1 一个简单的自然语言理解系统

1·1 自动作图世界

设想我们已经在微型计算机上建成了一个自动作图系统,该系统能接受由简单的作图语言(一种人工的机器内部语言)编制的指令,并能按照指令的要求储存信息,或在显示屏上作图。

现在我们给该系统装备一个用户友好接口，该接口能够接受用户用汉语发出的指令，并把这些指令翻译为相应的作图语言的指令。这样，我们就建成了一个类似于维纳格雷德的‘积木世界’[①] 的作图世界。显然，在这个简单的作图世界里所需使用的自然语言词汇、句子等是十分有限的,因此,要建成这样一个用户友好接口并不是很困难的。然而，由这个简单的作图世界出发，我们可以对涉及自然语言理解的方方

① 参见 Winograd(1972)。

面面的问题展开充分的阐释，并进而引向一些更为复杂的方法和策略方面问题的探讨。

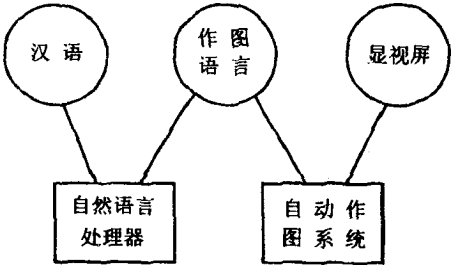

图 1·1

图 1·1 显示了我们这个作图世界的结构框架。由用户终端输入的是数量有限的一些汉语句子。这些句子受到一个严格而有限的语法控制，构成汉语句子集合中的一个子集。输入的句子经过自然语言处理器的处理，被翻译成意义上等值的作图语言的指令，然后再输入到自动作图系统中去。自动作图系统则按照作图语言表示的指令作出反应，把点、线、几何图形等输出到显示屏的相应位置上去。

从自然语言理解角度看，这里的关键就在于能否把汉语的句子翻译成计算机内部相应的作图语言的指令。如果一个输入的汉语句子能被译成作图语言的指令，该句子就是有意义的，或能被理解的；不然，就是没有意义的，或不能被理解的。前者处于作图世界之内，后者则在这个世界的边界之外。

1·2 内部文本的分类及其形式

能直接促进自动作图系统动作的，是用作图语言表达的指令，我们称之为“内部文本”。内部文本可以分为两类。一类是点的赋值语句，如在一个从标为 1000×1000 的显示屏上确定一个点。这类语句的形式为：

point(〈用户定义的点的名称〉(数字)〈数字〉)；'point'是关键词，表示要在显示屏上确定一个点。后面括弧内的第一项是用户可以任意定义的点的名称，它可以用一个小写字母来表示，也可以由一个以小写字母开头的字符串来表示（字符串的长度一般都有一定的规定）。后两项是两个 0 到 1000 之间的数字，代表用户定义的点的座标值。下面这些都是合格的点的赋值语句：

point (p, 123, 456)
point (q, 678, 901)
point (r, 987, 654)

内部文本的另一类是连接显示屏上任意两个点的动作语句。这类语句的形式为：

line－seg(〈用户定义的点的名称〉，〈用户定义的点的名称〉)

或：

line－seg(〈数字〉，〈数字〉，〈数字〉，〈数字〉)

“line－seg”是关键词，表示要在显示屏上作一条线段。后面括弧内的是用户定义的两个点 既可以是点的名称,也可以是点的座标值。下面这些都是合格的动作语句：

line－seg(p,q)
line－seg(q,r)
line－seg(r,p)
line－seg(123,456,678,890)

1·3　输入文本的大致范围

作图语言作为一种人工语言，在它的语句和意义之间有严格的一一对应的关系。就是说，一定形式的语句必定表示一定的意义；反过来一样，一定的意义必定由一定形式的语句来表示。不存在发生任何歧义的可能性。因此，计算机能顺利地将它解释或编辑为由一串数字来表示的机器指令。

自然语言的情况就不同了。在自然语言的语句和意义之间并不存在严格对应的关系。常常是，同一个意思，可以这么说,也可以那么说 或者同一个语句,可以这么理解,也可以那么理解。前者是所谓同义异构现象，后者是所谓的同构异义现象。

我们暂时只考虑同义异构现象。假定用户要求自动作图系统在显示屏上作一个三角形，他可以在计算机终端上敲入这样几句话：

设点 p、q、r 的座标分别为 234,456;678,901;

