# Gödel Users' Manual

(Release 1.3, June 1993)

A.F. Bowers and J. Wang

Department of Computer Science
University of Bristol
University Walk
Bristol BS8 1TR
UK

email: {bowers, jiwei}@compsci.bristol.ac.uk

# Contents

# 1 Introduction

This document explains the accompanying implementation of the Gödel language. It also serves as a supplement to *The Gödel Programming Language* by P.M. Hill and J.W. Lloyd, concentrating on the practical use of the Gödel Language.

The supplied Gödel system is implemented in SICStus Prolog. The Gödel parser runs at about 120 lines per sec. on a SPARC 2. Gödel programs are transformed to SICStus Prolog and compiled to native code. Without any delay declarations, negations or IF-THEN-ELSE constructs, Gödel programs run at the same speed as equivalent SICStus programs (about 600K LIPS on a SPARC 2). A carefully written Gödel program (with all features) should run at about half of the speed of SICStus.

Please send bug reports to goedel@compsci.bristol.ac.uk.

All suggestions for improvement are welcome.

# 2 Using the Gödel system

To use the Gödel system, you need to run the executable file called 'goedel'. (Instructions for installing the Gödel system can be found in the README file in the top level directory of this release.) If you do not have SICStus Prolog, your Gödel system will be based on the SICStus runtime system. There are a few differences between this *Runtime Gödel* and Gödel based on full SICStus Prolog, and these will be indicated where appropriate. Having entered the Gödel system, you will see a prompt '`[] <-`' at which you may type commands or Gödel queries. Commands always begin with a semi-colon (;) character to distinguish them from queries. All commands and Gödel queries end with '.' followed by a carriage return (CR).

The commands available in the Gödel system are presented below. Some commands take one or two parameters. The parameters are either a BigName (a sequence of alphanumeric characters starting with a capital letter, e.g. `Module`) or a String (any sequence of characters enclosed in double quotes, e.g. `"File"`). The syntax of each command is given in brackets after its name. The commands are divided into four groups: basic commands and commands for compiling and loading Gödel programs; utilities for performing other operations on Gödel files; debugging utilities, and an interface to Unix.

## 2.1 Basic system commands

**Help** (`;h.`) Gives rudimentary help information and lists the abbreviations that may be used.

**Quit** (`;quit.`) This command returns you to the Unix shell.

**Compile** (`;c Module.`) This compiles the given module. In compiling a module, the Gödel system looks for files whose name consists of the module name and the extension '.loc' or '.exp'. Error and warning messages are printed out should there be any syntax errors. Programs with warning messages may still be correct, but our advice is "don't ignore any warning message". There may be several error messages for a single error, due to Gödelś overloading mechanism. Not all the error messages are relevant, but it's advisable to read through them all.

If there is no error in the program, three files will be generated. The names of the files consist of the module name and extensions '.lng', '.pl' or '.ql'. (If you are using Runtime Gödel, only the '.lng' and '.pl' files are generated. In this case, the '.pl' file is not compatible with that generated by a normal Gödel system.) The '.lng' file contains the language of the module. The '.pl' file is the compiled Gödel code in Prolog; and the '.ql' file is the '.pl' file compiled by SICStus.

If the given module imports some other modules, and those modules have already been compiled, the ';c' command will load their '.lng' files instead of parsing those modules.

**Make** (`;m Module.`) This compiles the given module, and also compiles all the user modules it depends upon, whether or not these have already been compiled..

**Load** (`;l Module.`) This loads the compiled module and the user modules it depends upon from their '.lng' and '.ql' files ('.pl' files in the case of Runtime Gödel). Absence of any of these files causes the load command to abort. After loading a Gödel program, the system prompt changes to '`[Module] <-`', where 'Module' is the name of the main module in the program.

**Make and Load** (`;ml Module.`) This makes and loads the module into the system. Any error causes the `;ml` command to abort.

**Check Syntax** (`;cs Module.`) This command checks the syntax of the given module but does not create any files. It is useful to determine the syntactical correctness of an incomplete program.

**Save States** (`;save File Goal.`) This instructs Gödel to save the current state (the loaded Gödel program) into the executable file `File`. When `File` is run, Gödel will start the execution of `Goal`. This command is not available in the Runtime Gödel.

## 2.2 Gödel utilities

**Program Compile** (`;pc Module.`) This creates the ground representation of the specified program whose main module is `Module` in a file whose name consists of the main module name and '.prm' extension. If there is any syntax error in the specified module and its dependent user modules, the ground representation of the module is not created.

**Program Decompile** (`;pd "File".`) This takes the ground representation of a program from the file whose name consists of the module name and '.prm' extension, and generates source files for all the component user modules of this program. Where '.exp' and '.loc' files already exist for one of these modules, they will be renamed with '.exp.old' and '.loc.old' extensions respectively, so that the existing source is not immediately destroyed.

**Script Compile** (`;sc Script.`) This compiles a generated script in the file whose name consists of the script name and '.scr' extension and produces '.lng', '.pl' and '.ql' files so as to be loaded for execution. If you are using Runtime Gödel, the '.ql' file is not created.

**Flock Compile** (`;fc "File" "Flock".`) This takes a flock and returns a Gödel internal representation of the flock in a file whose name consists of the flock name and '.flk' extension. If there is an existing flock of the same name, the existing flock will be overwritten. If there is any syntax error in the file of units, the ';fc' commands aborts.

**Flock Decompile** (`;fd "Flock" "File".`) This takes a Gödel flock and returns a file of units. The flock should be in a file with the name consisting of the flock name and '.flk' extension. If there is an existing file of the same name, the existing file will be overwritten.

**Canonicalise Prolog** (`;cp "File1" "File2".`) This utility converts Prolog clauses or terms in File1 into their canonical form in File2. Variable names are not preserved.

**Decanonicalise Prolog** (`;dp "File1" "File2".`) This converts Prolog clauses or terms in canonical form in File1 into standard form in File2.

## 2.3 Debugging facilities

**Debug Compile** (`;dc Module.`) This has the same effect as the ';compile' command except that it also incorporates debugging information into the compiled code for the given module. Debug compiled modules run much more slowly.

**Trace** (`;trace.`) Switches on the tracer.

**Notrace** (`;notrace.`) Switches off the tracer.

**Spy** (`;spy Predicate.`) Sets spy points at all predicates or propositions with declared name `Predicate`, regardless of their arity and the module in which they are declared.

**Nospy** (`;nospy Predicate.`) Removes spy points at all predicates or propositions with declared name `Predicate`, regardless of their arity and the module in which they are declared.

**Nospyall** (`;nospyall.`) Removes all spy points.

**Type** (`;t Symbol.`) Displays the declarations of all the constants, functions, propositions and predicates in the goal language of the loaded program that have the declared name `Symbol`.

## 2.4 Interface to Unix

**cd** (`;cd "Directory".`) Invokes the Unix 'cd' command. The 'Directory' argument must be a string. If it is omitted, ';cd' changes to the user's home directory.

**ls** (`;ls "Directory".`) Invokes the Unix 'ls' command. The 'Directory' argument must be a string. If it is omitted, ';ls' lists the current directory.

**more** (`;more "File".`) Invokes the Unix 'more' command for the specified file.

**Unix** (`;unix "Command"`) Execute a Unix command. If 'Command' is omitted, this enters a Unix shell.

**pwd** (`;pwd.`) Invokes the Unix 'pwd' command.

## 2.5 Executing queries

A Gödel query should be a goal in the goal language of the program whose main module is the current module. The Gödel system attempts to solve the query and return answer bindings for the free variables appearing in it. If the query finitely fails, 'No' is printed. If it can be solved with the empty answer substitution, 'Yes' is printed. Otherwise, the answer substitution appears in the form:

```
x = Term,
```

for each free variable in the query. The system then prompts with a `?`. Typing in `;` after the question mark causes the system to search for another solution. A carriage return (CR) returns you to the main prompt.

An example session of using the Gödel system follows.

```
% goedel
Goedel 1.3
Type ;h. for help.
[] <- ;l Lists.
[Lists] <- Append(x, y, [1, 3]).

x = [],
y = [1,3] ? ;

x = [1],
y = [3] ? ;

x = [1,3],
y = [] ? ;
No
[Lists] <- ;l Integers.
[Integers] <- x^2 + y^2 = z^2  &  0 < x < 50  &  0 < y < 50  &  0 < z.

x = 3,
y = 4,
z = 5 ? ;

x = 4,
y = 3,
z = 5 ? ;

x = 5,
y = 12,
z = 13 ?
Yes
[Integers] <- ;ml EightQueens.  % the EightQueens program can be found in
```

```
                                % released directory program/demo/
Reading file "EightQueens.loc" ...
Parsing module "EightQueens" ...
Compiling module "EightQueens" ...
Module "EightQueens" compiled.
Loading module "EightQueens" ...
[EightQueens] <- Queen(x).

x = [1,5,8,6,3,7,2,4] ? ;

x = [1,6,8,3,7,4,2,5] ? ;

x = [1,7,4,6,8,2,5,3] ?
Yes
[EightQueens] <- ;q.
%
```

# 3 Use of the Gödel tracer

There are two modes of tracing in the Gödel system, namely, *trace* and *spy*. Trace starts tracing from the query, while spy only triggers the tracing at specified predicates. Trace and spy work only with debug-compiled modules (see **Debug Compile** in the previous section).

When the tracer is invoked, execution of the `call`, `exit`, `fail` and `redo` ports of each debug-compiled procedure will be displayed. Each `call` has an unique index and each `exit`, `fail` and `redo` has an index corresponding to their original `call`. At each entry, you are prompted with `?` at which the following commands can be used.

```
a - abort:     abort current query.
c - continue:  continue execution without trace info.
f - free:      let the tracer run free and print out all the trace info.
h - help:      print this message.
l - leap:      leap to the next spy point.
n - next:      next goal, the same as a carriage return.
r - redo:      reenter the failed goal, valid only in "fail" entries.
s - skip:      skip the trace info for executing the current goal.
1-9:           set term display depth in the tracer.
```

Suspended and awakened goals are also printed out under entries `suspend` and `awaken`, respectively. In a similar way to `call`, each `suspend` also has an unique index. Each `awaken` has an index corresponding to its `suspend`.

In this release, the tracer does not mask off terms defined in system modules as abstract data types. In a future release, a more sophisticated debugging tool will provide facilities for properly displaying terms in an abstract data type.

The Gödel tracer displays variables in the format `_N`. Variables in a suspended goal are renamed when the goal is awoken. Module prefixes are attached to the predicate with a `:`, e.g. `Lists:Append([1,2], [3,4], _12345)`. Syntactic sugars are sometimes displayed in the sugared form, e.g. lists and sets, and sometimes in the original form, e.g. the `Interval/3` function in the `Integers` module. Constraints are replaced by a variable and cannot be seen in the trace. This can be seen in the following example.

```
[Integers] <- ;trace.
Tracing on
[Integers] <- x^2 + y^2 = z^2 & 1 < x < 9 & 1 < y < 9.
   0 call: _34205=_33909 ?
   0 exit: _34710=_34710 ?
   1 call: Integers:Interval(2,_34104,8) ?
   1 exit: Integers:Interval(2,2,8) ?
   2 call: Integers:Interval(2,_34008,8) ?
   2 fail: Integers:Interval(2,_34008,8) ?
   1 redo: Integers:Interval(2,2,8) ?
   1 exit: Integers:Interval(2,3,8) ?
   3 call: Integers:Interval(2,_34008,8) ?
```

```
   3 exit: Integers:Interval(2,4,8) ?

x = 3,
y = 4,
z = 5 ?
```

Users of the Gödel tracer need to be aware that Gödel programs are converted into normal form using the Lloyd-Topor transformations (see Section 4.1 and [Lloyd 87] p. 113) and it is the normal forms which are traced.

Commits are invisible in the Gödel tracer. Furthermore, commits are 'switched off' when solving a negative goal (to be elaborated in the next section). The tracer does not give any indication about commits being switched off.

The 'redo' command in the Gödel tracer is limited to failed goals, due to implementation difficulties. However, it is recommended that programmers use the non-failing style of programming. That is, predicates are designed to succeed in the intended computation, therefore the failure of a goal indicates program errors. In this case, the 'redo' command should be useful to investigate the errors. The Gödel tracer differs from the Prolog tracer in that it does not have 'break'. It is probably not necessary to have 'break' in the Gödel system, because one cannot modify the running program from the break level.

# 4 Computational model

The accompanying implementation of the Gödel language maps Gödel programs into Prolog. This has allowed us to build prototypes of the language quickly by using existing logic programming techniques. However, the computational model of the Gödel system is limited by the existing implementation of Prolog.

This implementation of the Gödel system uses a variant of SLDNF-resolution as its procedural semantics. In this section, we shall try to explain this computational model.

## 4.1 Lloyd-Topor transformation

Let a program statement be `Head <- Body`, and a goal be `<- Goal`. Gödel allows arbitrary first-order formulas in `Body` and `Goal`. In order to execute Gödel programs on an SLDNF-resolution based system, Gödel programs have to be transformed using the Lloyd-Topor transformation [Lloyd 87, p113]. Because `W1 \/ W2` and `~SOME [x1, ..., xn] W` are handled by the underlying system, the transformation we use is syntactically different from that in [Lloyd 87], but semantically the same. The difference is mainly that no new clauses are generated in the transformation.

1. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \vee \neg W) \wedge W_{i+1} \wedge \ldots \wedge W_m$

2. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \leftrightarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \rightarrow W) \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$

3. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n W \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

4. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \forall x_1 \ldots \forall x_n W \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg\exists x_1 \ldots \exists x_n \neg W \wedge W_{i+1} \wedge \ldots \wedge W_m$

5. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg(V \wedge W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (\neg V \vee \neg W) \wedge W_{i+1} \wedge \ldots \wedge W_m$

6. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg(V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg V \wedge \neg W \wedge W_{i+1} \wedge \ldots \wedge W_m$

7. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg(V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge \neg V \wedge W_{i+1} \wedge \ldots \wedge W_m$

8. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg\neg W \wedge W_{i+1} \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_m$

9. Replace $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg\forall x_1 \ldots \forall x_n W \wedge \ldots \wedge W_m$
   by $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_n \neg W \wedge \ldots \wedge W_m$

## 4.2 Safe negation

The safeness condition is imposed in the Gödel system. In other words, negated subformulas of a goal are delayed until they contain no free variables. Furthermore, to ensure the soundness of the negation as failure rule, commits are disabled inside the execution of a negated goal.

These two restrictions are also applied to the execution of the condition in IF-THEN-ELSE constructs. This is because `IF Cond THEN ThenPart ELSE ElsePart` is semantically equivalent to `Cond & ThenPart \/ ~Cond & ElsePart`.

## 4.3 Computation rule

The computation rule in the Gödel system, called the *faithful computation rule*, generally selects the leftmost unsuspended goal, subject to the following caveats:

- constraints may be solved in any order that the system finds convenient;
- in some circumstances the order in which suspended goals are woken is undefined (due to a limitation of SICStus Prolog).

If all the goals are suspended, the computation flounders.

Once the execution enters the scope of a commit, it must solve the scope entirely before pruning can occur. If no literal in the scope can be selected, so that execution must leave the scope of the commit before solving it completely, then that commit becomes permanently disabled, that is it will never prune even if its scope is eventually solved.

The order in which clauses are selected in the execution of a Gödel program is not defined.

## 4.4 Constraint solving

Implementation of constraint solving in the modules `Integers` and `Rationals` is primitive. The basic idea in the implementation is to use the coroutining mechanism to suspend constraints until they become solvable. Sophisticated constraint solving methods are not supported yet.

It's hard to define what is currently available. Generally, this release supports:

1. Evaluating expressions with data structures defined in system modules `Integers`, `Rationals`, `Strings` and `Sets`. For example, the following procedures can be executed.

```
PREDICATE  P : Integer.
P((2*3+4) Div 6).

PREDICATE  Sum : Set(Integer) * Integer.
Sum({}, 0).
Sum(set, sub_total + x) <-
   x In set &
```

```
   Sum(set\{x}, sub_total).

PREDICATE  AddPostfix : String * String.
AddPostfix(file_name, file_name ++ ".prm").
```

2. Generating integers in an interval. E.g. `1 < x =< 5`.

3. Solving linear equations which are in triangular form. For example,

```
2*x+1 = y+2 & 3*y-2 = 1.

x/2 + y/3 = 5/6 & y/2 + 1/3 = 5/6.
```

4. Exhaustive search with coroutining.

```
x^2 + y^2 = z^2 & 1 < x < 50 & 1 < y < 50 & 0 < z.
```

## 4.5 Set unification

Full set unification is not implemented, because it would be too inefficient to support it at the Prolog level. Unification can be performed for ground sets, and a variable and a set; attempts to do more complex set unifications will flounder. For example,

```
[Sets] <- {1,2,3} = {3,2,1,2}.
Yes
[Sets] <- {1,2,3} = {3,2,1,4}.
No
[Sets] <- x = {3,2,1,2,3}.

x = {1,2,3} ?
Yes
```

Set operations which contains non-ground sets will be delayed until the non-ground sets become ground. For example,

```
[Sets] <- x In ({y} + {z}) & y=1 & z=2.

x = 1,
y = 1,
z = 2 ? ;

x = 2,
y = 1,
z = 2 ? ;
No
[Sets] <- x In {y} + {z} & y=1.
Floundered.  Unsolved goals are:
[user:sort([_37844],_38505),user:union([1],_38505,_38507),Sets:In(_37544,_38507)]
```

# 5 Some hints for good programming in Gödel

Following are some hints on good programming style and how to effectively use the Gödel system.

- Gödel programs are very close to logic specifications. Usually the program itself serves as the specification. It is highly recommended that the naming of variables and symbols be done properly in the sense that the name reflects the intended interpretation of the object. Sometimes, it is difficult to give an object a good name. But remember that if you cannot find a name for an idea, the idea is probably wrong.

- Singleton variables are most often produced by misspelling and unfinished clauses. So do not ignore singleton variable warnings and unused quantified variable warnings.

- Be careful of the limitations of safe negation, and note that all pruning is disabled within the condition of an IF-THEN-ELSE as well as within explicitly negated calls. It is better to write procedures to be determinate wherever possible without using commit to enforce determinacy. Experience suggests that it is a bad idea to use the failure of a procedure to return information, even the information that an error occurred, because testing for failure may compromise the efficiency of the procedure. Return error indications in arguments instead. Failure is best reserved for indicating programming errors.

- How to find out why your program is floundering.

  It is our experience that if the logic of a program is correct the program should run. When a goal flounders, it is useful to look at the quantifiers, negations, IF-THEN-ELSE constructs and intensional sets. The floundering message may give you some clue which part of program to look at. Particularly, local variables in the condition part of IF-THEN-ELSE should all be quantified. Beware also of attempts to unify nonground sets.

- How to write more efficient Gödel programs.

  The underlying Prolog system supports first argument indexing and tail recursion optimisation. Because the mapping from Gödel to Prolog is direct, the Gödel system does too. When properly used, first argument indexing and tail recursion optimisation can improve the speed by a factor of anything from 2 to 100.

- What are Gödel's counterparts to Prolog *var, nonvar, =.., assert and retract*?

  Prolog's *var, nonvar, assert* and *retract* are not supported in Gödel because they are not declarative. Univ (=..) would break Gödel's type system. However, the meta-logical use of *var and nonvar* is supported by Gödel meta programming facilities, i.e. `Variable/1` and `NonVarTerm/1` in the `Syntax` module.

  It is a common practice among Prolog programmers to write multi-moded predicates (i.e. predicates that can run "backwards") by using *var* to select between several

(hopefully equivalent) logics according to the mode of the call. Because Gödel does not provide *var*, it is not possible to use this trick to have one entry point for several procedures, and only in simple cases can efficient multi-moded procedures be constructed. Where a relation is required to be computed in different modes, the recommended technique is to implement a distinct predicate for each mode, with a name suited to its mode. The modes can be enforced by delay declarations. Of course, this means that the mode must be known at the point of call, but in practical programs this is rarely a problem.

The meta-logical use of Prolog's *assert* and *retract* is provided by Gödel's meta-programming facilities, i.e. `InsertStatement/4` and `DeleteStatement/4` respectively in the `Programs` module. There is no equivalent to *assert* and *retract* for implementing global variables in Gödel; instead add an extra argument to the relevant predicates in order to pass around state information.

# 6 Facilities absent from this release

Several facilities of the Gödel language are not implemented in this release.

1. The occur check is not implemented.
2. Full commit is not supported. Only bar commit and one solution commit are allowed. Co-routining across the bar commit and one solution commit causes the commit to be switched off.
3. Implementation of constraint solving is primitive as has been discussed in Section 4.4.
4. The Gödel system utilities: `script-view`, `theory-compile` and `theory-decompile` are not available.
5. The following system modules are not available: `Floats, Numbers, NumbersIO, Theories,` and `TheoriesIO`.

# 7 Known bugs and limitations

There are a few known bugs or limitations which we have not had time to fix or cannot be fixed in this implementation. These are as follows:

1. The parser is slow at sorting out heavily ambiguous experssions. For example, it takes a few minutes for the parser to parse `x = 1+2+3+4+5+6+7+8+9` in a module which imports `Integers`, `Rationals` and `Sets` modules, because `+` has been declared in all these three modules. Future releases will fix this problem.
2. A string cannot be longer than 512 characters.
3. Arithmetic functions and repeated variables in the head of a DELAY declaration may cause DELAY declarations to behave incorrectly in some rare cases.
4. In both the `Integers` and `Rationals` modules, if you use the power function backwards to compute the root of a number, when the answer is greater than `10^14`, it is usually wrong. For instance,

   ```
   [Integers] <- x^2 = 10^28.
   x = 100000000000000 ?
   Yes
   [Integers] <- x^2 = 10^30.
   No.
   ```

   This is because there is no general algorithm for this class of problems and the inaccurate logarithm function has to be used.
5. The Flock Compile command (`;fc`) cannot parse a single identifier unit written as, e.g. "`proposition.`". However, the command works if there is a layout character in between the identifier and the terminator, i.e. "`proposition .`".
6. No opaque terms are generated by the meta modules. This has little practical consequence, but means that it is sometimes possible to access the internal structure of terms that should be hidden inside closed system modules. Programs that rely on this will cease to work once opaque terms are implemented correctly.
7. The `Succeed` and `Compute` predicates in `Programs` can go wrong in certain very obscure circumstances where the object program itself calls Succeed or Compute on a different object program with the same name.
8. If you use SICStus Prolog 2.1 #6 (or earlier) to compile the Gödel system, you may be affected by a SICStus bug which causes a flounder message to be printed out even when your program has terminated normally. This bug was fixed in SICStus patch #7.

# 8 Acknowledgements

John Lloyd and Pat Hill are the originators of Gödel. Constructive criticisms and testing and debugging efforts were contributed by Dominic Binks, Corin Gurr, Feliks Kluzniak, Nick Moffat and André de Waal. Corin Gurr implemented the `Syntax` module and assisted with the implementation of other meta modules. We thank Mats Carlsson and Stefan Andersson at SICS for their prompt and unfailing technical support.

This work was partly supported by the ESPRIT Basic Research Action 3012 (Compulog) and Project 6810 (Compulog 2), SERC Grants GR/F/26256, and the University of Bristol.

# 9 References

[Hill&Lloyd 92] P.M. Hill & J.W. Lloyd. *The Gödel Programming Language*, Bristol University Technical Report CSTR-92-27, October 1992. Revised May 1993.

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.