。在对字符的数字表示进行标记化之前，将字符的数字表示转换为正规 ASCII 字符。通过这样做，用同样方式来处理所有字符。方括号围起的输入串被分配给 Set 标记。Set 标记可以具有某一离散字符集合、或某一字符范围。当集合内的值被处理为标示属于该集合的每个单字符的位集合时，Set 标记被转换为 CharSet。利用语法规则中的“择一”算子关联在一起的字符也被归合到 CharSet 中。

算子标记是自明 (self-explanatory) 的。这些算子用于语法规则中，用来组合和混合语言基本实体，以形成更复杂的实体。属于算子种类的标记有：OpExpInto；OpOr；及 OpExclude。在 EBNF 记号中 OpExpInto 是“::=”符号。OpExpInto 向软件指示，标记序列将紧接着该标记之后，并且它们将形成刚好在该标记之前出现的左手边符号的扩展规则。OpOr 是“或”算子，在 EBNF 记号中以“|”符号表示。OpExclude 是“排除”算子，在 EBNF 记号中以“-”符号表示。早先在形式语法部分中描述了这两个算子。

属性标记用于描述语言特定规则中的符号的允许出现频率。属性种类中的标记包括: AttZeroOrOne; AttZeroOrMany; 及 AttOneOrMany。AttZeroOrOne 在 EBNF 记号中以 “?” 字符来表示, 并用于指示刚刚在该标记之前出现的符号是任选的符号。在语言内的该特殊上下文中, 那个任选符号可以出现 0 次, 或刚好出现一次。AttZeroOrMany 在 EBNF 中以 “*” 字符来表示, 并用于指示刚刚在该标记之前出现的符号可以在当前上下文中出现 0 次或多次。同时, AttOneOrMany 类似地允许先前标记化的符号出现一次或多次, 并且在 EBNF 中以 “+” 字符来表示。

组种类(Group category)具有两种定义的标记: LParen 和 RParen。LParen 表示组开始, 而 RParen 表示组结束。通过左括号和右括号所围起的表达式, 来定义组。组内的整个表达式被处理为单元。组可以被嵌入另一个组内。

杂项种类 (Misc category) 包含元标记。这些标记包括: BlockStart; BlockEnd; 及 RecExp。这些标记被插入内部产生表所存储的语法规则中, 主要供调试之用。作为状态转换产生过程的一部分, 从 “语言起始符号” 开始成行扩展语法规则, 直到所有符号都变为终结符号或递归符号为止。当然不成行扩展递归符号, 这是因为递归扩展将导致无限循环, 如上所述。为帮助调试, 将 BlockStart 和 BlockEnd 标记插入在成行扩展期间得到的规则中, 以识别扩展的规则内的规则段的开始和结束。标记包含来自原始输入产生规则的左手边符号名, 以帮助识别。RecExp 指示递归表达式。

未知标记种类是一种在解析未知标记时、或者在把未知标记作为错误报告给用户之前, 被软件用来临时保存该未知标记的位置容器种类。

Token 类提供构造程序和以下方法:

Token

equals

setToken

getCategory

isCategoryControl

```
isCategorySymbol  
isCategoryOperator  
isCategoryAttribute  
isCategoryGroup  
isCategoryMisc  
print
```

Token 构造程序和 setToken 方法允许调用程序从头开始构造标记。调用程序可以利用 getCategory、equals 和各种 isCategoryXXXX 方法，来执行标记查询。print 方法将向屏幕打印与标记有关的所有信息。

RuleMgr

RuleMgr 类提供一种在被称为 ruleTable 的散列表 (hash table) 中创建并维护语法产生规则的工具。语法产生规则的右手边表达式被存储为标记矢量。通过把产生规则的左手边符号用作散列关键字，将矢量保存到散列表中。

RuleMgr 构造程序提供一种初始化 RuleMgr 类的普通机制。RuleMgr 类提供其它方法来帮助构造 ruleTable，以便查询 ruleTable、执行转换、以及支持调试。这些方法是：

```
parseEBNFRules  
checkRule  
componentLength  
extractCharSet  
replaceGroupsWithCharsets  
convertCharSetEntities  
findExclusion  
findAlternation  
groupRightAltParam  
goupLeftAltParam  
groupAltParams
```

`printRule`

`replaceRule`

`parseEBNFRules` 是 `RuleMgr` 类提供的一种重要方法。`parseEBNFRules` 允许调用程序从输入语法文件中提取语法规则规范。`parseEBNFRules` 方法利用传入的 `InputMgr` 来读取语法文件。然后，`parseEBNFRules` 方法将每条产生规则重新构造为标记矢量。规则被保存到 `ruleTable` 中，并且通过规则的左手边符号来检索每条规则。

`checkRule` 方法允许调用程序确定 `ruleTable` 中是否已定义规则。这消除了调用程序直接访问实施 `ruleTable` 的散列表的需要。

给定语法规则的符号名，`componentLength` 方法返回为定义语法规则所需的标记数。该方法的典型用途是，确定在语法规则表达式中规则是否只有单一组成部分（例如集合）。

`extractCharSet` 方法检查如作为输入的一对索引所指定的语法产生规则的一段标记矢量，并确定是否可以将表达式子集分解为 `CharSet`。如果可以将表达式子集变换为 `CharSet`，则 `extractCharSet` 方法将把 `CharSet` 返回给调用程序。该方法支持 `convertCharSetEntities` 方法。

`replaceGroupsWithCharsets` 方法遍历传入的包含标记序列的矢量，并用字符集（`CharSet`）代替所有合适的表达式子集。该方法支持 `convertCharSetEntities` 方法。

`convertCharSetEntities` 方法遍历整个 `ruleTable`，并将所有集合和符合条件的表达式子集变换为 `CharSet`。

`findExclusion` 方法遍历整个 `ruleTable`，并找到包含“排除”算子的所有语法产生规则。在完成后，该方法以矢量形式返回那些语法规则。

`findAlternation` 方法遍历整个 `ruleTable`，并找到包含“或”算子的所有语法产生规则。在完成后，该方法以矢量形式返回那些语法规则。

如果还没有用括号来分组子表达式，`groupRightAltParam` 方法在语法规则中“或”算子的右手边子表达式周围添加一对括号。

如果还没有用括号来分组子表达式，`groupLeftAltParam` 方法在语法规则中“或”算子的左手边子表达式周围添加一对括号。

如果还没有用括号来分组子表达式，`groupAltParam` 方法在语法规则中“或”算子两边的两个子表达式周围添加一对括号。

`printRule` 方法通过向屏幕打印用输入左手边符号命名为标记序列的语法规则，来提供调试支持。

`replaceRule` 方法代替如用输入符号命名的语法规则的标记矢量。

ExpandedRule

`ExpandedRule` 类的主要用途是，提供一种从起始符号开始扩展语法规则，并继续成行扩展所有产生规则、直到所有规则符号都被改进为字符集、字符串文字或递归符号为止的工具。字符集和字符串文字是能够被进一步改进的终结符号。由于递归符号递归进入相同状态的性质，递归符号需要堆栈执行其状态转换。单独的特殊过程将被执行，以处理递归符号。尽管为规则扩展起见，它们也被处理为好像是终结符号。

提供两个构造程序，来扩展传入的 `RuleMgr` 对象中包含的语法产生规则。为提供对多个规则表的独立处理，`RuleMgr` 成为构造程序的输入变元。构造程序所需的另一个输入变元是“语言起始符号”。这向构造程序提供扩展规则的起始点。两个构造程序之一还需要布尔标志变元，以指示是否需要压缩所得到的扩展产生规则。通过避免产生主要为调试目的而产生的标记、尤其是杂项标记，并积极将规则段变换为字符集，来执行压缩。这些构造程序是调用程序需要用来扩展语法规则的主要接口。构造程序将调用内部私有方法来成行扩展产生规则，导致了覆盖整个语言的单一语法规则。在扩展规则的过程中，这些方法也将识别递归符号。在扩展工作中，这些递归符号被处理为好像是终结符号。构造程序也将递归符号保存到 `RecursiveSymbolMgr` 所维护的表中，以便以后进行处理。在最高级产生规则已被扩展之后，调用程序可以调用“`expandAllRS`”方法，来扩展被构造程序识别和保存的