543,654。连接 pq,qr,pr。

也可以敲入那样几句话：

设 p 是 234,456;q 是 678,901;r 是 543,654。

从 p 分别引直线 pq 和 pr 再从 q 引直线 qr。

显然，相同的意思可以有许许多多不同的表达法。为简便易做起见，我们希望对允许输入的汉语句子能有一定的限制。这一点可以通过下面两个办法来实现，即

(1)规定一个允许使用的词汇子集；

(2) 规定一组允许使用的语法规则。

即使作了这样的限制，比起内部文本来，输入文本的仍然有丰富得多、灵活得多的表达方式。

1·4 输入文本的分类和分析

输入文本尽管有较为丰富、灵活的表达方式，但是从它所表达的意义或内容来看，仍然跟内部文本一样，只可分为两类：一类是赋值语句，另一类是动作语句。下面我们分别来分析这两类语句。

1·4·1 赋值语句的分析

输入允许的合格的赋值语句采取下面这些形式：

设点 p 为 x,y

设点 p 为 x　y

设 p 为 x, y

设 p 为 x　y

设点 p 的座标为 x, y

设 p 的座标为 x, y

p 的座标是 x, y

点 p 的座标为 x, y

p 的座标为 x, y

p 为 x, y

p 是 x, y

设 x, y 是 p 的座标

x, y 为点 p 的座标

x, y 为 p

x, y 是 p

等 等 。

这些赋值语句从最高层次上看可以分为两类：一类是句首带“设”的，另一类可以看作省略了“设”的。我们把它们概括为“设〈点的定义〉”和〈点的定义〉”这样两类。〈点的定义〉本身则由三个部分组成，分别为〈点的名称〉、〈动词〉和〈坐标值〉。用尖括号括起来的成分代表一个类别的集合，可以用它所包含的元素来定义；元素本身仍然可以是一个类别的集合，当然也可以是单个的原子，即词本身。

现在我们用 BNF(Backus－Naur Form)表述式来定义这些类别的集合：

(a) 赋值语句 :: = 设(点的定义) | 〈点的定义〉

(b) 点的定义 :: = (点的名称) 〈动词〉 〈座标短语)

| 〈座标短语〉 〈动词〉 〈点的名称〉

(c)动词 :: = 为 | 是

(d) 坐标短语 :: = 〈数字〉, 〈数字〉| 〈数字〉〈数字〉

(e) 数字 :: = 任何一个小于或等于 1000 的自然数

(f) 点的名称 :: = (点)〈用户定义的名称〉 |(点)

〈用户定义的名称〉的座标

(g) 用户定义的名称 :: = 除 x, y, u, v 之外的任何一个由用户指定的小写字母。

上述每个 BNF 表述式都代表一条规则。每条规则都由头部和尾部两部分组成。头部是被定义项,尾部是定义项,中间由“ :: = ”连结。“ :: = ”读作“ 定义为 ”,尾部出现的竖杠“ | ”,意为“ 或者 ”,圆括弧表示其中的成分可出现可不出现。BNF 表述式在排列顺序上是完全自由的。只是为了以后指称起来方便一点,才给它们编了次序。

上述 7 条规则组成一部语法,规定了赋值语句可以形成的模式或句型。

下面我们来讨论一下赋值语句自动分析的实现。

从刚才提出的那部由 BNF 表述式表述的语法中可以看出,赋值语句可由“ 点的定义 ”或“ 点的定义 ”前加“ 设 ”构成,而“ 点的定义 ”又由“ 点的名称 ”后接一个动词,再接一个“ 座标短语 ”构成,或者倒过来,由“ 座标短语 ”后接一个动词,再接一个

“点的名称”构成。因此,赋值语句的自动分析可以分别通过“点的名称”的自动分析,“动词”的自动分析,以及“座标短语”的自动分析来实现。

1·4·1·1 “点的名称”的自动分析

赋值语句中点的名称可以有下面4种形式:

p[①]

点 p

点 p 的座标

p 的座标

自动分析的结果,就是要把它们中的任何一种形式都翻译成内部文本的“point(p”[②]。

假定输入文本是以一个原子表的形式进入系统的,如:

〔点p 的座标为 123 ,456〕

表中每一个原子代表一个词,一个数字,或一个标点符号。计算机程序自左至右扫描输入文本,搜寻表的开头部分有没有点的名称。如果有,程序便得到三个结果:

(1) true(成功)

(2) 内部文本形式的〔point(p〕(即用作图语言表达的赋值语句的一部分)

(3) 输入文本的其余部分

① 这里假定“p”是用户定义的名称。

② “pointlp”是内部文本赋值语句的前半个部分。

如果表的开头部分没有找到点的名称，程序便得到另外三个结果：

(1)false(失败)

(2)nil(零)

(3)输入文本

下面是程序执行过程的实例。

例一：

输入文本= 〔点p 的座标为 123 , 456〕

结果 = true

〔point(P〕

〔为123 ,456〕

例二：

输入文本 = 〔123 , 456 是点 P〕

结果 = false

nil

〔123 , 456 是点 P〕

注意：上述程序是专为寻找、翻译出现在句首的‘点的名称’而设计的。一旦找到便把它抽取出来，并且翻译成内部文本的形式。随即进入下一步处理过程，即寻找、翻译后一半输入文本，构建出内部文本形式的〔123, 456〕。最后，将所得的两个内部文本形式的子表连结成一个内部文本形式的完整的赋值语句，即〔point(p, 123, 456)〕.

如果‘点的名称’不出现在句首，而是出现在句末(象例二的输入文本那样)，那么，上述程序便失败，送还原先的输入文

本,并等候调用专门处理‘点的名称’出现在句末的程序。

下面我们引进一个能够覆盖输入文本中各种不同形式的“点的名称”的自动分析手段——“有限状态图”。

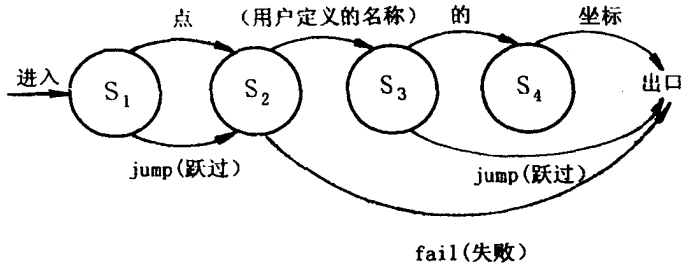

图 1·2 “点的名称”的有限状态图

图 1.2 是一个由 4 个状态(S_1, S_2, S_3 和 S_4)组成的有限状态图。每一状态代表分析过程中的一个期望状态；整个状态图则代表对输入文本中“点的名称”的全部可能的分析过程。分析过程跟输入文本自左至右的扫描同步进行。当分析开始时,过程处于 S_1 状态,这时,它期望找到‘点’这个词。一旦找到了,过程就由 S_1 转移到 S_2。接着，过程又期望找到一个由用户定义的名称。一旦找到，就又由 S_2 转移到 S_3。接着，又期望顺序找到‘的’‘坐标’这两个词。一旦找到,过程便从 S_3 转移到 S_4,并经由 S_4 出口，同时留下前面指出过的分析结果。

如果输入文本的‘点的名称’中没有‘点’字出现,或没有“的”“坐标”两个词出现,那么分析过程仍然可以经过 S_2、S_3 出口。当然，实际经由的路径有所不同，例如‘p的坐标’的分

析路径为：$\xrightarrow{\text{进入}} S_1 \xrightarrow[\text{jump}]{} S_2 \xrightarrow[\text{“P”}]{} S_3 \xrightarrow[\text{的}]{} S_4 \xrightarrow[\text{座标}]{}$出口。而‘p 的路径为：$\xrightarrow{\text{进入}} S_1 \xrightarrow[\text{jump}]{} S_2 \xrightarrow[\text{“P”}]{} S_3 \xrightarrow[\text{jump}]{}$出口。过程也可经由 S_2 出口，不过那样的话，就表明该分析过程的失败，即没有找到点的名称。注意：图 1.2 中带‘jump(跃过)’标记的路径可以不经过跟输入词语的匹配，直接从一个状态跃入另一个状态。带 fail(失败)”标记的路径则往往会引起回溯或重新开始分析。

现在，我们能轻而易举地把代表“点的名称”的分析过程的有限状态图翻译为用 Prolog 程序语言编写的程序了。Prolog 是近年来被学术界公认为最适合用于人工智能方面的计算机程序语言。我们这里采用的是 Borland 国际有限公司推出的已经由南京大学计算机系经过汉化处理的 Turbo－Prolog 程序语言。Prolog 程序语言的最大特点是把整个解题过程看作一个定理的推导和证明的过程。程序员只要给出解题所需的规则、事实和目标，Prolog 内部固有的合一机制便会自动地依据这些规则和事实来搜寻所追求的目标。这跟一般的程序语言截然不同。一般的程序语言要求把解题过程中每一步骤都分解出来，编成程序，因而被称为‘过程语言’。与此对立的 Prolog 程序语言便被称为‘非过程语言’。

关于 Turbo－Prolog 的较为详细的介绍，请参看南京大学计算机系编写的《Turbo－Prolog 程序语言》(油印本)。

为了便于理解，我们打算给本书中出现的每一段具体的程序都安上一个简要的注释。按照 Turbo－Prolog 的规定，

这些注释都放在斜杠、星号'／*"和星号、斜杠'*／'之间。

```
ptname([WLR],[point(|W],R) - isname(W)
                             not(de(R,R1))。
/*"〈点的名称〉"定义之一。这个定义专门用来          */
/* 分析单独由'〈用户定义的名称〉'构成的            */
/*"〈点的名称〉"的。表头中第三个参量'R'=          */
/* 输入文本中去掉'〈点的名称〉'之后所余下的        */
/*部分。中间那个参量[point(|W]则是所生成的        */
/* 〈点的名称〉的内部文本。"not"是 Prolog 程       */
/*序语言的内部谓词,相当于'逻辑非"它前面          */
/* 的逗号(,),则相当于'逻辑并"。                   */

ptname([W|R],[point(|W],R2) - isname(W),
                              de(R,R1),
                              zuo-biao(R1,R2)。
/*"〈点的名称〉"定义之二。这个定义专门用          */
/*来分析由〈用户定义的名称〉"后接'的"座           */
/*标"构成的'〈点的名称〉"的。                     */

ptname(["点"|T],P,R) - ptname(T,P,R)。
/*"〈点的名称〉"定义之三。这个定义可以用来        */
/*分析由'点"后接'〈用户定义的名称〉"构成的        */
/*〈点的名称〉,也可用来分析由'点"后接             */
```

```
/*"(用户定义的名称)",再接'的'"座标"构成                    */
/*的'(点的名称)"注意这是一个递归定义。                      */

isname(W) :- ismemb(W,[a,b,c,d,e,f,p,q,r]).
/*"(用户定义的名称)"的定义。用户定义的名                    */
/*称可以不止上面列举的"a,b,c"等9个。这                      */
/*里只列9个，是为了说明的方便。                              */

ismemb(X,[X|_]).
ismemb(X,[Y|L]) :- ismemb(X,L).
/*这也是一个递归定义：如果一个变量是某个                    */
/*表中的成员,那么,要么它是该表中的'头",                     */
/*要么它在该表的尾部。前一种情况由第一个                    */
/*"ismemb"Prolog子句来表示，后一种情况由第二                */
/*个"ismemb"Prolog子句来表示。"ismemb"其实是                */
/*Prolog的内部谓词，不须用户自己定义。                      */

de([W1|R1],R1) :- W1='的'.
/*当输入文本中紧接在(用户定义的名称)之                      */
/*后的是'的'这个词的时候,该子句为真。                       */

zuo-biao([W2|R2],R2) :- W2='座标'.
/*当'的'字之后是'座标'这个词的时候，                        */
/*该子句为真。                                              */
```

程序 1.1“点的名称”的 Turbo－Prolog 程序

1·4·1·2 “动词短语”的自动分析

输入文本中动词短语采取两种形式：

(1)…… 为……

(2)…… 是……

这两种形式的选择，完全是语境自由、不受任何限制的。不管哪一种形式，翻译成内部文本时都要被省略。

动词短语的分析可以用图 1.3 中的有限状态图来表示：

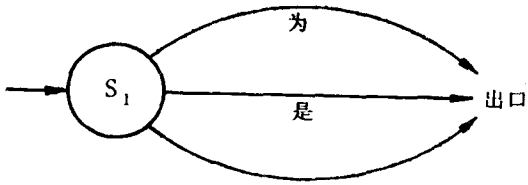

图 1·3 “动词短语”的有限状态图之一

下面是实现上述分析的 Prolog 程序：

```
verb－ph([W|R],R) :- W=“为”。
/*“〈动词短语〉”定义之一。此时,动词短语由          */
/*“为”来担任。                                    */

verb－ph([W|R],R) :- W=“是”。
/*“〈动词短语〉”定义之二。此时,动词短语由          */
/*“是”来担任。                                    */
```

