An Introduction to ISETL Version 3.0

Gary Marc Levin
Bitnet: gary@clutx
Internet: gary@clutx.clarkson.edu

September 18, 2021

Abstract

ISETL is an interactive implementation of SETL[1], a programming language built around mathematical notation and objects, primarily sets and functions. It contains the usual collection of statements common to procedural languages, but a richer set of expressions.

The objects of ISETL include: integers, floating point numbers, funcs (sub-programs), strings, sets, and tuples (finite sequences). The composite objects, sets and tuples, may contain any mixture of ISETL objects, nested to arbitrary depth.

This introduction is intended for people who have had no previous experience with ISETL, but who are reasonably comfortable with learning a new programming language. Few examples are given here, but many examples are distributed with the software.

This documentation is a useful supplement to *Learning Discrete Mathematics with* ISETL, a discrete math text written by Nancy Baxter, Ed Dubinsky, and Gary Levin, from Springer-Verlag. That text uses ISETL as a tool for teaching discrete mathematics.

[1]SETL was developed at the Courant Institute, by Schwartz. See Schwartz, J.T., *et al.* Programming with sets: An introduction to SETL. Springer-Verlag, 1986.

Contents

1 Running ISETL

ISETL is an interpreted, interactive version of the programming language SETL. It is invoked by typing a command line with the executable name, say `isetl`, along with optional file names that are discussed below.[2]

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. These actions are explained below. In the case of expressions, the result includes its value being printed. 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. In MS-DOS, ctrl-Z and ctrl-D will work.

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

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

   (a) If there is an initialization file[3] in the current directory, then the first thing ISETL will do is read this file.

   (b) Next, if the command line has any file names listed, ISETL will read each of these in turn.[4]

   Thus, if the command line reads,

```
isetl file.1 blue green
```

[2]The Macintosh version is clickable.

[3]Initialization files are called either `.isetlrc` or `isetl.ini`. The file is looked for in:

i. the current directory
ii. the directory containing `isetl.exe` (MSDOS and Mac)
iii. the home directory (Unix, VMS) or root (MSDOS)
iv. in the symbol `ISETLINI:` (VMS only)

Only one initialization file is read. The same pattern is searched for the `ile` initialization file.

[4]This feature is system dependent. To provide this feature in VMS, you must define `isetl :== $your$disk:[your.dir]isetl.exe` in your `login.com`. The leading `$` makes this a *foreign command*. The rest is the complete path to the executable version of ISETL.

ISETL will first read from ".isetlrc" if it exists, and then from "file.1", then "blue", and then "green". Finally, it is ready for input from the terminal.

(c) If there is a file available — say "file.2" — and ISETL is given (at any time), the following line of input,

```
!include file.2
```

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 at the keyboard,* and it can be followed on subsequent lines 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 end of the line is ignored by ISETL.

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 section 5 for more detail on scope.)

2 Characters, Keywords, and Identifiers

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.

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

The characters "`:`" and "`|`" may be used interchangably.

2.2 Keywords

The following is a list of `ISETL` keywords.

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

2.3 Identifiers

1. An identifier is a sequence of alphanumeric characters along with the underscore, caret, and prime — "`_^ '`". 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.

3 Simple Data Types

3.1 Integers

1. There is no limit to the size of integers.[5]

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

3. On input and output, long integers may be broken to accommodate limited line length. A backslash (“\”) at the end of a sequence of digits indicates that the integer is continued on the next line.

```
>       123456\
>>        789;
123456789;
```

3.2 Rationals

1. Rationals are only created when the directive `!rational on` has been used.

2. Rationals are created by dividing integers.

3. Arithmetic remains rational as long as possible.

3.3 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 zero or more digits, followed by a decimal point, followed by zero or more digits. Thus, `2.0`, `.5`, and `2.` are all legal.

   A floating_point constant 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 in scientific notation. Hence, for example, `0.2`, `2.0e-1`, `20.0e-2` are all equivalent. As with integers, it is unsigned.

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

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 Macintosh prints "+++++" and the Sun prints "Infinity".

3.4 Booleans

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

3.5 Strings

1. A string constant is any sequence of characters preceded and followed by a double quote, "`"`". A string may not be split across lines. Large strings may be constructed using the operation of concatenation. Strings may also be surrounded by single quotes, "`'`".

   The backslash convention may be used to enter special characters. When pretty-printing, these conventions are used for output. In the case of formated output, the special characters are printed.

\b	backspace
\f	formfeed (new page)
\n	newline (prints as CR-LF)
\q	double quote
\r	carriage return (CR)
\t	tab
\octal	character represented by *octal* Refer to an ASCII chart for meaning.
\other	*other* — may be any character not listed above.

   In particular, `"\\"` is a single backslash. You may type, `"\""` for double quote, but the pretty printer will print as `"\q"`. ASCII values are limited to `'\001'` to `'\377'`.

```
>       %+ [char(i): i in [1..127]];
"\001\002\003\004\005\006\007\b\t\n\013\f"
+"\r\016\017\020\021\022\023\024\025\026"
+"\027\030\031\032\033\034\035\036\037 !"
+"\q#$%&'()*+,-./0123456789:;<=>?@ABCDEF"
```

```
    +"GHIJKLMNOPQRSTUVWXYZ[\\]^_‘abcdefghijk"
    +"lmnopqrstuvwxyz{|}~\177";
```

3.6 Atoms

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

2. The keyword `newat` has as its value an atom never before seen in this session of `ISETL`.

3.7 Files

1. A file is an `ISETL` value that corresponds to an external file in the operating system environment.

2. They are created as a result of applying one of the pre-defined functions `openr`, `opena`, `openw`. (See section 7.8.)

3.8 Undefined

1. The data type undefined has a single value — `OM`. It may also be entered as `om`.

2. Any identifier that has not been assigned a value has the value `OM`.

4 Compound Data Types

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

3. 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.

4. Zero or more expressions, separated by commas and enclosed in braces ("{" and "}") evaluates to the set whose elements are the values of the enclosed expressions.

   Note that as a special case, the empty set is denoted by { }.

5. There are syntactic forms, explained in the grammar, for a finite set that is an arithmetic progression of integers, 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,y in {-1,-3..-100} | x /= y };
   ```

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

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

3. The order of the components of a tuple is significant. By treating the tuple as a function over the positive integers, you can extract individual components and contiguous subsequences (slices) of the tuple.

4. Zero or more expressions, separated by commas and enclosed in square brackets ("`[`" and "`]`") evaluates to the tuple whose defined components are the values of the enclosed expressions.

   Note that as a special case, the empty tuple is denoted by `[ ]`. This tuple is undefined everywhere.

5. The syntactic forms for tuples 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.

6. 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.

7. Tuples usually are indexed starting at 1, but they can have different starting indices. See page 31 and page 37 for definitions.

8. Tuples created by a `FORMER` have the default origin. See `origin` for how to redefine the default.

9. Tuples that result from operations on other tuples inherit their origin. Generally, the result inherits the origin of the leftmost tuple argument.

4.3 Maps

Maps 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`; if there is no such pair, it evaluates to `OM`; if there are many such pairs, an error is reported.

   (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 is no such pair, 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`.

5 Funcs

1. A `func` (`proc`) 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. (`procs` return an non-printing version of OM and should only be used in the place of statements.) 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. The `return` statement is only meaningful inside a func. Its effect is to terminate execution of the func and return a value to the caller. The form "`return` *expr*`;`" returns the value of *expr*; "`return;`" returns `OM`.

   `ISETL` inserts `return;` just before the `end` of every func.

3. 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 tuples and maps may be modified at a point by assignment, so can funcs. 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);
   "a";
   >       x(97) := "q";
   >       x(97);
   "q";
   >       x(97)(1) := "abc";
   ! Error: Only one level of selection allowed
   ```

   `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.

4. A number of functions have been pre-defined as funcs in `ISETL`. A list of their definitions is given in section 7. These are not keywords and may be changed by the user. They may not be modified at a point, however.

5. 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
   ```

   Alternately, one may write

```
: list-of-parameters -> result :
```

if the function simply consists of evaluating an expression.

(a) The declaration of `local` ids may be omitted if no local variables are needed. The declaration of `value` 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 may be evaluated, but the value has 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. It is an error to pass too many or too few arguments. It is possible to make some parameters *optional*.

```
f := func(a,b,c opt x,y,z); ... end;
```

`f` can be called with 3, 4, 5, or 6 arguments. If there are fewer than 6 arguments, the missing arguments are considered to be `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 :x -> f(g(x)) :
           end;
twice :=   :a -> 2*a: ;
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 is an example of functions modified at a point and functions that capture the current value of a global.

```
f := func(x);
         return x + 4;
     end func;
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`).

6 The `ISETL` Grammar — Annotated

6.1 Terminology

1. In what follows, the symbol `ID` refers to identifiers, and `INTEGER`, `FLOATING_POINT`, `BOOLEAN`, and `STRING` refer to constants of type integer, floating_point, Boolean, and string, which have been explained above. Any other symbol in capital letters is explained in the grammar.

2. Definitions appear as:

   `STMT` → `LHS := EXPR ;`

   `STMT` → **`if`** `EXPR` **`then`** `STMTS ELSE-IFS ELSE-PART` **`end`**

   indicating that `STMT` can be either an assignment statement or a conditional statement. The definitions for `ELSE-IFS` and `ELSE-PART` are in the section for statements, and `EXPR` in the section for expressions.

3. Rules are sometimes given informally in English. The rule is then quoted.

4. Spaces are not allowed within any of the character pairs listed in section 2, nor within an `ID`, `INTEGER` constant, `FLOATING_POINT` constant, or keyword. Spaces are required between keywords, `ID`s, `INTEGER` constants, and `FLOATING_POINT` constants.

5. `ISETL` treats ends of line 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 `!` directives, which are ended with a carriage return, and the fact that a quoted string cannot be typed on more than one line.

The annotated grammar below is divided into sections relating to the major parts of the language.

6.2 Input at the Prompt

`INPUT` → `PROGRAM`

`INPUT` → `STMT`

`INPUT` → `EXPR ;`
The `EXPR` is evaluated and the value is printed.

6.3 Program

Programs are usually read from a file, only because they tend to be long.

`PROGRAM` → `program ID ; LOCALS VALUES STMTS end ;`
Of course, it can appear on several lines. One may optionally close with `end program`. `LOCALS` and `VALUES` are defined in section 6.10.

6.4 Statements

`STMT` → `LHS := EXPR ;`
First, the left hand side (`LHS`) is evaluated to determine the target(s) for the assignment, then the right hand side is evaluated. Finally, the assignment is made. If there are some targets for which there are no values to be assigned, 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.

`STMT` → `EXPR ;`
The expression is evaluated and the value ignored. This is usually used to invoke procedures.

`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. One may optionally close with `end if`. See the end of this section for the definitions of `ELSE-IFS` and `ELSE-PART`.

`STMT` → `for ITERATOR do STMTS end ;`
The `STMTS` are executed for each instance generated by the iterator. One may optionally close with `end for`.

`STMT` → `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. One may optionally close with `end while`.

`STMT` → `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`. As usual, terminate the expressions with a semicolon. *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.

`STMT` → `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 section 7.8).

`STMT` → `readf PAIR-LIST ;`

`STMT` → `readf PAIR-LIST from EXPR ;`
The relation between these two forms is the same as the relation between the two forms of `read`, with the second one coming from a file. The elements in the `PAIR-LIST` define the formating used. See `PAIR-LIST` at the end of this section.

`STMT` → `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.

`STMT` → `print EXPR-LIST to EXPR ;`

As in `read...from...`, `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 writing by one of the pre-defined functions `openw` or `opena` (see section 7.8).

`STMT` → `printf PAIR-LIST ;`

`STMT` → `printf PAIR-LIST to EXPR ;`

The relation between these two forms is the same as the relation between the two forms of `print`, with the second one going to a file. The elements in the `PAIR-LIST` define the formating used. See `PAIR-LIST` at the end of this section. See `write` and `writeln` below.

`STMT` → `return ;`

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

`STMT` → `return EXPR ;`

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

`STMT` → `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.

`STMT` → `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).

`STMT` → `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 $(i = 1, 2, \ldots)$.

`STMT` → `write PAIR-LIST ;`

`STMT` → `write PAIR-LIST to EXPR ;`

`STMT` → `writeln PAIR-LIST ;`

```
>       readf x;
   1.34
>       x;
1.34000e+00;

>       readf y;
123,456
>       y;
"123,456";
```

Figure 1: `readf` example

`STMT` → `writeln PAIR-LIST to EXPR ;`
`write` is equivalent to `printf`, provided for the convenience of the Pascal user. `writeln` is equivalent to `write`, with `'\n'` as the last item of the list. This is also provided for user convenience.

`STMTS` → "One or more instances of `STMT`. The final semicolon is optional."

`ELSE-IFS` → "Zero or more instances of `ELSE-IF`."

`ELSE-IF` → `elseif EXPR then STMTS`

`ELSE-PART` → `else STMTS`
"May be omitted."

`PAIR-LIST` → "One or more instances of `PAIR`, separated by commas."

`PAIR` → `EXPR : EXPR`

`PAIR` → `EXPR`
When a `PAIR` appears in a `readf`, the first `EXPR` must be a `LHS`. The meaning of the `PAIR` and the default value when the second `EXPR` is omitted depends on whether the `PAIR` occurs in `readf` or `printf`. The second `EXPR` (or its default value) defines the format.

- Input: Input formats are integers.

  The integer gives the maximum number of characters to be read. If the first sequence of non-white space characters can be interpreted

```
>       printf 1/3: 15.10,  1/3:15.1,  1/3:15.01, "\n";
0.3333333135   0.3333333135            0.3

printf 1/3: -17.10, 1/3:-17.1, 1/3:-17.01, "\n";
3.3333331347e-01 3.3333331347e-01          3.3e-01
```

Figure 2: `printf` example

```
>       printf 3*[""]+[1..30] : 7*[3] with "\n";
           1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30
>       x := [ [i,j,i+j] : i,j in [1..3] ];
>       printf x: 5*[ [0,"+",0, "=", 0], "\t" ]
>>                  with "\n", "\n";
1+1=2   1+2=3   1+3=4   2+1=3   2+2=4
2+3=5   3+1=4   3+2=5   3+3=6
```

Figure 3: `printf` with structure example

as a number, that is the value read. Otherwise, the first non-white sequence is returned as a string.

If the integer is negative (say $-i$), exactly i characters will be read and returned as a string. Therefore `c:-1` will read one character into `c`.

If no integer is given, there is no maximum to the number of characters that will be read.

See figure 1.

- Output: Output formats are: integers, floating_point numbers, strings, or tuples of output formats.

  Integers (and the integer part of floating_point numbers) represent the minimal number of columns to be used. The fractional part of a floating_point number is used to specify precision, in terms of hundredths. The precision controls the number of places used in floating_point numbers, and where breaks occur in very long integers.

  Negative values cause floating_point numbers to be printed in scientific notation.

  Notice that there is a limit to the number of useful digits. Also notice that 15.1 is the same as 15.10; hence, both would use 15 columns and 10 decimal places. See figure 2.

