ICOT Technical Memorandum: TM-0366

TM-0366

PROLOG上のGHC処理系の作成技法

上田和紀

July, 1987

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

Prolog上のＧＨＣ処理系の作成技法

上田　和紀
（財）新世代コンピュータ技術開発機構

(Flat)ＧＨＣのプログラムをPrologプログラムにコンパイルする技法について解説する．生成するPrologプログラムは一見複雑であるが，使っている各技法に，特に難しいところはない．目的プログラムは，コンパイラ・ベースのProlog処理系上での効率を重視して設計されている．本処理系のような性格の処理系は，言語の普及・習得や，小～中規模のプログラミングに重要な役割を果たすと期待される．なお本論文は，ＧＨＣに関する基本的な知識を仮定している．

１．はじめに

プログラミング言語の普及を考えるとき，ある程度実用に堪え，しかも移植性のよい処理系を早期に開発することがきわめて重要である．一昔前にPascalが一世を風靡したのも，上のような処理系が自由に入手できたことが大きな原因になっている．

また，そのような処理系を開発することは，プログラム言語の実用的価値を立証するためにも重要である．せっかくよい言語を設計しても，その処理系として簡単なインタプリタしか存在しなければ，実用言語としての評価は生まれてこない．たとえば，Concurrent Prolog の場合，当初はProlog上のインタプリタしか存在しなかった[4]．このインタプリタはごく小さなプログラムを走らせるのには便利だったが，同等の，つまり同じ入出力関係をもつPrologプログラムを同じProlog処理系で走らせたときと比べた速度低下が２桁にものぼった．このため大きなプログラムを走らせることができなかったし，また並列論理型言語の実用性を説得する材料としても不十分であった．その後，本論文で紹介するＧＨＣ処理系の前身のConcurrent Prolog 処理系[6]や，Ｃなどの汎用言語による処理系[3]が開発されてはじめて，Concurrent Prolog の本格的な使用が始まったと言ってよい．

本論文に紹介する(Flat)ＧＨＣ処理系は，ＧＨＣ[7,8]の普及初期での使用を目的に開発したものである．Prolog上の処理系であるがゆえに，第３節に述べるようないくつかの制約があるが，小さく，移植が容易であるうえ，効率も十分高い．ＧＨＣの習得や，小～中規模のプログラミングに十分役立つと考えられる．また，記述性や効率に関する問題点を蓄積し，それを次に開発するべき，より本格的な処理系に反映させることも，本処理系の役割のひとつである．本処理系は，今後開発するＧＨＣ処理系の性能に関する大まかな指標を与えるという役目も果たしている．

2．処理の概要

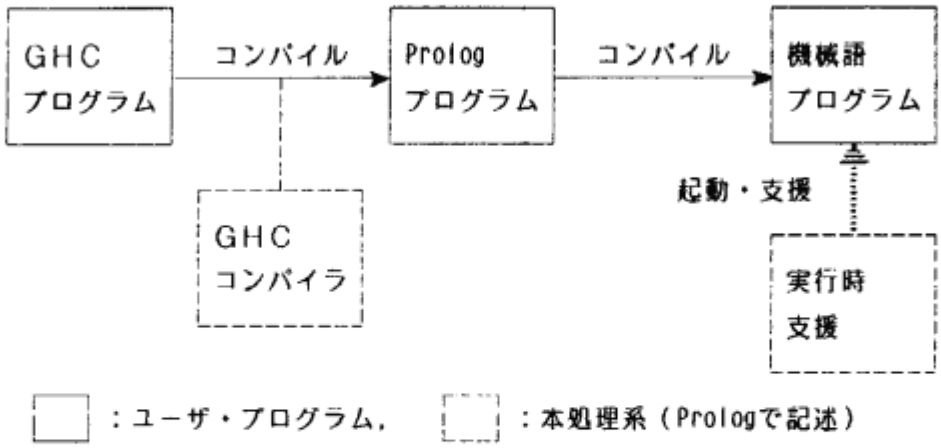

図１．　GHCプログラムの処理の概要

本処理系の概念図は，上のとおりである．ユーザの書いたＧＨＣプログラムは，まずPrologで書いたＧＨＣコンパイラによってPrologプログラムに変換する．これをさらにPrologコンパイラによって機械語にコンパイルし，やはりPrologで書いた実行時支援の下で実行する．コンパイラのターゲットおよび記述言語にPrologを選んだのは，次の理由による：

① Prologプログラムは，効率のよい機械語にコンパイルできる[9]．
② ソース言語（ＧＨＣ）とPrologとの類似性，およびPrologシステムが提供するすぐれた対話的プログラミング環境により，開発，改良，保守が容易にできる．特に，入出力をはじめとした実行時支援の作成の手間が少なくてすむ．
③ Prologは比較的よく標準化されているので，移植が容易である．

3．言語仕様

まず構文であるが，本処理系はDEC-10 Prolog [1]上に開発したものであり，受けつけるＧＨＣプログラムの構文も，Prologの事実上の標準であるDEC-10 Prolog にならっている．ＧＨＣとPrologの構文上の差異はコミット演算子 '|' の有無だが，本処理系では，コミット演算子を，ガード・ゴール群とボディ・ゴール群を引数にもつ２引数の演算子として扱っている．

次に意味的な側面であるが，本処理系が扱うＧＨＣプログラムは一般にFlat GHC（FGHC）と呼ばれているクラスに属するものである．つまり，各節のガードのゴールとして，組込述語の呼出ししか書くことができない．組込述語といっても，入出力プロセスのような

“すぐに終了しない”，“プロセス的な”ものは除かれ，書けるのは，同一化や算術比較など，直ちに終了ないしは中断する述語だけである．具体的にどのようなものが使用できるかは，付録１を参照してほしい．ここに紹介する処理系の組込述語はきわめて限られている．それで不足の場合はPrologへのインタフェース（prolog/1* ）を用いてコーディングすることになるが，この機能については本論文ではふれない．

ボディに書くことのできる組込述語は，同一化のための'='/2 ，算術演算のための ':='/2，入出力のためのinstream/1・outstream/1 ，およびPrologとのインタフェース prolog/1である．

本処理系は，多くの節をもつ述語の効率を改善するために，モード宣言機能を備えている．詳細は５．３節(iii) を参照してほしい．

４．擬似並列処理の実現

本処理系は，ＧＨＣの並列実行を逐次Prologプログラムで模擬するものである．このような擬似並列処理を行なうときは，並列実行の粒(granule) ，すなわちその単位どうしは並列に動くが，個々の処理は逐次的に行なうような処理単位を定め，それをうまくスケジューリングしてやる必要がある．ここではそのような粒として，“あるゴールが起動されてからコミットないしは中断するまでの処理”を採用している．実行すべきゴールは，一本の待ち行列で管理している．

具体的には，処理は次のように進む．いま，行列の先頭からゴールＧを取り出したとする．もしＧが組込述語の呼出しであれば，各組込述語の処理ルーチンによって処理する．さもなければ，各候補節 C_i について，Ｇが C_i にコミットできるかどうかを調べる．つまり， C_i を

$$H :- A_1 , \ldots , A_k \mid B_1 , \ldots , B_m .$$

とするとき，頭部の同一化Ｇ＝Ｈとガード・ゴール $A_1 , \ldots , A_k$ が，Ｇを具体化しないというＧＨＣの意味規則の下で解けるかどうかを調べる．もし解ければ，ゴールＧは C_i にコミットし，残ったボディ・ゴール $B_1 , \ldots , B_m$ をＧにかえて待ち行列に入れる．もしいかなる候補節にもコミットできなければ，Ｇの処理を中断する．つまり，Ｇ自身を再び待ち行列の終端に戻し，かわりに待ち行列の先頭のゴールを呼び出す．これは，この処理系がデータの動的待合せ(busy waiting)方式をとっていることを意味する．

本処理系は，標準のスケジューリング方式として，100 段の制限つき深さ優先(bounded

---

* '/n'という記法は，その左側に書いた述語記号，ないしは関数記号がｎ引数のものであることを示すためのものである．

depth-first) 方式を採用している．ここで，n段の制限つき深さ優先スケジューリングとは，待ち行列の終端につくゴールが，n段書換え可能であることである．ゴールGがn段書換え可能であるとは，Gが節

$$H :- A_1 , \ldots , A_K \mid B_1 , \ldots , B_m .$$

によってゴール $B_1 , \ldots , B_m$ に書き換えられるとき，$B_1 , \ldots , B_m$ が(n-1) 段書換え可能なゴールとして待ち行列の先頭につくことである．Gが0段書換え可能であるとは，Gを待ち行列の終端につけ，待ち行列の先頭のゴールを新たに実行すべきことを意味する．

上の定義から，1段の制限つき深さ優先は，幅優先(breadth-first) と等価であり，無限段の制限つき深さ優先は，単なる深さ優先(depth-first) と等価であることが理解されよう．

段数の値は，端末からゴール節を入力するときに指示することもできる．段数を小さくすると，各ゴールを小刻みに少しずつ実行するようになり，各ゴールの待ち時間は少なくなる．しかし，プロセス切替，すなわち待ち行列操作のオーバーヘッドが増え，また一般に，一度中断したゴールを，その中断の原因のなくならないうちに再び実行することが多くなるので，効率が低下する．一方，深さを無制限にすると，停止しないプログラムに対して不公平(unfair)なスケジューリングになってしまう．深さの標準値100 は，プロセス切替と中断による効率低下が目立たなくなるように選んだものである．

待ち行列の中には，サイクル・マーカと呼ばれる特別の（ダミー）ゴールがひとつはいっている．サイクル・マーカは，トップレベルのゴール節からはじまる実行の終了判定を行なう．サイクル・マーカを実行したとき，待ち行列に他にゴールがなければ，すべてのゴールが成功したわけであるから，成功を報告して止まる．待ち行列が空でないときは2通りの可能性がある．まず，前回サイクル・マーカを実行した後，中断せずに処理できたゴールがひとつでもあるかどうかを調べ，あればサイクル・マーカ自身を再び待ち行列の終端につけ，待ち行列の先頭のゴールを実行する．一方，すべてのゴールが中断していたら，前回のサイクル・マーカの実行以降計算は全く進んでいないことになる．このときは，かりに再び各ゴールを実行しても，（状況が全く変っていないから）すべて中断するはずである．そこでサイクル・マーカは，計算の行きづまり(deadlock)を報告して止まる．中断しなかったゴールがあるかどうかは，フラグによって検出している．

なお，プログラム節のボディの同一化ゴールは，その節へのコミットの直後に，待ち行列に入れることなくその場で実行し，スケジューリングの手間を省いている．これは，Flat GHCにおいては，ボディの同一化ゴールがガードの意味規則によって中断することはないからである．

また，ボディの算術演算ゴール（':='/2の呼出し）も，コミット直後に中断なしに実行できるものについては，待ち行列に入れずに実行している．たとえば，

```
gen(N, [X | Xs]) :- N>0 | X=N, N1:=N-1, gen(N1, Xs).
```

という節があったとすると，コミット時にはNが正整数に具体化していることが保証される（なぜなら，そうでなければガード・ゴールN>0 が成功せず，コミットできないからである）．そこで，ゴールN1:=N-1 は，中断に関する配慮をせずにコミット後直ちに実行している．だが，

```
gen(N, [X | Xs]) :- true | X=N, N1:=N-1, gen(N1, Xs).
```

の場合は，X=N は実行できるが，N1:=N-1 は，Nが不定ならば後回しにしなければならない．このため，一般のボディ・ゴールと同様の，待ち行列操作を伴うコードを生成している．

なお，本処理系では，成功しなかった計算はすべて行きづまりとみなしている．一般に，あるユーザ定義述語の呼出しGがどの候補節にもコミットできない理由はふたつ考えられる：

① Gが，いずれかの節にコミットできるところまで具体化していない．
② Gが，どの節にもコミットできない形に具体化してしまっている．

両者を区別すれば，実用的な観点からみた処理系の使い勝手は向上するかもしれない．たとえば，行きづまり時のメッセージの詳細化，行きづまりの早期検出，あるいはConcurrent Prolog で最初に提案された特殊な組込述語otherwise [5]の導入などが可能になる．しかし，①と②を区別することは，少なくともPrologの上の処理系にとっては容易でないので，本処理系でもこの区別を行なっていない．

5．コンパイル技法

5．1　スケジューリング

本処理系はコンパイラと実行時支援ルーチンからなる．コンパイラは，各ＧＨＣの手続き（同じ述語名を頭部にもつプログラム節の集まり）ごとにひとつのPrologの手続きを生成する．生成した目的コードは，全体を統括するスケジューラがあって，その下で起動されるのではなく，自ら，引数として受けとった待ち行列を操作し，以降の仕事のスケジューリングを行なう．このように，引数として受けとる待ち行列を，ここでは継続(continuation) と呼ぶ．これは，計算機科学における通常の継続と同様，“残りの仕事”を表現したものである．

さて，コンパイルしてできたPrologの各述語の仕事は，もとのＧＨＣの述語を呼び出したゴールが候補節のいずれかにコミットできるかを調べ，その結果に応じて，次に実行す

べきゴールをきめて起動することである．このとき，起動したゴール以外の実行すべきゴールは，起動したゴールに継続として与える．各ゴールは，存在しない述語を呼び出さない限り，失敗することなく“次の”ゴールを起動しつづける．このゴール起動の連鎖がおわるのは，先述のサイクル・マーカに対応するゴールが，トップレベルのゴール節の成功ないしは行きづまりを検出したときである．

個々のPrologの述語は，次のよっつの部分からなる：

① ゴールの起動を端末に知らせる節
② もとのＧＨＣの各プログラム節に対応する節の群
③ どの候補節にもコミットしなかったことを端末に知らせる節
④ コミットしなかった場合に，自分自身を待ち行列に戻して他ゴールを起動する節

①と③は，トレース・モード（後述）でコンパイルしたときだけ生成される．図２に，ＧＨＣで記述したクイック・ソートプログラムを，図３に，その目的コード（非トレース・モード）を示す．図３からも明らかなように，非トレース・モードでは，各述語の目的コードはもとの述語より一本ずつ多い候補節をもつ．

コンパイルしてできる各述語は，もとの述語の引数に対応するもののほかに，７個の追加引数をもつ：

① 制限つき深さ優先スケジューリングの“現在の”段数（RC）
②・③ 継続を表わす差分リストの頭と尾（Ch,Ct）
④ 前回のサイクル・マーカの実行以後，コミットしたゴールがあるか否かを示すフラグ（Dnd）（次のサイクル・マーカの実行時に検査される）
⑤ 制限つき深さ優先スケジューリングの段数の初期値（RCmax）
⑥ 当該ゴールの識別番号（Myid）
⑦ 次に生成するゴールに与えるべき識別番号（Nextid）

図３以下のプログラムでは，これらの引数は上記（ ）内の変数名に，必要に応じて接尾辞 0,1,x,y などをつけたものによって示している．

⑥と⑦は，トレース出力で，ゴールに識別番号をつけて表示するために用いる．プログラムのトレースをしないときはこの両者は実質的には利用せず，⑥は０に設定し，⑦は更新せずに同じ値を持ち回る．図３のコードもそうしている．トレース・モードでコンパイルしたときだけ，⑥と⑦を更新，管理するためのコードが出る．

継続の各要素は，次の形をしている：

```
$(Goal, Ch, Ct, Dnd, Nextid)
```

```
qsort(Xs, Ys) :- true | qsort(Xs, Ys, []).

qsort([X|Xs], Ys0, Ys3) :- true |
    part(Xs, X, S, L),
    qsort(S, Ys0, [X|Ys2]), qsort(L, Ys2, Ys3).
qsort([],     Ys0, Ys1) :- true | Ys0=Ys1.

part([X|Xs], A, S,  L0) :- A <  X |
    L0=[X|L1], part(Xs, A, S,  L1).
part([X|Xs], A, S0, L ) :- A >= X |
    S0=[X|S1], part(Xs, A, S1, L ).
part([],     _, S,  L ) :- true   | S=[], L=[].
```

Figure 2. A quicksort program.

```
:- public qsort/9.
:- mode qsort(?,?,+,+,-,+,+,+,+).
qsort(Xs,Ys,RC0,Ch,Ct,_,RCmax,_,Nextid) :-
  RC0>0, !,
  RC is RC0-1,
  qsort(Xs,Ys,[],RC,Ch,Ct,nd,RCmax,0,Nextid).
qsort(A1,A2,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(qsort(A1,A2,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                        Chx,Ctx,Dndx,           Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).

:- public qsort/10.
:- mode qsort(?,?,?,+,+,-,+,+,+,+).
qsort(A1,Ys0,Ys3,RC0,Ch0,Ct,_,RCmax,_,Nextid) :-
  nonvar(A1), ulist(A1,X,Xs),
  RC0>0, !,
  RC is RC0-1,
  part(Xs,X,S,L,RC,
    [$(qsort(S,Ys0,[X|Ys2],RC,Chx,Ctx,Dndx,RCmax,0,Nextidx),
                               Chx,Ctx,Dndx,        Nextidx),
     $(qsort(L,Ys2,Ys3,RC,Chy,Cty,Dndy,RCmax,0,Nextidy),
                          Chy,Cty,Dndy,        Nextidy)|Ch0],
     Ct,nd,RCmax,0,Nextid).
```

Figure 3. Object code of the quicksort program (bounded depth-first mode)
(continued on the next page).

```
qsort(A1,Ys0,Ys1,RC0,Ch0,Ct,_,_,_,Nextid) :-
  nonvar(A1), unil(A1),
  RC0>0, !,
  ubody(Ys1,Ys0),
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
qsort(A1,A2,A3,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(qsort(A1,A2,A3,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                          Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).

:- public part/11.
:- mode part(?,?,?,?,+,+,-,+,+,+,+).
part(A1,A,S,L0,RC0,Ch,Ct,_,RCmax,_,Nextid) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A<X,
  RC0>0, !,
  ubody([X|L1],L0),
  RC is RC0-1,
  part(Xs,A,S,L1,RC,Ch,Ct,nd,RCmax,0,Nextid).
part(A1,A,S0,L,RC0,Ch,Ct,_,RCmax,_,Nextid) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A>=X,
  RC0>0, !,
  ubody([X|S1],S0),
  RC is RC0-1,
  part(Xs,A,S1,L,RC,Ch,Ct,nd,RCmax,0,Nextid).
part(A1,_,S,L,RC,Ch0,Ct,_,_,_,Nextid) :-
  nonvar(A1), unil(A1),
  RC>0, !,
  ubody([],S), ubody([],L),
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
part(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(part(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                            Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 3. Object code of the quicksort program (bounded depth-first mode) (continued from the previous page).

ここでGoalは，あるＧＨＣゴールに対応するPrologゴールである．このPrologゴールの第②，③，④，⑦追加引数は，この継続要素を作るときにCh, Ct, Dnd, Nextid とそれぞれ同一にしてある．いいかえれば，Ch, Ct, Dnd, Nextid は，Goalから出た端子である．

Goalを実行するときは，その追加引数に，適当な値を与えなければならない．そのうちのいくつか（①，⑤，⑥）は，継続の終端にGoalを入れるときに決められるが，他（②，③，④，⑦）はGoalの起動時にならないときまらない．だが，Goalを起動する側からみると，それらの引数の絶対位置は各ゴールごとに異なるので，値を供給するのが容易でない．そこで，このようにGoalから端子を出して，端子経由でGoalに値を供給するようにしたわけである．継続の各要素における各端子の位置はGoalの引数の数によらず一定だから，それらに値を供給するのは同一化によってきわめて安価にできる．

各節の構造の説明にはいる前に，もっとも簡単な図２のqsort/2 が，具体的にどのようにコンパイルされているかをみてみよう．図３のように，ＧＨＣのqsort/2 はPrologのqsort/9 にコンパイルされている．最初のふたつのゴール節は，DEC-10 Prolog 用のpublicおよびmode宣言であるが，これはDEC-10 Prolog 固有のものなのでここでは無視する．プログラム節の第１節は，スケジューリングの制限段数RC0 が正ならば，その値を１減じた新しい段数RCを与えて，ＧＨＣのゴールqsort(Xs, Ys, [j) に対応するPrologのゴールを呼んでいる．このとき継続Ch・Ct，制限段数の初期値RCmax ，および次のゴール番号Nextidは，qsort/9 が受けとったものをそのまま渡している．行きづまり検出用フラグは，受けとった値を捨てて，定数nd（no deadlock を表わす）にセットしている．またゴール識別番号は()としている．

qsort/9 の第２節は，制限段数が()のときだけ用いられる．この節では，頭部と呼出し側の同一化によって，継続に対する操作を行なっている．すなわち，継続の先頭（追加引数②）を

```
[$(Gx,Ch,Ct,Dnd,Nextid) | Ct]
```

と同一化して先頭要素を取り出し，これと同時に，継続の終端（追加引数③）を

```
[$(qsort(A1,A2,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                        Chx,Ctx,Dndx,           Nextidx) | Ct]
```

と同一化して，現在処理中のqsort/9 の呼出しを付加している．次に起動するゴールは，取り出してきたGxであるが，Gxに与えるべき新しい継続は，受けとった継続からGxを除き，qsort/9 の呼出しを加えた，差分リストCh-Ct である．そこで，先頭要素を取り出すときに，端子経由でこの新しい継続をGxに供給している．また，Gxに与える行きづまり検出用フラグと次のゴール識別番号は，“現在”の値 Dnd, Nextid（追加引数④，⑦）を端子経

由で供給している．結局，追加引数②と，[$(Gx,Ch,Ct,Dnd,Nextid) | Ct] との同一化では，Gxの取り出しと同時にCh，Ct，Dnd ，Nextidの設定を行なっているわけである．

一方，継続の終端につけるqsort/9 の呼出しの方は，ふたつの本来の引数A1とA2，制限段数の初期値RCmax ，および自分自身の識別番号Myidについては，現在の値をそのまま引き継いでいる．一方，現在の段数値（追加引数①）は，0であったのを初期値RCmax にリセットしている．その他の追加引数②，③，④，⑦は，当ゴールの起動時に設定できるように，対応する端子と同一のものにしている．

第２節は，以上の（大がかりな）設定を頭部の同一化ですませたあと，ゴール incore(Gx)で，“次の”ゴールGxを起動している．incore/1は，引数として与えたゴールを処理するコンパイル・コードを呼ぶためのDEC-10 Prolog の組込述語で，コンパイル・コードとインタプリティブ・コードの区別を無視すれば，他のProlog処理系にもあるcall/1と同じと考えてよい．

さて一般論に戻ろう．ＧＨＣの節"Head :- Guard | Body."は，次のような形のPrologの節にコンパイルされる：

```
（引数の受取り）：－
    （Headと呼出し側との同一化），
    （Guard の実行），
    （制限段数の検査），
    !，
    （Bodyの同一化ゴールの実行），
    （Bodyの算術演算（中断しないもの）の実行），
    （制限段数の更新），
    （残りのBody中のゴールのスケジューリング）．
```

Headと呼出し側との同一化とGuard の実行は，擬似並列ではなく逐次的に行なう．つまり，まずHeadとその呼出し側の各引数の同一化を左から順に行ない，ついでGuard の各ゴールを左から順に実行する．それらのいずれかが中断したら，以後の同一化ないしはガード・ゴールの実行は試みずに次の節の検査に移る．ただし，ガード・ゴールの中の同一化ゴールについては，コンパイル時に実行してしまっている．つまりその同一化によって生じる結合を，節全体に施し，同一化ゴール自体を消去してから，他の部分のコンパイルを行なっている．たとえば，節

```
qsort(A, Ys0, Ys1) :- A=[] | Ys0=Ys1.
```

は，まずA=[]を実行し，

```
qsort([], Ys0, Ys1) :- true | Ys0=Ys1.
```

に変換してからコンパイルしている．これは，部分評価(partial evaluation)技法の応用である．

ガード・ゴールの実行順序を固定したことにより，節のガードの実行が，中断すべきでないときに中断してしまうことがある．たとえば，ゴールp(3, 4, 35) は，本来ならば節

```
p(X, Y, C) :- D>=X, D<Y, D:=(C-1)/10 | ... .
```

にコミットできなければならないが，本処理系では D>=X の評価が中断すると，次の候補節の実行に移ってしまう．しかし，本処理系が許すガード・ゴール(prolog/1 を除く）の範囲では，プログラマがゴールの並べ方に注意することによって，この問題を避けることができる．

ＧＨＣのコミット操作は，Prologの'!' にコンパイルされる．コミットの後は，Body中の同一化ゴールをまず実行し，次に，算術演算（':='/2の呼出し）のうち，コンパイラが中断のおそれがないと判断したものを実行する．これらの実行は，中断に対する考慮をしないので高速である．最後に，残ったゴールを，次のようにスケジュールする：

① もし残ったゴールがなければ，継続の先頭のゴールを起動する．
② もし残ったゴールがちょうどひとつならば，そのゴールを起動する．このとき，そのゴールには現在の継続をそのまま与える．
③ もし残ったゴールがふたつ以上あれば，最左のものを起動する．このとき，現在の継続の先頭に最左以外のゴールを付加したものを，起動するゴールに与える．

上のうち②，③は，起動するゴールがコンパイル時にわかっているので，高速に実行できる．これは効率上重要なことである．ＧＨＣプログラムには，ストリーム等のデータ構造の生成，消費を，終端再帰的(tail-recursive)な節を用いて行なうものがきわめて多い．本処理系は，このようなＧＨＣの終端再帰的な節を，Prologの終端再帰的な節にコンパイルする．Prologの本格的な処理系は，終端再帰呼出しあるいは一般に終端呼出し(last call) に対して特別の最適化[10] を行なう**ので，もとのＧＨＣプログラムに対しても良

** この最適化はいつもできるわけではない．その終端呼出しの実行時に，それが属する手続きの実行が決定的(determinate) である，つまりその手続きにとっての最後の選択肢を調べていることを，Prolog処理系が検出した場合だけ行なわれる．本処理系の生成するPrologプログラムは，常にこの最適化ができるような形になっている．

い性能を期待できる．特に大切なことは，終端（再帰）呼出しを最適化すると，戻り先や局所変数等を積んでおくためのスタックが伸びずにすみ，いくらでも長い計算ができるようになる点である．

生成するPrologの各述語の最後の節は，すべての候補節が中断した場合，およびスケジューリングの制限段数が0になった場合の処理をする節である．この節は，受けとった継続の終端に処理中のゴールを付加し，継続の先頭のゴールを起動する．

ＧＨＣプログラムを起動するゴール節は，そのままではPrologの目的コードを起動できない．Prologのコードを起動するためには，まずゴール節を継続（端子つきのゴールの差分リスト）の形に変換しなければならない．ＧＨＣプログラムの実行は，この継続を引数として，サイクル・マーカを呼出すことによって始まる．サイクル・マーカは，直ちに自分自身を継続の終端に移し，継続の先頭のゴールを起動する．以降のスケジューリングは，上述のように各ゴールが自ら継続を操作して行なう．

サイクル・マーカは，２種類用意してある．ひとつは，単に終了および行きづまりの判定を行なうもので，'$END'という実行時支援ルーチンによって実現している（付録２の214 行目）．もうひとつは，起動されたときに，ユーザからの指令に応じて継続の内容を端末に打ち出したり，トレースに関するモードを切り換えたりする機能をもったものであり，'$ENDSPY' というルーチンによって実現している（付録２の225 行目）．どちらを使うかは，端末からゴール節を入力するときに指定できるようになっている．

５．２　同一化

ＧＨＣのガードにおける同一化は，Prologのそれと異なり中断させなければならないことがある．このため，ＧＨＣの同一化は単純にPrologの同一化にコンパイルしてしまうわけにはいかず，一方向の同一化，つまりマッチングにコンパイルしなければならない．本処理系では，この一方向の同一化を，個々の場合に応じてできるかぎりインライン展開したコードによって処理し，良い効率を実現している．

実際には，中断の判定は，Prologの非論理的述語nonvar/1と'=='/2によって行なっている．たとえば図２において，qsort/3 の第１節の第１引数は[X | Xs] という形をしている．これに対応するPrologの目的コード（図３）では，まず呼出し側の引数を変数A1で受けとり，ゴールnonvar(A1)によってA1が非変数項に具体化しているか調べる．そのあとulist(A1,X,Xs)によってふたつに分解している．ulist(A1,X,Xs)は，A1=[X | Xs] と等価であり，単独で使用するとＧＨＣの意味規則に反してA1を具体化する可能性があるが，nonvar(A1)と併用することによってこれを避けている．なお，頭部の引数に，関数記号が入れ子構造になった項が書いてある場合は，その項のトップレベルを上の要領で分解したあと，その引数のうち非変数のものを同じ要領で分解する．

述語'=='/2よる中断の検査が必要になるのは，頭部に同じ変数が２回以上出現する場合である．たとえば，p(X,X)という頭部があるとき，これは“両引数が同一ならば”という

意味であって，“両引数が同一化できるならば”という意味ではない．したがって，これをp(X,X, ...) というPrologの節にコンパイルしてはならず，

```
p(A1,A2, ...) :- A1==A2, ...
```

という形のコードを生成しなければならない．ここで，'=='/2は，両辺の項が現在同一の形の項に具体化しているかを調べるPrologの組込述語である．

5．3　その他の機能

(i) 実行トレース機能

本処理系はコンパイラ方式ではあるが，実行トレースをとることができる．トレースをとるためには，トレース出力を出すためのコードを生成する必要がある．図４が，クイック・ソートの中の述語part/4をトレース・モードでコンパイルしたものである．通常のコード（図３）とちがうのは，次の点である：

① ゴールを起動したとき，およびそれがコミットできなかったときにメッセージを出す節が生成される．
② メッセージ中のゴールに識別番号を付加するために，それを計算するゴールが生成される．

(ii)深さ優先スケジューリング

たとえば，ストリームの生成プロセス，（いくつかの）変換プロセス，および消費プロセスからなるデータ駆動のストリーム処理プログラムを考えてみよう．ここで，各変換プロセスが，入力ストリーム上の１個の要素に対応して作る出力ストリーム上の要素の個数は，ある（入力ストリームに依存しない）上限によって押さえられているとする．このようなプログラムにおいて，制限つき深さ優先スケジューリングの深さ制限が本質的に意味をもつのは，ストリームを自律的に生成しつづけることのできる生成プロセスだけである．他のプロセスは，入力ストリーム中のデータがなくなれば，深さ優先スケジューリングの下でも中断せざるを得ない．したがって，ストリーム生成プロセス以外のプロセスから制限段数の検査を省略しても，制限つき深さ優先スケジューリングの性質は保存できる．また，ストリームの生成プロセスを含めて，すべてのプロセスを制限なしの深さ優先スケジューリングにしてしまっても，停止するプログラムに関しては正しい（実行すべきものは実行する）スケジューリングとなる．

以上のことから，本処理系は，制限なしの深さ優先スケジューリングを行なうコードを生成する機能をもっており，それを制限つき深さ優先スケジューリングのコードと混用することもできる．図５がpart/4の深さ優先のコードである．制限値の検査と更新のための

```
:- public part/11.
:- mode part(?,?,?,?,+,+,-,+,+,+,+).
part(A1,A2,A3,A4,_,_,_,_,_,Myid,_) :-
  ghcspied(part,4), trace_call(Myid,part(A1,A2,A3,A4)), fail.
part(A1,A,S,L0,RC0,Ch,Ct,_,RCmax,_,Nextid0) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A<X,
  RC0>0, !,
  'ubody?'([X|L1],L0,Nextid0),
  Nextid1 is Nextid0+1, Nextid is Nextid0+2,
  RC is RC0-1,
  part(Xs,A,S,L1,RC,Ch,Ct,nd,RCmax,Nextid1,Nextid).
part(A1,A,S0,L,RC0,Ch,Ct,_,RCmax,_,Nextid0) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A>=X,
  RC0>0, !,
  'ubody?'([X|S1],S0,Nextid0),
  Nextid1 is Nextid0+1, Nextid is Nextid0+2,
  RC is RC0-1,
  part(Xs,A,S1,L,RC,Ch,Ct,nd,RCmax,Nextid1,Nextid).
part(A1,_,S,L,RC0,Ch0,Ct,_,_,_,Nextid0) :-
  nonvar(A1), unil(A1),
  RC0>0, !,
  'ubody?'([],S,Nextid0),
  Nextid1 is Nextid0+1,
  'ubody?'([],L,Netxid1),
  Nextid is Nextid0+2,
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
part(_,_,_,_,RC,_,_,_,_,Myid,_) :-
  ghcspied(part,4), trace_suspension(RC,Myid), fail.
part(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(part(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                          Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 4. Predicate 'part' compiled in trace mode.

```
:- public part/11.
:- mode part(?,?,?,?,+,+,-,+,+,+,+).
part(A1,A,S,L0,RC,Ch,Ct,_,RCmax,_,Nextid) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A<X, !,
  ubody([X|L1],L0),
  part(Xs,A,S,L1,RC,Ch,Ct,nd,RCmax,0,Nextid).
part(A1,A,S0,L,RC,Ch,Ct,_,RCmax,_,Nextid) :-
  nonvar(A1), ulist(A1,X,Xs),
  integer(A), integer(X), A>=X, !,
  ubody([X|S1],S0),
  part(Xs,A,S1,L,RC,Ch,Ct,nd,RCmax,0,Nextid).
part(A1,_,S,L,_,Ch0,Ct,_,_,_,Nextid) :-
  nonvar(A1), unil(A1), !,
  ubody([],S),
  ubody([],L),
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
part(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(part(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                          Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 5. Predicate 'part' compiled in depth-first mode.

コードがなくなり，簡単になっている．

(iii) モード宣言

標準の目的コードは，図２のクイック・ソートプログラムのように，候補節が少数しかなく，しかも多くの場合最初の節にコミットするような述語からなるプログラムについては十分効率がよいが，候補節が多数あって，それらが呼出し側の同じ引数の値を待ち合わせる場合は，中断の検査を１回ですませた方が経済的である．このため，本処理系にはモード宣言機能がある．

ある述語ｐを呼び出したゴールが，第ｋ引数が非変数項に具体化していないとｐのどの候補節にもコミットできない場合，ｐの第ｋ引数を入力引数と呼ぶ．たとえば，図２のpart/4の場合，第１引数が入力引数である．第２引数は，第１，２節にコミットするためには具体化していなければならないが，第３節にコミットするためには具体化している必要がないので，入力引数とはいわない．本処理系では，入力引数がある場合，モード宣言でその位置を指定することによって，その引数が具体化しているかどうかの検査を１回ですませるコードを生成することができる（図６）．図６のように，入力引数の宣言をしたときの目的コードは，補助述語を用いた２段構造になる．まず第１段で入力引数に関する中断の検査を行ない，第２段で残りの仕事を行なう．２段構えになるためのオーバーヘッドを補って余りあると考えられる場合には，この機能を使うとよい．特に，第１引数が入力引数の場合，モード宣言を行なうと，DEC-10 Prolog では，候補節の数が多ければハッシングによって節を選択するコードが生成する．図７は，入力引数の宣言を行ない，さらにトレース・モードを指定したときの目的コードである．

(iv)算術演算

現在の処理系では，プログラム節のボディにある算術演算(':='/2)のうち，最左のユーザ定義述語の呼出しより左側にあるものについて，左から順に実行したとき中断のおそれなく実行できるところまでは，コミット直後に中断の考慮なしに実行するようなコードを生成している．残りの算術演算ゴールについては，各ゴールごとに，それを処理するための，スケジューラ機能をもった専用の述語を生成し，通常のボディ・ゴールと同様のスケジューリングを行なっている．図８に，（中断の可能性のある）算術演算を含むＧＨＣの述語（フィボナッチ数列生成器）を，図９にその目的コードを示す．

６．処理系の構造

処理系は，大きく実行時支援とコンパイラの２部に分かれている．が，両者は一緒に使用することを想定しているので，実行時支援は，コンパイラ中の一部のルーチンを借用している．

実行時支援は，次のようなものを含む：

```
:-public part/11.
:-mode part(?,?,?,?,+,+,-,+,+,+,+).
part(A1,A2,A3,A4,RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
  nonvar(A1), RC>0, !,
  '$$$part'(A1,A2,A3,A4,RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).
part(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(part(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                            Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).

:-public '$$$part'/11.
:-mode '$$$part'(?,?,?,?,+,+,-,+,+,+,+).
'$$$part'([X|Xs],A,S,L0,RC0,Ch,Ct,_,RCmax,_,Nextid) :-
  integer(A), integer(X), A<X, !,
  ubody([X|L1],L0),
  RC is RC0-1,
  part(Xs,A,S,L1,RC,Ch,Ct,nd,RCmax,0,Nextid).
'$$$part'([X|Xs],A,S0,L,RC0,Ch,Ct,_,RCmax,_,Nextid) :-
  integer(A), integer(X), A>=X, !,
  ubody([X|S1],S0),
  RC is RC0-1,
  part(Xs,A,S1,L,RC,Ch,Ct,nd,RCmax,0,Nextid).
'$$$part'([],_,S,L,_,Ch0,Ct,_,_,_,Nextid) :-
  !,
  ubody([],S),
  ubody([],L),
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
'$$$part'(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$('$$$part'(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                                Chx,Ctx,Dndx,          Nextidx)
  |Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 6. Predicate 'part' compiled with a mode declaration.

```
:-public part/11.
:-mode part(?,?,?,?,+,+,-,+,+,+,+).
part(A1,A2,A3,A4,RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
  nonvar(A1), RC>0, !,
  '$$$part'(A1,A2,A3,A4,RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).
part(A1,A2,A3,A4,RC,_,_,_,_,Myid,_) :-
  ghcspied(part,4),
  trace_call(Myid,part(A1,A2,A3,A4)),
  trace_suspension(RC,Myid), fail.
part(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(part(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                          Chx,Ctx,Dndx,          Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).

:-public '$$$part'/11.
:-mode '$$$part'(?,?,?,?,+,+,-,+,+,+,+).
'$$$part'(A1,A2,A3,A4,_,_,_,_,_,Myid,_) :-
  ghcspied(part,4), trace_call(Myid,part(A1,A2,A3,A4)), fail.
'$$$part'([X|Xs],A,S,L0,RC0,Ch,Ct,_,RCmax,_,Nextid0) :-
  integer(A), integer(X), A<X, !,
  'ubody?'([X|L1],L0,Nextid0),
  Nextid1 is Nextid0+1, Nextid is Nextid0+2,
  RC is RC0-1,
  part(Xs,A,S,L1,RC,Ch,Ct,nd,RCmax,Nextid1,Nextid).
'$$$part'([X|Xs],A,S0,L,RC0,Ch,Ct,_,RCmax,_,Nextid0) :-
  integer(A), integer(X), A>=X, !,
  'ubody?'([X|S1],S0,Nextid0),
  Nextid1 is Nextid0+1, Nextid is Nextid0,
  RC is RC0-1,
  part(Xs,A,S1,L,RC,Ch,Ct,nd,RCmax,Nextid1,Nextid).
'$$$part'([],_,S,L,_,Ch0,Ct,_,_,_,Nextid0) :-
  !,
  'ubody?'([],S,Nextid0),
  Nextid1 is Nextid0+1,
  'ubody?'([],L,Nextid1),
  Nextid is Nextid0+2,
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
'$$$part'(_,_,_,_,RC,_,_,_,_,Myid,_) :-
  ghcspied(part,4), trace_suspension(RC,Myid), fail.
'$$$part'(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$('$$$part'(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                                Chx,Ctx,Dndx,          Nextidx)
  |Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 7. Predicate 'part' compiled in trace mode with a mode declaration.

```
fib(Max,N1,N2,Ns0) :- Max >N2 |
  Ns0=[N2|Ns1], N3:=N1+N2, fib(Max,N2,N3,Ns1).
fib(Max,N1,N2,Ns0) :- Max=<N2 | Ns0=[].
```

Figure 8. Fibonacci sequence generator.

```
:-public fib/11.
:-mode fib(?,?,?,?,+,+,-,+,+,+,+).
fib(Max,N1,N2,Ns0,RC0,Ch0,Ct,_,RCmax,_,Nextid) :-
  integer(Max), integer(N2), Max>N2, RC0>0, !,
  ubody([N2|Ns1],Ns0),
  RC is RC0-1,
  '$$$Afib'(N3,N1,N2,RC,
    [$(fib(Max,N2,N3,Ns1,RC,Chx,Ctx,Dndx,RCmax,0,Nextidx),
                          Chx,Ctx,Dndx,        Nextidx)|Ch0],
    Ct,nd,RCmax,0,Nextid).
fib(Max,_,N2,Ns0,RC,Ch0,Ct,_,_,_,Nextid) :-
  integer(Max), integer(N2), Max=<N2, RC>0, !,
  ubody([],Ns0),
  Ch0=[$(Gx,Ch,Ct,nd,Nextid)|Ch], incore(Gx).
fib(A1,A2,A3,A4,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$(fib(A1,A2,A3,A4,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                       Chx,Ctx,Dndx,           Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).

:-public '$$$Afib'/10.
:-mode '$$$Afib'(?,?,?,+,+,-,+,+,+,+).
'$$$Afib'(N3,N1,N2,_,Ch,Ct,_,_,_,Nextid) :-
  integer(N1), Temp is N1+N2, !,
  ubody(Temp,N3), do_next(Ch,Ct,Nextid).
'$$$Afib'(N3,N1,N2,_,
  [$(Gx,Ch,Ct,Dnd,Nextid)|Ch],
  [$('$$$Afib'(N3,N1,N2,RCmax,Chx,Ctx,Dndx,RCmax,Myid,Nextidx),
                          Chx,Ctx,Dndx,           Nextidx)|Ct],
  Dnd,RCmax,Myid,Nextid) :-
  incore(Gx).
```

Figure 9. Object code of the predicate 'fib'.

① トップレベル・インタフェース（ＧＨＣゴール節のPrologゴールへの変換，その実行および結果の表示）
② 同一化（ガードおよびボディから起動される同一化および非同一性の検査('＼='/2)）
③ その他の組込述語（入出力等）
④ サイクル・マーカ
⑤ トレースおよびバックトレース（不成功時の，解けなかったゴールの表示）
⑥ その他

またコンパイラは，次のような部分からなる：

① ソース・プログラムの入力，モード宣言等の宣言の解析
② 目的プログラムの書出し
③ 宣言節，トレースのための節，および中断処理用の節の生成
④ 各節のコンパイル
　④-1 ソース・プログラムの節の分解，目的プログラムの節の合成
　④-2 ガードのコンパイル
　④-3 ボディのコンパイル
⑤ その他

付録２に，処理系の全体を示す．特にむずかしいところはないが，コンパイラにおいて，コンパイルするＧＨＣの節および目的コードのPrologの節の中に現れる変数を，コンパイラ自身の変数と区別しないで用いていることに注意してほしい．これは，論理プログラムとしては好ましくない技法である．本来ならば，処理系中の変数と，処理されるプログラム中の変数は区別する．つまり後者は特別の定数項などで表現するのが，より“論理的”な方法である．本処理系ではあえてこれを区別せず，ガード部の解析中に得られた各変数に関する情報を，その変数自身を具体化することによって記憶し，非論理的述語'=='/2によって参照している．この便法を用いたのは，効率上の理由によるものであるが，論理的な書換え可能配列(mutable array) [2]をProlog処理系がサポートしていれば，それを用いて変数表を作ることによって，この論理変数の非論理的用法を，効率をあまり犠牲にせずにやめることが可能であろう．

７．移植性

本処理系は，DEC-10 Prolog 系のProlog処理系には比較的容易に移植できる．だが，本処理系を実用処理系として使うためには，本格的な，つまり次のような機能を備えたProlog処理系を用いることが重要である：

① 終端呼出しの最適化(last-call optimization)
② ヒープのガーベジ・コレクション

本格的な処理系以外では，メモリ使用量の問題から，あまり大きな計算をさせることができない．付録3に，上の要件を満たすProlog処理系として代表的なQuintus Prologに本処理系を移植するときの手引を示す．

8．開発の経緯

本処理系は，すでに稼動中であったProlog上のConcurrent Prolog 処理系[6]を改良する形で作成した．主な改良点は，読出し専用標記(read-only annotation)による同期からガードの意味規則による同期への変更，および頭部と呼出し側の同一化の際に行なっていた出力同一化（結果を返すための同一化）のボディへの移動に伴うものであり，スケジューリングの基本方式と，それを実現する目的コードの基本構造は変わっていない．

処理系の設計にあたって最も重視したのは，目的コードの効率である．処理系開発に先立って，まず目的コードのモックアップをいくつか作り，効率を比較して，最適のものを選定した．この作業は，高級言語による高級言語のコンパイラ作成において，非常に重要な作業である．目的コードの設計が終わってしまえば，処理系が動きだすまでにそれほど時間はかからなかった．むしろ，動きだしたあとの処理系の整備に，かなりの時間をかけている．

謝辞

本処理系のデバッグ・改良にあたって，近山隆氏をはじめ，数名の方々から，有意義な指摘をいただいた．ここに感謝する．本処理系とその手引書（付録１）は，近山隆氏による初期のConcurrent Prolog 処理系（1984，未発表）をもとに，コンパイルの基本方式やＧＨＣへの変更などの大改訂と，度重なる小改訂を加えてきたものである．

参考文献

1) Bowen, D.L.(ed.), Byrd, L., Pereira, F.C.N., Pereira, L.M. and Warren, D.H.D., DECsystem-10 Prolog User's Manual. Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.
2) Eriksson, L.-H. and Rayner, M., Incorporating Mutable Arrays into Logic Programming. In roc. Second Int. Logic Programming Conf., Uppsala Univ., Sweden, 1984, pp.101-114.
3) Houri, A. and Shapiro, E., A Sequential Abstract Machine for Flat Concurrent Prolog. Tech. Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1986.

4) Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter. ICOT Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.
5) Shapiro, E. and Takeuchi, A., Object Oriented Programming in Concurrent Prolog. New Generation Computing, Vol.1, No.1 (1983), pp.25-48.
6) Ueda, K. and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog. In Proc. 1985 Symp. on Logic Programming, IEEE Computer Society, 1985, pp.119-126.
7) Ueda, K., Guarded Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. report TR-208, Institute for New Generation Computer Technology, Tokyo, 1986.
8) Ueda, K. Introduction to Guarded Horn Clauses. ICOT Tech. Report TR-209, Institute for New Generation Computer Technology, Tokyo, 1986.
9) Warren, D.H.D., Pereira, L.M. and Pereira, F., PROLOG--The language and Its Implementation Compared with Lisp. Sigplan Notices, Vol.12, No.8 (1977), pp.109-115.
10)Warren, D.H., Optimizing Tail Recursion in Prolog. in Logic Programming and Its Applications, Caneghem, M.van and Warren, D.H.D. (eds.), Albex, Norwood, New Jersey, 1986, pp.77-90.

（付録１）

ＧＨＣシステム使用手引書
（DEC-10 Prolog 版）

１．８版

１．ソースプログラム・ファイルの作成

ＧＨＣプログラムを走らせるにあたってまず最初にしなければならないことは，ソースプログラムのはいったファイルを作ることです．使い慣れたテキスト・エディタを使って作成してください．本システムの受けつけるＧＨＣプログラムの構文は，DEC-10 Prologの構文にならっています．下に，本手引書でこのあと利用するプログラム例を示します：

```
test(M, S) :- M=\=0 | S=[push(M), pop(N) | S1], N1:=N-1, test(N1, S1).
test(M, S) :- M=:=0 | S=[].

stack([push(X) | S], D      ) :- true | stack(S, [X | D]).
stack([pop(X) | S], [Y | D1]) :- true | X=Y, stack(S, D1).
stack([], _      ) :- true | true.
```

言語仕様上注意すべきことは，本システムでは，各節のガードから呼び出すことができる述語が，第８章に示す組込述語に限られるということです．その他の注意点については第７章を参照して下さい．

２．システムの起動

```
@ghc
```

と入力して下さい．以下，下線を引いたところが，利用者の入力する部分です．最後には必ずリターン・キーを押します．上のコマンドに対し，システムは，

```
*** GHC System Vers. 1.8 ...

yes
| ?-
```

と返答します．これで実は，ＧＨＣシステムの載ったProlog処理系のトップレベルにはいったことになります．つまり，本手引書で紹介するＧＨＣシステムの各機能以外に，必要ならばProlog処理系の各機能も，ここから利用することができます．

３．コンパイル

ファイルの中のプログラムを走らせるには，まず'ghccompile'というコマンドを使ってそれをコンパイルし，目的コードをロードする必要があります：

```
| ?- ghccompile(<ソースファイル名>).
test/2, stack/2, END.

-temp-.ghc compiled: 452 words,      0.91 sec.

yes
```

<ソースファイル名> は，Prologの定数記号の書き方にしたがって与えます．ファイル名にピリオドを含む場合は，'source.ghc'のように全体を単一引用符で囲みます．

このコマンドは，一時ファイルを作って目的コード（Prologプログラム）を書き出し，ついでそれをロード（すなわちPrologコンパイラでコンパイル）します．Prologへのコンパイルの間，システムは，コンパイル中の述語名を順次端末に表示します．それからメッセージ'END.'を表示し，目的コードのロードを開始します．

目的コードをファイルに保存したいときは，かわりに２引数の'ghccompile'を使用して下さい：

```
| ?- ghccompile(<ソースファイル名>, <目的ファイル名>).
```

このときは，<目的ファイル名> を名前とする新しいファイルを作成し，目的コードをそこに書き出します．が，目的コードのロードはしません．保存した目的コードを使いたいときは，それを第４節に述べる要領でロードして下さい．

プログラムは，複数個のファイルに分けて作っておいてもかまいません．このときは，次のようにファイル名のリストを与えてコンパイルします：

```
| ?- ghccompile(['s1.ghc', 's2.ghc', ...]).
```

しかし，ひとつの述語を定義するプログラム節が複数のファイルにまたがっていない限り，

これらのファイルを，次のようにばらばらにコンパイルすることもできます：

```
| ?- ghccompile('s1.ghc').
...
yes
| ?- ghccompile('s2.ghc').
...
```

実行してみたプログラムに誤りがあって，それを修正して起動中のＧＨＣシステムに再び戻ってきたとき，プログラム全体を再コンパイルする必要はありません．修正した述語だけを再コンパイルしてください．再コンパイルの時間を短縮することができます．

プログラムの実行トレースをとりたいときは，プログラムをあらかじめトレース・モードでコンパイルしておく必要があります．このときは，ソースプログラム・ファイルの名前に疑問符'?' をつけて下さい：

```
| ?- ghccompile(['s1.ghc'?, 's2.ghc']).  % こうすると's1.ghc'だけトレース・
                                         % モードでコンパイルします．
```

ただしトレース・モードでコンパイルすると，目的コードは大きく，遅くなります．実行トレースをとるには，さらにどの述語をトレースするかを指定する必要があります．第６節を参照して下さい．トレース・モードの目的コードと非トレース・モードの目的コードは互換性があり，互いに他を呼ぶことができます．

４．目的コードのロード

２引数の'ghccompile'で作成した目的コードをシステムにロードするときは，下のように，Prologコンパイラでコンパイルして下さい：

```
| ?- compile( <目的ファイル名>).
```

１引数の'ghccompile'を用いているときは，これを行なう必要はありません．

５．プログラムの実行

さていよいよ，プログラムの実行です．いまあなたは，Prolog処理系にはいっているわけですから，トップレベルで入力するゴール節のうちのどれを，ＧＨＣのゴール節として実行するかを指定しなければなりません．このために，コマンド'ghc' を使用します：

```
| ?- ghc  <ゴール列>.                    % 'ghc'は前置演算子として使えます.
```

<ゴール列> の実行が成功すると，システムはまず実行時間を表示します．次にProlog処理系が，（無名変数以外の変数がある場合）変数の具体化情報を報告し，'yes' というメッセージを返します：

```
| ?- ghc test(2,S), stack(S,[]).
4 msec.

S = [push(2),pop(2),push(1),pop(1)]

yes
```

<ゴール列> の実行が行きづまると（すなわち，計算を進めることのできるゴールがないことをシステムが検出すると），システムは，まず計算が中断(suspend) したまま待ち行列に残っているゴールを表示します．ついで，トップレベルのゴールが失敗して，メッセージ'no'が返ります：

```
| ?- ghc test(2,S), stack(s,[]).
Deaclock detected.  Suspended goals are:
: $$$Atest(A,B)                    % ゴールN1:=N-1 を表わす（第６章参照）
: test(A,G)                        % 第１引数は，上記代入ゴールの左辺と共通
2: stack(s,[])                     % ２は，このゴールの一連番号（第６章参照）

no
```

ただし，未定義の述語を呼んだときは，システムは単にその旨だけを報告して，トップレベルのゴールを失敗させます．中断したゴールや未定義述語の名前などの表示はありません：

```
| ?- ghc test(2,S), stak(S,[]).
You called an undefined predicate.

no
```

この場合は，述語の定義を忘れていないか，また述語名の綴り違いがないか，よく確かめ

て下さい．

節のボディから呼んだ組込述語'='(同一化）または':='（算術演算）が失敗したときは，メッセージは次のようになります：

```
| ?- ghc test(2,s), stack(S,[]).
You tried to unify [push(2),pop(_256)|_257] with s

no
| ?- ghc 6:=2+3.
You tried to unify 5 with 6

no
```

上記の報告機能はすべて，不必要ならば抑制することもできます．第６節を参照して下さい．

現在のシステムは，100 段の制限つき深さ優先スケジューリングを採用しています．つまり，簡単な終端再帰的な述語を呼び出した場合，待ち行列中の他のゴールを実行する前に，最初の呼出しを含めて最大100 回再帰呼出しを実行します．２引数の'ghc' を使えば，制限段数を100 以外にしてゴール節を実行することもできます：

```
| ?- ghc( <ゴール列>, <段数>).
```

<段数> を１にすれば，幅優先のスケジューリングになりますが，効率は非常に低下します．

制限つき深さ優先スケジューリングの段数制限が必要でないときは，'ghccompile'のかわりに'ghcdcompile' を使って下さい．目的コードが多少小さく，また速くなります．制限つき深さ優先スケジューリングのコードと，制限なしの深さ優先スケジューリングのコードは互換性があり，互いに他を呼びあうことができます．後者は単に制限段数を無視して走るわけです．組込述語は，常に制限なしの深さ優先スケジューリングで実行されます．

実行時間を（性能解析プログラムに渡すなどの理由から）引数経由で受けとりたいときは，３引数の'ghc' を使って下さい：

```
| ?- ghc( <ゴール列>, <段数>, <時間>).
```

<ゴール列> に疑問符'?' を付加すると，プログラムの実行が時々止まって，現在の実

行状況を調べたり変更したりするためのコマンドが端末から入力できるようになります．詳細は第６節を参照して下さい．

６．プログラムのトレース

プログラムの実行トレースを表示したいときは，

```
| ?- gncspy.
```

と入力して下さい．このコマンドの実行後は，節のガードから呼んだゴール以外のすべてのゴールの呼出し時と中断時に，メッセージが表示されます．ただしそのためには，プログラムをトレース・モードでコンパイルしておかなければなりません（第３節参照）．トレース出力をやめるときは，

```
| ?- gncnospy.
```

と入力します．

トレース・メッセージは，次のような形式で出てきます：

```
1: Call test(2,A)
3: Call [push(2),pop(A) | B]=C
4: Call A:=B-1
4: Suspended
5: Call test(A,B)
5: Suspended
2: Call stack([push(2),pop(A) | B],[])
6: Call stack([pop(A) | B],[2])
...
```

'Call'メッセージについた番号は，生成されたゴールにシステムがつけた一連番号です．メッセージ'N: Suspended'は，つい今しがた起動されたゴールＮが中断したことを意味します．

特定の述語の呼出しだけをトレースしたいときは，

```
| ?- gncspy <述語列>.
```

と入力して下さい．ここで，<述語列> はトレースすべき述語の名前か，それに引数の数

の指定'/n'（nは非負整数）をつけたものです．それらをコンマで区切って並べたものでもかまいません．特定のゴールのトレースをやめるときは，

```
| ?- ghcnospy <述語列>.
```

と入力します．ただし，トレース中止の要求は，より一般的なトレース要求と矛盾するときは受け付けられません．たとえば，'ghcspy p'をすでに実行してあるとき，'ghcnospy p'と'ghcnospy'は実行できますが，'ghcnospy p/2'を実行して，2引数のpだけトレースをやめることはできません．

どの述語がトレース中かは，

```
| ?- ghcspying.
```

で知ることができます．

デバッグのためのもうひとつの機能として，プログラムに各実行“サイクル”の最初にコマンドを受け付ける機能があります．“サイクル”とは，その開始時に存在したすべてのゴール，およびそれから導出されたゴールが，完全に解けるか，中断するか，あるいは（制限つき深さ優先スケジューリングの制限によって）待ち行列に追い出されるまでの一連の計算を意味します．次のように，トップレベルのゴール（の並び）に疑問符'?'をつけると，計算は各サイクルの開始時に止まり，端末からのコマンド入力待ちの状態になります：

```
| ?- ghc (test(2,S), stack(S,[]))? .  % 構文上の理由から，'?' と'.' の間には
Cycle 0; function (h for help)?       % 空白が必要．
```

ここで入力できるコマンドは，次のいずれかです：

```
 <改行>:  このサイクルの終わりまで実行せよ
w<改行>:  待ち行列の内容を表示せよ (Write goals)
n<改行>:  サイクルごとに止まらずに実行せよ (Nodebug mode)
h<改行>:  コマンド一覧を表示せよ (Help)
a<改行>:  実行を中止せよ (Abort)
@<改行>:  Prologのゴール（'ghcspy'など）をひとつ受けつけよ (Accept command)
```

この機能は，プログラムをトレース・モードでコンパイルしていなくても利用できます．

第5節に示したような，トップレベルのゴール節の終了時の報告メッセージは，いつで

も表示されるようになっています．この表示をやめる必要のあるとき，および再開するときは，それぞれ

| ?- ghcnobacktrace.　　および　| ?- ghcbacktrace.

と入力して下さい．

プログラムをトレース・モードでコンパイルしてあれば，システムはすべてのゴール（ガードから起動されるものは除く）に一連番号をつけ，トレース時や待ち行列の表示時にゴールと一緒に表示します．同じ述語の呼出しが複数個あるとき，一連番号でそれらを区別することができます．

表示されるゴールの中に，プログラムで使わなかった述語名が現れることがあります．これらは，入出力とボディの算術演算を処理するためにコンパイラが生成した述語です．またコンパイラは，最適化のために，組込述語の呼出しの引数の順序を入れ換えることがあり，入れ換えたものがトレース出力や待ち行列の表示の中に現れることがあります．

7．節のガードに関する制限

第１節でも述べましたが，節のガードから呼び出せる述語は，第８節にあげる組込述語に限られます．プログラマの定義した述語をガードから呼び出すことはできません．

本システムは，節のガードを，次のような順序で実行します：

① ガード・ゴールの中のすべての同一化ゴール
② 節頭部とその呼出し側との同一化
③ 残りのガード・ゴール中，述語prologの呼出し以外のもの（左から右へ）
④ ガード・ゴール中の述語prologの呼出し（左から右へ）

①～④の実行中，ある同一化またはゴールが中断すると，残りの同一化またはゴールは試みずに他の候補節（もしあれば）の実行に移ります．

8．組込述語

現在のシステムには，次の述語が組込述語として用意してあります．ガードから呼ぶことができる述語にはG，ボディから呼ぶことができる述語にはBとマーク欄に表示してあります．

マーク	述語	機能
GB	true	直ちに成功します.
GB	prolog(X)	Xを，Prologのゴールとして実行します．この述語は，真に必要な場合以外は使用しないようにして下さい．この述語をガードから呼ぶとき，そのガードを呼んだゴールを具体化しないように注意しなければなりません．GHCの原則からは，そのような具体化は中断させるべきなのですが，本システムはその処理を行なわないからです.
G−	wait(X)	Xが非変数に具体化すれば，あるいはしていれば成功します.
−B	instream(X)	Xには，次のようなPrologの入出力ゴールのリストを与えます： fileerrors* nofileerrors* see(Fg) seeing(f) seen close(Fg) read(T) * get0(C) get(C) skip(Cg) ttyget0(C) ttyget(C) ttyskip(Cg) tell(Fg) telling(Fg) told write(Tg) display(Tg) writeq(Tg) print(Tg) put(Cg) tab(Ng) nl ttyput(Cg) ttytab(Ng) ttyflush ttynl (* 下に重要な注意があります)

述語'instream'は，その引数Xの各要素が指定する入出力操作を，要素の出現順に実行します．上の操作のうち，引数名に接尾辞gがついているものは，その操作を処理するときに引数が基底項（変数を含まない項）になっていなければなりません．なっていないときは，基底項になるまで'instream'の処理が中断します．CgあるいはNgと書いてある引数は，（Prologでは整数を値とする式を書けますが，GHCでは）整数に具体化し（てい）なければなりません．上にあげた以外の操作を与えると，異常終了によってトップレベルに戻ってしまいます．また，トップレベルのゴール節からはじまる処理の中で'instream'を２度以上呼んだときも，異常終了によってトップレベルに戻ります.

出力関係の操作'write(X)'や'ttyflush'等ができるのは，出力

操作のなかには，入力促進メッセージの出力のように，入力操作との前後関係を保証しなければならないものがあるからです．

'read(X)' は，項をひとつ読み込み，Prologの組込述語 'numbervars'によって基底項化して，その結果をXと同一化します．基底項化によって，読み込んだ項の中のすべての変数は，'$VAR(0)' ，'$VAR(1)' ，'$VAR(2)' 等に具体化します．

'nofileerrors'は，異常処理の扱いを変更するための操作です．これを実行する前は，入力ファイルを終わりまで読んでしまったあとさらに'get0(C)' ，'get(C)'，'ttyget0(C)'および 'ttyget(C)' を実行しようとすると，異常終了によってトップレベルに戻ります．しかし，'nofileerrors'を実行すると，これらの操作は，ファイルの終わりにきたあと何度でもCを26に具体化するようになります．また，see(Fg) やtell(Fg)で指定するファイル名に誤りがあった場合も，異常終了ではなく処理中断を引き起すようになります．'fileerrors'はこのような変更扱いを解消し，もとの扱いに戻します．

−B　outstream(X)　Xには，次のようなPrologの入出力ゴールのリストを与えます：

fileerrors* nofileerrors*
tell(Fg)　telling(Fg)　told
write(Tg)　display(Tg)　writeq(Tg) print(Tg)
put(Cg)　tab(Ng)　nl
ttyput(Cg)　ttytab(Ng)　ttyflush　ttynl
(* 'instream'の説明を見て下さい)

述語'outstream' は，その引数Xの各要素が指定する出力操作を，要素の出現順に実行します．上で，引数名に接尾辞gがついているものは，その操作を処理するときに引数が基底項になっていなければなりません．なっていないときは，基底項になるまで 'outstream' の処理が中断します．CgあるいはNgと書いてある引数は，整数に具体化し（てい）なければなりません．上にあげた以外の操作を与えると，異常終了によってトップレベルに戻ってしまいます．また，トップレベルのゴール節からはじまる処理の中で'outstream' を２度以上呼んだときも，異常終了によってトップレベルに戻ります．

GB　X = Y　XとYを（GHCの同期規則の下で）同一化します．この述語は

主として，節のボディに出力のための同一化を記述するために用います．

G－　X ＼= Y　ＸとＹが，（今後両者がどのように具体化しても）同一化不可能であることがわかったとき成功します（cf. '=＼='）．

GB　V := E　整数式Ｅを評価し，結果の値をＶと同一化します．Ｅは，DEC-10 Prologにおける数式の構文にしたがって与えます．ゴール 'V=2+3' が，Ｖを項'2+3' に具体化するのに対し，'V:=2+3'はＶを５に具体化します．プログラム節のゴールの数式中に変数を書いた場合，その（あとからきまる）値は整数でなければならず，整数式であってはなりません．たとえばゴール列'X:=E, E=3+5' は，トップレベルのゴール節として与えない限り成功しません．ただし，DEC-10 Prolog では-1は整数とみなすので，'X:=E, E=(-1)' は必ず成功し，Ｘを-1に具体化します．

G－　E1 < E2
G－　E1 > E2
G－　E1 =< E2
G－　E1 >= E2
G－　E1 =:= E2
G－　E1 =＼=E2

整数式 E1 および E2 を評価し，それらの値が，述語記号に示す関係にあることがわかったとき，そしてそのときに限り成功します． E1 および E2 は，DEC-10 Prolog における数式の構文にしたがって与えます．数式中に変数を書いた場合，その（あとからきまる）値は整数でなければならず，整数式であってはなりません．X ＼=1とX=＼=1の違いに注意してください．前者は，Ｘが１以外のどの非変数項に具体化したときにも成功するのに対し，後者はＸが１以外の値をもつ整数式に具体化したときだけ成功します．X=:=Y は，ＸとＹが同じ値をもつ整数式に具体化したときに成功します．

９．モード宣言

本処理系は，モード宣言機能を備えています．モード宣言機能とは，同じ入力引数の値を持たなければならない候補節を数多くもつ述語の効率を改善するための機能です．モード宣言は，つぎのような節をソースプログラム中に書くことによって行ないます：

```
:- mode stack(+,?).
```

これは，述語'stack' の第１引数が入力引数で，第２引数が通常引数であることを宣言するものです．この宣言は，宣言対象の述語の最初のプログラム節より前に現れなければな

りません．

モードには入力と通常の２種類があります．入力引数は，宣言中では'+'と表記します．入力と宣言した引数の目的コードは，その引数のトップレベル（主関数記号）の値を待ち合わせる操作を，個々の候補節で毎回行なうかわりに，それらの候補節を調べる前に一度だけ行ないます．このモードを第１引数に宣言すると，プログラムによっては，インデクシングによって非逐次的に節を選択できる（Prologの）目的コードが生成されるようになります．入力引数の宣言が述語の意味を変えないためには，その述語の中に，入力引数の値の主関数記号が不定のまま選択できる節があってはなりません．

通常引数は，宣言中では'?'と表記します．通常引数は，モード宣言を与えなかったときと同様，各候補節ごとに独立に待ち合わせ等の処理を行ないます．モード宣言を与えなかったときの目的コードと，すべての引数を通常引数と宣言したときの目的コードは同一です．

なお，このモード宣言機能は，PARLOGのモード宣言機能とは全く異なるものであることに注意してください．

（付録２）

```
%%%%%%%%%%%%%%%%%% GHC Compiler System for DEC-10 Prolog %%%%%%%%%%%%%%%%%%
%                                                                             %
%                        Version 1.8  (June 19, 1987)                         %
%                                                                             %
%                              by Kazunori Ueda                               %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 1 :- op(1150,fx, (ghc)).
 2 :- op(1150,fx, (ghcspy)).
 3 :- op(1150,fx, (ghcnospy)).
 4 :- op(700, xfx,:=).           % becomes
 5 :- op(700, xfx,\=).           % dif
 6 :- op(50,  xf, ?).            % for specifying 'trace-mode'

   %%%% SAVING EXECUTABLE IMAGE %%%%
 7 :- public save/0.
 8 save :- plsys(core_image), nolog, ghcbacktrace, header.
 9 header :-
10     write('*** GHC System Vers. 1.8 (1987-06-19)  by Kazunori Ueda'), nl.



   %%%% RUN-TIME SUPPORT %%%%

   %%% TOP LEVEL %%%

11 :- public (ghc)/1, (ghc)/2, (ghc)/3.
12 ghc(Goals)            :- ghc(Goals, 100,   _).
13 ghc(Goals, Bound)     :- ghc(Goals, Bound, _).
14 ghc(_,     Bound, _) :- illegal_bound(Bound), !,
15     display('Illegal bound value: '), display(Bound), ttynl, fail.
16 ghc(Goals, Bound, T) :-
17     save_standard_io, reset_ioflag,
18     solve(Goals, Bound), !,
19     statistics(runtime,[_,T]), restore_standard_io,
20   ( recorded('$GHCBACKTRACE',_,_), !, display(T), display(' msec.'), ttynl;
21     true ).
22 ghc(_,     _,     _) :- /* failed in solving Goals */
23     restore_standard_io,
24     recorded('$GHCBACKTRACE',_,_), !,
25   ( recorded('$GHCBACKTRACE_END',_,Ref), !, erase(Ref);
26     recorded('$GHCUNIFYFAIL',    _,Ref), !, erase(Ref);
27     display('You called an undefined predicate.'), ttynl ), fail.

28 :- mode illegal_bound(?).
29 illegal_bound(Bound) :- integer(Bound), Bound>0, !, fail.
30 illegal_bound(_    ).

31 :- mode solve(?, +).
32 solve(Goals,  _    ) :- var(Goals), !, fail.
33 solve(Goals?, Bound) :- !,
34     prepare_goal_queue(Goals, Bound, Ch,Ct, Nextid),
35     statistics(runtime,_),
36     incore('$ENDSPY'(0, Ch,Ct, nd, Nextid)).
37 solve(Goals,  Bound) :- /* not_functor(Goals, ?, _), !, */
38     prepare_goal_queue(Goals, Bound, Ch,Ct, Nextid),
39     statistics(runtime,_),
40     incore('$END'(Ch,Ct, nd, Nextid)).
```

```
41 :- mode prepare_goal_queue(?, +, -,-, -).
42 prepare_goal_queue(Goals, Bound, Ch,Ct, Nextid) :-
43     put_queue(          % The definition of 'put_queue' is in the compiler.
44         Goals,          % Conjunctive goals to be put in the queue
45         Ch,Ct,          % The queue (d-list) of goals
46         Bound,Bound,    % Current and initial values of the reduction counter
47         Id_calc,true,   % D-conjunction of arithmetic goals to calculate the
48                         % IDs of Goals to be displayed on tracing
49         1,              % Initial value of the goal IDs
50         0,Id1,          % Initial and final values of the offsets of goal IDs
51         '$TOP',         % Predicate that called Goals
52         [],_,           % Old and new predicate dictionaries (for compiling
53                         % ':=' goals that are called not at the top level)
54         debug           % Debugging mode (debug/nodebug)
55         ),
56     Nextid is 1+Id1,    % Compute the ID of the next goal to be generated
57     call(Id_calc), !. % Provide the top-level goals with their IDs



   %%% UNIFICATION %%%
   % The following unification predicates fail on suspension.

   % Compound term and constant:  uskel(X,Skeleton)
58 :- mode uskel(+,+).
59 uskel(X,X).

   % Nil:  unil(X) (Hacked version of uskel)
60 :- mode unil(+).
61 unil([]).

   % List:  ulist(X,Car,Cdr) (Hacked version of uskel)
62 :- mode ulist(+,-,-).
63 ulist([H|T],H,T).

   % Unification in clause bodies:
64 :- mode ubody(?,?).
65 ubody(X,X) :- !.
66 ubody(X,Y) :- /* X and Y are ununifiable */
67     recorded('$GHCBACKTRACE',_,_),
68     recorda('$GHCUNIFYFAIL',_,_),
69     display('You tried to unify '), writeuser(X), display(' with '),
70     writeuser(Y), ttynl, fail.

71 :- mode 'ubody?'(?,?, +).
72 'ubody?'(X,Y, Myid) :- ghcspied('=',2), trace_call(Myid,X=Y), fail.
73 'ubody?'(X,Y, _    ) :- ubody(X,Y).

   % Ununifiability
   % ghcdif(X,Y) succeeds if and when X and Y proved to be ununifiable.
74 :- mode ghcdif(?,?).
75 ghcdif(X,_) :- var(X), !, fail.
76 ghcdif(_,Y) :- var(Y), !, fail.
   % Here, 1st & 2nd args are non-variables.
77 ghcdif(X,Y) :- functor(X,F,A), functor(Y,F,A), !, ghcdif_args(A,X,Y).
78 ghcdif(_,_).  % succeeds when the function symbols or the arities mismatch

79 :- mode ghcdif_args(+, +,+).
80 ghcdif_args(0, _,_) :- !, fail.
81 ghcdif_args(N, X,Y) :- /* N>0, */
```

```
 82      arg(N,X,Xn), arg(N,Y,Yn), ghcdif(Xn,Yn), !.
 83  ghcdif_args(N, X,Y) :- /* N>0, not yet dif(Xn,Yn) */
 84      N1 is N-1, ghcdif_args(N1, X,Y).



    %%% SYSTEM PREDICATES %%%
    % System predicates are executed even if the reduction counter indicates 0.

    %% Input and Output %%
    % Instream and outstream can be called only once for each.

 85  :-public instream/8.
 86  :-mode instream(?, +,+,-,+,+,+,+).
 87  instream(S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
 88      '$START_IO'(instream, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).

 89  :-public outstream/8.
 90  :-mode outstream(?, +,+,-,+,+,+,+).
 91  outstream(S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
 92      '$START_IO'(outstream, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).

 93  :- public '$START_IO'/9.
 94  :- mode '$START_IO'(+, ?, +,+,-,+,+,+,+).
 95  '$START_IO'(IO, _, _, _, _, _,  _,    _,   _     ) :-
 96      recorded('$GHCIOUSING',IO,_), !,
 97      display('Error: Instream/outstream cannot be called twice.'),
 98      ttynl, restore_standard_io, abort.
 99  '$START_IO'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
100      /* not_recorded('$GHCIOUSING',IO,_), !, */
101      recorda('$GHCIOUSING',IO,_),
102      '$IO'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).

103  :- public '$IO'/9.
104  :- mode '$IO'(+, ?, +,+,-,+,+,+,+).
105  '$IO'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
106      ghcspied(IO,1),
107      functor(G,IO,1), arg(1,G,S), trace_call(Myid, G), !,
108      '$IO?'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).
109  '$IO'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :-
110      '$IO!'(IO, S, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid).

111  :- mode '$IO?'(+, ?, +,+,-,+,+,+,+).
112  '$IO?'(IO, S, RC,Ch,Ct, _,  RCmax,_,   Nextid) :-
113      nonvar(S), ulist(S,H,T), nonvar(H), do_io(H,IO), !,
114      Nextid1 is Nextid+1,
115      '$IO'(IO, T, RC,Ch,Ct,nd,RCmax,Nextid,Nextid1).
116  '$IO?'(_,  S, _ ,Ch,Ct,_,  _,    _,   Nextid) :-
117      nonvar(S), unil(S), !, do_next(Ch,Ct, Nextid).
118  '$IO?'(IO, S, RC,[$(Gx, Ch,Ct,Dnd,Nextid)|Ch],
119                       [$('$IO'(IO,S, RCmax,Ch2,Ct2,Dnd2,RCmax,Myid,Nextid2),
120                                          Ch2,Ct2,Dnd2,            Nextid2)
121                       |Ct],
122                          Dnd,RCmax,Myid,Nextid) :-
123      trace_suspension(RC, Myid), !, incore(Gx).

124  :- public '$IO!'/9.
125  :- mode '$IO!'(+, ?, +,+,-,+,+,+,+).
126  '$IO!'(IO, S, RC,Ch,Ct,_,  RCmax,_,   Nextid) :-
127      nonvar(S), ulist(S,H,T), nonvar(H), do_io(H,IO), !,
```

```
128     Nextid1 is Nextid+1,
129     '$IO!'(IO, T, RC,Ch,Ct,nd,RCmax,Nextid,Nextid1).
130 '$IO!'(_,  S, _ ,Ch,Ct,_,  _,    _,    Nextid) :-
131     nonvar(S), unil(S), !, do_next(Ch,Ct, Nextid).
132 '$IO!'(IO, S, _, [$(Gx, Ch,Ct,Dnd,Nextid)|Ch],
133                     [$('$IO'(IO,S, RCmax,Ch2,Ct2,Dnd2,RCmax,Myid,Nextid2),
134                                        Ch2,Ct2,Dnd2,          Nextid2)
135                     |Ct],
136                        Dnd,RCmax,Myid,Nextid) :- incore(Gx).

137 :- mode do_io(+,+).
138 do_io(see(F),    instream) :- !, atom(F), see(F).
139 do_io(seeing(F), instream) :- !, seeing(F).
140 do_io(seen,      instream) :- !, seen.
141 do_io(tell(F),   _       ) :- !, atom(F), tell(F).
142 do_io(telling(F),_       ) :- !, telling(F).
143 do_io(told,      _       ) :- !, told.

144 do_io(close(F),     _       ) :- !, atom(F), my_close(F).
145 do_io(fileerrors,  _       ) :- !, fileerrors.
146 do_io(nofileerrors,_       ) :- !, nofileerrors.

147 do_io(read(X),   instream) :- !, my_read(X).
148 do_io(write(X),  _       ) :- !, ground(X), write(X).
149 do_io(writeq(X), _       ) :- !, ground(X), writeq(X).
150 do_io(print(X),  _       ) :- !, ground(X), print(X).
151 do_io(nl,        _       ) :- !, nl.
152 do_io(get0(C),   instream) :- !, my_get0(C).
153 do_io(get(C),    instream) :- !, my_get(C).
154 do_io(skip(C),   instream) :- !, ground(C), skip(C).
155 do_io(put(C),    _       ) :- !, ground(C), put(C).
156 do_io(tab(N),    _       ) :- !, ground(N), tab(N).

157 do_io(display(X),_       ) :- !, ground(X), display(X).
158 do_io(ttynl,     _       ) :- !, ttynl.
159 do_io(ttyflush,  _       ) :- !, ttyflush.
160 do_io(ttyget0(C),instream) :- !, my_ttyget0(C).
161 do_io(ttyget(C), instream) :- !, my_ttyget(C).
162 do_io(ttyskip(C),instream) :- !, ground(C), ttyskip(C).
163 do_io(ttyput(C), _       ) :- !, ground(C), ttyput(C).
164 do_io(ttytab(N), _       ) :- !, ground(N), ttytab(N).
165 do_io(G,         _       ) :-
166     display('Error: Illegal request in instream/outstream:  '),
167     writeuser(G), ttynl, restore_standard_io, abort.

    % If nofileerrors has been executed, input goals can be called arbitrarily
    % many times.
168 :- mode my_get0(?), my_ttyget0(?), my_get(?), my_ttyget(?), my_read(?).
169 my_get0(C )     :- get0(C), !.
170 my_get0(26).
171 my_ttyget0(C ) :- ttyget0(C), !.
172 my_ttyget0(26).
173 my_get(C )      :- get(C), !.
174 my_get(26).
175 my_ttyget(C )  :- ttyget(C), !.
176 my_ttyget(26).
177 my_read(X)      :- read(X), !, numbervars(X, 0,_).     % An input term is
178                                                        % made ground.
```

```
    % the following is for discarding an alternative created by the system
    % predicate close/1 when the file specified by its argument is currently
    % open.
179 :- mode my_close(+).
180 my_close(F) :- close(F), !.


    %% Other System Predicates Callable from Bodies %%

    % The following routine is used only when ':='/2 is called directly from
    % the top level; otherwise ':='/2 is compiled into a spcialized predicate.
181 :- public (:=)/9.
182 :- mode :=(?,?, +,+,-,+,+,+,+).
183 :=(X,Y, _,  _, _, _,  _,    Myid,_     ) :-
184     ghcspied(':=',2), trace_call(Myid, X:-Y), fail.
185 :=(X,Y, _,  Ch,Ct,_,  _,    _,   Nextid) :-
186     numbervars(Y,0,0), call(X2 is Y), !,
187     ubody(X2,X), do_next(Ch,Ct,Nextid).
188 :=(_,_, RC0,_, _, _,  _,    Myid,_     ) :-
189     ghcspied(':=',2), trace_suspension(RC0,Myid), fail.
190 :=(X,Y, _,  [$(Gx, Ch,Ct,Dnd,Nextid) | Ch],
191                [$(:=(X,Y, RCmax,Ch2,Ct2,Dnd2,RCmax,Myid,Nextid2),
192                                  Ch2,Ct2,Dnd2,          Nextid2) | Ct],
193                   Dnd,RCmax,Myid,Nextid) :- incore(Gx).

    % The following routine is used only when '='/2 is called directly from
    % the top level; otherwise '='/2 is compiled into 'ubody/2'.
194 :- public (=)/9.
195 :- mode =(?,?, +,+,-,+,+,+,+).
196 =(X,Y, _,_, _, _,_,Myid,_     ) :-
197     ghcspied('=',2), trace_call(Myid, X=Y), fail.
198 =(X,Y, _,Ch,Ct,_,_,_,   Nextid) :- ubody(X,Y), do_next(Ch,Ct, Nextid).
199
200 :- public prolog/8.
201 :- mode prolog(?, +,+,-,+,+,+,+).
202 prolog(X, _,  _, _, _,  _,    Myid,_     ) :-
203     ghcspied(prolog,1), trace_call(Myid, prolog(X)), fail.
204 prolog(X, _,  Ch,Ct,_,  _,    _,   Nextid) :-  call(X), !,
205     do_next(Ch,Ct,Nextid).
206 prolog(_, RC0,_, _, _,  _,    Myid,_     ) :-
207     ghcspied(prolog,1), trace_suspension(RC0, Myid), fail.
208 prolog(X, _,  [$(Gx, Ch,Ct,Dnd,Nextid) | Ch],
209                  [$(prolog(X, RCmax,Ch2,Ct2,Dnd2,RCmax,Myid,Nextid2),
210                                    Ch2,Ct2,Dnd2,          Nextid2) | Ct],
211                     Dnd,RCmax,Myid,Nextid) :- incore(Gx).