所有递归符号。

`expandAllRS` 和 `performSimpleExclude` 方法是 `ExpandedRule` 类中的所有其它外部接口。`expandAllRS` 方法从 `RecursiveSymbolMgr` 类得到所有递归符号的列表，并且每次一个地扩展每个递归符号。类似于最高级扩展，在扩展过程期间遇到的任何递归符号都将被处理为终结符号。这些递归符号将造成在状态转换表生成期间产生特殊动作码，使得该特殊动作码可以请求堆栈支持递归。

`performSimpleExclude` 方法遍历扩展的语法规则，以定位“排除(-)”算子。对于 `performSimpleExclude` 方法所遇到的每个“排除”算子，如果确定“排除”操作的操作数是具有字符文字的字符集、或两个字符集，则 `performSimpleExclude` 方法将立即执行“排除”操作，并用所得到的字符集来代替语法规则中的操作表达式。

`ExpandedRule` 中的其余方法是私有方法。这些方法是：

```
init  
isOnTheStack  
expand  
expandRS
```

`init` 方法帮助构造程序初始化类变量，以及起动语法规则成行扩展处理。

`isOnTheStack` 方法向构造程序提供内部支持，以确定语法符号是否为递归符号。软件通过将每个被扩展的符号推进堆栈中，来记住沿着扩展链的语法符号。一旦符号被完全扩展，该符号就从堆栈被托出。在扩展符号之前，代码检查符号是否已经在堆栈上。如果情况是这样的，则将符号识别为递归符号。

`expand` 方法是一种通过获得它所遇到的每个非终结符号的右边表达式、并用表达式来代替符号，来执行语法规则成行扩展的递归方法。`expand` 方法从起始符号开始，并且继续代替被扩展的规则中的每个符号，直到所有符号都变为终结符号或递归符号为止。堆栈用于在 `isOnTheStack` 方法中识别所有递归符号，如上所述。

`expandRS` 方法和上述 `expand` 方法很类似。`expandRS` 方法支持

expandAllRS 方法专门为递归符号扩展语法规则。类似于 expand 方法，通过复制代表用 ruleMgr 中的非终结符号命名的产生规则的标记矢量、并用标记矢量代替被扩展的规则中的符号，来执行扩展。连续重复该过程，直到被扩展的规则的所有符号都成为终结符号或递归符号为止。如果在扩展期间遇到递归符号，包括正被扩展的递归规则符号自己，则该递归符号被处理为好像是终结符号。

CharSet

CharSet 类支持一种用于存储语法产生规则中的表达式中所使用的有效字符集、或从语法规则中的子表达式得到的有效字符集的设置工具。最初在产生规则中指定的 EBNF 形式的字符集被封入一对方括号内。可以以多种方式表示方括号内的内容：

- 包含所有有效离散字符的字符序列

- 某一字符范围

- 被表示为十六进制值的单字符

- 利用十六进制值表示的字符范围

- 范围记号之外

- 以上的组合

CharSet 类所提供的方法将处理所有这些指定有效字符集的不同方式，并将它们转换为相对于调用程序透明的 CharSet 对象。从 CharSet 类可以得到允许调用程序维护 CharSet 对象的附加方法。

可以得到两个 CharSet 构造程序。无参数的构造程序允许调用程序设置一种要在稍后添加内容的 CharSet 对象。另一构造程序允许调用程序设置 CharSet，并通过指定用如上所述的信息格式化的串来初始化 CharSet 对象内容。

CharSet 类中定义的方法有：

- add

- remove

- isIn

- isEqual

```
print  
charCount  
iterator
```

有三种重载“add”方法。每种 add 方法都允许调用程序将更多字符添加到 CharSet 对象中。第一种变型允许调用程序利用如上所述的串格式指定多个字符。第二种 add 方法允许调用程序向 CharSet 对象添加字符。而第三种变型允许调用程序将另一 CharSet 对象的内容复制到当前对象中。

有两种重载“remove”方法。第一种型式允许调用程序从当前 CharSet 对象中删除字符。第二种型式接收 CharSet 对象作为输入参数。它从当前 CharSet 对象中删除在输入 CharSet 中发现的所有字符。

isIn 方法允许调用程序查明当前在 CharSet 对象中是否有特殊字符。

isEqual 方法把另一个 CharSet 对象和当前对象进行对比，以确定它们是否具有相同内容。

print 方法是为调试目的而设的。print 方法向屏幕打印 CharSet 对象的当前内容。

charCount 方法返回 CharSet 中当前的字符数。

iterator 方法将迭代程序对象返回给调用程序，允许调用程序每次一个地访问 CharSet 内的每个字符。

为支持 iterator 方法，CharSet 类也包含内部类 CharSetIterator。CharSetIterator 是 Iterator 接口的实施。

RecursiveSymbolMgr

RecursiveSymbolMgr 维护散列表，允许调用程序设置表，以包含本质上递归的产生规则。递归符号表被 InputMgr、ExpandedRule 和 NFAMgr 类使用。RecursiveSymbolMgr 类利用构造程序来生成 Java 散列表。因为是利用 Java 散列表来实施表的，所以利用散列表方法来执行对递归符号表的访问和维护。RecursiveSymbolMgr 类不定义任何附加方法。

RSEntry

RSEntry 类定义被实施为 RecursiveSymbolMgr 类中的散列表的递归符号表的项目结构。RSEntry 类的用途是定义数据结构。因而，只提供构造程序来初始化类变量。数据结构中的所有字段都可以利用它们的本来的（native）方法来直接访问。

NFAMgr

NFAMgr 类对把扩展的语法产生规则变换为 NFA 提供支持。NFAMgr 类封装用于存储从扩展的输入语法规则产生的状态转换信息的 StateMgr 类。用 NFAMgr 构造程序来例示 StateMgr。除构造程序以外，NFAMgr 类也定义以下方法：

```
genStates
genNFA
findLoopbackState
checkAttributeNext
eliminateDoubleEpsilons
optimizeEpsilonTransitions
```

genStates 方法允许调用程序起动将扩展的语法规则变换为 NFA 的处理。输入扩展语法规则作为标记矢量被传入。然后，genStates 方法调用递归 genNFA 方法，来将扩展的语法规则分解为可管理的段、并将这些段转换为状态转换。

genNFA 方法每次以递归形式处理一段输入扩展语法规则，直到整个语法规则被变换为完整的 NFA 为止。通过分组和识别语法规则定义中使用的普通子表达式，来执行处理，如图 5A 至 5I 所示。

图 5A 至 5I 通过各个图中包含的标示显示了几种通常出现的被描述为以上定义的 NFA 的语言模式。例如，图 5A 显示了代表“a”出现零次或多次的模式“a*”；图 5B 显示了代表“a”出现零次或一次的模式“a?”；等等。相应模式的这种记号和逻辑处理是编译器中用于具体表示这些模式的众所周知技术。然而，因为一个输入，如 ϵ （epsilon：

厄普西隆、空输入)，可以造成多种状态转换，如图 5D 中的步骤 2)，所以最后必须将这种表示改变为 DFA，如以上所提到的。

优选地，在这一点上不以最优形式执行变换，以便产生普通状态转换模式，使分组和组合语法规则子表达式的结果变得容易。一旦生成完整的 NFA 状态转换序列，就将消除冗余状态，并将组合普通状态。

`findLoopbackState` 方法支持 `checkAttributeNext` 方法中的属性（即*+?）变换处理，以确定当前语法子表达式组的起始状态，使得可以正确地为每一属性添加一个或多个转换弧（transition arcs）。

`checkAttributeNext` 方法查明是否为刚刚被变换为 NFA 序列的语法规则子表达式定义了属性。如果发现属性，则 `checkAttributeNext` 方法将在 NFA 中添加适当的转换，以满足属性规范。

