An Introduction to ISETL

Gary Marc Levin
Clarkson University
Dept of Math and Computer Science

1989

This introduction is intended for people who have had no previous experience with SETL or ISETL, but who are reasonably comfortable with learning a new programming language. Very few examples are given in the description, but a large number of examples is distributed with the software.

This documentation appears as an appendix in a discrete math text written by Nancy Baxter, Ed Dubinsky, and Gary Levin. That text uses ISETL as a tool for teaching discrete mathematics.

1 Running ISETL

ISETL is an interpreted, interactive version of the programming language SETL. Written in C, it is invoked by typing a command line with the executable name, say `isetl`, along with optional file names that are discussed below.[1]

There is no compiler for ISETL. When ISETL is running, it prompts for input with the character "`>`". Input consists of a sequence of expressions (each terminated by a semicolon "`;`"), statements, and programs. Each input is acted upon as soon as it is entered. The result of this action is explained below. In the case of expressions, the result includes its value being sent to standard output. If you have not completed your entry, you will receive the prompt "`>>`", indicating that more is expected.

1. ISETL is exited by typing "`!quit`". It may also be exited by ending the standard input. (In Unix, this is done by typing ctrl-D).

2. A common mistake is omitting the semicolon after an expression. ISETL will wait until it gets a semicolon before proceeding — even if a carriage return is entered. The doubled prompt "`>>`" indicates that ISETL is expecting more input.

3. ISETL can get its input from sources other than the standard input.

[1] The Macintosh version is clickable. Therefore, statements about the command line do not apply to the Mac.

(a) If there is a file with the name "`.isetlrc`" in the current directory, then the first thing ISETL will do is read this file. [2]

(b) Next, if the command line has any file names listed, ISETL will read each of these in turn. [3]

Thus, if the command line reads,

```
isetl file.1 foo bar
```

ISETL will first read from the "`.isetlrc`" file, if it exists, and then from the file "`file.1`", then from "`foo`", and then from "`bar`". Finally, it is ready for input from the terminal. (This is only available under UNIX and MSDOS.)

(c) If there is a file available with the name, say "`file.2`" and ISETL is given (at any time), the input,

```
!include file.2
```

(notice that there is *no* semicolon), then it will take its input from "`file.2`" before being ready for any further input. The material in such a file is treated *exactly as if it were typed directly to the terminal,* and it can be followed by any additional information that the user would like to enter.

Consider the following (rather contrived) example: Suppose that the file "`file.3`" contained the following data,

```
5, 6, 7, 3, -4, "the"
```

Then if the user typed,

```
> seta := {
>> !include file.3
!include file.3 completed
>> , x };
```

the effect would be exactly the same as if the user had entered,

```
> seta := {5, 6, 7, 3, -4, "the", x};
```

The line "`!include file.3 completed`" comes from ISETL and is always printed after an "`!include`".

4. Comments

If a dollar sign "`$`" appears on a line, then everything that appears until the next carriage return is ignored by ISETL.

[2] On systems that use file extentions, this file will be called "`isetl.ini`".

[3] This feature is system dependent. Currently, neither the VMS nor the Macintosh version has this feature.

5. After a program or statement has executed, the values of global variables persist. The user can then evaluate expressions in terms of these variables. (See below for more detail on scope.)

2 Characters and Keywords

2.1 Character set.

The following is a list of characters used by ISETL.

```
] [ ; :  = | } { ) ( .  # ?  * / + - _ " > < % ~ ,

a - z   A - Z   0 - 9
```

In addition, the following character-pairs are used.

```
:=   ..   **   /=   <=   >=
```

2.2 Keywords.

The following is a list of ISETL keywords.

```
and     from   newat    subset
div     fromb  not      take
do      frome  notin    then
else    func   of       to
elseif  if     om (OM)  true
end     impl   or       value
exists  in     print    while
false   less   program  where
for     local  read     with
forall  mod    return
```

3 Identifiers.

1. An identifier is a sequence of alphanumeric characters along with the underscore, "_". It must begin with a letter. Upper or lower case may be used, and ISETL preserves the distinction. (I.e.: `a_good_thing` and `A_Good_Thing` are both legal and are different.)

2. An identifier serves as a variable and can take on a value of any ISETL data type. The type of a variable is entirely determined by the value that is assigned to it and changes when a value of a different type is assigned to it.

4 Simple Data Types.

4.1 Integers.

1. There is no limit to the size of integers. [4]

2. An `integer` constant is a sequence of one or more digits. It represents an unsigned integer.

3. On output, long integers may be broken to accomodate limited line length. This format is not yet available for input.

4.2 Floating_Point Numbers.

1. The possible range of `floating_point` numbers is machine dependent. At a minimum, the values will have 5 place accuracy, with a range of approximately 10^{38}.

2. A `floating_point constant` is a sequence of one or more digits, followed by a period, "`.`", followed by one or more digits. Thus, `2.0` is legal, and `2.` and `.5` are illegal.

   It may be followed by an exponent. An exponent consists of one of the characters "`e`", "`E`", "`f`", "`F`" followed by a signed or unsigned `integer`. The value of a `floating_point` constant is determined as is usual with scientific notation. As with integers, it is unsigned. Hence, for example, `0.2`, `2.0e-1`, `20.0e-2` are all equivalent.

3. Different systems use different printed representations when floating point values are out of the machine's range. For example, when the value is too large, the Mac prints "+++++" and the Sun prints "Infinity".

4.3 Booleans.

1. A `Boolean` constant is one of the keywords `true` or `false`, with the obvious meaning for its value.

4.4 Strings.

1. A `string` constant is any sequence of characters preceded and followed by a double quote, "`"`". The double quote may not be used in a `string`. A `string` may not be split across lines. Large `string`s may be constructed using the operation of concatenation.

   Warning: Construction of `string`s containing double quote is possible, but the output of such `string`s may cause difficulties.

[4] No practical limit. Actually limited to about 20,000 digits per integer.

4.5 Atoms.

1. Atoms are "abstract points". They have no identifying properties other than their individual existence.

4.6 Files.

1. `Files` are ISETL values that are created as a result of applying one of the pre-defined functions `openr`, `opena`, `openw`. They correspond to external files in the operating system environment.

4.7 Undefined.

1. The data type undefined has a single value, `OM`. Any identifier that has not been given a value has the value `OM`. It may also be entered as `om`.

5 Compound Data Types.

5.1 Sets.

1. Only finite `sets` may be represented in ISETL. The elements may be of any type, mixed heterogeneously. Elements occur at most once per `set`.

2. The order of elements is not significant in a `set` and printing the value of a `set` twice in succession could display the elements in different orders.

3. `OM` may not be an element of a `set`. Any set that would contain `OM` is considered to be undefined.

4. An expression, or several expressions separated by commas, and, in either case, enclosed in { } evaluates to the `set` whose elements are the values of the enclosed expressions.

5. The empty `set` is denoted by { }.

6. There are syntactic forms, explained in the grammar, for a finite `set` that is an arithmetic progression of `integer`s, and also for a finite `set` obtained from a `set` former in standard mathematical notation.

   For example, the value of the following expression

   ```
   { x + y :  x in {-1,-3..-100}, y in {-1,-3..-100} | x /= y};
   ```

   is the `set` of all sums of two different odd negative `integers` larger than -100.

5.2 Tuples.

1. A `tuple` is an infinite sequence of components, of which only a finite number are defined. The components may be of any type, mixed heterogeneously. The values of components may be repeated.

2. The order of the components of a `tuple` is significant. By treating the `tuple` as a function over the integers, you can extract individual components and slices of the `tuple`.

3. `OM` is a legal value for a component.

4. An expression, or several expressions separated by commas, and, in either case, enclosed by `[ ]`, evaluates to a `tuple` whose defined components are the values of the enclosed expressions.

5. The empty `tuple` is denoted by `[ ]`.

6. The syntactic forms for `tuple`s of finite arithmetic progressions and `tuple` formers are similar to those provided for `sets`. The only difference is the use of square, rather than curly, brackets.

7. The length of a `tuple` is the largest index (counting from 1) for which a component is defined (that is, is not equal to `OM`). It can change at run-time.

5.3 Maps.

`Map`s form a subclass of `sets`.

1. A `map` is a `set` that is either empty or whose elements are all ordered pairs. An ordered pair is a `tuple` whose first two components and no others are defined.

2. There are two special operators for evaluating a `map` at a point in its domain. Suppose that `F` is a `map`.

   (a) `F(EXPR)` will evaluate to the value of the second component of the ordered pair whose first component is the value of `EXPR`, provided there is exactly one such ordered pair in `F`; otherwise, it evaluates to `OM`.

   (b) `F{EXPR}` will evaluate to the `set` of all values of second components of ordered pairs in `F` whose first component is the value of `EXPR`. If there are none such, its value is the empty `set`.

3. A `map` in which no value appears more than once as the first component of an ordered pair is called a *single-valued map* or *smap*; otherwise, the `map` is called a *multi-valued map* or *mmap*.

6 Funcs.

1. A `func` is an ISETL value that may be applied to zero or more values passed to it as arguments. It then returns a value specified by the definition of the `func`. Because it is a value, an ISETL `func` can be assigned to an identifier, passed as an argument, etc. Evaluation of an ISETL `func` can have side-effects determined by the statements in the definition of the `func`. Thus, it also serves the purpose of what is often called a procedure.

2. A `func` is the computational representation of a function, as a `map` is the ordered pair representation, and a `tuple` is the sequence representation. Just as `tuple`s and `map`s may be modified at a point by assignment, so can `func`s. However, if the value at a point is structured, you may not modify that at a point as well.

```
x := func(i); return char(i); end;
x(97) := "b";
x(97)(1) := "abc";
```

   `x` may be modified at a point. The assignment to `x(97)` is legal. However, the following assignment is not supported at this time, because you are trying to modify the structure of the value returned.

3. A number of functions have been pre-defined as `funcs` in ISETL. A list of their definitions is given later in this document. These are not keywords and may be changed by the user. They may not be modified at a point, however.

4. It is possible for the user to define her/his own `func`. This is done with the following syntax,

```
func(list-of-parameters);
    local list-of-local-ids;
    value list-of-global-ids;
    statements;
end
```

   (a) The declaration of local-ids may be omitted if no locals are needed. The declaration of global-ids represents global variables whose current values are to be remembered and used at the time of function invocation; these may be omitted if not needed. The list-of-parameters may be empty, but the pair of parentheses must be present.

