A Prolog Benchmark Suite for Aquarius

Ralph Haygood

Computer Science Division
University of California
Berkeley, CA 94720

April 30, 1989

Abstract

This report describes a suite of benchmarks for Prolog implementation research. It includes an explanation of the format of the suite, which is meant to facilitate use of the benchmarks. The principal idea of this format is to maintain for each benchmark a master file from which particular instances - for particular Prolog execution systems, for particular statistics to capture, etc. - are generated automatically using a preprocessor. A preprocessor provided with the suite for this purpose is described, along with a related utility and a simple framework for execution time measurement. Source code for these is appended. Possibilities for future work with respect both to this suite and to Prolog benchmarking more generally are discussed briefly. For each benchmark in the suite, source code and execution times under C Prolog and Quintus Prolog (compiled) on a Sun 3/60 are appended.

1 Introduction

This report describes a suite of Prolog benchmarks compiled for the Aquarius project at Berkeley. This suite is primarily intended as a tool for researchers trying to understand and improve Prolog implementations. It is not specialized to any single Prolog implementation (hardware or software) or application area. It includes benchmarks which have been collected by Aquarius over a period of several years during which the project has undertaken work on many aspects of Prolog implementation and application. [Des87] Though some of the benchmarks originated with members of Aquarius, most came from elsewhere. Many are well-known and have been widely-used to characterize Prolog implementations (see, for example, [Bur87], [DDP85], [Dob87], [DSP85], [NSD88], [Pon89], [Qui88], [Tic86], [Tic87], [War83]); a few, such as the set by David H. D. Warren, can fairly be called classics. Thus, besides the particular history of Aquarius, these benchmarks embody an important share of the experience and wisdom of the logic programming community. It is thus reasonable to hope this suite may be useful not only within Aquarius but beyond it as well.

This is **not** ''the ultimate Prolog benchmark suite.'' An ''ultimate'' benchmark suite, that is, one including an ideal set of benchmarks for every circumstance, is surely impossible. Indeed, no general suite such as this can include a ''complete'' set of benchmarks for nearly any circumstance. Characterizing an implementation of Prolog (or any other non-trivial programming system) is a complex project which requires more than results from a few standard benchmarks. Standard benchmarks well-analyzed can contribute a useful sketch of an implementation. They can provide approximate comparisons of alternative features and can suggest useful possibilities for more implementation-specific benchmarks to probe behavior more closely.

2 The Benchmarks

Appendix A is a catalog of the suite. It gives the name of each benchmark and a brief description of what the benchmark does. This information is also present in the files containing the benchmark code. Appendix G includes a listing of each of these files. Appendix A also notes references which further describe each benchmark and how it has been used.

As Appendix A indicates, the benchmarks are grouped into families, including the `warren` family from David H. D. Warren; the `berkeley` family from the Aquarius project at Berkeley, which has been used extensively to characterize the PLM and its successors [DSP85] [Dob87] [DDP85] [NSD88]; the `gabriel` family, which derives from a suite of Lisp benchmarks compiled by R. P. Gabriel [Gab85]; the `pereira` family from Fernando C. N. Pereira (courtesy of Quintus Computer Systems, Inc.), an outstanding contribution of twenty-six "microscopic" benchmarks which explore a multitude of Prolog implementation issues from structure unification to argument indexing [Bur87] [Qui88]; the `fft` family from Richard A. O'Keefe, which exercises Prolog floating point facilities with the fast fourier transform; the `tp` family from Ross Overbeek, which is a set of propositional theorem proving exercises; and the `asp` family from the ASP (Advanced Silicon compiler in Prolog) group of the Aquarius project, which executes stages in the silicon-compilation of several devices. [Bus88] Other benchmarks include the natural language system front-end `chat_parser` from David H. D. Warren and Fernando C. N. Pereira [WP82]; the intuitionistic logic interpreter `ili` from Seif Haridi [Tic87]; and the `plm_compiler` from Peter Van Roy. [Van84]

These benchmarks share the following characteristics:

- They are, on the whole, well-written, in a variety of accepted programming styles.
- They are nearly all well-known and have been used at Berkeley and elsewhere.
- They use sufficiently "vanilla" Prolog that they will run with little or no modification under most Prolog implementations. In particular, all of them will run under Ċ Prolog (version 1.5) and under Quintus Prolog (version 2.0).*

Several of the benchmarks include code in another language (`gabriel/lisp` [Gab85] and `tp/c`) whose performance can be compared with that of the Prolog code.

C Prolog and Quintus Prolog have been chosen as reference implementations not only because they are well-known and widely-used but because they typify alternatives among such implementations. C Prolog is a well-constructed but plain, non-commercial implementation with few "frills." By contrast, Quintus Prolog is a sophisticated, commercial implementation with features such as compilation, free mixing of compiled and interpreted code, first argument indexing, etc.

Appendix B tabulates execution times for the benchmarks under C Prolog and Quintus Prolog (compiled) on a Sun 3/60. Execution time is certainly not the only statistic relevant to characterizing an implementation, but it is significant, and relative execution times convey some sense of the relative amounts of "work" done by the benchmarks. This is useful for becoming acquainted with the suite.

* Several in the `tp` family fail to run to completion because they use too much memory.

Appendix C tabulates Prolog features used by the benchmarks. It is meant mainly to show what features an implementation must support to run a given benchmark, with emphasis on features which group into "functional sub-domains" of Prolog and which may not be realized in the early stages of a research implementation. Such features include integer or floating-point arithmetic (`is/2`, etc.), structure manipulation (`functor/3`, `arg/3`, `=../2`), and database editing (`assert/1`, `retract/1`, `abolish/2`). For full details about a particular benchmark, of course, it is necessary to examine the benchmark code directly, but Appendix C may provide a useful starting point.

It should be clear that the benchmarks chosen for this suite are by no means the only ones which might have been chosen. Many other possibilities exist. **It is expected that this suite will be revised over time.**

3 The Format

Benchmarks for implementors must run not only in complete, "production" systems (like C Prolog and Quintus Prolog) but in incomplete, experimental systems used for implementation research. For example, at Berkeley we have instruction-level and functional-unit-level simulator programs for various pieces of hardware intended to support high-performance Prolog execution. Such systems typically do not offer users the sort of interaction that conventional Prolog systems offer. In a conventional Prolog system, a benchmark might be invoked with a query to the interpreter like

```
?- run(a, 10, [cputime]).
```

where the form of `run/3` is

```
run( +Name, +Data, +Statistics )*
```

In systems devised for implementation research, however, there may well be no analogous way to specify at run-time what code to run, what data to use, or what statistics to capture. (This is true to various degrees of our simulator programs at Berkeley.) Instead, this information must be "hard-wired" into the code these systems are given to run. Of course, implementors could extend their systems to eliminate such restrictions, but they are likely to consider this a nuisance orthogonal to the thrust of their research.

The loss of flexibility is itself a significant nuisance, however. It is a burden to maintain a separate version of every benchmark for every system, for every statistic to capture, etc. Consistency and documentation are likely to deteriorate. This is especially so when a group of people work together to develop and characterize an implementation, as is commonly the case.

The benchmarks in this suite are set in a format which addresses these issues. The philosophy is to provide for each benchmark a master file from which files for particular cases are generated automatically by a preprocessor. A preprocessor called *pre* is provided with the suite for this purpose. It is designed to support easy and fast specification and documentation of particular cases and to be easy and fast to run. Also, a utility called *MAKE* is provided to expedite invocation of *pre*, and a simple framework, or "bench," for execution time measurement is provided, ready-to-run under systems such as C Prolog and Quintus Prolog. An interface to the bench is provided for each benchmark in the suite. The following sub-sections describe *pre*, *MAKE*, and the bench in more detail and then work through an example of their use with a benchmark from the suite.

* + before an argument indicates the argument is an input to the predicate.

3.1 *pre*

pre is a preprocessor offering, among other things, simple macro assignment and expansion, conditional processing, and file inclusion. Both syntactically and semantically, it is similar to ANSI standard C preprocessors, with omissions and extensions.* *pre* itself is implemented in ANSI standard C with assistance from the lexical analyzer generator LEX and the parser generator YACC. The source code is furnished with the suite and listed in Appendix D. 3.4 below introduces the major features of *pre*, and a manual for *pre* is in preparation. To get some flavor of what files for processing by *pre* look like, the reader may wish to inspect Figure 1 (p. 7) before reading further.

Many existing preprocessors do most of what *pre* does. The specialized features which motivated making *pre* center on facilitating the specification and documentation of "particular cases" of a "general scheme." Here we consider one example, the `option` directive. The `option` directive specifies a list of identifiers and a text string. Each identifier is presumably an "option" for *pre* processing, that is, a macro which, when assigned at *pre* invocation, contributes to deciding which "particular case" *pre* generates. The text string is presumably documentation of the roles and relationships of the specified option identifiers. The significance of this directive is that *pre* has a mode in which it searches its input for `option` directives. When it finds one, it can display the documentation text to the terminal, and/or it can list the specified options, one per line, to the terminal or to a file. The precise action *pre* takes is set by arguments given at invocation. In the first case (invoked by the -D (for Document) command line argument), the result is a list of what a user must know about how to process the file with *pre*: what options there are, and what they mean. This information is thus conveniently available both within the file and on-line upon demand. In the second case (invoked by the -L (for List) command line argument), the list, if directed to the terminal, may serve as abbreviated documentation, or, if directed to a file, may serve for subsequent use by *MAKE*, as described in 3.2 below.

A C preprocessor-like syntax was chosen for *pre* for two main reasons. First, for purposes of a Prolog benchmark suite, it seems desirable that *pre* syntax be easily distinguishable from that of Prolog so that *pre* directives in a benchmark master file stand out clearly from the benchmark code itself. Second, since most prospective users are likely to be familiar with C and hence with C preprocessors, a C preprocessor-like syntax is likely to be easy for them to learn.

It may be wondered why *pre* is implemented in C rather than in Prolog, since it is meant for use with a Prolog benchmark suite and since Prolog is in some obvious ways superior for processing language. The answer has to do primarily with portability. To support some of the features which are desirable for *pre* to offer, a Prolog implementation of *pre* would need to use some non-standard Prolog features. For example, to support a preassigned macro for the date, it would be necessary in most Prolog systems to make a foreign function call to an operating system utility. This sort of thing is done in quite different ways by different Prolog systems. Because there is not any one Prolog system which every potential user of the benchmark suite is likely to have, this is a problem. C, by contrast, is fairly standardized and widely available.†

* The major omissions are: macros with arguments; arithmetic in conditional expressions; the `include <...>` directive; the `line` directive. The first two may be supported in future revisions. Other ANSI standard C preprocessor directives - `define, include "...", error, if, ifdef, ifndef, elif, else, endif` - are supported (`define` as a synonym for `assign` and `elif` as a synonym for `elseif`). Error handling may also be improved in future revisions.

† *pre* is made with help from LEX and YACC, but since these generate C code, not every potential user must have them.

3.2 *MAKE*

MAKE is a utility which meshes with *pre* to expedite processing files with *pre*. It allows the user to specify concisely what file to process, what options to assign, and where to write the output. 3.4 below demonstrates the use of *MAKE*. The source code is included with the suite and listed in Appendix E. *MAKE* began as a C-shell (csh) script; primarily for the sake of speed, it is now implemented in C.

The main convenience *MAKE* provides is prefix expansion for *pre* processing options. If *MAKE* is invoked with the -L (for List) argument, it invokes *pre* with this same argument and writes output from *pre*, a sequence of option identifiers specified by `option` directives, to a file. It can subsequently use this file to expand prefixes of these option identifiers. For example, if `QUINTUS_PL` is an option that determines whether the output from *pre* is specialized for Quintus Prolog, rather than

```
pre -sQUINTUS_PL ...
```
[*]

to produce output thus specialized, it suffices to enter

```
MAKE Q ...
```

if no other option has first letter `Q`. Or, if `STRATEGY` is an option that determines which strategy the output for a state-space search program realizes, rather than

```
pre -aSTRATEGY=depth_first ...
```
[†]

to produce output which realizes depth-first search, it suffices to enter

```
MAKE S=depth_first ...
```

if no other option has first letter `S`. If a given prefix is ambiguous, *MAKE* resolves it to the first identifier in the option list file of which it is a prefix. If a given string is not a prefix of any identifier in the option list file, *MAKE* writes an error message and terminates. (This is meant to discourage the use of "undeclared" options. If users find this disagreeable, they can easily change *MAKE* to do something else, e.g., to pass such strings on to *pre* without modification.) This facility has proven most convenient in practice.

3.3 The Bench

The bench is a framework for measuring Prolog benchmark execution times. It is ready-to-run under C, BIM, Quintus, SB, and SICStus Prolog, and it can easily be extended to other implementations. It is provided primarily as a convenience for becoming acquainted with the benchmark suite or for comparing performance, to the extent of execution time, of experimental systems with well-known standard Prolog systems.

An interface to the bench is provided for each benchmark in the suite. These interfaces are in files separate from the benchmark master files. (They are thus out of the way when the bench is not in use.) The core of each interface is a clause of `benchmark/4` -

```
benchmark( +Name, -Action, -Control, -Iterations )
```
[‡]

---

[*] `-sQUINTUS_PL` means set macro `QUINTUS_PL`, that is, assign it value `1`.

[†] `-aSTRATEGY=depth_first` means assign macro `STRATEGY` value `depth_first`.

[‡] – before an argument indicates the argument is an output from the predicate.

where

> *Name* is the name of the benchmark;
>
> *Action* is a term such that `call(Action)` executes the benchmark;
>
> *Control* is a "dummy" term whose structure and instantiation are identical to those of *Action*;
>
> *Iterations* is the default number of iterations of *Action* and *Control* to execute in a run.*

For example, the clause for the benchmark `nreverse` is

```
benchmark(nreverse,
          nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                    13,14,15,16,17,18,19,20,21,
                    22,23,24,25,26,27,28,29,30],_),
          dummy([1,2,3,4,5,6,7,8,9,10,11,12,
                 13,14,15,16,17,18,19,20,21,
                 22,23,24,25,26,27,28,29,30],_),
          1000).
```

The "active" component of the bench is the "driver" listed in Appendix F. The top-level predicate is `driver/1` -

`driver(+Name)`

where

> *Name* is the name of the benchmark.

This predicate operates as follows:

(1) It calls `benchmark/4` (with the given *Name*) to find out how many *Iterations* of the *Action* and its *Control* to perform;

(2) It calls `get_cpu_time/2`;

(3) It repeats `call(Action)` the specified number of *Iterations*;

(4) It again calls `get_cpu_time/2`;

(5) It repeats `call(Control)` the specified number of *Iterations*;

(6) It yet again calls `get_cpu_time/2`;

(7) It calls `report/6` to which it passes *Name*, *Iterations*, and the results of the three calls to `get_cpu_time/2`.

The *Control* calls thus compensate for the overhead associated with the repetition and with the statistical predicate `get_cpu_time/2`. (If this rather cumbersome description is confusing, look at Appendix F - the code is simple.)

A clause of the form

`benchmark( +Name * +Iterations )`

is also defined to permit the user to vary the number of iterations from that specified in the `benchmark/4` clause.

* This notion of `benchmark/4` was invented by Fernando C. N. Pereira.

```
# /*
  nreverse.m: Warren benchmark nreverse master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

#if BENCH
#  include ".nreverse.bench"
#else
nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],_).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (nreverse/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
nreverse(_,_).
#else
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
#endif
```

Figure 1

The statistical predicate `get_cpu_time/2` is (like most statistical predicates) highly system-dependent. This system-dependency is conveniently taken care of by *pre*. If one of the options `BIM_PL`, `C_PL`, `QUINTUS_PL`, `SB_PL`, or `SICSTUS_PL` is assigned at *pre* invocation, then an appropriate definition for `get_cpu_time/2` is generated automatically. It is easy to add other options for other Prolog systems having mechanisms for obtaining execution time.

The output of a benchmark (if any) is generally unimportant in the context of execution time measurement - thus the second argument of `nreverse/2` in the `benchmark/4` clause above is anonymous. But output is sometimes useful for verifying that a given benchmark is executing completely and correctly. The interface to the bench for most of the benchmarks in the suite includes a clause of `show/1` -

`show(+Name)`

where

Name is the name of the benchmark.

This predicate is designed to show what the benchmark does. For example, the clause for `nreverse` is

```
show(nreverse) :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                            13,14,15,16,17,18,19,20,21,
                            22,23,24,25,26,27,28,29,30],R),
                  write('reverse of'), nl,
                  write([1,2,3,4,5,6,7,8,9,10,11,12,
                         13,14,15,16,17,18,19,20,21,
                         22,23,24,25,26,27,28,29,30]), nl,
                  write(is), nl,
                  write(R), nl.
```

3.4 An Example

We consider the classic benchmark `nreverse` of the `warren` family. The master file for this benchmark is called `nreverse.m`. It resides in a directory which also contains: `.nreverse.bench`, a symbolic link to `set-up.nreverse`, which contains the interface for `nreverse` to the bench; `MAKE`, a symbolic link to the executable *MAKE*; and `.pre`, a symbolic link to the executable *pre*. This scheme and the format of the files `nreverse.m` and `set-up.nreverse` are typical for the whole suite.

Figure 1 (p. 7) is a listing of `nreverse.m`. Lines beginning with `#` are *pre* directive lines. Ordinarily, *pre* directives may not extend over more than one line; however, newlines are permitted within comments, delimited by `/*` and `*/`, and within text strings, delimited by `"`'s.* The first three lines of `nreverse.m` form a header for the file by means of a *pre* comment. This is never written to the output; a header for the output is formed by the next block of eight lines. Two features of these lines warrant attention. On the first line, the preassigned macros `__MDAY__`, `__MONTH__`, and `__YEAR__` are expanded to the numerical day of the month, the name of the month, and the year including the century for the day on which *pre* is run. This generates a "time stamp" on the output, e.g., `30 April 1989`. The preassigned macro `__OPTIONS__` on the second line is expanded to the set of option identifiers assigned at *pre*

* Backslash followed by newline can be used to extend a *pre* directive cosmetically over more than one line. This is allowed anywhere that whitespace is allowed.

invocation - separated by spaces, and followed by = and assigned value for those which are assigned with the -a argument rather than simply set with the -s argument. The idea is to put an "options stamp" on the output. However, `__OPTIONS__` alone would not do this, for as soon as it was expanded to the set of assigned option identifiers, the expansion itself would be scanned for macros, and the option identifiers would be expanded to their values. (*pre* is like C preprocessors in this respect.) The `$identifier$` construct overcomes this obstacle. An identifier such as `__OPTIONS__` surrounded by `$`'s is scanned only once (`$-$` is quasi-mnemonic for "single-scan"); if the identifier has a macro expansion, the expansion is not scanned.

The next block of seven lines takes care of defining the top-level predicate for the benchmark. If option `BENCH` is set, then the interface to the bench in `set-up.nreverse` is included in the output. (Remember, `.nreverse.bench` is a symbolic link to `set-up.nreverse`.) `set-up.nreverse` is listed in Figure 2 (p. 10). The final *pre* directive in this file causes the bench driver to be included in the output. Note that (1) whether `BENCH` is set or not, the top-level predicate for the benchmark is the first executable Prolog in the output, and (2) the name of this predicate is the name of the benchmark, and the arity of this predicate is zero - in this case, it is `nreverse/0`. These are characteristics not only of `nreverse` but of every benchmark in the suite.

The final block of twenty lines defines the benchmark itself. The `DUMMY` option is explained by the documentation string specified by the `option` directive. A `DUMMY` option of this sort is provided for every benchmark in the suite. The style of the documentation text - indented eight spaces from the left and bordered on the left by a column of >'s and another column of spaces - is typical for the whole suite. Note that the structure of the 'dummy' call which is generated when `DUMMY` is set is identical to that of the 'dummy' call indicated for the bench by the `benchmark/4` clause in `set-up.nreverse`. This compatibility is also a characteristic of every benchmark in the suite.

Figures 3-5 (pp. 11-13) indicate the results of *MAKE*'ing the `nreverse` benchmark with various arguments and options. In figure 3,

```
MAKE -f nreverse.m -D -L
```

generates the documentation text shown (-D argument) and writes the "options list" file .nreverse.option for subsequent use by *MAKE* (-L argument). No Prolog output is generated. The first block of documentation text and the first six options listed in .nreverse.option are specified by the `option` directive near the beginning of the bench driver (see Appendix F). (Note that if `nreverse.m` is the only file in the directory named with a `.m` extension, as is in fact the case in the current on-disk configuration of the suite, then the `-f nreverse.m` argument is not strictly necessary - *MAKE* will assume the first and only "`.m`" file in the directory to be the input. Note also that *MAKE* is not sensitive to the order in which arguments are given to it.)

In figure 4,

```
MAKE -f nreverse.m -o nreverse.pl
```

generates Prolog output in `nreverse.pl`. This is the plain, "no-frills" version of the `nreverse` benchmark. For most of the benchmarks in the suite, *MAKE* with no options produces such a version.

In figure 5,

```
MAKE -f nreverse.m B C
```

```
# /*
  set-up.nreverse: bench set-up for nreverse
  */
nreverse :- driver(nreverse).

benchmark(nreverse,
          nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                    13,14,15,16,17,18,19,20,21,
                    22,23,24,25,26,27,28,29,30],_),
          dummy([1,2,3,4,5,6,7,8,9,10,11,12,
                 13,14,15,16,17,18,19,20,21,
                 22,23,24,25,26,27,28,29,30],_),
          1000).

show(nreverse) :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                            13,14,15,16,17,18,19,20,21,
                            22,23,24,25,26,27,28,29,30],R),
                  write('reverse of'), nl,
                  write([1,2,3,4,5,6,7,8,9,10,11,12,
                         13,14,15,16,17,18,19,20,21,
                         22,23,24,25,26,27,28,29,30]), nl,
                  write(is), nl,
                  write(R), nl.


#include "driver"
```

Figure 2

generates Prolog output which includes the bench with an appropriate definition for `get_cpu_time/2`. This time the output is in `out.pl`; *MAKE* writes to this file by default when no file is specified with the -o argument. Note the ''options stamp'' at the top. Note also how *MAKE* has resolved the ambiguous prefix `B` to `BENCH`, the first match in .nreverse.option.

Suppose we want Prolog output for `nreverse` incorporating the functionality of the bench predicate `show/1` but without including the bench. This sort of modification is fast and easy with *pre*. Figure 6 (p. 14) is a listing of a revised `nreverse.m` which provides what we want through a new option, `SHOW`. Figure 7 (p. 15) indicates how to use the new option. First,

```
MAKE -f nreverse.m -L
```

''installs'' it in .nreverse.option. Then,

```
MAKE -f nreverse.m SH
```

generates a new `out.pl` with the new `nreverse/0`. A `SHOW` option of this sort has in fact been added for every benchmark in the suite for which the bench predicate `show/1` is available.

```
haygood@vega> MAKE -f nreverse.m -D -L

        > Option BENCH includes the 'bench' for execution
        > time measurement.
        >
        > The 'bench' uses the system-dependent predicate
        > get_cpu_time/2.  If one of
        >
        > BIM_PL C_PL QUINTUS_PL SB_PL SICSTUS_PL
        >
        > is selected, then an appropriate definition for
        > get_cpu_time/2 is generated automatically.

        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (nreverse/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected.

haygood@vega> cat .nreverse.option
BENCH
BIM_PL
C_PL
QUINTUS_PL
SB_PL
SICSTUS_PL
DUMMY
```

Figure 3

```
haygood@vega> MAKE -f nreverse.m -o nreverse.pl
haygood@vega> cat nreverse.pl
% generated: 30 April 1989
% option(s):
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],_).

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

Figure 4

```
haygood@vega> MAKE -f nreverse.m B C
haygood@vega> cat out.pl
% generated: 30 April 1989
% option(s): BENCH C_PL
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

nreverse :- driver(nreverse).

benchmark(nreverse,
          nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                    13,14,15,16,17,18,19,20,21,
                    22,23,24,25,26,27,28,29,30],_),
          dummy([1,2,3,4,5,6,7,8,9,10,11,12,
                 13,14,15,16,17,18,19,20,21,
                 22,23,24,25,26,27,28,29,30],_),
          1000).

show(nreverse) :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                           13,14,15,16,17,18,19,20,21,
                           22,23,24,25,26,27,28,29,30],R),
                  write('reverse of'), nl,
                  write([1,2,3,4,5,6,7,8,9,10,11,12,
                         13,14,15,16,17,18,19,20,21,
                         22,23,24,25,26,27,28,29,30]), nl,
                  write(is), nl,
                  write(R), nl.


% driver(Name*Iterations) :-
%   Call benchmark/4 to find out the Action and its Control, perform
%   the specified number of Iterations of them, and report the times.

driver(Name*Iterations) :-
        integer(Iterations),
        Iterations >= 1,
        !,
        benchmark(Name, Action, Control, _),
        get_cpu_time(T0, Unit),
        (   repeat(Iterations), call(Action), fail
        ;   get_cpu_time(T1, Unit)
        ),
        (   repeat(Iterations), call(Control), fail
        ;   get_cpu_time(T2, Unit)
        ),
        report(Name, Iterations, T0, T1, T2, Unit).

% driver(Name) :-
%   Call benchmark/4 to find out how many Iterations of the Action
%   and its Control to perform, perform them, and report the times.

driver(Name) :-
        benchmark(Name, Action, Control, Iterations),
        get_cpu_time(T0, Unit),
        (   repeat(Iterations), call(Action), fail
        ;   get_cpu_time(T1, Unit)
        ),
        (   repeat(Iterations), call(Control), fail
        ;   get_cpu_time(T2, Unit)
        ),
        report(Name, Iterations, T0, T1, T2, Unit).
```

```
% get_cpu_time(T, seconds) :- T is the current cpu time
%                              (in seconds for C Prolog).

get_cpu_time(T, seconds) :- T is cputime.


% report(Name, N, T0, T1, T2, Unit) :-
%   Take the number of iterations and the three times yielded by
%   get_cpu_time/2 and write the total, overhead, and average.

report(Name, N, T0, T1, T2, Unit) :-
        TestTime is T1-T0,
        =(TestTime, Unit, TestTime_out, Unit_out),
        Overhead is T2-T1,
        =(Overhead, Unit, Overhead_out, Unit_out),
        Average_out is (TestTime_out-Overhead_out)/N,
        write(Name), write(' took '),
        write((TestTime_out-Overhead_out)/N=Average_out),
        write(' '), write(Unit_out), write('/iteration'), nl.


% repeat(N) :- succeed precisely N times.
% This is designed solely for use in this application; for a general
% way of doing this use the standard library predicate between/3, or
% perhaps repeat/0.

repeat(N) :- N > 0, from(1, N).

from(I, I) :- !.
from(L, U) :- M is (L+U) >> 1,     from(L, M).
from(L, U) :- M is (L+U) >> 1 + 1, from(M, U).


% =(T1, Unit1, T2, Unit2) :- T1 Unit1 = T2 Unit2.
%   The purpose of =/4 is unit conversion - the intended usage is
%   from T1 Unit1 to T2 Unit2.  In particular, the purpose is time
%   unit conversion for report/6.  Preferentially, times convert to
%   milli-seconds.  However, clauses may be added to convert to any
%   unit desired.

=(T1, seconds, T2, 'milli-seconds') :- !, T2 is T1*1000.
=(T, Unit, T, Unit).    % "catch-all" identity


% Trivial predicates for use as controls.

dummy.

dummy(_).

dummy(_, _).

dummy(_, _, _).

dummy(_, _, _, _).

dummy(_, _, _, _, _).


nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

Figure 5

```
# /*
  nreverse.m: Warren benchmark nreverse master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

#if BENCH
#  include ".nreverse.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],R),
            write('reverse of'), nl,
            write([1,2,3,4,5,6,7,8,9,10,11,12,
                   13,14,15,16,17,18,19,20,21,
                   22,23,24,25,26,27,28,29,30]), nl,
            write(is), nl,
            write(R), nl.
#  else
nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],_).
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (nreverse/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
nreverse(_,_).
#else
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
#endif
```

Figure 6

```
haygood@vega> MAKE -f nreverse.m -L
haygood@vega> MAKE -f nreverse.m SH
haygood@vega> cat out.pl
% generated: 30 April 1989
% option(s): SHOW
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],R),
            write('reverse of'), nl,
            write([1,2,3,4,5,6,7,8,9,10,11,12,
                   13,14,15,16,17,18,19,20,21,
                   22,23,24,25,26,27,28,29,30]), nl,
            write(is), nl,
            write(R), nl.

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

Figure 7

4 Future Work

There are many potential improvements to the suite. These include:

- More ''macroscopic'' benchmarks - for example, a serious expert system.
- Benchmarks written with parallel execution in mind. There are few of these yet. This is surely an important future direction.
- More apparatus for evaluating benchmark performance. The present suite incorporates a simple framework for execution time measurement. Some external performance analysis techniques are also available. (For example, one can compile C Prolog with the -pg option, run a Prolog benchmark under C Prolog, and obtain an execution profile with gprof.) In the future an integrated analysis ''workbench'' such as Gauge [GK88] may be incorporated into the suite.
- More analysis of what results for these benchmarks mean for Prolog implementations. Benchmarking is as much art as science. Figuring out what statistics from a set of benchmarks imply about the multitude of decisions embodied in an implementation is a formidable task. Important work in this direction has been done over the last few years, but more is needed. There is not yet any work in Prolog benchmarking fully comparable to, say, R. P. Gabriel's work in Lisp benchmarking. [Gab85]

Acknowledgements

Many people have contributed indispensably to this work. To begin with, I wish to thank the authors of the benchmarks themselves, including Alvin Despain, Tep Dobry, R. P. Gabriel, Seif Haridi, Rick McGeer, Richard A. O'Keefe, Ross Overbeek, Fernando C. N. Pereira, Evan Tick, Hervé Touati, Peter Van Roy, David H. D. Warren, and the ASP group. I also wish to thank my associates at Berkeley, particularly Bill Bush, Bruce Holmer, Tam Nguyen, Carl Ponder, Wesley Reed, Vason Srini, Hervé Touati, and Peter Van Roy for stimulating discussions and helpful suggestions along the way. I am particularly grateful to Tam Nguyen, who first suggested to me the advantages of using a preprocessor and later suggested to me what became the `option` directive. Finally, I wish to thank all who contributed to the workshop on Prolog benchmarking held at The Aerospace Corporation in Los Angeles during February 1988.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD) and monitored by the Office of Naval Research under Contract No. N00014-88-K-0579 through the Electronics Research Laboratory, University of California, Berkeley

References

[Bur87] L. Burkholder et al. "Fourteen Product Wrap-up: Prolog for the People." *AI Expert*, 2(6) (June 1987), 63-84.

[Bus88] W. Bush et al. *A Prototype Silicon Compiler in Prolog*. University of California (Technical Report UCB/CSD 88/476), Berkeley, California, 1988.

[DDP85] T. P. Dobry, A. M. Despain, Y. N. Patt. "Performance Studies of a Prolog Machine Architecture." In *Conference Proceedings of the 12th Annual International Symposium on Computer Architecture* (June 1985). 180-190.

[Des87] A. M. Despain et al. "Aquarius." *Computer Architecture News*, 15(1) (March 1987), 22-34.

[Dob87] T. P. Dobry. *A High Performance Architecture for Prolog*. University of California (Technical Report UCB/CSD 87/352), Berkeley, California, 1987.

[DSP85] A. M. Despain, V. P. Srini, Y. N. Patt. "Comparative Performance Evaluation of the PLM and NCR 9300 System." In *Proceedings of the 18th Annual Workshop on Microprogramming and Microarchitecture (Micro 18)* (1985).

[Gab85] R. P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, Massachusetts, 1985.

[GK88] M. M. Gorlick, C. F. Kesselman. "Gauge: A Workbench for the Performance Analysis of Logic Programs." In *Proceedings of the Fifth International Conference and Symposium on Logic Programming* (August 1988). MIT Press, Cambridge, Massachusetts, 1988.

[NSD88] T. M. Nguyen, V. P. Srini, A. M. Despain. "A Two-Tier Memory Architecture for High-Performance Multiprocessor Systems." In *Proceedings of the 1988 ACM International Conference on Supercomputing* (July 1988).

[Pon89] C. Ponder. *Prolog vs. Lisp*. University of California (Technical Report to be announced), Berkeley, California, 1989.

[Qui88] Quintus Computer Systems, Inc. *Release Notes for Quintus Prolog 2.0*. Quintus Computer Systems, Inc., Mountain View, California, 1988, 8-10.

[Tic86] E. Tick. "Memory Performance of Lisp and Prolog Programs." In *Proceedings of the Third International Conference on Logic Programming* (July 1986). Springer-Verlag (Lecture Notes in Computer Science 225), Berlin, 1986, 642-649.

[Tic87] E. Tick. *Studies In Prolog Architectures*. Stanford University (Technical Report No. CSL-TR-87-329), Stanford, California, 1987.

[Van84] P. Van Roy. *A Prolog Compiler for the PLM*. University of California (Technical Report UCB/CSD 84/203), Berkeley, California, 1984.

[War83] D. H. D. Warren. *Applied Logic - its Use and Implementation as a Programming Tool*. Artificial Intelligence Center (Technical Note 290), SRI International, 1983.

[WP82] D. H. D. Warren, F. C. N. Pereira. "An Efficient Easily Adaptable System for Interpreting Natural Language Queries." *American Journal of Computational Linguistics*, 8(3-4) (July- December 1982), 110-122.

Appendix A • Benchmark Suite Catalog

asp†		
name	*task*	*references*‡
inverter	compact VLSI inverter cell	[Bus88]
random_logic	compact VLSI random logic cell	[Bus88]
sml	synthesize structural description for simple microprocessor	[Bus88]

berkeley		
name	*task*	*references*
adder	design adder (with NAND's)	
mux[1]	design 2-1 MUX (with NAND's)	[Dob87] [DSP85]
concat_1[2]	concat [a,b,c] to [d,e]	[Dob87] [DSP85]
concat_6[3]	nondeterminate list concatenation	[DSP85]
hanoi_8[4]	8-disk tower of hanoi	[DSP85]
hanoi_16	16-disk tower of hanoi	
mu[5]	prove μ-math theorem	[Dob87] [DSP85] [NSD88]
prime_100[6]	find every prime < 100	[DSP85]
prime_1000	find every prime < 1000	
queens_4	4-queens problem	
queens_8	8-queens problem	

chat_parser		
name	*task*	*references*
chat_parser	parse natural language	[Tic87] [WP82]

† asp, berkeley, chat_parser, etc. name families of related benchmarks.
‡ These point to the References section at the end of the main text.
1 mux has also been known as ckt2.
2 concat_1 has also been known as con1.
3 concat_6 has also been known as con6.
4 hanoi_8 has also been known as hanoi.
5 mu has also been known as mumath and mutest.
6 prime_100 has also been known as pri2.

fft		
name	*task*	*references*
fft_4	fast fourier transform f(x) = x on 16 (= 2^4) points	
fft_8	fast fourier transform f(x) = x on 256 (= 2^16) points	

gabriel		
name	*task*	*references*
boyer	prove arithmetic theorem	[Gab85] [Pon89] [Tic86]
browse	build and query database	[Dob87] [Gab85] [Pon89]
poly_5	raise 1+x+y+z to 5th power	[Gab85] [Pon89]
poly_10	raise 1+x+y+z to 10th power	[Gab85] [Pon89]
poly_15	raise 1+x+y+z to 15th power	[Gab85] [Pon89]
puzzle	solve geometric puzzle	[Gab85] [Pon89] [Tic86]
tak	recursive arithmetic	[Gab85] [Pon89] [Tic86]

ili		
name	*task*	*references*
ili	natural deduction theorem proving	[Tic87]

	pereira	
name	*task*	*references*
`floating_add`	100 floating point additions	[Bur87] [Qui88]
`integer_add`	100 integer additions	[Bur87] [Qui88]
`arg_1`	100 calls to argument at position 1	[Bur87] [Qui88]
`arg_2`	100 calls to argument at position 2	[Bur87] [Qui88]
`arg_4`	100 calls to argument at position 4	[Bur87] [Qui88]
`arg_8`	100 calls to argument at position 8	[Bur87] [Qui88]
`arg_16`	100 calls to argument at position 16	[Bur87] [Qui88]
`assert_unit`	assert 1000 clauses	[Bur87] [Qui88]
`access_unit`	access 100 (dynamic) clauses with 1st argument instantiated	[Bur87] [Qui88]
`slow_access_unit`	access 100 (dynamic) clauses with 2nd argument instantiated	[Bur87] [Qui88]
`shallow_backtracking`	99 shallow failures	[Bur87] [Qui88]
`deep_backtracking`	99 deep failures	[Bur87] [Qui88]
`tail_call_atom_atom`	100 determinate tail calls	[Bur87] [Qui88]
`binary_call_atom_atom`	63 determinate nontail calls, 64 determinate tail calls	[Bur87] [Qui88]
`choice_point`	push 100 choice points	[Bur87] [Qui88]
`trail_variables`	push 100 choice points, trail 100 variables	[Bur87] [Qui88]
`index`	100 first-argument-determinate calls	[Bur87] [Qui88]
`cons_list`	construct 100-element list, nonrecursively	[Bur87] [Qui88]
`walk_list`	walk down 100-element list, nonrecursively	[Bur87] [Qui88]
`walk_list_rec`	walk down 100-element list, recursively	[Bur87] [Qui88]
`args_2`	walk down 2 copies of a 100-element list, recursively	[Bur87] [Qui88]
`args_4`	walk down 4 copies of a 100-element list, recursively	[Bur87] [Qui88]
`args_8`	walk down 8 copies of a 100-element list, recursively	[Bur87] [Qui88]
`args_16`	walk down 16 copies of a 100-element list, recursively	[Bur87] [Qui88]
`setof`	`setof(X, Y^pr(X, Y), _)`	[Bur87] [Qui88]
`pair_setof`	`setof((X, Y), pr(X, Y), _)`	[Bur87] [Qui88]
`double_setof`	`setof((X, S), setof(Y, pr(X, Y), S), _)`	[Bur87] [Qui88]
`bagof`	`bagof(X, Y^pr(X, Y), _)`	[Bur87] [Qui88]
`cons_term`	construct 100-node term, nonrecursively	[Bur87] [Qui88]
`walk_term`	walk down 100-node term, nonrecursively	[Bur87] [Qui88]
`walk_term_rec`	walk down 100-node term, recursively	[Bur87] [Qui88]
`medium_unify`	unify structures 5 deep	[Bur87] [Qui88]
`deep_unify`	unify structures 11 deep	[Bur87] [Qui88]

plm compiler		
name	*task*	*references*
plm_compiler	compile small Prolog file to PLM code	[Tic87] [Van84]

tp		
name	*task*	*references*
boys	prove propositional theorem	
ct_2	prove propositional theorem	
ct_3	prove propositional theorem	
ct_4	prove propositional theorem	
ct_5	prove propositional theorem	
ct_6	prove propositional theorem	

warren		
name	*task*	*references*
divide10	symbolic differentiation	[Dob87] [DSP85] [NSD88] [War83]
log10	symbolic differentiation	[Dob87] [DSP85] [War83]
ops8	symbolic differentiation	[Dob87] [DSP85] [War83]
times10	symbolic differentiation	[Dob87] [DSP85] [NSD88] [War83]
nreverse[7]	reverse 30-element list	[Dob87] [DSP85] [War83]
qsort[8]	quicksort 50-element list	[Dob87] [DSP85] [War83]
query	query small database	[Dob87] [DSP85] [NSD88] [War83]
serialise[9]	itemize 25-element list	[Dob87] [DSP85] [War83]

7 nreverse has also been known as nrev.

8 qsort has also been known as qs4.

9 serialise has also been known as palin25.

Appendix B • Execution Times

asp		
name	*Quintus Prolog*[*]	*C Prolog*[†]
inverter	2320	4220
random_logic	26700	92200
sml	3960	3320

berkeley		
name	*Quintus Prolog*	*C Prolog*
adder	2150	13700
mux	31.3	225
concat_1	0.017	1.03
concat_6	0.372	3.20
hanoi_8	19.8	282
hanoi_16	5060	72400[‡]
mu	64.9	615
prime_100	50.0	723
prime_1000	1680	22800[‡]
queens_4	3.70	63.8
queens_8	127	2900

chat_parser		
name	*Quintus Prolog*	*C Prolog*
chat_parser	3590	33600

fft		
name	*Quintus Prolog*	*C Prolog*
fft_4	55.8	484
fft_8	1790	14300[‡]

[*] Execution time in milli-seconds for Quintus Prolog 2.0 (compiled) on a Sun 3/60, to three significant figures (in most cases), averaged over many iterations. A '?' indicates the benchmark did not terminate within several hours.

[†] Execution time in milli-seconds for C Prolog 1.5 on a Sun 3/60, to three significant figures (in most cases), averaged over many iterations. '*memory*' indicates failure due to one memory-related problem or another.

[‡] These consume more memory than C Prolog allocates by default, but they will run if global stack size and/or local stack size is manually enlarged. Three Mbytes each is adequate.

gabriel		
name	*Quintus Prolog*	*C Prolog*
boyer	18200	*memory*
browse	23300	347000‡
poly_5	105	1050
poly_10	1420	14600‡
poly_15	7200	73300‡
puzzle	10500	27600
tak	3300	73000‡

ili		
name	*Quintus Prolog*	*C Prolog*
ili	1310	13000

pereira		
name	*Quintus Prolog*	*C Prolog*
floating_add	5.43	75.7
integer_add	1.32	76.3
arg_1	3.04	57.0
arg_2	3.06	57.0
arg_4	3.05	57.0
arg_8	3.06	57.0
arg_16	3.08	57.0
assert_unit	2180	1040
access_unit	24.7	623
slow_access_unit	798	873
shallow_backtracking	0.917	7.97
deep_backtracking	1.83	32.7
tail_call_atom_atom	0.850	13.7
binary_call_atom_atom	1.34	17.6
choice_point	1.73	13.8
trail_variables	2.45	16.3
index	1.39	458
cons_list	1.35	15.5
walk_list	0.708	19.2
walk_list_rec	0.550	25.9
args_2	1.03	34.7
args_4	1.77	52.1
args_8	3.89	86.6
args_16	7.72	156
setof	162	1700
pair_setof	163	1740
double_setof	522	3690
bagof	96.7	516
cons_term	1.51	15.4
walk_term	1.03	19.3
walk_term_rec	0.875	25.9
medium_unify	1.13	3.93
deep_unify	75.3	256

plm compiler		
name	*Quintus Prolog*	*C Prolog*
plm_compiler	1980	13200

tp		
name	*Quintus Prolog*	*C Prolog*
boys	14530	*memory*
ct_2	137	3790
ct_3	1150	36400‡
ct_4	39200	*memory*
ct_5	?	*memory*
ct_6	?	*memory*

warren		
name	*Quintus Prolog*	*C Prolog*
divide10	1.27	11.9
log10	0.468	8.50
ops8	0.767	8.83
times10	1.05	11.2
nreverse	4.87	183
qsort	16.9	230
query	185	2290
serialise	10.8	125

Appendix C • Prolog Features

Prolog Feature Classes*

class	*features*
1	cut [!]
2	disjunction [P ; Q]
3	if-then [P -> Q]
4	simple integer arithmetic [X is Y+1, X is Y−1, X =:= Y, X < Y, etc.]†
5	less simple integer arithmetic [X is Y+Z, Y*Z, Y<<Z, Y/\Z, etc.]†
6	floating point arithmetic
7	structure manipulation [functor/3, arg/3, =../2]
8	constant-text conversion [name/2]
9	database editing [assert/1, retract/1, abolish/2]
10	term comparison [T1 == T2, T1 @< T2, etc.]
11	negation-by-failure [\+ P]
12	call/1
13	reading [get/1, read/1, etc.]
14	writing [put/1, write/1, etc.]‡

* Note also:
(1) The asp benchmarks use consult/1; if this is inconvenient, the files they consult can be #include'd instead.
(2) setof/3 and bagof/3 are used only by the pereira benchmarks setof, pair_setof, double_setof, and bagof which focus on them.
(3) No benchmark in the suite uses grammar rules.

† Precisely, "simple integer arithmetic" is arithmetic comparison (X =:= Y, X =\= Y, X < Y, X > Y, X =< Y, X >= Y) with integer operands and increment or decrement by 1 (X is Y+1, X is Y−1) with an integer operand. Many benchmarks need only these arithmetic features. "Less simple integer arithmetic" is any other arithmetic with integer operands except division (X is Y/Z), whose result is always floating point.

‡ Benchmarks which offer the SHOW option are noted as requiring these features only if they use put/1, write/1, etc. even when SHOW is **not** selected.

asp														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
`inverter`	●	●	●	●	●	●	●	●	●					●
`random_logic`	●	●	●	●	●	●	●	●	●					●
`sml`	●	●[1]	●[1]	●			●	●	●		●			●

berkeley														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
`adder`	●			●										
`mux`	●			●										
`concat_1`														
`concat_6`														
`hanoi_8`	●			●										
`hanoi_16`	●			●										
`mu`				●										
`prime_100`	●			●	●						●			
`prime_1000`	●			●	●						●			
`queens_4`	●			●	●						●			
`queens_8`	●			●	●						●			

chat_parser														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
`chat_parser`	●[2]			●[2]										

fft														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
`fft_4`	●				●	●								
`fft_8`	●				●	●								

[1] These features (disjunction and if-then) are used only in connection with showing what `sml` does (with the `SHOW` option or with the bench predicate `show/1`). They can be removed easily.

[2] These features (cut and simple integer arithmetic) are each used exactly once in `chat_parser`. They can be removed easily.

gabriel														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
boyer	●	●	●	●			●							
browse	●	●	●	●	●		●							
poly_5	●			●	●					●				
poly_10	●			●	●					●				
poly_15	●			●	●					●				
puzzle	●	●[3]		●					●[3]					
tak	●			●										

ili														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
ili	●	●	●				●			●	●	●		

pereira														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
floating_add						●								
integer_add					●									
arg_1	●			●			●							
arg_2	●			●			●							
arg_4	●			●			●							
arg_8	●			●			●							
arg_16	●			●			●							
assert_unit	●			●	●				●					
access_unit	●	●		●	●				●					
slow_access_unit	●	●		●	●				●					
shallow_backtracking														
deep_backtracking														
tail_call_atom_atom														
binary_call_atom_atom														
choice_point	●													
trail_variables	●													
index														
cons_list														
walk_list														
walk_list_rec														
args_2														
args_4														
args_8														
args_16														
setof														
pair_setof														
double_setof														
bagof														
cons_term														
walk_term														
walk_term_rec														
medium_unify														
deep_unify														

plm compiler														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
plm_compiler	●	●	●	●	●		●	●	●	●	●	●	●	●

tp														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
boys	●	●	●	●	●		●			●	●		●	
ct_2	●	●	●	●	●		●			●	●		●	
ct_3	●	●	●	●	●		●			●	●		●	
ct_4	●	●	●	●	●		●			●	●		●	
ct_5	●	●	●	●	●		●			●	●		●	
ct_6	●	●	●	●	●		●			●	●		●	

warren														
name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
divide10	●													
log10	●													
ops8	●													
times10	●													
nreverse														
qsort	●			●										
query				●	●									
serialise	●			●										

[3] These features (disjunction and database editing) are necessary for `puzzle` only with Prolog systems which do not support `set/2` and `access/2` (see `puzzle.m`).

Appendix D • *pre*

Makefile

```
pre: pre.c y.tab.c lex.yy.c
        gcc -o pre pre.c -ll
        rm -f y.tab.c
        rm -f lex.yy.c

y.tab.c: pre.yacc
        yacc pre.yacc

lex.yy.c: pre.lex
        lex pre.lex
```

pre.c

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

/* for set_lex_start */
#define Normal  0
#define Mark    1
#define Assign  2

#define YES     1
#define NO      0

/* prototypes */
void set_lex_start(int start_condition);

/* variables */
FILE *infp = stdin, *outfp = stdout;
int normal_scan = YES, document_scan = NO, list_scan = NO;

main(int argc, char *argv[])
{
  /* prototypes */
  void process_args(int argc, char *argv[]);
  void assign_predefined(void);
  int yyparse(void);

  /* process command-line arguments */
  process_args(argc, argv);

  /* assign predefined macros */
  if (normal_scan) assign_predefined();

  /* initialize lexical analyzer */
  set_lex_start(Normal);

  /* process file */
  yyparse();
}

char *strdup(char *s)   /* duplicate string */
{
  char *p;

  p = (char *) malloc(strlen(s)+1);
  if (p != NULL)
    strcpy(p, s);
  return p;
}

#define HASHSIZE    107

static struct nlist *mactab[HASHSIZE];

unsigned hash(char *s)
{
    unsigned h = 0;

    while(*s) {
        if (isdigit(*s))
            h = 63*h + (*s-'0');
        else if (isupper(*s))
            h = 63*h + (*s-'A'+10);
        else if (islower(*s))
            h = 63*h + (*s-'a'+36);
        else /* *s == '_' */
            h = 63*(h+1);
        s++;
    }
    return h % HASHSIZE;
}
```

```
struct nlist {   /* macro table entry */
    struct nlist *next; /* next entry in bucket */
    char *name;         /* macro name */
    char *expansion;    /* macro expansion */
};

struct nlist *lookup(char *s)
{
    struct nlist *np = mactab[hash(s)];

    while(np /* != NULL */)
        if (strcmp(s, np->name) == 0)
            return np;
        else
            np = np->next;
    return NULL;
}

char *assign(char *name, char *expansion)
{
    struct nlist *np = lookup(name);
    unsigned h;

    if (np == NULL) {    /* NOT found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        h = hash(name);
        np->next = mactab[h];
        mactab[h] = np;
    } else      /* found */
        free(np->expansion);
    return np->expansion = strdup(expansion);
}

char *expand(char *name)
{
    struct nlist *np = lookup(name);

    return (np /* != NULL */) ? np->expansion : NULL;
}

#define isunder(c)   c == '_'

char *get_identifier(char *s)
{
  int i = 1;
  char *p;

  if (!(isalpha(*s) || isunder(*s)))
    return NULL;
  while (isalnum(*(s+i)) || isunder(*(s+i)))
    i++;
  p = (char *) malloc(i+1);
  if (p != NULL) {
    strncpy(p, s, i);
    *(p+i) = NULL;      /* this IS necessary */
  }
  return p;
}
```

```
void process_args(int argc, char *argv[])
{
  int i = 0;
  char c;
  char *name;
  char options[1024] = { '\0' };/* clumsy but adequate */

  while (--argc > 0 && *argv[++i] == '-')
    switch (c = *++argv[i]) {
      case 'a':
        name = get_identifier(++argv[i]);
        if (name == NULL || (c = *(argv[i] += strlen(name))) != '=') {
          fprintf(stderr, "Usage: pre -aname=expansion\n");
          exit(-1);
        }
        assign(name, ++argv[i]);
        strcat(options, " ");
        strcat(options, name);
        strcat(options, "=");
        strcat(options, argv[i]);
        free(name);
        break;
      case 's':
        name = get_identifier(++argv[i]);
        if (name == NULL) {
          fprintf(stderr, "Usage: pre -sname\n");
          exit(-1);
        }
        assign(name, "1");
        strcat(options, " ");
        strcat(options, name);
        free(name);
        break;
      case 'D':
        normal_scan = NO;
        document_scan = YES;
        break;
      case 'L':
        normal_scan = NO;
        list_scan = YES;
        break;
      default:
        fprintf(stderr, "Illegal option %c\n", c);
        exit(-1);
    }
  if (argc != 0) {
    fprintf(stderr, "Usage: pre [-D] [-L] [-aname=expansion] [-sname]\n");
    exit(-1);
  }
  if (normal_scan && !lookup("__OPTIONS__"))
    if (strlen(options) /* != 0 */)
      assign("__OPTIONS__", options+1);
    else
      assign("__OPTIONS__", "");
}
```

```
#include <sys/time.h>

void assign_predefined(void)
{
  char *mname[] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  struct timeval *tvp = (struct timeval *) malloc(sizeof(struct timeval));
  struct tm *tmp;
  char buf[5];

  /* TIME MACROS */

  gettimeofday(tvp, NULL);
  tmp = localtime(&tvp->tv_sec);

  /* month name abbreviation (three-letter) */
  if (!lookup("__MABB3__")) {
    buf[3] = '\0';
    assign("__MABB3__", strncpy(buf, mname[tmp->tm_mon], 3));
  }

  /* day of the month */
  if (!lookup("__MDAY__")) {
    sprintf(buf, "%u", tmp->tm_mday);
    assign("__MDAY__", buf);
  }

  /* month name (full) */
  if (!lookup("__MONTH__"))
    assign("__MONTH__", mname[tmp->tm_mon]);

  /* year including century */
  if (!lookup("__YEAR__")) {
    sprintf(buf, "%u", 1900+tmp->tm_year);
    assign("__YEAR__", buf);
  }

  /* year not including century */
  if (!lookup("__YABB2__")) {
    sprintf(buf, "%u", tmp->tm_year%100);
    assign("__YABB2__", buf);
  }
}


void yyerror(char *s)
{
  fprintf(stderr, "%s\n", s);
}


#include "y.tab.c"
```

pre.yacc

```
%{

/* prototypes */
void include(char *fname);
void setup_if(int condition);
void continue_if(int condition);
void wrapup_if(void);
char *dequote(char *s);

/* variables */
static int eliding = NO;
static int expanding = YES;
static char linbuf[2048];

%}

%start  lines

%union  { int i;
          char *s;
        }

%token <i> INTEGER
%token <s> GENERIC
%token <s> IDENTIFIER
%token <s> STRING
%token AND
%token ASSIGN
%token CLEAR
%token DEFINE
%token ELIF
%token ELSE
%token ELSEIF
%token ENDIF
%token EQ
%token ERROR
%token GE
%token GT
%token HALT
%token IF
%token IFDEF
%token IFNDEF
%token INCLUDE
%token LE
%token LT
%token MARK
%token MESSAGE
%token NE
%token NOT
%token OPTION
%token OR
%token SET

%type <i> conditional_expression
%type <i> equality_expression
%type <i> logical_AND_expression
%type <i> logical_NOT_expression
%type <i> logical_OR_expression
%type <i> primary_expression
%type <i> relational_expression
%type <s> generics
%type <s> non_control_line
```

pre.yacc

```
%%

lines   :       line
        |       lines line
        ;

line    :       non_control_line
                    { if (!eliding && normal_scan) fputs($1, outfp); }
        |       control_line
        ;

non_control_line    :   generics '\n'
                            { $$ = (!eliding && normal_scan) ?
                                strcat($1, "\n") : NULL; }
                    ;

generics     :  /* empty */
                    { $$ = (!eliding && normal_scan) ?
                        strcpy(linbuf, "") : NULL; }
             |  generics GENERIC
                    { $$ = (!eliding && normal_scan) ?
                        strcat($1, $2) : NULL; }
             ;

control_line      :   assignment_line
                  |   comment_line
                  |   error_line
                  |   halt_line
                  |   include_line
                  |   message_line
                  |   option_line
                  |   conditional
                  ;

assignment_line :   assign_line
                |   set_line
                |   clear_line
                ;

assign_line :   mark
                assign_token
                    { expanding = NO; }
                IDENTIFIER
                    { set_lex_start(Assign); }
                generics
                norm
                    { expanding = YES;
                      if (!eliding && normal_scan)
                        { assign($4, $6); free($4); } }
            ;

assign_token    :   ASSIGN
                |   DEFINE
                ;

set_line    :   mark
                SET
                    { expanding = NO; }
                IDENTIFIER
                norm
                    { expanding = YES;
                      if (!eliding && normal_scan)
                        { assign($4, "1"); free($4); } }
            ;
```

```
clear_line   :  mark
                CLEAR
                    { expanding = NO; }
                IDENTIFIER
                norm
                    { expanding = YES;
                      if (!eliding && normal_scan)
                        { assign($4, "0"); free($4); } }
             ;

comment_line    :   mark norm
                ;

error_line   :  mark
                ERROR
                    { set_lex_start(Normal); }
                generics
                '\n'
                    { if (!eliding && normal_scan)
                        { fprintf(stderr, "%s\n", $4+strspn($4, " \t"));
                          exit(0); } }
             ;

halt_line    :  mark HALT norm
                    { if (!eliding && normal_scan) exit(0); }
             ;

include_line    :   mark INCLUDE STRING norm
                        { if (!eliding) include(dequote($3)); free($3); }
                ;

message_line    :   mark MESSAGE STRING norm
                        { if (!eliding && normal_scan)
                            fprintf(stderr, "%s\n", dequote($3)); free($3); }
                ;

option_line     :   mark
                    OPTION
                        { expanding = NO; }
                    option_list
                        { expanding = YES; }
                    document_part
                    norm
                ;

option_list  :  /* empty */
             |  option_list IDENTIFIER
                    { if (list_scan) fprintf(outfp, "%s\n", $2); free($2); }

document_part   :   /* empty */
                |   STRING
                        { if (document_scan)
                            fprintf(stderr, "%s\n", dequote($1)); free($1); }
                ;


conditional  :  if_part lines endif_part
             |  if_part lines else_part lines endif_part
             |  if_part lines elseif_parts lines endif_part
             |  if_part lines elseif_parts lines else_part lines endif_part
             ;
```

```
if_part :     mark IF conditional_expression norm
                  { if (normal_scan) setup_if($3); }
        |     mark
              IFDEF
                  { expanding = NO; }
              IDENTIFIER
              norm
                  { expanding = YES;
                    if (normal_scan) setup_if(lookup($4) != NULL); free($4); }
        |     mark
              IFNDEF
                  { expanding = NO; }
              IDENTIFIER
              norm
                  { expanding = YES;
                    if (normal_scan) setup_if(lookup($4) == NULL); free($4); }
        ;

else_part    :    mark ELSE norm
                      { if (normal_scan) continue_if(1); }
             ;

elseif_parts     :    elseif_part
                 |    elseif_parts lines elseif_part
                 ;

elseif_part :    mark elseif_token conditional_expression norm
                     { if (normal_scan) continue_if($3); }
             ;

elseif_token     :    ELSEIF
                 |    ELIF
                 ;

endif_part   :    mark ENDIF norm
                      { if (normal_scan) wrapup_if(); }
             ;

conditional_expression   :    logical_OR_expression
                         ;

logical_OR_expression    :    logical_AND_expression
                         |    logical_OR_expression OR logical_AND_expression
                                  { $$ = $1 || $3; }
                         ;

logical_AND_expression   :    logical_NOT_expression
                         |    logical_AND_expression AND logical_NOT_expression
                                  { $$ = $1 && $3; }
                         ;

logical_NOT_expression   :    equality_expression
                         |    NOT equality_expression
                                  { $$ = !$2; }
                         ;

equality_expression      :    relational_expression
                         |    equality_expression EQ relational_expression
                                  { $$ = $1 == $3; }
                         |    equality_expression NE relational_expression
                                  { $$ = $1 != $3; }
                         ;
```

```
relational_expression      :   primary_expression
                           |   relational_expression LT primary_expression
                                   { $$ = $1 < $3; }
                           |   relational_expression GT primary_expression
                                   { $$ = $1 > $3; }
                           |   relational_expression LE primary_expression
                                   { $$ = $1 <= $3; }
                           |   relational_expression GE primary_expression
                                   { $$ = $1 >= $3; }
                           ;

primary_expression         :   INTEGER
                           |   IDENTIFIER
                                   { free($1); $$ = 0; }
                           |   '(' conditional_expression ')'
                                   { $$ = $2; }


mark    :       MARK      { set_lex_start(Mark); } ;

norm    :       '0        { set_lex_start(Normal); } ;

%%

static char *inpath = "./";     /* (/-terminated) input path */

char *path(char *fspec) /* return (/-terminated) path */
{
    char *p = strdup(fspec);
    char *q = p + strlen(fspec);

    while (q != p && *(--q) != '/')
        *q = NULL;
    return p;
}

char *fspec(char *path, char *fname)
{
    char *q, *r;

    if (*fname == '/')  /* absolute file specification */
        q = strdup(fname);
    else {
        q = (r = (char *) malloc(strlen(path)+strlen(fname)+1));
        if (r /* != NULL */) {
            while (*r = *path++) r++;
            while (*r++ = *fname++) ;
        }
    }
    return q;
}

typedef struct f_frame {
    struct f_frame *next;
    FILE *fp;
    char *path;
};

static struct f_frame *f_stack = NULL;

void push_f(FILE *fp, char *path)
{
    struct f_frame *new = (struct f_frame *) malloc(sizeof(struct f_frame));

    new->next = f_stack;
    new->fp = fp;
    new->path = path;

    f_stack = new;
}
```

```
struct f_frame *pop_f(void)
{
    struct f_frame *top = f_stack;

    if (top /* != NULL */) {
        f_stack = top->next;
        return top;
    } else {
        fprintf(stderr, "FATAL: pop_f: stack underflow\n");
        exit(-1);
    }
}

#include <sys/types.h>
#include <sys/stat.h>

#define MAXLNKLEN   128

void include(char *name)
{
    char *spec = fspec(inpath, name);
    struct stat *sp = (struct stat *) malloc(sizeof(struct stat));
    char lnkbuf[MAXLNKLEN];
    int lnklen;
    struct f_frame *top;

    push_f(infp, inpath);
    inpath = path(spec);
    if (lstat(spec, sp) == -1) {
        fprintf(stderr, "FATAL: include_line: lstat error: %s\n", spec);
        exit(-1);
    }
    while ((sp->st_mode & S_IFMT) == S_IFLNK) {
        if ((lnklen = readlink(spec, lnkbuf, MAXLNKLEN)) == -1) {
            fprintf(stderr, "FATAL: include_line: readlink error: %s\n", spec);
            exit(-1);
        }
        lnkbuf[lnklen] = NULL;
        free(spec);
        spec = fspec(inpath, lnkbuf);
        free(inpath);
        inpath = path(spec);
        if (lstat(spec, sp) == -1) {
            fprintf(stderr, "FATAL: include_line: lstat error: %s\n", spec);
            exit(-1);
        }
    }
    free(sp);
    if ((infp = fopen(spec, "r")) == NULL) {
        fprintf(stderr, "FATAL: include_line: fopen error: %s\n", spec);
        exit(-1);
    }
    free(spec);
    if (yyparse() == 1) {
        fprintf(stderr, "FATAL: include_line: yyparse error\n");
        exit(-1);
    }
    fclose(infp);
    free(inpath);
    top = pop_f();
    infp = top->fp;
    inpath = top->path;
    free(top);
    yychar = -1;        /* force a call to yylex() */
}
```

pre.yacc

```
#define TURN_ON     1
#define LEAVE_ON    0
#define TURN_OFF   -1

typedef struct if_frame {
    struct if_frame *next;
    int upon_continue_if;        /* TURN_ON or LEAVE_ON or TURN_OFF eliding  */
    int upon_wrapup_if;          /* YES or NO - eliding turned on in this if */
};

static struct if_frame *if_stack = NULL;

void push_if(int for_continue, int for_wrapup)
{
    struct if_frame *new = (struct if_frame *) malloc(sizeof(struct if_frame));

    new->next = if_stack;
    new->upon_continue_if = for_continue;
    new->upon_wrapup_if = for_wrapup;

    if_stack = new;
}

struct if_frame *pop_if(void)
{
    struct if_frame *top = if_stack;

    if (top /* != NULL */) {
        if_stack = top->next;
        return top;
    } else {
        fprintf(stderr, "FATAL: pop_if: stack underflow\n");
        exit(-1);
    }
}

void setup_if(int condition)
{
    if (!eliding) {
        if (condition)
            push_if(TURN_ON, YES);
        else {
            eliding = YES;
            push_if(TURN_OFF, NO);
        }
    } else
        push_if(LEAVE_ON, NO);
}

void continue_if(int condition)
{
    struct if_frame *top = pop_if();

    switch (top->upon_continue_if) {
        case TURN_ON:
            eliding = YES;
            break;
        case TURN_OFF:
            eliding = NO;
            /* break; */
    }
    if (!eliding) {
        if (condition)
            push_if(TURN_ON, YES);
        else {
            eliding = YES;
            push_if(TURN_OFF, NO);
        }
    } else
        push_if(LEAVE_ON, top->upon_wrapup_if);
}
```

```
void wrapup_if(void)
{
    struct if_frame *top = pop_if();

    switch (top->upon_continue_if) {
        /* case TURN_ON:
            eliding = YES;
            break; */
        case TURN_OFF:
            eliding = NO;
            /* break; */
    }
    if (top->upon_wrapup_if)
        eliding = NO;
}


char *dequote(char *s)  /* remove " from each end of "'ed string */
{
    /* no error checking! */
    *strrchr(s, '"') = NULL;
    return ++s;
}


#include "lex.yy.c"
```

pre.lex

```
%{

/* redefine LEX buffer size */
#undef YYLMAX
#define YYLMAX  2048

/* cancel LEX defaults */
#undef input()
#undef unput(c)

/* prototypes */
void discard(int n);
void insert(char *s);

/* variables */
char *exp;

%}

BN      (\\n)+
ID      [_A-Za-z][_A-Za-z0-9]*
WH      [ \t]+

%S      N M A0 A1

%%

<M>assign       return ASSIGN;
<M>clear        return CLEAR;
<M>define       return DEFINE;
<M>elif         return ELIF;
<M>else         return ELSE;
<M>elseif       return ELSEIF;
<M>endif        return ENDIF;
<M>error        return ERROR;
<M>halt         return HALT;
<M>if           return IF;
<M>ifdef        return IFDEF;
<M>ifndef       return IFNDEF;
<M>include      return INCLUDE;
<M>message      return MESSAGE;
<M>option       return OPTION;
<M>set          return SET;
<M>\${ID}\$     { yytext[yyleng-1] = '\0';
                  if (!normal_scan || !expanding ||
                     (exp = expand(yytext+1)) == NULL) {
                      yytext[yyleng-1] = '$';
                      yylval.s = yytext;
                      return GENERIC;
                  } else {  /* single expansion */
                      yylval.s = exp;
                      return GENERIC;
                  }
                }
<M>{ID}         { if (!normal_scan || !expanding ||
                     (exp = expand(yytext)) == NULL) {
                      yylval.s = strdup(yytext);
                      return IDENTIFIER;
                  } else {  /* expansion */
                      discard(yyleng);          /* replace macro  */
                      insert(exp);              /* with expansion */
                      yymore();
                  }
                }
```

pre.lex

```
<M>\"[^"]*\"     { if (yytext[yyleng-2] != '\\') {    /* return string */
                        yylval.s = strdup(yytext);
                        return STRING;
                   } else {  /* "...\" */
                        yyless(yyleng-1);
                        yymore();
                   }
                 }
<M>\-[1-9][0-9]*       |
<M>[1-9][0-9]*         { sscanf(yytext, "%d", &yylval.i);
                         return INTEGER;
                       }

<M>\-0[0-7]*           |
<M>0[0-7]*             { sscanf(yytext, "%o", &yylval.i);
                         return INTEGER;
                       }
<M>\-0[Xx][0-9A-F]*    |
<M>0[Xx][0-9A-F]*      { sscanf(yytext, "%x", &yylval.i);
                         return INTEGER;
                       }
<M>"||"          return OR;
<M>&&            return AND;
<M>!             return NOT;
<M>==            return EQ;
<M>!=            return NE;
<M>"<"           return LT;
<M>">"           return GT;
<M>"<="          return LE;
<M>">="          return GE;
<M>"("           return '(';
<M>")"           return ')';

<M,A0>{WH}       { discard(yyleng);       /* skip (<A0> leading) white-space */
                   yymore();
                 }

<A0>.            { yyless(0);
                   yymore();
                   BEGIN A1;
                 }

<A1>{WH}\n       return '\n';    /* skip trailing white-space */
<A1>{WH}{BN}     { discard(2);   /* skip trailing backslash-newline */
                   yymore();
                 }

<M,A0,A1>{BN}    { discard(yyleng);       /* skip backslash-newlines */
                   yymore();
                 }
<M,A0,A1>"/*"[^/]*"/" { if (yytext[yyleng-2] == '*')  /* skip comments */
                             discard(yyleng);
                           else {  /* /*.../ */
                             discard(yyleng-2);
                             yyless(0);
                           }
                           yymore();
                         }

<N,A1>\${ID}\$    { yytext[yyleng-1] = '\0';
                    if (!normal_scan || !expanding ||
                        (exp = expand(yytext+1)) == NULL) {
                         yytext[yyleng-1] = '$';
                         yylval.s = yytext;
                         return GENERIC;
                    } else {  /* single expansion */
                         yylval.s = exp;
                         return GENERIC;
                    }
                  }
```

```
<N,A1>{ID}       { if (!normal_scan || !expanding ||
                      (exp = expand(yytext)) == NULL) {
                       yylval.s = yytext;
                       return GENERIC;
                   } else {  /* expansion */
                       discard(yyleng);            /* replace macro  */
                       insert(exp);                /* with expansion */
                       yymore();
                   }
                 }
<N,A1>\"[^"]*\"  { if (yytext[yyleng-2] != '\\') {    /* return string */
                       yylval.s = yytext;
                       return GENERIC;
                   } else {  /* "...\" */
                       yyless(yyleng-1);
                       yymore();
                   }
                 }

^#               return MARK;

\n               return '\n';

.                { yylval.s = yytext;
                   return GENERIC;
                 }

%%

void set_lex_start(int start_condition)
{
    /* Assign, Mark, and Normal are #define'd in pre.c */
    switch (start_condition) {
        case Assign:
            BEGIN A0;
            break;
        case Mark:
            BEGIN M;
            break;
        case Normal:
            BEGIN N;
    }
}

#define BUFSIZE 2048

static char buf[BUFSIZE];        /* managed as stack */
static int bufp = 0;

static int nlf = 0;      /* newline-flag */

char input(void)         /* get next character for LEX */
{
    char c;

    if ((c = (bufp > 0) ? buf[--bufp] : getc(infp)) == EOF)
        if (nlf)
            return NULL;
        else {
            buf[bufp++] = EOF;
            c = '\n';
        }
    nlf = (c == '\n');
    return c;
}
```

```
static int dis = 0;

void discard(int n)     /* discard last n characters of yytext */
{
    dis = n;
    yyless(yyleng-n);
}

void unput(char c)      /* unget character c for LEX */
{
    if (dis > 0)
        dis--;
    else {
        if (bufp >= BUFSIZE) {
            fprintf(stderr, "FATAL: unput: buffer overflow\n");
            exit(-1);
        } else if (c /* != NULL */)     /* never unput end-of-file */
            buf[bufp++] = c;
    }
}

void insert(char *s)    /* insert string in buf */
{
    char *p = s;

    while (*p)
        p++;
    if (bufp+(p-s) > BUFSIZE) {
        fprintf(stderr, "FATAL: insert: buffer overflow\n");
        exit(-1);
    } else
        while (p != s)
            buf[bufp++] = *--p;
}

int yywrap(void)        /* for LEX at EOF */
{
    return -1;
}
```

Appendix E • *MAKE*

Makefile

```
MAKE: MAKE.c
	gcc -o MAKE MAKE.c
```

MAKE.c

```
#include <stdio.h>
#include <string.h>

#define FALSE   0
#define TRUE    1


char *strdup(char *s)   /* duplicate string */
{
  char *p;

  p = (char *) malloc(strlen(s)+1);
  if (p != NULL)
    strcpy(p, s);
  return p;
}


int is_prefix(char *prefix, char *string)
{
  int i = strlen(prefix);

  if (i <= strlen(string) && strncmp(string, prefix, i) == 0)
    return 1;
  else
    return 0;
}

int is_suffix(char *suffix, char *string)
{
  int i = strlen(string)-strlen(suffix);

  if (i >= 0 && strcmp(string+i, suffix) == 0)
    return 1;
  else
    return 0;
}


#include <sys/types.h>
#include <sys/dir.h>

main(int argc, char *argv[])
{
  char *inf = NULL, *outf = NULL, *optf = NULL;
  /* input file name, output file name, option list file name */

  char *optv[128] = { ".pre" };

  int argi = 0, opti = 0, len;
  char c;

  int user_optf = FALSE; /* optf user-specified (via -l) */

  DIR *dirfp;
  struct direct *dirp;

  FILE *optfp;
  char opt[64]; /* space for one option - clumsy but effective */
  char *pre, *def;
```

```
/* 1st scan through arguments - process -f -o -l */
while (++argi < argc)
  if (*argv[argi] == '-')
    switch (c = *(argv[argi]+1)) {
    case 'f':
      inf = strdup(argv[++argi]);
      break;
    case 'o':
      outf = strdup(argv[++argi]);
      break;
    case 'l':
      optf = strdup(argv[++argi]);
      user_optf = TRUE;
    }

if (inf == NULL) /* get first ".m" file in current directory */ {
  if ((dirfp = opendir(".")) == NULL) {
    fprintf(stderr, "Fatal: opendir error: current directory .\n");
    exit(-1);
  }
  do
    if ((dirp = readdir(dirfp)) == NULL) {
      fprintf(stderr, "Fatal: no \".m\" file in current directory\n");
      exit(-1);
    } else if (is_suffix(".m", dirp->d_name)) {
      inf = strdup(dirp->d_name);
      break;
    }
  while (TRUE);
  closedir(dirfp);
}

if (optf == NULL) {
  len = strlen(inf);
  if (is_suffix(".m", inf)) {
    optf = (char *) malloc(1+(len-2)+7+1); /* for .<inf-".m">.option\0 */
    strcpy(optf, ".");
    strncat(optf, inf, len-2);
    strncat(optf, ".option", 7);
  } else {
    optf = (char *) malloc(1+len+7+1); /* for .<inf>.option\0 */
    strcpy(optf, ".");
    strncat(optf, inf, len);
    strncat(optf, ".option", 7);
  }
}

argi = 0; /* reset for 2nd scan */

/* 2nd scan through arguments - process rest */
while (++argi < argc)
  if (*argv[argi] == '-')
    switch (c = *++argv[argi]) {
    case 'f':
    case 'o':
    case 'l':
      ++argi; /* skip next argument (already processed) */
      break;
    case 'D':
      optv[++opti] = strdup("-D");
      if (outf == NULL)
        outf = strdup("/dev/null");
      break;
  case 'L':
    optv[++opti] = strdup("-L");
    if (outf == NULL || strcmp(outf, "/dev/null") == 0)
      if (user_optf) /* must figure out option list file name */ {
        len = strlen(inf);
        if (is_suffix(".m", inf)) {
          outf = (char *) malloc(1+(len-2)+7+1);
          /* for .<inf-".m">.option\0 */
          strcpy(outf, ".");
          strncat(outf, inf, len-2);
          strncat(outf, ".option", 7);
        } else {
          outf = (char *) malloc(1+len+7+1);
          /* for .<inf>.option\0 */
          strcpy(optf, ".");
          strncat(optf, inf, len);
          strncat(optf, ".option", 7);
        }
      } else /* option list file name already figured out - copy it */
        outf = strdup(optf);
  }
else /* get first option (in option list file)
        of which argument is a valid prefix */ {
  if ((optfp = fopen(optf, "r")) == NULL) {
    fprintf(stderr, "Fatal: fopen error: option list file %s\n", optf);
    exit(-1);
  }
  def = strchr(argv[argi], '=');
  if (def /* != NULL */) /* extract prefix */ {
    len = def-argv[argi];
    pre = (char *) malloc(len+1);
    *pre = '\0';
    strncat(pre, argv[argi], len);
  } else
    pre = argv[argi];
  do
    if (fscanf(optfp, "%s\n", opt) == EOF) {
      fprintf(stderr, "Fatal: unknown option: %s\n", pre);
      exit(-1);
    } else if (is_prefix(pre, opt)) {
      if (def /* != NULL */) {
        len = strlen(def);
        optv[++opti] = (char *) malloc(2+strlen(opt)+len+1);
        /* for -a<opt>=...\0 */
        strcpy(optv[opti], "-a");
        strcat(optv[opti], opt);
        strncat(optv[opti], def, len);
        free(pre);
      } else {
        len = strlen(opt);
        optv[++opti] = (char *) malloc(2+len+1);
        /* for -s<opt>\0 */
        strcpy(optv[opti], "-s");
        strncat(optv[opti], opt, len);
      }
      break;
    }
  while (TRUE);
  fclose(optfp);
}
```

```
  if (outf == NULL)
    outf = strdup("out.pl");

  /* redirect standard input */
  if (freopen(inf, "r", stdin) == NULL) {
    fprintf(stderr, "Fatal: fopen error: input file %s\n", inf);
    exit(-1);
  }

  /* redirect standard output */
  if (freopen(outf, "w", stdout) == NULL) {
    fprintf(stderr, "Fatal: fopen error: output file %s\n", outf);
    exit(-1);
  }

  optv[++opti] = NULL; /* NULL-terminate option vector */

  execv(".pre", optv);
}
```