`eliminateDoubleEpsilons` 方法优化 NFA 转换序列，以消除冗余状态转换。

`optimizeEpsilonTransitions` 方法消除完整 NFA 状态转换序列内的外来转换。

StateMgr

`StateMgr` 类支持状态转换表的创建和维护。`StateMgr` 类对 `NFAMgr` 类和 `DFAMgr` 类两者提供支持。类构造程序初始化类变量，并为状态转换表分配内存。另外，构造程序创建将 NFA 状态（旧状态）映射到 DFA 状态（新状态）的散列表，来支持 DFA 变换。`StateMgr` 类中定义的其他方法有：

```
assignNewState  
recycleState  
addStateTransition  
removeStateTransition  
getAllOutTransitions  
getAllInTransitions  
getEpsilonOutTransitions  
getEpsilonInTransitions
```

```
getEpsilonArcs  
getNonEpsilonOutTransitions  
getNonEpsilonInTransitions  
getNonEpsilonArcs  
allocateEntry  
recycleEntry  
updateEntry  
getEntry  
locateState  
printStatistics  
printStateWithExt  
printState  
listStatesWithNFASet  
listStatesWithClosureStateSet  
peekNextNewStateNum  
writeXMLOutput
```

`assignNewState` 方法保留状态表项目，并返回要用于新转换状态的相应状态号。

`recycleState` 方法允许调用程序将状态表项目释放回到池中，以便重新分配。

`addStateTransition` 方法根据输入转换信息，来创建从当前状态到下一状态的转换弧。`addStateTransition` 方法也创建相对于调用程序透明的从下一状态返回到当前状态的反向链接。

`removeStateTransition` 方法删除两种状态之间的转换弧。`removeStateTransition` 方法删除关于两种状态之间的相同转换的正向和反向链接。

`getAllOutTransitions` 方法把与指定状态相关的所有外出转换 (outbound transition) 列表返回给调用程序。

`getAllInTransitions` 方法把与指定状态相关的所有进入转换 (inbound transition) 列表返回给调用程序。

`getEpsilonOutTransitions` 方法把与指定状态相关的、由“空”输入造成的外出厄普西隆转换 (outbound epsilon transition) 列表返回给调用程序。

`getEpsilonInTransitions` 方法把与指定状态相关的进入厄普西隆转换 (inbound epsilon transition) 列表返回给调用程序。

`getEpsilonArcs` 方法返回与从传入的转换列表中取出的厄普西隆输入相关的转换列表。该方法主要为支持 `getEpsilonOutTransitions` 和 `getEpsilonInTransitions` 方法而存在。

`getNonEpsilonOutTransitions` 方法向调用程序返回把与指定状态相关的厄普西隆转换排除在外的所有外出转换列表。

`getNonEpsilonInTransitions` 方法向调用程序返回把与指定状态相关的厄普西隆转换排除在外的所有进入转换列表。

`getNonEpsilonArcs` 方法返回与从传入的转换列表中取出的厄普西隆输入不相关的转换列表。该方法主要为支持 `getNonEpsilonOutTransitions` 和 `getNonEpsilonInTransitions` 方法而存在。

`allocateEntry` 方法从本地控制的状态表项目矢量中分配状态表项目。

`recycleEntry` 方法将状态表项目放到要重新使用的状态表项目列表上。

`updateEntry` 方法将状态项信息复制到 `StateMgr` 类对象内部维护的状态表矢量中的适当位置中。

`getEntry` 方法从内部状态表矢量检索与状态相关的信息。

`locateState` 方法对 DFA 变换提供支持。如果存在为匹配输入参数的一组 NFA 状态而生成的匹配 DFA 状态, `locateState` 方法将找到该匹配 DFA 状态。

`printStatistics` 方法提供调试支持。`printStatistics` 方法向屏幕打印出与内部受控的状态表相关的使用信息。

`printStateWithExt` 方法提供调试支持。`printStateWithExt` 方法

打印与具有为支持 DFA 变换而维护的附加信息的状态相关的所有信息。

`printState` 方法提供调试支持。`printState` 方法打印与状态相关的所有信息。

`listStatesWithNFASet` 方法返回包括指定 NFA 状态集的 DFA 状态列表。

`listStatesWithClosureSet` 方法返回作为厄普西隆闭包 (epsilon closure) 一部分的状态列表。

`peekNextNewStateNum` 方法返回要分配给下一新状态的状态号。

`writeXMLOutput` 方法支持以 XML 格式将状态表写出到输出文件流。

StateEntry

`StateEntry` 类定义状态表项目的内容。状态项包含三个主要字段：状态号、外出转换弧列表、以及进入转换弧列表。有两个为支持 DFA 变换而定义的附加字段：被替代的 NFA 状态集，以及空输入转换闭态集。类构造程序初始化字段，并创建关于外出弧和进入弧的矢量。`StateEntry` 类支持状态表项目的创建和维护，`StateEntry` 类也定义以下方法：

```
addToArc
addFromArc
removeToArc
removeFromArc
doesTransitionExist
removeArc
compareNFASets
printToArcs
printFromArcs
printArc
printExtension
```

`isInNFASet`

`isInClosureSet`

`writeXMLOutput`

`addToArc` 方法把当前状态的外出转换项添加到外出转换弧矢量上。

`addFromArc` 方法把当前状态的进入转换项添加到进入转换弧矢量上。

`removeToArc` 方法从外出转换弧矢量中删除当前状态的外出转换项。

`removeFromArc` 方法从进入转换弧矢量中删除当前状态的进入转换项。

`doesTransitionExist` 方法允许调用程序执行查询，以确定指定的转换是否和外出转换弧矢量中的任一转换项匹配。

`removeArc` 方法支持 `removeToArc` 和 `removeFromArc` 方法从传入的转换弧矢量中删除特殊转换项。

`compareNFASets` 方法比较输入的 NFA 状态集是否和正被当前 DFA 状态代替的 NFA 状态集匹配。

`printToArcs` 方法提供调试支持，以便打印出当前状态的所有外出转换弧的信息。

`printFromArcs` 方法提供调试支持，以便打印出当前状态的所有进入转换弧的信息。

`printArc` 方法支持 `printToArcs` 和 `printFromArcs` 方法向屏幕打印出传入的转换弧矢量中存储的所有转换项信息。

`printExtension` 方法提供调试支持，以便向屏幕打印出状态项中维护的 DFA 变换支持信息。

`isInNFASet` 方法提供 DFA 变换支持，以检查在当前状态项内维护的 NFA 状态集中是否已经包括特殊 NFA 状态。

`isInClosureSet` 方法提供 DFA 变换支持，以检查在当前状态项内维护的空输入闭态集中是否已经包括特殊 NFA 状态。

`writeXMLOutput` 方法支持以 XML 格式将状态表项目写出到输出文

件。

TransitionEntry

TransitionEntry 类为用于描述从一种状态转到另一种状态的转换弧的信息，定义数据字段。该信息包括造成状态转换的输入的类型；造成状态转换的输入的实际值；以及该状态转换所造成的下一状态的状态号。有六个类构造程序可用于初始化和设置适当数据字段中的输入数据信息，使得转换项已准备好使用。这些构造程序具有不同的输入参数来匹配转换输入数据类型。为 TransitionEntry 类定义了以下允许调用程序访问和更新数据字段的方法：

