TR-106

Qute処理系操作説明書

桜井貴文（東京大学）
藤田正幸（三菱総合研究所）

April, 1985

「本研究は第５世代コンピュータプロジェクトの
一環として行なわれた」

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

目次

Ｐａｒｔ１　Ｑｕｔｅの定義

１．序

Ｑｕｔｅは、Ｐｒｏｌｏｇ及び大部分の論理型言語の特徴であるｕｎｉｆｉｃａｔｉｏｎを取り入れた関数型プログラミング言語である。　即ち、大部分の関数型言語が変数に値を与える際（例えば引数の受け渡しの時）一方向の代入によるのに対し、Ｑｕｔｅは両方向性を持ったｕｎｉｆｉｃａｔｉｏｎによっている。この様に論理型言語と関数型言語が融合されているので、Ｑｕｔｅでは論理型言語であるＰｒｏｌｏｇ風のプログラムを書く事もできるし、関数型言語であるＭＬ風のプログラムを書く事もできる。　Ｑｕｔｅのプログラムは並列に評価する事ができる（例えば並列ａｎｄ）。しかも評価の順序にかかわらず結果は同一である。ＱｕｔｅではＰｒｏｌｏｇにおけるｏｒの代わりにｉｆ－ｔｈｅｎ－ｅｌｓｅが基本構成要素になっているが、結果が同一である事を保証するためにｉｆ－ｔｈｅｎ－ｅｌｓｅの定義は、ｕｎｉｆｉｃａｔｉｏｎを取り入れた事と相まって、普通の言語における場合とは異なる。即ち、ある条件（ｉｆ部の変数が‘十分’にｉｎｓｔａｎｔｉａｔｅされる）が満たされるまでは、ｔｈｅｎ部またはｅｌｓｅ部の評価に移らない。従ってＱｕｔｅではｉｆ－ｔｈｅｎ－ｅｌｓｅを使ってある種の同期を必要とするプログラムを書く事ができる。

Ｑｕｔｅのプログラムが終了した時の状態はＣｏｎｃｕｒｒｅｎｔ　Ｐｒｏｌｏｇと同様に、ｓｕｃｃｅｅｄ，ｆａｉｌ，ｓｕｓｐｅｎｄの３つのいずれかである。但しｓｕｃｃｅｅｄの時は、値を持つ。

ここではＱｕｔｅはどの様なプログラミング言語であるかをｉｎｆｏｒｍａｌに説明する。ｆｏｒｍａｌな定義は、佐藤雅彦・桜井貴文による「ＱＵＴＥ：Ａ　Ｆｕｎｃｔｉｏｎａｌ　Ｌａｎｇｕａｇｅ　Ｂａｓｅｄ　ｏｎ　Ｕｎｉｆｉｃａｔｉｏｎ」（ＦＧＣＳ　’８４　Ｐｒｏｃｅｅｄｉｎｇｓ）を参照されたい。

２．Ｑｕｔｅのｓｙｎｔａｘ

Ｑｕｔｅのプログラムはｅｘｐｒｅｓｓｉｏｎと呼ばれる。ｅｘｐｒｅｓｓｉｏｎは変数と定数からいくつかのｃｏｎｓｔｒｕｃｔｏｒによって順々に構成される。またｅｘｐｒｅｓｓｉｏｎのうちある形をしたものはｐａｔｔｅｒｎと呼ばれ、後に説明する様に　‘データ’の役割を持っている。

2．1　定数と変数

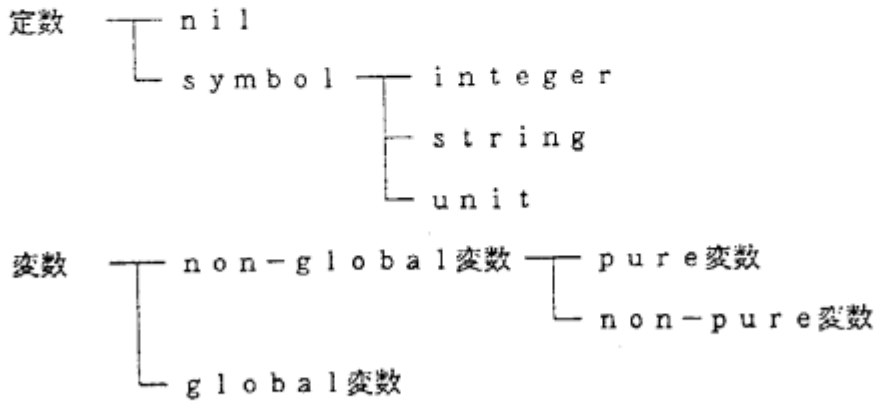
定数はｎｉｌとｓｙｍｂｏｌの２種類から成る。ｎｉｌと呼ばれる定数は唯一つで、＊と書かれる。ｓｙｍｂｏｌはｉｎｔｅｇｅｒ，ｓｔｒｉｎｇ，ｕｎｉｔの３種類から成る。ｉｎｔｅｇｅｒは十進表示で表現され、ｓｔｒｉｎｇは" "で囲まれた文字列で表現される。但し、文字列の中に"を入れたい時は＼でｅｓｃａｐｅする。またｕｎｉｔと呼ばれる定数も唯一で（　）と書かれる。

例　ｉｎｔｅｇｅｒ　・・・・　０　２　１２３
　　ｓｔｒｉｎｇ　・・・・　"ａｐｐｌｅ"　"Ｑｕｔｅ"
　　"ｄｏｕｂｌｅ　ｑｕｏｔｅ＼""

変数はｎｏｎ－ｇｌｏｂａｌ変数とｇｌｏｂａｌ変数の２種類から成る。ｎｏｎ－ｇｌｏｂａｌ変数はｐｕｒｅ変数とｎｏｎ－ｐｕｒｅ変数から成り、ｐｕｒｅ変数は大文字で始まる文字列または１文字の英字の後に０個以上の数字列を付けた文字列で表される。ｎｏｎ－ｐｕｒｅ変数はｐｕｒｅ変数の前に１個以上の＃を付けたものである。ｇｌｏｂａｌ変数は小文字で始まる長さ２以上の英数字列でｐｕｒｅ変数でない文字列で表される。

例　ｐｕｒｅ変数　・・・・　Ｘ　Ｑｕｔｅ　ｘ　ｙ１
　　ｎｏｎ－ｐｕｒｅ変数　・・・・　＃Ｙ　＃＃ｘ１２
　　ｇｌｏｂａｌ変数　・・・・　ｆｏｏ　ｑｕｔｅ

まとめると、



2. 2 expression

expressionは次のいずれかである。但しaとbとcはexpression である。

(E1) 変数

(E2) symbol

(E3) * % nil

(E4) 〔a, b〕 % list

(E5) a, b % and

(E6) λa. b % function

(E7) a ; b % sequential-and

(E8) if a then b else c,
% if-then-else

(E9) ¬a % negation

(E10) a b % application

(E11) a=b % unification

(E3) ～ (E11) をそれぞれ%以降に書いた名で呼ぶ。

(E3), (E4) はLispのnil, dotted pairに対応するもので、Lispと同様のlist notationをもつ。

例 〔a, b〕は〔a. 〔b. *〕〕
〔 〕 は *

丸かっこはgroupingのために用いる。また以下の省略形を用いる。

省略形	expression
<a>	(λ ( ). a) ( )
fail	(0=1)
fa<-b	f = (λa. b)

ａｎｄとｓｅｑｕｅｎｔｉａｌ－ａｎｄは右に、ａｐｐｌｉｃａｔｉｏｎは左にａｓｓｏｃｉａｔｅする。

例　a，b，c　　は　　a，（b，c）
　　a；b；c　　は　　a；（b；c）
　　a　b　c　　は　　（a b）c

これらのｅｘｐｒｅｓｓｉｏｎのｉｍｐｌｅｍｅｎｔ上での表現及び優先度については付録Ａを見よ。優先度については以下それに従う。

```
例　append（X，Y）　<－
　　if ▲#X＝〔 〕 then
　　　　#Y
　　else if　#X＝〔X1．X2〕 then
　　　　〔X1．append（X2，#Y）〕
　　else
　　　　fail
```

2．3　ｐａｔｔｅｒｎ

ｐａｔｔｅｒｎは特別な形のｅｘｐｒｅｓｓｉｏｎであり、次のいずれかである。但しａとｂはｅｘｐｒｅｓｓｉｏｎを、ｐとｑはｐａｔｔｅｒｎを表す。

（Ｐ１）変数
（Ｐ２）ｓｙｍｂｏｌ
（Ｐ３）＊
（Ｐ４）ｐ，ｑ
（Ｐ５）〔ｐ，ｑ〕
（Ｐ６）λａ，ｂ

例　"ａｐｐｌｅ"

x, 1, 2

〔"apple", "orange", "grape"〕

λx. foo x

2. 4 自由変数

expression中に現れるfunction及びif-then-elseのif部とthen部はlocalなscopeを形成する。

例えば次のexpression

X, λX. X

において、2番目と3番目に現れるXは同じ変数であるが、最初に現れるXと2番目に現れるXは異なる変数である。また

X, if X=1 then X else X

において1番目と4番目のXは同じ変数、2番目と3番目のXは同じ変数であるが、1番目と2番目のXは異なる変数である。

scope内から外の変数にaccessしたい時は、変数の前に#をつける。

例えば

X, λX. #X

において、1番目のXと#X は同じ変数を表している。更に

X, λ#X. #X

とすればすべての変数は同じ変数を表すことになる。if-then-elseにおいても同様で

X, if #X=1 then #X else X

とすればすべての変数（Xと#X）は同じものとなる。

scopeはnestしてもよい。その場合#1個につき1つのscopeから出る。

従って

X¹,
if #X¹=X² then
  if #X²=X³ then ##X¹ else foo X²
else
  X¹

において右肩に同じ添字をもつ変数が同じものである。ｓｃｏｐｅはｇｌｏｂａｌ変数に対しては効力を持たない。

ｅｘｐｒｅｓｓｉｏｎにおいて一番外のｓｃｏｐｅから見える変数（ｇｌｏｂａｌ変数を含める）を自由変数という。上のｅｘｐｒｅｓｓｉｏｎにおいて自由変数はX'とｆｏｏである。

例　V＝（X，Y）；〔1，X〕＝Y；（λx．〔x，＃V〕）Z
　　の自由変数はV　X　Y　Z

＃は変数の前にのみ現れるが、以下では一般のｐａｔｔｅｒｎ　ｐに対し＃ｐという表記を許し＃ｐの自由変数はｐの自由変数で、ｇｌｏｂａｌでない変数には＃を加えたものと定義する。ｐａｔｔｅｒｎの前に＃を２個以上付ける時も同様である。

例　＃〔X，Y，1〕の自由変数　　＃X　＃Y
　　＃（λx．ｆｏｏ（x，＃x））の自由変数　　ｆｏｏ　＃x

自由変数と同様に自由ｐａｔｔｅｒｎを定義することができる。但し、変数以外の自由ｐａｔｔｅｒｎは＃・・・＃ｐという形の一部分としてのみ表れるとする。

例　　〔1，X〕＝Y；（λx．〔x，＃（X，Y）〕）z
　　の自由ｐａｔｔｅｒｎは、
　　X　Y　（X，Y）　z
　　であり、
　　λx．（＃〔1，x〕，λx．＃＃〔2，x〕）
　　の自由ｐａｔｔｅｒｎは、
　　〔1，x〕　〔2，x〕
　　であり、
　　（λz．　ｆｏｏ（＃X，Z，＃＃W））Y
　　の自由ｐａｔｔｅｒｎは、

foo X #W

である。

3. unification

unificationはpatternについてのみ定義する（expression については定義しない）。二つのpatternの自由変数に適当な代入を行って同じpatternのなるとき、 2つのpatternはunifyできる、と定義する。さらにその2つのpatternがunifyしてできたpatternとは2つのpatternをunifyする代入のうち最も一般的な代入によってできたpatternとする。

例1 〔X, Y〕と〔1, 2〕はunifyできる

できたpatternは〔1, 2〕

X, 2, Zと1, Yはunifyできる

できたpatternは1, 2, Z

λx. foo xとλy. foo yはunifyできない

λx. foo #Xとλx. foo #Yはunifyできる

できたpatternはλx. foo #XまたはλX. foo #Y

λx. foo #Xとλx. foo 〔1〕はunifyできない

λx. foo #Xとλx. foo #〔1, Y〕はunifyできる

できたpatternはλx. foo #〔1, Y〕

occur checkするかしないかは定めないが、する場合でもfunctionの中に現れる変数についてはcheckしない。

例2 Xとλx. #Xはunifyできる

できたpatternはλx. #（λx. #（λx. . . .））

いう無限patternである

この様に無限patternを許す様にpatternの定義は拡張されるが、これは

ｒｅｃｕｒｓｉｖｅ　ｆｕｎｃｔｉｏｎを定義可能にするためである。

実はここでのｕｎｉｆｉｃａｔｉｏｎの定義はｆｕｎｃｔｉｏｎに関しては１節であげた論文での定義より拡張されている。論文の定義では２つのｆｕｎｃｔｉｏｎが全く同じものである時のみｕｎｉｆｙできる。

例３　例１の４組のｆｕｎｃｔｉｏｎは論文の定義ではいずれもｕｎｉｆｙしない。

４．Ｑｕｔｅのｓｅｍａｎｔｉｃｓ

ｅｘｐｒｅｓｓｉｏｎの評価はｅｘｐｒｅｓｓｉｏｎの書き換え可能のｓｕｂ－ｅｘｐｒｅｓｓｉｏｎ（ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎと呼ばれる）を次々と書き換えることによって進む。ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎに対して書き換え規則を適用しようとすると

ｒ１）書き換え規則が適用されて別のｓｕｂｅｘｐｒｅｓｓｉｏｎに書き換えられる。

ｒ２）書き換え規則が適用されて書き換えが失敗する。

ｒ３）書き換え規則が適用できない。

のいずれか１つが起こり、ｅｘｐｒｅｓｓｉｏｎの評価はつぎの３つの場合に終了する。

ｉ）ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎがなくなったとき

ii）書き換えに失敗したとき

iii）どのｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎに対しても書き換え規則が適用できないとき

