

The SALAD Cookbook: A User's Guide

Danette Chimenti Ruben Gamboa

MCC
3500 West Balcones Center Drive
Austin, TX 78759-6509

July 18, 1989

Abstract

This document serves as the user's guide to the SALAD (System for Advanced Logic Applications on Data) implementation of \mathcal{LDL} . It describes how \mathcal{LDL} programs and databases are created, compiled, and executed under SALAD. It provides a description of all SALAD commands and a comprehensive treatment of diagnostics.

1 Introduction

This document is the primary user's guide to the SALAD (System for Advanced Logic Applications on Data) implementation of \mathcal{LDL} . It assumes familiarity with the language and/or access to *A Logic Language for Data and Knowledge Bases*, by S. Naqvi and S. Tsur [NT89], where most of the language is described in detail. Certain features were added to \mathcal{LDL} after the publication of [NT89], such as modules and externals, which are described in [CGK89], and I/O facilities, which are described in appendix B. Moreover, the format of the schema file has changed considerably. The new schema syntax, as well as new capabilities in the schema, are described in appendix A.

The only way to learn a programming system is to sit down and write programs in it. Hence, we devote the first section of this paper to guide the user through his/her first programming session in SALAD. The second section provides a reference guide to the system, containing all commands and options available. In the final section, we provide a comprehensive list of error messages generated by SALAD with an explanation of each.

2 A First Session

In this section, we present a thorough walk-through of the creation and execution of a simple program. There are many ways to actually run a program from SALAD; we present only the simplest of these, using a small subset of the commands actually available in SALAD. The details can be found in the reference guide.

2.1 Setting Up the Database

An application is divided into four main sections: the data or known facts, the schema which describes the structure of the data, a set of rules that can be used to derive new facts, and query forms which supply information on how programs are to be invoked. In the traditional literature, the facts and rules are referred to as the extensional and intentional databases, respectively.

The format of the data is specified in the file *schema_name.sch*. We refer to this file as the “schema file.” This file specifies the names and arity (number of arguments) of all base relations in the database. It may also specify type information or names for the columns in a relation, but this is not necessary. The complete format of this file can be found in Chapter 10 of [NT89].

The actual facts are stored in the file *facts_name.fac*, which we refer to as the “fact file.” All relations in this file must have a corresponding (and agreeing) specification in the schema file, though it is not necessary for all relations in the schema to have corresponding facts. It is important to remember that this file contains the initial set of facts only. In the course of execution of a program, the actual facts known to the system may change, e.g. updates, but these changes are not reflected in the fact file. To reflect the changes, the user must invoke the **save** command.

The rules for the program are stored in the file *rules_name.rul*. Initial query forms may also be defined in this file or in a separate query form file such as *qforms_name.qf*.

SALAD creates and maintains directories and subdirectories to store internal data structures. The main directories created by SALAD are given an extension of *.ldl*. The user must not modify these directories and should avoid using the extension *.ldl* in file names.

2.2 The First Program

We are now ready to write our first program in *LDL*. We choose to implement the ancestor problem, which has become the classic example of a logic program. First of all, we need to define the base relations we'll be using. In this case, these are **father** and **mother**. Both of these relations have arity 2, and we interpret them so that `father(mary, john)` makes sense.

The `ancestor.sch` file looks as follows:

```
database( { mother( string, string ),
            father( string, string ) } ).
```

Notice how the name and arity of the relations were specified, as well as the type of their columns. If a type specification of **any** is given, all objects in the *LDL* universe will match.

The facts can now be entered in the `ancestor.fac` file. For our example, we can use the following facts:

```
mother( bill, ann ).
mother( sally, ann ).
mother( liz, jane ).
mother( mary, linda ).
mother( max, linda ).
mother( ann, nicole ).
mother( ed, sally ).

father( jane, bill ).
father( liz, ed ).
father( ed, henry ).
father( mary, henry ).
father( max, henry ).
father( ann, john ).
father( bill, kent ).
father( sally, kent ).
```

The ancestor rules are entered in `ancestor.rul`:

```
parent( X, Y ) <- mother( X, Y ).
parent( X, Y ) <- father( X, Y ).
```

```
anc( X, Y ) <- anc( X, AX ), parent( AX, Y ).  
anc( X, Y ) <- parent( X, Y ).
```

Finally, the query forms for ancestor are entered in `ancestor.qf`:

```
qform compile=no anc( X, Y ).  
qform anc( $X, Y ).
```

When the above query form file is opened, the first query form is defined but not compiled while the latter will be defined and automatically compiled. The uncompiled query form may later be compiled using the `compile` command.

2.3 Running the Program

We are now ready to actually invoke SALAD. To do this, simply type

```
salad
```

at the system prompt. If the system is unable to find SALAD, the search path is probably not set up correctly. Assuming that all is well and SALAD does come up, you should see a prompt as follows:

```
:-)
```

We now open the ancestor database by typing

```
open ancestor.sch ancestor.fac ancestor.rul ancestor.qf
```

SALAD will proceed to open the schema, fact, rule and query form files. Before processing each file, it prints a status message to inform the user of its progress. If you have problems during opening, the last section lists some error messages and tips on debugging. Note that since these files all have been given the same base name, namely `ancestor`, a more succinct version of the open command may be used:

```
open ancestor
```

Using this version of the command will result in SALAD opening any schema, fact, rule or query form files that are found with `ancestor` as the base name.

Now we are ready to execute some queries. For example, we can find the ancestors of `sally` by typing

```
?anc( sally, Y )
```

The system will respond with

```
anc( sally, ann ).
anc( sally, kent ).
anc( sally, nicole ).
anc( sally, john ).
    4 solutions
```

Of course, we can also find the ancestors of any of the people in our database, and the system will return the correct information. However, it doesn't take long to exhaust the possibilities of this query form. When this happens, we may be tempted to find, say, all pairs of ancestors and descendants. The necessary query form was defined in the query form file but not compiled. We can compile this query form by typing

```
compile
```

After the compilation is completed, the query may be given as follows:

```
?anc( X, Y )
```

This should generate the following results:

```
anc( liz, nicole ).
anc( liz, john ).
anc( ed, nicole).
    :
anc( bill, kent ).
anc( sally, kent ).
    34 solutions
```

Additional query forms, referred to as ad-hoc query forms, can be defined (and compiled) with the `qform` command. For example, we may wish to find only the immediate ancestors of `sally`. To do this we can define a query form on the `parent` relation by typing

```
qform parent( $X, Y )
```

The query form will be automatically compiled. If the `compile=no` option is used, the query form will only be defined and can be compiled later using the `compile` command.

Suppose now that we wish to add additional rules, e.g. defining the same generation relation. We can revise the existing rule file `ancestor.rul` to include the new rules and then open the file again. However, SALAD would close the existing version of the rule file before reading in the modified version. Thus, the data structures already constructed for `ancestor` and `parent` would be deleted and then reconstructed along with the new data structures for `same_generation`. Alternatively, we could place the rules for `same_generation` in a new file and open this file. This would result in the rules for `same_generation` being added to the existing rules, i.e. the new data structures would be added without affecting the existing ones.

The new rules are entered in `same_generation.rul`:

```
same_gen( X, Y ) <- same_gen( X1, Y1 ), parent( X, X1 ),
                    parent( Y, Y1 ).
same_gen( X, X ) <- human( X ).
human( X ) <- parent( X, - ).
human( X ) <- parent( -, X ).
```

We add these rules to the existing ones by typing

```
open same_generation.rul
```

If we are no longer interested in using the rules for `ancestor`, we can close the file `ancestor.rul`

```
close ancestor.rul
```

Even though the ancestor rules will now be unavailable, all previously compiled query forms are still available since the query form file has not been closed. Note that if any query forms had been defined in the closed rule file, they would have been deleted and, thus, made unavailable. Naturally, the rules (and associated data structures) for `same_generation` remain intact. The `display` command can be used to verify this as follows:

```
display files compiled rules
```

However, we note that since these rules use the `parent` relation, which was defined in `ancestor.rul`, SALAD will no longer allow compilation of `same_generation`. The `parent` relation must be defined (by opening `ancestor.rul` or otherwise) before compilation is possible.

The current environment, that is, all currently open files and compiled query forms can be stored for later use by using the `save` command:

```
save family.env
```

To restore the saved state, the `restore` command can be used:

```
restore family.env
```

This would be a good time to go ahead and experiment with your own *LDL* programs. Use the `help` command to get a complete listing of the commands and/or detailed information on any one of them. When you are through experimenting, type

```
exit
```

at the SALAD prompt to leave the system.

3 Command Description

In this section, we describe the commands available from SALAD. This is intended to be a fairly complete description, in the spirit of a reference guide, not a tutorial. Hence, commands are presented in alphabetical order, for easy reference.

The SALAD user interface does not require (or allow) a period at the end of a command. A command can be continued on the next line by using a comma before the newline. Commas may also be used to separate command arguments; that is, they are considered white space.

3.1 Conventions

The command descriptions adhere to the following typographical conventions. We use `typewriter` font to highlight keywords and *italics* to denote user-specified values. Square brackets (`[]`) enclose optional items, and a vertical bar (`|`) separates alternate items. Finally, ellipsis (`...`) are used to indicate repetition.

In the command descriptions, *filename* refers to a regular Unix file, whereas, *ldlfilename* refers to special types of files particular to SALAD, e.g., schema files, rule files, etc. Commands process a file differently depending on its type. The type can be specified explicitly using the notation *type:filename* or implicitly when default extensions are used. For example, the following are alternate ways of referring to the same schema file:

- schema:family
- schema:family.sch
- family.sch

Note that when `schema:family` is specified, SALAD first looks for the file `family` and if that file does not exist, it looks for `family.sch`. This explains why the first two file specifications above are equivalent. Also note that when the special keyword `adhoc` is used as an *ldlfilename*, it is understood by SALAD to mean all of the ad-hoc query forms defined by the user. Ad-hoc query forms are those defined interactively using the SALAD `qform` command rather than in a query form or rule file.

For convenience, two different ways to specify the type of a file have been provided, i.e., type names and file extensions. A user need only remember the type names or the file extensions, not both. Type names may be abbreviated, in which case they are identical to the extensions. A complete table of file types and the corresponding extensions follow.

Type	Extension	File Description
sch[ema]	.sch	Schema
fac[ts]	.fac or .db	Facts
rul[es]	.rul or .idb	Rules
qf[orms]	.qf	Query Form
env[ironment]	.env	Saved Environment
com[piled]	.com	Compiled Query Forms

Note that the `.db` and `.idb` extensions are included only for compatibility with the previous version of the system.

In the command descriptions, *queryform* refers to a generic query used to supply information on how a particular program will be invoked, i.e. which arguments will be variables and which will be constant terms. The format of a query form is the same as in [NT89], with the exception that

a module name can be given to qualify an otherwise ambiguous predicate. For example,

```
family.ancestor( $X, Y )
```

refers to the predicate `ancestor` in the `family` module¹. Note that the first argument of the above query form is a dummy constant that serves as a place holder for a constant that will be supplied when the program is invoked. These dummy constants are termed *deferred constants*. Only variables, deferred constants and functors (including list and set functors) are permitted in query forms.

3.2 + — Adding a Fact

Synopsis:

```
+tuple
```

Description:

This command adds a tuple to a base relation in the database. There must be an entry in the schema for the base relation. The tuple must be ground, i.e. must not contain variables. If the tuple does not satisfy any of the type of key constraints specified in the schema, it is not added to the database.

Examples:

```
+father( mary, john )
```

3.3 - — Deleting a Fact

Synopsis:

```
-tuple
```

Description:

This command deletes a tuple, if it exists, from a base relation in the database. There must be an entry in the schema for the base relation. If the tuple specified contains variables, all tuples in the database that unify with it will be deleted.

¹Modules are described in [CGK89].

Examples:

```
-father( mary, john )  
-father( _, john )
```

3.4 ? — Executing a Query

Synopsis:

```
[ ? ]query [ > outfile ]
```

Description:

This command is used to execute a query, which must match one of the compiled query forms. If the output file *outfile* is specified, the output of the query is redirected to that file; otherwise, it appears on the screen. The ? character may be omitted.

Examples:

```
? parent( mary, Y )  
? ancestor( liz, Y ) > ancestors_of_liz  
same_gen( ed, Y )
```

3.5 CD — Changing Directory

Synopsis:

```
cd [ newdir ]
```

Description:

This command changes the working directory of SALAD. If no directory is specified, the user's home directory is assumed. In *newdir*, the tilde (~) character refers to a home directory. If it is immediately followed by a login name, it expands to the home directory of the account named; otherwise, the user's home directory is assumed.

Examples:

```
cd  
cd ~salad/demo
```

3.6 CLOSE — Closing Files

Synopsis:

```
close [ ldlfilename ] ...
close queryform ...
close adhoc
```

Description:

The first form of this command closes the file *ldlfilename*. If no file is specified, all files currently open are closed. The actions taken when a file is closed depend on the type of the file. Closing a schema file also automatically closes all open files, since this is analogous to closing the database system. Closing a rule file makes all rules and query forms defined in the rule file unavailable. Compiling a query form that uses one of these rules is no longer possible. However, query forms already compiled elsewhere, i.e., as ad-hoc query forms or in a query form file, are not affected even if they use rules in the now closed rule file. Closing a query form file makes all query forms defined in it unavailable. Closing a fact file makes the current set of facts unavailable. Moreover, executing a query that uses the database results in a run-time error until a new fact file is open. Note that more than one file may be closed with a single `close` command.

The second form of this command is used to close the query form *queryform*. Note that more than one query form may be closed with a single `close` command. The last form of the command is used to close all ad-hoc query forms.

Examples:

```
close
close family.fac
close family.sch
close adhoc
close ancestor( $X, Y )
close ancestor( $X, Y ) ancestor( X, $Y )
```

3.7 COMPILE — Compiling Query Forms

Synopsis:

```
compile [ compile_options ] [ ldlfilename ] ...
```

Description:

This command is used to compile the query forms defined in the file *ldlfilename*. The special keyword `adhoc` is used to indicate all ad-hoc query forms. Hence, compiling this “file” compiles all query forms defined using the SALAD `qform` command. If no file is specified all query forms defined are compiled. If the type is not specified in *ldlfilename*, the type `qforms` is assumed. Note that more than one file may be compiled with the `compile` command.

If a file is not open prior to its compilation, SALAD automatically opens it. However, if the file has been modified since it was opened, SALAD asks the user whether it should be re-read prior to compilation. This also applies to any files which are used by the file being compiled.

There are several options available when compiling a file. These can be set directly in the *compile_options*. If no *compile_options* are specified, their values are taken from the current defaults as specified by the `default` command. The options can also be set when the query form is specified with the `qform` command, in which case they override *compile_options* as well as the `default` options. If the value of the `CYCLES` option is `YES`, the system will evaluate recursive rules in such a manner as to avoid infinite loops in the presence of cycles. If the value of this option is `NO`, the system will disregard safety concerns and, hence, be able to use more efficient methods for computing recursive goals. The option `DUPLICATES` declares how SALAD should treat duplicate answers. If its value is `YES`, SALAD allows duplicates in the answer and any intermediate derived relations; otherwise, SALAD removes any duplicates. Allowing duplicates can speed up the execution of queries, since there is no need to check an answer with the previously computed ones. However, when large relations are being joined, duplicate elimination may considerably speed up the execution, more than offsetting the extra overhead of checking for duplicate answers. If the value of the `MEMORY` option is `YES`, the memory used to store the database is released before compiling the query form. This means SALAD must re-read the database from disk after the compilation is complete. It is recommended that this option be

used when compiling large programs, since the translator (specifically the C compiler) requires large amounts of memory. If the value of the `OPTIMIZER` option is `YES`, the optimizer is invoked when a query form is being compiled. The last option currently recognized is `REORDER`. This option declares how the goals in a rule are to be ordered. If its value is `YES`, `SALAD` chooses the execution order of the goals; otherwise, the order specified by the user is preserved. Note that if the optimizer is not invoked, the order specified by the user is always the one chosen by the system. These options are summarized in the following table.

Option	Value	Description
CYCLES	YES	Allow for cycles in the database
	NO	Assume that no cycles exist in the database
DUPLICATES	YES	Allow duplicates in derived relations (intermediate and final results)
	NO	Remove duplicates from derived relations
MEMORY	YES	Return database memory to the operating system during compilation
	NO	Retain database memory during compilation
OPTIMIZER	YES	Query forms are optimized
	NO	Query forms remain un-optimized
REORDER	YES	Goals can be re-ordered by the optimizer
	NO	Goals are ordered by the user

Examples:

```

compile family
compile rules:family.ldl
compile family.rul
compile optimizer=yes reorder=no family.qf
compile cycles=no duplicates=yes memory=yes adhoc

```

3.8 DEFAULT — Setting or Querying SALAD Options

Synopsis:

```
default [ option [ = value ] ] ...
```

Description:

This command sets or displays the value of the global SALAD options. If *value* is not specified, the current value of *option* is printed; otherwise, it is set to the new *value*. If no options are specified, the current value of all options is displayed. The options and values are not case sensitive. Note that more than one value may be set or queried in a single default command.

The following table includes the options currently available.

Option	Value	See Command
COMPILE	YES NO	open, qform
CYCLES	YES NO	compile, open, qform
DUPLICATES	YES NO	compile, open, qform
EXCLUSIVE	YES NO	open
MAXFACTS	<i>number</i>	save
MEMORY	YES NO	compile, open, qform
OPTIMIZER	YES NO	compile, open, qform
REORDER	YES NO	compile, open, qform
VERBOSE	YES NO	run
VERIFY	YES NO	save

Examples:

```
default
default optimizer reorder
default optimizer=yes
default optimizer=yes reorder=yes duplicates=no
```

3.9 DISPLAY — Printing SALAD Data Structures

Synopsis:

```
display [ schema |
  facts [ in relname... ] |
  rules [ in predname... ] |
  qforms [ in predname... | in queryform... |
           in ldlfilename... ] |
  compiled [ in predname... | in queryform... |
            in ldlfilename... ] |
  uncompiled [ in predname... | in queryform... |
              in ldlfilename... ] |
  saved |
  files |
  modules [ in modulename... ] |
  types |
  indices [ in relname... ] |
  keys [ in relname... ] ] ...[ > outfile ]
```

Description:

This command is used to print the requested data structures. Available options include printing the schema, facts, rules, query forms, compiled query forms, uncompiled query forms, saved SALAD states, open files, *LDL* modules, defined types, indices, and key columns on the base relations. Some of these options accept arguments that further qualify the desired output. For example, `display facts` prints all facts that have been defined, while `display facts in father` prints only those facts in the `father` relation. If no options are specified, the schema, facts, rules, and query forms are printed. Note that more than one option may be specified. Options must

be separated by a comma if the `in` clause is being used. The output of this command will be directed to the file *outfile* if it is specified. If no output file is given, the results will appear on the screen.

Examples:

```
display
display schema facts rules qforms > program_listing
display facts in father, rules in ancestor
display compiled in family.qf
display compiled in adhoc
```

3.10 EMACS — Editing a File

Synopsis:

```
emacs filename
```

Description:

This command invokes the emacs editor on the file *filename*.

Examples:

```
emacs family.rul
```

3.11 EXIT — Exiting SALAD

Synopsis:

```
exit
```

Description:

This command simply exits SALAD. If there are any open files or compiled query forms that have not been saved before exiting, SALAD asks the user whether he wishes to save the environment.

Examples:

```
exit
```

3.12 HELP — Getting Help from SALAD

Synopsis:

```
help [ command ] [ > outfile ]
```

Description:

This command prints information about *command*. If no command is specified, a brief description of each command is displayed. The output of this command can be redirected to a file for later perusal.

Examples:

```
help
help qform
help compile > compile.help
```

3.13 INDEX — Specifying Indices on Base Relations

Synopsis:

```
index [ rename col1 ... ] ...
```

Description:

This command creates an index on the relation *rename*. Up to five columns may be specified on a single index, and any number of indices may be specified for a given relation. Note that more than one index may be specified with a single `index` command. This practice is recommended over multiple `index` commands, since the tables for the optimizer must be updated to reflect the new index after the command executes. Clearly, it is better to update these tables once after creating many indices than once after each index is created. If no index is specified, SALAD prompts for a relation to modify and then for the columns of the new index. It continues prompting until a newline is pressed at the relation prompt to indicate that no more relations are to be modified. This is convenient when many indices are to be created.

Note that creating an index on a given relation does not affect the currently compiled query forms involving that relation. We suggest recompiling the query forms that the user suspects may be affected by the new index.

Examples:

```
index
index father 1
index mother 2 mother 1 2 father 1 2
```

3.14 LS — Listing a Directory

Synopsis:

```
ls [ dirname ]
```

Description:

This command lists the contents of the directory *dirname*. If no directory is specified, the contents of the current directory are listed.

Examples:

```
ls
ls ~salad/demo
```

3.15 MAKE — Updating Modified Files

Synopsis:

```
make
```

Description:

This command reopens any schema, fact, rule, and query form files that have been modified since they were last opened or reopened. This command is similar to the **reopen** command and is provided as a convenience for program development. All details relevant to the **reopen** command apply here as well.

Examples:

```
make
```

3.16 MORE — Paging Through a File

Synopsis:

```
more filename
```

Description:

This command types the file *filename* on the screen. It waits for the user to press a space after each page of text is displayed.

Examples:

```
more ~/salad/demo/shuttle.rul
```

3.17 NOTRACE — Turning off Tracing

Synopsis:

```
notrace [ compiler_module ] ...
```

Description:

This command discontinues tracing of the SALAD *compiler_module*. If *compiler_module* is not specified, all tracing is discontinued. A complete listing of the compiler modules available for tracing is given with the `trace` command.

Examples:

```
notrace  
notrace qf
```

3.18 OPEN — Opening Files

Synopsis:

```
open [ exclusive = yes | no ] [ compile = yes | no ]  
    [ compile_options ] ldlfilename ...
```

Description:

This command is used to open the file *ldlfilename*. Note that more than one file may be opened with a single `open` command. When a file is opened with the `EXCLUSIVE` option set to `NO`, other users can also open the file; however, none of these users can write or modify the opened file. If the file is opened with the `EXCLUSIVE` option set to `YES`, it is effectively locked from all other users, and the current user is then able to modify the file. If the file to be opened is currently locked, SALAD prints an error message and denies access to the file.

The action taken on opening a file depends on the type of the file. When a schema file is opened, the previous schema file is closed (and hence all previously opened files are closed — see the `close` command) and then the new schema file is read in. If the schema file is being opened for the first time, SALAD creates a directory with the extension `.ld1`, where SALAD's internal version of facts, rules, etc. will be stored. If the schema file is modified and subsequently opened, it is possible that the modifications invalidate SALAD's internal data structures. Some changes are completely transparent to the user, such as the addition of a new base relation. In some cases, however, SALAD is unable to reconcile the facts with the new schema, for example, if a base relation, or a column of a base relation, is deleted. In these cases, SALAD must forget its internal version of the facts, and rebuild them from the ASCII version. Since any updates performed on the database would be lost, the user is warned before this happens. Finally, changes to the schema constraints, such as new type definitions, can invalidate some of the tuples in the database. In this case, the user is warned, so that he can remedy the situation.

When a fact file is opened, the previous fact file is closed and the new one read in. If multiple fact files are opened in a single `open` command, they are merged into a single database. Note that SALAD tries to read in its internal version of the database prior to looking at the user-supplied fact file. In this way, the fact file does not have to be translated into SALAD's

format each time it is opened. If the user-supplied fact file has changed since SALAD's internal version was created, SALAD asks the user whether it should re-read the user-specified fact file. However, the user should exercise caution when re-reading the fact file, since any updates performed and not specifically saved from within SALAD are lost.

When a rule file is opened, the rules are read in by SALAD and any query forms are defined. If the `COMPILE` option is set to `YES` with the `default` command, then these query forms are compiled. Options set with the `default` command are said to be global. If a value for the `COMPILE` option is given in the `open` command, it takes precedence over the global default. Similarly, if a `COMPILE` option is specified in the query form inside the file, it takes precedence over the global and `open` defaults. The *compile_options* are the same as those accepted by the `compile` command. As with `COMPILE`, if an option is specified in the query form, it takes precedence over the global and `open` defaults.

Finally, opening a query form file defines the query forms read from the file. If a type is not given in *ldlfilename*, then the schema, facts, rule, and query form files with principal name *ldlfilename* and the appropriate default extensions are opened.

Rules and facts that are distributed among several files with numbered extensions can be treated by SALAD as a single large file. For example, a large fact file can be split into the three files `family.fac.1`, `family.fac.2`, and `family.fac.3`. When the "file" `family.fac` is opened, all three of these files are loaded in the sequence given. This is also true when `facts:family` or `facts:family.fac` are opened. (The `save` command also creates fact files in this manner.) A similar partition can be applied to a rule file. It is important to note that the extensions must be numeric. Also note that if a fact file or rule file has been split, the individual parts of it can be loaded separately by fully specifying the extension. For example, opening `family.fac.1` reads in only those facts in this file.

Examples:

```
open family
open family.sch father.fac mother.fac
open schema:family facts:father mother
open family.sch family.fac ancestor.rul same_gen.rul
open ancestor.qf
open exclusive=yes compile=yes memory=yes ancestor.qf
```

```
open family.rul.1
```

3.19 QFORM — Defining Ad-Hoc Query Forms

Synopsis:

```
qform [ compile = yes | no ] [ compile_options ]  
      queryform ...
```

Description:

This command defines the new query form *queryform*. All query forms defined using this command are referred to as ad-hoc query forms. Ad-hoc query forms are those not associated with a particular query form or rule file. Note that more than one query form may be defined with the `qform` command. If the value of the `COMPILE` option is `YES`, then *queryform* is also compiled; otherwise, it is not compiled until the `compile` command is used explicitly.

The *compile_options* are the same options accepted by the `compile` command. They override the options specified by both the `default` and the `compile` commands.

Examples:

```
qform parent( $X, Y )  
qform compile=yes ancestor( $X, Y )  
qform duplicates=no reorder=no same_gen( $X, Y )
```

3.20 REOPEN — Reopening Files

Synopsis:

```
reopen [ schema | facts | rules | qforms ] ...
```

Description:

This command is used to reopen the current schema, fact, rule, and/or query form file(s). If no options are specified, all files currently open are

reopened. If the `schema` option is given, the current schema file is re-read, without closing any of the currently open files (cf. `open` command). When the `reopen` command is used, only incremental changes are allowed in the schema. For example, adding a new base relation is allowed, whereas deleting one is not. This guarantees that any open facts or rules files are not adversely affected. If a non-incremental change is desired, the `open` command must be used instead; however, this will close all open files before reading in the new schema.

If the `facts` argument is given, the current fact files are re-read from their original ASCII version. This option can be useful when developing a new program involving updates, since, after some testing, the user may wish to reset the facts to an initial, known state. The same can be achieved by a combination of `save` and `restore` commands.

If the `rules` option is used, all current rule files are reopened. Any query forms defined in the rule files will be recompiled if the `COMPILE` option is set (in the query form itself or with the `default` command). Note that if a query form is defined with the `COMPILE` option off and later compiled during the SALAD session, reopening the file in which the query form was defined will result in the deletion of the compiled version of the query form. If the `qforms` option is used, all current query form files are reopened. As with the `rules` option, query forms will be recompiled as appropriate.

Examples:

```
reopen facts
reopen rules qforms
reopen
```

3.21 RESTORE — Restoring a Saved State

Synopsis:

```
restore ldlfilename
```

Description:

This command reloads the state in *ldlfilename*, which must have been previously saved using the `save` command. Currently, the types of state information which can be saved are facts, compiled query forms, and the environment. If no type is specified, `environment` is assumed.

Examples:

```
restore family_programs.com
restore compiled:family_programs
restore family.env
restore environment:family
restore family_jan17.fac
restore facts:family_jan17
```

3.22 RUN — Executing a SALAD-Script

Synopsis:

```
run [ verbose = yes | no ] cmd_file [ > outfile ].
```

Description:

This command executes the SALAD-script *cmd_file*. The file *cmd_file* may contain any SALAD command, including another **run** command. However, we recommend the user not use recursive invocations of a SALAD-script, since the result will be non-terminating. If the **VERBOSE** option is **YES**, the commands in *cmd_file* are echoed to the screen before being executed; otherwise, only their output will be visible. This flag can be set in the command line or globally with the **default** command. The output of all commands in *cmd_file* (and the commands themselves, if **VERBOSE** is **YES**) are written to *outfile*. If *outfile* is not specified, the output goes to the terminal.

Examples:

```
run initialize_parameters
run verbose=yes compile_family > results_of_compilation
```

3.23 SAVE — Saving the Current State of SALAD

Synopsis:

```
save ldlfilename
save [ verify = yes | no ] ldlfilename
save [ maxfacts = n ] ldlfilename
```

Description:

This command is used to save the current SALAD state in the file *ldlfilename*. This state can subsequently be reclaimed using the **restore** command. Different parts of the state can be saved. If *ldlfilename* is an environment file, the entire SALAD environment is saved. This includes all the schema, rule, and query form files currently open. If any of these files is subsequently modified, when the environment is restored the user is warned of the possible inconsistency and asked whether the system should load the new version or proceed with the old, saved one. It also stores the name of the current fact file, but not its contents. Thus, an environment can be loaded many times, and its query forms executed with the updates of any session preserved for subsequent sessions. If *ldlfilename* is not given the default extension, it is automatically supplied by SALAD.

If *ldlfilename* is a compiled query form file, all currently compiled query forms are saved, but not the original sources, i.e., rule files. Hence, when the saved state is restored, the user can only execute the query forms specified, not compile new ones or modify existing ones. Note that after restoring, the query forms can not be closed individually. However, the *close* command can still be used to close the entire state. The **VERIFY** option is used when saving compiled query forms. If its value is set to **YES**, the user is queried before saving each compiled query form. In this way, a subset of the compiled query forms can be saved. If *ldlfilename* is not given the default extension, it is automatically supplied by SALAD.

Finally, if *ldlfilename* is a fact file, the current facts state is saved. This is advisable when debugging a new application that updates the database. We urge the user to exercise care in choosing the name of a fact file, since any existing file with the name as given will be overwritten. In particular, SALAD does not supply the default extension if it is not specified by the user, so that the user may later use this file in an **open** command. When there is a large database, it is inconvenient, although possible, to store the facts in a single file. If the option **MAXFACTS** is set to n , then at most n facts are written in a fact file. If necessary the resulting fact file is split by using numbered extensions as described in the **open** command. Thus, if **MAXFACTS** has value 100, there are 350 facts in the database, and the facts are saved to the file *family.fac*, then the files *family.fac.1*, *family.fac.2*, *family.fac.3*, and *family.fac.4* are created, the first three with 100 facts each, and the last with the remaining 50 facts. If **MAXFACTS** has value 0, this option is disallowed, thus all facts are saved in a single file.

Examples:

```
save family_programs.com
save compiled:family_programs      % Same as above
save family.env
save environment:family            % Same as above
save family-jan17.fac
save facts:family-jan17           % NOT same as above
```

3.24 STATS — Getting Statistics on the Database

Synopsis:

```
stats [ relname ] [ > outfile ]
```

Description:

This command prints the current indices, number of tuples, and number of distinct values in each column of relation *relname*. If *relname* is not specified, this information is printed for each relation in the database. The output of this command is sent to the file *outfile* if the file is specified; otherwise, it is sent to the terminal.

Examples:

```
stats father
stats > family_stats
```

3.25 TRACE — Tracing

Synopsis:

```
trace [ level = entry | data ] [ compiler_module ] ...
```

Description:

This command sets the trace mode on for the SALAD module *compiler_module*. Note that more than one *compiler_module* may be specified

in a single `trace` command. If `compiler_module` is not specified, all modules in SALAD are traced. The `level` option is used to specify the depth of information to be traced. If `entry` is given then SALAD traces the entry and exit points of `compiler_module`. Permanent data structures relevant to `compiler_module` are also printed if `data` is specified. The `data` level of tracing is intended for use by system implementors only, hence the default level is `entry`. The following table gives all modules of the compiler that are currently available for tracing.

Module	Description
<code>par</code>	Parser
<code>gpg</code>	Global PCG (only for <code>data</code> level)
<code>rca</code>	Recursive Clique Analyzer
<code>rm</code>	Rule Manager, includes <code>par</code> , <code>gpg</code> , <code>rca</code>
<code>rpcg</code>	Relevant PCG Generator
<code>popt</code>	Pre-Optimizer
<code>opt</code>	Optimizer, includes <code>popt</code>
<code>penh</code>	Pre-Enhancer
<code>enh</code>	Enhancer, includes <code>penh</code>
<code>sr</code>	Set Rewriter
<code>cg_compile</code>	C Compiler
<code>cg_link</code>	System Linker
<code>cg</code>	Code Generator, includes <code>cg_compile</code> , <code>cg_link</code>
<code>qf</code>	Query Form Manager, includes <code>rpcg</code> , <code>opt</code> , <code>enh</code> , <code>sr</code> , <code>cg</code>

Examples:

```
trace level=data rpcg enh
trace level=data rm
trace qf
```

3.26 UNIX — Executing a Unix Command

Synopsis:

```
unix [ unix_cmd ]
```

Description:

This command passes the command *unix_cmd* to the Unix operating system. If no command is specified, it creates a unix shell.

Examples:

```
unix
unix cat family.rul
unix ps -u
```

3.27 VI — Editing a File

Synopsis:

```
vi filename
```

Description:

This command invokes the vi editor on the file *filename*.

Examples:

```
vi family.rul
```

4 Diagnostics

In this section we present a list of the diagnostic messages generated by SALAD and their (probable) causes. SALAD prints two different types of diagnostics. An error message is printed when the problem encountered does not allow the system to proceed in a reasonable manner. For example, if an update operation is attempted on a derived relation, the rule involved can not be executed, hence SALAD responds with an error condition. A warning, on the other hand, is printed to inform the user of a possible problem that SALAD has recognized and either ignored or taken action to correct. For example, if a rule is found that can never be satisfied, say it contains the goals $X = 1, \dots, X > 2$, then this rule is deleted from the rule base, and the user is warned.

We present SALAD diagnostics by categorizing them according to the time at which they occur. For example, we group messages that appear when a schema file is opened. Within this classification, we attempt to group related messages together. Note, however, that this does not take into consideration the importance of the problem causing the diagnostic. In particular, fatal errors and simple warnings may appear side by side if they both relate to, say, the compilation of an update goal.

4.1 Initializing SALAD

SALAD consists of two major run-time modules: the translator and the fact manager. The translator is responsible for compiling *LDL* programs. The fact manager processes user requests, stores the database and runs the compiled programs. When SALAD is invoked, it is the fact manager process that is created first. The fact manager is a memory-resident database engine. When it is created, its first action is to allocate sufficient memory to store the largest possible database. If it is unable to do so, it prints

Internal Error (fatal): Out of memory

and terminates. In some instances, sufficient memory may be released by killing off extraneous processes. This is particularly true in window-based environments, where it is too easy to create a large number of essentially dormant processes.

The fact manager attempts to allocate memory at a specified location. This is important, because the copy of the database stored in disk does not contain relocation information, i.e., it must always be loaded at the

same base address. If the fact manager is unable to allocate memory where required, it prints

```
Internal Error (fatal): Memory not allocated at base address
```

and then terminates. If this error is encountered, SALAD can not run with your system configuration, since memory is evidently not allocated the way SALAD assumes. Both the amount and starting location of memory required by SALAD can be configured at installation time. This can be useful if system resources are limited.

After initializing, the fact manager attempts to create a translator process. If the translator process is not spawned successfully, one of the following messages is printed, depending on the cause of failure.

```
Cannot fork
```

```
Cannot exec program: ProgName
```

```
Internal Error (fatal): Cannot invoke Prolog
```

Typically, the problem can be traced to insufficient memory for the translator. Killing off extraneous processes, thus releasing memory to the operating system, will help in these cases. If the problem persists, contact your system manager.

4.2 Invoking a Command

In this section, we consider errors that occur immediately after a command is issued (and before it is executed). For example, when opening a fact file, if this file has been locked by another user, SALAD responds with an error message before opening the file. Such a diagnostic would be included in this section. On the other hand, if SALAD finds a syntax error in the fact file, this error would occur as the command is executed. Hence, any diagnostics generated would be found in the section corresponding to opening fact files.

4.2.1 Opening Files

File names can contain wildcard characters, such as * and ?, but only in the relative portion of the pathname. That is, wildcards are not allowed to specify any directory leading to the file in question. If wildcards are found in a directory component of a file name, SALAD prints

```
No wild cards allowed in directory name: FileName
```

where *FileName* is the invalid file name.

Only one schema file may be open at a time. Hence, opening a schema file automatically closes the previous schema (and thus all files previously open). However, if more than one schema file is opened with the same `open` command, SALAD is unable to choose between them, and prints

Only one schema may be specified

In this case no files are opened.

On the other hand, a schema must be open before a fact file can be opened, since the schema is supposed to describe the format of the facts. If a fact file is opened before opening a schema file, SALAD prints

No schema is currently open

The fact file is not opened; however, any other files in the `open` command are processed normally.

SALAD associates permanent directories with the schema and fact files. When a schema or fact file is opened for the first time, SALAD automatically creates these directories if the exclusive option is set. If the permanent directory already exists, but was not created by SALAD, the command is aborted. This is likely to happen when the user chooses the same name for a file as SALAD. If this occurs, SALAD prints

**Internal Error: Directory not in proper format: *DirName*
Please remove and try again**

where *DirName* is the name of the directory SALAD is trying to access.

When a file is opened without exclusive access, any permanent directories associated with the file must already exist. If the directories do not exist, SALAD will print

Can not create schema dir without exclusive access

Can not create facts dir without exclusive access

as appropriate. Moreover, to create the facts directory, the schema must also be open exclusively. Otherwise, SALAD prints

Can not create facts dir without locking schema

When the schema file changes non-incrementally, for example, the addition of a column to a relation, the SALAD directory associated with the schema file and all subsidiary facts directories must be deleted by SALAD. This can only happen if the user has exclusive access to the schema. Otherwise, SALAD prints

Can not modify schema without exclusive access

Opening a file can fail if it is currently locked. If this happens, SALAD responds with

File is currently locked: *DirName*

where *DirName* is the name of the directory SALAD is trying to access. The file may not be opened until the person locking the file releases it. Unfortunately, this may take an arbitrarily long time.

If many processes attempt to lock a file simultaneously, some of them may be “starved.” If this happens, SALAD prints

Time out while locking file: *DirName*

where *DirName* is the name of the directory SALAD is trying to access. This does not mean the file has been locked. In particular, the user may attempt to open file again, until he is successful, or someone else locks it.

4.2.2 Closing Files

If more than one fact file is open, it is not possible to close only some of these files. They must all be closed at the same time. Otherwise, SALAD prints

Can not close fact file incrementally

Note that all fact files are automatically closed if a new fact file is opened.

4.2.3 Accessing the Facts

If an operation is done which requires access to the database, such as an update or `stats` command, but no fact file is open, SALAD prints

No database is currently open

If a fact containing variables is inserted into the database, SALAD prints

Can only add ground tuples to database: *Fact*

where *Fact* is the unground fact.

If a fact is added to a non-existent relation, SALAD prints

Unknown relation name: *Rel*

where *Rel* is the relation name.

4.2.4 Defining Query Forms

If a query form that matches more than one global predicate is defined, SALAD prints

Ambiguous predicate name: *Pred*

If no global predicates match the query form, SALAD prints

Unknown predicate name: *Pred*

where *Pred* is the unknown or ambiguous query form predicate name.

The query form must contain only variables or deferred constants. If constants are found, SALAD prints

Query form contains constant terms: *QueryForm*

where *QueryForm* is the invalid query form.

4.2.5 Executing Queries

If a query is to be executed, but none of the currently compiled query forms unify with it, SALAD prints one of the following

No query forms match query: *Query*

No compiled query forms match query: *Query*

where *Query* is the requested execution query. To compile a matching query form, simply use the `qform` or `compile` command.

If SALAD is unable to execute a compiled query form, SALAD prints one of the following messages

Internal Error: Cannot open query form file

Internal Error: Compiled query form is too large

In the first case, closing and reopening the rule files should fix the problem. In the latter, SALAD's internal structures must be enlarged to run the current program.

4.2.6 Executing a SALAD-Script

If more than twenty nested `run` commands are found, SALAD complains

Too many nested runs (recursion?)

The most likely cause of this is a recursive run file, which can never terminate, since there are no conditional commands in SALAD.

4.3 Opening the Schema

These error conditions can occur when SALAD is opening a schema file. When a fatal error occurs opening the schema, the `open` command does not proceed opening any of the other files, such as facts or rules, since the latter depend on schema information.

4.3.1 Basic Syntax

This section enumerates syntax errors found when the schema file is being parsed. When a syntax error is detected, SALAD will discontinue processing of the immediate term or statement and continue processing subsequent statements in order to flag as many syntax errors as possible. If any syntax errors are discovered, the schema file will not be opened.

There are six types of statements in the schema file, namely `define`, `database`, `index`, `key`, `scratch` and `temp` statements. If SALAD encounters a statement it can not classify, it prints

```
Invalid statement in schema file: Cmd
```

If this happens, SALAD can not open the schema file.

If an attribute, type, or relation name is malformed or simply missing where expected in a key, name, or database statement, SALAD prints one of the following

```
Invalid attribute name: Attr
```

```
Invalid type name: Type
```

```
Invalid relation name: Rel
```

The set of columns specified must not be empty in an index or key statement. If it is, SALAD prints one of the following as appropriate

```
Cmd attribute list can not be empty
```

where *Cmd* is either index or key, as appropriate.

4.3.2 Database Definition

Only a single database definition is permitted in each schema file. If more than one is specified, SALAD warns

```
More than one database statement found in schema
```

If a base relation is defined more than once in a schema statement, SALAD prints the following error

Base relation is multiply-defined: *Rel/Arity*

where *Rel* is the base relation and *Arity* its arity.

If a type is used in a schema statement, but is not defined, SALAD prints the error

Undefined type: *Type*

If two columns of a relation are given the same attribute name, SALAD prints the error

Duplicate attribute name in relation: *Rel, AttrName*

where *Rel* is the relation and *AttrName* is the duplicated attribute name.

If a relation is declared as temporary in a temp or scratch statement, but it is not defined in the database statement, SALAD prints the error

Unknown relation name: *Rel*

4.3.3 Index and Key Definitions

If an index or key is declared on a relation not defined in the database statement, SALAD prints the error

Unknown relation name: *Rel*

If an attribute name is used in an index or key declaration, but is not defined, SALAD warns

Undefined named attribute for *Rel*: *AttrName*

where *Rel* is the relation and *AttrName* the undefined attribute name.

If a column is specified in an index or key declaration, but is outside the range of the corresponding base relation, SALAD prints the error

Column out of range in *Cmd* statement: *Rel, Col*

where *Cmd* is one of key or index, *Rel* is the relation, and *Col* is the non-existent column.

4.3.4 Type Definition

If a define statement is found for one of SALAD's built-in types (integer, real, string, and any), SALAD prints the error

Attempt to redefine built-in type: *Type*

SALAD does not allow cyclic (define(X,Y) and define(Y,X)) type definitions. A non-trivial cyclic definition will result in the following error

Cyclic type declaration: *Type*

where *Type* is the mal-formed type. A trivial cyclic definition (define(X,X)) results in the warning

Trivial type declaration: *Type*

4.4 Opening the Facts

In this section we describe errors that occur when SALAD is opening a fact file. Note that these errors can only happen when SALAD is actually reading the fact file. At this time, SALAD creates a permanent directory that is associated with the fact file being opened. Subsequently, whenever the same fact file is opened, SALAD reads the information contained in the permanent directory, not the fact file itself. Hence, these errors can only occur when a fact file is being opened for the first time.

First of all, if the fact file contains a fact for a base relation that is not defined in the schema, SALAD prints

Unknown relation: *Rel/Arity*

where *Rel* is the base relation and *Arity* its arity. In this case, the fact file is not opened. In particular, the permanent directory associated with the fact file is not created.

All facts must be terminated by a period. If a fact is followed by anything other than a period, SALAD prints

Expecting a period after reading a fact

Again, the fact file is not opened, and the permanent directory is not created.

In dealing with very large data bases, tables internal to the fact manager may conceivably be exceeded. If this happens, the system proceeds opening the fact file; however, performance may be greatly reduced. SALAD warns the user by printing one of the following:

Internal Error: Out of atom uniqueness hash space
Internal Error: Out of functor uniqueness hash space
Too many tuples for index hash space: *Rel*

4.5 Opening the Rules

In this section we describe errors that occur when SALAD is opening a rule file. When a fatal error occurs opening a rule file, that rule file is not opened; however, the `open` command will proceed to open any other rule files specified. Note that query forms can be defined in the rule file. After all files specified in the open command are processed, these query forms will automatically be compiled if the compile option is set. Thus, errors relating to the opening of query forms (section 4.6) as well as to the compiling of query forms (section 4.7) may be applicable here as well.

4.5.1 Basic Syntax

This section enumerates syntax errors found when the rule file is being parsed. When a syntax error is detected, SALAD will discontinue processing of the immediate rule and continue processing subsequent rules in order to flag as many syntax errors as possible. If any syntax errors are discovered, the rule file will not be opened.

If a string or comment is found which is not terminated before the end of file, the system responds with one of

```
string terminates with eof  
comment terminates with eof
```

as appropriate.

If a malformed list term is found the system prints

```
unbalanced list parenthesis detected
```

All variables occurring within a grouping operator must be bound by the bodies in the grouping clause. Hence, if SALAD finds a grouping operator with an anonymous variable, it prints the error

```
anonymous variable disallowed in set grouping
```

If SALAD is unable to find a valid term where it expects one, for example, in a relational expression, it prints

missing or invalid term

A valid predicate symbol is alphanumeric and must start with a lowercase letter. If a predicate symbol is found which does not satisfy these criteria, SALAD prints

invalid predicate symbol

If a malformed predicate is found, SALAD prints

invalid literal

If the malformed predicate is negated, on the other hand, SALAD prints one of the following as appropriate.

invalid negated literal

invalid negated relational or comparison literal

If an invalid arithmetic operator is used, SALAD prints

invalid arithmetic expression

If SALAD finds an incomplete rule, it prints

missing <- or .

If an equal sign is not found in a named attribute, SALAD prints

missing =

4.5.2 Aggregates

An aggregate definition consists of a triplet of rules covering the empty, single and multiple cases. If a multiple case is specified, then the single case must be specified as well. If not, SALAD prints the error

Incomplete definition for aggregate *AggrOp* in module *Mod*

where *AggrOp* is the operation name given to the aggregate and *Mod* is the name of the module where it is defined.

If the multiple or empty case is not specified for an aggregate, SALAD warns

No empty case defined for aggregate *AggrOp* in module *Mod*

No multiple case defined for aggregate *AggrOp* in module *Mod*

where *AggrOp* is the operation name given to the aggregate and *Mod* is the name of the module where it is defined.

If the definitions for an aggregate are split among module components, SALAD prints the error

Multiple definitions for aggregate *AggrOp* in module *Mod*

where *AggrOp* is the operation name given to the aggregate and *Mod* is the name of the module where it is defined.

If an aggregate is defined but is not used, SALAD warns

Unused aggregate: *AggrOp* in module *Mod*

where *AggrOp* is the operation name given to the aggregate and *Mod* is the name of the module where it is defined.

Conversely, if an aggregate is used but is not defined, SALAD warns

Undefined aggregate: *AggrOp* in module *Mod*

where *AggrOp* is the operation name given to the aggregate and *Mod* is the name of the module where it is defined. Note that no query forms may be compiled for *Mod* while undefined aggregates exist.

4.5.3 Modules

The use of modules (and externals) in SALAD is described in detail in [CGK89]. It is important to note that a module may be split among several rule files and that predicates defined in a module will be visible only within that module unless they are exported from the module and explicitly imported into other modules. Thus, the same predicate name used in two different modules will refer to two completely different predicates. Note also that when a module is split among rule files, each portion of the module that exists in a file is referred to as a module component.

If a predicate is defined in more than one context, for example, is imported and defined in the rules, SALAD gives the error

Predicate *Pred/Arity* in module *Mod* is multiply-defined

where *Pred* is the predicate, *Arity* its arity, and *Mod* the name of the module where the multiple definitions are found.

If a predicate is defined but is not used, SALAD warns

Unused predicate: *Pred/Arity* in module *Mod*

where *Pred* is the predicate, *Arity* its arity, and *Mod* the name of the module where it is defined.

If a predicate is used but is not defined, SALAD warns

Undefined predicate: *Pred/Arity* in module *Mod*

where *Pred* is the predicate, *Arity* its arity, and *Mod* the name of the module where it is used. Note that no query forms may be compiled for *Mod* while undefined predicates exist.

Only a single module component of a given module may be defined in a file. If more than one module component of the same module occurs in a file, SALAD prints the error

All rules for module must be together when in the same file

If the import statement does not specify the exporting module name and more than one module exports the given predicate, SALAD prints the error

Multiple exports of imported predicate *Pred/Arity*

where *Pred* is the imported predicate and *Arity* its arity.

On the other hand, if the imported query form does not have a corresponding export statement, SALAD warns

Predicate *Pred/Arity* **is not exported as needed by module** *Mod*

where *Pred* is the imported predicate, *Arity* its arity, and *Mod* the name of the module where it is imported.

It is possible that a previously defined imported query form becomes undefined when the file exporting the query form is closed. If this happens, SALAD warns

Predicate *Pred/Arity* **is no longer exported from module** *Mod*

where *Pred* is the exported predicate, *Arity* its arity, and *Mod* the name of the module where it was exported.

4.5.4 Named Attributes

If positional and named-attribute notations are mixed in a base relation, SALAD prints either

Ordinal number expected in using base relation *Pred/Arity*

Attribute name rejected: *AttrName*

or

Attribute name expected in using base relation *Pred/Arity*
Ordinal number rejected: *Ord*

where *Pred* is the name of the base relation, *Arity* its arity, *Ord* and *AttrName* are the ordinal number and the attribute name used, respectively.

Named attributes can not be used in a derived relation. The only way to use a non-positional notation in this case is to use ordinal numbers directly. If named attributes are found on a derived relation, SALAD prints the error

Ordinal number expected in using derived relation *Pred/Arity*
Attribute name rejected: *AttrName*

where *Pred* is the name of the derived relation, *Arity* its arity and *AttrName* is the invalid attribute name.

When using ordinal numbers, no column position outside the arity range of the relation may be used. If a non-existent column is specified, SALAD prints the error

Ordinal number in *Pred/Arity* is out of range: *Ord*

where *Pred* is the predicate name, *Arity* its arity and *Ord* the bad column number.

If a named attribute is used which is not defined in the schema file, SALAD prints the error

No such attribute name for base relation *Rel/Arity*: *AttrName*

where *Rel* is the base relation, *Arity* its arity, and *AttrName* the undefined attribute name. To define the attribute, use the name command in the schema.

When a named attribute is defined on an undefined predicate, SALAD prints the error

Predicate with named attribute with unknown arity at this
point: *Pred*

where *Pred* is the undefined predicate. The predicate can be defined in the schema or earlier in the rule file.

If a named attribute is used more than once in a predicate, SALAD prints the warning

Duplicate named attribute in predicate *Pred/Arity*: *AttrName*

where *Pred* is the predicate, *Arity* its arity, and *AttrName* the attribute name.

4.5.5 Recursion

\mathcal{LDL} requires that every group of mutually recursive predicates have at least one non-recursive rule, i.e., an exit rule. If a recursive group (clique) is found which does not satisfy this constraint, SALAD prints the error

Recursive clique has no exit rule: *RecPreds*

where *RecPreds* is a list of the recursive predicates belonging to the clique.

4.5.6 Reducible Predicates

If a predicate is found which always evaluates to true, it is eliminated from the rule set, and SALAD warns

Predicate is always true in rule, hence is deleted: *Pred*

where *Pred* is the tautology.

Similarly, if a predicate is found which always evaluates to false, then the rule in which it is found can never be satisfied and is deleted. In this case, SALAD warns

Rule is always false, hence is deleted: *Rule*

where *Rule* is the contradiction.

4.5.7 Sets

If a set containing multiple copies of a term (e.g., $\{X,X\}$) is encountered, all but one of the copies is removed, and SALAD prints the warning

Duplicate set term in set is removed: *Set*

where *Set* is the redundant set.

4.5.8 Singleton Variables

If a variable occurs only once in a rule, SALAD warns

Singleton variable(s) in clause *Pred/Arity*: *Variables*

where *Pred* is the predicate, *Arity* its arity, and *Variables* the list of singleton variables.

4.5.9 Updates

If SALAD finds an update operation done on a derived relation, it prints the error

```
Non-base predicate can not be updated
```

4.6 Opening Query Forms

In this section we describe errors that occur when SALAD is processing a query form that occurs within a rule file. The errors occurring when query forms are processed from a query form file are the same as those generated when query forms are defined with the `qform` command and can be found in section 4.2.4.

4.6.1 Basic Syntax

If a query form is not syntactically correct or does not appear where expected, the system responds with the following error

```
missing or invalid query form predicate
```

The arguments allowed in a query form include deferred constants, variables, set terms, lists, tuples and other complex terms. If one of the query form arguments is invalid or not well-formed the system will respond with the following error

```
missing or invalid query form term
```

The constants `{}`, `[]`, and `nil` are not allowed as query form arguments. If they are used as such, SALAD prints one of the following errors

```
empty set is a constant, not allowed  
empty list is a constant, not allowed  
nil is a constant, not allowed
```

If the variable name of a deferred constant is not valid, SALAD prints the error

```
invalid deferred constant type
```

4.7 Compiling Query Forms

4.7.1 External Functions

Externals are defined with certain binding restrictions in the import statement. If an imported goal is then used with insufficient bindings, SALAD prints the error

```
Bindings for external Pred are incorrect
```

where *Pred* is the external predicate name.

Complex terms can not be passed to an external function. If an external goal with complex terms is specified, SALAD responds with the error message

```
Complex objects in call to external: Goal
```

where *Goal* is the external goal.

4.7.2 Modules

If the query form to be compiled is defined in a module which contains undefined predicates (imported or local) or incomplete recursion, SALAD prints the following message

```
Module Mod is not complete and, thus, can not compile
```

where *Mod* is the name of the incomplete module. If the problem is caused by undefined predicates, opening more files may solve the problem. On the other hand, if the problem is found in recursion, and is not due to a missing predicate (e.g., missing exit rule for a recursive clique), the rules need to be modified and the rule file reopened.

If the predicates to be imported during compilation form a recursive clique, SALAD prints the error

```
Mutual recursion across modules for query form: QueryForm
```

where *QueryForm* is the problematic query form.

If there is no imported query form corresponding to the binding pattern for the current imported goal, SALAD prints the error

```
No match for imported predicate Pred/Arity
```

where *Pred* is the imported predicate name and *Arity* its arity.

4.7.3 Recursion

Currently, set terms may not appear within a recursive goal. If this restriction is violated, SALAD prints the error

Set term(s) within recursive goal: *Goal*

where *Goal* is the goal containing the misplaced set term.

Moreover, choice is not allowed to appear within recursion, either. If this restriction is not adhered to, SALAD prints the error

Choice within materialization or recursion: *Goal*

where *Goal* is the invalid choice goal.

Stratification does not allow a grouping operation to be performed on a recursive rule. Nor does it allow the negation of a recursive goal in a recursive rule. (See [NT89] for more details.) If these restrictions are violated, SALAD prints one of the following errors:

Grouping in the head of a recursive rule

Recursive clique is unstratified with negated predicate: *Pred*

where *Pred* is the negated, recursive goal.

4.7.4 Safety

A rule is said to be covered (safe) when all variables in its head are also found in the body. If a rule does not satisfy this condition, SALAD prints the error

Unsafe argument(s): *Args*

where *Args* are the variables in the head which are not bound in the body.

A negated goal may not be used to instantiate variables, only to test their value. If negation is used to bind a variable, SALAD prints the error

Unsafe negated goal: *Goal*

where *Goal* is the negated goal.

Both the **then** and the **else** branch of an if statement must bind all the variables yielded by the if statement. In particular, note that if there is only one branch (i.e., no else) then it must not bind any variables. Moreover, the condition must not bind any arguments used by the **else** branch. If this is not the case, SALAD prints the following error message

Unsafe if stmt: *Goal*

An evaluable predicate, such as an arithmetic or relational operation, must have all of its variables bound in order for the set of answers to be finite. For example, the expression $X < Y$ can only be evaluated if X and Y are bound. If the minimum binding requirement is not met, SALAD prints the error

Unsafe Evaluable Predicate: *Pred*

where *Pred* is the evaluable predicate.

Similarly, the set operations can only be evaluated for certain binding patterns. The set operators along with the acceptable binding patterns for each are given in the following table where **b** represents a bound argument and **f** represents a free argument (variable).

Set Operator	Bindings
union	bbb bbf ffb bfb fbb
subset	bb fb
member	bb fb
intersection	bbb bbf
difference	bbb bbf

For example, the expression `member(E,Set)` can only be evaluated if `Set` is bound. If the minimum binding requirement is not met, SALAD prints the error

Bindings for set primitive are incorrect: *AdornedPred*

where *AdornedPred* is the adorned set primitive, such as `member(f,b)`.

4.7.5 Unification

If the query form to be compiled does not match any predicate definition, SALAD prints the message

Query form *Pred/Arity* does not match any definition in module
Mod

where *Pred* is the query form predicate name, *Arity* its arity and *Mod* the module.

If no definitions are found for a goal, SALAD prints the error

Goal always fails as no rule or import matches it: *Goal*

where *Goal* is the goal that has no support.

If no definitions are found for a negated goal, SALAD warns

Negated goal *Goal* always succeeds as no rule matches it

where *Goal* is the goal that has no support.

If a unification is performed which results in a self-reference, SALAD prints the message

Occurs check error: *Term1* and *Term2*

where *Term1* and *Term2* are the self-referencing terms. Note that a rule which fails on the occur check is deleted. SALAD warns the user of this by printing

Rule fails on occur check, hence is deleted: *Rule*

where *Rule* is the malformed rule.

4.7.6 Updates

The current semantics of \mathcal{LDL} do not allow a failing goal after an update operation has been made. Hence, if a (potentially) failing goal is found after an update, SALAD prints the error

Failing goal found after an update: *Goal*

where *Goal* is the misplaced failing goal.

The semantics also do not allow updates to occur below a goal with multiple definitions. If this happens, SALAD prints the error

Update goal found in multiple or-branch: *Goal*

where *Goal* is the misplaced update goal.

Insertion updates may only be performed on fully bound objects. That is, all variables in an update must be bound before the update can be performed. If this is not the case, SALAD prints the error

Update Goal not ground: *Goal*

where *Goal* is the unsafe update goal.

Updates (deletion or insertion) must not bind any variables. If they attempt to, SALAD prints the error

Attempt to assign values from update goal: *Goal*

where *Goal* is the unsafe update goal.

Currently, SALAD does not accept updates within recursion, negation, or grouping. If such an update is found, SALAD prints the error

Update within recursion, negation or grouping: *Goal*

where *Goal* is the misplaced update goal.

4.8 Executing a Query

In this section we present the run-time error messages provided by SALAD. Most are given as warnings and pertain to type-checking. For example, adding two strings or sets will generate a warning.

4.8.1 Arithmetic Operators

If the system attempts to perform an arithmetic operation on two non-numeric terms, it will print the warning

Attempt to *Arith_op* *Term1* and *Term2*

where *Arith_op* is one of `add`, `subtract`, `multiply`, `divide`, or `modulo`, and *Term1* and *Term2* are the offending terms.

The only other arithmetic error that is trapped is division by zero. If this happens, SALAD warns

Divide by zero!

4.8.2 Set Operators and Aggregates

If a set operation or aggregate is attempted on any object other than a set, SALAD responds with the warning

Expecting set instead of *Term*

where *Term* is the offending object.

4.8.3 External Procedures

Since external procedures have typed parameters, these types must agree with the actual objects passed at run-time. If they do not, SALAD warns

```
Expecting Type instead of Term
```

where *Type* is one of `integer`, `real`, or `string` and *Term* is the non-conforming object.

4.8.4 Updates

Updates can not be performed without having exclusive access on the fact file. If an update is attempted while the facts are not locked, SALAD prints the error message

```
Can not update without exclusive access
```

A Schema Files

The format of the schema file has changed significantly since the publication of [NT89]. In this appendix, we describe the new syntax of the schema file and define the new capabilities of the system.

The schema describes all of the base relations in the database. The minimum information required to declare a base relation is its name and arity. Optionally, a column of a base relation can be typed. If a type is defined, SALAD will enforce it by checking each tuple as it is inserted into the database. A column (or group of columns) can also be declared as a key in the relation. Again, this constraint is enforced as tuples are added to the database. Finally, indices can be declared in the schema.²

In the following sections, we describe how each of these can be defined. First of all, we present the `define` statement, which is used to compose new data types from the basic data types known to SALAD. Then, we show how base relations are declared, including the type and name of each column in the relation. We also show how the duration of the facts in the relations can be controlled — this makes it easy to store temporary results in base relations. Finally, we show how keys and indices can be declared.

²Even if no indices are defined in the schema, an index is always assumed over the first column of the relation. Moreover, as queries are compiled, SALAD may choose to add its own indices to the database, transparently to the user.

A.1 Composing New Data Types

Synopsis:

```
define( typename, type_specification ).
```

The `define` statement is used to compose a new data type from the basic data types known to SALAD, i.e., `integer`, `real`, `string`, and `any`. The first three basic data types are self-descriptive. `Any` is used to refer to all objects in the \mathcal{LDL} universe — it is convenient when no typing is desired.

Types can be composed in one of two ways. First of all, more complex types can be built by using complex objects. For example, the statement

```
define( QUANTITY, qty( string, integer ) ).
```

defines a type of complex objects with functor `qty`, arity 2, and with a `string` and `integer` for its first and second arguments, respectively. As a special case, lists and sets can be specified by `[typename]` and `{ typename }`, as appropriate. For example,

```
define( NAME_LIST, [ string ] ).
```

specifies a type consisting of all (possibly empty) lists of strings.

Types can also be specified as the union of simpler types. This is accomplished by using multiple `define` statements. Thus, the type `NUMBER` can be defined by

```
define( NUMBER, integer ).  
define( NUMBER, float ).
```

SALAD also allows the definition of recursive types. For example, an arithmetic expression could be defined as

```
define( EXPR, NUMBER ).  
define( EXPR, plus( EXPR, EXPR ) ).
```

As with recursive predicates, it is necessary that at least one of the type definitions be non-recursive.

The extension to mutually recursive types is quite natural. In the next example, we present the type definition of a simple AND/OR graph.

```
define( OR, NUMBER ).  
define( OR, or( [ AND ] ) ).  
define( AND, and( [ OR ] ) ).
```

Type constraints are checked when a new tuple is inserted into the database. If the tuple does not satisfy one of the constraints, the update operation *aborts*. That is, the update is not performed. Moreover, the query currently executing is terminated, and any updates performed by that query are retracted — the database returns to the state prior to execution of the query.

A.2 Declaring Base Relations

Synopsis:

```
database( { rel_name( [attr_name:]type_spec,...),... } ).
```

The `database` statement is used to define all the base relations. There should be at most one `database` statement in the schema file.

The definition for each base relation consists of a template, identifying the relation name, as well as the type and, optionally, attribute name of each argument. For example, the following statement declares the base relations used in a family database:

```
database( { mother( Child:string, Mother:string ),
           father( Child:string, Father:string ) } ).
```

In general, a type specification can be used instead of a type name. In the family database, for example, we can associate each parent with the set of his children. This structure can be defined as

```
database( { mother( Mother:string, Children:{string} ),
           father( Father:string, Children:{string} ) } ).
```

Naturally, union types (and hence recursive types) can not be declared implicitly in the `database` statement, but must be named in a `define` statement.

The lifetime of the facts stored in the database can also be declared in the schema. This is useful to denote relations which store temporary facts, used in the computation of a single query. For example, in a typical assembler or compiler, a first phase is used to build a symbol table, which is then used by the second phase. In \mathcal{LDL} , the symbol table can be stored in a base relation, declared as follows:

```
database( { symtab( Name:string, Loc:integer ) } ).
```

Of course, the facts stored in this table are not necessary after the *LDL* program terminates. This can be specified by using a `scratch` statement, such as

```
scratch( symtab ).
```

SALAD will now ensure that the relation `symtab` is always empty when a new query is invoked.

In some applications, it is inconvenient to delete the temporary facts after each query is executed. For example, in a CAD/CAM application, a base relation can be used to store the current transformation matrix used to display the graphics model. This matrix can be modified by specialized *LDL* programs to do rotations, translations, etc. Unlike the symbol table described above, the transformation matrix is needed after each query terminates. However, it is temporary, in the sense that subsequent invocations of the CAD/CAM application need not be aware of its last state.³ This can be specified by using a `temp` statement, such as

```
temp( trans_matrix ).
```

SALAD will ensure that all `temp` relations are empty at the beginning of each new session — that is, when a new fact file is opened.

A.3 Declaring Indices and Key Constraints

Synopsis:

```
index( rel_name, [ col1,... ] ).  
key( rel_name, [ col1,... ] ).
```

SALAD allows the user to specify an arbitrary number of indices on any of the base relations. The columns for the index are specified in a list, which is not optional. If more than one column is specified in an `index` statement, a single index is declared over all the columns. Up to five columns may be specified for each index. For example, in the family database defined above, the following indices can be declared,

```
index( mother, [ Mother ] ).  
index( mother, [ Child ] ).  
index( mother, [ Child, Mother ] ).
```

³In fact, if more than one person is using the application, storing the transformation matrix between sessions could lead to serious disagreements.

Note that the last statement declares a single, composite index over `Child` and `Mother`, not two separate indices.

If attribute names are not defined in the `database` statement, it is still possible to define indices, by using ordinal numbers instead. Thus, the indices declared above can also be declared by

```
index( mother, [ 2 ] ).  
index( mother, [ 1 ] ).  
index( mother, [ 1, 2 ] ).
```

SALAD always assumes an index on the first column on each relation. Thus, the index on `Child` above is redundant. It is important to note that, as queries are compiled, SALAD may choose to define its own indices. However, SALAD never deletes an index specified by the user. Also note that simply declaring an index does not guarantee that it will be used — the particular index chosen (if any) is left up to the optimizer and the code generator of SALAD.

Keys are declared in much the same way as indices. SALAD enforces key constraints by checking each tuple as it is added to the database. As with type constraints, when an update is performed that violates a key constraint, the update is aborted, terminating the query and retracting any other updates performed by it.

Declaring a key on a given column (or group of columns) automatically declares an index on this column. Hence, keys are subject to the same restriction as indices — up to five columns may be specified, but no more. In the example above, we may wish to specify a key on the `Child` as follows

```
key( mother, [ Child ] ).
```

This declares that a child has at most one mother. Specifying a key on `Mother` is unduly restrictive, since a single mother may have many children. A key can also be specified on both `Child` and `Mother`; however, it is unnecessary for two reasons. First of all, it is subsumed by the key on `Child` alone. Secondly, there are no other columns on the `mother` relation. Since SALAD removes duplicates from base relations, the key is automatically assumed.

B Input/Output Primitives

In this appendix, we describe the input/output primitives of *LDL*. Incorporating I/O into a purely declarative language, such as *LDL*, poses some interesting problems. For example, in the program

```
read( X ), read( Y )
```

it is presumably important that the order of the `read` statements not be reversed. Similar problems occur in the presence of updates, where it is important that the order of updates be preserved. In \mathcal{LDL} , it has been the programmer's responsibility to order update goals correctly — we extend this to I/O primitives. In the above example, `X` is read before `Y`. For the same reason, we restrict I/O primitives so that they can not occur as subgoals of a predicate that contains more than one rule. Note that this includes all recursive predicates. For a full discussion of why these restrictions are necessary, the reader is referred to [NK88].

I/O operations can result in run-time errors, for example, when writing to a full file-system or opening a non-existent file for read access. When this occurs, the I/O predicate *aborts*, which terminates the entire query and retracts any updates performed by it. As will be seen later, I/O operations can also *fail*. The two notions should not be confused, as the former is much more drastic.

In the following sections, we describe the input operations. Subsequently, we describe the basic output primitives, followed by the formatted output primitive. Finally, we show how files can be used in \mathcal{LDL} .

B.1 Basic Input Primitives

Synopsis:

```
get( Char ).  
read( Term ).
```

The `get` and `read` primitives are used to read \mathcal{LDL} terms from the standard input. `get` returns the next character from the standard input, while `read` scans a complete \mathcal{LDL} term (ignoring comments and whitespace). Both of these predicates return the atom `end_of_file` to indicate end of file.⁴ If the argument to either `get` or `read` is bound or partially bound (e.g., a complex object), it is possible that the I/O operation fails. That is, the next term to be read in does not match the argument specified. This does not, however, mean that the goal *aborts* — the reader should keep this distinction in mind.

⁴In fact, when reading from the keyboard with `read`, the string `end_of_file` can be used to indicate end of file, as well as the EOT character. This is particularly useful when the input is in a runfile which is being executed.

We illustrate these primitives with an example. Continuing with the family example, we show an \mathcal{LDL} program that reads **mother** and **father** facts in the same format as in the fact file — that is, a sequence of tuples, separated by a period — and inserts them into the database.

```

forever ( read( Tuple ),
          Tuple ~= end_of_file,
          get( '.' ),
          if ( Tuple = mother( Child, Mother ) then
              + mother( Child, Mother )
          else
              Tuple = father( Child, Father ),
              + father( Child, Father ) ) ).

```

B.2 Basic Output Primitives

Synopsis:

```

write( Term ).
writeq( Term ).
tab.
tab( Count ).
nl.
nl( Count ).
flush.

```

The `write` predicate is used to output an arbitrary *LDL* term to standard output. In general, a term written in this way can not be read back in using `read`. For example, the atom `'hello world'` contains an embedded blank. If it is written using `write`, a subsequent `read` statement will find the two atoms `hello` and `world`. The `writeq` predicate, on the other hand, prints quotes when they are necessary, hence terms written with `writeq` can be read in later using `read`.

A simple example using `write` is shown below.

```

forever ( get( Char ),
          Char ~= end_of_file,
          write( Char ) ).

```

The program simply copies its standard input onto its standard output. The use of `write` instead of `writeq` here is not arbitrary. Recall that `get` returns the next character of the input, including whitespace. Had we used `writeq`, quotes would be placed around such characters, for example newlines.

The `nl` and `tab` commands are used to print a newline and tab, respectively. The optional argument denotes the number of such characters to print. For example,

```
nl( 2 ).
```

can be used to double-space the output. With no argument, one newline or tab is printed.

Finally, the `flush` command is used to flush the output buffers. This is useful mainly in interactive applications. For example, `flush` may be called after a prompt is displayed by a command interpreter.

B.3 Formatted Output Primitives

Synopsis:

```
format( format_spec, [ arg1,... ] ).
```

The `format` statement gives the programmer wide control over the output of an *LDL* program. The *format_spec* is a string which specifies the way the arguments in the arguments list are to be printed. Note that the argument list is not optional, but may be empty. The format specifiers in *format_spec* all begin with a tilde (~), and are summarized in figure 1. Any character found in *format_spec* that is not a format specifier is printed verbatim. The sequence `~~` is used to print a single tilde (~).

The format specifiers can be broken into four major categories. Any object can be printed using the `~w` and `~q` specifiers. These print the appropriate argument using `write` or `writeq`, respectively. Thus, the goal

```
format( 'X = w', [X] ).
```

is simply a more succinct version of

```
write( 'X = ' ), write( X ).
```

The next category of format specifiers can be used to print simple *LDL* objects — that is, atoms and numbers. The specifier `~a` is used to print an atom. If the corresponding object is not an atom, an error message is printed and the `format` command aborts. Numbers can be printed with either the `~d` or the `~f` specifier. The first is used to print integers. If the corresponding argument is real, only its decimal portion is printed. The second specifier is used to print real numbers. The number of digits to print after the decimal can be specified, as in

```
format( 'Pay this amount: $~2f', [ Price ] ).
```

SPECIFIER	DESCRIPTION
<code>~w</code>	Argument is printed using <code>write</code>
<code>~q</code>	Argument is printed using <code>writeq</code>
<code>~a</code>	Argument must be an atom, and is printed using <code>write</code>
<code>~d</code>	Argument must be an integer, and is printed using <code>write</code>
<code>~f</code>	Argument must be a real, and is printed using <code>write</code>
<code>~nf</code>	Argument must be a real, and is printed with n digits after the decimal point
<code>~b</code>	Page break
<code>~nb</code>	n page breaks
<code>~n</code>	Newline
<code>~nm</code>	n newlines
<code>~t</code>	Tab
<code>~nt</code>	n tabs
<code>~s</code>	Space
<code>~ns</code>	n spaces
<code>~ </code>	Set tab at current position
<code>~n </code>	Set tab at position n , and goto this position
<code>~+</code>	Set tab at next position that is a multiple of eight, and goto this position
<code>~n+</code>	Set tab n spaces after current tab, and goto this position
<code>~*</code>	Fill allotted space with blanks
<code>~c*</code>	Fill allotted space with c characters
<code>~~</code>	Tilda (~)

Figure 1: Format Specifiers

If the number of digits is not specified, SALAD prints six digits after the decimal point. There is a subtle difference between `~d` and `~0f`. Both can be used to print a number, and both print only the decimal part — no decimal point is used. However, the first one truncates the real number, whereas the second rounds it before printing.

The third category of format specifiers can be used to display whitespace. Page breaks, newlines, tabs and spaces can be printed using `~b`, `~n`, `~t` and `~s`, respectively. All of these specifiers accept an optional count, thus

```
format( '~2n', [ ] ).
```

can be used to double-space the output.

The last category of format specifiers can be used to define tables in the output format. An absolute tab position can be specified using `~n|`, which places the tab at position *n*. For example, the following statement

```
format( '~15|Hello World!~n', [ ] ).
```

prints the string `Hello World!` starting at column 15. A relative tab position is specified using `~n+`, which places the tab *n* columns after the current tab position. Thus, the statement

```
format( '~15|Hello~10+World!~n', [ ] ).
```

prints `Hello` starting at column 15 (as before) and `World!` starting at column 25. It is sometimes useful to specify a tab at the current cursor position, especially when text is mixed in with the format specifier. The tab can be declared simply by using `~|`. For example, this can be used to build a table as follows

```
format( 'Material: ~|~a~20+Quantity: ~d n',  
        [ Material, Qty ] ).
```

Notice how the combination `~|~a~20+` is used to produce an entry with twenty columns of width, without having to specify its starting column. The space between column boundaries can be controlled using `~c*`, where *c* is a single printable character. Normally, whitespace is used such that columns are aligned on the left. When the `~c*` command is used, the whitespace is filled with as many characters *c* as are needed. For example, the following `format` statement shows how the string `Hello World` can be right aligned.

```
format( '~15|~ * Hello World!~30+~n', [ ] ).
```

In this case, the fill character used is a blank. The explicit blank is actually not necessary because `~*` defaults to using a blank as the fill character. If more than one `~*` command is used, the whitespace is filled evenly between them. For example, the following `format` statement centers the string `Hello World`, using `-` as the fill character.

```
format( '~15|~-* Hello World! ~-*~30+~n', [] ).
```

B.4 I/O on Files

Synopsis:

```
open( FileName, Mode, FilePtr ).  
close( FilePtr ).
```

The `open` and `close` commands are used to open and close a file, respectively. In the `open` predicate, the *FileName* and *Mode* must be supplied, and the *FilePtr* is returned. The *FilePtr* is used in all subsequent I/O operations, such as in the `close` predicate shown above. The *Mode* is one of `read`, `write` or `append`. If `read` is specified, the file *FileName* must exist; otherwise, the `open` predicate aborts. If the file does not exist, and either `write` or `append` is specified, it is simply created.

All of the I/O operations described in the previous sections can be used with files — the *FilePtr* is simply passed as the first argument to each I/O predicate. As an example, we present a new version of the copy program shown before.

```
open( InputFile, read, InPtr ),  
open( OutputFile, write, OutPtr ),  
forever ( get( InPtr, Char ),  
          Char ~= end_of_file,  
          write( OutPtr, Char ) ),  
close( InPtr ),  
close( OutPtr ).
```

After a query terminates, SALAD automatically closes all open files, so the `close` statements above are not strictly necessary.

The whitespace commands deserve a special mention. Recall that `n1` is used to print a single newline on the standard output, and `n1(2)` prints two such newlines. When using files, `n1(FilePtr)` is not correct, since

the *FilePtr* is confused with the count above. The correct version of the command is

```
n1( FilePtr, Count ).
```

Hence, even if a single newline is desired, the count must be specified, as in

```
n1( OutPtr, 1 ).
```

This also applies to the `tab` command.

References

- [CGK89] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Using modules and externals in *LDL*. Technical Report ACA-ST-036-89, MCC, 1989.
- [NK88] Shamim Naqvi and Ravi Krishnamurthy. Database updates in logic programming. In *Proceedings of the Seventh Annual Association of Computing Machinery Symposium on Principles of Database Systems*, 1988.
- [NT89] Shamim Naqvi and Shalom Tsur. *A Logic Language for Data and Knowledge Bases*. W.H. Freeman, 1989.