Appendix F • bench "driver"

driver

```
# /*
  (.bench) driver

  Ralph Haygood based on code by Richard O'Keefe in turn based
  on code by Paul Wilk, Fernando Pereira, David Warren, et al.

  defines driver/1 for execution time measurement (via get_cpu_time/2)
  */
#option BENCH BIM_PL C_PL QUINTUS_PL SB_PL SICSTUS_PL "
        > Option BENCH includes the 'bench' for execution
        > time measurement.
        >
        > The 'bench' uses the system-dependent predicate
        > get_cpu_time/2.  If one of
        >
        > BIM_PL C_PL QUINTUS_PL SB_PL SICSTUS_PL
        >
        > is selected, then an appropriate definition for
        > get_cpu_time/2 is generated automatically."
% driver(Name*Iterations) :-
%   Call benchmark/4 to find out the Action and its Control, perform
%   the specified number of Iterations of them, and report the times.

driver(Name*Iterations) :-
        integer(Iterations),
        Iterations >= 1,
        !,
        benchmark(Name, Action, Control, _),
        get_cpu_time(T0, Unit),
        (   repeat(Iterations), call(Action), fail
        ;   get_cpu_time(T1, Unit)
        ),
        (   repeat(Iterations), call(Control), fail
        ;   get_cpu_time(T2, Unit)
        ),
        report(Name, Iterations, T0, T1, T2, Unit).

% driver(Name) :-
%   Call benchmark/4 to find out how many Iterations of the Action
%   and its Control to perform, perform them, and report the times.

driver(Name) :-
        benchmark(Name, Action, Control, Iterations),
        get_cpu_time(T0, Unit),
        (   repeat(Iterations), call(Action), fail
        ;   get_cpu_time(T1, Unit)
        ),
        (   repeat(Iterations), call(Control), fail
        ;   get_cpu_time(T2, Unit)
        ),
        report(Name, Iterations, T0, T1, T2, Unit).
```

```
#if BIM_PL
% get_cpu_time(T, seconds) :- T is the current cpu time
%                             (in seconds for BIM Prolog).

get_cpu_time(T, seconds) :- cputime(T).
#elseif C_PL
% get_cpu_time(T, seconds) :- T is the current cpu time
%                             (in seconds for C Prolog).

get_cpu_time(T, seconds) :- T is cputime.
#elseif QUINTUS_PL
% get_cpu_time(T, 'milli-seconds') :- T is the current cpu time
%                                     (in milli-seconds for Quintus Prolog).

get_cpu_time(T, 'milli-seconds') :- statistics(runtime,[T,_]).
                                    % We can't use the second element
                                    % of the list, as some tests will
                                    % call statistics/2 and reset it.
#elseif SB_PL
% get_cpu_time(T, 'milli-seconds') :- T is the current cpu time
%                                     (in milli-seconds for SB Prolog).

get_cpu_time(T, 'milli-seconds') :- cputime(T).
#elseif SICSTUS_PL
% get_cpu_time(T, 'milli-seconds') :- T is the current cpu time
%                                     (in milli-seconds for SICStus Prolog).

get_cpu_time(T, 'milli-seconds') :- statistics(runtime,[T,_]).
                                    % We can't use the second element
                                    % of the list, as some tests will
                                    % call statistics/2 and reset it.
#else
#  message "WARNING: get_cpu_time/2 must be defined"
#endif


% report(Name, N, T0, T1, T2, Unit) :-
%   Take the number of iterations and the three times yielded by
%   get_cpu_time/2 and write the total, overhead, and average.

report(Name, N, T0, T1, T2, Unit) :-
        TestTime is T1-T0,
        =(TestTime, Unit, TestTime_out, Unit_out),
        Overhead is T2-T1,
        =(Overhead, Unit, Overhead_out, Unit_out),
        Average_out is (TestTime_out-Overhead_out)/N,
        write(Name), write(' took '),
        write((TestTime_out-Overhead_out)/N=Average_out),
        write(' '), write(Unit_out), write('/iteration'), nl.


% repeat(N) :- succeed precisely N times.
% This is designed solely for use in this application; for a general
% way of doing this use the standard library predicate between/3, or
% perhaps repeat/0.

repeat(N) :- N > 0, from(1, N).

from(I, I) :- !.
from(L, U) :- M is (L+U) >> 1,     from(L, M).
from(L, U) :- M is (L+U) >> 1 + 1, from(M, U).
```

```
% =(T1, Unit1, T2, Unit2) :- T1 Unit1 = T2 Unit2.
%   The purpose of =/4 is unit conversion - the intended usage is
%   from T1 Unit1 to T2 Unit2.  In particular, the purpose is time
%   unit conversion for report/6.  Preferentially, times convert to
%   milli-seconds.  However, clauses may be added to convert to any
%   unit desired.

=(T1, seconds, T2, 'milli-seconds') :- !, T2 is T1*1000.
=(T, Unit, T, Unit).    % "catch-all" identity


% Trivial predicates for use as controls.

dummy.

dummy(_).

dummy(_, _).

dummy(_, _, _).

dummy(_, _, _, _).

dummy(_, _, _, _, _).
```

Appendix G • Benchmark Suite Listing

- This listing of the benchmark suite is broken down by family.
- Page numbering is independent from one family to another (each family starts with page number 1).
- A table of contents precedes the listing for each family.
- At bottom center of each page is the name of the appropriate family followed by the page number within the family.
- At top center of each page is the name of the file appearing on the page; each new file starts a new page.
- Files appear in the following general order: master files (`.m` extension); files shared by more than one master file via `include` directives (no extension); input data files (where required); bench interface files (`.bench.`*`name`*); files associated with code in a language other than Prolog (Lisp files for `gabriel` and C files for `tp`).

asp

inverter.m

```
# /*
  inverter.m: benchmark (compactor) inverter master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (compactor) inverter
%
%   The ASP Group
%
%   (contact: Bill Bush
%              Computer Science Division
%              University of California
%              Berkeley, CA 94720
%              bush@ophiuchus.Berkeley.EDU)
%
%   compact inverter cell

#if BENCH
#  include ".inverter.bench"
#else
inverter :- consult('examples/in/inverter.sip'),
            compact('examples/out/inverter').
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (compact/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
compact(_).
#else
#  include "compactor"  /* code for compactor */
#  message "NOTE: The compactor does not clean up the database when it is finished,"
#  message "      so this benchmark should not be run several times in succession -"
#  message "      the Prolog system should be stopped and restarted after each run."
#endif
```

random_logic.m

```
# /*
 random_logic.m: benchmark (compactor) random_logic master file
 */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (compactor) random_logic
%
%   The ASP Group
%
%   (contact: Bill Bush
%             Computer Science Division
%             University of California
%             Berkeley, CA 94720
%             bush@ophiuchus.Berkeley.EDU)
%
%   compact random logic cell (for a chess chip)

#if BENCH
#  include ".random_logic.bench"
#else
random_logic :- consult('examples/in/random_logic.sip'),
                compact('examples/out/random_logic').
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (compact/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
compact(_).
#else
#  include "compactor"  /* code for compactor */
#  message "NOTE: The compactor does not clean up the database when it is finished,"
#  message "      so this benchmark should not be run several times in succession -"
#  message "      the Prolog system should be stopped and restarted after each run."
#endif
```

compactor

```
# /*
  compactor: code for ASP compactor component
  */
%   (c) 1988 Regents of the University of California
%
%   This is a version of the cell compactor developed at the University
%   of California, Berkeley, as an element of the ASP (Advanced Silicon
%   compiler in Prolog) system. The compactor is a CAD tool for VLSI
%   design. It uses a four-pass deterministic algorithm to transform
%   an input cell into an output cell of near-minimum silicon area. The
%   input language is Sticks, a virtual device specification language,
%   and the main output language is CIF, a physical layout description
%   language. The compactor transforms a virtual device (grid) repre-
%   sentation into a physical layout representation using fabrication
%   design rules.
%
%   From a Sticks specification in file name (assumed already loaded),
%   the compactor generates four output files:
%
%       name.bbox
%       name.bdr
%       name.cif
%       name.space
%
%   When show/0 is provable, the compactor produces output (intended
%   for a terminal screen) indicating the number of the virtual row
%   or column it is currently processing (row on the first and second
%   passes and column on the third and fourth passes).
%
%   The compactor uses two predicates, floor/2 and sqrt/2, which are
%   evaluated differently under different Prolog systems. These must
%   be defined for any Prolog system under which the compactor is run.
%
%   The compactor does not clean up the database when it is finished,
%   so it should not be run several times in succession.  The Prolog
%   system should be stopped and restarted after each run.


#option "
        > For use with Quintus Prolog, compactor requires
        > some Quintus Prolog-specific directives.  These
        > are generated if option QUINTUS_PL is selected."
#if QUINTUS_PL
:- no_style_check(single_var).

:- unknown(_, fail).


#endif
compact(Name) :-
   brktrans,
   cmp,
   genbox,
   balance,
   expand(Name),
   writefile(Name), !.


brktrans :-
   trans(Type,pt(Sx,Sy),pt(Gx,Gy),pt(Dx,Dy),W,L,Sn,Gn,Dn),
   genconst1(Type, Sx, Sy, Dx, Dy, W, L, Sn, Gn, Dn),
   genconst(Type, Sx, Sy, Gx, Gy, Dx, Dy, W, L, Sn, Gn, Dn),
   fail.
brktrans.

genconst1(Type, Sx, Sy, Dx, Dy, W, L, Sn, Gn, Dn) :-
   Wovl is W/L,
   larger(1, Wovl, Dntcr, Wlrat),
   transtype(Type, Layer),
   assert(wire(Layer, pt(Sx, Sy), pt(Dx, Dy), Wlrat, tnods(Sn, Gn, Dn))), !.
```

```
transtype(pd, pdtrans).
transtype(nd, ndtrans).

genconst(Type, Sx, Sy, Gx, Gy, Dx, Dy, W, L, Sn, Gn, Dn) :-
   Sx = Gx,
   width(p, Pwid),
   Lw is L/W,
   larger(Lw, 1, S, Larg),
   Sspace is Larg*Pwid,
   assert(gethpic(Gy,Gx,Gn, Sspace)), !.
genconst(Type, Sx, Sy, Gx, Gy, Dx, Dy, W, L, Sn, Gn, Dn) :-
   Sy = Gy,
   width(p, Pwid),
   Lw is L/W,
   larger(Lw, 1, S, Larg),
   Sspace is Larg*Pwid,
   assert(getvpic(Gx,Gy,Gn, Sspace)), !.


cmp :-
   xcompact,
   ydtoxd,
   ycompact,
   trueydist(0, 0),
   truexdist(0, 0),
   ordconst,
   makediag,
   doubled, !.


% compaction for x dimension.
xcompact :-
   rmconst,
   initmap,
   maxrow(Row1),
   initconst(Row1),
   (show -> write('beginning pass 1...'), nl ; true),
   (show -> write('current row:'), nl ; true),
   xcoordex(0), !,
   rmmap(right),
   initmap,
   (show -> write('beginning pass 2...'), nl ; true),
   (show -> write('current row:'), nl ; true),
   xcoordexb(Row1),
   rmmap(left), !.

% remove any existing dist values.
rmconst :-
   retract(ydist(_,_,_)),
   fail.
rmconst :-
   retract(xdist(_,_,_)),
   fail.
rmconst.

% initialize elists.
initmap :-
   assert(elist(p, [])),
   assert(elist(dif, [])),
   assert(elist(m1, [])),
   assert(elist(m2, [])).

% set all ydist values to zero.
initconst(0).
initconst(Row1) :-
   Lastrow is Row1 - 1,
   assert(ydist(Lastrow, Row1, 0)),
   initconst(Lastrow), !.
```

```
rmmap(Side) :-
   retract(elist(Layer, Elements)),
   assert(border(Side, Layer, Elements)),
   fail.
rmmap(_).

% compact in the x dimension, 0->maxrow.
xcoordex(Row) :-
   maxrow(Row1),
   Row1 < Row, !.
xcoordex(Row) :-
   getrow(Row),
   (show -> write(Row), nl ; true),
   Nextrow is Row + 1,
   xcoordex(Nextrow), !.

% compact in the x dimension, maxrow->0.
xcoordexb(Row) :-
   Row < 0, !.
xcoordexb(Row) :-
   getrow(Row),
   (show -> write(Row), nl ; true),
   Nextrow is Row - 1, !,
   xcoordexb(Nextrow), !.

% get all contacts in a row.
getrow(Row) :-
   getcel(Row, Type, Y, Node),
   contlayer(Type, Layer1, Layer2),
   appendel(Row, Layer1, Y, Y, 1, Node, Type),
   appendel(Row, Layer2, Y, Y, 1, Node, Type),
   fail, !.
% get all wires in a row.
getrow(Row) :-
   getwel(Row, Layer, Y1, Y2, Wid, Node),
   larger(Y1, Y2, S, L),
   appendel(Row, Layer, S, L, Wid, Node, Layer),
   fail, !.
% get all transistor pickets in a row.
getrow(Row) :-
   getvpic(Row, Y, Node, Width),
   lretract(pd, Elist),
   addeltomap(Row, tspot, Y, Y, Width, Node, Elist, Elistout),
   lassert(pd, Elistout),
   fail, !.
getrow(Row).

% add poly or diffusion to elist.
% they must check more than 1 elist....
appendel(Row, Layer, S, L, Wid, Node, Type) :-
   test(Layer),
   lretract(Layer, Elist),
   addeltomap(Row, Type, S, L, Wid, Node, Elist, Elistout), !,
   lassert(Layer, Elistout),
   dual(Layer, Duolayer),
   lbind(Duolayer, Duoelist),
   addeltomap(Row, Type, S, L, Wid, Node, Duoelist, Dontcare), !.
% add m2 or m1 to elist.
appendel(Row, Layer, S, L, Wid, Node, Type) :-
   lretract(Layer, Elist),
   addeltomap(Row, Type, S, L, Wid, Node, Elist, Elistout),
   lassert(Layer, Elistout), !.

addeltomap(Row, Layer, S, L, Wid, Node, Elist, Elistout) :-
   searchlist(Row, Layer, S, L, Wid, Node, Elist, Glist, Llist, Withinlist, x),
   diagadd(Row, Layer, S, L, Wid, Node, Glist, Llist,Withinlist, Elistout), !.
```

```
% add diagonal constraints if necessary.
diagadd(Row, Layer, S, L, Wid, Node, Glist, Llist, Withinlist, Elistout) :-
   rev(Llist, Newllist),
   adducons(Row, Layer, S, L, Wid, Node, Glist),
   addlcons(Row, Layer, S, L, Wid, Node, Newllist),
   listio(Layer, S, L, Wid, Node, Row, Elementpac),
   append([Elementpac], Withinlist, NewWithin),
   append(NewWithin, Glist, Newglist),
   append(Llist, Newglist, Elistout), !.

% add upper constraint.
adducons(Row, Layer, S, L, Wid, Node, []).
adducons(Row, Layer, S, L, Wid, Node, List) :-
   Layer = tspot, !.
adducons(Row, Layer, S, L, Wid, Node, [Element|Glist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Row = Rowf, !.
adducons(Row, Layer, S, L, Wid, Node, [Element|Glist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Nodef = Node,
   Nodef =\= -1, !.
adducons(Row, Layer, S, L, Wid, Node, [Element|Glist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Layerf = tspot,
   adducons(Row, Layer, S, L, Wid, Node, Glist), !.
adducons(Row, Layer, S, L, Wid, Node, [Element|Glist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   findcontact(Row, Y1,  Widf, Layerf, Cwid, Clayer),
   addconst(L, Y1, Layer, Wid, Row, Layerf, Widf, Rowf), !.

findcontact(X, Y, Widf, Layerf, Cwid, Clayer) :-
   cont(Type, pt(X, Y), Oset, Node),
   width(Type, Conw),
   width(Layerf, Fwid),
   Nfw is Fwid * Widf,
   (Nfw > Conw->
      Clayer = Layerf,
      Cwid = Widf;
      Clayer = Type,
      Cwid =:= 1), !.
findcontact(X, Y, Widf, Layerf, Widf, Layerf) :- !.

addconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl) :-
   tyconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl);
   tyconst(S, L, Layerl, Widl, Rows, Layers, Wids, Rowl), !.
addconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl) :-
   larger(Rows, Rowl, Sr, Lr),
   assert(tyconst(S, L, Layers, Wids, Sr, Layerl, Widl, Lr)), !.

% add lower constraint.
addlcons(Row, Layer, S, L, Wid, Node, []).
addlcons(Row, Layer, S, L, Wid, Node, List) :-
   Layer = tspot, !.
addlcons(Row, Layer, S, L, Wid, Node, [Element|Llist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Row = Rowf, !.
addlcons(Row, Layer, S, L, Wid, Node, [Element|Llist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Nodef = Node, !.
addlcons(Row, Layer, S, L, Wid, Node, [Element|Llist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   Layerf = tspot,
   addlcons(Row, Layer, S, L, Wid, Node, Llist), !.
addlcons(Row, Layer, S, L, Wid, Node, [Element|Llist]) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Element),
   findcontact(Row, Y2, Widf, Layerf, Cwid, Clayer),
   addconst(Y2, S, Layerf, Widf, Rowf, Layer, Wid, Row), !.
```

```
test(p).
test(pd).
test(pdtrans).
test(nd).
test(ndtrans).

rep(m1, m1).
rep(m2, m2).
rep(pd, dif).
rep(nd, dif).
rep(pdtrans, dif).
rep(pdtrans, p).
rep(ndtrans, dif).
rep(ndtrans, p).
rep(p, p).

% can be pd or nd, choose nd.
dual(p, nd).
dual(nd, p).
dual(pd, p).

% get elist.
lretract(Layer, List) :-
   rep(Layer, Rep),
   retract(elist(Rep, List)), !.

% get elist without retracting
lbind(Layer,List) :-
   rep(Layer, Rep),
   elist(Rep, List),!.

% put elist.
lassert(Layer, List) :-
   rep(Layer, Rep),
   assert(elist(Rep, List)), !.

% get contacts .
getcel(Row, Type, Y, Node) :-
   cont(Type, pt(Row, Y), Oset, Node).

% get wires.
getwel(Row, Layer, Y1, Y2, Wid, Node) :-
   wire(Layer, pt(Row, Y1), pt(Row,  Y2), Wid, Node),
   (var(Wid)->
       Wid = 1;
       true).


% convert ydist values to xdist values (change name).
ydtoxd :-
   retract(ydist(R1, R2, D)),
   assert(xdist(R1, R2, D)),
   ydtoxd, !.
ydtoxd.


% compaction for y dimension.
ycompact :-
   maxcol(Col1),
   initmap,
   inityconst(Col1),
   (show -> write('beginning pass 3...'), nl ; true),
   (show -> write('current column:'), nl ; true),
   ycoordey(0), !,
   rmmap(top),
   initmap,
   (show -> write('beginning pass 4...'), nl ; true),
   (show -> write('current column:'), nl ; true),
   ycoordeyb(Col1),
   rmmap(bottom), !.
```

```
inityconst(0).
inityconst(Col1) :-
   Lastcol is Col1 - 1,
   ydist(Lastcol, Col1, 0),
   initconst(Lastcol), !.
inityconst(Col1) :-
   Lastcol is Col1 - 1,
   assert(ydist(Lastcol, Col1, 0)),
   initconst(Lastcol), !.

ycoordey(Col) :-
   maxcol(Col1),
   Col1 < Col, !.
ycoordey(Col) :-
   getcol(Col),
   (show -> write(Col), nl ; true),
   Nextcol is Col + 1, !,
   ycoordey(Nextcol), !.

ycoordeyb(Col) :-
   Col < 0, !.
ycoordeyb(Col) :-
   getcol(Col),
   (show -> write(Col), nl ; true),
   Nextcol is Col - 1,
   ycoordeyb(Nextcol), !.

getcol(Col) :-
   getcyel(Col, Type, Y, Node),
   contlayer(Type, Layer1, Layer2),
   yappendel(Col, Layer1, Y, Y, 1, Node, Type),
   yappendel(Col, Layer2, Y, Y, 1, Node, Type),
   fail, !.
getcol(Col) :-
   getwyel(Col, Layer, X1, X2, Wid, Node),
   larger(X1, X2, S, L),
   yappendel(Col, Layer, S, L, Wid, Node, Layer),
   fail, !.
getcol(Col) :-
   gethpic(Col, X, Node, Width),
   lretract(pd, Elist),
   addyeltomap(Col, tspot, X, X, Width, Node, Elist, Elistout),
   lassert(pd, Elistout),
   fail, !.
getcol(Col).

yappendel(Col, Layer, S, L, Wid, Node, Type) :-
   test(Layer),
   lretract(Layer, Elist),
   addyeltomap(Col, Type, S, L, Wid, Node, Elist, Elistout), !,
   lassert(Layer, Elistout),
   dual(Layer, Duolayer),
   lretract(Duolayer, Duoelist),
   addyeltomap(Col, Type, S, L, Wid, Node, Duoelist, Dontcare),
   lassert(Duolayer, Duoelist), !.
yappendel(Col, Layer, S, L, Wid, Node, Type) :-
   lretract(Layer, Elist),
   addyeltomap(Col, Type, S, L, Wid, Node, Elist, Elistout),
   lassert(Layer, Elistout), !.

addyeltomap(Col, Layer, S, L, Wid, Node, Elist, Elistout) :-
   searchlist(Col, Layer, S, L, Wid, Node, Elist, Glist, Llist, Withinlist, y),
   yeladd(Col, Layer, S, L, Wid, Node, Glist, Llist, Withinlist, Elistout), !.

yeladd(Col, Layer, S, L, Wid, Node, Glist, Llist, Withinlist, Elistout) :-
   listio(Layer, S, L, Wid, Node, Col, Elementpac),
   append([Elementpac], Withinlist, NewWithin),
   append(NewWithin, Glist, Newglist),
   append(Llist, Newglist, Elistout), !.
```

```
getcyel(Col, Type, X, Node) :-
   cont(Type, pt(X, Col), Oset, Node).

getwyel(Col, Layer, X1, X2, Wid, Node) :-
   wire(Layer, pt(X1, Col), pt(X2, Col), Wid, Node),
   (var(Wid)->
      Wid = 1;
      true).

searchlist(Row, Layer, S, L, Wid, Node, [], [], [], [], Direction).
searchlist(Row, Layer, S, L, Wid, Node, [], _, _, _, Direction).
searchlist(Row, Layer, S, L, Wid, Node, [Elem|Elistin], [Elem|Glist], Llist,
           Withinlist, Direction ) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Elem),
   Y1 > L, !,
   searchlist(Row, Layer, S, L, Wid, Node, Elistin, Glist, Llist,
              Withinlist, Direction), !.
searchlist(Row, Layer, S, L, Wid, Node, [Elem|Elistin], Glist, [Elem|Llist],
           Withinlist, Direction) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Elem),
   S > Y2, !,
   searchlist(Row, Layer, S, L, Wid, Node, Elistin, Glist, Llist,
              Withinlist, Direction), !.
searchlist(Row, Layer, S, L, Wid, Node, [Elem|Elistin], Glist, Llist,
           [Elem|Withinlist], Direction) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Elem),
   Row = Rowf, !,
   searchlist(Row, Layer, S, L, Wid, Node, Elistin, Glist, Llist, Withinlist,
              Direction), !.
searchlist(Row, Layer, S, L, Wid, Node, [Elem|Elistin], Glist, Llist,
           Withinlist, Direction) :-
   listio(Layerf, Y1, Y2, Widf, Nodef, Rowf, Elem), !,
   checonst(Layerf, Widf, Nodef, Rowf, Layer, Wid, Node, Row), !,
   searchlist(Row, Layer, S, L, Wid, Node, Elistin, Glist, Llist, Withinlist,
              Direction), !.

listio(Layer, S, L, Wid, Node, Grid, Element) :-
   Element = [Layer, S, L, Wid, Node, Grid], !.

checonst(Layerf, Widf, Nodef, Rowf, Layer, Wid, Node, Row) :-
   contacts(Layerf, Layer),
   larger(Row, Rowf, Rows, Rowl),
   findrdist(Rows, Rowl, 0, Dist), !,
   mindist(Layer, Wid, Layerf, Widf, Mindist), !,
   direction(Row, Rowf, Lastrow), !,
   compdist(Dist, Mindist, Row, Lastrow), !.
checonst(Layerf, Widf, Nodef, Rowf, Layer, Wid, Node, Row) :-
   Nodef = Node,
   Nodef =\= -1, !.
checonst(Layerf, Widf, Nodef, Rowf, Layer, Wid, Node, Row) :-
   larger(Row, Rowf, Rows, Rowl),
   findrdist(Rows, Rowl, 0, Dist), !,
   mindist(Layer, Wid, Layerf, Widf, Mindist), !,
   direction(Row, Rowf, Lastrow), !,
   compdist(Dist, Mindist, Row, Lastrow), !.

contacts(Layer1, Layer2) :-
   contlayer(Layer1, _, _),
   contlayer(Layer2, _, _), !.

findrdist(Colg, Colg, Idist, Idist).
findrdist(Coll, Colg, Idist, Dist) :-
   Nextcol is Coll + 1, !,
   ydist(Coll, Nextcol, Distbet),
   Newidist is Idist + Distbet,
   findrdist(Nextcol, Colg, Newidist, Dist), !.
```

```
%space is spacing distance in units....
%Wspace is half of the minimum width space for the layer.
mindist(Layer, Wid, Layerf, Widf, Dbetob) :-
   space(Layer, Layerf, Dist),
   width(Layer, Wspace),
   Widmod is (Wid * Wspace),
   width(Layerf, Wspacef),
   Widfmod is (Widf * Wspacef),
   Dbetob is Dist + Widmod + Widfmod.

direction(Row, Crow, Lrow) :-
   Crow < Row,
   Lrow is Row - 1, !.
direction(Row, Crow, Lrow) :-
   Lrow is Row + 1, !.

compdist(Dist, Dbetob, Row, Lastrow) :-
   Dist >= Dbetob, !.
compdist(Dist, Dbetob, Row, Lastrow) :-
   larger(Row, Lastrow, Srow, Grow),
   retract(ydist(Srow, Grow, Distbet)),
   Difdist is Dbetob - Dist,
   Newdist is Difdist + Distbet,
   assert(ydist(Srow, Grow, Newdist)), !.


trueydist(Col, Dist) :-
   maxcol(Maxcol),
   Col >= Maxcol,
   assert(hcol(0, 0)), !.
trueydist(Col, Dist) :-
   Nextcol is Col + 1,
   ydist(Col, Nextcol, Ddist),
   Newdist is Dist + Ddist,
   assert(hcol(Nextcol, Newdist)),
   trueydist(Nextcol, Newdist), !.


truexdist(Row, Dist) :-
   maxrow(Maxrow),
   Row >= Maxrow,
   assert(hrow(0, 0)), !.
truexdist(Row, Dist) :-
   Nextrow is Row + 1,
   xdist(Row, Nextrow, Ddist),
   Newdist is Dist + Ddist,
   assert(hrow(Nextrow, Newdist)),
   truexdist(Nextrow, Newdist), !.


ordconst :-
   getconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl),
   makehard(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl),
   fail, !.
ordconst.

getconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl) :-
   retract(tyconst(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl)).

% remove redundant diagonal constraints and
% determine direction of diagonal stretch.
```

```
makehard(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl) :-
   findistx(Rows, Rowl, Xdist),
   findisty(S, L, Ydist),
   mindiagdist(Xdist, Ydist, Layers, Wids, Layerl, Widl, Dbetob),
   Difdist is (Dbetob * Dbetob),
   Xdistsq is (Xdist * Xdist),
   Ydistsq is (Ydist * Ydist),
   Alreadist is Xdistsq + Ydistsq,
   (Alreadist < Difdist->
      (Xdistsq >= Ydistsq ->
         assert(rcon(Rows, Rowl, S, L, Wids, Widl, Layers, Layerl, Difdist));
         assert(ccon(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist)));
      true), !.
makehard(S, L, Layers, Wids, Rows, Layerl, Widl, Rowl).

findistx(Rowl, Rowg, Dist) :-
   hrow(Rowg, Rowgv),
   hrow(Rowl, Rowlv),
   Dist is Rowgv - Rowlv, !.

findisty(Coll, Colg, Dist) :-
   hcol(Colg, Colgv),
   hcol(Coll, Collv),
   Dist is Colgv - Collv, !.


% generate and sort the list of diagonal constraints
makediag :-
   assembler(Xcon),
   quisort(Xcon, Newxcon),
   assemblec(Ycon),
   quisort(Ycon, Newycon),
   settlercon(Newxcon),
   settleccon(Newycon), !.

% put diagonal constraints into a list
assembler([Rc|List]) :-
   retract(rcon(Rows, Rowl, S, L, Wids, Widl, Layers, Layerl, Difdist)),
   Rc = con(Rows, Rowl, S, L, Wids, Widl, Layers, Layerl, Difdist),
   assembler(List), !.
assembler([]).

assemblec([Rc|List]) :-
   retract(ccon(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist)),
   Rc = con(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist),
   assembler(List), !.
assemblec([]).

% Resolve diagonal constraints

settlercon([]).
settlercon([[]]).
settlercon([Elem|List]) :-
   Elem = con(Rows, Rowl, S, L, Wids, Widl, Layers, Layerl, Difdist),
   findistx(Rows, Rowl, Xdist),
   findisty(S, L, Ydist),
   Xdistsq is (Xdist * Xdist),
   Ydistsq is (Ydist * Ydist),
   Alreadist is Xdistsq + Ydistsq,
   (Difdist > Alreadist->
   Newxdist is Difdist - Ydistsq,
   sqrt(Newxdist, Nxs),
   sqrt(Xdistsq, Xdis),
   Pnxs is Nxs - Xdis,
   maxrow(Mr),
   Row is Rows + 1,
   adjustrow(Row, Mr, Pnxs);
   true),
   settlercon(List), !.
```

```
settleccon([]).
settleccon([[]]).
settleccon([Elem|List]) :-
   Elem = con(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist),
   findistx(Rows, Rowl, Xdist),
   findisty(S, L, Ydist),
   mindiagdist(Xdist, Ydist, Layers, Wids, Layerl, Widl, Dbetob),
   Difdist is (Dbetob * Dbetob),
   Xdistsq is (Xdist * Xdist),
   Ydistsq is (Ydist * Ydist),
   Alreadist is Xdistsq + Ydistsq,
   (Difdist > Alreadist->
      Newydist is Difdist - Xdistsq,
      sqrt(Newydist, Nys),
      sqrt(Ydistsq, Ydis),
      Pnys is Nys - Ydis,
      maxcol(Mc),
      Col is S + 1,
      adjustcol(Col, Mc, Pnys);
      true),
   settleccon([List]), !.

% Stretch due to diagonal constraints; note xdistt and ydist are not
% updated.  The change from xdist and ydist to hrow and hcol is to
% save time (less frequent changes, so we don't pay as much for having
% to update all the hcols and hrows greater than the hrow or hcol to
% be modified.)

adjustrow(Row, Maxrow, Dist) :-
   Row > Maxrow, !.
adjustrow(Row, Maxrow, Dist) :-
   retract(hrow(Row, Fdist)),
   Ndist is Fdist + Dist,
   Newrow is Row + 1,
   assert(hrow(Row, Ndist)),
   adjustrow(Newrow, Maxrow, Dist), !.

adjustcol(Col, Maxcol, Dist) :-
   Col > Maxcol, !.
adjustcol(Col, Maxcol, Dist) :-
   retract(hcol(Col, Fdist)),
   Ndist is Fdist + Dist,
   Newcol is Col + 1,
   assert(hcol(Col, Ndist)),
   adjustcol(Newcol, Maxcol, Dist), !.

% determine the minimum distance possible between two diagonal elements
mindiagdist(0, _, Layer, Wid, Layerf, Widf, Dbetob) :-
   mindist(Layer, Wid, Layerf, Widf, Dbetob), !.
mindiagdist(_, 0, Layer, Wid, Layerf, Widf, Dbetob) :-
   mindist(Layer, Wid, Layerf, Widf, Dbetob), !.
mindiagdist(Xdist, Ydist, Layer, Wid, Layerf, Widf, Dbetob) :-
   space(Layer, Layerf, Dist),
   width(Layer, Wspace),
   Widmod is (Wid * Wspace * 141421 / 100000 ),
   width(Layerf, Wspacef),
   Widfmod is (Widf * Wspacef * 141421 / 100000 ),
   Dbetob is Dist + Widmod + Widfmod, !.


% double all distances (CIF hates fractions)
doubled :-
   retract(hcol(Nextcol, Newdist)),
   Double is Newdist*2+1,
   floor(Double, Vnewdist),
   assert(col(Nextcol, Vnewdist)),
   doubled, !.
```

```
doubled :-
   retract(hrow(Nextrow, Newdist)),
   Double is Newdist*2+1,
   floor(Double, Vnewdist),
   assert(row(Nextrow, Vnewdist)),
   doubled, !.
doubled.


split(H, [A|X], [A|Y], Z) :- order(A, H), split(H, X, Y, Z).
split(H, [A|X], Y, [A|Z]) :- order(H, A), split(H, X, Y, Z).
split(_, [], [], []).

quisort([H|T], S) :-
   split(H, T, A, B),
   quisort(A, A1),
   quisort(B, B1),
   append(A1, [H|B1], S).
quisort([], []).

order(A, H) :-
   con(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist) = A,
   con(S1, L1, Rows1, Rowl1, Wids1, Widl1, Layers1, Layerl1, Difdist1) = H,
   isin(S, L, S1, L1), !.
order(A, H) :-
   con(S, L, Rows, Rowl, Wids, Widl, Layers, Layerl, Difdist) = A,
   con(S1, L1, Rows1, Rowl1, Wids1, Widl1, Layers1, Layerl1, Difdist1) = H,
   S =< S1, !.
order(A, H) :-
   term(Side, Loc, Lay, Wid, Nod) = A,
   term(Side2, Loc2, Lay2, Wid2, Nod2) = H,
   Loc =< Loc2, !.


layer(p).         % Poly
layer(m1).        % Metal 1.
layer(m2).        % Metal 2.
layer(nd).        % N Diffusion
layer(pd).        % P Diffusion.
layer(nw).        % N Well.
layer(pw).        % P Well.
layer(ccut).      % Generic Contact Cut.
layer(active).    % Active area
layer(m1m2).      % Contact -- Metal 1 to Metal 2.
layer(m1p).       % Contact -- Metal 1 to Poly.

% lambda(1.5).   Lambda value.
lambda(X) :- X is (15/10).

ciflayer([m1, m1m2, m1p], 'CMF').
ciflayer([m2, m1m2], 'CMS').
ciflayer([active, nd, pd], 'CAA').
ciflayer([p, m1p], 'CPG').
ciflayer([nd], 'CSN').
ciflayer([pd], 'CSP').
ciflayer([ccut], 'CCA').
ciflayer([nw], 'CWN').
ciflayer([pw], 'CWP').
ciflayer([m1m2], 'CVA').
ciflayer([m1p], 'CCP').
```

```
% MOSIS rules (in 2 units per mosis unit).

% Width rules (half of actual)
width(nd, 2).
width(ndtrans, 2).
width(pdtrans, 2).
width(pd, 2).
width(p, 2).
width(m2, 3).
width(m1, 3).
width(m1m2, 4).
width(m1p, 4).
width(m1nd, 4).
width(m1pd, 4).
width(tspot, 2).  % same as poly
width(ccut, 2).

% Spacing rules (full distances)

maxspace(m1, 12).
maxspace(m2, 16).
maxspace(pd, 48).
maxspace(nd, 48).
maxspace(pdtrans, 48).
maxspace(ndtrans, 48).
maxspace(p, 12).
maxspace(m1m2, 16).
maxspace(m1p, 12).
maxspace(m1nd, 48).
maxspace(m1pd, 48).

space(nd, nd, 6) :- !.
space(nd, ndtrans, 6) :- !.
space(ndtrans, ndtrans, 6) :- !.
space(pd, pd, 6) :- !.
space(pd, pdtrans, 6) :- !.
space(pdtrans, pdtrans, 6) :- !.
space(ndtrans, pdtrans, 6) :- !.
space(ndtrans, pdtrans, 6) :- !.
space(p, p, 4) :- !.
space(m1, m1, 6) :- !.
space(m2, m2, 8) :- !.
space(m1, m1m2, 6) :- !.
space(m2, m1m2, 8) :- !.
space(m1m2, m1, 6) :- !.
space(m1m2, m2, 8) :- !.
space(pd, nd, 24) :- !.
space(pd, ndtrans, 24) :- !.
space(nd, pd, 24) :- !.
space(nd, pdtrans, 24) :- !.
space(ndtrans, pdtrans, 24) :- !.
space(p, pd, 4) :- !.
space(p, pdtrans, 6) :- !. %p to dif + overhang distance
space(p, ndtrans, 6) :- !. %p to dif + overhang distance
space(p, nd, 4) :- !.
space(tspot, tspot, 6) :- !.
space(tspot, m1pd, 6) :- !.
space(m1pd, tspot, 6) :- !.
space(tspot, pd, 6) :- !.
space(tspot, nd, 6) :- !.
space(tspot, pdtrans, 8) :- !.
space(tspot, ndtrans, 8) :- !.
space(tspot, m1nd, 6) :- !.
space(m1nd, tspot, 6) :- !.
space(p, tspot, 8) :- !.
space(m1m2, m1m2, Dist) :-
    space(m2, m2, Dist), !.
space(m1pd, m1nd, Dist) :-
    space(pd, nd, Dist), !.
space(m1nd, m1pd, Dist) :-
   space(pd, nd, Dist), !.
space(L1, L2, Dist) :-
   contlayer(L1, _, SL1),
   contlayer(L2, _, SL2),
   space(m1, m1, Dist), !.
space(L1, m1, Dist) :-
   contlayer(L1, _, _),
   space(m1, m1, Dist), !.
space(m1, L1, Dist) :-
   contlayer(L1, _, _),
   space(m1, m1, Dist), !.
space(L1, L2, Dist) :-
   contlayer(L1, _, M11),
   space(M11, L2, Dist), !.
space(L1, L2, Dist) :-
   contlayer(L2, _, M12),
   space(L1, M12, Dist), !.
space(L1, L2, Dist) :-
   space(L2, L1, Dist), !.

pohang(2).


contlayer(m1m2, m1, m2).
contlayer(m1p, m1, p).
contlayer(m1nd, m1, nd).
contlayer(m1pd, m1, pd).

diftoed(6).


% The following code will take in a sticks description on a lambda grid
% and produce a CIF description, using a specified value of lambda.
% The name of output file is X, without any . extensions.

genbox :-
   makebox,
   makewell,
   makelabel, !.

makebox :-
   wire(Layer, pt(X1, Y1), pt(X2, Y2), Wid, Node),
   (var(Wid)->
       Wid = 1;
       true),
   getlayer(Layer, Rlayer),
   (X1 = X2 ->
      procxwire(Rlayer, X1, Y1, Y2, Wid, Node);
      procywire(Rlayer, Y1, X1, X2, Wid, Node)),
   fail, !.
makebox :-
   cont(Type, pt(Row, Y), Oset, _),
   procont(Type, pt(Row, Y), Oset, _),
   fail, !.
makebox :-
   trans(Type, pt(Sx, Sy), pt(Gx, Gy), pt(Dx, Dy), W, L, Sn, Gn, Dn),
   (Sx = Dx ->
      proctrans(Type, Gx, Gy, W, L, x);
      proctrans(Type, Gx, Gy, W, L, y)),
   fail, !.
makebox.

getlayer(Layer, Rlayer) :-
   transtype(Rlayer, Layer), !.
getlayer(Layer, Layer).
```

```
procxwire(Layer, Row, Y1, Y2, Wid, Node) :-
   width(Layer, Minwid),
   Boxwidth is 4*Minwid*Wid,
   larger(Y1, Y2, Ys, Yl),
   col(Ys, Sloc),
   col(Yl, Lloc),
   maxcol(Col),
   maxrow(Mrow),
   (Ys =:= 0->
      assert(term(bottom, Row, Layer, Wid, Node));
      true),
   (Yl = Col->
      assert(term(top, Row, Layer, Wid, Node));
      true),
   (Row =:= 0->
      assert(jterm(left, Y1, Y2, Layer, Wid, Node));
      true),
   (Row = Mrow->
      assert(jterm(right, Y1, Y2, Layer, Wid, Node));
      true),
   row(Row, Centerx),
   Boxlength is (Lloc - Sloc + 4*Minwid),
   Centery is (Lloc + Sloc)/2,
   assert(pbox(Layer, Boxwidth, Boxlength, Centerx, Centery)), !.

procywire(Layer, Col, X1, X2, Wid, Node) :-
   width(Layer, Minwid),
   Boxlength is 4*Minwid*Wid,
   larger(X1, X2, Xs, Xl),
   row(Xs, Sloc),
   row(Xl, Lloc),
   maxrow(Row),
   maxcol(Mcol),
   (Xs =:= 0->
      assert(term(left, Col, Layer, Wid, Node));
      true),
   (Xl = Row->
      assert(term(right, Col, Layer, Wid, Node));
      true),
   (Col =:= 0->
      assert(jterm(bottom, X1, X2, Layer, Wid, Node));
      true),
   (Col = Mcol->
      assert(jterm(top, X1, X2, Layer, Wid, Node));
      true),
   col(Col, Centery),
   Boxwidth is (Lloc - Sloc + 4*Minwid),
   Centerx is (Lloc + Sloc)/2,
   assert(pbox(Layer, Boxwidth, Boxlength, Centerx, Centery)), !.

procont(Type, pt(X, Y), Oset, _) :-
   contlayer(Type, Layr1, Layr2),
   width(Type, Minwid),
   width(ccut, Cminwid),
   Cwid is 4 * Cminwid,
   Boxlength is 4*Minwid,
   col(Y, Centery),
   row(X, Centerx),
   (Type = m1pd ->
      assert(pbox(Layr1, Boxlength, Boxlength, Centerx, Centery)),
      assert(pbox(Layr2, Boxlength, Boxlength, Centerx, Centery)),
      assert(pbox(ccut, Cwid, Cwid, Centerx, Centery));
      true),
   (Type = m1nd ->
      assert(pbox(Layr1, Boxlength, Boxlength, Centerx, Centery)),
      assert(pbox(Layr2, Boxlength, Boxlength, Centerx, Centery)),
      assert(pbox(ccut, Cwid, Cwid, Centerx, Centery));
      true),
   (Type = m1m2 ->
      assert(pbox(m1m2, Boxlength, Boxlength, Centerx, Centery));
      assert(pbox(ccut, Cwid, Cwid, Centerx, Centery));
      true),
   (Type = m1p ->
      assert(pbox(m1p, Boxlength, Boxlength, Centerx, Centery));
      assert(pbox(ccut, Cwid, Cwid, Centerx, Centery));
      true), !.

proctrans(Type, Gx, Gy, W, L, Orient) :-
   width(p, Pwidth),
   width(Type, Dwid),
   pohang(Ohang),
   row(Gx, Centerx),
   col(Gy, Centery),
   Lovw is L/W,
   larger(1, Lovw, Dc, Pwid),
   Wovl is W/L,
   larger(1, Wovl, Dcr, Pht),
   Boxwid is 4*Pwid*Pwidth,
   Boxlen is 4*(Pht*Dwid + Ohang),
   (Orient = x ->
      assert(pbox(p, Boxlen, Boxwid, Centerx, Centery));
      assert(pbox(p, Boxwid, Boxlen, Centerx, Centery))), !.

makewell :-
        pbox(pd,L,W,X,Y),
        diftoed(Edist),
        Newl is L + Edist * 4,
        Neww is W + Edist * 4,
        (Newl < 24->
           Vnewl = 24;
           Vnewl = Newl),
        (Neww < 24->
           Vneww = 24;
           Vneww = Neww),
        assert(pbox(pw,Vnewl,Vneww,X,Y)),
        assert(pbox(active,L,W,X,Y)),
        fail.
makewell :-
        pbox(nd,L,W,X,Y),
        diftoed(Edist),
        Newl is L + Edist * 4,
        Neww is W + Edist * 4,
        (Newl < 24->
           Vnewl = 24;
           Vnewl = Newl),
        (Neww < 24->
           Vneww = 24;
           Vneww = Neww),
        assert(pbox(nw,Vnewl,Vneww,X,Y)),
        assert(pbox(active,L,W,X,Y)),
        fail.
makewell.

%makewell :-
%       grow(pw),
%       mergeboxes,
%       shrink(pw),
%       grow(nw),
%       mergeboxes,
%       shrink(nw), !.
```

```
makelabel :-
        retract(node(X,Y,Label, Type)),
        (contlayer(Type, Layr1, Layr2)->
         Ilayer = m1;
         Ilayer = Type),
        row(X, Xdist),
        col(Y, Ydist),
        ciflayer(Ilayer, CLayer),
        assert(plabel(CLayer, Xdist, Ydist, Label)),
        fail.
makelabel :-
        pin(Dir, pt(X,Y),Type, Wid, Label, Name, Cell),
        (contlayer(Type, Layr1, Layr2)->
         Ilayer = m1;
         Ilayer = Type),
        row(X, Xdist),
        col(Y, Ydist),
        ciflayer(Ilayer, CLayer),
        assert(plabel(CLayer, Xdist, Ydist, Label)),
        fail.
makelabel :-
        pin(pt(X,Y),Type, Wid, Label, Name),
        (contlayer(Type, Layr1, Layr2)->
         Ilayer = m1;
         Ilayer = Type),
        row(X, Xdist),
        col(Y, Ydist),
        ciflayer(Ilayer, CLayer),
        assert(plabel(CLayer, Xdist, Ydist, Label)),
        fail.
makelabel.


balance :-
   assert(lowy(0)),
   assert(lowx(0)),
   assert(right(0)),
   assert(top(0)),
   shift,
   adjustbox,
   mods, !.

shift :-
   pbox(_, L, W, X, Y),
   xlow(L, X),
   ylow(W, Y),
   fail, !.
shift.

xlow(L, X) :-
   lowx(Lowx),
   right(Hix),
   Newlowx is X - (L/2),
   Newhix is X + (L/2),
   (Newlowx < Lowx->
      retract(lowx(_)),
      assert(lowx(Newlowx));
      true),
   (Newhix > Hix->
      retract(right(_)),
      assert(right(Newhix));
      true), !.
```

```
ylow(W, Y) :-
   lowy(Lowy),
   top(Hiy),
   Newlowy is Y - (W/2),
   Newhiy is Y + (W/2),
   (Newlowy < Lowy->
      retract(lowy(_)),
      assert(lowy(Newlowy));
      true),
   (Newhiy > Hiy->
      retract(top(_)),
      assert(top(Newhiy));
      true), !.

adjustbox :-
   lowy(YS),
   lowx(XS),
   retract(pbox(D, L, W, X, Y)),
   Newx is (X - XS),
   Newy is (Y - YS),
   assert(box(D, L, W, Newx, Newy)),
   fail, !.
adjustbox :-
   lowy(YS),
   lowx(XS),
   retract(plabel(CLayer, Xdist, Ydist, Label)),
   Newx is (Xdist - XS),
   Newy is (Ydist - YS),
   assert(label(CLayer, Newx, Newy, Label)),
   fail, !.
adjustbox :-
   lowy(Ly),
   lowx(Lx),
   retract(right(Ohix)),
   retract(top(Ohiy)),
   Newx is (Ohix - Lx),
   Newy is (Ohiy - Ly),
   assert(right(Newx)),
   assert(top(Newy)), !.

mods :-
   lowx(Lx),
   row(Row, Val),
   NewVal is (Val-Lx),
   assert(trow(Row, NewVal)),
   fail, !.
mods :-
   lowy(Ly),
   col(Col, Val),
   NewVal is (Val-Ly),
   assert(tcol(Col, NewVal)),
   fail, !.
mods.


expand(Name) :-
   open_file(Name),
   writecellbegin(Name, 1),
   writeboxes,
   writelabels,
   writecellend, !.

%Take boxes from database and write out CIF file.
writeboxes :-
        ciflayer(Layerlist, Layername),
        write('L '), write(Layername), write(';'), nl,
        writeboxes1(Layerlist),
        fail.
writeboxes.
```

```
writeboxes1([]).
writeboxes1([Layer|Rest]) :-
        box(Layer,L,W,X,Y),
        write('B '),
        write(L), write(' '),
        write(W), write(' '),
        write(X), write(' '),
        write(Y), write(';'),
        nl,
        fail.
writeboxes1([Layer|Rest]) :-
   writeboxes1(Rest), !.

% remove(X,Y,Label,Type) :-
%          retract(node(X,Y,Label,Type)),
%        fail, !.
% remove(X,Y,Label,Type).

% writelabels :-
%        node(X,Y,Label,Type),
%        assert(onode(X,Y,Label,Type)),
%        remove(X,Y,Label,Type),
%        writelabels.

writelabels :-
        label(Clayer, Xdist, Ydist, CLabel),
        write('94 '), write(CLabel),
        tab(1), write(Xdist),
        tab(1), write(Ydist),
        write(' '), write(Clayer), write(';'), nl,
        fail.
writelabels.

%Write out definition start line of CIF cell.
writecellbegin(X, N) :-
        lambda(L),               %Fetch lambda value.
        A is L*100,              %Find first scale factor.
        write('DS'),
        write(N),                %Hard wired cell number for now.
        write(' '),
        write(A),                %First scale factor.
        write(' 4;'),            %Second scale factor.
        nl,
        write('9 '), write(X), write(';'), nl.  %Module name.

%Write end cell definition instruction on CIF file.
writecellend :-
        write('DF;'), nl,
        write('C 1;'), nl,
        write('End'), nl,
        told, !.

%Open file with .cif extension.
open_file(X) :-
        name(X,L),
        append(L,".cif",L1),
        name(Y,L1),
        tell(Y).


writefile(Name) :-
   writespace(Name),
   writebbox(Name),
   name(Name, S0),
   append(S0, ".bdr", S1),
   name(Name1, S1),
   tell(Name1),
   writeterm(Name),
   writedge(Name),
   told, !.
```

```
writespace(Name) :-
   name(Name, S0),
   append(S0, ".space", S1),
   name(Name1, S1),
   tell(Name1),
   writex(0, Name),
   writey(0, Name),
   writerc(Name), !,
   told.

writebbox(Name) :-
   name(Name, S0),
   append(S0, ".bbox", S1),
   name(Name1, S1),
   tell(Name1),
   writepin(Name),
   writerc(Name), !,
   told.

writepin(Cellname) :-
   pin(Dir, pt(X, Y), Layer, Wid, Node, Nane, Cellname),
   trow(X, Xloc),
   tcol(Y, Yloc),
   P=pin(Dir, pt(Xloc, Yloc), Layer, Wid, Node, Nane, Cellname),
   write(P),
   write('.'),
   nl,
   fail, !.
writepin(Cellname) :-
   pin(pt(X, Y), Layer, Wid, Node, Nane),
   trow(X, Xloc),
   tcol(Y, Yloc),
   findwire(X, Y, Layer, Wid, Node, Dir),
   P=pin(Dir, pt(Xloc, Yloc), Layer, Wid, Node, Nane, Cellname),
   write(P),
   write('.'),
   nl,
   fail, !.
writepin(Cellname).

findwire(X, Y, Layer, Wid, Node, Dir) :-
   wire(Layer, pt(X, Y), pt(Ox, Oy), Wid, Node),
   findir(X, Ox, Y, Oy, Dir), !.
findwire(X, Y, Layer, Wid, Node, Dir) :-
   wire(Layer, pt(Ox, Oy), pt(X, Y), Wid, Node),
   findir(X, Ox, Y, Oy, Dir), !.

findir(X, X, Y, Oy, loy) :-
   Y = 0.
findir(X, X, Y, Oy, hiy).
findir(X, Ox, Y, Y, lox) :-
   X = 0.
findir(X, Ox, Y, Y, hix).

writex(R1, Cellname) :-
   maxrow(Rmax),
   R1 > Rmax, !.
writex(R1, Cellname) :-
   R2 is R1 + 1,
   trow(R1, Dist),
   Z = row(R1, Dist, Cellname),
   write(Z),
   write('.'),
   nl,
   writex(R2, Cellname), !.
```

```
writey(C1, Cellname) :-
   maxcol(Cmax),
   C1 > Cmax, !.
writey(R1, Cellname) :-
   R2 is R1 + 1,
   tcol(R1, Dist),
   Z = col(R1, Dist, Cellname),
   write(Z),
   write('.'),
   nl,
   writey(R2, Cellname), !.


xmatch(Xloc) :-
   retract(xdist(X1, X2, Dist)),
   Newx1 is X1 + Xloc,
   Newx2 is X2 + Xloc,
   xresolve(Newx1, Newx2, Dist),
   fail, !.
xmatch(Xloc).

xresolve(X1, X2, Dist) :-
   retract(gxdist(X1, X2, Gdist)),
   larger(Dist, Gdist, Dontcare, Ndist),
   assert(gxdist(X1, X2, Ndist)), !.
xresolve(X1, X2, Dist) :-
   assert(gxdist(X1, X2, Dist)), !.


ymatch(Yloc) :-
   retract(ydist(Y1, Y2, Dist)),
   Newy1 is Y1 + Yloc,
   Newy2 is Y2 + Yloc,
   yresolve(Newy1, Newy2, Dist),
   fail, !.
ymatch(Yloc).

yresolve(Y1, Y2, Dist) :-
   retract(gydist(Y1, Y2, Gdist)),
   larger(Dist, Gdist, Dontcare, Ndist),
   assert(gydist(Y1, Y2, Ndist)), !.
yresolve(Y1, Y2, Dist) :-
   assert(gydist(Y1, Y2, Dist)), !.


geterm(Bdr, B1, B2, [E1|Terms]) :-
   term(Bdr, X, Layer, Width, Node),
   in(X, B1, B2),
   termstr(X, Layer, Width, Node, E1),
   retract(term(Bdr, X, Layer, Width, Node)),
   geterm(Bdr, B1, B2, Terms), !.
geterm(Bdr, B1, B2, []).

in(X, B1, B2) :-
   X >= B1,
   X =< B2.

termstr(X, Layer, Width, Node, [X, Layer, Width, Node]).

seterm(Tbl, Lrl, Ntbl, Nlrl) :-
   retract(term(Side, Grid, Layer, Wid, Node)),
   Term = term(Side, Grid, Layer, Wid, Node),
   addlist(Side, Term, Tbl, Lrl, Newtbl, Newlrl),
   seterm(Newtbl, Newlrl, Ntbl, Nlrl), !.
seterm(Tbl, Lrl, Tbl, Lrl).

addlist(top, Term, Tbl, Lrl, [Term|Tbl], Lrl).
addlist(bottom, Term, Tbl, Lrl, [Term|Tbl], Lrl).
addlist(left, Term, Tbl, Lrl, Tbl, [Term|Lrl]).
addlist(right, Term, Tbl, Lrl, Tbl, [Term|Lrl]).
```

```
writeterm(Name) :-
   retract(jterm(Bdr, B1, B2, Layer, Wid, Node)),
   geterm(Bdr, B1, B2, Terms),
   Z = cell(Name,jterm(Bdr, B1, B2, Layer, Wid, Node, Terms)),
   write(Z),
   write('.'),
   nl,
   fail, !.
writeterm(Name) :-
   seterm([], [], Tbl, Lrl),
   sorterm(Tbl, Stbl),
   sorterm(Lrl, Slrl),
   Z = cell(Name,tbot,Stbl),
   write(Z),
   write('.'),
   nl,
   S = cell(Name,lrit,Slrl),
   write(S),
   write('.'),
   nl, !.
writeterm(Name).

spliterm(H, [A|X], [A|Y], Z) :- orderterm(A, H), spliterm(H, X, Y, Z).
spliterm(H, [A|X], Y, [A|Z]) :- orderterm(H, A), spliterm(H, X, Y, Z).
spliterm(_, [], [], []).

sorterm([H|T], S) :-
   spliterm(H, T, A, B),
   sorterm(A, A1),
   sorterm(B, B1),
   append(A1, [H|B1], S).
   sorterm([], []).

orderterm(A, H) :-
   term(Side, Grid, Layer, Wid, Node) = A,
   term(Side2, Grid2, Layer2, Wid2, Node2) = H,
   Grid =< Grid2, !.

distoside(bottom, Grid, Col) :-
   tcol(Grid, Col), !.
distoside(left, Grid, Row) :-
   trow(Grid, Row), !.
distoside(top, Grid, Idist) :-
   maxcol(Tc),
   tcol(Tc, Top),
   tcol(Grid, Col),
   Idist is Top - Col  , !.
distoside(right, Grid, Idist) :-
   maxrow(Tr),
   trow(Tr, Right),
   trow(Grid, Row),
   Idist is Right - Row, !.

side(top).
side(bottom).
side(left).
side(right).

vert(top).
vert(bottom).

listypes(dif).
listypes(p).
listypes(m1).
listypes(m2).
```

```
truebord(Side, [], []).
truebord(Side, [Element|List], [EL|Newlist]) :-
   listio(Layer, P1, P2, Wid, Node, Grid, Element),
   maxspace(Layer, Sdist),
   width(Layer, Wdist),
   Edged is 2 * Wdist * Wid,
   Dist is Sdist + Edged,
   distoside(Side, Grid, Bspace),
   Idist is Dist - Bspace,
   Idist > 0,
   bel(Layer, Wid, P1, P2, Grid, Node, EL),
   truebord(Side, List, Newlist), !.
truebord(Side, [Element|List], Newlist) :-
   truebord(Side, List, Newlist).

bel(Layer, Wid, P1, P2, Grid, Node, EL) :-
   EL = [Layer, Wid, P1, P2, Grid, Node], !.

writedge(Name) :-
   side(Side),
   listypes(Layer),
   border(Side, Layer, List),
   truebord(Side, List, Newlist),
   Z = bound(Side, Layer, Newlist, Name),
   write(Z),
   write('.'),
   nl,
   fail, !.
writedge(Name).

writerc(Name) :-
   writebound(Name),
   maxrow(Row),
   Z = maxrow(Row, Name),
   maxcol(Col),
   P = maxcol(Col, Name),
   top(Hix),
   Q = hiy(Hix, Name),
   right(Hiy),
   D = hix(Hiy, Name),
   write(Z),
   write('.'),
   nl,
   write(P),
   write('.'),
   nl,
   write(Q),
   write('.'),
   nl,
   write(D),
   write('.'),
   nl, !.

writebound(Name) :-
        rowbound(Llbound, Hlbound),
        Z = xbound(Llbound, Hlbound, Name),
        write(Z),
        write('.'),
        nl,
        colbound(Lcbound, Hcbound),
        A = ybound(Lcbound, Hcbound, Name),
        write(A),
        write('.'),
        nl, !.
```

```
writebound(Name) :-
        maxrow(Maxrow),
        maxcol(Maxcol),
        trow(0, R0),
        trow(Maxrow, Rm),
        tcol(0, C0),
        tcol(Maxcol, Cm),
        R = rowbound(R0, Rm, Name),
        write(R),
        write('.'),
        nl,
        C = colbound(C0, Cm, Name),
        write(C),
        write('.'),
        nl, !.

prpr([]) :- nl.
prpr([H|T]) :- write(H), tab(1), prpr(T).

clears :-
   retract(row(_, _)),
   fail, !.
clears :-
   retract(col(_, _)),
   fail, !.
clears.

symbfile :-
   clears,
   setsymbrow(0, 0),
   setsymbcol(0, 0), !.

setsymbrow(Row, Val) :-
   maxrow(Maxrow),
   Row > Maxrow, !.
setsymbrow(Row, Val) :-
   assert(row(Row, Val)),
   Newrow is Row+1,
   NewVal is Val+50,
   setsymbrow(Newrow, NewVal), !.

setsymbcol(Col, Val) :-
   maxcol(Maxcol),
   Col > Maxcol, !.
setsymbcol(Col, Val) :-
   assert(col(Col, Val)),
   Newcol is Col+1,
   NewVal is Val+50,
   setsymbcol(Newcol, NewVal), !.

ptobox :-
   retract(pbox(D, L, W, X, Y)),
   assert(box(D, L, W, X, Y)),
   fail, !.
ptobox.

isin(Ps, Pl, Bs, Bl) :-
   Ps >= Bs,
   Pl =< Bl, !.
isin(Ps, Pl, Bs, Bl) :-
   Pl >= Bs,
   Pl =< Bl, !.
isin(Ps, Pl, Bs, Bl) :-
   Pl >= Bl,
   Ps =< Bs, !.

growfactor(12).
```

```
grow(Layer) :-
   growfactor(Grow),
   retract(mwbox(Layer, Wid, Len, X, Y)),
   Newid is Grow + (Wid/2),
   Newle is Grow + (Len/2),
   X1 is X-Newid,
   Y1 is Y-Newle,
   X2 is X+Newid,
   Y2 is Y+Newle,
   assert(gbox(pt(X1, Y1), pt(X2, Y2))),
   fail, !.
grow(Layer).

mergeboxes :-
   retract(gbox(pt(X1, Y1), pt(X2, Y2))),
   findwithin(X1, Y1, X2, Y2, Gx1, Gy1, Gx2, Gy2),
   assert(fbox(pt(Gx1, Gy1), pt(Gx2, Gy2))),
   mergeboxes, !.
mergeboxes.

findwithin(Ix1, Iy1, Ix2, Iy2, Gx1, Gy1, Gx2, Gy2) :-
   gbox(pt(X1g, Y1g), pt(X2g, Y2g)),
   checkin(X1g, Y1g, X2g, Y2g, Ix1, Iy1, Ix2, Iy2),
   larger(Ix1, X1g, Smx1, Dc1),
   larger(Iy1, Y1g, Smy1, Dc2),
   larger(Ix2, X2g, Dc3, Lgx2),
   larger(Iy2, Y2g, Dc4, Lgy2),
   retract(gbox(pt(X1g, Y1g), pt(X2g, Y2g))),
   findwithin(Smx1, Smy1, Lgx2, Lgy2, Gx1, Gy1, Gx2, Gy2), !.
findwithin(X1, Y1, X2, Y2, X1, Y1, X2, Y2).

checkin(X1g, Y1g, X2g, Y2g, X1, Y1, X2, Y2) :-
   smaller(X1, Y1, X2g, Y2g),
   smaller(X2g, Y2g, X2, Y2), !.

smaller(X1g, Y1g, X1, Y1) :-
   X1g =< X1;
   Y1g =< Y1.

shrink(Layer) :-
   growfactor(Grow),
   retract(fbox(pt(X1, Y1), pt(X2, Y2))),
   Wid is X2-X1-2*Grow,
   Len is Y2-Y1-2*Grow,
   Cx is (X2 + X1)/2,
   Cy is (Y2 + Y1)/2,
   assert(pbox(Layer, Wid, Len, Cx, Cy)),
   fail, !.
shrink(Layer).

larger(E1, E2, E2, E1) :-
   E1 > E2, !.
larger(E1, E2, E1, E2).

append([], L, L).
append([X|L1], L2, [X|L3]) :-
   append(L1, L2, L3).

rev(O1,N1) :-
   nrev(O1,[],N1).

nrev([],Result,Result).
nrev([H|T],Sofar,Result) :-
   nrev(T,[H|Sofar],Result).
```

```
#option "
        > compactor uses floor/2 and sqrt/2.  If one of
        >
        > C_PL QUINTUS_PL
        >
        > is selected, then appropriate definitions for
        > floor/2 and sqrt/2 are included automatically."
#if C_PL
floor(X, I) :-
   I is floor(X).

sqrt(X, Y) :-
   Y is sqrt(X).
#elseif QUINTUS_PL
:- multifile tmp1/1.
:- multifile tmp2/1.
:- multifile tmp3/1.
:- multifile tmp4/1.
:- multifile wire/5.
:- multifile cont/4.
:- multifile pin/5.
:- multifile pin/6.
:- multifile pin/7.
:- multifile pbox/5.
:- multifile trans/9.
:- multifile node/4.
:- multifile tmpRowNum/1.
:- multifile tmpColNum/1.
:- multifile box/5.
:- multifile rowbound/2.
:- multifile rowbound/3.
:- multifile maxrow/1.
:- multifile maxcol/1.
:- multifile nwire/5.
:- multifile ntrans/9.
:- multifile ncont/4.
:- multifile npin/6.
:- multifile ncol/2.
:- multifile col/2.
:- multifile nrow/2.
:- multifile row/2.
:- multifile nbox/5.
:- multifile nlabel/4.
:- multifile label/4.
:- multifile nwire/5.
:- multifile nnode/4.
:- multifile ncont/4.
:- multifile node/4.
:- multifile term/5.
:- multifile jterm/6.
:- multifile plabel/4.
:- multifile lowx/1.
:- multifile right/1.
:- multifile lowy/1.
:- multifile top/1.
:- multifile gethpic/4.
:- multifile getvpic/4.
:- multifile tyconst/8.
:- multifile elist/2.
:- dynamic tmp1/1.
:- dynamic tmp2/1.
:- dynamic tmp3/1.
:- dynamic tmp4/1.
:- dynamic wire/5.
:- dynamic cont/4.
:- dynamic pin/5.
:- dynamic pin/6.
:- dynamic pin/7.
:- dynamic pbox/5.
:- dynamic trans/9.
:- dynamic node/4.
:- dynamic tmpRowNum/1.
:- dynamic tmpColNum/1.
:- dynamic box/5.
:- dynamic rowbound/2.
:- dynamic rowbound/3.
:- dynamic maxrow/1.
:- dynamic maxcol/1.
:- dynamic nwire/5.
:- dynamic ntrans/9.
:- dynamic ncont/4.
:- dynamic npin/6.
:- dynamic ncol/2.
:- dynamic col/2.
:- dynamic nrow/2.
:- dynamic row/2.
:- dynamic nbox/5.
:- dynamic nlabel/4.
:- dynamic label/4.
:- dynamic nwire/5.
:- dynamic nnode/4.
:- dynamic ncont/4.
:- dynamic node/4.
:- dynamic term/5.
:- dynamic jterm/6.
:- dynamic plabel/4.
:- dynamic lowx/1.
:- dynamic right/1.
:- dynamic lowy/1.
:- dynamic top/1.
:- dynamic gethpic/4.
:- dynamic getvpic/4.
:- dynamic tyconst/8.
:- dynamic elist/2.

:- ensure_loaded(library(math)).    % ffloor/2, sqrt/2

floor(X, I) :-
   ffloor(X, Y),
   I is integer(Y).
#else
#  message "WARNING: floor/2 and sqrt/2 must be defined"
#endif
```

inverter.sip

```
wire(m1, pt(0,0), pt(5,0),1,1).
wire(m1, pt(1,0), pt(1,2),1,1).
wire(m1, pt(0,10), pt(5,10),1,2).
wire(m1, pt(1,10), pt(1,8),1,2).
wire(m1, pt(3,8), pt(3,2),1,3).
wire(m1, pt(3,6), pt(5,6),1,3).
wire(p, pt(2,8), pt(2,2),1,4).
wire(p, pt(0,6), pt(2,6),1,4).
trans(nd, pt(1,2), pt(2,2), pt(3,2), 4, 2, 1, 4, 3).
trans(pd, pt(1,8), pt(2,8), pt(3,8), 2, 2, 2, 4, 3).
cont(m1nd, pt(1,2), na, 1).
cont(m1nd, pt(3,2), na, 3).
cont(m1pd, pt(1,8), na, 2).
cont(m1pd, pt(3,8), na, 3).
pin(top, (0, 6), p, 1, 4).
pin(bottom, (5, 6), m1, 1, 3).
maxrow(5).
maxcol(10).
```

```
wire(p,pt(1,2),pt(11,2),_57055,colorbus).
wire(p,pt(1,5),pt(11,5),_57252,p3).
wire(p,pt(1,8),pt(11,8),_57453,valid).
wire(p,pt(1,10),pt(11,10),_57654,colorbus).
wire(p,pt(1,12),pt(11,12),_57855,colbusbar).
wire(p,pt(1,14),pt(11,14),_58056,attacked).
wire(p,pt(1,17),pt(11,17),_58257,p3).
wire(p,pt(1,20),pt(11,20),_58458,valid).
wire(p,pt(1,22),pt(11,22),_58659,colbusbar).
wire(p,pt(1,24),pt(11,24),_58860,colorbus).
wire(p,pt(1,26),pt(11,26),_59061,attacked).
wire(p,pt(1,29),pt(11,29),_59262,p3bar).
wire(m1,pt(3,0),pt(3,5),_59478,p3).
wire(m1,pt(3,5),pt(3,17),_59797,p3).
wire(m1,pt(3,28),pt(3,30),_60120,whiteact).
wire(m1,pt(3,30),pt(3,31),_60443,_36238).
wire(m2,pt(10,28),pt(3,28),_60762,whiteact).
wire(m2,pt(1,30),pt(3,30),_61085,_36238).
wire(m1,pt(4,0),pt(4,8),_61408,valid).
wire(m1,pt(4,8),pt(4,20),_61727,valid).
wire(m1,pt(5,0),pt(5,2),_62050,colorbus).
wire(m1,pt(5,2),pt(5,10),_62365,colorbus).
wire(m1,pt(5,10),pt(5,24),_62684,colorbus).
wire(m1,pt(6,0),pt(6,14),_63007,attacked).
wire(m1,pt(6,14),pt(6,26),_63326,attacked).
wire(m1,pt(7,1),pt(7,12),_63788,colbusbar).
wire(m1,pt(7,12),pt(7,22),_64107,colbusbar).
wire(m2,pt(10,1),pt(7,1),_64430,colbusbar).
wire(m2,pt(2,1),pt(7,1),_64745,colbusbar).
wire(m1,pt(8,4),pt(8,29),_65203,p3bar).
wire(m2,pt(10,4),pt(8,4),_65526,p3bar).
wire(m2,pt(2,4),pt(8,4),_65849,p3bar).
wire(m1,pt(9,16),pt(9,18),_66172,blackact).
wire(m1,pt(9,18),pt(9,31),_66495,_28140).
wire(m2,pt(10,16),pt(9,16),_66814,blackact).
wire(m2,pt(1,18),pt(9,18),_67137,_28140).
wire(pd,pt(1,1),pt(0,1),_67460,vdd).
wire(nd,pt(11,1),pt(12,1),_67775,gnd).
wire(m1,pt(2,1),pt(2,3),_68090,colbusbar).
wire(pd,pt(1,3),pt(2,3),_68405,colbusbar).
wire(m1,pt(10,1),pt(10,3),_68720,colbusbar).
wire(nd,pt(11,3),pt(10,3),_69035,colbusbar).
wire(pd,pt(1,4),pt(0,4),_69350,vdd).
wire(nd,pt(11,4),pt(12,4),_69673,gnd).
wire(m1,pt(2,4),pt(2,6),_69996,p3bar).
wire(pd,pt(1,6),pt(2,6),_70319,p3bar).
wire(m1,pt(10,4),pt(10,6),_70642,p3bar).
wire(nd,pt(11,6),pt(10,6),_70965,p3bar).
wire(pd,pt(1,11),pt(0,11),_71288,vdd).
wire(nd,pt(11,13),pt(12,13),_71611,gnd).
wire(nd,pt(11,16),pt(12,16),_71934,gnd).
wire(m1,pt(1,7),pt(1,15),_72543,g1).
wire(m1,pt(1,15),pt(1,16),_72870,g1).
wire(m1,pt(11,7),pt(11,11),_73197,g4).
wire(m1,pt(11,11),pt(11,15),_73524,g4).
wire(m1,pt(10,9),pt(10,18),_73851,blackact).
wire(nd,pt(11,9),pt(10,9),_74174,blackact).
wire(m1,pt(11,18),pt(10,18),_74497,blackact).
wire(pd,pt(1,23),pt(0,23),_74822,vdd).
wire(nd,pt(11,25),pt(12,25),_75145,gnd).
wire(nd,pt(11,28),pt(12,28),_75468,gnd).
wire(m1,pt(1,19),pt(1,27),_76077,g5).
wire(m1,pt(1,27),pt(1,28),_76404,g5).
wire(m1,pt(11,19),pt(11,23),_76731,g8).
wire(m1,pt(11,23),pt(11,27),_77058,g8).
wire(m1,pt(10,21),pt(10,30),_77385,whiteact).
wire(nd,pt(11,21),pt(10,21),_77708,whiteact).
wire(m1,pt(11,30),pt(10,30),_78031,whiteact).
wire(m1,pt(1,0),pt(1,32),_56809,vdd).
wire(m1,pt(12,0),pt(12,32),_56810,gnd).
```

```
node(3,0,p3,m1).
node(3,5,p3,m1p).
node(3,5,p3,m1p).
node(3,17,p3,m1p).
node(3,28,whiteact,m1m2).
node(3,30,_36238,m1m2).
node(3,30,_36238,m1m2).
node(3,31,whiteact,m1).
node(10,28,whiteact,m1m2).
node(3,28,whiteact,m1m2).
node(1,30,_36238,m1m2).
node(3,30,_36238,m1m2).
node(4,0,valid,m1).
node(4,8,valid,m1p).
node(4,8,valid,m1p).
node(4,20,valid,m1p).
node(5,0,colorbus,m1).
node(5,2,colorbus,m1p).
node(5,2,colorbus,m1p).
node(5,10,colorbus,m1p).
node(5,10,colorbus,m1p).
node(5,24,colorbus,m1p).
node(6,0,attacked,m1).
node(6,14,attacked,m1p).
node(6,14,attacked,m1p).
node(6,26,attacked,m1p).
node(7,1,colbusbar,m1m2).
node(7,12,colbusbar,m1p).
node(7,12,colbusbar,m1p).
node(7,22,colbusbar,m1p).
node(10,1,colbusbar,m1m2).
node(7,1,colbusbar,m1m2).
node(2,1,colbusbar,m1m2).
node(7,1,colbusbar,m1m2).
node(8,4,p3bar,m1m2).
node(8,29,p3bar,m1p).
node(10,4,p3bar,m1m2).
node(8,4,p3bar,m1m2).
node(2,4,p3bar,m1m2).
node(8,4,p3bar,m1m2).
node(9,16,blackact,m1m2).
node(9,18,_28140,m1m2).
node(9,18,_28140,m1m2).
node(9,31,blackact,m1).
node(10,16,blackact,m1m2).
node(9,16,blackact,m1m2).
node(1,18,_28140,m1m2).
node(9,18,_28140,m1m2).
node(1,1,vdd,pd).
node(0,1,vdd,m1pd).
node(11,1,gnd,nd).
node(12,1,gnd,m1nd).
node(2,1,colbusbar,m1m2).
node(2,3,colbusbar,m1pd).
node(1,3,colbusbar,pd).
node(2,3,colbusbar,m1pd).
node(10,1,colbusbar,m1m2).
node(10,3,colbusbar,m1nd).
node(11,3,colbusbar,nd).
node(10,3,colbusbar,m1nd).
node(1,4,vdd,pd).
node(0,4,vdd,m1pd).
node(11,4,gnd,nd).
node(12,4,gnd,m1nd).
node(2,4,p3bar,m1m2).
node(2,6,p3bar,m1pd).
node(1,6,p3bar,pd).
node(2,6,p3bar,m1pd).
node(10,4,p3bar,m1m2).
node(10,6,p3bar,m1nd).
node(11,6,p3bar,nd).
node(10,6,p3bar,m1nd).
node(1,11,vdd,pd).
node(0,11,vdd,m1pd).
node(11,13,gnd,nd).
node(12,13,gnd,m1nd).
node(11,16,gnd,nd).
node(12,16,gnd,m1nd).
node(1,7,g1,m1pd).
node(1,15,g1,m1pd).
node(1,15,g1,m1pd).
node(1,16,g1,m1pd).
node(11,7,g4,m1nd).
node(11,11,g4,m1nd).
node(11,11,g4,m1nd).
node(11,15,g4,m1nd).
node(10,9,blackact,m1nd).
node(10,18,blackact,m1).
node(11,9,blackact,nd).
node(10,9,blackact,m1nd).
node(11,18,blackact,m1nd).
node(10,18,blackact,m1).
node(1,23,vdd,pd).
node(0,23,vdd,m1pd).
node(11,25,gnd,nd).
node(12,25,gnd,m1nd).
node(11,28,gnd,nd).
node(12,28,gnd,m1nd).
node(1,19,g5,m1pd).
node(1,27,g5,m1pd).
node(1,27,g5,m1pd).
node(1,28,g5,m1pd).
node(11,19,g8,m1nd).
node(11,23,g8,m1nd).
node(11,23,g8,m1nd).
node(11,27,g8,m1nd).
node(10,21,whiteact,m1nd).
node(10,30,whiteact,m1).
node(11,21,whiteact,nd).
node(10,21,whiteact,m1nd).
node(11,30,whiteact,m1nd).
node(10,30,whiteact,m1).
node(1,0,vdd).
node(1,32,vdd).
node(12,0,gnd).
node(12,32,gnd).
trans(pd,pt(1,1),pt(1,2),pt(1,3),1,1,vdd,colorbus,colbusbar).
trans(nd,pt(11,1),pt(11,2),pt(11,3),1,1,gnd,colorbus,colbusbar).
trans(pd,pt(1,4),pt(1,5),pt(1,6),1,1,vdd,p3,p3bar).
trans(nd,pt(11,4),pt(11,5),pt(11,6),1,1,gnd,p3,p3bar).
trans(pd,pt(1,7),pt(1,8),pt(1,9),1,1,g1,valid,g3).
trans(pd,pt(1,9),pt(1,10),pt(1,11),1,1,g3,colorbus,vdd).
trans(pd,pt(1,11),pt(1,12),pt(1,13),1,1,vdd,colbusbar,g2).
trans(pd,pt(1,13),pt(1,14),pt(1,15),1,1,g2,attacked,g1).
trans(pd,pt(1,16),pt(1,17),pt(1,18),1,1,g1,p3,blackact).
trans(nd,pt(11,7),pt(11,8),pt(11,9),1,1,g4,valid,blackact).
trans(nd,pt(11,9),pt(11,10),pt(11,11),1,1,blackact,colorbus,g4).
trans(nd,pt(11,11),pt(11,12),pt(11,13),1,1,g4,colbusbar,gnd).
trans(nd,pt(11,13),pt(11,14),pt(11,15),1,1,gnd,attacked,g4).
trans(nd,pt(11,16),pt(11,17),pt(11,18),1,1,gnd,p3,blackact).
trans(pd,pt(1,19),pt(1,20),pt(1,21),1,1,g5,valid,g7).
trans(pd,pt(1,21),pt(1,22),pt(1,23),1,1,g7,colbusbar,vdd).
trans(pd,pt(1,23),pt(1,24),pt(1,25),1,1,vdd,colorbus,g6).
trans(pd,pt(1,25),pt(1,26),pt(1,27),1,1,g6,attacked,g5).
trans(pd,pt(1,28),pt(1,29),pt(1,30),1,1,g5,p3bar,whiteact).
trans(nd,pt(11,19),pt(11,20),pt(11,21),1,1,g8,valid,whiteact).
trans(nd,pt(11,21),pt(11,22),pt(11,23),1,1,whiteact,colbusbar,g8).
trans(nd,pt(11,23),pt(11,24),pt(11,25),1,1,g8,colorbus,gnd).
trans(nd,pt(11,25),pt(11,26),pt(11,27),1,1,gnd,attacked,g8).
trans(nd,pt(11,28),pt(11,29),pt(11,30),1,1,gnd,p3bar,whiteact).
```

```
cont(m1pd,pt(2,3),no,colbusbar).
cont(m1m2,pt(2,1),no,colbusbar).
cont(m1nd,pt(10,3),no,colbusbar).
cont(m1m2,pt(10,1),no,colbusbar).
cont(m1nd,pt(12,1),no,gnd).
cont(m1pd,pt(0,1),no,vdd).
cont(m1pd,pt(2,6),no,p3bar).
cont(m1m2,pt(2,4),no,p3bar).
cont(m1nd,pt(10,6),no,p3bar).
cont(m1m2,pt(10,4),no,p3bar).
cont(m1nd,pt(12,4),no,gnd).
cont(m1pd,pt(0,4),no,vdd).
cont(m1pd,pt(1,7),no,g1).
cont(m1pd,pt(1,15),no,g1).
cont(m1pd,pt(1,16),no,g1).
cont(m1nd,pt(11,7),no,g4).
cont(m1nd,pt(11,11),no,g4).
cont(m1nd,pt(11,15),no,g4).
cont(m1nd,pt(11,18),no,blackact).
cont(m1nd,pt(10,9),no,blackact).
cont(m1m2,pt(10,16),no,blackact).
cont(m1nd,pt(12,13),no,gnd).
cont(m1nd,pt(12,16),no,gnd).
cont(m1pd,pt(0,11),no,vdd).
cont(m1pd,pt(1,19),no,g5).
cont(m1pd,pt(1,27),no,g5).
cont(m1pd,pt(1,28),no,g5).
cont(m1nd,pt(11,19),no,g8).
cont(m1nd,pt(11,23),no,g8).
cont(m1nd,pt(11,27),no,g8).
cont(m1nd,pt(11,30),no,whiteact).
cont(m1nd,pt(10,21),no,whiteact).
cont(m1m2,pt(10,28),no,whiteact).
cont(m1nd,pt(12,25),no,gnd).
cont(m1nd,pt(12,28),no,gnd).
cont(m1pd,pt(0,23),no,vdd).
cont(m1p,pt(5,2),no,colorbus).
cont(m1p,pt(3,5),no,p3).
cont(m1p,pt(4,8),no,valid).
cont(m1p,pt(5,10),no,colorbus).
cont(m1p,pt(7,12),no,colbusbar).
cont(m1p,pt(6,14),no,attacked).
cont(m1p,pt(3,17),no,p3).
cont(m1p,pt(4,20),no,valid).
cont(m1p,pt(7,22),no,colbusbar).
cont(m1p,pt(5,24),no,colorbus).
cont(m1p,pt(6,26),no,attacked).
cont(m1p,pt(8,29),no,p3bar).
cont(m1m2,pt(3,28),no,whiteact).
cont(m1m2,pt(3,30),no,_36238).
cont(m1m2,pt(7,1),no,colbusbar).
cont(m1m2,pt(7,1),no,colbusbar).
cont(m1m2,pt(8,4),no,p3bar).
cont(m1m2,pt(8,4),no,p3bar).
cont(m1m2,pt(9,16),no,blackact).
cont(m1m2,pt(9,18),no,_28140).
maxrow(12).
maxcol(32).
```

```
# /*
  sml.m: benchmark (viper) sml master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (viper) sml
%
%   The ASP Group
%
%   (contact: Bill Bush
%             Computer Science Division
%             University of California
%             Berkeley, CA 94720
%             bush@ophiuchus.Berkeley.EDU)
%
%   run viper on simple microprocessor specification (sml)

#if BENCH
#  include ".sml.bench"
#else
sml :- reconsult('examples/in/sml'),
       viper('examples/out/sml').
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#if QUINTUS_PL
:- multifile execute/1, fetch/0, run/0.

:- dynamic execute/1, fetch/0, run/0.

#endif
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (viper/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
viper(_).
#else
#  include "viper"      /* code for viper */
#endif
```

```
# /*
  viper: code for ASP viper component
  */
%   (c) 1988 Regents of the University of California
%
%   Viper is the high-level synthesis component of the ASP (Advanced
%   Silicon compiler in Prolog) system developed at the University of
%   California, Berkeley.  Viper generates structural hardware de-
%   scriptions from instruction-set level specifications written in
%   standard Prolog.  It translates Prolog constructs into hardware
%   equivalents and creates and allocates hardware resources while
%   satifying various constraints.
%
%   Viper operates in four phases: register allocation, translation
%   of the Prolog specification into an RTL-based form, data path
%   construction, and structural description generation.
%
%   For a detailed explanation of viper, see W. Bush et al., "A
%   Prototype Silicon Compiler in Prolog," University of California
%   (Technical Report UCB/CSD 88/476), Berkeley, California, 1988.
%
%   Five output files are generated from a specification file name:
%
%       namebus
%       namegoto
%       namertl
%       namesched
%       nameunit
%
%   When show/0 is provable, viper produces output (intended for
%   a terminal screen) indicating its progress.


#option "
        > For use with Quintus Prolog, viper requires some
        > Quintus Prolog-specific directives.  These are
        > generated if option QUINTUS_PL is selected."
#if QUINTUS_PL
:- no_style_check(single_var).

:- unknown(_, fail).

:- op(400, fx, \).


#endif
```

```
%** Viper top level

viper(Name) :-
        scan(transfer, run, 0),
        makeFileName(Name, rtl, N1),
        scanWrite(N1),
        (show -> nl, write('Register Transfers...'), nl, nl ; true),
        sched,
        makeFileName(Name, sched, N2),
        schedWrite(N2),
        (show -> nl, write('Schedule...'), nl, nl ; true),
        branch,
        makeFileName(Name, goto, N3),
        branchWrite(N3),
        (show -> nl, write('Branches...'), nl, nl ; true),
        alloc,
        makeFileName(Name, unit, N4),
        allocWrite(N4),
        (show -> nl, write('Functional Units...'), nl, nl ; true),
        conn,
        makeFileName(Name, bus, N5),
        connWrite(N5),
        (show -> nl, write('Buses...'), nl, nl ; true),
        !.

makeFileName(Root, Number, Symbol) :-
        name(Root, RL),
        name(Number, NL),
        makeFileString(RL, NL, SL),
        name(Symbol, SL).

makeFileString([], X, X) :- !.
makeFileString([A|B], C, [A|D]) :- makeFileString(B, C, D).

%*
%*  utility
%*

flush(Functor, Arity) :-
        abolish(Functor, Arity).


%** Scan Prolog, instantiating variables
%**     (declare all scanned procedures dynamic)

%** data base items
%*      scanIndex(<root>,<index>)
%*          v(c(p(<name>,<arity>),<clause-index>),<variable-index>)
%*      scanPass(<pass-name>)
%*      scanError(<pass-name>,<type-of-error>)
%*    transfer generation pass
%*      transferSrc(<register>,<variable>)
%*      transferExp(<src1>,<src2>,<op>,<dst>)
%*      transferDst(<register>,<variable>)
%*      transfer(<id>,<block>,<src1>,<src2>,<op>,<dstreg>)
%*          <register> ::= <register-name> | constant(<atom>)
%*              | field(<register-name>,<field-name>)
%*    label and jump generation
%*      label(<clause-name>,<tag>,<block>)
%*      jump(<block>,<type>,<clause-name>)
%*          <type> ::= case | cond | jrst

%*  main routine, invoked with prime clause functor and arity
scan(Pass, ProcFunctor, ProcArity) :-
        (show -> write('>>> '), write(Pass), nl, nl ; true),
        flush(scanPass, 1),
        assert(( scanPass(Pass) )), !,
        scanInitialize,
        scanClauses(p(ProcFunctor, ProcArity), ProcFunctor, ProcArity).
scanClauses(ProcName, ProcFunctor, ProcArity) :-
        functor(ClauseHead, ProcFunctor, ProcArity),
        clause(ClauseHead, ClauseBody),
        scanNewName(c, ProcName, ClauseName),
        scanClause(ClauseName, ClauseHead, ClauseBody),
        fail.
scanClauses(_, _, _).

scanClause(ClauseName, ClauseHead, ClauseBody) :-
        % case arm (indicated by arity 1)
        scanPass(transfer),
        functor(ClauseHead, _, 1), !,
        scanArgs(ClauseName, ClauseHead, 1),
        scanNewBlock(ClauseName, ClauseHead),
        scanGoal(ClauseName, ClauseBody),
        % add jump to end of case arm
        scanOldName(end, EndLabel),
        scanJump(jrst, EndLabel),
        (show -> write(ClauseHead), nl ; true), !.
scanClause(ClauseName, ClauseHead, ClauseBody) :-
        scanPass(transfer),
        functor(ClauseHead, _, 0), !,
        scanArgs(ClauseName, ClauseHead, 1),
        scanOldBlock(ClauseName),
        scanGoal(ClauseName, ClauseBody),
        (show -> write(ClauseHead), nl ; true), !.
scanClause(ClauseName, ClauseHead, ClauseBody) :-
        scanArgs(ClauseName, ClauseHead, 1),
        scanGoal(ClauseName, ClauseBody),
        (show -> write(ClauseHead), nl, tab(4), write(ClauseBody), nl
        ;
         true
        ), !.

scanArgs(ClauseName, ClauseHead, ArgIndex) :-
        arg(ArgIndex, ClauseHead, ClauseHeadArg),
        scanArg(ClauseName, ClauseHeadArg),
        NewIndex is ArgIndex + 1,
        scanArgs(ClauseName, ClauseHead, NewIndex).
scanArgs(_, _, _).

scanArg(_, Arg) :-
        atomic(Arg), !.
scanArg(ClauseName, Arg) :-
        var(Arg), !,
        scanVariable(ClauseName, Arg).
scanArg(ClauseName, v(ClauseName, _)) :-
        !.
scanArg(ClauseName, [L]) :-
        (show -> write('... List argument '), write(L),
                 write(' in '), write(ClauseName), nl
        ;
         true
        ),
        scanPass(Pass), assert(( scanError(Pass, list) )), !.
scanArg(ClauseName, S) :-
        (show -> write('... Structure argument '), write(S),
                 write(' in '), write(ClauseName), nl
        ;
         true
        ),
        scanPass(Pass), assert(( scanError(Pass, structure) )), !.

% and (,) terms
scanGoal(ClauseName, (Goal, Goals)) :-
        scanGoal(ClauseName, Goal),
        scanGoal(ClauseName, Goals), !.
```

```
% or (;) terms
scanGoal(ClauseName, (Goal; Goals)) :-
        (show -> write('... or '), write(Goal), nl ; true),
        scanPass(Pass), assert(( scanError(Pass, or) )),
        scanGoal(ClauseName, Goal),
        scanGoal(ClauseName, Goals), !.
% if-then (->)
scanGoal(ClauseName, (Goal -> Goals)) :-
        (show -> write('... if '), write(Goal), nl ; true),
        scanPass(Pass), assert(( scanError(Pass, if) )),
        scanGoal(ClauseName, Goal),
        scanGoal(ClauseName, Goals), !.
% not
scanGoal(ClauseName, not(InnerGoal)) :-
        (show -> write('... not '), write(InnerGoal), nl ; true),
        scanPass(Pass), assert(( scanError(Pass, not) )),
        scanGoal(ClauseName, InnerGoal), !.
% is
scanGoal(ClauseName, (LeftSide is RightSide)) :-
        scanPass(transfer), !,
        scanVariable(ClauseName, LeftSide),
        scanNumerics(ClauseName, RightSide, LeftSide).
scanGoal(ClauseName, (LeftSide is RightSide)) :-
        scanVariable(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% comparison (=:=)
scanGoal(ClauseName, (LeftSide =:= RightSide)) :-
        scanComparison(ClauseName, (LeftSide =:= RightSide)), !.
% comparison (>)
scanGoal(ClauseName, (LeftSide > RightSide)) :-
        scanComparison(ClauseName, (LeftSide > RightSide)), !.
% comparison (<)
scanGoal(ClauseName, (LeftSide < RightSide)) :-
        scanComparison(ClauseName, (LeftSide < RightSide)), !.
% comparison (=<)
scanGoal(ClauseName, (LeftSide =< RightSide)) :-
        scanComparison(ClauseName, (LeftSide =< RightSide)), !.
% comparison (>=)
scanGoal(ClauseName, (LeftSide >= RightSide)) :-
        scanComparison(ClauseName, (LeftSide >= RightSide)), !.
% cut
scanGoal(ClauseName, !) :- !.
% null goal
scanGoal(ClauseName, true) :- !.
% fail
scanGoal(ClauseName, fail) :- !.
% assert
scanGoal(ClauseName, assert(InnerGoal)) :-
        (show -> write('... assert '), write(InnerGoal), nl ; true),
        scanPass(Pass), assert(( scanError(Pass, assert) )),
        scanGoal(ClauseName, InnerGoal), !.
% retract
scanGoal(ClauseName, retract(InnerGoal)) :-
        (show -> write('... retract '), write(InnerGoal), nl ; true),
        scanPass(Pass), assert(( scanError(Pass, retract) )),
        scanGoal(ClauseName, InnerGoal), !.
% debugging goals
scanGoal(ClauseName, write(_)) :-
        !.
scanGoal(ClauseName, tab(_)) :-
        !.
scanGoal(ClauseName, nl) :-
        !.
% general Viper-specific goals
scanGoal(_, mem_read) :-
        scanPass(transfer), !,
        scanNewName(rt, ID),
        scanOldName(block, Block),
        assert(( transfer(ID, Block, memAR, none, mem_read, memDR) )).
```

```
scanGoal(ClauseName, mem_read) :-
        !.
scanGoal(_, mem_write) :-
        scanPass(transfer), !,
        scanNewName(rt, ID),
        scanOldName(block, Block),
        assert(( transfer(ID, Block, memAR, memDR, mem_write, none) )).
scanGoal(ClauseName, mem_write) :-
        !.
scanGoal(ClauseName, stateDefine) :-
        !.
scanGoal(ClauseName, stateInitialize) :-
        !.
scanGoal(_, stateUpdate) :-
        scanPass(transfer), !,
        scanOldName(rt, ID),
        scanOldName(block, Block),
        assert(( scanUpdatePost(ID, Block) )).
scanGoal(ClauseName, stateUpdate) :-
        !.
scanGoal(ClauseName, stateList) :-
        !.
scanGoal(ClauseName, statePrint) :-
        !.
scanGoal(ClauseName, stateCount(_)) :-
        !.
% access
scanGoal(ClauseName, access(Register, Variable)) :-
        scanPass(transfer), !,
        scanVariable(ClauseName, Variable),
        assert(( transferSrc(Register, Variable) )).
scanGoal(ClauseName, access(Register, Variable)) :-
        scanVariable(ClauseName, Variable), !.
scanGoal(ClauseName, access(Register, Field, Variable)) :-
        scanPass(transfer), !,
        scanVariable(ClauseName, Variable),
        scanGoal(ClauseName, access(field(Register, Field), Variable)).
scanGoal(ClauseName, access(Register, Field, Variable)) :-
        scanVariable(ClauseName, Variable), !.
% set
scanGoal(ClauseName, set(Register, Variable)) :-
        scanPass(transfer), !,
        scanVariable(ClauseName, Variable),
        scanSet(ClauseName, Register, Variable).
scanGoal(ClauseName, set(Register, Variable)) :-
        scanVariable(ClauseName, Variable), !.
scanGoal(ClauseName, set(Register, Field, Variable)) :-
        scanPass(transfer), !,
        scanVariable(ClauseName, Variable),
        scanSet(ClauseName, field(Register, Field), Variable).
scanGoal(ClauseName, set(Register, Field, Variable)) :-
        scanVariable(ClauseName, Variable), !.
% general goal
scanGoal(ClauseName, Goal) :-
        % case (indicated by goal arity 1)
        scanPass(transfer),
        functor(Goal, ProcName, 1), !,
        % add case dispatch
        scanJump(case, p(ProcName, 1)),
        scanNewName(end, EndLabel),
        scanActuals(ClauseName, Goal, 1),
        scanCall(Goal),
        % add label at end of case
        scanNewName(block, Block),
        assert(( label(EndLabel, none, Block) )).
```

```
scanGoal(ClauseName, Goal) :-
        % tail recursion
        scanPass(transfer),
        scanRecursion(ClauseName, Goal), !,
        scanArgs(ClauseName, Goal, 1),
        functor(Goal, GoalFunctor, GoalArity),
        scanProcedure(p(GoalFunctor, GoalArity), GoalFunctor, GoalArity),
        scanJump(jrst, c(p(GoalFunctor, GoalArity), 1)).
scanGoal(ClauseName, Goal) :-
        scanArgs(ClauseName, Goal, 1),
        scanCall(Goal), !.

scanComparison(ClauseName, Expression) :-
        scanPass(transfer), !,
        scanNewName(control, Name),
        scanNumerics(ClauseName, Expression, Name),
        transferExp(SrcVar1, SrcVar2, Op, Name),
        scanTransfer(SrcVar1, SrcVar2, Op, Name, control),
        ClauseName = c(ProcName, ThisClause),
        NextClause is ThisClause + 1,
        scanJump(cond, c(ProcName, NextClause)),
        scanNewBlock(Name, none).
scanComparison(ClauseName, Expression) :-
        arg(1, Expression, LeftSide),
        arg(2, Expression, RightSide),
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.

% atomic
scanNumeric(_, Object) :-
        atomic(Object), !.
% variable
scanNumeric(ClauseName, Object) :-
        var(Object), !,
        scanVariable(ClauseName, Object), !.
% touched variable
scanNumeric(ClauseName, v(ClauseName, _)) :-
        !.
% addition (+)
scanNumeric(ClauseName, (LeftSide + RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% subtraction (-)
scanNumeric(ClauseName, (LeftSide - RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% unary minus (-)
scanNumeric(ClauseName, (- InnerGoal)) :-
        scanNumeric(ClauseName, InnerGoal), !.
% multiplication (*)
scanNumeric(ClauseName, (LeftSide * RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% division (/)
scanNumeric(ClauseName, (LeftSide / RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% and (/\)
scanNumeric(ClauseName, (LeftSide /\ RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% or (\/)
scanNumeric(ClauseName, (LeftSide \/ RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% left shift (<<)
scanNumeric(ClauseName, (LeftSide << RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
```

```
% right shift (>>)
scanNumeric(ClauseName, (LeftSide >> RightSide)) :-
        scanNumeric(ClauseName, LeftSide),
        scanNumeric(ClauseName, RightSide), !.
% complement (\)
scanNumeric(ClauseName, (\ InnerGoal)) :-
        scanNumeric(ClauseName, InnerGoal), !.
% default
scanNumeric(ClauseName, Object) :-
        (show -> write('... unknown numeric in '), write(ClauseName), nl
        ;
         true
        ), !,
        scanPass(Pass), assert(( scanError(Pass, numeric) )).

scanVariable(ClauseName, Object) :-
        var(Object), !,
        scanNewName(v, ClauseName, Object).
scanVariable(ClauseName, Object).

scanCall(ClauseGoal) :-
        functor(ClauseGoal, ProcFunctor, ProcArity),
        scanProcedure(p(ProcFunctor, ProcArity), ProcFunctor, ProcArity).

% 1) already processed
scanProcedure(ProcName, _, _) :-
        scanPass(Pass),
        scanIndex(ProcName, _), !.
% 2) unit-ground clause
scanProcedure(ProcName, ProcFunctor, ProcArity) :-
        scanUnit(ProcFunctor, ProcArity), !,
        scanPass(Pass),
        assert(( scanIndex(ProcName, 0) )).
% 3) recurse
scanProcedure(ProcName, ProcFunctor, ProcArity) :-
        functor(ClauseHead, ProcFunctor, ProcArity),
        clause(ClauseHead, _), !,
        scanClauses(ProcName, ProcFunctor, ProcArity).
% 4) unknown
scanProcedure(ProcName, _, _) :-
        (show -> write('... unknown procedure '), write(ProcName), nl
        ;
         true
        ), !,
        scanPass(Pass), assert(( scanError(Pass, procedure) )).

scanUnit(ProcFunctor, ProcArity) :-
        functor(ClauseHead, ProcFunctor, ProcArity),
        clause(ClauseHead, true), !,
        scanGround(ClauseHead, 1, ProcArity).

scanGround(_, ArgIndex, ClauseArity) :-
        ArgIndex > ClauseArity, !.
scanGround(ClauseHead, ArgIndex, ClauseArity) :-
        arg(ArgIndex, ClauseHead, Arg), !,
        atomic(Arg),
        NewIndex is ArgIndex + 1,
        scanGround(ClauseHead, NewIndex, ClauseArity).

%*
%*  transfer-specifc procedures
%*

scanActuals(ClauseName, ClauseHead, ArgIndex) :-
        arg(ArgIndex, ClauseHead, ClauseHeadArg),
        scanActual(ClauseName, ClauseHeadArg),
        NewIndex is ArgIndex + 1,
        scanActuals(ClauseName, ClauseHead, NewIndex).
scanActuals(_, _, _).
```

```
scanActual(_, Variable) :-
        transferSrc(SrcReg, Variable),
        \+ (transferDst(_, Variable)), !,
        scanTransfer(Variable, none, case, Variable, control).
scanActual(ClauseName, Arg) :-
        scanVariable(ClauseName, Arg), !.

% simple transfer -- register <- constant via "is"
scanNumerics(ClauseName, Object, Result) :-
        functor(Object, Constant, 0),
        nonvar(Result), !,
        scanTransfer(Object, none, move, Object, Result).
% simple object -- result is self
scanNumerics(ClauseName, Object, Object) :-
        functor(Object, Constant, 0), !.
% unary operator
scanNumerics(ClauseName, Expression, Result) :-
        functor(Expression, Op, 1), !,
        arg(1, Expression, SubExpression),
        scanExp(SubExpression, none, Op, Result).
% binary operator
scanNumerics(ClauseName, Expression, Result) :-
        functor(Expression, Op, 2), !,
        arg(1, Expression, LeftSide),
        arg(2, Expression, RightSide),
        scanExp(LeftSide, RightSide, Op, Result).

% simple expression, destination known
scanExp(LeftSide, RightSide, Op, Destination) :-
        nonvar(Destination), !,
        assert(( transferExp(LeftSide, RightSide, Op, Destination) )).
% expression using temporaries
scanExp(LeftSide, RightSide, Op, Destination) :-
        scanNewName(temp, Destination),
        assert(( transferSrc(Destination, Destination) )), !,
        scanTransfer(LeftSide, RightSide, Op, Op, Destination).

%  1) an expression: exp -> reg
scanSet(ClauseName, DstReg, DstVar) :-
        transferExp(SrcVar1, SrcVar2, Op, DstVar), !,
        scanTransfer(SrcVar1, SrcVar2, Op, DstVar, DstReg).
%  2) a simple transfer: reg -> reg or constant -> reg
scanSet(ClauseName, DstReg, Variable) :-
        transferSrc(SrcReg, Variable), !,
        scanTransfer(Variable, none, move, Variable, DstReg).

scanTransfer(SrcVar1, SrcVar2, Op, DstVar, DstReg) :-
        scanNewName(rt, ID),
        scanOldName(block, Block),
        scanTransferSrc(SrcVar1, SrcReg1),
        scanTransferSrc(SrcVar2, SrcReg2),
        assert(( transferDst(DstReg, DstVar) )),
        assert(( transfer(ID, Block, SrcReg1, SrcReg2, Op, DstReg) )), !.

scanTransferSrc(Variable, Register) :-
        transferSrc(Register, Variable), !.
scanTransferSrc(none, none).
scanTransferSrc(Constant, constant(Constant)).

scanNewBlock(Name, ClauseHead) :-
        scanTag(ClauseHead, Tag),
        Name = c(ProcName, _),
        label(c(ProcName, _), Tag, _), !,
        scanNewName(block, Block),
        assert(( label(Name, none, Block) )).
scanNewBlock(Name, ClauseHead) :-
        scanTag(ClauseHead, Tag),
        scanNewName(block, Block),
        assert(( label(Name, Tag, Block) )), !.
```

```
scanOldBlock(Name) :-
        \+ (scanIndex(block, _)), !,
        assert(( scanIndex(block, 1) )),
        assert(( label(Name, none, block(1)) )).
scanOldBlock(_).

scanJump(Type, Label) :-
        scanOldName(block, Block),
        assert(( jump(Block, Type, Label) )), !.

scanRecursion(c(p(ProcName, ProcArity), _), Goal) :-
        functor(Goal, ProcName, ProcArity).

scanTag(ClauseHead, Tag) :-
        arg(1, ClauseHead, Tag), !.
scanTag(_, none).

%*
%*  utilities
%*

% gensym with general root (not simply symbol) and functor
scanNewName(Functor, Root, Name) :-
        scanIndex(Root, OldIndex), !,
        retract(( scanIndex(Root, OldIndex) )),
        NewIndex is OldIndex + 1,
        assert(( scanIndex(Root, NewIndex) )),
        Name =.. [Functor, Root, NewIndex].
scanNewName(Functor, Root, Name) :-
        assert(( scanIndex(Root, 1) )),
        Name =.. [Functor, Root, 1], !.

scanOldName(Functor, Root, Name) :-
        scanIndex(Functor, Root, Index),
        Name =.. [Functor, Root, Index], !.

% gensym with general root (not simply symbol)
scanNewName(Functor, Name) :-
        scanIndex(Functor, OldIndex), !,
        retract(( scanIndex(Functor, OldIndex) )),
        NewIndex is OldIndex + 1,
        assert(( scanIndex(Functor, NewIndex) )),
        Name =.. [Functor, NewIndex].
scanNewName(Functor, Name) :-
        assert(( scanIndex(Functor, 1) )),
        Name =.. [Functor, 1], !.

scanOldName(Functor, Name) :-
        scanIndex(Functor, Index), !,
        Name =.. [Functor, Index].
scanOldName(Functor, Name) :-
        (show -> write('... Undefined name '), write(Functor), nl ; true), !,
        scanPass(Pass), assert(( scanError(Pass, name) )),
        assert(( scanIndex(Functor, 1) )),
        Name =.. [Functor, 1], !.

scanInitialize :-
        flush(scanIndex, 2),
        flush(transferSrc, 2),
        flush(transferExp, 4),
        flush(transferDst, 2),
        flush(transfer, 6),
        flush(label, 3),
        flush(jump, 3),
        flush(scanUpdatePost, 2),
        flush(scanError, 2).
```

```
scanList :-
        listing(scanIndex),
        listing(transferSrc),
        listing(transferExp),
        listing(transferDst),
        listing(transfer),
        listing(label),
        listing(jump),
        listing(scanUpdatePost),
        listing(scanError).

scanWrite(File) :-
        tell(File),
        scanList,
        close(File).



%** RTL Scheduler
%**     Schedule abstract transfers and produce dependency information

%** data base items
%*      (input: label, transfer)
%*      cycle(<rtl-ID>,<block>,<cycle>)
%*      schedDep(<resource>,<successor-ID>,<predecessor-ID>)
%*      lastUse(<resource>,<last-user-ID>)

%*  main routine
sched :-
        schedInitialize,
        schedBlocks.

%*  process all blocks (each has one label)
schedBlocks :-
        label(_, _, Block),
        (show -> write(Block) ; true),
        schedBlock(Block),
        fail.
schedBlocks.

%*  process all transfers in a block
schedBlock(Block) :-
        transfer(ID, Block, Src1, Src2, OpType, Dst),
        (show -> tab(1), write(ID) ; true),
        schedTransfer(ID, Src1, Src2, OpType, Dst, Block),
        fail.
schedBlock(Block) :-
        flush(lastUse, 2),
        (show -> nl ; true), !.

schedTransfer(ID, Src1, Src2, OpType, Dst, Block) :-
        schedResource(ID, Src1, 0, CycleM1),
        schedResource(ID, Src2, CycleM1, CycleM2),
        schedResource(ID, Dst, CycleM2, MaxOldCycle),
        NewCycle is MaxOldCycle + 1,
        assert(( cycle(ID, Block, NewCycle) )), !.
```

```
%*  resource: none
schedResource(ID, none, Cycle, Cycle).
%*
%*  resource: integer
%*      this assumes it is always available;
%*          it may be in a constant ROM for which there is contention
schedResource(ID, constant(_), Cycle, Cycle).
%*
%*  resource: field
%*      this assumes that two fields cannot be accessed at once
%*          remember fields in lastUse and check for overlap
schedResource(ID, field(Resource, _), InCycle, OutCycle) :-
        schedResource(ID, Resource, InCycle, OutCycle), !.
%*
%*  resource: general
schedResource(ID, Resource, InCycle, OutCycle) :-
        % transfer of last occurrence
        lastUse(Resource, LastTransfer),
        % cycle of last occurrence
        cycle(LastTransfer, _, LastCycle), !,
        schedMax(InCycle, LastCycle, OutCycle),
        retract(( lastUse(Resource, LastTransfer) )),
        assert(( lastUse(Resource, ID) )),
        assert(( schedDep(Resource, ID, LastTransfer) )).
%*
%*  resource: general, first occurrence
schedResource(ID, Resource, Cycle, Cycle) :-
        assert(( lastUse(Resource, ID) )).

%*
%*  utilities
%*

schedInitialize :-
        flush(cycle, 3),
        flush(schedDep, 3),
        flush(lastUse, 2), !.

schedList :-
        listing(cycle),
        listing(schedDep).

schedWrite(File) :-
        tell(File),
        schedList,
        close(File).

schedMax(X, Y, X) :- X > Y.
schedMax(X, Y, Y).



%** Branch generator
%**     Generate state transitions, removing extra cycles

%** data base items
%*      (input: label, jump, transfer, cycle)
%*      goto(<from-block>,<cycle>,<condition>,<to-block>)
%*      unreachable(<block>)

%*  main routine
branch :-
        branchInitialize,
        branchBlocks,
        branchDeadBlocks.
```

```
%*  process all blocks (each has one label)
branchBlocks :-
        label(_, _, Block),
        jump(Block, Type, Target),
        (show -> tab(2), write(Block), tab(1), write(Type), nl ; true),
        branchBlock(Block, Type, Target),
        fail.
branchBlocks.

%*  process an unconditional jump
branchBlock(FromBlock, jrst, Target) :-
        branchTarget(Target, ToBlock),
        branchCycle(FromBlock, 0, Cycle),
        assert(( goto(FromBlock, Cycle, true, ToBlock) )), !.
%*  process a conditional jump
branchBlock(FromBlock, cond, FailTarget) :-
        transfer(_, FromBlock, Src1, Src2, Op, control),
        FromBlock = block(OldIndex),
        NewIndex is OldIndex + 1,
        SuccessBlock = block(NewIndex),
        % (we could check for a null SuccessBlock target)
        branchCycle(FromBlock, 0, Cycle),
        assert(( goto(FromBlock, Cycle, cond(Op, Src1, Src2), SuccessBlock) )),
        branchTarget(FailTarget, FailBlock),
        assert(( goto(FromBlock, Cycle, cond(not(Op, Src1, Src2)),
                        FailBlock) )), !.
%*  process a case
branchBlock(FromBlock, case, Target) :-
        transfer(_, FromBlock, Value, none, case, control),
        % (this assumes only one value in one transfer is used for dispatch)
        branchCases(FromBlock, Value, Target), !.

%*  process all case arm labels
branchCases(FromBlock, Value, ToProc) :-
        label(c(ToProc, _), Tag, ToBlock),
        branchCaseArm(FromBlock, Value, Tag, ToBlock),
        fail.
branchCases(_, _, _).

%*  process each case arm label
branchCaseArm(_, _, none, _) :- !.
        % ignore untagged case arms
branchCaseArm(FromBlock, Value, Tag, ToBlock) :-
        branchCycle(FromBlock, 0, Cycle),
        assert(( goto(FromBlock, Cycle, case(Value, Tag), ToBlock) )), !.

%*  non-null block
branchTarget(Target, ToBlock) :-
        label(Target, none, ToBlock),
        transfer(_, ToBlock, _, _, _, _), !.
%*  null block -- follow jump
branchTarget(Target1, ToBlock) :-
        label(Target1, none, IndirectBlock),
        jump(IndirectBlock, jrst, Target2),
        branchTarget(Target2, ToBlock), !.

%*  find last cycle in a block
branchCycle(Block, PreviousCycle, FinalCycle) :-
        ThisCycle is PreviousCycle + 1,
        cycle(_, Block, ThisCycle), !,
        branchCycle(Block, ThisCycle, FinalCycle).
branchCycle(_, Cycle, Cycle).

%*  check all blocks for unreachable ones
branchDeadBlocks :-
        label(_, _, Block),
        branchDeadBlock(Block),
        fail.
branchDeadBlocks.
```

```
%*  mark an unreachable block
branchDeadBlock(Block) :-
        goto(_, _, _, Block), !.
branchDeadBlock(Block) :-
        (show -> tab(2), write(Block), write(' is unreachable'), nl ; true),
        assert(( unreachable(Block) )), !.

%*
%*  utilities
%*

branchInitialize :-
        flush(goto, 4),
        flush(unreachable, 1), !.

branchList :-
        listing(goto),
        listing(unreachable).

branchWrite(File) :-
        tell(File),
        branchList,
        close(File).



%** Data Path Allocator
%**     Allocate data path elements

%** data base items
%*
%*  library input
%*      libOperator(Op, Fn, Class)
%*      libUnit(Type)
%*      libFunction(Type, Function)
%*  RTL input
%*      transfer, label, cycle
%*
%*  intermediate results
%*      allocCombFn(Class, Fn, Arg).
%*        (functions needed -- unique triples)
%*      allocCombPar(Block, Cycle, Fn, Arg).
%*        (concurrent resource use)
%*
%*  output
%*      unit(Unit, Type)
%*      functionBinding(Unit, Function)
%*      functionUse(ID, Block, Cycle, Fn, Arg).
%*      argRebinding(ID, Src, Dst).

%*  main routine
alloc :-
        allocInitialize,
        allocScanBlocks,
        allocUnits.

%*  process all blocks (each has one label) -- determine needs
allocScanBlocks :-
        label(_, _, Block),
        (show -> write(Block) ; true),
        allocScanBlock(Block),
        fail.
allocScanBlocks.

%*  process all transfers in a block
allocScanBlock(Block) :-
        transfer(ID, Block, Src1, Src2, Op, Dst),
        (show -> tab(1), write(ID) ; true),
        allocScanTransfer(ID, Src1, Src2, Op, Dst),
        fail.
```

```
allocScanBlock(_) :-
        (show -> nl ; true), !.

% transfer: move
allocScanTransfer(ID, Src, none, move, Dst) :- !,
        allocReg(Src),
        allocReg(Dst), !.
% transfer: special case -- increment
allocScanTransfer(ID, Counter, constant(1), '+', Counter) :- !,
        allocReg(Counter),
        allocAssertCombFn(ID, count, inc, Counter), !.
% transfer: special case -- shift one
allocScanTransfer(ID, Src, constant(1), '>>', Dst) :- !,
        allocReg(Src),
        allocReg(Dst),
        allocAssertCombFn(ID, shift, shr1, none), !.
% transfer: special case -- less than zero test
allocScanTransfer(ID, Reg, constant(0), '<', control) :- !,
        allocReg(Reg),
        allocAssertCombFn(ID, control, ltzero, Reg), !.
% transfer: special case -- case test
allocScanTransfer(ID, Reg, none, case, control) :- !,
        allocReg(Reg),
        allocAssertCombFn(ID, control, case, Reg), !.
% one operand functions
allocScanTransfer(ID, Src, none, Op, Dst) :- !,
        allocReg(Src),
        allocReg(Dst),
        libOperator(Op, Fn, Class),
        allocAssertCombFn(ID, Class, Fn, none), !.
% two operand functions
allocScanTransfer(ID, Src1, Src2, Op, Dst) :- !,
        allocReg(Src1),
        allocReg(Src2),
        allocReg(Dst),
        libOperator(Op, Fn, Class),
        allocAssertCombFn(ID, Class, Fn, none), !.

% note combinational functions
allocAssertCombFn(ID, Class, Fn, Arg) :-
        allocCombFn(Class, Fn, Arg), !,
        allocAssertCombUse(ID, Fn, Arg).
allocAssertCombFn(ID, Class, Fn, Arg) :-
        assert(( allocCombFn(Class, Fn, Arg) )),
        allocAssertCombUse(ID, Fn, Arg), !.
        allocAssertFnList(FU, Tail).

% note cycles with parallelism
allocAssertCombUse(ID, Fn, Arg) :-
        cycle(ID, Block, Cycle),
        functionUse(_, Block, Cycle, Fn, Arg), !,
        (show -> write('... concurrency in '), write(Block),
                 write(-), write(Cycle), write(' with '),
                 write(Fn), write(-), write(Arg), nl
        ;
         true
        ),
        assert(( allocCombPar(Block, Cycle, Fn, Arg) )),
        assert(( functionUse(ID, Block, Cycle, Fn, Arg) )).
allocAssertCombUse(ID, Fn, Arg) :-
        cycle(ID, Block, Cycle),
        assert(( functionUse(ID, Block, Cycle, Fn, Arg) )), !.

allocReg(none) :-
        !.
allocReg(constant(Constant)) :-
        (show -> write('... constant register '), write(Constant), nl
        ;
         true
        ), !.
```

```
allocReg(field(Reg, Field)) :-
        allocReg(Reg), !.
allocReg(Reg) :-
        unit(Reg, reg), !.
allocReg(Reg) :-
        assert(( unit(Reg, reg) )), !.

%*
%*  allocate functional units
%*

allocUnits :-
        allocArithLogicals,
        allocShifts,
        allocControls,
        allocMemFns.

%*  allocate all arithmetic-logical units
allocArithLogicals :-
        setof(X, T^(allocCombFn(arlog, X, T)), S),
        allocArithLogical(S), !.
allocArithLogicals.

% special case: no arlog
allocArithLogical([]).
% special case: add only
allocArithLogical([add]) :-
        libUnit(adder), !,
        allocUnitName(adder, Unit),
        assert(( functionBinding(Unit, add) )),
        (allocCombFn(count, inc, _) ->
                assert(( functionBinding(Unit, inc) )) ),
        allocRebindOneArg(inc).
% general case: ALU
allocArithLogical(S) :-
        libUnit(alu), !,
        allocUnitName(alu, Unit),
        allocUnitFns(Unit, S),
        allocIncrement(Unit).
% error
allocArithLogical(S) :-
        (show -> write('... unable to implement ALU functions '), write(S), nl
        ;
         true
        ).

% special case: increment register
allocIncrement(ALU) :-
        % (do for all allocCombFn's and schedule to disambiguate)
        allocCombFn(count, inc, Counter),
        libUnit(increg), !,
        functionUse(ID, Block, Cycle, inc, Counter),
        assert(( functionBinding(Counter, inc) )),
        allocRebindNoArgs(ID).
% special case: increment ALU
allocIncrement(ALU) :-
        % (do for all allocCombFn's and schedule to disambiguate)
        allocCombFn(count, inc, Counter), !,
        assert(( functionBinding(ALU, inc) )),
        allocRebindOneArg(inc).
% general case: no increment
allocIncrement(_).

allocUnitFns(_, []).
allocUnitFns(Unit, [Function | Tail]) :-
        unit(Unit, Type),
        libFunction(Type, Function), !,
        assert(( functionBinding(Unit, Function) )),
        allocUnitFns(Unit, Tail).
allocUnitFns(Unit, [Function | Tail]) :-
        (show -> write('... function '), write(Function), write(' in '),
                 write(Unit), write(' is unsupported'), nl
        ;
         true
        ),
        allocUnitFns(Unit, Tail).

allocShifts :-
        setof(X, T^(allocCombFn(shift, X, T)), S),
        allocShift(S), !.
allocShifts.

% special case shift right one
allocShift([shr1]) :-
        libFunction(alu, shr1),
        unit(ALU, alu), !,
        % assumes no conflict between shift and alu operations
        assert(( functionBinding(ALU, shr1) )),
        allocRebindOneArg(shr1).
allocShift([shr1]) :-
        libUnit(shfone), !,
        allocUnitName(shfone, Unit),
        assert(( functionBinding(Unit, shr1) )),
        allocRebindOneArg(shr1).
allocShift(S) :-
        (show -> write('... unable to implement shift functions '),
                 write(S), nl
        ;
         true
        ).

allocControls :-
        setof(X, T^(allocCombFn(control, X, T)), S),
        allocControl(S), !.
allocControls.

allocControl([]).
allocControl([case | Tail]) :-
        % (do for all such allocCombFn's)
        allocCombFn(control, case, Reg),
        assert(( functionBinding(Reg, case) )),
        allocControl(Tail), !.
allocControl([ltzero | Tail]) :-
        % (do for all such allocCombFn's)
        allocCombFn(control, ltzero, Reg),
        assert(( functionBinding(Reg, ltzero) )),
        allocControl(Tail), !.
allocControl([C | Tail]) :-
        (show -> write('... unknown control function '), write(C), nl ; true),
        allocControl(Tail), !.

allocMemFns :-
        setof(X, T^(allocCombFn(mem, X, T)), S),
        allocMemFn(S), !.
allocMemFns.

allocMemFn([]).
allocMemFn([mem_read | Tail]) :-
        assert(( functionBinding(mem, mem_read) )),
        allocMemFn(Tail), !.
allocMemFn([mem_write | Tail]) :-
        assert(( functionBinding(mem, mem_write) )),
        allocMemFn(Tail), !.
allocMemFn([M | Tail]) :-
        (show -> write('... unknown memory function '), write(M), nl ; true),
        allocMemFn(Tail), !.
```

```
allocUnitName(Type, Name) :-
        allocUnitIndex(Type, 1, NewIndex),
        Name =.. [Type, NewIndex],
        assert(( unit(Name, Type) )), !.

allocUnitIndex(Type, ThisIndex, LastIndex) :-
        Name =.. [Type, ThisIndex],
        unit(Name, Type), !,
        NextIndex is ThisIndex + 1,
        allocUnitIndex(Type, NextIndex, LastIndex).
allocUnitIndex(_, Index, Index).

allocRebindOneArg(Fn) :-
        % (do for all functionUse's)
        functionUse(ID, _, _, Fn, _),
        transfer(ID, _, Src, _, _, Dst),
        assert(( argRebinding(ID, Src, Dst) )), !.
allocRebindOneArg(Fn) :-
        (show -> write('... rebind error for '), write(ID), nl ; true).

allocRebindNoArgs(ID) :-
        assert(( argRebinding(ID, none, none) )).

%*
%*  utilities
%*

allocInitialize :-
        flush(unit, 2),
        flush(functionBinding, 2),
        flush(functionUse, 5),
        flush(argRebinding, 3),
        flush(allocCombFn, 3),
        flush(allocCombPar, 4), !.

allocList :-
        listing(unit),
        listing(functionBinding),
        listing(functionUse),
        listing(argRebinding),
        listing(allocCombFn),
        listing(allocCombPar), !.

allocWrite(File) :-
        tell(File),
        allocList,
        close(File).



%** Data Path Connecter and Scheduler
%**     Connect and schedule functional units

%** data base items:
%*
%*      input: transfer, label, cycle, libTwoPorts,
%*              unit, functionBinding, functionUse, argRebinding
%*
%*      busSrc(Bus, Resource)
%*      busDst(Bus, Resource)
%*
%*      bus(Bus)
%*      do(Unit, Fn, Block, Cycle, ID)
%*      move(Bus, Src, Dst, Block, Cycle, ID)

%* main routine
conn :-
        connInitialize,
        connBlocks.
```

```
%*  process all blocks
connBlocks :-
        label(_, _, Block),
        (show -> write(Block) ; true),
        connBlock(Block),
        fail.
connBlocks.

%*  process all transfers in a block
connBlock(Block) :-
        transfer(ID, Block, Src1, Src2, OpType, Dst),
        (show -> tab(1), write(ID) ; true),
        connTransfer(ID, Src1, Src2, OpType, Dst),
        fail.
connBlock(Block) :-
        (show -> nl ; true), !.

% transfer: move
connTransfer(ID, Src, none, move, Dst) :-
        connSchedBus(ID, Src, Dst), !.
%
% transfer: memory
connTransfer(ID, _, _, mem_read, _) :-
        connSchedUnit(ID, Unit), !.
connTransfer(ID, _, _, mem_write, _) :-
        connSchedUnit(ID, Unit), !.
%
% transfer: control (passive)
connTransfer(ID, _, _, _, control) :- !.
%
% transfer: rebound, no arguments
connTransfer(ID, _, _, _, _) :-
        argRebinding(ID, none, none), !,
        connSchedUnit(ID, Unit).
%
% transfer: rebound, one argument and one destination
connTransfer(ID, _, _, _, _) :-
        argRebinding(ID, Src, Dst), !,
        connSchedUnit(ID, Unit),
        connSchedBus(ID, Src, Unit),
        connSchedBus(ID, Unit, Dst).
%
% transfer: one operand function
connTransfer(ID, Src, none, _, Dst) :-
        connSchedUnit(ID, Unit),
        connSchedBus(ID, Src, Unit),
        connSchedBus(ID, Unit, Dst), !.
%
% transfer: two operand function
connTransfer(ID, Src1, Src2, _, Dst) :-
        connSchedUnit(ID, Unit),
        connSchedBus(ID, Src1, Src2, Unit),
        connSchedBus(ID, Unit, Dst), !.

% already scheduled (by alloc)
connSchedUnit(ID, Unit) :-
        do(Unit, _, _, _, ID), !.
% schedule from alloc information
connSchedUnit(ID, Unit) :-
        functionBinding(Unit, Fn),
        functionUse(ID, Block, Cycle, Fn, Arg),
        assert(( do(Unit, Fn, Block, Cycle, ID) )), !.

connSchedBus(ID, Src, Dst) :-
        cycle(ID, Block, Cycle),
        connGetFreeBus(Block, Cycle, Src, Dst, Bus),
        assert(( move(Bus, Src, Dst, Block, Cycle, ID) )).
```

```
connSchedBus(ID, Src1, Src2, Unit) :-
        connPortName(Unit, 1, Dst1),
        connSchedBus(ID, Src1, Dst1),
        connPortName(Unit, 2, Dst2),
        connSchedBus(ID, Src2, Dst2).

connGetFreeBus(Block, Cycle, Src, Dst, Bus) :-
        % bus connects and is available
        busSrc(Bus, Src),
        busDst(Bus, Dst),
        \+ (move(Bus, _, _, Block, Cycle, _)), !.
connGetFreeBus(Block, Cycle, Src, Dst, Bus) :-
        % bus connects to src and is available
        busSrc(Bus, Src),
        \+ (busDst(Bus, Dst)),
        \+ (move(Bus, _, _, Block, Cycle, _)), !,
        assert(( busDst(Bus, Dst) )).
connGetFreeBus(Block, Cycle, Src, Dst, Bus) :-
        % bus connects to dst and is available
        \+ (busSrc(Bus, Src)),
        busDst(Bus, Dst),
        \+ (move(Bus, _, _, Block, Cycle, _)), !,
        assert(( busSrc(Bus, Src) )).
connGetFreeBus(Block, Cycle, Src, Dst, bus(Index)) :-
        % bus is available
        bus(Index),
        \+ (move(bus(Index), _, _, Block, Cycle, _)), !,
        assert(( busSrc(bus(Index), Src) )),
        assert(( busDst(bus(Index), Dst) )).
connGetFreeBus(Block, Cycle, Src, Dst, Bus) :-
        % create new bus
        connBusName(Bus),
        assert(( busSrc(Bus, Src) )),
        assert(( busDst(Bus, Dst) )), !.

connPortName(Unit, Index, port(Unit, Index)).

connBusName(bus(NewIndex)) :-
        connBusIndex(1, NewIndex),
        assert(( bus(NewIndex) )), !.

connBusIndex(ThisIndex, LastIndex) :-
        bus(ThisIndex), !,
        NextIndex is ThisIndex + 1,
        connBusIndex(NextIndex, LastIndex).
connBusIndex(Index, Index).

%*
%*  utilities
%*

connInitialize :-
        flush(connIndex, 2),
        flush(bus, 1),
        flush(do, 5),
        flush(move, 6),
        flush(busSrc, 2),
        flush(busDst, 2), !.

connList :-
        listing(bus),
        listing(do),
        listing(move),
        listing(busSrc),
        listing(busDst), !.

connWrite(File) :-
        tell(File),
        connList,
        close(File).
```

```
%** Library

libOperator('+', add, arlog).
libOperator('-', sub, arlog).
libOperator('/\', and, arlog).
libOperator('\/', or, arlog).
libOperator('\', comp, arlog).
libOperator('>>', shr, shift).
libOperator('<<', shl, shift).
libOperator(mem_read, mem_read, mem).
libOperator(mem_write, mem_write, mem).

libUnit(increg).% incremented register
libUnit(adder).         % adder
libUnit(alu).           % ALU
libUnit(shfone).% shifter

libFunction(increg, inc).
libFunction(adder, add).
libFunction(adder, inc).
libFunction(alu, add).
libFunction(alu, inc).
libFunction(alu, and).
%libFunction(alu, or).
%libFunction(alu, shrl).
%libFunction(alu, shll).
libFunction(shfone, shrl).
%libFunction(shfone, shll).

libTwoPorts(adder).
libTwoPorts(alu).
```

```
% symbolic SM1

stateRegister(ac, 16).
stateRegister(pc, 16).
stateRegister(memAR, 16).
stateRegister(memDR, 16).
stateField(memDR, inst, opcode, 1).
stateField(memDR, inst, address, 2).

run :-
        write('--fetch '),stateCount(C1),write(C1),nl,
    fetch, !,
        write('--update '),stateCount(C2),write(C2),nl,
    stateUpdate, !,
        write('--access'),nl,
    access(memDR, opcode, OP), !,
        write('--execute '),write(OP),nl,
    !, execute(OP), !,
        write('--update '),stateCount(C3),write(C3),nl,
    stateUpdate, !,
        write('--recurse'),nl,
    run.
run :- true.

fetch :-
    access(pc, PC), set(memAR, PC),
    mem_read,
    access(pc, PC), P1 is PC+1, set(pc, P1).

execute(halt) :- !,
    fail.
execute(add) :- !,
    access(memDR, address, X), set(memAR, X),
    mem_read,
    access(memDR, T), access(ac, AC), A is T+AC, set(ac, A).
execute(and) :- !,
    access(memDR, address, X), set(memAR, X),
    mem_read,
    access(memDR, T), access(ac, AC), A is T/\AC, set(ac, A).
execute(shr) :- !,
    access(ac, AC), A is AC>>1, set(ac, A).
execute(load) :- !,
    access(memDR, address, X), set(memAR, X),
    mem_read,
    access(memDR, T), set(ac, T).
execute(stor) :- !,
    access(memDR, address, X), set(memAR, X),
    access(ac, T), set(memDR, T),
    mem_write.
execute(jump) :- !,
    access(memDR, address, T), set(pc, T).
execute(brn) :-
    access(ac, AC), AC<0, !,
    access(memDR, address, T), set(pc, T).
execute(brn) :- !,
    true.
```

.inverter.bench

```
# /*
  set-up.inverter: bench set-up for (asp/compactor) inverter
  */
inverter :- driver(inverter).

benchmark(inverter,
          compact('examples/out/inverter'),
          dummy('examples/out/inverter'),
          1) :-
        consult('examples/in/inverter.sip').

show(inverter) :- reconsult('examples/in/inverter.sip'),
                  assert(show),
                  compact('examples/out/inverter'),
                  retract(show).


#include "driver"
```

.random_logic.bench

```
# /*
  set-up.random_logic: bench set-up for (asp/compactor) random_logic
  */
random_logic :- driver(random_logic).

benchmark(random_logic,
          compact('examples/out/random_logic'),
          dummy('examples/out/random_logic'),
          1) :-
        consult('examples/in/random_logic.sip').

show(random_logic) :- reconsult('examples/in/random_logic.sip'),
                      assert(show),
                      compact('examples/out/random_logic'),
                      retract(show).


#include "driver"
```

.sm1.bench

```
# /*
  set-up.sm1: bench set-up for (asp/viper) sm1
  */
sm1 :- driver(sm1).

benchmark(sm1, viper('examples/out/sm1'), dummy('examples/out/sm1'), 10) :-
        reconsult('examples/in/sm1').

show(viper) :- reconsult('examples/in/sm1'),
               assert(show),
               viper('examples/out/sm1').
               retact(show).


#include "driver"
```

berkeley

```
# /*
  adder.m: benchmark (circuit) adder master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (circuit) adder
%
%   Alvin M. Despain (despain@cse.usc.edu)
%
%   September 1986
%
%   design a (full) adder using 2-input NAND gates

#assign ADDER_SPEC      [0,0,0,1,0,1,1,1]
#
#if BENCH
#  include ".adder.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
adder :- circuit(ADDER_SPEC, Solution),
         write(adder), write(': '), write(Solution), nl.
#  else
adder :- circuit(ADDER_SPEC, _).
#  endif
#endif

#include "circuit"      /* code for circuit design */
```

```
# /*
  mux.m: benchmark (circuit) mux master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (circuit) mux
%
%   Alvin M. Despain (despain@cse.usc.edu)
%
%   September 1986
%
%   design a 2-1 mux using 2-input NAND gates

#assign MUX_SPEC        [0,1,0,1,0,0,1,1]
#
#if BENCH
#  include ".mux.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
mux :- circuit(MUX_SPEC, Solution),
       write(mux), write(': '), write(Solution), nl.
#  else
mux :- circuit(MUX_SPEC, _).
#  endif
#endif

#include "circuit"      /* code for circuit design */
```

```
# /*
  circuit: code for circuit design
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (circuit/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
circuit(_, _).
#else
%   This is a program to design a 3-input digital circuit
%   using 2-input NAND gates given the truth table of the
%   desired circuit.  For example,
%
%   ?- circuit([0,1,0,1,0,0,1,1], Solution).
%
%   designs a 2 to 1 MUX.  ([0,1,0,1,0,0,1,1] means SAB =
%   000 <=> out = 0, SAB = 001 <=> out = 1, et cetera.)
%
%   The strategy is breadth-first search, where circuits
%   at level N of the search tree contain N gates.
%
%   (Clauses of signals/2 could be added to deal with
%   circuits having other than 3 inputs.)


circuit(Specification, Solution) :-
        num(Depth_limit),
        search(Depth_limit, 0, Specification, Solution), !.


search(_Depth_limit, _Depth, Table, Solution) :-
        signals(Solution, Table).

search(Depth_limit, Depth, Table, nand(Sl1,Sl2)) :-
        Depth < Depth_limit,
        D is Depth + 1,
        search(Depth_limit, D, Sp1, Sl1),
        ngate(Table, Sp1, Sp2),
        search(Depth_limit, D, Sp2, Sl2).


% Input signals are free and terminate the search.

signals( 0 , [0,1,0,1,0,1,0,1]).
signals( 1 , [0,0,1,1,0,0,1,1]).
signals( 2 , [0,0,0,0,1,1,1,1]).
signals( v , [1,1,1,1,1,1,1,1]). % Turn a NAND gate into an inverter.
signals(i0 , [1,0,1,0,1,0,1,0]).
signals(i1 , [1,1,0,0,1,1,0,0]).
signals(i2 , [1,1,1,1,0,0,0,0]).
```

```
% Optimized for "side" gate signal transformation.

ngate([], [], []).
ngate([1|T0], [0|T1], [_|T2]) :- ngate(T0, T1, T2).
ngate([1|T0], [1|T1], [0|T2]) :- ngate(T0, T1, T2).
ngate([0|T0], [1|T1], [1|T2]) :- ngate(T0, T1, T2).


num(0).
num(N) :- num(M), N is M + 1.
#endif
```

concat_1.m

```
# /*
  concat_1.m: benchmark (concat) concat_1 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (concat) concat_1
%
%   (deterministically) concatenate [a,b,c] and [d,e]

#if BENCH
#  include ".concat_1.bench"
#else
concat_1 :- concat([a,b,c],[d,e],_).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (concat/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
concat(_,_,_).
#else
concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
#endif
```

concat_6.m

```
# /*
  concat_6.m: benchmark (concat) concat_6 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (concat) concat_6
%
%   (nondeterministically) "deconcatenate" [a,b,c,d,e] (6 possibilities exist)

#if BENCH
#  include ".concat_6.bench"
#else
concat_6 :- run_concat_6.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (run_concat_6/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
run_concat_6.
#else
run_concat_6 :- concat(_,_,[a,b,c,d,e]), fail.
run_concat_6.

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
#endif
```

hanoi_8.m

```
# /*
  hanoi_8.m: benchmark (hanoi) hanoi_8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (hanoi) hanoi_8
%
%   solve the 8-disk towers of Hanoi problem

#if BENCH
#  include ".hanoi_8.bench"
#else
hanoi_8 :- hanoi(8).
#endif

#include "hanoi"        /* code for solving the N-disk towers of Hanoi */
```

hanoi_16.m

```
# /*
  hanoi_16.m: benchmark (hanoi) hanoi_16 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (hanoi) hanoi_16
%
%   solve the 16-disk towers of Hanoi problem

#if BENCH
#  include ".hanoi_16.bench"
#else
hanoi_16 :- hanoi(16).
#endif

#include "hanoi"        /* code for solving the N-disk towers of Hanoi */
```

```
# /*
  hanoi: code for solving the N-disk towers of Hanoi problem
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (hanoi/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
hanoi(_).
#else
%   This program solves the towers of Hanoi problem:  move a tower of
%   N (punctured) disks of various diameters from one peg to another
%   with the help of an auxiliary peg and according to the rules (1)
%   only one disk can be moved at a time and (2) a larger disk cannot
%   be put on top of a smaller disk.  The algorithm is deterministic
%   and highly recursive.
%
%   No static representation of the solution is accumulated by this
%   implementation of the algorithm.

hanoi(N) :- move(N,a,b,c).

move(0,_,_,_) :- !.
move(N,A,B,C) :- M is N-1, move(M,A,C,B), move(M,C,B,A).
#endif
```

```
# /*
  mu.m: benchmark mu master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   mu
%
%   derived from Douglas R. Hofstadter, "Godel, Escher, Bach," pages 33-35.
%
%   prove "mu-math" theorem muiiu

#if BENCH
#  include ".mu.bench"
#else
mu :- theorem(5, [m,u,i,i,u]).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (theorem/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
theorem(_, _).
#else
theorem(_, [m,i]).
theorem(_, []) :-
    fail.
theorem(Depth, R) :-
    Depth > 0,
    D is Depth-1,
    theorem(D, S),
    rules(S, R).

rules(S, R) :- rule1(S, R).
rules(S, R) :- rule2(S, R).
rules(S, R) :- rule3(S, R).
rules(S, R) :- rule4(S, R).

rule1(S, R) :-
    append(X, [i], S),
    append(X, [i,u], R).

rule2([m|T], [m|R]) :-
    append(T, T, R).

rule3([], -) :-
    fail.
rule3(R, T) :-
    append([i,i,i], S, R),
    append([u], S, T).
rule3([H|T], [H|R]) :-
    rule3(T, R).

rule4([], -) :-
    fail.
rule4(R, T) :-
    append([u,u], T, R).
rule4([H|T], [H|R]) :-
    rule4(T, R).
```

```
append([], X, X).
append([A|B], X, [A|B1]) :-
    append(B, X, B1).
#endif
```

```
# /*
  prime_100.m: benchmark (prime) prime_100 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (prime) prime_100
%
%   from Clocksin and Mellish, "Programming in Prolog" (edition 1), page 157.
%
%   find every prime number less than 100

#if BENCH
#  include ".prime_100.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
prime_100 :- primes(100, Ps), write(Ps), nl.
#  else
prime_100 :- primes(100, _).
#  endif
#endif

#include "prime"        /* code to find every prime less than N */
```

prime_1000.m

```
# /*
  prime_1000.m: benchmark (prime) prime_1000 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (prime) prime_1000
%
%   from Clocksin and Mellish, "Programming in Prolog" (edition 1), page 157.
%
%   find every prime number less than 1000

#if BENCH
#  include ".prime_1000.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
prime_1000 :- primes(1000, Ps), write(Ps), nl.
#  else
prime_1000 :- primes(1000, _).
#  endif
#endif

#include "prime"        /* code to find every prime less than N */
```

```
# /*
  prime: code to find every prime less than N
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (prime/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
prime(_, _).
#else
%   This program uses a version of the Sieve of Erastosthenes
%   to make a list of every prime number less than N.

primes(N, Ps) :- integers(2, N, Is), sift(Is, Ps).

integers(Low, High, [Low|Rest]) :-
        Low < High, !, M is Low+1, integers(M, High, Rest).
integers(_, _, []).

sift([], []).
sift([I|Is], [I|Ps]) :- remove(I, Is, New), sift(New, Ps).

remove(_, [], []).
remove(P, [I|Is], [I|Nis]) :-
        \+ (0 is I mod P), !, remove(P, Is, Nis).
remove(P, [I|Is], Nis) :-
        0 is I mod P, remove(P, Is, Nis).
#endif
```

queens_4.m

```
# /*
  queens_4.m: benchmark (queens) queens_4 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (queens) queens_4
%
%   from Sterling and Shapiro, "The Art of Prolog," page 211.
%
%   solve the 4 queens problem

#if BENCH
#  include ".queens_4.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
queens_4 :- queens(4,Qs), !,
            write(queens_4), write(': '), write(Qs), nl.
#  else
queens_4 :- queens(4,_), !.
#  endif
#endif

#include "queens"       /* code for solving the N queens problem */
```

```
# /*
  queens_8.m: benchmark (queens) queens_8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (queens) queens_8
%
%   from Sterling and Shapiro, "The Art of Prolog," page 211.
%
%   solve the 8 queens problem

#if BENCH
#  include ".queens_8.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
queens_8 :- queens(8,Qs), !,
            write(queens_8), write(': '), write(Qs), nl.
#  else
queens_8 :- queens(8,_), !.
#  endif
#endif

#include "queens"       /* code for solving the N queens problem */
```

```
# /*
  queens: code for solving the N queens problem
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (queens/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
queens(_,_).
#else
%   This program solves the N queens problem:  place N pieces on an N
%   by N rectangular board so that no two pieces are on the same line
%   - horizontal, vertical, or diagonal.  (N queens so placed on an N
%   by N chessboard are unable to attack each other in a single move
%   under the rules of chess.)  The strategy is incremental generate-
%   and-test.
%
%   A solution is specified by a permutation of the list of numbers 1 to
%   N.  The first element of the list is the row number for the queen in
%   the first column, the second element is the row number for the queen
%   in the second column, et cetera.  This scheme implicitly incorporates
%   the observation that any solution of the problem has exactly one queen
%   in each column.
%
%   The program distinguishes symmetric solutions.  For example,
%
%   ?- queens(4, Qs).
%
%   produces
%
%   Qs = [3,1,4,2] ;
%
%   Qs = [2,4,1,3]

queens(N,Qs) :-
        range(1,N,Ns),
        queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
        select(Q,UnplacedQs,UnplacedQs1),
        \+ attack(Q,SafeQs),
        queens(UnplacedQs1,[Q|SafeQs],Qs).

attack(X,Xs) :-
        attack(X,1,Xs).

attack(X,N,[Y|_Ys]) :-
        X is Y+N ; X is Y-N.
attack(X,N,[_Y|Ys]) :-
        N1 is N+1,
        attack(X,N1,Ys).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :-
        M < N,
        M1 is M+1,
        range(M1,N,Ns).
#endif
```

```
# /*
  set-up.adder: bench set-up for (circuit) adder
  */
adder :- driver(adder).

benchmark(adder,
          circuit(ADDER_SPEC, _),
          dummy(ADDER_SPEC, _),
          5).

show(adder) :- circuit(ADDER_SPEC, Solution),
               write(adder), write(': '), write(Solution), nl.


#include "driver"
```

```
# /*
  set-up.mux: bench set-up for (circuit) mux
  */
mux :- driver(mux).

benchmark(mux,
          circuit(MUX_SPEC, _),
          dummy(MUX_SPEC, _),
          50).

show(mux) :- circuit(MUX_SPEC, Solution),
             write(mux), write(': '), write(Solution), nl.


#include "driver"
```

.concat_1.bench

```
# /*
  set-up.concat_1: bench set-up for (concat) concat_1
  */
concat_1 :- driver(concat_1).

benchmark(concat_1, concat([a,b,c],[d,e],_), dummy([a,b,c],[d,e],_), 25000).

show(concat_1) :- concat([a,b,c],[d,e],Z),
                  write('concat([a,b,c],[d,e],'),
                  write(Z), write(').'), nl.


#include "driver"
```

.concat_6.bench

```
# /*
  set-up.concat_6: bench set-up for (concat) concat_6
  */
concat_6 :- driver(concat_6).

benchmark(concat_6, run_concat_6, dummy, 25000).

show(concat_6) :- concat(X,Y,[a,b,c,d,e]),
                  write('concat('),
                  write(X), write(','),
                  write(Y), write(',[a,b,c,d,e]).'), nl,
                  fail.
show(concat_6).


#include "driver"
```

.hanoi_8.bench

```
# /*
  set-up.hanoi_8: bench set-up for (hanoi) hanoi_8
  */
hanoi_8 :- driver(hanoi_8).

benchmark(hanoi_8, hanoi(8), dummy(8), 750).

#message "NOTE: show/1 is NOT defined for hanoi_8"


#include "driver"
```

.hanoi_16.bench

```
# /*
  set-up.hanoi_16: bench set-up for (hanoi) hanoi_16
  */
hanoi_16 :- driver(hanoi_16).

benchmark(hanoi_16, hanoi(16), dummy(16), 3).

#message "NOTE: show/1 is NOT defined for hanoi_16"


#include "driver"
```

```
# /*
  set-up.mu: bench set-up for mu
  */
mu :- driver(mu).

benchmark(mu, theorem(5, [m,u,i,i,u]), dummy(5, [m,u,i,i,u]), 250).

#message "NOTE: show/1 is NOT defined for mu"


#include "driver"
```

.prime_100.bench

```
# /*
  set-up.prime_100: bench set-up for (prime) prime_100
  */
prime_100 :- driver(prime_100).

benchmark(prime_100, primes(100, _), dummy(100, _), 30).

show(prime_100) :- primes(100, Ps), write(Ps), nl.


#include "driver"
```

```
# /*
  set-up.prime_1000: bench set-up for (prime) prime_1000
  */
prime_1000 :- driver(prime_1000).

benchmark(prime_1000, primes(1000, _), dummy(1000, _), 3).

show(prime_1000) :- primes(1000, Ps), write(Ps), nl.


#include "driver"
```

.queens_4.bench

```
# /*
  set-up.queens_4: bench set-up for (queens) queens_4
  */
queens_4 :- driver(queens_4).

benchmark(queens_4, (queens(4,_), !), (dummy(4,_), !), 1000).

show(queens_4) :- queens(4,Qs), !,
                  write(queens_4), write(': '), write(Qs), nl.


#include "driver"
```

```
# /*
  set-up.queens_8: bench set-up for (queens) queens_8
  */
queens_8 :- driver(queens_8).

benchmark(queens_8, (queens(8,_), !), (dummy(8,_), !), 50).

show(queens_8) :- queens(8,Qs), !,
                  write(queens_8), write(': '), write(Qs), nl.


#include "driver"
```

chat_parser

```
# /*
  chat_parser.m: benchmark chat_parser master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%  chat_parser
%
%  Fernando C. N. Pereira and David H. D. Warren

#if BENCH
#  include ".chat_parser.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
chat_parser :- string(X),
               write(X), nl,
               determinate_say(X,Y),
               write(Y), nl, nl,
               fail.
chat_parser.
#  else
chat_parser :- string(X),
               determinate_say(X,_),
               fail.
chat_parser.
#  endif
#endif

#option "
        > The chat parser includes many clauses with
        > single occurences of variables.  If option
        > QUINTUS_PL is selected, then the directive
        >
        > :- no_style_check(single_var).
        >
        > is generated to silence Quintus Prolog's
        > complaining about these."
#if QUINTUS_PL

:- no_style_check(single_var).

#endif
```

```
%  query set

string([what,rivers,are,there,?]).
string([does,afghanistan,border,china,?]).
string([what,is,the,capital,of,upper_volta,?]).
string([where,is,the,largest,country,?]).
string([which,country,'`',s,capital,is,london,?]).
string([which,countries,are,european,?]).
string([how,large,is,the,smallest,american,country,?]).
string([what,is,the,ocean,that,borders,african,countries,
        and,that,borders,asian,countries,?]).
string([what,are,the,capitals,of,the,countries,bordering,the,baltic,?]).
string([which,countries,are,bordered,by,two,seas,?]).
string([how,many,countries,does,the,danube,flow,through,?]).
string([what,is,the,total,area,of,countries,south,of,the,equator,
        and,not,in,australasia,?]).
string([what,is,the,average,area,of,the,countries,in,each,continent,?]).
string([is,there,more,than,one,country,in,each,continent,?]).
string([is,there,some,ocean,that,does,not,border,any,country,?]).
string([what,are,the,countries,from,which,a,river,flows,
        into,the,black_sea,?]).


%  determinate_say

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (determinate_say/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
determinate_say(_,_).
#halt
#endif
determinate_say(X,Y) :-
   say(X,Y), !.


%-----------------------------------------------------------------------------
%
%  xgrun
%
%-----------------------------------------------------------------------------

terminal(T,S,S,x(_,terminal,T,X),X).
terminal(T,[T|S],S,X,X) :-
   gap(X).

gap(x(gap,_,_,_)).
gap([]).

virtual(NT,x(_,nonterminal,NT,X),X).
```

```
%-----------------------------------------------------------------------------
%
%  clotab
%
%-----------------------------------------------------------------------------

% normal form masks

is_pp(#(1,_,_,_)).

is_pred(#(_,1,_,_)).

is_trace(#(_,_,1,_)).

is_adv(#(_,_,_,1)).

trace(#(_,_,1,_),#(0,0,0,0)).

trace(#(0,0,1,0)).

adv(#(0,0,0,1)).

empty(#(0,0,0,0)).

np_all(#(1,1,1,0)).

s_all(#(1,0,1,1)).

np_no_trace(#(1,1,0,0)).

% mask operations

myplus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
   or(B1,C1,D1),
   or(B2,C2,D2),
   or(B3,C3,D3),
   or(B4,C4,D4).

minus(#(B1,B2,B3,B4),#(C1,C2,C3,C4),#(D1,D2,D3,D4)) :-
   anot(B1,C1,D1),
   anot(B2,C2,D2),
   anot(B3,C3,D3),
   anot(B4,C4,D4).

or(1,_,1).
or(0,1,1).
or(0,0,0).

anot(X,0,X).
anot(X,1,0).

% noun phrase position features

role(subj,_,#(1,0,0)).
role(compl,_,#(0,_,_)).
role(undef,main,#(_,0,_)).
role(undef,aux,#(0,_,_)).
role(undef,decl,_).
role(nil,_,_).

subj_case(#(1,0,0)).
verb_case(#(0,1,0)).
prep_case(#(0,0,1)).
compl_case(#(0,_,_)).
```

```
%---------------------------------------------------------------------------
%
%  newg
%
%---------------------------------------------------------------------------

say(X,Y) :-
   sentence(Y,X,[],[],[]).


sentence(B,C,D,E,F) :-
   declarative(B,C,G,E,H),
   terminator(.,G,D,H,F).
sentence(B,C,D,E,F) :-
   wh_question(B,C,G,E,H),
   terminator(?,G,D,H,F).
sentence(B,C,D,E,F) :-
   topic(C,G,E,H),
   wh_question(B,G,I,H,J),
   terminator(?,I,D,J,F).
sentence(B,C,D,E,F) :-
   yn_question(B,C,G,E,H),
   terminator(?,G,D,H,F).
sentence(B,C,D,E,F) :-
   imperative(B,C,G,E,H),
   terminator(!,G,D,H,F).


pp(B,C,D,E,F,F,G,H) :-
   virtual(pp(B,C,D,E),G,H).
pp(pp(B,C),D,E,F,G,H,I,J) :-
   prep(B,G,K,I,L),
   prep_case(M),
   np(C,N,M,O,D,E,F,K,H,L,J).


topic(B,C,D,x(gap,nonterminal,pp(E,compl,F,G),H)) :-
   pp(E,compl,F,G,B,I,D,J),
   opt_comma(I,C,J,H).


opt_comma(B,C,D,E) :-
   `(',',B,C,D,E).
opt_comma(B,B,C,C).


declarative(decl(B),C,D,E,F) :-
   s(B,G,C,D,E,F).


wh_question(whq(B,C),D,E,F,G) :-
   variable_q(B,H,I,J,D,K,F,L),
   question(I,J,C,K,E,L,G).


np(B,C,D,E,F,G,H,I,I,J,K) :-
   virtual(np(B,C,D,E,F,G,H),J,K).
np(np(B,C,[]),B,D,def,E,F,G,H,I,J,K) :-
   is_pp(F),
   pers_pron(C,B,L,H,I,J,K),
   empty(G),
   role(L,decl,D).
np(np(B,C,D),B,E,F,G,H,I,J,K,L,M) :-
   is_pp(H),
   np_head(C,B,F+N,O,D,J,P,L,Q),
   np_all(R),
   np_compls(N,B,G,O,R,I,P,K,Q,M).
```

```
np(part(B,C),3+D,E,indef,F,G,H,I,J,K,L) :-
   is_pp(G),
   determiner(B,D,indef,I,M,K,N),
   '(of,M,O,N,P),
   s_all(Q),
   prep_case(R),
   np(C,3+plu,R,def,F,Q,H,O,J,P,L).


variable_q(B,C,D,E,F,G,H,x(gap,nonterminal,np(I,C,E,J,K,L,M),N)) :-
   whq(B,C,I,D,F,G,H,N),
   trace(L,M).
variable_q(B,C,compl,D,E,F,G,x(gap,nonterminal,pp(pp(H,I),compl,J,K),L)) :-
   prep(H,E,M,G,N),
   whq(B,C,I,O,M,F,N,L),
   trace(J,K),
   compl_case(D).
variable_q(B,C,compl,D,E,F,G,x(gap,nonterminal,
           adv_phrase(pp(H,np(C,np_head(int_det(B),[],I),[])),J,K),L)) :-
   context_pron(H,I,E,F,G,L),
   trace(J,K),
   verb_case(D).
variable_q(B,C,compl,D,E,F,G,
           x(gap,nonterminal,predicate(adj,value(H,wh(B)),I),J)) :-
   '(how,E,K,G,L),
   adj(quant,H,K,F,L,J),
   empty(I),
   verb_case(D).


adv_phrase(B,C,D,E,E,F,G) :-
   virtual(adv_phrase(B,C,D),F,G).
adv_phrase(pp(B,C),D,E,F,G,H,I) :-
   loc_pred(B,F,J,H,K),
   pp(pp(prep(of),C),compl,D,E,J,G,K,I).


predicate(B,C,D,E,E,F,G) :-
   virtual(predicate(B,C,D),F,G).
predicate(B,C,D,E,F,G,H) :-
   adj_phrase(C,D,E,F,G,H).
predicate(neg,B,C,D,E,F,G) :-
   s_all(H),
   pp(B,compl,H,C,D,E,F,G).
predicate(B,C,D,E,F,G,H) :-
   s_all(I),
   adv_phrase(C,I,D,E,F,G,H).


whq(B,C,D,undef,E,F,G,H) :-
   int_det(B,C,E,I,G,J),
   s_all(K),
   np(D,C,L,M,subj,K,N,I,F,J,H).
whq(B,3+C,np(3+C,wh(B),[]),D,E,F,G,H) :-
   int_pron(D,E,F,G,H).


int_det(B,3+C,D,E,F,G) :-
   whose(B,C,D,E,F,G).
int_det(B,3+C,D,E,F,G) :-
   int_art(B,C,D,E,F,G).


gen_marker(B,B,C,D) :-
   virtual(gen_marker,C,D).
gen_marker(B,C,D,E) :-
   '('',B,F,D,G),
   an_s(F,C,G,E).
```

```
whose(B,C,D,E,F,x(nogap,nonterminal,np_head0(wh(B),C,proper),
      x(nogap,nonterminal,gen_marker,G))) :-
    `(whose,D,E,F,G).


question(B,C,D,E,F,G,H) :-
   subj_question(B),
   role(subj,I,C),
   s(D,J,E,F,G,H).
question(B,C,D,E,F,G,H) :-
   fronted_verb(B,C,E,I,G,J),
   s(D,K,I,F,J,H).


det(B,C,D,E,E,F,G) :-
   virtual(det(B,C,D),F,G).
det(det(B),C,D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   det(I,C,B,D).
det(generic,B,generic,C,C,D,D).


int_art(B,C,D,E,F,x(nogap,nonterminal,det(G,C,def),H)) :-
   int_art(B,C,G,D,E,F,H).


subj_question(subj).
subj_question(undef).


yn_question(q(B),C,D,E,F) :-
   fronted_verb(nil,G,C,H,E,I),
   s(B,J,H,D,I,F).


verb_form(B,C,D,E,F,F,G,H) :-
   virtual(verb_form(B,C,D,E),G,H).
verb_form(B,C,D,E,F,G,H,I) :-
   terminal(J,F,G,H,I),
   verb_form(J,B,C,D).


neg(B,C,D,D,E,F) :-
   virtual(neg(B,C),E,F).
neg(aux+B,neg,C,D,E,F) :-
   `(not,C,D,E,F).
neg(B,pos,C,C,D,D).


fronted_verb(B,C,D,E,F,x(gap,nonterminal,verb_form(G,H,I,J),
             x(nogap,nonterminal,neg(K,L),M))) :-
   verb_form(G,H,I,N,D,O,F,P),
   verb_type(G,aux+Q),
   role(B,J,C),
   neg(R,L,O,E,P,M).


imperative(imp(B),C,D,E,F) :-
   imperative_verb(C,G,E,H),
   s(B,I,G,D,H,F).


imperative_verb(B,C,D,x(nogap,terminal,you,x(nogap,nonterminal,
                verb_form(E,imp+fin,2+sin,main),F))) :-
   verb_form(E,inf,G,H,B,C,D,F).
```

```
s(s(B,C,D,E),F,G,H,I,J) :-
   subj(B,K,L,G,M,I,N),
   verb(C,K,L,O,M,P,N,Q),
   empty(R),
   s_all(S),
   verb_args(L,O,D,R,T,P,U,Q,V),
   minus(S,T,W),
   myplus(S,T,X),
   verb_mods(E,W,X,F,U,H,V,J).


subj(there,B,C+be,D,E,F,G) :-
   '(there,D,E,F,G).
subj(B,C,D,E,F,G,H) :-
   s_all(I),
   subj_case(J),
   np(B,C,J,K,subj,I,L,E,F,G,H).


np_head(B,C,D,E,F,G,H,I,J) :-
   np_head0(K,L,M,G,N,I,O),
   possessive(K,L,M,P,P,B,C,D,E,F,N,H,O,J).


np_head0(B,C,D,E,E,F,G) :-
   virtual(np_head0(B,C,D),F,G).
np_head0(name(B),3+sin,def+proper,C,D,E,F) :-
   name(B,C,D,E,F).
np_head0(np_head(B,C,D),3+E,F+common,G,H,I,J) :-
   determiner(B,E,F,G,K,I,L),
   adjs(C,K,M,L,N),
   noun(D,E,M,H,N,J).
np_head0(B,C,def+proper,D,E,F,x(nogap,nonterminal,gen_marker,G)) :-
   poss_pron(B,C,D,E,F,G).
np_head0(np_head(B,[],C),3+sin,indef+common,D,E,F,G) :-
   quantifier_pron(B,C,D,E,F,G).


np_compls(proper,B,C,[],D,E,F,F,G,G) :-
   empty(E).
np_compls(common,B,C,D,E,F,G,H,I,J) :-
   np_all(K),
   np_mods(B,C,L,D,E,M,K,N,G,O,I,P),
   relative(B,L,M,N,F,O,H,P,J).


possessive(B,C,D,[],E,F,G,H,I,J,K,L,M,N) :-
   gen_case(K,O,M,P),
   np_head0(Q,R,S,O,T,P,U),
   possessive(Q,R,S,V,[pp(poss,np(C,B,E))|V],F,G,H,I,J,T,L,U,N).
possessive(B,C,D,E,F,B,C,D,E,F,G,G,H,H).


gen_case(B,C,D,x(nogap,terminal,the,E)) :-
   gen_marker(B,C,D,E).


an_s(B,C,D,E) :-
   '(s,B,C,D,E).
an_s(B,B,C,C).


determiner(B,C,D,E,F,G,H) :-
   det(B,C,D,E,F,G,H).
determiner(B,C,D,E,F,G,H) :-
   quant_phrase(B,C,D,E,F,G,H).
quant_phrase(quant(B,C),D,E,F,G,H,I) :-
   quant(B,E,F,J,H,K),
   number(C,D,J,G,K,I).


quant(B,indef,C,D,E,F) :-
   neg_adv(G,B,C,H,E,I),
   comp_adv(G,H,J,I,K),
   '(than,J,D,K,F).
quant(B,indef,C,D,E,F) :-
   '(at,C,G,E,H),
   sup_adv(I,G,D,H,F),
   sup_op(I,B).
quant(the,def,B,C,D,E) :-
   '(the,B,C,D,E).
quant(same,indef,B,B,C,C).


neg_adv(B,not+B,C,D,E,F) :-
   '(not,C,D,E,F).
neg_adv(B,B,C,C,D,D).


sup_op(least,not+less).
sup_op(most,not+more).


np_mods(B,C,D,[E|F],G,H,I,J,K,L,M,N) :-
   np_mod(B,C,E,G,O,K,P,M,Q),
   trace(R),
   myplus(R,O,S),
   minus(G,S,T),
   myplus(O,G,U),
   np_mods(B,C,D,F,T,H,U,J,P,L,Q,N).
np_mods(B,C,D,D,E,E,F,F,G,G,H,H).


np_mod(B,C,D,E,F,G,H,I,J) :-
   pp(D,C,E,F,G,H,I,J).
np_mod(B,C,D,E,F,G,H,I,J) :-
   reduced_relative(B,D,E,F,G,H,I,J).


verb_mods([B|C],D,E,F,G,H,I,J) :-
   verb_mod(B,D,K,G,L,I,M),
   trace(N),
   myplus(N,K,O),
   minus(D,O,P),
   myplus(K,D,Q),
   verb_mods(C,P,Q,F,L,H,M,J).
verb_mods([],B,C,C,D,D,E,E).


verb_mod(B,C,D,E,F,G,H) :-
   adv_phrase(B,C,D,E,F,G,H).
verb_mod(B,C,D,E,F,G,H) :-
   is_adv(C),
   adverb(B,E,F,G,H),
   empty(D).
verb_mod(B,C,D,E,F,G,H) :-
   pp(B,compl,C,D,E,F,G,H).


adjs([B|C],D,E,F,G) :-
   pre_adj(B,D,H,F,I),
   adjs(C,H,E,I,G).
adjs([],B,B,C,C).
```

```
pre_adj(B,C,D,E,F) :-
   adj(G,B,C,D,E,F).
pre_adj(B,C,D,E,F) :-
   sup_phrase(B,C,D,E,F).


sup_phrase(sup(most,B),C,D,E,F) :-
   sup_adj(B,C,D,E,F).
sup_phrase(sup(B,C),D,E,F,G) :-
   sup_adv(B,D,I,F,J),
   adj(quant,C,I,E,J,G).


comp_phrase(comp(B,C,D),E,F,G,H,I) :-
   comp(B,C,F,J,H,K),
   np_no_trace(L),
   prep_case(M),
   np(D,N,M,O,compl,L,E,J,G,K,I).


comp(B,C,D,E,F,G) :-
   comp_adv(B,D,H,F,I),
   adj(quant,C,H,J,I,K),
   '(than,J,E,K,G).
comp(more,B,C,D,E,F) :-
   rel_adj(B,C,G,E,H),
   '(than,G,D,H,F).
comp(same,B,C,D,E,F) :-
   '(as,C,G,E,H),
   adj(quant,B,G,I,H,J),
   '(as,I,D,J,F).


relative(B,[C],D,E,F,G,H,I,J) :-
   is_pred(D),
   rel_conj(B,K,C,F,G,H,I,J).
relative(B,[],C,D,D,E,E,F,F).


rel_conj(B,C,D,E,F,G,H,I) :-
   rel(B,J,K,F,L,H,M),
   rel_rest(B,C,J,D,K,E,L,G,M,I).


rel_rest(B,C,D,E,F,G,H,I,J,K) :-
   conj(C,L,D,M,E,H,N,J,O),
   rel_conj(B,L,M,G,N,I,O,K).
rel_rest(B,C,D,D,E,E,F,F,G,G).


rel(B,rel(C,D),E,F,G,H,I) :-
   open(F,J,H,K),
   variable(B,C,J,L,K,M),
   s(D,N,L,O,M,P),
   trace(Q),
   minus(N,Q,E),
   close(O,G,P,I).


variable(B,C,D,E,F,x(gap,nonterminal,np(np(B,wh(C),[]),B,G,H,I,J,K),L)) :-
   '(that,D,E,F,L),
   trace(J,K).
variable(B,C,D,E,F,x(gap,nonterminal,np(G,H,I,J,K,L,M),N)) :-
   wh(C,B,G,H,I,D,E,F,N),
   trace(L,M).
variable(B,C,D,E,F,x(gap,nonterminal,pp(pp(G,H),compl,I,J),K)) :-
   prep(G,D,L,F,M),
   wh(C,B,H,N,O,L,E,M,K),
   trace(I,J),
   compl_case(O).
```

```
wh(B,C,np(C,wh(B),[]),C,D,E,F,G,H) :-
   rel_pron(I,E,F,G,H),
   role(I,decl,D).
wh(B,C,np(D,E,[pp(F,G)]),D,H,I,J,K,L) :-
   np_head0(E,D,M+common,I,N,K,O),
   prep(F,N,P,O,Q),
   wh(B,C,G,R,S,P,J,Q,L).
wh(B,C,D,E,F,G,H,I,J) :-
   whose(B,C,G,K,I,L),
   s_all(M),
   np(D,E,F,def,subj,M,N,K,H,L,J).


reduced_relative(B,C,D,E,F,G,H,I) :-
   is_pred(D),
   reduced_rel_conj(B,J,C,E,F,G,H,I).


reduced_rel_conj(B,C,D,E,F,G,H,I) :-
   reduced_rel(B,J,K,F,L,H,M),
   reduced_rel_rest(B,C,J,D,K,E,L,G,M,I).


reduced_rel_rest(B,C,D,E,F,G,H,I,J,K) :-
   conj(C,L,D,M,E,H,N,J,O),
   reduced_rel_conj(B,L,M,G,N,I,O,K).
reduced_rel_rest(B,C,D,D,E,E,F,F,G,G).


reduced_rel(B,reduced_rel(C,D),E,F,G,H,I) :-
   open(F,J,H,K),
   reduced_wh(B,C,J,L,K,M),
   s(D,N,L,O,M,P),
   trace(Q),
   minus(N,Q,E),
   close(O,G,P,I).


reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
           np(np(B,wh(C),[]),B,G,H,I,J,K),x(nogap,nonterminal,
           verb_form(be,pres+fin,B,main),x(nogap,nonterminal,
           neg(L,M),x(nogap,nonterminal,predicate(M,N,O),P))))) :-
   neg(Q,M,D,R,F,S),
   predicate(M,N,O,R,E,S,P),
   trace(J,K),
   subj_case(G).
reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
           np(np(B,wh(C),[]),B,G,H,I,J,K),x(nogap,nonterminal,
           verb(L,M,N,O),P))) :-
   participle(L,N,O,D,E,F,P),
   trace(J,K),
   subj_case(G).
reduced_wh(B,C,D,E,F,x(nogap,nonterminal,
           np(G,H,I,J,K,L,M),x(gap,nonterminal,
           np(np(B,wh(C),[]),B,N,O,P,Q,R),S))) :-
   s_all(T),
   subj_case(I),
   verb_case(N),
   np(G,H,U,J,subj,T,V,D,E,F,S),
   trace(L,M),
   trace(Q,R).
```

```
verb(B,C,D,E,F,F,G,H) :-
   virtual(verb(B,C,D,E),G,H).
verb(verb(B,C,D+fin,E,F),G,H,C,I,J,K,L) :-
   verb_form(M,D+fin,G,N,I,O,K,P),
   verb_type(M,Q),
   neg(Q,F,O,R,P,S),
   rest_verb(N,M,B,C,E,R,J,S,L),
   verb_type(B,H).


rest_verb(aux,have,B,C,[perf|D],E,F,G,H) :-
   verb_form(I,past+part,J,K,E,L,G,M),
   have(I,B,C,D,L,F,M,H).
rest_verb(aux,be,B,C,D,E,F,G,H) :-
   verb_form(I,J,K,L,E,M,G,N),
   be(J,I,B,C,D,M,F,N,H).
rest_verb(aux,do,B,active,[],C,D,E,F) :-
   verb_form(B,inf,G,H,C,D,E,F).
rest_verb(main,B,B,active,[],C,C,D,D).


have(be,B,C,D,E,F,G,H) :-
   verb_form(I,J,K,L,E,M,G,N),
   be(J,I,B,C,D,M,F,N,H).
have(B,B,active,[],C,C,D,D).


be(past+part,B,B,passive,[],C,C,D,D).
be(pres+part,B,C,D,[prog],E,F,G,H) :-
   passive(B,C,D,E,F,G,H).


passive(be,B,passive,C,D,E,F) :-
   verb_form(B,past+part,G,H,C,D,E,F),
   verb_type(B,I),
   passive(I).
passive(B,B,active,C,C,D,D).


participle(verb(B,C,inf,D,E),F,C,G,H,I,J) :-
   neg(K,E,G,L,I,M),
   verb_form(B,N,O,P,L,H,M,J),
   participle(N,C,D),
   verb_type(B,F).


passive(B+trans).
passive(B+ditrans).


participle(pres+part,active,[prog]).
participle(past+part,passive,[]).


close(B,B,C,D) :-
   virtual(close,C,D).


open(B,B,C,x(gap,nonterminal,close,C)).
```

```
verb_args(B+C,D,E,F,G,H,I,J,K) :-
   advs(E,L,M,H,N,J,O),
   verb_args(C,D,L,F,G,N,I,O,K).
verb_args(trans,active,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
verb_args(ditrans,B,[arg(C,D)|E],F,G,H,I,J,K) :-
   verb_arg(np,D,L,H,M,J,N),
   object(C,E,L,G,M,I,N,K).
verb_args(be,B,[void],C,C,D,E,F,G) :-
   terminal(there,D,E,F,G).
verb_args(be,B,[arg(predicate,C)],D,E,F,G,H,I) :-
   pred_conj(J,C,E,F,G,H,I).
verb_args(be,B,[arg(dir,C)],D,E,F,G,H,I) :-
   verb_arg(np,C,E,F,G,H,I).
verb_args(have,active,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
verb_args(B,C,[],D,D,E,E,F,F) :-
   no_args(B).


object(B,C,D,E,F,G,H,I) :-
   adv(J),
   minus(J,D,K),
   advs(C,L,K,F,M,H,N),
   obj(B,L,D,E,M,G,N,I).


obj(ind,[arg(dir,B)],C,D,E,F,G,H) :-
   verb_arg(np,B,D,E,F,G,H).
obj(dir,[],B,B,C,C,D,D).


pred_conj(B,C,D,E,F,G,H) :-
   predicate(I,J,K,E,L,G,M),
   pred_rest(B,J,C,K,D,L,F,M,H).


pred_rest(B,C,D,E,F,G,H,I,J) :-
   conj(B,K,C,L,D,G,M,I,N),
   pred_conj(K,L,F,M,H,N,J).
pred_rest(B,C,C,D,D,E,E,F,F).


verb_arg(np,B,C,D,E,F,G) :-
   s_all(H),
   verb_case(I),
   np(B,J,I,K,compl,H,C,D,E,F,G).


advs([B|C],D,E,F,G,H,I) :-
   is_adv(E),
   adverb(B,F,J,H,K),
   advs(C,D,E,J,G,K,I).
advs(B,B,C,D,D,E,E).


adj_phrase(B,C,D,E,F,G) :-
   adj(H,B,D,E,F,G),
   empty(C).
adj_phrase(B,C,D,E,F,G) :-
   comp_phrase(B,C,D,E,F,G).


no_args(trans).
no_args(ditrans).
no_args(intrans).


conj(conj(B,C),conj(B,D),E,F,conj(B,E,F),G,H,I,J) :-
   conj(B,C,D,G,H,I,J).
```

```
noun(B,C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   noun_form(H,B,C).


adj(B,adj(C),D,E,F,G) :-
   terminal(C,D,E,F,G),
   adj(C,B).


prep(prep(B),C,D,E,F) :-
   terminal(B,C,D,E,F),
   prep(B).


rel_adj(adj(B),C,D,E,F) :-
   terminal(G,C,D,E,F),
   rel_adj(G,B).


sup_adj(adj(B),C,D,E,F) :-
   terminal(G,C,D,E,F),
   sup_adj(G,B).


comp_adv(less,B,C,D,E) :-
   '(less,B,C,D,E).
comp_adv(more,B,C,D,E) :-
   '(more,B,C,D,E).


sup_adv(least,B,C,D,E) :-
   '(least,B,C,D,E).
sup_adv(most,B,C,D,E) :-
   '(most,B,C,D,E).


rel_pron(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   rel_pron(G,B).


name(B,C,D,E,F) :-
   opt_the(C,G,E,H),
   terminal(B,G,D,H,F),
   name(B).


int_art(B,plu,quant(same,wh(B)),C,D,E,F) :-
   '(how,C,G,E,H),
   '(many,G,D,H,F).
int_art(B,C,D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   int_art(I,B,C,D).


int_pron(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   int_pron(G,B).


adverb(adv(B),C,D,E,F) :-
   terminal(B,C,D,E,F),
   adverb(B).


poss_pron(pronoun(B),C+D,E,F,G,H) :-
   terminal(I,E,F,G,H),
   poss_pron(I,B,C,D).
```

```
pers_pron(pronoun(B),C+D,E,F,G,H,I) :-
   terminal(J,F,G,H,I),
   pers_pron(J,B,C,D,E).


quantifier_pron(B,C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   quantifier_pron(H,B,C).


context_pron(prep(in),place,B,C,D,E) :-
   '(where,B,C,D,E).
context_pron(prep(at),time,B,C,D,E) :-
   '(when,B,C,D,E).


number(nb(B),C,D,E,F,G) :-
   terminal(H,D,E,F,G),
   number(H,B,C).


terminator(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   terminator(G,B).


opt_the(B,B,C,C).
opt_the(B,C,D,E) :-
   '(the,B,C,D,E).


conj(B,list,list,C,D,E,F) :-
   terminal(',',C,D,E,F).
conj(B,list,'end',C,D,E,F) :-
   terminal(B,C,D,E,F),
   conj(B).


loc_pred(B,C,D,E,F) :-
   terminal(G,C,D,E,F),
   loc_pred(G,B).


'(B,C,D,E,F) :-
   terminal(B,C,D,E,F),
   '(B).
```

```
%-----------------------------------------------------------------------
%
%  newdic
%
%-----------------------------------------------------------------------

word(Word)  :- `(Word).
word(Word)  :- conj(Word).
word(Word)  :- adverb(Word).
word(Word)  :- sup_adj(Word,_).
word(Word)  :- rel_adj(Word,_).
word(Word)  :- adj(Word,_).
word(Word)  :- name(Word).
word(Word)  :- terminator(Word,_).
word(Word)  :- pers_pron(Word,_,_,_,_).
word(Word)  :- poss_pron(Word,_,_,_).
word(Word)  :- rel_pron(Word,_).
word(Word)  :- verb_form(Word,_,_,_).
word(Word)  :- noun_form(Word,_,_).
word(Word)  :- prep(Word).
word(Word)  :- quantifier_pron(Word,_,_).
word(Word)  :- number(Word,_,_).
word(Word)  :- det(Word,_,_,_).
word(Word)  :- int_art(Word,_,_,_).
word(Word)  :- int_pron(Word,_).
word(Word)  :- loc_pred(Word,_).

`(how).
`(whose).
`(there).
`(of).
`('`).          % use ` instead of ' to help assembler
`(',').
`(s).
`(than).
`(at).
`(the).
`(not).
`(as).
`(that).
`(less).
`(more).
`(least).
`(most).
`(many).
`(where).
`(when).

conj(and).
conj(or).

int_pron(what,undef).
int_pron(which,undef).
int_pron(who,subj).
int_pron(whom,compl).

int_art(what,X,_,int_det(X)).
int_art(which,X,_,int_det(X)).

det(the,No,the(No),def).
det(a,sin,a,indef).
det(an,sin,a,indef).
det(every,sin,every,indef).
det(some,_,some,indef).
det(any,_,any,indef).
det(all,plu,all,indef).
det(each,sin,each,indef).
det(no,_,no,indef).
```

```
number(W,I,Nb) :-
   tr_number(W,I),
   ag_number(I,Nb).

tr_number(nb(I),I).
tr_number(one,1).
tr_number(two,2).
tr_number(three,3).
tr_number(four,4).
tr_number(five,5).
tr_number(six,6).
tr_number(seven,7).
tr_number(eight,8).
tr_number(nine,9).
tr_number(ten,10).

ag_number(1,sin).
ag_number(N,plu) :- N>1.

quantifier_pron(everybody,every,person).
quantifier_pron(everyone,every,person).
quantifier_pron(everything,every,thing).
quantifier_pron(somebody,some,person).
quantifier_pron(someone,some,person).
quantifier_pron(something,some,thing).
quantifier_pron(anybody,any,person).
quantifier_pron(anyone,any,person).
quantifier_pron(anything,any,thing).
quantifier_pron(nobody,no,person).
quantifier_pron(nothing,no,thing).

prep(as).
prep(at).
prep(of).
prep(to).
prep(by).
prep(with).
prep(in).
prep(on).
prep(from).
prep(into).
prep(through).

noun_form(Plu,Sin,plu) :- noun_plu(Plu,Sin).
noun_form(Sin,Sin,sin) :- noun_sin(Sin).
noun_form(proportion,proportion,_).
noun_form(percentage,percentage,_).

root_form(1+sin).
root_form(2+_).
root_form(1+plu).
root_form(3+plu).

verb_root(be).
verb_root(have).
verb_root(do).
verb_root(border).
verb_root(contain).
verb_root(drain).
verb_root(exceed).
verb_root(flow).
verb_root(rise).
```

```
regular_pres(have).
regular_pres(do).
regular_pres(rise).
regular_pres(border).
regular_pres(contain).
regular_pres(drain).
regular_pres(exceed).
regular_pres(flow).

regular_past(had,have).
regular_past(bordered,border).
regular_past(contained,contain).
regular_past(drained,drain).
regular_past(exceeded,exceed).
regular_past(flowed,flow).

rel_pron(who,subj).
rel_pron(whom,compl).
rel_pron(which,undef).

poss_pron(my,_,1,sin).
poss_pron(your,_,2,_).
poss_pron(his,masc,3,sin).
poss_pron(her,fem,3,sin).
poss_pron(its,neut,3,sin).
poss_pron(our,_,1,plu).
poss_pron(their,_,3,plu).

pers_pron(i,_,1,sin,subj).
pers_pron(you,_,2,_,_).
pers_pron(he,masc,3,sin,subj).
pers_pron(she,fem,3,sin,subj).
pers_pron(it,neut,3,sin,_).
pers_pron(we,_,1,plu,subj).
pers_pron(them,_,3,plu,subj).
pers_pron(me,_,1,sin,compl(_)).
pers_pron(him,masc,3,sin,compl(_)).
pers_pron(her,fem,3,sin,compl(_)).
pers_pron(us,_,1,plu,compl(_)).
pers_pron(them,_,3,plu,compl(_)).

terminator(.,_).
terminator(?,?).
terminator(!,!).

name(_).
```

```
% ==========================================================================

% specialised dictionary

loc_pred(east,prep(eastof)).
loc_pred(west,prep(westof)).
loc_pred(north,prep(northof)).
loc_pred(south,prep(southof)).

adj(minimum,restr).
adj(maximum,restr).
adj(average,restr).
adj(total,restr).
adj(african,restr).
adj(american,restr).
adj(asian,restr).
adj(european,restr).
adj(great,quant).
adj(big,quant).
adj(small,quant).
adj(large,quant).
adj(old,quant).
adj(new,quant).
adj(populous,quant).

rel_adj(greater,great).
rel_adj(less,small).
rel_adj(bigger,big).
rel_adj(smaller,small).
rel_adj(larger,large).
rel_adj(older,old).
rel_adj(newer,new).

sup_adj(biggest,big).
sup_adj(smallest,small).
sup_adj(largest,large).
sup_adj(oldest,old).
sup_adj(newest,new).

noun_sin(average).
noun_sin(total).
noun_sin(sum).
noun_sin(degree).
noun_sin(sqmile).
noun_sin(ksqmile).
noun_sin(thousand).
noun_sin(million).
noun_sin(time).
noun_sin(place).
noun_sin(area).
noun_sin(capital).
noun_sin(city).
noun_sin(continent).
noun_sin(country).
noun_sin(latitude).
noun_sin(longitude).
noun_sin(ocean).
noun_sin(person).
noun_sin(population).
noun_sin(region).
noun_sin(river).
noun_sin(sea).
noun_sin(seamass).
noun_sin(number).
```

```
noun_plu(averages,average).
noun_plu(totals,total).
noun_plu(sums,sum).
noun_plu(degrees,degree).
noun_plu(sqmiles,sqmile).
noun_plu(ksqmiles,ksqmile).
noun_plu(million,million).
noun_plu(thousand,thousand).
noun_plu(times,time).
noun_plu(places,place).
noun_plu(areas,area).
noun_plu(capitals,capital).
noun_plu(cities,city).
noun_plu(continents,continent).
noun_plu(countries,country).
noun_plu(latitudes,latitude).
noun_plu(longitudes,longitude).
noun_plu(oceans,ocean).
noun_plu(persons,person).  noun_plu(people,person).
noun_plu(populations,population).
noun_plu(regions,region).
noun_plu(rivers,river).
noun_plu(seas,sea).
noun_plu(seamasses,seamass).
noun_plu(numbers,number).

verb_form(V,V,inf,_) :- verb_root(V).
verb_form(V,V,pres+fin,Agmt) :-
   regular_pres(V),
   root_form(Agmt),
   verb_root(V).
verb_form(Past,Root,past+_,_) :-
   regular_past(Past,Root).

verb_form(am,be,pres+fin,1+sin).
verb_form(are,be,pres+fin,2+sin).
verb_form(is,be,pres+fin,3+sin).
verb_form(are,be,pres+fin,_+plu).
verb_form(was,be,past+fin,1+sin).
verb_form(were,be,past+fin,2+sin).
verb_form(was,be,past+fin,3+sin).
verb_form(were,be,past+fin,_+plu).
verb_form(been,be,past+part,_).
verb_form(being,be,pres+part,_).
verb_form(has,have,pres+fin,3+sin).
verb_form(having,have,pres+part,_).
verb_form(does,do,pres+fin,3+sin).
verb_form(did,do,past+fin,_).
verb_form(doing,do,pres+part,_).
verb_form(done,do,past+part,_).
verb_form(flows,flow,pres+fin,3+sin).
verb_form(flowing,flow,pres+part,_).
verb_form(rises,rise,pres+fin,3+sin).
verb_form(rose,rise,past+fin,_).
verb_form(risen,rise,past+part,_).
verb_form(borders,border,pres+fin,3+sin).
verb_form(bordering,border,pres+part,_).
verb_form(contains,contain,pres+fin,3+sin).
verb_form(containing,contain,pres+part,_).
verb_form(drains,drain,pres+fin,3+sin).
verb_form(draining,drain,pres+part,_).
verb_form(exceeds,exceed,pres+fin,3+sin).
verb_form(exceeding,exceed,pres+part,_).
```

```
verb_type(have,aux+have).
verb_type(be,aux+be).
verb_type(do,aux+ditrans).
verb_type(rise,main+intrans).
verb_type(border,main+trans).
verb_type(contain,main+trans).
verb_type(drain,main+intrans).
verb_type(exceed,main+trans).
verb_type(flow,main+intrans).

adverb(yesterday).
adverb(tomorrow).
```

```
# /*
  set-up.chat_parser: bench set-up for chat_parser
  */
chat_parser :- driver(chat_parser).

benchmark(chat_parser, run_chat_parser, run_dummy, 10).

run_chat_parser :- string(X),
                   determinate_say(X,_),
                   fail.
run_chat_parser.

run_dummy :- string(X),
             dummy(X,_),
             fail.
run_dummy.

show(chat_parser) :- string(X),
                     write(X), nl,
                     determinate_say(X,Y),
                     write(Y), nl, nl,
                     fail.
show(chat_parser).


#include "driver"
```

fft

fft_4.m

```
# /*
  fft_4.m: benchmark (fft) fft_4 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   fft_4
%
%   Richard A. O'Keefe
%
%   FFT (fast fourier transform) of f(x) = x on 2^4 points

#if BENCH
#  include ".fft_4.bench"
#else
fft_4 :- numlist(1, 16, Raw),    % 16 is 2^4
         fwd_fft(Raw, _).
#endif

numlist(I, N, []) :-
        I > N, !.
numlist(I, N, [I|L]) :-
        I =< N, J is I+1,
        numlist(J, N, L).

#include "fft"          /* code for N-point FFT */
```

```
# /*
  fft_8.m: benchmark (fft) fft_8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   fft_8
%
%   Richard A. O'Keefe
%
%   FFT (fast fourier transform) of f(x) = x on 2^8 points

#if BENCH
#  include ".fft_8.bench"
#else
fft_8 :- numlist(1, 256, Raw),  % 256 is 2^8
         fwd_fft(Raw, _).
#endif

numlist(I, N, []) :-
        I > N, !.
numlist(I, N, [I|L]) :-
        I =< N, J is I+1,
        numlist(J, N, L).

#include "fft"          /* code for N-point FFT */
```

```
# /*
  fft: code for N-point FFT
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (fwd_fft/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
fwd_fft(_, _).
#else
%   This is a Prolog implementation of the Fast Fourier Transform:
%
%   F(k;N) = sum[j=0..N-1] exp(2.pi.i.j.k/N) . f(j)
%
%          = sum[j=0..N/2-1] exp(2.pi.i.k.(2j)/N)   . f(2j)
%          + sum[j=0..N/2-1] exp(2.pi.i.k.(2j+1)/N) . f(2j+1)
%
%          =       sum[j=0..N/2-1] exp(2.pi.i.j.k/(N/2)) . f(2j)
%          + W^k . sum[j=0..N/2-1] exp(2.pi.i.j.k/(N/2)) . f(2j+1)
%
%          [where W = exp(2.pi.i/N)]
%
%   F(k;I) = F(k;E) + exp(2.pi.i.k/length(I)) . F(k;O)
%
%          [where evens_and_odds(I, E, O)]
%
%   It stresses floating-point arithmetic.  Note that the foreign
%   function interface problem is avoided by using a table of the
%   necessary sines and cosines.

fwd_fft(Raw, FFT) :-
        length(Raw, N),
        fft(N, Raw, FFT, fwd).

inv_fft(FFT, Raw) :-
        length(FFT, N),
        fft(N, FFT, Mid, inv),
        scale(Mid, N, Raw).

fft(1, [X], [C], _) :- !,
        complex_val(X, C).
fft(N, Raw, FFT, Dir) :-
        n_cos_sin(N, Cos, Sin),
        pack_w(Dir, Cos, Sin, W),
        M is N>>1,
        evens_and_odds(Raw, E, O),
        fft(M, E, Ef, Dir),
        fft(M, O, Of, Dir),
        fft(Ef, Of, W, (1.0,0.0), Z, FFT, FF2),
        fft(Ef, Of, W, Z, _, FF2, []).

pack_w(fwd, C, S, (C,S)).
pack_w(inv, C, S, (C,Z)) :- Z is -S.

fft([], [], _, Z, Z, F, F).
fft([E|Es], [O|Os], W, Z0, Z, [F|Fs], Fl) :-
        complex_mul(Z0, O, Zt),
        complex_add(Zt, E, F),
        complex_mul(Z0, W, Z1),
        fft(Es, Os, W, Z1, Z, Fs, Fl).
```

```
evens_and_odds([], [], []).
evens_and_odds([E,O|EOs], [E|Es], [O|Os]) :-
        evens_and_odds(EOs, Es, Os).

scale([], _, []).
scale([(Ra,Ia)|Xs], Scale, [(Rs,Is)|Ys]) :-
        Rs is Ra/Scale,
        Is is Ia/Scale,
        scale(Xs, Scale, Ys).

complex_val((Ra,Ia), (Rs,Is)) :- !,
        Rs is Ra*1.0,
        Is is Ia*1.0.
complex_val(Ra, (Rs,0.0)) :-
        Rs is Ra*1.0.

complex_add((Ra,Ia), (Rb,Ib), (Rs,Is)) :-
        Rs is Ra+Rb,
        Is is Ia+Ib.

complex_mul((Ra,Ia), (Rb,Ib), (Rs,Is)) :-
        Rs is Ra*Rb-Ia*Ib,
        Is is Ra*Ib+Rb*Ia.

%complex_exp(Ang, (Rs,Is)) :-
%       cos(Ang, Rs),
%       sin(Ang, Is).

% n_cos_sin(N, C, S) :- N is 2^K for K=1..23,
%                       C is cos(2.pi/N),
%                       S is sin(2.pi/N).

n_cos_sin(        2, -1.00000000,  0.00000000).
n_cos_sin(        4,  0.00000000,  1.00000000).
n_cos_sin(        8,  0.707106781,  0.707106781).
n_cos_sin(       16,  0.923879533,  0.382683432).
n_cos_sin(       32,  0.980785280,  0.195090322).
n_cos_sin(       64,  0.995184727,  0.0980171403).
n_cos_sin(      128,  0.998795456,  0.0490676743).
n_cos_sin(      256,  0.999698819,  0.0245412285).
n_cos_sin(      512,  0.999924702,  0.0122715383).
n_cos_sin(     1024,  0.999981175,  0.00613588465).
n_cos_sin(     2048,  0.999995294,  0.00306795676).
n_cos_sin(     4096,  0.999998823,  0.00153398019).
n_cos_sin(     8192,  0.999999706,  0.000766990319).
n_cos_sin(    16384,  0.999999926,  0.000383495188).
n_cos_sin(    32768,  0.999999982,  0.000191747597).
n_cos_sin(    65536,  0.999999995,  0.0000958737991).
n_cos_sin(   131072,  0.999999999,  0.0000479368996).
n_cos_sin(   262144,  1.00000000,   0.0000239684498).
n_cos_sin(   524288,  1.00000000,   0.0000119842249).
n_cos_sin(  1048576,  1.00000000,   0.00000599211245).
n_cos_sin(  2097152,  1.00000000,   0.00000299605623).
n_cos_sin(  4194304,  1.00000000,   0.00000149802811).
n_cos_sin(  8388608,  1.00000000,   0.000000749014057).
#endif
```

```
# /*
  set-up.fft_4: bench set-up for (fft) fft_4
  */
fft_4 :- driver(fft_4).

benchmark(fft_4, fwd_fft(Raw, _), dummy(Raw, _), 100) :- numlist(1, 16, Raw).
                                                         % 16 is 2^4

#message "NOTE: show/1 is NOT defined for (fft) fft_4"


#include "driver"
```

.fft_8.bench

```
# /*
  set-up.fft_8: bench set-up for (fft) fft_8
  */
fft_8 :- driver(fft_8).

benchmark(fft_8, fwd_fft(Raw, _), dummy(Raw, _), 5) :- numlist(1, 256, Raw).
                                                        % 256 is 2^8

#message "NOTE: show/1 is NOT defined for fft_8"


#include "driver"
```

gabriel

```
# /*
  boyer.m: Gabriel benchmark boyer master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   boyer
%
%   Evan Tick (from Lisp version by R. P. Gabriel)
%
%   November 1985
%
%   prove arithmetic theorem

#if BENCH
#  include ".boyer.bench"
#else
boyer :- run_boyer.
#endif


#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (run_boyer/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
run_boyer.
#halt
#endif
run_boyer :- wff(Wff),
             rewrite(Wff,NewWff),
             tautology(NewWff,[],[]).


wff(implies(and(implies(X,Y),
                and(implies(Y,Z),
                    and(implies(Z,U),
                        implies(U,W)))),
            implies(X,W))) :-
        X = f(plus(plus(a,b),plus(c,zero))),
        Y = f(times(times(a,b),plus(c,d))),
        Z = f(reverse(append(append(a,b),[]))),
        U = equal(plus(a,b),difference(x,y)),
        W = lessp(remainder(a,b),member(a,length(b))).

tautology(Wff) :-
        write('rewriting...'),nl,
        rewrite(Wff,NewWff),
        write('proving...'),nl,
        tautology(NewWff,[],[]).

tautology(Wff,Tlist,Flist) :-
        (truep(Wff,Tlist) -> true
        ;falsep(Wff,Flist) -> fail
        ;Wff = if(If,Then,Else) ->
                (truep(If,Tlist) -> tautology(Then,Tlist,Flist)
                ;falsep(If,Flist) -> tautology(Else,Tlist,Flist)
                ;tautology(Then,[If|Tlist],Flist),      % both must hold
                 tautology(Else,Tlist,[If|Flist])
                )
        ),!.
```

```
rewrite(Atom,Atom) :-
        atomic(Atom),!.
rewrite(Old,New) :-
        functor(Old,F,N),
        functor(Mid,F,N),
        rewrite_args(N,Old,Mid),
        ( equal(Mid,Next),          % should be ->, but is compiler smart
          rewrite(Next,New)         % enough to generate cut for -> ?
        ; New=Mid
        ),!.

rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        rewrite(OldArg,MidArg),
        N1 is N-1,
        rewrite_args(N1,Old,Mid).

truep(t,_) :- !.
truep(Wff,Tlist) :- member(Wff,Tlist).

falsep(f,_) :- !.
falsep(Wff,Flist) :- member(Wff,Flist).

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).


equal(  and(P,Q),
        if(P,if(Q,t,f),f)
        ).
equal(  append(append(X,Y),Z),
        append(X,append(Y,Z))
        ).
equal(  assignment(X,append(A,B)),
        if(assignedp(X,A),
           assignment(X,A),
           assignment(X,B))
        ).
equal(  assume_false(Var,Alist),
        cons(cons(Var,f),Alist)
        ).
equal(  assume_true(Var,Alist),
        cons(cons(Var,t),Alist)
        ).
equal(  boolean(X),
        or(equal(X,t),equal(X,f))
        ).
equal(  car(gopher(X)),
        if(listp(X),
        car(flatten(X)),
        zero)
        ).
equal(  compile(Form),
        reverse(codegen(optimize(Form),[]))
        ).
equal(  count_list(Z,sort_lp(X,Y)),
        plus(count_list(Z,X),
             count_list(Z,Y))
        ).
equal(  countps_(L,Pred),
        countps_loop(L,Pred,zero)
        ).
equal(  difference(A,B),
        C
        ) :- difference(A,B,C).
equal(  divides(X,Y),
        zerop(remainder(Y,X))
        ).
```

```
equal(  dsort(X),
        sort2(X)
        ).
equal(  eqp(X,Y),
        equal(fix(X),fix(Y))
        ).
equal(  equal(A,B),
        C
        ) :- eq(A,B,C).
equal(  even1(X),
        if(zerop(X),t,odd(decr(X)))
        ).
equal(  exec(append(X,Y),Pds,Envrn),
        exec(Y,exec(X,Pds,Envrn),Envrn)
        ).
equal(  exp(A,B),
        C
        ) :- exp(A,B,C).
equal(  fact_(I),
        fact_loop(I,1)
        ).
equal(  falsify(X),
        falsify1(normalize(X),[])
        ).
equal(  fix(X),
        if(numberp(X),X,zero)
        ).
equal(  flatten(cdr(gopher(X))),
        if(listp(X),
           cdr(flatten(X)),
           cons(zero,[]))
        ).
equal(  gcd(A,B),
        C
        ) :- gcd(A,B,C).
equal(  get(J,set(I,Val,Mem)),
        if(eqp(J,I),Val,get(J,Mem))
        ).
equal(  greatereqp(X,Y),
        not(lessp(X,Y))
        ).
equal(  greatereqpr(X,Y),
        not(lessp(X,Y))
        ).
equal(  greaterp(X,Y),
        lessp(Y,X)
        ).
equal(  if(if(A,B,C),D,E),
        if(A,if(B,D,E),if(C,D,E))
        ).
equal(  iff(X,Y),
        and(implies(X,Y),implies(Y,X))
        ).
equal(  implies(P,Q),
        if(P,if(Q,t,f),t)
        ).
equal(  last(append(A,B)),
        if(listp(B),
           last(B),
           if(listp(A),
              cons(car(last(A))),
              B))
        ).
equal(  length(A),
        B
        ) :- mylength(A,B).
equal(        lesseqp(X,Y),
        not(lessp(Y,X))
        ).
```

```
equal(  lessp(A,B),
        C
        ) :- lessp(A,B,C).
equal(  listp(gopher(X)),
        listp(X)
        ).
equal(  mc_flatten(X,Y),
        append(flatten(X),Y)
        ).
equal(  meaning(A,B),
        C
        ) :- meaning(A,B,C).
equal(  member(A,B),
        C
        ) :- mymember(A,B,C).
equal(  not(P),
        if(P,f,t)
        ).
equal(  nth(A,B),
        C
        ) :- nth(A,B,C).
equal(  numberp(greatest_factor(X,Y)),
        not(and(or(zerop(Y),equal(Y,1)),
                not(numberp(X))))
        ).
equal(  or(P,Q),
        if(P,t,if(Q,t,f),f)
        ).
equal(  plus(A,B),
        C
        ) :- plus(A,B,C).
equal(  power_eval(A,B),
        C
        ) :- power_eval(A,B,C).
equal(  prime(X),
        and(not(zerop(X)),
            and(not(equal(X,add1(zero))),
                prime1(X,decr(X))))
        ).
equal(  prime_list(append(X,Y)),
        and(prime_list(X),prime_list(Y))
        ).
equal(  quotient(A,B),
        C
        ) :- quotient(A,B,C).
equal(  remainder(A,B),
        C
        ) :- remainder(A,B,C).
equal(  reverse_(X),
        reverse_loop(X,[])
        ).
equal(  reverse(append(A,B)),
        append(reverse(B),reverse(A))
        ).
equal(  reverse_loop(A,B),
        C
        ) :- reverse_loop(A,B,C).
equal(  samefringe(X,Y),
        equal(flatten(X),flatten(Y))
        ).
equal(  sigma(zero,I),
        quotient(times(I,add1(I)),2)
        ).
equal(  sort2(delete(X,L)),
        delete(X,sort2(L))
        ).
equal(  tautology_checker(X),
        tautologyp(normalize(X),[])
        ).
```

```
equal(  times(A,B),
        C
        ) :- times(A,B,C).
equal(  times_list(append(X,Y)),
        times(times_list(X),times_list(Y))
        ).
equal(  value(normalize(X),A),
        value(X,A)
        ).
equal(  zerop(X),
        or(equal(X,zero),not(numberp(X)))
        ).

difference(X, X, zero) :- !.
difference(plus(X,Y), X, fix(Y)) :- !.
difference(plus(Y,X), X, fix(Y)) :- !.
difference(plus(X,Y), plus(X,Z), difference(Y,Z)) :- !.
difference(plus(B,plus(A,C)), A, plus(B,C)) :- !.
difference(add1(plus(Y,Z)), Z, add1(Y)) :- !.
difference(add1(add1(X)), 2, fix(X)).

eq(plus(A,B), zero, and(zerop(A),zerop(B))) :- !.
eq(plus(A,B), plus(A,C), equal(fix(B),fix(C))) :- !.
eq(zero, difference(X,Y),not(lessp(Y,X))) :- !.
eq(X, difference(X,Y),and(numberp(X),
                          and(or(equal(X,zero),
                                 zerop(Y))))) :- !.
eq(times(X,Y), zero, or(zerop(X),zerop(Y))) :- !.
eq(append(A,B), append(A,C), equal(B,C)) :- !.
eq(flatten(X), cons(Y,[]), and(nlistp(X),equal(X,Y))) :- !.
eq(greatest_factor(X,Y),zero, and(or(zerop(Y),equal(Y,1)),
                                     equal(X,zero))) :- !.
eq(greatest_factor(X,_),1, equal(X,1)) :- !.
eq(Z, times(W,Z), and(numberp(Z),
                      or(equal(Z,zero),
                         equal(W,1)))) :- !.
eq(X, times(X,Y), or(equal(X,zero),
                     and(numberp(X),equal(Y,1)))) :- !.
eq(times(A,B), 1, and(not(equal(A,zero)),
                      and(not(equal(B,zero)),
                          and(numberp(A),
                              and(numberp(B),
                                  and(equal(decr(A),zero),
                                      equal(decr(B),zero))))))) :- !.
eq(difference(X,Y), difference(Z,Y),if(lessp(X,Y),
                                       not(lessp(Y,Z)),
                                       if(lessp(Z,Y),
                                          not(lessp(Y,X)),
                                          equal(fix(X),fix(Z))))) :- !.
eq(lessp(X,Y), Z, if(lessp(X,Y),
                     equal(t,Z),
                     equal(f,Z))).

exp(I, plus(J,K), times(exp(I,J),exp(I,K))) :- !.
exp(I, times(J,K), exp(exp(I,J),K)).

gcd(X, Y, gcd(Y,X)) :- !.
gcd(times(X,Z), times(Y,Z), times(Z,gcd(X,Y))).

mylength(reverse(X),length(X)).
mylength(cons(_,cons(_,cons(_,cons(_,cons(_,cons(_,X7)))))),
         plus(6,length(X7))).

lessp(remainder(_,Y), Y, not(zerop(Y))) :- !.
lessp(quotient(I,J), I, and(not(zerop(I)),
                            or(zerop(J),
                               not(equal(J,1))))) :- !.
```

```
lessp(remainder(X,Y), X, and(not(zerop(Y)),
                             and(not(zerop(X)),
                                 not(lessp(X,Y))))) :- !.
lessp(plus(X,Y), plus(X,Z), lessp(Y,Z)) :- !.
lessp(times(X,Z), times(Y,Z), and(not(zerop(Z)),
                                   lessp(X,Y))) :- !.
lessp(Y, plus(X,Y), not(zerop(X))) :- !.
lessp(length(delete(X,L)), length(L), member(X,L)).

meaning(plus_tree(append(X,Y)),A,
        plus(meaning(plus_tree(X),A),
             meaning(plus_tree(Y),A))) :- !.
meaning(plus_tree(plus_fringe(X)),A,
        fix(meaning(X,A))) :- !.
meaning(plus_tree(delete(X,Y)),A,
        if(member(X,Y),
           difference(meaning(plus_tree(Y),A),
                      meaning(X,A)),
           meaning(plus_tree(Y),A))).

mymember(X,append(A,B),or(member(X,A),member(X,B))) :- !.
mymember(X,reverse(Y),member(X,Y)) :- !.
mymember(A,intersect(B,C),and(member(A,B),member(A,C))).

nth(zero,_,zero).
nth([],I,if(zerop(I),[],zero)).
nth(append(A,B),I,append(nth(A,I),nth(B,difference(I,length(A))))).

plus(plus(X,Y),Z,
     plus(X,plus(Y,Z))) :- !.
plus(remainder(X,Y),
     times(Y,quotient(X,Y)),
     fix(X)) :- !.
plus(X,add1(Y),
     if(numberp(Y),
        add1(plus(X,Y)),
        add1(X))).

power_eval(big_plus1(L,I,Base),Base,
           plus(power_eval(L,Base),I)) :- !.
power_eval(power_rep(I,Base),Base,
           fix(I)) :- !.
power_eval(big_plus(X,Y,I,Base),Base,
           plus(I,plus(power_eval(X,Base),
                       power_eval(Y,Base)))) :- !.
power_eval(big_plus(power_rep(I,Base),
                    power_rep(J,Base),
                    zero,
                    Base),
           Base,
           plus(I,J)).

quotient(plus(X,plus(X,Y)),2,plus(X,quotient(Y,2))).
quotient(times(Y,X),Y,if(zerop(Y),zero,fix(X))).

remainder(_,         1,zero) :- !.
remainder(X,         X,zero) :- !.
remainder(times(_,Z),Z,zero) :- !.
remainder(times(Y,_),Y,zero).

reverse_loop(X,Y,  append(reverse(X),Y)) :- !.
reverse_loop(X,[], reverse(X)          ).

times(X,         plus(Y,Z),       plus(times(X,Y),times(X,Z))       ) :- !.
times(times(X,Y),Z,               times(X,times(Y,Z))               ) :- !.
times(X,         difference(C,W),difference(times(C,X),times(W,X))) :- !.
times(X,         add1(Y),         if(numberp(Y),
                                     plus(X,times(X,Y)),
                                     fix(X))                        ).
```

```
# /*
  browse.m: Gabriel benchmark browse master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   browse
%
%   Tep Dobry (from Lisp version by R. P. Gabriel)
%
%   (modified January 1987 by Herve' Touati)

#if BENCH
#  include ".browse.bench"
#else
browse :- run_browse.
#endif


#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (run_browse/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
run_browse.
#halt
#endif
run_browse :-
    init(100,10,4,
         [[a,a,a,b,b,b,b,a,a,a,a,a,b,b,a,a,a],
          [a,a,b,b,b,b,a,a,[a,a],[b,b]],
          [a,a,a,b,[b,a],b,a,b,a]
         ],
         Symbols),
    randomize(Symbols,RSymbols,21),!,
    investigate(RSymbols,
                [[star(SA),B,star(SB),B,a,star(SA),a,star(SB),star(SA)],
                 [star(SA),star(SB),star(SB),star(SA),[star(SA)],[star(SB)]],
                 [_,_,star(_),[b,a],star(_),_,_]
                ]).


init(N,M,Npats,Ipats,Result) :- init(N,M,M,Npats,Ipats,Result).

init(0,_,_,_,_,_) :- !.
init(N,I,M,Npats,Ipats,[Symb|Rest]) :-
    fill(I,[],L),
    get_pats(Npats,Ipats,Ppats),
    J is M - I,
    fill(J,[pattern(Ppats)|L],Symb),
    N1 is N - 1,
    (I =:= 0 -> I1 is M; I1 is I - 1),
    init(N1,I1,M,Npats,Ipats,Rest).

fill(0,L,L) :- !.
fill(N,L,[dummy([])|Rest]) :-
    N1 is N - 1,
    fill(N1,L,Rest).
```

```
randomize([],[],_) :- !.
randomize(In,[X|Out],Rand) :-
    length(In,Lin),
    Rand1 is (Rand * 17) mod 251,
    N is Rand1 mod Lin,
    split(N,In,X,In1),
    randomize(In1,Out,Rand1).

split(0,[X|Xs],X,Xs) :- !.
split(N,[X|Xs],RemovedElt,[X|Ys]) :-
    N1 is N - 1,
    split(N1,Xs,RemovedElt,Ys).

investigate([],_) :- !.
investigate([U|Units],Patterns) :-
    property(U,pattern,Data),
    p_investigate(Data,Patterns),
    investigate(Units,Patterns).

get_pats(Npats,Ipats,Result) :- get_pats(Npats,Ipats,Result,Ipats).

get_pats(0,_,[],_) :- !.
get_pats(N,[X|Xs],[X|Ys],Ipats) :-
    N1 is N - 1,
    get_pats(N1,Xs,Ys,Ipats).
get_pats(N,[],Ys,Ipats) :-
    get_pats(N,Ipats,Ys,Ipats).

property([],_,_) :- fail.       /* don't really need this */
property([Prop|_],P,Val) :-
    functor(Prop,P,_),!,
    arg(1,Prop,Val).
property([_|RProps],P,Val) :-
    property(RProps,P,Val).

p_investigate([],_).
p_investigate([D|Data],Patterns) :-
    p_match(Patterns,D),
    p_investigate(Data,Patterns).

p_match([],_).
p_match([P|Patterns],D) :-
    (match(D,P),fail; true),
    p_match(Patterns,D).

match([],[]) :- !.
match([X|PRest],[Y|SRest]) :-
    var(Y),!,X = Y,
    match(PRest,SRest).
match(List,[Y|Rest]) :-
    nonvar(Y),Y = star(X),!,
    concat(X,SRest,List),
    match(SRest,Rest).
match([X|PRest],[Y|SRest]) :-
    (atom(X) -> X = Y; match(X,Y)),
    match(PRest,SRest).

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

poly_5.m

```
# /*
  poly_5.m: Gabriel benchmark (frpoly) poly_5 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (frpoly) poly_5
%
%   Rick McGeer (from Lisp version by R. P. Gabriel)
%
%   raise a polynomial (1+x+y+z) to the 5th power (symbolically)

#if BENCH
#  include ".poly_5.bench"
#else
poly_5 :- test_poly(P), run_frpoly(5, P), !.
#endif


% test polynomial definition

test_poly(P) :-
    poly_add(poly(y,[term(1,1)]),poly(x,[term(0,1),term(1,1)]),Q),
    poly_add(poly(z,[term(1,1)]),Q,P).


#include "frpoly"       /* code for symbolic polynomial exponentiation */
```

```
# /*
  poly_10.m: Gabriel benchmark (frpoly) poly_10 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (frpoly) poly_10
%
%   Rick McGeer (from Lisp version by R. P. Gabriel)
%
%   raise a polynomial (1+x+y+z) to the 10th power (symbolically)

#if BENCH
#  include ".poly_10.bench"
#else
poly_10 :- test_poly(P), run_frpoly(10, P), !.
#endif


% test polynomial definition

test_poly(P) :-
    poly_add(poly(y,[term(1,1)]),poly(x,[term(0,1),term(1,1)]),Q),
    poly_add(poly(z,[term(1,1)]),Q,P).


#include "frpoly"       /* code for symbolic polynomial exponentiation */
```

poly_15.m

```
# /*
  poly_15.m: Gabriel benchmark (frpoly) poly_15 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (frpoly) poly_15
%
%   Rick McGeer (from Lisp version by R. P. Gabriel)
%
%   raise a polynomial (1+x+y+z) to the 15th power (symbolically)

#if BENCH
#  include ".poly_15.bench"
#else
poly_15 :- test_poly(P), run_frpoly(15, P), !.
#endif


% test polynomial definition

test_poly(P) :-
    poly_add(poly(y,[term(1,1)]),poly(x,[term(0,1),term(1,1)]),Q),
    poly_add(poly(z,[term(1,1)]),Q,P).


#include "frpoly"       /* code for symbolic polynomial exponentiation */
```

```
# /*
  frpoly: Gabriel code for symbolic polynomial exponentiation
  */
% polynomial addition

poly_add( poly(Var, Terms1), poly(Var, Terms2), poly(Var, Terms3) ) :-
    !,
    add_terms(Terms1, Terms2, Terms3).

poly_add( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var1, Terms3) ) :-
    Var2 @> Var1,
    !,
    add_To_Zero_Term(Terms1, poly(Var2, Terms2), Terms3 ).

poly_add( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var2, Terms3) ) :-
    Var1 @> Var2,
    !,
    add_To_Zero_Term(Terms2, poly(Var1, Terms1), Terms3 ).

poly_add( poly(Var1, Terms1), N, poly(Var1, Terms3)) :-
    !,
    add_To_Zero_Term(Terms1, N, Terms3 ).

poly_add( N, poly(Var2, Terms2), poly(Var2, Terms3)) :-
    !,
    add_To_Zero_Term(Terms2, N, Terms3 ).

% plain numerical addition

poly_add(N, M, T) :-
    T is N + M.

% term addition

add_terms([],X,X) :- !.

add_terms(X,[],X) :- !.

add_terms([term(Exp,C1)|Terms1],[term(Exp,C2)|Terms2],[term(Exp,C)|Terms]) :-
    !,
    poly_add(C1, C2, C),
    add_terms(Terms1, Terms2, Terms).

add_terms([term(E1,C1)|Terms1],[term(E2,C2)|Terms2],[term(E1,C1)|Terms]) :-
    E1 < E2,
    !,
    add_terms(Terms1,[term(E2,C2)|Terms2],Terms).

add_terms(Terms1,[term(E2,C2)|Terms2],[term(E2,C2)|Terms]) :-
    add_terms(Terms1, Terms2, Terms).

add_To_Zero_Term([term(0,C1)|Terms],C2,[term(0,C)|Terms]) :-
    !,
    poly_add(C1, C2, C).

add_To_Zero_Term(Terms,C,[term(0,C)|Terms]).
```

frpoly

```
% run_frpoly definition

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (run_frpoly/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
run_frpoly(_, _).
#halt
#endif
run_frpoly(N, P) :-
    poly_expt(N, P, _).


% polynomial multiplication

poly_mult( poly(Var, Terms1), poly(Var, Terms2), poly(Var, Terms3)) :-
    form_poly_product(Terms1, Terms2, Terms3).

poly_mult( poly(Var1, Terms1), poly(Var2, Terms2), poly(Var1, Terms3) ) :-
    Var2 @> Var1,
    !,
    multiply_through(Terms1, poly(Var2, Terms2), Terms3 ).

poly_mult( Poly1, poly( Var2, Terms2), poly(Var2, Terms3) ) :-
    !,
    multiply_through(Terms2, Poly1, Terms3 ).

poly_mult( poly( Var2, Terms2), Poly1, poly(Var2, Terms3) ) :-
    !,
    multiply_through(Terms2, Poly1, Terms3 ).

poly_mult(C1, C2, C) :-
    C is C1 * C2.

multiply_through([], _, []) :- !.

multiply_through([term(N,T1)|Terms], Poly, [term(N,NewT1)|NewTerms]) :-
    poly_mult(T1, Poly, NewT1),
    multiply_through(Terms, Poly, NewTerms).

form_poly_product([],_,[]) :- !.

form_poly_product(_,[],[]) :- !.

form_poly_product([T1|Terms], Terms2, Terms3) :-
    form_single_product(Terms2, T1, Ta),
    form_poly_product(Terms, Terms2, Tb),
    add_terms(Ta, Tb, Terms3).

form_single_product([],_,[]) :- !.

form_single_product([term(Exp1,C1)|Terms],
                    term(Exp2,C2),
                    [term(Exp,C)|Products]) :-
    Exp is Exp1 + Exp2,
    poly_mult(C1, C2, C),
    form_single_product(Terms, term(Exp2,C2), Products).
```

```
% polynomial exponentiation

poly_expt(0, _, 1) :- !.

poly_expt(N, P, Result) :-
    evenP(N),
    !,
    M is N // 2,
    poly_expt(M, P, NextRes),
    poly_mult(NextRes, NextRes, Result).

poly_expt(N, P, Result) :-
    M is N - 1,
    poly_expt(M, P, NextRes),
    poly_mult(P, NextRes, Result).


%poly_expt(N, P, Result) :-
%   poly_expt( N, P, 1, Result).
%
%poly_expt(0, _, Result, Result) :- !.
%
%poly_expt(N, P, ResSoFar, Result) :-
%    evenP(N),
%    !,
%    M is N // 2,
%    poly_mult(ResSoFar, ResSoFar, NextRes),
%    poly_expt(M, P, NextRes, Result).
%
%poly_expt(N, P, ResSoFar, Result) :-
%    M is N - 1,
%    poly_mult(P, ResSoFar, NextRes),
%    poly_expt(M, P, NextRes, Result).


evenP(X) :-
    N is X // 2,
    X is N * 2.
```

```
# /*

% polynomial writing

print_poly(poly(Var, Terms)) :-
    !,
    print_Terms(Terms, Var).

print_poly(X) :-
    write(X).

print_Terms([],_) :- !.

print_Terms([term(_, 0)|Terms],Var) :-
    !,
    print_Terms(Terms, Var).

print_Terms([Term],Var) :-
    !,
    print_Term(Term, Var).

print_Terms([Term|Terms],Var) :-
    print_Term(Term, Var),
    write(' + '),
    print_Terms(Terms, Var).

print_Term(term(0, P), _) :-
    !,
    print_poly(P).

print_Term(term(1, C), Var) :-
    !,
    print_Coeff(C),
    write(Var).

print_Term(term(Exp,C), Var) :-
    print_Coeff(C),
    write(Var),
    write('^'),
    write(Exp).

print_Coeff(1) :- !.

print_Coeff(N) :-
    atomic(N),
    !,
    write(N),
    write('*').

print_Coeff(P) :-
    !,
    write('('),
    print_poly(P),
    write(')'),
    write('*').
  */
```

```
# /*
  puzzle.m: Gabriel benchmark puzzle master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   puzzle
%
%   Evan Tick (from Lisp version by R. P. Gabriel)

#if BENCH
#  include ".puzzle.bench"
#else
#option SHOW "
         > Option SHOW introduces code which writes output
         > to show what the benchmark does.  This may help
         > verify that the benchmark operates correctly.
         >
         > SHOW has no effect when BENCH is selected.  The
         > functionality of SHOW is then available through
         > show/1."
#  if SHOW
puzzle :- make_board(Board),
          initialize(Board,Pieces),
          play(Board,Pieces), !,
          access(0, N),
          write('success in '), write(N), write(' trials'), nl.

#  else
puzzle :- make_board(Board),
          initialize(Board,Pieces),
          play(Board,Pieces), !.

#  endif
#endif

make_board(Level0) :-
        make_level(Level0-Level1,Level1-_),
        make_level(Level1-Level2,Level2-_),
        make_level(Level2-Level3,Level3-_),
        make_level(Level3-Level4,Level4-_),
        make_level(Level4-[],X-[]),
        X = [z,z,z,z,z, z,z,z,z,z, z,z,z,z,z, z,z,z,z,z, z,z,z,z,z].

make_level(C-Link,Z-L) :-
        C = [C00,C10,C20,C30,C40,
                C01,C11,C21,C31,C41,
                C02,C12,C22,C32,C42,
                C03,C13,C23,C33,C43,
                C04,C14,C24,C34,C44|Link],
        Z = [Z00,Z10,Z20,Z30,Z40,
                Z01,Z11,Z21,Z31,Z41,
                Z02,Z12,Z22,Z32,Z42,
                Z03,Z13,Z23,Z33,Z43,
                Z04,Z14,Z24,Z34,Z44|L],

        C00 = s(_,C10,C01,Z00),
        C10 = s(_,C20,C11,Z10),
        C20 = s(_,C30,C21,Z20),
        C30 = s(_,C40,C31,Z30),
        C40 = s(_,  z,C41,Z40),

        C01 = s(_,C11,C02,Z01),
        C11 = s(_,C21,C12,Z11),
        C21 = s(_,C31,C22,Z21),
        C31 = s(_,C41,C32,Z31),
        C41 = s(_,  z,C42,Z41),
        C02 = s(_,C12,C03,Z02),
        C12 = s(_,C22,C13,Z12),
        C22 = s(_,C32,C23,Z22),
        C32 = s(_,C42,C33,Z32),
        C42 = s(_,  z,C43,Z42),

        C03 = s(_,C13,C04,Z03),
        C13 = s(_,C23,C14,Z13),
        C23 = s(_,C33,C24,Z23),
        C33 = s(_,C43,C34,Z33),
        C43 = s(_,  z,C44,Z43),

        C04 = s(_,C14,  z,Z04),
        C14 = s(_,C24,  z,Z14),
        C24 = s(_,C34,  z,Z24),
        C34 = s(_,C44,  z,Z34),
        C44 = s(_,  z,  z,Z44).


initialize([Spot|_],[[b,c,d,e,f,g,h,i,j,k,l,m],[n,o,p],[q],[r]]) :-
        set(0,0),
        pl(a,Spot).

#option "
        > puzzle uses set/2 and access/2.  If one of
        >
        > C_PL QUINTUS_PL
        >
        > is selected, then set/1 and access/1 are defined using
        > assert/1 and retract/1.  If the Prolog system does not
        > offer set/1 and access/1 (as built-ins) but does offer
        > assert/1 and retract/1, then you may add an option for
        > the Prolog system to the list above."
#if C_PL || QUINTUS_PL
set(N, A) :-
        (retract('$set'(N, _)); true),
        assert('$set'(N, A)), !.
access(N, A) :-
        '$set'(N,A), !.

#else
#  message "WARNING: set/2 and access/2 must be defined."
#endif
% 4-2-1
pl(M,s(M,s(M,s(M,s(M,_,C13,_),C12,_),C11,_),s(M,C11,_,_),_)) :-
        C13 = s(M,_,_,_),
        C12 = s(M,C13,_,_),
        C11 = s(M,C12,_,_).
% 2-1-4
pl(M,s(M,s(M,_,_,C11),_,s(M,C11,_,s(M,C12,_,s(M,C13,_,_))))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,_,C13),
        C11 = s(M,_,_,C12).
% 1-4-2
pl(M,s(M,_,s(M,_,s(M,_,s(M,_,_,C13),C12),C11),s(M,_,C11,_))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,C13,_),
        C11 = s(M,_,C12,_).
% 2-4-1
pl(M,s(M,s(M,_,C11,_),s(M,C11,s(M,C12,s(M,C13,_,_),_),_),_)) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,C13,_),
        C11 = s(M,_,C12,_).
% 4-1-2
pl(M,s(M,s(M,s(M,s(M,_,_,C13),_,C12),_,C11),_,s(M,C11,_,_))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,C13,_,_),
        C11 = s(M,C12,_,_).
```

```
% 1-2-4
p1(M,s(M,_,s(M,_,_,C11),s(M,_,C11,s(M,_,C12,s(M,_,C13,_))))) :-
        C13 = s(M,_,_,_),
        C12 = s(M,_,_,C13),
        C11 = s(M,_,_,C12).


#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (play/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
play(_,_).
#halt
#endif
play(Board,Pieces) :- play(Board,Pieces,Board).

play([],_,_).
play([s(V,_,_,_)|Rest],Pieces,Board) :-
        nonvar(V), !,
        play(Rest,Pieces,Board).
play([Spot|Rest],Pieces,Board) :-
        fill(Spot,Pieces,NewPieces),
        incr,
        play(Rest,NewPieces,Board).

incr :- access(0, Count),
        NCount is Count + 1,
        set(0, NCount).

fill(Spot,[[Mark|P1]|T],[P1|T]) :- p1(Mark,Spot).
fill(Spot,[P1,[Mark|P2]|T],[P1,P2|T]) :- p2(Mark,Spot).
fill(Spot,[P1,P2,[Mark|P3]|T],[P1,P2,P3|T]) :- p3(Mark,Spot).
fill(Spot,[P1,P2,P3,[Mark|P4]|T],[P1,P2,P3,P4|T]) :- p4(Mark,Spot).

p2(M,s(M,s(M,s(M,_,_,_),_,_),_,_)).
p2(M,s(M,_,s(M,_,s(M,_,_,_),_),_)).
p2(M,s(M,_,_,s(M,_,_,s(M,_,_,_)))).

p3(M,s(M,s(M,_,C,_),s(M,C,_,_),_)) :-
        C = s(M,_,_,_).
p3(M,s(M,s(M,_,_,C),_,s(M,C,_,_))) :-
        C = s(M,_,_,_).
p3(M,s(M,_,s(M,_,_,C),s(M,_,C,_))) :-
        C = s(M,_,_,_).

p4(M,s(M,s(M,_,C110,C101),s(M,C110,_,s(M,C111,_,_)),s(M,C101,C011,_))) :-
        C110 = s(M,_,_,C111),
        C101 = s(M,_,C111,_),
        C011 = s(M,C111,_,_),
        C111 = s(M,_,_,_).
```

```
# /*
  tak.m: Gabriel benchmark tak master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   tak
%
%   Evan Tick (from Lisp version by R. P. Gabriel)
%
%   (almost) Takeuchi function (recursive arithmetic)

#if BENCH
#  include ".tak.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
tak :- tak(18,12,6,A), write('tak(18 12 6) = '), write(A), nl.
#  else
tak :- tak(18,12,6,_).
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (tak/4).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
tak(_,_,_,_).
#halt
#endif
tak(X,Y,Z,A) :-
        X =< Y, !,
        Z = A.
tak(X,Y,Z,A) :-
        X1 is X - 1,
        tak(X1,Y,Z,A1),
        Y1 is Y - 1,
        tak(Y1,Z,X,A2),
        Z1 is Z - 1,
        tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).
```

.boyer.bench

```
# /*
  set-up.boyer: bench set-up for boyer
  */
boyer :- driver(boyer).

benchmark(boyer, run_boyer, dummy, 1).

#message "NOTE: show/1 is NOT defined for boyer"


#include "driver"
```

```
# /*
  set-up.browse: bench set-up for browse
  */
browse :- driver(browse).

benchmark(browse, run_browse, dummy, 1).

#message "NOTE: show/1 is NOT defined for browse"


#include "driver"
```

```
# /*
  set-up.poly_5: bench set-up for (frpoly) poly_5
  */
poly_5 :- driver(poly_5).

benchmark(poly_5, (run_frpoly(5, P), !), (dummy(5, P), !), 200) :-
    test_poly(P).

#message "NOTE: show/1 is NOT defined for (frpoly) poly_5"


#include "driver"
```

.poly_10.bench

```
# /*
  set-up.poly_10: bench set-up for (frpoly) poly_10
  */
poly_10 :- driver(poly_10).

benchmark(poly_10, (run_frpoly(10, P), !), (dummy(10, P), !), 15) :-
    test_poly(P).

#message "NOTE: show/1 is NOT defined for (frpoly) poly_10"


#include "driver"
```

.poly_15.bench

```
# /*
  set-up.poly_15: bench set-up for (frpoly) poly_15
  */
poly_15 :- driver(poly_15).

benchmark(poly_15, (run_frpoly(15, P), !), (dummy(15, P), !), 3) :-
    test_poly(P).

#message "NOTE: show/1 is NOT defined for (frpoly) poly_15"


#include "driver"
```

.puzzle.bench

```
# /*
  set-up.puzzle: bench set-up for puzzle
  */
puzzle :- driver(puzzle).

benchmark(puzzle, (play(Board, Pieces), !), (dummy(Board, Pieces), !), 5) :-
        make_board(Board),
        initialize(Board, Pieces).

show(puzzle) :- make_board(Board),
                initialize(Board, Pieces),
                play(Board, Pieces), !,
                access(0, N),
                write('success in '), write(N), write(' trials'), nl.


#include "driver"
```

```
# /*
  set-up.tak: bench set-up for tak
  */
tak :- driver(tak).

benchmark(tak, tak(18,12,6,_), dummy(18,12,6,_), 10).

show(tak) :- tak(18,12,6,A), write('tak(18 12 6) = '), write(A), nl.


#include "driver"
```

[lisp] README

```
;;;; File   : Gabriel/Lisp/README
;;;; Updated: 8 February 1989

;;;; Mode: LISP; Package: COMMON-LISP-USER; Syntax: Common-Lisp

;;;; These files contain common lisp versions of the lisp performance
;;;; benchmarks from Stanford.  They were translated and tested using
;;;; Symbolics Common Lisp on a Symbolics 3600.  They have not been
;;;; "tuned" to any particular implementation.  There is no Common
;;;; Lisp timing function - these are highly system dependent.

;;;; See R. P. Gabriel, "Performance and Evaluation of Lisp Systems,"
;;;; MIT Press, Cambridge, Massachusetts, 1985.
```

```
;;; BOYER

;;; Logic programming benchmark, originally written by Bob Boyer.
;;; Fairly CONS intensive.

;;; run (setup), then call: (test)

(defvar unify-subst)
(defvar temp-temp)

(defun add-lemma (term)
  (cond ((and (not (atom term))
              (eq (car term)
                  (quote equal))
              (not (atom (cadr term))))
         (setf (get (car (cadr term)) (quote lemmas))
               (cons term (get (car (cadr term)) (quote lemmas)))))
        (t (error "~%ADD-LEMMA did not like term:  ~a" term))))

(defun add-lemma-lst (lst)
  (cond ((null lst)
         t)
        (t (add-lemma (car lst))
           (add-lemma-lst (cdr lst)))))

(defun apply-subst (alist term)
  (cond ((atom term)
         (cond ((setq temp-temp (assq term alist))
                (cdr temp-temp))
               (t term)))
        (t (cons (car term)
                 (apply-subst-lst alist (cdr term))))))

(defun apply-subst-lst (alist lst)
  (cond ((null lst)
         nil)
        (t (cons (apply-subst alist (car lst))
                 (apply-subst-lst alist (cdr lst))))))

(defun falsep (x lst)
  (or (equal x (quote (f)))
      (member x lst)))

(defun one-way-unify (term1 term2)
  (progn (setq unify-subst nil)
         (one-way-unify1 term1 term2)))

(defun one-way-unify1 (term1 term2)
  (cond ((atom term2)
         (cond ((setq temp-temp (assq term2 unify-subst))
                (equal term1 (cdr temp-temp)))
               (t (setq unify-subst (cons (cons term2 term1)
                                          unify-subst))
                  t)))
        ((atom term1)
         nil)
        ((eq (car term1)
             (car term2))
         (one-way-unify1-lst (cdr term1)
                             (cdr term2)))
        (t nil)))

(defun one-way-unify1-lst (lst1 lst2)
  (cond ((null lst1)
         t)
        ((one-way-unify1 (car lst1)
                         (car lst2))
         (one-way-unify1-lst (cdr lst1)
                             (cdr lst2)))
        (t nil)))
```

```
(defun rewrite (term)
  (cond ((atom term)
         term)
        (t (rewrite-with-lemmas (cons (car term)
                                      (rewrite-args (cdr term)))
                                (get (car term)
                                     (quote lemmas))))))

(defun rewrite-args (lst)
  (cond ((null lst)
         nil)
        (t (cons (rewrite (car lst))
                 (rewrite-args (cdr lst))))))

(defun rewrite-with-lemmas (term lst)
  (cond ((null lst)
         term)
        ((one-way-unify term (cadr (car lst)))
         (rewrite (apply-subst unify-subst (caddr (car lst)))))
        (t (rewrite-with-lemmas term (cdr lst)))))

(defun setup ()
  (add-lemma-lst
    (quote ((equal (compile form)
                   (reverse (codegen (optimize form)
                                     (nil))))
            (equal (eqp x y)
                   (equal (fix x)
                          (fix y)))
            (equal (greaterp x y)
                   (lessp y x))
            (equal (lesseqp x y)
                   (not (lessp y x)))
            (equal (greatereqp x y)
                   (not (lessp x y)))
            (equal (boolean x)
                   (or (equal x (t))
                       (equal x (f))))
            (equal (iff x y)
                   (and (implies x y)
                        (implies y x)))
            (equal (evenl x)
                   (if (zerop x)
                       (t)
                       (odd (1- x))))
            (equal (countps- l pred)
                   (countps-loop l pred (zero)))
            (equal (fact- i)
                   (fact-loop i 1))
            (equal (reverse- x)
                   (reverse-loop x (nil)))
            (equal (divides x y)
                   (zerop (remainder y x)))
            (equal (assume-true var alist)
                   (cons (cons var (t))
                         alist))
            (equal (assume-false var alist)
                   (cons (cons var (f))
                         alist))
            (equal (tautology-checker x)
                   (tautologyp (normalize x)
                               (nil)))
            (equal (falsify x)
                   (falsify1 (normalize x)
                             (nil)))
            (equal (prime x)
                   (and (not (zerop x))
                        (not (equal x (add1 (zero))))
                        (prime1 x (1- x))))
```

```
(equal (and p q)
       (if p (if q (t)
                   (f))
           (f)))
(equal (or p q)
       (if p (t)
           (if q (t)
                 (f))
           (f)))
(equal (not p)
       (if p (f)
           (t)))
(equal (implies p q)
       (if p (if q (t)
                   (f))
           (t)))
(equal (fix x)
       (if (numberp x)
           x
           (zero)))
(equal (if (if a b c)
           d e)
       (if a (if b d e)
           (if c d e)))
(equal (zerop x)
       (or (equal x (zero))
           (not (numberp x))))
(equal (plus (plus x y)
             z)
       (plus x (plus y z)))
(equal (equal (plus a b)
              (zero))
       (and (zerop a)
            (zerop b)))
(equal (difference x x)
       (zero))
(equal (equal (plus a b)
              (plus a c))
       (equal (fix b)
              (fix c)))
(equal (equal (zero)
              (difference x y))
       (not (lessp y x)))
(equal (equal x (difference x y))
       (and (numberp x)
            (or (equal x (zero))
                (zerop y))))
(equal (meaning (plus-tree (append x y))
                a)
       (plus (meaning (plus-tree x)
                      a)
             (meaning (plus-tree y)
                      a)))
(equal (meaning (plus-tree (plus-fringe x))
                a)
       (fix (meaning x a)))
(equal (append (append x y)
               z)
       (append x (append y z)))
(equal (reverse (append a b))
       (append (reverse b)
               (reverse a)))
(equal (times x (plus y z))
       (plus (times x y)
             (times x z)))
(equal (times (times x y)
              z)
       (times x (times y z)))
```

```
(equal (equal (times x y)
              (zero))
       (or (zerop x)
           (zerop y)))
(equal (exec (append x y)
             pds envrn)
       (exec y (exec x pds envrn)
             envrn))
(equal (mc-flatten x y)
       (append (flatten x)
               y))
(equal (member x (append a b))
       (or (member x a)
           (member x b)))
(equal (member x (reverse y))
       (member x y))
(equal (length (reverse x))
       (length x))
(equal (member a (intersect b c))
       (and (member a b)
            (member a c)))
(equal (nth (zero)
            i)
       (zero))
(equal (exp i (plus j k))
       (times (exp i j)
              (exp i k)))
(equal (exp i (times j k))
       (exp (exp i j)
            k))
(equal (reverse-loop x y)
       (append (reverse x)
               y))
(equal (reverse-loop x (nil))
       (reverse x))
(equal (count-list z (sort-lp x y))
       (plus (count-list z x)
             (count-list z y)))
(equal (equal (append a b)
              (append a c))
       (equal b c))
(equal (plus (remainder x y)
             (times y (quotient x y)))
       (fix x))
(equal (power-eval (big-plus1 l i base)
                   base)
       (plus (power-eval l base)
             i))
(equal (power-eval (big-plus x y i base)
                   base)
       (plus i (plus (power-eval x base)
                     (power-eval y base))))
(equal (remainder y 1)
       (zero))
(equal (lessp (remainder x y)
              y)
       (not (zerop y)))
(equal (remainder x x)
       (zero))
(equal (lessp (quotient i j)
              i)
       (and (not (zerop i))
            (or (zerop j)
                (not (equal j 1)))))
(equal (lessp (remainder x y)
              x)
       (and (not (zerop y))
            (not (zerop x))
            (not (lessp x y))))
```

```
(equal (power-eval (power-rep i base)
                   base)
       (fix i))
(equal (power-eval (big-plus (power-rep i base)
                             (power-rep j base)
                             (zero)
                             base)
                   base)
       (plus i j))
(equal (gcd x y)
       (gcd y x))
(equal (nth (append a b)
            i)
       (append (nth a i)
               (nth b (difference i (length a)))))
(equal (difference (plus x y)
                   x)
       (fix y))
(equal (difference (plus y x)
                   x)
       (fix y))
(equal (difference (plus x y)
                   (plus x z))
       (difference y z))
(equal (times x (difference c w))
       (difference (times c x)
                   (times w x)))
(equal (remainder (times x z)
                  z)
       (zero))
(equal (difference (plus b (plus a c))
                   a)
       (plus b c))
(equal (difference (add1 (plus y z))
                   z)
       (add1 y))
(equal (lessp (plus x y)
              (plus x z))
       (lessp y z))
(equal (lessp (times x z)
              (times y z))
       (and (not (zerop z))
            (lessp x y)))
(equal (lessp y (plus x y))
       (not (zerop x)))
(equal (gcd (times x z)
            (times y z))
       (times z (gcd x y)))
(equal (value (normalize x)
              a)
       (value x a))
(equal (equal (flatten x)
              (cons y (nil)))
       (and (nlistp x)
            (equal x y)))
(equal (listp (gopher x))
       (listp x))
(equal (samefringe x y)
       (equal (flatten x)
              (flatten y)))
(equal (equal (greatest-factor x y)
              (zero))
       (and (or (zerop y)
                (equal y 1))
            (equal x (zero))))
(equal (equal (greatest-factor x y)
              1)
       (equal x 1))
```

```
(equal (numberp (greatest-factor x y))
       (not (and (or (zerop y)
                     (equal y 1))
                 (not (numberp x)))))
(equal (times-list (append x y))
       (times (times-list x)
              (times-list y)))
(equal (prime-list (append x y))
       (and (prime-list x)
            (prime-list y)))
(equal (equal z (times w z))
       (and (numberp z)
            (or (equal z (zero))
                (equal w 1))))
(equal (greatereqpr x y)
       (not (lessp x y)))
(equal (equal x (times x y))
       (or (equal x (zero))
           (and (numberp x)
                (equal y 1))))
(equal (remainder (times y x)
                  y)
       (zero))
(equal (equal (times a b)
              1)
       (and (not (equal a (zero)))
            (not (equal b (zero)))
            (numberp a)
            (numberp b)
            (equal (1- a)
                   (zero))
            (equal (1- b)
                   (zero))))
(equal (lessp (length (delete x l))
              (length l))
       (member x l))
(equal (sort2 (delete x l))
       (delete x (sort2 l)))
(equal (dsort x)
       (sort2 x))
(equal (length (cons x1
                     (cons x2
                           (cons x3 (cons x4
                                          (cons x5
                                                (cons x6 x7)))))))
       (plus 6 (length x7)))
(equal (difference (add1 (add1 x))
                   2)
       (fix x))
(equal (quotient (plus x (plus x y))
                 2)
       (plus x (quotient y 2)))
(equal (sigma (zero)
              i)
       (quotient (times i (add1 i))
                 2))
(equal (plus x (add1 y))
       (if (numberp y)
           (add1 (plus x y))
           (add1 x)))
(equal (equal (difference x y)
              (difference z y))
       (if (lessp x y)
           (not (lessp y z))
           (if (lessp z y)
               (not (lessp y x))
               (equal (fix x)
                      (fix z)))))
```

```
(equal (meaning (plus-tree (delete x y))
                a)
       (if (member x y)
           (difference (meaning (plus-tree y)
                                a)
                       (meaning x a))
           (meaning (plus-tree y)
                    a)))
(equal (times x (add1 y))
       (if (numberp y)
           (plus x (times x y))
           (fix x)))
(equal (nth (nil)
            i)
       (if (zerop i)
           (nil)
           (zero)))
(equal (last (append a b))
       (if (listp b)
           (last b)
           (if (listp a)
               (cons (car (last a))
                     b)
               b)))
(equal (equal (lessp x y)
              z)
       (if (lessp x y)
           (equal t z)
           (equal f z)))
(equal (assignment x (append a b))
       (if (assignedp x a)
           (assignment x a)
           (assignment x b)))
(equal (car (gopher x))
       (if (listp x)
           (car (flatten x))
           (zero)))
(equal (flatten (cdr (gopher x)))
       (if (listp x)
           (cdr (flatten x))
           (cons (zero)
                 (nil))))
(equal (quotient (times y x)
                 y)
       (if (zerop y)
           (zero)
           (fix x)))
(equal (get j (set i val mem))
       (if (eqp j i)
           val
           (get j mem))))))
```

```
(defun tautologyp (x true-lst false-lst)
  (cond ((truep x true-lst)
         t)
        ((falsep x false-lst)
         nil)
        ((atom x)
         nil)
        ((eq (car x)
             (quote if))
         (cond ((truep (cadr x)
                       true-lst)
                (tautologyp (caddr x)
                            true-lst false-lst))
               ((falsep (cadr x)
                        false-lst)
                (tautologyp (cadddr x)
                            true-lst false-lst))
               (t (and (tautologyp (caddr x)
                                   (cons (cadr x)
                                         true-lst)
                                   false-lst)
                       (tautologyp (cadddr x)
                                   true-lst
                                   (cons (cadr x)
                                         false-lst))))))
        (t nil)))

(defun tautp (x)
  (tautologyp (rewrite x)
              nil nil))

(defun test ()
  (prog (ans term)
        (setq term
              (apply-subst
                (quote ((x f (plus (plus a b)
                                   (plus c (zero))))
                        (y f (times (times a b)
                                    (plus c d)))
                        (z f (reverse (append (append a b)
                                              (nil))))
                        (u equal (plus a b)
                           (difference x y))
                        (w lessp (remainder a b)
                           (member a (length b)))))
                (quote (implies (and (implies x y)
                                     (and (implies y z)
                                          (and (implies z u)
                                               (implies u w))))
                                (implies x w)))))
        (setq ans (tautp term))))

(defun trans-of-implies (n)
  (list (quote implies)
        (trans-of-implies1 n)
        (list (quote implies)
              0 n)))

(defun trans-of-implies1 (n)
  (cond ((equal n 1)                    ; I think (eql n 1) may work here
         (list (quote implies)
               0 1))
        (t (list (quote and)
                 (list (quote implies)
                       (1- n)
                       n)
                 (trans-of-implies1 (1- n))))))
```

```
(defun truep (x lst)
       (or (equal x (quote (t)))
           (member x lst)))

(eval-when (load eval)
  (setup))
```

```
;;; BROWSE

;;; Benchmark to create and browse through an AI-like data base of
;;; units.

;;; call: (browse)

;;; n is # of symbols
;;; m is maximum amount of stuff on the plist
;;; npats is the number of basic patterns on the unit
;;; ipats is the instantiated copies of the patterns

(defvar rand 21.)

(defun seed () (setq rand 21.))

(defmacro char1 (x) `(aref (string ,x) 0))      ; maybe SYMBOL-NAME

(defun init (n m npats ipats)
  (let ((ipats (copy-tree ipats)))
    (do ((p ipats (cdr p)))
        ((null (cdr p)) (rplacd p ipats)))
    (do ((n n (1- n))
         (i m (cond ((= i 0) m)
                    (t (1- i))))
         (name (gensym) (gensym))
         (a ()))
        ((= n 0) a)
      (push name a)
      (do ((i i (1- i)))
          ((= i 0))
        (setf (get name (gensym)) nil))
      (setf (get name 'pattern)
            (do ((i npats (1- i))
                 (ipats ipats (cdr ipats))
                 (a ()))
                ((= i 0) a)
              (push (car ipats) a)))
      (do ((j (- m i) (1- j)))
          ((= j 0))
        (setf (get name (gensym)) nil)))))


(defun browse-random ()
  (setq rand (mod (* rand 17.) 251.)))

(defun randomize (l)
  (do ((a '()))
      ((null l) a)
    (let ((n (mod (browse-random) (length l))))
      (cond ((= n 0)
             (push (car l) a)
             (setq l (cdr l)))
            (t
             (do ((n n (1- n))
                  (x l (cdr x)))
                 ((= n 1)
                  (push (cadr x) a)
                  (rplacd x (cddr x))))))))))
```

```
(defun match (pat dat alist)
  (cond ((null pat)
         (null dat))
        ((null dat) ())
        ((or (eq (car pat) '?)
             (eq (car pat)
                 (car dat)))
         (match (cdr pat) (cdr dat) alist))
        ((eq (car pat) '*)
         (or (match (cdr pat) dat alist)
             (match (cdr pat) (cdr dat) alist)
             (match pat (cdr dat) alist)))
        (t (cond ((atom (car pat))
                   (cond ((eq (char1 (car pat)) #\?)
                          (let ((val (assoc (car pat) alist)))
                            (cond (val (match (cons (cdr val)
                                                    (cdr pat))
                                              dat alist))
                                  (t (match (cdr pat)
                                            (cdr dat)
                                            (cons (cons (car pat)
                                                        (car dat))
                                                  alist))))))
                         ((eq (char1 (car pat)) #\*)
                          (let ((val (assoc (car pat) alist)))
                            (cond (val (match (append (cdr val)
                                                      (cdr pat))
                                              dat alist))
                                  (t
                                   (do ((l () (nconc l (cons (car d) nil)))
                                        (e (cons () dat) (cdr e))
                                        (d dat (cdr d)))
                                       ((null e) ())
                                     (cond ((match (cdr pat) d
                                                   (cons (cons (car pat) l)
                                                         alist))
                                            (return t))))))))))
                  (t (and
                       (not (atom (car dat)))
                       (match (car pat)
                              (car dat) alist)
                       (match (cdr pat)
                              (cdr dat) alist)))))))

(defun browse ()
  (seed)
  (investigate (randomize
                 (init 100. 10. 4. '((a a a b b b b a a a a a b b a a a)
                                     (a a b b b b a a
                                         (a a)(b b))
                                     (a a a b (b a) b a b a))))
               '((*a ?b *b ?b a *a a *b *a)
                 (*a *b *b *a (*a) (*b))
                 (? ? * (b a) * ? ?))))

(defun investigate (units pats)
  (do ((units units (cdr units)))
      ((null units))
    (do ((pats pats (cdr pats)))
        ((null pats))
      (do ((p (get (car units) 'pattern)
              (cdr p)))
          ((null p))
        (match (car pats) (car p) ())))))
```

```
;;; CTAK

;;; A version of the TAKeuchi function that uses the CATCH/THROW
;;; facility.

;;; call: (ctak 18. 12. 6.)

(defun ctak (x y z)
  (catch 'ctak (ctak-aux x y z)))

(defun ctak-aux (x y z)
  (cond ((not (< y x))   ;xy
         (throw 'ctak z))
        (t (ctak-aux
             (catch 'ctak
               (ctak-aux (1- x)
                         y
                         z))
             (catch 'ctak
               (ctak-aux (1- y)
                         z
                         x))
             (catch 'ctak
               (ctak-aux (1- z)
                         x
                         y))))))
```

```
;;; FRPOLY

;;; Benchmark from Berkeley based on polynomial arithmetic.
;;; Originally writen in Franz Lisp by Richard Fateman.  PDIFFER1
;;; appears in the code, but it is not defined; it is not used in
;;; this test, however.

;;; There are four sets of three tests - call:
;;;  (pexptsq r 2)  (pexptsq r2 2)  (pexptsq r3 2)
;;;  (pexptsq r 5)  (pexptsq r2 5)  (pexptsq r3 5)
;;;  (pexptsq r 10) (pexptsq r2 10) (pexptsq r3 10)
;;;  (pexptsq r 15) (pexptsq r2 15) (pexptsq r3 15)

(defvar ans)
(defvar coef)
(defvar f)
(defvar inc)
(defvar i)
(defvar qq)
(defvar ss)
(defvar v)
(defvar *x*)
(defvar *alpha*)
(defvar *a*)
(defvar *b*)
(defvar *chk)
(defvar *l)
(defvar *p)
(defvar q*)
(defvar u*)
(defvar *var)
(defvar *y*)
(defvar r)
(defvar r2)
(defvar r3)
(defvar start)
(defvar res1)
(defvar res2)
(defvar res3)

(defmacro pointergp (x y) `(> (get ,x 'order)(get ,y 'order)))
(defmacro pcoefp (e) `(atom ,e))

(defmacro pzerop (x)
  `(if (numberp ,x)                                    ; no signp in CL
       (zerop ,x)))
(defmacro pzero () 0)
(defmacro cplus (x y) `(+ ,x ,y))
(defmacro ctimes (x y) `(* ,x ,y))

(defun pcoefadd (e c x)
  (if (pzerop c)
      x
      (cons e (cons c x))))

(defun pcplus (c p)
  (if (pcoefp p)
      (cplus p c)
      (psimp (car p) (pcplus1 c (cdr p)))))

(defun pcplus1 (c x)
  (cond ((null x)
         (if (pzerop c)
             nil
             (cons 0 (cons c nil))))
        ((pzerop (car x))
         (pcoefadd 0 (pplus c (cadr x)) nil))
        (t
         (cons (car x) (cons (cadr x) (pcplus1 c (cddr x)))))))
```

```
(defun pctimes (c p)
  (if (pcoefp p)
      (ctimes c p)
      (psimp (car p) (pctimes1 c (cdr p)))))

(defun pctimes1 (c x)
  (if (null x)
      nil
      (pcoefadd (car x)
                (ptimes c (cadr x))
                (pctimes1 c (cddr x)))))

(defun pplus (x y)
  (cond ((pcoefp x)
         (pcplus x y))
        ((pcoefp y)
         (pcplus y x))
        ((eq (car x) (car y))
         (psimp (car x) (pplus1 (cdr y) (cdr x))))
        ((pointergp (car x) (car y))
         (psimp (car x) (pcplus1 y (cdr x))))
        (t
         (psimp (car y) (pcplus1 x (cdr y))))))

(defun pplus1 (x y)
  (cond ((null x) y)
        ((null y) x)
        ((= (car x) (car y))
         (pcoefadd (car x)
                   (pplus (cadr x) (cadr y))
                   (pplus1 (cddr x) (cddr y))))
        ((> (car x) (car y))
         (cons (car x) (cons (cadr x) (pplus1 (cddr x) y))))
        (t (cons (car y) (cons (cadr y) (pplus1 x (cddr y)))))))

(defun psimp (var x)
  (cond ((null x) 0)
        ((atom x) x)
        ((zerop (car x))
         (cadr x))
        (t
         (cons var x))))

(defun ptimes (x y)
  (cond ((or (pzerop x) (pzerop y))
         (pzero))
        ((pcoefp x)
         (pctimes x y))
        ((pcoefp y)
         (pctimes y x))
        ((eq (car x) (car y))
         (psimp (car x) (ptimes1 (cdr x) (cdr y))))
        ((pointergp (car x) (car y))
         (psimp (car x) (pctimes1 y (cdr x))))
        (t
         (psimp (car y) (pctimes1 x (cdr y))))))

(defun ptimes1 (*x* y)
  (prog (u* v)
        (setq v (setq u* (ptimes2 y)))
     a
        (setq *x* (cddr *x*))
        (if (null *x*)
            (return u*))
        (ptimes3 y)
        (go a)))
```

```
(defun ptimes2 (y)
  (if (null y)
      nil
      (pcoefadd (+ (car *x*) (car y))
                (ptimes (cadr *x*) (cadr y))
                (ptimes2 (cddr y)))))

(defun ptimes3 (y)
  (prog (e u c)
     a1 (if (null y)
            (return nil))
        (setq e (+ (car *x*) (car y))
              c (ptimes (cadr y) (cadr *x*) ))
        (cond ((pzerop c)
               (setq y (cddr y))
               (go a1))
              ((or (null v) (> e (car v)))
               (setq u* (setq v (pplus1 u* (list e c))))
               (setq y (cddr y))
               (go a1))
              ((= e (car v))
               (setq c (pplus c (cadr v)))
               (if (pzerop c)                     ; never true, evidently
                   (setq u* (setq v (pdiffer1 u* (list (car v) (cadr v)))))
                   (rplaca (cdr v) c))
               (setq y (cddr y))
               (go a1)))
     a  (cond ((and (cddr v) (> (caddr v) e))
               (setq v (cddr v))
               (go a)))
        (setq u (cdr v))
     b  (if (or (null (cdr u)) (< (cadr u) e))
            (rplacd u (cons e (cons c (cdr u)))) (go e))
        (cond ((pzerop (setq c (pplus (caddr u) c)))
               (rplacd u (cdddr u))
               (go d))
              (t
               (rplaca (cddr u) c)))
     e  (setq u (cddr u))
     d  (setq y (cddr y))
        (if (null y)
            (return nil))
        (setq e (+ (car *x*) (car y))
              c (ptimes (cadr y) (cadr *x*)))
     c  (cond ((and (cdr u) (> (cadr u) e))
               (setq u (cddr u))
               (go c)))
        (go b)))

(defun pexptsq (p n)
  (do ((n (floor n 2) (floor n 2))
       (s (if (oddp n) p 1)))
      ((zerop n) s)
    (setq p (ptimes p p))
    (and (oddp n) (setq s (ptimes s p)))))

(eval-when (load eval)
  (setf (get 'x 'order) 1)
  (setf (get 'y 'order) 2)
  (setf (get 'z 'order) 3)
  (setq r (pplus '(x 1 1 0 1) (pplus '(y 1 1) '(z 1 1)))  ; r= x+y+z+1
        r2 (ptimes r 100000)                              ; r2 = 100000*r
        r3 (ptimes r 1.0))                                ; r3 = r with
                                                          ;  floating point
                                                          ;  coefficients
```

[lisp] puzzle.1

```
;;; PUZZLE

;;; Forest Baskett's Puzzle benchmark, originally written in Pascal.

;;; call: (start)

(eval-when (load eval)
  (defconstant size 511.)
  (defconstant classmax 3.)
  (defconstant typemax 12.))

(defvar iii 0)
(defvar kount 0)
(defvar d 8.)

(defvar piece-count (make-array (1+ classmax) :initial-element 0))
(defvar class (make-array (1+ typemax) :initial-element 0))
(defvar piecemax (make-array (1+ typemax) :initial-element 0))
(defvar puzzle (make-array (1+ size)))
(defvar p (make-array (list (1+ typemax) (1+ size))))

(defun fit (i j)
  (let ((end (aref piecemax i)))
    (do ((k 0 (1+ k)))
        ((> k end) t)
      (cond ((aref p i k)
             (cond ((aref puzzle (+ j k))
                    (return nil))))))))

(defun place (i j)
  (let ((end (aref piecemax i)))
    (do ((k 0 (1+ k)))
        ((> k end))
      (cond ((aref p i k)
             (setf (aref puzzle (+ j k)) t))))
    (setf (aref piece-count (aref class i)) (- (aref piece-count (aref class i)) 1))
    (do ((k j (1+ k)))
        ((> k size)
#|       (terpri)
         (princ "Puzzle filled")|#
         0)
      (cond ((not (aref puzzle k))
             (return k))))))

(defun puzzle-remove (i j)
  (let ((end (aref piecemax i)))
    (do ((k 0 (1+ k)))
        ((> k end))
      (cond ((aref p i k)
             (setf (aref puzzle (+ j k))  nil))))
      (setf (aref piece-count (aref class i))(+ (aref piece-count (aref class i)) 1))))

#|(defun puzzle-remove (i j)
  (let ((end (aref piecemax i)))
    (do ((k 0 (1+ k)))
        ((> k end))
      (cond ((aref p i k) (setf (aref puzzle (+ j k))  nil)))
      (setf (aref piece-count (aref class i)) (+ (aref piece-count (aref class i)) 1)))))|#
```

```
(defun trial (j)
  (let ((k 0))
    (do ((i 0 (1+ i)))
        ((> i typemax) (setq kount (1+ kount))   nil)
      (cond ((not (= (aref piece-count (aref class i)) 0))
             (cond ((fit i j)
                    (setq k (place i j))
                    (cond ((or (trial k)
                               (= k 0))
;                           (format t "~%Piece ~4D at ~4D." (+ i 1) (+ k 1))
                           (setq kount (+ kount 1))
                           (return t))
                          (t (puzzle-remove i j))))))))))

(defun define-piece (iclass ii jj kk)
  (let ((index 0))
    (do ((i 0 (1+ i)))
        ((> i ii))
      (do ((j 0 (1+ j)))
          ((> j jj))
        (do ((k 0 (1+ k)))
            ((> k kk))
          (setq index  (+ i (* d (+ j (* d k)))))
          (setf (aref p iii index)  t))))
    (setf (aref class iii) iclass)
    (setf (aref piecemax iii) index)
    (cond ((not (= iii typemax))
           (setq iii (+ iii 1))))))

(defun start ()
  (do ((m 0 (1+ m)))
      ((> m size))
    (setf (aref puzzle m) t))
  (do ((i 1 (1+ i)))
      ((> i 5))
    (do ((j 1 (1+ j)))
        ((> j 5))
      (do ((k 1 (1+ k)))
          ((> k 5))
        (setf (aref puzzle (+ i (* d (+ j (* d k))))) nil))))
  (do ((i 0 (1+ i)))
      ((> i typemax))
    (do ((m 0 (1+ m)))
        ((> m size))
      (setf (aref p i m)  nil)))
  (setq iii 0)
  (define-piece 0 3 1 0)
  (define-piece 0 1 0 3)
  (define-piece 0 0 3 1)
  (define-piece 0 1 3 0)
  (define-piece 0 3 0 1)
  (define-piece 0 0 1 3)

  (define-piece 1 2 0 0)
  (define-piece 1 0 2 0)
  (define-piece 1 0 0 2)

  (define-piece 2 1 1 0)
  (define-piece 2 1 0 1)
  (define-piece 2 0 1 1)

  (define-piece 3 1 1 1)
(setf (aref piece-count 0) 13.)
(setf (aref piece-count 1) 3)
(setf (aref piece-count 2) 1)
(setf (aref piece-count 3) 1)
(let ((m (+ 1 (* d (+ 1 d))))
      (n 0)(kount 0))
  (cond ((fit 0 m) (setq n (place 0 m)))
        (t (format t "~%Error.")))
  (cond ((trial n)
         (format t "~%Success in ~4D trials." kount))
        (t (format t "~%Failure.")))))
```

```
;;; STAK

;;; The TAKeuchi function with special variables instead of parameter
;;; passing.

;;; call: (stak 18. 12. 6.))

(defvar x)
(defvar y)
(defvar z)

(defun stak (x y z)
  (stak-aux))

(defun stak-aux ()
  (if (not (< y x))              ; xy
      z
      (let ((x (let ((x (1- x))
                     (y y)
                     (z z))
                 (stak-aux)))
            (y (let ((x (1- y))
                     (y z)
                     (z x))
                 (stak-aux)))
            (z (let ((x (1- z))
                     (y x)
                     (z y))
                 (stak-aux))))
        (stak-aux))))
```

```
;;; TAK

;;; A vanilla version of the TAKeuchi function and one with tail
;;; recursion removed.

;;; call: (tak 18. 12. 6.)

(defun tak (x y z)
  (if (not (< y x))               ; xy
      z
      (tak (tak (1- x) y z)
              (tak (1- y) z x)
              (tak (1- z) x y))))

;;; call: (trtak 18. 12. 6.)

(defun trtak (x y z)
  (prog ()
     tak
        (if (not (< y x))
            (return z)
            (let ((a (tak (1- x) y z))
                  (b (tak (1- y) z x)))
              (setq z (tak (1- z) x y)
                    x a
                    y b)
              (go tak)))))
```

```
;;; TAKL

;;; The TAKeuchi function using lists as counters.

;;; call: (mas 18l 12l 6l)

(defun listn (n)
  (if (not (= 0 n))
      (cons n (listn (1- n)))))

(defvar 18l (listn 18.))
(defvar 12l (listn 12.))
(defvar  6l (listn 6.))

(defun mas (x y z)
  (if (not (shorterp y x))
      z
      (mas (mas (cdr x)
                y z)
            (mas (cdr y)
                z x)
            (mas (cdr z)
                x y))))

(defun shorterp (x y)
  (and y (or (null x)
             (shorterp (cdr x)
                       (cdr y)))))
```

ili

```
# /*
  ili.m: benchmark ili master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   ili
%
%   Seif Haridi (Swedish Institute of Computer Science)
%
%   (modified August 1986 by Evan Tick)
%
%   intuitionistic logic interpreter

#if BENCH
#  include ".ili.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
ili :- q(I, R, P),
       (s(P) -> R = provable ; R = unprovable),
       write('Query '), write(I), write(': '), write(P), nl,
       write(success), nl, nl,
       fail.
ili.
#  else
ili :- q(_, _, P),
       s(P),
       fail.
ili.
#  endif
#endif

#option "
        > ili code includes several (non-standard) operators.
        > If one of
        >
        > C_PL QUINTUS_PL SICSTUS_PL
        >
        > is selected, then op/3 directives are generated for
        > these automatically.  If your Prolog system handles
        > op/3, then you may add an option for it to the list
        > above."
#if C_PL || QUINTUS_PL || SICSTUS_PL

#  include "operator"   /* op/3 directives */

#else
#  message "WARNING: load op/3 directives from (file) operator"
#endif
```

```
% query set

q(1,   provable,   a & b <-> b & a                                  ).
q(2,   provable,   a # b <-> b # a                                  ).
q(3,   provable,   (a & b) & c <-> (a & b) & c                      ).
q(4,   provable,   (a # b) #c <-> a# (b # c)                        ).
q(5,   provable,   a # (b & c) <-> (a # b) & (a # c)                ).
q(6,   provable,   a & (b # c) <-> (a & b) # (a & c)                ).
q(7,   provable,   a => ((a => ff) => ff)                           ).
q(8,   provable,   (a => (b => c)) => (a & b => c)                  ).
q(9,   provable,   a => (b => a)                                    ).
q(10,  provable,   a => ((a => ff) => b)                            ).
q(11,  provable,   ((a # b) => ff) => (a => ff) & (b => ff)         ).
q(12,  provable,   ((a => ff) # (b => ff)) => (a & b => ff)         ).
q(13,  provable,   ((a => ff) # b) => (a => b)                      ).
q(14,  provable,   (a => b) => ((b => ff) => (a => ff))             ).
q(15,  provable,   (a => b) => ((b => c) => (a => c))               ).
q(16,  provable,   ff <-> a & (a => ff)                             ).
q(17,  provable,   e(X, p(X) # r(X)) => e(Y, p(Y)) # e(Z, r(Z))    ).
q(18,  provable,   a(X, p(X) & r(X)) => a(Y, p(Y)) & a(Z,r(Z))      ).
q(19,  provable,   (e(X, p(X)) => ff) => a(Y, p(Y) => ff)           ).
q(20,  provable,   e(X, p(X) => ff) => (a(Y, p(Y)) => ff)           ).
q(21,  provable,   a(X, b => p(X)) => (b => a(Y, p(Y)))             ).
q(22,  provable,   e(X, a => p(X)) => (a => e(Y, p(Y)))             ).
q(23,  provable,   b # a(X, p(X)) => a(Y, b # p(Y))                 ).
q(24,  provable,   b & e(X, p(X)) => e(Y, b & p(Y))                 ).
q(25,  provable,   e(X, p(X) => b) => (a(Y, p(Y)) => b)             ).
q(26,  provable,   a(X, p(X) => b) => (e(Y, p(Y)) => b)             ).
q(27,  provable,   a(X, e(Y, Y = X))                                ).
q(28,  unprovable, e(X, a(Y, Y = X))                                ).
q(29,  unprovable, e(X, a(Y, f(Y) = X))                             ).
q(30,  provable,   a(X, professor(X) => e(Y, teaches(X,Y)))         ).
q(31,  provable,   e(X, e(Y, (X = Y => ff)))                        ).
q(32,  provable,   e(X, e(Y, (X = Y => ff) & X = 1 & Y = 2))        ).
q(33,  provable,   e(X, a(Y, X = Y => X = Y))                       ).
q(34,  unprovable, e(X, a(Y, X = Y => ff))                          ).
q(35,  unprovable, a(X, e(Y, X = Y => ff))                          ).
q(36,  provable,   a(X, e(Y,(X = Y => Y = 1) & Y=1))                ).
q(37,  unprovable, e(X, e(Y,(X = Y => ff) & X = 1 & Y = 1))         ).
q(38,  provable,   a(X, brother(X, _) => male(X))                   ).
/* added by ET 08-10-86 */
q(101, provable,   e(X, lessall(X,[b,c,d]))                         ).
/* added by ET 08-10-86 */
q(102, unprovable, e(X, lessall(X,[a,b,c,d]))                       ).


% intuitionistic logic interpreter

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (s/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
s(_).
#halt
#endif
s(P) :- sb([],P,[]).
```

```
sb(_L,tt,_CE) :- !.
sb(L,F1&F2,CE) :- !,
    sb(L,F1,CE),
    sb(L,F2,CE).
sb(L,F1#F2,CE) :- !,
    sf(L,F1#F2,CE).
sb(L,a(V,F),CE) :- !,
    replace(X/V,a(V,F),a(X,F1)),
    star(X),
    freeVars(a(X,F1),VL),
    sb(L,F1,CE),
    checkBinding(VL,X).
sb(L,e(V,F),CE) :- !,
    sf(L,e(V,F),CE).
sb(L,F1<->F2,CE) :- !,
    sb(L,F1=>F2,CE),
    sb(L,F2=>F1,CE).
sb(L,F1=>F2,CE) :- !,
    sb([F1|L],F2,CE).
sb(L,T1=T2,CE) :- !,
    sf(L,T1=T2,CE).
sb(_L,P,_CE) :- my_builtin(P), !,         /* INCOMPLETE */
    call(P).
sb(L,A,CE) :-
    sf(L,A,CE).

sf([],e(V,F),CE) :- !,
    replace(X/V,e(V,F),e(X,F1)),
    sb([],F1,CE).
sf([],F1#F2,CE) :- !,
    (sb([],F1,CE) ; sb([],F2,CE)).
sf([],T1=T2,CE) :- !,
    unifyb(T1,T2,CE).
sf([],A,CE) :- !,
    (findStatement(A,H<-B),
     unifyb(A,H,CE),
     sb([],B,CE) ;
     findAtom(A,H,CE),
     unifyb(A,H,CE)).
sf([ff|_FR],_CF,_CE) :- !.
sf([F1#F2|FR],CF,CE) :- !,
    sf([F1|FR],CF,CE),
    sf([F2|FR],CF,CE).
sf([F1&F2|FR],CF,CE) :- !,
    sf([F1,F2|FR],CF,CE).
sf([a(V,F)|FR],CF,CE) :- !,
    replace(X/V,a(V,F),a(X,F1)),
    sf([F1|FR],CF,CE).
sf([e(V,F)|FR],CF,CE) :- !,
    replace(X/V,e(V,F),e(X,F1)),
    star(X),
    freeVars(e(X,F1),VL),
    sf([F1|FR],CF,CE),
    checkBinding(VL,X).

sf([F1<->F2|FR],CF,CE) :- !,
    sf([F1=>F2,F2=>F1|FR],CF,CE).
sf([F1=>F2|FR],CF,CE) :- !,
    (sb(FR,F1,CE),
     sf([F2|FR],CF,CE) ;
     sf(FR,CF,CE)).
sf([T1=T2|FR],CF,CE) :- !,
    X=sf([T1=T2|FR],CF,CE),
    unifyf(T1,T2,CE,CE1,X),
    (CE1=fail ; sf(FR,CF,CE1)).
sf([Atom|FR],CF,CE) :-
    findStatement(Atom,H<->B),
    unifyb(Atom,H,CE),
    append(FR,[B],FR1),
    sf(FR1,CF,CE).
sf([Atom|FR],CF,CE) :-
    sf(FR,CF,[Atom|CE]).

star(*(_)).

findAtom(A,A,E) :- atom(A), member(A,E), !.
findAtom(A,H,E) :-
    functor(A,F,N),
    functor(H,F,N),
    member(H,E).

freeVars(X,[]) :- isstar(X), !.
freeVars(X,[X]) :- var(X), !.
freeVars(X,L) :- isdelay(X), !,
    dereference(X,X2,[]),
    (isdelay(X2) -> L=[X2] ; L=[]).
freeVars(A,[]) :- atomic(A), !.
freeVars(a(X,F),V) :- !,
    freeVars(F,V1),
    del(X,V1,V).
freeVars(e(X,F),V) :- !,
    freeVars(F,V1),
    del(X,V1,V).
freeVars(T,V) :- compound(T),
    T =.. [_|Ts],
    freeVarsList(Ts,V).

freeVarsList([],[]).
freeVarsList([T|Ts],V) :- !,
    freeVars(T,V1),
    freeVarsList(Ts,V2),
    append(V1,V2,V).

/* checkBinding(L,V) :-  \+ in(L,V), !. */

checkBinding([],_) :- !.
checkBinding([X|L],V) :- (var(X) ; isstar(X)), !,
    X \== V,
    checkBinding(L,V).
checkBinding([X|L],V) :- isdelay(X), !,
    dereference(X,X2,[]),
    (isdelay(X2) ->
      makedelay(X2,checkBinding([X],V)),
      checkBinding(L,V) ;
     checkBinding([X2|L],V)).
checkBinding([X|L],V) :- atomic(X), !,
    checkBinding(L,V).
checkBinding([X|L],V) :- compound(X), !,
    X =.. [_|T],
    checkBinding(T,V),
    checkBinding(L,V).

findStatement(_#_,_) :- !, fail.
findStatement(_&_,_) :- !, fail.
findStatement(P,H<-B) :-
    functor(P,F,N),
    functor(H,F,N),
    H <- B.
findStatement(P,H<-B) :-
    findStatement(P,H<->B).
findStatement(P,H<->B) :-
    functor(P,F,N),
    functor(H,F,N),
    H <-> B.

unifyb(T1,T2,E) :-
    unifybl([T1],[T2],E).
```

```
unifyb1([],[],_) :- !.
unifyb1([X|L1],[Y|L2],E) :-
    dereference(X,X1,E),
    dereference(Y,Y1,E),
    unifyb2([X1|L1],[Y1|L2],E).

unifyb2([X|L1],[Y|L2],E) :- var(X), !,
    X=Y,
    unifyb1(L1,L2,E).
unifyb2([X|L1],[Y|L2],E) :- isstar(X), !,
    (var(Y) -> X=Y ;
     isstar(Y) -> X == Y ;
     (isdelay(Y), binddelay(Y,X))),
    unifyb1(L1,L2,E).
unifyb2([X|L1],[Y|L2],E) :- isdelay(X), !,
    (var(Y) -> X=Y ;
     isdelay(Y) -> (X\==Y -> joindelay(X,Y)) ;
     binddelay(X,Y)),
    unifyb1(L1,L2,E).
unifyb2([X|L1],[Y|L2],E) :- atomic(X), !,
    (var(Y) -> X=Y ;
     isdelay(Y) -> binddelay(Y,X) ;
     atomic(Y) -> X=Y ;
     fail),
    unifyb1(L1,L2,E).
unifyb2([X|L1],[Y|L2],E) :- compound(X), !,
    (var(Y) -> (X=Y, unifyb1(L1,L2,E)) ;
     isdelay(Y) -> (binddelay(Y,X), unifyb1(L1,L2,E)) ;
     compound(Y) ->
      (functor(X,F,N), functor(Y,F,N),
       (X =.. [F|S1]), (Y =.. [F|S2]),
       append(S1,L1,M1), append(S2,L2,M2),
       unifyb1(M1,M2,E)) ;
     fail).

dereference(X,X1,E) :-
    (var(X) -> X=X1 ;
     isstar(X) ->
      (binding(X,E,X2) -> dereference(X2,X1,E); X=X1) ;
     isdelay(X) ->
      (hasdelayvalue(X,V) -> dereference(V,X1,E); X=X1) ;
     X=X1).

isstar(X) :- nonvar(X), X = *(_).

isdelay(X) :- nonvar(X), X=delay(_,_).

hasdelayvalue(delay(V,_),V2) :- nonvar(V), V=V2.

makedelay(delay(_,F),X) :-
    appvar(F,[X|_]).

binddelay(delay(V,F),V) :-
    calllist(F).

calllist(E) :- var(E), !.
calllist([H|T]) :-
    call(H),
    calllist(T).

joindelay(delay(V1,F1),delay(V2,F2)) :- V1=delay(V2,F2),
    appvar(F2,F1).

/* not in use...
clean(Dirty, Clean) :-
    clean(Dirty, 0, Clean, _).
clean(Var, Index, Var, Index) :- var(Var), !.
clean(delay(NonVar, _), Index, NonVar, Index) :- nonvar(NonVar), !.
clean(delay($(Index0),DirtyGoals), Index0, $(Index0):CleanGoals, Index) :- !,
    Index1 is Index0 + 1,
    clean(DirtyGoals, Index1, CleanGoals, Index).
clean(DirtyTerm, Index0, CleanTerm, Index) :-
    DirtyTerm=..[F|DirtyArgs],
    cleanlist(DirtyArgs, Index0, CleanArgs, Index),
    CleanTerm =.. [F|CleanArgs].

cleanlist([], Index, [], Index).
cleanlist([Dirty|DirtyArgs], Index0, [Clean|CleanArgs], Index) :-
    clean(Dirty, Index0, Clean, Index1),
    cleanlist(DirtyArgs, Index1, CleanArgs, Index).

portray(Dirty) :-
    \+ \+ (clean(Dirty, Clean), write(Clean)).
*/

unifyf(T1,T2,E1,E2,D) :-
    unifyf1([T1],[T2],E1,E2,D).

unifyf1([],[],E,E,_) :- !.
unifyf1([X|L1],[Y|L2],E1,E2,D) :-
    dereference(X,X1,E1),
    dereference(Y,Y1,E1),
    unifyf2([X1|L1],[Y1|L2],E1,E2,D).

unifyf2([X|L1],[Y|L2],E1,E2,D) :- var(X), !,
     (isstar(Y) -> (E3=[(Y=X)|E1], unifyf1(L1,L2,E3,E2,D)) ;
      (makedelay(X,D), E2=fail)).

unifyf2([X|L1],[Y|L2],E1,E2,D) :- isstar(X), !,
    E3=[(X=Y)|E1], unifyf1(L1,L2,E3,E2,D).

unifyf2([X|L1],[Y|L2],E1,E2,D) :- isdelay(X), !,
     (isstar(Y)  -> (E3=[(Y=X)|E1], unifyf1(L1,L2,E3,E2,D)) ;
      (makedelay(X,D), E2=fail)).

unifyf2([X|L1],[Y|L2],E1,E2,D) :- atomic(X), !,
     (atomic(Y) -> (X==Y -> unifyf1(L1,L2,E1,E2,D); E2=fail) ;
      compound(Y) -> E2=fail ;
      isstar(Y) -> (E3=[(Y=X)|E1], unifyf1(L1,L2,E3,E2,D)) ;
      (makedelay(Y,D), E2=fail)).

unifyf2([X|L1],[Y|L2],E1,E2,D) :- compound(X), !,
     (compound(Y) ->
        ((functor(X,F,N), functor(Y,F,N)) ->
          ((X =.. [F|S1]), (Y =.. [F|S2]),
           append(S1,L1,M1), append(S2,L2,M2),
           unifyf1(M1,M2,E1,E2,D)) ;
         E2=fail) ;
      atomic(Y) -> E2=fail ;
      isstar(Y) -> (E3=[(Y=X)|E1], unifyf1(L1,L2,E3,E2,D)) ;
      (makedelay(Y,D), E2=fail)).

binding(X,[Y=T|_],T) :-
    X == Y, !.
binding(X,[_|L],T) :-
    binding(X,L,T).

deref(X,E,T) :-
    binding(X,E,T1), !,
    deref(T1,E,T).
deref(X,_,X).
```

```
% utilities

rev(L1,L2) :-
    rev([],L1,L2).
rev(L,[],L).
rev(L1,[X|L2],L3) :-
    rev([X|L1],L2,L3).

apply_all(_,[]).
apply_all(R,[X|Y]) :-
    apply(R,[X]),
    apply_all(R,Y).

apply_all(_,_,[]) :- !.
apply_all(R,C,[X|Y]) :-
    apply(R,[X,C]), !,
    apply_all(R,C,Y).

apply_either(R,C,M,[X|Y]) :-
    (apply(R,[X,C,M]) ;
     apply_either(R,C,M,Y)).

apply_or(R,C,M,[X|Y]) :-
    (apply(R,[X,C,M]) ;
     apply_or(R,C,M,Y)).

apply_or(M,[X|Y],E) :-
    (apply(M,[X,E]) ;
     apply_or(M,Y,E)).

apply(R,Ts) :-
    X =.. [R|Ts],
    call(X).

apply_list(_,[],[]) :- !.
apply_list(R,[X1|L1],[X2|L2]) :-
    apply(R,[X1,X2]), !,
    apply_list(R,L1,L2).

apply_list(_,[],_,[]) :- !.
apply_list(R,[X1|L1],C,[X2|L2]) :-
    apply(R,[X1,C,X2]),
    apply_list(R,L1,C,L2).

apply_list(_,[],_,_,[]) :- !.
apply_list(R,[X1|L1],C1,C2,[X2|L2]) :-
    apply(R,[X1,C1,C2,X2]),
    apply_list(R,L1,C1,C2,L2).

iterate(_,[],X,X) :- !.
iterate(R,[X|L],B,B2) :-
    apply(R,[X,B,B1]),
    iterate(R,L,B1,B2).

/* not in use...
flatten([],[]) :- !.
flatten([X|L1],L2) :-
    element(X), !,
    flatten(L1,L3),
    union([X],L3,L2).
flatten([X|L1],L2) :-
    flatten(X,X1),
    flatten(L1,L3),
    union(X1,L3,L2).
*/

element(X) :- \+ list(X).
```

```
list([]).
list([_|_]).

compound(T) :-
    nonvar(T),
    \+ atomic(T), !.

member(X,[X|_]).
member(X,[_|Z]) :- member(X,Z).

writel([]) :- !.
writel([X|L]) :-
    nl, write(X), !,
    write(L).

/* replace(structure,oldv,newv,newstructure) */

replace(N/O,X,N) :-
    O == X, !.
replace(_/_,X,X) :-
    (atomic(X) ; var(X) ; isstar(X)), !.
replace(N/O,S,S1) :-
    compound(S),
    S =.. [F|Ts],
    apply_list1(replace,Ts,N/O,Ts1),
    S1 =.. [F|Ts1].

apply_list1(_,[],_,[]) :- !.
apply_list1(R,[X1|L1],C,[X2|L2]) :-
    apply(R,[C,X1,X2]),
    apply_list1(R,L1,C,L2).

in(X,Y) :-
    X == Y.
in(P,X) :-
    compound(P),
    P =.. [_|Ts],
    apply_or(in,Ts,X).

del(_,[],[]).
del(X,[Y|L],L1) :-
    X==Y, !,
    del(X,L,L1).
del(X,[Y|L],[Y|L1]) :-
    del(X,L,L1).

delete(L1,L2,L3) :-
    iterate(del,L1,L2,L3).

head(H) :- (atom(H) ; integer(H) ; functor(H,F,_), F \== (:-)).

append([],L,L).
append([X|L1],L2,[X|L3]) :-
    append(L1,L2,L3).

appvar(X,L) :- var(X), !,
    X=L.
appvar([_|X],L) :-
    appvar(X,L).
```

```
/* my_builtin - INCOMPLETE */

my_builtin(sum(_,_,_)).
my_builtin(diff(_,_,_)).
my_builtin(prod(_,_,_)).
my_builtin(quot(_,_,_)).
my_builtin(rem(_,_,_)).
my_builtin(eq(_,_)).
my_builtin(ne(_,_)).
my_builtin(gt(_,_)).
my_builtin(ge(_,_)).
my_builtin(le(_,_)).
my_builtin(lt(_,_)).
my_builtin(atom(_)).
my_builtin(int(_)).
my_builtin(var(_)).
my_builtin(skel(_)).
my_builtin(op(_,_,_)).
my_builtin(write(_)).
my_builtin(ax(_,_)).
my_builtin(delax(_)).
my_builtin(fail).
my_builtin('=/'(_,_)).


% test formula set

country(france) <- tt.
country(spain) <- tt.
country(switzerland) <- tt.
border(france, spain) <- tt.
border(france, switzerland) <- tt.
contain(europe, france) <- tt.
contain(europe, C) <- country(C) & border(C, france).

teaches(r, mmk) <- tt.
teaches(r, spv) <- tt.
teaches(t, lp) <- tt.

lessl(a, b) <- tt.
lessl(b, c) <- tt.
lessl(c, d) <- tt.
less(X, Y) <-> lessl(X, Y) # e(Z, lessl(Z, Y) & less(X, Z)).
lessall(X, L) <-> a(Y, listMember(Y, L) => less(X, Y)).

unique(X) <-> a(E1, a(E2, listMember(E1, X) & listMember(E2, X) => E1 = E2)).

u([]) <-> tt.
u([X|L]) <-> a(E, m(E, L)  => E = X).

ul([]) <-> tt.
ul([X|L]) <-> ul(X, L).
ul(_, []) <-> tt.
ul(X, [X|L]) <-> ul(X, L).

m(X, L) <-> e(U, e(L1, L = [U|L1] & ( X = U # m(X, L1)))).

brother(X, Y) <-> male(X) & sibling(X, Y).

uniqMember(X, L) <->
    e(T, L = [X|T] & (memberList(X, T) => ff)) #
    e(T, e(X2, L = [X2|T] & (X = X2 => ff) & uniqMember(X, T))).

uniqUnion(X, Y, Z) <->
    a(E, uniqMember(E, X) # uniqMember(E, Y) <-> uniqMember(E, Z)).

professor(X) <-> X = r # X = t.
```

```
employeeList(D, L) <-> a(X, employee(D, X) <-> listMember(X, L)).
employee(D, X) <->
     (D = cs & (X = r # X = s)) #
     (D = ts & (X = j # X = a)).

el(D, L) <-> a(X, em(D, X) <-> listMember(X, L)).
em(D, X) <->
     (D = cs & X = s) #
     (D = ts & X = a).

listMember(_, []) <-> ff.
listMember(X, [U|R]) <-> X = U # listMember(X, R).

mathMajor(X) <-> a(Y, mathCourse(Y) => takes(X, Y)).
mathCourse(Z) <-> Z = c1 # Z = c3.
takes(X, Y) <->
     (X = d & Y = c3) #
     (X = j & Y = c1) #
     (X = j & Y = c3).

subset(X, Y) <-> a(E, listMember(E, X) => listMember(E, Y)).
equalSets(X, Y) <-> subset(X, Y) & subset(Y, X).
```

```
% op/3 directives

:- op(900, xfx, <->).
:- op(890, xfy, =>).
:- op(880, xfx, <-).
:- op(870, xfy, #).
:- op(860, xfy, &).
:- op(500, xfx, :).
```

.ili.bench

```
# /*
  set-up.ili: bench set-up for ili
  */
ili :- driver(ili).

benchmark(ili, run_ili, run_dummy, 25).

run_ili :- q(_, _, P),
           s(P),
           fail.
run_ili.

run_dummy :- q(_, _, P),
             dummy(P),
             fail.
run_dummy.

show(ili) :- q(I, R, P),
             (s(P) -> R = provable ; R = unprovable),
             write('Query '), write(I), write(': '), write(P), nl,
             write(success), nl, nl,
             fail.
show(ili).


#include "driver"
```

pereira

floating_add.m

```
# /*
  floating_add.m: Pereira benchmark floating_add master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   floating_add
%
%   Fernando C. N. Pereira
%
%   Do 100 floating additions nonrecursively,
%   avoiding obvious compiler optimizations.

#if BENCH
#  include ".floating_add.bench"
#else
floating_add :- fa1(0.1, 1.1, R).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (fa1/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
fa1(_, _, _).
#else
fa1(M, K, P) :- N is M + K, fa2(N, 2.1, P).
fa2(M, K, P) :- N is M + K, fa3(N, 3.1, P).
fa3(M, K, P) :- N is M + K, fa4(N, 4.1, P).
fa4(M, K, P) :- N is M + K, fa5(N, 5.1, P).
fa5(M, K, P) :- N is M + K, fa6(N, 6.1, P).
fa6(M, K, P) :- N is M + K, fa7(N, 7.1, P).
fa7(M, K, P) :- N is M + K, fa8(N, 8.1, P).
fa8(M, K, P) :- N is M + K, fa9(N, 9.1, P).
fa9(M, K, P) :- N is M + K, fa10(N, 10.1, P).
fa10(M, K, P) :- N is M + K, fa11(N, 11.1, P).
fa11(M, K, P) :- N is M + K, fa12(N, 12.1, P).
fa12(M, K, P) :- N is M + K, fa13(N, 13.1, P).
fa13(M, K, P) :- N is M + K, fa14(N, 14.1, P).
fa14(M, K, P) :- N is M + K, fa15(N, 15.1, P).
fa15(M, K, P) :- N is M + K, fa16(N, 16.1, P).
fa16(M, K, P) :- N is M + K, fa17(N, 17.1, P).
fa17(M, K, P) :- N is M + K, fa18(N, 18.1, P).
fa18(M, K, P) :- N is M + K, fa19(N, 19.1, P).
fa19(M, K, P) :- N is M + K, fa20(N, 20.1, P).
fa20(M, K, P) :- N is M + K, fa21(N, 21.1, P).
fa21(M, K, P) :- N is M + K, fa22(N, 22.1, P).
fa22(M, K, P) :- N is M + K, fa23(N, 23.1, P).
fa23(M, K, P) :- N is M + K, fa24(N, 24.1, P).
fa24(M, K, P) :- N is M + K, fa25(N, 25.1, P).
fa25(M, K, P) :- N is M + K, fa26(N, 26.1, P).
fa26(M, K, P) :- N is M + K, fa27(N, 27.1, P).
fa27(M, K, P) :- N is M + K, fa28(N, 28.1, P).
fa28(M, K, P) :- N is M + K, fa29(N, 29.1, P).
fa29(M, K, P) :- N is M + K, fa30(N, 30.1, P).
fa30(M, K, P) :- N is M + K, fa31(N, 31.1, P).
fa31(M, K, P) :- N is M + K, fa32(N, 32.1, P).
fa32(M, K, P) :- N is M + K, fa33(N, 33.1, P).
fa33(M, K, P) :- N is M + K, fa34(N, 34.1, P).
fa34(M, K, P) :- N is M + K, fa35(N, 35.1, P).
fa35(M, K, P) :- N is M + K, fa36(N, 36.1, P).
```

```
fa36(M, K, P) :- N is M + K, fa37(N, 37.1, P).
fa37(M, K, P) :- N is M + K, fa38(N, 38.1, P).
fa38(M, K, P) :- N is M + K, fa39(N, 39.1, P).
fa39(M, K, P) :- N is M + K, fa40(N, 40.1, P).
fa40(M, K, P) :- N is M + K, fa41(N, 41.1, P).
fa41(M, K, P) :- N is M + K, fa42(N, 42.1, P).
fa42(M, K, P) :- N is M + K, fa43(N, 43.1, P).
fa43(M, K, P) :- N is M + K, fa44(N, 44.1, P).
fa44(M, K, P) :- N is M + K, fa45(N, 45.1, P).
fa45(M, K, P) :- N is M + K, fa46(N, 46.1, P).
fa46(M, K, P) :- N is M + K, fa47(N, 47.1, P).
fa47(M, K, P) :- N is M + K, fa48(N, 48.1, P).
fa48(M, K, P) :- N is M + K, fa49(N, 49.1, P).
fa49(M, K, P) :- N is M + K, fa50(N, 50.1, P).
fa50(M, K, P) :- N is M + K, fa51(N, 51.1, P).
fa51(M, K, P) :- N is M + K, fa52(N, 52.1, P).
fa52(M, K, P) :- N is M + K, fa53(N, 53.1, P).
fa53(M, K, P) :- N is M + K, fa54(N, 54.1, P).
fa54(M, K, P) :- N is M + K, fa55(N, 55.1, P).
fa55(M, K, P) :- N is M + K, fa56(N, 56.1, P).
fa56(M, K, P) :- N is M + K, fa57(N, 57.1, P).
fa57(M, K, P) :- N is M + K, fa58(N, 58.1, P).
fa58(M, K, P) :- N is M + K, fa59(N, 59.1, P).
fa59(M, K, P) :- N is M + K, fa60(N, 60.1, P).
fa60(M, K, P) :- N is M + K, fa61(N, 61.1, P).
fa61(M, K, P) :- N is M + K, fa62(N, 62.1, P).
fa62(M, K, P) :- N is M + K, fa63(N, 63.1, P).
fa63(M, K, P) :- N is M + K, fa64(N, 64.1, P).
fa64(M, K, P) :- N is M + K, fa65(N, 65.1, P).
fa65(M, K, P) :- N is M + K, fa66(N, 66.1, P).
fa66(M, K, P) :- N is M + K, fa67(N, 67.1, P).
fa67(M, K, P) :- N is M + K, fa68(N, 68.1, P).
fa68(M, K, P) :- N is M + K, fa69(N, 69.1, P).
fa69(M, K, P) :- N is M + K, fa70(N, 70.1, P).
fa70(M, K, P) :- N is M + K, fa71(N, 71.1, P).
fa71(M, K, P) :- N is M + K, fa72(N, 72.1, P).
fa72(M, K, P) :- N is M + K, fa73(N, 73.1, P).
fa73(M, K, P) :- N is M + K, fa74(N, 74.1, P).
fa74(M, K, P) :- N is M + K, fa75(N, 75.1, P).
fa75(M, K, P) :- N is M + K, fa76(N, 76.1, P).
fa76(M, K, P) :- N is M + K, fa77(N, 77.1, P).
fa77(M, K, P) :- N is M + K, fa78(N, 78.1, P).
fa78(M, K, P) :- N is M + K, fa79(N, 79.1, P).
fa79(M, K, P) :- N is M + K, fa80(N, 80.1, P).
fa80(M, K, P) :- N is M + K, fa81(N, 81.1, P).
fa81(M, K, P) :- N is M + K, fa82(N, 82.1, P).
fa82(M, K, P) :- N is M + K, fa83(N, 83.1, P).
fa83(M, K, P) :- N is M + K, fa84(N, 84.1, P).
fa84(M, K, P) :- N is M + K, fa85(N, 85.1, P).
fa85(M, K, P) :- N is M + K, fa86(N, 86.1, P).
fa86(M, K, P) :- N is M + K, fa87(N, 87.1, P).
fa87(M, K, P) :- N is M + K, fa88(N, 88.1, P).
fa88(M, K, P) :- N is M + K, fa89(N, 89.1, P).
fa89(M, K, P) :- N is M + K, fa90(N, 90.1, P).
fa90(M, K, P) :- N is M + K, fa91(N, 91.1, P).
fa91(M, K, P) :- N is M + K, fa92(N, 92.1, P).
fa92(M, K, P) :- N is M + K, fa93(N, 93.1, P).
fa93(M, K, P) :- N is M + K, fa94(N, 94.1, P).
fa94(M, K, P) :- N is M + K, fa95(N, 95.1, P).
fa95(M, K, P) :- N is M + K, fa96(N, 96.1, P).
fa96(M, K, P) :- N is M + K, fa97(N, 97.1, P).
fa97(M, K, P) :- N is M + K, fa98(N, 98.1, P).
fa98(M, K, P) :- N is M + K, fa99(N, 99.1, P).
fa99(M, K, P) :- N is M + K, fa100(N, 100.1, P).
fa100(M, K, P) :- P is M + K.
#endif
```

integer_add.m

```
# /*
  integer_add.m: Pereira benchmark integer_add master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   integer_add
%
%   Fernando C. N. Pereira
%
%   Do 100 integer additions nonrecursively,
%   avoiding obvious compiler optimizations.

#if BENCH
#  include ".integer_add.bench"
#else
integer_add :- a1(0, 1, R).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (a1/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
a1(_, _, _).
#else
a1(M, K, P) :- N is M + K, a2(N, 2, P).
a2(M, K, P) :- N is M + K, a3(N, 3, P).
a3(M, K, P) :- N is M + K, a4(N, 4, P).
a4(M, K, P) :- N is M + K, a5(N, 5, P).
a5(M, K, P) :- N is M + K, a6(N, 6, P).
a6(M, K, P) :- N is M + K, a7(N, 7, P).
a7(M, K, P) :- N is M + K, a8(N, 8, P).
a8(M, K, P) :- N is M + K, a9(N, 9, P).
a9(M, K, P) :- N is M + K, a10(N, 10, P).
a10(M, K, P) :- N is M + K, a11(N, 11, P).
a11(M, K, P) :- N is M + K, a12(N, 12, P).
a12(M, K, P) :- N is M + K, a13(N, 13, P).
a13(M, K, P) :- N is M + K, a14(N, 14, P).
a14(M, K, P) :- N is M + K, a15(N, 15, P).
a15(M, K, P) :- N is M + K, a16(N, 16, P).
a16(M, K, P) :- N is M + K, a17(N, 17, P).
a17(M, K, P) :- N is M + K, a18(N, 18, P).
a18(M, K, P) :- N is M + K, a19(N, 19, P).
a19(M, K, P) :- N is M + K, a20(N, 20, P).
a20(M, K, P) :- N is M + K, a21(N, 21, P).
a21(M, K, P) :- N is M + K, a22(N, 22, P).
a22(M, K, P) :- N is M + K, a23(N, 23, P).
a23(M, K, P) :- N is M + K, a24(N, 24, P).
a24(M, K, P) :- N is M + K, a25(N, 25, P).
a25(M, K, P) :- N is M + K, a26(N, 26, P).
a26(M, K, P) :- N is M + K, a27(N, 27, P).
a27(M, K, P) :- N is M + K, a28(N, 28, P).
a28(M, K, P) :- N is M + K, a29(N, 29, P).
a29(M, K, P) :- N is M + K, a30(N, 30, P).
a30(M, K, P) :- N is M + K, a31(N, 31, P).
a31(M, K, P) :- N is M + K, a32(N, 32, P).
a32(M, K, P) :- N is M + K, a33(N, 33, P).
a33(M, K, P) :- N is M + K, a34(N, 34, P).
a34(M, K, P) :- N is M + K, a35(N, 35, P).
a35(M, K, P) :- N is M + K, a36(N, 36, P).
```

```
a36(M, K, P) :- N is M + K, a37(N, 37, P).
a37(M, K, P) :- N is M + K, a38(N, 38, P).
a38(M, K, P) :- N is M + K, a39(N, 39, P).
a39(M, K, P) :- N is M + K, a40(N, 40, P).
a40(M, K, P) :- N is M + K, a41(N, 41, P).
a41(M, K, P) :- N is M + K, a42(N, 42, P).
a42(M, K, P) :- N is M + K, a43(N, 43, P).
a43(M, K, P) :- N is M + K, a44(N, 44, P).
a44(M, K, P) :- N is M + K, a45(N, 45, P).
a45(M, K, P) :- N is M + K, a46(N, 46, P).
a46(M, K, P) :- N is M + K, a47(N, 47, P).
a47(M, K, P) :- N is M + K, a48(N, 48, P).
a48(M, K, P) :- N is M + K, a49(N, 49, P).
a49(M, K, P) :- N is M + K, a50(N, 50, P).
a50(M, K, P) :- N is M + K, a51(N, 51, P).
a51(M, K, P) :- N is M + K, a52(N, 52, P).
a52(M, K, P) :- N is M + K, a53(N, 53, P).
a53(M, K, P) :- N is M + K, a54(N, 54, P).
a54(M, K, P) :- N is M + K, a55(N, 55, P).
a55(M, K, P) :- N is M + K, a56(N, 56, P).
a56(M, K, P) :- N is M + K, a57(N, 57, P).
a57(M, K, P) :- N is M + K, a58(N, 58, P).
a58(M, K, P) :- N is M + K, a59(N, 59, P).
a59(M, K, P) :- N is M + K, a60(N, 60, P).
a60(M, K, P) :- N is M + K, a61(N, 61, P).
a61(M, K, P) :- N is M + K, a62(N, 62, P).
a62(M, K, P) :- N is M + K, a63(N, 63, P).
a63(M, K, P) :- N is M + K, a64(N, 64, P).
a64(M, K, P) :- N is M + K, a65(N, 65, P).
a65(M, K, P) :- N is M + K, a66(N, 66, P).
a66(M, K, P) :- N is M + K, a67(N, 67, P).
a67(M, K, P) :- N is M + K, a68(N, 68, P).
a68(M, K, P) :- N is M + K, a69(N, 69, P).
a69(M, K, P) :- N is M + K, a70(N, 70, P).
a70(M, K, P) :- N is M + K, a71(N, 71, P).
a71(M, K, P) :- N is M + K, a72(N, 72, P).
a72(M, K, P) :- N is M + K, a73(N, 73, P).
a73(M, K, P) :- N is M + K, a74(N, 74, P).
a74(M, K, P) :- N is M + K, a75(N, 75, P).
a75(M, K, P) :- N is M + K, a76(N, 76, P).
a76(M, K, P) :- N is M + K, a77(N, 77, P).
a77(M, K, P) :- N is M + K, a78(N, 78, P).
a78(M, K, P) :- N is M + K, a79(N, 79, P).
a79(M, K, P) :- N is M + K, a80(N, 80, P).
a80(M, K, P) :- N is M + K, a81(N, 81, P).
a81(M, K, P) :- N is M + K, a82(N, 82, P).
a82(M, K, P) :- N is M + K, a83(N, 83, P).
a83(M, K, P) :- N is M + K, a84(N, 84, P).
a84(M, K, P) :- N is M + K, a85(N, 85, P).
a85(M, K, P) :- N is M + K, a86(N, 86, P).
a86(M, K, P) :- N is M + K, a87(N, 87, P).
a87(M, K, P) :- N is M + K, a88(N, 88, P).
a88(M, K, P) :- N is M + K, a89(N, 89, P).
a89(M, K, P) :- N is M + K, a90(N, 90, P).
a90(M, K, P) :- N is M + K, a91(N, 91, P).
a91(M, K, P) :- N is M + K, a92(N, 92, P).
a92(M, K, P) :- N is M + K, a93(N, 93, P).
a93(M, K, P) :- N is M + K, a94(N, 94, P).
a94(M, K, P) :- N is M + K, a95(N, 95, P).
a95(M, K, P) :- N is M + K, a96(N, 96, P).
a96(M, K, P) :- N is M + K, a97(N, 97, P).
a97(M, K, P) :- N is M + K, a98(N, 98, P).
a98(M, K, P) :- N is M + K, a99(N, 99, P).
a99(M, K, P) :- N is M + K, a100(N, 100, P).
a100(M, K, P) :- P is M + K.
#endif
```

```
# /*
  arg_1.m: Pereira benchmark (arg) arg_1 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (arg) arg_1
%
%   Fernando C. N. Pereira
%
%   100 calls to arg at position 1

#if BENCH
#  include ".arg_1.bench"
#else
arg_1 :- complex_nary_term(100, 1, Term),
         arg1(1, Term, _).
#endif

#include "arg"          /* code for arg */
```

```
# /*
  arg_2.m: Pereira benchmark (arg) arg_2 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (arg) arg_2
%
%   Fernando C. N. Pereira
%
%   100 calls to arg at position 2

#if BENCH
#  include ".arg_2.bench"
#else
arg_2 :- complex_nary_term(100, 2, Term),
         arg1(2, Term, _).
#endif

#include "arg"          /* code for arg */
```

arg_4.m

```
# /*
  arg_4.m: Pereira benchmark (arg) arg_4 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (arg) arg_4
%
%   Fernando C. N. Pereira
%
%   100 calls to arg at position 4

#if BENCH
#  include ".arg_4.bench"
#else
arg_4 :- complex_nary_term(100, 4, Term),
         arg1(4, Term, _).
#endif

#include "arg"          /* code for arg */
```

```
# /*
  arg_8.m: Pereira benchmark (arg) arg_8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (arg) arg_8
%
%   Fernando C. N. Pereira
%
%   100 calls to arg at position 8

#if BENCH
#  include ".arg_8.bench"
#else
arg_8 :- complex_nary_term(100, 8, Term),
         arg1(8, Term, _).
#endif

#include "arg"          /* code for arg */
```

arg_16.m

```
# /*
  arg_16.m: Pereira benchmark (arg) arg_16 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (arg) arg_16
%
%   Fernando C. N. Pereira
%
%   100 calls to arg at position 16

#if BENCH
#  include ".arg_16.bench"
#else
arg_16 :- complex_nary_term(100, 16, Term),
          arg1(16, Term, _).
#endif

#include "arg"          /* code for arg */
```

```
# /*
  arg: Pereira code for 100 calls to arg at position N
  */
complex_nary_term(0, N, N) :- !.
complex_nary_term(I, N, Term) :-
   I > 0, J is I - 1,
   complex_nary_term(J, N, SubTerm),
   nary_term(N, SubTerm, Term).

nary_term(N, SubTerm, Term) :-
   functor(Term, f, N),
   fill_nary_term(N, SubTerm, Term).

fill_nary_term(0, _, _) :- !.
fill_nary_term(N, SubTerm, Term) :-
   N > 0, M is N - 1,
   arg(N, Term, SubTerm),
   fill_nary_term(M, SubTerm, Term).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (arg1/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
arg1(_, _, _).
#else
arg1(N, T, R) :- arg(N, T, X), arg2(N, X, R).
arg2(N, T, R) :- arg(N, T, X), arg3(N, X, R).
arg3(N, T, R) :- arg(N, T, X), arg4(N, X, R).
arg4(N, T, R) :- arg(N, T, X), arg5(N, X, R).
arg5(N, T, R) :- arg(N, T, X), arg6(N, X, R).
arg6(N, T, R) :- arg(N, T, X), arg7(N, X, R).
arg7(N, T, R) :- arg(N, T, X), arg8(N, X, R).
arg8(N, T, R) :- arg(N, T, X), arg9(N, X, R).
arg9(N, T, R) :- arg(N, T, X), arg10(N, X, R).
arg10(N, T, R) :- arg(N; T, X), arg11(N, X, R).
arg11(N, T, R) :- arg(N, T, X), arg12(N, X, R).
arg12(N, T, R) :- arg(N, T, X), arg13(N, X, R).
arg13(N, T, R) :- arg(N, T, X), arg14(N, X, R).
arg14(N, T, R) :- arg(N, T, X), arg15(N, X, R).
arg15(N, T, R) :- arg(N, T, X), arg16(N, X, R).
arg16(N, T, R) :- arg(N, T, X), arg17(N, X, R).
arg17(N, T, R) :- arg(N, T, X), arg18(N, X, R).
arg18(N, T, R) :- arg(N, T, X), arg19(N, X, R).
arg19(N, T, R) :- arg(N, T, X), arg20(N, X, R).
arg20(N, T, R) :- arg(N, T, X), arg21(N, X, R).
arg21(N, T, R) :- arg(N, T, X), arg22(N, X, R).
arg22(N, T, R) :- arg(N, T, X), arg23(N, X, R).
arg23(N, T, R) :- arg(N, T, X), arg24(N, X, R).
arg24(N, T, R) :- arg(N, T, X), arg25(N, X, R).
arg25(N, T, R) :- arg(N, T, X), arg26(N, X, R).
arg26(N, T, R) :- arg(N, T, X), arg27(N, X, R).
arg27(N, T, R) :- arg(N, T, X), arg28(N, X, R).
arg28(N, T, R) :- arg(N, T, X), arg29(N, X, R).
arg29(N, T, R) :- arg(N, T, X), arg30(N, X, R).
arg30(N, T, R) :- arg(N, T, X), arg31(N, X, R).
arg31(N, T, R) :- arg(N, T, X), arg32(N, X, R).
arg32(N, T, R) :- arg(N, T, X), arg33(N, X, R).
arg33(N, T, R) :- arg(N, T, X), arg34(N, X, R).
arg34(N, T, R) :- arg(N, T, X), arg35(N, X, R).
arg35(N, T, R) :- arg(N, T, X), arg36(N, X, R).
```

```
arg36(N, T, R) :- arg(N, T, X), arg37(N, X, R).
arg37(N, T, R) :- arg(N, T, X), arg38(N, X, R).
arg38(N, T, R) :- arg(N, T, X), arg39(N, X, R).
arg39(N, T, R) :- arg(N, T, X), arg40(N, X, R).
arg40(N, T, R) :- arg(N, T, X), arg41(N, X, R).
arg41(N, T, R) :- arg(N, T, X), arg42(N, X, R).
arg42(N, T, R) :- arg(N, T, X), arg43(N, X, R).
arg43(N, T, R) :- arg(N, T, X), arg44(N, X, R).
arg44(N, T, R) :- arg(N, T, X), arg45(N, X, R).
arg45(N, T, R) :- arg(N, T, X), arg46(N, X, R).
arg46(N, T, R) :- arg(N, T, X), arg47(N, X, R).
arg47(N, T, R) :- arg(N, T, X), arg48(N, X, R).
arg48(N, T, R) :- arg(N, T, X), arg49(N, X, R).
arg49(N, T, R) :- arg(N, T, X), arg50(N, X, R).
arg50(N, T, R) :- arg(N, T, X), arg51(N, X, R).
arg51(N, T, R) :- arg(N, T, X), arg52(N, X, R).
arg52(N, T, R) :- arg(N, T, X), arg53(N, X, R).
arg53(N, T, R) :- arg(N, T, X), arg54(N, X, R).
arg54(N, T, R) :- arg(N, T, X), arg55(N, X, R).
arg55(N, T, R) :- arg(N, T, X), arg56(N, X, R).
arg56(N, T, R) :- arg(N, T, X), arg57(N, X, R).
arg57(N, T, R) :- arg(N, T, X), arg58(N, X, R).
arg58(N, T, R) :- arg(N, T, X), arg59(N, X, R).
arg59(N, T, R) :- arg(N, T, X), arg60(N, X, R).
arg60(N, T, R) :- arg(N, T, X), arg61(N, X, R).
arg61(N, T, R) :- arg(N, T, X), arg62(N, X, R).
arg62(N, T, R) :- arg(N, T, X), arg63(N, X, R).
arg63(N, T, R) :- arg(N, T, X), arg64(N, X, R).
arg64(N, T, R) :- arg(N, T, X), arg65(N, X, R).
arg65(N, T, R) :- arg(N, T, X), arg66(N, X, R).
arg66(N, T, R) :- arg(N, T, X), arg67(N, X, R).
arg67(N, T, R) :- arg(N, T, X), arg68(N, X, R).
arg68(N, T, R) :- arg(N, T, X), arg69(N, X, R).
arg69(N, T, R) :- arg(N, T, X), arg70(N, X, R).
arg70(N, T, R) :- arg(N, T, X), arg71(N, X, R).
arg71(N, T, R) :- arg(N, T, X), arg72(N, X, R).
arg72(N, T, R) :- arg(N, T, X), arg73(N, X, R).
arg73(N, T, R) :- arg(N, T, X), arg74(N, X, R).
arg74(N, T, R) :- arg(N, T, X), arg75(N, X, R).
arg75(N, T, R) :- arg(N, T, X), arg76(N, X, R).
arg76(N, T, R) :- arg(N, T, X), arg77(N, X, R).
arg77(N, T, R) :- arg(N, T, X), arg78(N, X, R).
arg78(N, T, R) :- arg(N, T, X), arg79(N, X, R).
arg79(N, T, R) :- arg(N, T, X), arg80(N, X, R).
arg80(N, T, R) :- arg(N, T, X), arg81(N, X, R).
arg81(N, T, R) :- arg(N, T, X), arg82(N, X, R).
arg82(N, T, R) :- arg(N, T, X), arg83(N, X, R).
arg83(N, T, R) :- arg(N, T, X), arg84(N, X, R).
arg84(N, T, R) :- arg(N, T, X), arg85(N, X, R).
arg85(N, T, R) :- arg(N, T, X), arg86(N, X, R).
arg86(N, T, R) :- arg(N, T, X), arg87(N, X, R).
arg87(N, T, R) :- arg(N, T, X), arg88(N, X, R).
arg88(N, T, R) :- arg(N, T, X), arg89(N, X, R).
arg89(N, T, R) :- arg(N, T, X), arg90(N, X, R).
arg90(N, T, R) :- arg(N, T, X), arg91(N, X, R).
arg91(N, T, R) :- arg(N, T, X), arg92(N, X, R).
arg92(N, T, R) :- arg(N, T, X), arg93(N, X, R).
arg93(N, T, R) :- arg(N, T, X), arg94(N, X, R).
arg94(N, T, R) :- arg(N, T, X), arg95(N, X, R).
arg95(N, T, R) :- arg(N, T, X), arg96(N, X, R).
arg96(N, T, R) :- arg(N, T, X), arg97(N, X, R).
arg97(N, T, R) :- arg(N, T, X), arg98(N, X, R).
arg98(N, T, R) :- arg(N, T, X), arg99(N, X, R).
arg99(N, T, R) :- arg(N, T, X), arg100(N, X, R).
arg100(N, T, R) :- arg(N, T, R).
#endif
```

```
# /*
  assert_unit.m: Pereira benchmark assert_unit master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   assert_unit
%
%   Fernando C. N. Pereira
%
%   Assert 1000 unit clauses.

#if BENCH
#  include ".assert_unit.bench"
#else
assert_unit :- abolish(ua, 3),
               create_units(1, 1000, L),
               assert_clauses(L).
#endif

create_units(I, N, []) :- I > N, !.
create_units(I, N, [ua(K, X, f(K, X))|Rest]) :-
    K is I * (1 + I//100),
    J is I + 1,
    create_units(J, N, Rest).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (assert_clauses/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
assert_clauses(_).
#else
assert_clauses([]).
assert_clauses([Clause|Rest]) :-
   assert(Clause),
   assert_clauses(Rest).
#endif
```

```
# /*
  access_unit.m: Pereira benchmark access_unit master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   access_unit
%
%   Fernando C. N. Pereira
%
%   Access 100 (dynamic) unit clauses with 1st argument instantiated.

#if BENCH
#  include ".access_unit.bench"
#else
access_unit :- abolish(dix, 2),
               dix_clauses(1, 100, L),
               assert_clauses(L),
               access_dix(1, 1).
#endif

dix_clauses(I, N, []) :- I > N, !.
dix_clauses(I, N, [dix(P, Q) | L]) :-
   I =< N,
   P is I*I,
   R is 1 + (I+N-2) mod N,
   Q is R*R,
   J is I + 1,
   dix_clauses(J, N, L).

assert_clauses([]).
assert_clauses([Clause|Rest]) :-
   assert(Clause),
   assert_clauses(Rest).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (access_dix/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
access_dix(_, _).
#else
access_dix(Start, End) :-
   dix(Start, Where),
   (  Where = End, !
   ;  access_dix(Where, End)
   ).
#endif
```

slow_access_unit.m

```
# /*
  slow_access_unit.m: Pereira benchmark slow_access_unit master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   slow_access_unit
%
%   Fernando C. N. Pereira
%
%   Access 100 dynamic clauses with 2nd argument instantiated.

#if BENCH
#  include ".slow_access_unit.bench"
#else
slow_access_unit :- abolish(dix, 2),
                    dix_clauses(1, 100, L),
                    assert_clauses(L),
                    access_back(1, 1).
#endif

dix_clauses(I, N, []) :- I > N, !.
dix_clauses(I, N, [dix(P, Q) | L]) :-
   I =< N,
   P is I*I,
   R is 1 + (I+N-2) mod N,
   Q is R*R,
   J is I + 1,
   dix_clauses(J, N, L).

assert_clauses([]).
assert_clauses([Clause|Rest]) :-
   assert(Clause),
   assert_clauses(Rest).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (access_back/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
access_back(_, _).
#else
access_back(Start, End) :-
   dix(Where, Start),
   (  Where = End, !
   ;  access_back(Where, End)
   ).
#endif
```

```
# /*
  shallow_backtracking.m: Pereira benchmark shallow_backtracking master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   shallow_backtracking
%
%   Fernando C. N. Pereira
%
%   99 shallow failures (assumes no indexing on second argument).

#if BENCH
#  include ".shallow_backtracking.bench"
#else
shallow_backtracking :- shallow.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (shallow/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
shallow.
#halt
#endif
shallow :- b(_X, 100).

#include "b"
```

deep_backtracking.m

```
# /*
  deep_backtracking.m: Pereira benchmark deep_backtracking master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   deep_backtracking
%
%   Fernando C. N. Pereira
%
%   99 deep failures.

#if BENCH
#  include ".deep_backtracking.bench"
#else
deep_backtracking :- deep.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (deep/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
deep.
#halt
#endif
deep :- b(_X, Y), Y = 100.

#include "b"
```

```
# /*
  b: (Pereira) b/2 for shallow_backtracking and deep_backtracking
  */
b(_X, 1).
b(_X, 2).
b(_X, 3).
b(_X, 4).
b(_X, 5).
b(_X, 6).
b(_X, 7).
b(_X, 8).
b(_X, 9).
b(_X, 10).
b(_X, 11).
b(_X, 12).
b(_X, 13).
b(_X, 14).
b(_X, 15).
b(_X, 16).
b(_X, 17).
b(_X, 18).
b(_X, 19).
b(_X, 20).
b(_X, 21).
b(_X, 22).
b(_X, 23).
b(_X, 24).
b(_X, 25).
b(_X, 26).
b(_X, 27).
b(_X, 28).
b(_X, 29).
b(_X, 30).
b(_X, 31).
b(_X, 32).
b(_X, 33).
b(_X, 34).
b(_X, 35).
b(_X, 36).
b(_X, 37).
b(_X, 38).
b(_X, 39).
b(_X, 40).
b(_X, 41).
b(_X, 42).
b(_X, 43).
b(_X, 44).
b(_X, 45).
b(_X, 46).
b(_X, 47).
b(_X, 48).
b(_X, 49).
b(_X, 50).
```

```
b(_X, 51).
b(_X, 52).
b(_X, 53).
b(_X, 54).
b(_X, 55).
b(_X, 56).
b(_X, 57).
b(_X, 58).
b(_X, 59).
b(_X, 60).
b(_X, 61).
b(_X, 62).
b(_X, 63).
b(_X, 64).
b(_X, 65).
b(_X, 66).
b(_X, 67).
b(_X, 68).
b(_X, 69).
b(_X, 70).
b(_X, 71).
b(_X, 72).
b(_X, 73).
b(_X, 74).
b(_X, 75).
b(_X, 76).
b(_X, 77).
b(_X, 78).
b(_X, 79).
b(_X, 80).
b(_X, 81).
b(_X, 82).
b(_X, 83).
b(_X, 84).
b(_X, 85).
b(_X, 86).
b(_X, 87).
b(_X, 88).
b(_X, 89).
b(_X, 90).
b(_X, 91).
b(_X, 92).
b(_X, 93).
b(_X, 94).
b(_X, 95).
b(_X, 96).
b(_X, 97).
b(_X, 98).
b(_X, 99).
b(_X, 100).
```

```
# /*
  tail_call_atom_atom.m: Pereira benchmark tail_call_atom_atom master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   tail_call_atom_atom
%
%   Fernando C. N. Pereira
%
%   100 determinate tail calls

#if BENCH
#  include ".tail_call_atom_atom.bench"
#else
tail_call_atom_atom :- p1(a).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (p1/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
p1(_).
#else
p1(a) :- p2(a).
p2(a) :- p3(a).
p3(a) :- p4(a).
p4(a) :- p5(a).
p5(a) :- p6(a).
p6(a) :- p7(a).
p7(a) :- p8(a).
p8(a) :- p9(a).
p9(a) :- p10(a).
p10(a) :- p11(a).
p11(a) :- p12(a).
p12(a) :- p13(a).
p13(a) :- p14(a).
p14(a) :- p15(a).
p15(a) :- p16(a).
p16(a) :- p17(a).
p17(a) :- p18(a).
p18(a) :- p19(a).
p19(a) :- p20(a).
p20(a) :- p21(a).
p21(a) :- p22(a).
p22(a) :- p23(a).
p23(a) :- p24(a).
p24(a) :- p25(a).
p25(a) :- p26(a).
p26(a) :- p27(a).
p27(a) :- p28(a).
p28(a) :- p29(a).
p29(a) :- p30(a).
p30(a) :- p31(a).
p31(a) :- p32(a).
p32(a) :- p33(a).
p33(a) :- p34(a).
p34(a) :- p35(a).
p35(a) :- p36(a).
```

```
p36(a) :- p37(a).
p37(a) :- p38(a).
p38(a) :- p39(a).
p39(a) :- p40(a).
p40(a) :- p41(a).
p41(a) :- p42(a).
p42(a) :- p43(a).
p43(a) :- p44(a).
p44(a) :- p45(a).
p45(a) :- p46(a).
p46(a) :- p47(a).
p47(a) :- p48(a).
p48(a) :- p49(a).
p49(a) :- p50(a).
p50(a) :- p51(a).
p51(a) :- p52(a).
p52(a) :- p53(a).
p53(a) :- p54(a).
p54(a) :- p55(a).
p55(a) :- p56(a).
p56(a) :- p57(a).
p57(a) :- p58(a).
p58(a) :- p59(a).
p59(a) :- p60(a).
p60(a) :- p61(a).
p61(a) :- p62(a).
p62(a) :- p63(a).
p63(a) :- p64(a).
p64(a) :- p65(a).
p65(a) :- p66(a).
p66(a) :- p67(a).
p67(a) :- p68(a).
p68(a) :- p69(a).
p69(a) :- p70(a).
p70(a) :- p71(a).
p71(a) :- p72(a).
p72(a) :- p73(a).
p73(a) :- p74(a).
p74(a) :- p75(a).
p75(a) :- p76(a).
p76(a) :- p77(a).
p77(a) :- p78(a).
p78(a) :- p79(a).
p79(a) :- p80(a).
p80(a) :- p81(a).
p81(a) :- p82(a).
p82(a) :- p83(a).
p83(a) :- p84(a).
p84(a) :- p85(a).
p85(a) :- p86(a).
p86(a) :- p87(a).
p87(a) :- p88(a).
p88(a) :- p89(a).
p89(a) :- p90(a).
p90(a) :- p91(a).
p91(a) :- p92(a).
p92(a) :- p93(a).
p93(a) :- p94(a).
p94(a) :- p95(a).
p95(a) :- p96(a).
p96(a) :- p97(a).
p97(a) :- p98(a).
p98(a) :- p99(a).
p99(a) :- p100(a).
p100(a).
#endif
```

binary_call_atom_atom.m

```
# /*
  binary_call_atom_atom.m: Pereira benchmark binary_call_atom_atom master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   binary_call_atom_atom
%
%   Fernando C. N. Pereira
%
%   63 determinate nontail calls, 64 determinate tail calls.

#if BENCH
#  include ".binary_call_atom_atom.bench"
#else
binary_call_atom_atom :- q1(a).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (q1/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
q1(_).
#else
q1(a) :- q2(a), q3(a).
q2(a) :- q4(a), q5(a).
q3(a) :- q6(a), q7(a).
q4(a) :- q8(a), q9(a).
q5(a) :- q10(a), q11(a).
q6(a) :- q12(a), q13(a).
q7(a) :- q14(a), q15(a).
q8(a) :- q16(a), q17(a).
q9(a) :- q18(a), q19(a).
q10(a) :- q20(a), q21(a).
q11(a) :- q22(a), q23(a).
q12(a) :- q24(a), q25(a).
q13(a) :- q26(a), q27(a).
q14(a) :- q28(a), q29(a).
q15(a) :- q30(a), q31(a).
q16(a) :- q32(a), q33(a).
q17(a) :- q34(a), q35(a).
q18(a) :- q36(a), q37(a).
q19(a) :- q38(a), q39(a).
q20(a) :- q40(a), q41(a).
q21(a) :- q42(a), q43(a).
q22(a) :- q44(a), q45(a).
q23(a) :- q46(a), q47(a).
q24(a) :- q48(a), q49(a).
q25(a) :- q50(a), q51(a).
q26(a) :- q52(a), q53(a).
q27(a) :- q54(a), q55(a).
q28(a) :- q56(a), q57(a).
q29(a) :- q58(a), q59(a).
q30(a) :- q60(a), q61(a).
q31(a) :- q62(a), q63(a).
q32(a) :- q64(a), q65(a).
q33(a) :- q66(a), q67(a).
q34(a) :- q68(a), q69(a).
q35(a) :- q70(a), q71(a).
```

```
q36(a) :- q72(a), q73(a).
q37(a) :- q74(a), q75(a).
q38(a) :- q76(a), q77(a).
q39(a) :- q78(a), q79(a).
q40(a) :- q80(a), q81(a).
q41(a) :- q82(a), q83(a).
q42(a) :- q84(a), q85(a).
q43(a) :- q86(a), q87(a).
q44(a) :- q88(a), q89(a).
q45(a) :- q90(a), q91(a).
q46(a) :- q92(a), q93(a).
q47(a) :- q94(a), q95(a).
q48(a) :- q96(a), q97(a).
q49(a) :- q98(a), q99(a).
q50(a) :- q100(a), q101(a).
q51(a) :- q102(a), q103(a).
q52(a) :- q104(a), q105(a).
q53(a) :- q106(a), q107(a).
q54(a) :- q108(a), q109(a).
q55(a) :- q110(a), q111(a).
q56(a) :- q112(a), q113(a).
q57(a) :- q114(a), q115(a).
q58(a) :- q116(a), q117(a).
q59(a) :- q118(a), q119(a).
q60(a) :- q120(a), q121(a).
q61(a) :- q122(a), q123(a).
q62(a) :- q124(a), q125(a).
q63(a) :- q126(a), q127(a).
q64(a).
q65(a).
q66(a).
q67(a).
q68(a).
q69(a).
q70(a).
q71(a).
q72(a).
q73(a).
q74(a).
q75(a).
q76(a).
q77(a).
q78(a).
q79(a).
q80(a).
q81(a).
q82(a).
q83(a).
q84(a).
q85(a).
q86(a).
q87(a).
q88(a).
q89(a).
q90(a).
q91(a).
q92(a).
q93(a).
q94(a).
q95(a).
q96(a).
q97(a).
q98(a).
q99(a).
q100(a).
```

```
q101(a).
q102(a).
q103(a).
q104(a).
q105(a).
q106(a).
q107(a).
q108(a).
q109(a).
q110(a).
q111(a).
q112(a).
q113(a).
q114(a).
q115(a).
q116(a).
q117(a).
q118(a).
q119(a).
q120(a).
q121(a).
q122(a).
q123(a).
q124(a).
q125(a).
q126(a).
q127(a).
#endif
```

choice_point.m

```
# /*
  choice_point.m: Pereira benchmark choice_point master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   choice_point
%
%   Fernando C. N. Pereira
%
%   Create 100 choice points (assumes no clever multi-predicate optimizer).

#if BENCH
#  include ".choice_point.bench"
#else
choice_point :- choice.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (choice/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
choice.
#else
choice :- c1(a), !.

c1(a) :- c2(a).
c1(a).
c2(a) :- c3(a).
c2(a).
c3(a) :- c4(a).
c3(a).
c4(a) :- c5(a).
c4(a).
c5(a) :- c6(a).
c5(a).
c6(a) :- c7(a).
c6(a).
c7(a) :- c8(a).
c7(a).
c8(a) :- c9(a).
c8(a).
c9(a) :- c10(a).
c9(a).
c10(a) :- c11(a).
c10(a).
c11(a) :- c12(a).
c11(a).
c12(a) :- c13(a).
c12(a).
c13(a) :- c14(a).
c13(a).
c14(a) :- c15(a).
c14(a).
c15(a) :- c16(a).
c15(a).
```

```
c16(a) :- c17(a).
c16(a).
c17(a) :- c18(a).
c17(a).
c18(a) :- c19(a).
c18(a).
c19(a) :- c20(a).
c19(a).
c20(a) :- c21(a).
c20(a).
c21(a) :- c22(a).
c21(a).
c22(a) :- c23(a).
c22(a).
c23(a) :- c24(a).
c23(a).
c24(a) :- c25(a).
c24(a).
c25(a) :- c26(a).
c25(a).
c26(a) :- c27(a).
c26(a).
c27(a) :- c28(a).
c27(a).
c28(a) :- c29(a).
c28(a).
c29(a) :- c30(a).
c29(a).
c30(a) :- c31(a).
c30(a).
c31(a) :- c32(a).
c31(a).
c32(a) :- c33(a).
c32(a).
c33(a) :- c34(a).
c33(a).
c34(a) :- c35(a).
c34(a).
c35(a) :- c36(a).
c35(a).
c36(a) :- c37(a).
c36(a).
c37(a) :- c38(a).
c37(a).
c38(a) :- c39(a).
c38(a).
c39(a) :- c40(a).
c39(a).
c40(a) :- c41(a).
c40(a).
c41(a) :- c42(a).
c41(a).
c42(a) :- c43(a).
c42(a).
c43(a) :- c44(a).
c43(a).
c44(a) :- c45(a).
c44(a).
c45(a) :- c46(a).
c45(a).
c46(a) :- c47(a).
c46(a).
c47(a) :- c48(a).
c47(a).
c48(a) :- c49(a).
c48(a).
c49(a) :- c50(a).
c49(a).
c50(a) :- c51(a).
c50(a).
```

```
c51(a) :- c52(a).
c51(a).
c52(a) :- c53(a).
c52(a).
c53(a) :- c54(a).
c53(a).
c54(a) :- c55(a).
c54(a).
c55(a) :- c56(a).
c55(a).
c56(a) :- c57(a).
c56(a).
c57(a) :- c58(a).
c57(a).
c58(a) :- c59(a).
c58(a).
c59(a) :- c60(a).
c59(a).
c60(a) :- c61(a).
c60(a).
c61(a) :- c62(a).
c61(a).
c62(a) :- c63(a).
c62(a).
c63(a) :- c64(a).
c63(a).
c64(a) :- c65(a).
c64(a).
c65(a) :- c66(a).
c65(a).
c66(a) :- c67(a).
c66(a).
c67(a) :- c68(a).
c67(a).
c68(a) :- c69(a).
c68(a).
c69(a) :- c70(a).
c69(a).
c70(a) :- c71(a).
c70(a).
c71(a) :- c72(a).
c71(a).
c72(a) :- c73(a).
c72(a).
c73(a) :- c74(a).
c73(a).
c74(a) :- c75(a).
c74(a).
c75(a) :- c76(a).
c75(a).
c76(a) :- c77(a).
c76(a).
c77(a) :- c78(a).
c77(a).
c78(a) :- c79(a).
c78(a).
c79(a) :- c80(a).
c79(a).
c80(a) :- c81(a).
c80(a).
c81(a) :- c82(a).
c81(a).
c82(a) :- c83(a).
c82(a).
c83(a) :- c84(a).
c83(a).
c84(a) :- c85(a).
c84(a).
c85(a) :- c86(a).
c85(a).
```

```
c86(a) :- c87(a).
c86(a).
c87(a) :- c88(a).
c87(a).
c88(a) :- c89(a).
c88(a).
c89(a) :- c90(a).
c89(a).
c90(a) :- c91(a).
c90(a).
c91(a) :- c92(a).
c91(a).
c92(a) :- c93(a).
c92(a).
c93(a) :- c94(a).
c93(a).
c94(a) :- c95(a).
c94(a).
c95(a) :- c96(a).
c95(a).
c96(a) :- c97(a).
c96(a).
c97(a) :- c98(a).
c97(a).
c98(a) :- c99(a).
c98(a).
c99(a) :- c100(a).
c99(a).
c100(a).
c100(a).
#endif
```

```
# /*
  trail_variables.m: Pereira benchmark trail_variables master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   trail_variables
%
%   Fernando C. N. Pereira
%
%   Create 100 choice points and trail 100 variables.

#if BENCH
#  include ".trail_variables.bench"
#else
trail_variables :- trail.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (trail/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
trail.
#else
trail :- t1(_X), !.

t1(a) :- t2(_X).
t1(b).
t2(a) :- t3(_X).
t2(b).
t3(a) :- t4(_X).
t3(b).
t4(a) :- t5(_X).
t4(b).
t5(a) :- t6(_X).
t5(b).
t6(a) :- t7(_X).
t6(b).
t7(a) :- t8(_X).
t7(b).
t8(a) :- t9(_X).
t8(b).
t9(a) :- t10(_X).
t9(b).
t10(a) :- t11(_X).
t10(b).
t11(a) :- t12(_X).
t11(b).
t12(a) :- t13(_X).
t12(b).
t13(a) :- t14(_X).
t13(b).
t14(a) :- t15(_X).
t14(b).
t15(a) :- t16(_X).
t15(b).
```

```
t16(a) :- t17(_X).
t16(b).
t17(a) :- t18(_X).
t17(b).
t18(a) :- t19(_X).
t18(b).
t19(a) :- t20(_X).
t19(b).
t20(a) :- t21(_X).
t20(b).
t21(a) :- t22(_X).
t21(b).
t22(a) :- t23(_X).
t22(b).
t23(a) :- t24(_X).
t23(b).
t24(a) :- t25(_X).
t24(b).
t25(a) :- t26(_X).
t25(b).
t26(a) :- t27(_X).
t26(b).
t27(a) :- t28(_X).
t27(b).
t28(a) :- t29(_X).
t28(b).
t29(a) :- t30(_X).
t29(b).
t30(a) :- t31(_X).
t30(b).
t31(a) :- t32(_X).
t31(b).
t32(a) :- t33(_X).
t32(b).
t33(a) :- t34(_X).
t33(b).
t34(a) :- t35(_X).
t34(b).
t35(a) :- t36(_X).
t35(b).
t36(a) :- t37(_X).
t36(b).
t37(a) :- t38(_X).
t37(b).
t38(a) :- t39(_X).
t38(b).
t39(a) :- t40(_X).
t39(b).
t40(a) :- t41(_X).
t40(b).
t41(a) :- t42(_X).
t41(b).
t42(a) :- t43(_X).
t42(b).
t43(a) :- t44(_X).
t43(b).
t44(a) :- t45(_X).
t44(b).
t45(a) :- t46(_X).
t45(b).
t46(a) :- t47(_X).
t46(b).
t47(a) :- t48(_X).
t47(b).
t48(a) :- t49(_X).
t48(b).
t49(a) :- t50(_X).
t49(b).
t50(a) :- t51(_X).
t50(b).
```

```
t51(a) :- t52(_X).
t51(b).
t52(a) :- t53(_X).
t52(b).
t53(a) :- t54(_X).
t53(b).
t54(a) :- t55(_X).
t54(b).
t55(a) :- t56(_X).
t55(b).
t56(a) :- t57(_X).
t56(b).
t57(a) :- t58(_X).
t57(b).
t58(a) :- t59(_X).
t58(b).
t59(a) :- t60(_X).
t59(b).
t60(a) :- t61(_X).
t60(b).
t61(a) :- t62(_X).
t61(b).
t62(a) :- t63(_X).
t62(b).
t63(a) :- t64(_X).
t63(b).
t64(a) :- t65(_X).
t64(b).
t65(a) :- t66(_X).
t65(b).
t66(a) :- t67(_X).
t66(b).
t67(a) :- t68(_X).
t67(b).
t68(a) :- t69(_X).
t68(b).
t69(a) :- t70(_X).
t69(b).
t70(a) :- t71(_X).
t70(b).
t71(a) :- t72(_X).
t71(b).
t72(a) :- t73(_X).
t72(b).
t73(a) :- t74(_X).
t73(b).
t74(a) :- t75(_X).
t74(b).
t75(a) :- t76(_X).
t75(b).
t76(a) :- t77(_X).
t76(b).
t77(a) :- t78(_X).
t77(b).
t78(a) :- t79(_X).
t78(b).
t79(a) :- t80(_X).
t79(b).
t80(a) :- t81(_X).
t80(b).
t81(a) :- t82(_X).
t81(b).
t82(a) :- t83(_X).
t82(b).
t83(a) :- t84(_X).
t83(b).
t84(a) :- t85(_X).
t84(b).
t85(a) :- t86(_X).
t85(b).
```

```
t86(a) :- t87(_X).
t86(b).
t87(a) :- t88(_X).
t87(b).
t88(a) :- t89(_X).
t88(b).
t89(a) :- t90(_X).
t89(b).
t90(a) :- t91(_X).
t90(b).
t91(a) :- t92(_X).
t91(b).
t92(a) :- t93(_X).
t92(b).
t93(a) :- t94(_X).
t93(b).
t94(a) :- t95(_X).
t94(b).
t95(a) :- t96(_X).
t95(b).
t96(a) :- t97(_X).
t96(b).
t97(a) :- t98(_X).
t97(b).
t98(a) :- t99(_X).
t98(b).
t99(a) :- t100(_X).
t99(b).
t100(a).
t100(b).
#endif
```

```
# /*
  index.m: Pereira benchmark index master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   index
%
%   Fernando C. N. Pereira
%
%   100 first-argument-determinate calls; some systems may
%   need extra declarations to index on the first argument.

#if BENCH
#  include ".index.bench"
#else
index :- ix(1).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (ix/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
ix(_).
#else
ix(1) :- ix(10000).
ix(4).
ix(9) :- ix(4).
ix(16) :- ix(9).
ix(25) :- ix(16).
ix(36) :- ix(25).
ix(49) :- ix(36).
ix(64) :- ix(49).
ix(81) :- ix(64).
ix(100) :- ix(81).
ix(121) :- ix(100).
ix(144) :- ix(121).
ix(169) :- ix(144).
ix(196) :- ix(169).
ix(225) :- ix(196).
ix(256) :- ix(225).
ix(289) :- ix(256).
ix(324) :- ix(289).
ix(361) :- ix(324).
ix(400) :- ix(361).
ix(441) :- ix(400).
ix(484) :- ix(441).
ix(529) :- ix(484).
ix(576) :- ix(529).
ix(625) :- ix(576).
ix(676) :- ix(625).
ix(729) :- ix(676).
ix(784) :- ix(729).
ix(841) :- ix(784).
ix(900) :- ix(841).
ix(961) :- ix(900).
ix(1024) :- ix(961).
ix(1089) :- ix(1024).
ix(1156) :- ix(1089).
ix(1225) :- ix(1156).
ix(1296) :- ix(1225).
ix(1369) :- ix(1296).
ix(1444) :- ix(1369).
ix(1521) :- ix(1444).
ix(1600) :- ix(1521).
ix(1681) :- ix(1600).
ix(1764) :- ix(1681).
ix(1849) :- ix(1764).
ix(1936) :- ix(1849).
ix(2025) :- ix(1936).
ix(2116) :- ix(2025).
ix(2209) :- ix(2116).
ix(2304) :- ix(2209).
ix(2401) :- ix(2304).
ix(2500) :- ix(2401).
ix(2601) :- ix(2500).
ix(2704) :- ix(2601).
ix(2809) :- ix(2704).
ix(2916) :- ix(2809).
ix(3025) :- ix(2916).
ix(3136) :- ix(3025).
ix(3249) :- ix(3136).
ix(3364) :- ix(3249).
ix(3481) :- ix(3364).
ix(3600) :- ix(3481).
ix(3721) :- ix(3600).
ix(3844) :- ix(3721).
ix(3969) :- ix(3844).
ix(4096) :- ix(3969).
ix(4225) :- ix(4096).
ix(4356) :- ix(4225).
ix(4489) :- ix(4356).
ix(4624) :- ix(4489).
ix(4761) :- ix(4624).
ix(4900) :- ix(4761).
ix(5041) :- ix(4900).
ix(5184) :- ix(5041).
ix(5329) :- ix(5184).
ix(5476) :- ix(5329).
ix(5625) :- ix(5476).
ix(5776) :- ix(5625).
ix(5929) :- ix(5776).
ix(6084) :- ix(5929).
ix(6241) :- ix(6084).
ix(6400) :- ix(6241).
ix(6561) :- ix(6400).
ix(6724) :- ix(6561).
ix(6889) :- ix(6724).
ix(7056) :- ix(6889).
ix(7225) :- ix(7056).
ix(7396) :- ix(7225).
ix(7569) :- ix(7396).
ix(7744) :- ix(7569).
ix(7921) :- ix(7744).
ix(8100) :- ix(7921).
ix(8281) :- ix(8100).
ix(8464) :- ix(8281).
ix(8649) :- ix(8464).
ix(8836) :- ix(8649).
ix(9025) :- ix(8836).
ix(9216) :- ix(9025).
ix(9409) :- ix(9216).
ix(9604) :- ix(9409).
ix(9801) :- ix(9604).
ix(10000) :- ix(9801).
#endif
```

cons_list.m

```
# /*
  cons_list.m: Pereira benchmark cons_list master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   cons_list
%
%   Fernando C. N. Pereira
%
%   Construct a 100 element list nonrecursively.

#if BENCH
#  include ".cons_list.bench"
#else
cons_list :- r1(_).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (r1/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
r1(_).
#halt
#endif
#include "r1"
```

walk_list.m

```
# /*
  walk_list.m: Pereira benchmark walk_list master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   walk_list
%
%   Fernando C. N. Pereira
%
%   Walk down a 100 element list nonrecursively.

#if BENCH
#  include ".walk_list.bench"
#else
walk_list :- r1(L),
             wl(L).
#endif

#include "r1"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (wl/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
wl(_).
#halt
#endif
wl([1|R]) :- r2(R).
```

walk_list_rec.m

```
# /*
  walk_list_rec.m: Pereira benchmark walk_list_rec master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   walk_list_rec
%
%   Fernando C. N. Pereira
%
%   Walk down a 100 element list recursively.

#if BENCH
#  include ".walk_list_rec.bench"
#else
walk_list_rec :- r1(L),
                 wlr(L).
#endif

#include "r1"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (wlr/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
wlr(_).
#halt
#endif
% recursive list cruncher

wlr([]).
wlr([_|L]) :- wlr(L).
```

```
# /*
  args_2.m: Pereira benchmark (args) args_2 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (args) args_2
%
%   Fernando C. N. Pereira
%
%   Walk down 2 copies of the same 100 element list recursively.

#if BENCH
#  include ".args_2.bench"
#else
args_2 :- rl(L),
          args(2, L).
#endif

#include "rl"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (args/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
args(_, _).
#halt
#endif
args(2, L) :- wlr(L, L).

wlr([], []).
wlr([_|L1], [_|L2]) :- wlr(L1, L2).
```

```
# /*
  args_4.m: Pereira benchmark (args) args_4 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (args) args_4
%
%   Fernando C. N. Pereira
%
%   Walk down 4 copies of the same 100 element list recursively.

#if BENCH
#  include ".args_4.bench"
#else
args_4 :- rl(L),
          args(4, L).
#endif

#include "rl"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (args/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
args(_, _).
#halt
#endif
args(4, L) :- wlr(L, L, L, L).

wlr([], [], [], []).
wlr([_|L1], [_|L2], [_|L3], [_|L4]) :- wlr(L1, L2, L3, L4).
```

args_8.m

```
# /*
  args_8.m: Pereira benchmark (args) args_8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (args) args_8
%
%   Fernando C. N. Pereira
%
%   Walk down 8 copies of the same 100 element list recursively.

#if BENCH
#  include ".args_8.bench"
#else
args_8 :- rl(L),
          args(8, L).
#endif

#include "rl"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (args/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
args(_, _).
#halt
#endif
args(8, L) :- wlr(L, L, L, L, L, L, L, L).

wlr([], [], [], [], [], [], [], []).
wlr([_|L1], [_|L2], [_|L3], [_|L4], [_|L5], [_|L6], [_|L7], [_|L8]) :-
   wlr(L1, L2, L3, L4, L5, L6, L7, L8).
```

args_16.m

```
# /*
  args_16.m: Pereira benchmark (args) args_16 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (args) args_16
%
%   Fernando C. N. Pereira
%
%   Walk down 16 copies of the same 100 element list recursively.

#if BENCH
#  include ".args_16.bench"
#else
args_16 :- rl(L),
           args(16, L).
#endif

#include "rl"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (args/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
args(_, _).
#halt
#endif
args(16, L) :- wlr(L, L, L, L, L, L, L, L, L, L, L, L, L, L, L, L).

wlr([], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []).
wlr([_|L1], [_|L2], [_|L3], [_|L4], [_|L5], [_|L6], [_|L7], [_|L8],
    [_|L9], [_|L10], [_|L11], [_|L12], [_|L13], [_|L14], [_|L15], [_|L16]) :-
   wlr(L1, L2, L3, L4, L5, L6, L7, L8, L9, L10, L11, L12, L13, L14, L15, L16).
```

```
# /*
  r1: (Pereira) nonrecursive list cruncher
  */
% nonrecursive list cruncher

r1([1|R]) :- r2(R).
r2([2|R]) :- r3(R).
r3([3|R]) :- r4(R).
r4([4|R]) :- r5(R).
r5([5|R]) :- r6(R).
r6([6|R]) :- r7(R).
r7([7|R]) :- r8(R).
r8([8|R]) :- r9(R).
r9([9|R]) :- r10(R).
r10([10|R]) :- r11(R).
r11([11|R]) :- r12(R).
r12([12|R]) :- r13(R).
r13([13|R]) :- r14(R).
r14([14|R]) :- r15(R).
r15([15|R]) :- r16(R).
r16([16|R]) :- r17(R).
r17([17|R]) :- r18(R).
r18([18|R]) :- r19(R).
r19([19|R]) :- r20(R).
r20([20|R]) :- r21(R).
r21([21|R]) :- r22(R).
r22([22|R]) :- r23(R).
r23([23|R]) :- r24(R).
r24([24|R]) :- r25(R).
r25([25|R]) :- r26(R).
r26([26|R]) :- r27(R).
r27([27|R]) :- r28(R).
r28([28|R]) :- r29(R).
r29([29|R]) :- r30(R).
r30([30|R]) :- r31(R).
r31([31|R]) :- r32(R).
r32([32|R]) :- r33(R).
r33([33|R]) :- r34(R).
r34([34|R]) :- r35(R).
r35([35|R]) :- r36(R).
r36([36|R]) :- r37(R).
r37([37|R]) :- r38(R).
r38([38|R]) :- r39(R).
r39([39|R]) :- r40(R).
r40([40|R]) :- r41(R).
r41([41|R]) :- r42(R).
r42([42|R]) :- r43(R).
r43([43|R]) :- r44(R).
r44([44|R]) :- r45(R).
r45([45|R]) :- r46(R).
r46([46|R]) :- r47(R).
r47([47|R]) :- r48(R).
r48([48|R]) :- r49(R).
r49([49|R]) :- r50(R).
```

```
r50([50|R]) :- r51(R).
r51([51|R]) :- r52(R).
r52([52|R]) :- r53(R).
r53([53|R]) :- r54(R).
r54([54|R]) :- r55(R).
r55([55|R]) :- r56(R).
r56([56|R]) :- r57(R).
r57([57|R]) :- r58(R).
r58([58|R]) :- r59(R).
r59([59|R]) :- r60(R).
r60([60|R]) :- r61(R).
r61([61|R]) :- r62(R).
r62([62|R]) :- r63(R).
r63([63|R]) :- r64(R).
r64([64|R]) :- r65(R).
r65([65|R]) :- r66(R).
r66([66|R]) :- r67(R).
r67([67|R]) :- r68(R).
r68([68|R]) :- r69(R).
r69([69|R]) :- r70(R).
r70([70|R]) :- r71(R).
r71([71|R]) :- r72(R).
r72([72|R]) :- r73(R).
r73([73|R]) :- r74(R).
r74([74|R]) :- r75(R).
r75([75|R]) :- r76(R).
r76([76|R]) :- r77(R).
r77([77|R]) :- r78(R).
r78([78|R]) :- r79(R).
r79([79|R]) :- r80(R).
r80([80|R]) :- r81(R).
r81([81|R]) :- r82(R).
r82([82|R]) :- r83(R).
r83([83|R]) :- r84(R).
r84([84|R]) :- r85(R).
r85([85|R]) :- r86(R).
r86([86|R]) :- r87(R).
r87([87|R]) :- r88(R).
r88([88|R]) :- r89(R).
r89([89|R]) :- r90(R).
r90([90|R]) :- r91(R).
r91([91|R]) :- r92(R).
r92([92|R]) :- r93(R).
r93([93|R]) :- r94(R).
r94([94|R]) :- r95(R).
r95([95|R]) :- r96(R).
r96([96|R]) :- r97(R).
r97([97|R]) :- r98(R).
r98([98|R]) :- r99(R).
r99([99|R]) :- r100(R).
r100([100|R]) :- r101(R).
r101([]).
```

```
# /*
  setof.m: Pereira benchmark setof master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   setof
%
%   Fernando C. N. Pereira

#if BENCH
#  include ".setof.bench"
#else
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (setof/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#  if DUMMY
setof :- dummy(X, Y^pr(X, Y), _).

dummy(_, _, _).
#  else
setof :- setof(X, Y^pr(X, Y), _).
#  endif
#endif

#include "pr"
```

pair_setof.m

```
# /*
  pair_setof.m: Pereira benchmark pair_setof master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   pair_setof
%
%   Fernando C. N. Pereira

#if BENCH
#  include ".pair_setof.bench"
#else
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (setof/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#  if DUMMY
pair_setof :- dummy((X,Y), pr(X, Y), S).

dummy(_, _, _).
#  else
pair_setof :- setof((X,Y), pr(X, Y), S).
#  endif
#endif

#include "pr"
```

double_setof.m

```
# /*
  double_setof.m: Pereira benchmark double_setof master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   double_setof
%
%   Fernando C. N. Pereira

#if BENCH
#  include ".double_setof.bench"
#else
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (setof/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#  if DUMMY
double_setof :- dummy((X,S), setof(Y, pr(X, Y), S), T).

dummy(_, _, _).
#  else
double_setof :- setof((X,S), setof(Y, pr(X, Y), S), T).
#  endif
#endif

#include "pr"
```

bagof.m

```
# /*
  bagof.m: Pereira benchmark bagof master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   bagof
%
%   Fernando C. N. Pereira

#if BENCH
#  include ".bagof.bench"
#else
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (bagof/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#  if DUMMY
bagof :- dummy(X, Y^pr(X, Y), S).

dummy(_, _, _).
#  else
bagof :- bagof(X, Y^pr(X, Y), S).
#  endif
#endif

#include "pr"
```

```
# /*
  pr: (Pereira) pr/2 for setof, pair_setof, double_setof, and bagof
  */
pr(99, 1).
pr(98, 2).
pr(97, 3).
pr(96, 4).
pr(95, 5).
pr(94, 6).
pr(93, 7).
pr(92, 8).
pr(91, 9).
pr(90, 10).
pr(89, 11).
pr(88, 12).
pr(87, 13).
pr(86, 14).
pr(85, 15).
pr(84, 16).
pr(83, 17).
pr(82, 18).
pr(81, 19).
pr(80, 20).
pr(79, 21).
pr(78, 22).
pr(77, 23).
pr(76, 24).
pr(75, 25).
pr(74, 26).
pr(73, 27).
pr(72, 28).
pr(71, 29).
pr(70, 30).
pr(69, 31).
pr(68, 32).
pr(67, 33).
pr(66, 34).
pr(65, 35).
pr(64, 36).
pr(63, 37).
pr(62, 38).
pr(61, 39).
pr(60, 40).
pr(59, 41).
pr(58, 42).
pr(57, 43).
pr(56, 44).
pr(55, 45).
pr(54, 46).
pr(53, 47).
pr(52, 48).
pr(51, 49).
pr(50, 50).
pr(49, 51).
pr(48, 52).
pr(47, 53).
pr(46, 54).
pr(45, 55).
pr(44, 56).
pr(43, 57).
pr(42, 58).
pr(41, 59).
pr(40, 60).
pr(39, 61).
pr(38, 62).
pr(37, 63).
pr(36, 64).
pr(35, 65).
pr(34, 66).
pr(33, 67).
pr(32, 68).
pr(31, 69).
pr(30, 70).
pr(29, 71).
pr(28, 72).
pr(27, 73).
pr(26, 74).
pr(25, 75).
pr(24, 76).
pr(23, 77).
pr(22, 78).
pr(21, 79).
pr(20, 80).
pr(19, 81).
pr(18, 82).
pr(17, 83).
pr(16, 84).
pr(15, 85).
pr(14, 86).
pr(13, 87).
pr(12, 88).
pr(11, 89).
pr(10, 90).
pr(9, 91).
pr(8, 92).
pr(7, 93).
pr(6, 94).
pr(5, 95).
pr(4, 96).
pr(3, 97).
pr(2, 98).
pr(1, 99).
pr(0, 100).
```

```
# /*
  cons_term.m: Pereira benchmark cons_term master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   cons_term
%
%   Fernando C. N. Pereira
%
%   Construct a term with 100 nodes nonrecursively.

#if BENCH
#  include ".cons_term.bench"
#else
cons_term :- s1(_).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (s1/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
s1(_).
#halt
#endif
#include "s1"
```

walk_term.m

```
# /*
  walk_term.m: Pereira benchmark walk_term master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   walk_term
%
%   Fernando C. N. Pereira
%
%   Walk down a term with 100 nodes nonrecursively.

#if BENCH
#  include ".walk_term.bench"
#else
walk_term :- s1(T),
             wt(T).
#endif

#include "s1"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (wt/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
wt(_).
#halt
#endif
wt(f(1, R)) :- s2(R).
```

```
# /*
  walk_term_rec.m: Pereira benchmark walk_term_rec master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   walk_term_rec
%
%   Fernando C. N. Pereira
%
%   Walk down a term with 100 nodes recursively.

#if BENCH
#  include ".walk_term_rec.bench"
#else
walk_term_rec :- s1(L),
                 wtr(L).
#endif

#include "s1"

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (wtr/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
wtr(_).
#halt
#endif
% recursive term cruncher

wtr(nil).
wtr(f(_,R)) :- wtr(R).
```

```
# /*
  s1: (Pereira) nonrecursive term cruncher
  */
% nonrecursive term cruncher

s1(f(1, R)) :- s2(R).
s2(f(2, R)) :- s3(R).
s3(f(3, R)) :- s4(R).
s4(f(4, R)) :- s5(R).
s5(f(5, R)) :- s6(R).
s6(f(6, R)) :- s7(R).
s7(f(7, R)) :- s8(R).
s8(f(8, R)) :- s9(R).
s9(f(9, R)) :- s10(R).
s10(f(10, R)) :- s11(R).
s11(f(11, R)) :- s12(R).
s12(f(12, R)) :- s13(R).
s13(f(13, R)) :- s14(R).
s14(f(14, R)) :- s15(R).
s15(f(15, R)) :- s16(R).
s16(f(16, R)) :- s17(R).
s17(f(17, R)) :- s18(R).
s18(f(18, R)) :- s19(R).
s19(f(19, R)) :- s20(R).
s20(f(20, R)) :- s21(R).
s21(f(21, R)) :- s22(R).
s22(f(22, R)) :- s23(R).
s23(f(23, R)) :- s24(R).
s24(f(24, R)) :- s25(R).
s25(f(25, R)) :- s26(R).
s26(f(26, R)) :- s27(R).
s27(f(27, R)) :- s28(R).
s28(f(28, R)) :- s29(R).
s29(f(29, R)) :- s30(R).
s30(f(30, R)) :- s31(R).
s31(f(31, R)) :- s32(R).
s32(f(32, R)) :- s33(R).
s33(f(33, R)) :- s34(R).
s34(f(34, R)) :- s35(R).
s35(f(35, R)) :- s36(R).
s36(f(36, R)) :- s37(R).
s37(f(37, R)) :- s38(R).
s38(f(38, R)) :- s39(R).
s39(f(39, R)) :- s40(R).
s40(f(40, R)) :- s41(R).
s41(f(41, R)) :- s42(R).
s42(f(42, R)) :- s43(R).
s43(f(43, R)) :- s44(R).
s44(f(44, R)) :- s45(R).
s45(f(45, R)) :- s46(R).
s46(f(46, R)) :- s47(R).
s47(f(47, R)) :- s48(R).
s48(f(48, R)) :- s49(R).
s49(f(49, R)) :- s50(R).
s50(f(50, R)) :- s51(R).
```

```
s51(f(51, R)) :- s52(R).
s52(f(52, R)) :- s53(R).
s53(f(53, R)) :- s54(R).
s54(f(54, R)) :- s55(R).
s55(f(55, R)) :- s56(R).
s56(f(56, R)) :- s57(R).
s57(f(57, R)) :- s58(R).
s58(f(58, R)) :- s59(R).
s59(f(59, R)) :- s60(R).
s60(f(60, R)) :- s61(R).
s61(f(61, R)) :- s62(R).
s62(f(62, R)) :- s63(R).
s63(f(63, R)) :- s64(R).
s64(f(64, R)) :- s65(R).
s65(f(65, R)) :- s66(R).
s66(f(66, R)) :- s67(R).
s67(f(67, R)) :- s68(R).
s68(f(68, R)) :- s69(R).
s69(f(69, R)) :- s70(R).
s70(f(70, R)) :- s71(R).
s71(f(71, R)) :- s72(R).
s72(f(72, R)) :- s73(R).
s73(f(73, R)) :- s74(R).
s74(f(74, R)) :- s75(R).
s75(f(75, R)) :- s76(R).
s76(f(76, R)) :- s77(R).
s77(f(77, R)) :- s78(R).
s78(f(78, R)) :- s79(R).
s79(f(79, R)) :- s80(R).
s80(f(80, R)) :- s81(R).
s81(f(81, R)) :- s82(R).
s82(f(82, R)) :- s83(R).
s83(f(83, R)) :- s84(R).
s84(f(84, R)) :- s85(R).
s85(f(85, R)) :- s86(R).
s86(f(86, R)) :- s87(R).
s87(f(87, R)) :- s88(R).
s88(f(88, R)) :- s89(R).
s89(f(89, R)) :- s90(R).
s90(f(90, R)) :- s91(R).
s91(f(91, R)) :- s92(R).
s92(f(92, R)) :- s93(R).
s93(f(93, R)) :- s94(R).
s94(f(94, R)) :- s95(R).
s95(f(95, R)) :- s96(R).
s96(f(96, R)) :- s97(R).
s97(f(97, R)) :- s98(R).
s98(f(98, R)) :- s99(R).
s99(f(99, R)) :- s100(R).
s100(f(100, R)) :- s101(R).
s101(nil).
```

medium_unify.m

```
# /*
  medium_unify.m: Pereira benchmark medium_unify master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   medium_unify
%
%   Fernando C. N. Pereira
%
%   Unify structures 5 deep.

#if BENCH
#  include ".medium_unify.bench"
#else
medium_unify :- term64(Term1),
                term64(Term2),
                equal(Term1, Term2).
#endif

term64(X1) :-
   X1 = f(X2, X2),
   X2 = f(X4, X4),
   X4 = f(X8, X8),
   X8 = f(X16, X16),
   X16 = f(X32, X32),
   X32 = f(X64, X64).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (equal/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
equal(_, _).
#else
equal(X, X).
#endif
```

deep_unify.m

```
# /*
  deep_unify.m: Pereira benchmark deep_unify master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   deep_unify
%
%   Fernando C. N. Pereira
%
%   Unify structures 11 deep.

#if BENCH
#  include ".deep_unify.bench"
#else
deep_unify :- term4096(Term1),
              term4096(Term2),
              equal(Term1, Term2).
#endif

term4096(X1) :-
   X1 = f(X2, X2),
   X2 = f(X4, X4),
   X4 = f(X8, X8),
   X8 = f(X16, X16),
   X16 = f(X32, X32),
   X32 = f(X64, X64),
   X64 = f(X128, X128),
   X128 = f(X256, X256),
   X256 = f(X512, X512),
   X512 = f(X1024, X1024),
   X1024 = f(X2048, X2048),
   X2048 = f(X4096, X4096).

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (equal/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
equal(_, _).
#else
equal(X, X).
#endif
```

.floating_add.bench

```
# /*
  set-up.floating_add: bench set-up for floating_add
  */
floating_add :- driver(floating_add).

benchmark(floating_add, fa1(0.1, 1.1, R), dummy(0.1, 1.1, R), 1000).

#message "NOTE: show/1 is NOT defined for floating_add"


#include "driver"
```

.integer_add.bench

```
# /*
  set-up.integer_add: bench set-up for integer_add
  */
integer_add :- driver(integer_add).

benchmark(integer_add, a1(0, 1, R), dummy(0, 1, R), 1000).

#message "NOTE: show/1 is NOT defined for integer_add"


#include "driver"
```

```
# /*
  set-up.arg_1: bench set-up for (arg) arg_1
  */
arg_1 :- driver(arg_1).

benchmark(arg_1, arg1(1, Term, _), dummy(1, Term, _), 2000) :-
   complex_nary_term(100, 1, Term).

#message "NOTE: show/1 is NOT defined for arg_1"


#include "driver"
```

```
# /*
  set-up.arg_2: bench set-up for (arg) arg_2
  */
arg_2 :- driver(arg_2).

benchmark(arg_2, arg1(2, Term, _), dummy(2, Term, _), 2000) :-
   complex_nary_term(100, 2, Term).

#message "NOTE: show/1 is NOT defined for arg_2"


#include "driver"
```

.arg_4.bench

```
# /*
  set-up.arg_4: bench set-up for (arg) arg_4
  */
arg_4 :- driver(arg_4).

benchmark(arg_4, arg1(4, Term, _), dummy(4, Term, _), 2000) :-
   complex_nary_term(100, 4, Term).

#message "NOTE: show/1 is NOT defined for arg_4"


#include "driver"
```

```
# /*
  set-up.arg_8: bench set-up for (arg) arg_8
  */
arg_8 :- driver(arg_8).

benchmark(arg_8, arg1(8, Term, _), dummy(8, Term, _), 2000) :-
   complex_nary_term(100, 8, Term).

#message "NOTE: show/1 is NOT defined for arg_8"


#include "driver"
```

.arg_16.bench

```
# /*
  set-up.arg_16: bench set-up for (arg) arg_16
  */
arg_16 :- driver(arg_16).

benchmark(arg_16, arg1(16, Term, _), dummy(16, Term, _), 2000) :-
   complex_nary_term(100, 16, Term).

#message "NOTE: show/1 is NOT defined for arg_16"


#include "driver"
```

.assert_unit.bench

```
# /*
  set-up.assert_unit: bench set-up for assert_unit
  */
assert_unit :- driver(assert_unit).

benchmark(assert_unit, assert_clauses(L), dummy(L), 5) :-
   abolish(ua, 3),
   create_units(1, 1000, L).

#message "NOTE: show/1 is NOT defined for assert_unit"


#include "driver"
```

.access_unit.bench

```
# /*
  set-up.access_unit: bench set-up for access_unit
  */
access_unit :- driver(access_unit).

benchmark(access_unit, access_dix(1, 1), dummy(1, 1), 100) :-
   abolish(dix, 2),
   dix_clauses(1, 100, L),
   assert_clauses(L).

#message "NOTE: show/1 is NOT defined for access_unit"


#include "driver"
```

.slow_access_unit.bench

```
# /*
  set-up.slow_access_unit: bench set-up for slow_access_unit
  */
slow_access_unit :- driver(slow_access_unit).

benchmark(slow_access_unit, access_back(1, 1), dummy(1, 1), 10) :-
   abolish(dix, 2),
   dix_clauses(1, 100, L),
   assert_clauses(L).

#message "NOTE: show/1 is NOT defined for slow_access_unit"


#include "driver"
```

.shallow_backtracking.bench

```
# /*
  set-up.shallow_backtracking: bench set-up for shallow_backtracking
  */
shallow_backtracking :- driver(shallow_backtracking).

benchmark(shallow_backtracking, shallow, dummy, 2000).

#message "NOTE: show/1 is NOT defined for shallow_backtracking"


#include "driver"
```

.deep_backtracking.bench

```
# /*
  set-up.deep_backtracking: bench set-up for deep_backtracking
  */
deep_backtracking :- driver(deep_backtracking).

benchmark(deep_backtracking, deep, dummy, 2000).

#message "NOTE: show/1 is NOT defined for deep_backtracking"


#include "driver"
```

.tail_call_atom_atom.bench

```
# /*
  set-up.tail_call_atom_atom: bench set-up for tail_call_atom_atom
  */
tail_call_atom_atom :- driver(tail_call_atom_atom).

benchmark(tail_call_atom_atom, p1(a), dummy(a), 2000).

#message "NOTE: show/1 is NOT defined for tail_call_atom_atom"


#include "driver"
```

.binary_call_atom_atom.bench

```
# /*
  set-up.binary_call_atom_atom: bench set-up for binary_call_atom_atom
  */
binary_call_atom_atom :- driver(binary_call_atom_atom).

benchmark(binary_call_atom_atom, q1(a), dummy(a), 2000).

#message "NOTE: show/1 is NOT defined for binary_call_atom_atom"


#include "driver"
```

```
# /*
  set-up.choice_point: bench set-up for choice_point
  */
choice_point :- driver(choice_point).

benchmark(choice_point, choice, dummy, 2000).

#message "NOTE: show/1 is NOT defined for choice_point"


#include "driver"
```

```
# /*
  set-up.trail_variables: bench set-up for trail_variables
  */
trail_variables :- driver(trail_variables).

benchmark(trail_variables, trail, dummy, 2000).

#message "NOTE: show/1 is NOT defined for trail_variables"


#include "driver"
```

.index.bench

```
# /*
  set-up.index: bench set-up for index
  */
index :- driver(index).

benchmark(index, ix(1), dummy(1), 2000).

#message "NOTE: show/1 is NOT defined for index"


#include "driver"
```

```
# /*
  set-up.cons_list: bench set-up for cons_list
  */
cons_list :- driver(cons_list).

benchmark(cons_list, r1(_), dummy(_), 2000).

#message "NOTE: show/1 is NOT defined for cons_list"


#include "driver"
```

```
# /*
  set-up.walk_list: bench set-up for walk_list
  */
walk_list :- driver(walk_list).

benchmark(walk_list, wl(L), dummy(L), 2000) :- rl(L).

#message "NOTE: show/1 is NOT defined for walk_list"


#include "driver"
```

```
# /*
  set-up.walk_list_rec: bench set-up for walk_list_rec
  */
walk_list_rec :- driver(walk_list_rec).

benchmark(walk_list_rec, wlr(L), dummy(L), 2000) :- rl(L).

#message "NOTE: show/1 is NOT defined for walk_list_rec"


#include "driver"
```

.args_2.bench

```
# /*
  set-up.args_2: bench set-up for (args) args_2
  */
args_2 :- driver(args_2).

benchmark(args_2, args(2, L), dummy(2, L), 2000) :- r1(L).

#message "NOTE: show/1 is NOT defined for args_2"


#include "driver"
```

```
# /*
  set-up.args_4: bench set-up for (args) args_4
  */
args_4 :- driver(args_4).

benchmark(args_4, args(4, L), dummy(4, L), 2000) :- r1(L).

#message "NOTE: show/1 is NOT defined for args_4"


#include "driver"
```

.args_8.bench

```
# /*
  set-up.args_8: bench set-up for (args) args_8
  */
args_8 :- driver(args_8).

benchmark(args_8, args(8, L), dummy(8, L), 2000) :- r1(L).

#message "NOTE: show/1 is NOT defined for args_8"


#include "driver"
```

```
# /*
  set-up.args_16: bench set-up for (args) args_16
  */
args_16 :- driver(args_16).

benchmark(args_16, args(16, L), dummy(16, L), 2000) :- r1(L).

#message "NOTE: show/1 is NOT defined for args_16"


#include "driver"
```

.setof.bench

```
# /*
  set-up.setof: bench set-up for setof
  */
setof :- driver(setof).

#if DUMMY
benchmark(setof, dummy(X, Y^pr(X, Y), S), dummy(X, Y^pr(X, Y), S), 10).
#else
benchmark(setof, setof(X, Y^pr(X, Y), S), dummy(X, Y^pr(X, Y), S), 10).
#endif

#message "NOTE: show/1 is NOT defined for setof"


#include "driver"
```

.pair_setof.bench

```
# /*
  set-up.pair_setof: bench set-up for pair_setof
  */
pair_setof :- driver(pair_setof).

#if DUMMY
benchmark(pair_setof, dummy((X,Y), pr(X, Y), S),
                      dummy((X,Y), pr(X, Y), S), 10).
#else
benchmark(pair_setof, setof((X,Y), pr(X, Y), S),
                      dummy((X,Y), pr(X, Y), S), 10).
#endif

#message "NOTE: show/1 is NOT defined for pair_setof"


#include "driver"
```

.double_setof.bench

```
# /*
  set-up.double_setof: bench set-up for double_setof
  */
double_setof :- driver(double_setof).

#if DUMMY
benchmark(double_setof, dummy((X,S), setof(Y, pr(X, Y), S), T),
                        dummy((X,S), setof(Y, pr(X, Y), S), T), 10).
#else
benchmark(double_setof, setof((X,S), setof(Y, pr(X, Y), S), T),
                        dummy((X,S), setof(Y, pr(X, Y), S), T), 10).
#endif

#message "NOTE: show/1 is NOT defined for double_setof"


#include "driver"
```

```
# /*
  set-up.bagof: bench set-up for bagof
  */
bagof :- driver(bagof).

#if DUMMY
benchmark(bagof, dummy(X, Y^pr(X, Y), S), dummy(X, Y^pr(X, Y), S), 10).
#else
benchmark(bagof, bagof(X, Y^pr(X, Y), S), dummy(X, Y^pr(X, Y), S), 10).
#endif

#message "NOTE: show/1 is NOT defined for bagof"


#include "driver"
```

.cons_term.bench

```
# /*
  set-up.cons_term: bench set-up for cons_term
  */
cons_term :- driver(cons_term).

benchmark(cons_term, s1(_), dummy(_), 2000).

#message "NOTE: show/1 is NOT defined for cons_term"


#include "driver"
```

.walk_term.bench

```
# /*
  set-up.walk_term: bench set-up for walk_term
  */
walk_term :- driver(walk_term).

benchmark(walk_term, wt(T), dummy(T), 2000) :- s1(T).

#message "NOTE: show/1 is NOT defined for walk_term"


#include "driver"
```

.walk_term_rec.bench

```
# /*
  set-up.walk_term_rec: bench set-up for walk_term_rec
  */
walk_term_rec :- driver(walk_term_rec).

benchmark(walk_term_rec, wtr(T), dummy(T), 2000) :- s1(T).

#message "NOTE: show/1 is NOT defined for walk_term_rec"


#include "driver"
```

.medium_unify.bench

```
# /*
  set-up.medium_unify: bench set-up for medium_unify
  */
medium_unify :- driver(medium_unify).

benchmark(medium_unify, equal(Term1, Term2), dummy(Term1, Term2), 2000) :-
   term64(Term1),
   term64(Term2).

#message "NOTE: show/1 is NOT defined for medium_unify"


#include "driver"
```

.deep_unify.bench

```
# /*
  set-up.deep_unify: bench set-up for deep_unify
  */
deep_unify :- driver(deep_unify).

benchmark(deep_unify, equal(Term1, Term2), dummy(Term1, Term2), 100) :-
   term4096(Term1),
   term4096(Term2).

#message "NOTE: show/1 is NOT defined for deep_unify"


#include "driver"
```

plm_compiler

```
# /*
  plm_compiler.m: benchmark plm_compiler master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   plm_compiler
%
%   compile small Prolog file to PLM code

#option /* BIM_PL C_PL QUINTUS_PL SB_PL SICSTUS_PL
           are #option'd in .plm_compiler.bench */ "
        > The PLM compiler includes system-dependent code.
        > If one of
        >
        > BIM_PL C_PL QUINTUS_PL SB_PL SICSTUS_PL
        >
        > is selected, then appropriate code is generated."
#if BENCH
#  include ".plm_compiler.bench"
#else
#  if BIM_PROLOG
plm_compiler :- bim,
#  elseif C_PROLOG
plm_compiler :- c,
#  elseif QUINTUS_PROLOG
:- no_style_check(single_var).

plm_compiler :- quintus,
#  elseif SB_PROLOG
plm_compiler :- sb,
#  elseif SICTUS_PROLOG
plm_compiler :- sicstus,
#  else
plm_compiler :-
#  endif
                options(test, []),
                see(test), read_clauses(CI), seen,
                cap(CI).
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (cap/1).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
cap(_).
#else
cap(CI) :- tell('test.w'), compileallprocs(CI), told.
#endif


#include "plm_compiler" /* compiler for the PLM */
```

```
# /*
  plm_compiler: compiler for the PLM
  */
% Here is the latest version of the PLM compiler as described
% in my Master's report (UCB/CSD 84/203), slightly modified
% for benchmarking.  I ask that you reference the report in
% any published work which uses it.  This version has been
% modified to run under five Prolog systems - BIM Prolog,
% C-Prolog, Quintus Prolog, SB Prolog, and SICStus Prolog.
% The call plm(FileName) will compile FileName.  Note that
% one of the five calls bim, c, quintus, sb, or sicstus must
% be done first for correct execution under your system.
% WAM code will be put on FileName.w.
%
% Compiler options:
%
%   plm(filename, optionlist)
%
%   where the options are:
%       a       use no-argument allocate with dummy call.
%               (default: use allocate with environment size argument)
%       l       output in Prolog-readable list form.
%               (default: output in human-readable form)
%       u       do not expand is/2 into is/4.
%               (default: expand is/2 into is/4)
%       s       compile into +,-,\/,/\ instead of is/4.  Other operators
%               are still compiled into is/4.
%               (default: use is/4 for all operators)
%       q       quote all atoms.  Only has effect if option l is not used.
%               (default: quote only those atoms that need it)
%       a(X)    append _X to all labels in the human-readable code.
%               (default: append nothing to labels)
%
%   A single option does not have to be put in a list.
%
%   plm(filename)       is the same as plm(filename,[]).
%
%   plm_help            gives on-line help with the options.
%
% If you have ideas for improvements or if you
% find any bugs, I will be happy to hear it.
%
% Peter Van Roy (vanroy@bellatrix.Berkeley.EDU)

/**********************************************************************/
/* Copyright (C) 1987 by Peter Van Roy                               */
/* on behalf of the Regents of the University of California.         */
/**********************************************************************/

% Call to change Prolog version at run time:

bim     :- abolish(prolog_version,1), assert(prolog_version(bimprolog)).
c       :- abolish(prolog_version,1), assert(prolog_version(cprolog)).
quintus :- abolish(prolog_version,1), assert(prolog_version(quintusprolog)).
sb      :- abolish(prolog_version,1), assert(prolog_version(sbprolog)).
sicstus :- abolish(prolog_version,1), assert(prolog_version(sicstusprolog)).

/**********************************************************************/
```

```
% CPU time handling:

get_cpu_time(T) :- prolog_version(bimprolog), !,
                   cputime(T).
get_cpu_time(T) :- prolog_version(cprolog), !,
                   T is cputime.
get_cpu_time(T) :- prolog_version(quintusprolog), !,
                   statistics(runtime, [T, _]).
get_cpu_time(T) :- prolog_version(sbprolog), !,
                   cputime(T).
get_cpu_time(T) :- prolog_version(sicstusprolog), !,
                   statistics(runtime, [T, _]).

cpu_time_unit(seconds) :-
        prolog_version(bimprolog), !.
cpu_time_unit(seconds) :-
        prolog_version(cprolog), !.
cpu_time_unit('milli-seconds') :-
        prolog_version(quintusprolog), !.
cpu_time_unit('milli-seconds') :-
        prolog_version(sbprolog), !.
cpu_time_unit('milli-seconds') :-
        prolog_version(sicstusprolog), !.

/**********************************************************************/

% Compile 'FileName' and put results in 'FileName.w':
% Default: no special options.
plm(FileName) :- !, plm(FileName, []).
plm(FileName, One) :-
        atomic(One),
        (prolog_version(sbprolog) -> not(One=[]); \+(One=[])), !,
        plm(FileName, [One]).
plm(FileName, One) :-
        (prolog_version(sbprolog) -> not(list(One)); \+(list(One))),
        (prolog_version(sbprolog) -> not(One=[]); \+(One=[])), !,
        plm(FileName, [One]).
plm(FileName, OptionList) :-
        % Handle options:
        options(FileName, OptionList),
        % Read input file:
        see(FileName), read_clauses(CI), seen,
        write('Finished reading '), write(FileName), nl,
        name(FileName, NL),
        name('.w', DOTW),
        concat(NL, DOTW, OF),
        name(OutFile, OF),
        % Compile & write output file:
          % get_cpu_time/2 is defined in .bench/driver
        ( clause(get_cpu_time(T0, Unit), Body0), !, call(Body0)
        ; get_cpu_time(T0) ),
        tell(OutFile),
        compileallprocs(CI),
        told,
        ( clause(get_cpu_time(T1, Unit), Body1), !, call(Body1)
        ; get_cpu_time(T1) ),
        Time is T1-T0,
          % =/4 is defined in .bench/driver
        ( clause(=(Time, Unit, Time_out, Unit_out), Body), !, call(Body),
          write('compilation took '),
          write(Time_out), write(' '),
          write(Unit_out), nl
        ; cpu_time_unit(Unit),
          write('compilation took '),
          write(Time), write(' '),
          write(Unit), nl ),
        fail.
% Clean up all heap space used.
plm(_, _).
```

```
% Add options to data base:
options(FileName, OptionList) :-
        abolish(compile_options, 1),
        assert(compile_options(none)),
        atom(FileName),full_list(OptionList), add_options(OptionList), !.
options(FileName, OptionList) :-
        write('First param is name of source file (atom)'),
        nl,
      write('Second param is one option or a list of options (ground terms)'),
        nl,
        abort, !.

add_options([Opt|OptionList]) :-
        nonvar(Opt), !,
        assert(compile_options(Opt)),
        add_options(OptionList).
add_options([]).

read_clauses(ClauseInfo) :-
        prolog_version(X),
        my_member(X, [quintusprolog,cprolog,sbprolog,sicstusprolog]), !,
        c_read_clauses(ClauseInfo), !.
read_clauses(ClauseInfo) :-
        prolog_version(bimprolog),
        b_read_clauses(ClauseInfo), !.

c_read_clauses(ClauseInfo) :- !,
        read(Clause),
        (Clause=end_of_file -> ClauseInfo=[];
                getname(Clause, NameAr),
                ClauseInfo=[source(NameAr,Clause)|Rest],
                c_read_clauses(Rest)), !.

b_read_clauses(ClauseInfo) :-
        read(Clause),
        getname(Clause, NameAr),
        ClauseInfo=[source(NameAr,Clause)|Rest],
        b_read_clauses(Rest), !.
b_read_clauses([]).



getname(Clause, Name/Arity) :- !,
        (Clause=(Head:-Body); Clause=Head),
        Head=..[Name|Args],
        my_length(Args, Arity).

% Generate and write code for all procedures in ClauseInfo:
compileallprocs([]) :-
        alloc_option,
        list_option, !.
compileallprocs(ClauseInfo) :-
        filteroneproc(ClauseInfo, NextCI, NameAr, OneProc),
        gc(compileproc(NameAr, OneProc, Code-[])),
        write_plm(NameAr, Code),
        compileallprocs(NextCI), !.

        % Take care of old-new allocate option:
        alloc_option :-
                compile_options(a),
                write_plm(allocate_dummy/0, [proceed]), !.
        alloc_option.

        % Procedure's end:
        list_option :- compile_options(l), !.
        list_option :- write(end), nl, nl, !.
```

```
filteroneproc([], [], _, []) :- !.
filteroneproc([source(NameAr,C)|Rest], NextCI, NameAr, [C|OneProc]) :-
        filteroneproc(Rest, NextCI, NameAr, OneProc), !.
filteroneproc([source(N,C)|Rest], [source(N,C)|NextCI], NameAr, OneProc) :-
        filteroneproc(Rest, NextCI, NameAr, OneProc), !.

/*******************************************************************/

% Compile one procedure.
% Input is a list of clauses in unaltered form.
% Output is complete code for the procedure.
% The labels remain uninstantiated.
% The special compilation for lists, constants, structures
% is not needed if:
%   1. Arity=0, no first arguments.
%   2. procedure consists of just one clause.
%   3. all first arguments are variables.
% Also recognized are the cases where all first arguments are either
% variables or one other kind.

compileproc(_/Arity, Clauses, Code-Link) :-
        compileclauses(Clauses, CompC),
        var_block(CompC, VarLbl, VCode-VLink),
        cp(Arity, CompC, VarLbl, VCode, VLink, Code, Link), !.

% Easy optimizations
cp(Arity, _, _, VCode, VLink, VCode, VLink) :- Arity=0, !.
cp(_, CompC, _, VCode, VLink, VCode, VLink) :- my_length(CompC,1), !.
cp(_, CompC, _, VCode, VLink, VCode, VLink) :- all_var(CompC), !.

% Only variables and one other kind present:
cp(_, CompC, VarLbl, VCode, VLink, Code, Link) :-
        same_or_var(CompC, Kind), !,
        filterv(CompC, VarC),
        try_block(VarC, TryLbl, VLink-TLink),
        cp_sub(Kind, CompC, TryLbl, VarLbl, TLink, Link, CLS),
        Switch=..[switch_on_term|CLS],
        Code=[Switch|VCode].

% General case: code for list, constant, and structure
cp(_, CompC, _, VCode, VLink, Code, Link) :-
        filterlcs(CompC, ListC, ConstC, StrucC),
        try_block(ListC, ListLbl, VLink-LLink),
        cs_block(ConstC, ConstLbl, LLink-CLink, _),
        cs_block(StrucC, StrucLbl, CLink-Link, _),
        Code=[switch_on_term(ConstLbl,ListLbl,StrucLbl)|VCode].

% Part of var & one other kind optimization:
cp_sub(list, _, TryLbl, VarLbl, TLink, TLink, CLS) :- !,
        CLS=[TryLbl, VarLbl, TryLbl], !.
cp_sub(Kind, CompC, TryLbl, VarLbl, TLink, Link, CLS) :-
        (prolog_version(sbprolog) -> not(Kind=list); \+(Kind=list)), !,
        cs_block(CompC, BlkLbl, BlkCode-BlkLink, Hashed),
        cp_hash(Hashed,CSLbl,TLink,VarLbl,Link,BlkLbl,BlkCode,BlkLink),
        cp_const_struc(Kind, CLS, CSLbl, TryLbl).

        cp_hash(no_hash, VarLbl, Link,    VarLbl, Link, _, _, _).
        cp_hash(yes_hash,BlkLbl, BlkCode, _,      Link, BlkLbl, BlkCode, Link).

        cp_const_struc(constant, [CSLbl,TryLbl,TryLbl], CSLbl, TryLbl).
        cp_const_struc(structure,[TryLbl,TryLbl,CSLbl], CSLbl, TryLbl).

% Succeeds if first arguments are all variable and one other kind:
same_or_var([clause(FArg,_,_)|Rest], Kind) :-
        kind(FArg, K),
        (K=variable; K=Kind),
        same_or_var(Rest, Kind).
same_or_var([], _).
```

```
% Succeeds if first arguments are all variables:
all_var(CompC) :- same_or_var(CompC, variable).


compileclauses([C|Clauses], [clause(FArg,Lbl,[label(Lbl)|Code]-Link)|Rest]) :-
        % !! getfirstarg must come before compileclause, since
        % compileclause instantiates variables in the head to registers.
        getfirstarg(C, FArg),
        gc(compileclause(C, Code, Link)), % garbage collect it.
        compileclauses(Clauses, Rest).
compileclauses([], []).

getfirstarg(Clause, FArg) :-
        (Clause=(Head:-Body); Clause=Head),
        Head=..[Name|HArgs],
        (HArgs=[Arg1|_]; true),
        gfa(Arg1, FArg).

gfa(Arg1, FArg) :- var(Arg1), !.
gfa(Arg1, Arg1) :- atomic(Arg1), !.
gfa(Arg1, Struc/Arity) :-
        Arg1=..[Struc|Args],
        my_length(Args, Arity).

/*******************************************************************/

% Generate code for the four blocks:

% First block:
% Link the clauses together with try_elses.
% Jumped to if the calling argument is a variable.
% Correctly handles cases of 1, 2, or more clauses.

var_block([clause(_,Lbl,Code-Link)], Lbl, Code-Link).
var_block(Clauses, Lbl, Code-Link) :-
        var_block(try, Clauses, Lbl, Code-Link).

var_block(_, [clause(_,_,C-L)], Lbl, [label(Lbl),trust(else,fail)|C]-L).
var_block(Type, [clause(_,_,C-L)|Clauses], PrevLbl,
                [label(PrevLbl),Instr|C]-Link) :-
        Instr=..[Type,else,NextLbl],
        var_block(retry, Clauses, NextLbl, L-Link).


% Filter out clauses which could match with a list, const, or struc
% as first argument.  Note that a variable as first argument matches
% with all of them.

filterlcs([], [], [], []).
filterlcs([X|Rest], [X|ListLbls], [X|ConstLbls], [X|StrucLbls]) :-
        X=clause(FArg, _, _),
        var(FArg), !,
        filterlcs(Rest, ListLbls, ConstLbls, StrucLbls).
filterlcs([X|Rest], [X|ListLbls], ConstLbls, StrucLbls) :-
        X=clause('.'/2, _, _),
        filterlcs(Rest, ListLbls, ConstLbls, StrucLbls).
filterlcs([X|Rest], ListLbls, [X|ConstLbls], StrucLbls) :-
        X=clause(FArg, _, _),
        atomic(FArg), !,
        filterlcs(Rest, ListLbls, ConstLbls, StrucLbls).
filterlcs([X|Rest], ListLbls, ConstLbls, [X|StrucLbls]) :-
        filterlcs(Rest, ListLbls, ConstLbls, StrucLbls).

% Filter out clauses with variables as first argument:

filterv([], []).
filterv([X|Rest], [X|VarLbls]) :-
        X=clause(FArg, _, _),
        var(FArg), !,
        filterv(Rest, VarLbls).
filterv([_|Rest], VarLbls) :-
        filterv(Rest, VarLbls).


% Try block: Generate a generic try-block to try
% all clauses in the given list.
% Optimizes code if only 0 or 1 clauses are given.

try_block([], fail, Link-Link).
try_block([clause(_,Lbl,_)], Lbl, Link-Link).
try_block([clause(_,Lbl,_)|Clauses], Label,
            [label(Label),try(Lbl)|LCode]-Link) :-
        try_block(Clauses, LCode-Link).

try_block([clause(_,Lbl,_)], [trust(Lbl)|Link]-Link) :- !.
try_block([clause(_,Lbl,_)|Clauses], [retry(Lbl)|LCode]-Link) :-
        try_block(Clauses, LCode-Link).


% Const and Struc block: First argument is a constant or a structure.
% This routine works for both constants and structures.
% Difference with try_block: generates hash tables if needed.
% Variable Hashed indicates if hash tables were generated.
% It is either no_hash or yes_hash.

cs_block([], fail, Link-Link, no_hash).
cs_block([clause(_,Lbl,_)], Lbl, Link-Link, no_hash).
cs_block(Clauses, Lbl, [label(Lbl)|Code]-Link, Hashed) :-
        cs_gather(Clauses, [], Gather-[], Hashed),
        set_hashed(Hashed),
        cs_link(try, Gather, Code-Link).

        % Instantiate argument:
        set_hashed(no_hash) :- !.
        set_hashed(yes_hash).

% Gather contiguous arguments which are not variables together.
% The other arguments are left separate.

cs_gather([X|Rest], Collect, Gather-Link, H) :-
        X=clause(FArg, Lbl, _),
        var(FArg), !,
        dump(Collect, Gather-G, H),
        G=[X|G2],
        cs_gather(Rest, [], G2-Link, H).
cs_gather([X|Rest], Collect, Gather-Link, H) :-
        X=clause(FArg, Lbl, _),
        my_member(clause(FArg,_,_), Collect), !,
        dump(Collect, Gather-G, H),
        cs_gather(Rest, [X], G-Link, H).
cs_gather([X|Rest], Collect, Gather-Link, H) :-
        cs_gather(Rest, [X|Collect], Gather-Link, H).
cs_gather([], Collect, Gather-Link, H) :-
        dump(Collect, Gather-Link, H).

% Convert a collection of clause(s) to a member of Gather:
% If Collect is longer than one, it (as list) is a member.
% Else just its element clause is member.
dump([], L-L, _).
dump([X], [X|L]-L, _) :- X=clause(_, _, _).
dump(Collect, [Collect|L]-L, yes_hash).

% Link all elements of Gather together with try, retry, trust:

cs_link(Type, [Gr], Code-Link) :-
        cs_endlink(Gr, Type, Code, Link).
cs_link(Type, [Gr|Rest], Code-Link) :-
        cs_midlink(Gr, Type, Code, L),
        cs_link(retry, Rest, L-Link).
        % Middle hash table or (re)try instruction:
        cs_midlink(clause(_,Lbl,_), Type, [Instr|L], L) :- !,
                Instr=..[Type,Lbl].
        cs_midlink(Gr, Type, [Instr|Hash], L) :-
                hash(Gr, Hash-HLink),
                Instr=..[Type,else,ElseLbl],
                HLink=[label(ElseLbl)|L].

        % Last hash table or trust instruction:
        cs_endlink(clause(_,Lbl,_), _, [trust(Lbl)|Link], Link) :- !.
        cs_endlink(Gr, Type, Code, Link) :-
                hash(Gr, Hash-Link),
                cs_addtrust(Type, Code, Hash).

        % Add a trust if necessary:
        cs_addtrust(try, Hash, Hash) :- !.
        cs_addtrust(_, [trust(else,fail)|Hash], Hash).

% Generate hash table with switch instruction:
% This routine is mainly cosmetic.
hash(Gr, Code-Link) :-
        hash_table(Gr, HashTbl-Link, 0, HashLen),
        Mask is 2*HashLen-1,
        cs_kind(Gr, Kind),
        Code=[switch(Kind,Mask,Label),label(Label)|HashTbl].

% See if Gr is a bunch of constants or structures:
% No parameter needs to be passed to cs_block for this.
cs_kind([clause(FArg,_,_)|_], Kind) :- kind(FArg, Kind).

% Construct hash table.
% Dummy code here:

% put final pair on end, pad with fail instructions
hash_table([clause(FArg,Lbl,_)], [cdrpair(FArg,Lbl)|FailList]-Link,SoFar,Len)
        :-
        SoFar1 is SoFar + 1,
        ceil_2(SoFar1,Len),     % hash table length must be power of 2.
        PadLen is Len - SoFar1,
        failpad(FailList,PadLen,Link).
hash_table([clause(FArg,Lbl,_)|Rest], [pair(FArg,Lbl)|Hash]-Link, SoFar, Len)
        :-
        SoFar1 is SoFar + 1,
        hash_table(Rest, Hash-Link, SoFar1, Len).

% General utility: Returns kind of argument, can be
% 'variable', 'list', 'constant', 'structure'.
% Argument is in form struc/arity for lists and structures.
kind(Arg, variable) :- var(Arg), !.
kind(Arg, constant) :- atomic(Arg), !.
kind('.'/2, list) :- !.
kind(_, structure).

% Pad end of hash table with pairs of fails.
failpad(Link,0,Link).
failpad([cdrpair(fail,fail)|Rest],More,Link) :-
        M1 is More - 1, failpad(Rest,M1,Link).

% Find smallest power of two larger than given value.
ceil_2(In,Out) :- ceil_2(In,Out,1).
ceil_2(In,Power,Power) :- In =< Power, !.
ceil_2(In,Out,Power) :- Power2 is Power*2, ceil_2(In,Out,Power2).

/**********************************************************************/
```

```
% Compile a Clause:

compileclause(Clause, Finalcode, Link) :-
        pretrans(Clause, Pretrans),
        Pretrans=[Head|Body], colvars(Head, HeadVars),
                permvars(Pretrans, Vars, Perms),
        unravel(Pretrans, Unravel, Perms),
        partobj(Unravel, PartObj, Perms),
                permalloc(Perms),
        valvar(PartObj, HeadVars),
                        varlist(Unravel, VarList),
                        lifetime(VarList, LifeList, Forward, Backward),
        varinit(Forward, Backward, PartObj, Newobj),
                        tempalloc(VarList, LifeList),
        objcode(Newobj, ObjCode),
        excess(ObjCode,ObjCode2),
        envsize(ObjCode2, MaxSize),
        voidalloc(ObjCode2, VCode),
        assn_elim(VCode, ACode),
        peephole(ACode, Finalcode, Link, MaxSize),
        !.

/*********************************************************************/

% Set utilities used in the PLM compiler.
% "v" at the end of a name means no unification done.


in(X, L) :- memberv(X, L).
notin(X, L) :- memberv(X, L), !, fail.
notin(X, L).

unionv(S1, S2, S1) :- S1==S2.
unionv([X|S1], S2, Res) :-
        memberv(X, S2), !,
        unionv(S1, S2, Res).
unionv([X|S1], S2, [X|Res]) :-
        unionv(S1, S2, Res).
unionv([], S, S).

diffv([X|S1], S2, Res) :-
        memberv(X, S2), !,
        diffv(S1, S2, Res).
diffv([X|S1], S2, [X|Res]) :-
        diffv(S1, S2, Res).
diffv([], _, []).

intersectv([X|Set1], Set2, Res) :-
        (in(X, Set1); notin(X, Set2)), !,
        intersectv(Set1, Set2, Res).
intersectv([X|Set1], Set2, [X|Res]) :-
        intersectv(Set1, Set2, Res).
intersectv([], _, []).

includev(X, S1, S1) :- in(X, S1), !.
includev(X, S1, [X|S1]).

/*********************************************************************/

% List processing utilities used in the PLM compiler.
% These are a subset of a much larger collection.

%    list(L) succeeds if and only if L is a list.
%    nonlist(S) succeeds if and only if S is not a list.
%    No unification is done.

list(Term) :- nonvar(Term), Term=[_|_].

nonlist(Term) :- list(Term), !, fail.
nonlist(_).
```

```
%    full_list(L) succeeds if and only if L is a complete list (all
%    the cdrs are also lists) or [].
%    No unification is done.

full_list(L) :- var(L), !, fail.
full_list([]) :- !.
full_list([_|L]) :- full_list(L).


