Cantor: a Tutorial and a User's Guide

(prototyping, set theory and all that)

Jean-Pierre Keller

volume II
Cantor: a Tutorial and a User's Guide

(prototyping, set theory and all that)

Jean-Pierre Keller

Kepler
8 rue des haies, F-75020 Paris

> what is prototyping?
deliver (i.e. give life and shape to) an abstraction
> why set theory?
because all the abstractions we may think of have models in set theory
> and what is all that?
that will be discussed now

Volume II: a Cantor User's Guide
Table of Contents

Cantor short history ............................................................ 1
Cantor technical profile ........................................................ 3
Cantor user's guide and overview ............................................ 4
  1 Running Cantor ............................................................ 4
  2 Characters, Keywords, and Identifiers ........................................... 9
    2.1 Character Set ................................................................... 9
    2.2 Keywords ..................................................................... 9
    2.3 Identifiers ................................................................... 9
  3 Simple Data Types ................................................................. 9
    3.1 Integers ................................................................. 10
       operations on integers .................................................. 10
       predicates on integers .................................................. 10
       some primitive functions on integers ................................. 11
       notes ...................................................................... 11
    3.2 Real (Floating_Point) Numbers ........................................... 12
       operations on floating point numbers (real numbers) ............... 12
       predicates on floating point numbers (real numbers) ............... 12
       some primitive functions on floating point numbers (real
       numbers) ................................................................ 13
       notes ...................................................................... 13
    3.3 Booleans ............................................................... 14
       operations on booleans ................................................. 14
       predicates on booleans .................................................. 14
       some primitive functions on booleans ................................... 15
    3.4 Strings ................................................................. 15
       operations on strings .................................................... 15
       predicates on strings .................................................... 15
       some primitive functions on strings ................................... 15
       notes ...................................................................... 16
    3.5 Atoms ................................................................. 19
       operations on atoms ..................................................... 19
       predicates on atoms ..................................................... 19
       some primitive functions on atoms .................................... 20
       notes ...................................................................... 20
    3.6 Files ................................................................. 20
       operations on files ....................................................... 20
       predicates on files ....................................................... 20
       some primitive functions on files ...................................... 20
       notes ...................................................................... 21
    3.7 Undefined ............................................................ 21
       operations on Undefined ............................................... 21
       predicates on Undefined ................................................ 21
       notes ...................................................................... 21
  4 Compound Data Types ................................................................. 22
    4.1 Sets ........................................................................ 22
       set expressions .......................................................... 22
       operations on sets ....................................................... 22
       predicates on sets ....................................................... 23
       some primitive functions on sets ..................................... 23
       notes ..................................................................... 24
    4.2 Tuples ................................................................... 24
       tuple expressions ........................................................ 24
       operations on tuples ..................................................... 25
       predicates on tuple ........................................................ 26
       some primitive functions on tuples .................................... 26
4.3 Maps ........................................................................ 27
operations on maps .......................................................... 27
some primitive functions on maps ....................................... 27
notes ........................................................................... 28
4.4 Formers ..................................................................... 28
4.5 Compound operators ................................................... 30
4.6 Quantifiers .................................................................. 31
4.7 Sample session ........................................................... 32
4.8 Exercises .................................................................. 33

5 Funcs ........................................................................ 33
5.1 func = l-expression + smap .......................................... 33
5.2 func specific semantics ................................................. 34
5.3 the pointer operation: ->, the scope designation: this ....... 35
5.4 some primitive functions of funcs and scopes ................ 37
5.5 Exercises .................................................................. 38

6 Abstract Syntax Trees .................................................. 38
6.1 operations on ast objects .............................................. 39
6.2 predicates on ast objects .............................................. 40
6.3 some ast analysis and interpretation primitive functions ... 40
6.4 a sample tutorial session on Ast ................................. 42
6.5 pattern matching ........................................................ 43
6.6 Exercises .................................................................. 44

7 Grammar ................................................................... 44
7.1 Terminology ............................................................. 44
7.2. Interactive Input ....................................................... 45
7.3 Program ................................................................... 45
7.4 Statements ............................................................... 45
    assignment statement ...................................................... 45
    call for expression evaluation ........................................... 46
    if statement .................................................................. 46
    for statement .................................................................. 46
    while statement ............................................................. 47
    read statement ............................................................. 47
    print statement ............................................................. 47
    return statement ........................................................... 48
    take .. from statement ...................................................... 48
    write statement ............................................................ 49
    formats ....................................................................... 49
7.5 Iterators .................................................................... 51
7.6 Formers .................................................................... 52
7.7 Selectors .................................................................... 52
7.8 Left Hand Sides .......................................................... 53
7.9 Expressions ............................................................... 54
7.10 User defined functions ............................................... 58
7.11 Precedence Rules ...................................................... 59

8 Directives .................................................................. 59
8.1 Cantor Commands ...................................................... 59
8.2 Cantor switches .......................................................... 61
8.3 !allocate and !memory .................................................. 63
8.4 !watch and !unwatch ................................................... 63
8.5 !record, !recordOutput .................................................. 64

9. The Cantor Grammar: condensed ................................. 64
9.1 Interactive Input ........................................................ 64
9.2 Program .................................................................... 64
9.3 Statements ............................................................... 65
9.4 Iterators ................................................................. 65
9.5 Selectors ............................................................................. 66
9.6 Left Hand Sides ....................................................................... 66
9.7 Expressions and Formers .......................................................... 66
9.8 User defined functions ............................................................ 67
10 Debugging ................................................................................ 67
  10.1 Runtime Errors ................................................................... 68
  10.2 Fatal Errors ......................................................................... 68
  10.3 Operator Related Messages ................................................ 68
  10.4 General Errors ..................................................................... 69
  10.5 Advanced trace and debugging facilities .............................. 69
      inserting code instead of stopping at a breakpoint ................ 70
      setting, re-setting breakpoints according to their kind .......... 70
appendix: Predefined Functions ..................................................... 72
Cantor short history

The history of Cantor belongs to that of set-oriented languages. The motto of this branch of computer science could be "programming is part of creative mathematics, i.e. exploration, formalization, design, modification, verification, ...and proof". That history is not very old, since its stone age is 1970.

1972 SETL (@ NYU)

1983 Ada/Ed (@ NYU)

1985 SED (ESPRIT) iSetl (@ Clarkson)

1989 Setl2 (@ NYU) Cantor (@ Kepler) eSetl (@ Essen)

Cantor's history is much younger, since it starts in 1985. This is partly a personal story:

The first time I really witnessed the need for a prototyping tool, I worked within a Research and Development Department of a large multinational corporation. The 'research' part of the activity had little to do with fundamental research. It consisted mainly in prototype developments for various applications. At that stage, applications were roughly identified
with a few essential functions: these were either an evolution of an existing well-
experimented service, or new ones 'defined' by gluing together expectations, e.g. functions
offered by related applications and known only from hearsay or a demonstration, say, at a
conference. No one had a very precise idea of the exact requirements. There was a distinctive
need for a comprehensive exploration of the feasibility of building a coherent application or
system to satisfy fuzzy requirements. Practically, the goal was to build a prototype
application, identifying all the functions, subfunctions and parameters, of a real industrial
application; this prototype could then be used and reviewed by a selected group of users to
determine the adequacy of the proposed services. This goal combined several studies: that of
a comprehensive set of requirements, with that of the feasibility and validity of planning an
effort for an industrial version. The objectives included also the identification of all the
expected pitfalls of a real industrial application design. Of course, that part of the objectives
was a mere wish. All the applications shared a somewhat similar profile: heavy symbolic
processing, together with numeric computations which did not require extreme precision and
an extremely diversified need for manipulating collections, relations and graphs (e.g.
dependency-reachability-graphs and their transitive closures). From this profile emerged the
need for a robust, flexible, and easy to use set-oriented programming environment.

This coincided in part with the objectives of the SETL programming environment. The SETL
project, some 10 years after its inception attracted much attention in 1983, when less than 2
weeks after the final definition of the Ada language was agreed upon, a working Ada system
was certified by the US-DOD. This Ada system is known as Ada/Ed\textsuperscript{1}. This was by no means
an industrial version, however it was working and ready for experimenting with the real
language not only with paper examples. Ada/Ed was used for several months by a large
community of users developing the Ada Test Suite for the later industrial versions.
Ada/Ed's development effort was evaluated at ca. three man-years (at most). Ada/Ed's
volume of source code was equivalent to the size of the Ada definition manual.

This prompted several investigation efforts, among which the SED ESPRIT project (1227) to
evaluate the adequacy of SETL as an industrial prototyping tool. This was a stimulating and
successful project which demonstrated several things, including how:
- complex computational geometry algorithms could be easily implemented and applied to
critically improve the working conditions of the personnel working on a specific cartography
application\textsuperscript{2}
- automatically, SETL code could be translated into a common programming language (Ada
was used as a target language, C would have been simpler)\textsuperscript{3}
- improving (i.e. optimizing) an algorithm from a 'naive' abstract problem statement, stated
in SETL, to an efficient solution was possible and could be automated as was done with the
RAPTS transformational system\textsuperscript{4}.
- complex semantic properties (e.g. the type of an object) could be inferred from a declaration-
free algorithm (i.e. declarations may be useless in a prototyping environment)\textsuperscript{5}.

\textsuperscript{1} Ed stands for 'Educational'. But many believe that this is just the first name of its Principal Investigator, Ed
Schonberg, from NYU.
\textsuperscript{2} Typically: manual processing of maps, consists in turning aerial or satellite pictures containing millions of
points into less than 2000 points. Picture processing tools are extremely helpful but no match to a human
operator for numerous tasks. SED developed an algorithm to reduce by a factor 10 to 30 the number of
interactive operations left to the human operator.
\textsuperscript{3} This result is reported in Doberkat E.-E., Gutenbeil U. : SETL to Ada - Tree Transformations Applied.
Information and Software Technology, 29, pp.548-557, 1987. When Doberkat and his team undertook this
translation research, all the experts told him: "this is impossible!".
\textsuperscript{4} RAPTS was developed by R. Paige. Many difficult algorithms have been since designed in this way:
transforming an abstract problem statement into an efficient implementation in SETL, and than translated
mechanically into C, or directly implemented into C.
\textsuperscript{5} This was the work of several people at CNAM and INRIA, including V. Donzeau-Gouge, C. Dubois, Ph.
Facon, and is reported in C. Dubois's Doctoral Dissertation: "Determination Statique des Types pour le
Langage SETL" (CNAM, juin 89)
SED produced several recommendations, which guided the design of Cantor whose current status is summarized below.

**Cantor technical profile**

Cantor is a very high-level programming language built around mathematical notation and objects, primarily sets and functions.

In its current version, Cantor has been derived from iSETL (copyright Gary Levin). Cantor has the usual collection of statements common to procedural languages, but a richer and simpler set of expressions. Primitive Cantor objects include:

- integers,
- floating point numbers,
- funcs (sub-programs),
- strings,
- sets,
- tuples (finite unbounded sequences).

Cantor objects include less traditional objects like:

- Abstract-Syntax Trees (AST),
- Windows,
- Buttons,
- Menus,
- Events.

The composite objects, sets and tuples, may contain any mixture of Cantor objects, nested to arbitrary depths. Cantor is essentially a declaration-free, weakly typed set-based language.

Among the major advanced features are the availability of:

1. functions as first-class objects and (dynamically constructable) lambda-expressions,
2. modularity, objects, classes, inheritance,
3. mechanisms to save (on a file) and restore objects of an arbitrary complexity (possibly including executable functions),
4. mechanisms for analyzing and transforming programs,
5. (interactive) graphics and text objects for multiwindow menu-oriented applications with their event processing mechanisms,
6. support to the 2-way interoperability with C-oriented applications.

The following comments will clarify these points:

1. Essential mechanisms for Software Engineering are available. For instance, the static binding discipline allows retention of links between objects created in deeply nested blocks, even beyond the exit of a higher level block. This implements in a very elegant and safe way information hiding. Cantor supports several very clean and efficient representations for objects, class properties, methods selection, inheritance.

2. The construction of the data-flow graphs is a rather simple program. Abstract Syntax Trees being a native type, Cantor programs may be analyzed and transformed by other Cantor programs: AST Pattern matching, unification and transformation primitives are available.

3. Cantor has both a traditional command console and a multiwindow menu-oriented user

---

6A major achievement by iSETL designers, inherited by Cantor
interface, and may be used for multiwindow applications. Cantor's event management allows
the development of reactive applications, in a very natural set-oriented way, mapping events
to their processing functions. Actually, all the computational resources may eventually be
captured as Cantor objects and subjected to set-based processing (e.g. to graph algorithms
for reachability, cycle-testing, topological sorting) or associated via maps to other objects or
processing functions (e.g. mapping Events or Event types onto specific functions).

5-Cantor programs may invoke-or be invoked from-applications using parameter passing
conventions compatible with those of C. It is indeed relatively easy to link Cantor with such
applications or packages. Parameters passed may be individual data items or homogeneous
collections.

This combination of features make Cantor a truly multiparadigm language, where imperative
and object oriented programming styles can be used simultaneously. The relative ease of
development of a Prolog interpreter in such a context makes declarative programming with
sets accessible too.

This tutorial is an introduction to the main features. Graphics, event processing, menus and
interactive user interface are not covered here.

This document is divided basically into:
-what is Cantor?, i.e., what is its syntax and what are its constructs?
-how to use Cantor?, i.e., what are application domains where Cantor is relevant, and how
could one use Cantor?

To answer the first question we will provide an illustrated Reference Manual. To answer the
second question we will review a set of programming examples.

Cantor user's guide and overview

Prerequisites are:
- a basic command of naive set theory
- the ability to understand grammars defined by their BNF (Backus-Naur Form)

This user's guide is concerned only with the basic methods and tools available with Cantor.
A number of advanced topics are not covered here. Mostly, event management, windowing
and graphics have been left out of the present guide, which concentrates on Cantor specific
algorithmic tools.

1 Running Cantor

Double-click the Cantor application icon or that of any already recorded Cantor document:

the Cantor application

a Cantor created Text document

a Cantor created binary document

---

7This section, as well as other part of this document has been borrowing in many occasions from G. Levin's
iSETL manual and test examples. Indeed, Cantor is a fully upward compatible extension of that of iSETL.
The following sections are those which owe the most to G. Levin's: §2,7,8,9

Kepler

Cantor: a User's Guide 4
This will create two windows, one titled 'console', and one titled 'stderr'. Copyright information is displayed in the console window, followed by a prompt ‘>’:

CANTOR interface alpha tests

**AEOpenDoc: BasicTest.r fdType(hex) 5854c538
Type(hex) 5854c538
macintosh environement installed
desired memory: 2048000
CANTOR v0.46.19 Kepler
based upon ISETL (1.9)
Last updated on 28/sept/94 at 18:17.
Copyright 1987,1988 (c):
==>
Gary Levin, Clarkson University
This version was compiled with THINK C 6.0(c)
==>
Yo Keller, Emmanuel Viennet, Marc Keller, Kepler
Copyright 1993,1994 (c):
==>
Yo Keller, Kepler
Enter !quit to exit.
constant buf preallocation: 191488 bytes
>
At this point the cursor is displayed idling on the right of the prompt. Cantor is an interactive system: it is ready for commands and instructions, coming either from the keyboard or from menu-selections.

When Cantor is running, it prompts for input with the characters >, ?, or %, depending upon its operating mode. Input consists of:

i- a sequence of expressions (each terminated by a semicolon ';'), statements, and programs. These follow the Cantor syntax (see §7). 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 double prompt >> (resp ??, % %), indicating that more is expected.

ii- directives

!<directive> [opt. param]*
for such task as compiling, redirecting input, (re-)setting debug options, loading, quitting, help etc. These directives are available as text oriented commands as well as menu options. Some directives may be invoked as instructions from Cantor program (see §8).

We will illustrate now some of these features.

Type now on the right of the prompt

```
1+2;
```

(the text of the expression followed by a semi-colon) and hit the ENTER or the Carriage Return key. The symbol ';' is the instruction terminator. Cantor understand this instruction as:

- compute the value of the expression 1+2
- display the value of this expression

Here is this session:

```
> 1+2;
3;
>
```

Type at the prompt:

```
!help cos
```
this is intended to supply the list of all the built-in functions whose name contains 'cos'. Here is this part of the session:

```plaintext
!help cos

$ .. 1
r := cos(x)
$ .. 2
r := acos(x)
$ .. 3
r := cosh(x)
$ .. 4
r := acosh(x)
```

The exclamation point introduces what is called a Cantor directive - in this example the help directive. Most directives may be obtained as interactive command selection in one of the menus in the Cantor menu bar. For instance, upon selecting help in the pulldown menu Cantor, a small dialog window opens up. The user is requested to complete the command, e.g. by typing in the dialog area cos and then the carriage return or enter key to validate that information. The dialog window disappears, and on the console window is displayed the same information as above. See §8 for a comprehensive list of directives. The most common directives and associated commands have short cuts, and therefore may be activated by typing only the short-cut at the key board (without an exclamation point). The most important ones are:

<table>
<thead>
<tr>
<th>interactive menu short cuts</th>
<th>full command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmd-H</code></td>
<td>!help &lt;pattern&gt;</td>
</tr>
<tr>
<td><code>cmd-I</code></td>
<td>!include &lt;filename&gt;</td>
</tr>
<tr>
<td><code>cmd-L</code></td>
<td>!load &lt;filename&gt;</td>
</tr>
<tr>
<td><code>cmd-R</code></td>
<td>!recordOutput &lt;filename&gt;</td>
</tr>
<tr>
<td><code>cmd-S</code></td>
<td>!save &lt;filename&gt;</td>
</tr>
<tr>
<td><code>cmd-W</code></td>
<td>!suspend</td>
</tr>
</tbody>
</table>

The parameter information <nnnn> is supplied by the user either on the directive line, following the exclamation point and the directive name, or in the dialog window generated by the menu selection process.

The help primitive is extremely useful. It tells which primitive are available and how to invoke them as in the following example:

```plaintext
$ .. 1
bool := is_window(x)
$ .. 2
idWin := openwindow(anAttrMap);
$ .. 3
closewindow(); $close current window
```

8 short cuts are case-insensitive, e.g., cmd-Q and cmd-q have the same effect: abort Cantor
When Cantor is running, it prompts for input with the characters >, ?, or %, depending upon its operating mode. There are three operating modes - the standard mode in which most user interaction will take place. The prompt is >. At launch time Cantor is in standard mode - the read mode in which Cantor is waiting for input at the request of a read instruction. The prompt is ?. - the nested mode, for which the prompt is %. The user may start a nested Cantor session by executing the instruction interp(""); or the command suspend (whose short cut is: cmd-w)

In nested mode Cantor behaves like in standard mode. However, normally hidden data, e.g. local variable values, may become visible if the nested mode is invoked when the execution is taking place within the scope of these hidden objects (i.e. during the execution of a func sub-program). Cantor's interactive debugging facilities use the nested mode at each breakpoint (see the watch and breakpoint directives). Following the detection of an execution error, often - depending how severe the error is- the execution flow is interrupted, and Cantor is placed in nested mode.

1. Cantor is exited by typing !quit. It may also be exited on the Macintosh by the quit command in the 'compiler' menu or its short cut cmd-q.

2. To exit from a read mode session one needs to complete as many syntactically correct expressions as are requested. Here is an example:

```
> read x,y;
? 10:{1,2..10};
> y;
{5, 6, 10, 9, 7, 8, 3, 4, 1, 2};
> x;
10;
> read x,y;
```
3. To exit from a nested mode session one needs to enter a `return;` instruction.

4. A common mistake is omitting the semicolon after an expression. Cantor will wait until it gets a semicolon before proceeding. The doubled prompt `>>` (resp `??`, `% %`) indicates that Cantor is expecting more input.

5. Cantor can get its input from sources other than the standard input.

(a) If there is a file with the name `.cantorrc` or `cantor.ini` in the current folder, then the first thing Cantor will do is read this file and execute the directives and instructions stated in it. These directives often require to include one or more files, defining an application (see below)

(b) If there is an available Cantor text file --- say `file.1` --- and Cantor is given (at any time), the following line of input,

```
!include file.1
```

then it will take its input from `file.1` before being ready for any further input. The contents of 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.2` contained the following data:

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

Then if the user typed,

```
> seta := {
  >> !include file.2
  !include file.2 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.2 completed` comes from Cantor and is always printed after an `!include`.

(c) If there is an available Cantor database (binary) file --- say `file.cntr` --- and Cantor is given (at any time), the following line of input,

```
!load file.cntr
```

then it will take its input from `file.cntr` before being ready for any further input. The contents of `file.cntr` is assumed to be a collection of persistent Cantor objects: data, (sub-)programs or a combination of both. These objects are restored as a separate component, with their Cantor identifiers (both global- and nested local- identifier scopes are restored).
6. Comments
If a dollar sign $ appears on a line, then everything that appears on the right of the $-sign until the end of the line is a comment and is ignored by Cantor.

7. 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.) However there is no automatic persistence in the sense: the variable values are automatically saved in a database. Instead, the user may organize for its own persistence needs. The user may save a whole session (with the !save directive) or a specific expression e.g. a map or a list of variables organized as a tuple, (with the save or store built-in functions)

2 Characters, Keywords, and Identifiers

2.1 Character Set
The following is a list of special characters used by Cantor.

[] }; = ! { } . # ? /* - + _ " < > % @ \ & $ 

In addition Cantor uses the standard alphanumeric characters:

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

and the following character-pairs .

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

2.2 Keywords
The following is a list of Cantor keywords.

and false iff not program true div for impl notin real read union

do forall in of readf value else from inter om return where elseif

fromb less opt subset while end frome local or take with exists

func mod print then write expr if newat printf to writeln this

2.3 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 Cantor 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 Cantor 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
The type of an expression x may be obtained either from the type testing predicates

bool := is_bignum(x); $ ...
bool := is_integer(x); $ ...
bool := is_floating(x); $ ...
bool := is_number(x); $ ...
bool := is_file(x); $ ...

Kepler Cantor: a User's Guide
bool := is_atom(x); $ ...
bool := is_boolean(x); $ ...

bool := is_cm(x); $ ...
bool := is_defined(x); $ ...

bool := is_string(x); $ ...

bool := is_func(x); $ ...
bool := is_ast(x); $ ...
bool := is_textpane(x); $ ...

bool := is_table(x); $ a map with domain and range
$ elements of simple type ...
bool := is_set(x); $ ...
bool := is_tuple(x); $ ...
bool := is_map(x); $ ...

or from the type function
type(x);

3.1 Integers

A cantor expression x is of type integer if
is_integer(x) = true;
holds. Actually, in that case, type(x) may be
"Integer" or "Bignum"
one has is_bignum(x) = true when x >= 2**15:
> x := 2**15;
> is_bignum(x);
true;
> is_bignum(x-1);
false;

operations on integers

Let x and y be Cantor expressions of type Integer or Bignum

x+y addition of x and y
x-y subtraction of x and y
+x x, without change
-x sign change for x
x*y product of x and y
x**y x to the power y:
  if y = 0 then 1
  elseif y < 0 then om
  else x * (x **(y-1))?!1
end
x div y integer division of x and y
x / y real-precision division of x and y
x mod y remainder in the division of x by y

predicates on integers

all predicates are expressions evaluating to either true or false

Kepler Cantor: a User's Guide 10
x=y  equality of x and y
x /= y  inequality of x and y
x < y  comparison for less than
x > y  comparison for greater than
x <= y  comparison for less than or equal to
x >= y  comparison for greater than or equal to
even(x)  test if x is a multiple of 2
odd(x)  test if x is odd
is_number(x)  true if x is an integer or a real

some primitive functions on integers
abs(x)  absolute value of x
ord(char)  n := ord(char); $ integer value of a character
char(n)  s := char(n); $ (ascii) char value of an integer
float(n)  convert n to a real number
random(root)  generate a random integer in the range (0..root) when root is an integer > 0
randomize (seed)  re-initialize the random number generator with a new seed
sgn(n)  sign of n
max(x,y)  the largest of x and y
min(x,y)  the smallest of x and y

some primitive functions on integers

1. There is no limit to the size of integers.9
   > the following expression computes 222!
   > $ apply the compound operator multiply = product
   > $ to all the numbers from 1 to 222
   > fact_222 := %* [ 1..222 ];
   > fact_222:
   112050758006441391828246578742888503316182344\n
9No practical limit. Actually limited to about 15,000 digits per integer on a Macintosh

Kepler  Cantor: a User's Guide  11
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.