- clear
- setSymbolName
- setInput
- setTransition
- setCheckedFlag
- getInputType
- getCharSet
- getInputChar
- getTransition
- getSymbolName
- setCheckedFlag
- isEqual
- compareInput
- copyInput
- print
- writeXMLCharInput
- writeXMLOutput

clear 方法将所有数据字段都设置为一种初始已知状态。

setSymbolName 方法将转换输入类型设置为“RELOCATE”，以指示可能需要分支转到另一状态表来处理递归符号。符号名作为输入参数

被传入，并且被保存在符号名字段中以便以后参考。

setInput 方法由三种重载方法组成，它们的不同之处仅在于输入参数。第一种 setInput 型式不需要任何输入。它把转换项的转换输入类型设置为空（厄普西隆）输入。第二种型式需要字符输入参数。该方法将转换项输入类型设置为字符类型，并保存输入字符值。第三种型式需要 CharSet 输入参数。它将转换项输入类型设置为 CharSet，并保存 CharSet 值。

setTransition 方法允许调用程序指定要转到的转换状态号。

setCheckedFlag 方法支持 DFA 变换。它允许 DFA 变换处理标明该转换项，使得该项只被处理一次，以便加速变换。

getInputType 方法把该转换项的输入类型返回给调用程序。

getCharSet 方法把该转换项的输入 CharSet 值返回给调用程序。

getInputChar 方法把该转换项的输入字符值返回给调用程序。

getTransition 方法返回该转换项中指定的转换状态号。

getSymbolName 方法把该项中存储的输入符号值返回给调用程序。

getCheckedFlag 方法把该项中的 CheckedFlag 当前标志设置返回给调用程序。

isEqual 方法对包括作为输入参数传入的转换项中存储的转换状态信息的所有值和该转换项中存储的那些值进行比较。如果这些值相同，则 isEqual 方法返回真；否则，返回假。

compareInput 方法对作为输入参数传入的转换项中存储的输入类型及输入值和该转换项中存储的输入类型及输入值进行比较。如果这些值相同，则 compareInput 方法返回真；否则，返回假。

copyInput 方法允许调用程序把输入类型和输入值信息从作为输入参数传入的转换项复制到当前项。

print 方法提供调试支持，以便向屏幕打印出该转换项的内容。

writeXMLCharInput 方法通过确定输入字符是否为可打印 ASCII 字符，来支持 writeXMLOutput 方法，并以适当的 XML 格式将输入字符输出到输出文件流。

`writeXMLOutput` 方法支持以 XML 格式将状态转换信息写出到输出文件流。

DFAMgr

DFAMgr 类支持将 NFA 变换为 DFA。DFAMgr 类构造程序接收包含要被变换为 DFA 的 NFA 状态表的 NFAMgr，作为输入。DFAMgr 类构造程序还需要两个附加参数来指定 NFA 起始状态和 NFA 最终状态，使得 DFAMgr 能够将它们映射为 DFA 起始状态和 DFA 最终状态。构造程序创建新 StateMgr，来维护要产生的新 DFA 状态。在 DFAMgr 类对象被构造之后，调用程序可以调用 `NFA2DFA` 方法来执行 DFA 变换。以下是 DFAMgr 所定义的方法列表：

```
createDFAState  
NFA2DFA  
addEpsilonOutStates  
eClosure  
getNFATransitionSet  
extractNFAInputSet  
extractNFATargetStateSet  
findDFAFinalStates  
printFinalStates  
writeXMLOutput
```

`createDFAState` 方法支持 `NFA2DFA` 方法执行 DFA 变换。`createDFAState` 方法为新 DFA 状态创建状态表项目。在创建状态项之后，`createDFAState` 方法用关联的 NFA 状态集和厄普西隆闭集来初始化状态项。

`NFA2DFA` 方法是用于执行将 NFA 变换为 DFA 的主要方法。`NFA2DFA` 方法使用某些公知编译器构造技术来将 NFA 变换为 DFA。

`addEpsilonOutStates` 是一种为支持 `eClosure` 方法而存在的递归方法。`addEpsilonOutStates` 方法以一种递归方式将厄普西隆（空输入）转换状态添加到来源于被映射到 DFA 状态的 NFA 状态集的闭集。

`eClosure` 方法建立并返回与作为输入参数传入的 NFA 状态集关联的厄普西隆闭态集。

`getNFATransitionSet` 方法建立并返回与作为输入参数传入的状态集关联的非厄普西隆转换项集合。

`extractNFAInputSet` 方法查看作为输入参数传入的转换项集合，并把从这些转换项中提取的输入集返回给调用程序。

`extractNFATargetStateSet` 方法查看作为第一输入参数传入的转换项集合，并返回具有与作为该方法第二输入参数传入的转换项中指定的输入匹配的输入的目标状态集。

`findDFAFinalStates` 方法返回被指定为 DFA 状态表中允许最终状态的 DFA 状态集。该 DFA 状态集是根据作为输入参数传入的原始 NFA 最终状态来确定的。

`printFinalStates` 方法提供调试支持，以便向屏幕打印出如通过 `NFA2DFA` 方法确定的 DFA 最终状态集。

`writeXMLOutput` 方法支持以 XML 格式把与 `DFAMgr` 创建的 DFA 相对应的状态表写出到输出文件流。

参考图 6，图 6 显示了被表示为 XML 文件的状态转换规范输出的例子。600 的文件头识别文件内容、产生文件的日期、以及语法规则输入源。610 的文件下一部分提供某些关于被指定的状态表的身份和布局的一般信息。在 611，它识别该文件中描述的逻辑状态表数。加载器可以通过把来自后续逻辑状态表的状态附加到第一逻辑状态表上，并据此调节它们的转换，来把这些逻辑状态表组合成一个单物理状态表。（例如，如果物理状态表中的当前最后状态是 1205。物理状态表中的下一可用状态项是 1206。为了将下一逻辑状态表附加到物理状态表上，把在逻辑上被标定为状态 0 的初始状态加载到物理状态表项 1206 上。从逻辑状态表的所有状态转换都将被调节 1206 的偏移量。因此，如果有到逻辑状态表的状态 5 的转换，则在物理状态表中该转换将变为 1211（1206+5）。）在 612，它识别逻辑表的名称。递归符号它们自己用作递归符号逻辑状态表的名称。在 613，它提供用于标定物理状态表列（状态输入）的信息。620 的文件下一段提供关于每个

逻辑状态表的详细规范。621 的部分提供对该文件所指定的逻辑状态表的完整描述。它通过 622 的名称来识别表。然后，它在 623 识别该状态表的逻辑初始状态。624 列出了允许最终状态。625 指定了该逻辑状态表的状态数。626 的文件部分识别该逻辑状态表的所有不同状态及其转换的详细信息。它首先提供如 627 所示的逻辑状态号。然后，它在 628 列出在各种输入的情况下，来源于该状态的所有转换。在 629 识别具有到该逻辑状态的转换的状态。对逻辑状态表中的每种状态都重复 626 的文件部分。并且，对每一逻辑状态表都重复在 621 指定的信息。这向加载器提供用于使硬件加速器个性化的完整信息。

由以上描述来看，可以看到，本发明能够优选地以诸如 BNF 或其派生物的形式记号，直接自动地从语言或功能规范、为任何计算机语言或其它目的提供无差错状态表数据。过程可以迅速地执行，并以低成本产生无差错状态表。从而，本发明允许随意迅速地改变 FSM 的个性，以适应或提供不同功能、或反映所关心的不同语言或字符串。

