be set to the atom `all` in which case all public predicates are imported. Same as `load_files(`*File*`, [if(changed),imports(Imports)])`.

`use_module(-`*Module*`, :`*File*`, +`*Imports*`)`
`use_module(+`*Module*`, :`*File*`, +`*Imports*`)`
If used with `+`*Module*, and that module already exists, this merely imports *Imports* from that module. Otherwise, this is equivalent to `use_module(`*File*`,` *Imports*`)` with the addition that *Module* is unified with the loaded module.

`fcompile(:`*Files*`)` **[Obsolescent]**
Compiles the source file or list of files specified by *Files*. If *Files* are prefixed by a module name, that module name will be used for module name expansion during the compilation (see Section 5.4 [Considerations], page 66). The suffix '`.pl`' is added to the given filenames to yield the real source filenames. The compiled code is placed on the '`.ql`' file or list of files formed by adding the suffix '`.ql`' to the given filenames. (This predicate is not available in runtime systems.)

`source_file(?`*File*`)`
File is the absolute name of a source file currently in the system.

`source_file(:`*Head*`,?`*File*`)`
`source_file(-`*Head*`,?`*File*`)`
Head is the most general goal for a predicate loaded from *File*.

`require(:`*PredSpecOrSpecs*`)`
PredSpecOrSpecs is a *predicate spec* or a list or a conjunction of such. The predicate will check if the specified predicates are loaded and if not, will try to load or import them using `use_module/2`. The file containing the predicate definitions will be located in the following way:

- The directories specified with `user:library_directory/1` are searched for a file '`INDEX.pl`'. This file is taken to contain relations between all exported predicates of the module-files in the library directory and its subdirectories. If an '`INDEX.pl`' is not found, `require/1` will try to create one by loading the library package `mkindex` and calling `make_index:make_library_index(`*Directory*`)` (see Chapter 12 [The Prolog Library], page 249).
- The first index entry for the requested predicate will be used to determine the file to load. An exception is raised if the predicate can't be located.
- Once an '`INDEX.pl`' is read, it is cached internally for use in subsequent calls to `require/1`.
- Not available in runtime systems.

7.1.2 Term and Goal Expansion

When a program is being read in, SICStus Prolog provides hooks that enable the terms being read in to be source-to-source transformed before the usual processing of clauses or directives. The hooks consist in user-defined predicates that define the transformations. One transformation is always available, however: *definite clause grammars*, a convenient notation for expressing grammar rules. See [Colmerauer 75] and [Pereira & Warren 80].

Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form

head `-->` *body*`.`

meaning "a possible form for *head* is *body*". Both *body* and *head* are sequences of one or more items linked by the standard Prolog conjunction operator '`,`'.

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).
2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list '`[]`'. If the terminal symbols are character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, '`[]`' or '`""`'.
3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in '`{}`' brackets.
4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
5. Disjunction, if-then, if-then-else, and not-provable may be stated explicitly in the right-hand side of a grammar rule, using the operators '`;`' ('`|`'), '`->`', and '`\+`' as in a Prolog clause.
6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in '`{}`' brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

In the last rule, C is the character code of some digit.

The query

```
| ?- expr(Z, "-2+3*5+1", []).
```

will compute Z=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

```
p(X, S0, S) :- q(X, S0, S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) -->
        q(X),
        r(X, Y),
        s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X, Y, S0, S) :-
        q(X, S0, S1),
        r(X, Y, S1, S2),
        r(Y, S2, S).
```

Terminals are translated using the built-in predicate `'C'(`*S1*`,` *X*`,` *S2*`)`, read as "point *S1* is connected by terminal *X* to point *S2*", and defined by the single clause

```
'C'([X|S], X, S).
```

(This predicate is not normally useful in itself; it has been given the name upper-case `c` simply to avoid using up a more useful name.) Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated by

```
p(X, S0, S) :-
        'C'(S0, go, S1),
        'C'(S1, to, S2),
        q(X, S2, S3),
        'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X, S0, S) :-
        'C'(S0, X, S1),
        integer(X),
        X>0,
        q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals are translated using the built-in predicate `'C'`(*S1*, *X*, *S2*), read as "point *S1* is connected by terminal *X* to point *S2*", and defined by the single clause

Terminals on the left-hand side of a rule are also translated using `'C'/3`, connecting them to the output argument of the head non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N, S0, S) :-
        'C'(S0, aint, S1),
        'C'(S, not, S1).
```

Disjunction has a fairly obvious translation, e.g.

```
args(X, Y) -->
        (   dir(X), [to], indir(Y)
        ;   indir(Y), dir(X)
        ).
```

translates to

```
args(X, Y, S0, S) :-
        (   dir(X, S0, S1),
            'C'(S1, to, S2),
            indir(Y, S2, S)
        ;   indir(Y, S0, S1),
            dir(X, S1, S)
        ).
```

Similarly for if-then, if-then-else, and not-provable.

The built-in predicates which are concerned with grammar rules and other compile/consult time transformations are as follows:

`expand_term(+`*Term1*`,?`*Term2*`)`

If *Term1* is a term that can be transformed, *Term2* is the result. Otherwise *Term2* is just *Term1* unchanged. This transformation takes place automatically when grammar rules are read in, but sometimes it is useful to be able to perform it explicitly. Grammar rule expansion is not the only transformation available; the user may define clauses for the predicate `user:term_expansion/[2,4]` to perform other transformations. `user:term_`

`expansion(`*Term1*`[,`*Layout1*`],`*Term2*`[,`*Layout2*`])` is called first, and only if it fails is the standard expansion used.

