ICOT Technical Memorandum: TM-0177

TM-0177

FGHCシステム使用手引き

宮崎敏彦

June, 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

ICOT　1研　宮崎

1986/6/27

ＦＧＨＣシステム使用手引き

(Version 1.33)

（１）言語の特徴

本処理系で実行される言語FGHC(Flat Guarded Horn Clauses) はGHC[Ueda 85]を基に幾つかの拡張と制限を加えた言語である。FGHCの個々のclauseのシンタックスや名称、及び直感的なプログラムの意味はGHC と同様である。FGHCの主な特徴を以下に示す。

①guard 部に書ける述語を一部の組込述語のみに限定している。（組込述語については後述する。）
②passive part（head及びguard 部）はテキストに書かれた順序に従って左から右に逐次的に実行される。
③候補節のテストはテキストに書かれた順序に従って上から下に逐次的に試される。つまり、ある節はその節よりも上に書かれた節が失敗もしくは中断したときのみ実行の対象となる。
④上の②，③により失敗の検出は積極的には行なわない。つまり、次の様なプログラムは常に中断する。

```
  goal:    :- p(X, a).
clause:    p(true, b) :- true | true.
```

⑤全ての候補節が失敗するとゴールが失敗する。
⑥全てのゴールが中断した場合のみデッドロックを検出する。
⑦passive partにおける変数の同一性のチェックは行っていない。つまり、次の様なプログラムはゴールの実行順に因らずデッドロックする。

```
  goal:    :- X=Y, p(X, Y).
clause:    p(A, A) :- true | true.
```

⑧モジュール間の述語名の衝突をなくす、という程度の簡単なモジュール化機能がある。
⑨ESP の述語を呼び出す為の組込述語が用意されている。
⑩active part に書かれたユニファイ(=) はtrust の直後に実行される。ユニファイの失敗は呼び出したゴールの失敗として扱われる。

（２）データ型

本処理系（Version 1.30）で提供しているデータ型を以下に示す。ただし、システムに依存したデータ型は省く。

		例
①アトム	……	abc , 'ABC'
②整数	……	123
③構造体（ゼロ要素を含む一次元配列）	……	{a,Y,[A,B]} , p(X), {}
④リスト	……	[a | X]
⑤ESP におけるオブジェクト		

【注意】

現在の処理系はESP で実現されている都合上、cdr 部がnil 以外のアトムである様なリストは２要素の構造体と見做される（例えば[a | b]はb(a)とコンパイルされる）。つまり、ソース・テキスト上で書かれたその様なリストと、プログラムの実行によって作られたリストはユニファイ出来ない。

（３）組込述語

組込述語の一覧は付録を参照していただくとして、以下ではそのうちの幾つかに付いて簡単に説明を加える。（以下で説明されない組込述語で一般のユーザが使ってはいけないものが幾つかあるので注意。）

(a) unbound(X)

この述語を実行しようとした時点で Xが未定義の場合のみ成功それ以外は失敗。

(b) atom(X), integer(X), list(X), structure(X, ^Length), object(X)

X がテストしている型に具体化されたら成功、未定義の場合は具体化されるまで実行を中断、具体化されたものがテストしている型でなければ失敗。Lengthには構造体の長さがユニファイされる。

(c) new-structure(^Structure, Length)

長さがLengthである構造体を新に生成しStructure とユニファイする。生成された構造体の要素は全て未定義。Lengthが未定義の場合は具体化されるまで実行を中断。

(d) element(Structure, Position, ^Element)

Structure のPosition番目を取り出しElement とユニファイする。Structure 及びPositionが未定義の場合は具体化されるまで実行を中断。

(e) make-symbol(CharacterCodeList, ^Symbol)

要素が文字コードであるリスト(CharacterCodeList) からシンボルを生成しSymbolとユニファイする。CharacterCodeList が未定義の場合は具体化されるまで実行を中断。

(f) symbol-name(Symbol, ^CharacterCodeList)

与えられたSymbolからそのシンボルの印字表現を構成する文字コードのリストをCharacterCodeList とユニファイする。Symbolが未定義の場合は具体化されるまで実行を中断。

(g) wait(X)

X が具体化されるまで実行を中断し、具体化されると成功する。

(h) ground-term(X)