程序 1.2“动词短语”的 Turbo－Prolog 程序之一

动词短语的分析也可以用图 1.4 中的有限状态图来表示：

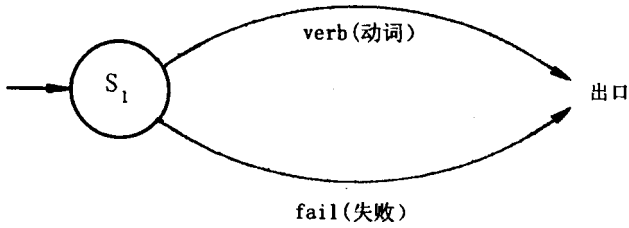

图 1·4 “动词短语”的有限状态图之二

相应的 Prolog 程序为：

```
verb－ph([W|R],R) - isverb(W).
/*“〈动词短语〉”的定义                                    */

isverb(W) - ismemb(W, ["为";是"])
/*“动词”的定义。                                         */
```

程序 1.3“动词短语”的 Turbo－Prolog 程序之二

1·4·1·3 词以及词的分类概念

动词短语的上述两种不同的分析和处理方法，看上去差异不大，其实包含着观念上的重大区别。前者是以词为出发点，用词来定义句子。其结果必然是：只要组成句子的词一有变化，就得重新给句子下定义。因此，上述赋值语句中的动词

部分用‘是’不用‘为’;在第一种处理方式下,动词部分的定义乃至整个语句的定义就必须作出相应的调整。后者是以词的分类为出发点,用词类来定义句子。这样,组成句子的词可以由同一词类中的其他词来替换而不影响句子的定义。因此,在上述第三种处理方式下,不管赋值语句的动词部分用哪一个动词,动词部分或整个句子的定义都不需变动。

某一语言中所包含的具体句子的数量是无限多的。从词出发,就无法对这无限多的具体句子进行概括。从词类出发,才可能把无限多的句子归纳为数量上有限的句型。例如下面这些句子:

他撒网。
我捕鱼。
他听收音机。
我看电视机。
等等

都可以概括为

〈代词〉〈动词〉〈名词〉

这样一个句型。这里的‘代词’‘动词’‘名词’等,便是词的分类概念。

传统语法常常自觉或不自觉地倾向于把词的分类看作是词的意义范畴。例如,把动词看作是表示行为、动作、变化等概念的词,而把名词看作是代表事物概念的词,等等。其实,词的分类虽然跟意义范畴有些瓜葛,但绝不是以意义范畴为标准。这一点无论是在汉语中还是在其他语言中都是如此。

例如汉语的两个词“战争”和“打仗”,从意义范畴看,没有什么不同,但从词类看,却分属“名词”和“动词”两大类。这是因为这两个词跟其他词语之间的组合能力不同:“战争”前面可以出现数量词(“一场战争”),但不能出现否定副词(“不战争”);“打仗”前面不能出现数量词(“一场打仗”),但可以出现否定副词(“不打仗”)。又如英语的两个词“assign”“assignment”,都是“分配”的意思。然而“assign”可以充当谓语,有时态变化,后面可以带一个名词性成分作直接宾语和一个介词结构作间接宾语,如“assign two rooms to somebody”;“assignment”则不能充当谓语,没有时态变化,也不可以带宾语,但能在其前加上冠词或所有格代词,能充当主语、宾语或表语等等,如“receive an assignment”。

由此可见,词的分类只跟词的语法功能有关,即同词和词的组合能力有关,或跟词在句中出现的位置有关。利用这一点,可以有效地控制句子或短语的分析。例如,一个名词后面紧跟着一个动词,往往组成一个主谓结构的短语,或一个句子。

词类的概念推广开来,就可以得到“形式类”的概念,即把短语的外部功能类别也包括进来了。如果一个短语的外部组合能力相当于一个名词,就叫做“名词性短语”,用“NP”表示;相当于一个动词,就叫做“动词性短语”,用“VP”,表示。运用形式类的概念,可以对句型或短语结构作更进一步的归纳。例如下列句子:

王冕的父亲死了。