ｉ）ii）iii）の場合をそれぞれｓｕｃｃｅｅｄ，ｆａｉｌ，ｓｕｓｐｅｎｄと呼ぶ。ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎは一般に複数個存在するが、どの順序で書き換えても終了する限り結果は同じである。

以下、ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎの定義と書き換え規則の説明をする。

４．１　ｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎ

ｅｘｐｒｅｓｉｏｎ　ａに対しそのｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓ－ｓｉｏｎの集合Σａを次の様に定義する。

・pが変数、定数またはｆｕｎｃｔｉｏｎのとき　Σｐ≡φ

・Σ｛a，ｂ｝≡　ΣａＵΣｂ

・Σ（a，ｂ）≡　ΣａＵΣｂ

・Σ（a＝ｂ）≡　ΣａＵΣｂ　ｉｆ　Σａ≡φ　ｏｒ　Σｂ≡φ
　　｛｛a＝ｂ｝　ｏｔｈｅｒｗｉｓｅ

・Σ（a；ｂ）≡　Σａ　　ｉｆ　Σａ≡φ
　　Σｂ　　ｉｆ　Σａ≡φ　ａｎｄ　Σｂ≡φ
　　｛a；ｂ｝　ｏｔｈｅｒｗｉｓｅ

・Σ（a　ｂ）≡　｛ΣａＵΣｂ　ｉｆ　Σａ≡φ　ｏｒ　Σｂ≡φ
　　｛｛a　ｂ｝　ｏｔｈｅｒｗｉｓｅ

・Σ（ｉｆ　a，　ｔｈｅｎ　ｂ　ｅｌｓｅ　ｃ）≡
　　｛ｉｆ　ａ　ｔｈｅｎ　ｂ　ｅｌｓｅ　ｃ｝

・Σ（¬a）≡｛¬a｝

Σａが空であることとａがｐａｔｔｅｒｎであることは同値である。従ってｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎの定義から次のことがわかる。

(1) 評価がｓｕｃｃｅｅｄしたとき結果はｐａｔｔｅｒｎ。従ってｐａｔｔｅｒｎはｅｘｐｒｅｓｓｉｏｎの標準形であり、プログラミング言語Ｑｕｔｅにおける‘データ’と見なすことができる。

(2) ｕｎｉｆｉｃａｔｉｏｎの形のｓｕｂｅｘｐｒｅｓｓｉｏｎは＝の左辺と右辺がｐａｔｔｅｒｎになるまで書き換えられた後で書き換えられる。

(3) ｓｅｑｕｅｎｔｉａｌ－ａｎｄの形のｓｕｂｅｘｐｒｅｓｓｉｏｎは、まず；の前のｅｘｐｒｅｓｓｉｏｎがｐａｔｔｅｒｎになるまで書き換えられ、次に；の後のｅｘｐｒｅｓｓｉｏｎがｐａｔｔｅｒｎになるまで書き換えられる。

(4) ａｐｐｌｉｃａｔｉｏｎの形のｓｕｂｅｘｐｒｅｓｓｉｏｎは、関数部と引数部がｐａｔｔｅｒｎになるまで書き換えられた後で書き換えられる。これは関数適用はｃａｌｌ－ｂｙ－ｖａｌｕｅにより計算されることを意味している。

(5) if-then-elseとnegationの形のsubexpression はそのsubexpressionが先に書き換えられることはない。

evaluable subexpressionは次のいずれかの形をしている。(p, qはpattern, a, b, cはexpression)

i) p＝q

ii) p ; q

iii) p q

iv) if a then b else c

v) ¬a

例 次のexpression

x＝1, f (g x, y＝h x), (z＝〔u, v〕; k z)

のevaluable subexpressionは

x＝1 g x h x z＝〔u, v〕

である。

expression aの自由patternに現れるpure変数を新しいpure変数に換え、non-pure変数の#を1つ取ってできたexpressionを、aをrenameしたexpressionという。

例 X＝Y, (λZ. foo (#X, Z, ##W)) Y, #X＝Z

をrenameすると

X0＝Y0, (λZ. foo (#X0, Z, #W)) Y0, X＝Z0

であり

λx. foo (x, #〔y, z〕), baa (X, #〔Y, Z〕)

をrenameすると

λx. foo (x, #〔y0, z0〕), baa (X0, 〔Y, Z〕)

4．2　書き換え規則

ここではｅｖａｌｕａｂｌｅ　ｓｕｂｅｘｐｒｅｓｓｉｏｎがどのような規則に従って書き換えられるかを説明する。前節ｉ）～ｖ）の形のｅｖａｌｕａｂｌｅ　ｓｕｂ－ｅｘｐｒｅｓｓｉｏｎに対してはそれぞれ次の（1）～（5）の規則によって書き換えが試みられる。その際４節の初めで述べた様に３つの場合ｒ１），ｒ２），ｒ３）のいずれかが起こる。以下ｒ１）が起こってｅｘｐｒｅｓｓｉｏｎがＥ１からＥ２に書き換わるとき

Ｅ１　▷　Ｅ２

という記法を用いる。

（1）ｕｎｉｆｉｃａｔｉｏｎ

両辺のｐａｔｔｅｒｎのｕｎｉｆｉｃａｔｉｏｎが行われる。そして

ｒ１）ｕｎｉｆｉｃａｔｉｏｎが成功した時

ｐ＝ｑ　▷　ｐとｑをｕｎｉｆｙしてできたｐａｔｔｅｒｎ

ｒ２）ｕｎｉｆｉｃａｔｉｏｎが失敗した時

となる。ｒ１）の場合ｕｎｉｆｉｃａｔｉｏｎの結果変数に代入が起こるのでｅｘｐｒｅｓｓｉｏｎの他の部分の同じ変数も書き換えられる。

例　・Ｘ＝１，ｆｏｏ　Ｘ　▷　１，ｆｏｏ　１

・Ｘ＝〔Ｙ〕，Ｙ＝１，ｉｆ　＃Ｘ＝〔Ｙ〕ｔｈｅｎ．．．

▷〔Ｙ〕，Ｙ＝１，ｉｆ　＃〔Ｙ〕＝〔Ｙ〕ｔｈｅｎ．．．

▷〔１〕，１，ｉｆ　＃〔１〕＝〔Ｙ〕ｔｈｅｎ．．．

２番目の例で、ｉｆ－ｔｈｅｎ－ｅｌｓｅのｉｆ部のＹと外のＹは違う事に注意。

注）１節であげた論文の定義では変数を書き換えるのではなく、ｅｎｖｉ－ｒｏｎｍｅｎｔに登録する。３節で述べたようにｆｕｎｃｔｉｏｎのｕｎｉｆｉｃａｔｉｏｎにも違いがあるので、ｓｅｍａｔｉｃｓに若干の違いが生ずる。ここでは、例をあげるにとどめる。