```plaintext
> 123456\n>> 789;
123456789;
```

### 3.2 Real (Floating_Point) Numbers

A cantor expression x is of type real or floating point if

```plaintext
is_floating(x) = true;
```

holds. Actually, in that case, type(x) is

"Real"

this is illustrated here:

```plaintext
> type(1.0e2);
"Real";
> is_floating(1.0e2); true;
```

operations on floating point numbers (real numbers)

Let x and y be Cantor expressions of type real

- `x+y`: addition of x and y
- `x-y`: subtraction of x and y
- `+x`: x, without change
- `-x`: sign change for x
- `x*y`: product of x and y
- `x**y`: x to the power y:
  - if y = 0 then 1
  - elseif y < 0 then om
  - else x * (x **(y-1))?1
- `x/y`: real-precision division of x and y

```plaintext
> 2 ** 10.5; 1448.155;
> 2 ** 10.5 = 1024*sqrt(2); true;
> 7.8/19.1e-2; 40.838;
```

predicates on floating point numbers (real numbers)

all predicates are expressions evaluating to either true or false

- `x=y`: equality of x and y
- `x /= y`: inequality of x and y
\begin{itemize}
\item \texttt{x < y} \hspace{1cm} \text{comparison for less than}
\item \texttt{x > y} \hspace{1cm} \text{comparison for greater than}
\item \texttt{x <= y} \hspace{1cm} \text{comparison for less than or equal to}
\item \texttt{x >= y} \hspace{1cm} \text{comparison for greater than or equal to}
\item \texttt{is\_number(x)} \hspace{1cm} \text{true if \textit{x} is an integer or a real}
\end{itemize}

\textbf{some primitive functions on floating point numbers (real numbers)}

\begin{itemize}
\item \texttt{abs(x)} \hspace{1cm} \text{absolute value of \textit{x}}
\item \texttt{ceil(x)} \hspace{1cm} \text{n := ceil(real); \$ int. approx. of a real, see also floor, fix, round}
\item \texttt{fix(x)} \hspace{1cm} \text{n := fix(real); \$ int. approx. of a real, see also ceil, floor, round}
\item \texttt{floor(x)} \hspace{1cm} \text{n := floor(real); \$ int. approx. of a real, see also ceil, fix, round}
\item \texttt{round(x)} \hspace{1cm} \text{n := round(real); \$ int. approx. of a real, see also ceil, fix, floor}
\item \texttt{random(root)} \hspace{1cm} \text{generate a random real in the range (0..root) when root is a real > 0}
\item \texttt{randomize (seed)} \hspace{1cm} \text{re-initialize the random number generator with a new real seed}
\item \texttt{sgn(x)} \hspace{1cm} \text{sign of real expression \textit{x}}
\item \texttt{max(x,y)} \hspace{1cm} \text{the largest of \textit{x} and \textit{y}}
\item \texttt{min(x,y)} \hspace{1cm} \text{the smallest of \textit{x} and \textit{y}}
\item \texttt{trigonometry} \hspace{1cm} \text{the trigonometric functions: cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh}
\item \texttt{sqrt(x)} \hspace{1cm} \text{the square root of \textit{x}, equivalent to \textit{x} **0.5}
\item \texttt{logarithms} \hspace{1cm} \text{the ln (neperian log), log (base 10 log), exp (neperian exponentiation)}
\end{itemize}

\textbf{notes}
1. The possible range and precision of floating point numbers is machine dependent. At a minimum, the values will have 5 place accuracy, with a range of approximately 10^{3}.

2. A floating point constant is a sequence of one or more digits, followed by a decimal point, followed by zero or more digits. Thus, 2.0 and 2. are legal, but .5 is illegal. 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.

   > 1.0e-30 / 10;
   1.00000e-31;

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 INF (infinity):

   > 1.0e+125 ** 10;
   1.00000e+250;
   INF;

4. Cantor is weakly typed, and its primitive operations support polymorphic operations on numbers. An expression x is a number if is_number(x) is true. All the standard arithmetic operations (+, -, *, /, **) as well as the basic mathematics primitives (max, min, trigonometric functions) work with numbers:

   > cos(2);
   -0.416;
   > cos(2.0);
   -0.416;
   > 1. max 0.5;
   1;
   > 1. max 1.5;
   1.500;

3.3 Booleans

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

A cantor expression x is of type boolean if

   is_boolean(x) = true;

holds. Actually, in that case, type(x) is

   "Boolean"

   operations on booleans

Let x and y be Cantor expressions of type Boolean

   x or y or of x and y
   x and y and of x and y
   not x opposite of x
   x impl y x imply y, i.e. (not x) or y
   x iff y x impl y and y impl x

   predicates on booleans

Let u and v be arbitrary Cantor expressions
u = v  
inequality of u and v

some primitive functions on booleans

random(root)  
generate a random boolean in the range if

root is a boolean

3.4 Strings

A cantor expression s is of type string if

is_string(s) = true;

holds. Actually, in that case, type(x) is

"String"

operations on strings

Let s and t be Cantor expressions of type String and let i be an integer > 0

s + t  
concatenation of s and t

s(i)  
extracts the i-th character of s

s(i..j)  
substring containing all chars from i-th to

the j-th

s(..j)  
substring containing all the chars in s until

the j-th

s(i..)  
substring containing all the chars in s

starting with the i-th

s*i  
repeat - i.e. replicate - s i times. When i is

0, returns the empty string

or

i*s

predicates on strings

s = t  
equality of s and t

s /= t  
inequality of s and t

s in t  
true if s is a substring of t

some primitive functions on strings

random(true);
false;
random(true);
true;

> 'to be ...' + or not to be...';
"to be ... or not to be...";

> '123456789'(4);
"4";
>'123456789'(10);
OM;

> '123456789'(4..6);
"456";

> '123456789'((4..6);
"123456";

> '123456789'(4..);
"456789";

> 'to be'*3;
"to beto beto be";
> 3*'to be';
"to beto beto be";
> 'to be'*0;
"";

> 'o b' in ' or not to be...';
true;
> 'ob' in ' or not to be...';
false;
random(s) extracts randomly a character from the string s

rank(s,t) if s in t, returns the position of the 1st char of s in the leftmost occurrence of s in t, otherwise returns 0

ato

real := ato(floatingNbrString);

rtoa

str := rtoa(realNbr);

atoi

n := atoi(nbrAsString);

itoa

str := itoa(n); $ integer (or atom-value) to string conversion

ord

n := ord(char); $ integer value of a character

char

s := char(s); $ (ascii) char value of an integer

hash

int = hash(x) $ hash value

date

str := date(); $ current date, with the precision of a second

uclcase

s := uclcase((Uu)(Ll),string); $ convert string into upper (resp. lower) case

strsubst string := strsubst(pat, string,by); $ replace all occurrences of pat in string with by

scan

token_stream := scan(FilefileNamelonstring,
textScanIntextAndNumScan,strScan);

setScanStop aChar := setScanStop(aChar); $ aChar becomes the new scan stop char; default ScanStop is '.'

The important string function is scan, is described in detail, below in the notes.

notes

1. A string constant is any sequence of characters preceded and followed by double quotes. 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. I.e.
   "a sample string", 'another string' are two valid strings. A single quote (resp. double quote) may be freely used within a double-quote (resp. simple quote) bound string:
   "a string quote: ' may be used", 'a string quote: "may be used" are two valid strings.

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

2. the scan function is used to decompose a text file or a string into its token, i.e. its basic lexical components: words, operation symbols, punctuation, numbers. It always returns a tuple. Multiple switches control the behaviour of the scan function:

the input data: the 1st argument represents the input data. The second and third argument are used to control the scanning mode and the interpretation of the 1st argument.

1st and 3rd arguments

<table>
<thead>
<tr>
<th>1st arg is a File</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if that file is opened and is a text file, its contents, until the end of file or the text terminator - whichever comes first - is the input data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st arg is a String (3rd arg = om)</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if the third argument is undefined (om) that string represents a file name. scan will attempt to open that file as a text file for reading, and if successful, process it until the end of file or the terminator</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st arg is a String (3rd arg \ne om)</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>the first argument is the input text upon which scan will operate, if the third arg is different from OM</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st arg is om</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>use the console standard input as the scan input. The user should enter at the end of the text the current terminator symbol (by default it is ',')</td>
<td></td>
</tr>
</tbody>
</table>

By default, the terminator is ', '. This terminator character may be changed by calling setScanStop. If the terminator is encountered in the input data, the scan function stops processing the data:

```c
> s := "abrac cada bra";
> scan(s,1,1);
["abrac", "cada", "bra"];
> s := "abrac ca\da bra";
```

**Cantor: a User's Guide**
> scan(s,1,1);
> setScanStop('.');
> s := "abrac ca";
> scan(s,1,1);
> scan("-1 .0675",1,1);
> setScanStop('"');
> scan(om,om,om);
> turlutut chapeau pointu!
> setScanStop('§');
> scan('New York is a big City', "L", l);
> scan('New York is a big City', 'u', l);
> scan('New York is a big City', om, l);
> scan('New York is a big City', 1, 1);
> scan("-1.035", l, l);
> scan("-1.035", "", l);

Ordinary simple or double quotes, e.g. "a beautiful house", or 'a beautiful house' are used as string delimiters, and are considered ordinary special symbols:
> s := "a beautiful house";
> scan(s,1,1);
> s := "'a beautiful house'";
> scan(s,1,1);
> s := "he sings §a beautiful house§ while riding ";
> scan(s,1,1);

The second argument is used as a control switch:

<table>
<thead>
<tr>
<th>2nd arg</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'l', 'L'</td>
<td>the text is to be output in lower-case letters</td>
<td>&gt; scan('New York is a big City', 'L', l);</td>
</tr>
<tr>
<td>'u', 'U'</td>
<td>the text is to be output in upper-case letters</td>
<td>&gt; scan('New York is a big City', 'u', l);</td>
</tr>
<tr>
<td>om, &quot;&quot;</td>
<td>the text is to be output respecting the input case</td>
<td>scan('New York is a big City', om, l);</td>
</tr>
<tr>
<td>1</td>
<td>keep the settings used in the previous scan</td>
<td>&gt; scan('New York is a big City', 'L', l);</td>
</tr>
<tr>
<td>&quot;&quot;, &quot;ab&quot;</td>
<td>at least two characters: keep correct number formats</td>
<td>&gt; scan(&quot;1.035&quot;, 1, 1);</td>
</tr>
</tbody>
</table>

Kepler Cantor: a User's Guide 18
combine settings for upper or lower case and number formatting

The Cantor system variable `cantor_AlphaNumSet` is by default undefined. The user may set it to a string, or a set or tuple of strings. Each member of `cantor_AlphaNumSet` is then considered by the scan function as an ordinary alphanumeric symbol:

```plaintext
> cantor_AlphaNumSet;
OM;
> cantor_AlphaNumSet := "+";
> scan("a+b","",1);
["a+b"];
> cantor_AlphaNumSet := om;
> scan("a+b","",1);
["a", "+", "b"];
```

or even, demoting the ',' as a separator:

```plaintext
> cantor_AlphaNumSet := om;
> scan("a+b","",1);
["a", "+", "b"];
> scan("a+1,b+2",om,1);
["a", "+", 1, ",", "b", "+", 2];
> cantor_AlphaNumSet := "+";
> scan("a+1,b+2",om,1);
["a+1,b+2"];
> scan("a+1, b+2",om,1);
["a+1", ",", "b+2"];
```

When the 1st argument is the input string, it is destroyed by the scanning process:

```plaintext
> scan(s,'l',1);
["abrac", "cada", "bra"];
> s;
"
```

To force the use of a copy, add an empty string, that is, force the creation of a temporary expression which actually evaluates to the same value as the original string:

```plaintext
> s := "abrac cada bra";
> scan(s+'','l',1);
["abrac", "cada", "bra"];
```

3.5 Atoms

A cantor expression at is of type atom if

```plaintext
is_atom(at) = true;
```

holds. Actually, in that case, type(at) is "Atom"

```plaintext
operations on atoms
newat atom creation. This atom is unique

newat
```

predicates on atoms
some primitive functions on atoms

\texttt{itoat} \quad \text{atom(resp. int) := itoat(int(resp. atom))};$
\text{int (resp atom-value) to atom-value (resp int) conversion}

\text{if there is no atom corresponding to the given integer, returns an error}

\texttt{setBaseAtom} \quad \text{setBaseAtom(atom|om); $\text{set Base}_{-}\text{Atom to atom}$ !0!}

notes

1. Atoms are abstract points. They have no identifying properties other than their individual existence. The only operation on atoms is comparing two atoms for identity.

2. The keyword \texttt{newat} represents a constructor, acting as a function. \texttt{newat} has as its value an atom never before seen in this session of Cantor.

3.6 Files

A cantor expression $f$ is of type File if
\text{is\_file}(f) = true;
holds. Actually, in that case, type($f$) is "File"

operations on files

let $f$, $g$ be expressions of type File, let $\text{nam}$ be a string

predicates on files

\texttt{f=g} \quad \text{equality of s and t}
\texttt{f /= g} \quad \text{inequality of s and t}
\texttt{eof(f)} \quad \text{true if file pointer is at the end of file}

some primitive functions on files

\texttt{close(f)} \quad \text{close File f}
\texttt{opena(nam)} \quad f := opena(nam); $\text{open append text file nam}$
\texttt{openab(nam)} \quad f := openab(nam); $\text{open append binary file nam}$
\texttt{openr(nam)} \quad f := openr(nam); $\text{open read only text file nam}$
\texttt{openrb(nam)} \quad f := openrb(nam); $\text{open read only binary file nam}$
\texttt{openrw(nam)} \quad f := openrw(nam); $\text{open read-write text file nam}$

\text{10} In the current version atoms may be saved and restored: their uniqueness may not be guaranteed across sessions. The function \texttt{setBaseAtom} may be used to correct this situation.
openrwb(nam) \( f := \text{openrwb}(\text{nam}); \) \$open read-write binary file nam
openw(nam) \( f := \text{openw}(\text{nam}); \) \$open write text file nam
openwb(nam) \( f := \text{openwb}(\text{nam}); \) \$open write binary file nam

fwrite \( f \text{write}(\text{item},f); \$ \text{item type: integer, string, bignum} \\
\)
fread \( \text{value}_{\text{read}} := \text{fread} (\text{File}, 'int\text{'str\text{'big'}, count}); \$ \text{item type: integer, string, bignum count: always 1 for int} \\
fseek \( f\text{seek}(f, f_{\text{position}}); \$ \text{move file pointer to given file position} \\
f\text{tell} \( f_{\text{position}} := \text{ftell}(f); \$ \text{returns current file position} \\
\text{rewind} \( \text{rewind}(f); \$ \text{set file position to the beginning of the file} \\
\text{toend} \( \text{toend}(f); \$ \text{set file position to the end of the file} \\
\text{flen} \( n := \text{flen}(\text{ffileName}); \$ \text{file size} \\
\text{lc} \( n := \text{lc}(\text{ffileName}); \$ \text{text file line count} \\
\text{fgets} \( \text{string} := \text{fgets}(n,f); \$ \text{read a line of at most n char} \\

\text{notes} \\

1. A file is generally a Cantor value that corresponds to an \textit{external file} pointer in the \textit{operating system environment}. There are however two other kinds of files: the \textit{pane file} (corresponding to the data streams in a Cantor text-oriented window) and the \textit{data-base} (corresponding to files keeping persistent Cantor objects, including programs)

2. Common external files are created as a result of applying one of the pre-defined functions \textit{openr}, \textit{opena}, \textit{openw}, \textit{openrw} for text files and \textit{openrb}, \textit{openab}, \textit{openwb}, \textit{openrwb} for non-text (binary) files.

3. Pane files are created by applying the predefined functions \textit{open pane file}

4. Databases are created by the \textit{save}, \textit{store} or \textit{compile} predefined functions.

\textbf{3.7 Undefined} \\
A cantor expression \( x \) is of type Undefined if \( \text{is_om}(x) = \text{true}; \) holds. Actually, in that case, type(\( x \)) is "Undefined" \\
And the value of an undefined variable is OM

\textbf{operations on Undefined} \\
Let \( x, y \) be two arbitrary expressions

\( x?y \) this expression has value \( x \) if \( x \neq \text{om}, \) otherwise, has value \( y \)

\textbf{predicates on Undefined} \\
\( x=\text{om} \) true if \( \text{is_om}(x) = \text{true} \) \\
\( x\neq \text{om} \) true if \( \text{is_om}(x) = \text{false} \) \\
\( \text{is_defined}(x) \) true if \( \text{is_om}(x) = \text{false} \)

\textbf{notes} \\
1. The data type undefined has a single value --- \textit{OM}. It may also be entered as \textit{om}. 

\textit{Kepler} \hspace{1cm} \textit{Cantor: a User's Guide} \hspace{1cm} 21
2. Any identifier that has not been assigned a value has the value *OM*.

### 4 Compound Data Types

#### 4.1 Sets

A cantor expression *x* is of type set if

\[ \text{is set}(x) = \text{true}; \]

holds. Actually, in that case, \( \text{type}(x) \) is "Set"

---

**set expressions**

- **set in extension**: Zero or more expressions, separated by commas and enclosed in braces (\{ and \}) evaluate to the set whose elements are the values of the enclosed expressions. Note that as a special case, the empty set is denoted by \{\}.

- **slices**: A set of integers, may be defined by a slice \{i..j\}, meaning that its members are exactly all the integers from *i* to *j* (incl.), or by a slice with increment \{i1,i2..j\}, meaning all the integers *i* in increment \*k = i2-i1, starting at *i1* and such that \*li ≤ \*lj.

---

**set formers**

- **(in comprehension)**: A set may be defined as the subcollection of a given collection -set, tuple or string-, containing all the elements satisfying a given condition:

\[ \{t : t \in X \mid K(t)\} \]

or given a subcollection, as a derived set of expressions:

\[ \{ \text{exprm}(t) : t \in X \mid K(t)\} \]

the syntax supports very complex set formers. (see § 4.4)

---

**operations on sets**

Let *x* and *y* be sets. Let *t* be an arbitrary expression

- **union**: \( x+y \) or \( x \cup y \)

- **set difference**: \( x-y \) or \( x \setminus y \)

- **intersection**: \( x * y \) or \( x \cap y \)

---

---

---

---

---
form a new set by adding element t to the set x

form a new set by removing t from x

cardinality of x

remove an arbitrary element of x and assign it to variable t

observe that x has been changed

predicates on sets

true if x is a subset of y, i.e., all the members of x are members of y too

true if t is a member of x

true if t is not a member of x

some primitive functions on sets

if s is non-empty returns an 'arbitrary' element from s, otherwise returns om

power set of a given set s: the set of all subsets
npow(s,nmax), npow2(s,nmax)

set_collection := npow(s,nmax); $ all the subsets of s having exactly nmax elements

npow2 is a faster algorithm for the same

random(s) extracts randomly a member from the set

size(s) n := size(s); $ size in byte of s and its dependants

notes

1. Only finite sets may be represented in Cantor. 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. However OM is considered a neutral element in most set addition and deletion operations\(^{11}\): e.g. if the variable \(x\) has a set value, \(x\ with\ om\) has the same value.

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\(^{12}\).

4.2 Tuples

A cantor expression \(x\) is of type tuple if

\[
\text{is\_tuple}(x) = \text{true};
\]

holds. Actually, in that case, type(\(x\)) is

"Tuple"

tuple expressions

Syntactically, the rules for defining sets and tuples are very similar. Their main difference is the use of square brackets \([...]\) as delimiters of a tuple expression instead of \{...\} for sets.

\(^{11}\) iSETL users should be warned that in contrast, for iSETL, any set that would contain OM is considered to be undefined. I.e. in iSETL \(x := \{...\} with\ OM\) has the effect of setting \(x\) to OM

\(^{12}\) the sorted() predefined func allows some level of control over the enumeration ordering of sets. See §7.3
tuple in extension

Zero or more expressions, separated by commas and enclosed in square brackets ([ and ]) evaluates to the tuple whose elements are the values of the enclosed expressions, in the given order. Note that as a special case, the empty tuple is denoted by [ ].

slices

A tuple of integers, may be defined by a slice [i..j], meaning that its members are exactly all the integers from i to j (incl.) in that order, or by a slice with increment [i1..i2..j], meaning all the integers i in increment k = i2-i1, starting at i1 and such that |i1| ≤ |j|, in that order.

Slices are arithmetic progressions

A tuple may be defined as the ordered subcollection of a given collection, containing all the elements satisfying a given condition:

\[ \{ t : t \in x \mid K(t) \} \]

or given a subcollection, as a derived tuple of expressions:

\[ \{ \text{expr}(t) : t \in x \mid K(t) \} \]

the syntax supports very complex tuple formers. (see § 4.4)

operations on tuples

Let x and y be tuples, let t be an arbitrary expression, let i be an integer

\[ x+y \]

concatenation of x and y

\[ x*i \]

new tuple obtained by the replication i times of tuple x

or

i*x

\[ x(\ldots j) \]

form a new tuple made of all elements from the i-th to the j-th included

\[ x(\ldots ) \]

form a new tuple made of all elements from the 1st to the j-th included

\[ x(i..) \]

form a new tuple made of all elements from the i-th to the last one

\#x

cardinality of x
take \( t \) from \( x \) remove the first element of \( x \) and assign it to variable \( t \) observe that \( x \) has been changed (fromb is from the begining)

\[
x := [2,4..10];
take tb fromb x;
tb;
x;
\]

\( [4, 6, 8, 10] \);

\[
2;
\]

take \( t \) from \( e \) \( x \) remove the last element of \( x \) and assign it to variable \( t \) observe that \( x \) has been changed (frome is from the end)

\[
x := [2,4..10];
take te frome x;
te;
x;
\]

\( [2, 4, 6, 8] \);

\[
10;
\]

predicates on tuple

\( t \) \( \in \) \( x \) true if \( t \) is a member of \( x \)

\[
8 \in [2,4..10];
true;
\]

\( t \) \( \notin \) \( x \) true if \( t \) is not a member of \( x \)

\[
5 \in [2,4..10];
false;
\]

\[
5 \not\in [2,4..10];
true;
\]

some primitive functions on tuples

\( \text{rank}(t,x) \) if \( t \) is a member of \( x \), the rank is \( i \) iff \( t(i) = x \), and \( i \) is the smallest integer having this property, otherwise the rank is 0

\[
\text{rank}(8,[2,4..10]);
4;
\]

\[
[2,4..10](\text{rank}(8,[2,4..10])) = 8;
true;
\]

\[
\text{rank}('',[2,9]);
0;
\]

notes

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

2. OM is a legal value for a tuple member.

3. The order of the tuple members 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. The length or cardinality 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. It is obtained by applying the unary \# operation to a tuple expression.

\[
[1,3..100] \quad \text{a tuple of all positive odd integers less than 100}
t := [OM,'a string',10,[1..20],'another string',OM,[]];
\quad \text{a tuple of length 7.} t(4) \text{ is the set of all integers ranging from 1 to 20.} t(#t) \text{ is the empty tuple.}
\quad \text{For any integer i>7} t(i) \text{ has the value OM.}
\]

5. The function arb(s) is polymorphic and apply to all collections: sets, tuples, strings. Observe that \#s and size(s), althgough related, are independant. Similarly,

- the operations in, notin, \# (cardinality) are polymorphic over all collections,
- the operations + (concatenation), *(replication), the slice extraction operations and the function rank(s,t) are polymorphic over ordered collections (tuple, strings)
-the functions size(s), hash(s), random(s) are polymorphic over all types

6. The cardinality #s is an abstraction of the collection s: the number of elements in s. The size size(s) is the number of bytes, this implementation instance requires for representing s and its dependants.

4.3 Maps

A cantor expression x is of type map if

\[
\text{is map(x) = true;}
\]

holds. Actually, in that case, type(x) is "Map"

Maps form actually a subclass of sets. Thus,

\[
\text{is map(x) = true --> is set(x) = true}
\]

A map is exactly a table representing a binary relation, i.e. a set of pairs, e.g.

\{
[a,b],[c,d],....
\}

operations on maps

let m be a map

\[
m\{x\}
\]

it is by definition the image set i.e. the set of all images of x:

\{y: [x,y] in m \}

see notes 2,3 below

\[
m(x)
\]

if exists u in m \(m(1) = x\), and if this u is unique then

m(x) is u(2)

see notes 2,3 below

\[
m\{x\} := \text{aSet} \text{ (re-)defining m in x, i.e replacing the set of all pairs [x,y] in m by \{[x,u] \text{ in aSet}\}
\]

\[
m(x) := y \text{ (re-)defining m in x, by deleting from m, if there are any, all the pairs with x as 1st element, and adding to m the pair [x,y]}
\]

some primitive functions on maps

\[
\text{domain(m)}
\]

it is by definition:

\{x: [x,y] in m \}

the set of all pre-images, or of all 1st components of all the members of m
range(m)  it is by definition:  range(m);
    {y: [x,y] in m}

or  "binary op", "unary op";

image(m)

notes

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.

   F(EXPR) should be used only if F is a smap (see note 3).

   > s := ([arg1',10],[arg2',{}],[arg3','example']);
   > s(arg3);
     'example';
   > s(arg4);
     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 is no such pair, its value is
   the empty set.

   > s := { [arg1',10],[arg2',{}], [arg3','example'], [arg2',20]};
   > s('arg2');
     ! Error -- Bad mapping(multiple images):
     { ! Set! } ( "arg2" ) ;
   > s('arg2');
     (20, {});
   > s('arg3');
     "example";
   > s('arg4');
     OM;
   > s('arg4');
     {};

F(EXPR) may be used both for smap and mmap (see note 3). However F(EXPR) is
defined if F is not a map (i.e. a set of pairs)

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.
I.e., in the smap m, if [a,b] in m, then there is no member [a,c] of m with c ≠ b.

4.4 Formers

Formers are syntactic expressions to express an enumeration or an iteration. Sets and tuples
being collections, it is useful to collect here all the formation rules. Former are used in
defining expressions:

EXPR  -->  [ FORMER ]
EXPR --> { FORMER }

FORMER --> ε  
  empty, i.e. as in {}, []
FORMER --> EXPR-LIST  
  as in {expn1,expn2,expn3}, [expn1,expn2,expn3]
FORMER --> EXPR .. EXPR  
  i.e. a slice or arithmetic progression of 1, as in {1..10}
FORMER --> EXPR , EXPR .. EXPR  
  i.e. a slice or arithmetic progression of expr2-exprl, as in {1,-3..-10}
FORMER --> EXPR : ITERATOR  
  e.g. #x: x in s | 'a' notin x  
  or  x+y: x in s,y int I x > y**2

The syntax for ITERATOR is extremely versatile:

ITERATOR --> ITER-LIST  
ITERATOR --> ITER-LIST I EXPR  
  i.e. the expr here is a boolean expression, playing the role of a selection criteria as in  
  #x: x in s | 'a' notin x  
  consider only the elements x in s which satisfy 'a' notin x
The most common form of ITER-LIST is:

ITER-LIST --> SIMPLE-ITERATOR+  
  separated by commas
SIMPLE-ITERATOR --> BOUND-LIST in EXPR  
BOUND-LIST --> BOUND+  
  separated by commas
BOUND --> ID  
  in  x+y+z: x in s,y,z in t | x > (y**2+z) the ITER-LIST has two elements: x in s and  
y,z in t. In the 1st SIMPLE-ITERATOR the BOUND-LIST has a single element: x. In  
the 2nd SIMPLE-ITERATOR, the BOUND-LIST has 2 elements : y,z. As a whole, the  
bound variables in this example are x,y,z

However this is only the most common form. We provide here the full ITER-LIST grammar  
and then a set of running examples.

ITER-LIST --> SIMPLE-ITERATOR+  
  separated by commas
SIMPLE-ITERATOR --> BOUND-LIST in EXPR  
SIMPLE-ITERATOR --> BOUND = ID (BOUND-LIST )
SIMPLE-ITERATOR --> BOUND = ID {BOUND-LIST}
BOUND-LIST --> BOUND+  
  separated by commas
BOUND --> ~  
BOUND --> ID  
BOUND --> [ BOUND-LIST ]

We illustrate some of the possibilities of this with the following session:

> lt;
{[2, 5], [2, 4], [2, 3], [1, 5],
[1, 4], [1, 3], [1, 2], [4, 5],
[3, 5], [3, 4]};

> sentence := "un exemple";
> [[i,c] : c = sentence(i) | c = 'e'];
4.5 Compound operators

Let us consider an operation \( \text{op} \) on \( A \times B \rightarrow A' \), where \( A' \subseteq A \).

An operation like this could be one of the built-in binary operations, e.g.,
\( +, *, **, \div, \text{mod} \),
or any (built-in or user defined) 2-ary function \( f: A \times B \rightarrow A' \), e.g.,
\( \min, \max, \text{npow} \).

In the above examples, \( A, A' \) and \( B, \) are number sets (real \( R \) or integer \( N \)) or \( S \), the collection of all expressions of type set, \( T \), that of tuples, \( S' \) that of strings, e.g.
\( \text{tdiv}: N \times N \rightarrow N \)
\( /: (R+N) \times (R+N) \rightarrow R \) where \( R \subseteq R+N \)
\( \text{npow}: S \times N \rightarrow S \).

For any such operation or function, a repeated application over a given collection is possible.
Let a \( \in A \) and let \( [b_1, b_2, ..., b_n] \) be a tuple of elements of \( B \). Then
\[ a \ \text{op} \ b_1 \ \text{op} \ b_2 \ \text{op} \ ... \ \text{op} \ b_n \]
is well-defined and may be written, in Cantor as a compound operator, signaled by the \% (percent) sign:
\[ a \ \%\text{op} \ [b_1, b_2, ..., b_n] \]
or
\[ \%\text{op} \ [a, b_1, b_2, ..., b_n] \]
For instance:
\[ \%+ \ [1..j] \] is the sum of all integers from 1 to \( j \) and \( \%**[2,2,2] \) is
\[ 2^{2^2} \]
Cantor allows the application of compound operators to unordered collection (sets):
\[ a \ \%\text{op} \ [b_1, b_2, ..., b_n] \]
\[ \%\text{op} \ [a, b_1, b_2, ..., b_n] \]
In that case the enumeration \( b_1, b_2, ..., b_n \) of the elements is in an arbitrary order. And repeated computations of \( a \ %\text{op} \ [b_1, b_2, ..., b_n] \) or \( \%\text{op} \ [a, b_1, b_2, ..., b_n] \) may yield different results if the commutativity properties of the operation are not guaranteed. If however
op is a well-defined and commutative operation $A \times A \to A$, than a $a \% op \{ b_1, b_2, \ldots, b_n \}$ may be written $\% op \{ a, b_1, b_2, \ldots, b_n \}$. In that case, the left most term in $a \ op b_1 \ op b_2 \ op \ldots \ op b_n$ i.e. the term playing the role of $a$, is selected arbitrarily in the argument set.

Formally:
- $\% op \[]$ is om
- $\% op \{}$ is om
- $\% op \{ b \}$ is $b$
- $\% op \{ b \}$ is $b$
- $\% op \{ t \ with \ b \}$ is:
  - $(\% op \ t) \ op b$ if $op$ is a binary operation or
  - $op \ (\% op \ t, b)$ if $op$ is a function with 2 arguments.

4.6 Quantifiers

Given the formation rules for composite expressions, it is a relatively easy task to introduce the quantifiers exists (corresponding to $\exists$) and forall (corresponding to $\forall$)

$EXPR \rightarrow \text{exists} \ \text{ITER-LIST} \ \text{I} \ \text{EXPR}$

EXPR evaluates 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.