    %% Calling the Next Goal in the Queue %%

212 :- mode do_next(+,-, +).
213 do_next([$(Gx, Ch2,Ct,nd,Nextid)|Ch2],Ct, Nextid) :-  incore(Gx).



    %%% CYCLE MARKERS %%%

    %%  Standard Version %%

214 :- public '$END'/4.
215 :- mode '$END'(?,-,+,+).
```

```
'$END'([],_, _, _     ) :- !. % succeeds if the queue is empty, i.e.,
                              % the first arg is UNINSTANTIATED.
'$END'([$(Gx, Ch,Ct,d,Nextid)|Ch],
          [$('$END'(Ch2,Ct2,Dnd2,Nextid2),Ch2,Ct2,Dnd2,Nextid2)|Ct],
             nd,Nextid) :- !,
    incore(Gx).               % otherwise, if some goals have been reduced
                              % in the last 'cycle' (i.e., the deadlock
                              % flag is 'nd'), then try  another 'cycle'.
'$END'(Gs,_, _, _     ) :- backtrace(Gs), fail.


    %% Cycle Marker with Spy Facilities %%

:- public '$ENDSPY'/5.
:- mode '$ENDSPY'(+, ?,-,+,+).
'$ENDSPY'(Endid, Ch,Ct,Dnd,Nextid) :-
    display('Cycle '), display(Endid),
    display('; function (h for help)? '), ttyflush, ttyget0(C),
  ( C=\=31/*newline*/, !, ttyskip(31); true ),
    endspy(C, Endid, Ch,Ct,Dnd,Nextid).

:- mode endspy(+, +, ?,-,+,+).
endspy(31 /*NL*/, Endid, Ch,Ct,Dnd,Nextid) :- !,          % CONTINUE
    endspy2(Endid, Ch,Ct,Dnd,Nextid).
endspy(119/*w*/,  Endid, Ch,Ct,Dnd,Nextid) :- !,          % WRITE QUEUE
    display_queue(Ch), '$ENDSPY'(Endid, Ch,Ct,Dnd,Nextid).
endspy(110/*n*/,  _    , Ch,Ct,Dnd,Nextid) :- !,          % NODEBUG MODE
    '$END'(Ch,Ct,Dnd,Nextid).
endspy(104/*h*/,  Endid, Ch,Ct,Dnd,Nextid) :- !,          % HELP
    display('<cr>: continue'), ttynl,
    display('   w: Write goals in the queue'), ttynl,
    display('   n: continue in Nodebug mode'), ttynl,
    display('   h: Help'), ttynl,
    display('   a: Abort current execution'), ttynl,
    display('   @: Accept a Prolog goal'), ttynl,
    '$ENDSPY'(Endid,Ch,Ct,Dnd,Nextid).
endspy(97 /*a*/,  _,     _, _, _,  _     ) :- !,          % ABORT
    restore_standard_io, abort.
endspy(64 /*@*/,  Endid, Ch,Ct,Dnd,Nextid) :- !,          % ACCEPT COMMAND
    display('| :- '), ttyflush, seeing(F), see(user), read(G), see(F),
  ( call(G), !, display('yes'); display('no') ), ttynl,
    '$ENDSPY'(Endid, Ch,Ct,Dnd,Nextid).
endspy(_,         Endid, Ch,Ct,Dnd,Nextid) :- /* !, */ % Other chars are
    endspy(104, Endid, Ch,Ct,Dnd,Nextid).              % interpreted as 'h'

:- mode endspy2(+, ?,-,+,+).
endspy2(_,     [],_, _, _     ) :- !.
endspy2(Endid, [$(Gx, Ch,Ct,d,Nextid)|Ch],
                  [$('$ENDSPY'(Endid2, Ch2,Ct2,Dnd2,Nextid2),
                                       Ch2,Ct2,Dnd2,Nextid2)|Ct],
                     nd,Nextid) :- Endid2 is Endid+1, !, incore(Gx).
endspy2(_,     Gs,_, _, _     ) :- backtrace(Gs), fail.


    %%% TRACING AND BACKTRACING %%%

:- public
    (ghcspy)/0, (ghcspy)/1, (ghcnospy)/0, (ghcnospy)/1, ghcspying/0,
    ghcbacktrace/0, ghcnobacktrace/0.
```

```
%% Tracing %%

% Setting, Resetting and Showing Spy Points

ghcspy :-
  ( recorded('$GHCSPY',X,_), var(X), !,
      display('Spy points have already been set on all goals.'), ttynl;
    ( erase1, fail; recorda('$GHCSPY',_,_) ) ).
:- mode ghcspy(?).
ghcspy(X) :- atom(X), !,
  ( recorded('$GHCSPY',X/V,_), var(V), !, complain_double_spy(X);
    ( erase1(X), fail; recorda('$GHCSPY',X/_,_) ) ).
ghcspy(X) :- nonvar(X), X=P/A, atom(P), integer(A), !,
  ( recorded('$GHCSPY',X,_), !, complain_double_spy(X);
    recorda('$GHCSPY',X,_) ).
ghcspy(X) :- nonvar(X), X=(X1,X2), !, ghcspy(X1), ghcspy(X2).
ghcspy(X) :- complain_predicate_spec(X).

ghcnospy :-
  ( erase1, !, ( erase1, fail; true ); complain_nothing_to_reset ).
:- mode ghcnospy(?).
ghcnospy(X) :- atom(X), !,
  ( erase1(X), !, ( erase1(X), fail; true );
    recorded('$GHCSPY',Y,_), var(Y), !, complain_too_specific(X);
    complain_nothing_to_reset(X) ).
ghcnospy(X) :- nonvar(X), X=P/A, atom(P), integer(A), !,
  ( erase1(P,A), !;
    recorded('$GHCSPY',Y,_), Y=P/V, var(V), !,
      complain_too_specific(X);
    complain_nothing_to_reset(X) ).
ghcnospy(X) :- nonvar(X), X=(X1,X2), !, ghcnospy(X1), ghcnospy(X2).
ghcnospy(X) :- complain_predicate_spec(X).

erase1 :- recorded('$GHCSPY',_,R), erase(R).
:- mode erase1(+).
erase1(P) :- recorded('$GHCSPY',X,R), nonvar(X), X=P/_,  erase(R).
:- mode erase1(+,+).
erase1(P,A) :- recorded('$GHCSPY',X,R), nonvar(X), X=P/B, A==B, erase(R).

:- mode complain_double_spy(+).
complain_double_spy(X) :-
    display('Spy point on '), writeuser(X),
    display(' has already been set.'), ttynl.
:- mode complain_predicate_spec(?).
complain_predicate_spec(X) :-
    display('Illegal predicate specification:  '), writeuser(X), ttynl.
complain_nothing_to_reset :-
    display('No GHC goals are spied.'), ttynl.
:- mode complain_nothing_to_reset(+).
complain_nothing_to_reset(X) :-
    display('Spy point has not been set on '),
    writeuser(X), display('.'), ttynl.
:- mode complain_too_specific(+).
complain_too_specific(X) :-
    display('Spy point cannot be reset selectively on '),
    writeuser(X), display('.'), ttynl.

ghcspying :- setof($(X), R^recorded('$GHCSPY', X, R), S), !,
  ( S=[$(Y)|_], var(Y), !, display('All GHC goals are spied.');
    display('GHC spy-points have been set on '), write_spied(S) ), ttynl.
```

```
317 ghcspying :- display('No GHC goals are spied.'), ttynl.

318 :- mode write_spied(+).
319 write_spied([$(P/A)|Ps]) :- display(P),
320    ( nonvar(A), !, ttyput("/"), display(A); true ),
321    ( Ps\==[], !, display(', '), write_spied(Ps); ttyput(".") ).

    % Runtime Support for Tracing

322 :- mode ghcspied(+,+).
323 ghcspied(P,A) :-  recorded('$GHCSPY',P/A,_), !.

324 :- mode trace_call(+, +).
325 trace_call(Id, G) :-
326     numbervars(G,0,_), display_id(Id), display(': Call '),
327     writeuser(G), ttynl, fail.
328 trace_call(_,  _).
329
330 :- mode display_id(+).
331 display_id(0 ) :- !.
332 display_id(Id) :- /* Id>0, !, */ display(Id).

333 :- mode trace_suspension(+, +).
334 trace_suspension(0, Id) :- !,
335     display_id(Id), display(': Swapped out'), ttynl.
336 trace_suspension(N, Id) :- /* N>0, !, */
337     display_id(Id), display(': Suspended'), ttynl.


    %% Backtracing %%

    % Setting and Resetting
338 ghcbacktrace   :- recorda('$GHCBACKTRACE',_,_).
339 ghcnobacktrace :- recorded('$GHCBACKTRACE',_,R), erase(R), fail.
340 ghcnobacktrace.

    % Runtime Support
341 :- mode backtrace(+).
342 backtrace(Gs) :-
343     recorded('$GHCBACKTRACE',_,_),
344     display('Deadlock detected.  Suspended goals are:'), ttynl,
345     display_queue(Gs), recorda('$GHCBACKTRACE_END',_,_).


    %% Runtime Support for Displaying the Goal Queue %%

346 :- mode display_queue(+).
347 display_queue(Gs) :- numbervars(Gs,0,_), dq2(Gs), fail.
348 display_queue(_ ).

349 :- mode dq2(+).
350 dq2([G|Gs]) :- !, display_goal(G), dq2(Gs).
351 dq2(_).

352 :- mode display_goal(+).
353 display_goal($(Gx, _,_,_,_)) :-
354     functor(Gx,P,A7), A is A7-7, functor(G,P,A),
355     copy_args(A, Gx,G),
356     A6 is A+6, arg(A6,Gx,Id), display_id(Id),
357     display(': '), writeuser(G), ttynl.
```

```
    %%% MISCELLANEOUS %%%

:- mode ground(?).
ground(X) :- var(X), !, fail.
ground(X) :- /* !, */ functor(X,_,Arity), ground_args(Arity, X).

:- mode ground_args(+, +).
ground_args(0, _) :- !.
ground_args(N, X) :- /* !, */
    arg(N,X,T), ground(T), N1 is N-1, ground_args(N1, X).

reset_ioflag :- recorded('$GHCIOUSING',_,R), erase(R), fail.
reset_ioflag.

save_standard_io :-
    seeing(In),   recorda('$GHCSTDIN', In, _),
    telling(Out), recorda('$GHCSTDOUT',Out,_).

restore_standard_io :- fileerrors,
    recorded('$GHCSTDIN', In, Ref1), erase(Ref1), see(In),
    recorded('$GHCSTDOUT',Out,Ref2), erase(Ref2), tell(Out).

    %%%% COMPILATION %%%%

    %%% READING PROGRAMS %%%

:- public ghccompile/1, ghccompile/2, ghcdcompile/1, ghcdcompile/2.
:- mode ghccompile(?), ghccompile(?,?), ghcdcompile(?), ghcdcompile(?,?).
ghccompile(S) :- /* !, */
    ghccompile(S, '-temp-.ghc', nodepth), compile('-temp-.ghc').
ghccompile(S,O) :- /* !, */ ghccompile(S,O, nodepth).
ghcdcompile(S) :- /* !, */
    ghccompile(S, '-temp-.ghc', depth),   compile('-temp-.ghc').
ghcdcompile(S,O) :- /* !, */ ghccompile(S,O, depth).

ghccompile(S,O, Depth) :-
    telling(Old), nofileerrors, tell(O), !,
    ghccomp(S, [],Dict, Depth),
    write_last_clause_group(Dict, Depth),
    told, tell(Old), display('END.'), ttynl.
ghccompile(_,O, _    ) :- /* failed to open O */
    complain_unable_to_open(O).

    % ghccomp(Files, Current_dict,New_dict, Depth first?).
    % Each member of the predicate dictionary has the form
    % $$(Functor/Arity, Name_of_internal_subpredicate,
    %    Mode_decl, Debug_mode, #_of_body_assignment+"@").
:- mode ghccomp(?, +,-, +).
ghccomp(F,     Dict0,Dict1, _    ) :- var(F), !,
    Dict0=Dict1, complain_unable_to_open(F).
ghccomp(F,     Dict0,Dict1, Depth) :-
    atom(F), seeing(Old), nofileerrors, see(F), !,
    read(X), ghccomp1(X, Dict0,Dict1, nodebug, Depth), seen, see(Old).
ghccomp(F?,    Dict0,Dict1, Depth) :-
    atom(F), seeing(Old), nofileerrors, see(F), !,
    read(X), ghccomp1(X, Dict0,Dict1, debug,   Depth), seen, see(Old).
ghccomp([H|T], Dict0,Dict2, Depth) :- !,
    ghccomp(H, Dict0,Dict1, Depth), ghccomp(T, Dict1,Dict2, Depth).
ghccomp([],    Dict0,Dict1, _    ) :- !, Dict0=Dict1.
ghccomp(F,     Dict0,Dict1, _    ) :- /* failed to open F */
```

```
401     Dict0=Dict1, complain unable_to_open(F).

402 :- mode ghccompl(?, +,-, +, +).
403 ghccompl(X,                Dict0,Dict1, Debug, Depth) :- var(X), !,
404     display('Uninstantiated clause found'), ttynl,
405     read(Next), ghccompl(Next, Dict0,Dict1, Debug, Depth).
406 ghccompl(end_of_file,      Dict0,Dict1, _,     _     ) :- !, Dict0=Dict1.
407 ghccompl((:- mode Mdecl), Dict0,Dict2, Debug, Depth) :- nonvar(Mdecl), !,
408     functor(Mdecl, P, A),
409   ( lookup_dict(P/A, Dict0, _,_,_), !,
410       errmsg('Duplicate or misplaced mode declaration: ', Mdecl),
411       Dict1=Dict0;
412     notify_compilation(P,A), write_mode_decl(P,A),
413   ( check_mode(Mdecl), !,
414       write_transmit_clause(Mdecl), make_internal_name(P,IP),
415       write_mode_decl(IP,A);
416     IP=P ),
417     write_callmsg_clause(IP,A,P, Debug),
418     Dict1=[$$(P/A, IP,Mdecl,Debug,64/*@*/)|Dict0] ),
419     read(Next), ghccompl(Next, Dict1,Dict2, Debug, Depth).
420 ghccompl((:- op(Priority,Type,Op)), Dict0,Dict1, Debug, Depth) :-
421     op(Priority,Type,Op), !,                  % Operator decl must be exe-
422     write_clause((:- op(Priority,Type,Op))), % cuted and also copied.
423     read(Next), ghccompl(Next, Dict0,Dict1, Debug, Depth).
424 ghccompl((:- X),       Dict0,Dict1, Debug, Depth) :-
425  /* not_functor(X, mode, 1), not_functor(X, op, 3) */ !,
426     errmsg('Illegal directive: ', (:- X)),
427     read(Next), ghccompl(Next, Dict0,Dict1, Debug, Depth).
428 ghccompl(X,            Dict0,Dict3, Debug, Depth) :-
429  /* clause_with_positive_atom(X), !, */
430   ( X=(Head:-_), !; X=Head ),
431     functor(Head,P,A),
432   ( lookup_dict(P/A, Dict0, _,_,_), !, Dict1=Dict0;
433     notify_compilation(P,A),
434     write_mode_decl(P,A), write_callmsg_clause(P,A,P, Debug),
435     Dict1=[$$(P/A, P,'$NOMODE',Debug,64/*@*/)|Dict0] ),
436   ( c_clause(X, C, Dict1,Dict2, Debug, Depth), !, write_clause(C);
437     errmsg('Cannot compile: ', X), Dict2=Dict1 ),
438     read(Next), ghccompl(Next, Dict2,Dict3, Debug, Depth).

439 :- mode complain_unable_to_open(?).
440 complain_unable_to_open(F) :- /* !, */
441     fileerrors, display('Cannot open file: '), writeuser(F), ttynl.



    %%% WRITING OBJECT CLAUSES %%%

442 :- mode write_clause(?).
443 write_clause(X) :-
444     make_writable(X,Y, 0,_), writeq(Y), put("."), nl, fail.
445 write_clause(_).