虽然用单个优选实施例描述了本发明，但是本领域技术人员应该认识到，可以在所附权利要求的精神和范围进行修改来实施本发明。

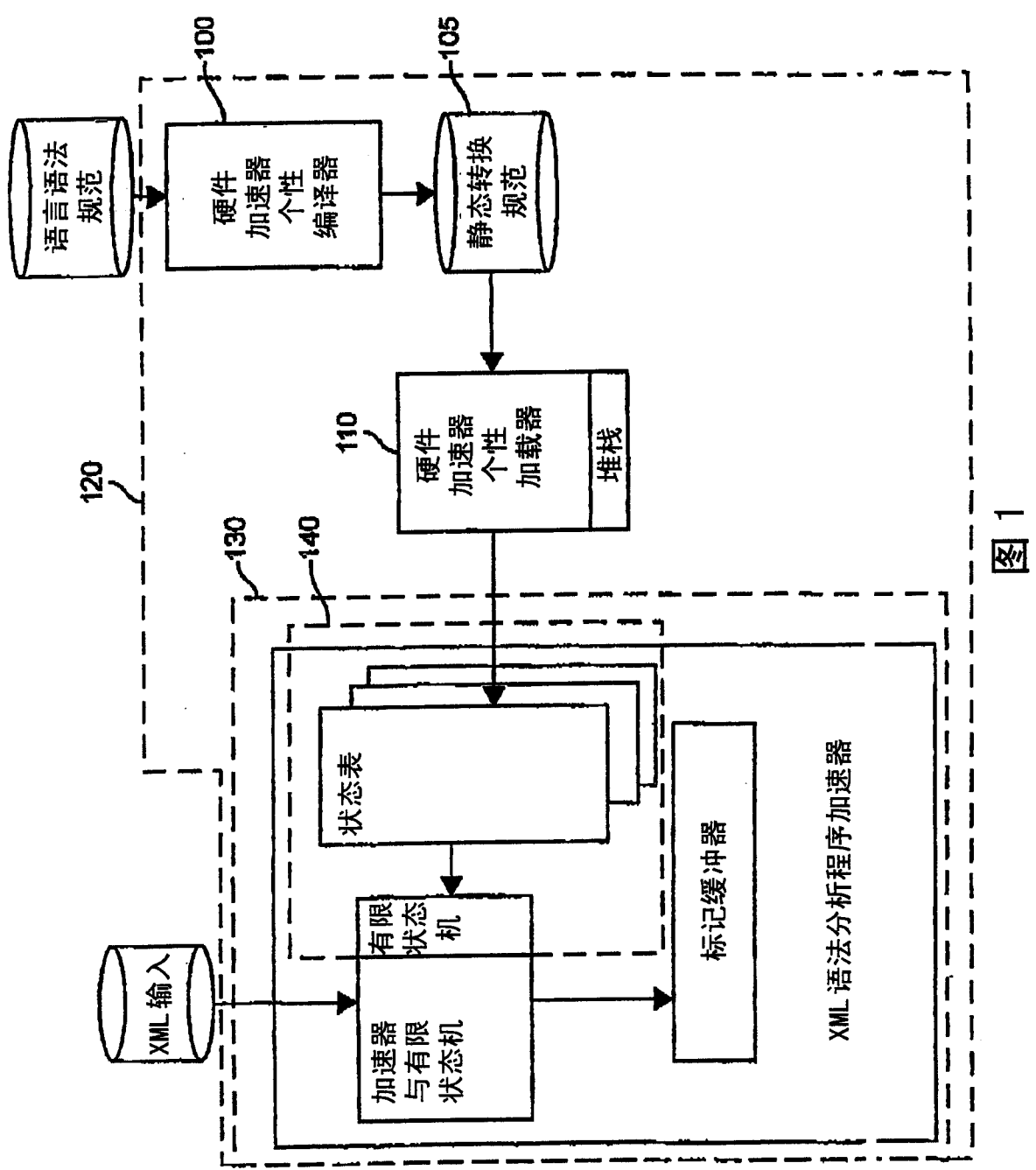


图 1

XML 语法分析程序状态表例子

ASCII 值

索引 基地址	00	01	02	03		FD	FE	FF
状态 0	保持在 状态 0	保持在 状态 0	转到 状态 1	保持在 状态 0		保持在 状态 0 特殊中断	保持在 状态 0	保持在 状态 0
状态 1	保持在 状态 1	转到 状态 2	转到 状态 0	保持在 状态 1		转到 状态 4	转到 状态 0	转到 状态 0
状态 2	保持在 状态 2	转到 状态 3	转到 状态 0	转到 状态 3 特殊中断		转到 状态 0	转到 状态 0	转到 状态 0
状态 3	保持在 状态 3	转到 状态 0	转到 状态 0 存储标记	转到 状态 0		转到 状态 0	转到 状态 0	转到 状态 0
状态 4	保持在 状态 4	转到 状态 0 存储标记	转到 状态 0	转到 状态 0		转到 状态 0	转到 状态 5	转到 状态 0
状态 5	保持在 状态 5	转到 状态 0	转到 状态 0	转到 状态 0		转到 状态 0	转到 状态 0	转到 状态 0 存储标记

图 2A

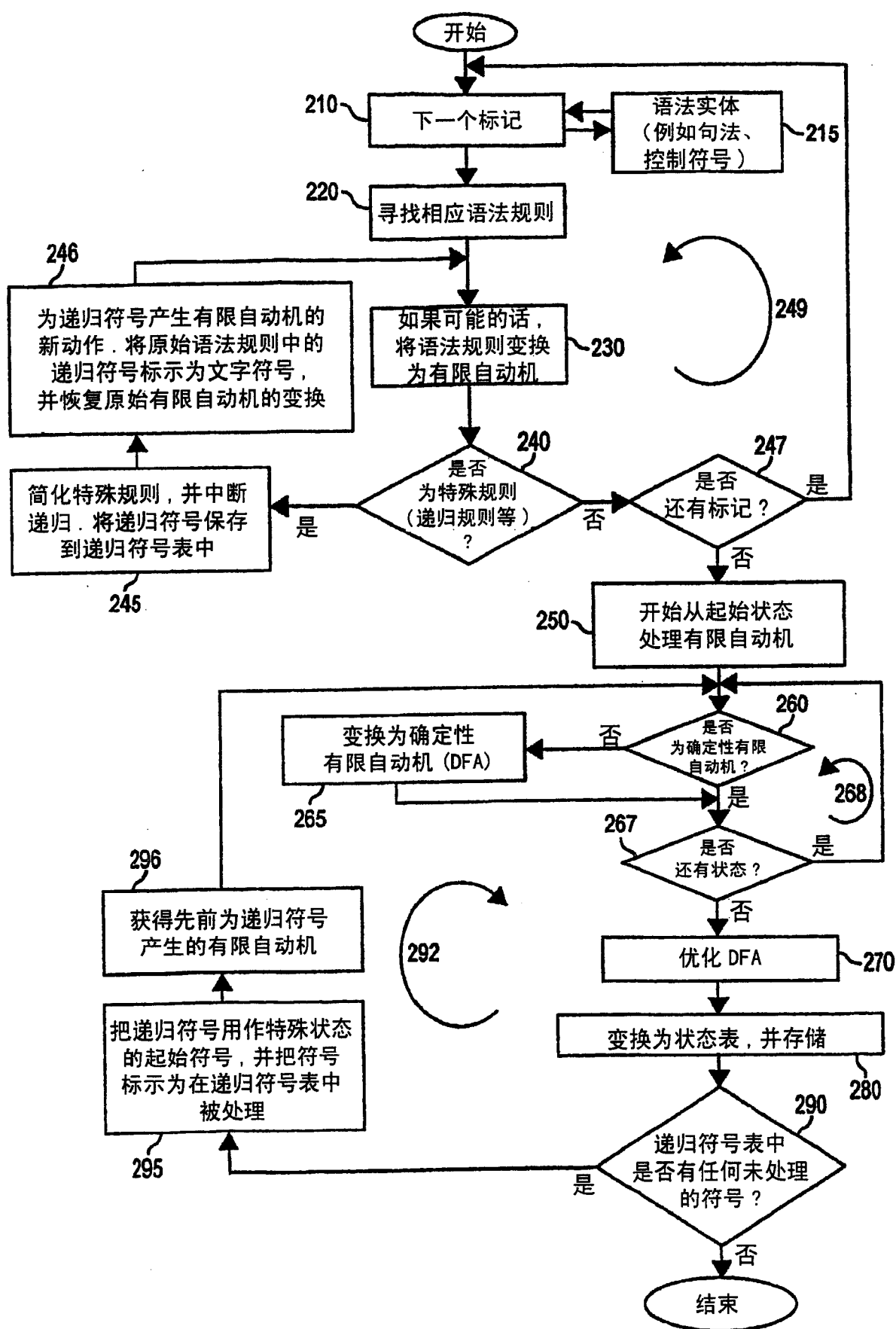


图 2B

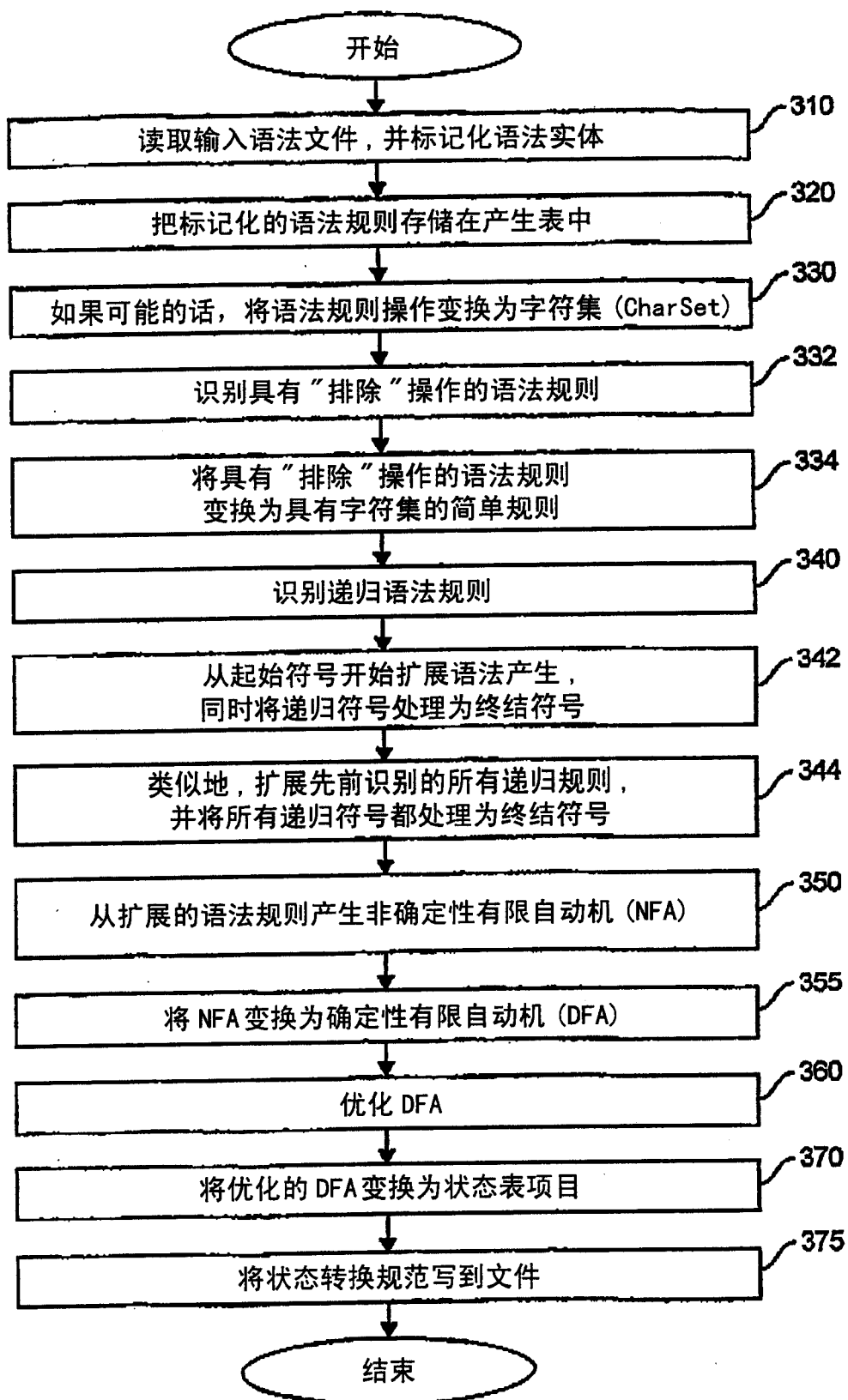


图 3

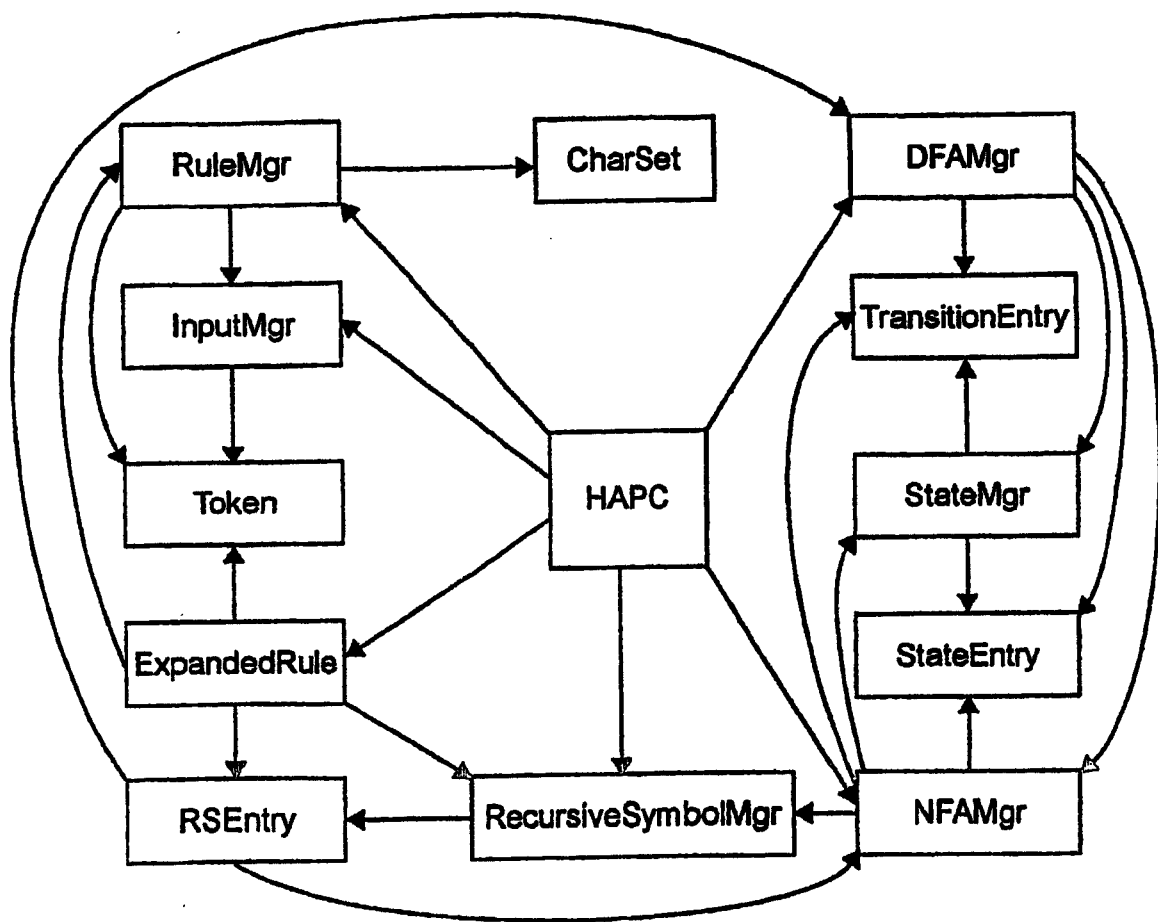


图 4

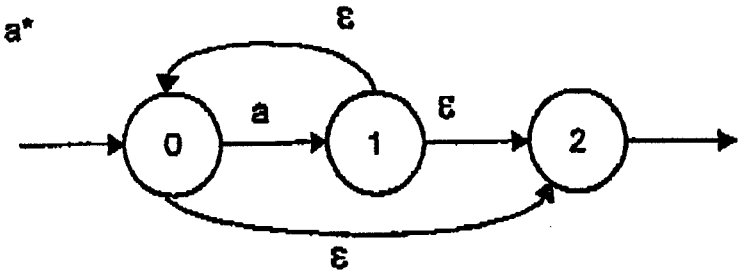


图 5A

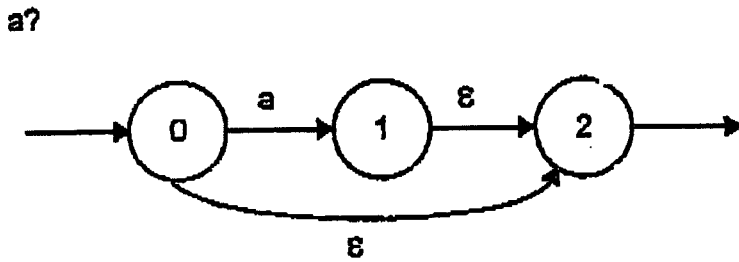


图 5B

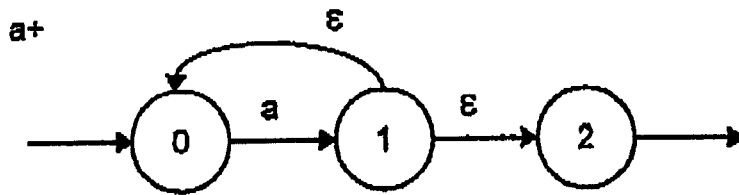


图 5C

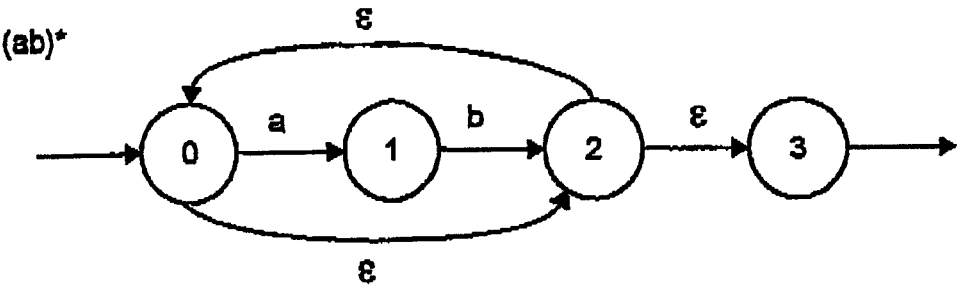


图 5D

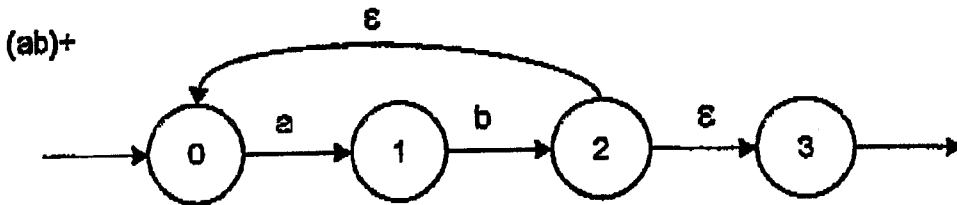


图 5E

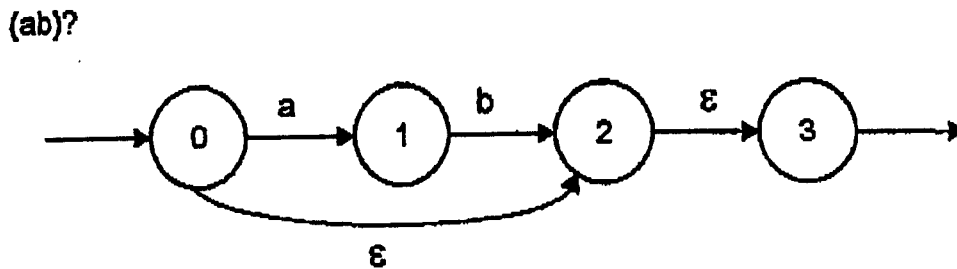


图 5F

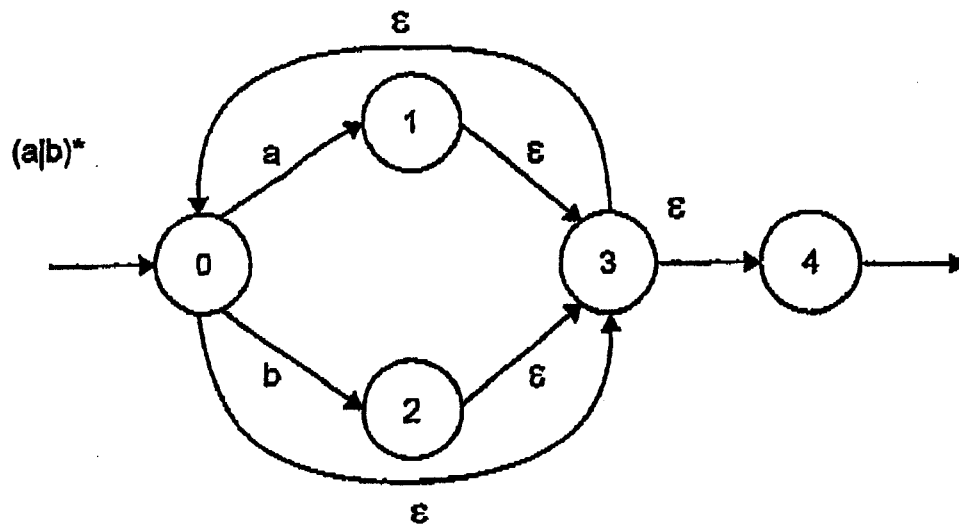


图 5G

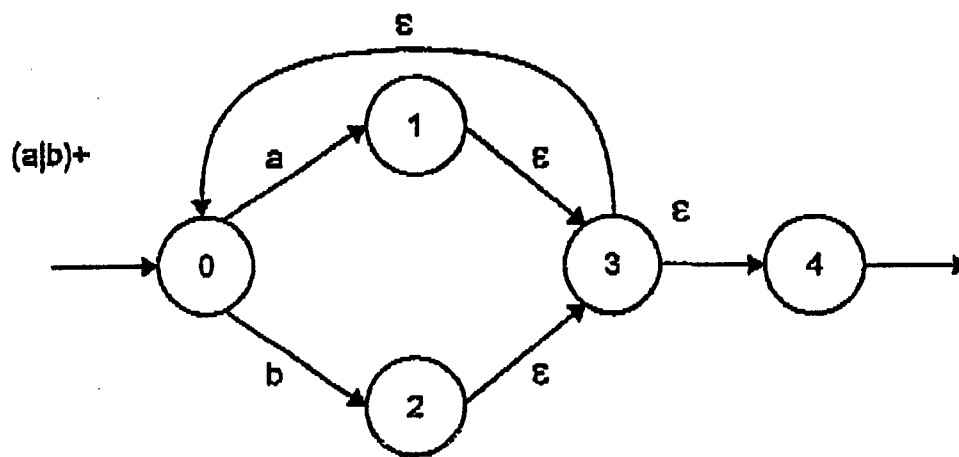


图 5H

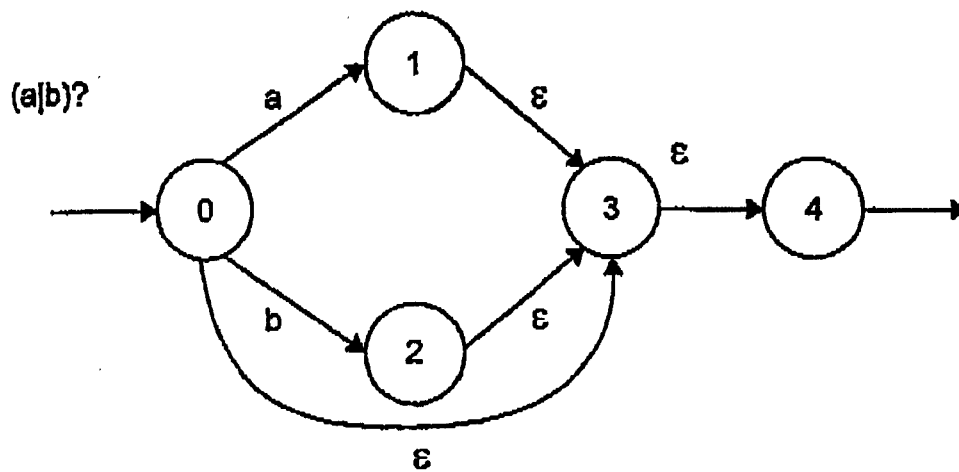


图 5 I

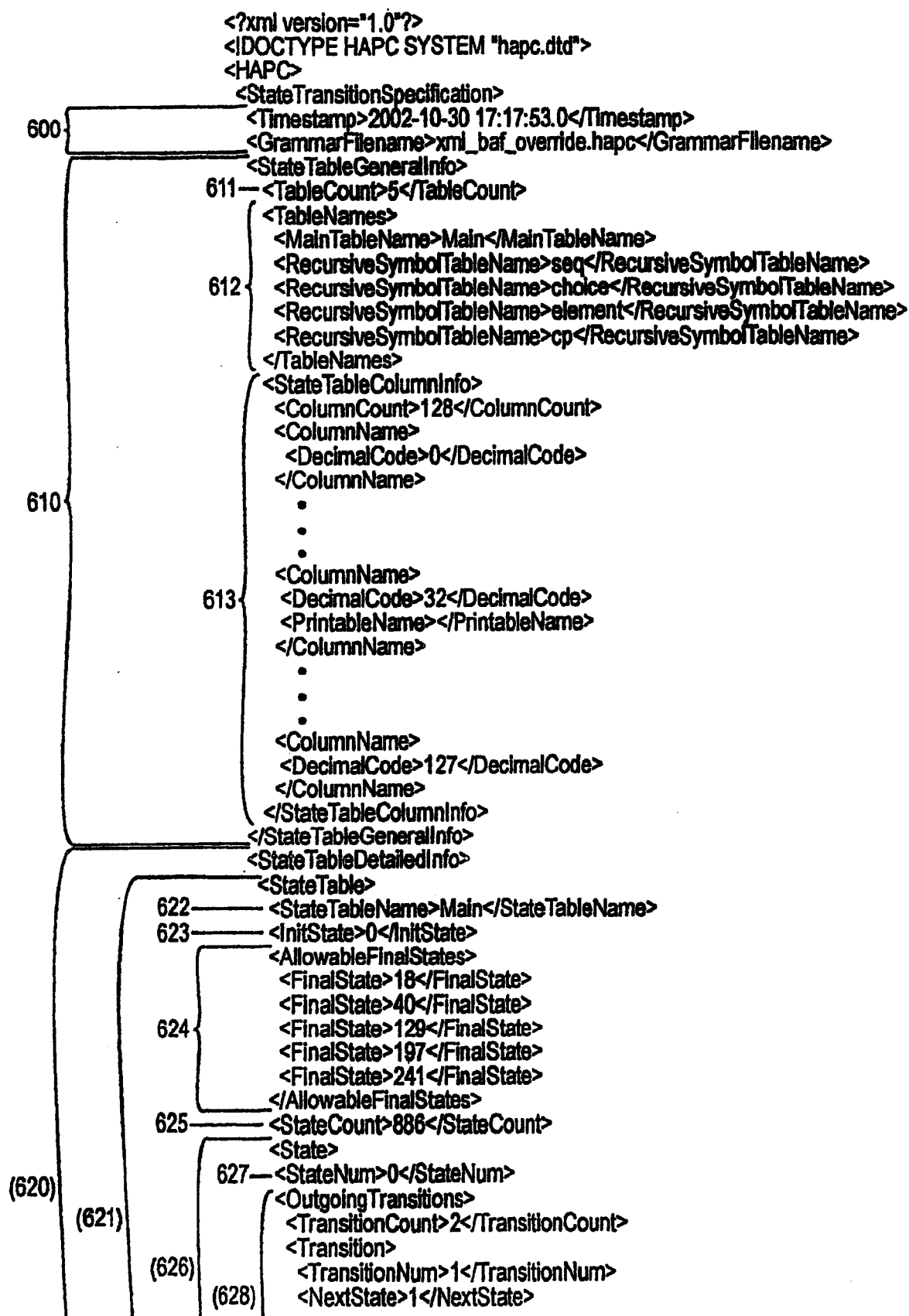


图 6A

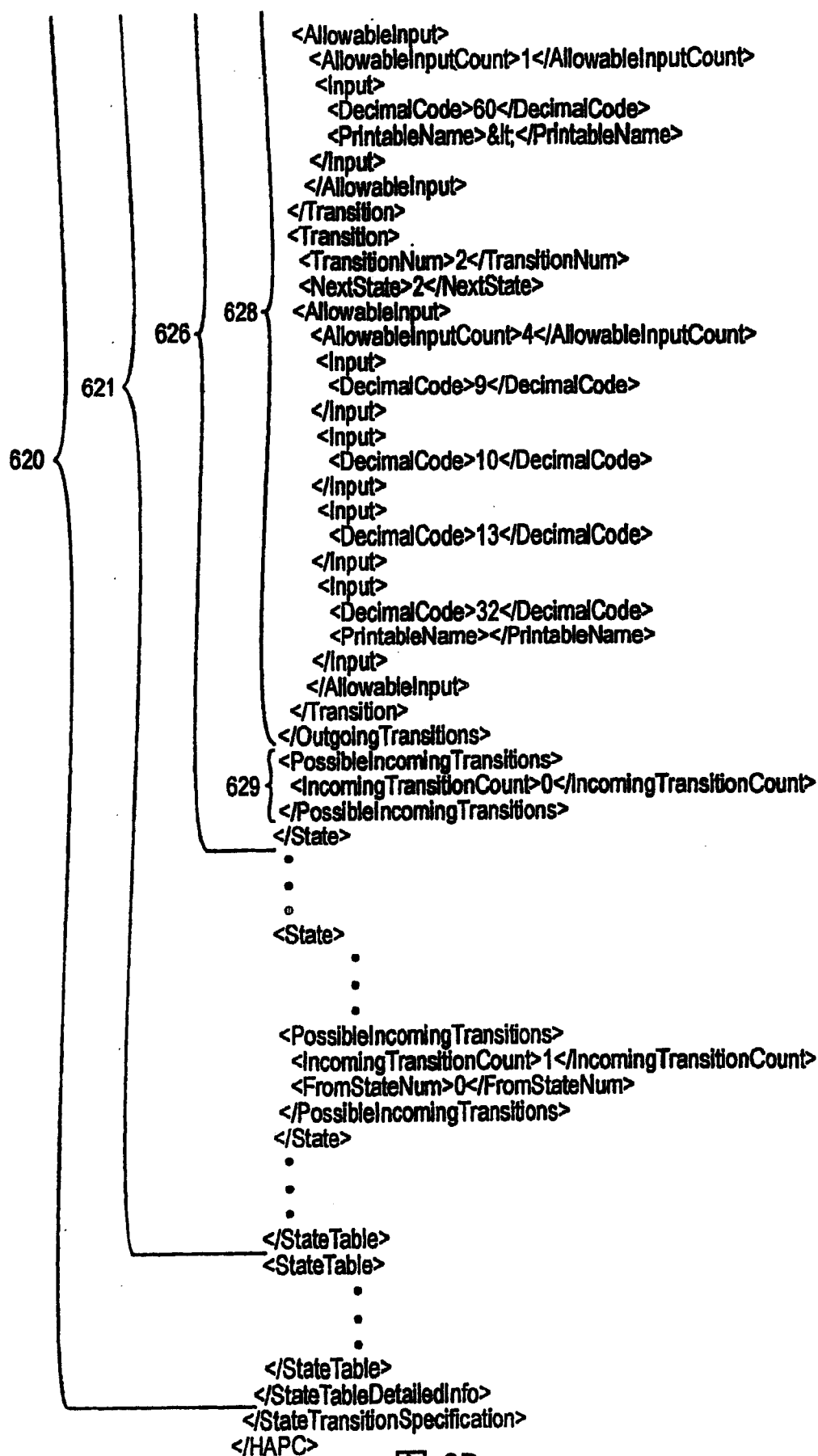


图 6B