`term_expansion(+`*Term1*`,?`*TermOrTerms*`)` *[Hook]*
`term_expansion(+`*Term1*`,+`*Layout1*`,?`*TermOrTerms*`,?`*Layout2*`)` *[Hook]*
`user:term_expansion(+`*Term1*`,?`*TermOrTerms*`)`
`user:term_expansion(+`*Term1*`,+`*Layout1*`,?`*TermOrTerms*`,?`*Layout2*`)`

Defines transformations on terms read while a program is consulted or compiled. It is called for every *Term1* read, including at end of file, represented as the term `end_of_file`. If it succeeds, *TermOrTerms* is used for further processing, otherwise the default grammar rule expansion is attempted. It is often useful to let a term expand to a list of directives and clauses, which will then be processed sequentially.

The 4 arguments version also defines transformations on the layout of the term read, so that the source-linked debugger can display accurate source code lines if the transformed code needs debugging. *Layout1* is the layout corresponding to *Term1*, and *Layout2* should be a valid layout of *TermOrTerms* (see Section 7.1.3 [Term I/O], page 108).

For accessing aspects of the load context, e.g. the name of the file being compiled, the predicate `prolog_load_context/2` (see Section 7.6 [State Info], page 137) can be used.

`user:term_expansion/[2,4]` may also be used to transform queries entered at the terminal in response to the '`| ?- `' prompt. In this case, it will be called with *Term1* = `?-(`*Query*`)` and should succeed with *TermOrTerms* = `?-(`*ExpandedQuery*`)`.

`goal_expansion(+`*Goal*`,+`*Module*`,?`*NewGoal*`)` *[Hook]*
`user:goal_expansion(+`*Goal*`,+`*Module*`,?`*NewGoal*`)`

Defines transformations on goals while clauses are being consulted, compiled or asserted, *after* any processing by `user:term_expansion/[2,4]` of the terms being read in. It is called for every simple *Goal* encountered in the calling context *Module*. If it succeeds, *Goal* is replaced by *NewGoal*, otherwise *Goal* is left unchanged. *NewGoal* may be an arbitrarily complex goal, and `user:goal_expansion/3` is recursively applied to its subgoals.

This predicate is also used to resolve meta-calls to *Goal* at runtime via the same mechanism. If the transformation succeeds, *NewGoal* is simply called instead of *Goal*. Otherwise, if *Goal* is a goal of an existing predicate, that predicate is invoked. Otherwise, error recovery is attempted by `user:unknown_predicate_handler/3` as described below.

`user:goal_expansion/3` can be regarded as a macro expansion facility. It is used for this purpose to support the interface to attributed variables in `library(atts)`, which defines the predicates *M*`:get_atts/2` and *M*`:put_atts/2` to access module-specific variable attributes. These "predicates" are actually implemented via the `user:goal_expansion/3` mechanism. This has the effect that calls to the interface predicates are expanded at compile time to efficient code.

For accessing aspects of the load context, e.g. the name of the file being compiled, the predicate `prolog_load_context/2` (see Section 7.6 [State Info], page 137) can be used.

`phrase(`*:Phrase*`,`*?List*`)`
`phrase(`*:Phrase*`,`*?List*`,+`*Remainder*`)`
The list *List* is a phrase of type *Phrase* (according to the current grammar rules), where *Phrase* is either a non-terminal or more generally a grammar rule body. *Remainder* is what remains of the list after a phrase has been found. If called with 2 arguments, the remainder has to be the empty list.

`'C'(`*?S1*`,`*?Terminal*`,`*?S2*`)`
Not normally of direct use to the user, this built-in predicate is used in the expansion of grammar rules (see above). It is defined as if by the clause `'C'([X|S], X, S).`

7.1.3 Input and Output of Terms

Most of the following predicates come in two versions, with or without a stream argument. Predicates without a stream argument operate on the current input or output stream, depending on context. Predicates with a stream argument can take stream reference or an alias in this argument position, the alias being replaced by the stream it was associated with.

Some of these predicates support a notation for terms containing multiple occurrences of the same subterm (cycles and DAGs). The notation is `@(`*Template*`,`*Substitution*`)` where *Substitution* is a list of *Var*=*Term* pairs where the *Var* occurs in *Template* or in one of the *Terms*. This notation stands for the instance of *Template* obtained by binding each *Var* to its corresponding *Term*. The purpose of this notation is to provide a finite printed representation of cyclic terms. This notation is not used by default, and `@/2` has no special meaning except in this context.

`read(`*?Term*`)` *[ISO]*
`read(+`*Stream*`,`*?Term*`)` *[ISO]*
The next term, delimited by a full-stop (i.e. a `.`, possibly followed by layout text), is read from *Stream* and is unified with *Term*. The syntax of the term must agree with current operator declarations. If a call `read(`*Stream*, *Term*`)` causes the end of *Stream* to be reached, *Term* is unified with the term `end_of_file`. Further calls to `read/2` for the same stream will then raise an exception, unless the stream is connected to the terminal. The characters read are subject to character-conversion, see below.

`read_term(`*?Term*`,+`*Options*`)` *[ISO]*
`read_term(+`*Stream*`,`*?Term*`,+`*Options*`)` *[ISO]*
Same as `read/[1,2]` with a list of options to provide extra control or information about the term. *Options* is a list of zero or more of: