ICOT Technical Memorandum: TM-0242

TM-0242

CIL言語マニュアル

大沼敏幸(三菱電機)
向井国昭，奥西稔幸

November, 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ＣＩＬ言語マニュアル

初版　第１刷　昭和61年 7月17日
第２刷　昭和61年11月14日

向井国昭　奥西稔幸
（財団法人　新世代コンピュータ技術開発機構）

天沼敏幸
（三菱電機東部コンピュータシステム株式会社）

本マニュアルは、現在のバージョンに関する仕様をまとめたものです。
CIL言語は、発展途上／実験中の言語であり、今後断りなく仕様が変更されることがありますのでご了承ください。

目　次

Ⅰ．言語の概要

本言語は、自然言語処理にあらわれる複雑な構造を持つデータおよびそれらの間の制約関係を、より柔軟に記述するために、Prologを拡張した言語である。本言語により、GPSG/LFG等の構文論およびsituation semantics 等の意味論をはじめとする自然言語処理システムが容易に記述できる。

［特徴］
CILは論理型言語Prologをベースとした言語であり，その実行メカニズムは単一化(unification)と後戻り(backtracking)が中心となる。

・単一化(unification)

・後戻り(backtracking)

CILは新たに次の２つの基本機能を導入した。

・部分項(partially specified term)

・遅延実行制御(lazy evaluation)

これらの基本機能をもとに幾つかの有用なシンタックス・シュガーを標準マクロとして提供する。

・シンタックス・シュガー(syntax-sugar)

［概要］
本章以下では基本機能の概要を述べる。

Ⅰ．1　部分項

自然言語処理をはじめとする大規模なプログラムを記述しようとする場合，複雑なデータ構造を表現する機能が必要となる。自然言語処理では談話構造、知識等がその対象となる。
Prologでは複合項(compound term)を用いることで構造体を表現することを可能とする。しかし複合項は、
・引数の位置順序とその意味（役割）は記述者が管理する必要がある、
・いつもすべての引数を書かなければならない、
などの欠点をもち、意味ネットワーク等の複雑なデータを表現・操作をするにはけっして十分とは言えない。

ここで部分項という概念を導入する。

部分項の概念は下に掲げるような計算機言語やデータベース，文法理論等，計算機工学の他の分野で広く見られるデータ構造からの抽象化と見ることができる。

1) 一階述語論理の項
2) データベースにおけるレコード
3) 属性/属性値対リスト
4) Lispのプロパティリスト(property list)
5) C.Pollardのanadic relation (状況意味論の１つのヴァージョン)
6) GPSGのカテゴリ
7) LFGの機能構造(functional structure)
8) Barwiseの状況理論における割り当て(assignment)や事態(state of affair)
9) Minskyのフレーム(frame)

部分項の記述は引数位置名と引数の対（引数位置名-引数対と呼ぶ）の順序のないリストで行う。具体的には次のような形で記述される。

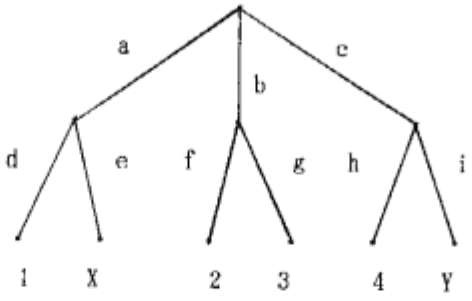
{ a1/b1, a2/b2, ... , an/bn }　　n ≧ 0, ai ≠ aj (i ≠ j)

ここでaiは引数位置名を表し、biは引数を表す。 ai/biの出現順序には意味がない。aiには通常の（変数を含まない）項が、biには部分項も含めたCILの全てのデータ型が記述できる。これにより知識をそれが持つ属性の名前を用いて表すことが可能になる。もしbiに自分自身を記述すれば，自己参照を含むデータを表現・操作することができ、最近の状況理論(situation theory)の数学的性質に対応できる。

X = {a / X}.

実装面から眺めると部分項は次図のようなラベル付けされた順序なし木(labelled unordered tree)のことである。

{a/ {d/1, e/X},
 b/ {f/2, g/3},
 c/ {h/4, i/Y}}.



部分項の名前は項の必要な部分のみを指定する記述法という面を強調したものである。
部分項に対するデータのアクセスは引数位置名aiを用いて行う。そのための述語やマクロをCILでは準備する。

部分項の導入に伴い，単一化はこの新しい構造を持つ対象に対して拡張されている。

更に，部分項を意味ネットワーク(semantic network)と考えた場合に,簡単な 照合操作(pattern matching)を可能にする述語も用意している。

部分項およびそれに関する組み込み述語の詳細な仕様についてはII,IIIで述べる。

１．２　遅延実行制御

CILの遅延実行制御のオリジナルはProlog-Ⅱ(Colmerauer)のフリーズ制御である。PSiでもバインドフック(bind_hook)と呼ばれる述語で提供される。
これらは「ある変数に対し，その変数が具体化された時点で，特定の述語(デモン)を実行するように指定できる機能」を持つ述語である。

遅延実行制御を用いると次のようなプログラミング（"Generate and Test"手法）で効率を上げることができる。

generate(X)で次々と生成されるXをprocess(X)で処理する。ただしXが意図されたものかどうかはtest(X)で調べる。これを実現するには下のゴール列ではいかにも効率が悪い。

```
..... generate(X), process(X), test(X), .....
```

このような時は遅延実行制御を用いると次のように記述することができる。

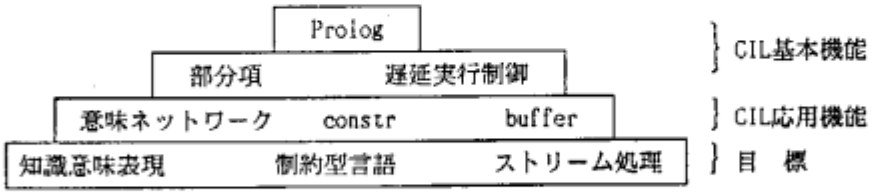
```
..... test(X?), generate(X), process(X), ....
```

ここでtest(X?)は遅延実行制御のCILマクロ表記で「変数Xに対してデモンとして述語testを登録する」という意味をもつ。generateでXが生成されるたびにtestが起動され正しいXだけがprocessにわたるので効率は改善される。testをgenerate中などの適当な位置に埋め込めば同様の効率が得られるが、それでは記述者がtestすべき位置を管理する必要が生じ大きな負担となる。
したがって遅延実行制御は実行順序を意識せずに、いわゆる宣言的にプログラムを記述するための１つの方法を与えていることになる。上の場合ではtest(X?)は「Xはtestを満足する」という宣言であると考えられる。

CILでは制約評価のための述語constrを持つ。これは構文処理、意味処理等のモジュールの宣言的記述を高めるのに使うことができる。

また遅延実行制御を用いたストリームプログラミングは数個のモジュールが密接に関連しながら動作する、いわゆるコルーチン処理プログラムの記述には有効な機能を果たす。

最後にCILの目指すところと現状を図にして本章の締めとする。



［本マニュアルに関する注意］

1．説明範囲

CILはDEC-10 Prologと逐次型推論マシンPSI上で実現されている。本マニュアルはPSI上の処理系を対象としているが，PSI/SIMPOSおよびそのシステム記述言語ESP，機械語KL0，更にCILのベースになっているPrologについては既知の知識としてここでは述べられていない。本マニュアルではCILの基本機能に関する事項のみ記述している。

2．部分項に関する用語

簡単のため引数位置名をラベル，引数を値，引数位置名-引数対をラベル-値対と呼ぶこともある。

II．言語仕様・文法

本章ではCILを用いたプログラミングの際に必要とされる文法・仕様などについて述べる。最初にCILプログラムの基本的なシンタックスがPrologと同じであることを述べる。次にCILの基本機能である部分項と部分項を対象にした拡張された単一化について述べる。さらに遅延実行制御およびそれらから組み立てられる制約のシンタックスを述べる。最後にCILの標準マクロを紹介する。

II．1　節／述語／データ型

CILプログラムはホーン節の集りからなり、各節の基本的なシンタックスはPrologと同じである。述語定義の方法もPrologと同じである。現在のところ、プログラム記述の面からは、クラス等のESPシンタックスは取り入れておらず、ESPのメソッド、クラス等はCIL処理系が管理する。（「付録-II　ESP/KLOとの関連」参照）

データ型に関しては、ESPで許されているデータ（アトム、スタックベクタ、ヒープベクタ、ストリング等）も使用可能である。これらと部分項をあわせてCILデータ型とする。

II．2　部分項／拡張単一化

II．2．1　部分項

［文法］

〈部分項〉　　::=　"{"〈引数位置名-引数対リスト〉"}"
〈引数位置名-引数対リスト〉::=　〈引数位置名-引数対〉　{ "," 〈引数位置名-引数対〉 }
〈引数位置名-引数対〉　::=　〈引数位置名〉"/"〈引数〉

〈引数位置名〉　::=　〈基礎項〉
〈引数〉　　::=　〈一般項〉

〈一般項〉　　::=　〈通常項〉　|　〈部分項〉
〈通常項〉　　::=　〈〈ESPデータ型〉〉
〈基礎項〉　　::=　〈〈変数を含まない通常項〉〉

［例］　{ a/1, f(1)/X, c/{d/g(Y)}, e/{d/h(Z)} }

```
{ 関係 / 愛する,
  主格 / { 名前/ 健,
           年齢/ 25 },
  対格 / { 名前/ 奈緒美,
           年齢/ 24 } }
```

［注意］同一レベルの同一部分項内には同じ引数位置名を持つ矛盾した引数位置名-引数対リストが存在してはいけない

X = { a/1, b/2, a/2 }.　のような記述は動作を保証しない。

II．2．2　拡張単一化

部分項の導入に伴い単一化の仕様は次のように拡張されている。
この拡張単一化の実質的意味は”マージ（併合）”である。

●部分項が関与しない単一化（通常項同士の単一化）

KLOのunify述語が処理する。（詳細仕様は『KLO組込述語説明書』を参照）

```
f(X) = f(a)              ==>    X = a.
g(b, a) = g(a, b)               失敗
```

●部分項と通常項の単一化

①部分項と未定義変数の単一化

単一化はつねに成功する。未定義変数は相手の部分項と同じになる。

```
h(X) = h({x/a, y/b})     ==>    X = {x/a, y/b}
```

②部分項と未定義変数以外の単一化

単一化はつねに失敗する。引数に変化はない。

```
{f/p, a1/x, a2/y} = p(x,y)      失敗
```

●部分項同士の単一化

①２つの部分項ともに存在するラベルに関しては，ラベルに対応するそれぞれの値の単一化（この単一化もここで述べている仕様に従って行われる）を行う。

```
{a/{c/1}, b/X} = {b/2, a/Y}     ==>    X = 2, Y = {c/1}
{a/{b/X}} = {a/{b/2}}           ==>    X = 2
{a/1} = {a/2}                          失敗
```

②一方の部分項にしか存在しないラベルに関しては，そのラベルと値からなる対を新たに他方の部分項に追加する。

```
X = {a/1}, Y = {b/1}, X = Y             ==> X = Y = {a/1, b/1}
X = {x/1, y/1}, Y = {z/1, x/1}, X = Y   ==> X = Y = {x/1, y/1, z/1}
```

II．3　遅延実行制御／制約

II．3．1　遅延実行制御

CILの遅延実行制御はPSIのbind_hookを用いて実現される。bind_hookの仕様は次のとおりである。

```
... bind_hook(X, demon(X)), ...
```

のような記述は、X に値が入った時点でdemon(X)を実行するように指定しているものである。

```
... bind_hook(X, demon(X)), ... p(X, Y), ...

=>  p(a, Y) :- demon(a), ...
                  └──デモンの起動がこの時点に割込む。
```

もしbind_hook実行時に、既にX が具体化されていればデモンは即座に実行される。

```
... X = a, .. bind_hook(X, demon(X)), ...

=>  X = a, .. bind_hook(X, demon(X)), demon(a), ..
                                        └── デモンが即座に実行される。
```

同じ変数に複数回フックをかけると、その変数が具体化されたときには両方が起動される。例えば、

```
... bind_hook(X, p(X)), bind_hook(X, q(X)), ...
```

というゴール列は、

```
... bind_hook(X, (p(X), q(X))), ...
```

と同じである。（ただし、起動される順序は保証しない）
またフックのかかった変数同士を単一化すると、その結果の変数には両方のフックがかかった状態になる。例えば

```
... bind_hook(X, p(X)), bind_hook(Y, q(Y)),X = Y, ...
```

というゴール列は

```
... bind_hook(X, p(X)), bind_hook(X, q(X)),X = Y, ...
```

と同じことになる。

Ⅱ．３．２　制約

constrの第１引数である制約の文法を以下に示す。

```
〈制約〉 ::=〈基本制約〉
    | "not(" 〈制約〉 ")"
    | 〈順接続子〉 "(" 〈制約〉 "," 〈制約〉 ")"
    | 〈制約〉〈接続子〉〈制約〉
    | 〈DEFCONヘッド部〉
    | 〈IF述語〉
    | 〈ASSIGN述語〉
```

〈基本制約〉 ::= 〈論理制約〉 | 〈原子制約〉 | 〈計算制約〉
〈論理制約〉 ::= "true" | "false" | 〈変数〉
〈原子制約〉 ::= 〈項〉 "=" 〈項〉 | 〈項〉 "\==" 〈項〉
〈項〉 ::= 〈数〉 | 〈変数〉 | 〈アトム〉 | 〈計算式〉
〈計算制約〉 ::= 〈計算式〉〈計算関係式〉〈計算式〉
〈計算関係式〉 ::= "=:=" | "=\=" | "<" | ">"
〈計算式〉 ::= 〈数〉 | 〈変数〉 | 〈計算式〉〈演算子〉〈計算式〉
〈演算子〉 ::= "+" | "-" | "*" | "/" | "mod"

〈順接続子〉 ::= "exor" | "nor" | "nand" | "and" | "or" | "imply"
〈接続子〉 ::= "," | ";" | "->" | "<->"

〈IF述語〉 ::= "if(" 〈制約〉, 〈制約〉, 〈制約〉 ")"
〈ASSIGN述語〉 ::= "assign(" 〈項〉 , 〈項〉 ")"
〈DEFCONヘッド部〉 ::= 〈通常項〉

[例]　　constr(``(X!refl = (+) -> X!gr = sbj)).

注：``(..)は、ESPマクロ抑制記法である。
　　詳細は「付録-II　ESP/KLOとの関連」参照

II. 4　シンタックス・シュガー

CILでは以下の諸記法が使用できる。

（1）　引数指示記法
「部分項Pの引数位置名Lに置かれている引数」をP!Lと書く。
P!Lは組込述語role(L,P,V)に展開され、Vの値を持つ。

```
... f(X!a), ...
```
は
```
... role(a, X, V), f(V), ...
```

に展開される。（述語roleの仕様についてはIII. 6「部分項」参照）

[例1] {a/1, b/X}!b = 3. を実行すると、X = 3 となる。
[例2] X = {a/1, b/2}, X!b = Y. を実行すると、Y = 2 となる。
[例3] X = {a/1, b/2}, X!c = 3. を実行すると、X = {a/1, b/2, c/3} となる。

（2）　条件付項
「条件Cを満たすX」をX:Cと書く。C には述語が記述できる。項記述となっている。

[例]　T = X:father_of(X, 太郎).
　　これは「太郎の父（であるX）をT と単一化しなさい」
　　という意味である。

（３） 遅延実行制御記法

「引数が未定義の場合、具体化されるまで実行を中断する述語中の変数」をX?と書く。
print(X?)はbind_hook(X, print(X))に対するマクロ記法である。

［例］ print(X?), X = OK. を実行すると OK が表示される。
print(X?). だけでは、X は未定義のままで何も表示されない。

（４） ＃記法

X#Yは、X:(X=Y) の略記である。XはYと同格の項であることを示す。

［例］ X# {a/1, b/X!a} = Y. を実行すると、
X = Y = {a/1, b/1} となる。

（５） @記法 （遅延実行型の条件付項）

@pは、X:p(X?) の略記であり、X@pはX:p(X?) の略記である。

［例］ X = {a/b}, Y = {a / @print}, X = Y. を実行すると、
b が表示され、X = Y = {a/b} となる。

この記法と部分項を用いることで知識表現でいう継承関係を表すことができる。

［例］ man(@animal@clever). は
man(X) :- animal(X), clever(X). とほぼ同じである。

（６）??記法

X??は、T:(T?=X)の略記である。

［例］ X = a(Y??) はfreeze(T, (T=Y)), X=a(T) と同じである。
これを実行するとY の状態にかかわらずX は具体化する。
しかし、X = a(Y?) はfreeze(Y, X=a(Y)) と同じのため、
Y が未定義の場合、X は具体化しない。

III. 組込み述語

CIL処理系では、以下の機能のための組込み述語を提供している。

1．単一化
2．等価性
3．複写
4．部分項
　4．1　基本述語
　4．2　照合操作
5．制御
6．制約
　6．1　制約の述語
　6．2　基本制約
　6．3　制約の結合
　6．4　IF述語
　6．5　ASSIGN述語
　6．6　DEFCON述語
7．論理演算
8．計算
9．Prolog／入出力
10．その他

CIL処理系では、KLO／ESPの述語／マクロも使用できる。以上のCIL組込み述語だけで不足している機能についてはそれらを用いて実現できる。

［注意］
1．この章における単一化はCIL用の拡張された単一化のことである。
2．遅延実行引数とは、その引数が変数であるかまたは変数を含んでいる場合には、その変数が具体化されるまで述語の実行が中断されるような性質を持つ引数の事である。

Ⅲ．1　単一化

unify_cil(X,Y)　　［マクロ記法］X = Y

X とY を単一化する。

［例］ {a/P, b/2} = {b/Q, a/1} を実行すると P = 1, Q = 2 となる。
X = {f/1}, Y = {g/2}, X = Y を実行すると X = Y = {f/1, g/2} となる。

［注］ Ⅱ．2 で述べたCILの単一化のアルゴリズムはこの単一化述語を用いてのみ実現される。現在のところ，ヘッド内の引数間の単一化（ヘッド内ユニフィケーション）ではCILの単一化は行われない。従って部分項を含むような構造体をヘッド内ユニフィケーションする場合は，それらの単一化は上の述語を用いてボディに展開しておく必要がある。

```
p(X, X) :- !, q, r ....
```

なる述語pに対しては

```
p({a/1}, {b/2})
```

は正しく動作しない。pの定義を次のように変える必要がある。

```
p(X, Y) :- X = Y , !, q, r ....
```

assign(X,Y,T)

X とY を一方向の単一化を行う。unify_cil/2 と次の３点において異なる。
① X に含まれる変数への束縛はサスペンドされる。
② 単一化が成功すればTはtrue、失敗すればfalseとなり、
いずれの場合も述語の実行は成功する。
③ X，Yには部分項は含まれてはいけない。

［例］ assign(f(X), f(Y), T). を実行すると、T = true, X = Y であるが値は持たない。
assign(f(X), f(a), T). を実行すると、T, X は値を持たない。
assign(f(a), f(X), T). を実行すると、T = true, X = a となる。

Ⅲ．2 等価性

same(X,Y)

X とY が等しいかどうか調べる。
X とY の単一化はされない。

［例］ same({a/1, b/X}, {b/2, a/1}). は失敗する。

X = Y, same(X, Y) は成功する。

dif(X,Y)

X とY が等しくないかどうか調べる。
ただし，X，Yは遅延実行引数である。

[例] dif(X,Y), X = 1, Y = 2. は成功する。
dif(X,1), print(ok), X = 1. は ok を表示後、失敗する。

equal(X,Y,C,R,_)

XとYの等価性を調べる。制御値Cの指定により、次の３つの動きをする。
① C が未定義の場合、XとYが等しいかどうか調べる。
ただし、X、Yは遅延実行引数である。
XとYが等しければ、C、Rともtrueとなり、述語は成功する。
XとYが等しくなければ、C、Rともfalseとなり、述語は成功する。
② C = false と指定した場合，XとYが等しくないかどうか調べる。
但し、X、Yは遅延実行引数である。
XとYが等しくなければ、Rはfalseとなり、述語は成功する。
XとYが等しければ、述語は失敗する。
③ C = true と指定した場合、XとYを単一化する。
単一化が失敗すれば、述語は失敗する。
単一化が成功すれば、Rはtrueとなり，述語は成功する。
引数C，Rの役割については「Ⅲ．６　制約」でも述べる。

[例] equal(X, Y, false, R, _), X = 1, Y = 4. の結果、
R = false, X = 1, Y = 4 となる。

[注] 述語equalの第５引数はシステムが用いる制御値で、この値がセットされればその時にサスペンドされている述語は全てキャンセルされる。特にユーザが意識する必要はない。組込み述語で引数が'_'になっているのは全て同じ働きをする述語である。

t_equal(X,Y,C,R,_)

equal/5 とほぼ同機能であるが、次の点で異なる。X とY は数である。また、X とY は遅延実行に関して対象でない。X が具体化されるまで中断するが、Xが具体化されればY が未定義でもX とY が単一化される。

[例] t_equal(X,Y,C,R,_), X = 1. の結果、
X = 1, Y = 1, C = true, R = true となる。

Ⅲ．３　複写

fullCopy(X,Y)

Y にX を複写する。ただし、遅延実行の制御（変数にかかったデモン）は複写されない。

[例] X = {a/ @print, b/X}, fullCopy(X, Y), Y!b!a = ok.
を実行しても、ok は表示されない。

Ⅲ．4　部分項

この節での説明では、P は部分項、L は引数位置名（ラベル）、V は引数（値）を示す。

Ⅲ．4．1　基本述語

partial(P)

Pが部分項ならば成功する。

[例] partial({a/1, b/X}). は成功する。

role(P,L,V)　　[マクロ表記] P!L

部分項P の引数位置名L に対応する引数をV と単一化する。

[注] 「Ⅱ．2 拡張単一化」のアルゴリズムに従って、P 中にL というラベルを持つラベルー値対がなければ，値をVとする対を新たにP に追加する。

[例] X = {a/1, b/2}, role(L, X, 3), L = c. を実行すると、
X = {a/1, b/2, c/3} となる。

locate(P,L,V)

部分項P の引数位置名L に対応する引数をV と単一化する。
P中にLという引数位置名を持つ対がなければ、述語は失敗する。

[例] locate({a/1, b/2}), b, V). を実行すると、V = 2 となる。
locate({a/1, b/2}), c, V). を実行すると、失敗する。

getRole(P,L,V)

部分項PTの引数位置名、引数をそれぞれL,Vと単一化する。
再実行させることでPのすべての引数位置名、引数をつぎつぎとL,Vに単一化する。

[例] getRole({a/1, b/2}, L, V). を実行すると、
L = a, V = 1 ;
L = b, V = 2　となる。

setOfKeys(P,S)

部分項P中に含まれるすべての引数位置名からなるリスト（集合）をSと単一化する。

[例] setOfKeys({a/1, b/2}, S). を実行すると、
S = {a,b}. となる。

record(P,R)

部分項Pをリスト（レコード）に変換し、Rと単一化する。

[例] record({a/1, b/2}, R). を実行すると、
R = [(a,1), (b,2)] となる。

buffer(P,B)

リストB の長さだけ部分項P をリスト（バッファ）に変換し、Bと単一化する。
P の中の対がB の長さより少なければ、最後にend がはいる。
B が変数の間は実行は中断される。

[例] buffer({a/1, b/3}, B), B = {}. を実行すると、B = {} となる。
buffer({a/1, b/3}, [X,Y,Z,W]). を実行すると、
X = (a,1), Y = (b,3), Z = end となり、W は未定義となる。

Ⅲ．４．２　照合操作

この節では照合操作のための幾つかの述語を述べる。部分項を意味ネットワークとして利用する時、有効になる機能である。

glue(P1,P2)

P1、P2に共通の引数位置名をもつ引数同士の単一化を行う。P1、P2とも対の拡張はされずに共通名を持つ引数の単一化が行われるだけである。

[例] P1 = {a/1, b/2}, P2 = {a/X, c/3}, glue(P1, P2) を実行すると、
X = 1, P1 = {a/1, b/2}, P2 = {a/1, c/3} となる。

merge(P1,P2)

P1をP2に併合し、結果をP2に単一化する。
P1はglueと、P2は単一化と同様の機能を持つ。
すなわち併合の結果、P2の対が拡張されるのに対し、P1は拡張されずにP2と共通の引数位置名をもつ引数の単一化が行われるだけである。

[例] P1 = {a/1, b/X}, P2 = {b/2, c/3}, merge(P1, P2). を実行すると、
X = 2, P1 = {a/1, b/2}, P2 = {a/1, b/2, c/3} となる。

t_merge(P1,P2)

mergeと同様の機能を持つが，次の２点で異なる。
① P1はP2と共通の引数位置名をもつ引数の単一化も行われない。
(P1は全く変化しない。)
② 引数が部分項であれば再帰的にt_mergeを適用する。

[例] t_merge({b/1, a/{c/2, b/3}}, {a/{b/Y}}). を実行すると、
Y = 3 となる。

d_merge(P1,P2)

mergeと同様の機能を持つが，次の１点で異なる。
① 単一化できない引数が存在しても、述語は失敗せずにその対がそのまま残る。

[例] d_merge(P1#{a/1, b/2}, P2#{a/X, b/3}). を実行すると、
X = 1, P1 = {a/1, b/2}, P2 = {a/1, b/3}. となる。

delete(L,P1,P2)

Lというラベルの対をP1から削除し（なければそのまま）、P2に併合する。
P1, P2の機能はmerge と同様である。

[例] delete(a, {a/1, b/2}, P). を実行すると、
P = {b/2} となる。

masked_merge(P1,P2,P3)

P1からP2を削除し（なければそのまま）、P3に併合する。

[例] masked_merge({a/1, b/1, c/1}, {a/_, b/_}, U#{a/2}). を実行すると、
U = {a/2, c/1} となる。

subpat(P1,P2,D)

P1がP2のサブパターンである。すなわち，P1の引数位置名の集合がP2の引数位置名の集合の部分集合である。（引数の等価性は問わない）
P1の引数位置名とそれぞれ対応するP1, P2の引数の三つ組のリスト（集合）が D に差分リストの形式で返る。

[例] P1 = {b/1, c/X}, P2 = {a/2, b/Y, c/2, d/2}, subpat(P1, P2, D). を実行すると
D = {(b,1,Y), (c,X,2)|Z}-Z となる。

t_subpat(P1,P2)

P1がP2のサブパターンである。
引数が部分項であれば再帰的にt_subpatを適用する。

［例］t_subpat({a/{b/Y}}, {b/1, a/{c/2, b/3}}). は成功する。
t_subpat({a/{b/Y, c/U}}, {b/1, a/{a/2, b/3}}). は失敗する。

extend(P1,P2,D)

P1がP2のサブパターンになるようにP2が拡張される。
D はsubpatと同様。

［例］P1 = {a/1}, P2 = {b/2}, extend(P1, P2, D). を実行すると、
P1 = {a/1}, P2 = {a/X, b/2}, D = [(a,1,X)|Y]-Y となる。

meet(P1,P2,D)

P1とP2の共通パターンを作る。すなわち，
P1とP2の共通の引数位置名について，差分リストDを作る。

［例］meet({a/1, b/2}, {b/3, c/4}, `(X-[])). を実行すると、
X = [(b,2,3)] となる。

frontier(X,Y,D)

XとYが単一化可能であるとき，そのための最低条件となる等式の集合を差分リストの形式でDに提示する。単一化不可能ならば、述語は失敗する。
但し、X、Y は部分項であってはいけない。

［例］frontier(f(a,g(b)), f(A,B), `(L-[])). を実行すると、
L = [(a=A), (g(b)=B)] となる。
frontier(a, b, `(L-[])). を実行すると失敗する。

match(X,Y,D)

XとYが単一化できるための最低条件である等式の集合を差分リストの形式でDに提示する。(frontierと異なりXとYは単一化不可能でもよい)
但し、X、Y は部分項であってはいけない。

［例］match(a, b, `(L-[])). を実行すると、
L = [(a=b)] となる。

Ⅲ．5　制御

以下の説明では、Gはゴール（述語）で、T は真理値（trueまたはfalse）を示す。

freeze(X,G)　　[KLO] bind_hook(X,G)

X が未定義の場合、具体化されるまでG は中断する。
X が具体化されていれば、即座にG を実行する。

freeze(X,Y,G)

X とY がどちらもが未定義の場合、どちらかが具体化されるまで、G は中断する。
X とY どちらか又は両方ともが具体化されていれば、即座にG を実行する。

if(T,G)

T がtrueまたは未定義ならば、G を実行する。それ以外ならば何もせず成功する。
Gの失敗により、述語自身も失敗する。

if(T,G1,G2)

T がtrueまたは未定義ならば、G1を実行し、それ以外はG2を実行する。
G1（またはG2）の失敗により、述語自身も失敗する。

ifBound(X,G)

X が具体化されていれば、G を実行する。未定義ならば、何もせず成功する。
Gの失敗により、述語自身も失敗する。

ifUnbound(X,G)

X が未定義ならば、G を実行する。具体化されていれば、何もせず成功する。
Gの失敗により、述語自身も失敗する。

wif(T,G)

T が未定義の場合、具体化されるまで、T の評価を中断し、
T がtrueに具体化されれば、Gを実行する。それ以外の値を持てば何もせず成功する。
Gの失敗により、述語自身も失敗する。

wif(T,G1, G2)

T が未定義の場合、具体化されるまで、T の評価を中断し、
T がtrueに具体化されれば、G1を実行する。それ以外の値を持てばG2を実行する。
G1（またはG2）の失敗により、述語自身も失敗する。

when(X,G)

X の構造体中の変数のすべてが具体化するまで、G を中断する。

ただし、X は部分項であってはならない。

pv(F, SuspG)

Fが未定義ならば，F を1 にしてSuspGを実行する。
Fが定義済ならば，SuspGは実行しない。
pvは複数個の遅延実行制御のどれか１つだけを実行すればよいという時に用いる。

solve(G)

Gを実行するいわゆるメタコールである。
Gに記述できるのは，CIL述語，KLO述語，CILユーザ定義述語(CIL変換系でコンパイルされた述語）である。

［例］solve(print(ok)) を実行すると ok が表示される。

Ⅲ．６　制約

Ⅲ．６．１　制約の述語

constr(X,C,R)

X は制約として、「Ⅱ．３．２ 制約」で示した文法で記述できる。
C は制約の制御値、Rは制約の結果であり、ともにtrueかfalseの値をとる。

C が未定義ならば制約X は遅延実行される。
C がtrueに具体化された時、制約の中の変数は制約Xがtrueになるように具体化される。しかし、非決定的な制約中の変数は具体化されない。
C がfalseに具体化された時も、制約Xがfalseになるように制約X中の変数が決まるならば、具体化される。

［例］constr(`(X=1), true, _). を実行すると、X = 1 となる。
constr(`(X=1), false, _). を実行すると、X は未定義のままである。
constr(`(X=1), _, _). を実行すると、X は未定義のままである。
constr(`(X\==2), false, U). を実行すると、X = 2, U は未定義のままである。

引数の異なる述語も用意されており、その内容は以下のとおりである。

```
constr(X) :- T? = true, constr(X, true, T).
constr(X, Y) :- ocnstr(X, true, Y).
```

Ⅲ．６．２　基本制約

（１）論理制約

constr(true)は成功し、constr(false)は失敗する。
constr(X) の場合、Xはtrueかfalseの値をとる。
Xが未定義の場合、具体化されるまで中断する。

［例］constr(X), X = true. は成功する。

（２）原始制約

項１= 項２ は項１と項２が等しいことを示す。
項１\== 項２ は項１と項２が等しくないことを示す。

［例］constr(`(X=1)). を実行すると、X = 1 となる。

（３）計算制約

計算関係子の意味は、以下のとおりである。

=:= 等しい、 =\= 等しくない
〈 小さい、 〉 大きい

［例］constr(``(1+X=:=Y),_), Y = 4. を実行すると、X = 3, Y = 4 となる。
constr(``(1+X=\=Y)), X = 3, Y = 4. を実行すると失敗する。

Ⅲ．６．３ 制約の結合

（１）否定

not(制約)は、制約の否定を示す。

（２）接続子

, は論理積、; は論理和、-〉は包含、〈-〉は必要十分を示す。
１つめの制約が具体化していなくても、２つめの制約の評価を行う。

（３）順接続子

and は積、orは和、implyは包含、exorは排他論理和、norは和の否定、
nandは積の否定を示す。
１つめの制約が具体化していなければ、２つめの制約の評価は中断される。

［例］constr(and(X,Y)). を実行すると、X = true, Y = true となる。
constr(and(X,Y),false,Z). を実行すると、X, Y, Z は未定義となる。

Ⅲ．６．４ IF述語

if(X,Y,Z)でXが成立すればYを評価し、成立しなければY を評価する。
X,Y,Z は制約である。

［例］constr(``(if(P=1, Q=1, R=1))), P=1, Q=1, R=2. を実行すると成功する。

Ⅲ．６．５ ASSIGN述語

assign(X,Y) は片方向の単一化を行う。

［例］constr(assign(X,Y)), X = 1. を実行すると、X = 1, Y = 1 となる。

Ⅲ．６．６　DEFCON述語

ユーザがconstr述語の外で定義した制約を起動するのに用いる。
defcon(DEFCONヘッド部、制約). の形式で定義した制約は、
DEFCONヘッド部を記述することで用いることができる。

［例］リストのメンバを調べる述語の定義。
defcon(mem(X,Y), (and(assign(Y, [H|T]), (`(X=H);mem(X,T))).と定義。
constr(mem(X,[Y,Z], true, A), Y = 1, Z = [U,B], X = 2, B = 3.
を実行すると、U = 2, A = true, X = 2, B = 3, Y = 1, Z = [2,3] となる。

Ⅲ．７　論理演算

ここで説明する論理演算に表われる変数は、trueまたはfalseである。その決定は遅延実行制御の対象となる。すなわち、論理関係を満たす変数が２組以上ある時は、その可能性が１つになるまで変数への値の束縛はサスペンドされる。明らかに１つしかない時は即座に束縛される。変数が関係を満たしていなかったり、関係を満たす変数が１つもなかったりした場合は失敗する。

not(X,Y)

X はY の否定である。

and_cil(X,Y,Z)

Z はX とY の積である。

andx(X,Y,Z)

X とY の積がtrueならば成功し、falseならば失敗する。

or_cil(X,Y,Z)

Z はX とY の和である。

orx(X,Y,Z)

X とY の和がtrueならば成功し、falseならば失敗する。

exor(X,Y,Z)

Z はX とY の排他的論理和である。

nand(X,Y,Z)

Z はX とY の積の否定である。

nor(X,Y,Z)

Z はX とY の和の否定である。

imply(X,Y,Z)

Z はX とY の包含である。

iff(X,Y,Z)

Z はX とY の必要十分である。

［例］and_cil(X, Y, true). を実行すると、X = Y = true となる。
or_cil(X, Y, true). を実行すると、X、Y の値はまだ決まらない。
or_cil(X, false, true). を実行すると、X = true となる。

Ⅲ. 8 計算

ここでの説明では、X,Y,Z は数であり、C,R はtrueかfalseである。X,Y は計算の入力であり、Z は計算結果である。C は計算結果が正しいかどうかの制御値であり、R はその結果である。いずれの値も遅延実行制御の対象となる。すなわち、C にtrueが具体化するとX,Y,Z が四則関係を満たすように計算を行い、関係を満たす変数の組が２つ以上あれば値の束縛はサスペンドされ、１つになった時に値が決まる。 C がfalseの時は、X,Y,Z が四則関係を満たさないように計算を行う。

add(X,Y,Z,C,R,_)

Z はX とY の和である。

multiply(X,Y,Z,C,R,_)

Z はX とY の積である。

mod(X,Y,Z,C,R,_)

Z はX とY の法である。

evaluate(E,V,C,R,_)

E はX+Y, X-Y, X*Y, X/Y, X mod Y の形のいずれかをもち、Vはその計算結果である。

X とY に関しては遅延実行制御されないため、必要ならばX?+Y?の記法を使う。

exp(E,V,C,R,_)

E はX+Y, X-Y, X*Y, X/Y, X mod Y の形のいずれかをもち、Vはその計算結果である。
X とY に関しては、遅延実行制御される。

Ⅲ. 9 Prolog／入出力

CIL処理系では、以下のDEC-10 Prolog仕様にほぼ準拠した述語を提供している。

statistics(runtime, X)

var(X)

nonvar(X)

\+(X)

name(X,Y)

また、以下のDEC-10 Prolog仕様準拠の入出力述語も用意している。
ただし、入出力ポートはCILランタイムウィンドウを対象にしている。

read(X)

get(X)

print(X)

write(X)

nl

tab(X)

SIMPOSのメソッドを用いて、ファイルやウィンドウなどのオブジェクトを生成して
入出力を行うことも出来る。

Ⅲ. 10 その他

pp(X)

X をCILランタイムウィンドウにプリティ・プリントする。

ppf(X,F)

X をファイルF にプリティ・プリントする。
ファイル名F はアトムまたはストリングで指定。

ppwf(X,F)

X をCILランタイムウィンドウとファイルF にプリティ・プリントする。

genSym(X,Y)

アトムX をプレフィックスとするシンボルをY につくる。
副作用をもち、２回目以降の別のシンボルを作る。

［謝辞］

本報告を纏める機会を与えてくださった横井俊夫第２研究室室長に感謝いたします。また日頃、有益な御討論を頂いたICOT自然言語処理グループの三吉秀夫、田中裕一、瀧塚孝志、杉村領一、木村和広、赤坂宏二の各氏に感謝いたします。

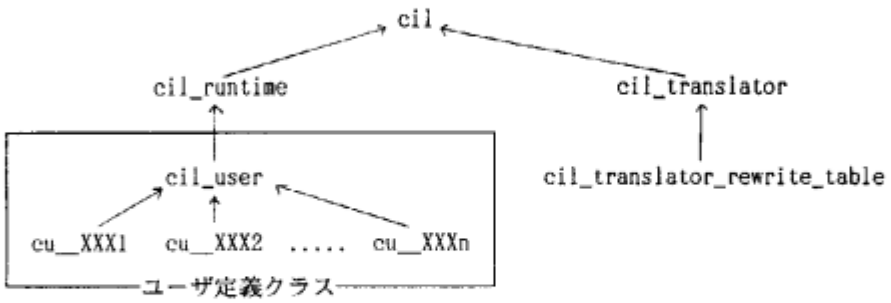
［参考文献］

向井国昭：自然言語処理のための単一化の拡張と遅延的実行制御、ICOT-TR 215，1986.

付録－Ｉ　ＣＩＬ処理系クラス構成

PSI上のCIL処理系は次のESPクラスを以下に示す。

●実行系／変換系



① cilクラス

cilのトップレベルクラスで、cilシステム全体の起動を行う。その際、
ランタイムウィンドウの生成など、必要な初期設定を行う。
さらにゴールの入力や結果の表示などのモニタ機能を実現している。
（ただし、現在のところトレース機能は独自に持たない）

② cil_runtimeクラス

CILの組込み述語を構成するクラス。述語呼出しのためのエントリとしては
全てクラスメソッドである。

③ cil_translatorクラス

CILプログラムをESPに変換するクラス。具体的には、CILマクロの展開を行う。
ESPに変換されたプログラムはSIMPOSのコンパイラで実行形式になる。

④ cil_translator_rewrite_tableクラス

③で用いるクラス。CILプログラムをESPに変換する際に組込み述語を
クラスメソッドに変換するためのテーブル。

⑤ ユーザ定義クラス（cil_userクラスおよびcu__XXXiクラス）

ユーザが記述したCILプログラムがESPに変換される際に生成されるクラス。
１ファイルが１クラスに対応し、クラス名は、CILプログラムファイル名より
与えられる。ファイル名がXXXiであれば、cu__XXXiクラスとなる。
cu__XXX1, cu__XXX2, ... cu__XXXnクラスを継承したクラスcil_userクラス
により、cil_runtimeクラスのリンクをとる。
CILのプログラムをコンパイルする（述語t,tr,tcを実行する）たびに、
これらのユーザ定義クラスが新たに生成される。

●入出力系

以下のクラスのインスタンスオブジェクト(ファイル,ウィンドウ)はCILのオペレータを

受付ける。これらのインスタンスオブジェクトに対するメソッドとしてはSIMPOS標準入出力述語が全て使える。

⑥ cil_window
　　ウィンドウクラス

⑦ cil_input_file
　　読み込みだけができるファイル用のクラス

⑧ cil_output_file
　　書き込みだけができるファイル用のクラス

付録－ロ　ESP/KLOとの関連

以下はCILプログラムを記述する場合に、ESP/KLO機能を用いたい時の注意事項である。

（1）マクロ抑制記号

ESPマクロ抑制　` または ``　（`はバッククオート）
CILマクロ抑制　$

これらは次のようなときに用いる

a. CILマクロと衝突しているESPマクロを(ESPマクロとして)用いる場合，$を用いる。

現在，'!'，':'(infixのみ)，'#'(iinfixのみ) が衝突している。

'!'	スロットアクセス	スロットアクセス
':'	条件付き項	クラス指定メソッドコール
'#'	同格項	文字コード，文字列

〔例〕　W = $(Obj!window), :putc(W, $(key#tab))

b. 述語constrで 制約の記述に =, ==,), <, +, -, *, / のESPマクロ対象記号を用いる場合，` または `` を用いる。
〔例〕　constr(`(A=B),...) .

c. マクロ記号を普通の関数名として用いる時。

〔例〕 sentence(A, B, '(S0-S1))　　/* d-リストとして用いる */

〔注〕 {}はESPのベクタ記法であって，ESPマクロでないので，$ で囲っても
CILの理想リストの意味しかもたない。

（2）他ファイルの述語の呼出し

他のCILファイルで定義した述語を用いている場合は，それを呼び出すメソッドコールを，用いる側のファイルに追加する必要がある。

〔例〕 ファイルtest1.cil内の述語が,ファイルtest2.cilで定義されている
foo/3を呼び出している場合，ファイルtest1.cil内の任意の場所に

```
foo(X, Y, Z) :- :foo(#cu__test2, X, Y, Z).
```

と，定義しておく必要がある。
(もちろん，foo/3を直接書き直してもOK)

〔注〕ESPプログラムとして作成した述語（メソッド）をCILプログラムから使うことができる。

（3）オペレータ宣言

オペレータはファイルの先頭（述語の定義のの前）に，ESPと同様の記法で宣言できる。
但し，現在のところCILプログラムではinclude文は使えない。

```
add_operator(Name, Type, Prec)   オペレータの追加
remove_operator(Name)            オペレータの削除
```

〔注〕DEC-10Prologの　:- op(Prec, Type, Name)　はadd_operatorと
見なされ処理される。

付録－III　プログラム例

```
%
%       例題 1 ：状況を用いた談話の解釈のモデル
%
example(Interpretation):-
        discourse_constraint(
              {{sit/ {soa(speaking,  (jack, _), yes),
                      soa(addressing,(betty, _),yes)|_],
                exp/ [i,love,you],
                dl/  loc(1)]@discourse_situation,
               {exp/ [i,love,you]}@discourse_situation],
              Interpretation).

   % sit: 状況(situation)
   % soa: 事態(state of affair)
   % exp: 表現(expression)
   % dl : 談話時空間(discourse location)

% 談話状況(DISCOURSE SITUATION)の定義
discourse_situation({sit/S, sp/I, hr/ You, dl/ Here, exp/ Exp}):-
        member(soa(speaking, (I, Here),yes),S),      % sp :  話者
        member(soa(addressing,(You, Here),yes),S),   % hr :  聴者
        member(soa(utter,(Exp, Here),yes), S).

% メンバの定義
member(X, [X|Y]).
member(X,[Y|Z]):-member(X,Z).


% 談話制約(DISCOURSE CONSTRAINT)の定義
discourse_constraint([],[]):-!.
discourse_constraint([X],[Y]):-!,meaning(X,Y).
        % Y はX の解釈である
discourse_constraint([X,Y|Z], [Mx,My|R]):-
        meaning(X,Mx),          % Mx はX の解釈である
        change_role(X, Y),      % 話者〈―〉聴者
        time_precedent(X, Y), % X はY より時間的に先行している
        discourse_constraint([Y|Z],[My|R]).

change_role({hr/X,sp/Y},{hr/Y,sp/X}@discourse_situation).

time_precedent({dl/loc(X)},{dl/loc(Y)}):-
        constr(``(X+1=:=Y)).    %  遅延評価のための組み込み述語
```

```
% 文法 - DCG(DEFINITE CLAUSE GRAMMAR) -
meaning(X#{exp/E},Y):-sentence(`(E-[]),{ip/Y,ds/X}). % ip  : 解釈

sentence(`(A-B), {ip/SOA,ds/DS}):-
        noun(`(A-A1),  {ip/Ag,ds/ DS}),
        verb(`(A1-A2), {ip/SOA, ds/DS, ag/Ag, obj/ Obj}), % ag : 行為者
        noun(`(A2-B),  {ip/Obj, ds/ DS}).                 % obj: 対象者

% 語彙項目
noun(`({jack|Q]-Q), {ip/jack}).          % Jack
noun(`({betty|Q]-Q),{ip/betty}).         % Betty
noun(`({i|Q]-Q),    {ip/X, ds/{sp/X}}).  % I
noun(`({you|Q]-Q),  {ip/X, ds/{hr/X}}).  % You
verb(`({love|Q]-Q), {ip/ soa(love,(X, Y, Loc), yes), % love
                 ds/ {dl/Loc},
                 ag/ X,
                 obj/Y}).

% 実行結果

% 同じ文が談話状況(discourse situations)によって
% いろいろな解釈(interpretation)を持つ

\ example(Interpretation).

Interpretation =>
    [soa(love,(jack,betty,loc(1)),yes),    % 文1   "I love you."
     soa(love,(betty,jack,loc(2)),yes)]    % 文2   "I love you."
```

```
%
%       例題  2
%
:-public tomorrow/2.
tomorrow({year/ Y1, month/ M1, week/ W1, day/ D1},
         {year/ Y2, month/ M2, week/ W2, day/ D2}):-
        orderInWeek(W1,W2),
        endOfMonth(Y1,M1,L1),
        bump(D1,L1,D2,Cm),
        monthOfTomorrow(Cm,M1,M2,Cy),
        constr(``(Y1+Cy=:=Y2)).

leapYear(X,V):-constr(``((X-1984) mod 4 =:= 0),_,V).

endOfMonth(Y,M,L):-wif(``(M=\=february), defaultEnd(M,L), endOfMonth1(Y,L)).

endOfMonth1(Y,L):-leapYear(Y,V), wif(V, ``(L=29), ``(L=28)).

:-public bump/4.
:-public bump1/4.
bump(D1,L,D2,C):-
        pv(X, bump1(D1,L,D2?,C)),
        pv(X, bump1(D1?,L?,D2,C)),
        pv(X, bump1(D1?,L,D2,C?)),
        pv(X, bump1(D1,L?,D2,C?)).

bump1(X,X,1,1):-!.
bump1(X,Y,Z,0):-!, constr(``(X @( Y, X+1=:=Z)).

:-public monthOfTomorrow/4.
monthOfTomorrow(C,M1,M2,Cy):-
        pv(X, monthOfTomorrow1(C,M1?,M2?,Cy)),
        pv(X, monthOfTomorrow1(C?,M1?,M2,Cy)),
        pv(X, monthOfTomorrow1(C?,M1,M2?,Cy)),
        pv(X, monthOfTomorrow1(C?,M1,M2,Cy?)).

:-public monthOfTomorrow1/4.
monthOfTomorrow1(0,X,X,0):-!.
monthOfTomorrow1(1,december,january,1):-!.
monthOfTomorrow1(1,X,Y,0):-!,orderMonth(X,Y).

:-public defaultEnd/2.
defaultEnd(january,  31).
defaultEnd(february, 28).
defaultEnd(march,    31).
defaultEnd(april,    30).
```

```
defaultEnd(may,       31).
defaultEnd(june,      30).
defaultEnd(july,      31).
defaultEnd(august,    31).
defaultEnd(september,30).
defaultEnd(october,   31).
defaultEnd(november,  30).
defaultEnd(december,  31).

:-public orderMonth/2.
orderMonth(january,    february).
orderMonth(february,   march).
orderMonth(march,      april).
orderMonth(april,      may).
orderMonth(may,        june),
orderMonth(june,       july).
orderMonth(july,       august).
orderMonth(august,     september).
orderMonth(september,  october).
orderMonth(october,    november).
orderMonth(november,   december).
orderMonth(december,   january).


:-public orderWeek/2.

orderInWeek(X, Y):-freeze(X,Y, orderInWeek1(X,Y)).

:-public orderWeek1/2.
orderInWeek1(sunday,    monday    ).
orderInWeek1(monday,    tuesday   ).
orderInWeek1(tuesday,   wednesday).
orderInWeek1(wednesday,thursday ).
orderInWeek1(thursday,  friday    ).
orderInWeek1(friday,    saturday  ).
orderInWeek1(saturday,  sunday    ).
```

```
%
%       例題  3
%
:-public send/2.
send(M,T):- statistics(runtime,_),send(M),statistics(runtime,T).
send([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]):-
        different([S,E,N,D,M,O,R,Y]) ,
        M? \== 0 , S? \== 0 ,
        sum(R1,0,0,M,0),
        sum(R2,S,M,O,R1),
        sum(R3,E,O,N,R2),
        sum(R4,N,R,E,R3),
        sum(O ,D,E,Y,R4).

remainder(1).   remainder(0).

digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).

different([]).
different([X|Y]):-out_of(X,Y),different(Y).

out_of(X,[]).
out_of(X,[A|L]):-X? \== A?, out_of(X, L).

sum(R,X,Y,Z,R1):-
        remainder(R),
        digit(X) ,
        digit(Y) ,
        divide_with_remainder(R + X + Y, 10, R1, Z).

:-public gerald/2.
gerald(M,T):-statistics(runtime,_),gerald(M),statistics(runtime,T).
gerald([[G,E,R,A,L,D],[D,O,N,A,L,D],[R,O,B,E,R,T]]):-
        different([G,E,R,A,L,D,O,N,B,T]) ,
        G? \== 0 , D? \== 0 , R? \== 0 ,
        sum(R1,G,D,R,O),
        sum(R2,E,O,O,R1),
        sum(R3,R,N,B,R2),
        sum(R4,A,A,E,R3),
        sum(R5,L,L,R,R4),
        sum(O ,D,D,T,R5).
```