%   concat(Part1, Part2, Combined) and
%   my_append(Part1, Part2, Combined)
%   are true when all three arguments are lists, and the members of Combined
%   are the members of Part1 followed by the members of Part2.  It may be
%   used to form Combined from a given Part1 and Part2, or to take a given
%   Combined apart.  E.g. we could define member/2 (from SetUtl.Pl) as
%       member(X, L) :- my_append(_, [X|_], L).

concat([], L, L).
concat([H|L1], L2, [H|Res]) :- concat(L1, L2, Res).

my_append(A, B, C) :- concat(A, B, C).


%   concat(Part1, Part2, Part3, Combined)
%   concat(Part1, Part2, Part3, Part4, Combined)
%   concat(Part1, Part2, Part3, Part4, Part5, Combined)
%   are extensions of concat for three, four, and five sublists respectively.
%   Concat can also be used to decompose lists into all combinations of
%   three, four, and five parts.

concat([], L2, L3, Res) :- concat(L2, L3, Res).
concat([H|L1], L2, L3, [H|Res]) :- concat(L1, L2, L3, Res).

concat([], L2, L3, L4, Res) :- concat(L2, L3, L4, Res).
concat([H|L1], L2, L3, L4, [H|Res]) :- concat(L1, L2, L3, L4, Res).

concat([], L2, L3, L4, L5, Res) :- concat(L2, L3, L4, L5, Res).
concat([H|L1], L2, L3, L4, L5, [H|Res]) :- concat(L1, L2, L3, L4, L5, Res).


% length of a list

my_length([], 0).
my_length([_|L], N) :- my_length(L, N1), N is N1+1 .

%   last(List, Last)
%   is true when List is a list and Last is its last element.
%   This could be defined as last(L,X) :- my_append(_, [X], L).

last([Last], Last) :- !.
last([_|List], Last) :- last(List, Last).


%   my_member(Elem, List)
%   is true if Elem is a member of List.  This can be used as a checker,
%   as a generator of elements, or as a generator of lists.

my_member(X, [X|_]).
my_member(X, [_|L]) :- my_member(X, L).


%   memberchk(Elem, List)
%   same as member, but used only to test membership.
%   This is faster and uses less memory than the more general version.
%   memberv does not use unification to test for membership.

memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :- memberchk(X, L).
```

```
memberv(X, [Y|_]) :- X==Y, !.
memberv(X, [_|L]) :- memberv(X, L).


%   reverse(List, Reversed)
%   is true when List and Reversed are lists with the same elements
%   but in opposite orders.  rev/2 is a synonym for reverse/2.

rev(List, Reversed) :- reverse(List, [], Reversed).

reverse(List, Reversed) :- reverse(List, [], Reversed).

reverse([], Reversed, Reversed).
reverse([Head|Tail], Sofar, Reversed) :-
        reverse(Tail, [Head|Sofar], Reversed).



%   flatten(List, FlatList-Link)
%   flattens a list by removing all nesting.  FlatList consists of
%   all atoms nested to any depth in List, but all on one level.

flatten([], Link-Link).
flatten([A|L], [A|F]-Link) :-
        (atomic(A);var(A)), !, flatten(L, F-Link).
flatten([A|L], F-Link) :-
        flatten(A, F-FL), flatten(L, FL-Link).


%   mapcar(Structure, List1, List2)
%   Calls the goal Structure+elem of List1+elem of List2 for each pair
%   of elements of List1 and List2.
%   generalization of the Lisp function mapcar.

mapcar(Call, List1, List2) :- !,
        Call=..[Func|Args],
        xmapcar(Func, Args, List1, List2).

xmapcar(Func, Args, [A|L1], [B|L2]) :- !,
        concat(Args, [A,B], GoalArgs),
        Goal=..[Func|GoalArgs],
        call(Goal), !,
        xmapcar(Func, Args, L1, L2).
xmapcar(_, _, [], []).

%   mapcar(Functor, List1, List2, List3)
%   same as Lisp's mapcar, except has three arguments.

mapcar(Func, [A|L1], [B|L2], [C|L3]) :- !,
        Term=..[Func, A, B, C],
        call(Term), !,
        mapcar(Func, L1, L2, L3).
mapcar(_, [], [], []).



%   listify(Structure, ListForm)
%   converts a general structure to a Lisp-like list form.

listify(X, X) :- (atomic(X); var(X)), !.
listify(Structure, [Func|LArgs]) :-
        Structure=..[Func|Args],
        mapcar(listify, Args, LArgs).


%   linkify(List, DiffList-Link)
%   converts a list into a difference list.
```

```
linkify([], Link-Link).
linkify([A|List], [A|DiffList]-Link) :-
        linkify(List, DiffList-Link).

/*********************************************************************/

% Special utilities used by the clause compiler:

% Built-in procedures which do not destroy any
% argument registers:
% Includes arity so user can define routines with same
% name but different arity.
escape_builtin(Goal) :-
        Goal=..[Name|Args],
        my_length(Args, Arity),
        escape_builtin(Name, Arity).

% Note: The escape_builtins not,=,+ are done in pretrans.
% The unify operator '=' is part & parcel of the compiler.
% However, all four must be listed here for correct compilation.

% 12/4 - added escape_builtin routines to
% handle global variables, set and access.
% - Wayne

% Some of the escape_builtins are implemented with existing instructions:
escape_builtin(!, 0).
escape_builtin('->',0). % For correct compilation of if-then-else.
escape_builtin(nl, 0).
escape_builtin(true, 0).
escape_builtin(fail, 0).
escape_builtin(repeat, 0).
% escape_builtin('+', 1).
escape_builtin(var, 1).
escape_builtin('not', 1).
escape_builtin(atom, 1).
escape_builtin(list, 1).
escape_builtin(write, 1).
escape_builtin(writeq, 1).
escape_builtin(nonvar, 1).
escape_builtin(atomic, 1).
escape_builtin(number, 1).
escape_builtin(integer, 1).
escape_builtin(nonlist, 1).
escape_builtin(structure, 1).
escape_builtin('=', 2).
escape_builtin('<', 2).
escape_builtin('>', 2).
escape_builtin('==', 2).
% escape_builtin('=', 2).
escape_builtin('<=', 2).
escape_builtin('=<', 2).
escape_builtin('>=', 2).
% escape_builtin('==', 2).
escape_builtin('=..', 2).
escape_builtin(set,2).          % for global variables
escape_builtin(access,2).       % for global variables
escape_builtin('is', 2).
escape_builtin('+', 3).
escape_builtin('-', 3).
% escape_builtin('/', 3).
% escape_builtin('/', 3).
escape_builtin('is', 4).
escape_builtin(functor, 3).     % Added 11/15/86.
escape_builtin(arg, 3).         % Added 1/15/87.
escape_builtin(name,2).
escape_builtin(system,1).
escape_builtin(consult,1).
escape_builtin(reconsult,1).
```

```
% additional built-ins not in original list.
% escape_builtin('=\=',2).
escape_builtin('<>',2).
escape_builtin(abolish,2).
escape_builtin(assert,1).
% escape_builtin(call,1).          Because call/1 kills temporaries. 11/16/86.
escape_builtin(length,2).
escape_builtin(put,1).
escape_builtin(get,1).
escape_builtin(get0,1).
escape_builtin(read,1).
escape_builtin(retract,1).
escape_builtin(see,1).
escape_builtin(seen,0).
escape_builtin(tab,1).
escape_builtin(tell,1).
escape_builtin(told,0).


% Get type and argument of an instruction:
type_arg(get(T,R,X), T, R).
type_arg(put(T,R,X), T, R).
type_arg(unify(T,R), T, R).

% Maximum:
max(A, B, A) :- A>=B, !.
max(A, B, B) :- A=<B, !.

% Collect variables in a structure.
colvars(S, Vars) :-
        S=..[_|SL],
        split_avs(SL, Vars).
split_avs([A|Args], Vars) :-
        atomic(A), !,
        split_avs(Args, Vars).
split_avs([V|Args], Vars) :-
        var(V), !,
        split_avs(Args, VL),
        includev(V, VL, Vars).
split_avs([S|Args], Vars) :-
        S=..[_|SA],
        split_avs(SA, VL1),
        split_avs(Args, VL2),
        unionv(VL1, VL2, Vars).
split_avs([], []).

% Extract all variable terms from input list
% and put them in a difference list:
getvars(V, [V|Link]-Link) :- var(V), !.
getvars(V, Link-Link) :- nonlist(V), !.
getvars([V|List], Out) :-
        var(V), !,
        Out=[V|Vars]-Link, % Changed for bug in v2.1 BIM-Prolog
        getvars(List, Vars-Link).
getvars([X|List], Vars-Link) :-
        nonvar(X), !,
        getvars(List, Vars-Link).
getvars([], Link-Link).


% Mapping utilities for G.P. traversing of
% clause code.

% 1. Map over a clause (no dependencies):
% Result has same structure as input.
% Call may be a structure with one argument.
mapclause(Call, [X|XRest], [Y|YRest]) :-
        X=(_;_), !,
        mapdis(Call, X, Y),
        mapclause(Call, XRest, YRest).
mapclause(Call, [X|XRest], [Y|YRest]) :-
        Call=..List,
        concat(List, [X,Y], GoalList),
        G=..GoalList,
        call(G),
        mapclause(Call, XRest, YRest).
mapclause(_, [], []).

mapdis(Call, (X;XRest), (Y;YRest)) :-
        mapclause(Call, X, Y),
        mapdis(Call, XRest, YRest).
mapdis(Call, X, Y) :-
        mapclause(Call, X, Y).


% 2. Mapclause with three inputs:
mapclause(Call, [X|XRest], [Y|YRest], [Z|ZRest]) :-
        X=(_;_), !,
        mapdis(Call, X, Y, Z),
        mapclause(Call, XRest, YRest, ZRest).
mapclause(Call, [X|XRest], [Y|YRest], [Z|ZRest]) :-
        Call=..[A],
        G=..[A,X,Y,Z],
        call(G),
        mapclause(Call, XRest, YRest, ZRest).
mapclause(_, [], [], []).

mapdis(Call, (X;XRest), (Y;YRest), (Z;ZRest)) :-
        mapclause(Call, X, Y, Z),
        mapdis(Call, XRest, YRest, ZRest).
mapdis(Call, X, Y, Z) :-
        mapclause(Call, X, Y, Z).


% Repeat loop in Prolog.
% by Warren.
% range(10,I,30) succeeds with I=10, 11, ..., 30, and then fails.
% range(L,L,L) :- !.
% range(L,I,H) :-
%    K is (H+L)//2,
%    range(L,I,K).
% range(L,I,H) :-
%    K is 1+(H+L)//2,
%    range(K,I,H).

range(L,L,H).
range(L,I,H) :- L<H, L1 is L+1, range(L1,I,H).

/*********************************************************************/

% Memory management: cleaning up of the heap.

gc(Call) :-
%       prolog_version(cprolog),
        c_gc(Call).
% Can use same trick on bimprolog
%gc(Call) :-
%       prolog_version(bimprolog),
%       call(Call).

c_gc(Call) :- one_call(Call), lock(Call).
c_gc(Call) :- unlock(Call).

one_call(Call) :- call(Call), !.

lock(Term) :-
        abolish(info_lock, 1),
        assert(info_lock(Term)), fail.
```

```
unlock(Term) :-
        retract(info_lock(Term)),
        abolish(info_lock, 1).

/**********************************************************************/

% IO Package

% Arity of the compiled code never goes above 7, but the actual arity
% of the predicate is output here in order to distinguish predicates
% with arities>7.

write_plm(NameArity, List) :-
        compile_options(l), !,
        write_plm_list(NameArity, List), nl, nl, !.
write_plm(NameArity, List) :-
        (prolog_version(sbprolog) -> not(compile_options(l));
                                     \+(compile_options(l))), !,
        write_plm_nice(NameArity, List), nl, nl, !.

% Write the procedure code in human-readable form:
write_plm_nice(NameArity, List) :-
        write('procedure  '), write(NameArity), nl, nl,
        write_plm_nice(List).

write_plm_nice([I|List]) :-
        winstr(I),
        write_plm_nice(List), !.
write_plm_nice([]).

% Write the procedure code in list-form, able to be read by read/1:
write_plm_list(NameArity, List) :-
        write('[procedure('), writeq(NameArity), write('),'), nl,
        write_plm_list(List),
        write('].').

write_plm_list([I]) :-
        writeq(I).
write_plm_list([I|List]) :-
        writeq(I), comma, nl,
        write_plm_list(List).
write_plm_list([]).

% Write arguments separated by commas:
wcomma([A]) :- warg(A), nl.
wcomma([A|L]) :- warg(A), comma, wcomma(L).
wcomma([]) :- nl.

% Write a label or constant label:
wlbl(L) :- var(L), compile_options(a(A)), atomic(A), !,
           write(L), und, write(A).
wlbl(L) :- var(L), !, write(L).
wlbl(X) :- write(X).

% Write an argument:
warg(Lbl) :- var(Lbl), wlbl(Lbl). % var/1 needed here.
warg(x(I)) :- write('X'), write(I), !.
warg(y(I)) :- write('Y'), write(I), !.
warg(N) :- number(N), write('&'), write(N), !.
warg(C) :- compile_options(q), write(''''),write(C),write(''''), !.
warg(C) :- (prolog_version(sbprolog) -> not(compile_options(q));
                                        \+(compile_options(q))), writeq(C), !.
```

```
% Write a single instruction on a line:
winstr(X) :- atomic(X), wtabln(X).
winstr(fail/0) :- wtabln(fail).
winstr(label(L)) :- wlbl(L), wln(':').
winstr(execute(L)) :- wtab(execute), space, wlbl(L), nl.
winstr(cutd(L)) :- wtab(cutd), space, wlbl(L), nl.
winstr(pair(A,B)) :- tabl, warg(A), nl, tabl, wlbl(B), nl.
winstr(cdrpair(A,B)) :- A==fail, B==fail,
                        wtab(fail), wtabln(tcdr), wtabln(fail).
winstr(cdrpair(A,B)) :- tabl, warg(A), wtabln(tcdr), tabl, wlbl(B), nl.
winstr(switch_on_term(A,B,C)) :-
        wtab(switch_on_term), space, wcomma([A,B,C]).
winstr(switch(Kind,Mask,Lbl)) :-
        wtab(switch_on_), write(Kind), space,
        write(Mask),comma,wlbl(Lbl),nl.
winstr(unify(void,N)) :- wtab(unify_void), space, wln(N).
winstr(Instr) :-
        Instr=..[Name,Type|Args],
        (Name=unify; Name=get; Name=put),
        wtab(Name), und, write(Type), space,
        wcomma(Args).
winstr(Instr) :-
        Instr=..[Name,Arg1|Args],
        (Name=try; Name=retry; Name=trust),
        wtab(Name),
        write_else(Arg1, Args).
winstr(Instr) :-
        Instr=..[Name,Arg],
        (Name=get_nil; Name=put_nil; Name=get_list; Name=put_list),
        wtab(Name), space, warg(Arg), nl.
winstr(Instr) :-
        Instr=..[Name,Arg],
        wtab(Name), space, wln(Arg).
winstr(call(Name,N)) :-
        wtab(call), space,
        write(Name),comma,wln(N).
winstr(Name/Arity) :-
        wtab(escape), space, wln(Name/Arity).

% Write a space, comma, or underline character:
space :- write(' ').
comma :- write(',').
und :- write('_').

% Tab before or newline after:
wtab(X) :- tabl, write(X).
wln(X) :- write(X), nl.
wtabln(X) :- tabl, write(X), nl.
tabl :- put(9).


w(Expr) :- X is Expr, write(X).

wl([A|Rest]) :- write(A), nl, wl(Rest).
wl([]) :- nl.

write_else(Arg1, Args) :- Arg1==else, !,
        write('_me_else '),
        Args=[L], wlbl(L), nl.
write_else(Arg1, _) :-
        space, wlbl(Arg1), nl.

/**********************************************************************/

% Pretransformations: Recognize source forms
% and transform to forms which can be compiled.
% Also transform conjunction into list form.
```

```
pretrans((Head:-Body), [PH|PB]) :-
        % Transform head according to change of 12/20 (below).
        arity_limit(Head, PH),
        pretrans(Body, PB, []), !.
pretrans(Head, [PH]) :- arity_limit(Head, PH).

% Addition - 4/16
% Transform list form of consult and reconsult
% into explicit calls to consult and reconsult.

% Transform a disjunction:
disjpretrans(Disj, (PX;PB)) :-
        disjtest(Disj, A, B),
        nonvar(A), A=(X -> Y), !,
        pretrans(X, PX, ['->'|PY]),
        pretrans(Y, PY, []),
        disjpretrans(B, PB).
disjpretrans(Disj, (PA;PB)) :-
        disjtest(Disj, A, B), !,
        pretrans(A, PA, []),
        disjpretrans(B, PB).
disjpretrans(Last, (PX;[fail])) :-
        nonvar(Last), Last=(X->Y), !,
        pretrans(X, PX, ['->'|PY]),
        pretrans(Y, PY, []).
disjpretrans(Last, PL) :-
        pretrans(Last, PL, []).

disjtest(Disj, A, B) :- nonvar(Disj), Disj=(A;B).

% Bug fix - 11/29/84
% Transform goal consisting of single variable to call.
% I'm not sure if this is really semantically correct, but goals
% of this form are not handled elsewhere and C-Prolog handles
% them this way. - Wayne
pretrans(X, [call(X)|Link], Link) :- var(X), !.

pretrans(call(X), PX, Link) :-
        pretrans(X, PX, Link).
pretrans(not(A), [(PA;[true])|Link], Link) :-
        pretrans(A, PA, ['->',fail]).
pretrans(\+(A),  [(PA;[true])|Link], Link) :-
        pretrans(A, PA, ['->',fail]).
% Lone (X->Y) not in a disjunction:
pretrans((X -> Y), [(PX;[fail])|Link], Link) :-
        pretrans(X, PX, ['->'|PY]),
        pretrans(Y, PY, []).
pretrans((Goal,Body), PG, Link) :-
        pretrans(Goal, PG, PB),
        pretrans(Body, PB, Link).
pretrans(Disj, [PD|Link], Link) :-
        Disj=(_;_),
        disjpretrans(Disj, PD).
% pretrans('\='(X,Y), [([X=Y,'->',fail];[true])|Link], Link).
% Transform is/2 into is/4 if not using option 'u'.
pretrans((V is Exp), Is4, Link) :-
        expr_nolist(Exp, NExp),
        top_expression(NExp, V, Is4, Link).

% transform list form of re/consult to explicit call
pretrans(List, Consult, Link) :-
        full_list(List),
        expand_consult(List, Consult, Link).

% Transform subgoal according to change of 12/20.
pretrans(Goal, [PG|Link], Link) :- arity_limit(Goal, PG).
```

```
% Transform an expression of the form X is Expr
% into a series of is/4 calls.
% Recognizes unary minus and converts all binary operators.
% Unrecognized forms are kept in is/2.

% Bug fix 1/15/87: does 'V is W' correctly, where V & W are vars.

% Top level call recognizes special cases of top level.
top_expression(Expr, X, [fail|Link], Link) :-
        % X is atom or struc or list.
        nonvar(X),
        (prolog_version(sbprolog) -> not(number(X)); \+(number(X))), !.
top_expression(Expr, X, [(X = Expr)|Link], Link) :-
        number(Expr), !.
top_expression(Expr, X, [(X is Expr)|Link], Link) :-
        (var(Expr); atomic(Expr)), !.
top_expression(Expr, X, Code, Link) :-
        (prolog_version(sbprolog) -> not(compile_options(u));
                                     \+(compile_options(u))), !,
        expression(Expr, X, Code, Link).
top_expression(Expr, X, [(X is Expr)|Link], Link) :-
        compile_options(u), !.

expression(IExpr, IExpr, Link, Link) :- var(IExpr), !.
expression(IExpr, IExpr, Link, Link) :- number(IExpr), !.
expression(-(E1), OExpr, Code, Link) :- !,
        expression(E1, A1, Code, ['is'(OExpr,0,'-',A1)|Link]).
expression(IExpr, OExpr, Code, Link) :-
        compile_options(s),
        IExpr=..[Op, E1, E2],
        my_member(Op, ['+', '-' /* , '/\', '\/' */]), !,
        Pred=..[Op, A1, A2, OExpr],
        expression(E1, A1, Code, L1),
        expression(E2, A2, L1, [Pred|Link]).
expression(IExpr, OExpr, Code, Link) :-
        IExpr=..[Op, E1, E2], !,
        expression(E1, A1, Code, L1),
        expression(E2, A2, L1, ['is'(OExpr,A1,Op,A2)|Link]).
expression(IExpr, OExpr, [(OExpr is IExpr)|Link], Link).

% Bug fix 5/17/87: recognizes [expr] correctly in expressions.

% Transform [expr] to expr recursively:
% This makes is/2 compatible with C-Prolog:
expr_nolist(IExpr, IExpr) :- var(IExpr), !.
expr_nolist(IExpr, IExpr) :- number(IExpr), !.
expr_nolist(IExpr, IExpr) :- atomic(IExpr), !.
expr_nolist([IExpr], OExpr) :-
        expr_nolist(IExpr, OExpr), !.
expr_nolist(IExpr, OExpr) :-
        IExpr=..[Op|IArgs],
        mapcar(expr_nolist,IArgs,OArgs),
        OExpr=..[Op|OArgs].

% Expand consult shorthand into explicit calls to re/consult:
% Ignore nonatomic items in the consult list.
expand_consult([], Link, Link).
expand_consult([File|List], [consult(File)|Consult], Link) :-
        (var(File);atom(File)), !,
        expand_consult(List, Consult, Link).
expand_consult([-File|List], [reconsult(File)|Consult], Link) :-
        (var(File);atom(File)), !,
        expand_consult(List, Consult, Link).
expand_consult([Other|List], Consult, Link) :-
        expand_consult(List, Consult, Link).
```

```
% Addition - 12/20
% If > 7 arguments to a call, force all arguments after the first six into
% a structure passed as the seventh argument.  Must be done for heads and
% subgoals in body. - Wayne
arity_limit(Pred, PH) :-
        functor(Pred, Functor, Arity), Arity >= 8, !,
        Pred =.. [Functor, A1, A2, A3, A4, A5, A6 | Rest],
        RestArgs =.. [dummy | Rest],
        rename_goal(Functor, Arity, NewFunctor),
        PH =.. [NewFunctor, A1, A2, A3, A4, A5, A6, RestArgs].
arity_limit(Pred, NewPred) :-
        functor(Pred, Functor, Arity),
        rename_goal(Functor, Arity, NewFunctor),
        Pred=..[Functor|Args],
        NewPred=..[NewFunctor|Args].

% Embed arity into the functor name:
% Only done for nonbuiltins.
rename_goal(Functor, Arity, Functor) :-
        escape_builtin(Functor, Arity), !.
rename_goal(Functor, Arity, NewFunctor) :-
        name(Functor, FList),
        to_string(Arity, AList, []),
        name('/', [Slash]),
        concat(FList, [Slash|AList], NFList),
        name(NewFunctor, NFList).

% Convert an integer into a string:
to_string(N, [D|Link], Link) :-
        N<10, !,
        name('0', [Zero]),
        D is N+Zero.
to_string(N, String, Link) :-
        name('0', [Zero]),
        D is (N mod 10) + Zero,
        N1 is (N // 10),
        to_string(N1, String, [D|Link]).

/*********************************************************************/

% All structures are unraveled into unify goals.
% All unify goals are of the form Var1=(Var2 or Atom or Struc),
% where Var1 is temporary or permanent and
% where Struc has only variables and atoms as arguments.
% If Var1 is permanent then so is Var2.
% Preexisting unify goals are transformed into this type.
% The structure of disjunctions remains the same (i.e.
% the operator ';' remains).  Only the content is unraveled.

% Bug fix - 7/31/85:
%   Handle case where the null list is an element of a list or a structure.
%   - Wayne

unravel([Head|Body], [NewHead|Ravel], Perms) :-
        spread(Head, NewHead, Ravel-L),
        xunravel(Body, L-[], Perms), !.

xunravel([Dis|Rest], [DRavel|Ravel]-Link, Perms) :-
        Dis=(_;_),
        disunravel(Dis, DRavel, Perms),
        xunravel(Rest, Ravel-Link, Perms).
xunravel([Goal|Rest], Ravel-Link, Perms) :-
        Goal=(_=_),
        varunify(Goal, Ravel-L, Perms),
        xunravel(Rest, L-Link, Perms).
xunravel([Goal|Rest], Ravel-Link, Perms) :-
        spread(Goal, NewGoal, Ravel-L),
        L=[NewGoal|L2],
        xunravel(Rest, L2-Link, Perms).
xunravel([], Link-Link, _).
```

```
disunravel((A;B), (ARavel;BRavel), Perms) :- !,
        xunravel(A, ARavel-[], Perms),
        disunravel(B, BRavel, Perms).
disunravel(A, ARavel, Perms) :-
        xunravel(A, ARavel-[], Perms).


% Unification optimization.
% Turn the general goal 'X=Y' into a sequence
% of simpler unifications of the form
% Var1=(Var2 or Atom or Struc),
% where Var1 is a temporary or permanent variable, and
% where Struc has only atoms and variables as arguments.
varunify(X=Y, Code-Link, Perms) :-
        (xvarunify(X=Y, Code-Link, Perms); Code=[fail|Link]).

% One argument is a temporary variable:
xvarunify(A=B, [A=NewB|L]-Link, Perms) :-
        var(A), notin(A,Perms), !,
        spread(B, NewB, L-Link).
xvarunify(A=B, [B=NewA|L]-Link, Perms) :-
        var(B), notin(B,Perms), !,
        spread(A, NewA, L-Link).
% One argument is a permanent variable:
xvarunify(A=B, [A=NewB|L]-Link, Perms) :-
        in(A,Perms), !,
        spread(B, NewB, L-Link).
xvarunify(A=B, [B=NewA|L]-Link, Perms) :-
        in(B,Perms), !,
        spread(A, NewA, L-Link).
% Both arguments are nonvariables:
xvarunify(A=B, Link-Link, Perms) :-
        atomic(A), !, atomic(B), A=B.
xvarunify(A=B, Code-Link, Perms) :-
        atomic(B), !, fail.
xvarunify(A=B, Code-Link, Perms) :- % A&B are structures
        A=..[Func|ArgsA],
        B=..[Func|ArgsB],
        lvarunify(ArgsA, ArgsB, Code-Link, Perms).

lvarunify([A|ArgsA], [B|ArgsB], Code-Link, Perms) :-
        xvarunify(A=B, Code-L, Perms), !,
        lvarunify(ArgsA, ArgsB, L-Link, Perms).
lvarunify([], [], Link-Link, Perms).


% Take a (possibly nested) structure apart into
% (1) a simple structure, and (2) a series of unify goals.
% A list is considered as a structure with variable arity.
% Its cdr field is given a separate unify goal to
% accommodate the unify_cdr instruction.
spread(Var, Var, Link-Link) :- var(Var), !.
spread(Atomic, Atomic, Link-Link) :- atomic(Atomic), !.
spread(List, SimpleList, Rest-Link) :-
        list(List), !,
        argspread(CdrUnify, List, SimpleList, Ravel-Link),
        check_cdr(CdrUnify, Ravel, Rest).
spread(Struc, SimpleStruc, Rest-Link) :-
        Struc=..[Name|Args],
        argspread(_, Args, VArgs, Rest-Link),
        SimpleStruc=..[Name|VArgs].

        check_cdr(none, Ravel, Ravel) :- !.
        check_cdr(CdrUnify, Ravel, [CdrUnify|Ravel]).

argspread(none, Cdr, Cdr, Link-Link) :-
        (var(Cdr);Cdr==[]), !.
argspread(T=SimpleCdr, Cdr, T, Ravel-Link) :-
        nonlist(Cdr), !,
        spread(Cdr, SimpleCdr, Ravel-Link).
```

```
% arg is null list
argspread(CdrUnify, [S|Args], [T|VArgs], [T=[]|L]-Link) :-
        nonvar(S), S = [], !,
        argspread(CdrUnify, Args, VArgs, L-Link).
argspread(CdrUnify, [A|Args], [A|VArgs], Ravel-Link) :-
        (atomic(A); var(A)), !,
        argspread(CdrUnify, Args, VArgs, Ravel-Link).
argspread(CdrUnify, [S|Args], [T|VArgs], Ravel-Link) :-
        Ravel=[T=V|L],
        spread(S, V, L-L2),
        argspread(CdrUnify, Args, VArgs, L2-Link).

/*********************************************************************/

% Convert unraveled code into partial object code:

partobj([Head|BodyGoals], [HeadObj|BodyObj], Perms) :-
        Head=..[_|Args],
        getputblock(get, Args, HeadObj, 1),
        xpartobj(BodyGoals, Perms, BodyObj, yes), !.


xpartobj([], _, [], _).
xpartobj([Dis|Rest], Perms, Result, Flag) :-
        Dis=(_;_), !,
        % Initialize permanent variables just before first disjunction:
        initperms(Flag, Perms, Result, [DisCode|RestCode]),
        dispartobj(Dis, Perms, DisCode),
        xpartobj(Rest, Perms, RestCode, no).
xpartobj([Goal|Rest], Perms, [GoalCode|RestCode], Flag) :-
        goalpartobj(Goal, Perms, GoalCode),
        xpartobj(Rest, Perms, RestCode, Flag).

        initperms(yes, Perms, [PermInit|R], R) :- !,
                initblock(Perms, PermInit).
        initperms(_, _, R, R).

dispartobj((A;B), Perms, (ACode;BCode)) :-
        xpartobj(A, Perms, ACode, no),
        dispartobj(B, Perms, BCode).
dispartobj(A, Perms, ACode) :-
        xpartobj(A, Perms, ACode, no).


% Convert goals into their object code:
% Recognizes !, true, unify goals, and calls with simple arguments:

% Convert '!' into cut instruction:
goalpartobj(!, _, [cut|Link]-Link).
% Cut in a disjunction is handled for objcode:
goalpartobj('->', _, cutd). % Note: not a list, so objcode is signaled.
% 'true' needs no code:
goalpartobj(true, _, Link-Link).
% translation of unify goals:
goalpartobj(V=W, Perms, [put(_,V,Temp)|Code]-Link) :-
        unify_temp(V, Perms, Temp),
        unify_2ndpart(W, Temp, Code-Link).
% translation of other goals:
goalpartobj(Goal, _, Code-Link) :-
        Goal=..[Name|Args],
        my_length(Args, Arity),
        getputblock(put, Args, Code-L, 1),
        goal_call(Name, Arity, L, Link).

        % Get the temporary variable for unify goals:
        unify_temp(V, Perms, x(8)) :- in(V, Perms), !.
        unify_temp(V, Perms, V).
        % Create the call:
        goal_call(Name, Arity, [Name/Arity|L], L) :-
                escape_builtin(Name,Arity), !.
        goal_call(Name, Arity, [call(Name,_)|L], L).

% Code for second argument of '=' predicate:
unify_2ndpart(W, Temp, [get(_,W,Temp)|Link]-Link) :-
        var(W), !.
unify_2ndpart(W, Temp, [get(constant,W,Temp)|Link]-Link) :-
        atomic(W), !.
unify_2ndpart(W, Temp, [get(structure,'.'/2,Temp)|L]-Link) :-
        list(W), !,
        unifyblock(list, W, L-Link).
unify_2ndpart(W, Temp, [get(structure,Name/Arity,Temp)|L]-Link) :- !,
        W=..[Name|Args], my_length(Args, Arity),
        unifyblock(nonlist, Args, L-Link).

% Initialization of variables:
% Uses register 8 as a holder.
initblock([], Link-Link).
initblock([V|Vars], [put(_,V,x(8))|Rest]-Link) :-
        initblock(Vars, Rest-Link).

% Get or put of all head arguments:
% (If Type is get or put).
getputblock(Type, [A|Args], [X|Rest]-Link, N) :-
        X=..[Type,T,A,x(N)],
        (atomic(A) -> T=constant; true),
        N1 is N+1,
        getputblock(Type, Args, Rest-Link, N1).
getputblock(_, [], Link-Link, _).

% Block of unify instructions to unify structures or lists:
unifyblock(nonlist, [], [unify_nil|Link]-Link).
unifyblock(list, V, [unify(cdr,x(8)),get(_,V,x(8))|Link]-Link) :- var(V), !.
unifyblock(list, [], [unify_nil|Link]-Link) :- !.
unifyblock(Type, [A|Args], [unify(T,A)|Rest]-Link) :-
        (atomic(A) -> T=constant; true),
        unifyblock(Type, Args, Rest-Link).

/*************************************************************************/

% Adding initialization instructions
% in disjunctions to variables which need it.
% Result is a modified PartObj.
% Traverses code once; passes over everything without
% a passing glance except disjunctions.

% Must be used before tempalloc.

varinit(Forward, Backward, Partobj, Newobj) :-
        xvarinit(Forward, Backward, Partobj, Newobj-[]), !.

xvarinit([_], _, X, R-L) :- linkify(X, R-L), !.

% The first two clauses traverse Forward, Backward, and PartObj
% until a disjunction is found:
xvarinit([_,FIn|Forward], [_,BIn|Backward], PartObj, NewObj) :-
        (prolog_version(sbprolog) -> not(FIn=(_;_)); '\+'(FIn=(_;_))), !,
        % Note: since Forward and Backward have identical
        % structure, only one must be tested.
        xvarinit([FIn|Forward], [BIn|Backward], PartObj, NewObj), !.
xvarinit(Forward, Backward, [G|PartObj], [G|NewObj]-Link) :-
        (prolog_version(sbprolog) -> not(G=(_;_)); '\+'(G=(_;_))), !,
        xvarinit(Forward, Backward, PartObj, NewObj-Link), !.
```

```
% At this stage all three arguments have disjunctions:
xvarinit([FLeft,(FA;FB),FRight|Forward],
         [BLeft,(BA;BB),BRight|Backward],
         [(A;B)|PartObj], [(NA;NB)|NewObj]-Link) :- !,
        diffv(FRight, FLeft, T),
        intersectv(T, BRight, V),
        dis_varinit(V, (FA;FB), (BA;BB), (A;B), (NA;NB)),
        xvarinit([FRight|Forward], [BRight|Backward], PartObj, NewObj-Link), !.

dis_varinit(V, (FA;FB), (BA;BB), (A;B), (NA;NB)) :-
        one_choice(V, FA, BA, A, NA),
        dis_varinit(V, FB, BB, B, NB).
dis_varinit(V, FA, BA, A, NA) :-
        one_choice(V, FA, BA, A, NA).

one_choice(V, FA, BA, A, NA) :-
        xvarinit(FA, BA, A, NA-Link),
        last(FA, FLast),
        diffv(V, FLast, InitVars),
        add_init_list(InitVars, Link).

        add_init_list([], []) :- !.
        add_init_list(InitVars, [InitInstr]) :- init_list(InitVars, InitInstr).

init_list([V|Vars], [put(variable,V,V)|Rest]-Link) :-
        init_list(Vars, Rest-Link).
init_list([], Link-Link).

/*******************************************************************/

% Turn partial object code, which still contains the
% hierarchy of goals and disjunctions, into a uniform list.
% The control instructions for disjunctions are compiled and
% the labels for the cut instructions are instantiated.
objcode(PartObj, ObjCode) :-
        xobjcode(PartObj, ObjCode-[], proc, _), !.

xobjcode([], Link-Link, _, _).
xobjcode([cutd|RestCode], [cutd(CutLbl)|C]-Link, CutLbl, yes) :-
        xobjcode(RestCode, C-Link, CutLbl, _).
xobjcode([Code-L|RestCode], Code-Link, CutLbl, IsCut) :-
        xobjcode(RestCode, L-Link, CutLbl, IsCut).
xobjcode([(X;Choices)|RestCode], [try(else,L1)|ChCode]-Link, CutLbl, IsCut) :-
        xobjcode(X, ChCode-ChLink, L1, _),
        ChLink=[execute(EndLbl),label(L1)|C3],
        xdiscode(Choices, C3-L, EndLbl),
        xobjcode(RestCode, L-Link, CutLbl, IsCut).

xdiscode((X;Choices), [retry(else,L2)|ChCode]-Link, EndLbl) :-
        xobjcode(X, ChCode-ChLink, L2, _),
        ChLink=[execute(EndLbl),label(L2)|C3],
        xdiscode(Choices, C3-Link, EndLbl).
xdiscode(LastChoice, Code-Link, EndLbl) :-
        xobjcode(LastChoice, ChCode-ChLink, CutLbl, IsCut),
        lastchoice(IsCut,CutLbl,EndLbl,Code,ChCode,ChLink,L),
        L=[label(EndLbl)|Link].

        % Handle case of cut in last choice:
        lastchoice(IsCut,CutLbl,EndLbl,Code,ChCode,ChLink,L) :-
            IsCut==yes, !,
            Code=[retry(else,CutLbl)|ChCode],
            ChLink=[execute(EndLbl),label(CutLbl),trust(else,fail),fail/0|L].
        lastchoice(IsCut,CutLbl,EndLbl,Code,ChCode,ChLink,L) :-
            Code=[trust(else,fail)|ChCode],
            ChLink=L.

/*******************************************************************/
```

```
% Value-variable annotation:
% Assumes that initializations of variables
% that needed it have been added to the code.
% Assumes that code still contains disjunction structure.

% Pass 1: First occurrences of all variables are
%         marked 'variable'.  All variables occurring
%         first in a 'put' are marked unsafe.  Later,
%         'excess' will only allow permanents to keep
%         the unsafe annotation.
% Pass 2: Do a reverse pass.  First encounters of
%         unsafe variables are marked 'unsafe_value',
%         unless they are already marked 'variable'.
%         All other encounters with variables are marked
%         'value'.

% Must be done before temporary variable allocation and
% after calculation of permanent variables.

% Variables encountered so far are kept in the set SoFar
% in both passes.  This set is passed in parallel across
% disjunctions, and the different SoFar's are united upon
% exiting disjunctions.

% Top level:
valvar(PartObj, HeadVars) :-
        valvar1(PartObj, [], PossUnSafe, [], _), !,
        diffv(PossUnSafe, HeadVars, UnSafe),
        valvar2(PartObj, UnSafe, [], _), !.

% Pass 1:
valvar1(V, UnSafe, UnSafe, SF, SF) :- (var(V);V=[]).

valvar1([(A;B)|RestCode], InUS, OutUS, SoFar, OutSF) :-
        disvalvar1((A;B), InUS, US1, SoFar, NewSF),
        valvar1(RestCode, US1, OutUS, NewSF, OutSF).
valvar1([G-L|RestCode], InUS, OutUS, SoFar, OutSF) :-
        valvar1(G, InUS, US1, SoFar, NewSF),
        valvar1(RestCode, US1, OutUS, NewSF, OutSF).
valvar1([I|RestInstr], InUS, OutUS, SoFar, OutSF) :-
        type_arg(I, T, X), !,
        (notin(X, SoFar) , T=variable; true),
        new_us(I, X, SoFar, InUS, US1),
        unionv([X], SoFar, NewSF),
        valvar1(RestInstr, US1, OutUS, NewSF, OutSF).
valvar1([_|RestInstr], InUS, OutUS, SoFar, OutSF) :-
        valvar1(RestInstr, InUS, OutUS, SoFar, OutSF).

        new_us(I, X, SoFar, InUS, US1) :-
                I=put(_,_,_), notin(X, SoFar), !,
                unionv([X], InUS, US1).
        new_us(I, X, SoFar, InUS, InUS).

disvalvar1((A;B), InUS, OutUS, SoFar, OutSF) :-
        valvar1(A, InUS, US1, SoFar, Out1),
        disvalvar1(B, US1, OutUS, SoFar, Out2),
        unionv(Out1, Out2, OutSF).
disvalvar1(B, InUS, OutUS, SoFar, OutSF) :-
        valvar1(B, InUS, OutUS, SoFar, OutSF).

% Pass 2:
valvar2(V, _, SF, SF) :- (var(V); V=[]).

valvar2([(A;B)|RestCode], UnSafe, SoFar, OutSF) :-
        valvar2(RestCode, UnSafe, SoFar, NewSF),
        disvalvar2((A;B), UnSafe, NewSF, OutSF).
valvar2([G-L|RestCode], UnSafe, SoFar, OutSF) :-
        valvar2(RestCode, UnSafe, SoFar, NewSF),
        valvar2(G, UnSafe, NewSF, OutSF).
```

```
valvar2([I|RestInstr], UnSafe, SoFar, OutSF) :-
        type_arg(I, T, X), !,
        valvar2(RestInstr, UnSafe, SoFar, NewSF),
        choose_annotation(X, UnSafe, NewSF, T),
        unionv([X], NewSF, OutSF).
valvar2([_|RestInstr], UnSafe, SoFar, OutSF) :-
        valvar2(RestInstr, UnSafe, SoFar, OutSF).

        choose_annotation(X, UnSafe, NewSF, T) :-
                in(X, UnSafe), notin(X, NewSF), !,
                make_unsafe_value(T).
        choose_annotation(X, UnSafe, NewSF, T) :-
                make_value(T).

disvalvar2((A;B), UnSafe, SoFar, OutSF) :-
        valvar2(A, UnSafe, SoFar, Out1),
        disvalvar2(B, UnSafe, SoFar, Out2),
        unionv(Out1, Out2, OutSF).
disvalvar2(B, UnSafe, SoFar, OutSF) :-
        valvar2(B, UnSafe, SoFar, OutSF).

% Make unsafe_value if possible
make_unsafe_value(unsafe_value) :- !.
make_unsafe_value(_) :- !.

% Make value if possible
make_value(value) :- !.
make_value(_) :- !.

/**********************************************************************/

% Find all permanent variables
permvars([Head|Body], Vars, Perms) :-
        colvars(Head, HeadVars),
        xpermvars(Body, [HeadVars,[],[]], [Vars,Half,Perms]), !.

xpermvars([], AllVars, AllVars).

% Disjunction:
xpermvars([Dis|Rest], SoFar, Out) :-
        Dis=(_;_), !,
        disxpermvars(Dis, SoFar, NewSoFar),
        xpermvars(Rest, NewSoFar, Out).

% Conjunction:
xpermvars([A|Rest], SoFar, Out) :-
        SoFar=[Vars, Half, Perms],
        colvars(A, AVars),
        intersectv(AVars, Half, P),
        unionv(Perms, P, NewPerms), % Fresh variables at end of NewPerms.
        unionv(AVars, Vars, NewVars),
        newhalf(A, Half, NewVars, NewHalf),
        NewSoFar=[NewVars, NewHalf, NewPerms],
        xpermvars(Rest, NewSoFar, Out).

        % calculate new Half permanent set:
        newhalf(A, Half, NewVars, Half) :-
                escape_builtin(A), !.
        newhalf(A, Half, NewVars, NewHalf) :-
                unionv(NewVars, Half, NewHalf).

        disxpermvars((A;B), SoFar, Out) :- !,
                xpermvars(A, SoFar, OutA),
                disxpermvars(B, SoFar, OutB),
                mapcar(unionv, OutA, OutB, Out). % Fresh vars at end of Perms.
        disxpermvars(B, SoFar, Out) :-
                xpermvars(B, SoFar, Out).

/**********************************************************************/
```

```
% Trivial permalloc
% Variables at end of list are numbered lowest.

permalloc(PermVars) :-
        permalloc(PermVars, _).

permalloc([y(I)|Vars], I) :-
        permalloc(Vars, I1),
        I is I1+1 .
permalloc([], 0).

/*********************************************************************/

% Calculate from the unraveled source code
% the varlist used for calculating lifetimes.
% All goal arguments (variables & atoms) are simply listed.
% For unify goals only the variables are listed.
% Goal arguments are delimited by one or both of arity(Arity) and fence(Name).
% This is determined as follows:
%       1. arity(Arity) allows tempalloc to do more optimal allocation.
%          It comes before the arguments.
%          It is generated for all goals, even built-ins (except unify,
%          or goals with arity zero).
%       2. fence(Name) is used in lifetime to kill temporaries.
%          It comes after the arguments.
%          It is not generated for built-ins or the head of the clause.
%
% 11/15/84:
% Correction - last line of item 1 used to be:
%
%          or goals with arity zero, or if all arguments are nonvariable).
%
% This is incorrect because even nonvariable arguments will use registers,
% so tempalloc will have to be made aware of them.
% Fourth line of goalsvars used to be
%       ((Arity=0;getvars(Args, []-[])) -> Vars=L;

varlist([Head|RestCode], [arity(Arity)|Vars]) :-
        Head=..[Name|Args],
        my_length(Args, Arity),
        linkify(Args, Vars-L),
        xvarlist(RestCode, L-[]), !.

xvarlist([X|RestCode], [Dis|Vars]-Link) :-
        X=(_;_),
        dislist(X, Dis),
        xvarlist(RestCode, Vars-Link).
xvarlist([Goal|RestCode], Vars-Link) :-
        goalsvars(Goal, Vars-L),
        xvarlist(RestCode, L-Link).
xvarlist([], Link-Link).

dislist((A;B), (AVars;BVars)) :-
        xvarlist(A, AVars-[]),
        dislist(B, BVars).
dislist(B, BVars) :-
        xvarlist(B, BVars-[]).

goalsvars(A=S, Vars_Link) :-
        var(S), !,
        getvars([A,S], Vars_Link).
goalsvars(A=S, Vars_Link) :-
        list(S), !,
        getvars([A|S], Vars_Link).
goalsvars(A=S, Vars_Link) :-
        atom(S), !,
        getvars([A], Vars_Link).
goalsvars(A=S, Vars_Link) :-
        S=..[_|SVars],
        getvars([A|SVars], Vars_Link).
```

```
goalsvars(Goal, Link-Link) :-
        atom(Goal), escape_builtin(Goal,0), !.
goalsvars(Goal, [fence(Name)|Link]-Link) :-
        atom(Goal), !.
goalsvars(Goal, [arity(Arity)|V]-Link) :-
        Goal=..[Name|Args],
        my_length(Args,Arity),
        escape_builtin(Name,Arity), !,
        linkify(Args, V-Link).
goalsvars(Goal, [arity(Arity)|V]-Link) :-
        Goal=..[Name|Args],
        my_length(Args,Arity),
        linkify(Args, V-[fence(Name)|Link]).

/*********************************************************************/

% Calculate lifetimes of all temporary
% variables using the varlist.
% (Permanents must be allocated beforehand)
% Uses fence(_) to forget temporaries.
% Two passes needed: Down & back up.
% Lots of verbose superfluous code used.

lifetime(VarList, LifeList, ForwList, BackList) :-
        ForwList=[[]|_],
        forward(VarList, ForwList, _),
        backward(VarList, BackList, []),
        mapclause(intersectv, ForwList, BackList, LifeList), !.


% Forward Pass:
% Watch out for data flow!
% FLast is an output, FLeft is given.
forward([X|Rest], [FLeft,FRight|FRest], FLast) :-
        var(X), !,
        unionv([X], FLeft, FRight),
        forward(Rest, [FRight|FRest], FLast).
forward([fence(_)|Rest], [_,[]|FRest], FLast) :-
        forward(Rest, [[]|FRest], FLast).
forward([Dis|Rest], [FLeft,FIn,FRight|FRest], FLast) :-
        Dis=(_;_),
        forwdis(Dis, [FLeft,FIn], FRight),
        forward(Rest, [FRight|FRest], FLast).
forward([_|Rest], [FLeft,FLeft|FRest], FLast) :-
        forward(Rest, [FLeft|FRest], FLast).
forward([], [FLast], FLast).

% Given: FLeft.
% To be calculated: AIn,BIn,FRight.
forwdis((A;B), [FLeft,(AIn;BIn)], FRight) :-
        AIn=[FLeft|_],
        forward(A, AIn, ARight),
        forwdis(B, [FLeft,BIn], BRight),
        unionv(ARight, BRight, FRight).
forwdis(B, [FLeft,BIn], FRight) :-
        BIn=[FLeft|_],
        forward(B, BIn, FRight).


% Backward Pass:
% Watch out for convoluted data flow!
% BLast is an input, others (BLeft, BRight) are outputs.
backward([X|Rest], [BLeft,BRight|BRest], BLast) :-
        var(X), !,
        backward(Rest, [BRight|BRest], BLast),
        unionv([X], BRight, BLeft).
backward([fence(_)|Rest], [[],L|BRest], BLast) :-
        backward(Rest, [L|BRest], BLast).
backward([Dis|Rest], [BLeft,BIn,BRight|BRest], BLast) :-
        Dis=(_;_),
        backward(Rest, [BRight|BRest], BLast),
        backdis(Dis, [BLeft,BIn,BRight]).
backward([_|Rest], [BLeft,BLeft|BRest], BLast) :-
        backward(Rest, [BLeft|BRest], BLast).
backward([], [BLast], BLast).

% Given: BRight.
% To be calculated: XIn,YIn,BLeft.
backdis((X;Y), [BLeft,(XIn;YIn),BRight]) :-
        XIn=[XLeft|_],
        backward(X, XIn, BRight),
        backdis(Y, [YLeft,YIn,BRight]),
        unionv(XLeft,YLeft,BLeft).
backdis(Y, [BLeft,YIn,BRight]) :-
        YIn=[BLeft|_],
        backward(Y, YIn, BRight).

/*********************************************************************/

% A new & possibly correct temporary allocation routine:

% Uses the variable list created by varlist
% and the lifetime list created by lifetime.
% Takes the overlap of registers caused by calls into account.
% The Life list does not have to contain any instantiated entries.

% Optimization - 11/16/84:
% Modified tempa so that it will identify temporaries which are not
% arguments in the head and aren't
% arguments of a call, by allocating them outside of the registers being
% currently used for arguments, thereby leaving them available for other
% allocation.  This allows a more efficient allocation and solves the
% 'determinate concat' optimization.

% Optimization - 12/4/84:
% Modified tempa so that if a variable first occurs between the head and the
% first clause, it will attempt to allocate into the next call's argument
% registers.  Modification done at statement (1) below.
% Similar modification can probably be done for calls after first call.

% bug fix - 3/27/86:
% cut inserted in alloc procedure so that retract will succeed only once.
% This was not a problem in 1.2, where retract only succeeded once,
% no matter how many unifiable clauses were available.
% Peter's algorithm took advantage of this bug.
% This bug doesn't exist in 1.5.
% In order to simulate the bug, the cut has been inserted.

% bug fix - 1/15/87:
% alloc/3 fixed.  Old version would generate a choice point for each recursive
% call, whereas correct version generates only one choice point per allocation.
% This bug sometimes led to an enormous increase in allocation time.

tempalloc([arity(HeadArity)|Vars], [_|Life]) :-
        abolish(cause, 1),
        assert(cause(none)),
        tempa(Vars, Life, 1, HeadArity, [], head), !.


% Fail if there is a conflict:
tempa(Vars, [Live|R], N, Arity, OK, Place) :-
        conflict_interval(Place, N, Arity, Interval),
        conflict(Live, Interval, I),
        notin(I,OK),
        abolish(cause, 1),
        assert(cause(I)),
        !, fail.
```

```
tempa([], _, _, _, _, _) :- !.

% Try to allocate to an argument:
tempa([X|Vars], [Left,Right|LifeList], N, Arity, OK, Place) :-
        var(X), in(X, Right), !,
        alloc_start_reg(Place, X, Vars, N, Arity, StartReg),
        alloc(X, Right, StartReg),
        update_params(X, N, Arity, OK, NewN, NewArity, NewOK),
        tempa(Vars, [Right|LifeList], NewN, NewArity, NewOK, Place).
                        % failure of tempa backtracks to alloc
                        % which redoes the allocation causing the conflict.
tempa([X|Vars], [_|LifeList], _, _, _, Place) :-
        nonvar(X), X=arity(Arity), !,
        tempa(Vars, LifeList, 1, Arity, [], body).
tempa([X|Vars], [Left,In,Right|LifeList], _, _, _, Place) :-
        nonvar(X), X=(_;_), In=(_;_), !,
        distempa(X, In),
        tempa(Vars, [Right|LifeList], 1, 0, [], body).
tempa([X|Vars], [_|LifeList], N, Arity, OK, Place) :-
        update_params(X, N, Arity, OK, NewN, NewArity, NewOK),
        tempa(Vars, LifeList, NewN, NewArity, NewOK, Place).

% Handle disjunctions:
distempa((A;B), (ALife;BLife)) :- !, % cut needed for correct conflict detect
        tempa(A, ALife, 1, 0, [], body),
        distempa(B, BLife).
distempa(B, BLife) :-
        tempa(B, BLife, 1, 0, [], body).

% Calculate conflict interval.
% Depends on place in a call sequence
conflict_interval(body, 1, _, empty) :- !.
conflict_interval(body, N, _, int(1,N1)) :- !, N1 is N-1 .
conflict_interval(head, N, Arity, empty) :- N>Arity, !.
conflict_interval(head, N, Arity, int(N,Arity)) :- !.

% Update parameters of tempa.
update_params(X, N, Arity, OK, NewN, NewArity, NewOK) :-
        N=<Arity, !,
        NewN is N+1,
        NewArity=Arity,
        newok(X, N, OK, NewOK).
update_params(_, _, _, _, 1, 0, []) :- !.

        % New value of OK list
        newok(X, N, OK, [N|OK]) :-
                nonvar(X), X=x(N), !.
        newok(X, N, OK, OK) :- !.

% Calculate register to start allocation with.

% If in head, avoid using arg. reg. of next call
alloc_start_reg(head, X, Vars, N, Arity, StartReg) :-
        N>Arity, notinnextcall(X, Vars, NextArity), !,
        StartReg is NextArity+1 .
alloc_start_reg(head, _, _,    N, Arity, StartReg) :-
        N=<Arity, !,
        StartReg = N.
% Default starting value is register 1
alloc_start_reg(head, _, _,    _,     _,         1) :- !.
alloc_start_reg(body, _, _,    N,     _,         N) :- !.

% Succeeds iff there is a register conflict:
% The interval [L, L+1, ..., H] is also considered as live registers.
% It is represented as int(L,H) or as the atom 'empty'.
conflict(Live, int(L,H), I) :-
        L=<H,
        range(L, I, H),
        in(x(I), Live).
```

```
conflict(Live, R, I) :-
        conflict(Live, I).

conflict([V|Live], I) :-
        nonvar(V), V=x(I),
        in(V, Live).
conflict([R|Live], I) :-
        conflict(Live, I).

% Allocate a register.
% When there is a conflict,
% supports sophisticated backtracking to the cause.
% Don't allocate X8.

% Bug fix Jan. 15: every recursive call generated a choice point,
% whereas only one choice point per allocation mey be generated.
alloc(X, Alive, N) :-
        (prolog_version(sbprolog) -> not(N=8); \+(N=8)),
        notin(x(N),Alive), X=x(N).
alloc(X, Alive, N) :-
        cause(none), !, % <- Bug fix: this cut is essential.
        N1 is N+1,
        alloc(X, Alive, N1).
alloc(X, Alive, N) :-
        cause(N), abolish(cause, 1), assert(cause(none)),
        N1 is N+1,
        alloc(X, Alive, N1).

% Find next call and return arity.
% Fails if no next call or if X is not an argument of it.
notinnextcall(X,Vars,NextArity) :-
        isnextcall(Vars,Call,NextArity),!,
        notin(X, Call).

isnextcall([V|RestVars],RestVars,NextArity) :-
        nonvar(V),
        V = arity(NextArity),!.
isnextcall([_|RestVars],Call,NextArity) :-
        isnextcall(RestVars,Call,NextArity).


/***********************************************************************/

% Fix code containing illegal (excess) temporary variables,
% those temporaries numbered X9 or higher.

% Excess phase contains three passes:
%       1.  Backwards pass to reallocate permanents and excess temporaries
%           as permanents.  As in permalloc, variables whose last use
%           is later in the program get lower numbered locations.
%       2.  Forward pass to fix up all get and put instructions whose
%           second operand is now a permanent.
%       3.  Forward pass to change the 'unsafe_value' annotation to a
%           'value' annotation for all temporaries.  This finishes the
%           work started by valvar.

excess(Objcode,Objcode4) :-
   excess(Objcode,Objcode2,_,_),
   cleanup(Objcode2,Objcode3),
   temp_value(Objcode3,Objcode4).

% Pass 1.

excess([I|Rest],[NI|NR],NewPerm,NewMap) :-
   excess(Rest,NR,NextPerm,Map),
   fix_excess(I,NI,NextPerm,NewPerm,Map,NewMap).

excess([],[],1,[]).
```

```
fix_excess(get(Ann,A,B),get(Ann,NA,NB),NextPerm,NewPerm,Map,NewMap) :-
   fix_temp(A,NA,NextPerm,NextPerm2,Map,NextMap),
   fix_temp(B,NB,NextPerm2,NewPerm,NextMap,NewMap).

fix_excess(put(Ann,A,B),put(Ann,NA,NB),NextPerm,NewPerm,Map,NewMap) :-
   fix_temp(A,NA,NextPerm,NextPerm2,Map,NextMap),
   fix_temp(B,NB,NextPerm2,NewPerm,NextMap,NewMap).

fix_excess(unify(Ann,A),unify(Ann,NA),NextPerm,NewPerm,Map,NewMap) :-
   fix_temp(A,NA,NextPerm,NewPerm,Map,NewMap).

fix_excess(I,I,NextPerm,NextPerm,Map,Map).

% allocate a new permanent in place of old permanent or excess temporary.

fix_temp(A,NA,NextPerm,NewPerm,Map,NewMap) :-
   nonvar(A), A=x(I), I>8, !,
   add_perm(A,NA,NextPerm,NewPerm,Map,NewMap).

fix_temp(A,NA,NextPerm,NewPerm,Map,NewMap) :-
   nonvar(A), A=y(_), !,
   add_perm(A,NA,NextPerm,NewPerm,Map,NewMap).

fix_temp(A,A,NextPerm,NextPerm,Map,Map).

        add_perm(A,NA,NextPerm,NewPerm,Map,NewMap) :-
                inmap(A,Map,NA), !,
                NewMap = Map, NewPerm = NextPerm.
        add_perm(A,NA,NextPerm,NewPerm,Map,NewMap) :-
                NA = y(NextPerm),
                NewPerm is NextPerm+1,
                NewMap = [pair(A,NA)|Map].

% check whether variable has been reallocated yet,
% and if so, what it has been reallocated to.

inmap(A,[pair(A,NA)|_],NA) :- !.
inmap(A,[_|Rest],NA) :- inmap(A,Rest,NA), !.

% Pass 2.

cleanup([put(Ann,A,B),get(structure,S,C)|Rest],
        [put(Ann,A,x(8)),get(structure,S,x(8))|NRest]) :-
   nonvar(A), nonvar(B), nonvar(C), A = y(_), A = B, B = C, !,
   cleanup(Rest,NRest).

cleanup([put(Ann,A,B)|Rest],
        [put(value,B,x(8)),put(structure,A,x(8))|NRest]) :-
   nonvar(Ann), Ann = structure, nonvar(B), B = y(_), !,
   cleanup(Rest,NRest).

cleanup([put(Ann,A,B)|Rest],
        [put(Ann,A,x(8)),get(variable,B,x(8))|NRest]) :-
   nonvar(B), B = y(_), !,
   cleanup(Rest,NRest).

cleanup([get(Ann,A,B)|Rest],
        [put(value,B,x(8)),get(Ann,A,x(8))|NRest]) :-
   nonvar(B), B = y(_), !,
   cleanup(Rest,NRest).

cleanup([I|Rest],[I|NRest]) :- cleanup(Rest,NRest).

cleanup([],[]).
```

```
% Pass 3.

temp_value([I|Rest], [NI|NRest]) :-
        I=..[N,unsafe_value,X|RI],
        nonvar(X), X=x(_), !,
        NI=..[N,value,X|RI],
        temp_value(Rest, NRest).
temp_value([I|Rest], [I|NRest]) :-
        temp_value(Rest, NRest).
temp_value([], []).

/*********************************************************************/

% Calculate environment sizes in all call instructions:
% Returns maximum environment size.

envsize([], 0) :- !.
envsize([call(_,EnvSize)|Code], EnvSize) :-
        envsize(Code, EnvSize), !.
envsize([I|Code], EnvSize) :-
        type_arg(I, T, R),
        nonvar(R), R=y(N1),
        envsize(Code, N2),
        max(N1, N2, EnvSize), !.
envsize([_|Code], EnvSize) :-
        envsize(Code, EnvSize).

/*********************************************************************/

% Take care of void variables:
%       (1) Remove gets
%       (2) Instantiate unallocated variables
%       (3) Collect unifys

% Bug fix Jan. 15, 1987: old version left some unallocated
% voids uninstantiated.  Two fixes were considered:
% source code transformation & simple use of x(8).
% For simplicity the latter was done here.

% Remove superfluous gets of voids:
voidalloc([get(_,A,_)|Code], VCode) :-
        var(A), !,
        voidalloc(Code, VCode).

% Collect unifies of voids and replace by unify_void N:
voidalloc(Code, [unify(void,N)|VCode]) :-
        collect_voids(Code, Rest, N), N>0, !,
        voidalloc(Rest, VCode).

% Instantiate puts of voids to registers:
voidalloc([X|Code], [X|VCode]) :-
        inst_void(X), !,
        voidalloc(Code, VCode).

% Default clause:
voidalloc([I|Code], [I|VCode]) :- !,
        voidalloc(Code, VCode).
voidalloc([], []).


        collect_voids([unify(_,Arg)|Code], Rest, N) :-
                var(Arg), !,
                collect_voids(Code, Rest, N1),
                N is N1+1 .
        collect_voids(Code, Rest, 0) :- Rest=Code.

% Bug fix Jan. 15: added this predicate.
% Instantiate variables left unallocated to x(8):
```

```
inst_void(unify(cdr,x(8))) :- !.
inst_void(put(variable,x(8),x(8))) :- !.
inst_void(put(variable,R,R)) :- !.
inst_void(get(structure,_,x(8))) :- !.
inst_void(put(structure,_,x(8))) :- !.

/**********************************************************************/

% eliminate redundant assignments
% We can remove a put_value Yj,Xi when:
%   1) it's before the first call, and
%   2) Yj was initialized by a get_variable, and
%   3) between the get_variable and the put_value, there's no get, put, or
%       unify instruction which references Xi.  (This is probably overkill,
%       but it is correct.)
% The purpose of this optimization is to fix code for clauses like:
%       a(X) :- b(X), x(X).
% which generates:
%       get_variable Y1,X1
%       put_value Y1,X1         < redundant instruction >
%       call b/1
%               -- Wayne (1/28)

assn_elim(Code, ACode) :-
        assn_elim(Code, ACode,live(no,no,no,no,no,no,no)).

assn_elim([I|Rest], [I|Rest],_) :-
        I = call(_,_), !.
assn_elim([], [], _) :- !.
assn_elim([get(variable,Y,R)|Rest],
          [get(variable,Y,R)|NewRest],
          Live) :-
        nonvar(Y),
        Y=y(J),
        R=x(I), I\==8, !,
        make_live(Live,I,J,NewLive),
        assn_elim(Rest,NewRest,NewLive).
assn_elim([put(value,Y,R)|Rest],
          NewRest,
          Live) :-
        nonvar(Y), Y=y(J), R=x(I), is_live(Live,I,J), !,
        assn_elim(Rest,NewRest,Live).
assn_elim([I|Rest], [I|NewRest], Live) :-
        (I=put(A,X,Y); I=get(A,X,Y); I=unify(A,X)),
        (nonvar(X), X=x(K) -> make_dead(Live,K,NewLive) ; NewLive = Live),
        (nonvar(Y), Y=x(J) -> make_dead(NewLive,J,NewLive2);
        NewLive2=NewLive),!,
        assn_elim(Rest,NewRest,NewLive2).
assn_elim([I|Rest], [I|NewRest], Live) :-
        assn_elim(Rest,NewRest,Live).

% Live structure has exactly seven elements.
make_live(live(A1,A2,A3,A4,A5,A6,A7),1,J,
          live(J,A2,A3,A4,A5,A6,A7)).

make_live(live(A1,A2,A3,A4,A5,A6,A7),2,J,
          live(A1,J,A3,A4,A5,A6,A7)).

make_live(live(A1,A2,A3,A4,A5,A6,A7),3,J,
          live(A1,A2,J,A4,A5,A6,A7)).

make_live(live(A1,A2,A3,A4,A5,A6,A7),4,J,
          live(A1,A2,A3,J,A5,A6,A7)).

make_live(live(A1,A2,A3,A4,A5,A6,A7),5,J,
          live(A1,A2,A3,A4,J,A6,A7)).

make_live(live(A1,A2,A3,A4,A5,A6,A7),6,J,
          live(A1,A2,A3,A4,A5,J,A7)).
```

```
make_live(live(A1,A2,A3,A4,A5,A6,A7),7,J,
          live(A1,A2,A3,A4,A5,A6,J)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),1,
          live(no,A2,A3,A4,A5,A6,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),2,
          live(A1,no,A3,A4,A5,A6,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),3,
          live(A1,A2,no,A4,A5,A6,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),4,
          live(A1,A2,A3,no,A5,A6,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),5,
          live(A1,A2,A3,A4,no,A6,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),6,
          live(A1,A2,A3,A4,A5,no,A7)).

make_dead(live(A1,A2,A3,A4,A5,A6,A7),7,
          live(A1,A2,A3,A4,A5,A6,no)).

is_live(Live,I,J) :- arg(I,Live,J).

/**********************************************************************/

% Do peephole optimization of several kinds:
%            (1) many special instruction sequences.
%            (2) code generation for some built-ins.
%            (3) allocate & deallocate instructions.
%            (4) last instruction (proceed or execute).
%            (5) customization of instructions.

peephole(Code, PCode, Link, MaxSize) :-
        peephole(Code, PCode, Link, no_alloc, MaxSize, no_dummy), !.

% The call/1 predicate must be an escape:
peephole([call(call/1,_)|Code], PCode, Link, Alloc, M, D) :- !,
        peephole([call/1|Code], PCode, Link, Alloc, M, D).

% Insert the allocate and deallocate instructions
% and take care of the last instruction.
peephole([call(G,0)], LastCode, Link, Alloc, M, D) :- !,
        lastcode(Alloc, LastCode, [execute(G)|Link]).
peephole([], LastCode, Link, Alloc, M, D) :- !,
        lastcode(Alloc, LastCode, [proceed|Link]).

% Insert the correct allocate instruction:
peephole([I|Code], [A|PCode], Link, no_alloc, M, D) :-
        alloc_needed(I), !,
        alloc_instruction(A, M),
        peephole([I|Code], PCode, Link, yes_alloc, M, D).

% Insert call to dummy procedure if using old allocate instruction:
% Must be done if 'try' or call/1 occurs as first call.
% This is needed to initialize the N register.
peephole([I|Code], [call(allocate_dummy/0,M),I|PCode], Link, yes_alloc, M, D)
        :-
        D=no_dummy,
        compile_options(a),
        (I=..[try|_]; I=call/1), !,
        peephole(Code, PCode, Link, yes_alloc, M, yes_dummy).

% Recognize and eliminate superfluous jumps:
peephole([label(Lbl),execute(Lbl)|Code],
         [execute(Lbl)|PCode], Link, Alloc, M, D) :- !,
        peephole(Code, PCode, Link, Alloc, M, D).
```

```
% Remove all code after a fail/0 until reaching a
% label, retry, or trust:
% (calls to peephole and f_remove must be in this order for best working!)
peephole([fail/0|Code], [fail/0|PCode], Link, Alloc, M, D) :- !,
        peephole(Code, MCode, Link, Alloc, M, D),
        f_remove(MCode, PCode).

% Optimize unify goals:
% First case: one variable is temporary or void:
peephole([put(variable,R,R),get(A,X,R)|Code], PCode, Link, Alloc, M, D) :-
        R=x(I),
        integer(I), !,
        peephole([put(A,X,R)|Code], PCode, Link, Alloc, M, D).
% Second case: both variables are permanent:
% What if X==Y???
peephole([put(A,X,x(8)),get(B,Y,x(8))|Code], PCode, Link, Alloc, M, D) :-
        X\==Y, X=y(N1), Y=y(N2), !,
        update_unsafe(A, B, NewA, NewB),
        PCode=[put(NewA,X,x(8)),get(NewB,Y,x(8))|MCode],
        peephole(Code, MCode, Link, Alloc, M, D).

% Optimize unify_cdr:
peephole([unify(cdr,x(8)),get(variable,X,x(8))|Code], PCode, Link, Alloc, M,
         D) :- !,
        peephole([unify(cdr,X)|Code], PCode, Link, Alloc, M, D).
peephole([unify(cdr,x(8)),get(unsafe_value,X,x(8))|Code],
         [unify(cdr,x(8)),get(value,X,x(8))|PCode], Link, Alloc, M, D) :- !,
        peephole(Code, PCode, Link, Alloc, M, D).

% Remove superfluous initializations of permanent variables:
peephole([put(value,y(_),x(8)),I|Code], PCode, Link, Alloc, M, D) :-
        I=..[Name|_], Name\==get, !,
        peephole([I|Code], PCode, Link, Alloc, M, D).

% Remove no-op register transfers:
peephole([I|Code], PCode, Link, Alloc, M, D) :-
        (I=get(variable,R,R); I=put(value,R,R)),
        R=x(_), !,
        peephole(Code, PCode, Link, Alloc, M, D).

% Remove remaining unsafe_values
peephole([get(unsafe_value,A,B)|Code], [get(value,A,B)|PCode], Link, Alloc, M,
         D) :- !,
        peephole(Code, PCode, Link, Alloc, M, D).

% Post-transformation:
% Generates code for some built-ins in terms of
% existing instructions.
peephole([Name/Arity|Code], PCode, Link, Alloc, M, D) :-
        post_trans(Name, Arity, TCode-Code), !,
        peephole(TCode, PCode, Link, Alloc, M, D).

% Customization of instructions:
peephole([I|Code], [CI|PCode], Link, Alloc, M, D) :-
        customize(I, CI), !,
        peephole(Code, PCode, Link, Alloc, M, D).

% Default:
peephole([I|Code], [I|PCode], Link, Alloc, M, D) :-
        peephole(Code, PCode, Link, Alloc, M, D).

% Update unsafe_value annotations of put-get sequence:
update_unsafe(A, unsafe_value, A, value) :- !.
update_unsafe(unsafe_value, B, value, B) :- !.
update_unsafe(A, B, A, B) :- !.
```

```
% Remove code until encountering a
% label, retry, or trust:
f_remove(V, V) :- var(V).
f_remove([Instr|Code], [Instr|Code]) :-
        Instr=..[N|_],
        (N=label; N=retry; N=trust), !.
f_remove([_|Code], RCode) :-
        f_remove(Code, RCode).

% Table of builtins with code:
post_trans(var, 1, [switch_on_term(fail,fail,fail)|L]-L).
post_trans(nonvar, 1, [switch_on_term(Lbl,Lbl,Lbl),fail/0,label(Lbl)|L]-L).
post_trans(atomic, 1, [switch_on_term(Lbl,fail,fail),fail/0,label(Lbl)|L]-L).
post_trans(nonatomic, 1, [switch_on_term(fail,Lbl,Lbl),label(Lbl)|L]-L).
post_trans(list, 1, [switch_on_term(fail,Lbl,fail),fail/0,label(Lbl)|L]-L).
post_trans(nonlist, 1, [switch_on_term(Lbl,fail,Lbl),label(Lbl)|L]-L).
post_trans(structure, 1, [switch_on_term(fail,fail,Lbl),fail/0,label(Lbl)|L]-L).
post_trans(composite, 1, [switch_on_term(fail,Lbl,Lbl),fail/0,label(Lbl)|L]-L).
post_trans(simple, 1, [switch_on_term(Lbl,fail,fail),label(Lbl)|L]-L).
post_trans(repeat, 0, [try(Lbl),label(Lbl)|L]-L).

% Customize one instruction:
customize(get(structure,'.'/2,B), get_list(B)).
customize(put(structure,'.'/2,B), put_list(B)).
customize(put(constant,[],A), put_nil(A)).
customize(get(constant,[],A), get_nil(A)).

% Succeeds if an allocate instruction is needed
% before instruction I:
alloc_needed(I) :-
        I=..[Name|_],
        (Name=call;Name=try;Name=cut).
alloc_needed(I) :-
        (I=get(_,V,_);I=put(_,V,_);I=unify(_,V)),
        nonvar(V), V=y(_).

% Deallocate at last code:
lastcode(yes_alloc, [deallocate|L], L).
lastcode(no_alloc,  L, L).

% The allocate instruction:
alloc_instruction(allocate, M) :- compile_options(a), !.
alloc_instruction(allocate(M), M).

/*********************************************************************/

% Help information.
% Invoked by the command plm_help or plm_help(option).

plm_help :-
        nl,
        write('The compiler is called as '),
        write('plm(filename) or plm(filename,optionlist).'), nl,
        write('The options in optionlist must be a subset of '),
        help_optionlist(OptList),
        write(OptList), write('.'), nl,
        write('Call plm_help(option) for more information on an option.'), nl.


plm_help(Option) :-
        nl,
        nonvar(Option),
        help_info(Option, String),
        put(9), write(String), nl,
        fail.
```

```
plm_help(Option) :-
        help_optionlist(OptList),
        ((prolog_version(sbprolog) -> not(help_member(Option,OptList));
                                      \+(help_member(Option,OptList)));
         var(Option)),
        put(9),write('The option '''),write(Option),write(''' is unknown.'),nl,
        put(9),write('The known options are in the set '),
        write(OptList), write('.'),nl.
plm_help(_).

help_optionlist([a,l,s,u,q,a(_)]).

help_info(a, 'Compile an allocate instruction without arguments.').
help_info(a, 'The default is to use a single-argument allocate with the').
help_info(a, 'environment size as argument.').
help_info(l, 'Write the output in Prolog-readable list form.').
help_info(l, 'The default is to write the output in human-readable form.').
help_info(u, 'Do not expand calls of is/2 into calls of is/4.').
help_info(u,
    'The default is to expand is/2 into is/4 whenever it is possible.').
help_info(u, 'Option u overrides option s.').
help_info(q, 'When output is in human-readable form, quote all atoms.').
help_info(q, 'The default is to quote only those atoms that need it.').
help_info(q,
    'Option q has no effect when option l (Prolog-readable form) is used.').
help_info(a(X),
    'The parameter of a(_) (which must be atomic) is appended to all').
help_info(a(X),
    'labels in the human-readable code.  The default is to append nothing.').
help_info(s,
    'Compile the operators +, -, \/, /\ in an expression as builtins,').
help_info(s, 'and only the others with is/4.  The default is to compile all').
help_info(s, 'operators with is/4.').
help_info(s,
    'Option s has no effect when option u (unexpanded expression) is used.').

help_member(X, [X|_]).
help_member(X, [_|L]) :- help_member(X, L).
```

test

```
concat([],L,L).
concat([X|L1],L2,[X|L3]):-concat(L1,L2,L3).

a(1).
a(2).
a(3).
a(4).

b([_]).
b([a]).
b(X).

a(A,B,C,D,E,F) :- a([A|B], [C|D], F,D,E,C).
a(A,B,C,s,E,F) :- a(F,E, [A|C], s(E,F), A,B).
```

.plm_compiler.bench

```
# /*
  set-up.plm_compiler: bench set-up for plm_compiler
  */
plm_compiler :- driver(plm_compiler).

benchmark(plm_compiler, cap(CI), dummy(CI), 10) :-
#if BIM_PROLOG
        bim,
#elseif C_PROLOG
        c,
#elseif QUINTUS_PROLOG
        quintus,
#elseif SB_PROLOG
        sb,
#elseif SICTUS_PROLOG
        sicstus,
#endif
        options(test, []),
        see(test), read_clauses(CI), seen.

#message "NOTE: show/1 is NOT defined for plm_compiler"


#include "driver"
```

tp

```
# /*
  boys.m: benchmark (tp) boys master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                           %%%%
%%%%    (tp) boys                                              %%%%
%%%%                                                           %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                  %%%%
%%%%                                                           %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".boys.bench"
#else
boys :- do('examples/boys.ax','examples/boys.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"         /* code for propositional theorem prover */
#endif
```

```
# /*
  ct_2.m: benchmark (tp) ct_2 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                          %%%%
%%%%    (tp) ct_2                                             %%%%
%%%%                                                          %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                 %%%%
%%%%                                                          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".ct_2.bench"
#else
ct_2 :- do('examples/empty','examples/ct_2.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"         /* code for propositional theorem prover */
#endif
```

```
# /*
  ct_3.m: benchmark (tp) ct_3 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                          %%%%
%%%%    (tp) ct_3                                             %%%%
%%%%                                                          %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                 %%%%
%%%%                                                          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".ct_3.bench"
#else
ct_3 :- do('examples/empty','examples/ct_3.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"          /* code for propositional theorem prover */
#endif
```

```
# /*
  ct_4.m: benchmark (tp) ct_4 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                            %%%%
%%%%    (tp) ct_4                                               %%%%
%%%%                                                            %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                   %%%%
%%%%                                                            %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".ct_4.bench"
#else
ct_4 :- do('examples/empty','examples/ct_4.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"         /* code for propositional theorem prover */
#endif
```

```
# /*
  ct_5.m: benchmark (tp) ct_5 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                          %%%%
%%%%    (tp) ct_5                                             %%%%
%%%%                                                          %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                 %%%%
%%%%                                                          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".ct_5.bench"
#else
ct_5 :- do('examples/empty','examples/ct_5.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"         /* code for propositional theorem prover */
#endif
```

```
# /*
  ct_6.m: benchmark (tp) ct_6 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                           %%%%
%%%%    (tp) ct_6                                              %%%%
%%%%                                                           %%%%
%%%%    Ross Overbeek (overbeek@anl-mcs.arpa)                  %%%%
%%%%                                                           %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#if BENCH
#  include ".ct_6.bench"
#else
ct_6 :- do('examples/empty','examples/ct_6.sos'), !.
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW

show.
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (do/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
do(_,_).
#else
#  include "tp"        /* code for propositional theorem prover */
#endif
```

```
# /*
  tp: code for propositional theorem prover (Prolog version)
  */
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                          %%%%
%%%%    Uniprocessor Version of Propositional Theorem Prover  %%%%
%%%%                                                          %%%%
%%%%    This version corresponds to the version in C. It      %%%%
%%%%    accepts input in the same format as the C version.    %%%%
%%%%                                                          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*****************************************************************

To understand what this benchmark is designed to stress, one needs
to know a little about how theorem provers run.  Essentially, this
program reads in two sets of formulas:

        The first set is called the "axioms".  These formulas are
        placed into the "usable formulas" list.

