Appendix 4

Maintenance documentation

In this appendix we give the maintenance documentation for `am`, the implemented abstract definitive machine simulator.

Appendix 4 Maintenance documentation

The purpose of this appendix is to describe how the `am` program was implemented. We indicate in outline how the program executes, giving the input syntax in bnf, and then describe the constants, abstract data types, structures, modules and functions used in the program.

Appendix 4.1 Program execution

The program `am` is a simulator for the abstract definitive machine, written in **C**. On invocation the command line arguments are scanned and the appropriate assignments made to the four control variables `nflag`, `aflag`, `silent` and `iterations` from within `main()`. A non-returning call is then made to `yyparse()`, which passes control to the parser generated using yacc from the am.y module. Using the lexical analyser generated using lex from the am.l module the input is read and interpreted according to the grammar in am.y.

Function calls are made from within am.y to the print.c module to print the definition and evaluation of a variable or to list the contents of the stores, to the instantiation.c module for static instantiations and redefinitions, and to simulate.c to start or continue simulation, or to load the run set.

During simulation all references to a variable in D are recorded both with the variable in D and at the place where the variable is used. This allows for the efficient accessing of the variable when its value is required or it is being redefined, but imposes an overhead of maintaining reference information, particularly when entities are instantiated or deleted.

Appendix 4.2 Input syntax

The bnf for the input grammar is:

```
<spec> ::= <spec> <entity> | <entity>
<entity> ::=     ENTITY <id> ( <param_list> ) <body>
               | <id> ( <wparam_list> )
               | define <definition> ;
               | l en
               | l ds
               | l as
               | l runset
               | ? ( <id> <wparameter> )
               | l in
               | status
               | set nflag = <bool>
               | set aflag = <bool>
               | set silent = <bool>
               | set iterations = <integer>
               | load runset
               | cont
               | start | run
               | quit
<bool> ::= true | false
<definition> ::= <def> | <definition> , <def>
<def> ::= <id> <wparameter> = <nexpr>
<wparameter> ::= [ <wparam_list> ]
                  |
<wparam_list> ::=   <wexpr>
                    | <wparam_list> , <wexpr>
                    |

<wexpr> ::=  ( <wexpr> )
             | if <wexpr> then <wexpr> else <wexpr>
             | <integer>
             | - <wexpr>
             | <wexpr> + <wexpr>
```

```
              | <wexpr> - <wexpr>
              | <wexpr> * <wexpr>
              | <wexpr> / <wexpr>
<nexpr> ::=  ( <nexpr> )
              | rand ( <integer> )
              | if <nexpr> then <nexpr> else <nexpr>
              | <integer>
              | <id> <wparameter>
              | - <nexpr>
              | <nexpr> < <nexpr>
              | <nexpr> <= <nexpr>
              | <nexpr> == <nexpr>
              | <nexpr> > <nexpr>
              | <nexpr> >= <nexpr>
              | <nexpr> <> <nexpr>
              | <nexpr> != <nexpr>
              | <nexpr> && <nexpr>
              | <nexpr> and <nexpr>
              | <nexpr> "||" <nexpr>
              | <nexpr> or <nexpr>
              | <nexpr> + <nexpr>
              | <nexpr> - <nexpr>
              | <nexpr> * <nexpr>
              | <nexpr> / <nexpr>
              | @
              | true
              | false
              | not <nexpr>
              | ! <nexpr>
              | "|" <nexpr> "|"
<param_list> ::=  _ <id>
                   | <param_list> , _ <id>
                   |
<pparam_list> ::=   <pexpr>
                    | <pparam_list> , <pexpr>
                    |
<body> ::= { <decl_list> <action_list> }
<decl_list> ::=  definition <var_list>
                  |
<var_list> ::=  <id> <pparameter> <initial>
                 | <var_list> , <id> <pparameter> <initial>
<initial> ::=  = <expr>
                |
<pparameter> ::=  [ <pparams> ]
                   |
<pparams> ::=  <pexpr>
                | <pparams> , <pexpr>
<pexpr> ::=     ( <pexpr> )
               | if <pexpr> then <pexpr> else <pexpr>
               | <integer>
               | _ <id>
               | - <pexpr>
               | <pexpr> + <pexpr>
               | <pexpr> - <pexpr>
               | <pexpr> * <pexpr>
               | <pexpr> / <pexpr>
<action_list> ::= action <act_list>
                   |
<act_list> ::=   <act_list> , <action>
                  | <action>
<action> ::= <lexpr> <procact> - > <command_list>
<lexpr> ::=   ( <lexpr> )
              | rand ( <integer> )
```

```
                | if <lexpr> then <lexpr> else <lexpr>
                | <id> <pparameter>
                | _ <id>
                | <lexpr> < <lexpr>
                | <lexpr> <= <lexpr>
                | <lexpr> == <lexpr>
                | <lexpr> > <lexpr>
                | <lexpr> >= <lexpr>
                | <lexpr> != <lexpr>
                | <lexpr> <> <lexpr>
                | <lexpr> and <lexpr>
                | <lexpr> && <lexpr>
                | <lexpr> or <lexpr>
                | <lexpr> "||" <lexpr>
                | <lexpr> + <lexpr>
                | <lexpr> - <lexpr>
                | <lexpr> * <lexpr>
                | <lexpr> / <lexpr>
                | - <lexpr>
                | true
                | false
                | @
                | not <lexpr>
                | ! <lexpr>
                | <integer>

<expr> ::=   ( <expr> )
                | rand ( <integer> )
                | if <expr> then <expr> else <expr>
                | <integer>
                | <id> <pparameter>
                | _ <id>
                | - <expr>
                | <expr> < <expr>
                | <expr> <= <expr>
                | <expr> == <expr>
                | <expr> > <expr>
                | <expr> >= <expr>
                | <expr> != <expr>
                | <expr> <> <expr>
                | <expr> and <expr>
                | <expr> && <expr>
                | <expr> or <expr>
                | <expr> "||" <expr>
                | <expr> + <expr>
                | <expr> - <expr>
                | <expr> * <expr>
                | <expr> / <expr>
                | @
                | true
                | false
                | not <expr>
                | ! <expr>
                | "|" <expr> "|"
<procact> ::= print ( <message> )
<message> ::=   <msg>
                | <message> , <msg>
<msg> ::=   <quote>
          | <expr>
<command_list> ::=   <clist>
                     |
<clist> ::=   <clist> ; <command>
```

```
              | <command>
<command> ::=   <id> <cmd>
                | <delete>
                | stop
<cmd> ::=   <pparameter> = <expr>
          | ( <pparam_list> )
<delete> ::= delete <id> ( <pparam_list )
```

where the syntactic categories `<id>`, `<integer>` and `<quote>` are recognised within the lexical analyser as (respectively) identifiers, integers and quoted text. Identifiers consist of a letter followed by any combination of letters, digits, underscores and periods. An integer consists of a string of digits. Quoted text consists of double quotes, any text, and double quotes.