  Strings should not be used as formats outside of tuples.

  Compound objects (tuples and sets) iterate over the format. If the format is a number, it is used as the format for each element. If the format is a tuple, the elements of the tuple are cycled among, with strings printed literally and other items used as formats. See figure 3.

  Default values are:

Type	Columns	Precision
Float	20	5
Integer	10	50 (for breaking large ints)
String	0	
Anything else	10	

6.5 Iterators

These constructs are used to iterate through a collection of values, assigning these values one at a time to a variable. Iterators are used in the `for` statement, quantifiers, and set formers.

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` statements, quantifiers, formers, etc.)

`ITERATOR` → `ITER-LIST`

`ITERATOR` → `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` → `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`.

`SIMPLE-ITERATOR` → `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.

`SIMPLE-ITERATOR` → `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.

`BOUND` → `ID`
Corresponding value is assigned to `ID`.

`BOUND` → `[ BOUND-LIST ]`
Corresponding value must be a tuple, and elements of the tuple are assigned to corresponding elements in the `BOUND-LIST`.

6.6 Formers

Generates the elements of a set or tuple.

`FORMER` → "Empty"
  Generates the empty set or tuple.

`FORMER` → `EXPR-LIST`
  Values are explicitly listed.

`FORMER` → `EXPR .. EXPR`
  Both occurrences of `EXPR` must evaluate to integers or characters (strings of length 1). Generates all integers (characters) 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. The characters are generated in ASCII order.

`FORMER` → `EXPR , EXPR .. EXPR`
  All three occurrences of `EXPR` must evaluate to numbers. Generates all numbers 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 values that are not greater than the third `EXPR`. If the difference is negative, it generates those values that are not less than the third `EXPR`. If the difference is zero, no values are generated.

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

6.7 Selectors

Selectors fall into three categories: function application, mmap images, and slices. 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 `SELECTOR`s can be used in all four cases.

`SELECTOR` → `( EXPR-LIST )`
  Must be used with an smap, tuple, string, or func.

  If used with a tuple or string, then `EXPR-LIST` can only have one element, which must evaluate to a positive integer.

  If used with a func, arguments are passed to corresponding parameters. There must be as many arguments as required parameters and no more than the optional parameters permit.

If used with an smap and `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 tuple whose components are the individual variables.

`SELECTOR` → { `EXPR-LIST` }
Must be used with an mmap, tuple, or string. Tuples and strings will either select a singleton set or the empty set. The case in which the list has more than one element is handled as above.

`SELECTOR` → `( 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 integers 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 empty tuple.
$a > b + 1$	Run-time error.

`SELECTOR` → `( .. EXPR )`
Means the same as `(low .. EXPR)`, where `low` is 1 for strings and `lo(T)` for tuple `T`.

`SELECTOR` → `( EXPR .. )`
Means the same as `( EXPR .. high )`, where `high` is `#s` for string `s` and `hi(T)` for tuple `T`.

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

6.8 Left Hand Sides

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

`LHS` → `ID`

`LHS` → `@ EXPR`

`LHS` → `@ ( EXPR , EXPR , ... , EXPR)`

These are variables and may be used wherever a variable is needed. *NB:* The ids in declarations and binding positions (iterators) are not variables and cannot be `@`-expressions.

The expressions may be strings or integers. The integers are converted to strings and the strings are then concatenated to produce the variable name.

`LHS` → `LHS SELECTOR`

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

`LHS` → `[ 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.

6.9 Expressions

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

`EXPR` → `ID`

`EXPR` → `INTEGER`

`EXPR` → `FLOATING-POINT`

`EXPR` → `STRING`

`EXPR` → `true`

`EXPR` → `false`

`EXPR` → `OM`

`EXPR` → `newat`
The value is a new atom, different from any other atom that has appeared before.

`EXPR` → `FUNC-CONST`
A user-defined func. See section 6.10.

`EXPR` → `if EXPR then EXPR ELSE-IFS ELSE-PART end ;`
See definition of `if` under `STMT`, page16. If `ELSE-PART` is omitted, it is replaced by "`else OM`". Each part contains an expression rather than statements.

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

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

`EXPR` → { `FORMER` }
Evaluates to the set of those values generated by `FORMER`.

`EXPR` → `# 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.

`EXPR` → `not EXPR`
Logical negation. `EXPR` must evaluate to Boolean.

`EXPR` → `+ EXPR`
Identity function. `EXPR` must evaluate to a number.

`EXPR` → `- EXPR`
Negative of `EXPR`. `EXPR` must evaluate to a number.

`EXPR` → `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 section 6.7.

`EXPR` → `EXPR . ID EXPR`
This is equivalent to `ID(EXPR,EXPR)`. It lets you use a binary function as an infix operator. The space after the "`.`" is optional.

`EXPR` → `EXPR . (EXPR) EXPR`
: This is equivalent to `(EXPR)(EXPR,EXPR)`. It lets you use a binary function as an infix operator. The space after the "`.`" is optional.

In general, arithmetic operators and comparisons may mix integers and floating_point. The result of an arithmetic operation 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.
Possible operators are:

```
     + - * / div mod **
          with less
        = /= < > <= >=
union inter in notin subset
       and or impl iff
```

See section 8 for precedence rules.
Any cases not covered in the explanation for an operator will result in an error. For an explanation of errors, see section 11.

`EXPR` → `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 union EXPR`
: An alternate form of `+`. It is intended that it be used with sets, but it is in all ways equivalent to `+`.

`EXPR` → `EXPR - EXPR`
: If both instances of `EXPR` evaluate to numbers, this is subtraction. If both instances of `EXPR` evaluate to sets, then this is set difference.

`EXPR` → `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 inter EXPR`
: An alternate form of `*`. It is intended that it be used with sets, but it is in all ways equivalent to `*`.

`EXPR` $\rightarrow$ `EXPR / EXPR`
Both instances of `EXPR` must evaluate to numbers. The value is the result of division and is of type floating_point.

`EXPR` $\rightarrow$ `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,

$$\begin{array}{ll} (a \text{ div } b) * b + (a \text{ mod } b) = a & \text{for } b > 0 \\ a \text{ div } (-b) = -(a \text{ div } b) & \text{for } b < 0. \end{array}$$

`EXPR` $\rightarrow$ `EXPR mod EXPR`
Both instances of `EXPR` must evaluate to integer and the second must be non-zero. The result is the remainder, and the following condition is always satisfied,

$$0 \leq a \text{ mod } b < |b|.$$

`EXPR` $\rightarrow$ `EXPR ** EXPR`
The values of the two expressions must be numbers. The operation is exponentiation.

`EXPR` $\rightarrow$ `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` $\rightarrow$ `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; the value of the first `EXPR`, if the second was not present.

`EXPR` $\rightarrow$ `EXPR = EXPR`
The test for equality of any two `ISETL` values.

`EXPR` $\rightarrow$ `EXPR /= EXPR`
Negation of `EXPR=EXPR`.

`EXPR` $\rightarrow$ `EXPR < EXPR`

`EXPR` $\rightarrow$ `EXPR > EXPR`

`EXPR` $\rightarrow$ `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 numbers, this is the test for the standard arithmetic ordering; for strings, it is the test for lexicographic ordering.

`EXPR` → `EXPR in EXPR`

The second `EXPR` must be a set, tuple, or string. For sets and tuples, this is the test for membership of the first in the second. For strings, it is the test for substring.

`EXPR` → `EXPR notin EXPR`

Negation of `EXPR in EXPR`.

`EXPR` → `EXPR subset EXPR`

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

`EXPR` → `EXPR and EXPR`

Logical conjunction. Both instances of `EXPR` should evaluate to a Boolean. If the left operand is false, the right operand is not evaluated. Actually returns the second argument, if the first is `true`. While the user may depend on the left-to-right evaluation order, it is recommended that they not depend on the behavior when the second argument is not Boolean.

`EXPR` → `EXPR or EXPR`

Logical disjunction. Both instances of `EXPR` should evaluate to a Boolean. If the left operand is true, the right operand is not evaluated. Actually returns the second argument, if the first is `false`. While the user may depend on the left-to-right evaluation order, it is recommended that they not depend on the behavior when the second argument is not Boolean.

`EXPR` → `EXPR impl EXPR`

Logical implication. Both instances of `EXPR` must evaluate to a Boolean.

`EXPR` → `EXPR iff EXPR`

Logical equivalence. Both instances of `EXPR` should evaluate to a Boolean. It actually checks for equality, like `=`, but it has a different precedence. It is recommended that the user not depend on `iff` to work with arguments other than Booleans.

`EXPR` $\rightarrow$ `% 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, `%`$\oplus$ `EXPR` equals

`x1` $\oplus$ `x2` $\oplus \cdots \oplus$ `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` $\rightarrow$ `EXPR % BINOP EXPR`

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

`a` $\oplus$ `x1` $\oplus$ `x2` $\oplus \cdots \oplus$ `xN`

associating to the left.

`EXPR` $\rightarrow$ `EXPR ? EXPR`

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

`EXPR` $\rightarrow$ `choose ITER-LIST | EXPR`

Returns the first set of iterator values that satisfies EXPR. The value returned depends on the type of iterators.

Iterator	Returns
========	=======
`x in S`	`x`
`x in S, y in T`	`[x,y]`
`x,y in S`	`[x,y]`
`y=f(x)`	`[x,y]`
`y=f(x), a=g(b)`	`[[x,y], [a,b]]`
`y=f{x}`	`[x,y], where y is a set`

`EXPR` $\rightarrow$ `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.

`EXPR` $\rightarrow$ `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` → `EXPR where DEFNS end`
The value is the value of the `EXPR` preceding `where`, evaluated in the current environment with the `ID`s in the `DEFNS` added to the environment and initialized to the corresponding `EXPR`s. The scope of the `ID`s is limited to the `where` expression. The `DEFNS` can modify `ID`s defined in earlier `DEFNS` in the same `where` expression.

`EXPR` → `EXPR @ EXPR`
The first expression must be an integer `i` and the second a tuple `T`. The result is a tuple consisting of the same sequence as `T`, but with the first index being `i`.

`BINOP` → "Any binary operator or an `ID` or expression in parentheses whose value is a function of two parameters. The `ID` and parenthesized expression may be preceded by a period."
The acceptable binary operators are: `+`, `-`, `*`, `**`, `union`, `inter`, `/`, `div`, `mod`, `with`, `less`, `and`, `or`, `impl`.

`DEFNS` → "Zero or more instances of `DEFN`. The final semicolon is optional."

`DEFN` → `BOUND := EXPR ;`

`DEFN` → `ID SELECTOR := EXPR ;`

`EXPR-LIST` → "One or more instances of `EXPR` separated by commas."

6.10 Function Constants

`FUNC-CONST` → `FUNC-HEAD LOCALS VALUES STMTS end`
This is the syntax for user-defined funcs. One may optionally close with `end func`. `VALUES` and `LOCALS` may be repeated or omitted and appear in any order.
See `return` on page 18.

`FUNC-CONST` → `:  ID-LIST OPT-PART -> EXPR :`
An abbreviation for `func( ID-LIST OPT-PART ); return EXPR; end`

`FUNC-HEAD` → `func ( ID-LIST OPT-PART ) ;`
In this case, there are parameters. The parameters in the `OPT-PART` receive the value `om` if there are no corresponding arguments.

`FUNC-HEAD` → `func ( OPT-PART ) ;`
In this case, there are no required parameters.

`FUNC-HEAD` → `proc ( ID-LIST OPT-PART ) ;`

`FUNC-HEAD` → `func ( OPT-PART ) ;`
Just like `func`, but no value may be returned in the return statement.
Values of type `proc` should only be used as statements.

`OPT-PART` → `opt ID-LIST`
"May be omitted."

`LOCALS` → `local ID-LIST ;`

`VALUES` → `value ID-LIST ;`

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

7 Pre-defined Functions

All pre-defined functions are initially locked, preventing accidental modification. You can unlock the id with the `!unlock` directive. If no return value is specified, the function is a `proc` and should be used as a statement.

7.1 Functions on Integers

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

1. `even(EXPR)` — Is `EXPR` even?
2. `odd(EXPR)` — Is `EXPR` odd?
3. `float(EXPR)` — The value of `EXPR` converted to floating_point.
4. `char(EXPR)` — The one-character string whose (machine dependent) index is the value of `EXPR`.

7.2 Functions on Rationals

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

1. `den(EXPR)` — returns the denominator of `EXPR`.
2. `num(EXPR)` — returns the numerator of `EXPR`.

7.3 Functions on Floating Point Numbers

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

1. `ceil(EXPR)` — The smallest integer not smaller than the value of `EXPR`.
2. `floor(EXPR)` — The largest integer not larger than the value of `EXPR`.
3. `fix(EXPR)` — 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.

7.4 Functions on Sets

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

1. `pow(EXPR)` — The set of all subsets of the value of `EXPR`.
2. `npow(EXPR,EXPR)` — One `EXPR` must be a set and the other a non-negative integer. The set of all subsets of the set whose cardinality is equal to the integer.

7.5 Functions on Maps

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

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

7.6 Standard Mathematical Functions

1. Each of the following takes a single floating_point argument. The result is a floating_point approximation to the value of the corresponding mathematical function. `exp`, `ln`, `log`, `sqrt`, `sin`, `cos`, `tan`, `sec`, `csc`, `cot`, `asec`, `acsc`, `atan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.
2. In each of the following, `EXPR` must evaluate to integer or floating_point. The result 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)` — This resets the random number generator. EXPR should be an integer. This may be used to select a new sequence of random numbers.

3. In each of the following, both occurrences of `EXPR` must evaluate to a number or string. The result is always one of the two `EXPR`, according to the usual mathematical definition.

   (a) `max(EXPR,EXPR)`

   (b) `min(EXPR,EXPR)`

7.7 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_defined(EXPR)` — Negation of `is_om`.
4. `is_file(EXPR)`
5. `is_floating(EXPR)`
6. `is_func(EXPR)`
7. `is_integer(EXPR)`
8. `is_map(EXPR)`
9. `is_number(EXPR)` — true for integer and floating_point.
10. `is_om(EXPR)`
11. `is_rational(EXPR)`
12. `is_set(EXPR)`
13. `is_string(EXPR)`
14. `is_tuple(EXPR)`

7.8 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 used in `read... from...`, `readf... from...`, `print... to...`, `printf... to...`, and the function `eof` 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`.

   A special case is the file named `"CONSOLE"`. Opening `"CONSOLE"` for reading provides a way to read from the console, even if you are currently reading from an include file. If you have directed `stdin` from a file, it may read from that file or it may read from the console; this is machine dependent.

   (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 added to the end of 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)` — Closes the file.

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.

7.9 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. `arb(EXPR)` — An element of `EXPR` selected arbitrarily. If the value of `EXPR` is empty, then the value of the function is `OM`. `EXPR` may be a set, tuple, or string.
4. `random(EXPR)` — An element of `EXPR` selected with uniform probability. If the value of `EXPR` is empty, then the value of the function is `OM`. `EXPR` may be a set, tuple, or string.
5. `max_line(EXPR)` — `EXPR` must be an integer. The maximum number of columns used when pretty-printing is set to the value of `EXPR`.
6. `system(EXPR)` — `EXPR` must be a string. The string is passed to the operating system as a command line. Available under Unix, VMS, and MSDOS.
7. `precision(EXPR)` — `EXPR` must be an integer. This sets the number of decimal places shown by `print`. If `EXPR` is negative, it indicates that `print` should use scientific notation.
8. `video(EXPR)` — MSDOS only. `EXPR` must be a boolean. This controls how the screen is managed under MSDOS. Generally, `true` is faster output, and `false` is less likely to run into trouble with compatibility questions. Also controled by `-v` on the command line. See section 9.1.

7.10 Tuple

1. `lo(EXPR)` — `EXPR` must be a tuple. Returns the low bound of the tuple.
2. `hi(EXPR)` — `EXPR` must be a tuple. Returns the high bound of the tuple.
3. `origin(EXPR)` — `EXPR` must be an integer. Sets the default lower bound for tuples.

7.11 Graphics

The following routines work on the PC and Mac versions only.

1. `graphics(bool)` — Call with true before any of the following commands. Call with false to return to editor mode (or close window). On PC, `^ G` will switch between graphics and editor. You must have the appropriate `*.BGI` file for your PC graphics adaptor in the same directory as `isetl.exe`.

2. `scale(minX,maxX,minY,maxY)` — Sets up graphing region. Will plot points within the rectangle described by args, scaling your values to fit on the screen. Call before any of the following commands.

3. `move(x,y)` — Move current point (CP) to `(x,y)`. Nothing drawn.

4. `draw(x,y)` — Draw line from CP to `(x,y)`. Change CP to `(x,y)`.

5. `textout(x,y,text)` — Write text starting at `(x,y)` and writing left to right. If `(x,y)` are omitted, writes at CP.

6. `get_coord()` — Returns `[x,y,c]` when a key is pressed. `(x,y)` is the point on the screen and `c` is the key pressed. `c=''` if mouse was clicked.

7. `thickline(bool)` — Should lines be thick? Returns old value.

8. `title_window(text)` — Writes near top of graph, on PC. Writes in title bar on Mac.

9. `clear_screen()` — Clears screen and undefines objects.

10. `new_object()` — Returns an integer i identifying the new object. Anything plotted up to the next `new_object` is part of this object and may be deleted with `del_object(i)` and added back again with `add_object(i)`.

11. `add_object(i)` — See `new_object`.

12. `del_object(i)` — See `new_object`.

13. `resolution()` — Returns `[minX,minY]`, distance between pixels.

14. `XtoYratio()` — Returns ratio of x pixel size to y pixel size.

15. `erasable` is used for small objects that come and go quickly. Drawn in xor mode, so complex figures or text may look funny.

    `erasable(true)` — Plotting is made part of the erasable object.

    `erasable(false)` — Plotting is no longer part of erasable object.

    `erasable()` — Erase erasable object.

16. `char_mult(m)` — Magnify text by `m`. Returns old value.

17. `point(x,y,size)` — Draw square `2*size+1` pixels on a side, centered at `(x,y)`. CP becomes `(x,y)`. If `x,y` are missing, draws at current CP. If size is negative, fills square.

18. `rectangle(xlow,xhigh,ylow,yhigh,hollow)` — Draw rectangle. If `hollow` is true or is omitted, rectangle is hollow. If `hollow` is false, rectangle is filled in.

19. `save_graph(filename)` — Write current graph to `filename`. On PC, empty string pops up a window to let you enter the name of the file. On Mac, use SAVE from menu.

20. `load_graph(filename)` — Read `filename` and add to current object. Uses the current scale, which may be incorrect for this graph. You can call scale after a plot. On PC, empty string pops up a window to let you enter the name of the file. On Mac, this is an error.

8 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.
`# - + @`	unary operators
`@`	nonassociative
`?`	nonassociative
`%`	nonassociative
`**`	right associative
`* / mod div inter`	
`+ - with less union`	
`.ID`	infix use of binary function
`in notin subset`	
`< <= = /= > >=`	nonassociative
`not`	unary
`and`	
`or`	
`impl`	
`iff`	
`exists forall`	
`where`	

9 Directives

9.1 Brief Descriptions

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

On the command line, the following switches control aspects of `ISETL`.

`-d` indicates *direct input*. This suppresses the interactive line editor or the screen editor in MSDOS.

`-e` implies `-d` and provides the `!edit` directive described below.

`-s` indicates *silent mode*. In silent mode, the header and all prompts are suppressed.

`-v` (MSDOS only) controls the initial value of `video`. `-v` sets it to safe (on all PC compatibles), but slow (on most video boards).

The rest of the directives are `!` commands. `[ a | b ]` indicates a choice between `a` and `b`.

9.1.1 Commands

- `!quit` — Exit `ISETL`.
- `!include <filename>` — Replace `<filename>` with a file/pathname according to the rules of your operating system. `ISETL` will insert your file.
- `!clear` — Throw away all input back to the last single prompt.
- `!edit` — Edit all the input back to the last single prompt. Unavailable on systems with the interactive line editor.
- `!memory [nnn]` — Change the legal upper bound to `nnn`. May not be lower than the currently allocated memory. Without `nnn`, shows how much memory has been allocated.
- `!allocate nnn` — Increase the currently allocated memory to `nnn`. Will not exceed the upper bound set by `!memory`, nor the actual limits of the machine.
- `!record [ file-name ]` — Begins recording input to "`file-name`". This lets you experiment and keep a record of the work performed.

- `!system command-line` — Sends the `command-line` to the system for execution. Not available on the Macintosh.
- `!lock list-of-ids` — Prevent future assignments to the ids in the list. Predefined functions are locked by default.
- `!unlock list-of-ids` — Permit future assignments to the ids in the list.
- `!ids` — Lists all non-locked identifiers that have been defined.
- `!locked` — Lists all locked identifiers that have been defined.
- `!oms` — Lists all identifiers that have been used, but not defined.
- `!alias id command-line` — Makes `!id` equivalent to `!command-line`.
- `!version` — Prints version information for `ISETL`.
- `!credits` — Print some copyright information.

9.1.2 Toggles

Toggles take arguments `on` or `off`. Without arguments, they echo the toggle's current state.

- `!verbose` — Controls the amount of information provided by runtime error messages. See section 11. Default is off.
- `!echo` — 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.
- `!code` — When on, you get a pseudo-assembly listing for the program. Default is off.
- `!trace` — When on, you get an execution trace, using the same notation as `!code`. When desperate, this can be used to trace the execution of your program. Really intended for debugging `ISETL`. Default is off.
- `!source` — Saves source for debugging. See `!pp`, `!stack`, and `!slow`.
- `!stack` — Show calls when errors occur.

- `!setrandom` — When off, sets are printed in a canonical order. Default is on.
- `!rational` — When on, `int / int` produces a rational result. Default is off.

9.1.3 Debugging

1. `!watch list-of-ids` — Traces assignment and evaluation of ids.
2. `!unwatch list-of-ids` — Turns off tracing for ids.
3. `!pp id [ file-name ]` — Prints the source for function `id`. When present, output goes to `file-name`; otherwise, output goes to last file. `!pp` returns the file to `stdout` (usually the screen).
4. `!slow` — Execution steps by source lines. See section 9.1.3.
5. `!fast` — Return to normal execution speed.

When the system is stopped for debugging, in the `!slow` mode, you get the `?>` prompt. Responses at this point are:

`f` — go to fast mode.

`l` — leap mode (calls are executed as one step).

`c` — crawl mode (trace execution within calls).

`e` — evaluate. Enter an expression at the `!` prompt.

`RET` — Execute the next step.

9.2 `!clear` and `!edit`

1. The user can edit[6] 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.

[6]Turn this on with the `-e` switch.

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`.

9.3 `!allocate` and `!memory`

The `!memory` directive adjusts the upper limit on permitted memory allocation. This is mainly to protect mainframe systems, so that one user doesn't use all the available space.

The `!allocate` directive increases the amount of memory *currently* available for `ISETL` objects. This space is automatically increased up to the limit set by `!memory`, but by allocating it early, some large programs may run more quickly.

If you want to grab as much memory as possible, particularly on single user systems, this is what we would recommend. First, determine the amount of memory available, by attempting to allocate everything. Then subtract from that 10K for `ISETL`'s scratch area plus any other space you may wish to save for use by the `!system` directive. You can then set the memory limit and pre-allocate in your `isetl.ini` (or `.isetlrc`) files.

See figure 4. Having tried to allocate 800K, there was only room for 500K. Deciding to leave 200K for other work, a limit of 300K was placed on `ISETL`, and 150K was pre-allocated. The lines below "..." are in another session, because one cannot decrease the GC (garbage collected) memory.

9.4 `!watch` and `!unwatch`

The two commands `!watch` and `!unwatch` control which identifiers are traced during execution. Tracing consists of reporting assignments and function evaluation.

An identifier is watched by the directive:

```
!watch id id1 id2 id3
```

where "`id`" is the name of the identifier to be watched. More than one identifier may be listed, separated by blanks.

While being watched, any assignment to a variable named with that identifier is echoed on the standard output. This includes assignments to slices and maps. If the identifier is used as a function (smap, mmap, tuple, func), a line is printed indicating that the expression is being evaluated and a second line is printed reporting the value returned.

It is significant that identifiers are watched, rather than variables. If `i` is being watched, then *all* variables named `i` are watched.

You can stop watching an identifier with the directive:

```
!unwatch id
```

See figure 5 for an example of the output.

9.5 `!record`

The `!record` directive channels all input from standard input into a file. This allows you to capture your work and later edit it for including.

A directive of the form: `!record test` changes to recording on file `test`. If you had been recording elsewhere, the other file is closed. `!record` with no file name turns off recording altogether. The recording is appended to an existing file.

By combining this with the `!echo` directive, one can create terminal sessions.

9.6 `!system`

This allows you to execute one command in the operating system without leaving `ISETL`. This feature is not available on the Macintosh version. See section 9.3 for hints on making sure that there is enough room to invoke the command from the system.

You could list your directory on MS-DOS using the command:

```
!system dir
```

Assuming that you had enough memory, you could escape to an editor, edit a file, exit the editor, and then include the file.

If you type `!system` by itself, you will enter a new copy of your operating system. You can execute anything that fits in the remaining memory.

10 Editors

The original view of ISETL was a program that read lines of text, recognizing programs and expressions, and then evaluating them. The introduction of editors adds a second level to this. In each of the editors, there is some way to *send text to* ISETL. This phrase refers to taking the text and treating it as if those lines had been typed directly in.

10.1 MSDOS Screen Editor

In the MSDOS editor, you send lines to ISETL by typing RETURN. If you are on the last line of the window, that line is sent. You may send other lines by selecting a *region*. The first line of a region is called the TAG. The last line is the line containing the cursor. Regions are written in reverse video.

If you want to send lines from the edit window, edit them *first*, then TAG the first line you wish sent by typing ^ T, move to the last line that you wish sent, and type RETURN. Prompts at the beginning of a line are ignored.[7]

To make it easy to check a region, you can find TAG by typing ^ X (control-X). This will exchange the cursor and TAG. Type ^ X again to return.

To make it easy to find the last region sent, type ^ B. This finds the BOOK_MARK, which is left behind at the old TAG after a region is sent to ISETL.

There is a menu (ESC). You can execute commands from the menu by typing ESC followed by the capital letter in the command or by moving to the command with the arrows and then typing return.

A hint. You can read a file that contains prompts and then execute it. Note that you cannot `!include` a file with prompts, because they are not syntactically correct, but prompts are stripped from the beginning of the line when the editor sends lines to ISETL.

10.2 Mac Screen Editor

The Macintosh version has an editor that needs no introduction. All operations are reachable from the menus, and follow the standard Macintosh usages.

The only unusual feature is that highlighted regions can be run — sent to the execution window and sent to ISETL as input. You can run by:

- Typing RETURN,

[7]N.B.: If you try to type a line starting with '>', '?', or '!' you must leave a blank in front of them to prevent their removal. Blanks are automatically inserted after the prompts.

Table 1: Important keys for MSDOS Editor(`^X` = control-X)

`ESC`	Get menu
`Arrow Keys`	Motion
`INS`	Break line
`DEL`	Delete under
`Backspace`	Delete left
`Home`	Left of line
`End`	Right of line
`PgUp (PgDn)`	Up (down) 8 lines
`^PgUp (^PgDn)`	Top (bottom) of buffer
`Return (LF)`	If below tag, execute; o.w. insert return
`^A`	Mark previous region
`^B`	Go back to bookmark (previous tag)
`^E`	Erase current line
`^G`	Show graphics screen
`^L`	Refresh screen
`^T`	Tag top of region
`^X`	Exchange tag and cursor
`^Z`	Escape to DOS. Use “exit” to return

Help is available through menu or `F1`.

- Selecting `Run` from the menu, or
- Typing `clover-R`.

In addition, typing `RETURN` on the last line of the Execution window causes that line to be sent to `ISETL`.

10.3 Interactive Line Editor (ILE)

10.3.1 Brief description

The `left` and `right arrows` will move you within a line, permitting insertions of characters. `delete` removes the character at the cursor, `backspace` deletes the character left of the cursor. The interesting feature is that the `up arrow` moves you back thru the last hundred lines entered, with `down arrow` moving you forward. You can’t go past the last entered line.

Table 2: Menu for MSDOS Editor

`Copy region`	Copies region to the end of the buffer.
`Print region`	Send region to printer.
`Save region`	Appends region to a file. Omits prompts.
`Read file`	Placed at end of buffer.
`save Buffer`	Saves buffer, with prompts.
`Quit`	Like !quit.
`buffer Info`	Information on buffer size and cursor position.
`Help`	Describes key maps.
`clear region`	Erase region.
`clear buffer`	Erase all lines in buffer.

You need to use `!clear` if you want to throw away your current input (since the last `>`) so that you can edit it.
Example:

```
>   a := b +
>>       c +
>>  !clear
>           =up=>      c + =up=>  a := b +
>>          =up=> a := b + =up=>       c +  =edit=> c;
```

The `!clear` had `ISETL` throw away the earlier input, but left it for subsequent editting. `=up=>` means typing the up arrow, followed by the new value displayed on that line. `=edit=>` means editing the line to produce the desired result.

Below is a complete description of the new editor.

10.3.2 Default key bindings

The interactive line editor is an input line editor that provides both line editing and a history mechanism to edit and re-enter previous lines.

`ISETL` looks in the `ile` initialization file. See page 4 for more information.

Not everyone wants to have to figure out yet another initialization file format so we provide a complete set of default bindings for all its operations.

The following table shows the default bindings of keys and key sequences provided by `ile`. These are based on the emacs key bindings for similar operations.