（λｘ．λｙ．＃ｘ）１＝（λｘ．λｙ．＃ｘ）１を評価する場合ここでの定義に従うとｓｕｃｃｅｅｄして値はλｙ．１であるが、論文での定義に従うとｆａｉｌする。

(2) sequential-and

r1) 常時

p ; q ▷ q

例 Z = 〔X. Y〕 ; X = 1 ; Y = 2 ; Z

▷ 〔X. Y〕 ; X = 1 ; Y = 2 ; 〔X. Y〕

▷ 〔1. Y〕 ; 1 ; Y = 2 ; 〔1. Y〕

▷ 〔1. 2〕 ; 1 ; 2 ; 〔1. 2〕

▷ 1 ; 2 ; 〔1. 2〕

▷ 2 ; 〔1. 2〕

▷ 〔1. 2〕

(3) application

pqという形のevaluable subexpressionにこの規則を適用しようとする場合

r1) pがfunctionの時

pはλa. bでありa, bをrenameしたexpressionをa', b' であるとすると、

pq ▷ a' = q ; b'

r2) pがfunctionでも変数でもない時

r3) pが変数の時

となる。pが変数の時r3) であるのは、その変数が後にfunctionになるかそれ以外のpatternになるかわからないので、規則の適用を保留するのである。

例 (λx, y. 〔x. y〕) (1, 2)

▷ (x0, y0) = (1, 2) ; 〔x0. y0〕

▷ (1, 2) ; 〔1, 2〕

▷ 〔1. 2〕

(4) if-then-else

evaluable subexpressionはif a then b else c であるとする。if a then * else * の自由変数の列をVとし、a，bをrenameしたexpressionをa'，b'とする。 そしてa'を他の subexpressionとは独立に評価する。即ち、評価中に起こった変数への代入は他のsubexpressionに影響しないとする。a'の評価がsucceedした時その結果起こる代入をσとする。

r1) i) a'の評価がsucceedし、VσがVのvariantである時

if a then b else c ▷ b'σ

ii) a'の評価がfailした時

if a then b else c ▷ c

r2) なし

r3) iii) a'の評価がsucceedし、VσはVのvariantではない時

iv) a'の評価がsuspendした時

iii)の時r3)であるのはσとこのif-then-else以外の部分で後にVになされ得る代入がcompatibleであるかどうかわからないからである。i)の様に他の部分とは無関係に評価が進む時は、書き換えてよい。また ii)の時は、a'の評価がfailするならa'をinstantiateしたexpressionの評価もfailするので書き換えてよい。

例 if部とthen部がlocalなscopeを形成する（2．4節）ので、次のexpressionでXとYは局所変数である。従ってVは空である。

・if 〔1, 2〕 = 〔X, Y〕 then foo (X, Y)
else fail
▷ foo (1, 〔2〕)

次の2つはr3)の例であり、評価はsuspendする

・if #Z = 〔X, Y〕 then. . .

・if #X = #Y then. . .

次のexpressionでZ = 〔1, 2〕とif. . . はいずれもevaluable subexpressionであるが最初は

Z＝〔1，2〕の方のみが書き換え可能である。

・Z＝〔1，2〕，

if　#Z＝〔X，Y〕then　foo（X，Y）

else　fail

▷〔1，2〕，

if　#Z＝〔X，Y〕then　foo（X，Y）

else　fail

▷〔1，2〕，foo（1，〔2〕）

また

・Z＝〔A，A〕，if　#Z＝〔X，X〕then　foo　X

▷〔A，A〕，if　#〔A，A〕＝〔X，X〕then　....

▷〔A，A〕，foo　A

であるが、次のexpは

・Z＝〔A，B〕，if　#Z＝〔X，X〕then　....

▷〔A，B〕，if　#〔A，B〕＝〔X，X〕then　....

となった所でsuspendする。

この様にif－then－elseは自由変数が'十分'にinstantiateされるまで書き換えられないので、一種の同期機構として働く。（Part 2の例を参照）

(5) negation

evaluable subexpressionは¬aであるとする。aの自由変数の列をVとし，aを他のsubexpressionとは独立に評価する。aの評価がsucceedした時その結果起こる代入をσとする。

r1) aの評価がfailした時

¬a ▷（ ）

r2) aの評価がsucceedし、VσはVのvariantである時

r3) aの評価がsucceedし、VσはVのvariantでない時、または

aの評価がsuspendした時

これからわかる様に¬aはif a' then fail else（ ）（但しa'はaの自由変数のうちglobalでないものに#を1つ付けたexpres－

ｓｉｏｎ）で定義できる。

5　ｉｍｐｌｅｍｅｎｔにおける制限

5．1　特殊文字と予約語

ｅｘｐｒｅｓｓｉｏｎの定義で使われている特殊文字（列）は以下の様に普通の文字（列）に対応させる

λ	ｌａｍｂｄａ
ｉｆ	ｉｆ
ｔｈｅｎ	ｔｈｅｎ
ｅｌｓｅ	ｅｌｓｅ
¬	～
ｆａｉｌ	ｆａｉｌ
＜－	＜－

この時ｌａｍｂｄａ，ｉｆ，ｔｈｅｎ，ｅｌｓｅ，ｆａｉｌはｇｌｏｂａｌ変数と区別がつかなくなるので、これらは予約語として扱う。

5．2　ｓｙｍｂｏｌ

ｉｍｐｌｅｍｅｎｔでは（　）と”（　）”，〔　〕と”〔　〕”は同一視される。

5．3　関数定義とｇｌｏｂａｌ変数

ｆａ＜－ｂは2．2節で述べた様にｆ＝（λａ，ｂ）の略であるが、ｉｍｐｌｅ－ｍｅｎｔでは、ｆａ＜－ｂという表記はｔｏｐ　ｌｅｖｅｌに現れかつｆがｇｌｏ－ｂａｌ変数である時のみ許す。即ちこの表記はｕｎｉｆｉｃａｔｉｏｎというよりは一種のコマンドとみなす。ｆがすでに値（ｆｕｎｃｔｉｏｎ）を持っている時でもそれをｃａｎｃｅｌして代入する。

例　ｃｏｎｓ（ｘ，ｙ）＜－〔ｘ，ｙ〕
　　ｃｏｎｓ（1，2）

1行目でｃｏｎｓというｇｌｏｂａｌ変数にλｘ，ｙ．〔ｘ．ｙ〕という値を与え2行目でそれを使う。

さらにこのコマンドによってのみｇｌｏｂａｌ変数に値を与える事ができる。即ちｅｘｐｒｅｓｓｉｏｎの評価中にｇｌｏｂａｌ変数とｕｎｉｆｙできるのは変数のみであり、かつｇｌｏｂａｌ変数同士ｕｎｉｆｙする時は同じｇｌｏｂａｌ変数の時のみ成功する。4節のｓｅｍａｎｔｉｃｓに従えば変数に代入が起こった時は変数はその値で置き換えられるのであるが、ｇｌｏｂａｌ変数については、字づらだけが問題になると思えばよい。