```plaintext
> p := [1 .. 100];
> exists j in p \mid j < 0;
false;
> exists j in p, i in [2 .. j] \mid j = i**2;
true;
```

Note that in this example, the values $i$ and $j$ which satisfy the conditions are not accessible: these are bound variables. Previous settings for variables $i$ and $j$ has not been changed, by the side-effect free execution of this quantifier. We will see later, in the section on funcs how to create a side-effect to gain access to the values of the bound variables which meet the condition.

$EXPR \rightarrow \text{forall} \ \text{ITER-LIST} \ \text{I} \ \text{EXPR}$

EXPR evaluates 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.

```plaintext
> primes := [i: i in [2 .. 1000] \mid forall j in [2 .. floor(sqrt(i))]
> \mid i mod j /= 0];
> primes;
```
The following is a self-explanatory demonstration of bulk structures, i.e. entities of type set, tuple and map:

> $ bulk structures don't have to be homogeneous
> t := t with [-12.0];
> #t;
31;
> (t[#1]);
[-1, 0, -2, -4, -3, -7, -8, -6, -5, -10, -9, -12, -11];
> t;
[1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, 73, 77, 81, 85, 89, 93, 97, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM, OM];
> s := [t[], 'a sample string'];
> #s;
3;
> u := s with t;
> #u;
3;
> u=s;
true;

> $ bulk structures may be defined by set-fromers (or tuple-formers):

> primes := [i in [2..1000] | forall j in [2..floor(sqrt(i))] -> i mod j /= 0];
> #primes;
168;
> type(uXv);
"Map";
> uXv;
[("a", "b")], ["a", "c"], ["a", "d"], ["b", "c"], ["b", "d"], ["c", "d"], ["a", "b", "c", "d"];
> type(uXv);
"Map";
> uXv;
[("a", "b")], ["a", "c"], ["a", "d"], ["b", "c"], ["b", "d"], ["a", "b", "c", "d"];
> type(uXv);
"Map";
> $ the relation represented by a map may be single-valued or
> multiply valued
> $ it depends upon the card of the image of each domain element
> elt := arb(domain(aMap));
> elt;
6;
> aMap[elt];
["a", "b", "c", "d"];
4.8 Exercises

- write an expression which evaluates to the set of all multiples of 7 or 11 less than 1000.
  What is the cardinality of that set
- compute the sum of all multiples of 7 or 11 less than 1000
- compute the product of all multiples of 7 or 11 less than 1000
- write an expression which evaluates to the list of all multiples of 7 or 11 less than 1000, in ascending order
  - verify that the following expression evaluates to the truth table of and:
    \[ \{([x, y], x \land y) : x, y \in \{\text{true}, \text{false}\}\} \]
    See section 7.4 for indications on formatted output: print out this expression in the format of a truth-table
- write an expression which evaluates to the truth table of xor (exclusive-or is not a built-in Cantor operator. It is defined as follows:
  - \( a \text{ xor } b = (a \land \neg b) \lor (b \land \neg a) \)
- are truth tables always maps? smaps? mmaps?
- is it possible to write a tuple former which evaluates exactly to the 1st 100 primes?
- write a set former \( \{ \text{EXPR: } x \text{ in m} \mid \ldots \} \) which evaluates to a largest map contained in m. Apply this to R and uXv above.

5 Funcs

A cantor expression \( x \) is of type \( \text{func} \) if
\[
\text{is_func}(x) = \text{true};
\]
holds. Actually, in that case, \( \text{type}(x) \) is either
"Closure" or "Predef".
Whenever type(\( x \)) = "Closure", \( x \) is a user-defined function, otherwise, \( x \) is a Cantor built-in or 'predefined' functions.

5.1 \( \text{func} = \lambda\)-expression + smap

1. A func is a Cantor 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, a Cantor func can be assigned to an identifier, passed as an argument, etc. A func is
what is often called a λ-expression. Evaluation of a Cantor 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

\[
\begin{align*}
& \text{return } expr; \quad \text{returns the value of } expr; \\
& \text{return;} \quad \text{returns } OM.
\end{align*}
\]

Cantor inserts a return; statement 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 assignments, so can funcs. However, if the value at a point is structured, you may not access or modify individually the members of this structure, at that point.

```cpp
> 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, because you are trying to modify the structure of the value returned.

4. A number of functions (over four hundred) have been pre-defined as funcs in Cantor. Their list and a short description, equivalent to that provided by the online help, is given in section 9. 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 funcs. This is done with the following func syntax:

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

5.2 func specific semantics

1. The declaration of local ids may be omitted if no local variables are needed. The ids declared in a value list represent 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.

2. Parameters and local-ids are local to the func. See below, alinea #5, for a discussion of scope.

3. 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 assignment Cantor will be able to evaluate an expression such as cube_plus(2,5) as 13.

4. 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.

5. 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.

6. 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 Cantor 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.

7. 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).

5.3 the pointer operation: ->, the scope designation: this

Pointer expressions may be defined as follows:

    f -> EXPR

the expression f on the left of the -> (pointer sign) designates the scope in which f was created. The expression on the right of the pointer sign is an expression which must be evaluated in that scope. Observe that the only expressions which might appear meaningfully on the lhs of -> are expressions which evaluate to a func. For all other expression types,
since no scope creation is recorded, they refer to the outermost (global) scope.

> cf := carol('deposit');
> cf->balance;
1000;
> cf->name;
"Carol Simon Levin";

The pointer operation may be used very efficiently to change any package (i.e. a set of nested func's sharing some private memory and functions) into a class structure with simple inheritance.

The scope designation this is analogous to the object designation this in C++ or the self of Smalltalk. When a func is designed to be invoked from many different scopes, the objects it is referring to may change from invocation to invocation. Indeed, not only the actual arguments passed participate in the computation but the whole scope, including its hidden objects, may participate too. The way to refer explicitly to the variable scope is by means of the scope designator this. For instance,

this->x
refers to the definition of x in the execution time scope.

The following example illustrates the role played by -> and this.

```c
$ "a "class" point, its constructor and methods
figures := func();
local pointSet, segmentSet, class, vars, methods;
vars := ['class', 'vars', 'methods'];
methods := {};
class := 'figures';
new := {};
$
pointSet := {};
point := func();

variables

local u, v, class, vars, methods; $ class
local translate, dist, homothetia; $ methods
$ par default les points sont à l'origine
u := 0; v := 0; class := 'point';
$ vars, methods : could be computed with
recollect...
vars := ['u', 'v'];
methods := ['translate', 'dist', 'homothetia'];
newP := func(x, y);
local zpt;
x := x + u;
y := y + v;
zpt := newP; pass; end;
pointSet := pointSet with zpt;
return zpt;
end; $ end newP
new('point') := newP;
translate := func(dx, dy);
this->x := this->x + dx;
this->y := this->y + dy;
end; $ end translate
dist := func(pt);
if pt->class /= 'point' then return
om; end;
return sqrt((pt->x - this->x)**2 + (pt->y - this->y)**2);
end; $ end dist
homothetia := func(factor);
$ homothetia: center is pt, factor
is scaling factor
coordinates are integers
if pt->class /= 'point' or not
is_number(factor) then
pl->x = fix((pl->x)*factor);
if pl->class = 'point' then
this->X := pl->X + fix((this->X -
pl->y := pt->y + fix((this->y -
end; $ end homothetia

$ invoke the constructor of the class 'figures'
figures(); $ figures is the super class of 'point'
end; $ end figures
end
$ random creation of points
for i in [1 .. 100] do

Kepler
Cantor: a User’s Guide 36
To analyze this program, notice that the text indentation describes the actual nesting of scopes at creation time. For instance, as a scope definition `point` is just an object which knows of all the private (e.g. 'local') variables of `figues`. Thus

> point->class;
> "figues";

While, as a func, `point` owns a private variable named `class`, whose value is not known in `point`-scope, but is known within the scope of any other variable created within the func `point`, e.g.:

> newP->class;
> "point";
> arb(point->pointSet)->class;
> "point";

Consider an arbitrary point `pt` in `point->pointSet`. When scaling is carried out by invoking `pt->homothetia(first_point,0.2);` the procedure `homothetia` within the scope of `pt` is invoked. While executing that procedure, `this` refers precisely to the scope within which it is invoked, i.e. the scope of `pt`, whence `this->x` is `pt->x`, `this->y` is `pt->y`, at that time.

Try running this program. Some modifications will be suggested in an exercise.

### 5.4 some primitive functions of funcs and scopes

The term 'environment' designates a specific scope. Each user defined func is characterized by its code, its environment, its redefinition (override) map. By default, the environment is the largest possible scope granting access to all the global objects available in the current session of Cantor.

- `applyEnv`: `fn1 := applyEnv(fn opt optEnv);` $set fn1 env to optEnv if specified, otherwise to the current env
- `applyNilEnv`: `fn1 := applyNilEnv(fn);` $set fn1 env to the current env
- `hasNilEnv`: `bool := hasNilEnv(fn);` $true if fn is a func with Nil Env
- `detachEnv`: `env := detachEnv();` $unlink the current func's env from creator's
- `codeOf`: `code := codeOf(func);` $code is a non-printable object
- `overrideOf`: `aMap := overrideOf(func);` $aMap is a s-map
- `envOf`: `env := envOf(func);` $env is a non-printable object
- `mkLocal`: `mkLocal(idName,aFunc);` $creates a local var in the scope of aFunc
- `tellfunc`: `aFunc := tellfunc();` $attempts to tell within which func is current program ptr
- `ref`: `ref(afunc);` $returns a map describing the list of the identifiers referenced by afunc. If the afunc has not been compiled under 'refcollect', returns only the parameter list, and the local and value identifiers. Under 'refcollect', the non-local identifiers used by afunc re also produced
5.5 Exercises

-the factorial function may be defined by:

```
fact := func(n);
if not is integer(n) then return om; end;
if n <=1 then return 1
else return n*fact(n-1);
end;
end; $ end fact
```

It is better programming practice to tabulate than always re-evaluate:

```
tab_fact := func(n);
if not is integer(n) then return om; end;
if n <=1 then return 1
else
  tab_fact(n) := n*tab_fact (n-1);
end;
return tab_fact (n);
end; $ end tab_fact
```

Compare the performance of fact and tab_fact for n = 5,10,15,20,100. Use the date() or the clock() primitives. Compare also with the expression %*[1..n] . See also §8.2, an execution trace for tab_fact.

- Create a func for computing the Fibonnacci sequence:

```
1,2, Fibonnacci (n+2) = Fibonnacci (n+1)+Fibonnacci (n).
```

Create the associated tab_Fibonnacci func and compare the performance for n = 5,10,15,20,100.

- Let Keep_ and gkeep() be defined as follows:

```
Keep_ := om; gkeep := func(x); Keep_ := x; return true; end;
```

Show how this may be used to inform on the status of bound variables in quantifiers. Illustrate this by exhibiting the first pair [i,j] which given p := [1..100]; satisfies:

```
exists j in p, i in [2 .. j] I j = i**2;
```
or violates

```
forall i in p, j in [2 .. i-1] I i = j**2;
```

- Modify the figues program above to include a rotate(center, angle) function among the point functions.

- Modify the figues program to associate with each point a string. Use gputs(aStr) to display a string at a given location in the graphics window.

- Modify the figues program to include new 'classes' correponding to point groupings: segments, triangles, as well as specific kinds of triangles: rectangular, isosceles, equilateral. For each kind provide direct way of creating a new object of that kind, of displaying that object in the graphics window, of translating, rotating, scaling the object.

6 Abstract Syntax Trees

A cantor expression x is of type Abstract Syntax Tree (ast) if

```
is ast(x) = true;
```
holds. Actually, in that case, type(x)

```
"AST"
```

Each syntactic category is characterized by its ast_kind:

<table>
<thead>
<tr>
<th>ast</th>
<th>ast kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>x+y</td>
<td>+''</td>
</tr>
<tr>
<td>x*y</td>
<td>*&quot;</td>
</tr>
<tr>
<td>exists x in s I K(x)</td>
<td>&quot;exists&quot;</td>
</tr>
</tbody>
</table>
Given an ast expression af, its ast_kind (in string form) is the value of which_ast(af). The ast_kind has an internal code: af('t'). The function which_ast converts all the forms into one another.

In the table below the major syntactic categories are listed (as unquoted strings):

<table>
<thead>
<tr>
<th>T_Pat</th>
<th>&lt;</th>
<th>etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_Cmt</td>
<td>&lt;=</td>
<td>T_Missing</td>
</tr>
<tr>
<td>T_Spec</td>
<td>=</td>
<td>take</td>
</tr>
<tr>
<td>T_Id</td>
<td>&gt;</td>
<td>do</td>
</tr>
<tr>
<td>T_String</td>
<td>&gt;=</td>
<td>INFIX</td>
</tr>
<tr>
<td>T_Real</td>
<td>+</td>
<td>INFIX</td>
</tr>
<tr>
<td>T_Boolean</td>
<td>-</td>
<td>end</td>
</tr>
<tr>
<td>CLEAR</td>
<td>:=</td>
<td>with</td>
</tr>
<tr>
<td>:=</td>
<td>where</td>
<td>less</td>
</tr>
<tr>
<td>:=</td>
<td>if</td>
<td>*</td>
</tr>
<tr>
<td>:=</td>
<td>impl</td>
<td>/</td>
</tr>
<tr>
<td>:=</td>
<td>or</td>
<td>mod</td>
</tr>
<tr>
<td>:=</td>
<td>and</td>
<td>div</td>
</tr>
<tr>
<td>:=</td>
<td>not</td>
<td>**</td>
</tr>
<tr>
<td>:=</td>
<td>in</td>
<td>%</td>
</tr>
<tr>
<td>:=</td>
<td>notin</td>
<td>UNARY</td>
</tr>
<tr>
<td>:=</td>
<td>subset</td>
<td>CALL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T_Of</th>
<th>-&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>om</td>
<td>?</td>
</tr>
<tr>
<td>opt</td>
<td>#</td>
</tr>
<tr>
<td>print</td>
<td>[</td>
</tr>
<tr>
<td>printf</td>
<td>{</td>
</tr>
<tr>
<td>do</td>
<td>program</td>
</tr>
<tr>
<td>end</td>
<td>return</td>
</tr>
<tr>
<td>with</td>
<td>exists</td>
</tr>
<tr>
<td>exists</td>
<td>then</td>
</tr>
<tr>
<td>less</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>value</td>
</tr>
<tr>
<td>forall</td>
<td>while</td>
</tr>
<tr>
<td>from</td>
<td>write</td>
</tr>
<tr>
<td>from</td>
<td>writeln</td>
</tr>
<tr>
<td>from</td>
<td>MAP</td>
</tr>
<tr>
<td>func</td>
<td>SELECTOR</td>
</tr>
<tr>
<td>if</td>
<td>ITERS</td>
</tr>
<tr>
<td>this</td>
<td></td>
</tr>
</tbody>
</table>

6.1 operations on ast objects

let af be an ast, and ley i be an integer in [0..2].

af(i)     subscripts indicate a filiation in the abstract syntax tree:

or

ast(af,i) - af(1) is the 1st or left subtree
- af(2) is the 12nd or right subtree
- af(0) is the ancestor tree, if af is an internal node within an ast

the binary operation ast is equivalent to subscripting an abstract syntax tree

at a leaf node, the subtrees are either a string or om

WARNING: ast is the name of a predefined (built-in) function

<p>| &gt; af; |
| x + y; |
| &gt; type(af); |
| &quot;AST&quot;; |
| &gt; is_ast(af); |
| true; |
| &gt; which_ast(af); |
| &quot;CALL&quot;; |
| &gt; which_ast(af(1)); |
| &quot;+&quot;; |
| &gt; b := af(1); |
| &gt; which_ast(b(0)); |
| &quot;CALL&quot;; |
| &gt; b(1); |
| x; |
| &gt; b(1)(1); |
| &quot;x&quot;; |</p>
<table>
<thead>
<tr>
<th>af('t')</th>
<th>'typ' or any initial segment thereof, is the subscript leading to the ast_kind code. See also which_ast()</th>
</tr>
</thead>
<tbody>
<tr>
<td>or ast(af,'t')</td>
<td></td>
</tr>
</tbody>
</table>

af %ast [...] | compound subscripting defines paths from a tree root to a subtree |
| or %ast [af ...] |                                                                 |

### 6.2 Predicates on Ast Objects

<table>
<thead>
<tr>
<th><strong>is_ast_leaf(a)</strong></th>
<th>true if a is a leaf node: identifier, or a constant node: identifier: &quot;T_Id&quot;, integer: &quot;T_Integer&quot;, real: &quot;T_Real&quot;, string: &quot;T_String&quot;, boolean: &quot;true&quot;, &quot;false&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.3 Some Ast Analysis and Interpretation Primitive Functions

<table>
<thead>
<tr>
<th><strong>which_ast(af)</strong></th>
<th>-if af is an ast, returns a string designating the ast_kind of af; -if af is an integer, it is considered as the internal code for an ast_kind, and the corresponding string is returned -if af is a string corresponding to a known ast_kind, its corresponding internal code is returned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>analyze</strong></td>
<td>analyze, either reads an input from the stdin input stream, or from a string argument, performs parsing and build an ast see also scan, construct</td>
</tr>
<tr>
<td><strong>construct</strong></td>
<td>similar to analyze. However, if an identifier in the parsed expression is the identifier for an abstract syntax tree, that tree is substituted in: ordinary Cantor variables may be used as tree variables see also scan, analyze</td>
</tr>
</tbody>
</table>

---
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAst</td>
<td>anAst := setAst(anAst,-1011121'type'l3, modif);$ modify anAst: -1:parent, 0:node itself,1:left,2:right,&gt;=3:more</td>
</tr>
</tbody>
</table>

N.B. unlike in ast subscripting, 0 does not represent an ancestor node, but the node itself

| pretty | produces a string which is the pretty print of the ast arg |
| ugly | displays the tree-like structure of its ast argument, including the ast_kind of each node |

| findAst | findAst(atyp,af);$ atyp = intlstring|set|om;\ produces a tuple of all the af sub-trees with the given type |

(see examples in session 6.4)

| eval(af) | evaluate an ast as an executable expression |

(see examples in session 6.4)

| interp(stmt_str) | interp is similar to eval, but takes as input a text string, which is supposed to represent a statement. That text is parsed and evaluated. If the stmt_str is an empty string, then Cantor suspends the current session and enters a nested session |

| scan | scan(File|file|Name|elom|string, , textScan|text|And|NumScan, strScan); $ produces a tuple -- ScanStop (terminator) is 'i'. textScan /= om --> spec. symbols (eg $, quotes) are parsed as tokens and returned strScan /= om --> 1st arg = input string char in cantor_AlphaNumSet are considered alpha-num |

see also analyze, construct, setScanStop, and the notes in section 3.4

| kw|d | kw|d(opt key, token_val); $ if key = om returns tuple of all keys, if key = " (empty string) reset all keywords to default $ if key = some_str returns the corresponding token, $ if token_val = 0 resets the key, else sets the key to the given token_val |

N.B. the above mentioned variable cantor_AlphaNumSet is a global variable which is accessed by the Cantor system, when parsing texts with scan, analyze, construct or scan.
6.4 a sample tutorial session on Ast

```lisp
# we will present here a subset of the available
# AST-processing
# tools:
# analyze          the basic parser
# pretty, ugly    pretty-printer, ugly-
# printer         evaluates an AST
# eval             to tell the type of an
# AST             sub-AST of a given type
# (or an identifier string)
# which_ast        refcollect a toggle for saving the
#                  reference data obtained at compile-time
# ref             to produce the
#                  reference map of the variables referenced within a
# func             sub-AST to link AST-nodes to
# parent node
# use the built-in help to find how to use these primitives
# help pretty
# prettyStrings(true|false); pretty print in mode: emphasize
# strings quoting ...
# str -> pretty(str); pretty print the ast arg
#
# help analyze
# ast := analyze(File|expression_string | om); File: the input
# stream, om: the standard input,
# expression_string: the expr to parse
# analyze is a parser ...

def flex_alloc
# use undefined objects:
> a := om; b := om;
> af := analyze("a+b");
> af;
> af;
a + b;
> ugly(af); (CALL :
(+
|  (T_id : a)
|  (T_id : b))
OM;

> $ keep the expr part:
> af := af(1);
> af;
> af;
a + b;
> $ since a and b have no values
> $ the expr evaluates to an error:
> eval(af);
  Error - Bad arguments in:
  OM + OM;
> a := 10; b := 2;
> eval(af);
12;
> a := "the horse ";
> b := "drinks vodka ";
> eval(af);
"the horse drinks vodka ";
> u := om;
> bf := analyze("u := a+b");
> bf;
> u := a + b;
> ugly(bf)( := :
(+
|  (T_id : u)
|  (T_id : a)
|  (T_id : b))
OM;
> zz := eval(bf);
type(zz);
  "Code"
> u;
  "the horse drinks vodka ";
> $ to navigate within a tree one may use findAst
> bf1 := findAst("x := (a+b)*(c+d)"");
> v := findAst("x",bf1);
> v;
  [c + d, a + b];
> type(v);
  "Tuple"
> #v;
2;
> ugly(v(1))( + :
    (T_id : c)
    (T_id : d))
OM;
> ugly(v(2))( + :
    (T_id : a)
    (T_id : b))
OM;
> v(1)(1)(1);
  "c"
> v(1)(1);
> which_ast(v(1));
  +
> which_ast(v(1));
280;

> $ if the AST is chained (i.e. each node is linked
> $ to its father-node), it is possible to move within
> $ the tree upwards:
> v(0);
  (a + b) * (c + d);
> v(0)(1);
a + b;
> v(0)(1) = bf(1);
false;

> $ if a func is compiled, its internal documentation
> $ is computed and then
> $ discarded. It is possible to keep the variable
> $ reference info and look at it.
> $ It has a natural presentation as a set/map structure

> $ let us experiment with a realistic program
> $ to explore graphs
> $ start collecting references
> refcollect(true);
OM;
> explore := func(g,s opt avoid);
>> local toUse,reach,access;
>> reach := [s];
>> toUse := [s];
>> while #toUse > 0 do
>>  v := arb (toUse);
```
if avoid = om then
  access := \{u(2): u in g(v)} - reach;
else
  access := \{u(2): u in g(v) | u(1) notin avoid\} - reach;
end;

if #access /= 0 then
  if #access /= 0 then
    w := arb (access);
    toUse := toUse with w;
    end;
  end;
else
  toUse := toUse less v;
end;

if #access /= 0 then
  ref_explore := ref(explore);
end;

w := arb (access);
reach := reach with w;

$ stop collecting references
refcollect(false);

The ast_kind T_Pat (internal code 100) is for an ast playing the role of a joker, matching any other ast. The pretty-print of a T_Pat ast node is $@@@

$$\text{parse_msg(bool); }$ false->no parsing msg, true->msg$$

$$\text{varsOf(af); }$$ produces a tuple of all the referenceable ids appearing in af

$$\text{varsIn(af); }$$ produces a tuple of all the variables appearing in af excluding Selectors

$$\text{match(ast1,ast2); }$$true if ast1, ast2 are equal or if ast2 contains patterns matching ast1 subtrees or if ast1 contains patterns matching ast2 subtrees which will then be subsituted into ast1

$$\text{varsOf(af);}$$

$$\text{varsIn(af);}$$

$$\text{match(ast1,ast2);}$$

$$\text{parse_msg(bool);}$$

$$\text{varsOf(af);}$$

$$\text{varsIn(af);}$$

$$\text{match(ast1,ast2);}$$

$$\text{parse_msg(bool);}$$
unify [unified_S, unif_map] := unify(S opt constants);
- S is a collection -set or tuple- of ASTs
- constants is a set or tuple (even a single string is OK) of strings, or AST T_Id's,
  representing constant symbol identifiers

specifying constants, prevents the symbols in that collection from being
substituted by variables (forces the unification, if possible, in the other
direction)

unified_S is the unified collection (should be a singleton, if successful); unification
is impossible if #unified_S > 1

unif_map is the corresponding unification map, which may be applied using the
ast_subs primitive

6.6 Exercises

- write a func to list all the occurrences of a given identifier, within a given ast
- write a func to list all the occurrences of a given ast as subtree of another given ast
- write a func to list all the occurrences of a matching subtree of a given ast as subtree of
  another given ast (to subtrees are 'matchable' if they are unifiable)
- write a func to tranform a simple for-loop (with a single iterator) into a while loop
- a conjunction is a formula of type f1 and f2 and f3 and ... fn; the conjuncts are f1,f2, .. fn.
  Write a func to transform the ast of any conjunction into the set of its conjuncts.
- Write a func boundsOf to produce the set of all bound variables occurring in a quantifer ast, e.g.
  boundsOf(exists x in s, y in K(x) | F(y,0) = x+t ) = {x,y}
- Write a func to produce the set of all bound variables occurring in (set, tuple) -formers
- Write a func to produce the set of all bound variables occurring in for-loops
- Write a func to produce the set of all local and value declared variables occurring in funcs
- Similar exercises, but instead with free or non-local variables occurring in the given scope

7 Grammar

7.1 Terminology

Here are some preliminary observations concerning our BNF presentation of Cantor
grammar.

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.

Kepler Cantor: a User's Guide 44
2. Definitions appear as:

\[
\begin{align*}
\text{STMT} & \rightarrow \text{LHS} := \text{EXPR} ; \\
\text{STMT} & \rightarrow \text{if} \ \text{EXPR} \ \text{then} \ \text{STMTS} \ \text{ELSE-IFS} \ \text{ELSE-PART} \ \text{end}
\end{align*}
\]
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 in smaller case or in italic.

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, IDs, INTEGER constants, and FLOATING_POINT constants.

5. Cantor treats ends of line and tab as spaces. Any input can be spread across lines without changing the meaning, and Cantor 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.

7.2. Interactive Input

\[
\begin{align*}
\text{INPUT} & \rightarrow \text{PROGRAM} \\
\text{INPUT} & \rightarrow \text{STMT} \\
\text{INPUT} & \rightarrow \text{EXPR} ;
\end{align*}
\]

The EXPR is evaluated and the value is printed.

7.3 Program

Programs are usually read (i.e. included or read at launch-time) from a file, only because they tend to be long.

\[
\text{PROGRAM} \rightarrow \text{program ID} ; \ \text{STMTS} \ \text{end} ;
\]

Of course, it can appear on several lines. One may optionally close with \textit{end program}.

7.4 Statements

\[
\begin{align*}
\text{STMTS} & \rightarrow \text{STMT+} \\
& \text{NB-> One or more instances of STMT. The final semicolon is optional.}
\end{align*}
\]

\textbf{assignment statement}
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:

\[ a := 4; \] \quad \text{a is changed to contain the value 4.}
\[ [a,b] := [1,2]; \] \quad \text{a is assigned 1 and b is assigned 2.}
\[ [x,y] := [y,x]; \] \quad \text{Swap x and y.}
\[ f(3) := 7; \] \quad \text{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, -although not a predefined func- then f(3) will be 7, and all other values of f will be as they were before the assignment.}

call for expression evaluation

STMT --> EXPR ;

if statement

The expression is evaluated and the value ignored. This is usually used to invoke procedures or to display the current value of a variable.

STMT --> if EXPR then STMTS ELSE-IFS ELSE-PART end ;

The EXPRs 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.

ELSE-IFS --> ELSE-IF *
NB-> Zero or more instances of ELSE-IF.

ELSE-IF --> elseif EXPR then STMTS

ELSE-PART --> else STMTS
NB-> May be omitted.

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

for statement

STMT --> for ITERATOR do STMTS end ;

The STMTS are executed for each instance generated by the iterator. One may optionally close with end for.
while statement

\[
\text{STMT} \rightarrow \text{while EXPR do STMTS end ;}
\]

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 \textit{end while}.

read statement

\[
\text{STMT} \rightarrow \text{read LHS-LIST ;}
\]

Cantor gives a question mark (?) (Cantor is then in read mode) 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 Cantor will look at the next input in that file for the expression(s) to be read.

\[
\text{STMT} \rightarrow \text{read LHS-LIST from EXPR ;}
\]

This is the same as read LHS-LIST; except that EXPR must have a value of type file, i.e. designate an external file or a pane file (a file associated to a text window). The values to be read are then taken from the external file or the pane stream 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 \textit{eof} will return true when given the file as parameter. Before this statement is executed, the external or pane file in question must have been opened for reading by the proper pre-defined function (see section 3.6).

\[
\text{STMT} \rightarrow \text{readf PAIR-LIST ;}
\]

\[
\text{STMT} \rightarrow \text{readf PAIR-LIST from EXPR ;}
\]

\[
> \text{readf } x ;
1.34
> x ;
1.34000e+00 ;
\
> \text{readf } y ;
123,456
> y ;
"123,456" ;
\]

Figure 1: readf example

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 formatting used. See PAIR-LIST at the end of this section.

print statement

\[
\text{STMT} \rightarrow \text{print EXPR-LIST ;}
\]
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.

\[
\text{STMT} \rightarrow \text{print EXPR-LIST to EXPR ;}
\]

As in read...from..., EXPR must be a value of type file. The values are written to the external or pane 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 (e.g. openw or opena for external text files. See section 3.6).

\[
\begin{align*}
\text{STMT} & \rightarrow \text{printf PAIR-LIST;} \\
\text{STMT} & \rightarrow \text{printf PAIR-LIST to EXPR ;}
\end{align*}
\]

> 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

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.

\[
\text{return statement}
\]

\[
\text{STMT} \rightarrow \text{return ;}
\]

return is only meaningful inside a func. Its effect is to terminate execution of the func and return OM to the caller. Cantor 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.

\[
\text{STMT} \rightarrow \text{return EXPR ;}
\]

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

\[
\text{take .. from statement}
\]

\[
\text{STMT} \rightarrow \text{take LHS from LHS ;}
\]

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.

\[
\text{STMT} \rightarrow \text{take LHS frome LHS ;}
\]

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 from 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 ith component is the old value of the (i+1)st component (i=1,2,...).

write statement

STMT  -->  write PAIR-LIST;
STMT  -->  write PAIR-LIST to EXPR;
STMT  -->  writeln PAIR-LIST;
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.

formats

PAIR-LIST  -->  PAIR+
        NB-> 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

```plaintext
> 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

* 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 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.

\[
\begin{array}{ll}
\text{format} & \rightarrow \text{INT} \\
\text{format} & \rightarrow \text{INT.FRACT}
\end{array}
\]

INT is an Integer (and the integer part of floating_point numbers). INT represents the minimal number of columns to be used. FRACT, the fractional part of a floating_point number is used to specify precision, in terms of hundredths:

\[
\text{precision} = \text{0.FRACT} \times 100
\]

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 figures 3 and 4.

Default values are:

<table>
<thead>
<tr>
<th>type</th>
<th>int part</th>
<th>fract.part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Integer</td>
<td>0</td>
<td>50 (for breaking large ints)</td>
</tr>
<tr>
<td>String</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Anything else</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

in the example of figure 4 the printf statement reads:

```
printf x:3*[10, ' --- ' ] with ' \n-> ', ' \nfin';
```

Two items are printed: x and the string '\nfin'. x has its output format specified by a tuple. '\nfin' uses the default format. x's format tuple is 3*[10, \n->] with \n->

a tuple of seven (7) elements. In this tuple only 3 elements are numbers, i.e. the format specification for 3 elements of x. Since in this example x has 17 elements, the format specification is cycled over 6 times.

> x;
["there", "are", 5, "output", "formats", "in", "version", 0.410, ":", "integers", "floating", "point", "numbers", "strings", 
or", "tupleof", "output"];
> printf x;
there are 5 output formats in version 0.41000: integers floating point numbers strings or tuple of output
> printf x:3*[10, \n-> ] with '\n-> ', '\nfin';
there are 5 ---
-\n output --- formats --- in ---
-\n version --- 0.41000 ---
-\n integers --- floating --- point ---
> printf x:3*[10.2,' -- ' with \n-->', '\nfin';
    there --- are --- 5 ---
> output --- formats --- in ---
> version --- 0.41000000000000000000 --- : ---
> integers --- floating --- point ---
> numbers --- strings --- or ---
> tupleof --- output
>
Figure 4: printf with structure example

7.5 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 or tuple 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 IDs that were used as local variables are restored. That is, these IDs are bound variables whose scope is the construction containing the iterator. (e.g., for statements, quantifiers, formers, etc.)

\[
\text{ITERATOR} \rightarrow \text{ITER-LIST}
\]
\[
\text{ITERATOR} \rightarrow \text{ITER-LIST} | \text{EXPR}
\]

EXPR must evaluate to a Boolean. Generates only those instances generated by ITER-LIST for which the value of EXPR is true.

\[
\text{ITER-LIST} \rightarrow \text{SIMPLE-ITERATOR} +
\]
\[
\text{ITER-LIST} \rightarrow \text{SIMPLE-ITERATOR} + | \text{EXPR}
\]

One or more SIMPLE-ITERATORS separated by commas.

Generates all possible instances for every combination of the SIMPLE-ITERATORS. The first SIMPLE-ITERATOR advances most slowly. Subsequent iterators may depend on previously bound values.

\[
\text{SIMPLE-ITERATOR} \rightarrow \text{BOUND-LIST in EXPR}
\]

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.

\[
\text{SIMPLE-ITERATOR} \rightarrow \text{BOUND} = \text{ID} (\text{BOUND-LIST})
\]

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.

\[
\text{SIMPLE-ITERATOR} \rightarrow \text{BOUND} = \text{ID} (\text{BOUND-LIST})
\]

Same as the previous one for the case in which ID is an mmap.
BOUND-LIST --> BOUND+  
once 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.

7.6 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. Generates all integers 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.

FORMER --> EXPR , EXPR .. EXPR  
All three occurrences of EXPR must evaluate to integer. Generates all integers 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 integers that are not greater than the third EXPR. If the difference is negative, it generates those integers that are not less than the third EXPR. If the difference is zero, no integers are generated.

FORMER --> EXPR : ITERATOR  
The value of EXPR for each instance generated by the ITERATOR.

7.7 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 SELECTORs 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.

\[
\text{SELECTOR} \rightarrow \{ \text{EXPR-LIST} \}
\]

Must be used with an mmap. The case in which the list has more than one element is handled as above.

\[
\text{SELECTOR} \rightarrow (\text{EXPR .. EXPR})
\]

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 \((a .. b)\).

- \(a <= b\) 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.

\[
\text{SELECTOR} \rightarrow (.. \text{EXPR})
\]

Means the same as \((1 .. \text{EXPR})\).

\[
\text{SELECTOR} \rightarrow (\text{EXPR .. })
\]

Means the same as \((\text{EXPR .. EXPR})\) where the second EXPR is equal to the length of the tuple or string.

\[
\text{SELECTOR} \rightarrow ()
\]

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

7.8 Left Hand Sides

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

\[
\text{LHS} \rightarrow \text{ID}
\]

\[
\text{LHS} \rightarrow \text{LHS SELECTOR}
\]

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

\[
\text{LHS} \rightarrow [\text{LHS-LIST}]
\]

\[
\text{LHS-LIST} \rightarrow \text{LHS+}
\]

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.

### 7.9 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 --> USER-FUNC**
  
  A user-defined func. See §5.

- **EXPR --> ifEXPR then EXPR ELSE-IFS ELSE-PART end;**
  
  See definition of if under STMT, page15. ELSE-PART is required, and 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 Cantor value that is, in the general sense, a function. That is, it must be a map, tuple, string, or func. See §4,5.

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

EXPR · (EXPR) EXPR 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 7.11 for precedence rules.

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 *.

\[ \text{EXPR} \rightarrow \text{EXPR} \div \text{EXPR} \]

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

\[ \text{EXPR} \rightarrow \text{EXPR} \mod \text{EXPR} \]

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,

\[
\begin{align*}
(a \div b)? b+ (a \bmod b) &= 0 \quad \text{for } b > 0 \\
 a \div (? b) &= ? (a \div b) \quad \text{for } b < 0.
\end{align*}
\]

\[ \text{EXPR} \rightarrow \text{EXPR} ** \text{EXPR} \]

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

\[ \text{EXPR} \rightarrow \text{EXPR} \text{ with } \text{EXPR} \]

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.

\[ \text{EXPR} \rightarrow \text{EXPR} \text{ less } \text{EXPR} \]

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.

Pointer expressions may be defined as follows:

\[ \text{EXPR} \rightarrow \text{EXPR} \rightarrow \text{EXPR} \]

the expression on the left of the -> (pointer sign) designates a scope. The expression on the right of the pointer sign is an expression which must be evaluated in that scope.

\[ \text{EXPR} \rightarrow \text{EXPR} = \text{EXPR} \]

The test for equality of any two Cantor values.

\[ \text{EXPR} \rightarrow \text{EXPR} /= \text{EXPR} \]

Negation of EXPR=EXPR.

\[ \text{EXPR} \rightarrow \text{EXPR} < \text{EXPR} \]

\[ \text{EXPR} \rightarrow \text{EXPR} > \text{EXPR} \]

\[ \text{EXPR} \rightarrow \text{EXPR} <= \text{EXPR} \]

\[ \text{EXPR} \rightarrow \text{EXPR} >= \text{EXPR} \]

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.

\[
\text{EXPR} \rightarrow \text{EXPR in EXPR}
\]

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.

\[
\text{EXPR} \rightarrow \text{EXPR notin EXPR}
\]

Negation of EXPR in EXPR.

\[
\text{EXPR} \rightarrow \text{EXPR subset EXPR}
\]

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.

\[
\text{EXPR} \rightarrow \text{EXPR and EXPR}
\]

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.

\[
\text{EXPR} \rightarrow \text{EXPR or EXPR}
\]

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.

\[
\text{EXPR} \rightarrow \text{EXPR impl EXPR}
\]

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

\[
\text{EXPR} \rightarrow \text{EXPR iff EXPR}
\]

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.

\[
\text{EXPR} \rightarrow \% \text{ BINOP EXPR}
\]

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, %? EXPR equals

\[
x1 \ ? \ x2 \ ? \ ???? \ xN
\]

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 --> EXPR % BINOP EXPR

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

   a ? x1 ? x2 ? ??? ? xN

associating to the left.

EXPR --> EXPR ? EXPR

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

EXPR --> 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 --> 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 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.

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.

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

DEFNS --> DEFN*
Zero or more instances of DEFN.

DEFN --> BOUND := EXPR ;
DEFN --> ID SELECTOR := EXPR ;

EXPR-LIST --> EXPR+
One or more instances of EXPR separated by commas.

7.10 User defined functions

USER-FUNC --> 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 in section 6.2.3.

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.

OPT-PART --> opt ID-LIST
May be omitted.

LOCALS --> local ID-LIST;

VALUES --> value ID-LIST;

ID-LIST --> ID+
One or more instances of ID separated by commas.

7.11 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 using parentheses to separate the scope of each.

- left associative
CALL anything that is a call
to a function
--- func, tuple, string, map, etc.

# - + unary operators

? nonassociative
%
** right associative
*/ mod div
+- with less union inter

.ID infix use of binary function
in notin subset
< <= /= >= nonassociative
not unary
and

or_impl
iff
exists forall
where

8 Directives

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

The other directives are ! commands. [ a | b ] indicates a choice between a and b. Most directives are available as interactive, menu oriented commands.

8.1 Cantor Commands
!quit --- Exit Cantor.

!reset --- Reset Cantor: it is useful for a fresh restart, without leaving Cantor; all memory resident Cantor objects are destroyed. The memory allocated until then is kept for the new Cantor session.

!suspend --- it is a menu oriented directive, not a line-oriented directive: it is used as a menu option to suspend the Cantor execution. It is very useful when one suspects a program to be trapped into an endless loop. The suspend directive is invoked by selecting with the mouse or cursor the suspend option in the menu. The cantor program is then suspended and Cantor enters a nested session in the break mode (prompt %). To resume program execution --perhaps after inspecting or changing some variables or funcs-- one executes the return; instruction.

!include <filename> --- Replace <filename> with a file/pathname according to the rules of your operating system. Cantor will insert your file. The same service is available with the Cantor primitive include(filename);

!load <filename> --- Begins loading a Cantor component. A similar service is available with the Cantor primitive restore(filename);

!save <filename> --- save the current status of the Cantor session as a component. It is mainly used for creating compiled components: one performs a !include of some source code, and one saves the result. It is recommended to make a !reset between any two compilations, otherwise the compilation results accumulate.A similar service is available with the Cantor primitives store(expr filename); and save(id_string filename);

!clear --- Throw away all input back to the last single prompt. 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 the user prefers to start again, !clear will clear the typing buffer and allow you to start the input afresh.

!memory --- Shows how much memory has been allocated --and is subject to garbage collection.

!memory <nnn> --- Change the legal upper bound to <nnn>. May not be lower than the currently allocated memory: if <nnn>is lower than what is currently allocated, returns how much memory has been allocated so far.

!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.

!watch list-of-ids --- Traces assignment and evaluation of ids. Any watch-ed id, when accessed, is considered a breakpoint, if the !breakpoint on switch has been set, and until it has been reset (/breakpoint off)

!unwatch list-of-ids --- Turns off tracing for ids, and possible breakpoints.

!record <filename> --- Begins recording (i.e. echoing) all input to <filename>. This lets you experiment and keep a record of the work performed.

!record --- the same directive, without argument, is used to turn-off recording
!recordOutput <filename> --- Begins recording (i.e. echoing) all that is displayed on-i.e. output to-the console to <filename>. This lets you keep a complete record of the work performed.

!recordOutput --- the same directive, without argument, is used to turn-off recording

!ids --- Lists all identifiers that have been defined. See also the Cantor primitive ids()

!oms --- Lists all identifiers that have been used, but not defined.

!work_ids --- upon this command, all user created variables, visible in the current session are listed, if they are defined (i.e. \( \neq \) OM)

!new_ids --- upon this command, all user created variables, visible in the current session are listed, whether they are defined (i.e. \( \neq \) OM) or not.

!version --- upon this commands the version information and copyright notice is displayed at the console.

!flex_min_size <nnn> --- When flex_alloc is on, the Cantor system optimizes the size of the contiguous memory blocks available for new dynamic memory requests. The default flex_min_size is 64 bytes.

obsolete: 8086, oldRreal

8.2 Cantor switches

!breakpoint [ on / off ] --- When on all access to a watch-ed id becomes a breakpoint. When arriving at a breakpoint, Cantor suspends the current execution and prompts with % (break mode). Most Cantor services are available then. However: the currently watch-ed id may not be unwatch-ed. After a break, to resume program execution --perhaps after inspecting or changing some variables or funcs-- one executes the return; instruction.

!changes [ on / off ] --- When on, all the changes to any Cantor variable, in any scope are recorded. Recording stops when the switch is reset. The changes keep accumulating in the same structure when the switch is set again, unless the resetchanges(); instruction is executed. (see the primitives allchanges, visiblechanges, resetchanges)

!code [ on / off ] --- When on, you get a pseudo-assembly listing for the program. Default is off.

!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.

!gc [ on / off ] --- When on, upon each garbage collection, statistics are displayed. It is useful to parametrize correctly the initial allocation. Note the difference with the Cantor primitive gc() which invokes the garbage collector.

!trace [ on / off ] --- 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 Cantor itself. Default is off.

!verbose [ on / off ] --- Controls the amount of trace information provided by runtime error messages. See section 11. Default is off.

!flex_alloc [ on/off ] --- When flex_alloc is on, the memory upper-limit, defined by the
Memory command may be bypassed in increments of `flex_min_size`. See the `flex_min_size` command.

`!passive_err [on/off]` --- When `passive_err` is on, Warnings issued by the Cantor system no longer trigger an interactive process. Instead the message is issued to the console. However, since a breakpoint is generated, the system is suspended, in a nested error session. To avoid such suspension, insert the instruction:

```
ignoreallbreakpt();$ all breakpoint commands set to noop
```
before running the program sequence containing the Warning.

`!annotate [on/off]` --- When `annotate` is on, comments are added as annotations to Abstract Syntax Trees. This is used when comments need to be processed.

`!time [on/off]` --- When `time` is on, along with the execution trace the time (date) of accesses is displayed. This illustrated by the following example, run under the option `verbose` on, and `time` on.

```plaintext
> fact := func(n);
  » if n <= 1 then return 1;
  » else fact(n) := n*fact(n-1);
  » return fact(n);
> end;
> end;
> watch fact
!fact' watched
> fact(10);
| fact(2) := 2;
| fact(3) := 6;
| fact(4) := 24;
| fact(5) := 120;
```

Figure 5: Tracing and timing

```
 obsolete: in_debug
```

---

Figure 5: Tracing and timing
### 8.3 `!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 Cantor 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, first, determine the amount of memory available (e.g. by checking 'About Finder'). Then subtract from that 250K for Cantor's scratch area plus any other space you may wish to save for use by the `!save` and `!load` directives or `save()`, `store()` and `restore()` instructions. A thumb rule is that a restore() or a `!load` require a memory allocation of 1.4 times the file size. This memory is not freed, it is used for the new objects restored. Its excess is just added to the garbage collectable memory. You can then set the memory limit and pre-allocate in your `.cantorrc` files.

See figure 6. 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 Cantor, and 150K was pre-allocated. The lines below "..." are in another session, because one cannot decrease the GC (garbage collected) memory.

```plaintext
> !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 6: Finding memory limits
```

If you have enough memory available, don't worry about memory allocation, just use the directive

`!flex_alloc on`

### 8.4 `!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` (resp `id1 id2 id3`) is the name(s) of the identifier(s) 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 7 for an example of the output.

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

!'f' watched
> f(1) := 1;
! f(1) := 1;

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

Figure 7: !watch examples

8.5 !record, !recordOutput

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. The !recordOutput directive is similar to !record, but concerns only the output (!record only the input).

9. The Cantor Grammar: condensed

9.1 Interactive Input

INPUT  -->  PROGRAM
INPUT  -->  STMT
INPUT  -->  EXPR;

9.2 Program
PROGRAM --> program ID ; STMTS end ;

9.3 Statements

STMT --> LHS := EXPR ;
STMT --> EXPR ;
STMT --> if EXPR then STMTS ELSE-IFS ELSE-PART end ;
ELSE-IFS --> 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 from 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 --> STMT+ 
The final semicolon is optional.
PAIR-LIST --> PAIR*
PAIR, separated by commas
PAIR --> EXPR : EXPR
PAIR --> EXPR

9.4 Iterators

ITERATOR --> ITER-LIST
ITERATOR --> ITER-LIST | EXPR
ITER-LIST --> SIMPLE-ITERATOR+
SIMPLE-ITERATOR --> BOUND-LIST in EXPR
SIMPLE-ITERATOR --> BOUND = ID ( BOUND-LIST )
SIMPLE-ITERATOR --> BOUND = ID { BOUND-LIST }
BOUND-LIST --> BOUND+
separated by commas

BOUND --> ~
BOUND --> ID
BOUND --> [ BOUND-LIST ]

9.5 Selectors

SELECTOR --> ( EXPR-LIST )
SELECTOR --> { EXPR-LIST }

SELECTOR --> ( EXPR .. EXPR )
SELECTOR --> ( .. EXPR )
SELECTOR --> ( EXPR .. )
SELECTOR --> ()

9.6 Left Hand Sides

LHS-LIST --> LHS+
    separated by commas

LHS --> ID
LHS --> LHS SELECTOR
LHS --> [ LHS-LIST ]

9.7 Expressions and Formers

EXPR-LIST --> EXPR+
    separated by commas
EXPR --> ID
EXPR --> INTEGER

EXPR --> FLOATING-POINT
EXPR --> STRING
EXPR --> true
EXPR --> false
EXPR --> OM

EXPR --> newat
EXPR --> USER-FUNC
EXPR --> if EXPR then EXPR ELSE-IFS ELSE-PART end ;
    the analyzer separates if-expressions from if-statements, by the context. In an if-
    expression each statement reduces to expr
EXPR --> ( EXPR )
EXPR --> [ FORMER ]
EXPR --> { FORMER }

FORMER --> ε
    empty
FORMER --> EXPR-LIST

FORMER --> EXPR .. EXPR
FORMER --> EXPR , EXPR .. EXPR
FORMER --> EXPR : ITERATOR

EXPR --> # EXPR
EXPR --> not EXPR

Kepler Cantor: a User’s Guide 66
EXPR --> + EXPR  
EXPR --> - EXPR  

EXPR --> EXPR SELECTOR  
EXPR --> EXPR . ID EXPR  
notice the period '.' preceding the 2-arg func identifier  
EXPR --> EXPR . (EXPR) EXPR  
notice the period '.' preceding the 2-arg lambda-expression  
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 --> exists ITER-LIST | EXPR  
EXPR --> forall ITER-LIST | EXPR  
EXPR --> EXPR where DEFNS end  

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 --> DEFN*  
DEFN --> BOUND := EXPR ;  
DEFN --> ID SELECTOR := EXPR ;  

9.8 User defined functions  
USER-FUNC --> FUNC-HEAD LOCALS VALUES STMTS end  
FUNC-HEAD --> func ( 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 --> ID+  
separated by commas  

10 Debugging  
This section covers both the general issue of debugging and that of run-time errors. The  
basic debugging technique, involves watching (under the verbose on switch for a more  
detailed reporting) variables and functions, and possibly setting a breakpoint. When a
breakpoint is reached, the system enters in a nested session. This is the case too when a severe error is detected and a notification/corrective action is expected from the user. Let us observe that almost all errors, except for syntax errors generate execution interruption.

A frequent problem is to find out where the error is detected. If it is detected within the execution of a non-anonymous func, it is often possible to know which one, by the instruction:

> tellfunc();

which returns the name a variable which was assigned recently that func. When an error is easily reproductible, a fine grain of breakpoints allows a relatively easy identification of its origin.

### 10.1 Runtime Errors

The runtime error messages describe most problems by printing the operation with the offending values of the arguments.

One possible problem is that some values are very big: \{1..10000\} for instance when not enough memory was set aside to accommodate for all the created data. Therefore, there are several forms of the error messages, controlled by the `!verbose` and `!passive_err` directives. By default both switches are off. When `passive_err` is off severe errors and warnings generate a special dialog, where the message is displayed into a warning window which disappears when clicking inside. When `verbose` is off 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 8 for an example.

```plaintext
> !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 8: Runtime errors

### 10.2 Fatal Errors

The following errors cause Cantor to exit. Generally they indicate that the problem is larger than Cantor can manage.

---

Allocated data memory exhausted

Use `!memory` to raise limit.

Includes too deeply nested Probably file includes itself.

### 10.3 Operator Related Messages

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

Bad arg to mcPrint   internal
Bad args in low,next..high   *
Bad args in low..high   *
Bad mmap in iterator   MMap iterator over non-map
Can't mmap string   Cannot perform selection
Can't mmap tuple   Cannot perform selection
Cannot edit except at top level   Edit not permitted within an include
Divide by zero   *
Input must be an expression   *
Iter_Next   internal

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
Slice lower bound too big   *
Slice upper bound too big   *
Stack Overflow   *
Stack Underflow   *
Too few arguments   *
Too many arguments   *
Top level return not allowed   *

10.5 Advanced trace and debugging facilities

the following different kinds of breakpoints are available:

"NO_BREAK_PT_bkPt",
"RET_CLOSURE_bkPt",
"PARAM_DEFN_bkPt",
"OPT_PARAM_DEFN_bkPt",
"LHS_ASSIGN_bkPt",
"LHS_ASSIGN_bkPt",...
"LOAD_ACCESS_bkPt",
"S_MAP_ACCESS_bkPt",
"M_MAP_ACCESS_bkPt",
"SLICE_DEFN_bkPt",

"WARNING_bkPt",
"RT_ERROR_bkPt",
"FATAL_bkPt",

Each of these breakpoints represent either a specific mode, a specific phase or both, in the execution process. The purpose of this section is to explain the versatility of breakpoint and tracing facilities in Cantor.

inserting code instead of stopping at a breakpoint

It is possible to replace an execution suspension at a breakpoint by the execution of a predefined collection of statements. Indeed, Cantor checks for each breakpoint kind, the value of the variable having as identifier the kind name, e.g. checks the variable RET_CLOSURE_bkPt upon the return of a func, or the variable S_MAP_ACCESS_bkPt just before computing the value of an expression f(i), etc. Let us call these variables "break variables" or bk_var. Let us assume a breakpoint has been set by combining a !watch request a the breakpoint on option. Furthermore, let us assume the program execution is entering the break-ing section:

- if the appropriate bk_var is undefined (has value OM) or is a null string (""), program execution is suspended in a nested Cantor session, waiting until the user exits from this session by entering at the console a "return;" statement
- if the appropriate bk_var is a string, the string text is considered as executable instructions and will be executed unless it is the string 'noop'. The text of bk_var is what we call a breakpoint command.

Actually, the Cantor system executes interp(bk_var) unless bk_var = 'noop'.

setting, re-setting breakpoints according to their kind

The breakpoints are all potentially activated when a variable is declared watched, and the breakpoint option is on. However, none of them is actually active. Making them active consists in setting the proper values for all the bk_vars. This is entirely programmable, by means of ordinary assignments to these 11 variables. The following primitives are helpful:

setallbreakpt();  $ all breakpoint commands set to OM
ignoreallbreakpt();  $ all breakpoint commands set to noop
allbreaktype();  $ all breakpoint types- this command is a help-command
breaktype();  $ current breakpoint type

Here is the text of an include file example to illustrate these possibilities:

!help break

ignoreallbreakpt();
RET_CLOSURE_bkPt;
S_MAP_ACCESS_bkPt;
S_MAP_ACCESS_bkPt := 'entry_time := clock();';
RET_CLOSURE_bkPt := 'delta := clock() - entry_time; printf delta*16;';$
16ms = 1 Tick mesure par clock
f := func(x); return x+10; end;
!watch f

!breakpoint on
Here is now the recording of two distinct sessions, one with `verbose off` then with `verbose on`. Note the use of the `clock()` primitive, which according to the on-line help:

```
tickCount := clock();$ approx. 1 Tick every 16 msec
```

returns a `tickCount`, incremented 60 times a second. Whence, a duration in seconds may be computed as `(clock_end - clock_begin)/60`

```c
f(10);
entry_time;
delta;
f(15);
```

```c
! Recording Output on bkPt_session_out

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;

> !help break
breakProcess() $ break current process ...
setallbreakpt();$ all breakpoint commands set to OM ...
ignoreallbreakpt();$ all breakpoint commands set to noop ...
allbreaktype();$ all breakpoint types ...
breaktype();$ current breakpoint type ...
break();$ force premature (loop) exit ...
OM;

> ignoreallbreakpt();
OM;
```

```c
> entry_time;
1531779;

> delta;
10;

> entry_time;
1532863;

> delta;
24;

> entry_time;
1533263;

> delta;
24;

> entry_time;
1533263;

> delta;
24;
```

```c
> entry_time;
1531779;

> delta;
10;

> entry_time;
1532863;
```

```c
Verbose on
```

```c
#include bkPt_session completed
```

Kepler

Cantor: a User's Guide 71
appendix: Predefined Functions

In this appendix find:
- the Cantor primitives list
- the Cantor primitives index

---

the Cantor primitives
and directives

release 26 octobre-94 Cantor 46.19

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic math</td>
<td>1</td>
</tr>
<tr>
<td>trigonometry</td>
<td>1</td>
</tr>
<tr>
<td>basic set primitives</td>
<td>1</td>
</tr>
<tr>
<td>type testing</td>
<td>1</td>
</tr>
<tr>
<td>source &amp; binary</td>
<td>2</td>
</tr>
<tr>
<td>misc</td>
<td>3</td>
</tr>
<tr>
<td>text-number conversion</td>
<td>3</td>
</tr>
<tr>
<td>basic file processing</td>
<td>4</td>
</tr>
<tr>
<td>advanced file primitives</td>
<td>4</td>
</tr>
<tr>
<td>scope control</td>
<td>4</td>
</tr>
<tr>
<td>drawing</td>
<td>4</td>
</tr>
<tr>
<td>windows</td>
<td>5</td>
</tr>
<tr>
<td>events</td>
<td>6</td>
</tr>
<tr>
<td>menus</td>
<td>6</td>
</tr>
<tr>
<td>regions</td>
<td>6</td>
</tr>
<tr>
<td>buttons</td>
<td>6</td>
</tr>
<tr>
<td>text pane</td>
<td>7</td>
</tr>
<tr>
<td>pane files</td>
<td>7</td>
</tr>
<tr>
<td>logo turtle</td>
<td>7</td>
</tr>
<tr>
<td>abstract syntax</td>
<td>8</td>
</tr>
<tr>
<td>interoperability types</td>
<td>9</td>
</tr>
<tr>
<td>interoperability functions</td>
<td>10</td>
</tr>
<tr>
<td>macintosh specific misc.</td>
<td>10</td>
</tr>
<tr>
<td>macintosh vector graphics</td>
<td>11</td>
</tr>
<tr>
<td>grids and cells</td>
<td>13</td>
</tr>
<tr>
<td>Cantor Global var</td>
<td>14</td>
</tr>
<tr>
<td>Cantor directives</td>
<td>14</td>
</tr>
<tr>
<td>experimental Cantor directives</td>
<td>15</td>
</tr>
<tr>
<td>pane &amp; file experiments</td>
<td>15</td>
</tr>
<tr>
<td>Cantor primitives index</td>
<td>i</td>
</tr>
<tr>
<td>basic math</td>
<td>i</td>
</tr>
</tbody>
</table>

Sorted Topic List
Sorted Item List

Kepler  the Cantor primitives  72
<table>
<thead>
<tr>
<th>primitive name</th>
<th>arg nbr</th>
<th>invocation, comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>basic math</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>1</td>
<td>$r := \exp(x)$; $\text{power of } e$ (inverse: see ln)</td>
</tr>
<tr>
<td>ln</td>
<td>1</td>
<td>$r := \ln(x)$; $\text{neper. log}$ (inverse: see exp)</td>
</tr>
<tr>
<td>log</td>
<td>1</td>
<td>$r := \log(x)$; $\text{base 10 log}$</td>
</tr>
<tr>
<td>max</td>
<td>2</td>
<td>$x := \max(a,b)$;</td>
</tr>
<tr>
<td>min</td>
<td>2</td>
<td>$x := \min(a,b)$;</td>
</tr>
<tr>
<td>abs</td>
<td>1</td>
<td>$y := \abs(x)$; $\text{absolute value}$</td>
</tr>
<tr>
<td>ceil</td>
<td>1</td>
<td>$n := \ceil\left(\text{real}\right)$; $\text{int. approx. of a real, see also floor, fix}$</td>
</tr>
<tr>
<td>fix</td>
<td>1</td>
<td>$n := \fix\left(\text{real}\right)$; $\text{int. approx. of a real, see also ceil, floor}$</td>
</tr>
<tr>
<td>floor</td>
<td>1</td>
<td>$n := \floor\left(\text{real}\right)$; $\text{int. approx. of a real, see also ceil, floor}$</td>
</tr>
<tr>
<td>even</td>
<td>1</td>
<td>$\text{bool} := \text{even}(n)$;</td>
</tr>
<tr>
<td>odd</td>
<td>1</td>
<td>$\text{bool} := \text{odd}(n)$;</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>$n := \random\left(\text{root}\right)$; $\text{type of root is the returned type}$</td>
</tr>
<tr>
<td>randomize</td>
<td>1</td>
<td>$x := \randomize\left(\text{seed}\right)$; $\text{set new seed for random gen.}$</td>
</tr>
<tr>
<td>round</td>
<td>1</td>
<td>$n := \round(x)$;</td>
</tr>
<tr>
<td>sgn</td>
<td>1</td>
<td>$n := \sgn\left(\text{int}\right)$; $\text{sign}$</td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
<td>$r := \sqrt(x)$; $\text{square root}$</td>
</tr>
<tr>
<td><strong>trigonometry</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cos</td>
<td>1</td>
<td>$r := \cos(x)$;</td>
</tr>
<tr>
<td>sin</td>
<td>1</td>
<td>$r := \sin(x)$;</td>
</tr>
<tr>
<td>tan</td>
<td>1</td>
<td>$r := \tan(x)$;</td>
</tr>
<tr>
<td>acos</td>
<td>1</td>
<td>$r := \acos(x)$;</td>
</tr>
<tr>
<td>asin</td>
<td>1</td>
<td>$r := \asin(x)$;</td>
</tr>
<tr>
<td>atan</td>
<td>1</td>
<td>$r := \atan(x)$;</td>
</tr>
<tr>
<td>cosh</td>
<td>1</td>
<td>$r := \cosh(x)$;</td>
</tr>
<tr>
<td>sinh</td>
<td>1</td>
<td>$r := \sinh(x)$;</td>
</tr>
<tr>
<td>tanh</td>
<td>1</td>
<td>$r := \tanh(x)$;</td>
</tr>
<tr>
<td>acosh</td>
<td>1</td>
<td>$r := \acosh(x)$;</td>
</tr>
<tr>
<td>asinh</td>
<td>1</td>
<td>$r := \sinh(x)$;</td>
</tr>
<tr>
<td>atanh</td>
<td>1</td>
<td>$r := \atanh(x)$;</td>
</tr>
<tr>
<td><strong>basic set primitives</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arb</td>
<td>1</td>
<td>$x := \arb\left(\text{set}\right)$; $\text{choice function}$</td>
</tr>
<tr>
<td>image</td>
<td>1</td>
<td>$\text{set} := \image\left(\text{map}\right)$; $\text{same as range}$</td>
</tr>
<tr>
<td>npow</td>
<td>2</td>
<td>$\text{set}_\text{collection} := \npow\left(\text{set, nmax}\right)$; $\text{the subsets of atmost nmax els}$</td>
</tr>
<tr>
<td>pow</td>
<td>1</td>
<td>$\text{power}_\text{set} := \pow\left(\text{set}\right)$;</td>
</tr>
<tr>
<td>range</td>
<td>1</td>
<td>$\text{set} := \range\left(\text{map}\right)$; $\text{same as image}$</td>
</tr>
<tr>
<td><strong>type testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_atom</td>
<td>1</td>
<td>$\text{bool} := \is_{\text{atom}}\left(\text{x}\right)$;</td>
</tr>
<tr>
<td>is_ast</td>
<td>1</td>
<td>$\text{bool} := \is\left(\text{ast}\right)$;</td>
</tr>
<tr>
<td>is_boolean</td>
<td>1</td>
<td>$\text{bool} := \is_{\text{boolean}}\left(\text{x}\right)$;</td>
</tr>
</tbody>
</table>

**Kepler**
a prototyping company

**Cantor**
a prototyping language
## Cantor

**primitives list**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_bignum(x)</td>
<td>bool := is_bignum(x);</td>
</tr>
<tr>
<td>is_defined(x)</td>
<td>bool := is_defined(x);</td>
</tr>
<tr>
<td>is_integer(x)</td>
<td>bool := is_integer(x);</td>
</tr>
<tr>
<td>is_file(x)</td>
<td>bool := is_file(x);</td>
</tr>
<tr>
<td>is_floating(x)</td>
<td>bool := is_floating(x);</td>
</tr>
<tr>
<td>is_func(x)</td>
<td>bool := is_func(x);</td>
</tr>
<tr>
<td>is_map(x)</td>
<td>bool := is_map(x);</td>
</tr>
<tr>
<td>is_func(x)</td>
<td>bool := is_func(x);</td>
</tr>
<tr>
<td>is_number(x)</td>
<td>bool := is_number(x);</td>
</tr>
<tr>
<td>is_om(x)</td>
<td>bool := is_om(x);</td>
</tr>
<tr>
<td>is_set(x)</td>
<td>bool := is_set(x);</td>
</tr>
<tr>
<td>is_string(x)</td>
<td>bool := is_string(x);</td>
</tr>
<tr>
<td>is_table(x)</td>
<td>bool := is_table(x); $ a map with domain and range elements of simple type</td>
</tr>
<tr>
<td>is_textpane(x)</td>
<td>bool := is_textpane(x);</td>
</tr>
<tr>
<td>is_tuple(x)</td>
<td>bool := is_tuple(x);</td>
</tr>
<tr>
<td>type(x)</td>
<td>str := type(x);</td>
</tr>
</tbody>
</table>

### source & binary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp(s)</td>
<td>$ interpret string s as a Cantor instruction or directive</td>
</tr>
<tr>
<td>include(filename)</td>
<td>$ program command for source inclusion (see also the directive !include)</td>
</tr>
<tr>
<td>components()</td>
<td>tuple := components(); $ tuple of currently restored components name</td>
</tr>
<tr>
<td>ids_in(component)</td>
<td>idList := ids_in(component);</td>
</tr>
<tr>
<td>newids()</td>
<td>tuple := newids(); $ list of defined obj and their type</td>
</tr>
<tr>
<td>newsymbols(x)</td>
<td>tuple := newsymbols(x) $x= om: list of defined obj</td>
</tr>
<tr>
<td></td>
<td>x= 0: list of defined and their type;</td>
</tr>
<tr>
<td></td>
<td>x&gt;0 : list of defined obj their type and their value</td>
</tr>
<tr>
<td>allchanges(x)</td>
<td>tuple := allchanges(x); $ x means the same as for newsymbols. tuple contains all ids which have changed</td>
</tr>
<tr>
<td>visiblechanges(x)</td>
<td>tuple := visiblechanges(x); $ x means the same as for newsymbols. tuple contains visible ids</td>
</tr>
<tr>
<td>resetchanges()</td>
<td>resetchanges();</td>
</tr>
<tr>
<td>ids(x)</td>
<td>tuple := ids(x); $ids(om): tuple of defined ids,</td>
</tr>
<tr>
<td></td>
<td>ids(0): tuple of [type, defined-id]</td>
</tr>
<tr>
<td></td>
<td>ids(1): tuple of [type, defined-id, value]</td>
</tr>
<tr>
<td>gc()</td>
<td>gc(); $ garbage collection invocation</td>
</tr>
<tr>
<td>restore(filename)</td>
<td>restore(filename); $ restore var. identifier and its value if file was 'save'-d or only a value if the file was created by a 'store'</td>
</tr>
<tr>
<td>save(varname)</td>
<td>save(varname, filename); $ if filename is OM or &quot;: interactive selection of file</td>
</tr>
<tr>
<td>store(expression)</td>
<td>store(expression, filename); $ if filename is OM or &quot;: interactive selection of file</td>
</tr>
<tr>
<td>ref(func)</td>
<td>ref(func); $ prints a list of the identifiers referenced by func</td>
</tr>
<tr>
<td>refcollect(truelfalse)</td>
<td>refcollect(truelfalse); $ sets -resp resets- ref collection</td>
</tr>
<tr>
<td>workComp(compName)</td>
<td>workComp(compName); $ cur env is that of compName</td>
</tr>
</tbody>
</table>

---

**Kepler**  
a prototyping company  

**Cantor**  
a prototyping language
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>misc</strong></td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>2</td>
</tr>
<tr>
<td>sorted</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
</tr>
<tr>
<td>blockcount</td>
<td>1</td>
</tr>
<tr>
<td>npow2</td>
<td>2</td>
</tr>
<tr>
<td>qsort</td>
<td>1</td>
</tr>
<tr>
<td>sort_index</td>
<td>1</td>
</tr>
<tr>
<td>break</td>
<td>0</td>
</tr>
<tr>
<td>setallbreakpt</td>
<td>0</td>
</tr>
<tr>
<td>ignoreallbreakpt</td>
<td>0</td>
</tr>
<tr>
<td>albreaktype</td>
<td>0</td>
</tr>
<tr>
<td>brektype</td>
<td>0</td>
</tr>
<tr>
<td>clock</td>
<td>0</td>
</tr>
<tr>
<td>pause</td>
<td>1</td>
</tr>
<tr>
<td>tellfunc</td>
<td>0</td>
</tr>
<tr>
<td>setBaseAtom</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>text-number conversion</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ator</td>
<td>1</td>
</tr>
<tr>
<td>rtoa</td>
<td>1</td>
</tr>
<tr>
<td>atoi</td>
<td>1</td>
</tr>
<tr>
<td>itoa</td>
<td>1</td>
</tr>
<tr>
<td>ord</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>hash</td>
<td>1</td>
</tr>
<tr>
<td>date</td>
<td>0</td>
</tr>
<tr>
<td>uclcase</td>
<td>2</td>
</tr>
<tr>
<td>strsubst</td>
<td>3</td>
</tr>
<tr>
<td>max_line</td>
<td>1</td>
</tr>
<tr>
<td>show_mode</td>
<td>1</td>
</tr>
<tr>
<td>show</td>
<td>n</td>
</tr>
</tbody>
</table>

### Cantor

**primitives list**

- **rank**: $x := \text{rank}(x,L)$; $x$ is an item in $L$ (list or tuple), or a substring if $L$ is a string. Returns 0 if $x$ not in $L$.
- **sorted**: $\text{sorted}(\text{true|false})$; $\$ for displaying sets in h-sorted | unsorted order.
- **size**: $n := \text{size}(\text{object})$; $n$ size in byte of the object and its dependants.
- **blockcount**: $n := \text{blockcount}(\text{object})$; $n$ size in # of basic elements in the object and its dependants.
- **npow2**: $x := \text{npow2}(\text{set},n\text{max})$; $\$the set of all subsets with exactly $n\text{max}$ elements! equivalent to npow, but much faster.
- **qsort**: $\text{tuple} := \text{qsort}(\text{collection})$; $\$ tuple contains the collection in a sorted order.
- **sort_index**: $\text{tuple} := \text{sort_index}(\text{collection})$; $\$ tuple contains the permutation index map to sort collection.
- **break**: $\text{break}();$ force premature (loop) exit.
- **setallbreakpt**: $\text{setallbreakpt}();$ all breakpoint commands set to OM.
- **ignoreallbreakpt**: $\text{ignoreallbreakpt}();$ all breakpoint commands set to noop.
- **albreaktype**: $\text{albreaktype}();$ all breakpoint types.
- **brektype**: $\text{brektype}();$ current breakpoint type.
- **clock**: $\text{tickCount} := \text{clock}();$$\$ approx. 1 Tick evry 16 msec.
- **pause**: $\text{pause}(\text{nb\_sec})$; $\$ suspend all processing for \text{nb\_sec}.
- **tellfunc**: $\text{aFunc} := \text{tellfunc}();$$\$ attempts to tell within which func is current progr ptr.
- **setBaseAtom**: $\text{setBaseAtom}(\text{atomlom})$; $\$ set Base\_Atom to atom 1 !10!.

---

**Kepler**
a prototyping company

**Cantor**
a prototyping language
<table>
<thead>
<tr>
<th>basic file processing</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>close(File);</td>
</tr>
<tr>
<td>eof</td>
<td>bool := eof(File);</td>
</tr>
<tr>
<td>opena</td>
<td>File := opena(file_name); $open append text file</td>
</tr>
<tr>
<td>openab</td>
<td>File := openab(file_name); $open append binary file</td>
</tr>
<tr>
<td>openr</td>
<td>File := openr(file_name); $open read text file</td>
</tr>
<tr>
<td>openrb</td>
<td>File := openrb(file_name); $open read binary file</td>
</tr>
<tr>
<td>openrw</td>
<td>File := openrw(file_name); $open read-write text file</td>
</tr>
<tr>
<td>openrwb</td>
<td>File := openrwb(file_name); $open read-write binary file</td>
</tr>
<tr>
<td>openw</td>
<td>File := openw(file_name); $open write text file</td>
</tr>
<tr>
<td>openwb</td>
<td>File := openwb(file_name); $open write binary file</td>
</tr>
<tr>
<td>fwrite</td>
<td>fwrite(item,File);</td>
</tr>
<tr>
<td>fread</td>
<td>value_read := fread(File,'int'l'string',count);</td>
</tr>
<tr>
<td>fseek</td>
<td>fseek(File,f_position);</td>
</tr>
<tr>
<td>ftell</td>
<td>f_position := ftell(File);</td>
</tr>
<tr>
<td>rewind</td>
<td>rewind(File);</td>
</tr>
<tr>
<td>toend</td>
<td>toend(File);</td>
</tr>
<tr>
<td>flen</td>
<td>n := flen(File,fileName); $file size</td>
</tr>
<tr>
<td>lc</td>
<td>n := lc(File,fileName); $text file line count</td>
</tr>
<tr>
<td>fgets</td>
<td>string := fgets(n,File); $read a line of at most n char;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>advanced file primitives</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcopy</td>
<td>n := fcopy(from,to); $copy from a File to a File</td>
</tr>
<tr>
<td>finsert</td>
<td>n := finsert(from,to); $copy from a File to a text pane</td>
</tr>
<tr>
<td>panecopy</td>
<td>n := panecopy(from,to); $copy from a text pane to a File</td>
</tr>
<tr>
<td>file_find_str</td>
<td>bool := file_find_str(aFile,string); $find a string within a file</td>
</tr>
<tr>
<td>rename</td>
<td>bool := rename(from,to); $rename from a File to a File; returns true iff successful</td>
</tr>
<tr>
<td>fdelete</td>
<td>bool := fdelete(filename); $delete a File; returns true iff successful</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>scope control</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>applyEnv</td>
<td>fn1 := applyEnv(fn1,env); $set fn1 env to env if specified, otherwise to the current env</td>
</tr>
<tr>
<td>applyNilEnv</td>
<td>fn1 := applyNilEnv(fn1); $set fn1 env to the current env</td>
</tr>
<tr>
<td>hasNilEnv</td>
<td>bool := hasNilEnv(fn1); $true if fn is a func with Nil Env</td>
</tr>
<tr>
<td>detachEnv</td>
<td>env := detachEnv(); $unlink the current func's env from creator's</td>
</tr>
<tr>
<td>codeOf</td>
<td>code := codeOf(func); $code is a non-printable object</td>
</tr>
<tr>
<td>overrideOf</td>
<td>aMap := overrideOf(func); $aMap is a s-map</td>
</tr>
<tr>
<td>envOf</td>
<td>env := envOf(func); $env is a non-printable object</td>
</tr>
<tr>
<td>mkLocal</td>
<td>mkLocal(idName,aFunc); $creates a local var in the environment of aFunc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>drawing</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearscreen</td>
<td>clearscreen(); $clears the screen</td>
</tr>
</tbody>
</table>

**Kantor**  
a prototyping language

**Cantor**  
primitives list
### Cantor

| primitives list |
|-----------------|-----------------|-----------------|
| **lineto**      | 2               | **lineto(x,y)**; $ draws a line from current pen location |
| **moveto**      | 2               | **moveto(x,y)**; $ move pen loc to (x,y) in (horiz, vert ) coordinate system |
| **arc**         | 4               | **arc( [x,y,rx,ry],startangle,arcangle ); $ frame an arc within the (rx,ry)-box at (x,y) with the given angles.** |
| **farc**        | 4               | **farc( [x,y,rx,ry],startangle,arcangle );$ fill arc** |
| **box**         | 4               | **box( [x,y,wx,wy] l x, y, wx, wy ); $ frame a box at (x,y) having: width = wx, height = wy** |
| **fbox**        | 4               | **fbox( [x,y,wx,wy] l x, y, wx, wy ); $ fill box** |
| **ellipse**     | 4               | **ellipse( [x,y,wx,wy] l x, y, wx, wy ); $ frame an oval within the rect at (x,y) having: width = wx, height = wy** |
| **fellipse**    | 4               | **fellipse( [x,y,wx,wy] l x, y, wx, wy ); $ fill ellipse** |
| **circle**      | 3               | **circle( x, y, r ); $ center: (x,y) radius: r** |
| **fcircle**     | 3               | **fcircle( x, y, r ); $ fill circle** |
| **polygon**     | 1               | **polygon(poly); $ input is: poly := [[x1,y1],[x2,y2]]** |
| **fpolygon**    | 1               | **fpolygon(poly);$ fill polygon poly: [[x1,y1],...,[xn,yn]]** |
| **font**        | 1               | **font_nbr := font(font_name_size)$ e.g. "monaco9", "helvetica12"** |
| **gputs**       | 1               | **gputs(string); $ draws string at current pen location** |
| **color**       | 1               | **color(gray_color | RGB_color)** |
| **hascolor**    | 0               | **bool := hascolor(); $ tells if current screen has color** |
| **allocolor**   | 3               | **allocolor(r,b,g);$ color allocation: r,b,g: real numbers** |
| **set_gc**      | 1               | **set_gc(gcMap);$ restore graf context described in gcMap** |
| **get_gc**      | 0               | **gcMap := get_gc(); $ save current graf context in gcMap** |
| **acquire**     | 0               | **acquire();$ offscreen acquire** |
| **release**     | 0               | **release();$ offscreen release** |
| **set_cursor**  | 1               | **setcursor(cursor_name); $ select a cursor by name** |
| **hilite**      | 4               | **hilite(win_no, aCodeStr, [xl,yl], [x2,y2]); $ aCodeStr is 'invert'l'rect'l'vector'l'ellipse'** |
| **compute_rect_text** | 4 | **[[x,y],[w,h]] := compute_rect_text(fontname, x, y, str);** |

### Windows

| windows |
|---------|---------|---------|---------|---------|---------|
| **openwindow** | 1 | **idWin := openwindow(anAttrMap);** |
| **closewindow** | 0 | **closewindow(); $ closes current window** |
| **show_hide** | 2 | **win_no := show_hide(win_no I -1, truefalse); if 2nd arg = true then show window, otherwise hide it** |
| **window** | 1 | **curwin_nbr := window(win_nbr); $ win_nbr = -1: tells which win ic current window, win_nbr >=0: change curwin to win_nbr** |
| **set_window** | 1 | **curwin_nbr := set_window(win_nbr); $ see window()** |
| **window_attributes** | 0 | **window_attributes();$ help: the window attribute map structure** |
| **set_window_attributes** | 1 | **set_window_attributes(anAttrMap); $ change win. attr** |
| **get_window_attributes** | 1 | **anAttrMap := get_window_attributes(anAttrSet); $ get anAttrMap with domain corresp. to anAttrSet** |
| **ask_str** | 4 | **anAnswer := ask_str( x, y, aMsg, aQuestion ); $ a dialog returning a string** |
| **ask_real** | 4 | **anAnswer := ask_real( x, y, aMsg, aQuestion ); $ a dialog returning a real nbr** |
| **ask_int** | 4 | **anAnswer := ask_int( x, y, aMsg, aQuestion ); $ a dialog returning an integer** |

---

**Kepler**
a prototyping company

**Cantor**
a prototyping language
<table>
<thead>
<tr>
<th>Cantor</th>
<th>primitives list</th>
</tr>
</thead>
<tbody>
<tr>
<td>ask_long</td>
<td>4 anAnswer := ask_long( x, y, aMsg, aQuestion ); $ a dialog returning a bignum</td>
</tr>
<tr>
<td>setTitle</td>
<td>1 settitle(title); $ change window title</td>
</tr>
<tr>
<td>tell</td>
<td>1 tell(msg); $ open a warning window</td>
</tr>
<tr>
<td>events</td>
<td></td>
</tr>
<tr>
<td>wait_mask_event</td>
<td>2 win_no := wait_mask_event( win_no,event_mask_set); $ nondiscriminating wait: wait_mask_event(-1,{})</td>
</tr>
<tr>
<td>check_mask_event</td>
<td>2 anEventMap := check_mask_event( win_no, event_mask_set); $check all events:check mask event(-1,{});</td>
</tr>
<tr>
<td>PostHEvent</td>
<td>2 PostHEvent(win_no, event_map); $ post the event defined in event_map:</td>
</tr>
<tr>
<td>interface_task</td>
<td>0 interface_task(); $ insert everywhere to allow queued events processing</td>
</tr>
<tr>
<td>event_mask</td>
<td>0 event_mask(); $ help: the event type structure</td>
</tr>
<tr>
<td>event_map</td>
<td>0 event_map(); $ help: the event map structure</td>
</tr>
<tr>
<td>event_domains</td>
<td>0 event_domains(); $ help: more on event map structure</td>
</tr>
<tr>
<td>event_convert</td>
<td>1 event_typenameevent_typeNbr := event_convert( event_typeNbr, event_typeName); $ for CantorDrvMap, convert type Nbr into name and conversely</td>
</tr>
<tr>
<td>menus</td>
<td></td>
</tr>
<tr>
<td>install_menu</td>
<td>4 menuID := install_menu(win_no, ['item1', ..,'itemN'], [ox,oy], [w,h]); $ ox,oy, [w,h] designates the active rectangle</td>
</tr>
<tr>
<td>remove_menu</td>
<td>1 remove_menu(win_no, menuID);</td>
</tr>
<tr>
<td>popupmenu</td>
<td>3 itemID := popupmenu(win_no, menuID, [ox,oy]); $ [ox,oy] indicates where to display the menu</td>
</tr>
<tr>
<td>getpopup</td>
<td>1 [(x,y),itemID] := getpopup(['item1', ..,'itemN']); $ returns the selection within popup item list</td>
</tr>
<tr>
<td>set_item</td>
<td>3 set_item(menuID,menuItem, item_string); $ replace in menu menuID the item # menuItem by item_string</td>
</tr>
<tr>
<td>set_item_status</td>
<td>3 set_item_status(menuID,menuItem, bool 10110m); $ 3rd arg: bool-&gt;check (resp uncheck) item, 0(resp 1,0m) -&gt; disable(resp enable) item</td>
</tr>
<tr>
<td>regions</td>
<td></td>
</tr>
<tr>
<td>clipwindow</td>
<td>1 clipwindow(region);</td>
</tr>
<tr>
<td>create_region</td>
<td>0 region := create_region();</td>
</tr>
<tr>
<td>rect_region</td>
<td>2 region := rect_region([ox,oy],[w,h]);</td>
</tr>
<tr>
<td>destroy_region</td>
<td>1 destroy_region(region);</td>
</tr>
<tr>
<td>union_region</td>
<td>2 region := union_region(region1, region2);</td>
</tr>
<tr>
<td>buttons</td>
<td></td>
</tr>
<tr>
<td>create_button</td>
<td>1 button := create_button(anAttrMap);</td>
</tr>
<tr>
<td>set_button_attributes</td>
<td>2 set_button_attributes(button, anAttrMap);</td>
</tr>
<tr>
<td>get_button_attributes</td>
<td>2 anAttrMap := get_button_attributes(button, anAttrSet);</td>
</tr>
<tr>
<td>destroy_button</td>
<td>1 destroy_button(button);</td>
</tr>
</tbody>
</table>

**Kepler**
a prototyping company

**Cantor**
a prototyping language
<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw_buttons</td>
<td>1</td>
<td>draw_buttons( win_no );</td>
</tr>
<tr>
<td>get_button_value</td>
<td>1</td>
<td>button_value := get_button_value(button);</td>
</tr>
<tr>
<td>button_attributes</td>
<td>0</td>
<td>button_attributes();$ help: the button attribute map structure</td>
</tr>
</tbody>
</table>

**text pane**

<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_text-pane</td>
<td>2</td>
<td>panelID := create_text-pane(win_no, tp_map);</td>
</tr>
<tr>
<td>delete_text-pane</td>
<td>1</td>
<td>delete_text-pane(panelID);</td>
</tr>
<tr>
<td>tp_get_select</td>
<td>1</td>
<td>[selStart, selEnd] := tp_get_select(panelID); $ ok for grids</td>
</tr>
<tr>
<td>tp_set_select</td>
<td>2</td>
<td>tp_set_select(panelID, [selStart, selEnd]); $ ok for grids</td>
</tr>
<tr>
<td>tp_return_select</td>
<td>1</td>
<td>selStr := tp_return_select(panelID); $ ok for grids</td>
</tr>
<tr>
<td>tp_show_selection</td>
<td>1</td>
<td>tp_show_selection(panelID);</td>
</tr>
<tr>
<td>tp_delete_selection</td>
<td>1</td>
<td>tp_delete_selection(panelID);</td>
</tr>
<tr>
<td>tp_insert</td>
<td>2</td>
<td>tp_insert(panelID, aString); $ at most 8000 char long; $ ok for grids</td>
</tr>
<tr>
<td>tp_text_length</td>
<td>1</td>
<td>tp_text_length(panelID);</td>
</tr>
<tr>
<td>tp_set_prompt</td>
<td>2</td>
<td>tp_set_prompt(panelID, aString);</td>
</tr>
<tr>
<td>tp_select_paragraph</td>
<td>1</td>
<td>tp_select_paragraph(panelID);</td>
</tr>
<tr>
<td>tp_set_focus</td>
<td>1</td>
<td>tp_set_focus(panelID);</td>
</tr>
<tr>
<td>tp_set_handler</td>
<td>2</td>
<td>tp_set_handler(panelID,'console'edit'field'data'sel'); $ set (change) the key_handler</td>
</tr>
<tr>
<td>textPane_attributes</td>
<td>0</td>
<td>textPane_attributes();$ the textPane attribute map structure</td>
</tr>
<tr>
<td>tp_set_handler</td>
<td>2</td>
<td>tp_set_handler(panelID,'console'edit'field'data'sel'); $ set (change) the key_handler</td>
</tr>
</tbody>
</table>

**pane files**

<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_pane_file</td>
<td>3</td>
<td>pane_file := open_pane_file(panelID, win_no, mode); $- &gt; mode == 'r' or 'w'</td>
</tr>
<tr>
<td>get_pane_id</td>
<td>1</td>
<td>panelID := get_pane_id(pane_file); $ --&gt; return -1 if pane_file is a disk file</td>
</tr>
<tr>
<td>get_file_window</td>
<td>1</td>
<td>win_no := get_file_window(pane_file); $ --&gt; return -1 if pane_file is a disk file</td>
</tr>
<tr>
<td>echo_pane_file</td>
<td>2</td>
<td>echo_pane_file(pane_file, echo_file);</td>
</tr>
<tr>
<td>remove_echo_file</td>
<td>1</td>
<td>remove_echo_file(echo_file);</td>
</tr>
</tbody>
</table>

**logo turtle**

<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearscreen, cs</td>
<td>0</td>
<td>clearscreen(); $ clears the screen cs(); $ clears the screen</td>
</tr>
<tr>
<td>round</td>
<td>1</td>
<td>n := round(x);</td>
</tr>
<tr>
<td>set_world</td>
<td>2</td>
<td>set_world([[x,y],[w,h]]); $ rect: [x,y],[w,h]</td>
</tr>
<tr>
<td>set_view</td>
<td>2</td>
<td>set_view([[x,y],[w,h]]);</td>
</tr>
<tr>
<td>wtov</td>
<td>1</td>
<td>v_pt := wtov(w_pt);</td>
</tr>
<tr>
<td>vtov</td>
<td>1</td>
<td>w_pt := vtov(v_pt);</td>
</tr>
<tr>
<td>move_to</td>
<td>1</td>
<td>move_to(w_pt);</td>
</tr>
<tr>
<td>line_to</td>
<td>1</td>
<td>line_to(w_pt);</td>
</tr>
<tr>
<td>dot</td>
<td>1</td>
<td>dot(w_pt);</td>
</tr>
</tbody>
</table>

**Kepler**
a prototyping company

**Cantor**
a prototyping language
Cantor
primitives list

<table>
<thead>
<tr>
<th>Cantor</th>
<th>primitives list</th>
</tr>
</thead>
<tbody>
<tr>
<td>cligne_line</td>
<td>2</td>
</tr>
<tr>
<td>is_pendown</td>
<td>0</td>
</tr>
<tr>
<td>set_pendown</td>
<td>1</td>
</tr>
<tr>
<td>pendown, pd</td>
<td>0</td>
</tr>
<tr>
<td>penup, pu</td>
<td>0</td>
</tr>
<tr>
<td>pos</td>
<td>0</td>
</tr>
<tr>
<td>set_pos</td>
<td>1</td>
</tr>
<tr>
<td>stdpt</td>
<td>1</td>
</tr>
<tr>
<td>home</td>
<td>0</td>
</tr>
<tr>
<td>std_dir</td>
<td>1</td>
</tr>
<tr>
<td>forward, fd</td>
<td>1</td>
</tr>
<tr>
<td>backward, bk</td>
<td>1</td>
</tr>
<tr>
<td>left, lt</td>
<td>1</td>
</tr>
<tr>
<td>right, rt</td>
<td>1</td>
</tr>
<tr>
<td>set_turtle</td>
<td>3</td>
</tr>
<tr>
<td>ngon</td>
<td>2</td>
</tr>
<tr>
<td>init_turtle</td>
<td>0</td>
</tr>
<tr>
<td>open_std_turtle</td>
<td>0</td>
</tr>
<tr>
<td>cngon</td>
<td>2</td>
</tr>
</tbody>
</table>

abstract syntax

| parsetree | 0 | parsetree(); $ returns the accumulated parsetrees |
| resetParses | 0 | resetParses(); $ set Parses to Nil |
| scan | 3 | scan (File|file Name|om| string, textScan|textAndNumScan, strScan); $ produces a tuple -- ScanStop (terminator) is '.' |
| textScan /= om --> spec. symbols (eg $, quotes) are parsed as tokens and returned |
| strScan /= om --> 1st arg = input string |
| char in cantor AlphaNumSet are considered alpha-num |
| setScanStop | 1 | aChar := setScanStop(aChar); $aChar becomes the new scan stop char; default ScanStop is '.' |
| which_ast | 1 | astType := which_ast(anAst, anInteger, aString); $ produces an Ast Type |
| is_ast_leaf | 1 | bool := is_ast_leaf(anAst, anInteger); $ id, int, real, spec, string are leaves |
| ast | 2 | ast(anAST,0|1|2|type'|3);$ |
| 0:father, 1:left, 2:right, >=3:more |
| copy_ast | 1 | aNewAST := copy_ast(anAST); |
| coref_ast | 1 | an := coref_ast(anAST); $if an = om then anAST is atree, else a DAG, and an is the 1st coref |
| setAst | 3 | anAST := setAst(anAST, 1|0|1|2|type'|3, modif);$ |
| -1:parent, 0:node itself, 1:left, 2:right, >=3:more |
| setCopyAst | 3 | anAST := setCopyAst(Ast1, 1|0|1|2|type'|3, modif);$ |
| -1:parent, 0:node itself, 1:left, 2:right, >=3:more |
| construct | 1 | ast := construct(File|expression_string | om); $ File: the input stream, om: the standard input, expression_string: the expr to parse. construct is a parser |
| analyze | 1 | ast := analyze(File|expression_string | om); $ File: the input stream, om: the standard input, expression_string: the expr to parse. analyze is a parser |
| upd_chain_ast | 1 | upd_chain_ast(ast); $ link all nodes to their father node |

Kepler
a prototyping company

Cantor
a prototyping language
is_chained 1 bool := is_chained(ast); $ test if there is a link from all nodes to their father node

chain_ast 1 chain_ast(ast); $ link all nodes to their father node

up 1 ast := up(ast); $ move up the tree to the father node

eval 1 eval(ast); $ generates the ast-code and executes

evalref 1 evalref(ast); $ generates and analyzes the ast-code

findAll 1 findAll(atyp,ast); $ produces a tuple of all the ast subtrees with the given type

varsOf 1 varsOf(ast); $ produces a tuple of all the referenceable ids appearing in ast

upTo 2 anAst := upTo(ast,atyp); $ move up the ast from ast to find a node with type atyp; produces om if no match; otherwise returns the node whose parent matches

ugly 1 str := ugly(ast); $ ugly print the ast arg\'n

pretty 1 str := pretty(ast); $ pretty print the ast arg\'n

prettyStrings 1 prettyStrings(true|false); $ pretty print in mode: emphasize strings quoting

findAst 2 findAst(atyp,ast); $ atyp = intstringset\::\ producers a tuple of all the ast sub-trees with the given type

match 2 match(ast1,ast2); $ true if ast1, ast2 are eq or if ast2 contains pattern matching ast1 subtrees or if ast1 contains patterns matching ast2 subtrees which will then be subsituted into ast1

well_defined 2 bool := well_defined(af opt sw); $ if sw = om: are all the variables occuring free in af well-defined? $ if sw /= om: is subtree a well-defined expression

varsIn 1 varsIn(ast); $ produces a tuple of all the variables appearing in ast, excluding Selectors

parse_msg 1 parse_msg(bool); $ false->no parsing msg, true->msg

unify 2 [unified_S, unif_map] := unify(S opt constants); $ $ \$ is a collection of ASTs, constants is a set or tuple (even a single string is OK) of strings, or AST T_Id\'s, representing constant symbol identifiers

unif_step 2 match_tree_pair := unif_step(ast1,ast2); $ if ast1, ast2 are unifiable, returns the 1st non-matched term pair

ast_subs 2 anAst := ast_subs(ast1,subs_map \::\ (string|cid,ast2)); $ subs_map map strings or T_Id trees onto other ASTs -- all occurrences of T_Ids in the domain of subs_map are subsituted

ast_id_str_subs 3 anAst := ast_id_str_subs(ast1,set_of_str \::\ tup_of_str \::\ str, om \::\ anything); $ replace all occurrences of T_Ids with name in the given collection by Strings into AST; do the converse if 3rd arg is non-om

kwd 2 kwd(opt key, token_val); $ if key = om returns tuple of all keys, if key = \" (empty string) reset all keywords to default $ if key = some_str returns the corresponding token, $ if token_val = 0 resets the key, else sets the key to the given token_val

interoperability types designate target Cantor types for C data or data collections converted to or from Cantor (see below mk_cantor_obj)

Kepler
a prototyping company

Cantor
a prototyping language
<table>
<thead>
<tr>
<th>type</th>
<th>primitives list</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define Boolean 2</td>
<td>Boolean 2</td>
</tr>
<tr>
<td>#define Integer 3</td>
<td>Integer 3</td>
</tr>
<tr>
<td>#define Bignum 4</td>
<td>Bignum 4</td>
</tr>
<tr>
<td>#define File 5</td>
<td>File 5</td>
</tr>
<tr>
<td>#define Real 6</td>
<td>Real 6</td>
</tr>
<tr>
<td>#define Set 7</td>
<td>Set 7</td>
</tr>
<tr>
<td>#define String 8</td>
<td>String 8</td>
</tr>
<tr>
<td>#define Tuple 9</td>
<td>Tuple 9</td>
</tr>
<tr>
<td>#define SetOfShortInt 72</td>
<td>SetOfShortInt 72</td>
</tr>
<tr>
<td>#define SetOfInt 73</td>
<td>SetOfInt 73</td>
</tr>
<tr>
<td>#define SetOfBn 74</td>
<td>SetOfBn 74</td>
</tr>
<tr>
<td>#define SetOfFile 75</td>
<td>SetOfFile 75</td>
</tr>
<tr>
<td>#define SetOfReal 76</td>
<td>SetOfReal 76</td>
</tr>
<tr>
<td>#define SetOfStr 78</td>
<td>SetOfStr 78</td>
</tr>
<tr>
<td>#define TupleOfFile 92</td>
<td>TupleOfFile 92</td>
</tr>
<tr>
<td>#define TupleOfInt 93</td>
<td>TupleOfInt 93</td>
</tr>
<tr>
<td>#define TupleOfBn 94</td>
<td>TupleOfBn 94</td>
</tr>
<tr>
<td>#define TupleOfFile 95</td>
<td>TupleOfFile 95</td>
</tr>
<tr>
<td>#define TupleOfReal 96</td>
<td>TupleOfReal 96</td>
</tr>
<tr>
<td>#define TupleOfStr 98</td>
<td>TupleOfStr 98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C types</th>
<th>interoperability functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_ Val /* a structure for converting data types */</td>
<td>C-primitives for C-to-Cantor or Cantor-to-C</td>
</tr>
<tr>
<td>Cantor_Ptr /* the Cantor data structure pointer */</td>
<td></td>
</tr>
</tbody>
</table>

| convert_cantor_id         | 1 | C_ Val *convert_cantor_id(P1(char *name)); /* returns the value of the Cantor variable name */ |
| convert_cantor_obj        | 1 | C_ Val *convert_cantor_obj(P1(Cantor_Ptr obj)); /* returns the value of the Cantor object obj */ |
| mk_cantor_obj             | 4 | Cantor_Ptr mk_cantor_obj(P4(char *name, short type, void *value, int siz)); /* create a Cantor object associated to the variable name, of the given type, value (and size for arrays) */ |
| mk_cantor_id              | 1 | Cantor_Ptr mk_cantor_id(P1(char *name)); /* create a Cantor object associated to the variable name */ |
| mk_local_id               | 1 | Cantor_Ptr mk_local_id(P1(char *name)); /* same as mk_cantor_id but creates a non-global variable */ |

<table>
<thead>
<tr>
<th>invocation (from Cantor) of C-procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>a specific packaging is required</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>invocation (from C) of Cantor</th>
</tr>
</thead>
<tbody>
<tr>
<td>self_interp (see interp)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>macintosh specific misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_in_file_name</td>
</tr>
<tr>
<td>get_out_file_name</td>
</tr>
<tr>
<td>setfilecreator</td>
</tr>
<tr>
<td>add_mac_menu</td>
</tr>
<tr>
<td>check_menu_item</td>
</tr>
</tbody>
</table>

**Kepler**

a prototyping company

**Cantor**

a prototyping language
<table>
<thead>
<tr>
<th>Cantor</th>
<th>primitives list</th>
</tr>
</thead>
<tbody>
<tr>
<td>openresfile</td>
<td>refnum := openresfile(resFileName); $ resFileName is a string</td>
</tr>
<tr>
<td>closeresfile</td>
<td>refnum := closeresfile(refnum); $ refnum is an int</td>
</tr>
<tr>
<td>getnewdialog</td>
<td>dialog := getnewdialog(dialogResld); $ dialogResld is an int</td>
</tr>
<tr>
<td>closedialog</td>
<td>dialog := closedialog(dialog); $ dialog was returned by getnewdialog( .. )</td>
</tr>
<tr>
<td>getditem</td>
<td>[itemType,itemHandle,box] := getditem(dialog,itemNo); $ dialog and itemHandle are Bignum; box is [x,y,w,h]</td>
</tr>
<tr>
<td>setditem</td>
<td>setditem(dialog,itemNo,itemType,itemHandle,box); $ dialog and itemHandle are Bignum; box is [x,y,w,h]</td>
</tr>
<tr>
<td>getitext</td>
<td>atext := getitext(itemHandle); $ itemHandle is a Bignum;</td>
</tr>
<tr>
<td>setitext</td>
<td>setitext(itemHandle,atext); $ itemHandle is a Bignum;</td>
</tr>
<tr>
<td>getctitle</td>
<td>atext := getctitle(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>setctitle</td>
<td>setctitle(ControlItemHandle,atext); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>hidecontrol</td>
<td>hidecontrol(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>showcontrol</td>
<td>showcontrol(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>getctlvalue</td>
<td>anInt := getctlvalue(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>setctlvalue</td>
<td>setctlvalue(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>getctlmin</td>
<td>anInt := getctlmin(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>setctlmin</td>
<td>setctlmin(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>getctlmax</td>
<td>anInt := getctlmax(ControlItemHandle); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>setctlmax</td>
<td>setctlmax(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;</td>
</tr>
<tr>
<td>hitdialog</td>
<td>hitdialog(dialog,itemHit); $ dialog is a Bignum; itemHit is an int</td>
</tr>
<tr>
<td>dialog</td>
<td>dialog(dialog); $ dialog is a Bignum; itemHit is an int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>macintosh vector graphics</th>
<th>QuickDraw graphics based: an experimental set of primitives, not optimally operational</th>
</tr>
</thead>
<tbody>
<tr>
<td>getobj</td>
<td>[x,y,w,h] := getobj(oml \text{r}[ect]</td>
</tr>
<tr>
<td>getrect</td>
<td>[x,y,w,h] := getrect( oml \text{r}[ect]</td>
</tr>
<tr>
<td>getvector</td>
<td>[x1,y1,x2,y2,win_no] := getvector( [x1,y1]lom, [x1,y1]lom, len{-0,2\text{-}4\text{-}8}); $ create interactively a vector: if len &gt; 0, create a vector of this length $ -2: take [x1,y1] as origin, -4: create vect parallel to given line\n$ -8: create vector with same length</td>
</tr>
</tbody>
</table>

Kepler
a prototyping company

Cantor
a prototyping language
<table>
<thead>
<tr>
<th>primitives list</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpoly</td>
<td>2</td>
</tr>
<tr>
<td>saveimage</td>
<td>2</td>
</tr>
<tr>
<td>restoreimage</td>
<td>2</td>
</tr>
<tr>
<td>spextra</td>
<td>1</td>
</tr>
<tr>
<td>DrawPicture</td>
<td>3</td>
</tr>
<tr>
<td>erasel pict</td>
<td>2</td>
</tr>
<tr>
<td>change picset</td>
<td>2</td>
</tr>
<tr>
<td>get sel pict</td>
<td>1</td>
</tr>
<tr>
<td>get port pict</td>
<td>1</td>
</tr>
<tr>
<td>set sel pict</td>
<td>2</td>
</tr>
<tr>
<td>set port pict</td>
<td>2</td>
</tr>
<tr>
<td>get sel in rect</td>
<td>3</td>
</tr>
<tr>
<td>get sel at point</td>
<td>3</td>
</tr>
<tr>
<td>move focus to</td>
<td>2</td>
</tr>
<tr>
<td>set origin</td>
<td>2</td>
</tr>
<tr>
<td>move select pict</td>
<td>2</td>
</tr>
<tr>
<td>move pict</td>
<td>3</td>
</tr>
<tr>
<td>translate pict</td>
<td>3</td>
</tr>
<tr>
<td>enum pict</td>
<td>1</td>
</tr>
<tr>
<td>kill pict</td>
<td>1</td>
</tr>
<tr>
<td>dupl pict</td>
<td>1</td>
</tr>
<tr>
<td>pic rect</td>
<td>1</td>
</tr>
<tr>
<td>clear sel pict</td>
<td>1</td>
</tr>
<tr>
<td>cut sel pict</td>
<td>1</td>
</tr>
<tr>
<td>copy sel pict</td>
<td>1</td>
</tr>
</tbody>
</table>

**Kepler**
a prototyping company

**Cantor**
a prototyping language
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pastepict</td>
<td>paste pict in window</td>
</tr>
<tr>
<td>portrect</td>
<td>port rect of window</td>
</tr>
<tr>
<td>portframe</td>
<td>port frame of window</td>
</tr>
<tr>
<td>viewframe</td>
<td>view frame of window</td>
</tr>
<tr>
<td>screen</td>
<td>screen frame</td>
</tr>
<tr>
<td>textbox</td>
<td>text box</td>
</tr>
<tr>
<td>dragpict</td>
<td>drag pict</td>
</tr>
<tr>
<td>deselect</td>
<td>deselect all items in window</td>
</tr>
<tr>
<td>redraw</td>
<td>redraw all items in window</td>
</tr>
<tr>
<td>inrect</td>
<td>inside rect of window</td>
</tr>
<tr>
<td>grid_attributes</td>
<td>grid attributes</td>
</tr>
<tr>
<td>grid_databounds</td>
<td>grid bounds</td>
</tr>
<tr>
<td>create grid pane</td>
<td>create grid pane</td>
</tr>
<tr>
<td>add_col</td>
<td>add col to grid</td>
</tr>
<tr>
<td>del_col</td>
<td>delete col from grid</td>
</tr>
<tr>
<td>add_row</td>
<td>add row to grid</td>
</tr>
<tr>
<td>del_row</td>
<td>delete row from grid</td>
</tr>
<tr>
<td>gridcopy</td>
<td>grid copy</td>
</tr>
<tr>
<td>grid_deselect</td>
<td>grid deselect</td>
</tr>
<tr>
<td>grid_selfflags</td>
<td>grid self flags</td>
</tr>
<tr>
<td>selected_cells</td>
<td>selected cells</td>
</tr>
<tr>
<td>grid_select_all</td>
<td>select all cells</td>
</tr>
<tr>
<td>grid_add_to</td>
<td>add to grid</td>
</tr>
</tbody>
</table>

### Kepler

a prototyping company

### Cantor

a prototyping language

**Grids and Cells**
an experimental set of primitives, extending text-pane primitives, not optimally operational

- **grid_attributes**: display grid attributes
- **grid_databounds**: grid bounds
- **create_grid-pane**: create grid pane
- **add_col**, **del_col**: add or delete col
- **add_row**, **del_row**: add or delete row
- **gridcopy**: copy grid
- **grid_deselect**: deselect all cells
- **grid_selfflags**: modify grid selection flags
- **selected_cells**: selected cells
- **grid_select_all**: select all cells
- **grid_add_to**: add to grid
<table>
<thead>
<tr>
<th>Cantor Global var</th>
<th>these variables have default values OM or { }</th>
</tr>
</thead>
<tbody>
<tr>
<td>CantorDrvrMap</td>
<td>CantorDrvrMap((win_no,event_type)) := aFunc; $</td>
</tr>
<tr>
<td></td>
<td>if an event of type event_type occurs in window win_no, and</td>
</tr>
<tr>
<td></td>
<td>has for event map ev, aFunc(ev) is run</td>
</tr>
<tr>
<td>MenuBarMap</td>
<td>MenuBarMap(win_no) := aFunc; $</td>
</tr>
<tr>
<td></td>
<td>if an event of type EVENT_MENU or EVENT_MENU_BAR occurs in</td>
</tr>
<tr>
<td></td>
<td>window win_no, and with selected item number isel, then</td>
</tr>
<tr>
<td></td>
<td>aFunc(isel) is run</td>
</tr>
<tr>
<td>StaticMenuDrvrMap</td>
<td>StaticMenuDrvrMap(selected_text) := aFunc; $</td>
</tr>
<tr>
<td></td>
<td>if an event of type EVENT_MENU occurs in any static window</td>
</tr>
<tr>
<td></td>
<td>and selects the string selected_text, then aFunc() is run</td>
</tr>
<tr>
<td>cantor_AlphaNumSet</td>
<td>contains all non standard alphanumeric character (used by</td>
</tr>
<tr>
<td></td>
<td>scan)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cantor directives</th>
<th>these are execution or compilation directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate</td>
<td>!allocate mem_size $ memory pre-allocation</td>
</tr>
<tr>
<td>breakpoint on/off</td>
<td>!breakpoint on/off $ enable/disable breakpoints over watched</td>
</tr>
<tr>
<td></td>
<td>objects</td>
</tr>
<tr>
<td>clear</td>
<td>!clear $ clear the text input buffer (e.g. after syntax error)</td>
</tr>
<tr>
<td>changes on/off</td>
<td>!changes on/off $ enable/disable changes monitoring</td>
</tr>
<tr>
<td>code on/off</td>
<td>!code on/off $ enable/disable code production trace</td>
</tr>
<tr>
<td>echo on/off</td>
<td>!echo on/off $ enable/disable echo of console input</td>
</tr>
<tr>
<td>flex_alloc on/off</td>
<td>!flex_alloc on/off $ enable/disable flexible self-adjustable</td>
</tr>
<tr>
<td></td>
<td>memory allocation policies (cf flex_min_size)</td>
</tr>
<tr>
<td>flex_min_size</td>
<td>!flex_min_size block_size $ suggest free block size; for most</td>
</tr>
<tr>
<td></td>
<td>applications 200 is ok (cf flex_alloc)</td>
</tr>
<tr>
<td>gc on/off</td>
<td>!gc on/off $ enable/disable garbage collection stat. trace</td>
</tr>
<tr>
<td>help</td>
<td>!help pattern $ supplies a list of all the primitives</td>
</tr>
<tr>
<td></td>
<td>matching the pattern</td>
</tr>
<tr>
<td>ids</td>
<td>!ids $ list of all defined (non-om) identifiers</td>
</tr>
<tr>
<td>include</td>
<td>!include filename $ insert as Cantor source the file's contents</td>
</tr>
<tr>
<td>load</td>
<td>!load filename $ insert as binary prog. component the file's</td>
</tr>
<tr>
<td></td>
<td>contents</td>
</tr>
<tr>
<td>memory</td>
<td>!memory memsize $ sets the upperbound of gc memory</td>
</tr>
<tr>
<td>new_ids</td>
<td>!new_ids $ lists all var created by the prog in the current</td>
</tr>
<tr>
<td></td>
<td>component (see work_ids, oms,ids)</td>
</tr>
<tr>
<td>oms</td>
<td>!oms $ lists all undefined var created by the prog in the</td>
</tr>
<tr>
<td></td>
<td>current component (see new_ids, work_ids, ids)</td>
</tr>
<tr>
<td>passive_err on/off</td>
<td>!passive_err on/off $ disable/enable interactive</td>
</tr>
<tr>
<td></td>
<td>acknowledgement of detected errors</td>
</tr>
<tr>
<td>quit</td>
<td>!quit $ quit Cantor session</td>
</tr>
<tr>
<td>record</td>
<td>!record filename $ record all input to console in file</td>
</tr>
<tr>
<td></td>
<td>!record $ no filename arg --&gt; stop recording</td>
</tr>
</tbody>
</table>

**Kepler**  
a prototyping company  

**Cantor**  
a prototyping language
<table>
<thead>
<tr>
<th>Cantor primitives list</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>recordOutput</td>
<td>!recordOutput $ record all console output in file</td>
</tr>
<tr>
<td></td>
<td>!recordOutput $ no filename arg --&gt; stop recording</td>
</tr>
<tr>
<td>reset</td>
<td>!reset $ reset all program variables. Doesn’t reset windows</td>
</tr>
<tr>
<td>save</td>
<td>!save filename $ create a prog. component file copy (see load)</td>
</tr>
<tr>
<td>suspend</td>
<td>!suspend $ suspend execution: same as interp(om); Resume execution by entering return:</td>
</tr>
<tr>
<td>time on/off</td>
<td>!time on/off $ enable/disable date display in traces</td>
</tr>
<tr>
<td>trace on/off</td>
<td>!trace on/off $ enable/disable Cantor Abstract Machine execution trace</td>
</tr>
<tr>
<td>unwatch</td>
<td>!unwatch name $ stop watching the object(s) identified by that name (see watch, breakpoint, allbreakpt())</td>
</tr>
<tr>
<td>verbose on/off</td>
<td>!verbose on/off $ enable/disable detailed display of data involved in traces</td>
</tr>
<tr>
<td>version</td>
<td>!version $ display current Cantor version</td>
</tr>
<tr>
<td>watch</td>
<td>!watch name $ start watching the object(s) identified by that name (see unwatch, breakpoint, allbreakpt())</td>
</tr>
<tr>
<td>work_ids</td>
<td>!work_ids $ lists all defined var created by the prog in the current component (see new_ids, oms, ids)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>experimental directives</th>
<th>Cantor directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotate on/off</td>
<td>!annotate on/off $ enable/disable AST annotation</td>
</tr>
<tr>
<td>in_debug on/off</td>
<td>!in_debug on/off $ enable/disable graphic driver primitives trace</td>
</tr>
<tr>
<td>oldBin</td>
<td>!oldBin $ no longer applicable</td>
</tr>
<tr>
<td>oldReal</td>
<td>!oldReal $ for upward compatibility with binaries containing real nbrs in version V0.45 or before</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pane &amp; file experiments</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>get_stdio</td>
<td>stdin,stdout] := get_stdio(); $--&gt; stdin, stdout : global variables in Cantor</td>
</tr>
<tr>
<td>stdin_from</td>
<td>stdin_from(paneID, win_no); $ redirects</td>
</tr>
<tr>
<td>stdout_to</td>
<td>stdout_to(paneID, win_no); $ redirects</td>
</tr>
<tr>
<td>get_file_value</td>
<td>file_value := get_file_value(file);</td>
</tr>
<tr>
<td>get_file_mode</td>
<td>file_value := get_file_mode(file);</td>
</tr>
<tr>
<td>getPane_file</td>
<td>file_value := get_Plane_file(paneID);</td>
</tr>
<tr>
<td>getOther_Pane_file</td>
<td>file_value := getOther_Pane_file(paneID, pane_file);</td>
</tr>
</tbody>
</table>

Kepler
a prototyping company

Cantor
a prototyping language
Cantor primitives index

**basic math**
- exp
- ln
- log
- max
- min
- abs
- ceil
- fix
- floor
- even
- odd
- random
- randomize
- round
- sgn
- sqrt

**trigonometry**
- cos
- sin
- tan
- acos
- asin
- atan
- cosh
- sinh
- tanh
- acosh
- asinh
- atanh

**basic set primitives**

**source & binary**
- interp
- include
- components
- ids_in
- newids
- newsymbols
- allchanges
- visiblechanges
- resetchanges
- ids
- gc
- restore
- save
- store
- ref
- refcollect
- workComp

**misc**
- rank
- sorted
- size
- blockcount
- npow2
- qsort
- sort_index
- break
- setallbreakpt
- ignoreallbreakpt
- allbreaktype
- breakevent
- clock
- pause
- tellfunc
- setBaseAtom

**type testing**
- is_atom
- is_ast
- is_boolean
- is_bignum
- isDefined
- is_integer
- is_file
- is_floating
- is_func

**conversion**
- ator
- rtoa
- atoi
- itoa
- itoa
- ord

**basic file processing**
- close
- eof
- opena
- openab
- openr
- openrb
- openrw
- openrw
- openwb
- fwrite
- fread
- fseek
- ftell
- rewind
- toend
- flen
- lc
- fgets

**advanced file primitives**
- fcopen
- finsert
- panecopy
- file_find_str
- rename
- fdelete

**scope control**
- applyEnv
- applyNilEnv
- hasNilEnv
- detachEnv
- codeOf
- overrideOf
- envOf
- mkLocal

**drawing**
- clearscreend
- lineto
- moveto
- arc
- farc
- box
<table>
<thead>
<tr>
<th>Cantor primitives list</th>
</tr>
</thead>
<tbody>
<tr>
<td>rect_region</td>
</tr>
<tr>
<td>destroy_region</td>
</tr>
<tr>
<td>union_region</td>
</tr>
<tr>
<td>buttons</td>
</tr>
<tr>
<td>create_button</td>
</tr>
<tr>
<td>set_button_attributes</td>
</tr>
<tr>
<td>get_button_attributes</td>
</tr>
<tr>
<td>destroy_button</td>
</tr>
<tr>
<td>draw_buttons</td>
</tr>
<tr>
<td>get_button_value</td>
</tr>
<tr>
<td>button_attributes</td>
</tr>
<tr>
<td>text pane</td>
</tr>
<tr>
<td>create_text_pan</td>
</tr>
<tr>
<td>delete_text_pan</td>
</tr>
<tr>
<td>tp_get_select</td>
</tr>
<tr>
<td>tp_set_select</td>
</tr>
<tr>
<td>tp_return_select</td>
</tr>
<tr>
<td>tp_show_selection</td>
</tr>
<tr>
<td>tp_delete_selection</td>
</tr>
<tr>
<td>tp_insert</td>
</tr>
<tr>
<td>tp_text_length</td>
</tr>
<tr>
<td>tp_set_prompt</td>
</tr>
<tr>
<td>tp_set_paragraph</td>
</tr>
<tr>
<td>tp_set_focus</td>
</tr>
<tr>
<td>tp_set_handler</td>
</tr>
<tr>
<td>tp_find_string</td>
</tr>
<tr>
<td>text_pane_attributes</td>
</tr>
<tr>
<td>tp_set_handler</td>
</tr>
<tr>
<td>pane files</td>
</tr>
<tr>
<td>open_pane_file</td>
</tr>
<tr>
<td>get_pane_id</td>
</tr>
<tr>
<td>get_file_window</td>
</tr>
<tr>
<td>echo_pane_file</td>
</tr>
<tr>
<td>remove_echo_file</td>
</tr>
<tr>
<td>logo turtle</td>
</tr>
<tr>
<td>cclearscreen,cs</td>
</tr>
<tr>
<td>set_world</td>
</tr>
<tr>
<td>set_view</td>
</tr>
<tr>
<td>wtv</td>
</tr>
<tr>
<td>vtw</td>
</tr>
<tr>
<td>move_to</td>
</tr>
<tr>
<td>line_to</td>
</tr>
<tr>
<td>dot</td>
</tr>
<tr>
<td>cligne_line</td>
</tr>
<tr>
<td>is_pendown</td>
</tr>
<tr>
<td>set_pendown</td>
</tr>
<tr>
<td>pendown,pd</td>
</tr>
<tr>
<td>penup,pu</td>
</tr>
<tr>
<td>pos</td>
</tr>
<tr>
<td>set_pos</td>
</tr>
<tr>
<td>stdpt</td>
</tr>
<tr>
<td>home</td>
</tr>
<tr>
<td>std_dir</td>
</tr>
<tr>
<td>forward, fd</td>
</tr>
<tr>
<td>backward, bk</td>
</tr>
<tr>
<td>left, lt</td>
</tr>
<tr>
<td>right, rt</td>
</tr>
<tr>
<td>set_turtle</td>
</tr>
<tr>
<td>ngon</td>
</tr>
<tr>
<td>init_turtle</td>
</tr>
<tr>
<td>open_std_turtle</td>
</tr>
<tr>
<td>cngon</td>
</tr>
<tr>
<td>abstract</td>
</tr>
<tr>
<td>syntax</td>
</tr>
<tr>
<td>parsetree</td>
</tr>
<tr>
<td>resetParses</td>
</tr>
<tr>
<td>scan</td>
</tr>
<tr>
<td>setScanStop</td>
</tr>
<tr>
<td>which_ast</td>
</tr>
<tr>
<td>is_ast_leaf</td>
</tr>
<tr>
<td>ast</td>
</tr>
<tr>
<td>copy_ast</td>
</tr>
<tr>
<td>coref_ast</td>
</tr>
<tr>
<td>setAst</td>
</tr>
<tr>
<td>setCopyAst</td>
</tr>
<tr>
<td>construct</td>
</tr>
<tr>
<td>analyze</td>
</tr>
<tr>
<td>is_chained</td>
</tr>
<tr>
<td>chain_ast</td>
</tr>
<tr>
<td>up</td>
</tr>
<tr>
<td>eval</td>
</tr>
<tr>
<td>evalref</td>
</tr>
<tr>
<td>findAll</td>
</tr>
<tr>
<td>varsOf</td>
</tr>
<tr>
<td>upTo</td>
</tr>
<tr>
<td>ugly</td>
</tr>
<tr>
<td>pretty</td>
</tr>
<tr>
<td>prettyStrings</td>
</tr>
<tr>
<td>findAst</td>
</tr>
<tr>
<td>match</td>
</tr>
<tr>
<td>well_defined</td>
</tr>
<tr>
<td>varsIn</td>
</tr>
<tr>
<td>parse_msg</td>
</tr>
<tr>
<td>unify</td>
</tr>
<tr>
<td>unify_step</td>
</tr>
<tr>
<td>ast_subs</td>
</tr>
<tr>
<td>ast_id_str_subs</td>
</tr>
<tr>
<td>kwd</td>
</tr>
<tr>
<td>interoperability</td>
</tr>
<tr>
<td>functions</td>
</tr>
<tr>
<td>convert_cantor_id</td>
</tr>
<tr>
<td>convert_cantor_obj</td>
</tr>
<tr>
<td>mk_cantor_obj</td>
</tr>
<tr>
<td>mk_cantor_id</td>
</tr>
<tr>
<td>mk_local_id</td>
</tr>
<tr>
<td>invocation (from Cantor)</td>
</tr>
<tr>
<td>of C-procedures</td>
</tr>
</tbody>
</table>
invocation (from C) of Cantor
  macintosh specific
  misc.

get_in_file_name
get_out_file_name
setfilecreator
add_mac_menu
check_menu_item
openresfile
closeresfile
calldialog
calldialog
setditem
setditem
gettext
gettile
settitle
hidecontrol
showcontrol
getctlvalue
setctlvalue
getctlmin
setctlmin
getctlmax
setctlmax
hidedialog
dialog
  macintosh
  vector
  graphics
getect
getrect
getvector
getpoly
saveimage
restoreimage
spextra
DrawPicture
erasepict
changepicset
getseelpict
getportpict
setselpict
setportpict
getselelnrect
getselelatpoint
movefocususto
setorigin
moveselectpict
movepict
translatepict
enumerator
killpict
duplpict
pictrect
clearselpict
cutselpict
copyselpict
pastepict
portrect
portframe
viewframe
screen
textbox
dragpict
deselect
redraw
inrect
  grids and
cells
  grid_attributes
  grid_databounds
create_gridPane
add_col
del_col
add_row
del_row
gridcopy
grid_deselect
grid_selflags
selected_cells
grid_select_all
grid_add_to
grid_get
grid_set
  Cantor Global
  var
  Cantor DrvrMap
  MenuBarMap
StaticMenuDrvrMap
cantor_AlphaNumSet
    Cantor
directives
allocate
breakpoint on/off
clear
changes on/off
code on/off
echo on/off
flex_alloc on/off
flex_min_size
gc on/off
help
ids
include
load
memory
new_ids
oms
passive_err on/off
quit
record
recordOutput
reset
save
suspend
time on/off
trace on/off
unwatch
verbose on/off
version
watch
work_ids
  experimental
  Cantor
directives
annotate on/off
in_debug on/off
oldBin
oldReal
  pane & file
  experiments
get_stdio
stdin_from
stdout_to
get_file_value
get_file_mode
get_pane_file
get_other_pane_file
  Sorted Topic List
abstract syntax
advanced file
primitives
basic file
processing
basic math
basic set
primitives
buttons
Cantor directives
Cantor Global var
drawing
events
experimental
  Cantor directives
  grids and cells
  interoperability
  functions

Kepler
Cantor primitives index
Cantor primitives list

Cantor

logo turtle
macintosh specific
misc.
macintosh vector
graphics
menus
misc
pane & file
   experiments
pane files
regions
scope control
source & binary
text pane
text-number
conversion
trigonometry
type testing
windows

Sorted Item List

abs
acos
acosh
acquire
add_col
add_mac_menu
add_row
allbreaktype
allchanges
allocate
alloccolor
analyze
annotate on/off
applyEnv
applyNilEnv
arb
arc
asin
asinh
ask_int
ask_long
ask_real
ask_str
ast
ast_id_str_subs
ast_subs
atan
atanh
atoi
ator
backward, bk
blockcount
box
break
breakpoint on/off
breaktype
button_attributes
CantorDrvrmAP
cantor_AlphaNumSet
clearscreen,cs
clearscreen
clearselpict
cligneline
clipwindow
clock
close
closedialog
closeresfile
closewindow
cngon
code on/off
codeOf
color
components
compute_rect_text
construct
convert_cantor_id
convert_cantor_obj
copyast
coref_ast
cos
cosh
create_button
create_grid-pane
create_region
create_text-pane
cutselpict
date
delete_text-pane
del_col
del_row
deselect
destroy_button
destroy_region
detachEnv
dialog
dot
dragpict
DrawPicture
draw_buttons
duplpict
echo on/off
echo_pane_file
echopane_file
enumerpict
envOf
eof
erasepict
eval
evalref
even
event_convert
event_domains
event_map
event_mask
exp
farc
fbox
fcircle
fcopy
fdelete
fellipse
fgets
file_find_str
findAll
findAst
insert
fix
flen
flex_alloc on/off
flex_min_size
floor
font
forward, fd
fpolygon
fread
fseek
ftell
fwrite
gc
gc on/off
getc_title
getctmax
getctlmin
getctlvalue
getditem
getext
getnewdialog
getobj
getpoly
getpopup
getportion
getrect
getselatpoint
getselinrect
getselpict
getvector
get_button_attributes
get_button_value
Cantor

primitives list

get_file_mode
get_file_value
get_file_window
get_gc
get_in_file_name
get_other_pane_file
get_out_file_name
get_pane_file
get_pane_id
get_stdio
get_window_attributes
gputs
gridcopy
grid_add_to
grid_attributes
grid_databounds
grid_deselect
grid_get
grid_select_all
grid_selflags
grid_set
hascolor
hash
hasNilEnv
help
hidecontrol
hilite
hiddialog
home
ids
ids
ids_in
ignoreallbreakpt
image
include
include
init_turtle
inrect
install_menu
interface_task
interp
invocation (from C) of Cantor
invocation (from Cantor) of C-procedures
in_debug on/off
is_ast
is_ast_leaf
is_atom
is_bignum
is_boolean
is_chained
is_defined
is_file
is_floating
is_func
is_integer
is_map
is_number
is_om
is_pendown
is_set
is_string
is_table
is_textpane
is_tuple
itoa
itoat
killpict
kwda
lc
left, lt
lineto
line_to
ln
load
log
match
max
max_line
memory
MenuBarMap
min
mkLocal
mk_cantor_id
mk_cantor_obj
mk_local_id
movefocusto
movepict
moveselectpict
moveto
move_to
newids
newsymbols
new_ids
ngon
npow
npow2
odd
oldBin
oldReal
oms
opena
openab
openr
openrb
openresfile
openrw
openrwb
openw
openwb
openwindow
open_pane_file
open_std_turtle
ord
overrideOf
panecopy
parseree
parse_msg
passive_err on/off
pastepict
pause
pendown, pd
penup, pu
pictrect
polygon
popupmenu
portframe
portrect
pos
PostHEvent
pow
pretty
prettyStrings
qsort
quit
random
randomize
range
rank
record
recordOutput
rect_region
redraw
ref
refcollect
release
remove_echo_file
remove_menu
rename
reset
resetchanges
resetParses
restore
restoreimage
rewind
right, rt
round
rtoa
save
save
saveimage
scan
screen
selected_cells
setallbreakpt
setAst
setBaseAtom
setCopyAst

Kepler

Cantor primitives index
setctitle tp_select_paragraph
setctlmax tp_set_focus
setctlmin tp_set_handler
setctvalue tp_set_handler
setditem tp_set_prompt
setfilecreator tp_set_select
setitext tp_show_selection
setorigin tp_text_length
setportpict trace on/off
setselpict translatepict
taptitle type
set_title uclcase
set_button_attributes ugly
set_cursor unify
set_gc unif_step
set_item union_region
set_item_status unwatch
set_pendown up
set_pos upTo
set_turtle varsIn
set_view varsOf
set_window verbose on/off
set_window_attributes version
set_window_attributes visiblechanges
set_window_attributes vtow
show showcontrol
show_hide wait_mask_event
show_mode watch
sin well_defined
sinh which_ast
size window
size window_attributes
sorted workComp
sort_index work_ids
spextra wtov
sqrt
StaticMenuDrvrMap
stdin_from
stdout_to
stdpt
std_dir
store
strsubst
suspend
tan
tanh
tell
tellfunc
textbox
text-pane_attributes
time on/off
toend
tp_delete_selection
tp_find_string
tp_get_select
tp_insert
tp_return_select
# Cantor

## Index

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>46</td>
</tr>
<tr>
<td>%</td>
<td>30; 57</td>
</tr>
<tr>
<td>% (functionals)</td>
<td>57</td>
</tr>
<tr>
<td>%ast</td>
<td>10; 12; 15; 22; 25; 55</td>
</tr>
<tr>
<td>-*</td>
<td>10; 12; 22</td>
</tr>
<tr>
<td>**</td>
<td>10; 12; 22; 25; 55</td>
</tr>
<tr>
<td>-</td>
<td>10; 12; 22</td>
</tr>
<tr>
<td>-&gt;</td>
<td>35; 56</td>
</tr>
<tr>
<td>/=</td>
<td>11; 56</td>
</tr>
<tr>
<td>=</td>
<td>56</td>
</tr>
<tr>
<td>?</td>
<td>21; 47</td>
</tr>
<tr>
<td>abs</td>
<td>11; 13</td>
</tr>
<tr>
<td>Abstract Syntax Tree</td>
<td>38</td>
</tr>
<tr>
<td>Ada/Ed</td>
<td>2</td>
</tr>
<tr>
<td>allbreaktype</td>
<td>70</td>
</tr>
<tr>
<td>allocate</td>
<td>60; 63</td>
</tr>
<tr>
<td>analyze</td>
<td>40</td>
</tr>
<tr>
<td>and</td>
<td>14; 57</td>
</tr>
<tr>
<td>annotate</td>
<td>62</td>
</tr>
<tr>
<td>applyEnv</td>
<td>37</td>
</tr>
<tr>
<td>applyNilEnv</td>
<td>37</td>
</tr>
<tr>
<td>arb</td>
<td>23</td>
</tr>
<tr>
<td>assignment</td>
<td>46; 54</td>
</tr>
<tr>
<td>assignment (simultaneous-)</td>
<td>53</td>
</tr>
<tr>
<td>ast</td>
<td>38; 39</td>
</tr>
<tr>
<td>ast_kind</td>
<td>38</td>
</tr>
<tr>
<td>ast_subs</td>
<td>44</td>
</tr>
<tr>
<td>atoi</td>
<td>16</td>
</tr>
<tr>
<td>atom</td>
<td>19</td>
</tr>
<tr>
<td>ator</td>
<td>16</td>
</tr>
<tr>
<td>backslash</td>
<td>12; 16</td>
</tr>
<tr>
<td>Bignum</td>
<td>10</td>
</tr>
<tr>
<td>binary function</td>
<td>55</td>
</tr>
<tr>
<td>BNF</td>
<td>4; 44</td>
</tr>
<tr>
<td>Boolean</td>
<td>14; 31; 44</td>
</tr>
<tr>
<td>bound variables</td>
<td>51</td>
</tr>
<tr>
<td>breakpoint</td>
<td>61</td>
</tr>
<tr>
<td>breakpoints</td>
<td>69</td>
</tr>
<tr>
<td>breaktype</td>
<td>70</td>
</tr>
<tr>
<td>call (selector)</td>
<td>52</td>
</tr>
<tr>
<td>cantor.ini</td>
<td>8</td>
</tr>
<tr>
<td>cantor_AlphaNumSet</td>
<td>19</td>
</tr>
<tr>
<td>cardinality</td>
<td>23; 25; 26</td>
</tr>
<tr>
<td>ceil()</td>
<td>13</td>
</tr>
<tr>
<td>changes</td>
<td>61</td>
</tr>
<tr>
<td>char</td>
<td>11; 16</td>
</tr>
<tr>
<td>character-pairs</td>
<td>9</td>
</tr>
<tr>
<td>class</td>
<td>36</td>
</tr>
<tr>
<td>clear</td>
<td>60</td>
</tr>
<tr>
<td>close</td>
<td>20</td>
</tr>
<tr>
<td>Closure</td>
<td>33</td>
</tr>
<tr>
<td>code</td>
<td>61</td>
</tr>
<tr>
<td>codeOf</td>
<td>37</td>
</tr>
<tr>
<td>command</td>
<td>6</td>
</tr>
<tr>
<td>comment</td>
<td>9</td>
</tr>
<tr>
<td>compound operator</td>
<td>30</td>
</tr>
<tr>
<td>construct</td>
<td>40</td>
</tr>
<tr>
<td>conversion</td>
<td>16; 20</td>
</tr>
<tr>
<td>data-base</td>
<td>21</td>
</tr>
<tr>
<td>date</td>
<td>16</td>
</tr>
<tr>
<td>debugging</td>
<td>67</td>
</tr>
<tr>
<td>declaration</td>
<td>34</td>
</tr>
<tr>
<td>detachEnv</td>
<td>37</td>
</tr>
<tr>
<td>directives</td>
<td>5; 59</td>
</tr>
<tr>
<td>div</td>
<td>10; 56</td>
</tr>
<tr>
<td>do</td>
<td>46; 47</td>
</tr>
<tr>
<td>domain</td>
<td>27</td>
</tr>
<tr>
<td>double quotes</td>
<td>16</td>
</tr>
<tr>
<td>echo</td>
<td>61</td>
</tr>
<tr>
<td>else</td>
<td>46</td>
</tr>
<tr>
<td>elseif</td>
<td>46</td>
</tr>
<tr>
<td>end of file</td>
<td>20</td>
</tr>
<tr>
<td>environment</td>
<td>37</td>
</tr>
<tr>
<td>envOf</td>
<td>37</td>
</tr>
<tr>
<td>eof</td>
<td>20</td>
</tr>
<tr>
<td>error messages</td>
<td>68</td>
</tr>
<tr>
<td>eval</td>
<td>41</td>
</tr>
<tr>
<td>even</td>
<td>11</td>
</tr>
<tr>
<td>exists</td>
<td>31; 58</td>
</tr>
<tr>
<td>exit</td>
<td>60</td>
</tr>
<tr>
<td>expression</td>
<td>54</td>
</tr>
<tr>
<td>external file</td>
<td>21</td>
</tr>
<tr>
<td>Fatal Errors</td>
<td>68</td>
</tr>
<tr>
<td>fgets</td>
<td>21</td>
</tr>
<tr>
<td>File</td>
<td>20; 47</td>
</tr>
<tr>
<td>file size</td>
<td>21</td>
</tr>
<tr>
<td>findAst</td>
<td>41</td>
</tr>
<tr>
<td>fix</td>
<td>13</td>
</tr>
<tr>
<td>flen</td>
<td>21</td>
</tr>
<tr>
<td>flex_alloc</td>
<td>61</td>
</tr>
<tr>
<td>flex_min_size</td>
<td>61</td>
</tr>
<tr>
<td>float(</td>
<td>11</td>
</tr>
<tr>
<td>floating point</td>
<td>12</td>
</tr>
<tr>
<td>FLOATING_POINT</td>
<td>44</td>
</tr>
<tr>
<td>floor</td>
<td>13</td>
</tr>
<tr>
<td>for</td>
<td>46</td>
</tr>
<tr>
<td>forall</td>
<td>31; 58</td>
</tr>
<tr>
<td>format</td>
<td>50</td>
</tr>
<tr>
<td>former (set-, tuple-)</td>
<td>52</td>
</tr>
<tr>
<td>Formers</td>
<td>28</td>
</tr>
<tr>
<td>fread</td>
<td>21</td>
</tr>
<tr>
<td>from</td>
<td>47; 48</td>
</tr>
<tr>
<td>fromb</td>
<td>49</td>
</tr>
<tr>
<td>frome</td>
<td>48</td>
</tr>
<tr>
<td>fseek</td>
<td>21</td>
</tr>
<tr>
<td>ftell</td>
<td>21</td>
</tr>
<tr>
<td>func</td>
<td>33; 58</td>
</tr>
<tr>
<td>func syntax</td>
<td>34</td>
</tr>
<tr>
<td>fwrite</td>
<td>21</td>
</tr>
<tr>
<td>garbage collection</td>
<td>61</td>
</tr>
<tr>
<td>Gary Levin</td>
<td>3</td>
</tr>
<tr>
<td>gc</td>
<td>61</td>
</tr>
<tr>
<td>global</td>
<td>34</td>
</tr>
<tr>
<td>grammar</td>
<td>44</td>
</tr>
<tr>
<td>hash</td>
<td>16</td>
</tr>
<tr>
<td>hasNilEnv</td>
<td>37</td>
</tr>
<tr>
<td>help</td>
<td>6</td>
</tr>
<tr>
<td>icon</td>
<td>4</td>
</tr>
<tr>
<td>ID</td>
<td>44</td>
</tr>
<tr>
<td>identifier</td>
<td>9</td>
</tr>
<tr>
<td>ids</td>
<td>61</td>
</tr>
<tr>
<td>if</td>
<td>46</td>
</tr>
<tr>
<td>iff</td>
<td>14; 57</td>
</tr>
<tr>
<td>ignoreallbreakpt</td>
<td>70</td>
</tr>
<tr>
<td>image</td>
<td>28</td>
</tr>
<tr>
<td>image set</td>
<td>27</td>
</tr>
<tr>
<td>impl</td>
<td>14; 57</td>
</tr>
<tr>
<td>in</td>
<td>15; 23; 26; 57</td>
</tr>
<tr>
<td>include</td>
<td>6; 8; 60</td>
</tr>
<tr>
<td>include file</td>
<td>7</td>
</tr>
<tr>
<td>inequality</td>
<td>56</td>
</tr>
<tr>
<td>INF</td>
<td>14</td>
</tr>
<tr>
<td>infix operator</td>
<td>55</td>
</tr>
<tr>
<td>Integer</td>
<td>10; 44</td>
</tr>
<tr>
<td>interp</td>
<td>41</td>
</tr>
<tr>
<td>iSETL</td>
<td>3</td>
</tr>
<tr>
<td>is_ast</td>
<td>38</td>
</tr>
<tr>
<td>is_ast_leaf</td>
<td>40</td>
</tr>
<tr>
<td>is_atom</td>
<td>19</td>
</tr>
<tr>
<td>is_bignum</td>
<td>10</td>
</tr>
<tr>
<td>is_boolean</td>
<td>14</td>
</tr>
<tr>
<td>is_defined</td>
<td>21</td>
</tr>
<tr>
<td>is_file</td>
<td>20</td>
</tr>
<tr>
<td>is_floating</td>
<td>12</td>
</tr>
<tr>
<td>is_func</td>
<td>33</td>
</tr>
<tr>
<td>is_integer</td>
<td>10</td>
</tr>
<tr>
<td>is_map</td>
<td>27</td>
</tr>
<tr>
<td>is_number</td>
<td>11; 13</td>
</tr>
<tr>
<td>is_OM</td>
<td>21</td>
</tr>
<tr>
<td>is_set</td>
<td>22; 27</td>
</tr>
<tr>
<td>is_string</td>
<td>15</td>
</tr>
<tr>
<td>is_tuple</td>
<td>24</td>
</tr>
<tr>
<td>iterator</td>
<td>51</td>
</tr>
<tr>
<td>itoa</td>
<td>16</td>
</tr>
<tr>
<td>itoa</td>
<td>20</td>
</tr>
<tr>
<td>keywords</td>
<td>9; 41</td>
</tr>
<tr>
<td>kwd</td>
<td>41</td>
</tr>
<tr>
<td>l-expression</td>
<td>34</td>
</tr>
<tr>
<td>lc</td>
<td>21</td>
</tr>
<tr>
<td>less</td>
<td>23; 56</td>
</tr>
<tr>
<td>line count</td>
<td>21</td>
</tr>
<tr>
<td>load</td>
<td>6; 8; 60</td>
</tr>
<tr>
<td>local</td>
<td>34; 59</td>
</tr>
<tr>
<td>logarithms</td>
<td>13</td>
</tr>
<tr>
<td>lower-case</td>
<td>18</td>
</tr>
</tbody>
</table>
map 27; 32
match 43
max 11; 13
memory 60; 63
min 11; 13
mkLocal 37
mmap 28; 52
mod 10; 56
modes 7
nested mode 7
newat 19
new_ids 61
noop 70
not 14
notin 23; 26; 57
npow 24
npow2 24
odd 11
OM 21
oms 61
opena 20
openab 20
openr 20
openrb 20
openrw 20
openrw 21
openw 21
openwb 21
operator 55
opt 59
or 14; 57
ord 11; 16
overrideOf 37
package 36
pair 28
pane file 21
parse_msg 43
parsing 40
passive_err 62
pattern matching 43
persistence 9
pointer 35
polymorphic 14; 26
pow 23
precision 14; 50
Predef 33
pretty 41
print 47
printf 48; 49
program 45
prompt 5
quantifiers 31
quit 6; 60
random 11; 13; 15; 16; 24
randomize 11; 13
range 28
rank 16; 26
RAPTS 2
read 47
read mode 7
readf 47; 49
real 12
record 60; 64
recordOutput 6; 61
ref 37
replicate 15
reset 60
restart 60
return 34
rewind 21
round 13
rtoa 16
runtime error 68
save 6; 60
scan 16; 17; 41
scope 35; 37
SED 2
separator 19
set 22; 32
set formers 22
setallbreakpt 70
setAst 41
setBaseAtom 20
SETL 2
setScanStop 16
sgn 11; 13
short cuts 6
simple data type 54
simple inheritance 36
single quotes 16
size 24
slice 53
slices 22; 25
smap 28; 52
special characters 9; 17
sqrt 13
standard mode 7
String 15; 44
strsubst 16
subset 23; 57
substring 15
suspend 6; 60
take 23; 26; 48
tellfunc 37; 68
terminator 17
then 46
this 36
time 62
to 48
toend 21
token 17
trace 61
trigonometry 13
tuple 24; 32