446 :- mode make_writable(?,-, +,-).
447 make_writable(X,          Y, V0,V1) :- var(X), !,
448     X='$VAR'(V0), Y=X, V1 is V0+1.
449 make_writable('$VAR'(X), Y, V0,V1) :- !, Y='$VAR'(X), V0=V1.
450                                % Variable already made ground ('numbervar'ed)
451 make_writable('$REF'(X), Y, V0,V1) :- !, make_writable(X,Y, V0,V1).
452 make_writable('$WAIT'(X),Y, V0,V1) :- !, make_writable(X,Y, V0,V1).
```

```
453 make_writable('$INT'(X), Y, V0,V1) :- !, make_writable(X,Y, V0,V1).
454 make_writable(X,          Y, V0,V1) :- /* nonvariable(X), !, */
455     functor(X,F,A), functor(Y,F,A), make_writable_args(0,A, X,Y, V0,V1).

456 :- mode make_writable_args(+,+, +,+, +,-).
457 make_writable_args(K,N, X,Y, V0,V2) :- K<N, !,
458     K1 is K+1, arg(K1,X,Xk), make_writable(Xk,Yk, V0,V1), arg(K1,Y,Yk),
459     make_writable_args(K1,N, X,Y, V1,V2).
460 make_writable_args(N,N, _,_, V0,V1) :- /* !, */ V0=V1.


    %%% MAKING ADDITIONAL CLAUSES %%%

461 :- mode write_mode_decl(+,+).
462 write_mode_decl(P,A) :- /* !, */
463     A7 is A+7, write_clause((:- public P/A7)),
464     functor(Mode,P,A), fill_questions(A, Mode),
465     extend_atom(Mode,Modex, +,+,-,+,+,+,+), write_clause((:- mode Modex)).

466 :- mode fill_questions(+, +).
467 fill_questions(0, _) :- !.
468 fill_questions(K, X) :- /* K>0, !, */
469     arg(K,X,?), K1 is K-1, fill_questions(K1, X).

470 :- mode write_callmsg_clause(+,+,+, +).
471 write_callmsg_clause(P,A,Pspy, debug) :- !,
472     functor(H,P,A), extend_atom(H,Headx, _, , ,_,_,Myid,_),
473     functor(Hspy,Pspy,A), copy_args(A, H,Hspy),
474     write_clause((Headx :- ghcspied(Pspy,A), trace_call(Myid,Hspy), fail)).
475 write_callmsg_clause(_,_,_,     _     ) /* :- ! */.

476 :- mode write_last_clause_group(+, +).
477 write_last_clause_group([$$(P/A, P, _,Debug,_)|L], Depth) :- !,
478     write suspmsg_clause(P, A, P, Debug, Depth),
479     write_susp_clause(P,A),
480     write_last_clause_group(L, Depth).
481 write_last_clause_group([$$(P/A, IP,_,Debug,_)|L], Depth) :- /*P\==IP,*/ !,
482     write_callsuspmsg_clause(P,A,P, Debug, Depth),
483     write_susp_clause(P,A),
484     write_suspmsg_clause(IP,A,P, Debug, Depth),
485     write_susp_clause(IP,A),
486     write_last_clause_group(L, Depth).
487 write_last_clause_group([],                          _     ) /* :- ! */.

488 :- mode write_suspmsg_clause(+,+,+, +, +).
489 write_suspmsg_clause(P,A,Pspy, debug, Depth) :- !,
490     functor(H,P,A), extend_atom(H,Headx, RC0,_,_,_,_,Myid,_),
491   ( Depth==nodepth, !, RC=RC0; RC=1 ),
492     write_clause((Headx :- ghcspied(Pspy,A),
493                            trace_suspension(RC,Myid), fail)).
494 write_suspmsg_clause(_,_,_,     _,       _     ) /* :- ! */.

495 :- mode write_callsuspmsg_clause(+,+,+, +, +).
496 write_callsuspmsg_clause(P,A,Pspy, debug, Depth) :- !,
497     functor(H,P,A), extend_atom(H,Headx, RC0,_,_,_,_,Myid,_),
498     functor(Hspy,Pspy,A), copy_args(A, H,Hspy),
499   ( Depth==nodepth, !, RC=RC0; RC=1 ),
500     write_clause((Headx :- ghcspied(Pspy,A), trace_call(Myid,Hspy),
501                            trace_suspension(RC,Myid), fail)).
```

```
502 write_callsuspmsg_clause(_,_,_,     _,          ) /* :- ! */.

503 :- mode write_susp_clause(+,+).
504 write_susp_clause(P,A) :- /* !, */
505     functor(H,P,A),
506     extend_atom(H,Goalx, RCmax,Ch2,Ct2,Dnd2,RCmax,Myid,Nextid2),
507     extend_atom(H,Headx, _,     [$(Gx, Ch,Ct,Dnd,Nextid)|Ch],
508                                     [$(Goalx, Ch2,Ct2,Dnd2,Nextid2)|Ct],
509                                         Dnd, RCmax,Myid,Nextid ),
510     write_clause((Headx :- incore(Gx))).

511 :- mode write_transmit_clause(+).
512 write_transmit_clause(Mdecl) :- /* !, */
513     functor(Mdecl,P,A), make_internal_name(P,IP), A7 is A+7,
514     functor(Headx,P,A7), functor(Goalx,IP,A7),
515     transmit_args(0,A7, Mdecl, Headx, Goalx,Body),
516     write_clause((Headx :- Body)).

    % transmit_args(Arg_position,Arity, Mode, Head, Subgoal,Body)
    % Body = suquence of calls to 'nonvar/1' followed by Subgoal.
517 :- mode transmit_args(+,+, +, +, ?,-).
518 transmit_args(K,N, Mdecl, Headx, Goalx,Body0) :- K<N, !,
519     K1 is K+1, arg(K1,Headx,Xk1), arg(K1,Goalx,Xk1),
520   ( arg(K1,Mdecl,+), !, Body0=(nonvar(Xk1), Body1);
521     Body0=Body1 ),                    % Mode is not '+' or K1>arity(Mdecl)
522     transmit_args(K1,N, Mdecl, Headx, Goalx,Body1).
523 transmit_args(N,N, _,     Headx, Goalx,Body ) :- /* !, */
524     N6 is N-6, arg(N6,Headx,Xn6), Body=(Xn6>0, !, Goalx).

525 :- mode make_internal_name(+,-).
526 make_internal_name(P,IP) :- /* !, */
527     name(P,Pl), name(IP,[36,36,36/*$$$*/|Pl]).



    %%% COMPILING CLAUSES %%%

    %% Decomposing and Composing Clauses %%

    % c_clause(GHC_clause, Prolog_clause,
    %          Current_dict, New_dict, Debug_mode?, Depth_first?).
    % c_clause may fail if an uninstantiated atom appears.
528 :- mode c_clause(+, -, +, +, ?, +).
529 c_clause((Head:-_),     _, _,     _,     _,          ) :- var(Head), !,
530     fail.
531 c_clause((Head:-X),     C, Dict0,Dict1, Debug, Depth) :-
532     nonvar(X), X=(Guard|Body), !,
533     c_head_guard(Head, Guard, PH, PB0,(!,PB1), Dict0, RC0, Depth),
534     extend_atom(PH,PHx, RC0,Ch,Ct,_,RCmax,_,Nextid),
535     separate_unify(Body, Bnounify, Bunify),
536     c_body_unify(Bunify, Nextid,0,Id1, PB1,PB2, Debug),
537     c_body_arith(Bnounify,Bnounify_rest, PB2,PB3, Debug),
538     put_queue(Bnounify_rest, Q0,Q1, RC1,RCmax,
539               PB3,PB4, Nextid,Id1,Id2, Head, Dict0,Dict1, Debug),
540     c_update_nextid(Debug, Nextid,Id2,Newid, PB4,PB5),
541     c_make_tail(Depth, Q0,Q1, RC0,RC1, Ch,Ct, Newid, PB5),
542     C=(PHx:-PB0).
543 c_clause((Head:-Body), C, Dict0,Dict1, Debug, Depth) :-
544  /* not_functor(Body, '|', 2) */ !,
545     errmsg('Warning: No commitment operator:  ', (Head :- Body)),
```

```
546     c_clause((Head :- true|Body), C, Dict0,Dict1, Debug, Depth).
547 c_clause(Head,          C, Dict0,Dict1, Debug, Depth) :-
548  /* not_functor(Head, ':-', 2) */ !,
549     errmsg('Warning: No commitment operator:  ', Head),
550     c_clause((Head :- true|true), C, Dict0,Dict1, Debug, Depth).

    % separate_unify(Goals, Nounify, Unify).
    % Nounify and Unify become LINEAR conjunction of goals.
551 :- mode separate_unify(?, -, -).
552 separate_unify(G, NU, U) :- separate_unify(G, NU,true, U,true).

553 :- mode separate_unify(?, -,?, -,?).
554 separate_unify(X,      _,  _,   _, _ ) :- variable(X), !, fail.
555 separate_unify((X,Y), NU0,NU2, U0,U2) :- !,
556     separate_unify(X, NU0,NU1, U0,U1), separate_unify(Y, NU1,NU2, U1,U2).
557 separate_unify(X=Y,   NU0,NU1, U0,U1) :- !, NU0=NU1, U0=(X=Y,U1).
558 separate_unify(G,     NU0,NU1, U0,U1) :- /* not_unification(G), !, */
559     NU0=(G,NU1), U0=U1.


    %% Compiling Heads and Guards %%

    % c_head_guard(GHC_head, GHC_guard,
    %     Prolog_head, Prolog_body, Prolog_body_tail, Current_dict,
    %     Current_reduction_count, Depth_first?)
560 :- mode c_head_guard(+, ?, -, -,+, +, -, +).
561 c_head_guard(Head, Guard, PH, PB0,PB4, Dict, RC, Depth) :-
562     functor(Head,P,A), lookup_dict(P/A, Dict, IP,Mode,_),
563     functor(PH,IP,A),
564   ( separate_unify(Guard, Gnounify, Gunify), !,
565     ( call(Gunify), !,
566         c_args(0,A, Head, PH, PB0,PB1, Mode),
567       ( c_guard(Gnounify, PB1,PB2, PB3,PB4), !,
568         ( P==IP, Depth==nodepth, !,
569             PB2=(RC>0,PB3);
570           PB2=PB3 );
571         errmsg('Guards can call predefined predicates only:  ', Guard),
572         PB1=(fail,PB4) );
573       errmsg('Warning: Unsucceedable guard:  ', Guard),
574       PB0=(fail,PB4) );
575     errmsg('Uninstantiated variables in guard:  ', Guard),
576     PB0=(fail,PB4) ).

577 :- mode c_args(+,+, +, -, -,?, +).
578 c_args(K,N, Head, PH, PB0,PB2, M) :- K<N, !,
579     K1 is K+1, arg(K1,Head,Ak), arg(K1,PH,PHk),
580   ( M=='$NOMODE', !, Mk=(?);  arg(K1,M,Mk) ),
581     c_unify(Ak, PHk, PB0,PB1, Mk), c_args(K1,N, Head, PH, PB1,PB2, M).
582 c_args(N,N, _,     _,  PB0,PB1, _) :- /* !, */ PB0=PB1.

    % c_unify(Original_argument, Generated_argument,
    %         Unify_code, Unify_code_tail, Mode).
    % Note that f(X,X) must be compiled into f(X,Y) :- X==Y.
    % This is because X and Y must be unified with no substitutions.
    % For integer comparison, =:= and =\= are faster than = and \=.
583 :- mode c_unify(?, -, -,?, +).
584 c_unify(V,         X, G0,G1, M) :- var(V), !,
585     X=V, G0=G1, V='$REF'(Y), mark(Y, M).
586 c_unify('$REF'(V), X, G0,G1, M) :- !, G0=(V==X,G1), mark(X, M).
587 c_unify([],        X, G0,G1, M) :- !,
```

```
588   ( M=='+', !, X=[], G0=G1; G0=(nonvar(X),unil(X),G1) ).
589 c_unify(A,          X, G0,G1, M) :- atomic(A), !,
590   ( M=='+', !, X=A, G0=G1; G0=(nonvar(X),uskel(X,A),G1) ).
591 c_unify([H|T],      X, G0,G3, M) :- !,
592   ( M=='+', !, X=[H0|T0], G0=G1; G0=(nonvar(X),ulist(X,H0,T0),G1) ),
593     c_unify(H, H0, G1,G2, ?), c_unify(T, T0, G2,G3, ?).
594 c_unify(S,          X, G0,G2, M) :- /* other_structure(S), !, */
595     functor(S,F,A), functor(S0,F,A),
596   ( M=='+', !, X=S0, G0=G1; G0=(nonvar(X),uskel(X,S0),G1) ),
597     c_unify_args(0,A, S, S0, G1,G2).

598 :- mode c_unify_args(+,+, +, +, -,?).
599 c_unify_args(K,N, S, X, G0,G2) :- K<N, !,
600     K1 is K+1, arg(K1,S,Sk), arg(K1,X,Xk),
601     c_unify(Sk, Xk, G0,G1, ?), c_unify_args(K1,N, S, X, G1,G2).
602 c_unify_args(N,N, _, _, G0,G1) :- /* !, */ G0=G1.

    % c_guard(Original goals, Before_goals, Before_goals_tail,
    %         After_goals, After_goals_tail)
    % Before_goals are called before checking the reduction counter, while
    % After_goals are not.
603 :- mode c_guard(+, -,?, -,?).
604 c_guard(true,      Gb0,Gb1, Ga0,Ga1) :- !, Gb0=Gb1, Ga0=Ga1.
605 c_guard((X,Y),     Gb0,Gb2, Ga0,Ga2) :- !,
606     c_guard(X, Gb0,Gb1, Ga0,Ga1), c_guard(Y, Gb1,Gb2, Ga1,Ga2).
607 c_guard(prolog(X), Gb0,Gb1, Ga0,Ga1) :- !, Gb0=Gb1, Ga0=(X,Ga1).
608                       % prolog(X) must be executed after checking the
609                       % reduction counter, since it may cause side effects.
610 c_guard(X,          Gb0,Gb1, Ga0,Ga1) :- /* other_form(X), !, */
611     c_guard_2(X, Gb0,Gb1), Ga0=Ga1.

    % c_guard_2 fails if the first arg is not a system predicate.
612 :- mode c_guard_2(+, -,?).
613 c_guard_2(wait(X), Gb0,Gb1) :- !, c_w(X, Gb0,Gb1).
614 c_guard_2(X\=Y,    Gb0,Gb1) :- atomic(X), !, c_w(Y, Gb0,(X\==Y,Gb1)).
615 c_guard_2(X\=Y,    Gb0,Gb1) :- atomic(Y), !, c_w(X, Gb0,(X\==Y,Gb1)).
616 c_guard_2(X\=Y,    Gb0,Gb1) :- /* non_atomic(X), non_atomic(Y) */ !,
617     Gb0=(ghcdif(X,Y),Gb1).
618 c_guard_2(X:=Y,    Gb0,Gb2) :- appeared_in_head(X), !,
619     c_iw(X, Gb0,Gb1), c_iw_all(Y, Gb1,(X is Y,Gb2)).
620 c_guard_2(X:=Y,    Gb0,Gb1) :- /* not_appeared_in_head(X) */ !,
621     c_iw_all(Y, Gb0,(X is Y,Gb1)).
622 c_guard_2(X<Y,     Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X<Y,  Gb1)).
623 c_guard_2(X>Y,     Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X>Y,  Gb1)).
624 c_guard_2(X=<Y,    Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X=<Y, Gb1)).
625 c_guard_2(X>=Y,    Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X>=Y, Gb1)).
626 c_guard_2(X=:=Y,   Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X=:=Y,Gb1)).
627 c_guard_2(X=\=Y,   Gb0,Gb1) :- !, c_iw_2(X, Y, Gb0,(X=\=Y,Gb1)).

628 :- mode c_w(?, -,?).
629 c_w(X,          G0,G1) :- var(X), !, G0=(nonvar(X),G1), X='$WAIT'(_).
630 c_w('$REF'(X), G0,G1) :- !, c_w(X, G0,G1).
631 c_w(X,          G0,G1) :-
632  /* ( X='$WAIT'(_); X='$INT'(_); not_variable(X) ), !, */ G0=G1.

633 :- mode c_iw(?, -,?).
634 c_iw(X,          G0,G1) :- var(X), !, G0=(integer(X),G1), X='$INT'(_).
635 c_iw('$REF'(X), G0,G1) :- !, c_iw(X, G0,G1).
636 c_iw('$WAIT'(X), G0,G1) :- !, c_iw(X, G0,G1).
637 c_iw(_,          G0,G1) :- /* ( X='$INT'(_); not_variable(X) ), ! */ G0=G1.
```

```
638 :- mode c_iw_2(?,?, -,?).
639 c_iw_2(X,Y, G0,G2) :- /* !, */ c_iw_all(X, G0,G1), c_iw_all(Y, G1,G2).

640 :- mode c_iw_all(?, -,?).
641 c_iw_all(X,          G0,G1) :- var(X), !, G0=(integer(X),G1), X='$INT'(_).
642 c_iw_all('$REF'(X),  G0,G1) :- !, c_iw_all(X, G0,G1).
643 c_iw_all('$WAIT'(X), G0,G1) :- !, c_iw_all(X, G0,G1).
644 c_iw_all('$INT'(_),  G0,G1) :- !, G0=G1.
645 c_iw_all(X,          G0,G1) :- /* structure(X), !, */
646     functor(X,_,A), c_iw_args(0,A, X, G0,G1).

647 :- mode c_iw_args(+,+, ?, -,?).
648 c_iw_args(K,N, X, G0,G2) :- K<N, !,
649     K1 is K+1, arg(K1,X,Xk), c_iw_all(Xk, G0,G1),
650     c_iw_args(K1,N, X, G1,G2).
651 c_iw_args(N,N, _, G0,G1) :- /* !, */ G0=G1.


    %% Compiling Bodies %%

652 :- mode c_body_unify(+, -,+,-, -,?, +).
653 c_body_unify((X-Y,Bunify), Id,Id0,Id2, PB0,PB3, Debug) :- !,
654     c_id_setup(Debug, PB0,PB1, Id,Id0,Id1, Myid),
655   ( variable(X), !, R=X, L=Y;  % for efficiency on DEC-10 Prolog
656     L=X, R=Y ),
657   ( Debug==nodebug, !, PB1=(ubody(L,R), PB2);
658     PB1=('ubody?'(L,R,Myid), PB2) ),
659     c_body_unify(Bunify, Id,Id1,Id2, PB2,PB3, Debug).
660 c_body_unify(true,          _, Id0,Id1, PB0,PB1, _    ) :- /* !, */
661     Id0=Id1, PB0=PB1.

662 :- mode c_id_setup(+, -,?, ?,+,-, -).
663 c_id_setup(nodebug, Ig0,Ig1, _, Id0,Id1, Myid) :- !,
664     Myid=0, Ig0=Ig1, Id1 is Id0 .
665 c_id_setup(debug,   Ig0,Ig1, Id,Id0,Id1, Myid) :- /* !, */
666     ( Id0==0, !, Myid=Id, Ig0=Tg1; Tg0=(Myid is Id+Id0, Ig1) ),
667     Id1 is Id0+1.

    % Initial sequence of assignments that need no waiting is processed on the
    % spot.
668 :- mode c_body_arith(+,-, -,?, +).
669 c_body_arith((V:-E,Body0),Body1, PB0,PB2, nodebug) :-
670     c_iw_all(E, W,true), W=true, !,  % check if waiting is needed for E
671   ( integer(E), !, PB0=(ubody(E,V), PB1);
672     PB0=(V2 is E, ubody(V2,V), PB1) ),
673     c_iw(V, _,_),                    % mark V as 'having an integer value'
674     c_body_arith(Body0,Body1, PB1,PB2, nodebug).
675 c_body_arith(Body0,       Body1, PB0,PB1, _      ) :-
676  /* top element of Body0 is not a non-suspending assignment, !, */
677     Body0=Body1, PB0=PB1.

    % Put_queue is called also at run time to make a goal queue.  It fails if
    % there exists an uninstantiated goal.
    % put_queue(Body goals, Goal_queue_head, Goal_queue_tail,
    %           New_reduction_count, Max_reductions,
    %           Id_calc_goal_head, Id_calc_goal_tail,
    %           Next_id, Initial_id_offset, Final_id_offset,
    %           Head, Dict_old, Dict_new, Debug_mode?)
678 :- mode put_queue(?, -,-, ?,?, -,?, ?,+,-, +, +,-, +).
679 put_queue(X,      _, _,  _,_,_,_,_, _,  _,   _,    _,    _,     _    ) :-
```

```
680     variable(X), !, fail.
681 put_queue((X,Y), Q0,Q2, RC1,RCmax,
682                       Ig0,Ig2, Id,Id0,Id2, Head, Dict0,Dict2, Debug) :- !,
683     put_queue(X, Q0,Q1, RC1,RCmax,
684                       Ig0,Ig1, Id,Id0,Id1, Head, Dict0,Dict1, Debug),
685     put_queue(Y, Q1,Q2, RC1,RCmax,
686                       Ig1,Ig2, Id,Id1,Id2, Head, Dict1,Dict2, Debug).
687 put_queue(true,  Q0,Q1, _, _,
688                       Ig0,Ig1, _, Id0,Id1, _,    Dict0,Dict1, _    ) :- !,
689     Q0=Q1, Ig0=Ig1, Id0=Id1, Dict0=Dict1.
690 put_queue(V:=E,  Q0,Q1, RC1,RCmax,
691                       Ig0,Ig1, Id,Id0,Id1, Head, Dict0,Dict1, Debug) :-
692     Head\=='$TOP', !,
693     make_arith_head(Head, V,E, AH,AP,AA, Dict0,Dict1),
694     write_mode_decl(AP,AA),
695     extend_atom(AH,AHx, RC1,Ch,Ct,Dnd,RCmax,Myid,Nextid),
696   ( Debug==debug, !,
697       write_clause((AHx :- ghcspied(':=',2), trace_call(Myid,V:=E), fail));
698     true ),
699     c_iw all(E, G,(V2 is E, !, ubody(V2,V), do_next(Ch,Ct,Nextid))),
700     write_clause((AHx :- G)),
701   ( Debug==debug, !,
702       write_clause((AHx :- ghcspied(':=',2),
703                             trace_suspension(1,Myid), fail));
704     true ),
705     write_susp_clause(AP,AA),
706     Q0=[$(AHx, Ch,Ct,Dnd,Nextid) | Q1],
707     c_id_setup(Debug, Ig0,Ig1, Id,Id0,Id1, Myid).
708 put_queue(X,      Q0,Q1, RC1,RCmax,
709                       Ig0,Ig1, Id,Id0,Id1, _,    Dict0,Dict1, Debug) :-
710  /* user_defined_goal(X), !, */
711     extend_atom(X,Xx, RC1,Ch,Ct,Dnd,RCmax,Myid,Nextid),
712     Q0=[$(Xx, Ch,Ct,Dnd,Nextid) | Q1],
713     c_id_setup(Debug, Ig0,Ig1, Id,Id0,Id1, Myid),
714     Dict0=Dict1.

715 :- mode make_arith_head(+, ?,?, -,-,-, +,-).
716 make_arith_head(Head, V,E, AH,AP,AA, Dict0,Dict1) :-
717     functor(Head,P,A), inc_arith(P/A, Dict0,Dict1, Arithid),
718     name(P,Pl), name(AP,[36,36,36,Arithid|Pl]),
719     analyze_exp(E, L,[]),
720     AH =.. [AP,V|L], functor(AH,_,AA).

721 :- mode analyze_exp(?, -,?).
722 analyze_exp(E, L0,L1) :- variable(E), !, L0=[E|L1].
723 analyze_exp(E, L0,L1) :- /* not_variable(E), !, */
724     functor(E, _, A), analyze_exp_args(0,A, E, L0,L1).

725 :- mode analyze_exp_args(+,+, +, -,?).
726 analyze_exp_args(K,N, E, L0,L2) :- K<N, !,
727     K1 is K+1, arg(K1,E,Ek), analyze_exp(Ek, L0,L1),
728     analyze_exp_args(K1,N, E, L1,L2).
729 analyze_exp_args(N,N, _, L0,L1)  :- /* !, */ L0=L1.

730 :- mode c_update_nextid(+, -,+,-, -,-).
731 c_update_nextid(debug, Nextid,Id1,Newid, PB4,PB5) :- Id1>0, !,
732     PB4=(Newid is Nextid+Id1, PB5).
733 c_update_nextid(Debug, Nextid,Id1,Newid, PB4,PB5) :-
734  /* ( Debug==nodebug; Id1=<0 ), !, */
735     Newid=Nextid, PB4=PB5.
```

```
    % c_make_tail(Depth_first?, Body_queue_head,Body_queue_tail,
    %              Reduction_count_old,Reduction_count_new,
    %              Cont_h,Cont_t, New_goal_id, Prolog_body).
736 :- mode c_make_tail(+, ?, -,-,-,-,-,-,-).
737 c_make_tail(_,     H,          T, _,  _,   Ch,Ct, Newid, PB5) :- H==T, !,
738     PB5=(Ch=[$(Gx, Ch1,Ct,nd,Newid)|Ch1], incore(Gx)).
739 c_make_tail(Depth, [Qitem|H1],T, RC0,RC1, Ch,Ct, Newid, PB5) :- /* !, */
740     c_update_rc(Depth, RC0,RC1, PB5,PB6),
741     Qitem= $(PB6, H1,Ct,nd,Newid),
742     T=Ch.                    % The d-list [Qitem|H1]-T is connected to Ch-Ct

743 :- mode c_update_rc(+, -,-, -,-).
744 c_update_rc(nodepth, RC0,RC1, PB5,PB6) :- !,
745     PB5=(RC1 is RC0-1, PB6).
746 c_update_rc(depth,   RC0,RC1, PB5,PB6) :- /* !, */
747     PB5=PB6, RC1=RC0.



    %%% MISCELLANEOUS ROUTINES %%%

    %% Predicate Dictionary Management %%

748 :- mode lookup_dict(+, +, -,-,-).
749 lookup_dict(Pred, [$$(Pred, IP,Mode,Debug,_)|_], IP,Mode,Debug) :- !.
750 lookup_dict(Pred, [Item|Dict],                    IP,Mode,Debug) :-
751  /* Item=$$(Pred2, _,_,_,_), Pred\==Pred2, !, */
752     lookup_dict(Pred, Dict, IP,Mode,Debug).

753 :- mode inc_arith(+, +,-, -).
754 inc_arith(Pred, Dict0,Dict1, Arithid) :- /* !, */
755     find_item(Pred, Dict0,Dict_rest, Item),
756     Item= $$(Pred, IP,Mode,Debug,Arithid0), Arithid is Arithid0+1,
757     Dict1=[ $$(Pred, IP,Mode,Debug,Arithid)|Dict_rest].

758 :- mode find_item(+, +,-, -).
759 find_item(Pred, [I|Dict],Dict_rest, Item) :- arg(1,I,Pred), !,
760     Dict_rest=Dict, Item=I.
761 find_item(Pred, [I|Dict], Dict_rest, Item) :- /* not_arg(1,I,Pred), !, */
762     Dict_rest=[I|Dict_rest1], find_item(Pred, Dict,Dict_rest1, Item).


    %% Constructing Atomic Formulas %%

763 :- mode extend_atom(+, -, ?,?,?,?,?,?,?).
764 extend_atom(X, Xx, RC,Ch,Ct,Dnd,RCmax,Myid,Nextid) :- /* !, */
765     functor(X,F,A),
766     A1 is A+1, A2 is A+2, A3 is A+3, A4 is A+4,
767     A5 is A+5, A6 is A+6, A7 is A+7,
768     functor(Xx,F,A7), copy_args(A,X,Xx),
769     arg(A1,Xx,RC),    arg(A2,Xx,Ch),   arg(A3,Xx,Ct),     arg(A4,Xx,Dnd),
770     arg(A5,Xx,RCmax), arg(A6,Xx,Myid), arg(A7,Xx,Nextid).

771 :- mode copy_args(+, +,+).
772 copy_args(0, _,_ ) :- !.
773 copy_args(K, X,Xx) :- /*K>0, !,*/
774     arg(K,X,Xk), arg(K,Xx,Xk), K1 is K-1, copy_args(K1, X,Xx).


    %% Analyzing Mode Declaration %%
```

```
    % check_mode(Mdecl) succeeds if Mdecl contains the '+' mode, fails
    % otherwise.  Also, it complains of modes other than '+' and '?'.
775 :- mode check_mode(!).
776 check_mode(Mdecl) :- /* !, */
777     functor(Mdecl,_,A), check_mode_args(0,A, Mdecl, no).
778 :- mode check_mode_args(+,+, +, +).
779 check_mode_args(K,N, Mdecl, YN0) :- K<N, !,
780     K1 is K+1, arg(K1,Mdecl,M), check_mode_arg(M, YN0,YN1),
781     check_mode_args(K1, N, Mdecl, YN1).
782 check_mode_args(N,N, _,     yes) /* :- ! */.  % fails if 4th arg is 'no'.

783 :- mode check_mode_arg(?, +,-).
784 check_mode_arg(+, _,  YN1) :- !, YN1=yes.
785 check_mode_arg(?, YN0,YN1) :- !, YN1=YN0.
786 check_mode_arg(M, YN0,YN1) :- /* !, */
787     display('Illegal mode specifier: '), display(M), ttynl, YN1=YN0.


    %% Analyzing Variables in Clauses %%

788 :- mode variable(?).
789 variable(X          ) :- var(X), !.
790 variable('$REF'( ) ) :- !.
791 variable('$WAIT'(_)) :- !.
792 variable('$INT'(_) ) /* :- ! */.

793 :- mode appeared_in_head(?).
794 appeared_in_head(X          ) :- var(X), !, fail.
795 appeared_in_head('$REF'(_)) /* :- ! */.

796 :- mode mark(-, +).
797 mark(X, +) :- !, X='$WAIT'(_).
798 mark(_, M) /* :- M\==+, ! */.


    %% Displaying Messages %%

799 :- mode notify_compilation(+,+).
800 notify_compilation(P,A) :-
801     display(P), ttyput("/"), display(A), display(', '), ttyflush.

802 :- mode errmsg(+, ?).
803 errmsg(Msg, Clause) :- /* !, */
804     telling(File), tell(user), nl,
805     write(Msg), write_clause(Clause), nl, tell(File).

806 :- mode writeuser(?).
807 writeuser(T) :- /* !, */ telling(File), tell(user), write(T), tell(File).
```

（付録３）

Quintus Prologへの移植の手引

本システムをQuintus Prologに移植するときは，次のことに注意して下さい：

① DEC-10 Prolog とQuintus Prologでは，特殊文字のコードに違いがあります．本システムのソースプログラムの中には，次に２種類のコードが整定数として書いてあるので，それらを修正する必要があります：

	DEC-10 Prolog	Quintus Prolog
ファイルの終わりを示すコード	26	-1
改行記号	31	10

② DEC-10 Prolog とQuintus Prologでは，使用できる入出力操作に多少の違いがあります．本システム中の述語 'do_io'/2 が，組込述語'instream'/1と'outstream'/1 に与えられた各操作が正しいものかどうかを検査し，実際の入出力を行なっているので，この述語を適当に修正する必要があります．

③ DEC-10 Prolog とQuintus Prologでは，算術式の構文に多少の違いがあります（たとえば，DEC-10 Prolog では整数の除算に'/' を使いますが，Quintus Prologでは'//' を使います）．しかし，本システムは，算術式の解析をPrologまかせにしているので，システム自身を修正する必要はありません．

④ 目的コードを構成する各述語の節は，目的ファイルの中で連続して現れません．Quintus Prologでは，このような述語に対してコンパイル時（本システムの用語ではローディング時）に警告メッセージを出すので，それを抑止するために，適当な宣言（たとえば':- no_style _check(all)' ）が目的コードの先頭に現れるようにコンパイラを直す必要があります．

⑤ ロギング出力を抑止するための述語'nolog'/0 は，Quintus Prologでは呼ぶ必要がありません．

⑥ 本システム自身の目的コードを保存するのには，Quintus Prologでは'pisys'/1 ではなくて'save'/1を使う必要があります．

⑦ 本システム自身と，それが生成するPrologの目的コードは，DEC-10 Prolog 用のmode宣言とpublic宣言を行なっています．これらはあってもかまいませんが，Quintus Prologでは単に無視されます．

DEC-10 Prolog 上のＧＨＣシステムとQuintus Prolog上のＧＨＣシステムの言語仕様上の違いは，Prolog処理系の違いからくる上記①，②，③の３点です．