例	ｃｏｎｓ＝ｘ	ｓｕｃｃｅｅｄ
	ｃｏｎｓ＝〔ｘ．ｙ〕	ｆａｉｌ
	ｃｏｎｓ＝ｃｏｎｓ	ｓｕｃｃｅｅｄ
	ｃｏｎｓ＝ａｄｄ	ｆａｉｌ

そしてｇｌｏｂａｌ変数の値は、そのｇｌｏｂａｌ変数がａｐｐｌｉｃａｔｉｏｎの関数部に現れた時のみその値を取り出されａｐｐｌｉｃａｔｉｏｎが計算される。

例　ｃｏｎｓが先の例の定義を持っているとすると

ｆ＝ｃｏｎｓ，ｆ（1，2）

▷ｃｏｎｓ，ｃｏｎｓ（1，2）

▷* ｃｏｎｓ，〔1，2〕

定義を持たないｇｌｏｂａｌ変数をａｐｐｌｙしようとすると4節で述べたｓｅｍａｎｔｉｃｓに従えば書き換え規則を適用できず他の部分を書き換えようとするのであるが、ｉｍｐｌｅｍｅｎｔではｅｒｒｏｒ　ｍｅｓｓａｇｅを出し実行を中断する。しかし、ｎｏｎ－ｇｌｏｂａｌ変数の時は正しくｓｅｍａｎｔｉｃｓに従う。

例　ｃａｒ〔1，2〕．．．．．ｅｒｒｏｒ

Ｃａｒ〔1，2〕．．．．．ｓｕｓｐｅｎｄ

Car〔1, 2〕, Car＝λ〔x, y〕. x

▷* 1, λ〔x, y〕. x

5. 4 unification

implementではoccur checkはしない。しかも（大部分のPrologと同様に）無限patternのunificationを正しくimplementしていないのでunificationが止まらないこともあり得る。

global変数のunificationについては5. 2節で述べた制限がある。

5. 5 suspend

4節で述べた様にif－then－elseには書き換え規則を適用できないことがあり、その判定はif部を評価することによりなされる。従って書き換え規則が適用可能になるまで何度もif部が評価されることになるが、if部の評価に手間がかかる場合何度も繰り返すのは無駄であるし、if部の評価中に副作用のある関数がある場合何度も評価するのは具合が悪い。implementでは、if部を評価した後の状態を保存しておくことより2度目以降規則が適用可能かどうかの判定は現在の状態とその状態を比較することによりなされる。従って評価は一度しかなされない。

negationについても事情は同じである。

例 次のexpressionにおいて

```
. . . , if  write  #X;  #X＝1  then
                "yes"
        else  "no", . . .
```

4節のsemanticsによればこのif－then－elseに規則の適用を試みる度にXの値が出力されるが、implementでは何回適用が試みられるかによらず、1回しか出力されない。

Ｐａｒｔ　２　Ｑｕｔｅの処理系の操作

１　Ｑｕｔｅの操作

１．１　起動

ＱｕｔｅをＤＥＣ　ＴＯＰＳ－２０上で起動するときには、モニタ・コマンドのトップレベルで

```
@ｑｕｔｅ
```

という命令を入力する。するとＱｕｔｅの処理系は以下のようにメッセージとプロンプト‘＞　’を返す。

```
Ｑｕｔｅ　ｖｅｒｓｉｏｎ　１．０
＞
```

この状態でＱｕｔｅ処理系は、Ｑｕｔｅ言語の文の入力を待っている。

また、Ｑｕｔｅ処理系はＤＥＣ－１０　Ｐｒｏｌｏｇで書かれているので、異常が発生すると、ＤＥＣ－１０　Ｐｒｏｌｏｇのトップレベルにもっどてしまうことがある。そのときは、

```
？－ｑｕｔｅ．
```

という述語を入力すると、再びＱｕｔｅのトップレベルを呼び出すことができる。このとき、関数の定義は異常発生以前と変わっていない。

１．２　関数定義

Ｑｕｔｅ処理系は以下の２つの方法による関数定義を受け入れる。

（１）端末から定義式を入力する。

（２）ファイルにある定義式を読み込む。

第一の方法では、例えば、

```
> cons(X, Y) <— [X, Y].
```

と端末から入力することにより関数定義が行われる。
第2の方法では、エディタにより書き込まれたソースプログラムがcons.pというファイルにあるとき、

```
> consult("cons.p").
```

という入力により、処理系はこのファイルからプログラムを読み込み定義する。
繰り返し同じ名前の関数を定義した場合には、最新の定義が残り、前の定義は消去される。
Qute処理系の受理する関数定義の構文は、以下の形に限定される。

```
<関数名><引数> <— <処理>.
```

但し <関数名>はglobal変数、<引数>と<処理>はそれぞれ付録のBNFの<list>と<exp>でなければならない。
ファイル読み込みの場合には関数をcompileすることもできる。それは、

```
> compile("cons.p").
```

という入力により実行することができる。compileされた関数はconsultされたものより数倍早い。
複数のファイルを、consultまたはcompileするときは、ファイル名をリストにして

```
> consult({"cons.p", "etc.p", ...}).
> compile({"cons.p", "etc.p", ...}).
```

と入力する。ファイルに実行文が含まれている時、ｃｏｎｓｕｌｔすればそれは実行され、ｃｏｍｐｉｌｅすれば無視される。

１．３　実行

Ｑｕｔｅ処理系における実行とは、ｅｘｐｒｅｓｓｉｏｎの評価の事である。例えば、先のｃｏｎｓという関数の場合

```
＞　ｃｏｎｓ（1，2）．
```

と入力すると以下のような結果が出力される。

```
ｓｕｃｃ：〔1．2〕
```

この様に、ｅｘｐｒｅｓｓｉｏｎを入力し、．（ピリオド）で終わると、そのｅｘｐｒｅｓｓｉｏｎが評価され結果が出力される。評価が終わったときの状態は、　Ｐａｒｔ１で述べたように３つの場合があり、次のように表示される。

ｓｕｃｃ：　　最終的にｐａｔｔｅｒｎが得られた場合
ｆａｉｌ：　　書き換えが失敗した場合
ｓｕｓｐｅｎｄ：　書き換えがこれ以上できない場合

さらに、ｓｕｃｃｅｅｄしたときは結果のｐａｔｔｅｒｎも出力される。

１．４　エラー

シンタックス・エラー

シンタックス・エラーのある場合、ＤＥＣ－１０　Ｐｒｏｌｏｇと同様に、パーザがエラーを見つけた場所を示して来る。

例

```
＞ｃｏｎｓ（1，2．
```

```
******SYNTAX ERROR******
cons (1, 2
***here***
```

その他のエラー

実行時のエラーは理論上はないが、Ｐａｒｔ１　５．３節で述べたように定義されていない関数を呼んだときにはＰｒｏｌｏｇのデバグモードに入り、定義されていない関数名を表示する。その関数呼出は、ｆａｉｌしたとして、さらに評価は続行される。トップレベルに戻ると、Ｐｒｏｌｏｇのデバッグモードから抜ける。

例
```
> append ([1, 2], [3]).
% Unknown procedure:Sappend/5
% Failing back・・・
Debug mode switched on
fail:
Debug mode switched off
>
```

この例では、評価はすぐにｆａｉｌするが、例えばｉｆ－ｔｈｅｎ－ｅｌｓｅのｉｆ部で未定義関数が呼ばれたときはｆａｉｌして、ｅｌｓｅ部の評価に移るので、Ｐｒｏｌｏｇのデバッガへの命令を与えなければならない。この様なときは、‘a’（ａｂｏｒｔ）と入力して評価を放棄し、Ｐｒｏｌｏｇのトップレベルに戻って、１．１で述べた方法で回復するとよい。

１・５　終了

Ｑｕｔｅの処理系からモニタ・コマンドのレベルに出るためには、ｅｘｉｔという関数に任意の引数を与えて評価すればよい。また、Ｐｒｏｌｏｇのトップレベルに出るためには、プロンプトが表示され入力待ちの状態になっているときに

、Ｃ－Ｚを入力するかａｂｏｒｔという組み込み関数に任意の引数を与えて評価すればよい。

１．６　ログの参照

ログはＰｒｏｌｏｇのログファイルにためられる。すなわち、

ｐｒｏｌｏｇ．ｌｏｇ

がログの入っているファイルである。

２．　デバッグ

２．１　デバッグのための基本的コマンド

Ｑｕｔｅ処理系にはＤＥＣ１０－Ｐｒｏｌｏｇのような逐次実行して停止するトレースモードはない。デバッグのために或る関数にトレーサを付けたいときは組み込み関数ｔｒａｃｅを用いて

ｔｒａｃｅ　＜関数名＞

を評価すればよい。（システム関数を除く任意の関数の名前に対してユーザはトレーサをつけることができる。）このトレーサはキーボードからのコントロールなしに関数の呼び出しと結果を内部処理の順番どおりに出力する。対応する呼び出しと結果は同じ番号のヘッダーが順次つけられる。

また、複数の関数にトレーサをつけたいときには、リストを使って

ｔｒａｃｅ　〔＜関数名＞，．．．〕

という形で、評価すればよい。

トレーサは関数名に対してつけられるので、同じ関数名で新たな関数を定義した時にもトレーサは残る。ある関数のトレーサを解除したいときには、組み込み関数ｎｏｔｒａｃｅを用いて、

notrace　<関数名>

という関数適用を評価すればよい。また複数の関数のトレーサをはずすときは、

notrace　〔<関数名>, . . . 〕

という形が使える。すべてのトレーサをはずすときには引数を（ ）として、

notrace ( ).

とすればよい。トレーサの付いた関数が呼びだされたときは、

n>　<関数名>　<引数>

succeedした時は

n<　<関数名>　<引数>　　=　　<値>

failした時は

n<　<関数名>　fail:

と出力される。ただし、nは1つの呼出し毎にに割り当てられる番号であり、これによりどの呼出しと終了が対応するかがわかる。

Quteでは、関数の本体を評価中suspendし、他の関数の本体の評価終了後再開することもあるので、呼出と終了は例えばPrologにおけるように入れ子にはならない。なお、評価中に起こるsuspendに関しては、トレーサによる出力はなされない。

2．2　デバッガの使用例

つぎは、ａｐｐｅｎｄをおこなう関数のトレースの実行例である。

```
> app (X, Y)  <-
>          if  #X  =  *  then  #Y
>          else  X  =  〔X1, X2〕;
>                    〔X1, app (X2, Y)〕.
function <app> is defined
> trace (app).
FUNCTION <app> HAS TRACER
succ:  app
> app (〔1, 2〕, 〔3〕).
0> app  〔1, 2〕, 〔3〕
1> app  〔2〕, 〔3〕
2> app  *, 〔3〕
2< app  *, 〔3〕        =  〔3〕
1< app  〔2〕, 〔3〕        =  〔2, 3〕
0< app  〔1, 2〕, 〔3〕        =  〔1, 2, 3〕
succ:  〔1, 2, 3〕
>
```

3．組み込み関数

Ｑｕｔｅ処理系のｖｅｒｓｉｏｎ 1.0 に組み込まれている関数には、以下のようなものがある。

3．1　算術演算

四則の演算子については中置記法のみが許され、その強さは一般的なものと同じである。（他のｅｘｐｒｅｓｓｉｏｎとの関係については、付録のＢＮＦを参照）

```
X + Y          ;;  XとYの和を返す
X - Y          ;;  XとYの差を返す
X @ Y          ;;  XとYの積を返す
X / Y          ;;  XとYの商を返す
mod (X, Y)     ;;  XのYによる剰余を返す
gt (X, Y)      ;;  X>Yならば ( ) を返し、それ以外のとき
               ;;  failする
lt (X, Y)      ;;  X<Yならば ( ) を返し、それ以外のとき
               ;;  failする
ge (X, Y)      ;;  X>=Yならば ( ) を返し、それ以外の
               ;;  ときfailする
le (X, Y)      ;;  X<=Yならば ( ) を返し、それ以外の
               ;;  ときfailする
```

以上の関数は、XまたはYが整数でも変数でもないときにfailし、その他のときは、suspendする。

3．2　入出力

```
read ( )
write (<expression>)
nl ( )
consult     ( <ファイル名> )
          | ( <ファイル名リスト> )
compile     ( <ファイル名> )
          | ( <ファイル名リスト> )
<ファイル名>      :=  <string>
<ファイル名リスト>  ::=  〔 <ファイル名>, ・・・〕
```

3．3　コマンド

```
listing    （ <関数名> ）
           |（ ）；；　定義式のリスティング
trace      （ <関数名> ）
           |（ <関数名リスト> ）　；；　トレーサを付ける
notrace    （ <関数名> ）
           |（ <関数名リスト> ）
           |（ ）　　；；　トレーサを解除する
<関数名リスト>　：：＝　〔 <関数名>，・・・〕

timer      （ "on" ）
           |（ "off" ）；；　タイマールーチンのｏｎとｏｆｆ
reset　（ ）　；；　すべての関数定義，トレーサ，タイマーを消す
abort　（ ）　；；　Ｐｒｏｌｏｇのモードになる
exit　（ ）　；；　モニタ・レベルになる
```

注）　|はｏｒ，；；は注釈をあらわす

4. プログラム例

今回、新しい言語であるＱｕｔｅを理解するときの参考にもなるように、Ｑｕｔｅのプログラム例として、つぎの七つのプログラムを作成した。

- ｅｉｇｈｔ　ｑｕｅｅｎｓ
- ｑｕｉｃｋ　ｓｏｒｔ（ｓｅｑｕｅｎｔｉａｌ，ｐａｒａｌｌｅｌ）
- Ｅｒａｓｔｏｓｔｈｅｎｅｓ'　ｓｉｅｖｅ
- ｍｉｓｓｉｏｎａｒｉｅｓ　ａｎｄ　ｃａｎｎｉｂａｌｓ
- ｍａｇｉｃ　ｓｅｒｉｅｓ
- ｈａｍｍｉｎｇ　ｓｅｑｕｅｎｃｅ
- ｅｘｐａｎｓｉｏｎ　ｏｆ　ｆｕｎｃｔｉｏｎ

それぞれのプログラムとその実行結果を、以下で説明する。

(i) e i g h t　q u e e n s

チェスの盤面に並べられた8個のｑｕｅｅｎが、互いにチェックさていないようにするパズルである。プログラムは、1～8までの整数の順列を作りながら、その順列をｑｕｅｅｎの配置にみたてて、互いにチェックされているかどうかを調べている。

結果

```
> queen(8).

succ: [5, 7, 2, 6, 3, 1, 4, 8]
```

プログラム

```
% **** Eight Queens ****

queen(N) <- SO = generate(N); try(N,SO,1,SO,[],[],[]).

generate(N) <-
        if #N = 0 then
                []
        else
                [N.generate(N-1)].

try(C,SR,SC,SO,Rs,CpRs,CmRs) <-
        if #C = 0 then
                #Rs
        else if #SR = [] then
                fail
        else
                SR = [R.SR1];
                if try1(#C,#R,#SC,#SO,#Rs,#CpRs,#CmRs) = 0 then
                        0
                else
                        try(C,SR1,SC+1,SO,Rs,CpRs,CmRs).

try1(C,R,SC,SO,Rs,CpRs,CmRs) <-
        CpR = C+R, notmem(CpR,CpRs),
        CmR = C-R, notmem(CmR,CmRs),
        SO1 = delete(SO,SC);
        try(C-1,SO1,1,SO1,[R.Rs],[CpR.CpRs],[CmR.CmRs]).

delete([A.L],K) <-
        if #K = 1 then
                #L
        else
                [A.delete(L,K-1)].

notmem(A,L) <-
        if #L = [] then
                ()
        else
                L = [B.L1];
                if ~(#A = #B) then
                        notmem(#A,#L1)
                else
                        fail.
```

(ii) quick sort

quick sort の方法でリストの整列を行うプログラムである。部分リストの整列を同時に行うものと、そうでないものと二種類作成した。

結果

```
> qsort([8, 7, 6, 5, 4, 3, 2, 1]).
succ: [1, 2, 3, 4, 5, 6, 7, 8]
```

プログラム

```
% Quick Sort (parallel sort)

qsort(X) <- qsort1(X, R, []); R.

qsort1(L, R, R0) <-
        if #L = [X . L0] then
                partition(L0, X, L1, L2);
                qsort1(L2, R1, #R0), qsort1(L1, #R, [X . R1])
        else
                R = R0.

partition(L, X, L1, L2) <-
        if #L = [X0 . L0] then
                if le(#X0, ##X) then
                        ##L1 = [#X0 . L1];
                        partition(#L0, ##X, L1, ##L2)
                else
                        #L2 = [X0 . L2];
                        partition(L0, #X, #L1, L2)
        else
                L1 = L2 = [].
```

QSORT2.Q.1

```
% Quick Sort (sequential sort)

qsort(X) <- qsort1(X, []).

qsort1(L, R0) <-
        if #L = [X . L0] then
                (L1, L2) = partition(L0, X);
                qsort1(L1, [X . qsort1(L2, #R0)])
        else
                R0.

partition(L, X) <-
        if #L = [X0 . L0] then
                (L1, L2) = partition(L0, #X);
                if le(#X0, ##X) then
                        ([#X0 . #L1], #L2)
                else
                        (L1, [X0 . L2])
        else
                ([], []).
```

(iii) Erastosthenes' sieve

素数を昇順に求めて行くプログラムである。整数を1から順に作るプログラムと、前に得られた素数でふるいにかけるプログラムが、同時にはしる。

結果

```
>primes ( ).
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
```

プログラム

```
% Eratosthenes' sieve

primes() <- integers(2, i), sift(i, J), outstream(J).

integers(x, [x . l]) <- integers(x+1, l).

sift(i, J) <-
        if $i = [p . I] then
                $J = [p . J]; filter(I, p, r), sift(r, J)
        else
                fail.

filter(i, p, r) <-
        if $i = [n . I] then
                if mod($n, $$p) = 0 then
                        filter($I, $$p, $$r)
                else
                        $r = [n . R]; filter(I, $p, R)
        else
                fail.

outstream(l) <-
        if $l = [x . L] then
                write(x); nl(); outstream(L)
        else
                fail.
```

(iv) m i s s i o n a i e s　a n d　c a n n i b a l s

それぞれ三人の宣教師と人喰い人種を、一隻のボートで川の向う岸まで渡す有名なパズルである。ボートは、三人乗りで、ボート、川岸どちらでも、宣教師の数が人喰い人種の数よりも少なくなってはいけない。

結果

```
> m_c ( ).
〔〔"M". 3〕, 〔"C". 3〕, "Boat"〕
〔〔"M". 3〕, 〔"C". 1〕〕
〔〔"M". 3〕, 〔"C". 2〕, "Boat"〕
〔〔"M". 3〕, 〔"C". 0〕〕
〔〔"M". 3〕, 〔"C". 1〕, "Boat"〕
〔〔"M". 1〕, 〔"C". 1〕〕
〔〔"M". 2〕, 〔"C". 2〕, "Boat"〕
〔〔"M". 0〕, 〔"C". 2〕〕
〔〔"M". 0〕, 〔"C". 3〕, "Boat"〕
〔〔"M". 0〕, 〔"C". 1〕〕
〔〔"M". 0〕, 〔"C". 2〕, "Boat"〕
〔〔"M". 0〕, 〔"C". 0〕〕
succ:  ( )
```

注) 〔"M", N〕は、宣教師がN人川のこちら側にいることをあらわす。

〔"C", N〕は、人喰い人種がN人川のこちら側にいることをあらわす。

"BOAT"は、ボートが川のこちら側にいることをあらわす。

プログラム

```
% MISSIONARIES AND CANNIBALS

m_c() <-
        Statelist =
        search([["M".3],["C".3],"Boat"],
               [["M".0],["C".0]],
               [[["M".0],["C".2]],
                [["M".1],["C".1]],
                [["M".2],["C".0]],
                [["M".0],["C".1]],
                [["M".1],["C".0]]],
               [[["M".3],["C".3],"Boat"]]);
        revoutstream(Statelist).

% search(leftbank state,rightbank state, move list)

search(Leftbank,Rightbank,Movelist,Statelist) <-
        Allmoves = [[["M".0],["C".2]],
                    [["M".1],["C".1]],
                    [["M".2],["C".0]],
                    [["M".0],["C".1]],
                    [["M".1],["C".0]]];
        if #Leftbank = [["M".0],["C".0]] then #Statelist
        else if #Movelist = * then fail
        else if boatat(#Leftbank) then
                if [New1.Newr] =
                   move(##Leftbank,##Rightbank,car(##Movelist));
                   newstate(New1,##Statelist) then
                   search(New1,Newr,##Allmoves,[New1.##Statelist])
                else
                   search(#Leftbank,#Rightbank,cdr(#Movelist),
                          #Statelist)
        else if [Newr.New1] =
                move(#Rightbank,#Leftbank,car(#Movelist)) ;
                newstate(New1,#Statelist) then
                search(New1,Newr,#Allmoves,[New1.#Statelist])
        else search(Leftbank,Rightbank,cdr(Movelist),Statelist).

boatat(Bank) <-
        if #Bank = [X,Y,"Boat"] then ()
        else fail.

move(From,To,Passengers) <-
        [depart(From,Passengers).arrive(To,Passengers)].

depart(Bank,Passengers) <-
        Bank = [["M".Bm],["C".Bc],"Boat"];
        Passengers = [["M".Pm],["C".Pc]];
        Xm = Bm - Pm; Xc = Bc - Pc;
        if lt(#Xm,0) then fail
        else if lt(#Xc,0) then fail
        else if gt(#Xm,0) then
                ge(#Xm,#Xc); [["M".#Xm],["C".#Xc]]
        else [["M".Xm],["C".Xc]].
```

```
arrive(Bank,Passengers) <-
        Bank = [["M".Bm],["C".Bc]];
        Passengers = [["M".Pm],["C".Pc]];
        Xm = Bm + Pm; Xc = Bc + Pc;
        if gt(#Xm,0) then
           ge(#Xm,#Xc); [["M".#Xm],["C".#Xc],'Boat']
        else [["M".Xm],["C".Xc],'Boat'].

newstate(X,Y) <-
        if #Y = * then ()
        else
                Y = [Y1.Yrest];
                if #X = #Y1 then fail
                else newstate(X,Yrest).

car([X.Y]) <- X.

cdr([X.Y]) <- Y.

revoutstream(List) <-
        if #List = * then ()
        else  List = [H.T];revoutstream(T);write(H);nl().
```

(v) magic series

与えられた正の整数が、奇数ならば三倍して一を加え、偶数ならば二で割る、という操作を繰り返して、一が得られたら止まるプログラムである。次の数が得られるとすぐに出力するようにプログラムはつくられている。これは、concurrent prolog のストリーム通信に相当する。

結果

```
> magic_series (321).
321  964  482  241  724  362  181

544  272  136  68  34  17  52  26

13  40  20  10  5  16  8  4  2  1  1
```

注意：実際の出力は一つずつ縦に並んで出力される。

プログラム

```
%MAGIC SERIES

magic_series(X) <-
        magic_series1(X,Y),outstream(Y);().

magic_series1(X,T) <-
        if #X = 1 then
                #T = [1]
        else if even(#X) then
                Z = #X / 2;
                #T = [#X.Tail];
                magic_series1(Z,Tail)
        else
                Z = X @ 3 + 1;
                magic_series1(Z,Tail), T = [X.Tail].

even(X) <-
        if mod(#X,2) = 0 then
                ()
        else
                fail.

outstream L <-
        if #L = [X . L1] then
                write X; nl(); outstream L1
        else if L = * then
                ()
        else
                fail.
```

(vi) h a m m i n g　s e q u e n c e

$2^l * 3^m * 5^n$　（l，m，nは0または正の整数）

で表される整数を昇順に出力するプログラムである。この数列とそれを各々二、三、五倍した数列を同時に作って行くことにより、実行が行われる。

プログラムで作られる数列

S	数列	1	2	3	4	5	6	8・・・
S2	二倍	2	4	6	8	10	12	16・・・
S3	三倍	3	6	9	12	15	18	24・・・
S5	五倍	5	10	15	20	25	30	40・・・

S，S2，S3，S5は、プログラム中の変数名

結果

```
> h a m m i n g ( ) .
  2  3  4  5  6  8  9  10  12  15  16  18

  20  24  25  27  30  32  36  40  45  48

  45  48  50  54  60  64  72  75  80  81

  96  ・・・
```

注意：実際の出力は一つずつ縦に並んで出力される。

プログラム

```
% Hamming sequence

hamming() <- ham S, outstream S.

ham S <-
        S = [1 . ST];
        smul(S, 2, S2), smul(S, 3, S3), smul(S, 5, S5),
        select(S2, S3, S5, ST).

smul(S, N, MS) <-
        if #S = [X . ST] then
                #MS = [#N@X . MST],
                smul(ST, #N, MST)
        else
                fail.

select(S2, S3, S5, S) <-
        if #S2 = [M2 . S2T], #S3 = [M3 . S3T], #S5 = [M5 . S5T] then
                #S = [Min = min(M2, M3, M5) . ST];
                select(news(Min, M2, S2T),
                       news(Min, M3, S3T),
                       news(Min, M5, S5T),
                       ST)
        else
                fail.

min(X, Y, Z) <- min2(min2(X, Y), Z).

min2(X, Y) <-
        if le(#X, #Y) then
                #X
        else
                Y.

news(Min, M, S) <-
        if lt(#Min, #M) then
                [#M . #S]
        else
                S.
```

```
outstream(l) <-
        if #l = [x . L] then
                write(x); nl(); outstream(L)
        else
                fail.
```

(vii) expansion of function

関数の展開の実行例として、次の二つの例を用意した。

和の展開

〔1，2，3〕という整数のリストの、和を求める。このリストが任意のものであっても、このプログラムは、和を求めることができる。これは、expandという関数定義が、関数を帰納的に展開しているからである。

合成関数の定義

car（cdr（cdr（〔1，2，3，4〕））を求める。ここで、Compseqとある関数は、関数のリストから合成関数を作る。

結果

succ：6

succ：3

プログラム

```
expand(Fun, Init) <-
        lambda Args.(
                if #Args = * then ##Init
                else (Args = [Fst . Rst];
                      #Fun(Fst, (expand(#Fun, #Init))(Rst))
                )
        ).


Plus = (lambda x, y . (x + y));
Sum = expand(Plus, 0);
Sum [1, 2, 3].

Comp = (lambda f, g. (lambda x. #f(#g x)));
Car = (lambda [x . y]. x);
Cdr = (lambda [x . y]. y);
Compseq = expand(Comp, (lambda x. x));
Caddr = Compseq [Car, Cdr, Cdr];
Caddr [1, 2, 3, 4].
```

索引

T

U

Y

＜特殊記号＞

付録 A QuteのBNF

BNF of Qute Syntax February 1985

```
<toplevel>        ::= <definition> .
                  |   <exp> .

<definition>      ::= <globalvar> <list> <- <exp>

<exp>             ::= <seqand>
<seqand>          ::= <and>
                  |   <and> ; <seqand>
<and>             ::= <ifthenelse>
                  |   <ifthenelse> , <and>
<ifthenelse>      ::= <unification>
                  |   if <exp> then <exp> else <exp>
<unification>     ::= <negation>
                  |   <negation> = <unification>
<negation>        ::= <function>
                  |   ~ <negation>
<function>        ::= <binop1>
                  |   lambda <exp> . <exp>
<binop1>          ::= <binop2>
                  |   <binop2> + <binop1>
                  |   <binop2> - <binop1>
<binop2>          ::= <unop>
                  |   <unop> @ <binop2>
                  |   <unop> / <binop2>
<unop>            ::= <application>
                  |   + <unop>
                  |   - <unop>
<application>     ::= <list>
                  |   <application> <list>
<list>            ::= <block>
                  |   [ <listseq> <listtail> ]
                  |   []
                  |   *
<listseq>         ::=
                  |   <unification>
                  |   <unification> , <listseq>
<listtail>        ::=
                  |   . <unification>
<block>           ::= <group>
                  |   < <exp> >
<group>           ::= <symbol>
                  |   <variable>
                  |   ( <exp> )
                  |   fail
<symbol>          ::= <integer>
                  |   <string>
                  |   ()
<variable>        ::= <nonglobalvar>
                  |   <globalvar>
<nonglobalvar>    ::= <purevar>
                  |   # <nonglobalvar>
<string>          ::= "<character string>"  ;; " is able to appear
                      only with \
<globalvar>       ;; a lowercase alphabet and characters without blank
<purevar>         ;; an uppercase alphabet and characters without blank
                  ;; or a lowercase alphabet and integer without blank
                  ;; or a lowercase alphabet
<characters>      ;; a list of non-blanc printable characters
```