DB—920006
总 0268

中国电子科技报告

逻辑型语言编译系统——LPL 系统研究报告

机械电子工业部 电子科技情报研究所

中国电子科技报告

编辑出版：机电部电子科技情报所
发　　行：北京 750 信箱 21 分箱
出版日期：1993 年 4 月
定　　价：5 元

J-92006(总 0099)

报告题名	逻辑型语言编译系统——LPL 系统研究报告		
完成单位	北京信息工程学院计算机系		
作者姓名 (职务或职称)	张海藩(教授)		
获奖情况	1991 年机电部科技进步三等奖		
报告密级		分 类 号	TP314
报告完成日期	1992 年 05 月	出版日期	1993 年 04 月
部门编号	军 9115012	总 页 数	15
审查批准人 (职务或职称)	甘圣宁(院长,教授)		
叙　　词	编译系统,PROLOG 语言		

摘要:LPL 语言是在 PROLOG 语言中增加函数计算功能和启发式搜索算法而设计出来的逻辑型程序设计语言,它兼容 DEC-10PROLOG,然而功能更强,解题效率更高。

LPL 语言的特点是决定了其编译程序有许多不同于常规编译技术的特殊处理,如变量分析,同名子句分段索引等。此外,本系统还包括一个比较庞大的设计巧妙的运行子系统,以实现目标代码运行时的人机交互,完成动态存贮管理,实现内部谓词和内部函数等。

LPL 系统已经商品化(商品名为 BITI PROLOG+编译系统)。它为用户提供了一个集成化的逻辑程序开发环境。

一、概　述

“逻辑型语言编译系统——LPL 系统”是 1986 年 10 月国防科工委和机电部下达给北京信息工程学院的“七五”预先研究项目，为“面向逻辑型语言处理机的研究”的阶段性成果。

LPL 语言是在 PROLOG 语言基础上，吸收函数型语言的基本特点，并以内部谓词形式增加启发式搜索策略，而设计出的一种功能更强，效率更高，使用更方便的逻辑型语言。

LPL 系统是 LPL 语言的纯编译系统。该系统在我国主流微机 0500 系列微型计算机上运行，在CCDOS 操作系统环境下充分支持汉字功能。该系统使用方便，操作简单，熟悉 PROLOG 语言的人不需专门的学习过程，即可操作。

LPL 系统于 1990 年 12 月获机电部教育司科技进步奖。1991 年 12 月获机电部科技进步三等奖。该系统已经实现了商品化，商品软件命名为“BITI　PROLOG＋编译系统”，它的用户界面友好灵活，带有一个全屏幕汉字编辑器，可使编辑、编译、调试、链接和运行等操作在一个集成环境中方便地完成。

LPL 系统的主要特点如下：

1、LPL 语言兼容 DEC-10　PROLOG 语言，便于利用大量现有的 PROLOG 程序。

2、增加了函数计算功能，用户程序中可定义和使用函数，克服了 PROLOG 语言表达算法类知识能力弱的缺点。

3、除 PROLOG 原有的深度优先搜索策略外，还以内部谓词形式提供了用户可方便选用的常用启发式搜索算法，以利用启发式知识求解问题，从根本上提高了解题效率。

4、在 CCDOS/GWDOS 支持下，具有很强的汉字功能，谓词名、函数名、参数、字符串、注解及窗口均可使用汉字，方便了国内用户。

5、运行速度快，目标代码短，运行时占用空间少。总观其运行效率，与当前在微机上流行的 Turbo Prolog 或 Arity Prolog 不相上下(参见表 1)。

6、编译生成的可执行文件在运行时仍然具有很强的人机对话功能，可针对源程序中的任何谓词或函数提问，也就是说，本系统以编译系统的速度和解释系统的交互方式工作，兼有两类系统的优点。

7、内部谓词相当丰富，目前已有 160 多个内部谓词。

8、以内部谓词形式提供了 IF-THEN-ELSE 型分支和 REPEAT-UNTIL 型循环等过程性控制结构，使得 LPL 语言在一定程度上兼有描述性语言和过程性语言两者的优点。

表 1　国内外有代表性的 PROLOG 系统性能比较

主要性能 \ 系统名称		Turbo Prolog V 1.1（美国 Borland 公司生产）	Arity Prolog V 5.0（美国 Arity 公司生产）	BITI PROLOG+ V 1.0（即商品化的 LPL 系统）
中文处理能力		只能在字符串中使用汉字	不能使用汉字	谓词名、函数名、参数、窗口内等均可用汉字
函数功能		无	无	有
目标代码大小		大	很大	小
解八皇后问题所需时间		5.1 秒	17.2 秒	13.0 秒
对 n 个元素的逆序表快速排序所需时间	n=126	1.8 秒	3.0 秒	3.3 秒
	n=1000	当 n>383 时 Heap 栈溢出	10 分 30 秒	5 分 15 秒
	n=2000		3 小时 22 分	21 分 2 秒
	n=4000		全局栈溢出	1 小时 23 分
解九宫问题所需时间	用内部谓词 astar 进行启发式搜索所需时间	无此内部谓词	无此内部谓词	1.15 秒
	用插入、删除等内部谓词实现启发搜索所需时间	2.0 秒	1.45 秒	1.32 秒

注：当 n>126 时，本系统进行快速排序时需要使用废料回收技术。

二、总体设计原则

LPL 系统是一个系统软件，它要为各种用户服务。从一开始研制，我们就坚持以软件工程方法学来指导和管理整个系统的开发。在总体设计过程中，着重考虑了下述几个方面的问题：

1、设计一个好的逻辑型语言，是设计面向逻辑型语言系统的第一步，也是非常重要的一步。我们对 LPL 语言的设计原则是，既不要脱离当前最流行的逻辑型语言 PROLOG 另搞一套，又不要受 PROLOG 语言的束缚。应该在 PROLOG 语言的基础上加以改进和扩充，在克服 PROLOG 语言缺点的同时，又向下兼容 PROLOG 语言，以便现有的大量 PROLOG 语言程序可以不经修改就在 LPL 系统中运行。

2、作为一个系统软件必须十分重视它所服务的对象，我们的原则是使 LPL 系统为最广大的用户服务。为此，选择了国内主流微机机型 0500 系列微型计算机，作为 LPL 系统的硬件环境，并在 CCDOS 操作系统支持下，使 LPL 系统具有很强的汉字功能，以方便广大的国内用户。

3、系统的可靠性，可维护性，实用性，运行效率，以及使用的方便程度等等，是决定一个系统软件是否有生命力的关键因素。我们的设计原则是：强调软件模块化和模块独立性，以提高系统的可靠性和可维护性；精心设计 LPL 语言，以满足各方面用户的需要，保证系统的实用性；扩充搜索策略，改进实现算法，以提高系统效率；简化使用方法，建立集成环境，充分支持汉字功能，以方便用户。

LPL 语言与传统的过程性语言不同，它是一种描述性语言。用户的源程序只需描述与问题有关的事实和规则，定义必要的函数，而不必规定计算机求解问题的一系列步骤。因此，LPL 编译程序不同于一般的编译程序，它必须根据源程序中已有的信息进行大量的分析和处理工作，才能把它们转换成一系列简单、确定的操作。由于这个原因，诸如语义处理、优化、存储分配和回收，以及运行机制的控制等等，都不能使用常规的编译技术实现。

为了解决上述这些编译实现中的特殊问题，必须特别注意中间语言的设计和编译过程中的预处理工作，并且需要一个运行子系统来支持和控制目标代码的运行过程，以及在各种情况下进行相应的动态存储分配。因此，LPL 系统由编译子系统和运行子系统这样两个相对独立的子系统组成。编译子系统把用户用 LPL 语言书写的源程序翻译成用 IBM-PC 宏汇编语言书写的目标代码；运行子系统接受用户的提问，控制目标代码的运行过程以求解用户的问题，显示对用户提问的回答。下面三节分别讲述上述几个方面的问题。

三、中间语言的设计

LPL 系统的编译对象是 Horn 子句和递归方程式形式的函数定义子句，必须根据其语义将它们翻译成一系列的推理计算过程。一般说来，把源程序直接翻译成汇编级或更低级的目标代码是困难的，也不利于优化处理。中间语言的形式应该既方便于编译又能有效地实现源语言的运行机制。

著名的 Warren 抽象指令集，是近年来人们所推崇的 PROLOG 编译程序的中间语言，它是 D. H. D. Warren 博士于 1983 年提出的一个非常高效的 PROLOG 执行模型，并被 P. Kursawe 从理论上证明了其正确性和高效性。Warren 指令集巧妙适当地设置了指令和数据空间，从而合乎逻辑地实现了 Horn 子句的执行过程。其数据区的分配策略和指令集的有机结合，组成了一个完整的按深度优先策略遍历搜索树的过程。

但是，Warren 指令集只是纯逻辑成分的执行模型，它对非逻辑成分的内部谓词并没有提供支持，也不适合表达算术运算。因此，我们对 Warren 指令集作适当扩充后作为 Horn 子句部分的中间语言，对函数定义和表达式部分则选用四元式形式作中间语言。

选择 Warren 指令集作为中间语言的另一个重要理由是，该指令系统具有层次较低，结构简单紧凑的特点。因此，不仅适于用软件实现，也容易用硬件实现，可以做成 Warren 指令集的硬件处理器。

LPL 系统用于逻辑成分的中间语言指令共有六类约 78 条，下面列出它们的分类情况：

1、get 类指令　用于将子句头的参数和调用目标的变元进行匹配。

例如，get __ list Ai　表示将子句头的类型为表的参数与在参数寄存器 Ai 中的目标变元相匹配。

2、put 类指令　用于将子句体中子目标的参数放入参数寄存器中。

例如，put __ value Xn，Ai　表示将子目标中临时变量 Xn 的值放入参数寄存器 Ai 中。

3、unify 类指令　这类指令对应于结构或表中的参数的合一操作，包括对已存在的结构或表的参数进行合一操作（在‘读’模式下），以及构造新的结构或表（在‘写’模式下）两类合一操作。

例如，unify __ nil　表示把空表放入全局栈中或用空表与全局栈中相应参数合一。

4、过程控制类指令　这类指令用于过程的调用以及环境栈的开辟和回收。

5、选择指令　这类指令用于推理时对子句的选择并控制选择点栈的开辟、更新和回收。

6、索引(开关)指令　这类指令的作用是有选择地执行过程子句，以避免执行肯定不能与调用目标匹配的子句。

四、编译子系统的设计

4.1　总体结构

编译子系统的功能是对用户的源程序进行语法分析，并根据其语义生成目标代码和初始数据，其总体结构如图1所示。

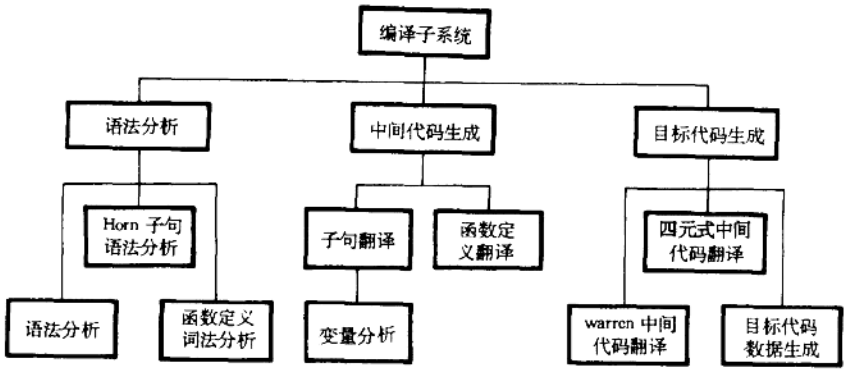

图1　编译子系统的总体结构

(1)词法分析模块

此模块的功能是对源程序进行词法分析以生成内部单词编码。源程序中注释部分的处理，汉字的接收和处理等也都由本模块完成。

为了提高编译速度，词法分析不作为独立的一遍扫描过程。本模块仅被语法分析模块调用。本模块采用直接分析法实现。

对名字或字符串的长度过分限制不便于用户使用，因此，LPL 语言在这方面没有限制。这样在词法分析过程中必须将谓词名、函数名、变量名、字符串等分别登记在几张不定长表中，编译过程中使用它们的内部值，最后由目标代码生成模块将这些不定长表转换成目标代码的初始数据。

(2)语法分析模块

采用自顶向下分析法，通过若干个递归下降子程序对源程序进行语法分析。这些子程序的主要功能是：

①接受文法结构正确的程序。

②发现一切非法的语法结构，并给出出错信息，指示出错位置。

③为生成中间代码作必要的预处理，进行各种统计、分类、登记生成 atom 表、clause 表、funatom 表和 funclause 表，以及 Lcatom 表。

atom 表存放源程序中包含的各个元素的信息，主要有元素的单词内部码，元素的单词类型，以及一个指向变量表(当元素为变量时)或 Lcatom 表(当元素为表或结构时)的指针。

atom 表和 Lcatom 表将源程序中 Horn 子句的基本信息登记下来，剔除了程序中的固定分

隔符，如“：—”、“，”、“(”和“)”等，并加上了为生成中间代码所需要的子目标或子句结束标记。

clause 表登记源程序中包含的各个 Horn 子句的信息，主要有子句或子目标第一个元素在 atom 表的位置，下一条子句或子目标在 clause 表的位置，子句或子目标的类型，参数数目和永久变量的个数等。clause 表中的这些信息是生成中间代码的重要依据。

函数定义子句与 Horn 子句的文法表示以及对应的中间代码形式均有较大差异。因此，将函数定义子句的信息填在 funatom 表和 funclause 表中，它们的内容与 atom 表和 clause 表的内容相似。

为了减少开销，提高编译速度，LPL 系统的出错处理采用如下策略：一旦发现源程序中的语法错误，立即显示有错误的那行源程序，指示出错位置，给出出错信息，并停止编译程序的运行。

(3)中间代码生成模块

常规的过程性高级语言编译程序生成中间代码的方法是，根据每一条源语句的语义直接翻译成完成相应操作的中间代码序列。LPL 语言虽然语法结构简单，用户使用方便，但同样形式的子句或子目标由于所处位置不同，它们所对应的中间代码却不尽相同。例如，同样的子句出现的位置不同要使用不同的选择指令，同样的子目标所处位置不同对应的调用代码也不相同，而程序中同样的变量名由于出现的次数、位置不同却可能有五种不同的中间代码与之对应。因此，不能使用通常的语法制导翻译法或直接翻译法生成 LPL 源程序的中间代码，需要对每个子句、子句的项、子目标等“瞻前顾后”才能翻译。在一条子句，一个子目标，一个项翻译成中间代码之前，必须前后扫描有关的信息，加上各种标志。为了减少扫描次数提高编译效率，这类工作大部分在语法分析阶段完成，得到的结果填在 atom 表和 clause 表中。变量使用情况的分析由变量分析模块完成，此模块对各子句中变量分布情况进行分析，对变量加标记和编号。此外，还需要根据各子句头的第一个参数的情况，进行分段和索引。

中间代码生成就是根据 atom 表、clause 表和 vartable 表的有关信息“综合考虑”，生成与源程序等价的中间代码序列。

中间代码生成模块调用 Horn 子句翻译模块和函数定义子句翻译模块。Horn 子句翻译模块根据语法分析模块产生的 atom 表，clause 表及 vartable 表生成 Warren 指令形式的中间代码序列。函数定义子句翻译模块，根据语法分析模块产生的 funatom 表和 funclause 表，生成四元式形式的中间代码序列。

①Horn 子句翻译模块

Horn 子句的翻译原则是，同名子句组成一个中间代码过程。在语法分析若干条连续排列的同名子句后调用此模块，生成一个 Warren 指令形式的中间代码过程，子句中的项对应一条或一段中间代码，子句间或子目标间的调用指令和控制指令则需根据前后有关信息才能确定。Horn 子句中非逻辑成分的内部谓词的翻译方法是根据每个内部谓词的功能翻译成相应的中间代码。

本模块调用如下几个模块：子句头翻译模块，子目标翻译模块，内部谓词翻译模块，表翻译模块，结构翻译模块等。

中间代码的优化处理也是本模块的一项重要工作。根据变量分析表的优化标志，生成中间代码时分不同情况分别生成可优化的代码序列或不可优化的代码序列。这样的优化处理策略可减少扫描次数和程序开销。

②函数定义子句翻译模块

函数定义子句的翻译原则是，同名函数定义语句翻译成的中间代码组成一个函数过程。语法分析一组同名函数定义子句后，调用函数定义子句翻译模块，生成一个四元形式的中间代码过程。

每条函数定义子句的翻译规则是，先将它的约束条件（由它的参数及子句后面的条件表示）翻译成条件转移中间代码，再处理它的定义表达式。

函数定义子句翻译模块采用算符优先分析法生成中间代码。

(4)目标代码生成模块

此模块将 Warren 指令形式或四元式形式的中间代码转换成 8088 宏汇编代码。由于两种中间代码形式不同，故分成两个模块分别完成目标代码生成工作：抽象指令翻译模块将每一条 Warren 指令翻译成相应的汇编代码；四元式代码翻译模块将每一条四元式翻译成相应的汇编代码。在翻译过程中对目标代码作适当的优化处理。

此模块还将几张不定长表，谓词名表（登记与谓词对应的过程编号），函数名表（登记与函数定义对应的过程编号），结构名表等转换成目标代码的初始数据。

4.2 变量分析

子句中的变量分为临时变量和永久变量两大类。

所谓临时变量是仅出现在一个子目标（子句头作为子句体中第一个子目标的一部分看待）中的变量，这类变量在运行时保存在数据区 PDL 中。

永久变量是指除了临时变量以外的变量，这类变量需保存在环境栈中。有一种特殊的永久变量称为不安全变量，这类变量不是首次出现在子句头或结构中，而是首次出现在子句的子目标中，是由 Put-variable 指令初始化的。

变量的作用域是它所在的子句。因此，目标代码中用 $y_i(i=1,2\cdots)$ 表示永久变量，用 $x_i(i=1,2\cdots)$ 表示临时变量。在每一条 put、get 或 unify 指令中对变量的操作还必须标以 variable、value 或 unsafe-value 等标记。其中 variable 标记表示该变量是第一次在子句中出现，将对它作初始化操作；value 标记表示该变量已不是第一次出现，已经在约束链中了；unsafe-value 标记表示不安全变量的最后一次出现，为避免产生悬空指针，可能需要把此变量移至全局栈中。

临时变量的编号方法，是根据它们在子目标中第一次出现时的位置从左到右顺序编排。对永久变量的编号方法与此相反，是按照各个变量最后一次出现时的位置从右到左顺序编排。永久变量的这种特殊的编号方法，是为了便于及时回收，不再使永久变量在环境栈中占用空间。

变量分析模块完成变量分类、标记和编号等工作。变量分析过程需要对子句中的变量正、反向扫描多次，因此设置了变量表和子目标变量情况表。

变量表的内容有：变量名，子目标号，参数号，高链，低链，变量编号，指令号。

子目标变量情况表有两个域：子目标第一个变量在变量表的位置和子目标在 clause 表的位置。具体说来，变量分析模块完成如下几项工作：

(1)根据 atom 表和 Lcatom 表中的信息，产生变量表和子目标变量情况表中的部分信息。

(2)对没有永久变量的子句进行临时变量的编号和指令编号。

(3)对有两个以上子目标的子句作如下处理：

①对每个变量建立同名变量的高、低链，并登记此变量第一、第二及最后一次出现的子目标号 a_1、a_2 和 a_3。

②由每个变量的 a_1、a_2、a_3 的情况判断它是永久变量还是临时变量，并加上指令标记。

③从子目标变量情况表的最后一个子目标开始，反向扫描各个子目标的变量，为每个永久

变量编号，并将各子目标永久变量最大序号回填到 clause 表中。

此外，变量分析模块还完成一些优化预处理工作，具体说有下述两个方面：

①对于像 p(X,Y,X):-…这样的子句，在子句头中有两个同名变量作为参数，而且这个变量不在其它子目标或表和结构中出现，那么可将原来的二条指令 get-var Y_1、A_1 和 get-value Y_1、A_3 优化为一条指令 get-value A_1、A_3。变量分析模块识别出这种可优化的情况，并在变量表的相应位置加上标记。

②对于子目标有下列情况：

…,X is Y+1,q(X,b),……

即，变量 X 仅临时用来存放 Y+1 的值，并立即在下一个用户谓词目标中作为参数，那么中间代码翻译时可优化为直接将 Y+1 的值放入 X 在下个子目标对应的 A_1 中，而不必先将 Y+1 的值放到 X 中，再将 X 的值放到 A_1 中。变量分析模块也必须对这种可优化的情况加以标记，以便中间代码生成时特殊处理。

4.3 同名子句分段索引

LPL 系统缺省的搜索策略，是深度优先遍历搜索树。为了提高搜索速度，我们利用编译技术对源程序进行了一些静态处理，以产生高效代码。主要进行了下述几方面的工作。

(1)寻找与调用目标名字及参数数目相同过程的工作，大部分在编译阶段完成。编译程序将同名子句(参数数目亦相同)翻译成目标代码的一个过程，并为之指定一个过程号。程序中所有对此子目标的调用，均使用该过程编号直接转向。为了使运行子系统，在接受用户提问的目标后，能准确地转向与之匹配的过程，程序中所有子句的名字，参数数目及对应的过程编号应该作为目标代码的初始数据由编译子系统生成，为此设置了一个谓词名表。

(2)LPL 系统深度优先自上而下的搜索过程，是通过一套选择指令来实现。在含有两条以上子句的过程中，每条子句的代码之前都有下列三条选择指令中的一条：try-me-else L，retry-me-else L 和 trust-me-else-fail。其中：第一条指令，作为此过程第一条子句的目标代码的第一条指令；第三条指令作为此过程最后一条子句的目标代码的第一条指令；第二条指令用于此过程的中间子句。这几条指令通过选择点栈的设置，保证了程序动态运行时正向搜索和反向回溯时的正确走向。但是，它们的作用是保证按照程序中子句的自然书写顺序去一条条地试探。对于显然无法与调用目标匹配的子句也必须经过合一过程，失败回溯后才走向下一条子句。

为了提高搜索效率，LPL 系统使用了一些索引指令，以便有选择地挑选子句去试探。主要的索引指令是：

switch-on-term vlb,ilb,alb,slb,nllb,stlb,llb，其中，vlb～llb 为标号，分别代表子句头的第一个参数为变量、整数、原子名、字符串、空表、结构、表的子句的代码地址。该指令根据调用目标的第一个参数的类型，直接跳转到上述七个标号之一指定的代码处。若某一类型有多条子句，则再用一套 try，retry 和 trust 指令逐一试探，或利用 hash 表直接转移到可能与之匹配的子句。

为了生成上述的选择和索引指令，必须根据同名子句的第一个参数的类型将它们分成若干段，分段以参数类型为变量的子句为界，若干条连续排列的类型为常量的子句为一段，类型为变量的子句一条为一段。每段开头放一条适当的选择指令。如果某段中有常量，则段内再用索引指令，以加快搜索速度。

为了生成适当的选择和索引指令，在语法分析过程将一些辅助性的信息登记在一张段表中。段表的主要内容如下：段属性(第一段，中间段，最后段)，段标号属性，段标号，段的起、止子

句标号，本段是否有 switch 指令，段的各类常量分布信息等。中间代码生成程序根据段表的有关信息，一段一段地进行翻译，各段间生成必要的控制指令。

4.4 函数和表达式的处理

增加函数成分主要是为了提高 LPL 语言表达算法类知识的能力。函数以递归方程式形式定义：

@f(a_1,a_2,…,a_n)=〈算术表达式〉

或@f(a_1,a_2,…,a_n)=〈算术表达式〉if〈条件〉

其中，f(a_1,a_2,…,a_n)为 n 元函数，算术表达式中可以有加、减、乘、除等运算，还可以出现用户定义的函数或系统提供的内部函数，函数的参数也可以是函数。可见，这种定义方式与数学中的函数方程相似，而且允许出现高阶函数。

在 LPL 系统中对函数的使用主要是求值，因此对函数的处理采用比消解法更高效的归约法来求解。也就是说，由编译程序作如下的静态处理：将同名函数的若干个定义方程组成一个过程，在此过程中首先根据每个函数方程的限制条件，对函数的输入项（调用此函数时的输入参数），进行归约（比较判断），选择某一个定义方程，然后按此方程的算术表达式计算出函数值，返回一个确定的调用结果。

由于函数允许递归定义，因此对运算过程使用的数据空间采用动态栈式分配策略。用一个 funstack 作为函数运算和谓词中复杂表达式的数据空间，数据的值直接存放在这个栈中。当在谓词子目标中出现函数调用或复杂表达式时，就需要在 funstack 与 PDL 或环境栈之间交换数据。为此编译系统设置了两类传送指令：

load A_i,V_j 表示将选择点栈中 A_i 指定的值放入到 funstack 的 V_j 中；

set V_j,A_i 表示与上条指令传送数据的方向相反。

五、运行子系统的设计

LPL 是非过程性的交互式语言，并具有动态修改数据库的功能，它的编译实现存在一些不同于传统语言的特殊问题。为了解决这些问题，LPL 系统中包含一个相当庞大的运行子系统，以实现系统和用户间的交互、支持和控制目标代码的运行过程，实现各种内部谓词和内部函数，并完成各种情况下的动态存储分配和回收。

5.1 总体结构

运行子系统的总体结构如图 2 所示。

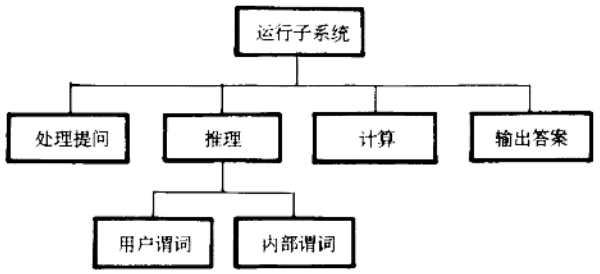

图 2 运行子系统的总体结构

(1)处理提问模块

本模块首先把用户键入的询问读到提问缓冲区中，然后从左至右逐个分析、处理缓冲区中的每个目标。先把一个目标内各参数的类型和值放入相应的参数寄存器，然后分析这个目标，若是谓词则据谓词名转到相应的目标代码过程进行推理，若是函数则转到相应的过程进行归约计算。

处理完一个目标以后，再重复上述过程，处理其右侧目标。当用户缓冲区为空时(所有目标都处理完了)，输出对用户询问的回答并等待接收下一个询问。如果某个目标不能被满足，则需回溯重新处理其左侧的目标。

(2)推理模块

本模块使用某种策略搜索知识库以满足一个目标。LPL 系统的缺省搜索策略与 PROLOG 相同，也是深度优先搜索法，推理过程是典型的线性归结过程。

为了从根本上提高解题效率，用户可以使用内部谓词以某种启发式搜索算法搜索知识库。启发式搜索算法由系统内部模块实现，对用户透明，用户只需给出算法所需要的基本信息即可。

(3)计算模块

本模块使用算符优先分析法和归约机制，完成函数或复杂表达式的计算。

(4)输出答案模块

本模块印出系统对用户提问的回答。如果用户提问中的参数都是常量，则目标可以满足时，回答“yes”；无法满足时回答“no”。如果提问中包含变量，则系统将自动回溯，输出所有可能的解。

5.2 数据结构

LPL 系统运行时的存储结构大致可分为如下几个部分：提问缓冲区、全局栈区(即 Heap)、环境栈区、选择点栈区、临时变量区(即 PDL)、踪迹栈区、截断栈区、函数栈区、OPEN 表区和动态数据库区。运行子系统的基本数据结构如图 3 所示。

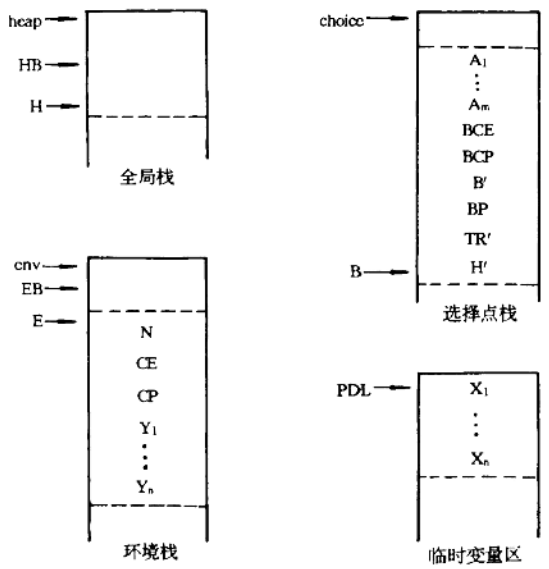

图 3 运行子系统的基本数据结构

在运行子系统中绝大多数数据元素都使用两个存储字来表示，包括类型和值两个域。

(1)全局栈

全局栈主要用来存放表和结构。

本系统对表和结构的处理采用非结构共享方式，即把表和结构中的每个元素都依次拷贝到全局栈的连续单元中。

在结构共享型系统中，当一个变量被约束于一个结构或表时，是通过产生一对指针来实现的，这一对指针和每个分量的偏置指针的使用，使得结构共享型系统所占用的全局栈空间往往超过非结构共享型系统所占用的全局栈空间。抽样统计结构表明，结构共享型系统占用的最大全局栈空间平均是非结构共享型系统占用的最大全局栈空间的 1.55 倍。此外，一个需要经常访问结构或表中的元素的程序，在非结构共享型系统中显然执行得比较快。因此，采用非结构共享方式有利于提高时空效率。

结构或表所占用的全局栈空间在回溯时进行回收。

(2)环境栈

环境栈主要用来保存子句中的永久变量。每当进入有多个子目标的子句时，系统即为该子句在环境栈中开辟一个环境(即分配一些存储单元)，保存永久变量个数 N，前一个环境的指针 CE，后续目标的地址 CP，以及每一个永久变量(Y_1～Y_N)。在执行确定性子句的最后一个子目标之前，清除该子句环境中的信息，从而实现了“尾递归优化”。所谓尾递归优化，更确切地说，应该称为“最后一次调用优化”。也就是说，在一定条件下执行最后一个子目标时，可以重新使用分配给其父目标的存储空间，从而使得递归程序可以和迭代程序一样在恒定空间内执行，而不是占用的空间随递归调用的次数线性增大。

能够进行尾递归优化的关键条件，就是从父目标消解到最后一个子目标的过程中没有选择点遗留下来，不可能回溯。Warren 建议把选择点和环境放在同一个栈内，这样在为新环境分配空间时只作一个简单的判断，就可实现尾递归优化。本系统把环境栈和选择点栈分开，为了也能实现尾递归优化，每当建立新的选择点栈时需要用指针 EB 记下当时环境栈顶的位置。将两个栈分开可增大每个栈的空间，可以按照每个栈的使用特点优化相应的栈区，并且易于实现下面谈到的对 Warren 原设计的改进。

(3)选择点栈

选择点栈保存在回溯时为了恢复先前的计算状态所需要的一切信息：调用该子句的参数(A_1～A_m)，下一个候选子句的地址(BP)，建立选择点时环境栈的指针(BCE)，前一个选择点的指针(B′)，下一个目标的地址(BCP)，踪迹栈的指针(TR′)和全局栈的指针(H′)。每当进入一个非确定性的过程时，系统即在选择点栈内创建一个选择点，在过程变为确定时删除这个选择点。

本系统对 Warren 设计所作的一个比较重要的改进，是把临时变量的参数分开存放，参数直接放入选择点栈中。按照 Warren 原设计，调用一个过程之前先用 put 指令把参数放入 PDL 中，若所调用的过程是非确定性的，在建立选择点时需把参数从 PDL 中拷贝到选择点中来，回溯时还需要再把参数从选择点中拷贝回 PDL。显然，这样反复传送参数是相当大的开销，为了提高效率，本系统的 put 指令直接把参数放到选择点栈中，如果所调用的过程是非确定性的，创建选择点时仅需再把其余信息放入即可；如果所调用的过程是确定性的，不需创建选择点，刚放入选择点栈中的参数在和子句头中的参数合一后即无需再保留了，它们所占用的空间可再用来存放调用下一个过程所需要的参数。为了实现上述改进，本系统的 get 指令直接从选择

点栈中取参数。当然，必须保持读参数时和写参数时所用指针的一致性。

(4)其它数据结构

踪迹栈中保存在合一过程中被约束的变量的地址，以便在回溯时用这些变量去约束。

截断栈是为实现内部谓词 cut 而设立的；OPEN 表是为实现内部谓词 astar 而设立的；动态数据库区是动态修改数据库的内部谓词使用的数据区；函数栈则是实现函数计算所使用的数据区。

5.3 内部谓词的实现

除了少数几个内部谓词可以由编译子系统译成扩充的 Warren 指令序列直接实现外，绝大多数内部谓词都使用运行子系统中相应模块来实现。下面介绍两类比较复杂的内部谓词的实现算法。

(1)实现启发式搜索算法的内部谓词

搜索技术是人工智能领域中使用十分广泛的机械化推理技术之一，特别是其中的启发式搜索原理(Heuristic Search Theory)，可以利用局部状态空间求解复杂问题，从而大大提高了解题效率。PROLOG 语言只有单一的搜索策略——深度优先搜索法。这种无信息的盲目搜索过程极其耗费时间，即使用编译系统代替解释系统，用并行处理代替顺序处理，在提高解题速度方面也只是治表不是治本。

几十年来人工智能研究的重大成就之一，就是利用启发式知识改进搜索过程，从根本上提高解复杂问题的效率。因此，本系统在保留深度优先搜索法作为缺省搜索策略的同时，增加了用户可方便地选用的或图最佳优先搜索法(A*算法)和 Alpha-Bata 剪枝法，以内部谓词形式提供。下面介绍 A*算法的实现。

A*算法用内部谓词 astar 实现，它的形式为：

```
astar(rulename,S0,T)
```

它含义是：若初始状态为 S0，则使用 A*算法得到的第一个满足终止条件的状态 T 就是最优解。状态用表来表示，规定表的第一个元素必须是该状态的评价函数值，表中其它元素的含义和形式同用户根据所求解的具体问题确定。内部谓词 astar 的第一个参数 rulename 是使用 A*算法所需要的产生式规则的名字。产生式规则由用户用 LPL 语言写出。其形式为：

```
rulename(X,Y)
```

它表示：将状态 X 展开，得到一个新状态 Y。规则中必须包含计算新状态 Y 的评价函数值的表达式。

内部谓词 astar 可以出现在用户谓词可以出现的任何地方，它的参数既可由用户直接给出，也可由合一消解过程传递过来，它的结果可供其它谓词继续求解使用。

为了实现内部谓词 astar，使用了专门的数据结构 OPEN 表，来保存用产生式规则得到的新状态。OPEN 表是一个单链表，每个结点包含两个域：指向下个结点的指针和表示状态的表在全局栈中的地址。

在 LPL 系统中，A*算法的搜索对象是一棵证明树。因此，其实现算法可以适当简化。若当前目标是 astar(rule,S0,T)，则本系统的求解过程相当于动态地生成并执行下列三条子句：

```
①astar(_,X,_):-rule(X,Y),enter(Y),fail.
②astar(_,_,T):-head(S),S=T,!.
③astar(_,_,_):-head(S),astar(_,S,T).
```

系统首先使用子句①。用当前目标第一个参数指定的产生式规则扩展出 S0 的第一个后续状态

Y，接着执行子目标 enter(Y)，把 Y 在全局栈中的地址存入 OPEN 表的最后一个结点中，并根据其评价函数值修改 OPEN 表的指针域，将此结点插到 OPEN 表的适当位置（评价函数值小的结点排在前面）。

内部谓词 fail 强迫回溯，扩展出 S0 的第二个后续状态并插到 OPEN 表中。如此重复下去，直到 S0 的所有后续状态都扩展出来并插入到 OPEN 表中之后，系统再执行 rule(X,Y)时失败，于是改用子句②去满足当前目标。

子句②的第一子目标 head(S)从全局栈中取出 OPEN 表第一个结点所对应的状态 S（即评价函数值最小的状态），然后将 S 与 T 合一，若合一成功已求出最优解，astar 运行结束；否则，进入子句③。把状态 S 作为新的初始状态，对新生成的目标 astar(_,S,T)重新使用上述三条子句求解。

(2)动态修改数据库的内部谓词

LPL 作为一种人工智能语言，自然应该提供某种知识获取手段。PROLOG 的内部谓词 asserta、assertz 和 retract 具有动态修改数据库的功能，可以说是最基本的知识获取机制。但是，在编译系统中实现这些内部谓词比在解释系统中实现要困难得多。本系统采用一个比较巧妙的策略实现了这些内部谓词，关键是把需要动态插入或删除的子句作为数据来处理，然而仍然使用扩充的 Warren 指令序列，来完成它们的合一操作，从而既简单又高效。

为了动态修改数据库，本系统设计了两个数据结构，它们是 DBI(Data Base Index)和 Data Base。

DBI 主要有三个域：

F　谓词名在名字表的下标

head　指向此谓词第一条子句在 Data Base 中的位置

tail　指向此谓词最后一条子句的位置

DataBase 的顶部指针为 DB_Top，主要有两个域：

next　指向同一谓词的下一条子句

PA　存放子句的所有参数

动态删除或插入的效果在任何情况下都不能因回溯而取消，因此，设置了专用数据区 DataBase，它的存储空间动态分配。为了和常规子句一样，也使用扩充的 Warren 指令序列，完成动态插入的子句的合一操作，在进行合一之前先将 DataBase 中的信息传送到全局栈中去。

当求解的当前目标是 asserta(X)或 assertz(X)时，实现的方法比较简单：首先在 DBI 中登记子句 X 的谓词名下标，然后把 X 的所有参数存放到 DataBase 的顶部，修改 next 和 head（或 tail）指针，将 X 相应地插在数据库的头或尾。

如果求解的当前目标是 retract(X)，则执行过程相当于动态生成，并执行如下两条子句：

①retract(X):－nextclause(Y),unify(X,Y),delete(Y).

②retract(X):－not(empty(X)),retract(X).

系统首先选用子句①，根据 head(或 next)指针到 DataBase 中，找出与 X 谓词名相同的下一条子句 Y，然后合一 X 和 Y 这两条子句，即对 X 和 Y 的相应参数分别合一，若合一成功则将子句 Y 从 DataBase 中删除；否则，子句①失败。系统选用子句②，首先判断 DataBase 中是否还有和 X 谓词名相同的子句，若已空则目标 retrat(X)失败，否则转向子句①，继续寻找可与 X 合一的子句。

对于动态数据库中的谓词，不仅可以插入和删除，也允许用户在源程序中直接引用它们。

如果求解的当前目标 X 是动态数据库中的子句，则系统的求解过程和求解目标 retract(X)时非常相似，唯一的不同之外在于，当动态数据库中某条子句与 X 合一成功时，并不将该条子句从 DataBase 中删掉。

5.4 函数计算功能的实现

LPL 语言中的函数以递归方程式定义，函数的参数还可以是函数，因此，对函数和复杂表达式的计算使用了一个专用的数据区——函数栈(funstack)。计算时需用的数据空间采用动态栈式分配策略。

使用函数栈中的数据进行计算时不需要分析处理数据类型，也不采用回溯机制，因此，其效率比使用谓词方式完成同样的计算时的效率高得多，而且使用起来非常方便。

参考文献

[1] D. H. D. Warren："An Abstract Prolog Instruction Set", Tech. Report 309, AI Center, SRI International, 1983

[2] 张海藩、林亨利等："逻辑型语言编译系统——LPL 系统"，《计算机技术》，1990. 1, pp. 27～31

[3] 张海藩、成渝等："逻辑型语言编译系统的运行子系统"，《计算机技术》，1990. 3, pp. 10～15

[4] 林亨利、张海藩等："逻辑语言编译系统的编译子系统"，《计算机技术》，1990. 4, pp. 43～49

[5] 张晨曦、慈云桂："Prolog 的编译实现"，《小型微型计算机系统》，1988 年，第 9 卷，第 1 期，pp. 14～19

[6] 郭剑昆等："PROLOG 编译系统的存贮结构与目标代码生成"，《小型微型计算机系统》，1988 年，第 9 卷，第 5 期，pp. 14～19