The keywords, i.e. `run`, `start`, `entity`, `define`, `print`, `delete`, `definition`, `action`, `true`, `false`, `stop`, `if`, `then` and `else` can consist of either all lower case or all upper case letters, e.g. both `stop` and `STOP` are allowed.

Appendix 4.3 Constants

In this appendix we list the constants used in `am`, and explain what they are used for. The constants are stored in main.h.

0	FALSE	used for boolean variables to mean false
1	TRUE	used for boolean variables to mean true
	Status of the simulation: different values for the `flag` variable in simulate.c	
2	ERROR	error causing termination of the simulation
3	OK	continue simulation
4	MORE	continue simulation
5	FINISHED	stop because `iterations` execution cycles done
6	TERMINATE	stop because `stop` command executed
7	LOADED	stop because run set has been loaded
	type of an element of a procedural action	
0	STRING	
1	EXPRESSION	
	type of brackets, used in print.c	
0	SQR	
1	ROUND	
	type of parameters, used in print.c	
0	PREINST	
1	POSTINST	
	type of simulation	
0	FULL	full simulation
1	PARTIAL	only load the run set
	type of command	
0	REDEF	redefinition
1	INVOKE	entity instantiation
2	DELETE	entity deletion
3	STOP	stop
	type of expression, used in EXPR structure	
1	AND	
2	OR	
3	NOT	
4	NUM	integer

5	ID	identifier
7	BOOL	boolean
8	PLUS	
9	MINUS	
10	TIMES	
11	DIV	
12	LT	less than
13	GT	greater than
14	EQ	equal to
15	LE	less than or equal to
16	GE	greater than or equal to
17	NE	not equal to
18	UMINUS	unary minus
19	EVAL	evaluation
20	UNDEF	undefined
21	PARAM	parameter
22	ALT	alternative (`if...then...else` construct)
23	THEN	second node in tree of `if...then...else` construct
24	RAND	random function

reason why expression cannot be evaluated

0	EXPR_UNDEF	no associated definition
1	NOT_FOUND	definition involves variable not in D
2	CIRCULAR	circular definition

Appendix 4.4 Abstract data types

Stacks are used for various purposes in `am`. Five stacks are implemented in stack.c, and are used for the following purposes:

1 checking for duplicate redefinitions in the run set by simulate.c
2 checking for two dynamic actions on the same entity in the run set by simulate.c
3 checking for deletion of a redefined variable by simulate.c
4 checking for circular definition by evaluation.c
5 checking for instantiation causing duplicate variables in D

The stacks were implemented as abstract data types, with the unnecessary stack operations being removed from stack.c.

Appendix 4.5 Structures

Several structures are used in `am`, for storing the contents of the various stores. The structure declaration is given and the use of each member is described in this appendix.

The `ENTITY` structure is used to store an entity description in P. The type declaration for `ENTITY` is:

```
typedef struct ent_node
{
struct var_node *ent_name ;
int num_param ;
struct var_node *owned ;
struct act_node *actions ;
struct inst_node *instance ;
int num_inst ;
struct ent_node *nxt ;
} ENTITY ;
```

The ent_name member points to the name and formal parameter list of the entity. The num_param member stores the number of parameters which the entity takes - if this were not stored explicitly then whenever a dynamic action on an entity with the same name was performed the parameter list in the ent_name member would need to be iterated through to count the number of parameters. The owned member points to all the variables (and associated initialisations) which are owned by the entity. The actions member points to all the guarded commands and procedural actions which are associated with the entity. The instance member points to a list of where the instantiations of the entity are in D and A, so that they can be removed if the instantiation is deleted. The num_inst member stores the number of instantiations of the entity. The nxt member points to the next entity description in P, being NULL if there are no more.

The PAR structure is used to store parameters. The type declaration for PAR is:

```
typedef struct par_node
{
char *name ;
int value ;
struct expr_node *expr ;
struct par_node *nxt ;
} PAR ;
```

The name member points to the parameter name. Parameters always start with an underscore. The value member stores the value of the parameter after instantiation. The expr member points to the expression which is supplied for the parameter value prior to instantiation. The nxt member points to the next parameter in the list, being NULL if there are none.

The VAR structure is used to store variables, with their parameters and definitions. The type declaration for the VAR structure is:

```
typedef struct var_node
{
char *name ;
struct par_node *parameters ;
struct expr_node *defn ;
struct var_node *nxt ;
} VAR ;
```

The name member points to the name of the variable. The parameters member points to the list of parameters which parameterise the variable, being NULL if the variable is not parametric. The defn member points to the definition associated with the variable, being either NULL or an EXPR with the op member set to UNDEF if the variable is undefined. The nxt member points to the next variable in the list, being NULL if there are none.

The ACTION structure is used to store the guards, procedural actions and command lists in P, A and the run set. The type declaration for the ACTION structure is:

```
typedef struct act_node
{
struct expr_node *guard ;
int is_true ;
struct pact_node *procact ;
struct cmd_node *cmd ;
struct act_node *nxt ;
} ACTION ;
```

The guard member points to the guard expression. The is_true member is used to tag the action as having a true guard during the evaluation part of an execution cycle. The procact member points to the procedural action. The cmd member points to the command list. The nxt member points to the next action in the list, being NULL if there is not one.

The DEFSTORE member is used to store the definition store D. The type declaration for the DEFSTORE structure is:

```
typedef struct ds_node
{
```

```
struct var_node *name ;
struct expr_node *defn ;
struct inst_node *inst ;
struct refs_node *refs ;
struct ds_node *back ;
struct ds_node *nxt ;
} DEFSTORE ;
```

The `name` member points to the name and parameters of the variable. The `defn` member points to the definition associated with the variable. The `inst` member points to the place in P where the entity instantiation is recorded in the entity description. The `refs` member points to a list of references to the variable in other guarded commands and definitions. These pointers must be maintained when dynamic actions are performed, so they are recorded in both directions. This allows for the efficient deletion of references when an entity is deleted. The `back` member is only used for the first owned variable from each instantiation, and points to the previous element of D, being NULL if it is the first variable in D. This allows D to be relinked when the owned variables of a deleted instantiation are removed. The `nxt` member points to the next element in D, being NULL if the variable is the last in D.

The `EXPR` structure is used to store expressions. The type declaration for the `EXPR` structure is:

```
typedef struct expr_node
{
struct var_node *name ;
struct ds_node *pos ;
int op ;
int value ;
struct expr_node *left ;
struct expr_node *right ;
} EXPR ;
```

The `name` member points to the name of the variable in the expression, and the `pos` member points to its position in D. The `op` member indicates what sort of node of the expression tree this is, its value being one of the constants described for the EXPR structure in Appendix 4.3. The `value` member stores integer or boolean values. The `left` member points to the operand of unary operators and the left hand operand of binary operators, and the first operand of ternary operators (i.e. the if...then...else construct). The `right` member points to the right hand operand of binary operators and the second and third operand (in its left and right member respectively) of ternary operators.

The `PROCACT` structure is used to store procedual actions. The type declaration for the `PROCACT` structure is:

```
typedef struct pact_node
{
int type ;
struct expr_node *expr ;
char *message ;
struct pact_node *nxt ;
} PROC ;
```

The `type` member stores the type of the element of the procedural action. The `type` member can have the value of the constant EXPRESSION indicating that it is an expression to be evaluated or the constant STRING indicating that it is a string of text to print. The `member` message points to the string to be printed. The `nxt` member points to the next element of the procedural action list, being NULL if it is the last element in the procedural action.

The `CMD` structure is used to store command lists. The type declaration for the `CMD` structure is:

```
typedef struct cmd_node
{
int command ;
struct var_node *name ;
struct ds_node *pos ;
struct expr_node *defn ;
struct ent_node *entity ;
struct cmd_node *nxt ;
```

```
} CMD ;
```

The command member indicates the type of the command by having the value of one of the constants REDEF, INVOKE, DELETE or STOP. If it has the value REDEF then the name member points to the name of the variable being redefined, the pos member points to its position in D, and the defn member points to its new definition. If the command member has the value INVOKE then the name member points to the name of the entity to be instantiated and the entity member points to its position in P. If the command member has the value DELETE then the name member points to the name of the entity to be deleted and the entity member points to its position in P. The nxt member points to the next command in the list, being NULL if there is not one.

The ACTSTORE structure is used to store the contents of the action store A and the run set. The type declaration for the ACTSTORE structure is:

```
typedef struct as_node
{
struct act_node *action ;
struct as_node *back ;
struct as_node *nxt ;
} ACTSTORE ;
```

The action member points to the action. The back member is only used for the first action from each instantiation, and points to the previous element of A, being NULL if it is the first action in A. This allows A to be relinked when the oactions of a deleted instantiation are removed. The nxt member points to the next element in A, being NULL if the variable is the last in A.

The INSTANCE structure is used to store information about the instantiations of each entity. The type declaration for the INSTANCE structure is:

```
typedef struct inst_node
{
struct var_node *name ;
struct ds_node *var_start ;
struct ds_node *var_end ;
struct as_node *gc_start ;
struct as_node *gc_end ;
struct inst_node *nxt ;
} INSTANCE ;
```

The name member points to the name of the entity which is instantiated. The var_start member points to the position in D of the first variable which must be removed when the entity is deleted. The var_end member points to the position in D of the last variable which must be removed when the entity is deleted. The owned variables are stored contiguously, so all variables between these two positions in D must also be deleted. The gc_start and gc_end members perform the same task for actions in the action store which must be deleted when the instantiation is deleted. The nxt member points to the next instance, being NULL if there is not one.

The REFS structure is used to store reference information about where a variable is used in definitions in D or actions in A. The type declaration for the REFS structure is:

```
typedef struct refs_node
{
struct expr_node *epos ;
struct cmd_node *cpos ;
struct refs_node *nxt ;
} REFS ;
```

The epos member points to a definition in D which refers to the variable. The cpos member points to a command in A which refers to the variable. The nxt member points to the next reference-storing structure. It is NULL when there are no more references to the variable in either definitions in D or commands in A. If there is (for instance) more references in definitions than commands then the cpos member will be NULL in the last elements.

Appendix 4.6 Modules

There are ten modules which are used to construct the am program. We describe their purpose and what other modules use them.

Module name: `error.c`
Purpose: to print error messages for the five types of error which can occur: static, notifiable, avoidance, fatal and death.
Used by: `simulate.c, lex.yy.c, tree.c, instantiation.c, evaluation.c, y.tab.c`

Module name: `tree.c`
Purpose: to perform memory allocation and deallocation operations.
Used by: `simulate.c, lex.yy.c, instantiation.c, evaluation.c, stack.c y.tab.c`

Module name: `main.c`
Purpose: to parse the command line arguments and then invoke the yacc-generated parser.

Module name: `y.tab.c` (generated by `yacc` from `am.y`)
Purpose: to parse the input and invoke the appropriate procedures.
Used by: `main.c`

Module name: `lex.yy.c` (generated by `lex` from `am.l`)
Purpose: to perform lexical analysis on the input by splitting it into tokens.
Used by: `y.tab.c`

Module name: `instantiation.c`
Purpose: to perform static redefinitions and instantiations (i.e. those performed in command mode).
Used by: `y.tab.c, simulate.c`

Module name: `simulate.c`
Purpose: to perform the simulation or load the run set.
Used by: `instantiation.c, y.tab.c`

Module name: `evaluation.c`
Purpose: to evaluate an EXPR expression.
Used by: `simulate.c, instantiation.c, y.tab.c, stack.c`

Module name: `print.c`
Purpose: to print information about the contents of the stores and the current state.
Used by: `y.tab.c`

Module name: `stack.c`
Purpose: to supply stack abstract data types.
Used by: `simulate.c, evaluation.c`

Appendix 4.7 Procedure index

In **C** a procedure is called a function, but we use the term procedure to avoid confusion with the mathematical function. In this appendix we give an alphabetical ordering of the 148 procedures:

Name	Module
`addACTION()`	`tree.c`
`addAS()`	`instantiation.c`

```
addCMD()                tree.c
addCREFS()              tree.c
addDS()                 instantiation.c
addENTITY()             tree.c
addEREFS()              tree.c
addINSTANCE()           tree.c
addPAR()                tree.c
addPROC()               tree.c
add_to_A()              simulate.c
add_to_D()              simulate.c
addVAR()                tree.c
aerror1()               error.c
aerror2()               error.c
can_evaluate()          simulate.c
cause_duplicate()       simulate.c
checkCMD()              instantiation.c
checkEXPR()             instantiation.c
checkPAR()              instantiation.c
checkPROC()             instantiation.c
copyCHAR()              tree.c
copyCMD()               tree.c
copyEXPR()              tree.c
copyPAR()               tree.c
copyVAR()               tree.c
copyPROC()              tree.c
correct()               instantiation.c
death()                 error.c
define()                instantiation.c
delete()                simulate.c
dequeue_runset()        simulate.c
duplicate()             instantiation.c
empty1()                stack.c
empty2()                stack.c
empty3()                stack.c
empty5()                stack.c
eval()                  evaluation.c
ferror1()               error.c
ferror2()               error.c
ferror3()               error.c
ferror4()               error.c
ferror5()               error.c
ferror6()               error.c
ferror7()               error.c
ferror8()               error.c
ferror9()               error.c
freeACTION()            tree.c
freeACTSTORE()          tree.c
freeCHAR()              tree.c
freeCMD()               tree.c
freeDEFSTORE()          tree.c
freeENTITY()            tree.c
freeEXPR()              tree.c
freeINSTANCE()          tree.c
freePAR()               tree.c
freePROC()              tree.c
freeREFS()              tree.c
freeVAR()               tree.c
guard_evaluation()      simulate.c
in_D()                  simulate.c
instantiate()           instantiation.c
is_in()                 instantiation.c
is_on1()                stack.c
is_on2()                stack.c
```

```
is_on3()                stack.c
is_on4()                stack.c
is_on5()                stack.c
isvalue()               instantiation.c
linkCMD()               simulate.c
linkENTITY()            simulate.c
linkEXPR()              simulate.c
linkPROC()              simulate.c
linkup()                simulate.c
linkVAR()               simulate.c
main()                  main.c
make_instance()         simulate.c
my_strcmp()             tree.c
nerror1()               error.c
nerror2()               error.c
nerror3()               error.c
nerror4()               error.c
nerror5()               error.c
nerror6()               error.c
newACTION()             tree.c
newACTSTORE()           tree.c
newCMD()                tree.c
newDEFSTORE()           tree.c
newENTITY()             tree.c
newEXPR()               tree.c
newINSTANCE()           tree.c
newPAR()                tree.c
newPROC()               tree.c
newREFS()               tree.c
newVAR()                tree.c
paramCMD()              instantiation.c
paramEXPR()             instantiation.c
paramPROC()             instantiation.c
pop4()                  stack.c
pos()                   instantiation.c
preprocess()            simulate.c
print_action()          print.c
print_as()              print.c
print_command()         print.c
print_ds()              print.c
print_entity()          print.c
print_expr()            print.c
print_instance()        print.c
print_name()            error.c
print_op()              print.c
print_procact()         print.c
print_runset()          print.c
print_val()             print.c
print_varlist()         print.c
proc_action()           simulate.c
process_runset()        simulate.c
push1()                 stack.c
push2()                 stack.c
push3()                 stack.c
push4()                 stack.c
push5()                 stack.c
reduce()                instantiation.c
remove_quotes()         tree.c
reset()                 main.c
rmCMD()                 simulate.c
rmCREFS()               simulate.c
rmEREFS()               simulate.c
```

rmEXPR()	simulate.c
rmPROC()	simulate.c
rm_refs()	simulate.c
rmREFS()	simulate.c
same()	evaluation.c
samePAR()	instantiation.c
serror1()	error.c
serror10()	error.c
serror2()	error.c
serror3()	error.c
serror4()	error.c
serror5()	error.c
serror6()	error.c
serror7()	error.c
serror8()	error.c
serror9()	error.c
simulate()	simulate.c
store_values()	simulate.c
valid()	instantiation.c
yylex()	lex.yy.c
yyparse()	y.tab.c

Appendix 4.8 Procedure descriptions

In this appendix we describe the use of each procedure, what procedures it uses, and what procedures it is called by:

Module: `main.c`
Procedure name: `main()`
Purpose: to parse the command line and then invoke the parser
Calls: `printf(), reset(), yyparse()`,
Called by:

Module: `main.c`
Procedure name: `reset()`
Purpose: to reset the control variables `nflag`, `aflag`, `silent` and `iterations` to their default values
Called by: `main()`

Module: `lex.yy.c`
Procedure name: `yylex()`
Purpose: to perform lexical analysis by splitting the input into tokens
Calls: `death()`
Called by: `yyparse()`

Module: `y.tab.c`
Procedure name: `yyparse()`
Purpose: to parse the input according to the specified grammar
Calls: `printf(), newVAR(), correct(), addENTITY(), freeENTITY(), freePAR(), freeCHAR(), instantiate(), print_entity(), print_ds(), print_as(), print_runset(), print_name(), samePAR(), my_strcmp(), print_expr(), eval(), print_instance(), print_val(), simulate(), exit(), define(), serror3(), freeEXPR(), newPAR(), addPAR(), newEXPR(), newENTITY(), addVAR(), addACTION(), newACTION(), addPROC(), newPROC(), remove_quotes(), addCMD(), newCMD(),`
Called by: `main()`

Module: `instantiation.c`
Procedure name: `correct()`
Purpose: to test that the static instantiation of an entity will not cause an error
Calls: `duplicate(), checkPAR(), serror2(), checkEXPR(), serror3(), serror9(), checkPROC(), serror8(), checkCMD(), serror10()`
Called by: `yyparse()`

Module: instantiation.c
Procedure name: duplicate()
Purpose: to test for duplicate parameters in a list
Calls: my_strcmp(), serror1()
Called by: correct()

Module: instantiation.c
Procedure name: checkPAR()
Purpose: to test that all parameter in a list can be instantiated
Calls: checkEXPR(), checkPAR()
Called by: correct(), checkPAR(), checkEXPR(), checkCMD()

Module: instantiation.c
Procedure name: checkEXPR()
Purpose: to check that the parameters in an EXPR can be instantiated
Calls: is_in(), checkPAR(), checkEXPR()
Called by: correct(), checkPAR(), checkEXPR(), checkPROC()

Module: instantiation.c
Procedure name: is_in()
Purpose: to test whether a parameter occurs in a list of parameters
Calls: my_strcmp()
Called by: checkEXPR(), checkPROC()

Module: instantiation.c
Procedure name: checkPROC()
Purpose: to check that a procedural action only involves instantiatable expressions
Calls: is_in(), checkEXPR(), checkPROC()
Called by: correct(), checkPROC()

Module: instantiation.c
Procedure name: checkCMD()
Purpose: to check that the parameters in a command are valid
Calls: checkPAR(), checkEXPR(), checkCMD()
Called by: correct(), checkCMD()

Module: instantiation.c
Procedure name: define()
Purpose: to test whether a redefinition results in an error
Calls: my_strcmp(), samePAR(), freeEXPR(),
Called by: yyparse()

Module: instantiation.c
Procedure name: samePAR()
Purpose: to test whether two lists of parameters are identical
Called by: yyparse(), define(), pos(), rm_refs(), delete()

Module: instantiation.c
Procedure name: instantiate()
Purpose: to statically instantiate an entity
Calls: my_strcmp(), death(), valid(), serror4(), newINSTANCE(), addDS(), freeINSTANCE(), addAS(), addINSTANCE(), serror5(), serror6()
Called by: yyparse()

Module: instantiation.c
Procedure name: valid()
Purpose: tests whether the list of parameters has already been used in an instantiation
Called by: instantiate(), process_runset()

Module: instantiation.c
Procedure name: addDS()
Purpose: to add a list of owned variables to D, testing that no duplicates are caused
Calls: newDEFSTORE(), copyVAR(), copyEXPR(), paramEXPR(), freeEXPR(), pos(), serror7(), freeDEFSTORE()
Called by: instantiate()

Module: instantiation.c
Procedure name: pos()
Purpose: to find the position of a the first occurrence of a variable in D
Calls: samePAR(), my_strcmp(), death()
Called by: addDS()

Module: instantiation.c
Procedure name: addAS()
Purpose: to add a list of actions to A
Calls: newACTSTORE(), copyEXPR(), paramEXPR(), copyPROC(), paramPROC(), copyCMD(), paramCMD(), newACTION()
Called by: instantiate()

Module: instantiation.c
Procedure name: paramEXPR()
Purpose: to substitue formal parameters with actual parameters in an EXPR
Calls: paramEXPR(), reduce(), my_strcmp(), freeVAR()
Called by: paramEXPR(), addDS(), add_to_D(), addAS(), add_to_A(), paramPROC(), paramEXPR(), cause_duplicate()

Module: instantiation.c
Procedure name: reduce()
Purpose: to reduce an expression to its simplest form
Calls: isvalue(), eval()
Called by: paramEXPR()

Module: instantiation.c
Procedure name: paramPROC()
Purpose: to substitue formal parameters with actual parameters in a PROC
Calls: paramEXPR(), paramPROC()
Called by: paramPROC(), addAS(), add_to_A()

Module: instantiation.c
Procedure name: paramCMD()
Purpose: to substitue formal parameters with actual parameters in a CMD
Calls: paramCMD(), paramEXPR()
Called by: paramCMD(), addAS(), add_to_A()

Module: instantiation.c
Procedure name: isvalue()
Purpose: test whether an EXPR is an integer or boolean value, or more complex
Called by: reduce()

Module: simulate.c
Procedure name: simulate()
Purpose: to perform the simulation
Calls: freeCMD(), linkup(), preprocess(), guard_evaluation(), process_runset(), empty1(), empty2(), empty3(), empty5(), dequeue_runset(), printf()
Called by: yyparse()

Module: simulate.c
Procedure name: linkup()
Purpose: to record references to variables which are in the definition store

Calls: linkEXPR(), linkPROC(), linkCMD()
Called by: simulate()

Module: simulate.c
Procedure name: linkPROC()
Purpose: to record references to variables in procedural actions
Calls: linkEXPR()
Called by: linkup(), add_to_A()

Module: simulate.c
Procedure name: linkCMD()
Purpose: to record references to variables in command lists
Calls: linkENTITY(), linkVAR(), addCREFS(), linkEXPR()
Called by: linkup(), add_to_A()

Module: simulate.c
Procedure name: linkENTITY()
Purpose: to test that dynamic actions are valid
Calls: my_strcmp(), nerror2(), nerror3()
Called by: linkCMD()

Module: simulate.c
Procedure name: linkEXPR()
Purpose: to store the position of variables in D occurring in expressions
Calls: linkEXPR(), linkVAR(), addEREFS()
Called by: linkup(), linkPROC(), linkCMD(), linkEXPR(), add_to_D(), add_to_A()

Module: simulate.c
Procedure name: linkVAR()
Purpose: to store the position of a variable in D
Calls: same(), nerror1()
Called by: linkCMD(), linkPROC()

Module: simulate.c
Procedure name: preprocess()
Purpose: to preprocess D, testing for notifiable errors caused by nonevaluable or nonexistent variables
Calls: nerror6(), eval(), nerror4(), nerror5()
Called by: simulate()

Module: simulate.c
Procedure name: guard_evaluation()
Purpose: to evaluate the guards, tagging those which are true
Calls: eval(), aerror1(), aerror2(), freeEXPR(), proc_action(), addCMD(), copyCMD(), store_values()
Called by: simulate()

Module: simulate.c
Procedure name: proc_action()
Purpose: to execute a procedural action
Calls: printf(), eval(), freeEXPR()
Called by: guard_evaluation()

Module: simulate.c
Procedure name: store_values()
Purpose: to perform the evaluation caused by the selection of a command list for execution
Calls: can_evaluate(), ferror1()
Called by: guard_evaluation()

Module: simulate.c

Procedure name: `can_evaluate()`
Purpose: to test that an expression can be evaluated
Calls: `eval(), freeEXPR(), can_evaluate()`
Called by: `store_values(), can_evaluate()`

Module: `simulate.c`
Procedure name: `process_runset()`
Purpose: to test whether the run set can be executed in parallel without interference
Calls: `ferror2(), is_on1(), ferror3(), push1(), is_on2(), ferror4(), push3(), ferror7(), ferror5(), push2(), valid(), ferror6(), ferror8(), is_on3(), cause_duplicate(), ferror9(), rm_refs()`
Called by: `simulate()`

Module: `simulate.c`
Procedure name: `cause_duplicate()`
Purpose: to test whether the instantiation will cause duplicate variables in D
Calls: `copyVAR(), paramEXPR(), eval(), death(), freeEXPR(), is_on5(), in_D(), push5(), freeVAR()`
Called by: `process_runset()`

Module: `simulate.c`
Procedure name: `in_D()`
Purpose: to test whether the variable is in D
Calls: `same()`
Called by: `cause_duplicate()`

Module: `simulate.c`
Procedure name: `rm_refs()`
Purpose: to remove referencingf information about the portions of A and D which are to be deleted
Calls: `samePAR(), rmEXPR(), rmREFS(), rmPROC(), rmCMD()`
Called by: `process_runset()`

Module: `simulate.c`
Procedure name: `rmREFS()`
Purpose: to remove references to EXPR and CMD structures by the variable in D
Called by: `rm_refs()`

Module: `simulate.c`
Procedure name: `rmEXPR()`
Purpose: to remove references to D from an EXPR
Calls: `rmEXPR(), rmEREFS()`
Called by: `rm_refs(), rmEXPR(), rmPROC(), rmCMD()`

Module: `simulate.c`
Procedure name: `rmPROC()`
Purpose: to remove references to D from a PROC
Calls: `rmEXPR()`
Called by: `rm_refs()`

Module: `simulate.c`
Procedure name: `rmCMD()`
Purpose: to remove references to D from variables in redefinitions
Calls: `rmEXPR(), rmCREFS()`
Called by: `rm_refs()`

Module: `simulate.c`
Procedure name: `rmEREFS()`
Purpose: to remove a reference to an EXPR from a list of references
Calls: `freeREFS(), addCREFS()`
Called by: `rmEXPR()`

Module: simulate.c
Procedure name: rmCREFS()
Purpose: to remove a reference to an EXPR from a list of references
Calls: freeREFS(), addEREFS()
Called by: rmCMD()

Module: simulate.c
Procedure name: dequeue_runset()
Purpose: to perform the commands in the run set
Calls: linkEXPR(), copyEXPR(), make_instance(), delete()
Called by: simulate()

Module: simulate.c
Procedure name: make_instance()
Purpose: to instantiate an entity
Calls: newINSTANCE(), copyCHAR(), copyPAR(), add_to_D(), add_to_A(), addINSTANCE()
Called by: dequeue_runset()

Module: simulate.c
Procedure name: add_to_D()
Purpose: add a list of owned variables to D
Calls: newDEFSTORE(), copyVAR(), copyEXPR(), paramEXPR(), linkEXPR(), eval(), death(), freeEXPR()
Called by: make_instance()

Module: simulate.c
Procedure name: add_to_A()
Purpose: to add a list of actions to A
Calls: newACTSTORE(), copyEXPR(), paramEXPR(), linkEXPR(), copyPROC(), paramPROC(), linkPROC(), copyCMD(), paramCMD(), linkCMD(), newACTION()
Called by: make_instance()

Module: simulate.c
Procedure name: delete()
Purpose: to delete an instantiation
Calls: samePAR(), freeDEFSTORE(), freeACTSTORE(), freeINSTANCE()
Called by: dequeue_runset()

Module: evaluation.c
Procedure name: eval()
Purpose: to evaluate an EXPR, returning an EXPR which represents either a value or undefined
Calls: newEXPR(), random(), copyVAR(), is_on4(), push4(), same(), pop4(), eval(), death()
Called by: yyparse(), reduce(), add_to_D(), preprocess(), eval(), guard_evaluation(), can_evaluate(), proc_action(), cause_duplicate()
Module: evaluation.c
Procedure name: same()
Purpose: test whether two variables have the same name and parameter list
Called by: linkVAR(), in_D(), eval(), is_on1(), is_on2(), is_on3(), is_on4(), is_on5()

Module: print.c
Procedure name: print_entity()
Purpose: to print the contents of P
Calls: printf(), print_varlist(), print_action()
Called by: yyparse()

Module: print.c
Procedure name: print_varlist()
Purpose: print a list of variables, using either the value or expr member of the parameter list respectively depending

on whether the parameters have been instantiated or not.
Calls: `printf(), print_expr()`
Called by: `print_entity()`

Module: `print.c`
Procedure name: `print_action()`
Purpose: to print an action, consisting of a guard, a procedural action, and a command list
Calls: `print_expr(), printf(), print_procact(), print_command()`
Called by: `print_entity(), print_as()`

Module: `print.c`
Procedure name: `print_expr()`
Purpose: to print an expression
Calls: `printf(), print_expr(), print_op()`
Called by: `yyparse(), print_varlist(), print_action(), print_ds(), print_expr(), print_procact(), print_command()`

Module: `print.c`
Procedure name: `print_op()`
Purpose: to print the appropriate binary operator
Calls: `printf()`
Called by: `print_expr()`

Module: `print.c`
Procedure name: `print_procact()`
Purpose: to print a procedural action
Calls: `printf(), print_expr()`
Called by: `print_action()`

Module: `print.c`
Procedure name: `print_command()`
Purpose: to print a command list
Calls: `print(), print_expr(),`
Called by: `print_action(), print_runset()`

Module: `print.c`
Procedure name: `print_runset()`
Purpose: to print the contents of the run set
Calls: `printf(), print_command()`
Called by: `yyparse()`

Module: `print.c`
Procedure name: `print_as()`
Purpose: to print the contents of A
Calls: `printf(), print_action()`
Called by: `yyparse()`

Module: `print.c`
Procedure name: `print_ds()`
Purpose: to print the contents of D
Calls: `printf(), print_expr()`
Called by: `yyparse()`

Module: `print.c`
Procedure name: `print_instance()`
Purpose: to print the list of current instances
Calls: `printf(),`
Called by: `yyparse()`

Module: `print.c`

Procedure name: `print_val()`
Purpose: to print a boolean value
Calls: `printf()`
Called by: `yyparse()`

Module: `tree.c`
Procedure name: `copyCHAR()`
Purpose: to return a copy of the string passed
Calls: `malloc(), strlen(), strcpy()`
Called by: `make_instance(), copyVAR(), copyPAR(), copyPROC()`

Module: `tree.c`
Procedure name: `freeCHAR()`
Purpose: to deallocate the memory allocated for the char*
Calls: `free()`
Called by: `yyparse(), freePAR(), freeVAR(), freePROC()`

Module: `tree.c`
Procedure name: `newENTITY()`
Purpose: to return a pointer to a new ENTITY structure
Calls: `malloc(), death()`
Called by: `yyparse(), copyENTITY()`

Module: `tree.c`
Procedure name: `addENTITY()`
Purpose: to add an ENTITY element to the end of a list
Called by: `yyparse()`

Module: `tree.c`
Procedure name: `freeENTITY()`
Purpose: to deallocate the memory allocated for the ENTITY list
Calls: `freeVAR(), freeACTION(), freeINSTANCE(), freeENTITY(), free()`
Called by: `yyparse(), freeENTITY()`

Module: `tree.c`
Procedure name: `newPAR()`
Purpose: to return a pointer to a new PAR structure
Calls: `malloc(), death()`
Called by: `yyparse(), copyPAR()`

Module: `tree.c`
Procedure name: `addPAR()`
Purpose: to add a PAR element to the end of a list
Called by: `yyparse()`

Module: `tree.c`
Procedure name: `copyPAR()`
Purpose: to return a copy of a PAR list
Calls: `NEWPAR(), copyCHAR(), copyEXPR(), copyPAR()`
Called by: `make_instance() copyPAR(), copyVAR()`

Module: `tree.c`
Procedure name: `freePAR()`
Purpose: to deallocate the memory allocated for the PAR list
Calls: `freeCHAR(), freeEXPR(), freePAR(), free()`
Called by: `yyparse(), freePAR(), freeVAR()`

Module: `tree.c`
Procedure name: `newVAR()`
Purpose: to return a pointer to a new VAR structure
Calls: `malloc(), death()`

Called by: yyparse(), copyVAR()

Module: tree.c
Procedure name: addVAR()
Purpose: to add a VAR element to the end of a list
Calls: malloc()
Called by: yyparse()

Module: tree.c
Procedure name: copyVAR()
Purpose: to return a copy of a VAR list
Calls: newVAR(), copyCHAR(), copyPAR(), copyEXPR(), copyVAR()
Called by: addDS(), cause_duplicate(), add_to_D(), eval(), copyCMD(), copyVAR(), copyEXPR()

Module: tree.c
Procedure name: freeVAR()
Purpose: to deallocate the memory allocated for the VAR list
Calls: freeCHAR(), freePAR(), freeVAR(), freeEXPR(), free()
Called by: paramEXPR(), freeVAR(), cause_duplicate(), freeENTITY(), freeINSTANCE(), freEXPR(), freeDEFSTORE(), freeCMD(), empty1(), empty2(), empty3(), empty5()

Module: tree.c
Procedure name: newACTION()
Purpose: to return a pointer to a new ACTION structure
Calls: malloc(), death()
Called by: yyparse(), addAS(), add_to_A()

Module: tree.c
Procedure name: addACTION()
Purpose: to add an ACTION element to the end of a list
Called by: yyparse()

Module: tree.c
Procedure name: freeACTION()
Purpose: to deallocate the memory allocated for the ACTION list
Calls: freeEXPR(), freePROC(), freeCMD(), freeACTION(), free()
Called by: freeENTITY(), freeACTION(), freeACTSTORE()

Module: tree.c
Procedure name: newEXPR()
Purpose: to return a pointer to a new EXPR structure
Calls: malloc(), death()
Called by: yyparse(), eval(), copyEXPR()

Module: tree.c
Procedure name: copyEXPR()
Purpose: to return a copy of an EXPR list
Calls: newEXPR(), copyVAR(), copyEXPR()
Called by: addDS(), addAS(), copyEXPR(), dequeue_runset(), copyVAR(), copyPAR(), add_to_A(), add_to_D(), copyCMD(), copyPROC()

Module: tree.c
Procedure name: freeEXPR()
Purpose: to deallocate the memory allocated for the EXPR list
Calls: freeVAR(), freeEXPR(), free()
Called by: yyparse(), define(), freePAR(), freeCMD(), addDS(), can_evaluate(), guard_evaluation(), cause_duplicate(), freePROC(), proc_action(),

freeDEFSTORE(), add_to_D(), freeACTION(), freeEXPR()

Module: tree.c
Procedure name: newPROC()
Purpose: to return a pointer to a new PROC structure
Calls: malloc(), death()
Called by: yyparse(), copyPROC()

Module: tree.c
Procedure name: addPROC()
Purpose: to add a PROC element to the end of a list
Called by: yyparse()

Module: tree.c
Procedure name: copyPROC()
Purpose: to return a copy of a PROC list
Calls: newPROC(), copyEXPR(), copyCHAR(), copyPROC()
Called by: addAS(), add_to_A(), copyPROC()

Module: tree.c
Procedure name: freePROC()
Purpose: to deallocate the memory allocated for the PROC list
Calls: freeEXPR(), freeCHAR(), freePROC(), free()
Called by: freeACTION(), freePROC()

Module: tree.c
Procedure name: remove_quotes()
Purpose: to remove quotes surrounding a string in a procedural action
Called by: yyparse()

Module: tree.c
Procedure name: newCMD()
Purpose: to return a pointer to a new CMD structure
Calls: malloc(), death()
Called by: yyparse(), copyCMD()

Module: tree.c
Procedure name: addCMD()
Purpose: to add a CMD element to the end of a list
Called by: yyparse(), guard_evaluation()

Module: tree.c
Procedure name: copyCMD()
Purpose: to return a copy of a CMD list
Calls: newCMD(), copyVAR(), copyEXPR(), copyCMD()
Called by: addAS(), guard_evaluation(), add_to_A(), copyCMD()

Module: tree.c
Procedure name: freeCMD()
Purpose: to deallocate the memory allocated for the CMD list
Calls: freeVAR(), freeEXPR(), freeCMD(), free()
Called by: simulate(), freeCMD(), freeACTION()

Module: tree.c
Procedure name: newINSTANCE()
Purpose: to return a pointer to a new INSTANCE structure
Calls: malloc(), death()
Called by: instantiate(), make_instance()

Module: tree.c
Procedure name: addINSTANCE()
Purpose: to add an INSTANCE element to the end of a list
Called by: instantiate()

Module: tree.c
Procedure name: freeINSTANCE()
Purpose: to deallocate the memory allocated for the INSTANCE list
Calls: freeVAR(), freeINSTANCE() , free()
Called by: instantiate(), delete(), freeENTITY(), freeINSTANCE()

Module: tree.c
Procedure name: newDEFSTORE()
Purpose: to return a pointer to a new DEFSTORE structure
Calls: malloc(), death()
Called by: addDS(), add_to_D()

Module: tree.c
Procedure name: freeDEFSTORE()
Purpose: to deallocate the memory allocated for the DEFSTORE list
Calls: freeVAR(), freeEXPR(), freeREFS(), freeDEFSTORE(), free()
Called by: addDS(), delete(), freeDEFSTORE()

Module: tree.c
Procedure name: newREFS()
Purpose: to return a pointer to a new REFS structure
Calls: malloc(), death()
Called by: addEREFS(), addCREFS()

Module: tree.c
Procedure name: addEREFS()
Purpose: to add an EXPR reference to the end of a list
Calls: newREFS()
Called by: linkEXPR(), rmCREFS()

Module: tree.c
Procedure name: addCREFS()
Purpose: to add a CMD reference to the end of a list
Calls: newREFS()
Called by: linkCMD(), rmCREFS()

Module: tree.c
Procedure name: freeREFS()
Purpose: to deallocate the memory allocated for the ENTITY list
Calls: freeREFS(), free()
Called by: rmCREFS(), rmEREFS(), freeDEFSTORE(), freeREFS()

Module: tree.c
Procedure name: newACTSTORE()
Purpose: to return a pointer to a new ACTSTORE structure
Calls: malloc(), death()
Called by: addAS(), add_to_A()

Module: tree.c
Procedure name: freeACTSTORE()
Purpose: to deallocate the memory allocated for the ACTSTORE list
Calls: freeACTION(), freeACTSTORE(), free()
Called by: freeACTSTORE(), delete()

Module: tree.c
Procedure name: my_strcmp()

Purpose: to mimic the `strcmp()` procedure but also handle comparison of NULL pointers
Calls: `strcmp()`
Called by: `yyparse(), duplicate(), linkENTITY(), paramEXPR(), is_in(), define(), instantiate(), pos()`

Module: `error.c`
Procedure name: `print_name()`
Purpose: to print a variable or entity name, with appropriate brackets
Calls: `printf()`
Called by: `yyparse(), serror1(), serror2(), serror3(), serror4(), serror7(), serror8(), serror9(), serror10(), nerror1(), nerror4(), nerror5(), nerror6(), aerror1(), aerror2(), ferror1(), ferror2(), ferror3(), ferror4(), ferror5(), ferror6(), ferror7(), ferror8(), ferror9()`

Module: `error.c`
Procedure name: `death()`
Purpose: to print an unrecoverable error message and exit
Calls: `printf(), exit()`
Called by: `yylex(), instantiate(), eval(), cause_duplicate(), pos(), add_to_D(), newREFS(), newENTITY(), newPAR(), newVAR(), newPROC(), newEXPR(), newACTION(), newACTSTORE(), newINSTANCE(), newDEFSTORE(), newCMD()`

Module: `error.c`
Procedure name: `serror1()`
Purpose: to print an error message when a static instantiation contains duplicate parameters
Calls: `printf(), print_name()`
Called by: `duplicate()`

Module: `error.c`
Procedure name: `serror2()`
Purpose: to print an error message when an owned variable uses an unrecognised parameter
Calls: `printf(), print_name()`
Called by: `correct()`

Module: `error.c`
Procedure name: `serror3()`
Purpose: to print an error message when a static redefinition defines a variable not in D
Calls: `printf(), print_name()`
Called by: `yyparse(), correct()`

Module: `error.c`
Procedure name: `serror4()`
Purpose: to print an error message when a static instantiation duplicates an extant instantiation
Calls: `printf(), print_name()`
Called by: `instantiate()`

Module: `error.c`
Procedure name: `serror5()`
Purpose: to print an error message when a static instantiation contains the wrong number of parameters
Calls: `printf()`
Called by: `instantiate()`

Module: `error.c`
Procedure name: `serror6()`
Purpose: to print an error message when a static instantiation refers to a nonexistent entity
Calls: `printf()`
Called by: `instantiate()`

Module: `error.c`
Procedure name: `serror7()`
Purpose: to print an error message when a static instantiation creates duplicate variables in D
Calls: `printf(), print_name()`
Called by: `addDS()`

Module: `error.c`
Procedure name: `serror8()`
Purpose: to print an error message when an entity description has a procedural action with an unrecognised parameter
Calls: `printf(), print_name()`
Called by: `correct()`

Module: `error.c`
Procedure name: `serror9()`
Purpose: to print an error message when an entity description has a guard with an unrecognised parameter
Calls: `printf(), print_name()`
Called by: `correct()`

Module: `error.c`
Procedure name: `serror10()`
Purpose: to print an error message when an entity description has a command with an unrecognised parameter
Calls: `printf(), print_name()`
Called by: `correct()`

Module: `error.c`
Procedure name: `nerror1()`
Purpose: to print an error message when a referred-to variable is not in D
Calls: `printf(), print_name()`
Called by: `linkVAR()`

Module: `error.c`
Procedure name: `nerror2()`
Purpose: to print an error message when a dynamic action in A has the wrong number of parameters
Calls: `printf()`
Called by: `linkENTITY()`

Module: `error.c`
Procedure name: `nerror3()`
Purpose: to print an error message when a dynamic action is performed on an entity not in P
Calls: `printf()`
Called by: `linkENTITY()`

Module: `error.c`
Procedure name: `nerror4()`
Purpose: to print an error message when a definition is circular
Calls: `printf(), print_name()`
Called by: `preprocess()`

Module: `error.c`
Procedure name: `nerror5()`
Purpose: to print an error message when a definition involves nonexistent or undefined variables
Calls: `printf(), print_name()`
Called by: `preprocess()`

Module: `error.c`
Procedure name: `nerror6()`
Purpose: to print an error message when a variable in D has no definition
Calls: `printf(), print_name()`
Called by: `preprocess()`

Module: `error.c`
Procedure name: `aerror1()`
Purpose: to print an error message when a guard cannot be evaluated because of an undefined variable
Calls: `printf(), print_name()`
Called by: `guard_evaluation()`

Module: `error.c`
Procedure name: `aerror2()`
Purpose: to print an error message when a guard cannot be evaluated because of a variable not in D
Calls: `printf(), print_name()`
Called by: `guard_evaluation()`

Module: `error.c`
Procedure name: `ferror1()`
Purpose: to print an error message when an evaluation in a redefinition cannot be performed
Calls: `printf(), print_name()`
Called by: `store_values()`

Module: `error.c`
Procedure name: `ferror2()`
Purpose: to print an error message when there is a redefinition of a nonexistent variable on the run set
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror3()`
Purpose: to print an error message when duplicate redefinitions are on the run set
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror4()`
Purpose: to print an error message when the run set contains a redefinition and deletion of a variable
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror5()`
Purpose: to print an error message when the run set contains two dynamic actions on the same entity
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror6()`
Purpose: to print an error message when the run set contains an instantiation of an already extant entity
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror7()`
Purpose: to print an error message when the run set contains a dynamic action on an entity not in P
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror8()`
Purpose: to print an error message when the run set contains a deletion of an entity which has not been instantiated
Calls: `printf(), print_name()`

Called by: `process_runset()`

Module: `error.c`
Procedure name: `ferror9()`
Purpose: to print an error message when the run set contains an instantiation causing duplicate variables in D
Calls: `printf(), print_name()`
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `push1()`
Purpose: to push an element on to stack 1
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `is_on1()`
Purpose: to test whether an element is in stack 1
Calls: `same()`
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `empty1()`
Purpose: to empty stack 1
Calls: `freeVAR()`
Called by: `simulate()`

Module: `stack.c`
Procedure name: `push2()`
Purpose: to push an element on to stack 2
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `is_on2()`
Purpose: to test whether an element is in stack 2
Calls: `same()`
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `empty2()`
Purpose: to empty stack 2
Calls: `freeVAR()`
Called by: `simulate()`

Module: `stack.c`
Procedure name: `push3()`
Purpose: to push an element on to stack 3
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `is_on3()`
Purpose: to test whether an element is in stack 3
Calls: `same()`
Called by: `process_runset()`

Module: `stack.c`
Procedure name: `empty3()`
Purpose: to empty stack 1
Calls: `freeVAR()`
Called by: `simulate()`

Module: `stack.c`
Procedure name: `push4()`

Purpose: to push an element on to stack 4
Called by: `eval()`

Module: `stack.c`
Procedure name: `pop4()`
Purpose: to pop an element from stack 4
Called by: `eval()`
Module: `stack.c`
Procedure name: `is_on4()`
Purpose: to test whether an element is in stack 4
Calls: `same()`
Called by: `eval()`

Module: `stack.c`
Procedure name: `push5()`
Purpose: to push an element on to stack 5
Called by: `cause_duplicate()`

Module: `stack.c`
Procedure name: `is_on5()`
Purpose: to test whether an element is in stack 5
Calls: `same()`
Called by: `cause_duplicate()`

Module: `stack.c`
Procedure name: `empty5()`
Purpose: to empty stack 5
Calls: `freeVAR()`
Called by: `simulate()`