X が変数を含まない場合成功、それ以外は中断。

(i) member(X, List)

Listの要素としてX が存在するかどうかテストし、存在すれば成功、存在しなければ失敗する。テストの方法はpassive partのユニファイと同じ（つまり変数同士の場合は中断）。X 及びListが未定義の場合は中断。

(j) otherwise

述語otherwise を含む節よりも上位に書れた節がすべて失敗である時otherwise は成功。それ以外は中断。上位に節が無い場合も成功。

(k) reduce(Goal, ^BodyGoals)

Goalに対する候補節のpassive part及びactive part の全てのユニファイ(=) を実行し、成功した節のボディのゴールをBodyGoals にユニファイする。Goalが未定義、あるいはGoalの実行が中断した場合は、reduceが中断。Goalの実行が失敗した場合は、BodyGoals にfailをユニファイする。

(l) call(Goals , ^Result, Control)

Goals の値をFGHCのゴールとして実行する。実行が成功するとResultにsuccess なるシンボルがユニファイされ、失敗するとfailがユニファイされる。Resultに対するユニファイが失敗しない限りcall自身は失敗しない。Control に値をバインドすると、Goals の実行は有限時間内に中止される。Goals が未定義の場合は中断。

(m) get-class-object(ClassName , ^Oject)

ClassName で指定されたESP のクラスオブジェクトをObjectにユニファイする。指定されたクラスオブジェクトが存在しない時は失敗する。ClassName が未定義の場合は中断。

(n) method-call(HethodName(Object, Arguments))

Objectに対しHethodNameで指示されるメソッドの呼び出しをArguments で与えられる環境で行なう。Arguments 内の変数は具体化される事があるので注意（passive partの呼び出しでも具体化は中断されない）。Object,HethodName 及びArguments が未定

義のとき中断。

(o) class-method-call(MethodName(ClassName, Arguments))

ClassName で指定されるクラスオブジェクトに対してメソッド呼び出しを行う。それ以外はmethod-call と同じ。

(p) create-io-stream(Type, Option, ^Stream)

Typeで指定されるI/O ストリームを生成しStreamにユニファイする。Typeで指定できるものは以下のもの。

Type	Option
window	size(X,Y), position(X,Y), title(name)
user	指定しても無視
file(FILE-NAME)	Version-1.30ではinput,output,append のいずれかを指定

windowはOptionで指定されたウィンドウを新たに作る。userはstandard-I/Oの意。
fileはFILE-NAME で指定されたファイルを作る。
Version-1.30でのストリームに対する命令は次のものが可能。

```
read(X)          ····  読み込んだ文字列を構文解析しX とユニファイする。
write(X)         ····  X の値をその印字表現として書き出す。
write-chars(Cs)        文字コードのリストCsを書き出す。
get(X)           ····  入力された文字の文字コードをX とユニファイする。
put(X)           ····  X を文字コードとする文字を書き出す。
nl               ····  改行コードを書き出す。
open             ····  ファイルをオープンする。
close            ····  ファイルをクローズする。
add-operator(Precedence, Type, Name)
```

(q) merge(MergedStream, RequestStream)

RequestStream を通して依頼されるストリームをマージしてMergedStreamに出す。各要素は先着順にマージされる。新たにマージしてほしいストリームがある場合には、RequestStream に対してmerge(Stream) をユニファイすれば良い。

【例】
```
:- p(X), q(Y),r(N),
     merge(Out, [merge(X),merge(Y) | N]), out(Out).

r(N) :- true | N=[merge(Z)], s(Z).
```

【ESP の呼び出しに対する注意】

FGHCからESP を呼び出す場合（あるいは値を返す場合）データ変換を行うため、無限のStructure を含む場合はシステムがエラーメッセージを出力する。

（４）マクロ機能

FGHCのプログラムに現れる特定のパターンに対しては、マクロ展開機能が働き、自動的にあらかじめシステムによって定められた、もしくはユーザによって定められたパターンに展開される。

（４－１）ユーザによるパターンの定義

パターンの定義はモジュール内のプロシジャーの定義よりも前で記述されなければならない。定義の方法は次のように行なう。

```
:- macro  パターン => 展開されるパターン.
```

```
[例]    :- macro list3 -> [1,2,3].
```

プログラム中での表記は#パターン。

```
[例]    p(X, #list3).
```

（４－２）システムによって規定されているパターン

基本的にはESP で許されている定数値の表記のためのマクロをそのまま使える。ただし本システムではストリングなるデータ・タイプが無いので、ｓｔｒｉｎｇ#“文字列”（単に“文字列”でも良い）は文字コードのリストに置き変えられる。

また、二つ以上のストリームをマージする際には、MergedStream名のつぎにRequestStream 名を並べて書くことができる。

```
[例]  merge(MergedStream,X,Y,Z) ->
                    merge(MergedStream,[merge(X),merge(Y),merge(Z)])
```

（５）使用法

FGHC System Version-1.30はSIMPOS（version 2.00以上）の下で稼動するシステムである。以下では本システムの使用法に付いて説明する。

（５－１）システムの作成

FGHC System Version-1.30はソース・プログラムかオブジェクト・コードで提供される予定である。

①　提供されたものがソース・プログラムの場合

以下の手順でオブジェクト・コードを作る。

（ａ）SIMPOSの"file manipulator"で、ソース・プログラムの入ったフロッピィをマウントする。

（ｂ）ソース・プログラムを、全て自分のディレクトリにコピーする。

（ｃ）SIMPOSの"Librarian" で"Execute" を選択し fxd("MAKEFGHC.COM") を実行。

（ｄ）システム・メニューでクラス fghc-systemを実行出来るよう"login.cmd" に登録する。

（ｅ）login しなおす。

（ｆ）システム・メニューを使いFGHC System を起動。（あるいは、"Debugger"で :go(#fghc-system) を実行してもよい。）

また、オブジェクト・コードをセーブしたい場合には、上記（ａ）～（ｃ）を実行済みの状態（オブジェクト・コードができている）で、

（ｇ）SIMPOSの"Librarian" で"Execute" を選択しfxd("SAVEFGHC.COM") を実行する。

②　提供されたものがオブジェクト・コードの場合

以下の手順でオブジェクト・コードをロードする。

（ａ）フロッピィをUnit.1にいれる。

（ｂ）SIMPOSの"Librarian" で、"Utility" を選択し、更に、"Load from Fdd" を選択する。

（ｃ）Unit.1を指定すると、フロッピィは、自動的にマウントされ、マウスの中一回クリックで、フロッピィに入っているクラス名が、メニュー形式で表示されるので"all but selected"を選び、実行する。

（ｄ）～（ｆ）は①と同じである。

また、オブジェクト・コードをセーブしたい場合には、"SAVEFGHC.COM"をフロッピィからコピーし（ｇ）を、実行する。（オブジェクト・コードのフロッピィには、オブジェクト・コード以外に、FGHCのソースとコマンド・ファイルの二種類が入っている。）

【注】オブジェクト・コードのセーブはコンパイルの直後、すなわちFGHCシステムを起動せずに行なうこと。

システムを起動するとstandard-I/Oと呼ぶウインドウが自動的に表示される。以後エラーメッセジ等はこのウインドウに出力される。またstandard-I/O内でマウスを一回クリックするとFGHC System のシステム・メニューが表示される。

（５－２）プログラムのロードとセーブ

本システムで実行可能なコードの形式はインタプリティブ・コードとコンパイルド・コードの２種である。前者は組み込み述語reduceで実行出来るコードであり実行のトレースも取る事が出来るが、後者は前者より高速である代りそのどちらも出来ない（もちろん両者の間での呼び出しは可能）。

ロードの方法はシステムメニューでConsult かCompile のいずれかを選びファイル名を入力すれば良い。ファイル名の属性は".ghc"がデフォルトであり、それ以外の属性のファイルを入力する場合はシングル・クウォートで括る必要がある。複数のファイルを一度に入力する場合はリスト形式で入力することができる。

ロードしたプログラムはSIMPOSの"librarian" を用いてセーブすることができる。このときクラス名の代りにFlatGHC のモジュール名を入力する。またトップレベルでmodules, listing(モジュール名) ,listing(モジュール名, 述語姓, 引き数の数）を使って既にロードされているモジュールに関する情報を得ることができる。

（５－３）実行

プログラムの実行はプロンプト ":-" に答えてゴールを入力すれば良い。ゴールの形式は、ユーザ定義の述語呼び出しであればモジュール名@ゴールである。組み込み述語の呼び出しはモジュール名を省いても良い。

以前に入力したゴールを再利用したい場合は次のコマンドを使う（プログラムのロードにおけるファイル名の入力の時にも使える）。

^x	····	再利用モードに入る
^p	····	次の候補を一つ表示
^n	····	次の候補を一つ表示
^u	····	表示された文字を全て消去
^w	····	表示された文字を１ワード単位で消去

この他スクロールのコントロールは ^s と ^q 、実行の停止は ^c もしくはabort キーを使う。

（５－４）状態の変更

システム・メニューでState を選択するとState Windowと呼ぶウインドウが表示される。ウインドウ内の各項目の意味を以下に述べる。

（ａ） mode

トレース・モードのスウィッチ。trace を選択するとインタプリティブ・コードの実行のトレースが出来る。細かな設定は下に表示されているモード・ウインドウで行なう。トレース情報の意味は下に示す通り。

*No Call	……	bound Noを持つゴールが呼び出された。 *はそのゴールがトレース・モードであることを意味する（トレース・モードでない場合は ｛）。
Susp(No)	……	ゴールの実行が中断した。Noは中断した番号。
Redu(No)	……	実行の結果No番目の節を使ってreduceされた。
reSu(No)	……	No番目に中断していたゴールがresumeされた。
Fail	……	ゴールの実行が失敗した。
Exit	……	ゴールの実行が終了した。

（ｂ）Exec bound

本システムは高速化の為にbounded depth first と呼ぶスケジュール法を用いており、そのbound を変更する。値は正の整数。

（ｃ） logging

standard-I/Oに出力される文字をファイルに書き出すかどうかスウィッチする。onの時は"-fghc-.log"というファイルに書き出す。既に"-fghc-.log"なるファイルが存在する場合は、そのファイルの後ろに書き足される。

（ｄ） print-length/depth

表示される構造体の深さと、リストの長さを規定する。

（ｅ） top goals re-display

入力したゴールを再表示するか否か。

（ｆ） backtrace

システムがdeadlockを検出したとき中断しているゴールを表示するかどうか。

（５－５）tracerの使用法

モード・ウィンドウで適当なゲートをオンにするとトレーサを使用する事ができる。また、ユーザはリーシュ時にリーシュ・コマンドを入力し、システムの実行の様子を知ることができる。

abort	……処理を中断。ＦＧＨＣシステムのトップ・レベルに戻る。
enqueue(e)	……当該ゴールの処理をあとまわしにする。
fail(f)	……当該ゴールを失敗させる。
help	……トレーサのヘルプ・メニューを表示する。
next goal(l)	……指定されているゴールの子孫まで、ゴールのトレース情報を表示しない。ゴールの指定は、set spied goalコマンドで、解除は、reset spied goalコマンドで行なう。
next procedure(p)	……指定されている述語まで、他の述語のトレース情報を表示しない。述語名の設定、解除はSet Spy Windowで行なう。
next spy	……指定されたゴールまたは述語まで、他の述語のトレース情報を表示しない。
no trace(n)	……以降のトレースは行なわない。
reset spied goal(r)	……スパイの指定を解除する。
set spied goal(t)	……スパイを指定する。
suppress(s)	……当該ゴールの以降のトレース情報は表示しない。
step	……次のトレース・ポイントに行く。

各コマンドの入力は、キーボードからの一文字で行われる。abort 、help、next spy、stepは、それぞれ、キーボードのABORT 、HELP(hでもよい）、LINE FEED 、RETURNの各キーにアサインされている。各コマンドについては、ヘルプ・コマンドにより、実行中に調べることもできる。

（５－６）algorithmic-debuggerの使用法

ゴールの実行がdeadlockで終了した場合、ユーザはゴールの変わりに“@”を入力することによってこのデバッガのモードに入ることができる。但しプログラムはインタプリティブ・コードでロードされていなければならない。

システムからの問い合わせは次の３種類のものがある。

(1) suspended(Goal)　:: Goalが中断した儘で終了した。
(2) open(Goal)　:: Goalの実行が終了していない。
(3) Goal　:: Goalの実行が終了してる。

ユーザはそれぞれの問い合わせに対し、その引き数の状態を見てイエスかノーを答える。

（６）モジュールの定義

モジュールの定義は `module モジュール名.` で始まり `end.` で終わる。他のモジュールに公開する述語はDEC-10 Prolog 風に `:- public PredName/Arity.` と宣言しなければならない。また、モジュール内で新たに定義したいオペレータは
`:- op(Precedence, Type, Name).` で宣言する。

〔付録〕

付録－１にVersion-1.30でサポートされている組込述語の一覧を示す。表において“ｐａｓｓ”はpassive part, “ａｃｔ”はactive part を意味しマークが付いているものはそれぞれの部位でその組込述語が書けることを意味する。また、“＊”はその述語を一般ユーザが使ってはならない事を意味する。

付録－２にプログラム例を示す。

付録－３に本システムで使用しているウインドウを示す。

付録－４にESP で定義している以外に本システムで定義しているオペレータを示す。

付録－1

	Pass	Act	system predicate
	0	0	true
	0	0	fail
		0	X = Y
	0		unbound(X)
	0		atom(X)
	0		atomic(X)
	0		integer(X)
	0		list(X)
	0		structure(X, ^Length)
	0		object(Object)
		0	new_structure(^Structure, Length)
	0	0	element(Structure, Position, ^Element)
	0		X \= Y
	0		<Expression> < <Expression>
	0		<Expression> > <Expression>
	0		<Expression> =< <Expression>
	0		<Expression> >= <Expression>
	0		<Integer Expression> =:= <Integer Expression>
	0		<Integer Expression> =\= <Integer Expression>
	0	0	I := <Integer Expression>
	0	0	minus(X, ^Y)
	0	0	make_symbol(CharacterCodeList, ^Symbol)
	0	0	symbol_name(Symbol, ^CharacterCodeList)
	0		otherwise
	0		wait(X)
	0		ground_term(X)
	0		member(X, List)
		0	reduce(Goal, ^BodyGoals)
		0	call(Goals, ^Result, Stop)
		0	get_class_object(ClassName, ^Object)
	0		method_call(MethodName(Object, Arguments))
	0		class_method_call(MethodName(ClassName, Arguments))
		0	create_io_stream(Type, Option, Stream)
		0	merge(MergedStream, RequestStream)
*	0		create_tty(^StreamId, Option)
*	0		create_file(FileName, ^StreamId, Option)
*	0		get(StreamId, ^X)
*	0		put(StreamId, X)
*	0		write(StreamId, X)
*	0		read(StreamId, ^X)
*	0		open(StreamId)
*	0		close_tty(StreamId)
*	0		close_file(StreamId)
*	0		new_line(StreamId)
*	0		enter_system_stream(Id, Stream)
*	0		system_call(Id, Request)
*	0		register(Name, ^Value)

<Expression> ::= <Term> | <Integer Expression>

<Integer Expression> ::=
　　<integer> |
　　<Integer Expression> <operator> <Integer Expression>

<operator> ::= "+" | "-" | "*" | "/" | "mod"

付録－3

```
FGHC System  Version 1.30
:- element([1,2,3], 1, E).
%%%  element([1,2,3],1,2)
2 msec

:- test2@nrev([1,2,3], X).
%%%  test2@nrev([1,2,3],[3,2,1])
14 msec

:- test2@nrev(X, Y).

Deadlock !!!

2 msec

:- ■
```

standard-I/Oウィンドウ

```
FGHC State Window
 mode:  trace  no trace
 backtrace:   on  off
   logging:   on  off
  Exec bound: 100
 print depth: 4
       length: 10
 top goals re-display:   on  off

 [Trace]   CALL   REDUCED  EXIT
           SUSPEND  RESUME  FAIL
 [Leash]   CALL   REDUCED  EXIT
           SUSPEND  RESUME  FAIL
```

State ウィンドウ

```
FGHC Menu
    State
  Set Spy
  Consult
  Compile
Style Check
(Developer)
```

システム・メニュー

```
FGHC Spy Window
 module> test2
 predicate>  nrev/2
                Spy them
 test2@nrev/2


 Abort  Reset  Expunge    EXIT
```

Spy ウィンドウ

```
FGHC Compiler
<< Consult >>
  Input File Name : test2

Loading File : test2.ghc
  Consulting module : test2
nrev5/0, primes/0, sift/2, filter/3, nrev/2, concat/3, esp/1, io/1
, merge1/1, merge2/2, merge21/3, gen/3, outstream/2, END.
  Transferring module : test2
  Running ESP compiler : test2

Next File (Y/N) : ■
```

コンパイラ・ウィンドウ

付録－4

Operator	Type	Precedence
(@)	xfy	90
module	fx	1180
library	fx	1180
(|)	xfx	1020
(\=)	xfx	700
(macro)	fx	1080

Source Program Floppy contains the following.

FGHC System source program
- FGHCOPR.ESP
- CNPWIN.ESP
- LOG.ESP
- TUPLE2.ESP
- TUPLE.ESP
- ASTRAP.ESP
- LBUILTIN.ESP
- LLIBRARY.ESP
- TRAFILE.ESP
- KB.ESP
- KINSTR.ESP
- KCOMP.ESP
- MACRO.ESP
- COMPILER.ESP
- UTIL.ESP
- SPYCRE.ESP
- STACRE.ESP
- SMENU.ESP
- DMENU.ESP
- STATE.ESP
- INPUT.ESP
- OUTPUT.ESP
- WINDOW.ESP
- COMP_CODE.ESP
- INTER.ESP
- ASTRAP.ESP
- REGISTERS.ESP
- LEASH.ESP
- DEBUG.ESP
- INST.ESP
- FGHCFILE.ESP
- FGHCWINDOW.ESP
- IO_CONTROL.ESP
- BLT1.ESP
- BLT2.ESP
- BLT3.ESP
- BLT4.ESP
- BUILTIN.ESP
- LIBRARY.ESP
- GOAL.ESP
- META.ESP
- SCHEDULER.ESP
- RUNTIME.ESP
- TOP.ESP
- ASTRANS.ESP
- TRANS.ESP
- VARCHK.ESP
- MLIB.KLB
- ALDEB.KLB
- ALDEB.ESP

Command Procedure
- MAKEFGHC.COM
- LOADFGHC.COM
- SAVEFGHC.COM

FGHC Source Program
- bench.ghc
- path.ghc

Object Code Floppy contains the following.

```
Object Code
   $$template1
         :
         :
   $$template49
   template.dir

Command Procedure
   MAKEFGHC.COM
   LOADFGHC.COM
   SAVEFGHC.COM

FGHC Source Program
    bench.ghc
    path.ghc
```

ICOT:ALDEB.HLP

%%%

Algorithmic Debugger for Flat GHC programs

Version 1.0 (May 10, 1986)

Akikazu Takeuchi

%%%

1. HOW TO START

```
    @aldeb
```

The system responds

```
*** Algorithmic GHC Debugger 1.0 (1986-05-09)  by Akikazu Takeuchi

            Type "?- helpdb.", if you need help.

yes
| ?-
```

There two top-level procedures, "debia" for debugging termination with incorrect answer and "debdl" for deadlock. They are invoked in the following way.

```
| ?- debia <Goals>.        %% if <Goals> succeeded with incorrect answer.
| ?- debdl <Goals>.        %% if <Goals> deadlocked.
```

2. QUERIES

The debugger can find a bug when a program terminates with incorrect answer or deadlocks. The debugger asks several questions at run-time whether a part of computation is correctly executed or not.

There are three types of questions.

(1) <goal>?
It indicates that <goal> terminates and asks whether the result is correct or not.
You should answer "yes" (or "y") if <goal> exhibits the correct result.
Otherwise, say "no" (or "n").

(2) open(<goal>)?
It indicates that some descendant of <goal> suspends and asks whether the result matches the intended one or not.
You should answer "yes" (or "y") if the result is correct.
Otherwise, say "no" (or "n").

(3) suspended(<goal>)?
It indicates that <goal> immediately suspends and asks whether the result matches the intended one or not.
You should answer "yes" (or "y") if the result is correct.
Otherwise, say "no" (or "n").

If some of questions are incorrectly answered, then the behavior of the debugger is no longer correct. Therefore the output of the debugger also becomes meaningless. Users have to answer each question after sufficiently considering its meaning.

3. CAUTIONS

Source program must be consulted, neither compiled nor ghccompiled.

If you reconsult the source program, then you must do

```
?- cleardef.
```