Key	**Effect**	**VMS differences**
`del`	delete char under	
`^A`	start of line	undefined
`^B`	backward char	
`^E`	end of line	
`^F`	forward char	
`^K`	erase to end of line	
`^L`	retype line	
`^N`	forward history	
`^P`	backward history	
`^U`	erase line	
`^V`	quote	
`^X`	delete char under	
`delete`	delete char under	delete char before
`back space`	delete char before	start of line
`return`	add to history	
`line feed`	add to history	
`home`	start of line	undefined
`end`	end of line	undefined
`^C`	interrupt	
`^Z`	end of file	
`^D`	end of file	
`left`	backward char	
`right`	forward char	
`up`	backward history	
`down`	forward history	

10.3.3 Initialization File

The `ile` initialization file is a list of table numbers, characters, and actions or strings. `ile` has 4 action tables. Each action table contains an action or string for each possible character. `ile` decides what to do with a character by looking it up in the table and executing the action associated with the character or by passing the string one character at a time into `ile` as if it had been typed by the user. Normally only table 0 is used. The escape actions cause the next character to be looked up in a different table. The escape actions make it possible to map multiple character sequences to actions.

By default, all entries in table 0 are bound to the insert action, and all entries in the other tables are bound to the bell action. User specified bindings override these defaults. The example in Table 3 is an initialization file that sets up the same key and delimiter bindings as the `ile` default bindings.

The first character on each key binding line is the index of the table to place the key binding in. Valid values for the index are 0, 1, 2, and 3.

The second character on the line is either the character to bind or an indicator that tells how to find out what character to bind. If the second character is any character besides '`^`' or '\' then the action is bound to that character.

If the second character on the line is '`^`' then the next character is taken as the name of a control character. So `^H` is backspace and `^[` is escape.

If the second character on the line is a '\' and the next character is a digit between 0 and 7 the the following characters are interpreted as an octal number that indicates which character to bind the action to. If the character immediately after the '\' is not an octal digit then the action is bound to that character. For example, to get the '`^`' character you would use '\`^`'.

The next character on the line is always '`=`'. Following the equal sign is the name of an action or a string. The actions are defined in the following table.

10.3.4 Actions

`bell` Send a bell (`^G`) character to the terminal. Hopefully the bell will ring. This action is a nice way to tell the user that an invalid sequence of keys has been typed.

`insert` Insert the character into the edit buffer. If there are already 75 characters in the buffer `ile` will beep and refuse to put the character in the buffer.

`delete_char` Delete the character directly to the left of the cursor from the edit buffer.

`delete_char_under` Delete the character under the cursor from the edit buffer.

`quote` The next character to come into `ile` will be inserted into the edit buffer. This allows you to put characters into the edit buffer that are bound to an action other than insert.

```
0\177=delete_char_under
0^@=escape_3
0^A=start_of_line
0^B=backward_char
0^C=pass_thru
0^D=pass_thru
0^E=end_of_line
0^F=forward_char
0^J=add_to_history
0^H=delete_char
0^K=erase_to_end_of_line
0^L=retype_line
0^M=add_to_history
0^N=forward_history
0^P=backward_history
0^U=erase_line
0^V=quote
0^X=delete_char_under
0^Z=pass_thru
0^[=escape_1

1[=escape_2

2A=backward_history
2B=forward_history
2C=forward_char
2D=backward_char

3\107=start_of_line
3\110=backward_history
3\113=backward_char
3\115=forward_char
3\117=end_of_line
3\120=forward_history
3\123=delete_char_under
```

Table 3: Example `ile.ini` file

`escape_1` Look up the next character in action table 1 instead of action table 0.

`escape_2` Look up the next character in action table 2 instead of action table 0.

`escape_3` Look up the next character in action table 3 instead of action table 0.

`start_of_line` Move the cursor to the left most character in the edit buffer.

`backward_char` Move the cursor to the left one character.

`end_of_line` Move the cursor past the last character in the edit buffer.

`forward_char` Move the cursor to the right one character.

`add_to_history` Add the contents of the edit buffer to the history buffer and pass the line along to the program running under `ile`.

`erase_line` Clear the line. Erase all characters on the line.

`erase_to_end_of_line` Delete the character under the cursor and all character to the left of the cursor from the edit buffer.

`retype_line` Retype the contents of the current edit buffer. This is handy when system messages or other asynchronous output has garbled the input line.

`forward_history` Display the next entry in the history buffer. If you are already at the most recent entry display a blank line. If you try to go forward past the blank line this command will beep at you.

`backward_history` Display the previous entry in the history buffer. If there are no older entries in the buffer, beep.

10.3.5 Strings

In addition to being able to bind a character sequence to an action `ile` allows characters sequences to be bound to strings of characters. When a string is invoked the characters in the string are treated as if they were typed by the user. For example, if the line:

```
0^G=ring^Ma^Mbell^M
```

was in your `ile.ini` file, typing control G would cause three lines to be typed as if the user typed them. Using the default bindings, unless there is a `^ J` or `^ M` in the string the string will be inserted in the current line but not sent along until the user actually presses return.

10.3.6 Error Messages

When `ile` encounters errors it prints a message and terminates. `ile` can print several standard error message. It can also print a few messages that are specific to `ile`.

- `ile:  '=' missing on line #`

  In a character binding line you left out the '`=`' character. Or, you did something that confused the initialization file reader into thinking there should be an '`=`' where you didn't think there should be one.

- `ile:  error in initialization file on line #`

  This means that the first character of a character binding line wasn't a newline or a 0, 1, 2, or 3. It could also mean that the initialization file reader is confused.

A misspelled action name in an `ile.ini` will be treated as a string. This means that typing the sequence of characters that should invoke the action will actually cause the misspelled name to be inserted in the input line.

10.3.7 Copyright

`ile` and this documentation was adapted from the program called `ile`. Permission to modify and distribute the program and its documentation is granted, subject to the inclusion of its copyright notice, which has been reproduced at the front of this manual.

11 Runtime Errors

Error messages describe most problems by printing the operation with the offending values of the arguments.

If `!source` was on when the program was read, you will get the source line where the error occurred. If `!stack` is on, lines containing the calls leading to this error will also be printed.

One possible problem is that some values are very big: {`1..1000`} for instance. Therefore, there are two forms of the error messages, controlled by the `!verbose` directive. By default, `verbose` is off and large values are represented by their type. The directive `!verbose on` results in full values being printed. `!verbose off` returns you to short messages. See figure 6 for an example.

11.1 Fatal Errors

The following errors cause `ISETL` to exit. Generally they indicate that the problem is larger than `ISETL` can manage. Please report cases where internal limits are exceeded to the author.

Message	**Explanation / Suggestions**
Includes too deeply nested	Probably file includes itself.
Out of parsing space	Internal limit exceeded.
Parser out of memory	Internal limit exceeded.
Too many locals	Internal limit exceeded.
Too many variables	Internal limit exceeded.

11.2 Operator Related Messages

Most errors print the offending expression with the values (or types) of the arguments. A few have additional information attached.

Additional	**Explanation**
`+`	May refer to `union`.
`*`	May refer to `inter`.
`<relation>`	Refers to any of the relational operators.
Boolean expected	May occur in `if`, `while`, `and`, `or`, `?`, and iterators.
Can't iterate over	Error in iterator.
in LHS of assignment	Error in selector on LHS.
Multiple images	Smap had multiple images.

11.3 General Errors

These errors do not provide context by printing the values involved, but they are generally more specific.

*	Used for self explanatory messages
internal	Messages the user should never see Please report to author.

Message	**Explanation**
Allocated data memory exhausted	Use `!memory` to raise limit.
Arithmetic error	Relates to machine limits
Bad arg to mcPrint	internal
Bad args in low,next..high	*
Bad args in low..high	*
Bad format in readf	*
Bad mmap in iterator	MMap iterator over non-map
Can't mmap string	Cannot perform selection in assignment
Can't mmap tuple	Cannot perform selection in assignment
Cannot edit except at top level	Edit not permitted within an include
Divide by zero	*
Exact format too big in readf	*
Floating point error	*
Input must be an expression	*
Internal object too large	*
Iter_Next	internal
Nesting too deep for pretty printer.	*
Only one level of selection allowed	See section 5
Return at top level	*
RHS in mmap assignment must be set	*
RHS in string slice assignment must be string	*
RHS in tuple slice assignment must be tuple	*
Return at top level	*
Slice lower bound too big	*
Slice upper bound too big	*
Stack Overflow	*
Stack Underflow	*
Too few arguments	*
Too many arguments	*
Wrong number of args	*

```
>       !memory
Current GC memory = 50060, Limit = 1024000
>       !allocate 800000
Current GC memory = 500600, Limit = 1024000
...
>       !memory 300000
Current GC memory = 50060, Limit = 300000
>       !allocate 150000
Current GC memory = 150180, Limit = 300000
```

Figure 4: Finding memory limits

```
>       f := func(i);
                 return f(i-1)+f(i-2);
             end;
>       !watch f
!'f' watched
>       f(1) := 1;
!  f(1) := 1;

>       f(2) := 1;
!  f(2) := 1;

>       f(4);
!  Evaluate:  f(4);
!  Evaluate:  f(3);
!  Evaluate:  f(2);
!  Yields:  1;
!  Evaluate:  f(1);
!  Yields:  1;
!  f returns:  2;
!  Evaluate:  f(2);
!  Yields:  1;
!  f returns:  3;
3;
```

Figure 5: `!watch` examples

```
>       !verbose on
>       {1..3} + 5;
!  Error -- Bad arguments in:
{3, 1, 2} + 5;

>       !verbose off
>       {1..3} + 5;
!  Error -- Bad arguments in:
!Set!  + 5;
```

Figure 6: Runtime errors

12 The ISETL Grammar — Compressed

12.1 Input at the Prompt

```
INPUT → PROGRAM
INPUT → STMT
INPUT → EXPR ;
```

12.2 Program

```
PROGRAM → program ID ; LOCALS VALUES STMTS end ;
```

12.3 Statements

```
STMT → LHS := EXPR ;
STMT → EXPR ;
STMT → if EXPR then STMTS ELSE-IFS ELSE-PART end ;

    ELSE-IFS → “Zero or more repetitions of ELSE-IF.”
    ELSE-IF → elseif EXPR then STMTS
    ELSE-PART → else STMTS

STMT → for ITERATOR do STMTS end ;
STMT → while EXPR do STMTS end ;
STMT → read LHS-LIST ;
STMT → read LHS-LIST from EXPR ;
STMT → readf PAIR-LIST ;
STMT → readf PAIR-LIST to EXPR ;
STMT → print EXPR-LIST ;
STMT → print EXPR-LIST to EXPR ;
STMT → printf PAIR-LIST ;
STMT → printf PAIR-LIST to EXPR ;
STMT → return ;
STMT → return EXPR ;
STMT → take LHS from LHS ;
STMT → take LHS frome LHS ;
STMT → take LHS fromb LHS ;
STMT → write PAIR-LIST ;
STMT → write PAIR-LIST to EXPR ;
STMT → writeln PAIR-LIST ;
```

`STMT` → `writeln PAIR-LIST to EXPR ;`

`STMTS` → "One or more instances of `STMT`. The final semicolon is optional."

`PAIR-LIST` → "One or more instances of `PAIR`, separated by commas."

`PAIR` → `EXPR : EXPR`
`PAIR` → `EXPR`

12.4 Iterators

`ITERATOR` → `ITER-LIST`
`ITERATOR` → `ITER-LIST | EXPR`

`ITER-LIST` → "One or more `SIMPLE-ITERATOR`s separated by commas."

`SIMPLE-ITERATOR` → `BOUND-LIST in EXPR`
`SIMPLE-ITERATOR` → `BOUND = ID ( BOUND-LIST )`
`SIMPLE-ITERATOR` → `BOUND = ID { BOUND-LIST }`

`BOUND-LIST` → "One or more instances of `BOUND`, separated by commas."

`BOUND` → `~`
`BOUND` → `ID`
`BOUND` → `[ BOUND-LIST ]`

12.5 Selectors

`SELECTOR` → `( EXPR-LIST )`
`SELECTOR` → `{ EXPR-LIST }`
`SELECTOR` → `( EXPR .. EXPR )`
`SELECTOR` → `( .. EXPR )`
`SELECTOR` → `( EXPR .. )`
`SELECTOR` → `( )`

12.6 Left Hand Sides

`LHS-LIST` → "One or more instances of `LHS`, separated by commas."

`LHS` → `ID`
`LHS` → `@ EXPR`
`LHS` → `@ ( EXPR , EXPR , ...  , EXPR)`
`LHS` → `LHS SELECTOR`
`LHS` → `[ LHS-LIST ]`

12.7 Expressions and Formers

`EXPR-LIST` → "One or more instances of `EXPR` separated by commas."

```
EXPR → ID
EXPR → INTEGER
EXPR → FLOATING-POINT
EXPR → STRING
EXPR → true
EXPR → false
EXPR → OM
EXPR → newat
EXPR → FUNC-CONST
EXPR → if EXPR then EXPR ELSE-IFS ELSE-PART end ;
EXPR → ( EXPR )
EXPR → [ FORMER ]
EXPR → { FORMER }
```

```
    FORMER → "Empty"
    FORMER → EXPR-LIST
    FORMER → EXPR .. EXPR
    FORMER → EXPR , EXPR .. EXPR
    FORMER → EXPR : ITERATOR
```

```
EXPR → # EXPR
EXPR → not EXPR
EXPR → + EXPR
EXPR → - EXPR
EXPR → EXPR SELECTOR
EXPR → EXPR . ID EXPR
EXPR → EXPR . (EXPR) EXPR
```

`EXPR` → `EXPR OP EXPR`
Possible operators (`OP`) are:

```
          + - * / div mod **
               with less
            = /= < > <= >=
     union inter in notin subset
           and or impl iff
```

`EXPR` → `% BINOP EXPR`
`EXPR` → `EXPR % BINOP EXPR`
`EXPR` → `EXPR ? EXPR`
`EXPR` → `choose ITER-LIST | EXPR`
`EXPR` → `exists ITER-LIST | EXPR`
`EXPR` → `forall ITER-LIST | EXPR`
`EXPR` → `EXPR where DEFNS end`
`EXPR` → `EXPR @ EXPR`

`BINOP` → "Any binary operator or an `ID` or expression in parentheses whose value is a function of two parameters. The `ID` and parenthesized expression may be preceded by a period."

The acceptable binary operators are: `+`, `-`, `*`, `**`, `union`, `inter`, `/`, `div`, `mod`, `with`, `less`, `and`, `or`, `impl`.

`DEFNS` → "Zero or more instances of `DEFN`. The final semicolon is optional."

`DEFN` → `BOUND := EXPR ;`
`DEFN` → `ID SELECTOR := EXPR ;`

12.8 Function Constants

`FUNC-CONST` → `FUNC-HEAD LOCALS VALUES STMTS end`
`FUNC-CONST` → `:  ID-LIST OPT-PART -> EXPR :`

`FUNC-HEAD` → `func ( ID-LIST OPT-PART ) ;`
`FUNC-HEAD` → `func ( OPT-PART ) ;`
`FUNC-HEAD` → `proc ( ID-LIST OPT-PART ) ;`
`FUNC-HEAD` → `func ( OPT-PART ) ;`

`OPT-PART` → `opt ID-LIST`
"May be omitted."

`LOCALS` → `local ID-LIST ;`

`VALUES` → `value ID-LIST ;`

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

Index