        The second set of formulas is the initial contents of the
        "set of support" list.

For convenience, the first set is processed by adding each formula
to the "set of support" and then moving each formula to the "usable
formulas" list.

I will try to use the term "formula" when I'm talking about a
clause in the propositional calculus (to distinguish these from
Prolog clauses).  All of the formulas input to the program or
derived by the program are clauses in the propositional calculus.

The program builds an initial database of the form

        database(Sos,NextId,IndexAll,IndexUsable,IdLookup)

Here,
        Sos is a structure of the form

                sos(<list of clauses that contain 1 literal>,
                    <list of clauses that contain 2 literals>,
                    <list of clauses that contain 3 literals>,
                    .
                    .
                    .
                    <list of clauses that contain 28 literals>)

        It is, as Richard O'Keefe pointed out, a priority queue.
        Entries are extracted when a new "given formula" is needed by
        selecting the first clause with the least number of literals.

        NextId is an integer which gives the value that can be
        assigned as an id to the next formula added to the database

        IndexAll is an "index" that is used to access all clauses that
        occur in either the set of support or the usable formulas list.

        IndexUsable is an "index" that is used to access all clauses that
        occur in the usable formulas list.

        IdLookup is an "array" used to locate a formula with a designated
        id.

        An "index" is a structure of the form

                index(PosIndex,NegIndex)
        where
                PosIndex is of the form

                        pos(<list of clauses containing v0>,
                            <list of clauses containing v1>,
                            <list of clauses containing v2>,
                                     .
                                     .
                                     .
                            <list of clauses containing v27>)

                NegIndex is of the form

                        neg(<list of clauses containing -v0>,
                            <list of clauses containing -v1>,
                            <list of clauses containing -v2>,
                                     .
                                     .
                                     .
                            <list of clauses containing -v27>)

This "database" is really a mechanism for efficiently accessing
the formulas composing the "set of support list" and the "usable
formulas list", which are just abstractions.  There is no actual
list called set-of-support (rather, Sos is a structure through
which these formulas are accessed) or usable-formulas (rather,
these formulas are accessed through the IndexUsable structure).

Execution of the program causes the input formulas to be used to
make an initial database.  Then, until the null clause is derived
or the set-of-support becomes empty, the following procedure is
just repeated:

        pick a clause from the set-of-support (with a minimum
            number of literals)

        move it to the usable-formulas list

        form all binary resolvents that can be formed from the given
            clause and another member of the usable-formulas list

        for each generated resolvent,

            if it is subsumed by an existing clause,
                or if it is a tautology,

                just ignore it

            else

                delete all clauses that already exist,
                    but that are subsumed by the generated clause

The deletion is a bit tricky.  Actually, we just accumulate a list
of the ids of clauses subsumed by new resolvents produced by a
single given clause; once generation of resolvents has completed
for the given formula, then all of the clauses to be deleted are
deleted.

The problem is that each addition to the formula database and each
deletion of the set of clauses "back-subsumed" by clauses derived
from a given clause produces a new database (built from contents of
the previous database).  A Prolog that is not smart enough to use
destructive assignment (and none are, at this time) accumulates a
massive number of structures on the heap.  The time spent to build
these structures and the garbage collection caused by this copying
constitute a real performance problem.

*****************************************************************/
```

```
#option "
        > For use with Quintus Prolog, tp requires a
        > Quintus Prolog-specific directive.  It is
        > generated if option QUINTUS_PL is selected."
#if QUINTUS_PL
:- unknown(_,fail).

#endif
# /*
tp :-   read_file_name(axioms,AxiomFileName),
        read_file_name(sos,SosFileName),
        do(AxiomFileName,SosFileName).

  */
do(AxiomFileName,SosFileName) :-
        initialize_database(Db),
        process_axiom_file(AxiomFileName,state(Db,false),State1,Symbols),
        process_sos_file(SosFileName,State1,State2,Symbols),
        process_events(State2,State3),
        ( show -> display_final_status(State3) ; true ).

display_final_status(state(_,true)) :- write('Proof found'), nl.
display_final_status(state(_,false)) :- write('Proof not found'), nl.

process_events(state(Db,true),state(Db,true)).
process_events(state(Db,false),state(Db,false)) :-
        sos(Db,Sos),
        empty_sos(Sos).
process_events(state(Db,false),NewState) :-
        sos(Db,Sos),
        pick_given_formula(Sos,Given),  % pick from set-of-support
        ( show -> writelist([given,Given]), nl ; true ),
        move_to_usable(Db,Given,Db1),
        gen_and_chk(Db1,Given,NewDb,NewStatus),
        process_events(state(NewDb,NewStatus),NewState).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Resolvent generation code                               %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen_and_chk(Db,Id,NewDb,Status) :-
        id_lookup(Db,IdLookUp),
        array(IdLookUp,Id,clause(Id,_,Pos,Neg)),
        get_literals(Pos,PosLits),
        get_literals(Neg,NegLits),
        index_usable(Db,index(PosIndex,NegIndex)),
        gen_resolvents(PosLits,Pos,Neg,Id,Db,NegIndex,pos,Db1,Status1),
        cont_generation(Status1,Pos,Neg,Id,Db1,PosIndex,neg,
                        NewDb,Status,NegLits).

cont_generation(true,_,_,_,Db,_,_,Db,true,_).
cont_generation(false,Pos,Neg,Id,Db,Index,Type,NewDb,Status,Lits) :-
        gen_resolvents(Lits,Pos,Neg,Id,Db,Index,Type,NewDb,Status).

gen_resolvents([],_,_,_,Db,_,_,Db,false).
gen_resolvents([ClashLit|T],Pos,Neg,Id,Db,Index,Type,NewDb,Status) :-
        arg(ClashLit,Index,ClauseList),
        gen_resolvents_on_lit(ClauseList,Pos,Neg,Id,
                              Db,ClashLit,Type,Db1,Status1,DelList),
        delete_from_database_clauses(DelList,Db1,Db2),
        cont_generation(Status1,Pos,Neg,Id,Db2,Index,Type,
                        NewDb,Status,T).

gen_resolvents_on_lit([],_,_,_,Db,_,_,Db,false,[]).
gen_resolvents_on_lit([Clause|T],Pos,Neg,Id,Db,ClashLit,Type,NewDb,Status,
                      DelList) :-
        gen_one_resolvent(Clause,Pos,Neg,Id,Db,ClashLit,Type,Db1, Status1,
                          DelList,NewEnd),
        cont_generation_on_lit(Status1,Pos,Neg,Id,Db1,ClashLit,Type, NewDb,
                               Status,T,NewEnd).

gen_one_resolvent(clause(Par2Id,_,Par2Pos,Par2Neg),Pos,Neg,
                  Par1Id,Db,ClashLit,Type,NewDb,Status,DelList,NewEnd) :-
        form_resolvent(Type,Pos,Neg,Par2Pos,Par2Neg,ClashLit,ResPos,ResNeg),
        (   empty_clause(ResPos,ResNeg) ->
                Status = true,
                ( show ->
                    writelist([derived,empty,clause,from,[Par1Id,Par2Id]]), nl
                ; /* otherwise -> */
                    true
                )
        ;   /* otherwise -> */
                Status = false,
                subsume_and_add(Db,ResPos,ResNeg,[Par1Id,Par2Id],NewDb,DelList,
                                NewEnd)
        ).

subsume_and_add(Db,ResPos,ResNeg,Parents,NewDb,DelList,NewEnd) :-
        ( (forward_subsumed(Db,ResPos,ResNeg) ; tautology(ResPos,ResNeg) ) ->
                NewDb = Db,
                NewEnd = DelList
        ;  /* otherwise -> */
                add_to_sos(Db,clause(Parents,ResPos,ResNeg),NewDb,NewId),
                ( show ->
                    write_added_mesg(NewId,clause(Parents,ResPos,ResNeg))
                ; /* otherwise -> */
                    true
                ),
                back_subsumption(Db,NewId,ResPos,ResNeg,DelList,NewEnd)
        ).

tautology(Pos,Neg) :-
        0 =\= Pos /\ Neg.

cont_generation_on_lit(true,_,_,_,Db,_,_,Db,true,_,[]).
cont_generation_on_lit(false,Pos,Neg,Id,Db,ClashLit,Type,NewDb,Status,
                       ClauseList,DelList) :-
        gen_resolvents_on_lit(ClauseList,Pos,Neg,Id,Db,ClashLit,Type,
                              NewDb,Status,DelList).

empty_clause(0,0).

form_resolvent(pos,Pos1,Neg1,Pos2,Neg2,ClashLit,ResPos,ResNeg) :-
        form_resolvent(Pos1,Neg1,Pos2,Neg2,ClashLit,ResPos,ResNeg).
form_resolvent(neg,Pos2,Neg2,Pos1,Neg1,ClashLit,ResPos,ResNeg) :-
        form_resolvent(Pos1,Neg1,Pos2,Neg2,ClashLit,ResPos,ResNeg).

form_resolvent(Par1Pos,Par1Neg,Par2Pos,Par2Neg,
               ClashLit,ResPosWord,ResNegWord) :-
        Mask is \(1 << (ClashLit - 1)),
        ResPosWord is ((Par1Pos /\ Mask) \/ Par2Pos),
        ResNegWord is ((Par2Neg /\ Mask) \/ Par1Neg).

delete_from_database_clauses(List,Db,NewDb) :-
        delete_from_database_clauses(List,[],Db,NewDb).

delete_from_database_clauses([],_,Db,Db).
delete_from_database_clauses([Id|Rest],L,Db,NewDb) :-
        (   u_member(Id,L) ->
                delete_from_database_clauses(Rest,L,Db,NewDb)
        ;   /* otherwise -> */
                delete_from_database(Db,Id,Db1),
                delete_from_database_clauses(Rest,[Id|L],Db1,NewDb)
        ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                                %%%
%%%     Subsumption code                                           %%%
%%%                                                                %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

forward_subsumed(Db,Pos,Neg) :-
        index_all(Db,index(PosIndex,_)),
        get_next_literal(Pos,PosLit),
        indexed_subsumed_by(PosLit,PosIndex,Pos,Neg).
forward_subsumed(Db,Pos,Neg) :-
        index_all(Db,index(_,NegIndex)),
        get_next_literal(Neg,NegLit),
        indexed_subsumed_by(NegLit,NegIndex,Pos,Neg).

indexed_subsumed_by(Lit,Index,Pos,Neg) :-
        arg(Lit,Index,ClauseList),
        u_member(clause(_,_,SubsumerPos,SubsumerNeg),ClauseList),
        subsumes(SubsumerPos,SubsumerNeg,Pos,Neg).

subsumes(Pos1,Neg1,Pos2,Neg2) :-
        Pos1 =:= (Pos1 /\ Pos2),
        Neg1 =:= (Neg1 /\ Neg2).

back_subsumption(Db,SubId,Pos,Neg,DelList,NewEnd) :-
        index_all(Db,IndexAll),
        indexed_back_subsumed_by(SubId,IndexAll,Pos,Neg,DelList,NewEnd).

indexed_back_subsumed_by(SubId,IndexAll,Pos,Neg,DelList,NewEnd) :-
        (   Pos =\= 0 ->
                get_first_literal(Pos,H),
                IndexAll = index(PosIndex,_),
                arg(H,PosIndex,ClauseList)
        ;   Neg =\= 0 ->
                get_first_literal(Neg,H),
                IndexAll = index(_,NegIndex),
                arg(H,NegIndex,ClauseList)
        ),
        backsub(ClauseList,SubId,Pos,Neg,DelList,NewEnd).

backsub([],_,_,_,X,X).
backsub([clause(Id,_,Pos,Neg)|T],SubId,SubsumerPos,SubsumerNeg,DelList,
        NewEnd) :-
        (   subsumes(SubsumerPos,SubsumerNeg,Pos,Neg) ->
                ( show -> write_subsumed_mesg(Id,SubId) ; true ),
                add_element(DelList,Id,EndList)
        ;   /* otherwise -> */
                EndList = DelList
        ),
        backsub(T,SubId,SubsumerPos,SubsumerNeg,EndList,NewEnd).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                         %%%
%%%     Database utilities                                  %%%
%%%                                                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sos(database(Sos,_,_,_,_),Sos).
next_id(database(_,NextId,_,_,_),NextId).
index_all(database(_,_,IndexAll,_,_),IndexAll).
index_usable(database(_,_,_,IndexUsable,_),IndexUsable).
id_lookup(database(_,_,_,_,IdLookup),IdLookup).
pos_index(index(Pos,_),Pos).
neg_index(index(_,Neg),Neg).

empty_sos(sos_by_weight([],[],[],[],[],[],[],[],[],[],[],[],[],[],
                        [],[],[],[],[],[],[],[],[],[],[],[],[],[])).

create_index(index(pos([],[],[],[],[],[],[],[],[],[],[],[],[],[],
                       [],[],[],[],[],[],[],[],[],[],[],[],[],[]),
                   neg([],[],[],[],[],[],[],[],[],[],[],[],[],[],
                       [],[],[],[],[],[],[],[],[],[],[],[],[],[]))).

database_vars(database(Sos,NextId,IndexAll,IndexUsable,IdLookup),
              Sos,NextId,IndexAll,IndexUsable,IdLookup).

make_database(Sos,NextId,IndexAll,IndexUsable,IdLookup,
              database(Sos,NextId,IndexAll,IndexUsable,IdLookup)).

initialize_database(database(Sos,1,IndexAll,IndexUsable,IdLookUp)) :-
        create_index(IndexAll),
        create_index(IndexUsable),
        new_array(IdLookUp),
        sos_by_weight(Sos).

sos_by_weight(Sos) :-
        functor(Sos,sos_by_weight,28),
        empty_sos(Sos).

pick_given_formula(Sos,Id) :-
        pick_given_formula(Sos,1,Id).

pick_given_formula(Sos,ArgNum,Id) :-
        ArgNum < 29,
        arg(ArgNum,Sos,Arg),
        (   Arg = [Id|_] ->
                true
        ;   /* otherwise -> */
                NewArgNum is ArgNum + 1,
                pick_given_formula(Sos, NewArgNum, Id)
        ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Input processing code                                   %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# /*
read_file_name(Type,FileName) :-
        write('Enter filename for '), write(Type), write(': '),
        read(FileName).

  */
process_axiom_file(File,State,NewState,Symbols) :-
        see(File),
        process_axiom_input(State,NewState,[],Symbols),
        seen.

process_sos_file(File,State,NewState,Symbols) :-
        see(File),
        process_sos_input(State,NewState,Symbols,_),
        seen.

process_axiom_input(state(Db,Status),state(NewDb,Status),Symbols,NewSym) :-
        (   read_one_clause(Clause,Symbols,Symbols2) ->
                add_to_sos(Db,Clause,Db1,Id),
                ( show -> write_added_mesg(Id,Clause) ; true ),
                move_to_usable(Db1,Id,Db2),
                process_axiom_input(state(Db2,Status),state(NewDb,Status),
                                    Symbols2,NewSym)
        ;   /* otherwise -> */
                NewSym = Symbols,
                NewDb = Db
        ).

process_sos_input(state(Db,Status),state(NewDb,Status),Symbols,NewSym) :-
        (   read_one_clause(Clause,Symbols,Symbols2) ->
                add_to_sos(Db,Clause,Db1,Id),
                ( show -> write_added_mesg(Id,Clause) ; true ),
                process_sos_input(state(Db1,Status),state(NewDb,Status),
                                    Symbols2,NewSym)
        ;   /* otherwise -> */
                NewSym = Symbols,
                NewDb = Db
        ).

read_one_clause(clause([-1,-1],Pos,Neg),Symbols,NewSym) :-
        read(Term),
        Term \== end_of_file,
        term_to_list(Term,CList),
        set_bits(CList,0,0,Pos,Neg,Symbols,NewSym).

term_to_list((First;Rest),[First|T]) :-
        term_to_list(Rest,T).
term_to_list(Term,[Term]) :-
        \+ Term = (_;_).

set_bits([],Pos,Neg,Pos,Neg,Symbols,Symbols).
set_bits([H|T],InitPos,InitNeg,NewPos,NewNeg,Symbols,NewSym) :-
        lookup_symbol(H,SymbolNumber,Type,Symbols,Symbols1),
        (   Type = pos ->
                turn_bit_on(InitPos,SymbolNumber,PosWord),
                set_bits(T,PosWord,InitNeg,NewPos,NewNeg,Symbols1,NewSym)
        ;   /* otherwise -> */
                turn_bit_on(InitNeg,SymbolNumber,NegWord),
                set_bits(T,InitPos,NegWord,NewPos,NewNeg,Symbols1,NewSym)
        ).

turn_bit_on(Word,BitPos,NewWord) :-
        NewWord is (Word \/ (1 << (BitPos - 1))).
```

```
lookup_symbol(-Symbol,SymbolNumber,neg,Symbols,NewSym) :-
        (   u_member((Symbol,SymbolNumber),Symbols) ->
                NewSym = Symbols
        ;   /* otherwise -> */
                u_length(Symbols,I),
                SymbolNumber is I+1,
                NewSym = [(Symbol,SymbolNumber)|Symbols]
        ).
lookup_symbol(Symbol,SymbolNumber,pos,Symbols,NewSym) :-
        (   u_member((Symbol,SymbolNumber),Symbols) ->
                NewSym = Symbols
        ;   /* otherwise -> */
                u_length(Symbols,I),
                SymbolNumber is I+1,
                NewSym = [(Symbol,SymbolNumber)|Symbols]
        ).

get_literals(Word,Literals) :- get_literals(Word,Literals,1).

get_literals(Word,List,N) :-
        (   (Word =:= 0 ; N =:= 29) ->
                List = []
        ;   /* otherwise -> */
                J is (N + 1),
                Word1 is (Word >> 1),
                (   0 =:= (Word /\ 1) ->
                        get_literals(Word1,List,J)
                ;   /* otherwise -> */
                        List = [N|List1],
                        get_literals(Word1,List1,J)
                )
        ).

get_first_literal(Word,Literal) :-
        get_next_literal(Word,Literal), !.

get_next_literal(Word,Literal) :- get_next_literal(Word,Literal,1).

get_next_literal(Word,N,N) :-
        N < 29,
        1 is (Word /\ 1).
get_next_literal(Word,Literal,N) :-
        N < 28,
        I is N+1,
        Word1 is (Word >> 1),
        get_next_literal(Word1,Literal,I).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                                    %%%
%%%     Indexing routines                                              %%%
%%%                                                                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_to_sos(Db,clause(Parents,Pos,Neg),NewDb,NextId) :-
        database_vars(Db,Sos,NextId,AllIndex,UsableIndex,IdLookUp),
        NewNextId is NextId + 1,
        Clause = clause(NextId,Parents,Pos,Neg),
        add_index_entry(AllIndex,Clause,Pos,Neg,NewAllIndex),
        num_lits(Pos,Neg,NumLits),
        add_by_weight_to_sos(Sos,NumLits,NextId,NewSos),
        array(IdLookUp,NextId,Clause),
        make_database(NewSos,
                      NewNextId,NewAllIndex,UsableIndex,IdLookUp,NewDb).

move_to_usable(Db,Id,NewDb) :-
        database_vars(Db,Sos,NextId,AllIndex,UsableIndex,IdLookUp),
        delete_from_sos(Sos,Id,NewSos,Clause,IdLookUp),
        Clause = clause(Id,_,Pos,Neg),
        add_index_entry(UsableIndex,Clause,Pos,Neg,NewUsableIndex),
        make_database(NewSos,
                      NextId,AllIndex,NewUsableIndex,IdLookUp,NewDb).

delete_from_database(Db,Id,NewDb) :-
        database_vars(Db,Sos,NextId,AllIndex,UsableIndex,IdLookUp),
        delete_from_sos(Sos,Id,NewSos,Clause,IdLookUp),
        !,
        Clause = clause(Id,_,Pos,Neg),
        delete_index_entry(AllIndex,Id,Pos,Neg,NewAllIndex),
        make_database(NewSos,NextId,
                      NewAllIndex,UsableIndex,IdLookUp,NewDb).
delete_from_database(Db,Id,NewDb) :-
        database_vars(Db,Sos,NextId,AllIndex,UsableIndex,IdLookUp),
        array(IdLookUp,Id,Clause),
        Clause = clause(Id,_,Pos,Neg),
        delete_index_entry(AllIndex,Id,Pos,Neg,NewAllIndex),
        delete_index_entry(UsableIndex,Id,Pos,Neg,NewUsableIndex),
        make_database(Sos,NextId,
                      NewAllIndex,NewUsableIndex,IdLookUp,NewDb).

add_index_entry(Index,PtrClause,Pos,Neg,NewIndex) :-
        pos_index(Index,PosIndex),
        get_literals(Pos,PosLits),
        add_to_index_list(PosLits,PosIndex,PtrClause,NewPosIndex),
        neg_index(Index,NegIndex),
        get_literals(Neg,NegLits),
        add_to_index_list(NegLits,NegIndex,PtrClause,NewNegIndex),
        pos_index(NewIndex,NewPosIndex),
        neg_index(NewIndex,NewNegIndex).

add_to_index_list(Lits,Index,Clause,NewIndex) :-
        add_to_index_lists(Lits,Index,Clause,InitChangeList),
        functor(Index,Functor,Arity),
        functor(NewIndex,Functor,Arity),
        form_updated_index(InitChangeList,Arity,Index,NewIndex).

add_to_index_lists([LiteralNum|Tail],Index,Clause,InitChange) :-
        arg(LiteralNum,Index,Arg),
        add_element(InitChange,(LiteralNum,[Clause|Arg]),NewChangeList),
        add_to_index_lists(Tail,Index,Clause,NewChangeList).
add_to_index_lists([],_,_,[]).
```

```
delete_index_entry(Index,ClauseId,Pos,Neg,NewIndex) :-
        pos_index(Index,PosIndex),
        get_literals(Pos,PosLits),
        delete_from_index_list(PosLits,PosIndex,ClauseId,NewPosIndex),
        neg_index(Index,NegIndex),
        get_literals(Neg,NegLits),
        delete_from_index_list(NegLits,NegIndex,ClauseId,NewNegIndex),
        pos_index(NewIndex,NewPosIndex),
        neg_index(NewIndex,NewNegIndex).

delete_from_index_list(Lits,Index,Clause,NewIndex) :-
        delete_from_index_lists(Lits,Index,Clause,InitChangeList),
        functor(Index,Functor,Arity),
        functor(NewIndex,Functor,Arity),
        form_updated_index(InitChangeList,Arity,Index,NewIndex).

delete_from_index_lists([LiteralNum|Tail],Index,ClauseId,InitChange) :-
        arg(LiteralNum,Index,Arg),
        delete(clause(ClauseId,_,_,_),Arg,NewArg),
        add_element(InitChange,(LiteralNum,NewArg),NewChangeList),
        delete_from_index_lists(Tail,Index,ClauseId,NewChangeList).
delete_from_index_lists([],_,_,[]).

num_lits(Pos,Neg,NumLits) :-
        get_literals(Pos,PosLits),
        get_literals(Neg,NegLits),
        u_length(PosLits,PosLength),
        u_length(NegLits,NegLength),
        NumLits is PosLength + NegLength,
        !.

add_by_weight_to_sos(Sos,NumLits,ClauseId,NewSos) :-
        arg(NumLits,Sos,LiteralArg),
        argrep(NumLits,Sos,[ClauseId|LiteralArg],NewSos).

delete_from_sos(Sos,Id,NewSos,Clause,IdLookUp) :-
        array(IdLookUp,Id,Clause),
        Clause = clause(_,_,Pos,Neg),
        num_lits(Pos,Neg,NumLits),
        arg(NumLits,Sos,Arg),
        delete(Id,Arg,NewArg),
        !,
        argrep(NumLits,Sos,NewArg,NewSos).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                              %%%
%%%     Utility routines                                 %%%
%%%                                                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

database(state(Db,_),Db).

proof_completed(state(_,ProofCompleted),ProofCompleted).

make_state(Db,ProofCompleted,state(Db,ProofCompleted)).

% u_ prefix to avoid confusion with member/2 which may or may not be built-in
u_member(X,[X|_]).
u_member(X,[_|Y]) :- u_member(X,Y).

% u_ prefix to avoid confusion with length/2 which may or may not be built-in
u_length([],0).
u_length([_|T],N) :- u_length(T,I), N is I+1.

delete(X,[X|Y],Y).
delete(X,[H|T],[H|T2]) :- delete(X,T,T2).

writelist([]).
writelist([H|T]) :- write(H), write(' '), writelist(T).

argrep(N,Old,Value,New) :-
        functor(Old,Functor,Arity),
        functor(New,Functor,Arity),
        argrep(Old,1,Arity,N,Value,New).

argrep(Term,ArgNo,Arity,Index,Value,NewTerm) :-
        (   ArgNo > Arity ->
                true
        ;   /* otherwise -> */
                (   ArgNo =:= Index ->
                        arg(ArgNo,NewTerm,Value)
                ;   /* otherwise -> */
                        arg(ArgNo,Term,ArgVal),
                        arg(ArgNo,NewTerm,ArgVal)
                ),
                NewArgNo is ArgNo + 1,
                argrep(Term,NewArgNo,Arity,Index,Value,NewTerm)
        ).

%-------------------------------------------------------------------

% The following two predicates are used to define an empty array
% and set values in it.

% new_array(-Array)
new_array(A) :-
        functor(A,array,100).

% array(+Array,+Subscript,?Value)
% J gives the offset for entry in array A, K gives the offset in the
% Jth entry of the array.  We consider that A is an array of arrays.

array(A,I,E) :-
        J is ((I - 1) // 100) + 1,
        K is ((I - 1) mod 100) + 1,
        arg(J,A,SubArray),
        (var(SubArray) -> functor(SubArray,array,100); true),
        arg(K,SubArray,E).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%      Routines for writing clauses in readable form          %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

convert_to_external_form(Pos,Neg) :-
        get_literals(Pos,PosList),
        write_pos_symbols(PosList),
        get_literals(Neg,NegList),
        (   (NegList = [_|_], PosList = [_|_]) ->
                write(' | ')
        ;   /* otherwise -> */
                true
        ),
        write_neg_symbols(NegList).

write_pos_symbols([]).
write_pos_symbols([H|T]) :-
        write(v), write(H),
        (   T = [] ->
                true
        ;   /* otherwise -> */
                write(' | ')
        ),
        write_pos_symbols(T).

write_neg_symbols([]).
write_neg_symbols([H|T]) :-
        write('-'),
        write(v), write(H),
        (   T = [] ->
                true
        ;   /* otherwise -> */
                write(' | ')
        ),
        write_neg_symbols(T).

form_updated_index(InitChangeList,Arity,Index,NewIndex) :-
        form_updated_index(InitChangeList,1,Arity,Index,NewIndex).

form_updated_index([(ArgNo,Arg)|T],InitArg,Arity,Index,NewIndex) :-
        NewArgNo is InitArg + 1,
        (   InitArg =\= ArgNo ->
                arg(InitArg,Index,ArgVal),
                arg(InitArg,NewIndex,ArgVal),
                form_updated_index([(ArgNo,Arg)|T],NewArgNo,Arity,Index,
                        NewIndex)
        ;   /* otherwise -> */
                arg(InitArg,NewIndex,Arg),
                form_updated_index(T,NewArgNo,Arity,Index,NewIndex)
        ).
form_updated_index([],InitArg,Arity,Index,NewIndex) :-
        (   InitArg =< Arity ->
                arg(InitArg,Index,ArgVal),
                arg(InitArg,NewIndex,ArgVal),
                NewArgNo is InitArg + 1,
                form_updated_index([],NewArgNo,Arity,Index,NewIndex)
        ;   /* otherwise -> */
                true
        ).

add_element(List,E,NewList) :- List = [E|NewList].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Status Messages                                         %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_subsumed_mesg(Id,SubId) :-
        writelist([clause,SubId,subsumes,Id]),
        nl.

write_added_mesg(Id,clause(Parents,Pos,Neg)) :-
        writelist(['added',Id,Parents]),
        convert_to_external_form(Pos,Neg),
        nl.
```

```
-p0 ; -p1 ; p2.
-p4 ; -p10 ; p12.
-p17 ; p18 ; p19.
-p2 ; -p3 ; p10.
-p17 ; p4 ; p13.
-p7 ; -p11 ; p17.
-p2 ; -p12 ; -p4.
-p18 ; -p13 ; -p10.
-p0 ; p1 ; -p7 ; -p17.
-p0 ; -p3 ; p11.
-p2 ; -p19 ; p4.
p0 ; p1.
p2 ; p3.
p4 ; p7.
p10 ; p11.
p12 ; p13.
p17 ; p18.
```

boys.sos

```
p0.
p7.
```

empty

ct_2.sos

```
p0.
p1.
-p0 ; -p1.
```

ct_3.sos

```
p0 ; p1.
p2 ; p3.
p4 ; p5.
-p0 ; -p2.
-p0 ; -p4.
-p2 ; -p4.
-p1 ; -p3.
-p1 ; -p5.
-p3 ; -p5.
```

ct_4.sos

```
p0 ; p1 ; p2.
p3 ; p4 ; p5.
p6 ; p7 ; p8.
p9 ; p10 ; p11.
-p0 ; -p3.
-p0 ; -p6.
-p0 ; -p9.
-p3 ; -p6.
-p3 ; -p9.
-p6 ; -p9.
-p1 ; -p4.
-p1 ; -p7.
-p1 ; -p10.
-p4 ; -p7.
-p4 ; -p10.
-p7 ; -p10.
-p2 ; -p5.
-p2 ; -p8.
-p2 ; -p11.
-p5 ; -p8.
-p5 ; -p11.
-p8 ; -p11.
```

ct_5.sos

```
p0 ; p1 ; p2 ; p3.
p4 ; p5 ; p6 ; p7.
p8 ; p9 ; p10 ; p11.
p12 ; p13 ; p14 ; p15.
p16 ; p17 ; p18 ; p19.
-p0 ; -p4.
-p0 ; -p8.
-p0 ; -p12.
-p0 ; -p16.
-p4 ; -p8.
-p4 ; -p12.
-p4 ; -p16.
-p8 ; -p12.
-p8 ; -p16.
-p12 ; -p16.
-p1 ; -p5.
-p1 ; -p9.
-p1 ; -p13.
-p1 ; -p17.
-p5 ; -p9.
-p5 ; -p13.
-p5 ; -p17.
-p9 ; -p13.
-p9 ; -p17.
-p13 ; -p17.
-p2 ; -p6.
-p2 ; -p10.
-p2 ; -p14.
-p2 ; -p18.
-p6 ; -p10.
-p6 ; -p14.
-p6 ; -p18.
-p10 ; -p14.
-p10 ; -p18.
-p14 ; -p18.
-p3 ; -p7.
-p3 ; -p11.
-p3 ; -p15.
-p3 ; -p19.
-p7 ; -p11.
-p7 ; -p15.
-p7 ; -p19.
-p11 ; -p15.
-p11 ; -p19.
-p15 ; -p19.
```

ct_6.sos

```
p0 ; p1 ; p2 ; p3 ; p4.
p5 ; p6 ; p7 ; p8 ; p9.
p10 ; p11 ; p12 ; p13 ; p14.
p15 ; p16 ; p17 ; p18 ; p19.
p20 ; p21 ; p22 ; p23 ; p24.
p25 ; p26 ; p27 ; p28 ; p29.
-p0 ; -p5.
-p0 ; -p10.
-p0 ; -p15.
-p0 ; -p20.
-p0 ; -p25.
-p5 ; -p10.
-p5 ; -p15.
-p5 ; -p20.
-p5 ; -p25.
-p10 ; -p15.
-p10 ; -p20.
-p10 ; -p25.
-p15 ; -p20.
-p15 ; -p25.
-p20 ; -p25.
-p1 ; -p6.
-p1 ; -p11.
-p1 ; -p16.
-p1 ; -p21.
-p1 ; -p26.
-p6 ; -p11.
-p6 ; -p16.
-p6 ; -p21.
-p6 ; -p26.
-p11 ; -p16.
-p11 ; -p21.
-p11 ; -p26.
-p16 ; -p21.
-p16 ; -p26.
-p21 ; -p26.
-p2 ; -p7.
-p2 ; -p12.
-p2 ; -p17.
-p2 ; -p22.
-p2 ; -p27.
-p7 ; -p12.
-p7 ; -p17.
-p7 ; -p22.
-p7 ; -p27.
-p12 ; -p17.
-p12 ; -p22.
-p12 ; -p27.
-p17 ; -p22.
-p17 ; -p27.
-p22 ; -p27.
-p3 ; -p8.
-p3 ; -p13.
-p3 ; -p18.
-p3 ; -p23.
-p3 ; -p28.
-p8 ; -p13.
-p8 ; -p18.
-p8 ; -p23.
-p8 ; -p28.
-p13 ; -p18.
-p13 ; -p23.
-p13 ; -p28.
-p18 ; -p23.
-p18 ; -p28.
-p23 ; -p28.
```

```
-p4 ; -p9.
-p4 ; -p14.
-p4 ; -p19.
-p4 ; -p24.
-p4 ; -p29.
-p9 ; -p14.
-p9 ; -p19.
-p9 ; -p24.
-p9 ; -p29.
-p14 ; -p19.
-p14 ; -p24.
-p14 ; -p29.
-p19 ; -p24.
-p19 ; -p29.
-p24 ; -p29.
```

.boys.bench

```
# /*
  set-up.boys: bench set-up for (tp) boys
  */
boys :- driver(boys).

benchmark(boys,
          (do('examples/boys.ax','examples/boys.sos'), !),
          (dummy('examples/boys.ax','examples/boys.sos'), !),
          1).

show(boys) :- assert(show),
              do('examples/boys.ax','examples/boys.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                                %%%
%%%     Execution time measurement code                            %%%
%%%                                                                %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

```
# /*
  set-up.ct_2: bench set-up for (tp) ct_2
  */
ct_2 :- driver(ct_2).

benchmark(ct_2,
          (do('examples/empty','examples/ct_2.sos'), !),
          (dummy('examples/empty','examples/ct_2.sos'), !),
          50).

show(ct_2) :- assert(show),
              do('examples/empty','examples/ct_2.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                            %%%
%%%     Execution time measurement code                        %%%
%%%                                                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

.ct_3.bench

```
# /*
  set-up.ct_3: bench set-up for (tp) ct_3
  */
ct_3 :- driver(ct_3).

benchmark(ct_3,
          (do('examples/empty','examples/ct_3.sos'), !),
          (dummy('examples/empty','examples/ct_3.sos'), !),
          15).

show(ct_3) :- assert(show),
              do('examples/empty','examples/ct_3.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Execution time measurement code                         %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

```
# /*
  set-up.ct_4: bench set-up for (tp) ct_4
  */
ct_4 :- driver(ct_4).

benchmark(ct_4,
          (do('examples/empty','examples/ct_4.sos'), !),
          (dummy('examples/empty','examples/ct_4.sos'), !),
          1).

show(ct_4) :- assert(show),
              do('examples/empty','examples/ct_4.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Execution time measurement code                         %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

.ct_5.bench

```
# /*
  set-up.ct_5: bench set-up for (tp) ct_5
  */
ct_5 :- driver(ct_5).

benchmark(ct_5,
          (do('examples/empty','examples/ct_5.sos'), !),
          (dummy('examples/empty','examples/ct_5.sos'), !),
          1).

show(ct_5) :- assert(show),
              do('examples/empty','examples/ct_5.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Execution time measurement code                         %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

.ct_6.bench

```
# /*
  set-up.ct_6: bench set-up for (tp) ct_6
  */
ct_6 :- driver(ct_6).

benchmark(ct_6,
          (do('examples/empty','examples/ct_6.sos'), !),
          (dummy('examples/empty','examples/ct_6.sos'), !),
          1).

show(ct_6) :- assert(show),
              do('examples/empty','examples/ct_6.sos'), !,
              retract(show).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                             %%%
%%%     Execution time measurement code                         %%%
%%%                                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include "driver"
```

[c] README

```
tp (C version)

Ross Overbeek
(312) 972-7856
overbeek@anl-mcs.arpa

This directory (tp/c) contains the C ver-
sion of the propositional theorem prover.

To make it on a new machine, do

    touch *.c
    make tp

Once the make completes, do

    tp < ../examples/boys.in > boys.out
    diff boys.out ../examples/boys.out.verify

to get output and verify that it is correct.

The execution should take only a few seconds.
```

[c] alloc.c

```
#include "tp.h"

init_formula_db(db)
struct formula_db *db;
{
int usable_sz, sos_sz, i;

    db->next_avl = db->formula_storage + 1;
    db->end_avl = db->formula_storage + MAX_CLAUSES;

    /* get storage for list of usable formulas */

    usable_sz = NUM_CLASH_ENTRIES * sizeof(struct formula *);
    db->usable.first = (struct formula **) malloc(usable_sz);
    if (db->usable.first == NULL)
    {
        printf("malloc for usable list failed in master\n");
        exit(1);
    }
    *(db->usable.first) = NULL;
    db->usable.next_avl = db->usable.first;
    db->usable.end_avl = db->usable.first + NUM_CLASH_ENTRIES;

    /* get storage for list of sos formulas     */
    /* number of bytes in the sos list:         */
    /* 2 * maxvar for 1 lit formulas            */
    /* maxvar * maxvar for 2 lit formulas       */
    /* 2000 entries each for 3-6 lit formulas   */
    /*  500 entries each for 7-64 lit formulas  */

    for (i=0; i < MAXVAR; i++)
    {
        if (i == 0)      /* formulas with 1 lit */
            sos_sz = 2 * MAXVAR;
        else if (i == 1)
            sos_sz = 2 * MAXVAR * MAXVAR;
        else if (i == 2)
            sos_sz = 8000;
        else
            sos_sz = 500;
        db->by_weight_in_sos[i].first =
           (struct formula **) malloc(sos_sz * sizeof(struct formula *));
        if (db->by_weight_in_sos[i].first == NULL)
        {
            printf("malloc for by_wt_in_sos list failed in master\n");
            exit(1);
        }
        *(db->by_weight_in_sos[i].first) = NULL;
        db->by_weight_in_sos[i].next_avl =
                db->by_weight_in_sos[i].first;
        db->by_weight_in_sos[i].end_avl =
                db->by_weight_in_sos[i].first + sos_sz;
    }

    /* get storage for subsumption and usable indices */

    alloc_index(&(db->sub_index),NUM_SUB_IDX_ENTRIES);
    alloc_index(&(db->clash_index),NUM_CLASH_IDX_ENTRIES);
}
```

[c] alloc.c

```
alloc_index(idx,num_entries)
struct cl_index *idx;
int num_entries;
{
int i;

    for (i=0; i < MAXVAR; i++)
    {
        idx->pos_lits[i].first =
                (struct formula **) malloc(num_entries * sizeof(struct formula *));
        if (idx->pos_lits[i].first == NULL)
        {
            printf("malloc for idx list failed in master\n");
            exit(1);
        }
        *(idx->pos_lits[i].first) = NULL;
        idx->pos_lits[i].next_avl = idx->pos_lits[i].first;
        idx->pos_lits[i].end_avl = idx->pos_lits[i].first + num_entries;

        idx->neg_lits[i].first =
            (struct formula **) malloc(num_entries * sizeof(struct formula *));
        if (idx->neg_lits[i].first == NULL)
        {
            printf("malloc for idx list failed in master\n");
            exit(1);
        }
        *(idx->neg_lits[i].first) = NULL;
        idx->neg_lits[i].next_avl = idx->neg_lits[i].first;
        idx->neg_lits[i].end_avl = idx->neg_lits[i].first + num_entries;
    }
}
```

[c] c.macros

```
#define TestBit(Bitvec,Var) ((Bitvec)->word1 & (1 << (Var)))

#define Numlits(C)  (count_bits((C)->positive.word1) +
                     count_bits((C)->negative.word1))

#define CopyBitvec(From,To) To = From;

#define Subsumes(C1,C2)
     ((((C1)->positive.word1 & (C2)->positive.word1) == (C1)->positive.word1) &&
     (((C1)->negative.word1 & (C2)->negative.word1) == (C1)->negative.word1))


#define MakeResolvent(C1,C2,V,R)
    (R).parents[0] = (C1)->id;
    (R).parents[1] = (C2)->id;
        (R).positive.word1 =
            ((C1)->positive.word1 | (C2)->positive.word1) ^ (1 << (V));
        (R).negative.word1 =
            ((C1)->negative.word1 | (C2)->negative.word1) ^ (1 << (V));

#define Printcls(C)
    {
        printf("%d [%d,%d] ",(C)->id,(C)->parents[0],(C)->parents[1]);
        print_clause(&((C)->positive),&((C)->negative));
        printf("\n");
    }
```

[c] clocks.c

```
#include "tp.h"

#include <sys/time.h>
#include <sys/resource.h>

struct clock {
    int accum_sec;   /* accumulated seconds */
    int accum_usec;  /* accumulated microseconds*/
    int curr_sec;
    int curr_usec;
    };

struct clock clocks[MAX_CLOCKS];

/*************
 *
 *    clock_init() - Initialize all clocks.
 *
 *************/

clock_init()
{
    int i;
    for (i=0; i<MAX_CLOCKS; i++)
        clock_reset(i);
}  /* clock_init */

/*************
 *
 *    cpu_time(sec, usec) - It has been sec seconds + usec microseconds
 *                          since the start of this process.
 *
 *************/

cpu_time(seconds, microseconds)
int *seconds, *microseconds;
{
    struct rusage r;

    getrusage(0, &r);
    *seconds = r.ru_utime.tv_sec;
    *microseconds = r.ru_utime.tv_usec;
}  /* cpu_time */

/*************
 *
 *    clock_start(clock_num) - Start or continue timing.
 *
 *        If the clock is already running, a warning message is printed.
 *
 *************/

clock_start(c)
int c;
{
    struct clock *cp;

    cp = &clocks[c];
    if (cp->curr_sec != -1) {
        fprintf(stderr, "WARNING, clock_start: clock %d already on.\n", c);
        printf("WARNING, clock_start: clock %d already on.\n", c);
        }
    else
        cpu_time(&cp->curr_sec, &cp->curr_usec);
}  /* clock_start */
```

```
/*************
 *
 *    clock_stop(clock_num) - Stop timing and add to accumulated total.
 *
 *        If the clock not running, a warning message is printed.
 *
 *************/

clock_stop(c)
{
    int sec, usec;
    struct clock *cp;

    cp = &clocks[c];
    if (cp->curr_sec == -1) {
        fprintf(stderr, "WARNING, clock_stop: clock %d already on.\n", c);
        printf("WARNING, clock_stop: clock %d already on.\n", c);
        }
    else {
        cpu_time(&sec, &usec);
        cp->accum_sec += sec - cp->curr_sec;
        cp->accum_usec += usec - cp->curr_usec;
        cp->curr_sec = -1;
        cp->curr_usec = -1;
        }
}  /* clock_stop */

/*************
 *
 *    int clock_val(clock_num) - Returns accumulated time in milliseconds.
 *
 *        Clock need not be stopped.
 *
 *************/

int clock_val(c)
int c;
{
    int i, j, sec, usec;

    i = (clocks[c].accum_sec * 1000) + (clocks[c].accum_usec / 1000);
    if (clocks[c].curr_sec == -1)
        return(i);
    else {
        cpu_time(&sec, &usec);
        j = ((sec - clocks[c].curr_sec) * 1000) +
            ((usec - clocks[c].curr_usec) / 1000);
        return(i+j);
        }
}  /* clock_val */

/*************
 *
 *    clock_reset(clock_num) - Clocks must be reset before being used.
 *
 *************/

clock_reset(c)
int c;
{
    clocks[c].accum_sec = clocks[c].accum_usec = 0;
    clocks[c].curr_sec = clocks[c].curr_usec = -1;
}  /* clock_reset */
```

[c] db.c

```
#include "tp.h"

add_to_sos(db,ptr_formula)
struct formula_db *db;
struct formula *ptr_formula;
{
int pos_ints[MAXVAR], neg_ints[MAXVAR];
struct formula *cls;
struct list *lst;

    /*
    if usable formulas storage is exhausted then simply terminate
    the program
    */

    if ( db->next_avl > db->end_avl)
    {
        printf("\n\n Usable Formula Area Overflow \n\n");
        exit(1);
    }

    /*
    place the usable formula at the next available position,
    as pointed to by the next_avl pointer, in the formula
    storage
    */

    cls = db->next_avl++;

    /* generate formula id */

    cls->id = (cls - db->formula_storage);

    cls->parents[0] = ptr_formula->parents[0];
    cls->parents[1] = ptr_formula->parents[1];
    CopyBitvec(ptr_formula->positive.word1,cls->positive.word1);
    CopyBitvec(ptr_formula->negative.word1,cls->negative.word1);

    /*
    get pointer to the array entry by_weight_in_sos, in
    whose list this particular formula is to be placed
    */

    lst = db->by_weight_in_sos + (Numlits(cls) - 1);
    add_to_list(lst,cls);

    add_to_index(&(db->sub_index),cls);
    return(cls->id);
}

move_to_usable(db,id)         /* mv cls from sos to usable */
struct formula_db *db;
int id;
{
struct formula *c;
int num_lits, pos_ints[MAXVAR], neg_ints[MAXVAR];


    c = db->formula_storage + id;
    if (del_from_list(&(db->by_weight_in_sos[Numlits(c)-1]),c))
    {
        add_to_list(&(db->usable),c);
        add_to_index(&(db->clash_index),c);
    }
    else
        printf("*** mv_cmd: invalid move of cl %d\n",id);
}
```

```
delete_from_database(db,id)
struct formula_db *db;
int id;
{
struct formula *c;


    c = db->formula_storage + id;
    if (del_from_list(&(db->usable),c))
    {
        del_from_index(&(db->clash_index),c);
    }
    else
    {
        del_from_list(&(db->by_weight_in_sos[Numlits(c)-1]),c);
    }
    del_from_index(&(db->sub_index),c);
}

int count_bits(v)
register unsigned int v;
{
register int i;
register unsigned int j;

    for (i=0,j=v; j; i++)
        j &= (j-1);
    return(i);
}

add_to_index(ndx,cl)
struct cl_index *ndx;
struct formula *cl;
{
int pos_ints[MAXVAR], neg_ints[MAXVAR];
int p,n,i;

    /*
        adds the formula to the list of formulas, maintained for
        each literal ( +ve or -ve )
    */
    p = lit_nums(&(cl->positive),pos_ints);
    n = lit_nums(&(cl->negative),neg_ints);

    /* add formula to sub_index for each lit represented */

    for (i=0; i < p; i++)
        add_to_list(&(ndx->pos_lits[pos_ints[i]]),cl);

    for (i=0; i < n; i++)
        add_to_list(&(ndx->neg_lits[neg_ints[i]]),cl);
}

lit_nums(bits,ints)
BITVEC *bits;
int *ints;
{
int count = 0;
int *ni = ints;
int i;
/*
    Returns the number of +ve or -ve literals in the formula.
    It also places in the array ints the entry of which literal
    ( out of all those available ) occured in a formula
*/

    for (i=0; i < MAXVAR; i++)
        if (TestBit(bits,i))
        {
            count++;
            *(ni++) = i;
        }

    return(count);
}

add_to_list(alist,aformula)
struct list *alist;
struct formula *aformula;
{
struct formula **p,**p1,**p2;
int i;

    if (alist->next_avl  >=  alist->end_avl)
    {
        i = (alist->next_avl - alist->first) * 2;
        p = (struct formula **) malloc(i * sizeof(struct formula *));
        if (!p)
        {
            printf("*** failure to allocate in add_to_list: aborting\n");
            exit(1);
        }

        for (p1=alist->first,p2=p; p1 < alist->next_avl;)
            *(p2++) = *(p1++);

        free(alist->first);
        alist->first = p;
        alist->next_avl = p2;
        alist->end_avl = p + i;
    }
    *(alist->next_avl) = aformula;
    (alist->next_avl)++;
}

del_from_index(ndx,cl)
struct cl_index *ndx;
struct formula *cl;
{
int pos_ints[MAXVAR], neg_ints[MAXVAR];
int p,n,i;

    p = lit_nums(&(cl->positive),pos_ints);
    n = lit_nums(&(cl->negative),neg_ints);

    for (i=0; i < p; i++)
        del_from_list(&(ndx->pos_lits[pos_ints[i]]),cl);

    for (i=0; i < n; i++)
        del_from_list(&(ndx->neg_lits[neg_ints[i]]),cl);
}
```

```
del_from_list(alist,aformula)
struct list *alist;
struct formula *aformula;
{
int rc;
struct formula **c;


    for (c=alist->first; c < alist->next_avl  &&  *c != aformula; c++)
        ;

    if (*c == aformula)
    {
        (alist->next_avl)--;
        while (c < alist->next_avl)
        {
            *c = *(c + 1);
            c++;
        }

        rc = TRUE;
    }
    else
        rc = FALSE;

    return(rc);
}
```

[c] io.c

```
#include "tp.h"

#define POS 1
#define NEG 2

char symtab[MAXVAR][MAX_SYMBOL_LEN];
static next_var = 0;

char *get_word(), *skip_white();

read_input(db)
struct formula_db *db;
{
int new_formula;
char axm_file[STRINGLEN], sos_file[STRINGLEN];
FILE *af, *sf;
struct formula clause;
BITVEC pos, neg;

    printf("Enter axm filename : ");
    scanf("%s",axm_file);
    printf("\n");

    if ((af = fopen(axm_file,"r")) == NULL)
    {
        fprintf(stderr,"Error opening axiom file %s: ",axm_file);
        perror("");
        exit(1);
    }

    printf("Enter sos filename : ");
    scanf("%s",sos_file);
    printf("\n");

    if ((sf = fopen(sos_file,"r")) == NULL)
    {
        fprintf(stderr,"Error opening sos file %s: ",sos_file);
        perror("");
        exit(1);
    }

    clause.parents[0] = -1;
    clause.parents[1] = -1;

    while(read_literals(af,&(clause.positive),&(clause.negative)))
    {
        new_formula = add_to_sos(db,&clause);

        printf("added ");
        Printcls(db->formula_storage+new_formula)

        move_to_usable(db,new_formula);
    }

    while(read_literals(sf,&(clause.positive),&(clause.negative)))
    {
        new_formula = add_to_sos(db,&clause);

        printf("added ");
        Printcls(db->formula_storage+new_formula)
    }
}

BOOL read_literals(fp,pos,neg)
FILE *fp;
BITVEC *pos, *neg;
{
int i, vnum, var, sign;
char *bptr, buf[MAXVAR * (MAX_SYMBOL_LEN + 10)];
char word[MAX_SYMBOL_LEN];
```

```
    if (next_var == 0)                      /* first call */
        for (i = 0; i < MAXVAR; i++)
            symtab[i][0] = '\0';

    pos->word1 = neg->word1 = 0;
    bptr = buf;
    while (((*bptr = getc(fp)) != '.') && (*bptr != EOF))
        if (*bptr != '\n')
            bptr++;

    if (*bptr == EOF)
        return FALSE;

    *bptr = '\0';

    bptr = buf;
    while (*bptr)
    {
        bptr = get_word(bptr,word,&sign);
        var = find_word_in_symtab(word);

        if (sign == POS)
            pos->word1 |= 1 << var;
        else
            neg->word1 |= 1 << var;
    }
    return TRUE;
}

char *get_word(bptr,word,sign)
char *bptr, *word;
int *sign;
{

    *sign = POS;

    bptr = skip_white(bptr);

    if (*bptr == '-')
    {
        *sign = NEG;
        bptr++;
    }

    bptr = skip_white(bptr);
    while (*bptr && (*bptr != ' '))
    {
        *word++ = *bptr++;
    }

    *word = '\0';
    if (*bptr)
    {
        bptr = skip_white(bptr);

        if ((*bptr == ';') || (*bptr == '.'))
            bptr++;
        else
        {
            fprintf(stderr, "Syntax error\n");
            return FALSE;
        }
    }

    return skip_white(bptr);
}
```

```
char *skip_white(ptr)
char *ptr;
{
    while (*ptr && (*ptr == ' ') || (*ptr == '	'))
        ptr++;

    return ptr;
}

int find_word_in_symtab(word)
char *word;
{
register int i;

    for (i = 0; i < next_var; i++)
        if (strcmp(symtab[i],word) == 0)
            return i;

    if (next_var < MAXVAR - 1)
    {
        strcpy(symtab[next_var],word);
        return(next_var++);
    }
    else
    {
        fprintf(stderr,"Too many variables\n");
        exit(1);
    }
}

print_clause(pos,neg)
BITVEC *pos, *neg;
{
int i;
unsigned int negword = neg->word1, posword = pos->word1;

    for (i = 0; i < MAXVAR; i++)
    {
        if (posword & 1)
        {
            posword >>= 1;
            printf("%s ",symtab[i]);
            if (posword || negword)
                printf("| ");
        }
        else
            posword >>= 1;
    }

    for (i = 0; i < MAXVAR; i++)
    {
        if (negword & 1)
        {
            negword >>= 1;
            printf("-%s ",symtab[i]);
            if (negword)
                printf("| ");
        }
        else
            negword >>= 1;
    }
}
```

[c] makefile

```
CFLAGS = -g

TP_FILES = tp.o subsump.o db.o alloc.o io.o clocks.o

tp : $(TP_FILES)
        cc $(CFLAGS) -o tp $(TP_FILES)
```

[c] subsump.c

```
#include "tp.h"

forward_sub(db,res)
struct formula_db *db;
struct formula *res;
{
int pos_lits[MAXVAR],neg_lits[MAXVAR];
struct formula **pt;
struct list *lst;
int p,n,i,*nl;
BOOL subsumed = FALSE;
int subsumer;

    p = lit_nums(&res->positive,pos_lits);

    for (i = 0,nl = pos_lits; !subsumed && (i < p); i++,nl++)
    {
        lst = &(db->sub_index.pos_lits[*nl]);
        for (pt = lst->first; !subsumed && (pt < lst->next_avl); pt++)
             if (Subsumes((*pt),res))
             {
                 subsumed = TRUE;
                 subsumer = (*pt)->id;
             }
    }

    if (!subsumed)
    {
        n = lit_nums(&res->negative,neg_lits);

        for (i = 0, nl = neg_lits; !subsumed && (i < n); i++, nl++)
        {
            lst = &(db->sub_index.neg_lits[*nl]);
            for (pt = lst->first; !subsumed && (pt < lst->next_avl); pt++)
            {
                if (Subsumes((*pt),res))
                {
                    subsumed = TRUE;
                    subsumer = (*pt)->id;
                }
            }
        }
    }

    return(subsumed ? subsumer : -1);
}

back_sub(db,formula_id)
int formula_id;
struct formula_db *db;
{
int lits[MAXVAR];
struct formula **pt;
struct list *lst;
int p,n,i,*nl;
struct formula *cl;

    cl = db->formula_storage + formula_id;

    if (p = lit_nums(&(cl->positive),lits))
        lst = &(db->sub_index.pos_lits[lits[0]]);
    else
    {
        n = lit_nums(&(cl->negative),lits);
        lst = &(db->sub_index.neg_lits[lits[0]]);
    }
    for (pt = lst->next_avl - 1; pt >= lst->first; pt--)
    {
        if ((formula_id != (*pt)->id) && Subsumes(cl,(*pt)))
        {
            printf("clause %d subsumes %d \n", formula_id,(*pt)->id);
            add_to_deleted_list((*pt)->id);
        }
    }
}

static int clauses_to_delete[10000];
static int next_to_delete = 0;

add_to_deleted_list(id)
int id;
{
register int i;

    for (i = 0; i < next_to_delete && clauses_to_delete[i] != id; i++)
        ;

    if (i == next_to_delete)
        clauses_to_delete[next_to_delete++] = id;
}

delete_saved_clauses(db)
struct formula_db *db;
{
int i;

     for (i = 0; i < next_to_delete; i++)
        delete_from_database(db,clauses_to_delete[i]);

    next_to_delete = 0;
}
```

[c] tp.c

```
#include "tp.h"

struct formula_db master_db;

main()
{
BOOL proof_completed;

    clock_init();

    init_formula_db(&master_db);
    read_input(&master_db);

    clock_start(CLK_RUNTIME);
    proof_completed = generate(&master_db);
    clock_stop(CLK_RUNTIME);

    if (proof_completed)
        printf("Proof found\n");
    else
        printf("Proof not found\n");

    printf("Total time is %f sec\n",(float)clock_val(CLK_RUNTIME) / 1000.0);
}

BOOL generate(db)
struct formula_db *db;
{
BOOL proof_completed = FALSE;
int given;

    while (((given = pick_given_formula(db)) != -1) && !proof_completed)
    {
        printf("given %d\n",given);
        move_to_usable(db,given);
        proof_completed = gen_from_given(db,given);
    }

    return proof_completed;
}

BOOL gen_from_given(db,given)
struct formula_db *db;
int given;
{
struct formula *c1, c2;
int p, n;
int *nl, i;
struct list *lst;
struct formula **pt;
int pos_lits[MAXVAR], neg_lits[MAXVAR];
BOOL proof_completed = FALSE;

    c1 = db->formula_storage + given;

    p = lit_nums(&(c1->positive),pos_lits);
    n = lit_nums(&(c1->negative),neg_lits);

    for (i = 0, nl = pos_lits; (i < p) && !proof_completed; i++, nl++)
    {
        lst = &(db->clash_index.neg_lits[*nl]);

        for (pt = lst->next_avl - 1; (pt>=lst->first) && !proof_completed; pt--)
            proof_completed = gen_one_resolvent(db,c1,*pt,*nl);

        delete_saved_clauses(db);
    }

    for (i = 0, nl = neg_lits; (i < n) && !proof_completed; i++, nl++)
    {
        lst = &(db->clash_index.pos_lits[*nl]);

        for (pt =lst->next_avl - 1;(pt >= lst->first) && !proof_completed; pt--)
            proof_completed = gen_one_resolvent(db,cl,*pt,*nl);

        delete_saved_clauses(db);
    }

    return proof_completed;
}

BOOL gen_one_resolvent(db,c1,c2,var)
struct formula_db *db;
struct formula *c1, *c2;
int var;
{
struct formula resolvent;
int id;

    MakeResolvent(c1, c2, var, resolvent);

    if (Numlits(&resolvent) == 0)
    {
        printf("derived null clause from %d and %d\n",resolvent.parents[0],
                resolvent.parents[1]);
        return TRUE;
    }

    if (tautology(&resolvent))
        return FALSE;

    if (forward_sub(db,&resolvent) == -1)
    {
        id = add_to_sos(db,&resolvent);

        printf("added ");
        Printcls(db->formula_storage + id);

        back_sub(db,id);
    }

    return FALSE;
}

BOOL tautology(cl)
struct formula *cl;
{
    return (cl->positive.word1 & cl->negative.word1) != 0;
}

int pick_given_formula(db)
struct formula_db *db;
{
int i,retval;

    /* returns next formula from set of support */

    for (i=0;
         i <= MAX_WEIGHT  &&
         (db->by_weight_in_sos[i].first == db->by_weight_in_sos[i].next_avl);
         i++)
         ;

    if (i <= MAX_WEIGHT)
        retval = (*(db->by_weight_in_sos[i].next_avl - 1))->id;
    else
        retval = -1;

    return(retval);
}
```

[c] tp.c

```
#include <stdio.h>

#define MAX_WEIGHT           31
#define MAXVAR               32
#define MAX_CLAUSES          20000
#define NEW_CLAUSE           0
#define MAX_SYMBOL_LEN       256
#define STRINGLEN            50

#define NUM_CLASH_ENTRIES      2000
#define NUM_SUB_IDX_ENTRIES    2000
#define NUM_CLASH_IDX_ENTRIES  2000

#define MAX_CLOCKS 30

#define CLK_RUNTIME 0

typedef int BOOL;
#define TRUE 1
#define FALSE 0

typedef struct {unsigned int word1;} BITVEC;

struct formula {
    int id;              /* id of the formula */
    int parents[2];      /* ids of parents ([-1,-1] for input formula) */
    BITVEC positive;     /* bits set to reflect positive literals:
                            rightmost bit represents p0
                         */
    BITVEC negative;     /* bits set to represent negative literals */
};

struct list {   /* one entry for signed prop. variable */
    struct formula **first;
    struct formula **next_avl;
    struct formula **end_avl;
};

struct cl_index {
    struct list pos_lits[MAXVAR];
    struct list neg_lits[MAXVAR];
};


struct formula_db {
    struct cl_index sub_index;          /* into all formulas */
    struct cl_index clash_index;        /* into clashable formulas */

    struct formula formula_storage[MAX_CLAUSES * sizeof(struct formula)];
    struct formula *next_avl;           /* points to entry in
                                           formula_storage (next available
                                           formula entry)
                                        */
    struct formula *end_avl;            /* last entry in formula_storage */

    struct list by_weight_in_sos[MAX_WEIGHT+1]; /* set-of-support */
    struct list usable;         /* usable formulas */


};

#include "c.macros"
```

warren

divide10.m

```
# /*
  divide10.m: Warren benchmark (deriv) divide10 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (deriv) divide10
%
%   David H. D. Warren
%
%   symbolic derivative of ((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x

#assign DIVIDE10_EXP    ((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x
#
#if BENCH
#  include ".divide10.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
divide10 :- d(DIVIDE10_EXP,x,D),
            write('(d/dx)('),
            write(DIVIDE10_EXP),
            write(') ='), nl,
            write(D), nl.
#  else
divide10 :- d(DIVIDE10_EXP,x,_).
#  endif
#endif

#include "deriv"        /* code for symbolic derivative */
```

```
# /*
  log10.m: Warren benchmark (deriv) log10 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (deriv) log10
%
%   David H. D. Warren
%
%   symbolic derivative of log(log(log(log(log(log(log(log(log(log(x))))))))))

#assign LOG10_EXP       log(log(log(log(log(log(log(log(log(log(x))))))))))
#
#if BENCH
#  include ".log10.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
log10 :- d(LOG10_EXP,x,D),
         write('(d/dx)('),
         write(LOG10_EXP),
         write(') ='), nl,
         write(D), nl.
#  else
log10 :- d(LOG10_EXP,x,_).
#  endif
#endif

#include "deriv"        /* code for symbolic derivative */
```

```
# /*
  ops8.m: Warren benchmark (deriv) ops8 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (deriv) ops8
%
%   David H. D. Warren
%
%   symbolic derivative of (x+1)*((^(x,2)+2)*(^(x,3)+3))

#assign OPS8_EXP        (x+1)*((^(x,2)+2)*(^(x,3)+3))
#
#if BENCH
#  include ".ops8.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
ops8 :- d(OPS8_EXP,x,D),
        write('(d/dx)('),
        write(OPS8_EXP),
        write(') ='), nl,
        write(D), nl.
#  else
ops8 :- d(OPS8_EXP,x,_).
#  endif
#endif

#include "deriv"        /* code for symbolic derivative */
```

```
# /*
  times10.m: Warren benchmark (deriv) times10 master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   (deriv) times10
%
%   David H. D. Warren
%
%   symbolic derivative of ((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x

#assign TIMES10_EXP     ((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x
#
#if BENCH
#  include ".times10.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
times10 :- d(TIMES10_EXP,x,D),
           write('(d/dx)('),
           write(TIMES10_EXP),
           write(') ='), nl,
           write(D), nl.
#  else
times10 :- d(TIMES10_EXP,x,_).
#  endif
#endif

#include "deriv"       /* code for symbolic derivative */
```

deriv

```
# /*
  deriv: Warren code for symbolic derivative
  */
#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (d/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
d(_,_,_).
#else
d(U+V,X,DU+DV) :- !,
    d(U,X,DU),
    d(V,X,DV).
d(U-V,X,DU-DV) :- !,
    d(U,X,DU),
    d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !,
    d(U,X,DU),
    d(V,X,DV).
d(U/V,X,(DU*V-U*DV)/(^(V,2))) :- !,
    d(U,X,DU),
    d(V,X,DV).
d(^(U,N),X,DU*N*(^(U,N1))) :- !,
    integer(N),
    N1 is N-1,
    d(U,X,DU).
d(-U,X,-DU) :- !,
    d(U,X,DU).
d(exp(U),X,exp(U)*DU) :- !,
    d(U,X,DU).
d(log(U),X,DU/U) :- !,
    d(U,X,DU).
d(X,X,1) :- !.
d(_,_,0).
#endif
```

```
# /*
  nreverse.m: Warren benchmark nreverse master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

#if BENCH
#  include ".nreverse.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],R),
            write('reverse of'), nl,
            write([1,2,3,4,5,6,7,8,9,10,11,12,
                   13,14,15,16,17,18,19,20,21,
                   22,23,24,25,26,27,28,29,30]), nl,
            write(is), nl,
            write(R), nl.
#  else
nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],_).
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (nreverse/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
nreverse(_,_).
#else
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
#endif
```

```
# /*
  qsort.m: Warren benchmark qsort master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   qsort
%
%   David H. D. Warren
%
%   quicksort a list of 50 integers

#if BENCH
#  include ".qsort.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
qsort :- qsort([27,74,17,33,94,18,46,83,65, 2,
                32,53,28,85,99,47,28,82, 6,11,
                55,29,39,81,90,37,10, 0,66,51,
                 7,21,85,27,31,63,75, 4,95,99,
                11,28,61,74,18,92,40,53,59, 8],S,[]),
         write('qsort of'), nl,
         write([27,74,17,33,94,18,46,83,65, 2,
                32,53,28,85,99,47,28,82, 6,11,
                55,29,39,81,90,37,10, 0,66,51,
                 7,21,85,27,31,63,75, 4,95,99,
                11,28,61,74,18,92,40,53,59, 8]), nl,
         write(is), nl,
         write(S), nl.
#  else
qsort :- qsort([27,74,17,33,94,18,46,83,65, 2,
                32,53,28,85,99,47,28,82, 6,11,
                55,29,39,81,90,37,10, 0,66,51,
                 7,21,85,27,31,63,75, 4,95,99,
                11,28,61,74,18,92,40,53,59, 8],_,[]).
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (qsort/3).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
```

```
#if DUMMY
qsort(_,_,_).
#else
qsort([X|L],R,R0) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :-
        X =< Y, !,
        partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
        partition(L,Y,L1,L2).
partition([],_,[],[]).
#endif
```

```
# /*
  query.m: Warren benchmark query master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   query
%
%   David H. D. Warren
%
%   query population and area database to find coun-
%   tries of approximately equal population density

#if BENCH
#  include ".query.bench"
#else
query :- run_query.
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (run_query/0).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
run_query.
#else
run_query :- query(_), fail.
run_query.

query([C1,D1,C2,D2]) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.

density(C,D) :-
    pop(C,P),
    area(C,A),
    D is (P*100)/A.
```

```
% populations in 100000's
pop(china,       8250).
pop(india,       5863).
pop(ussr,        2521).
pop(usa,         2119).
pop(indonesia,   1276).
pop(japan,       1097).
pop(brazil,      1042).
pop(bangladesh,   750).
pop(pakistan,     682).
pop(w_germany,    620).
pop(nigeria,      613).
pop(mexico,       581).
pop(uk,           559).
pop(italy,        554).
pop(france,       525).
pop(philippines,  415).
pop(thailand,     410).
pop(turkey,       383).
pop(egypt,        364).
pop(spain,        352).
pop(poland,       337).
pop(s_korea,      335).
pop(iran,         320).
pop(ethiopia,     272).
pop(argentina,    251).

% areas in 1000's of square miles
area(china,      3380).
area(india,      1139).
area(ussr,       8708).
area(usa,        3609).
area(indonesia,   570).
area(japan,       148).
area(brazil,     3288).
area(bangladesh,   55).
area(pakistan,    311).
area(w_germany,    96).
area(nigeria,     373).
area(mexico,      764).
area(uk,           86).
area(italy,       116).
area(france,      213).
area(philippines,  90).
area(thailand,    200).
area(turkey,      296).
area(egypt,       386).
area(spain,       190).
area(poland,      121).
area(s_korea,      37).
area(iran,        628).
area(ethiopia,    350).
area(argentina,  1080).
#endif
```

serialise.m

```
# /*
  serialise.m: Warren benchmark serialise master file
  */
% generated: __MDAY__ __MONTH__ __YEAR__
% option(s): $__OPTIONS__$
%
%   serialise
%
%   David H. D. Warren
%
%   itemize (pick a "serial number" for each
%   unique integer in) a list of 25 integers

#assign PALIN25         "ABLE WAS I ERE I SAW ELBA"
#
#if BENCH
#  include ".serialise.bench"
#else
#option SHOW "
        > Option SHOW introduces code which writes output
        > to show what the benchmark does.  This may help
        > verify that the benchmark operates correctly.
        >
        > SHOW has no effect when BENCH is selected.  The
        > functionality of SHOW is then available through
        > show/1."
#  if SHOW
serialise :- serialise(PALIN25,S),
             write('serialisation of'), nl,
             printstring(PALIN25), nl,
             write(is), nl,
             write(S), nl.

printstring([]).
printstring([H|T]) :- put(H), printstring(T).
#  else
serialise :- serialise(PALIN25,_).
#  endif
#endif

#option DUMMY "
        > To facilitate overhead subtraction for performance
        > statistics, option DUMMY substitutes a 'dummy' for
        > the benchmark execution predicate (serialise/2).
        >
        > To use this, generate code without DUMMY and run
        > it, generate code with DUMMY and run it, and take
        > the difference of the performance statistics.
        >
        > This functionality is automatically provided with
        > execution time measurement when BENCH is selected."
#if DUMMY
serialise(_,_).
#else
serialise(L,R) :-
    pairlists(L,R,A),
    arrange(A,T),
    numbered(T,1,_).

pairlists([X|L],[Y|R],[pair(X,Y)|A]) :- pairlists(L,R,A).
pairlists([],[],[]).
```

serialise.m

```
arrange([X|L],tree(T1,X,T2)) :-
    split(L,X,L1,L2),
    arrange(L1,T1),
    arrange(L2,T2).
arrange([],void).

split([X|L],X,L1,L2) :- !, split(L,X,L1,L2).
split([X|L],Y,[X|L1],L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([],_,[],[]).

before(pair(X1,_),pair(X2,_)) :- X1 < X2.

numbered(tree(T1,pair(_,N1),T2),N0,N) :-
    numbered(T1,N0,N1),
    N2 is N1+1,
    numbered(T2,N2,N).
numbered(void,N,N).
#endif
```

.divide10.bench

```
# /*
  set-up.divide10: bench set-up for (deriv) divide10
  */
divide10 :- driver(divide10).

benchmark(divide10,
          d(DIVIDE10_EXP,x,_),
          dummy(DIVIDE10_EXP,x,_),
          1000).

show(divide10) :- d(DIVIDE10_EXP,x,D),
                  write('(d/dx)('),
                  write(DIVIDE10_EXP),
                  write(') ='), nl,
                  write(D), nl.


#include "driver"
```

.log10.bench

```
# /*
  set-up.log10: bench set-up for (deriv) log10
  */
log10 :- driver(log10).

benchmark(log10,
          d(LOG10_EXP,x,_),
          dummy(LOG10_EXP,x,_),
          1000).

show(log10) :- d(LOG10_EXP,x,D),
               write('(d/dx)('),
               write(LOG10_EXP),
               write(') ='), nl,
               write(D), nl.


#include "driver"
```

```
# /*
  set-up.ops8: bench set-up for (deriv) ops8
  */
ops8 :- driver(ops8).

benchmark(ops8,
          d(OPS8_EXP,x,_),
          dummy(OPS8_EXP,x,_),
          1000).

show(ops8) :- d(OPS8_EXP,x,D),
              write('(d/dx)('),
              write(OPS8_EXP),
              write(') ='), nl,
              write(D), nl.


#include "driver"
```

```
# /*
  set-up.times10: bench set-up for (deriv) times10
  */
times10 :- driver(times10).

benchmark(times10,
          d(TIMES10_EXP,x,_),
          dummy(TIMES10_EXP,x,_),
          1000).

show(times10) :- d(TIMES10_EXP,x,D),
                 write('(d/dx)('),
                 write(TIMES10_EXP),
                 write(') ='), nl,
                 write(D), nl.


#include "driver"
```

```
# /*
  set-up.nreverse: bench set-up for nreverse
  */
nreverse :- driver(nreverse).

benchmark(nreverse,
          nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                    13,14,15,16,17,18,19,20,21,
                    22,23,24,25,26,27,28,29,30],_),
          dummy([1,2,3,4,5,6,7,8,9,10,11,12,
                 13,14,15,16,17,18,19,20,21,
                 22,23,24,25,26,27,28,29,30],_),
          1000).

show(nreverse) :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                            13,14,15,16,17,18,19,20,21,
                            22,23,24,25,26,27,28,29,30],R),
                  write('reverse of'), nl,
                  write([1,2,3,4,5,6,7,8,9,10,11,12,
                         13,14,15,16,17,18,19,20,21,
                         22,23,24,25,26,27,28,29,30]), nl,
                  write(is), nl,
                  write(R), nl.


#include "driver"
```

.qsort.bench

```
# /*
  set-up.qsort: bench set-up for qsort
  */
qsort :- driver(qsort).

benchmark(qsort,
          qsort([27,74,17,33,94,18,46,83,65, 2,
                 32,53,28,85,99,47,28,82, 6,11,
                 55,29,39,81,90,37,10, 0,66,51,
                  7,21,85,27,31,63,75, 4,95,99,
                 11,28,61,74,18,92,40,53,59, 8],_,[]),
          dummy([27,74,17,33,94,18,46,83,65, 2,
                 32,53,28,85,99,47,28,82, 6,11,
                 55,29,39,81,90,37,10, 0,66,51,
                  7,21,85,27,31,63,75, 4,95,99,
                 11,28,61,74,18,92,40,53,59, 8],_,[]),
          1000).

show(qsort) :- qsort([27,74,17,33,94,18,46,83,65, 2,
                      32,53,28,85,99,47,28,82, 6,11,
                      55,29,39,81,90,37,10, 0,66,51,
                       7,21,85,27,31,63,75, 4,95,99,
                      11,28,61,74,18,92,40,53,59, 8],S,[]),
               write('qsort of'), nl,
               write([27,74,17,33,94,18,46,83,65, 2,
                      32,53,28,85,99,47,28,82, 6,11,
                      55,29,39,81,90,37,10, 0,66,51,
                       7,21,85,27,31,63,75, 4,95,99,
                      11,28,61,74,18,92,40,53,59, 8]), nl,
               write(is), nl,
               write(S), nl.


#include "driver"
```

```
# /*
  set-up.query: bench set-up for query
  */
query :- driver(query).

benchmark(query, run_query, dummy, 100).

show(query) :- query(X), write(X), nl, fail.
show(query).


#include "driver"
```

```
# /*
  set-up.serialise: bench set-up for serialise
  */
serialise :- driver(serialise).

benchmark(serialise,
          serialise(PALIN25,_),
          dummy(PALIN25,_),
          1000).

show(serialise) :- serialise(PALIN25,S),
                   write('serialisation of'), nl,
                   printstring(PALIN25), nl,
                   write(is), nl,
                   write(S), nl.

printstring([]).
printstring([H|T]) :- put(H), printstring(T).


#include "driver"
```

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) Contract No. N00014-88-K-0579
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) University of California Berkeley, CA 94720	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A Prolog Benchmark Suite for Aquarius

12. PERSONAL AUTHOR(S) R. Haygood

13a. TYPE OF REPORT technical	13b. TIME COVERED FROM 07/01/88 TO 11/30/90	14. DATE OF REPORT (Year, Month, Day) * April 1989	15. PAGE COUNT * 17

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes a suite of benchmarks for Prolog implementation research. It includes an explanation of the format of the suite, which is meant to facilitate use of the benchmarks. The principal idea of this format is to maintain for each benchmark a master file from which particular instances - for particular Prolog execution systems, for particular statistics to capture, etc. - are generated automatically using a preprocessor. A preprocessor provided with the suite for this purpose is described, along with a related utility and a simple framework for execution time measurement. Source code for these is appended. Possibilities for future work with respect both to this suite and to Prolog benchmarking more generally are discussed briefly. For each benchmark in the suite, source code and execution times under C Prolog and Quintus Prolog (compiled) on a Sun 3/60 are appended.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE