ICOT Technical Memorandum: TM-0451

TM-0451

TOEST: A Prolog Benchmark

by
E. Tich & K. Susaki

January, 1988

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

Institute for New Generation Computer Technology

TOESP: A Prolog Benchmark

E. Tick* and K. Susaki

Institute for New Generation Computer Technology (ICOT) †

January 26, 1988

Abstract

Extended Self-Contained Prolog (ESP) is a programming language that unites logic programming and object-oriented programming paradigms. It introduces a modular programming capability based on an object-oriented paradigm into the logic programming language Prolog. Thus the advantages of modularity are incorporated with unification and backtracking. This paper describes a Prolog-to-ESP translator written in Prolog. The purpose of this paper is two-fold. First, interesting attributes and techniques used in the translation are described. Second, the program is analyzed from the point of view of a Prolog benchmark, for use in Prolog system performance measurements.

1 Introduction

Extended Self-Contained Prolog (ESP) [2,4,1,5], is the programming language of the Personal Sequential Inference (PSI) machines [8,9,14,7,6] developed for Fifth Generation Computer System (FGCS) [3]. ESP unites logic programming and object-oriented programming paradigms. It introduces a modular programming capability based on an object-oriented paradigm into the

*also: Computer Systems Laboratory, Stanford University

†Mita-Kokusai Building 21F, 4-28 Mita 1-Chome, Minato-ku, Tokyo 108, JAPAN

logic programming language Prolog. Thus the advantages of modularity are incorporated with unification and backtracking.

ESP is described in great detail in the "ESP" Guide [4], an introduction to object-oriented programming and a programmer's guide for ESP. It has been necessary at ICOT to port many Prolog programs to the PSI machines, i.e., to translate Prolog into ESP. Since ESP is a superset of Prolog, this translation is theoretically simple, although practical considerations make it nontrivial. Some of these considerations include the complexities and differences of the standard DEC-10 Prolog and ESP semantics, and the desire to retain the *readability* of the translated programs.

The Prolog-to-ESP translator, called `toesp`, is described in this paper. The program was written by T. Takizuka of ICOT. The translation process is outlined and a simple, annotated example is given. The source code of `toesp` is also included. The translator is also discussed from a different viewpoint: that of benchmarking. `toesp` is a medium-sized Prolog benchmark (1500 source lines) that is a *real-life* program, written by a *real* programmer. No attempt was made to "clean-up" the code or optimize it in hindsight. Compared to the so-called "classic" Prolog benchmarks [13], `toesp` has both similarities and differences. Firstly, it is deterministic (the translation process is almost a one-to-one mapping). Secondly, it performs a deal of file I/O and `assert/abolish`. Thus it represents using Prolog for symbolic manipulation without heavy crunching.

For further information about ESP and object-oriented programming in logic, the reader is refered to the following works[4,2,1,5]. For further information about Prolog benchmarking, the reader is refered to the following works [11][10][12].

2 What is Object-Oriented Programming?

The essential idea of object-oriented programming is describing a given application problem in terms of things called *objects*. An object has three characteristics:

- Each object has its own characteristics and internal state. It acts independently from other objects.

- Each object can examine and change its own internal state, but can't examine and change the state of other objects.
- Each object has a function which can send and receive messages from other objects. When an object receives a message, a definite procedure is executed. Objects act only upon receiving messages.

Figure 1 gives an example of a *definition of an object*, called a *class* (taken from [4]). Objects described by this class are essentially binary trees. This program uses slots heavily and is given more as an example of ESP programming than the Prolog-to-ESP translation capability. All aspects of this program are detailed in later sections.

3 Fuctions of Object-Oriented Languages

3.1 Logic Programming and Object-Oriented Programming

An object in ESP represents an *axiom set* in logic programming. An axiom set is a group of basic logical relationships from which predicates are defined. The same predicate call may have different semantics when applied in different axiom sets. The axiom set to be used in a certain call can be specified by giving an object as the first argument of the call and preceding the call with a colon (":") to notify the set that the semantics of the call should depend on the first argument.

3.2 Class Definition

A *class* defines the characteristics common to a group of similar objects. A class can be regarded as a *module* of a program. We write a program using a form of class definition. In the example, `binary_tree` is a *class name*; however, we can view it as a module which adds and traverses a binary tree.

```
% Binary Tree: A tree node is an object and key,data,left,right
% are slots which keep the value of that node.  Objects have
% methods that represent adding and searching nodes.

class binary_tree has
% Class method that makes an instance object of binary_tree
:create(Class,Obj):-
    :new(Class,Obj) ;                 % :new/2 is a reserved method
instance                              % slots definition
attribute key := '$empty',            % Slot keeps the key of the node
      data,                           % Slot keeps the data of the node
      left,right;                     % Slots keep their left and right nodes

% :add/3 adds a new node as an object to the binary_tree
:add(Obj,Key,Data) :-                 % set the value of the node to a new
    is_empty(Obj),!,                  % object
    set_values(Key,Data,Obj) ,        % create_branch/1 creates objects and
    create_branch(Obj) ;              % set the slots (left and right)
:add(Obj,Key,Data) :-                 % Obj!key designates the slot key which
    insert(Obj!key,Key,Data,Obj) ;    % is contained by the object Obj

:get_contents(Obj,list,List) :- !,
    :get(Obj,List,[]) ;               % gets the contents of the binary_tree

:get(Obj,List,List) :-                % traverses the binary_tree and
    :is_empty (Obj), !;               % get the value of nodes
:get(Obj,List,List0) :-
    :get(Obj!left,List,List1),
    get_values(Key,Data,Obj),
    List1 = [{Key,Data}|List2],
    :get(Obj!right,List2,List0) ;

% :is_empty/1 examine if the node has any branches
:is_empty(Obj) :- !,Obj!key = '$empty' ;

local
% insert a new node to the binary_tree sorted by the key
insert(Key0,Key,Data,Obj) :- Key0 > Key, !,
    :add(Obj!right,Key,Data) ;
insert(Key0,Key,Data,Obj) :- Key0 < Key, !,
    :add(Obj!right,Key,Data) ;
insert(Key0,Key,Data,Obj) :-
    set_values(Key,Data,Obj) ;
    Obj!data := Data;

% set the value of the node (key,data) to slots
set_values(Key,Data,Obj) :- !,
    Obj!key := Key, Obj!data := Data;

% get the value from the object (key,data)
get_values(Obj!key,Obj!data,Obj) :- ! ;

% create new objects and set them to slots left and right
create_branch(Obj) :- !,
    :create(#binary_tree,Left),       % call the class method of this class
    :create(#binary_tree,Right),      % and create new objects of binary_tree
    Obj!left := Left,                 % set them to slots
    Obj!right := Right ;
end.
```

Figure 1: Binary Tree Class

3.3 Class Objects and Instance Objects

Two objects are considered *similar* when they differ only by their slot values (more about slots in a later section). When the characteristics of an object is described in a class, the object is said to be an instance of the class or an *instance object*. We can make any number of instance objects from one class. A class itself is also an object. Such an object is sometimes called a *class object* to distinguish it from usual instance objects. We use class objects to control instance objects.

Because of their uniqueness, class objects can be written down as constants using their class names. In the example, we can write `#binary_tree` as the class object of `binary_tree`. Only class objects are static objects which have this explicit notation and are directly accessible. Instance objects are always created dynamically by using the method `:new(Class,Object)` which makes a instance object of the `Class` and binds the object to the variable `Object`. The instance object is accessed via the variable to which it is bound (in this case, `Object`). The class `binary_tree` itself is defined as an object and the method `:create(Class,Object)` makes an instance object by using this method in the form of `:create(#binary_tree,Object)`. When we make an instance object it contains the part of the class definition from the keyword `instance` to `end`.

3.4 Methods and Local Predicates

Predicates are the form of executable programs of ESP. In ESP there are two kinds of predicates: *methods* and *local predicates*. A method can be called not only from within the class where the method is defined, but also from outside the class. A local predicate is a predicate that can be referenced only within the class where it is defined. In ESP, messages that are exchanged between objects are always interpreted with a method. In the example, `:create/2`, `:add/3`, `:get_contents/2`, `:get/3`, and `:is_empty/1` are methods. For instance, `:add(Object,Tree,Key,Data) :- ...` is the method definition where `add` is the method name and `Object`, `Tree`, `Key`, and `Data` are the arguments. A method is distinguished from both local and builtin predicates by the colon. There are two kinds of methods (class and instance), which area related to class and instance objects respectively. In ESP method calls, the method

called is determined dynamically depending on the first argument of the call. In other words, the method called is determined depending on the combination of its predicate name, arity, and object.

3.5 Slots

An object may have *slots*. A slot is a variable that stores the characteristics, and internal state of an object. A value can be substituted by overriding the previous value. Note that a value is not reset by backtracking. These are differences between logical variables (e.g., in Prolog) and slots. The value of slot can be examined by certain predicates using the slot names, i.e., the slot values defines a part of the axiom set. The slot values can also be changed only by certain predicate calls. This corresponds to altering the axiom set represented by the object. This mechanism is similar to `assert` and `retract` of Prolog. The difference is that only the slot values can be changed in ESP, while any axioms can be altered in Prolog. In the example, `key`, `data`, `left`, and `right` are slots. In ESP, there are two kinds of slots: *attributes* and *components*. Attribute slots are the slots that can be accessed from other classes. Component slots cannot be accessed from other classes.

3.6 Inheritance

Inheritance is the function of one class incorporating *all* the methods of another class. In ESP when a class inherits some other classes, the object of that class has all methods and slots that are defined in its inherited classes, in addition to its own methods and slots. In other words, inheritance combines axiom sets.

There are two usages of inheritance: *single* and *multiple*. Single inheritance means inheriting only one defined class during a class definition. Multiple inheritance means inheriting more than one defined class during a class definition. To use inheritance effectively, one should extract program segments that are common to different classes, and make these common segments independent inherited classes.

To take advantage of this thoroughly, it is possible to inherit multiple defined classes in ESP. In the example, the class `binary_tree` inherits no classes. But if `binary_tree` inherited

a class, we would use the keyword `nature` to define the inheritance relationship. It would be written as follows:

```
class binary_tree has
nature abc;
:create(Class,Obj):- ....
```

Here the class `binary_tree` inherits the class `abc` that is already defined.

4 How the Program Works

In this section we give a small example of the translation using the previous binary tree example. Figure 2 lists a simplified Prolog version of the program. The Prolog program is pure, i.e., slot-like facilities, using `assert` and `retract` are not implemented. Figure 3 lists the translated ESP program files. Note that in contrast to Figure 1, no vectors or slots are generated in the translated ESP program. Even if the original Prolog program used `assert` and `retract` to simulate slots, the translator does not know enough to transform them into slots.

The `toesp` Prolog-to-ESP translation program, listed in Section 6, is now described. The program uses assertion and abolishment of facts heavily. The facts refer to the Prolog predicate names that correspond to ESP class methods. During the translation, there will be one assertion and one retraction for each predicate name in the program. The reason this information is asserted in the database instead of stored in a dynamic data structure was because the author felt that assertion/retraction was less costly than other alternatives on the DEC-20. Whether this is true or not, under DEC-10 Prolog or other Prologs, the program has not been changed from its original form. For example, in the benchmark test input data, eleven Prolog source files cause 861 assertions and 108 abolishes.

Seven different types of unit facts are asserted during the translation. The major ones are: `local_CLASS_method`, `public_CLASS_method`, and `assert_CLASS_method`. These are used to categorize the different types of methods: *local* is local to a file, *public* is exported to all other files, and *assert* is a special type of local method produced by an assertion. For example if we translated the `toesp.pl` file itself, all seven types of facts would be asserted as `assert_CLASS_method` during translation.

```
%%%%%%%%%%%%%%%%%%
%      FILE 1      %
%%%%%%%%%%%%%%%%%%

:- public insert/3,traverse/3.

% binary tree represented as: tree(Key,Data,Left_tree,Right_tree)

% insert/3 adds a node to the binary tree
insert(Tree, Key, Data) :-          % create a tree
    var(Tree), !,                % if none exists
    Tree = tree(Key,Data,_,_).

insert(tree(Key0,_,Left,_), Key, Data) :-  % search left subtree
    Key0 > Key, !,
    insert(Left, Key, Data).
insert(tree(Key0,_,_,_), Key, _) :-        % already exists so fail
    Key0 =:= Key, !, fail.
insert(tree(_,_,_,Right), Key, Data) :-    % search right subtree
    insert(Right, Key, Data).

% traverse/3 converts the binary tree into a list of nodes
% Note that arguments 2 and 3 form a different list (D-list)
traverse(Tree, List, List) :-           % at leaf, close D-list
    var(Tree), !.
traverse(tree(Key,Data,Left,Right), List, List0) :-
    traverse(Left, List, List1),
    List1 = [pair(Key,Data)|List2],
    traverse(Right, List2, List0).

%%%%%%%%%%%%%%%%%%
%      FILE 2      %
%%%%%%%%%%%%%%%%%%

build(Y) :-
    insert(X,1,1),
    insert(X,2,2),
    insert(X,3,3),
    traverse(X,Y,[]).
```

Figure 2: Prolog Files for Binary Tree Example

```
%%%%%%%%%%%%%%%%%%
%     FILE 1     %
%%%%%%%%%%%%%%%%%%
class foo1 has

:insert(My_Class,Tree,Key,Data) :-
    insert(Tree,Key,Data);
:traverse(My_Class,Tree,List,A) :-
    traverse(Tree,List,A);

local

% :- public insert/3, traverse/3.

% binary tree represented as: tree(Key,Data,Left_tree,Right_tree)
% insert/3 adds a node to the binary tree

insert(Tree,Key,Data) :-     % create a tree
    unbound(Tree), !,    % if none exists
    Tree = tree(Key,Data,_,_);
insert(tree(Key0,_,Left,_),Key,Data) :-     % search left subtree
    Key0 > Key, !,
    insert(Left,Key,Data);
insert(tree(Key0,_,_,_),Key,_) :-     % already exists so fail
    Key0 =:= Key, !,
    fail;
insert(tree(_,_,_,Right),Key,Data) :-     % search right subtree
    insert(Right,Key,Data);

% traverse/3 converts the binary tree into a list of nodes
% Note that arguments 2 and 3 form a different list (D-list)

traverse(Tree,List,List) :-     % at leaf, close D-list
    unbound(Tree), !;
traverse(tree(Key,Data,Left,Right),List,List0) :-
    traverse(Left,List,List1),
    List1 = [pair(Key,Data)|List2],
    traverse(Right,List2,List0);


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %     unchangable built in predicate           %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% "var/1" is replaced to unbound(


end.

%%%%%%%%%%%%%%%%%%
%     FILE 2     %
%%%%%%%%%%%%%%%%%%

class foo2 has

local

build(Y) :-
    :insert(#foo1,X,1,1),
    :insert(#foo1,X,2,2),
    :insert(#foo1,X,3,3),
    :traverse(#foo1,X,Y,[]);

end.
```

Figure 3: Translated ESP Program Files for Binary Tree Example

The general control flow of the program is as follows. The main procedure, `to_esp(I,O)` translates an input file `I` into an output file `O`. A temporary working file, `QXKZXX.DEL`, is used during this translation. The procedure `working_file_name/3` creates this file and the procedure `temp_file/2` does a pre-translation of the input file into the temporary file. The pre-translation is used to convert the entire input file text image into valid Prolog so that variable names, comments, etc., are not lost by the translation process. Variables are converted into unique atoms and comments are packaged during this pre-translation phase. The real work of translation is performed by `new__file/5`, which calls `pass_1/2` and `pass_2/2`. The first pass abolishes any previous local and assert method facts (but keeps public facts), and makes new asserts concerning the procedures defined in the current input file. Pass one cannot be 100% accurate because in assertions such as `assert(X)` the method name cannot be determined. Note that `record`, `abolish`, and `retract` do not affect the class structure. Pass two performs the guts of the translation, doing the actual rewriting and instantiating the method calls. Two asserted facts are used: `later_WARNING_method` and `is_ALREDY_included`. `later_WARNING_method` is used to store the names of Prolog builtins (that have no ESP equivalent) found in this file. Warnings will be issued for these builtins. `is_ALREDY_included` keeps track of builtins for which conversion is possible, and the conversion methods have been written in the output file. Examples of such builtins are `functor` and `name`. The program author has chosen by design to view these builtins are local methods. Therefore, the conversion suggestions are written to *every file containing the builtins.* For example, in the benchmark test input data, there are eleven Prolog source files, eight of which use `functor`. Thus the conversion generates about 600 characters of output, to each of eight files. This occurs for each instance of a convertable builtin.

The precise translation rules are not outlined in this paper. The reader is refered to Appendix G of the ESP Guide: "Precautions for Conversion from Prolog to ESP" [4]. This appendix gives great detail about the differences in data structures, control structures, builtin predicates, arithmetic and symbolic operations. We list here a few points specific to the decisions made in the `toesp` translator.

- **class selection**—when two or more classes both export a public method with the same name and arity, the translator choses only one. The translator will choose the first method

that it finds in its asserted public method database. Thus the *order* in which the files are translated defines the strength of the classes.

- **inheritance**—the translator does not introduce inheritance into the program. In other words, no global analysis of the program structure is performed, only a local passes.
- **slots** the translator does not introduce slots into the program.
- **user library and runtime procedures**—there is no real distinction made by the translator between these types.
- **vectors**—In ESP, structures are represented as vectors, e.g., `f(A,B)` is equivalent to `{f,A,B}`. However ESP vectors are more general, and can have a variable as the first element, e.g., `{X,Y,Z}`. The translator does *not* generate generalized vectors from Prolog structures. In other words, no inter-procedural analysis is performed to optimize restricted uses of structures.

5 Benchmark Considerations

The Prolog-to-ESP translator is a good Prolog benchmark because it is a real program (i.e., it solves a real application) of significant size. We feel that the program will have different characteristics that those seen in "classic" (small) Prolog benchmarks (e.g., Warren[13]). These differences are due to the I/O intensive nature of the application, with little computation inbetween file reads and writes. In addition, `assert/abolish` are used freely to implement a database of predicate names. Thus the performance of this benchmark will be largely influenced by the performance of I/O and dynamic database operations on the system measured.

The program has been modified to run under DEC-10 Prolog, Quintus Prolog, and SICStus Prolog. In addition, it has been rigged with a benchmark entry point, `go/0`, with which to run performance measurements. This entry point translates 11 related Prolog source files corresponding to a real input data set. Thus for each file, the original will be read, a temporary file will be written, the temporary will then be read, and an output file will be written. The same temporary is used for each file. As mentioned previously, this input data set causes 861 `asserts` and 108 `abolishes`.

6 Source Program Listing

```
/* Copyright (c) 1987, 1988, Institute for New Generation Computer Technology.
   All rights reserved.
   Permission to use this program is granted for academic use only. */

%%%%%%%%%%%%%%%%%%%%%%%%% Prolog-to-ESP Translator %%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
%                      Version 0.4  (February 19 1986)                       %
%                                                                            %
%                              T. Takizuka                                   %
%            Institute for New Generation Computer Technology                %
%             4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan                 %
%                                                                            %
%              phone: +81-3-456-3194    telex: ICOT J 32964                 %
%              csnet: takizuka%icot.jp@relay.cs.net                          %
%              uucp:  {enea,inria,kddlab,mit-eddie,ukc}!icot!takizuka        %
%                                                                            %
%       Modification for outside ICOT by E. Tick, 12-Jan-88                  %
%       This program runs on DEC-10 Prolog 3.52                              %
%                               Quintus Prolog                               %
%                               SICStus Prolog V0.5                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% following declarations needed for Quintus/SICStus Prologs
:- unknown(_,fail).
:- dynamic local_CLASS_method/2,public_CLASS_method/2,assert_CLASS_method/2.
:- dynamic other_CLASS_method/2,is_ALREDY_included/2,ctype/1.
:- dynamic later_WARNING_method/2.

my_nl(10).
%my_nl(31).

eof(-1).
%eof(26).

:- mode message(-,-).
message( '
Source:    TOESP.PL       (No other modules is required)
Version:   0.4            (Last Updated:  19 Feb 1986)
Author:    ICOT
Copyright: All rights are reserved in 1985.
').

/* ET 12-7-87: top-level benchmark: ?- go. */
go :- def_other([
    'oper.pl',
    'cntr.pl',
    'comp.pl',
    'port.pl',
    'kernel.pl',
    'macro.pl',
    'prep.pl',
    'fast.pl',
    'exec.pl',
    'lib.pl',
    'buil.pl']),
    to_esp('oper.pl','oper.esp'),
    to_esp('cntr.pl','cntr.esp'),
    to_esp('comp.pl','comp.esp'),
    to_esp('port.pl','port.esp'),
    to_esp('kernel.pl','kernel.esp'),
    to_esp('macro.pl','macro.esp'),
```

```
    to_esp('prep.pl','prep.esp'),
    to_esp('fast.pl','fast.esp'),
    to_esp('exec.pl','exec.esp'),
    to_esp('lib.pl','lib.esp'),
    to_esp('buil.pl','buil.esp').

:- public myhelp/0.
myhelp :- writelist([
'    Following programs support conversion from Prolog to ESP.',
'    Input_file  : file_name or list of file_name ( in DEC-20 Prolog ).',
'    Output_file : file_name ( in ESP style language ).',
'to_esp(Input_file,Output_file)    => Conversion main program.',
'to_esp_start(Input_file)    => Create temporary file named "XXQXKZ.DEL".',
'to_esp_continue(Output_file)    => Create ESP style program.',
'def_other(Input_file) => Read public definition as other classes methods.',
'             Operator definition is also read.',
'ctype    => Integer/ASCII_code handling option setting.',
'       Invoke this predicate and answer Y or N for each query.',
'easy     => Simple user interface which calls def_other and to_esp.',
'fdel     => Delete temporary file.',
'myhelp    => Write this message.']).

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                                             %
    %      User Predicate Conversion Table        %
    %                                             %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- mode user_PREDICATE_replace(+,+,-).    % predicate name replace
user_PREDICATE_replace('Arity_0',0,':arity_0(#some_class)').    % Example
user_PREDICATE_replace('Arity_2',2,':arity_2(#some_class,').    % Example

:- mode user_FUNCTOR_prelace(+,+,-).    % functor name replace
user_FUNCTOR_prelace('',1,'').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                                             %
    %      Builtin Predicate Conversion Table     %
    %                                             %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- mode replace_functor(+,+,-).
replace_functor(abolish,2,':abolish(#User_Library,').
replace_functor(abort,0,':abort(#RunTime)').
replace_functor(ancestors,1,':ancestors(#Interpreter,').
replace_functor(arg,3,':arg(#Functor,').
replace_functor(assert,1,':assert(#User_Library,').
replace_functor(assert,2,':assert(#User_Library,').
replace_functor(asserta,1,':asserta(#User_Library,').
replace_functor(asserta,2,':asserta(#User_Library,').
replace_functor(assertz,1,':assertz(#User_Library,').
replace_functor(assertz,2,':assertz(#User_Library,').
replace_functor(bagof,3,':bagof(#RunTime,').
replace_functor(break,0,':break(#RunTime)').
replace_functor(call,1,':call(#Interpreter,').
replace_functor(clause,2,':clause(#Interpreter,').
replace_functor(clause,3,':clause(#Interpreter,').
replace_functor(close,1,':close(#Files,').
replace_functor(compare,3,':compare(#Interpreter,').
replace_functor(compile,1,':compile(#Interpreter,').
replace_functor(consult,1,':consult(#Interpreter,').
```

```
replace_functor(current_atom,1,':current_atom(#Interpreter,').
replace_functor(current_functor,2,':current_functor(#Interpreter,').
replace_functor(current_predicate,2,':current_predicate(#Interpreter,').
replace_functor(current_op,3,':current_op(#Interpreter,').
replace_functor(debug,0,':debug(#RunTime)').
replace_functor(debugging,0,':debugging(#RunTime)').
replace_functor(depth,1,':depth(#RunTime,').
replace_functor(display,1,':display(#Files,').
replace_functor(erase,1,':erase(#User_Library,').
replace_functor(expand_term,2,':expand_term(#Interpreter,').
replace_functor(fileerrors,0,':fileerrors(#Files)').
replace_functor(functor,3,':functor(#RunTime,').
replace_functor(gc,0,':gc(#RunTime)').
replace_functor(gcguide,3,':gcguide(#RunTime,').
replace_functor(get,1,':getp(#Files,').
replace_functor(get0,1,':get(#Files,').
replace_functor(halt,0,':halt(#RunTime)').
replace_functor(incore,1,':incore(#Interpreter,').
replace_functor(instance,2,':instance(#User_Library,').
replace_functor(keysort,2,':keysort(#List,').
replace_functor(leash,1,':leash(#Interpreter,').
replace_functor(length,2,':length(#List,').
replace_functor(listing,0,':listing(#Interpreter)').
replace_functor(listing,1,':listing(#Interpreter,').
replace_functor(log,0,':log(#RunTime)').

replace_functor(maxdepth,1,':maxdepth(#RunTime,').
replace_functor(name,2,':name(#Symbolizer,').
replace_functor(nl,0,':new_line(#Files)').
replace_functor(nodebug,0,':nodebug(#RunTime)').
replace_functor(nofileerrors,0,':nofileerrors(#Files)').
replace_functor(nogc,0,':nogc(#RunTime)').
replace_functor(nolog,0,':nolog(#RunTime)').
replace_functor(nonvar,1,'not(unbound(').
replace_functor(nospy,1,':nospy(#RunTime,').
replace_functor(numbervars,3,':numbervars(#List,').
replace_functor(op,3,':op(#RunTime,').
replace_functor(phrase,2,':phrase(#Interpreter,').
replace_functor(plsys,1,':plsys(#RunTime,').
replace_functor(print,1,':print(#Files,').
replace_functor(prompt,2,':prompt(#Files,').
replace_functor(put,1,':put(#Files,').
replace_functor(read,1,':read(#Files,').
replace_functor(reconsult,1,':reconsult(#Interpreter,').
replace_functor(recorda,3,':recorda(#User_Library,').
replace_functor(recorded,3,':recorded(#User_Library,').
replace_functor(recordz,3,':recordz(#User_Library,').
replace_functor(reinitialise,0,':reinitialise(#RunTime)').
replace_functor(rename,2,':rename(#Files,').
replace_functor(repeat,0,'repeat').
replace_functor(restore,1,':restore(#Interpreter,').
replace_functor(retract,1,':retract(#User_Library,').
replace_functor(revive,2,':revive(#User_Library,').
replace_functor(save,1,':save(#Interpreter,').
replace_functor(save,2,':save(#Interpreter,').
replace_functor(see,1,':see(#Files,').
replace_functor(seeing,1,':seeing(#Files,').
replace_functor(seen,0,':seen(#Files)').
replace_functor(setof,3,':setof(#RunTime,').
```

```
replace_functor(skip,1,':skip(#Files,').
replace_functor(sort,2,':sort(#List,').
replace_functor(spy,1,':spy(#RunTime,').
replace_functor(statistics,0,':statistics(#RunTime)').
replace_functor(statistics,2,':statistics(#RunTime,').
replace_functor(subgoal_of,1,':subgoal_of(#Interpreter,').
replace_functor(tab,1,':tab(#Files,').
replace_functor(tell,1,':tell(#Files,').
replace_functor(telling,1,':telling(#Files,').
replace_functor(told,0,':told(#Files)').
replace_functor(trace,0,':trace(#RunTime)').
replace_functor(trimcore,0,':trimcore(#RunTime)').
replace_functor(ttyflush,0,':ttyflush(#Files)').
replace_functor(ttyget,1,':ttyget(#Files,').
replace_functor(ttyget0,1,':ttyget0(#Files,').
replace_functor(ttynl,0,':ttymy_nl(#Files)').
replace_functor(ttyput,1,':ttyput(#Files,').
replace_functor(ttyskip,1,':ttyskip(#Files,').
replace_functor(ttytab,1,':ttytab(#Files,').
replace_functor(unknown,2,':unknown(#RunTime,').
replace_functor(var,1,'unbound(').
replace_functor(version,0,':version(#RunTime)').
replace_functor(version,1,':version(#RunTime,').
replace_functor(write,1,':write_term(#Write,').
replace_functor(writeq,1,':write_term(#Writeq,').
replace_functor(' write',1,':write_lines(#Files,').

replace_functor('C',3,':c(#DCG,').
replace_functor('LC',0,':lc(#Interpreter)').
replace_functor('NOLC',0,':nolc(#Interpreter)').

replace_functor((\+),1,'not(').
replace_functor('+',1,':unary_plus(#Arithemitic,''+'',').
replace_functor((:-),1,':do_it(#Interpreter,'':-'',').
replace_functor((?-),1,':do_it(#Interpreter,''?-'',').

replace_functor('=..',2,':func_arg(#Functor,''=..'',').

replace_functor('@<',2,':atom_compare(#Atom,''@<'',').
replace_functor('@>',2,':atom_compare(#Atom,''@>'',').
replace_functor('@=<',2,':atom_compare(#Atom,''@=<'',').
replace_functor('@>=',2,':atom_compare(#Atom,''@>='',').

replace_functor('->',2,':undefined_binary(#Interpreter,''->'',').
replace_functor('-->',2,':undefined_binary(#Interpreter,''-->'',').
replace_functor('^',2,':undefined_binary(#Interpreter,''^'',').


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                                               %
    %      Builtin Conversion Sugestion Table       %
    %                                               %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% modified ET 12-9-87 to use lists of strings to avoid string length limits

:- mode include_functor(+,+,-,-).
include_functor(name,2,[],[
  'name(Atom,List) :- atomic(Atom), !,',
  '     :get_atom_token(#symbolizer,Atom,List), !;',
  'name(Atom,List) :- unbound(Atom), !,',
  '     :enter_atom(#symbolizer,Atom,List), !;']) :- !.
```

```
include_functor(functor,3,[],[
  'functor(Functor,Name,Arity) :- unbound(Functor), !,',
  '    atomic(Name), integer(Arity), !,',
  '    ( Arity==0,          !, Functor=Name ;',
  '      Arity==2, Name==''.'',     !, Functor=[_|_] ;',
  '      Arity >0, Length is Arity+1, new_stack_vector(Functor,Length),',
  '                   first(Functor,Name) ), !;',
  'functor(Functor,Name,0) :- atomic(Functor), !, Functor=Name, !;',
  'functor([X|Y],Name,Arity)    :- !,',
  '    ( atom(Y), Y\==[], Name=Y, Arity=1,',
  '      Name=''.'', Arity=2 ), !;',
  'functor(Functor,Name,Arity) :- stack_vector(Functor,Length), !,',
  '    Arity is Length-1, first(Functor,Name), atomic(Name), !;']) :- !.

include_functor('=..',2,[length,2],[
  '''=..''(Functor,Name_List) :- unbound(Functor), !,',
  '    Name_List=[Name|Args], atomic(Name), length(Args,Arity), !,',
  '    ( Arity==0, !,              Functor=Name ;',
  '      Name==''.'', Args=[X,Y], !,    Functor=[X|Y] ;',
  '      Length is Arity+1, new_stack_vector(Functor,Length), !,',
  '        ''Assign_List_Functor''(Name_List,Functor,0,Length) ), !;',
  '''=..''(Functor,Name_List) :- atomic(Functor), !, Name_List=[Functor], !;',
  '''=..''([X|Y],Name_List) :- !,',
  '    ( atom(Y), Y\==[], Name_List=[Y,X] ;',
  '      Name_List=[''.'',X,Y] ), !;',
  '''=..''(Functor,[Name|List]) :- stack_vector(Functor,Length), !,',
  '    first(Functor,Name), atomic(Name), !,',
  '     ''Assign_List_Functor''(List,Functor,1,Length), !;',
  '''Assign_List_Functor''([],_,Length,Length) :- !;',
  '''Assign_List_Functor''([Item|List],Functor,Index,Length) :- Index < Length,',
  '    vector_element(Functor,Index,Item), Position is Index+1, !,',
  '    ''Assign_List_Functor''(List,Functor,Position,Length);']) :- !.

include_functor(arg,3,[],[
  'arg(Index,Functor,X) :- unbound(Functor), !, fail;',
  'arg(Index,[P1|P2],X) :- !,    % arg(2,f(a),X) will return X=f',
  '    ( Index==1, P1=X ; Index==2, P2=X ), !;',
  'arg(Index,Functor,X) :- stack_vector(Functor,Length), !,',
  '    0 < Index, Index < Length, first(Functor,Name), atomic(Name),',
  '    vector_element(Functor,Index,X), !;']) :- !.

include_functor(length,2,[],[
  '% not loop check',
  'length(List,_)     :- unbound(List), !, fail;',
  'length([],0)     :- !;',
  'length([_|List],Length) :- length(List,Size), Length is Size+1, !;']) :- !.

include_functor(write,1,[],[
  'write(A) :- atom(A),',
  '    :get_atom_string(#symbolizer,A,S),',
  '    :write_lines(#Window,S), !;',
  'write(T) :-',
  '    :write_term(#Window,T), !; % single quotation mark may be added']) :- !.


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                                             %
    %    Predicate & Operator Definition          %
    %                                             %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- mode key_word_in_KLO(+,+,-).
key_word_in_KLO(add,3,'Add').
```

```
key_word_in_KL0(and,3,'And').
key_word_in_KL0(apply,3,'Apply').
key_word_in_KL0(code,3,'Code').
key_word_in_KL0(complement,2,'Complement').
key_word_in_KL0(decrement,2,'Decrement').
key_word_in_KL0(divide,3,'Divide').
key_word_in_KL0(equal,2,'Equal').
key_word_in_KL0(first,2,'First').
key_word_in_KL0(hash,2,'Hash').
key_word_in_KL0(identical,2,'Identical').
key_word_in_KL0(increment,2,'Increment').
key_word_in_KL0(level,1,'Level').
key_word_in_KL0(location,1,'Location').
key_word_in_KL0(minus,2,'Minus').
key_word_in_KL0(multiply,3,'Multiply').
key_word_in_KL0(number,1,'Number').
key_word_in_KL0(not,1,'Not').
key_word_in_KL0(or,3,'Or').
key_word_in_KL0(second,2,'Second').
key_word_in_KL0(slot,3,'Slot').
key_word_in_KL0(string,3,'String').
key_word_in_KL0(structure,1,'Structure').
key_word_in_KL0(substring,4,'Substring').
key_word_in_KL0(subtract,3,'Subtract').
key_word_in_KL0(subvector,4,'Subvector').
key_word_in_KL0(succeed,1,'Succeed').
key_word_in_KL0(type,2,'Type').
key_word_in_KL0(unbound,1,'Unbound').
key_word_in_KL0(unify,2,'Unify').
key_word_in_KL0(value,2,'Value').
key_word_in_KL0(xor,3,'Xor').

:- mode built_in_KL0(+,+).
built_in_KL0(!,0).
built_in_KL0(atom,1).
built_in_KL0(atomic,1).
built_in_KL0(fail,0).
built_in_KL0(integer,1).
built_in_KL0(true,0).

:- mode key(+,-).
key(7,bell).
key(8,bs).
key(9,tab).
key(10,lf).
key(13,cr).
key(27,esc).
key(127,del).


:- mode default_atom(+,-).
default_atom(',',''',''').
default_atom([],[]).
default_atom(true,true).
default_atom(fail,fail).
default_atom(+,+).
default_atom(-,-).
default_atom(?,?).
default_atom(!,!).
default_atom((:-),'(:-)').
default_atom((?-),'(?-)').
```

```
default_atom((\+),'(\+)').
default_atom((mode),'(mode)').
default_atom((public),'(public)').
default_atom((spy),'(spy)').
default_atom((nospy),'(nospy)').

:- mode is_prefix_esp(+).
is_prefix_esp(':').
is_prefix_esp('#').
is_prefix_esp('\').
is_prefix_esp(Op) :- is_prefix(Op).

:- mode is_infix_esp(+).
is_infix_esp('#').
is_infix_esp('!').
is_infix_esp(':').
is_infix_esp(':=').
is_infix_esp('=>').
is_infix_esp(Op) :- is_infix(Op).

:- mode is_prefix(+).
is_prefix('-').

:- mode is_infix(+).
is_infix('+').
is_infix('-').
is_infix('/\').
is_infix('\/').
is_infix('*').
is_infix('/').
is_infix('<<').
is_infix('>>').
is_infix('mod').
is_infix('=').
is_infix('==').
is_infix('\==').
is_infix(Op) :- right_eval_op(Op).

:- mode right_eval_op(+).
right_eval_op(is).
right_eval_op('=:=').
right_eval_op('=\=').
right_eval_op('<').
right_eval_op('>').
right_eval_op('=<').
right_eval_op('>=').

:- mode short_pass(+,-).
short_pass(qXkZ__,'_').
short_pass(qXkZ_A,'A').
short_pass(qXkZ_B,'B').
short_pass(qXkZ_C,'C').
short_pass(qXkZ_E,'E').
short_pass(qXkZ_F,'F').
short_pass(qXkZ_G,'G').
short_pass(qXkZ_H,'H').
short_pass(qXkZ_I,'I').
short_pass(qXkZ_J,'J').
short_pass(qXkZ_K,'K').
short_pass(qXkZ_L,'L').
short_pass(qXkZ_M,'M').
short_pass(qXkZ_N,'N').
short_pass(qXkZ_O,'O').
```

```
short_pass(qXkZ_P,'P').
short_pass(qXkZ_Q,'Q').
short_pass(qXkZ_R,'R').
short_pass(qXkZ_S,'S').
short_pass(qXkZ_T,'T').
short_pass(qXkZ_U,'U').
short_pass(qXkZ_V,'V').
short_pass(qXkZ_W,'W').
short_pass(qXkZ_X,'X').
short_pass(qXkZ_Y,'Y').
short_pass(qXkZ_Z,'Z').
short_pass(qXkZ__0,'_0').
short_pass(qXkZ__1,'_1').
short_pass(qXkZ__2,'_2').
short_pass(qXkZ__3,'_3').
short_pass(qXkZ__4,'_4').
short_pass(qXkZ__5,'_5').
short_pass(qXkZ__6,'_6').
short_pass(qXkZ__7,'_7').
short_pass(qXkZ__8,'_8').
short_pass(qXkZ__9,'_9').
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      %                                               %
      %              Main Predicates                  %
      %                                               %
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- public to_esp/2.
:- mode to_esp(+,+).
to_esp(I,O) :-
    working_file_name(F,C,D),
    temp_file(I,F), !,
    new__file(F,O,T0,T1,T2),
    delete_file(F),
    statistics(runtime,[_,T3]),
    TT is T0+T1+T2+T3,
    cpu_time_write(
    [to_esp_start(I),T0,
     to_esp_continue(O),'
    pass_1',T1,
    '    pass_2',T2,
     fdel,T3,
    'Total Time of',to_esp(I,O),TT]), !,
    user_continue(C,D).

:- public ctype/0.
ctype :-
    working_file_name(_,I,O),
    abolish(ctype,1), prompt(P,''),
    y_n_in([printable,key,control]),
    prompt(_,P), !, user_continue(I,O).

:- public fdel/0.
fdel :-
    working_file_name(F,I,O),
    delete_file(F),
    statistics(runtime,[_,T3]),
    cpu_time_write([fdel,T3]), !, user_continue(I,O).

:- mode delete_file(+).
```

```
delete_file(F) :- rename(F,[]), !.
delete_file(F) :- write('Can''t delete a File '), write(F), !, nl.

:- public to_esp_start/1.
:- mode to_esp_start(+).
to_esp_start(I) :-
    working_file_name(F,C,D),
    temp_file(I,F),
    statistics(runtime,[_,T0]),
    cpu_time_write([to_esp_start(I),T0]), !,
    user_continue(C,D).

:- public to_esp_continue/1.
:- mode to_esp_continue(+).
to_esp_continue(0) :-
    working_file_name(F,C,D),
    new__file(F,0,T0,T1,T2),
    cpu_time_write([to_esp_continue(0),'
    pass_1',T1,
    '    pass_2',T2]), !,
    user_continue(C,D).

:- public def_other/1.
:- mode def_other(+).
def_other([]) :- !.
def_other(F)  :- atom(F), !, read_other_class(F).
def_other([F|I]) :- read_other_class(F), !, def_other(I).

        %%%%%%%%%%%%%%%
        %  Interface  %
        %%%%%%%%%%%%%%%

:- mode new__file(+,+,-,-,-).
new__file(F,0,T0,T1,T2) :-
    statistics(runtime,[_,T0]),
    pass_1(F,0),
    statistics(runtime,[_,T1]),
    pass_2(F,0), !,
    statistics(runtime,[_,T2]).

        %%%%%%%%%%%%%%%%%%%%%%
        %  Common Predicate  %
        %%%%%%%%%%%%%%%%%%%%%%

:- mode working_file_name(-,-,-).
working_file_name('XXQXKZ.DEL',I,0) :-
    message(C),
    statistics(runtime,_),
    seeing(I), telling(0), seen, !, told.

:- mode cpu_time_write(+,-).
cpu_time_write([]) :- !.
cpu_time_write([T|L]) :-
    ( integer(T),
        write_cpu_time(T) ;
      ( atom(T),
          write(T) ;
        write_arg_content(T) ),
      L=[I|_],
      ( integer(I), write(' needs ') ; write(' ') ) ), !,
    cpu_time_write(L).
```

```
:- mode write_cpu_time(+).
write_cpu_time(T) :-
    write(100),nl.              % for debugging ET
%    S is T / 1000, M is T mod 1000,
%    time_put(S,3600,M), !, nl.

% ET 12-11-87
safe_div(0,0,_,0) :- !.
safe_div(X,R,T,U) :- X is T / U,
             R = U.          % DEBUGGING -- ET
%             R is T mod U.

:- mode time_put(+,+,+).
time_put(0,0,M) :- ( M == 0 ; write('.'), time_d(100,M) ), write(' sec').
time_put(0,_,M) :- ( M == 0 ; write(M), write(' millisec') ).
time_put(T,U,M) :-
    safe_div(X,R,T,U), time_unit(U,D,W),
    ( X == 0 ; write(X), write(W) ), !,
    time_put(R,D,M).

:- mode time_unit(+,-,-).
time_unit(3600,60,' hour ').
time_unit(  60, 1,' min ').
time_unit(   1, 0,'').

:- mode time_d(+,+).
time_d(10,M) :- write(M).
time_d(B,M)  :-
    safe_div(D,N,M,B), write(D), R is B/10, !,
    time_d(R,N).

:- mode user_continue(+,+).
user_continue(user,user) :- !.
user_continue(user,   0) :- !, tell(0).
user_continue(   I,user) :- !, see(I).
user_continue(   I,   0) :- see(I), !, tell(0).

:- mode io_open(+,+).
io_open(I,0) :- r_open(I), !, w_open(0).

:- mode r_open(+).
r_open(F) :- fclose(F), !, see(F).

:- mode w_open(+).
w_open(F) :- fclose(F), !, tell(F).

:- mode reset(+,+).
reset(I,0) :- seen, told, fclose(I), !, fclose(0).

:- mode fclose(+).
fclose(user)    :- !.
%fclose(F)    :- close(F), !.
fclose(_)    :- !.

:- mode name_from_file(+,-).
name_from_file(0,N) :-
    name(0,L), class_name(L,X,X,C), !, name(N,C).

:- mode class_name(+,?,-,-).
class_name([],C,[],C) :- !.
class_name([46|_],C,[],C) :- !.
class_name([60|L],_,_,C) :- !, skip_to_62(L,M), !, class_name(M,X,X,C).
class_name([58|L],_,_,C) :- !, class_name(L,X,X,C).
class_name([X|L],H,[X|T],C) :- !, class_name(L,H,T,C).

:- mode skip_to_62(+,-).
```

```
skip_to_62([],[]) :- !.
skip_to_62([62|L],L) :- !.
skip_to_62([_|L],M) :- !, skip_to_62(L,M).

nl_tab :- nl, !, put(9).


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %       Pre-processor            %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- mode temp_file(+,+).
temp_file(I,0) :-
    w_open(0),
    listed_input_file(I), told,
    fclose(0).

:- mode listed_input_file(+).
listed_input_file([])     :- !.
listed_input_file(F)      :- atom(F), !, do_rep_term(F).
listed_input_file([F|I]) :- do_rep_term(F), !, listed_input_file(I).

:- mode do_rep_term(+).
do_rep_term(F) :- r_open(F), rep_term, seen, !, fclose(F).

rep_term :- cvt_statement(0,0).
rep_term :- !.

:- mode cvt_statement(+,+).
cvt_statement(S,W) :- cvt_step(S,W,R,E), !, cvt_statement(R,E).

:- mode cvt_step(+,+,-,-).
cvt_step(0,W,R,E) :-
    get0(C), !, comment_manager(C,W,R,E).
cvt_step(1,W,R,E) :-
    get0(C), i_c(C,F), !, new_term(F,C,W,R,E).


:- mode comment_manager(+,+,-,-).
comment_manager(X,_,_,_) :- eof(X),!, fail.
comment_manager(46,_,0,1) :- !, put(46).
comment_manager(37,W,0,0) :- !, unique_comment(W,1).
comment_manager(47,W,R,E) :- get0(C), !, comment_skip(C,1,W,R,E).
comment_manager( C,_,R,E) :- C > 32, i_c(C,F), !, new_term(F,C,1,R,E).
comment_manager( C,W,R,E) :- comment_with_my_nl(C,W,U,N), !,
    comment_manager(N,U,R,E).

:- mode i_c(+,-).
i_c(  X,   _) :- eof(X),!, fail.
i_c(Code,Type) :- arg(Code,ascii(
%        ^A, ^B, ^C, ^D, ^E, ^F, ^G, ^H, ^I, ^J, ^K, ^L, ^M, ^N, ^O,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
%    ^P, ^Q, ^R, ^S, ^T, ^U, ^V, ^W, ^X, ^Y, ^Z, ^[, ^\, ^], ^^, ^_,
     1,  1,  1,  1,  1,  1,  1,  1,  1,  1, 99,  1,  1,  1,  1,  1,
%      ,  !,  ",  #,  $,  %,  &,  ',  (,  ),  *,  +,  ,,  -,  .,  /,
     1,  1,  2,  1,  1,  3,  1,  2,  1,  1,  1,  1,  1,  1,  0,  4,
%     0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  :,  ;,  <,  =,  >,  ?,
     5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  1,  1,  1,  1,  1,  1,
%     @,  A,  B,  C,  D,  E,  F,  G,  H,  I,  J,  K,  L,  M,  N,  O,
     1,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,
%     P,  Q,  R,  S,  T,  U,  V,  W,  X,  Y,  Z,  [,  \,  ],  ^,  _,
     6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  1,  1,  1,  1,  6,
%     `,  a,  b,  c,  d,  e,  f,  g,  h,  i,  j,  k,  l,  m,  n,  o,
     1,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,
%     p,  q,  r,  s,  t,  u,  v,  w,  x,  y,  z,  {,  |,  },  ~,   ,
```

```
     5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  1,  1,  1,  1,  1 ),
    Type), !.

:- mode new_term(+,+,+,-,-).
new_term(0,C,_,0,1) :- !, put(C).
new_term(2,C,_,1,1) :- put(C), get0(N),    !, skip_to_cc(N,C).
new_term(3,C,W,1,1) :- !, unique_comment(W,0).
new_term(4,_,W,R,E) :- get0(N), !, comment_skip(N,0,W,R,E).
new_term(F,C,_,R,E) :- new_do_type(F,C,X,T), i_c(X,Y), !, new_term(Y,X,T,R,E).

:- mode unique_comment(+,+).
unique_comment(W,S) :-
    comment_func_name(W,C), !,
    my_nl(X),comment_maker(C,X,S),
    comment_func_end(S), !, nl.

:- mode comment_skip(+,+,+,-,-).
comment_skip(42,S,W,R,1) :- R is 1-S,
    comment_func_name(W,C), !, comment_cont(C,S).
comment_skip( C,_,W,R,E) :-
    put(47), i_c(C,F), !,
    new_term(F,C,W,R,E).

:- mode comment_with_my_nl(+,+,-,-).
comment_with_my_nl(X,_,0,N) :- my_nl(X), nl, !, get0(N).
comment_with_my_nl( C,W,W,N) :- put(C),  !, get0(N).

:- mode comment_func_name(+,-).
comment_func_name(V,C) :- write('qXkZ_('), write(V), write(','''), !, get0(C).

:- mode comment_maker(+,+,+).
comment_maker(X,_,_):- eof(X),!, fail.
comment_maker(N,N,_) :- !.
comment_maker(G,N,S) :-
    ( my_nl(G),
        comment_func_end(S), nl,
        comment_func_name(1,C) ;
      G==39,
        write(''''''), get0(C) ;
      put(G), get0(C) ), !,
    comment_maker(C,N,S).

:- mode comment_func_end(+).
comment_func_end(0) :- !, write('''), ').
comment_func_end(1) :- !, write(''').').

:- mode comment_cont(+,+).
comment_cont(C,S) :- !,
    comment_maker(C,42,S), get0(F), !,
    comment_ender(F,S).

:- mode comment_ender(+,+).
comment_ender(47,S) :- !, comment_func_end(S).
comment_ender(C,S)  :- put(42), !, comment_cont(C,S).

:- mode skip_to_cc(+,+).
skip_to_cc(C,C) :- !, put(C).
skip_to_cc(P,C) :- put(P), get0(N), !, skip_to_cc(N,C).

:- mode new_do_type(+,+,-,-).
new_do_type(1,Y,X,0) :- my_nl(Y), nl, !, get0(X).
new_do_type(1, C,X,1) :- put( C), get0(X), my_nl(E),
    ( C\==44 ; X\==E ; write(' qXkZ_,') ), !.
new_do_type(5, C,X,1) :- put( C), get0(N), !, f_term(N,X).
```

```
new_do_type(6, C,X,1) :- write('qXkZ_'), put(C), get0(N), !, f_term(N,X).

:- mode f_term(+,-).
f_term(N,N) :- i_c(N,F), F < 5, !.
f_term(N,C) :- put(N), get0(T), !, f_term(T,C).


        %%%%%%%%%%%%%%%%%%%%
        %  Pass 1 Routine  %
        %%%%%%%%%%%%%%%%%%%%


:- mode pass_1(+,+).
pass_1(I,O) :-
    io_open(I,O), !, class_header(I,O).

:- mode class_header(+,+).
class_header(I,O) :-
    abolish(local_CLASS_method,2),
    abolish(public_CLASS_method,2),
    abolish(assert_CLASS_method,2),
    name_from_file(O,N),
    write('class '), write(N), write(' has'), nl,
    put_public, nl, nl,
    seen, !, r_open(I).

put_public :- read_public.
put_public :- public_CLASS_method(A,N), is_assed(A,N).
put_public :-
    abolish(public_CLASS_method,2), !,
    abolish(assert_CLASS_method,2).

read_public :- read(T), !, pub_action(T), !, read_public.

:- mode is_assed(+,+).
is_assed(N,A)    :- assert_CLASS_method(N,A), !, fail.
is_assed(N,A)    :- nl,
      write('% class method '), writeq(N), write('/'), write(A),
      write(' isn''t found.'), !, fail.

:- mode pub_action(+).
pub_action(end_of_file)    :- !, fail.
pub_action(P) :- pub_class(P), !.
pub_action(_) :- !.

:- mode pub_class(+).
pub_class((:- D))    :- !, enter_public(D).
pub_class((T :- D))    :- !, pub_checker(T).
pub_class((T --> D))    :- !, expand_pub(T,E), pub_checker(E).
pub_class(qXkZ_(_,_))    :- !.
pub_class(T)        :- !, pub_checker(T).

:- mode enter_public(+).
enter_public((public N)) :- !, to_pub_def(N).
enter_public(op(A,B,C))  :- !, op(A,B,C).
enter_public(_) :- !.

:- mode pub_checker(+).
pub_checker(T) :-
    functor(T,N,CA), count_arg(0,CA,T,0,A), !,
    global_method(N,A),      assert(local_CLASS_method(N,A)), !,
    public_CLASS_method(N,A), assert(assert_CLASS_method(N,A)), !, nl,
    write(':'), writeq(N), !,
    ( A==0,
        write('(My_Class) :-'), nl_tab,
```

```
        func_name_writer(N,0), write(';') ;
      functor(L,$,A),
        sum_up_arg(0,A,T,L),
        write_class_arg(0,A,L,'(My_Class,',65), write(') :-'),
        nl_tab, func_name_writer(N,A), !,
        write_class_arg(0,A,L,'(',65), write(');') ).

:- mode expand_pub(+,-).
expand_pub(T,E) :-
    functor(T,N,S), X is S+1, Y is S+2,
    functor(E,N,Y), arg(X,E,qXkZ__), arg(Y,E,qXkZ__),
    expand_arg(0,S,T,E).

:- mode to_pub_def(+).
to_pub_def((P,N)) :- ass_pub(P), !, to_pub_def(N).
to_pub_def(P)     :- !, ass_pub(P).

:- mode ass_pub(+).
ass_pub(N/A) :- public_CLASS_method(N,A),
      write('% class method '), writeq(N), write('/'), write(A),
      write(' is duplicated difined.'), !, nl.
ass_pub(N/A) :- !, assert(public_CLASS_method(N,A)).
ass_pub(_)   :- !.

:- mode global_method(+,+).
global_method(N,A) :- local_CLASS_method(N,A), !, fail.
global_method(_,_) :- !.

:- mode sum_up_arg(+,+,+,+).
sum_up_arg(A,A,T,L) :- !.
sum_up_arg(I,A,T,L) :- J is I+1, arg(J,T,X),
    give_name_to_arg(X,J,L), !,
    sum_up_arg(J,A,T,L).

:- mode write_class_arg(+,+,+,+,+,+).
write_class_arg(A,A,_,_,_,_).
write_class_arg(I,A,S,P,N) :- J is I+1, arg(J,S,E),
    ( nonvar(E), M=N ; write_one_class_arg(A,S,E,N,M) ),
    ( E==qXkZ_, Q=P ; write(P), write(E), Q=',', !,
    write_class_arg(J,A,S,Q,M).

:- mode give_name_to_arg(+,+,+).
give_name_to_arg(X,I,L) :- alow_var(X,I,L).
give_name_to_arg(_,_,_).

:- mode alow_var(+,+,+).
alow_var(qXkZ_,I,L) :- arg(I,L,qXkZ_).
alow_var(X,I,L) :- atom(X), X \== qXkZ__, !,
    ( short_pass(X,N) ; name(X,[113,88,107,90,95|T]), name(N,T) ), !,
    new_face(1,I,L,N), arg(I,L,N).
alow_var(qXkZ_(_,_),I,L) :- arg(I,L,qXkZ_).

:- mode new_face(+,+,+,+).
new_face(J,J,_,_).
new_face(I,J,L,N) :- arg(I,L,Y), !, Y \== N, K is I+1, !,
    new_face(K,J,L,N).

:- mode write_one_class_arg(+,+,+,+,+).
write_one_class_arg(A,S,E,N,M) :-
    name(Z,[N]), yet_used(0,A,S,Z), E=Z, M is N+1.
write_one_class_arg(A,S,E,N,M) :- R is N+1, !,
    write_one_class_arg(A,S,E,R,M).
```

```
:- mode yet_used(+,+,+,+).
yet_used(A,A,_,_).
yet_used(I,A,S,E) :- J is I+1, arg(J,S,X), !, E \== X, !,
    yet_used(J,A,S,E).

:- mode expand_arg(+,+,+,+).
expand_arg(S,S,_,_).
expand_arg(I,S,T,E) :- J is I+1, arg(J,T,X), arg(J,E,X), !,
    expand_arg(J,S,T,E).

        %%%%%%%%%%%
        %  Pass 2  %
        %%%%%%%%%%%

:- mode pass_2(+,+).
pass_2(I,0) :-
    cvt_form, !, reset(I,0).

cvt_form :-
    write(local), nl, nl,
    abolish(later_WARNING_method,2),
    abolish(is_ALREDY_included,2),
    ( cvt_form_rep(0) ; true ),
    later_warning,
    abolish(local_CLASS_method,2),
    nl, write('end.'), nl, !, nl.

:- mode cvt_form_rep(+).
cvt_form_rep(N) :-
    read(T),
    classify_statement(T,N,M), !,
    cvt_form_rep(M).

later_warning :- later_WARNING_method(F,A), !, later_warning_out.
later_warning :- !.

:- mode classify_statement(+,+,-).
classify_statement(end_of_file,_,_) :- nl, !, fail.
classify_statement(qXkZ_(F,CS),N,M) :-
    atom(CS), ( F==0 ; F==1 ),
    comment_my_nl(F,N,CS,M), !.
classify_statement(( :- T),N,0) :- !,
    nl_put(N,0), !,
    declaration(T).
classify_statement((H :- B),N,M) :- !,
    nl_put(N,M,H),
    write_arg_content(H), write(' :-'),
    state_test(B),
    write_mini_body(B,0,_), !, write(';').
classify_statement(T,N,M) :- T='-->'(_,_), !,
    expand_term(T,S),
    de_expand_comment(S,R), !,
    classify_statement(R,N,M).
classify_statement(H,N,M) :- !,
    nl_put(N,M,H),
    write_arg_content(H), !, write(';').

:- mode comment_my_nl(+,+,+,-).
comment_my_nl(_,qXkZ_,CS,qXkZ_) :- nl, write('%'), !, write(CS).
comment_my_nl(0,_,CS,qXkZ_) :- nl, nl, write('%'), !, write(CS).
comment_my_nl(1,N,CS,N)     :- put(9), write('%'), !, write(CS).

:- mode nl_put(+,+).
```

```
nl_put(0,_) :- nl.
nl_put(N,N) :- nl.
nl_put(_,_) :- nl, nl.

:- mode declaration(+).
declaration((public D)) :-
    write('% :- public '), !, write_pubs(D).
declaration((mode D)) :-
    write('% :- mode '), !, write_modes(D).
declaration(T) :-
    write('% :- '), write_a_functor(T), !, write('.').

:- mode nl_put(+,-,+).
nl_put(N,H,H) :- atomic(H), !, nl_put(N,H).
nl_put(N,M,H) :- functor(H,M,_), !, nl_put(N,M).

:- mode write_arg_content(+).
write_arg_content(F) :- atomic_writer(F), !.
write_arg_content(F) :- functor(F,N,A), !, mk_func_class(N,A,F).

:- mode write_arg_content(+,+).
write_arg_content(I,F) :- arg(I,F,X), !, write_arg_content(X).

:- mode state_test(+).
state_test((A,B))    :- ( state1_test(A,B), write(' ') ; nl_tab ), !.
state_test((_;_))    :- !, nl_tab.
state_test(_)        :- !, write(' ').

:- mode write_mini_body(+,+,-).
write_mini_body(!,_,1)    :- !,
    write(!).
write_mini_body((!,B),_,R) :-
    write('!,'),
    ( B=(qXkZ_,C) ;
      B=(qXkZ_(F,CS),C),
        comment_out(F,CS) ;
      B=C ), nl_tab,
    write_mini_body(C,0,R).
write_mini_body((X,B),S,R) :-
    write_body_2(X,B,S,T),
    write_mini_body(B,T,R).
write_mini_body(X,S,R)    :- !,
    check_or(X,S,R).

:- mode de_expand_comment(+,-).
de_expand_comment(H,R) :- rm_cmt(H,R), !, give_expand_name(R,0,_).

:- mode rm_cmt(+,-).
rm_cmt(H,H) :- ( var(H) ; atomic(H) ).
rm_cmt(qXkZ_(X,X),qXkZ_).
rm_cmt(qXkZ_(F,S,X,X),qXkZ_(F,S)).
rm_cmt(H,R) :- functor(H,N,A), functor(R,N,A), ea_fa_rm(0,A,H,R).

:- mode give_expand_name(+,+,-).
give_expand_name(S,N,M) :- var(S), M is N+1, arg(M,
    dcg_name(qXkZ_DCG_0,qXkZ_DCG_1,qXkZ_DCG_2,qXkZ_DCG_3,qXkZ_DCG_4,
    qXkZ_DCG_5,qXkZ_DCG_6,qXkZ_DCG_7,qXkZ_DCG_8,qXkZ_DCG_9,
    qXkZ_DCG_A,qXkZ_DCG_B,qXkZ_DCG_C,qXkZ_DCG_D,qXkZ_DCG_E,
    qXkZ_DCG_F,qXkZ_DCG_G,qXkZ_DCG_H,qXkZ_DCG_I,qXkZ_DCG_J,
    qXkZ_DCG_K,qXkZ_DCG_L,qXkZ_DCG_M,qXkZ_DCG_N,qXkZ_DCG_O,
    qXkZ_DCG_P,qXkZ_DCG_Q,qXkZ_DCG_R,qXkZ_DCG_S,qXkZ_DCG_T,
    qXkZ_DCG_U,qXkZ_DCG_V,qXkZ_DCG_W,qXkZ_DCG_X,qXkZ_DCG_Y,qXkZ_DCG_Z),S).
give_expand_name(S,N,M) :-
    functor(S,_,A), ea_gi_nm(0,A,S,N,M).
```

```
:- mode ea_fa_rm(+,+,+,+).
ea_fa_rm(A,A,_,_).
ea_fa_rm(I,A,H,R) :- J is I+1, arg(J,H,X), arg(J,R,Y), rm_cmt(X,Y), !,
    ea_fa_rm(J,A,H,R).

:- mode ea_gi_nm(+,+,+,+,-).
ea_gi_nm(A,A,_,N,N).
ea_gi_nm(I,A,S,N,M) :- J is I+1, arg(J,S,T), give_expand_name(T,N,K), !,
    ea_gi_nm(J,A,S,K,M).

:- mode write_pubs(+).
write_pubs((P,D)) :- put_pub_def(P,', '), !, write_pubs(D).
write_pubs(P)     :- !, put_pub_def(P,'.').

:- mode write_modes(+).
write_modes((M,D)) :- put_mode_def(M,'), '), !, write_modes(D).
write_modes(M)     :- !, put_mode_def(M,').').

:- mode write_a_functor(+).
write_a_functor(F) :- atomic_writer(F), !.
write_a_functor(F) :- functor(F,N,A), !, func_class(N,A,F).

:- mode write_a_functor(+,+).
write_a_functor(I,F) :- arg(I,F,X), write_a_functor(X).

:- mode put_pub_def(+,+).
put_pub_def(N/A,T):- writeq(N), write('/'), write(A), !, write(T).
put_pub_def(P,_)  :- com_in_com(P), !.
put_pub_def(P,T)  :- write(P), write(T), !,
    write('<= illeagal public definition').

:- mode com_in_com(+).
com_in_com(qXkZ_) :- !, write('
%    ').
com_in_com(qXkZ_(F,S)) :- atom(S), ( F==0 ; F==1 ),
    write('    %'), write(S), !, write('
%    ').

:- mode put_mode_def(+,+).
put_mode_def(M,_)  :- com_in_com(M), !.
put_mode_def(M,T)  :- functor(M,N,L), count_arg(0,L,M,0,A),
    writeq(N), write('('), write_mode_arg(0,L,M), write(T),
    ( local_CLASS_method(N,A) ;
      write(' <= method '), writeq(N), write('/'), write(A),
      write(' isn''t defined.'), nl, write('%'), put(9) ), !.

:- mode write_mode_arg(+,+,+).
write_mode_arg(L,L,_).
write_mode_arg(I,L,F) :- J is I+1, arg(J,F,M),
    ( ( M=='+' ; M=='-' ; M=='?' ), write(M),
      ( J==L ; write(',') ) ;
      com_in_com(M) ;
      write_a_functor(M), write(' <= illeagal mode,
%    ') ), !,
    write_mode_arg(J,L,F).

:- mode atomic_writer(+).
atomic_writer(F) :- atomic(F),
    ( integer(F),
        asc_trans(F) ;
      default_atom(F,R),
        write(R) ;
      func_name_writer(F,0) ), !.
atomic_writer(qXkZ_(F,CS)) :-
```

```
    comment_out(F,CS), !, nl_tab.
atomic_writer(L) :- L=[_|_],
    write('['), is_string(L), !, write(']').
atomic_writer((X,L)) :-
    write('('),  write_arg_content(X), write(','), !,
    put_para(L), !,
    write(')').
atomic_writer({X}) :-
    write('{'), put_para(X), !, write('}').

:- mode func_class(+,+,+).
func_class(N,1,F) :- is_prefix(N),
    write(N), !,
    write_a_functor(1,F).
func_class(N,2,F) :- is_infix(N), !,
    write_in_functor(N,F).
func_class(N,A,F) :-
    func_name_writer(N,A), !,
    put_args(0,A,'(',F).

:- mode asc_trans(+).
asc_trans(F) :- classify_asc_type(F), !.
asc_trans(F) :- write(F).

:- mode func_name_writer(+,+).
func_name_writer(N,A) :-
    key_word_in_KL0(N,A,R), !, writeq(R).
func_name_writer(N,A) :-
    user_FUNCTOR_prelace(N,A,R), !, write(R).
func_name_writer(N,_) :-
    N @>= qXkZ_A, N @< qXkZ_a,
    ( short_pass(N,X), write(X) ;
      name(N,[113,88,107,90,95,X|List]), put(X), list_put(List) ), !.
func_name_writer(N,0) :-
    N @>= ' ', N @< '!', ( N==' ' ; N=='  ' ; name(N,L), all_blank(L) ),
    writeq(N), !, write('    % <= **** this atom is made of all blanks ****
    ').
func_name_writer(N,_) :-
    writeq(N).

:- mode comment_out(+,+).
comment_out(0,CS) :- atom(CS), nl,    write('%'), !, write(CS).
comment_out(1,CS) :- atom(CS), put(9),    write('%'), !, write(CS).

:- mode is_string(+).
is_string(F) :- all_ascii(F), !, list_as_string(F).
is_string(F) :- put_list(F,'').

:- mode put_para(+).
put_para((X,L)) :-
    ( X==qXkZ_, nl_tab ;
      X= qXkZ_(F,CS), comment_out(1,CS), nl_tab ;
      write_arg_content(X), write(',') ), !,
    put_para(L).
put_para(L)    :-
    write_arg_content(L).

:- mode classify_asc_type(+).
classify_asc_type(C) :- key(C,K), !, ctype(key),
    write('key#'), !, write(K).
classify_asc_type(C) :- C < 32, !, C >= 0, ctype(control),
    write('control#"'), I is C+64, put(I), !, write('"').
classify_asc_type(C) :-  C < 127, !, ctype(printable),
    write('#"'), put(C), !, write('"').
```

```
:- mode list_put(+).
list_put([]) :- !.
list_put([X|List]) :- put(X), !, list_put(List).

:- mode all_blank(+).
all_blank([]) :- !.
all_blank([32|L]) :- all_blank(L).

:- mode all_ascii(+).
all_ascii([]) :- !.
all_ascii([X|L]) :- is_ascii(X), !, all_ascii(L).

:- mode list_as_string(+).
list_as_string([X])   :- !, char_const(X).
list_as_string([X|L]) :- char_const(X), write(','), !,
    list_as_string(L).

:- mode put_list(+,+).
put_list([],_)     :- !.
put_list([X|L],D):- skip_comlist(X,L,D,M,E), !,
    put_list(M,E).
put_list(L,_)     :-
    write('|'), !, write_arg_content(L).

:- mode is_ascii(+).
is_ascii(X) :- integer(X), X > 31, X < 127.

:- mode char_const(+).
char_const(X) :- write('#"'), put(X), !, write('"').

:- mode skip_comlist(+,+,+,-,-).
skip_comlist(qXkZ_(_,CS),[X|L],D,M,E) :-
    comment_out(1,CS), nl_tab, !, skip_comlist(X,L,D,M,E).
skip_comlist(qXkZ_,L,D,L,'') :-
    write(D), !, nl_tab.
skip_comlist(X,L,D,L,',') :-
    write(D), !, write_arg_content(X).

:- mode mk_func_class(+,+,+).
mk_func_class((:-),2,F) :-
    write('('), write_arg_content(1,F), write(' :- '),
    write_arg_content(2,F), !, write(')').
mk_func_class(N,1,F) :- is_prefix_esp(N),
    write('`('), write(N),
    write_arg_content(1,F), !, write(')').
mk_func_class(N,2,F) :- is_infix_esp(N),
    write('`('), write_arg_content(1,F), write(' '), write(N), write(' '),
    write_arg_content(2,F), write(')'),
    ( N\==';' ; write(' % *** in Prolog ''(a|b) => (a;b)'' when read ***
    ') ), !.
mk_func_class(N,A,F) :-
    func_name_writer(N,A), !,
    put_args(0,A,'(',F).

:- mode put_args(+,+,+,+).
put_args(A,A,_,_) :- !, write(')').
put_args(I,A,P,F) :- J is I+1, arg(J,F,X),
    write(P), args_com(X,T), !,
    put_args(J,A,T,F).

:- mode args_com(+,-).
args_com(qXkZ_,'') :-        !, nl_tab.
args_com(qXkZ_(F,CS),'') :-    comment_out(F,CS), !, nl_tab.
```

```
args_com(X,',') :-        write_arg_content(X).

:- mode write_in_functor(+,+).
write_in_functor(N,F) :- current_op(P,yfx,N), !,
    div_op_2(N,P,F).
write_in_functor(N,F) :- right_eval_op(N),
    write_arg_content(1,F),
    write(' '), write(N), write(' '), !,
    write_a_functor(2,F).
write_in_functor(N,F) :-
    write_arg_content(1,F),
    write(' '), write(N), write(' '), !,
    write_arg_content(2,F).

:- mode div_op_2(+,+,+).
div_op_2(N,P,F) :-
    arg(1,F,X),
    op_priority(X,P), !,
    after_op(N,P,F).
div_op_2(N,P,F) :-
    write_a_functor(1,F), !,
    after_op(N,P,F).

:- mode op_priority(+,+).
op_priority(X,P) :-
    functor(X,M,2), is_infix(M),
    current_op(Q,yfx,M), Q > P, !,
    write('('), div_op_2(M,Q,X), !, write(')').

:- mode after_op(+,+,+).
after_op(N,P,F) :-
    write(' '), write(N), arg(2,F,Y),
    ( op_priority(Y,P) ;
      write(' '), write_a_functor(Y) ), !.

:- mode state1_test(+,+).
state1_test(!,_).
state1_test(';'(_,_),_) :- !, fail.
state1_test(_,!).
state1_test(_,(!,_)).
state1_test(qXkZ_(1,_),_).

:- mode write_body_2(+,+,+,-).
write_body_2(qXkZ_,B,_,0) :-
    ( B=='!' ; B=(!,_) ; nl_tab ), !.
write_body_2(qXkZ_(F,CS),_,_,0) :-
    comment_out(F,CS), !, nl_tab.
write_body_2(X,_,S,T) :-
    check_or(X,S,T), !, write(', ').

:- mode check_or(+,+,-).
check_or(';'(B1,B2),S,2) :- !,
    ( S < 1 ; nl_tab ),
    write('( '),
    put_in_or_body(B1,0,_), write(' ;'),
    ( B2=(qXkZ_(F,CS),C), comment_out(F,CS) ;
      B2=C ), write('
      '), !,
    write_or_body(C).
check_or(X,S,1) :-
    ( S < 2 ; nl_tab ),
    ( is_var_call(X) ;
      functor(X,F,A),  count_arg(0,A,X,0,M), replacable(X,F,A,M) ), !.
```

```
:- mode put_in_or_body(+,+,-).
put_in_or_body(!,_,1)    :-
    write(!), !, to_esp_warning.
put_in_or_body((!,B),_,R) :-
    write('!,'), to_esp_warning,
    ( B=(qXkZ_,C) ;
      B=(qXkZ_(_,CS),C),
        comment_out(1,CS), write('
      ') ;
      B=C ), !,
    put_in_or_body(C,0,R).
put_in_or_body((X,B),S,R) :-
    write_body_2(X,B,S,T), !,
    put_in_or_body(B,T,R).
put_in_or_body(X,S,R)    :-
    check_or(X,S,R).

:- mode write_or_body(+).
write_or_body(';'(B1,B2)) :-
    put_in_or_body(B1,0,_),
    write(' ;'), nl_tab, write('  '), !,
    write_or_body(B2).
write_or_body(X) :-
    put_in_or_body(X,0,_), !, write(' )').

:- mode is_var_call(+).
is_var_call(X) :- atom(X), X @>= qXkZ_A, X @< qXkZ_a, !,
    replacable(call(X),call,1,1).

:- mode replacable(+,+,+,+).
replacable(X,',',2,_) :-
    nl_tab, write('( '),
    write_mini_body(X,0,_), !,
    write(' )').
replacable(X,F,A,M) :- built_in_KLO(F,M), write(F),
    ( A==0 ; write('('), write_func_arg(A,X) ), !.
replacable(X,F,A,M) :- user_PREDICATE_replace(F,M,R), !,
    write(R), !, write_func_arg(A,X).
replacable(X,F,A,M) :- replace_functor(F,M,R), !,
    registrate_replace(F,A,M,R,X).
replacable(X,F,A,M) :- undefined_method(F,M), !,
    write_func_arg(A,X).
replacable((X = [qXkZ__|qXkZ__]),_,_,_) :- atom(X),
    atomic_writer(X), !,
    write(' = [_|_]    % *** [_|_] is also unifiable with "f(x)" ***
    ').
replacable(X,_,_,_) :- write_a_functor(X).

:- mode count_arg(+,+,+,+,-).
count_arg(L,L,_,A,A) :- !.
count_arg(I,L,F,X,A) :- J is I+1,
    arg(J,F,C), ( functor(C,qXkZ_,_), Y=X ; Y is X+1 ), !,
    count_arg(J,L,F,Y,A).

to_esp_warning :-
    write('    % In ESP Warning ''**** Cut in OR block ****''
      ').

:- mode write_func_arg(+,+).
write_func_arg(0,_) :- !.
write_func_arg(_,X) :-
```

```
    functor(X,_,L), !, put_args(0,L,'',X).

:- mode registrate_replace(+,+,+,+,+).
registrate_replace(write,1,1,_,write(T)) :- real_atom(T),
    write(':write_lines(#Files,string#"'), write(T), write('")'), !,
    later_assert(' write',1).
registrate_replace(\+,1,1,R,\+X) :-
    write(R), ( X=(_,_), write('('), D='))' ; D=')' ),
    write_mini_body(X,0,_), write(D), !,
    later_assert(\+,1).
registrate_replace(F,A,M,R,X) :-
    write(R), write_func_arg(A,X),
    ( F \== nonvar ; write(')') ), !,
    later_assert(F,M).

:- mode undefined_method(+,+,+).
undefined_method(F,1) :- is_prefix(F), !, fail.
undefined_method(F,2) :- is_infix(F), !, fail.
undefined_method(F,A) :- local_CLASS_method(F,A), !, fail.
undefined_method(F,A) :- other_CLASS_method(F,A,O), !,
    put_other_class_name(F,A,O).
undefined_method(F,A) :-
    put_other_class_name(F,A,'Other_User_Class').

:- mode later_assert(+,+).
later_assert(F,N) :- later_WARNING_method(F,N).
later_assert(F,N) :- assert(later_WARNING_method(F,N)).

:- mode real_atom(+).
real_atom(T) :- integer(T).
real_atom(T) :- atom(T), !, ( T @< qXkZ_A ; T @>= qXkZ_a ).

:- mode put_other_class_name(+,+,+).
put_other_class_name(F,A,O) :-
    write(':'), func_name_writer(F,A),
    write('(#'), write(O),
    ( A==0, write(')') ; write(',') ), !.

later_warning_out :- write('

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %      unchangable built in predicate        %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

'),
    later_WARNING_method(F,A), warning_out(F,A).
later_warning_out :-
    have_knowledge,
    later_WARNING_method(F,A), include_functor(F,A,O,M),
    include_it(F,A,O,M).
later_warning_out :- nl,
    abolish(is_ALREDY_included,2),
    abolish(later_WARNING_method,2).

:- mode warning_out(+,+).
warning_out(F,A) :- write('% "'), writeq(F), write('/'), write(A),
    write('" is replaced to '), replace_functor(F,A,R),
    write(R), nl, !, fail.

have_knowledge :-
    later_WARNING_method(F,A), include_functor(F,A,_,_), !,
    write('
%    ***** Sample of Builtin Predicate Conversion *****
').
```

```
:- mode include_it(+,+,+,+).
include_it(F,A,O,M) :- \+is_ALREDY_included(F,A),
    assert(is_ALREDY_included(F,A)),
    writelist(M), nl, nl, !, additional_need(O).

:- mode additional_need(+).
additional_need([F,A|L]) :-
    include_functor(F,A,O,M), \+include_it(F,A,O,M), !,
    additional_need(L).

writelist([]) :- !.                % included ET 12-9-87
writelist([H|T]) :- write(H),nl,writelist(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %%%%%%%%%%%%%%%%%%%%%%%%
        %   Utility Programs   %
        %%%%%%%%%%%%%%%%%%%%%%%%

% For ctype

:- mode y_n_in(+).
y_n_in([]) :- !.
y_n_in([X|L]) :-
    write('An integer of "'), write(X),
    write('" character value is to be translated (y/n) ? '),
    ttyflush, read_line(Y),
    ( Y\==121 ; assert(ctype(X)) ), !, y_n_in(L).

:- mode read_line(-).
read_line(Y) :- get0(C), my_nl(E),
    ( C > 64, Y is C \/ 32, skip(E) ;
      C==E, Y = 110 ;
     !, read_line(Y) ).


% For def_other

:- mode read_other_class(+).
read_other_class(F) :-
    r_open(F), name_from_file(F,N), read_pub_as_cls(N), seen, fclose(F).

:- mode read_pub_as_cls(+).
read_pub_as_cls(C) :- read(T), r_pub(T,C), read_pub_as_cls(C).
read_pub_as_cls(_) :- !.

:- mode r_pub(+,+).
r_pub(end_of_file,_) :- !, fail.
r_pub((:- public P),C) :- ass_o_cls(P,C), !.
r_pub((:- op(Priority,Type,Name)),_) :- op(Priority,Type,Name), !.
r_pub(_,_) :- !.

:- mode ass_o_cls(+,+).
ass_o_cls((N/A,P),C)    :- ass_ea_cls(N,A,C), !, ass_o_cls(P,C).
ass_o_cls(N/A,C)    :- ass_ea_cls(N,A,C).

:- mode ass_ea_cls(+,+,+).
ass_ea_cls(N,A,C)  :- other_CLASS_method(N,A,X), !,
    ( X==C ;
      write('class method '), write(N), write('/'), write(A),
      write(' is also defined in file '), write(X), write('.'), nl ), !.
ass_ea_cls(N,A,C)  :- assert(other_CLASS_method(N,A,C)).


% For easy

:- public easy/0.
```

```
easy :- write('
Followings input files specification is possible.
    => [a,''b.pl'',''us2:<soft-ii.duals>xxx.pl''].
    => [a, b.pl,  us2:<soft-ii.duals>xxx.pl ]
    =>  a, b.pl,  us2:<soft-ii.duals>xxx.pl
    =>  a  b.pl   us2:<soft-ii.duals>xxx.pl

Output file names are given by input file name replacing extention with ".esp".
For Above example, output file name:
    ''a.esp'', ''b.esp'', and ''us2:<soft-ii.duals>xxx.esp'' are assumed.

Please key in all input file names to convert.
'),
    prompt(N,'    => '), not_eof_do(N).

:- mode not_eof_do(+).
not_eof_do(N) :- my_nl(E), get_name_list(0,E,D,D,L), prompt(_,N), !,
    def_other(L), !, do_all(L,D).
not_eof_do(N) :- prompt(_,N).

:- mode get_name_list(+,+,?,-,-).
get_name_list(1,_,_,[],[]).
get_name_list(0,C,H,OD,OL) :- !,
    get_c_name([],0,C,H,T,R,Q),
    ( T==[], OD=[], OL=[] ;
      OD=[T|D], OL=[N|L], name(N,T), !, get_name_list(R,Q,H,D,L) ).

:- mode get_c_name(+,+,+,?,-,-,-).
get_c_name(L,R,Q,_,L,R,Q) :- has_alpha(L).
get_c_name([],1,R,H,[],1,Q) :- nonvar(H).
get_c_name(_,_,K,H,T,R,Q) :- get0(C), get_a_name(C,K,M,W,Z), !,
    get_c_name(M,W,Z,H,T,R,Q).

:- mode has_alpha(+).
has_alpha([C|L]) :- I is C\/32, I > 96, I < 123.
has_alpha([_|L]) :- has_alpha(L).

:- mode get_a_name(+,+,-,-,-).
get_a_name(X, _, _,_,_) :- eof(X),!, fail.
get_a_name(39, K,[],0,K).
get_a_name(44, K,[],0,K).
get_a_name(91, K,[],0,93) :- my_nl(K).
get_a_name( C, C,[],1,C) :- ( my_nl(C) ; my_nl(E), skip(E) ).
get_a_name( C, K,[],0,K) :- C < 33.
get_a_name( C, K,[C|F],R,Q) :- get0(X), !,
    get_a_name(X,K,F,R,Q).

:- mode do_all(+,+).
do_all([],[]).
do_all([I|L],[E|D]) :-
    append_esp(E,A), name(O,A),
    to_esp(I,O), !, do_all(L,D).

:- mode append_esp(+,-).
append_esp([],".esp").
append_esp([46|_],".esp").
append_esp([C|E],[C|A]) :-
    ( C==60, cut_dir(E,A,M,N) ; M=E, N=A ), !,
    append_esp(M,N).

:- mode cut_dir(+,-,-,-).
cut_dir([],A,[],A).
cut_dir([62|E],[62|A],E,A).
cut_dir([ C|E],[ C|A],M,N) :- cut_dir(E,A,M,N).
```

```
end_of_file.
```

7 Acknowledgements

The authors would like to thank T. Takizuka for patiently explaining how the translator works. Kasumi Susaki wishes to thank the Director of ICOT, Dr. Kazuhiro Fuchi, and Dr. Shun-ichi Uchida for giving her an opportunity to write this paper. Evan Tick was supported by NSF Grant No. IRI-8704576.

References

[1] T. Chikayama. *ESP Reference Manual.* Technical Memorandum TM-044, ICOT, 4-28 Mita 1-chome, Minato-ku Tokyo 108 Japan.

[2] T. Chikayama. Unique Features of ESP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

[3] K. Fuchi and K. Furukawa. The Role of Logic Programming in the Fifth Generation Computer Project. In *Third International Conference on Logic Programming*, pages 1–24, Imperial College, July 1986.

[4] ICOT. *ESP Guide.* Technical Memorandum TM-388, ICOT, 4-28 Mita 1-chome, Minato-ku Tokyo 108 Japan, 1987.

[5] H. Ishibashi and T. Chikayama. ESP Tutorial. In *Logic Programming Conference '86*, pages 1–10, ICOT, June 1986.

[6] K. Nakajima and et. al. Evaluation of PSI Micro-Interpreter. In *COMPCON Spring 86*, pages 173–177, IEEE Computer Society, San Francisco, March 1986.

[7] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *1987 International Symposium on Logic Programming*, IEEE Computer Society, August 1987.

[8] H. Nishikawa, M. Yokota, A Yamamoto, K. Taki, and S. Uchida. The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture. In *Logic Programming Workshop '83*, pages 53–73, Universidade Nova de Lisboa, June 1983.

[9] K. Taki and et. al. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1984.

[10] E. Tick. *Lisp and Prolog Memory Performance.* Technical Report CSL-TR-86-291, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, January 1986.

[11] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[12] H. Touati and A. Despain. An Empirical Study of the Warren Abstract Machine. In *1987 International Symposium on Logic Programming*, IEEE Computer Society, August 1987.

[13] D. H. D. Warren. *Applied Logic—Its Use and Implementation as Programming Tool.* PhD thesis, University of Edinburgh, 1977. Also available as SRI Technical Note 290.

[14] M. Yokota and et. al. A Microprogrammed Interpreter for the Personal Sequential Inference Machine. In *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1984.