符号表达式的递归函数及其机器计算

约翰·麦卡锡，马赛诸塞州理工大学，马赛诸塞州，剑桥市.

1960年4月

1 介绍

M.I.T.的人工智能小组已为IBM 704计算机开发了一种名为LISP（用于列表处理）的编程系统。 该系统旨在方便使用被提议的称为“咨询建议者”(Advice Taker)的软件系统进行实验， 从而可以控制机器处理陈述性和命令性句子， 并且在其执行指令时可以表现出“常识”的特征。 给Advice Taker的原始提议[1]于1958年11月提出。 主要要求是一个程序系统， 用于操纵表示形式化的陈述性和命令性句子的表达式， 以便Advice Taker可以进行推理。

在其开发过程中，LISP系统经历了多个阶段的简化，最终基于的是表示特定类型的符号表达式的局部递归函数的方案。此表示独立于IBM 704计算机或任何其他电子计算机，现在可以很方便地通过从称为S-expressions的表达式类型和称为S-functions的函数类型来阐述系统。

在本文中， 我们首先描述一种用于递归定义函数的形式语言。 我们认为这种形式语言本身有作为编程语言的好处， 同时也是研究计算理论的载体。接下来，我们描述S表达式和S函数，给出一些示例，然后描述通用S函数的应用，它们在通用图灵机的理论作用和解释器的实际作用中发挥作用。然后，我们通过类似于Newell，Shaw和Simon [2]所使用的列表结构，来描述IBM 704内存中S表达式的表示，以及通过程序表示S函数。之后，我们介绍用于IBM 704的LISP编程系统的主要功能。接下来是用符号表达式描述计算的另一种方法，最后我们给出了流程图的递归函数解释。

我们希望在另一篇论文中描述已经使用LISP实现的一些符号计算程序，并希望在其他地方为数学逻辑和机械定理证明问题提供递归函数形式语言的一些应用。

2 函数和函数定义

我们将需要一些有关函数的数学思想和表示法。大多数想法是众所周知的，但是条件表达式的概念被认为是新的，条件表达式允许以更简洁的方式定义递归函数。

a，局部函数。一个局部函数是仅在其局部作用域中定义的函数。当函数由计算定义时，局部函数必定会出现，因为对于自变量的某些值，定义函数值的计算可能不会终止。因此，我们的一些基本函数将定义为局部函数。

b，命题表达式和谓词。命题表达式是其可能值为T（代表真）和F（代表虚假）的表达式。我们将假定读者熟悉命题连接词∧（and，与），∨（or，或）和¬（not，非）。典型的命题表达式是：

$$x < y$$
$$(x < y) \wedge (b = c)$$
$$x\text{是素数}$$

谓词是一个函数，其范围由真值T和F组成。

c，条件表达式。真值对其他种类的数量值的依赖关系在数学中通过谓词表示，而真值对其他真值的依赖关系则通过逻辑连接词表达。然而，用于象征性地表达其他种类的数量对真值的依赖性的符号是不充分的，因此英语单词和短语通常用于在象征性地描述其他依赖性的文本中表达这些依赖性。例如，函数| x | 通常用文字来定义。 条件表达式是一种表达量对命题量的依赖性的设备。

$$(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n)$$

其中p是命题表达式，e是任何形式的表达式。可以被读作“如果p_1则为e_1，如果为p_2则为e_2，...，否则，如果p_n则为e_n”，或“ p_1产生p_1，...， p_n产生e_n”。

现在我们给出确定

$$(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n)$$

的值的规则，如果条件为真，则确定它的值。从左到右检查p。如果在遇到值未定义的任何p之前遇到了值为T的p，则条件表达式的值就是相应e的值（如果已定义）。如果在真p之前遇到任何未定义的p，或者所有p均为假，或者与第一个真p对应的e未定义(e is undefined)，则条件表达式的值不确定。我们现在举一些例子。

$$(1 < 2 \rightarrow 4, 1 > 2 \rightarrow 3) = 4$$

$$(2 < 1 \rightarrow 4, 2 > 1 \rightarrow 3, 2 > 1 \rightarrow 2) = 3$$

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$\left(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3\right) = 3$$

$$\left(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}\right) = 3$$

$$\left(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}\right) \text{ is undefined}$$

$$(2 < 1 \rightarrow 3, 4 < 1 \rightarrow 4) \text{ is undefined}$$

条件表达式的一些最简单的应用是给出如下定义：

$$|x| = (x < 0 \to -x, T \to x)$$

$$\delta_{ij} = (i = j - 1, T \to 0)$$

$$sgn(x) = (x < 0 \to -1, x = 0 \to 0, T \to 1)$$

d，递归函数定义。通过使用条件表达式，我们可以在没有循环的情况下通过出现已定义函数的形式来定义函数。例如，我们这么写

$$n! = (n = 0 \to 1, T \to n \cdot (n-1)!)$$

当我们使用此公式求值 0! 时我们得到答案1；因为定义了条件表达式的求值方式，表达式$0 \cdot (0-1)$不会出现。求值 2! 时根据定义它会执行以下操作：

$$\begin{aligned} 2! &= (2 = 0 \to 1, T \to 2 \cdot (2-1)!) \\ &= 2 \cdot 1! \\ &= 2 \cdot (1 = 0 \to 1, T \to 1 \cdot (1-1)!) \\ &= 2 \cdot 1 \cdot 0! \\ &= 2 \cdot 1 \cdot (0 = 0 \to 1, T \to 0 \cdot (0-1)!) \\ &= 2 \cdot 1 \cdot 1 \\ &= 2 \end{aligned}$$

现在我们给出递归函数定义的另外两个应用。两个正整数m和n的最大公约数gcd（m，n）通过欧几里得算法计算。

$$gcd\,(m,n) = (m > n \to gcd(n,m), rem(n,m) = 0 \to m, T \to gcd(rem(n,m), m))$$

其中rem（n，m）表示当n除以m时剩下的余数。

牛顿迭代算法，用于获取数字a的近似平方根，该算法从初始近似值x开始，并要求满足$|y^2 - a| < \epsilon$可接受的近似值y，可以写成

$$sqrt(a,x,\epsilon) = \left(|x^2 - a| < \mathrm{e} \to x, T \to sqrt\left(a, \frac{1}{2}\left(x + \frac{a}{x}\right), \epsilon\right)\right)$$

多个函数的同时递归定义也是可能的，如果需要，我们将使用这些定义。

不能保证由递归定义确定的计算将永远终止，例如，尝试计算 n! 的时候，仅当n为非负整数时，才能成功。如果计算没有终止，则对于给定的参数，该函数必须视为未定义。

命题连接词本身可以由条件表达式定义。我们写为

$$\begin{aligned} p \wedge q &= (p \to q, T \to F) \\ p \wedge q &= (p \to T, T \to q) \\ \neg p &= (p \to F, T \to T) \\ p \supset q &= (p \to q, T \to T) \end{aligned}$$

显而易见，等式的右边具有正确的真值表。如果考虑p或q不确定的情况，则连接词∧和∨被视为不可交换的。例如，如果p为假，而q为未定义，则根据上面给出的定义，我们可以看到p ∨ q为假，而q ∧ p为未定义。对于我们的应用，这种不可交换性是可取的，因为p ∧ q是通过首先计算p来计算的，并且如果p为假，则不会计算q。如果p的计算没有终止，我们将永远不会绕开计算q。后面我们将在这层意义上使用命题连接词。

e，函数和形式。在数学中，除数学逻辑外，通常不精确地使用“函数”一词并将其应用于诸如$y^2 + x$的形式。因为稍后我们将使用函数的表达式进行计算，所以我们需要在函数和形式之间进行区分，并需要使用一种表示法来区分这种区别。说一句题外话，邱奇[3]给出了这种区别和描述它的记号。

令f为代表两个整数变量的函数的表达式。写为 f(3,4) 具有意义，并且该表达式的值应该确定。表达式 $y^2 + x$ 则不满足此要求；$y^2 + x$ (3, 4)不是传统的表示法，如果我们尝试对其进行定义，我们将不确定其值是否变为13或19。邱奇将 $y^2 + x$ 这样的表达式称为一个结构。如果我们可以确定结构中出现的变量与所需函数的参数的有序列表之间的对应关系，则可以将结构转换为函数。这是由邱奇的λ符号完成的。如果E是变量 $x_1, \cdots, x_n$ 的一个结构，则将$\lambda((x_1, \cdots, x_n), E)$视为n个变量的函数，其值通过将参数替换为n个变量 $x_1, \cdots, x_n$ 在E中按此顺序排列求值得来。

出现在λ表达式的变量列表中的变量是假设的或是有界的，就像微积分中的积分变量一样。那也就是说，我们更改函数表达式中绑定变量的名称而不更改表达式的值，只要我们对对应变量的每次出现都进行相同的更改并且不让两个变量的名字相同就可以了。因此，$\lambda((x,y), y^2 + x)$，$\lambda((u, v), v^2 + u)$ 和 $\lambda((y,x), x^2 + y)$ 表示相同的函数。

我们将经常使用某些变量受λ约束而另一些不受λ约束的表达式。这样的表达式可以被认为是用参数定义函数。未绑定的变量称为自由变量。

将函数与结构区分开的适当符号可以对函数的意义进行准确的处理。在这里举个例子可能会涉及很多题外话，但是在本报告的后面，我们将使用带有函数作为参数的函数。

组合由λ表达式或任何其他涉及变量的符号描述的函数时会遇到困难，因为不同的绑定变量可能用相同的符号表示。这称为绑定变量的冲突。有一种只涉及运算符的符号系统，称为组合子，用于在不使用变量的情况下组合函数。不幸的是，有意义的组合子的组合表达式往往冗长且难以理解。

f，递归函数的表达式。λ表示法不足以用递归定义函数。例如，使用λ，我们可以转换定义

$$sqrt(a,x,\epsilon) = \left(|x^2 - a| < \mathrm{e} \to x, T \to sqrt\left(a, \frac{1}{2}\left(x + \frac{a}{x}\right), \epsilon\right)\right)$$

为

$$sqrt = \lambda\left((a,x,\epsilon),\left(|x^2-a|<\mathrm{e}\rightarrow x, T\rightarrow sqrt\left(a,\frac{1}{2}\left(x+\frac{a}{x}\right),\epsilon\right)\right)\right)$$

但右侧不能用作该函数的表达式，因为没有任何迹象表明该表达式中对$sqrt$的引用代表整个表达式。

为了能够为递归函数编写表达式，我们引入了另一种表示法。$label(a,E)$ 表示表达式E，前提是在E内出现的a被解释为是指整个表达式，这样我们可以写下

$$label(sqrt, \lambda\left((a,x,\epsilon),\left(|x^2-a|<\mathrm{e}\rightarrow x, T\rightarrow sqrt\left(a,\frac{1}{2}\left(x+\frac{a}{x}\right),\epsilon\right)\right)\right))$$

作为我们sqrt函数的名称。

$label(a,E)$中的符号a也已绑定，也就是说，可以在不更改表达式含义的情况下进行系统修改。但是，它的行为不同于以λ为界的变量。

3 符号表达式的递归函数

我们将首先根据有序对和列表定义一类符号表达式。然后，我们将定义五个基本函数和谓词，并根据它们的构成，条件表达式和递归定义，利用它们构建广泛的函数类，我们将给出许多示例。然后，我们将展示如何将这些函数本身表达为符号表达式，并定义一个通用函数应用，该函数允许我们从表达式中计算出给定函数的值，以得出给定参数。最后，我们将定义一些以函数为参数的函数，并给出一些有用的示例。

a，一类符号表达式。现在我们将定义S表达式（S代表符号 $symbol$ ）。它们是通过使用特殊字符形成的

.
)
(

以及无限组可区分的原子符号。对于原子符号，我们将使用大写拉丁字母和数字字符串以及单个嵌入空格。原子符号可以写作

$$A$$
$$ABA$$
$$APPLE \quad PIE \quad NUMBER \quad 3$$

有两个原因令人放弃了将单个字母用于原子符号的通常数学实践。首先，计算机程序经常需要数百个可区分的符号，这些符号必须由IBM 704计算机可打印的47个字符组成。其次，出于助记符的原因，允许英语单词和短语代表原子实体很方便。这些符号是原子性的，因为它们可能会忽略字符序列中可能包含的任何子结构。我们假设只能区分不同的符号。然后，S表达式定义如下：

1. 原子符号是S-expressions.

2. 如果e1和e2是S-expressions,那么 (e1 · e2)也是.

实例如下：

$$AB$$
$$(A\cdot B)$$
$$((AB\cdot C)\cdot D)$$

那么，一个S表达式就是一个有序对，其对可以是原子符号或更简单的S表达式。我们可以用S表达式来表示任意长度的列表，如下所示。这个列表

$$(m_1, m_2, \cdots, m_n)$$

代表这个S-expression

$$(m_1\cdot(m_2\cdot(\cdots(m_n\cdot NIL)\cdots)))$$

NIL是用于终止列表的原子符号。由于我们处理的许多符号表达式都可以方便地表示为列表，我们只引入一个列表符号来缩写某些S表达式。我们有

1.(m)代表$(m\cdot NIL)$.

2.$(m_1,\cdots,m_n)$代表$(m_1\cdot(\cdots(m_n\cdot NIL)\cdots))$.

3.$(m_{1,}\cdots,m_n\cdot x)$代表$(m_1\cdot(\cdots(m_n\cdot x)\cdots))$.

子表达式可以类似地缩写。这些缩写的一些示例是

$$((AB,C),D) \quad for \quad ((AB\cdot(C\cdot NIL))\cdot(D\cdot NIL))$$
$$((A,B),C,D\cdot E) \quad for \quad ((A\cdot(B\cdot NIL))\cdot(C\cdot(D\cdot E)))$$

因为我们将逗号表示作为不涉及逗号的缩写，所以我们将它们全部称为S表达式。

b，S表达式的函数和表示它们的表达式。现在，我们定义一类S-expressions的功能。代表这些功能的表达式以常规功能符号编写。但是，为了清楚地将代表函数的表达式与S-expressions区别开来，对于函数名称和整个S表达式集内的变量，我们将使用小写字母序列。我们还使用中括号"[]"和分号";"（而不是括号"()"和逗号","）来表示函数在其参数上的应用。因此我们写为

$$car[x]$$
$$car[cons[(A\cdot B);x]]$$

在这些M表达式（meta-expressions，元表达式）中，任何出现的S表达式都代表自己。

c，基本S函数和谓词。我们介绍以下函数和谓词：

1.$atom$.根据x是否为原子符号,$atom[x]$的值为T或F.因此

$$atom[X]=T$$
$$atom[(X\cdot A)]=F$$

2.eq.当且仅当x和y均为原子时,才定义 eq[x; y]. eq[x;y] 中如果x和y是相同的符号,则 $eq[x;y]=T$,否则$eq[x;y]=F$.因此

$$eq[X;X]=T$$
$$eq[X;A]=F$$
$$eq[X;(X\cdot A)] \text{ is undefined}$$

3.car.当x不是原子的时候$car[x]$才有效.$car[(e_1\cdot e_2)]=e_1$.因此

$$car[X] \text{ is undefined}$$
$$car[(X\cdot A)]=X$$
$$car[((X\cdot A)\cdot Y)]=(X\cdot A)$$

4.cdr.当x不是原子的时候$cdr[x]$才有效.我们有$cdr[(e_1\cdot e_2)]=e_2$.因此

$$cdr[X] \text{ is undefined.}$$
$$cdr[(X\cdot A)]=A$$
$$cdr[((X\cdot A)\cdot Y)]=Y$$

5.$cons$.$cons[x;y]$可以为任意x和y定义.我们有$cons[e_1;e_2]=(e_1\cdot e_2)$.因此

$$cons[X;A]=(XA)$$
$$cons[(X\cdot A);Y]=((X\cdot A)Y)$$

car,cdr和$cons$很容易满足下列关系

$$\begin{aligned} car[cons[x;y]] &= x \\ cdr[cons[x;y]] &= y \\ cons[car[x];cdr[x]] &= x, \text{此处如果} x \text{不是原子的话.} \end{aligned}$$

仅当我们讨论计算机中系统的表示形式时，名称“car”和“cons”才具有助记符的含义。car和cdr在相应位置给出固定表达式的子表达式。cons的组成部分组成了给定结构的表达式。可以通过这种方式形成的函数类别非常有限，而且不是很有意义。

d，递归S函数。当我们允许自己通过条件表达式和递归定义形成S表达式的新函数时，我们得到的函数类型更多（实际上是所有可计算的函数）。现在，我们给出一些可以通过这种方式定义的功能的示例。

1. $ff[x]$.$ff[x]$的值是S表达式x的第一个原子符号,括号被忽略.因此

$$ff[((A\cdot B)\cdot C)]=A$$

我们有

$$ff[x]=[atom[x]\rightarrow x;T\rightarrow ff[car[x]]]$$

现在，我们详细跟踪求值的步骤$ff[((A\cdot B)\cdot C)]$:

$$\begin{aligned} ff[[(A\cdot B)\cdot C]] &= [atom[((A\cdot B)\cdot C)]\rightarrow((A\cdot B)\cdot C);T\rightarrow ff[car[((A\cdot B)\cdot C)]]] \\ &= [F\rightarrow((A\cdot B)\cdot C);T\rightarrow ff(car[((A\cdot B)\cdot C)])] \\ &= [T\rightarrow ff[car[((A\cdot B)\cdot C)]]] \\ &= ff[car[((A\cdot B)\cdot C)] \\ &= ff[(A\cdot B)] \\ &= [atom[(A\cdot B)]\rightarrow(A\cdot B);T\rightarrow ff[car[(A\cdot B)]]] \\ &= [F\rightarrow(A\cdot B);T\rightarrow ff[car[(A\cdot B)]]] \\ &= [T\rightarrow car[(A\cdot B)]] \\ &= ff[car[(A\cdot B)]] \\ &= ff[A] \\ &= [atom[A]\rightarrow A;T\rightarrow ff[car[A]]] \\ &= [T\rightarrow A;T\rightarrow ff[car[A]]] \\ &= A \end{aligned}$$

2.$subst[x;y;z]$.该函数给出用符号表达式x替换符号表达式z中所有出现的原子符号y的结果,其定义为

$$\begin{aligned} subst[x;y;z] &= [atom[z]\rightarrow[eq[z;y]\rightarrow x;T\rightarrow z], \\ &\quad [T\rightarrow cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]] \end{aligned}$$

一个例子如下

$$subst[(X\cdot A);B;((A\cdot B)\cdot C)]=((A\cdot(X\cdot A))\cdot C)$$

3.$equal[x;y]$. 如果x和y是相同的S表达式,则该谓词的值为T,否则为F.我们有

$$\begin{aligned}equal[x;y] \;=\; & [atom[x]\wedge atom[y]\wedge eq[x;y]]\\ & \vee[\neg atom[x]\wedge\neg atom[y]]\\ & \wedge equal[car[x];car[y]]\\ & \wedge equal[cdr[x];cdr[y]]]\end{aligned}$$

可以很方便地看到基本函数在缩写列表符号中的外观，读者可以轻松验证

$$\begin{aligned}&(i)\quad car[(m_1,m_2,\ldots,m_n)]=m_1\\ &(ii)\quad cdr[(m_s,m_2,\ldots,m_n)]=(m_2,\ldots,m_n)\\ &(iii)\quad cdr[(m)]=NIL\\ &(iv)\quad cons[m_1;(m_2,\ldots,m_n)]=[m_1,m_2,\ldots,m_n]\\ &(v)\quad cons[m;NIL]=(m)\end{aligned}$$

我们定义

$$null[x]=atom[x]\wedge eq[x;NIL]$$

该谓词在处理列表时很有用.

car和cdr的组合出现的频率很高,如果我们缩写,可以更简洁地编写许多表达式,cadr[x] 表示 car[cdr[x]], caddr[x] 表示 car[cdr[cdr[x]]], 等等.

另一个有用的缩写是书写列表[e1; e2;...; en] 代表 cons[e1; cons[e2;...; cons[en; NIL]...]].

该函数根据其元素给出列表(e1,...,en).将S表达式视为列表时,以下函数很有用.

1.$append[x;y]$.

$$append[x;y]=[null[x]\rightarrow y;T\rightarrow cons[car[x];append[cdr[x];y]]]$$

一个例子是

$$append[(A,B);(C,D,E)]=(A,B,C,D,E)$$

2.$among[x;y]$.如果符号表达式x出现在列表y的元素之间,则该谓词是true.我们有

$$among[x;y]=\neg null[y]\wedge[equal[x;car[y]]\vee among[x;cdr[y]]]$$

3.$pair[x;y]$.此函数提供列表x和y的对应元素对的列表.我们有

$$\begin{aligned}pair[x;y] \;=\; & [null[x]\wedge null[y]\rightarrow NIL;\\ & [\neg atom[x]\wedge\neg atom[y]\rightarrow\\ & cons[list[car[x];car[y]];pair[cdr[x];cdr[y]]]\end{aligned}$$

例子如下

$$pair[(A,B,C);(X,(Y,Z),U)]=((A,X),(B,(Y,Z)),(C,U))$$

4.$assoc[x;y]$.如果y是形式为$((u_1,v_1),\ldots,(u_n,v_n))$的列表,并且x是u中的符号,则assoc[x; y]是对应的v.我们有

$$assoc[x;y]=eq[caar[y];x]\rightarrow cadar[y];T\rightarrow assoc[x;cdr[y]]$$

一个例子是

$$assoc[X;((W,(A,B)),(X,(C,D)),(Y,(E,F)))]=(C,D)$$

5.$sublis[x;y]$.这里假设x具有成对列表的形式,$((u_1,v_1),\ldots,(u_n,v_n))$,其中u是原子,y可以是任何S表达式.sublis[x;y]是将每个v替换为y中对应的u的结果.为了定义sublis,我们首先定义一个辅助函数.我们有

$$sub2[x;z]=[null[x]\rightarrow z;eq[caar[x];z]\rightarrow c\,\mathrm{ad}\,ar[x];T\rightarrow sub2[cdr[x];z]]$$

然后

$$sublis[x;y]=[atom[y]\rightarrow sub2[x;y];T\rightarrow cons[sublis[x;car[y]];sublis[x;cdr[y]]]]$$

例

$$sublis[((X,(A,B)),(Y,(B,C)));(A,X\cdot Y)]=(A,(A,B),B,C)$$

e，通过S表达式表示S函数。S函数已由M表达式描述。现在，我们给出了一个将M表达式转换为S表达式的规则，以便能够使用S函数来使用S函数进行某些计算并回答有关S函数的某些问题。翻译由以下丰富的规则确定，我们用$\mathcal{E}^*$表示M表达式$\mathcal{E}$的翻译。

1.如果$\mathcal{E}$是S表达式$\mathcal{E}^*$是(QUOTE,$\mathcal{E}$).

2.用小写字母的字符串表示的变量和函数名称将转换为相应大写字母的相应字符串.因此car^*是CAR,$subst^*$是SUBST.

3.结构$f[e_1,\ldots,e_2]$被翻译为$(f^*,e_1^*,\ldots,e_n^*)$.因此cons[car[x];cdr[x]]*是(CONS,(CAR,X),(CDR,X)).

4.$\{[p_1\rightarrow e_1;\ldots;p_n\rightarrow e_n]\}^*$是(COND, $(p_1^*,e_1^*),\ldots,(p_n^*\cdot e_n^*)$).

5.$\{\lambda[[x_1;\ldots;x_n];\mathcal{E}]\}^*$是(LAMBDA, $(x_1^*,\ldots,x_n^*),\mathcal{E}^*$).

6.$\{label[a;\mathcal{E}]\}^*$是(LABEL, $a^*,\mathcal{E}^*$).

使用这些置换函数的约定，M表达式

$$
\begin{array}{l}
label[subst;\lambda[[x;y;z];\\
[atom[z]\rightarrow[eq[y;z]\rightarrow x;T\rightarrow z];\\
T\rightarrow cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]]]
\end{array}
$$

有对应的表达式

$$
\begin{array}{l}
(LABEL,SUBST,(LAMBDA,(X,Y,Z),\\
(COND((ATOM,Z),(COND,(EQ,Y,Z),X),((QUOTE,T),Z))),\\
((QUOTE,T),(CONS,(SUBST,X,Y,(CAR,Z)),(SUBST,X,Y,(CDR,Z))))
\end{array}
$$

该表示法是可写的，部分可读。可以使其结构更不规则，以使其更易于读写。如果计算机上有更多字符可用，则可以大大改善。

f，通用S函数apply。有一个S函数，其属性为：如果f是S函数f'的S表达式，而args是形式为$(arg_1,\ldots,arg_n)$的参数列表，其中$arg_1,\ldots,arg_n$是任意S表达式，然后$apply[f;args]$和$f'[arg_1,\ldots,arg_n]$是给相同的$arg_1,\ldots,arg_2$的值所定义，它们定义的时候是相等的。例如

$$
\begin{array}{l}
\lambda[[x;y];cons[car[x];y]][(A,B);(C,D)]\\
=apply[(LAMBDA,(X,Y),(CONS,(CAR,X),Y));((A,B),(C,D))]=(A,C,D)
\end{array}
$$

S函数apply被定义为

$$
\begin{array}{rcl}
apply[f;args] & = & eval[cons[f;appq[args]];NIL]\\
\text{这里} & & \\
appq[m] & = & [null[m]\rightarrow NIL;T\rightarrow cons[list[QUOTE;car[m]];appq[cdr[m]]]]\\
\text{其中} & & \\
eval[e;a] & = & [atom[e]\rightarrow assoc[e;a];\\
 & & atom[car[e]]\rightarrow[\\
 & & eq[car[e];QUOTE]\rightarrow cadr[e];\\
 & & eq[car[e];ATOM]\rightarrow atom[eval[cadr[e];a]];\\
 & & eq[car[e];EQ]\rightarrow[eval[cadr[e];a]=eval[caddr[e];a]];\\
 & & eq[car[e];COND]\rightarrow evcon[cdr[e];a];\\
 & & eq[car[e];CAR]\rightarrow car[eval[cadr[e];a]];\\
 & & eq[car[e];CDR]\rightarrow cdr[eval[cadr[e];a]];\\
 & & eq[car[e];CONS]\rightarrow cons[eval[cadr[e];a];[eval[caddr[e];\\
 & & a]];T\rightarrow eval[cons[assoc[car[e];a];evlis[cdr[e];a]];a]];\\
 & & eq[caar[e];LABEL]\rightarrow eval[cons[caddar[e];cdr[e]];\\
 & & cons[list[cadar[e];car[e];a]];\\
 & & eq[caar[e];LAMBDA]\rightarrow eval[caddar[e];\\
 & & append[pair[cadar[e];evlis[cdr[e];a];a]]]\\
\text{其中} & & \\
evcon[c;a] & = & [eval[caar[c];a]\rightarrow eval[cadar[c];a];T\rightarrow evcon[cdr[c];a]]\\
\text{其中} & & \\
evlis[m;a] & = & [null[m]\rightarrow NIL;T\rightarrow cons[eval[car\,[m];a];evlis[cdr\,[m];a]]]
\end{array}
$$

现在，我们解释有关这些定义的一些要点。

1.apply本身形成一个表达式,该表达式表示应用于参数的函数的值,并将对该表达式求值的结果放到函数eval上.它使用appq在每个参数周围加上引号,以便eval将其视为对自身的引用.

2.eval[e;a] 有两个参数,一个要求值的表达式e,和一个成对的列表a.每对的第一项是原子符号,第二项是该符号所代表的表达式.

3.如果要求值的表达式是原子表达式,则eval首先在列表a上求值与其匹配的值.

4.如果e不是原子的,而car[e]是原子的,则表达式具有以下形式之一:(QUOTE,e)或(ATOM,e)或(EQ,e1,e2)或(COND,(p1,e1), . . . ,(pn,en))或(CAR,e)或(CDR,e)或(CONS,e1,e2）或(f,e1,...,en),其中f是原子符号.

在(QUOTE,e)的情况下,采用表达式e本身.在(ATOM,e)或(CAR,e)或(CDR,e)的情况下,求值表达式e并执行适当的函数.在(EQ,e1,e2)或(CONS,e1,e2)的情况下,必须求值两个子表达式.在(COND,(p1,e1),...,(pn,en)))的情况下,必须对p进行求值,直到找到真p,然后必须求值相应的e.这是由evcon完成的.最后,在(f,e1,...,en)的情况下,我们求值将表达式f替换为列表a中与之匹配的表达式的结果.

5.((LABEL,f,E),e1,...,en)的求值是通过对(E,e1,...,en)进行匹配，并将匹配的(f,(LABEL,f,E))放在对应的前一个列表a的前面.

6.最后,((LAMBDA,(x1,...,xn),E),e1,...,en)的值,通过对E进行求值,将E与列表对((x1,e1),...,((xn,en)))放在上一个列表a的前面.

可以删除列表a,并通过用参数E替换表达式E中的变量来评估LAMBDA和LABEL表达式.不幸的是,出现了涉及绑定变量冲突的困难,但是通过使用列表a可以避免这些困难.

通过使用apply计算函数的值是比电子计算机更适合计算的活动.但是,作为说明,我们现在给出一些计算$apply[(LABEL, FF, (LAMBDA, (X), (\text{COND}, (\text{ATOM}, X), X), ((QUOTE, T), (FF, (CAR, X))))); (A \cdot B)] = A$的步骤.第一个参数是S表达式,它表示第三节中定义的函数ff.我们将使用字母ϕ进行缩写.我们有

$$\begin{aligned}
apply[\phi; ((A \cdot B))] &= eval[((LABEL, FF, \boldsymbol{\psi}), (QUOTE, (A \cdot B))); NIL] \\
&\quad \boldsymbol{\psi}\text{的地方是}\phi\text{从}(LAMBDA\text{开始的一部分} \\
&= eval[((LAMBDA, (X), \omega), (QUOTE, (A \cdot B))); ((FF, \phi))] \\
&\quad \omega\text{的地方是}\phi\text{从}(COND\text{开始的一部分} \\
&= eval[((COND, (\pi_1, \epsilon_1), (\pi_2, \epsilon_1)); ((X, (QUOTE, (A \cdot B))), (FF, \phi))] \\
&\quad \text{用}a\text{表示}((X, (QUOTE, (A \cdot B))), (FF, \phi)), \text{得到} \\
&= evcon[((\pi_1, \epsilon_1), (\pi_2, \epsilon_1)); a] \\
&\quad \text{这里涉及到求值}[\pi_1; a] \\
&= eval[(ATOM, X); a] \\
&= atom[eval[X; a]] \\
&= atom[eval[assoc[X; ((X, (QUOTE, (A \cdot B))), (FF, \phi))]; a]] \\
&= atom[eval[(QUOTE, (A \cdot B)); a]] \\
&= atom[(A \cdot B)] \\
&= F
\end{aligned}$$

我们的主要计算继续如下

$$\begin{aligned}
&apply[\phi; ((A \cdot B))] = evcon[((\pi_2, \epsilon_2)); a] \\
&\text{这里涉及到}eval[\pi_2, a] = eval[(QUOTE, T); a] = T
\end{aligned}$$

我们的主要计算再次继续

$$\begin{aligned}
apply[\varphi; ((A \cdot B))] &= eval[\epsilon_2; a] \\
&= eval[(FF, (CAR, X)); a] \\
&= eval[CONS[\phi; evlis[((CAR, X)); a]]] \\
&\quad \text{这里涉及到}evlis[((CAR, X)); a] \\
eval[(CAR, X); a] &= car[eval[X; a]] \\
&= car[(A \cdot B)] \\
&\quad \text{我们从}atom[eval[X; a]] = A\text{的较早计算中采取了步骤} \\
&\quad \text{于是}evlis[((CAR, X)); a]\text{就变成了} \\
&\quad list[list[QUOTE; A]] = ((QUOTE, A)) \\
&\quad \text{我们的主要运算变成了} \\
&= eval[(\phi, (QUOTE, A)); a]
\end{aligned}$$

按照计算开始的步骤进行后续步骤.LABEL和LAMBDA导致将新的对添加到a,从而给出新的序对列表a1.有条件的eval[(ATOM,X);a_1]的π_1结构有T值,因为X是a_1里面(QUOTE,A)的第一项,而不是像a那样是(QUOTE,$(A \cdot B)$).

因此,我们最终从evcon得到eval[X;a_1],这是A.

g，以函数作为参数的函数。有许多有用的函数，其中一些参数是函数。它们在定义其他函数时特别有用。这样的函数之一就是maplist [x; f]，带有S表达式参数x和参数f，该参数f是从S表达式到S表达式的函数。我们定义

$$maplist[x; f] = [null[x] \rightarrow NIL; T \rightarrow cons[f[x]; maplist[cdr[x]; f]]]$$

maplist的有用性由涉及x的和与乘积以及其他变量的表达式的x的偏导数的公式说明。我们将要区分的S表达式形成如下

1.原子符号是允许的表达式.

2.如果$e_1, e_2, \ldots, e_n$是允许的表达式,$(PLUS, e_1, \ldots, e_2)$和$(TIMES, e_1, \ldots, e_2)$也是,代表了总和与乘积,分别是$e_1, \ldots, e_n$.

本质上，这是波兰表达式的函数表示法(the Polish notation for functions)，不同之处在于，括号和逗号允许包含可变数量的参数的函数。允许的表达式的一个示例是(TIMES,X,(PLUS,X,A),Y)，其常规代数符号为X(X + A)Y。

我们的微分公式，给出y相对于x的导数是

$$\begin{aligned}
diff[y; x] &= [atom[y] \rightarrow [eq[y; x] \rightarrow ONE; T \rightarrow ZERO]; \\
&\quad eq[car[Y]; PLUS] \rightarrow cons[PLUS; maplist[cdr[y]; \lambda[[z]; \\
&\quad diff[car[z]; x]]]]; \\
&\quad eq[car[y]; TIMES] \rightarrow cons[PLUS; maplist[cdr[y]; \lambda[[z]; \\
&\quad cons[TIMES; maplist[cdr[y]; \lambda[[w]; \neg eq[z; w] \rightarrow car[w]; \\
&\quad T \rightarrow diff[car[[w]; x]]]]]]]]]
\end{aligned}$$

由该公式计算表达式(TIMES,X,(PLUS,X,A),Y)的导数是

$$\begin{aligned}
&(PLUS, (TIMES, ONE, (PLUS, X, A), Y), \\
&(TIMES, X, (PLUS, ONE, ZERO), Y), \\
&(TIMES, X, (PLUS, X, A), ZERO)
\end{aligned}$$

除了maplist之外，具有函数参数的另一个有用函数是search，其定义为

$$search[x;p;f;u] = [null[x] \rightarrow u; p[x] \rightarrow f[x]; T \rightarrow search[cdr[x];p;f;u]]$$

该函数用于在列表中搜索具有属性p的元素，如果找到了这样的元素，该元素的f被捕获。如果没有这样的元素，则不计算参数的函数u。

4 LISP编程系统

LISP编程系统是用于使用IBM 704计算机执行以下操作的系统：使用S表达式形式的符号信息进行计算。它已经或将用于以下目的：

1.编写编译器以将LISP程序编译为机器语言.

2.编写程序以检查形式逻辑系统中的证明.

3.编写程序以进行形式化的微分和积分.

4.编写程序以实现用于在谓词演算中生成证明的各种算法.

5.进行某些工程计算,其结果是公式而不是数字.

6.编写 the Advice Taker 系统.

该系统的基础是一种编写计算机程序以求值S函数的方法，以下各节将对此进行描述。

除了描述S函数的功能外，还有在FORTRAN[4]或ALGOL[5]的语句序列编写的程序中沿用S函数的一些功能。这些特性将不在本文中描述。

a，通过列表结构表示S表达式。列表结构是按图1a或1b排列的计算机单词的集合。列表结构的每个单词由图中细分的矩形之一表示。矩形的左框表示单词的地址(address)字段，右框表示减量(decrement)字段。从一个框到另一个矩形的箭头表示与该框相对应的字段包含与另一个矩形相对应的单词的位置。

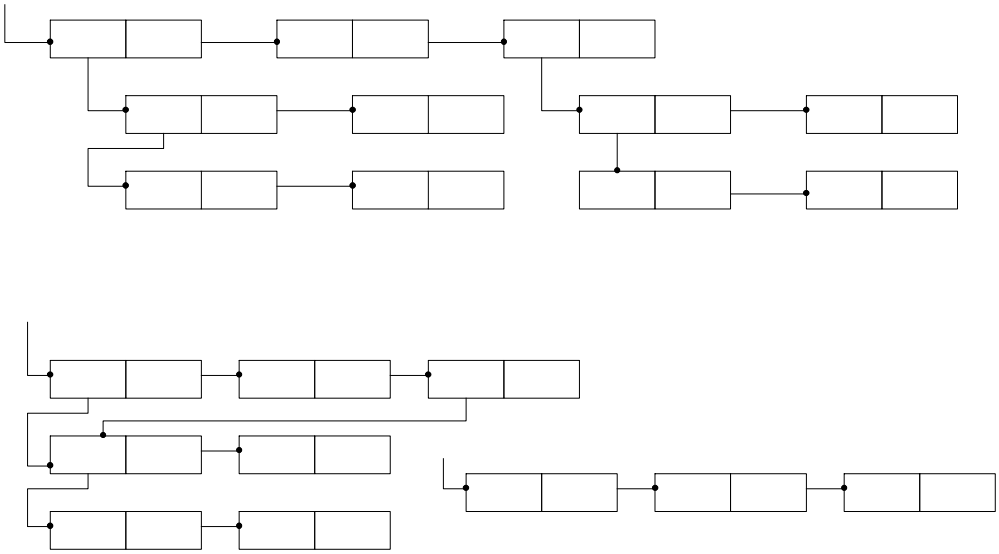


Figure 1

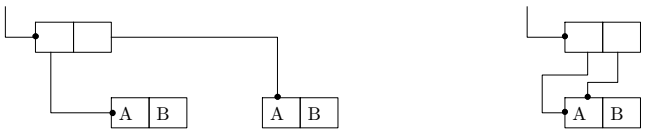
允许子结构出现在列表结构的多个位置，如图1b所示，但不允许结构具有循环，如图1c所示。原子符号在计算机中用特殊形式的列表结构表示，该列表结构称为符号的关联列表。第一个单词的地址字段包含一个特殊的常量，该常量使程序能够知道该单词代表一个原子符号。我们将在第4b节中描述关联列表。

非原子的S表达式x由一个单词表示，其地址和减量部分分别包含子表达式car[x]和cdr[x]的位置。如果我们使用符号A，B...，为了表示这些符号的关联列表的位置，则S表达式$((A \cdot B) \cdot (C \cdot (E \cdot F)))$由图2的列表结构a表示。转到S表达式的列表形式，我们看到S表达式(A,(B,C),D)，它是$(A \cdot ((B \cdot (C \cdot NIL)) \cdot (D \cdot NIL)))$，由图2b的列表结构表示。

(a) (b)

Figure 2

当列表结构被视为代表一个列表时，我们看到列表的每个元素都占据一个单词的地址部分，其递减部分指向包含下一个元素的单词，而最后一个单词的尾部4形式为NIL。具有给定子表达式多次出现的表达式可以用一种以上的方式表示。子表达式的列表结构是否重复取决于程序的历史记录。尽管重复子表达式是否出现在机器外部，但对程序结果的影响不会改变，尽管这会影响时间和存储要求。例如，S表达式$((A \cdot B) \cdot (A \cdot B))$可以由图3a或3b的列表结构表示。



(a) (b)

Fogure 3

禁止循环列表结构本质上是禁止将表达式表示为其自身的子表达式。在具有我们的拓扑结构的世界中，这样的表达不可能在纸上存在。循环列表结构在机器中将具有一些优势，例如，用于表示递归函数,但是在打印它们时以及在某些其他操作中存在困难，建议目前不使用它们。

列表结构用于存储符号表达式的优点是：

1.无法预先预测程序必须处理的表达式的大小,甚至数目.因此,难以布置固定长度的存储块以容纳它们.

2.不再需要寄存器时,可以将它们放回空闲存储列表中.即使返回到列表中的一个寄存器也很有价值,但是如果将表达式线性存储,则很难利用可能具有奇数大小的寄存器块.

3.作为多个表达式的子表达式出现的表达式只需要在存储中表示一次.

b，关联列表。在LISP编程系统中，我们在符号的关联列表中放置的内容超过了前面各节中描述的数学系统所需的数量，实际上，我们希望与该符号关联的任何信息都可以放在关联列表中。该信息可能包括：打印名称，即代表机器外部符号的字母和数字字符串；如果符号代表数字，则为数值；如果符号以某种方式充当其名称，则为另一个S表达式；或例程的位置（如果该符号表示一个具有机器语言子例程的函数）。所有这些暗示着，在机器系统中，比在数学系统的各节中描述的原始实体更多。

就目前而言，我们将仅描述如何在关联列表上表示打印名称，以便在读取或打印程序时可以在打孔卡，磁带或打印页面上的信息与机器内部的列表结构之间建立对应关系。符号DIFFER，ENTIATE的关联列表具有图4所示形式的一段。这里的pname是一个符号，指示该符号的打印名称的结构，该符号的关联列表挂在关联列表的下一个单词上。在图的第二行，我们列出了三个单词。这些单词中每个单词的地址部分都指向一个包含六个6位字符的单词。最后一个字用6位组合填充，该组合不代表计算机可打印的字符。（回想一下，IBM 7O4有一个36位的单词，每个可打印的字符都由6位表示。）带有字符信息的单词的存在意味着关联列表本身并不代表S表达式，并且在关联列表中只有一些用于处理S表达式的功能才有意义。

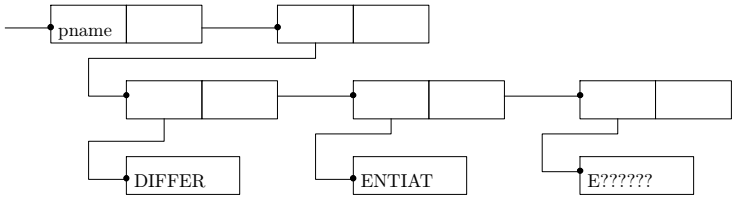


Figure 4

c，自由存储列表。在任何给定时间，仅保留用于列表结构的一部分内存实际上将用于存储S表达式。其余的寄存器（在我们的系统中，最初的数量大约为15,000）排列在一个称为自由存储列表的列表中。程序中的某个特定寄存器FREE包含此列表中第一个寄存器的位置。无需为用户编写将寄存器返回到空闲存储列表的程序。此返回自动发生，大致如下（需要在此报告中对此过程进行简化的描述）：程序中有一组固定的基址寄存器，其中包含程序可访问的列表结构的位置。当然，由于列表结构分支，因此可能涉及任意数量的寄存器。该程序可访问的每个寄存器都是可访问的，因为可以通过一连串的car和cdr操作从一个或多个基本寄存器访问它们。当基址寄存器的内容更改时，可能会发生car-cdr链无法从任何基址寄存器访问基址寄存器先前指向的寄存器的情况。此类寄存器可能被程序丢弃，因为任何可能的程序都无法找到其内容；因此，不再需要其内容，因此我们希望将其重新列入自由存储列表。这是通过以下方式实现的。

在程序耗尽可用存储空间之前，什么也不会发生。当需要一个空闲寄存器，并且空闲存储列表中没有剩余寄存器时，将开始一个回收周期。

首先，程序找到可从基址寄存器访问的所有寄存器，并使它们的符号为负。这是通过从每个基址寄存器开始并更改car-cdr链可以从中访问的每个寄存器的符号来实现的。如果程序在此过程中遇到一个已经带有负号的寄存器，则认为该寄存器已经到达。在所有可访问的寄存器的符号都更改之后，程序将遍历为存储列表结构而保留的内存区域，并将在上一步中未更改符号的所有寄存器放回自由存储列表，然后 使可访问寄存器的符号再次为正。

由于该过程是完全自动的，因此对于程序员而言，比必须跟踪并清除不需要的列表的系统更方便。它的效率取决于是否接近用可用列表耗尽可用内存。这是因为回收过程需要几秒钟的时间来执行，因此如果程序不打算将其大部分时间用于回收，则必须在空闲存储列表中添加至少数千个寄存器。

d，计算机中的基本S函数。现在我们将描述atom，=，car，cdr和cons的计算机表示。将S表达式传达给程序，该程序将表示功能的位置表示为代表单词的位置，并且程序以相同的形式给出S表达式的答案。

$atom$，如上所述，表示原子符号的单词在其地址部分具有一个特殊的常数：原子被编程为测试该零件的开放子程序。除非在条件表达式中作为条件出现M表示原子[e]，否则将根据测试结果生成符号T或F。在条件表达式的情况下，将使用条件转化，并且不会生成符号T或F。

eq，eq[e;f]程序涉及测试单词位置的数字相等性。之所以可行，是因为每个原子符号只有一个关联列表。与原子一样，结果是条件转化符号T或F之一。

car，计算car[x]涉及获取寄存器x的地址部分的内容，这本质上是通过单条指令CLA 0（即参数位于索引寄存器中，并且结果显示在累加器的地址部分中）来完成的。（我们认为，函数的定义中规定了函数取自变量以及将结果放入的位置，程序员或编译器有责任插入所需的数据移动指令以获取函数的位置。一个计算的结果在下一个位置。）（“car”是“寄存器地址部分的内容”（contents of the address part of register）的助记符。）

cdr，cdr的处理方式与car相同，除了结果出现在累加器的减量部分（“ cdr”代表“寄存器的减量部分的内容”（contents of the decrement part of register）。）

$cons$，cons[x; y]必须是地址分别为x和y的寄存器的位置。计算机中可能没有这样的寄存器，即使有，它也会很费时。实际上，我们要做的是从空闲存储列表中获取第一个可用的寄存器，将x和y分别放在地址和减量部分中，并使函数的值成为所取寄存器的位置。（“ cons”是“ construct”的缩写。）

当空闲存储列表用完时，cons的子例程将启动回收。在当前使用的系统版本中，cons由封闭的子例程表示。在编译版本中，cons是打开的。

e，通过程序表示S函数。手工或通过编译器程序对由car，cdr和cons组成的函数进行编译很简单。条件表达式没有什么麻烦，除了条件表达式必须编译为仅计算所需的p和e之外。但是，在递归函数的编译中会出现问题。

一般而言（我们将讨论异常），递归函数的例程将自身用作子例程。例如，subst [x; y; z]的程序本身作为子例程来评估替换为子表达式car [z]和cdr [z]的结果。而subst [x; y; cdr [z]]正在求值，先前求值subst [x; y; car [z]]的结果必须保存在临时存储寄存器中。使用公共下推列表的SAVE和UNSAVE例程解决了这种可能的冲突。在递归函数例程的开头输入SAVE例程，并请求保存给定的一组连续寄存器。为此，保留了一块称为公共下推列表的寄存器。SAVE例程具有一个索引，该索引告诉它下拉列表中已经使用了多少个寄存器。它将要保存的寄存器的内容移到下拉列表中的第一个未使用的寄存器中，前进列表的索引，并返回到控制来自的程序。然后，该程序可以自由使用这些寄存器进行临时存储。在例程退出之前，它使用UNSAVE，它将从下推列表中恢复临时寄存器的内容，并向后移动该列表的索引。这些约定的结果用编程术语来描述，方法是说递归子例程对临时存储寄存器是透明的。

f， LISP编程系统的状态(1960年2月)。在第5节中描述的函数apply的变体已被翻译为IBM 704的程序APPLY。由于此例程可以根据给定的S函数表达式和参数描述来计算S函数的值，它充当LISP编程语言的解释器，该语言以这种方式描述计算过程。

程序APPLY已嵌入到具有以下功能的LISP编程系统中：

1.程序员可以通过S表达式定义任意数量的S函数.这些函数可以相互引用,也可以指由机器语言程序表示的某些S功能.

2.可以计算已定义函数的值.

3.S表达式可以被读取和打印(直接或通过磁带).

4.包括一些错误诊断和选择性跟踪功能.

5.程序员可能已经选择了编译为机器语言程序的S函数,并将其放入核心存储器中.编译函数的值的计算速度大约是解释时的60倍,编译速度足够快,因此无需打孔已编译的程序以备将来使用.

6.“程序功能”允许程序以ALGOL样式包含赋值和go语句.

7.在系统中可以使用浮点数进行计算,但这效率很低.

8.正在准备程序员手册.LISP编程系统适用于计算,在该计算中,可以方便地将数据表示为符号表达式,允许使用与子表达式相同类型的表达式.正在准备IBM 709的系统版本.

5 符号表达式函数的其他形式

定义符号表达式函数的方法有很多，与我们采用的系统非常相似。它们每个都涉及三个基本函数，条件表达式和递归函数定义，但是与S表达式相对应的表达式类别不同，这些函数的精确定义也不同。我们将描述这些称为线性LISP的变体之一。

L表达式定义如下：

1.允许使用有限的字符列表.

2.L表达式中任何允许的字符字符串,这包括由Λ表示的空字符串.

3.字符串有三个功能:

1.first[x]是字符串x的第一个字符.

2.rest [x]是当删除字符串的第一个字符时保留的字符串.

rest [Λ]未定义.例如: rest[ABC] = BC.

3.combine[x; y]是通过在字符串y前面添加字符x形成的字符串.例如:combine[A;BC]=ABC.

字符串上有三个谓词:

1.char[x],x是单独的字符.

2.null[x],x是空字符.

3.x = y,为x和y字符定义.

线性LISP的优点是，没有字符被赋予特殊的角色，LISP中的括号，点和逗号也是如此。这允许使用可以线性编写的所有表达式进行计算。线性LISP的缺点是子表达式的提取是相当复杂的操作，而不是基本操作。用线性LISP编写与LISP基本功能相对应的函数并不难，因此，从数学上讲，线性LISP包括LISP。事实证明，这是最简单的编程方法，在线性LISP中，更复杂的操作。但是，如果要用计算机例程来表示功能，则LISP本质上会更快。

6 流程图和递归

由于计算机程序的通常形式和递归函数定义在计算上都是通用的，因此显示它们之间的关系很有趣。递归符号功能到计算机程序的翻译是本报告其余部分的主题。在本节中，我们至少从原理上说明了如何走另一条路。

计算期间任何时候机器的状态由多个变量的值给出。让这些变量组合成向量ξ。 考虑一个具有一个入口和一个出口的程序块。它定义并本质上由某个函数f定义，该函数将一种机器配置转换为另一种，即f的形式为$\xi' = f(\xi)$。让我们调用程序块的关联功能。现在，通过确定元素π将许多这样的块组合到程序中，这些决定元素决定每个块完成之后接下来要输入哪个块。不过，让整个程序还有一个入口和一个出口。

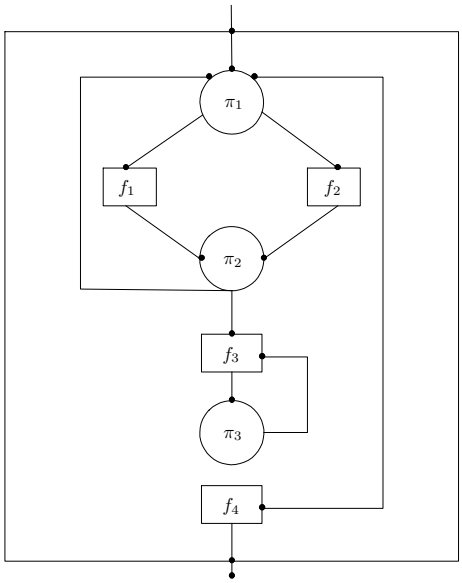


Figure 5

我们以图5的流程图为例。让我们描述函数 r[ξ] ，它给出了整个块的入口和出口之间的向量ξ的转换。我们将结合函数 s(ξ) 和 t[ξ] 对其进行定义，它们给出ξ在点S和T之间以及出口处分别经历的变换。我们有

$$\begin{aligned} r[\xi] &= [\pi_1 1[\xi] \to S[f_1[\xi]]; T \to S[f_2[\xi]]] \\ S[\xi] &= [\pi_2 1[\xi] \to r[\xi]; T \to t[f_3[\xi]]] \\ t[\xi] &= [\pi_3 1[\xi] \to f_4[\xi]; \pi_3 2[\xi] \to r[\xi]; T \to t[f_3[\xi]]] \end{aligned}$$

给定一个具有单个入口和单个出口的流程图，很容易记下递归函数，该递归函数根据计算块和分支谓词的相应函数给出从入口到出口的状态向量的转换。 通常，我们进行如下操作。在图6中，令β为n路分支点，令 $f_1, \ldots, f_n$ 为导致分支点 $\beta_1, \ldots, \beta_n$ 的计算。设 ϕ 是在β和图表的出口之间转换ξ的函数，而 $\phi_1, \ldots, \phi_n$ 是 $\beta_1, \ldots, \beta_n$ 的对应函数。然后我们写为

$$\phi[\xi] = [p_1[\xi] \to \phi_1[f_1[\xi]]; \ldots; p_1[\xi] \to \phi_1[\xi]]$$

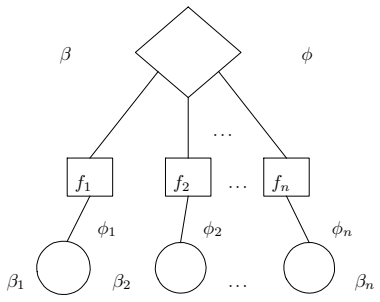


Figure 6

7 致谢

N. Rochester注意到λ符号不足以命名递归函数，他发现了这里使用的涉及标签的解决方案的替代方案。由cons的子例程的形式，使其与其他功能结合在一起，是由IBM公司的C. Gerberick和H. L. Gelernter发明的，并与另一个编程系统有关。LlSP编程系统由包括R. Brayton，D.Edwards，P.Fox，L.Hodes，D.Luckham，K.Maling，J.McCarthy，D.Park，S.Russell在内的团队开发。

该小组得到了M.I.T. 计算中心，由M.I.T. 电子研究实验室（部分由美国陆军（信号军），美国空军（科学研究室，空中研究与发展司令部）和美国海军（海军研究室）提供支持）。作者还深切感谢Alfred P. Sloan基金会的个人财政支持。

8 引用

1. J. McCARTHY, Programs with common sense, Paper presented at the Symposium on the Mechanization of Thought Processes, National Physical Laboratory, Teddington, England, Nov. 24-27, 1958. (Published in Proceedings of the Symposium by H. M. Stationery Office).

2. A. NEWELL AND J. C. SHAW, Programming the logic theory machine,Proc. Western Joint Computer Conference, Feb. 1957.

3. A. CHURCH, The Calculi of Lambda-Conversion (Princeton University Press, Princeton, N. J., 1941).

4. FORTRAN Programmer's Reference Manual, IBM Corporation, New York, Oct. 15, 1956.

5. A. J. PERLIS AND K. SAMELS0N, International algebraic language,Preliminary Report, Comm. Assoc. Comp. Mach., Dec. 1958.