   (b) Parameters and local-ids are local to the `func`. See below for a discussion of scope.

(c) The syntax described above is for an *expression* of type `func`. As with any expression, it has an existence as a value, but no name. Thus, the definition will typically be part of an assignment statement, or passed as a parameter. As a very simple example, consider:

```
cube_plus := func(x,y);
                 return x**3 + y;
             end;
```

After having executed this input, ISETL will evaluate an expression such as `cube_plus(2,5);` as `13`.

(d) Parameters are passed by value. If there are too many arguments, the extra arguments are ignored. If there are too few arguments, the extra parameters are assigned the value `OM`.

(e) Scope is lexical (static) with retention. *Lexical* means that references to global variables are determined by where the `func` was created, not by where it will be evaluated. *Retention* means that even if the scope that created the `func` has been exited, its variables persist and can be used by the `func`.

By default, references to global variables will use the value of the variable at the time the function is invoked. The `value` declaration causes the value of the global variable *at the time the* `func` *is created* to be used.

(f) Here is a more complicated example of the use of `func`. As defined below, `compose` takes two functions as arguments and creates their functional composition. The functions can be any ISETL values that may be applied to a single argument; e.g. `func, tuple, smap`.

```
compose := func (f,g);
                return func (x);
                            return f(g(x));
                       end;
           end;
twice :=   func (a);
                return 2*a;
           end;
times4 :=  compose(twice,twice);
```

Then the value of `times4(3)` would be `12`. The value of `times4` needs to refer to the values of `f` and `g`, and they remain accessible to `times4`, even though `compose` has returned.

(g) Finally, here are some examples of functions modified at a point, and functions that capture the current value of a global.

```
f := func (x);
          return x + 4;
     end ;
gs := [ func(x); value N; return x+3*N; end :
                        N in [1..3] ];
f(3) := 21;
```

After this is executed, `f(1)` is `5`, `f(2)` is `6`, but `f(3)` is `21`. `gs(2)(4)` is `10` (`4+3*2`).

7 The ISETL Grammar.

7.1 Terminology.

1. In what follows, the symbols `ID`, `INTEGER`, `FLOATING_POINT`, `BOOLEAN`, and `STRING` refer to the terms identifier and `integer`, `floating point` number, `Boolean`, and `string` constants, which have been specified above. Any other symbol in capital letters is explained in the grammar.

2. A statement enclosed in double quotes is an extra-grammatical explanation.

3. Spaces are not allowed within any of the character pairs listed in the section on Characters and Keywords, nor within an `ID`, `INTEGER` constant, `FLOATING_POINT` constant, or keyword. Spaces are required between keywords, `ID`s, `INTEGER` constants, and `FLOATING_POINT` constants.

4. ISETL treats carriage returns and tabs as spaces. Any input can be spread across lines without changing the meaning, and ISETL will not consider it to be complete until a semicolon "`;`" is entered. The only exceptions to this are the `!include` input, which is ended with a carriage return, and the fact that a quoted `string` cannot be typed on more than one line.

7.2 The Grammar.

The annotated grammar below is presented in three columns. The first column is the symbol being defined, the second column contains a list of the possibilities for that symbol, and the third column, when present, provides additional explanation.

`INPUT`

Can be typed at the terminal or read from a file.

`PROGRAM`

`STMT`

`EXPR ;`

The `EXPR` is evaluated and the result is sent to standard output.

`PROGRAM`

Usually read from a file.

`program ID ; STMTS ; end ;`

Of course, it can appear on several lines with the usual indented format.

`STMTS`

"One or more instances of `STMT`"

`STMT`

`LHS := EXPR ;`

The left hand side is evaluated to determine the target(s) for the assignment, and the right hand side is evaluated. Then the assignment is made. If there are some targets for which there are no values to be assigned, then they receive the value, `OM`. If there are values to be assigned but no corresponding targets, then the values are ignored.

Examples:

```
a := 4;
```

`a` is changed to contain the value `4`.

```
[a,b] := [1,2];
```

`a` is assigned `1` and `b` is assigned `2`.

```
[x,y] := [y,x];
```

Swap `x` and `y`.

```
f(3) := 7;
```

If `f` is a `tuple`, then the effect of this statement is to assign `7` as the value of the third component of `f`. If `f` is a `map`, then its effect is to replace all pairs beginning with `3` by the pair `[ 3 , 7 ]` in the set of ordered pairs `f`. If `f` is a `func`, then `f(3)` will be `7`, and all other values of `f` will be as they were before the assignment.

`EXPR ;`

The expression is evaluated and the value ignored.

`for ITERATOR do STMTS end ;`

The `STMTS` are executed for each instance generated by the iterator. Optionally, one may close with `end for`.

`while EXPR do STMTS end ;`

`EXPR` must evaluate to a `BOOLEAN` value. `EXPR` is evaluated and the `STMTS` are executed repetitively as long as this value is equal to `true`. Optionally, one may close with `end while`.

`take LHS frome LHS ;`

The second `LHS` must evaluate to a `tuple` (or a `string`). The value of its last defined component (or last character) is assigned to the first `LHS` and replaced by `OM` in the `tuple` (deleted from the `string`).

`take LHS fromb LHS ;`

The second LHS must evaluate to a tuple (or a string). The value of its first component (defined or not) (first character) is assigned to the first LHS and all components of the tuple (characters of the string) are shifted left one place. That is, the new value of the i^{th} component is the old value of the $(i+1)^{st}$ component $(i = 1, 2, ...)$.

`take LHS from LHS ;`

The second LHS must evaluate to a set. An arbitrary element of the set is assigned to the first LHS and removed from the set.

`read LHS_LIST ;`

ISETL gives a question mark "?" prompt and waits until an expression has been entered. This EXPR is evaluated and the result is assigned to the first item in LHS_LIST. This is repeated for each item in LHS_LIST. *Note*: If a read statement appears in an !include file, then ISETL will look at the next input in that file for the expression(s) to be read.

`read LHS_LIST from EXPR ;`

This is the same as read LHS_LIST; except that EXPR must have a value of type file. The values to be read are then taken from the external file specified by the value of EXPR. If there are more values in the file than items in LHS_LIST, then the extra values are left to be read later. If there are more items in LHS_LIST than values in the file, then the extra items are assigned the value OM. In the latter case, the function eof will return true, when given the file as parameter. Before this statement is executed, the external file in question must have been opened for reading by the pre-defined function openr (see the section on file functions).

`print EXPR_LIST ;`

Each expression in EXPR_LIST is evaluated and printed on standard output. The output values are formated to show their structure, with line breaks at reasonable positions and meaningful indentation. The only problem is with very long strings. If a string is too long to fit on the screen, the front of the string is printed, followed by ellipses and a length indicator.

`print EXPR_LIST to EXPR ;`

Again EXPR must be a value of type file. The values are written to the external file specified by the value of EXPR. Before executing this statement, the external file in question must have been opened for printing by one of the pre-defined functions openw or opena (see the section on file functions).

`return ;`

return is only meaningful inside a func. Its effect is to terminate execution of a func and return the value OM to the caller. ISETL inserts return; just before the end of every func. If return appears at the "top level", i.e. as input to the terminal, a run time error will occur.

`return EXPR ;`

Same as `return;` except that `EXPR` is evaluated and its value is returned as the value of the `func`.

`IF_STMT`

`if EXPR then STMTS ELSE_IFS ELSE_PART end`
The `EXPR`s after `if` and `elseif` are evaluated in order until one is found to be `true`. The `STMTS` following the associated `then` are executed. If no `EXPR` is found to be `true`, the `STMTS` in the `ELSE_PART` are executed. In this last case, if the `ELSE_PART` is omitted, this statement has no effect. Optionally, one may close with `end if`.

`ELSE_IFS`

"zero or more repetitions of the following"

`elseif EXPR then STMTS`

`ELSE_PART`

"may be omitted"

`else STMTS`

`ITERATOR`

`ITER_LIST`

`ITER_LIST | EXPR`
`EXPR` must evaluate to a `BOOLEAN`. Generates only those instances generated by `ITER_LIST` for which the value of `EXPR` is `true`.

`ITER_LIST`

"one or more `SIMPLE_ITERATOR`s separated by commas."
Generates all possible instances for every combination of the `SIMPLE_ITERATOR`s. The first `SIMPLE_ITERATOR` advances most slowly. Subsequent iterators may depend on previously bound values.

`SIMPLE_ITERATOR`
A `SIMPLE_ITERATOR` generates a number of instances for which an assignment is made. These assignments are local to the iterator and when it is exited, all previous values of `ID`s that were used as local variables are restored. That is, these `ID`s are "bound variables" whose scope is the construction containing the iterator. (e.g., `for`-loops, quantifiers, formers, etc.)

`BOUND_LIST in EXPR`
`EXPR` must evaluate to a `set`, `tuple`, or `string`. The instances generated are all possibilities in which each `BOUND` in `BOUND_LIST` is assigned a value that occurs in `EXPR`.

`BOUND = ID ( BOUND_LIST )`

Here ID must have the value of an smap, tuple, or string, and BOUND_LIST must have the correct number of occurrences of BOUND corresponding to the parameters of ID. The resulting instances are those for which all occurrences of BOUND in BOUND_LIST have all possible legal values and BOUND is assigned the corresponding value.

BOUND = ID { BOUND_LIST }
Same as the previous one for the case in which ID is an mmap.

BOUND_LIST

"one or more BOUND, separated by commas"

BOUND

~
Corresponding value is thrown away.

ID
Corresponding value is assigned to ID.

[BOUND_LIST]
Corresponding value must be a tuple, and elements of the tuple are assigned to corresponding elements in the BOUND_LIST.

SELECTOR
A tuple, string, map, or func (pre- or user- defined) may be followed by a SELECTOR, which has the effect of specifying a value or group of values in the range of the tuple, string, map, or func. Not all of the following SELECTORs can be used in all four cases.

{ EXPR_LIST }
Must be used with an mmap. If the EXPR_LIST has more than one element it is equivalent to what it would be if the list were enclosed in square brackets, [ ]. Thus a function of several variables is interpreted as a function of one variable — the vector (tuple) whose components are the individual variables.

(EXPR_LIST)
Must be used with an smap, tuple, string, or func. If it is used with a tuple or string, then EXPR_LIST can only have one element, which must evaluate to a positive integer. Same remark as previous for the case in which the list has more than one element.

(EXPR .. EXPR)
Must be used with a tuple or string, and both instances of EXPR must evaluate to a positive integer.

The value is the slice of the original tuple or string in the range specified by the two occurrences of EXPR. There are some special rules in this case. To describe them, suppose that the first EXPR has the value a and the second has the value b so that the selector is (a..b).

$a \leq b$	value is the `tuple` or `string` with components defined only at the `integer`s from 1 to $b - a + 1$, inclusive. The value of the i^{th} component is the value of the $(a + i - 1)^{st}$ component of the value of `EXPR`.
$a = b + 1$	value is the null `tuple`.
$a > b + 1$	run-time error.

`( .. EXPR )`
Means the same as `(1 .. EXPR)`.

`( EXPR .. )`
Means the same as `( EXPR .. EXPR )` where the second `EXPR` is equal to the length of the `tuple` or `string`.

`( )`
Used with a `func` that has no parameters. It also works with an `smap` with `[ ]` in its domain.

`FORMER`
Generates the values to be used in expressions that evaluate to a `set` or a `tuple`.

`EXPR : ITERATOR`
The value of `EXPR` for each instance generated by the `ITERATOR`.

`EXPR_LIST`
Values are explicitly listed.

`EXPR .. EXPR`
Both occurrences of `EXPR` must evaluate to `integer`. Generates all `integer`s beginning with the first `EXPR` and increasing by `1` for as long as the second `EXPR` is not exceeded. If the first `EXPR` is larger than the second, no values are generated.

`EXPR , EXPR .. EXPR`
All three occurrences of `EXPR` must evaluate to `integer`. Generates all `integer`s beginning with the first `EXPR` and incrementing by the value of the second `EXPR` minus the first `EXPR`. If this difference is positive, it generates those `integer`s that are not greater than the third `EXPR`. If the difference is negative, it generates those `integer`s that are not less than the third `EXPR`. If the difference is zero, no `integers` are generated.

`LHS`
The target for anything that has the effect of an assignment.

`ID`

`LHS SELECTOR`
`LHS` must evaluate to a `tuple`, `string`, or `map`. `LHS` is modified by replacing the components designated by selector.

`[ LHS_LIST ]`

`LHS_LIST`

"One or more instances of `LHS`, separated by commas"

Thus the input,

```
[A, B, C] := [1, 2, 3];
```

has the effect of replacing `A` by `1`, `B` by `2`, and `C` by `3`.

"Any `LHS` in the list can be replaced by `~` "

The effect is to omit any assignment to a `LHS` that has been so replaced. Thus the input,

```
[A, ~ , C] := [1, 2, 3];
```

replaces `A` by `1`, `C` by `3`.

`EXPR`

The first few in the following list are values of simple data types and they have been discussed before.

`ID`

`INTEGER`

`FLOATING_POINT`

`STRING`

`true`

`false`

`OM`

`newat`

The value is a new `atom`, different from any other `atom` that has appeared at the time it is invoked.

`FUNC_CONST`

A user-defined `func`.

`( EXPR )`

Any expression can be enclosed in parentheses. The value is the value of `EXPR` enclosed in parentheses.

`[ FORMER ]`

Evaluates to the `tuple` of those values generated by `FORMER` in the order that former generates them.

{ `FORMER` }

Evaluates to the `set` of those values generated by `FORMER`.

`# EXPR`

`EXPR` must be a `set`, `tuple`, or `string`. The value is the cardinality of the `set`, the length of the `tuple`, or the length of the `string` as the case may be.

`EXPR SELECTOR`

EXPR must evaluate to an ISETL value that is, in the general sense, a function. That is, it must be a map, tuple, string, or func. See SELECTOR.

EXPR . ID EXPR

This is equivalent to ID(EXPR,EXPR). (Space after the "." is optional.)

EXPR ? EXPR

The value of the first EXPR, if it is not OM; otherwise the value of the second EXPR.

In general, arithmetic and comparisons may mix integers and floating_point. The result is an integer if both operands are integers and floating_point, otherwise. For simplicity, we will use the term number to mean a value that is either integer or floating_point.

EXPR ** EXPR

The values of the two expressions must be numbers. The operation is exponentiation.

EXPR * EXPR

If both instances of EXPR evaluate to numbers, this is multiplication. If both evaluate to sets, this is intersection. If one instance of EXPR evaluates to integer, and the other to a tuple or string, then the value is the tuple or string, concatenated with itself the integer number of times, if the integer is positive; and the empty tuple or string, if the integer is less than or equal to zero.

EXPR / EXPR

Both instances of EXPR must evaluate to numbers. The value is the result of division, and is of type floating_point.

EXPR mod EXPR

Both instances of EXPR must evaluate to integer and the second must be positive. The result is the remainder, and the following condition is always satisfied,

$$0 \leq a \,\texttt{mod}\, b < b$$

.

EXPR div EXPR

Both instances of EXPR must evaluate to integer, and the second must be non-zero. The value is integer division defined by the following two relations,

$$\begin{aligned} (a \,\texttt{div}\, b) * b + (a \,\texttt{mod}\, b) &= 0 \qquad \text{for } b > 0 \\ a \,\texttt{div}\, (-b) &= -(a \,\texttt{div}\, b) \qquad \text{for } b < 0\,. \end{aligned}$$

EXPR + EXPR

If both instances of EXPR evaluate to numbers, this is addition. If both instances of EXPR evaluate to sets, then this is union. If both instances of EXPR evaluate to tuples or strings, then this is concatenation.

`EXPR - EXPR`
: If both instances of `EXPR` evaluate to `number`s, this is subtraction. If both instances of `EXPR` evaluate to `set`s, then this is `set` difference.

`EXPR with EXPR`
: The value of the first `EXPR` must be a `set` or `tuple`. If it is a `set`, the value is that `set` with the value of the second `EXPR` added as an element. If it is a `tuple`, the value of the second `EXPR` is assigned to the value of the first component after the last defined component of the `tuple`.

`EXPR less EXPR`
: The value of the first `EXPR` must be a `set`. The value is that `set` with the value of the second `EXPR` removed, if it was present.

`EXPR = EXPR`
: The test for equality of any two ISETL values.

`EXPR /= EXPR`
: Negation of `EXPR=EXPR`.

`EXPR < EXPR`

`EXPR > EXPR`

`EXPR <= EXPR`

`EXPR >= EXPR`
: For all the above inequalities, both instances of `EXPR` must evaluate to the same type, which must be `number` or `string`. For `number`s, this is the test for the standard arithmetic ordering, and for `string`s, it is the test for lexicographic ordering.

`EXPR in EXPR`
: The second `EXPR` must be a `set`, `tuple`, or `string`. For `set`s and `tuple`s, this is the test for membership. For `string`s, it is the test for substring.

`EXPR notin EXPR`
: Negation of `EXPR in EXPR`.

`EXPR subset EXPR`
: Both instances of `EXPR` must be `set`s. This is the test for the value of the first `EXPR` to be a subset of the value of the second `EXPR`.

`EXPR and EXPR`
: Both instances of `EXPR` must evaluate to `BOOLEAN`. Logical conjunction. If the left operand is `false`, the right operand is not evaluated.

`EXPR or EXPR`
: Both instances of `EXPR` must evaluate to `BOOLEAN`. Logical disjunction. If the left operand is `true`, the right operand is not evaluated.

`EXPR impl EXPR`
: Both instances of `EXPR` must evaluate to `BOOLEAN`. Logical implication.

`not EXPR`
: `EXPR` must evaluate to `BOOLEAN`. Logical negation.

`+ EXPR`

EXPR must evaluate to a number. Identity function.

- EXPR

EXPR must evaluate to a number. Negative of EXPR.

% BINOP EXPR

EXPR must evaluate to a set, tuple, or string. Say that the elements in EXPR are x1, x2,...,xN (N=#EXPR). If N=0, then the value is OM. If N=1, then the value is the single element. Otherwise, %BINOP EXPR equals

```
x1 BINOP x2 BINOP ... BINOP xN
```

associating to the left.

If EXPR is a set, then the selection of elements is made in arbitrary order, otherwise it is made in the order of the components of EXPR.

EXPR % BINOP EXPR

The second instance of EXPR must evaluate to a set, tuple, or string. If the first EXPR is a, and the values in the second are x1, x2,...,xN as above, then the value is:

```
a BINOP x1 BINOP x2 BINOP ... BINOP xN
```

associating to the left.

exists ITER_LIST | EXPR

EXPR must evaluate to a BOOLEAN. If ITER_LIST generates at least one instance in which EXPR evaluates to true, then the value is true; otherwise it is false.

forall ITER_LIST | EXPR

EXPR must evaluate to a BOOLEAN. If every instance generated by ITER_LIST is such that EXPR evaluates to true, then the value is true; otherwise it is false.

EXPR where DEFNS end

The value is the value of the EXPR preceeding where, evaluated in the current environment with the IDs in the DEFNS added to the environment and initialized to the corresponding EXPRs. The scope of the IDs is limited to the where expression. The DEFNS can modify IDs defined in earlier DEFNS in the same where expression.

`DEFNS`

"Zero or more instances of `DEFN`"

`DEFN`

`BOUND := EXPR ;`

`ID SELECTOR := EXPR ;`

`EXPR_LIST`

"One or more instances of `EXPR` separated by commas"

`BINOP`

"Any binary operator (`+`, `-`, `*`, `**`, `/`, `div`, `mod`, `with`, `less`, `and`, `or`, `impl`) or an `ID` whose value is a function of two parameters."

`FUNC_CONST`

`FUNC_HEAD LOCALS VALUES STMTS end`
: This is the syntax for user-defined `funcs`. Optionally, one may close with `end func`.

`FUNC_HEAD`

`func ( ID_LIST ) ;`
: In this case, there are parameters.

`func ( ) ;`
: In this case, there are no parameters.

`LOCALS`

`local ID_LIST ;`
: The appearance of this in `FUNC_CONST` is optional.

`VALUES`

`value ID_LIST ;`
: The appearance of this in `FUNC_CONST` is optional. `VALUES` and `LOCALS` may be repeated and appear in any order.

`ID_LIST`

"One or more instances of `ID` separated by commas."

8 Pre-defined Functions

8.1 Functions on integers.

In each of the following, `EXPR` must evaluate to `integer`.

1. `even(EXPR)`. The test for even.
2. `odd(EXPR)`. The test for odd.
3. `float(EXPR)`.

   The value is the value of `EXPR` converted to `floating_point`.
4. `char(EXPR)`. The value is the one-character `string` whose (machine dependent) index is the value of `EXPR`.

8.2 Functions on floating point numbers.

In each of the following, `EXPR` must evaluate to `floating_point`.

1. `ceil(EXPR)`. The value is the smallest `integer` not smaller than the value of `EXPR`.
2. `floor(EXPR)`. The value is the largest `integer` not larger than the value of `EXPR`.
3. `fix(EXPR)`. The value is the same as `floor(EXPR)` if `EXPR>=0`, and the same as `ceil(EXPR)` if the value of `EXPR<=0`. In other words, the fractional part is discarded.

8.3 Functions on Sets.

In each of following, `EXPR` must evaluate to a `set`.

1. `arb(EXPR)`. An element of `EXPR`, selected arbitrarily. If the value of `EXPR` is { }, then the value of the function is `OM`.
2. `pow(EXPR)`. The value is the `set` of all subsets of the value of `EXPR`.
3. `npow(EXPR,EXPR)`. One `EXPR` must be a `set`, and the other a non-negative `integer`. The value is the `set` of all subsets of the `set` whose cardinality is equal to the `integer`. This `func` is defined in `.isetlrc`. This will eventually change. If you do not have the standard `.isetlrc`, you do not have this `func`.

8.4 Functions on maps.

In each of the following, `EXPR` must evaluate to a `map`.

1. `domain(EXPR)`. The value is the `set` of all values that appear as the first component of an element of the value of `EXPR`.

2. `image(EXPR)`. The value is the `set` of all values that appear as the second component of an element of the value of `EXPR`.

8.5 Standard mathematical functions.

1. Each of the following takes a single `floating_point` argument. The value is a `floating_point` approximation to the value of the corresponding mathematical function.

   `exp`, `ln`, `log`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.

2. In each of the following, `EXPR` must evaluate to `integer` or `floating_point`. The value is the value of the mathematical function in the same type as the value of `EXPR`.

   (a) `sgn(EXPR)`. If `EXPR` is positive, then `1`; if `EXPR` is zero, then `0`; otherwise `-1`.

   (b) `random(EXPR)`. The value is a `number` selected at random in the interval from `0` to the value of `EXPR`, inclusive. There has been no statistical study made of the generators. Don't depend on them for highly sensitive work.

   (c) `randomize(EXPR)`. If EXPR is an `integer`, this resets the random number generator. This may be used to re-create a sequence of random numbers. On some systems, a non-integer argument resets the random number generator in an uncontrolled fashion.

3. In each of the following, both occurrences of `EXPR` must evaluate to `integer`, `floating_point`, or `string`. The value of the function is the value of the mathematical function in the same type as was the values of the two occurrences of `EXPR`.

   (a) `max(EXPR,EXPR)`.

   (b) `min(EXPR,EXPR)`.

8.6 Type Testers.

In each of the following, the value of `EXPR` can be any ISETL data type. The function is the test for the value of `EXPR` being the type indicated.

1. `is_atom(EXPR)`.

2. `is_boolean(EXPR)`.
3. `is_integer(EXPR)`.
4. `is_file(EXPR)`.
5. `is_floating(EXPR)`.
6. `is_func(EXPR)`.
7. `is_map(EXPR)`.
8. `is_number(EXPR)` – `true` for `integer` and `floating_point`.
9. `is_set(EXPR)`.
10. `is_string(EXPR)`.
11. `is_tuple(EXPR)`.
12. `is_om(EXPR)`.
13. `is_defined(EXPR)`. Negation of `is_om`.

8.7 Input/Output Functions.

1. In each of the following functions, the value of `EXPR` must be a `string` that is a file name consistent with the operating system's naming conventions. The value of the function has ISETL type `file` and may be in `read...from...` and `print...to...` statements to refer to that file.

   (a) `openr(EXPR)`. If the file named by the value of `EXPR` exists, then it is opened for reading, and the value of the function is of type `file`. If the file named by the value of `EXPR` does not exist, then the value of the function is `OM`.

   (b) `openw(EXPR)`. If the file named by the value of `EXPR` does not exist, then it is created by the operating system externally to ISETL. This file is opened for writing from the beginning, so that anything previously in the file is destroyed. The value of the function is of type `file`.

   (c) `opena(EXPR)`. The same as `openw(EXPR)` except that if the file exists its contents are not destroyed. Anything that is written is appended to the file.

2. In the following function, the value of `EXPR` must be of type `file`. The file specified by this value is closed. Output files must be closed to guarantee that all output has been stored by the operating system. All files are closed automatically when ISETL is exited. There is usually a system-imposed limit on the number of files that may be open at one time, however, so it is a good idea to close files when finished using them.

(a) `close(EXPR)`. The value of the function is `OM`.

3. In the following function the value of `EXPR` must be of type `file`.

   (a) `eof(EXPR)`. Test for having read past the end of an external file.

8.8 Miscellaneous.

1. `abs(EXPR)`. If the value of `EXPR` is `integer` or `floating_point`, then the value of the function is the standard absolute value.

2. `ord(EXPR)`. The inverse of `char`. `EXPR` must be a `string` of length 1.

3. `font(EXPR,EXPR)`. *Macintosh version only.* Both arguments should be `integer`. The first controls the font, the second the point size. The relation between numbers and fonts is dependent on the fonts loaded, so you need to experiment. If an argument is not an `integer`, that parameter is unchanged. The result is a pair, where the first element is the old font and the second element is the old size.

9 Precedence Rules

- Operators are listed from highest priority to lowest priority.
- Operators are left associative unless otherwise indicated.
- "nonassociative" means that you cannot use two operators on that line without parentheses.

`CALL`	anything that is a call to a function – `func`, `tuple`, `string`, `map`, etc.
`#`	
`?`	nonassociative
`%`	nonassociative
`**`	right associative
`* / mod div`	
`+ - with less`	
`INFIX .`	
`in notin subset`	
`< <= = /= > >=`	nonassociative
`not`	
`and`	
`or`	
`impl`	

10 Directives

There are a number of directives that can be given to ISETL to modify its behavior.

On the command line, `-s` indicates *silent mode*. In silent mode, the header and all prompts are supressed. This is useful when using ISETL as a filter. This feature is not available on the Macintosh version.)

The rest of the directives are `!` commands.

1. `!quit` – exit ISETL.
2. `!include <filename>` – Replace `<filename>` with a file/pathname according to the rules of your operating system. ISETL will insert your file.
3. `!clear` – throw away all input back to the last single prompt.
4. `!edit` – edit all the input back to the last single prompt.
5. `!echo [ on | off ]` – When on, all input is echoed. This is particularly useful when trying to find a syntax error in an `!include` file or input for a `read`. It is also useful for pedagogical purposes, as it can be used to interleave input and output.
6. `!memory` – shows how much memory has been allocated.
7. `!memory nnn` – increase the legal upper bound to `nnn`.
8. `!code [ on | off ]` – When on, you get a pseudo-assembly listing for the program. Default is off.
9. `!trace [ on | off ]` – When on, you get an execution trace, using the same notation as `!code`. When desperate, this can be used to watch the execution of your program. Really intended for debugging ISETL. Default is off.
10. `!ids` – Lists all identifiers which have been defined.

11 Editor

1. The user can edit whatever has been entered since the beginning of the current syntactic object, in response to a syntax error message, or if the user wants to change something previously typed. If you prefer to start again, "`!clear`" will clear the typing buffer and allow you to start the input afresh.
2. When the editor is invoked (by typing "`!edit`"), the user is prompted for the string that is to be modified. The user types the desired string, and the editor finds its first occurrence in the lines being edited.

3. The user is then prompted for the replacement of this string. When it is entered, the change is made.

4. The process repeats until the user enters a blank search line, at which time control is returned to ISETL.

12 Error Messages

The following list of error messages is given in two columns. The first is the message as it is printed to standard output, and the second gives a brief explanation of the meaning of the message.

*	used for self explanatory messages
internal	messages the user should never see
operator	messages that report bad args to the operator

RT Message	Explanation
'#'	operator
And	operator
Assignment	operator
Bad arg to mcPrint	internal
Bad args in low,next..high	*
Bad args in low..high	*
Bad modulus	operator mod
Bad selector to modify	lhs and selector incompatible
Bad slice assignment	slice on lhs out of range
Bounds must be integer	*
Cannot edit except at top level	Edit not permitted within an include
Cannot SMap	Cannot perform selection
Cannot MMap	Cannot perform selection
Cannot Slice	Cannot perform selection
Cnt2	bad args to `x..y`
Cnt3	bad args to `x,y..z`
Div	operator
Divide by zero	*
Exp	operator
From	operator
Fromb	operator
Frome	operator
IC2	bad args to iterator over `x..z`
IC3	bad args to iterator over `x,y..z`
Impl	operator
In	operator
Index < 1	*

Index out of bounds	*
Input must be an expression	*
Iter_Next	internal
Less	operator
Lower bound < 1	*
MMap	bad args to iterator for MMap
MMap/Str	bad args to iterator for MMap
MMap/Tuple	bad args to iterator for MMap
Minus	operator
Mk_Iter	bad LHS for iterator
Mod	operator
Negate	operator (unary `-`)
Negative slice	*
Non-bool to Br_False	May occur in `if`, `while`, `and`, `or`
Non-bool to Br_True	May occur in `if`, `while`, `and`, `or`
Non-integer index to string	*
Non-integer index to tuple	*
Not	operator
Only one level of selection allowed	Limitation on modification of funcs
Or	operator
Order Relation	incomparable args to `<`,`>`,`<=`,`>=`
Plus	operator
RHS of MMap must be a set	`f{x}`, `f` not a set of ordered pairs
Return at top level	*
Set Coersion	internal
Slash	operator
Slice out of bounds	*
Stack Overflow	*
Stack Underflow	* internal
String index too small	*
String requires integer index	*
Subset	operator
Times	operator
Top level return not allowed	*
Tuple index too small	*
Tuple requires integer index	*
With	operator