Cantor: a Tutorial and a User's Guide

(prototyping, set theory and all that)

Jean-Pierre Keller

volume II

Kepler *8 rue des haies, F-75020 Paris*

dec. 1994

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

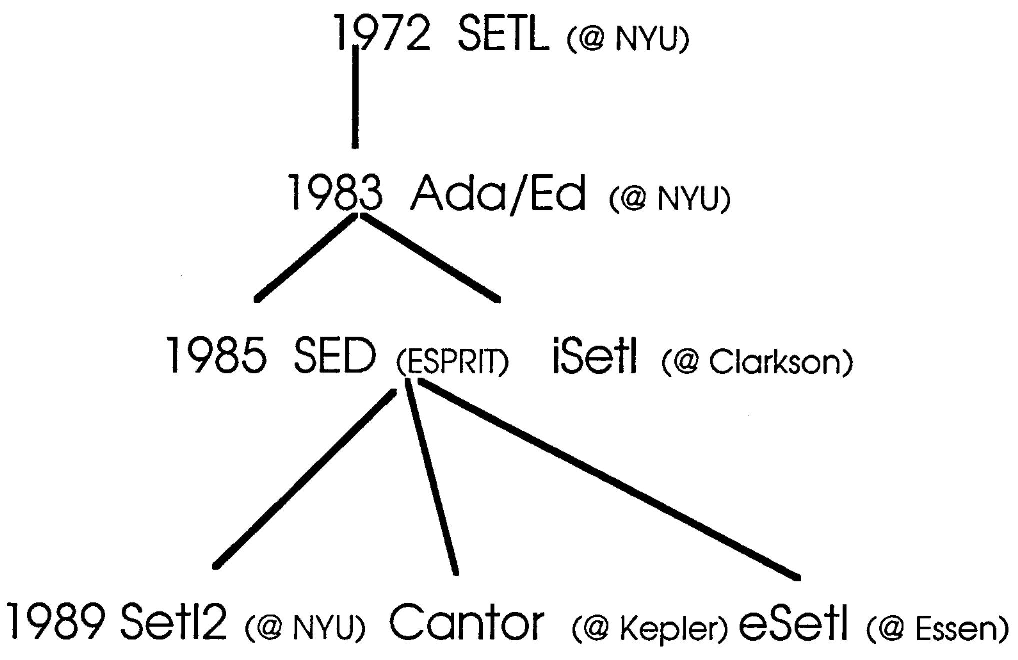
Volume II : a Cantor User's Guide

Jean-Pierre Keller, Kepler

8 rue des haies, F-75020 Paris

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.



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 and 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[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 developping 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[2]
- automatically, SETL code could be translated into a common programming language (Ada was used as a target language, C would have been simpler)[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[4].
- complex semantic properties (e.g. the type of an object) could be infered from a declaration-free algorithm (i.e. declarations may be useless in a prototyping environment)[5]

[1]Ed stands for 'Educational'. But many believe that this is just the first name of its Principal Investigator, Ed Schonberg, from NYU.

[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 developped an algorithm to reduce by a factor 10 to 30 the number of interactive operations left to the human operator.

[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!".

[4]RAPTS was developped 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.

[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 recommandations, 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 traditionnal objects like:
Abstract-Syntax Trees (AST),
and also
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[6] and (dynamically constructable) lambda-expressions, modularity, objects, classes, inheritance,
2-mechanisms to save (on a file) and restore objects of an arbitrary complexity (possibly including executable functions),
3-mechanisms for analyzing and transforming programs,
4-(interactive) graphics and text objects for multiwindow menu-oriented applications with their event processing mechanisms,
5-support to the 2-way interoperability with C-oriented applications.

The following comments will clarify these points:

1,2-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.

3-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.

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

[6]a 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 basicaly 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[7]

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

[7]This 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

a Cantor created Pict image

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)
Macintosh version (68000) Kepler corp.1989,1990,1991,1992,1993
Copyright 1989,1990,1991,1992 (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:

```
!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 <u>Cantor directive</u> -in this example the help directive. Most directives may be obtained as interactive <u>command</u> 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:

interactive menu short cuts[8]	full command	
cmd-H	!help <pattern>	list all cantor primitives whose names match <pattern>, together with simple help information
cmd-I	!include <filename>	include (read in and compile) the text file <filename>
cmd-L	!load <filename>	load the binary file <filename>
cmd-Q	!quit	quit Cantor
cmd-R	!recordOutput <filename>	record all output to the console in the file <filename>
cmd-S	!save <filename>	save the session into the binary file <filename>
cmd-W	!suspend	suspend current Cantor session, and return to the user with a prompt

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 :

```
> !help window

$ ..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

```
$ ..4
win_nbr := open_old_window( x, y, w, h );
$
..5
curwin_nbr := window(win_nbr);
$ ..6
curwin_nbr :=
set_window(win_nbr);
$ ..7
window_attributes();$ the window
attribute map structure
$
..8
set_window_attributes(anAttrMap);
$ ..9
anAttrMap :=
get_window_attributes(anAttrSet);
$ ..10
clipwindow(region);
$
..11
win_no := get_file_window(pane_file); $--> return -1 if
pane_file is a disk file
;
> !help clock

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

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;
```

```
? 'ask';
? 20;
> x;
"ask";
> y;
20;
>
```

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 automaticaaly 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	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); $ ...
```

```
bool := is_atom(x); $ ...
bool := is_boolean(x); $ ...

bool := is_om(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`
		`> 2 ** 10;`
		`1024;`

x div y	integer division of x and y	`>7 div 4;`
		`1;`

x / y	real-precision division of x and y	`> 7 / 4;`
		`1.750;`

x mod y	remainder in the division of x by y	`> 7 mod 4;`
		`3;`

predicates on integers

all predicates are expressions evaluating to either true of false

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	`> even(65);`
		`false;`

odd(x)	test if x is odd	`> odd(65);`
		`true;`

is_number(x)	true if x is an integer or a real	`> is_number(10);`
		`true;`

some primitive functions on integers

abs(x)	absolute value of x	`> abs(-31);`
		`31;`

ord(char)	n := ord(char); $ integer value of a character	`> ord('a');`
		`97;`
		`> ord('b');`
		`98;`
		`> ord(char(100));`
		`100;`

char(n)	s := char(n); $ (ascii) char value of an integer	`> char(97);`
		`"a";`
		`> char(ord('z'));`
		`"z";`

float(n)	convert n to a real number	`> float(65);`
		`65.000;`

random(root)	generate a random integer in the range (0..root) when root is an integer > 0	`> random(500);`
		`492;`
		`> random(500);`
		`268;`
		`> random(-500);`
		`OM;`

randomize (seed)	re-initialize the random number generator with a new seed	

sgn(n)	sign of n	`> sgn(-15);`
		`-1;`
		`> sgn(0);`
		`0;`
		`> sgn(74);`
		`1;`

max(x,y)	the largest of x and y	
min(x,y)	the smallest of x and y	

notes

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;
112050755800644139182824657874288503316182344\
```

[9]No practical limit. Actually limited to about 15,000 digits per integer on a Macintosh

```
836201072566418066442575170654489604988455473\
085891233152722255158215820835509118567770425\
555664949954615083500304129450159283620378895\
008790288025331140066449564826484508657579315\
925606917480955013780196392370141851418465252\
049263944145260911871147445328203745168510368\
854915637280099588264866194322947975660549095\
765165693992960000000000000000000000000000000\
0000000000000000000000;
>
```

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

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

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

3.2 Real (Floating_Point) Numbers

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

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

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

```
"Real"
```

this is illustrated here :

```
> 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`	`> 2 ** 10.5;`
	`elseif y < 0 then om`	`1448.155;`
	`else x * (x **(y-1))?1`	`> 2 ** 10.5 = 1024*sqrt(2);`
	`end`	`true;`
x / y	real-precisiondivision of x and y	`> 7.8/19.1e-2;`
		`40.838;`

`predicates on floating point numbers (real numbers)`

all predicates are expressions evaluating to either true of false

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	
is_number(x)	true if x is an integer or a real	`> is_number(10.486);`
		`true;`

some primitive functions on floating point numbers (real numbers)

abs(x)	absolute value of x	`> abs(7.8/19.1e-2);`
		`40.838;`
		`> abs(-7.8/19.1e-2);`
		`40.838;`
		`>`
ceil(x)	n := ceil(real); $ int. approx. of a real, see	`> ceil(13.7);`
	also floor, fix, round	`14;`
		`> ceil(-13.7);`
		`-13;`
fix(x)	n := fix(real); $ int. approx. of a real, see	`> fix(13.7);`
	also ceil,floor, round	`13;`
		`> fix(-13.7);`
		`-13;`
floor(x)	n := floor(real); $ int. approx. of a real,	`> floor(13.7);`
	see also ceil,fix, round	`13;`
		`> floor(-13.7);`
		`-14;`
round(x)	n := round(real); $ int. approx. of a real,	`> round(13.7);`
	see also ceil,fix, floor	`14;`
		`> round(-13.7);`
		`-14;`
random(root)	generate a random real in the range	`> random(500.0);`
	(0..root) when root is a real > 0	`256.927;`
		`> random(500.0);`
		`87.860;`
		`> random(-500.0);`
		`OM;`
randomize	re-initialize the random number generator	
(seed)	with a new real seed	
sgn(x)	sign of real expression x	`> sgn(-1.2);`
		`-1.000;`
		`> sgn(0.0);`
		`0.000;`
		`> sgn(1.2);`
		`1.000;`
max(x,y)	the largest of x and y	
min(x,y)	the smallest of x and y	
trigonometry	the trigonometric functions:	`> sin(3.14);`
	cos, sin, tan, acos, asin, atan, cosh, sinh,	`0.002;`
	tanh, acosh, asinh, atanh	`> acos(-1);`
		`3.142;`
sqrt(x)	the square root of x, equivalent to x **0.5	
logarithms	the ln (neperian log),log (base 10 log),	
	exp (neperian exponentiation)	

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 103 .

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;
> 1.0e+125 **100;
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	equality of u and v	
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	`> random(true);`
		`false;`
		`> random(true);`
		`true;`

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

s(i)	extracts the i-th character of s	`> '123456789'(4);`
		`"4";`
		`> '123456789'(10);`
		`OM;`

s(i..j)	substring containing all chars from i-th to the j-th	`> '123456789'(4..6);`
		`"456";`

s(..j)	substring containing all the chars in s until the j-th	`> '123456789'(..6);`
		`"123456";`

s(i..)	substring containing all the chars in s starting with the i-th	`> '123456789'(4..);`
		`"456789";`

s*i
or
i*s
	repeat -i.e. replicate - s i times . When i is 0, returns the empty string	`> 'to be'*3;`
		`"to beto beto be";`
		`> 3*'to be';`
		`"to beto beto be";`
		`> 'to be'*0;`
		`"";`

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	`> 'o b' in ' or not to be...';`
		`true;`
		`> 'ob' in ' or not to be...';`
		`false;`

some primitive functions on strings

random(s)	extracts randomly a character from the string s	`> random('123456789');`
		`"9";`
		`> random('123456789');`
		`"2";`
		`> random('123456789');`
		`"7";`

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	`> rank('ia','miam-miam');`
		`2;`
		`> rank('foo','miam-miam');`
		`0;`

ator	real := ator(floatingNbrString);	`> ator('1.5');`
		`1.500;`

rtoa	str := rtoa(realNbr);	`> rtoa(1.0e-1);`
		`"0.100";`

atoi	n := atoi(nbrAsString);	`> atoi('125');`
		`125;`

itoa	str := itoa(n);$ integer (or atom-value) to string conversion	`> itoa(2*125);`
		`"250";`

ord	n := ord(char); $ integer value of a character	`> ord(char(100));`
		`100;`

char	s := char(s); $ (ascii) char value of an integer	`> char(ord('z'));`
		`"z";`

hash	int = hash(x) $ hash value	`> hash('abc');`
		`489;`

date	str := date(); $ current date, with the precision of a second	`> date();`
		`"Mon Oct 24 15:43:59 1994\n";`

uclcase	s := uclcase('(Uu)|(Ll)',string); $ convert string into upper (resp. lower) case	`> uclcase('u',date());`
		`"MON OCT 24 15:44:05 1994\n";`
		`> uclcase('l',date());`
		`"mon oct 24 15:44:18 1994\n";`

strsubst	string := strsubst(pat, string,by);$replace all occurrences of pat in string with by	`> strsubst(' ','too foo is she?','_-_');`
		`"too_-_foo_-_is_-_she?";`

scan	token_stream := scan(File|fileName|om|string, textScan|textAndNumScan,strScan);	`> scan(" on dec 14, it rains",1,1);`
		`["on","dec",14,",","it","rains"];`

setScanStop	aChar := setScanStop(aChar); $aChar becomes the new scan stop char; default ScanStop is '¿'	`> setScanStop('£'); $ change the terminator char to '£'`
		`"£";`

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 formated output, the special characters are printed.

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

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

```
> %+ [char(i): i in [1..127]];

"\001\002\003\004\005\006\007\b\t\n\013\f"
+"\r\016\017\020\021\022\023\024\025\026"
+"\027\030\031\032\033\034\035\036\037 !"
+"\q#$%&'()*+,-./0123456789:;<=>?@ABCDEF"
+"GHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijk"
+"lmnopqrstuvwxyz{|}~\177";
```

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	*description*
1st arg is a File	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
1st arg is a String (3rd arg = om)	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 successfull, process it until the end of file or the terminator
1st arg is a String (3rd arg ≠ om)	the first argument is the input text upon which scan will operate, if the third arg is different from OM
1st arg is om	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 '¿')

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:

```
> s := "abrac cada bra";
> scan(s,1,1);
["abrac", "cada", "bra"];
> s := "abrac ca@da bra";
> scan(s,1,1);
["abrac", "ca"];
> setScanStop('.');
".";
> s := "abrac ca¿da bra";
> scan(s,1,1);
["abrac", "ca", "da", "bra"];
> scan("-1 .0675",1,1);
["-", 1];
> setScanStop('¿');
"¿";
> scan(om,om,om);
turlutut chapeau pointu!
¿
["turlutut", "chapeau", "pointu", "!"];
```

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);
["a", "beautiful", "house"];
> s := "'a beautiful house'";
> scan(s,1,1);
["'", "a", "beautiful", "house", "'"];
```

One needs a different symbol, a kind of 'super'-quote, which will not be ignored by the scan. That symbol is '§':

```
> s := "he sings §a beautiful house§ while riding ";
> scan(s,1,1);
["he", "sings", "§a beautiful house§", "while", "riding"];
```

The second argument is used as a control switch:

2nd arg	descritpion	example
'l','L'	the text is to be output in lower-case letters	`> scan('New York is a big CiTy','L',1);` `> ["new", "york", "is", "a", "big", "city"];`
'u','U'	the text is to be output in upper-case letters	`> scan('New York is a big CiTy','u',1);` `["NEW", "YORK", "IS", "A", "BIG", "CITY"];`
om, ""	the text is to be output rescpecting the input case	`scan('New York is a big CiTy',om,1);` `["New", "York", "is", "a", "big", "CiTy"];`
1	keep the settings used in the previous scan	`> scan('New York is a big CiTy','L',1);` `> ["new", "york", "is", "a", "big", "city"];` `> scan('New York is a big CiTy',1,1);` `> ["new", "york", "is", "a", "big", "city"];`
" ", "ab"	at least two characters: keep correct number formats	`> scan("1.035",1,1);` `[1.035];` `> scan("-1.035",1,1);` `["-", 1.035];` `> scan("-1.035","  ",1);` `[-1.035];`

"U ","Ux"
"l ","ly"
	combine settings for upper or lower case and number formatting	`> s := " a is -1.027 grams";`
		`> scan(s+"","U ",1);`
		`["A", "IS", -1.027, "GRAMS"];`
		`> t := "b is -9.57 ";`
		`> scan(t+"",1,1);`
		`["B", "IS", "-", 9.570];`

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:

```
> 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:

```
> 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:

```
> 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:

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

3.5 Atoms

A cantor expression at is of type atom if

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

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

```
"Atom"
```

```
operations on atoms
```

newat	atom creation. This atom is unique	`> s := newat;`
		`> t := newat;`
		`> type(s) = type(t);`
		`true;`
		`> s = t;`
		`false;`

```
predicates on atoms
```

s=t	equality of s and t
s /= t	inequality of s and t

```
some primitive functions on atoms
```

itoat	atom(resp. int) := itoat(int(resp. atom));$ int (resp atom-value) to atom-value (resp int) conversion
	
	if there is no atom corresponding to the given integer, returns an error
		`> at2;`
		`!5!;`
		`> itoat(at2);`
		`5;`
		`> itoat(7);`
		`!break point: WARNING_bkPt!`
		`;`
		`%`

setBaseAtom	setBaseAtom(atom|om); $ set Base_Atom to atom | !0!	

```
notes
```

1. Atoms are *abstract points*. They have no identifying properties other than their individual existence[10]. The only operation on atoms is comparing two atoms for identity.

2. The keyword *newat* represents a constructor, acting as a function. *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
```
  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 nam be a string

```
predicates on files
```

f=g	equality of s and t
f /= g	inequality of s and t
eof(f)	true if file pointer is at the end of file

```
some primitive functions on files
```

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

[10]In the current version atoms may be saved and restored: their uniqueness may not be garanteed accross sessions. The function setBaseAtom may be used to correct this situation.

openrwb(na m)	f := openrwb(nam); $open read-write binary file nam
openw(nam)	f := openw(nam); $open write text file nam
openwb(nam)	f := openwb(nam); $open write binary file nam
fwrite	fwrite(item,f);$ item type:integer, string, bignum
fread	value_read := fread (File, 'int'|'str'|'big' , count); $ item type: integer, string, bignum count: always 1 for int
fseek	fseek(f,f_position); $ move file pointer to given file position
ftell	f_position := ftell(f); $ returns current file position
rewind	rewind(f); $ set file position to the begining of the file
toend	toend(f);$ set file position to the end of the file
flen	n := flen(f|fileName); $ file size
lc	n := lc(f|fileName|om); $ text file line count
fgets	string := fgets(n,f); $read a line of at most n char;

notes

1. A file is generally a Cantor value that corresponds to an *external file* pointer in the operating system environment. There are however two other kinds of files: the *pane file* (corresponding to the data streams in a Cantor text-oriented window) and the *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 *openr, opena, openw, openrw* for text files and *openrb, openab, openwb, openrwb* for non-text (binary) files.

3. Pane files are created by applying the predefined functions *open_pane_file*

4. Databases are created by the *save, store* or *compile* predefined functions.

3.7 Undefined

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

operations on Undefined

Let x, y be two arbitrary expressions

x?y	this expression has value x if x /= om, otherwise, has value y

predicates on Undefined

x=om	true if is_om(x) = true
x /= om	true if is_om(x) = false
is_defined(x)	true if is_om(x) = false

notes

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

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

4 Compound Data Types

4.1 Sets

A cantor expression x is of type set if

```
   is_set(x) = true;
```

holds. Actually, in that case, 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 { }

```
>{1,'man','ape',2,45.75,
>> newat, {}};
{!7!, 45.750, "ape", "man", 2, {},
1};
```

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 |i| ≤ |j|. Slices are arithmetic progressions

```
> {1..5};
{2, 1, 3, 4, 5};
> {-3..2};
{1, 2, 0, -1, -2, -3};
> {0,5..25};
{10, 15, 25, 20, 0, 5};
> {0,5..26};
{15, 10, 20, 25, 0, 5};
```

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 | K(t)}
or given a subcollection, as a derived set of expressions:
{exprn(t): t in x | K(t)}
the syntax supports very complex set formers.(see § 4.4)

```
> {t: t in {1..15} | t mod 7
>> = 2};
{2, 9};
> { x+y : x,y in {-1,-3..-
>> 10} | x /= y };
{-8, -10, -4, -6, -14, -16, -12};
```

`operations on sets`

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

x+y
or
x union y

union of x and y

```
> {1} union {"a"};
{1, "a"};
> {1} + {"a"};
{1, "a"};
```

x-y

set difference of x and y

```
> {1,'a',2,'b'} - {'a','b'};
{1, 2};
```

x * y
or
x inter y

intersection of s and y

```
> {1,'a',2,'b'} inter {'a','b'};
{"a", "b"};
> {1,'a',2,'b'} * {'a','b'};
{"a", "b"};
```

x with t	form a new set by adding element t to the set x	`> {1,'a',2,'b'} with 'c';`
		`{1, 2, "c", "b", "a"};`
		`> {1,'a',2,'b'} with 'a' =`
		`{1,'a',2,'b'};`
		`true;`

x less t	form a new set by removing t from x	`> {1,'a',2,'b'} less 'a';`
		`{"b", 2, 1};`
		`> {1,'a',2,'b'} less 'c' =`
		`{1,'a',2,'b'};`
		`true;`

#x	cardinality of x	`> #{-1,-3..-100};`
		`50;`
		`> #{ x+y : x,y in {-1,-3..-10} | x`
		`/= y };`
		`7;`

take t from x	remove an arbitrary element of x and assign it to variable t
	observe that x has been changed
		`> x := {1, 2, "c", "b",`
		`>> "a"};`
		`> x;`
		`{"a", "b", "c", 2, 1};`
		`> take t from x;`
		`> t;`
		`"a";`
		`> x;`
		`{2, 1, "c", "b"};`

`predicates on sets`

x subset y	true if x is a subset of y, i.e. , all the members of x are members of y too	`>{1,2,3} subset {1..5};`
		`true;`
		`> {1,2,3} subset {1,2,'a'};`
		`false;`
		`>{1,2,3} subset {1,2,'a'} with 3;`
		`true;`

t in x	true if t is a member of x	`> {} in {'a','b',{}};`
		`true;`

t notin x	true if t is not a member of x	`> 'a' in {1,2, 3};`
		`false;`
		`> 'a' notin {1,2, 3};`
		`true;`
		`> 1 notin {};`
		`true;`

`some primitive functions on sets`

arb(s)	if s is non-empty returns an 'arbitrary' element from s, otherwise returns om	`> s := {1..50};`
		`> arb(s);`
		`16;`
		`> arb(s);`
		`27;`

pow(s)	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
		`> s := {1..10};`
		`> c1 := clock(); t12 :=`
		`>> npow2(s,#s-1);c2 :=`
		`>> clock();`
		`> #t12;`
		`10;`
		`> $ comput. time in sec`
		`> (c2-c1)/60;`
		`0.367;`

random(s)	extracts randomly a member from the set	`> s := {1..50};`
		`> random(s);`
		`26;`
		`> random(s);`
		`9;`

size(s)	n := size(s); $ size in byte of s and its dependants	`> s :={1..10};`
		`> t12 := npow2(s,2);`
		`> #t12;`
		`45;`
		`> size(s);`
		`864;`
		`> size(t12);`
		`6384;`

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
```
   is_tuple(x) = 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 []	`> [1,'man','ape',2,45.75,`
		`>> newat, {}];`
		`[1, "man", "ape", 2, 45.750, !8!,`
		`{}];`

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 $\lvert i\rvert \le \lvert j\rvert$, in that order.
	Slices are arithmetic progressions
		`> [1..5];`
		`[1, 2, 3, 4, 5];`
		`> [-3..2];`
		`[-3, -2, -1, 0, 1, 2];`
		`> [-1,-3..-10];`
		`[-1, -3, -5, -7, -9];`

tuple formers (in comprehen-sion)	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 | K(t)]
	or given a subcollection, as a derived tuple of expressions:
	[exprn(t): t in x | K(t)]
	the syntax supports very complex tuple formers. (see § 4.4)
		`> [t: t in {1..15} | t mod 7`
		`>> = 2];`
		` [2, 9];`
		`> [ x+y : x,y in [-1,-3..-`
		`>> 10] | x /= y ];`
		` [-4, -6, -8, -10, -4, -8, -10, -12,`
		`-6, -8, -12, -14, -8, -10,`
		` -12, -16, -10, -12, -14, -16];`

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	`> [2, 9]+[-1,-3..-10];`
		`[2, 9, -1, -3, -5, -7, -9];`

x*i
or
i*x
	new tuple obtained by the replication i times of tuple x	`> [2,9]*3;`
		`[2, 9, 2, 9, 2, 9];`
		`> 3*[2,9];`
		`[2, 9, 2, 9, 2, 9];`

x with t	form a new tuple by adding element t to the tuple x, as last element	`> [2,9] with 100;`
		`[2, 9, 100];`

x(i)	the i-the element of tuple x	`> [2,4..10](3);`
		`6;`

x(i..j)	form a new tuple made of all elements from the i-th to the j-th included	`> [2,4..10](2..4);`
		`[4, 6, 8];`

x(..j)	form a new tuple made of all elements from the 1st to the j-th included	`> [2,4..10](..4);`
		`[2, 4, 6, 8];`

x(i..)	form a new tuple made of all elements from the i-th to the last one	`> [2,4..10](2..);`
		`[4, 6, 8, 10];`

#x	cardinality of x	`> #[2,4..10];`
		`5;`

take t fromb 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;`
		`2;`
		`> x;`
		`[4, 6, 8, 10];`

take t frome 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;`
		`10;`
		`> x;`
		`[2, 4, 6, 8];`

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 notin [2,4..10];`
		`true;`

some primitive functions on tuples

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	`> rank(8,[2,4..10]);`
		`4;`
		`> [2,4..10](rank(8,[2,4..10])) = 8;`
		`true;`
		`> 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]   a tuple of all positive odd integers less than 100
t := [OM,'a string',10,{1..20},'another string',OM,[]];
   a tuple of length 7. t(4) is the set of all integers ranging from 1 to 20. t(#t) is the empty tuple. For any integer i>7 t(i) 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 implentation instance requires for representing s and its dependants.

4.3 Maps

A cantor expression x is of type map if

```
is_map(x) = true;
```

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

```
"Map"
```

Maps form actually a subclass of sets. Thus,

```
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 := {['+','binary op'],
>> ['-','binary op'], ['-',
>> 'unary >> op']};
> m{'-'};
{"unary op", "binary op"};
```

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('+');
"binary op";
> m('-');

!  Error  --  Bad  mapping(multiple
images):
{!Set!}("-");
```

m{x} := aSet

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

```
> m{'%'} := {"unary op",
>> "binary op"};
> m;
{["%",  "binary  op"],  ["%",  "unary
op"], ["+", "binary op"],
  ["-",  "unary  op"],  ["-",  "binary
op"]};
> m{'-'} := {};
> m;
{["%",  "binary  op"],  ["%",  "unary
op"], ["+", "binary op"]};
```

m(x) := y

(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]

```
> s := {[1,2],[1,3],[2,4]};
> s{1};
{2, 3};
> s(1) := 5;
> s;
{[1, 5],[2,4]};
```

```
some primitive functions on maps
```

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

```
> m;
{["-",  "unary  op"],  ["-",  "binary
op"], ["+", "binary op"]};
> domain(m);
{"+", "-"};
```

range(m) or image(m)	it is by definition: {y: [x,y] in m}	`> range(m);` `{"binary op", "unary op"};`

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 undefined 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 $c \neq b$.

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-expr1, as in {1,-3..-10}
FORMER --> EXPR : ITERATOR
e.g. #x: x in s | 'a' notin x *or* x+y: x in s,y in t | x > y**2

The syntax for ITERATOR is extremely versatile:

ITERATOR --> ITER-LIST
ITERATOR --> ITER-LIST | 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 := {[i,j] : i,j in [1..5] | i < j};
> 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, "e"], [6, "e"], [10, "e"]];

> [sentence(i..j): c=sentence(i), j in [i.. #sentence] | c = 'e'];
["e", "ex", "exe", "exem", "exemp",
 "exempl", "exemple", "e", "em", "emp",
 "empl", "emple", "e"];

> po := { [1,2], [1,3], [2,4], [2,5], [3,5],
>> [3,6], [4,8], [5,7], [6,7], [7,8] };
> op := {[x,y]: [y,x] in po};
> op;
{[6, 3], [5, 2], [5, 3], [7, 6],
 [7, 5], [8, 4], [8, 7], [4, 2],
 [3, 1], [2, 1]};


> domain(op) = {x: [x,~] in op};
true;

> image(op) = {x: [~,x] in op};
true;

> op_graph := { [y, x] : x=op{y} };
op_graph;
{[2, {1}], [3, {1}], [5, {3, 2}],
 [4, {2}], [7, {6, 5}], [8, {4, 7}],
 [6, {3}]};
```

4.5 Compound operators

Let us consider an operation op
  op: A x B -> A' where A' $\subseteq$ A
An operation like this could be one of the built-in binary operation, e.g.
  +,*,**,/,div,mod
or any (built-in or user defined) 2-ary function f: A x B -> A', e.g.
  max, min, 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, Str that of strings, e.g.
  tdiv: N x N -> N
  /: (R+N) x (R+N) -> R where R $\subseteq$ R+N
  npow: S x N -> 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 op b_1 op b_2 op .... op b_n
is well-defined and may be written, in Cantor as a compound operator, signaled by the % (percent) sign:
  a %op [b_1,b_2, ..., b_n]
or
  %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 %op {b_1,b_2, ..., b_n}
  %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 %op {b_1,b_2, ..., b_n}* or *%op {a,b_1,b_2, ..., b_n}* may yield different results if the commutativity properties of the operation are not garanteed. If however

op is a well-defined and commutative operation A x A -> A, than *a %op {b_1,b_2, ..., b_n}* may be written *%op {a,b_1,b_2, ..., b_n}*. In that case, the left most term in *a op b_1 op b_2 op .... 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 ∃) and forall (corresponding to∀)

EXPR --> exists ITER-LIST I 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.*

```
> p := [1..100];
> exists j in p | j < 0;
false;
> exists j in p, i in [2..j] | 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 --> forall ITER-LIST I 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.*

```
> primes := [i: i in [2..1000] | forall j in [2..floor(sqrt(i))]
>> | i mod j /= 0];
> primes;
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257,
 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331,
 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563,
 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709,
 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877,
 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967,
 971, 977, 983, 991, 997];
```

4.7 Sample session

The following is a self-explanatory demonstration of bulk structures, i.e. entities of type set, tuple and map:

```
> $ basic bulk structures are: set, maps, tuples,
> $ a holly trinity

> $ a set
> {1,5..100};
{69, 65, 77, 73, 85, 81, 89, 93, 97, 1, 5, 9, 13, 29, 25, 21,
 17, 61, 57, 53, 49, 37, 33, 41, 45};

> type({1,5..100});
"Set";

> $ this is an unordered structure
> {1,5..100};
{53, 49, 61, 57, 45, 41, 37, 33, 29, 25, 17, 21, 1, 5, 13, 9,
 69, 65, 73, 77, 85, 81, 93, 97, 89};


> $ a tuple, an ordered structure
> [1,5..100];
[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];

> type([1,5..100]);
"Tuple";


> t := [1,5..100];

> $ cardinality
> #t;
25;

> t(#t);
97;


> $ a tuple is an unbounded ordered structure
> t(30) := -4;
> #t;
30;

> t(#t);
-4;

> 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, -4];


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

> primes := [i: i in [2..1000] | forall j in [2..floor(sqrt(i))]
>> | i mod j /= 0];
> primes;
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257,
 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331,
 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563,
 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709,
 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877,
 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967,
 971, 977, 983, 991, 997];

> #primes;
168;

> $ bulk structures don't have to be homogeneous
> t := t with {-12..0};
> #t;
31;

> t(#t);
{-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, -4,
 {-6, -5, -7, -8, -12, -11, -10, -9, -4, -3, -1, 0, -2}];


> s := {t,{},'a sample string'};
> #s;
3;

> $ adding a member element does'nt change the set
> u := s with t;
> #u;
3;

> u=s;
true;


> $ maps are binary relations
> $ i.e. a set of pairs, a subset of a cartesian product

> aMap := {[1,'c'],[2,'a'],[3,'n'],[4,'t'],[5,'o'],[6,'r']};
> type(aMap);
"Map";

> aMap;
{[1, "c"], [2, "a"], [3, "n"], [4, "t"], [6, "r"], [5, "o"]};


> $ the sets involved in the cartesian product are
> domain(aMap);
{3, 4, 6, 5, 1, 2};

> image(aMap);
{"a", "c", "n", "o", "t", "r"};


> $ any cartesian product is a map
> u := {1..10};
> v := {'a','b','c'};
> uXv := {[x,y]: x in u,y in v};
> #uXv;
30;

> uXv;
{[10, "b"], [10, "c"], [10, "a"], [9, "b"], [9, "c"], [9, "a"],
 [7, "b"], [7, "c"], [7, "a"], [8, "a"], [8, "c"], [8, "b"],
 [5, "c"], [5, "b"], [5, "a"], [6, "b"], [6, "c"], [6, "a"],
 [1, "b"], [1, "c"], [1, "a"], [2, "a"], [2, "b"], [2, "c"],
 [3, "a"], [3, "b"], [3, "c"], [4, "a"], [4, "c"], [4, "b"]};

> 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};
{"r"};
```

```
> uelt := arb(domain(uXv));
> uelt;
7;

> uXv{uelt};
{"c", "a", "b"};


> $ single-valued map (smap)
> R := aMap;
> forall x in domain(R) | #R{x} = 1;
true;

> R := uXv;
> forall x in domain(R) | #R{x} = 1;
false;


$ multiple-valued map (mmap)
> R := aMap;
> exists x in domain(R) | #R{x} /= 1;
false;

> R := uXv;
> exists x in domain(R) | #R{x} /= 1;
true;
```

```
> $ single-valued map are functions defined over their domain
> aMap(elt);
"r";


> uXv(uelt);
! Error -- Bad mapping(multiple images):
{!Set!}(7);


> $ a tuple t has the semantics of a function defined over {1..#t}
> 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, -4,
 {-4, -3, 0, -1, -2, -10, -9, -11, -12, -5, -6, -7, -8}];

> #t;
31;

> domain(t);
{21, 22, 24, 23, 25, 31, 30, 17, 18, 19, 20, 11, 12, 9, 10, 13,
 14, 16, 15, 2, 1, 3, 4, 7, 8, 6, 5};

> image(t);
{21, 17, 25, 29, 13, 9, 5, 1, -4,
 {-3, -4, -1, 0, -2, -10, -9, -12, -11, -7, -8, -6, -5}, 33, 37,
 41, 45, 61, 57, 49, 53, 85, 81, 97, 93, 89, 73, 77, 69, 65};
```

4.8 Exercises

- write an expression which evaluates to the set of all multiples of 7 or 11 less then 1000. What is the cardinality of that set
- compute the sum of all multiples of 7 or 11 less then 1000
- compute the product of all multiples of 7 or 11 less then 1000
- write an expression which evaluates to the list of all multiples of 7 or 11 less then 1000, in ascending order
-verify that the following expression evaluates to the truth table of *and*:
    {[[x,y],x and y]: x,y in {true,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 xor b = (a and not b) or (b and not 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 { EXPR: x in m | ....} which evaluates to a largest smap contained in m. Apply this to R and uXv above.

5 Funcs

A cantor expression x is of type func if

```
   is_func(x) = true;
```

holds. Actually, in that case, 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 func = λ-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
return expr; returns the value of *expr;*
return; returns *OM*.
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.

```
> 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:

```
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.

```
$       the use of the pointer operation
$       to refer to hidden objects and data
$       is illustrated here
$       this ex. originates with an ex. taken from
$       Abelson & Sussman pp 167 et seq.
$       the use of maps make it far more readable
$       Note that the function make_account returns
$       a map whose domain is a set of strings, and
$       whose range is a set of lambda -expressions
$       here the -> (pointer) let us get into the private scope
$       of these maps.


> make_account :=
>>  func(name, balance);
>>      return
>>      {
>>        ["deposit", func(n); balance := balance + n; end],
>>        ["withdraw", func(n); balance := balance - n; end],
>>        ["balance", func(); return balance; end],
>>        ["name", func(); return name; end]
>>      };
>>   end;

> gary := make_account("Gary Levin", 1000);
> carol := make_account("Carol Simon Levin", 1000);

> gf := gary('deposit');
> cf := carol('deposit');

> gf->balance;
1000;

> gf->name;
"Gary Levin";

> $ compare with:
> gf('balance')();
1000;

> gf('name')();
"Gary Levin";

> cf->balance;
1000;

> cf->name;
"Carol Simon Levin";


> cf->balance := 300;
> carol('balance')();
300;

> cf->name := 'gribouille';
> carol('name')();
"gribouille";
```

The pointer operation may be used very efficiently to change any package (i.e. a set of nested funcs 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** .

```
$ a "class" point, its constructor and methods
figues := func();
        local pointSet,segmentSet, class,vars,methods;
        vars := {'class','vars','methods'};
        methods := {};
        class := 'figues';
        new := {};
        $_________________________________point
        pointSet := {};
        point := func();
                local u,v, class,vars,methods;   $    class
variables
                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
refcollect ...
                vars := {'u','v'};
                methods := {'translate','dist','homothetia'};
                newP := func(opt x,y);
                        local zpt;
                        x := x?u;
                        y := y?v;
                        zpt := func(); pass; end;
                        pointSet := pointSet with zpt;
                        return zpt;
                end; $ end newP
                new('point') := newP;
                translate := func(dh,dv);
                        this->x := this->x + dh;
                        this->y := this->y + dv;
                end; $ end translate
                dist := func(pt);       $ dist. to a point
                        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(pt,factor);
                        $ homothetia: center is pt, factor
is scaling factor
                        $   make   sure   resuling
coordinates are integers
                        if  pt->class  /=  'point'  or  not
is_number(factor) then
                                return; end;
                        this->x  :=  pt->x  +  fix((this->x  -
pt->x)*factor);
                        this->y  :=  pt->y  +  fix((this->y  -
pt->y)*factor);
                end; $ end homothetia
        end; $ end point
        point();$ invoke the constructor of the class "point"
end; $ end figues
                $ 'figues' is the super class of 'point'
figues(); $ invoke the constructor of the class "figues"

        $ random creation of points
        for i in [1..100] do
```

```
            newP(random(500),random(500));
    end;

    $ count those points
    #point->pointSet;

    $ display these point as a polyline
    w := openwindow();
    $ 'first point': picked arbitrarily
    first_point := arb(point->pointSet);
    $ display a polyline joining all the points
    $ set the initial pen position at the first point
    moveto(first_point->x,first_point->y);
    for pt in point->pointSet less first_point do
            lineto(pt->x,pt->y));
    end;

    $ scale down the set of points around first_point and
redisplay
    clearscreen();
    moveto(first_point->x,first_point->y);
    for pt in point->pointSet less first_point do
            pt->homothetia (first_point,0.2); $ scale
down to 20% of original size
            lineto(pt->x,pt->y));
    end;
```

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, ie. 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 progr ptr
ref	ref(afunc); $ returns a map describing the list of the identitiers 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);
          return tab_fact (n);
    end;
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] | j = i**2;
```
or violates
```
forall i in p, j in [2..i-1] | 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, isoceles, 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:

ast	*ast_kind*
x+y	"+"
x*y	"*"
exists x in s | K(x)	"exists"

'a text'	"T_STring"
x	"T_Id"
1	"T_Integer"
etc.	

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):

T_Pat	<	..	T_Of	->
T_Cmt	<=	|	om	?
T_Spec	=	T_Missing	opt	#
T_Id	/=	take	print	[
T_Integer	>	to	printf	{
T_String	>=	do	program	(
T_Real	INFIX	else	read	
T_Boolean	+	elseif	readf	
CLEAR	-	end	return	
:=	with	exists	then	
where	less	false	true	
iff	*	for	value	
impl	/	forall	while	
or	mod	from	write	
and	div	fromb	writeln	
not	**	frome	MAP	
in	%	func	SELECTOR	
notin	UNARY	if	ITERS	
subset	CALL	local	this	

6.1 operations on ast objects

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

af(i)
or
ast(af,i)

subscripts indicate a filiation in the abstract syntax tree:
- 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 aquivalent 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

```
> af;
x + y;;
> type(af);
"AST";
> is_ast(af);
true;
> which_ast(af);
"CALL";
> which_ast(af(1));
"+";
> b := af(1);
> which_ast(b(0));
"CALL";
> b(1);
x;
> b(1)(1);
"x";
```

af('t')
or
ast(af,'t')
	'typ' or any initial segment thereof, is the subscript leading to the ast_kind code. See also which_ast()	`> af('t');`
		`292;`
		`> which_ast(292);`
		`"CALL";`
		`> af(1)('t');`
		`281;`
		`> which_ast(281);`
		`"+";`

af %ast [...]
or
%ast [af ...]
	compound subscripting defines paths from a tree root to a subtree	`> af %ast [];`
		`x + y;;`
		`> %ast [af,1,1];`
		`x;`
		`> %ast [af,1,1,1];`
		`"x";`

6.2 predicates on ast objects

is_ast_leaf(a)	true if *a* is a leaf node: identifier, or a constant node : identifier: "T_Id", integer: "T_Integer", real: "T_Real", string: "T_String", boolean: "true", "false"	`> af;`
		`x + y;;`
		`> is_ast_leaf(af);`
		`false;`
		`> is_ast_leaf(af %ast`
		`>> [1,1]);`
		`true;`
		`> which_ast(%ast [af,1,1]);`
		`"T_Id";`

6.3 some ast analysis and interpretation primitive functions

which_ast(af)	-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 correponding string is returned
	-if af is a string corresponding to a known ast_kind, its corresponding internal code is returned
		`> af('t');`
		`292;`
		`> which_ast(292);`
		`"CALL";`
		`> which_ast(af);`
		`"CALL";`
		`> which_ast("CALL");`
		`292;`

analyze	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
		`$ take input from console`
		`> af := analyze();`
		`x+y;`
		`> af;`
		`x + y;;`
		`> bf := analyze('x+y');`
		`> af = bf;`
		`true;`

construct	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
		`> af;`
		`x + y;;`
		`> af := construct('af*2');`
		`> af;`
		`(x + y) * 2;;`

setAst	anAST := setAst(anAst,-1|0|1|2|'type'|3, modif);$ modify anAst: -1:parent, 0:node itself,1:left,2:right,>=3:more
	
	N.B. unlike in ast subscripting, 0 does not represent an ancestor node, but the node itself
		`> af := analyze('x+y');`
		`> af;`
		`x + y;;`
		`> af(1);`
		`x + y;`
		`> setAst(af(1),'t',`
		`>> which_ast('*'));`
		`x * y;`
		`> af;`
		`x * y;;`

pretty	produces a string which is the pretty print of the ast arg	`> af;`
		`(x + y) * 2;;`
		`> pretty(af);`
		`"(x + y) * 2;";`

ugly	displays the tree-like structure of its ast argument, including the ast_kind of each node	`> af;`
		`(x + y) * 2;;`
		`> ugly(af);`
		`( CALL :`
		` ( * :`
		`    ( + :`
		`            ( T_Id :       x )`
		`            ( T_Id :       y )`
		`     ( T_Integer :         2 ) OM;`

findAst	findAst(atyp,af); $ atyp = int|string|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|fileName|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
	
	see also analyze, construct, setScanStop, and the notes in section 3.4
		`> tokenstream := scan('mix\`
		` 100g of sugar with',1,1);`
		`["mix", 100, "g", "T_Of", "sugar", "with"];`
		`> tk := scan('mixons\`
		`le lait et 100g de beurre et 1.5 l d"eau',1,1);`
		`> tk;`
		`["mixons", "le", "lait", "et", 100, "g", "de", "beurre", "et",`
		` 1.500, "l", "d", "\q", "eau"];`

kwd	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	

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

```
$
$        we will present here a subset of the available
$        AST-processing
$        tools:
$                 analyze           the basic parser
$                 pretty, ugly      pretty-printer,ugly-
printer
$                 eval              evaluates an AST
$        (or an identifier string)
$                 which_ast         to tell the type of an
AST
$                 findAst           to produce a tuple of
$        sub-AST of a given type
$                 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
$                 chain_ast         to link AST-nodes to
their parent node
$ use the built-in help to find how to use these primitives
!help pret
prettyStrings(true|false);$ pretty print in mode: emphasize
strings quoting ...
str := pretty(ast);$ pretty print the ast arg
 ...
!help naly
ast := analyze(File| expression_string | om); $ File: the input
stream, om:the standard input,
expression_string: the expr to parse
analyze is a parser ...


!flex_alloc on
$ use undefined objects:
> a := om; b := om;

> af := analyze("a+b");
> af;
a + b;;

> ugly(af);( CALL :
 ( + :
          ( T_Id :    a )
          ( T_Id :    b )
OM;


> $ keep the expr part:
> af := af(1);
> 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 := analyze("u := (a+b)*(c+d)");
> v := findAst('+',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";

> $ it is always possible to move within
> $ the tree downwards:
> v := v(1); $ look at the 1st AST
> v(1);
c;

> v(2);
d;

> v('t');
280;

> which_ast(v('t'));
"+";

> which_ast(which_ast(v('t')));
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;


> $ when 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
>>                  w := arb (access);
>>                  reach := reach with w;
>>                  toUse := toUse with w;
>>        else
>>                  toUse := toUse less v;
>>        end;
>> end;
>> return reach;
>> end;


> $
> $ stop collecting references
> refcollect(false);
OM;
```

```
> $ look at the ref

> ref_explore := ref(explore);
> ref_explore;
{["Opt", ["avoid"]],
 ["all",
  ["w", "g", "access", "avoid", "arb", "v", "toUse", "s",
   "reach"]],
 ["Parameter", ["g", "s"]],
 ["Local", ["toUse", "reach", "access"]]};

> $ in this case what are the references to
> $ non-local(global) objects?
> {x: x in ref_explore('all') | x notin
>>          %+[ref_explore(u): u in ["Local","Parameter","Opt"]]};
{"arb", "v", "w"};
```

6.5 pattern matching and unification primitives

varsOf

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

```
> af;
x * y;;
> varsOf(af);
[y, x];
> bf := analyze('f(x+y)');
> varsOf(bf);
[y, x, f];
```

varsIn

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

```
> af;
x * y;;
> varsIn(af);
[y, x];
> bf := analyze('f(x+y)');
> varsIn(bf);
[y, x];
```

match

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

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 $@@@

```
> af := analyze()(1);
>> exists x,y,z in s | K(x,z);
> af;
exists x in s, y in s, z in s | K(x,
z);

> setAst(af(1)(2)(1),'t', 100);
$@@@;
> af;
exists x in s, $@@@, z in s | K(x,
z);
> bf := analyze()(1);
>> exists u,v,w in s | K(u,w);
> bf;
exists u in s, v in s, w in s | K(u,
w);
> match(af,bf);
true;
> af;
exists u in s, v in s, w in s | K(u,
w);
```

parse_msg

parse_msg(bool); $ false->no parsing msg, true->msg
suspends output of parse msgs

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

```
> a1 :=
>> analyze('p(x,f(x),y)');
>    a2 :=
>> analyze('p(g(z),w,y)');
>    a3 :=
>> analyze('p(x,u,u)');
> S := {a1,a2,a3};
> s1 := unify(S);
> s1;
[{p(g(z), f(g(z)), f(g(z)));},
 {[x,  g(z)],  [y,  w],  [u,  y],  [w,
f(g(z))]}];
> g1 := analyze('anc(x,W)');
> h1 :=
>> analyze("anc(Z0,y)");
> w :=
>> unify({g1,h1},['Z0','W']);
> w;
[{anc(Z0, W);}, {[y, W], [x, Z0]}];
```

ast_subs

anAst := ast_subs(ast1,subs_map | [string|t_id1,astt2]); $ subs_map map strings or T_Id trees onto other ASTs -- all occurrences of T_Ids in the domain of subs_map are subsituted

```
> a1;
p(x, f(x), y);;
> aMap;
{[x,  g(z)],  [y,  w],  [w,  f(g(z))],
[u, y]};
> ast_subs(a1,aMap);
p(g(z), f(g(z)), w);;
```

6.6 Exercises

- write a func to list all the occurences of a given identifier, within a given ast
- write a func to list all the occurences of a given ast as subtree of another given ast
- write a func to list all the occurences 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 occuring 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 occuring in (set, tuple) -formers
- Write a func to produce the set of all bound variables occuring in for-loops
- Write a func to produce the set of all local and value declared variables occuring in funcs
- Similar exercises, but instead with free or non-local variables occuring 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.

2. Definitions appear as:

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

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

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

```
INPUT  -->  PROGRAM

INPUT  -->  STMT

INPUT  -->  EXPR ;
```

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.

```
PROGRAM  -->  program ID ; STMTS end ;
```

Of course, it can appear on several lines. One may optionally close with *end program.*

7.4 Statements

```
STMTS  -->  STMT+
```

NB-> One or more instances of STMT. The final semicolon is optional.

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;          a is changed to contain the value 4.

[a,b] := [1,2];       a is assigned 1 and b is assigned 2.

[x,y] := [y,x];       Swap x and y.

f(3) := 7;
```

If f is a *tuple*, then the effect of this statement is to assign 7 as the value of the third component of f. If f is a *map*, then its effect is to replace all pairs beginning with 3 by the pair [3,7] in the set of ordered pairs f. If f is a *func*, -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

STMT --> while EXPR do STMTS end ;

EXPR must evaluate to a Boolean value. EXPR is evaluated and the STMTS are executed repetitively as long as this value is equal to true. One may optionally close with *end while.*

read statement

STMT --> 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.

STMT --> 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 *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).

STMT --> readf PAIR-LIST ;
STMT --> readf PAIR-LIST from EXPR ;

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

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

print statement

STMT --> print EXPR-LIST ;

Each expression in EXPR-LIST is evaluated and printed on standard output. The output values are formated to show their structure, with line breaks at reasonable positions and meaningful indentation.

STMT --> print EXPR-LIST to EXPR ;

As in read...from..., EXPR must be a value of type file. The values are
written to the external 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).

STMT --> printf PAIR-LIST ;
STMT --> printf PAIR-LIST to EXPR ;

```
> 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.

return statement

STMT --> 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.

STMT --> return EXPR ;

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

take .. from statement

STMT --> take LHS from LHS ;

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

STMT --> take LHS frome LHS ;

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

STMT --> take LHS fromb LHS ;

The second LHS must evaluate to a tuple (or a string). The value of its first component (defined or not) (first character) is assigned to the first LHS and all components of the tuple (characters of the string) are shifted left one place. That is, the new value of the ithcomponent 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

```
> 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.

format --> INT
format --> INT.FRACT

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:
precision = 0.FRACT * 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:

type	**int part**	**fract.part**
Float	20	5
Integer	0	50 (for breaking large ints)
String	0	
Anything else	10	

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 specifed by a tuple. **'\nfin'** uses the default format. x's format tuple is
3*[10,' --- '] 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 specifaction 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;
thereare                          5outputformatsinversion
0.41000:integersfloatingpointnumbersstringsortupleofoutput
> printf x:3*[10,' --- '] with '\n->', '\nfin';
     there ---        are ---         5 ---
->     output ---    formats ---         in ---
->    version ---    0.41000 ---          : ---
->   integers ---   floating ---      point ---
->   numbers ---    strings ---         or ---
->   tupleof ---     output
fin
> 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
fin
>
```

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.)

ITERATOR --> ITER-LIST
ITERATOR --> ITER-LIST | EXPR

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

ITER-LIST --> SIMPLE-ITERATOR+
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.

SIMPLE-ITERATOR --> BOUND-LIST in EXPR

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

SIMPLE-ITERATOR --> BOUND = ID (BOUND-LIST)

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

SIMPLE-ITERATOR --> BOUND = ID { BOUND-LIST }

Same as the previous one for the case in which ID is an mmap.

BOUND-LIST --> BOUND+
one or more BOUND, separated by commas

BOUND --> ~
Corresponding value is thrown away.

BOUND --> ID
Corresponding value is assigned to ID.

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

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.

SELECTOR --> { EXPR-LIST }

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

SELECTOR --> (EXPR .. EXPR)

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

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

a <=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 ith component is the value of the (a + i? 1)stcomponent of the value of EXPR.
a = b + 1	Value is the empty tuple.
a > b + 1	Run-time error.

SELECTOR --> (.. EXPR)

Means the same as (1 .. EXPR).

SELECTOR --> (EXPR ..)

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

SELECTOR --> ()
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.

LHS --> ID
LHS --> LHS SELECTOR

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

LHS --> [LHS-LIST]

LHS-LIST --> 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 --> if EXPR 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 *.

EXPR --> EXPR / EXPR

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

EXPR --> EXPR div EXPR

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

(a div b)? b+ (a mod b)= 0 for b> 0
a div (? b)= ? (a divb) for b< 0.

EXPR --> EXPR mod EXPR

Both instances of EXPR must evaluate to integer and the second must bepositive. The result is the remainder, and the following condition isalways satisfied,
0<=a modb < b.

EXPR --> EXPR ** EXPR

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

EXPR --> EXPR with EXPR

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

EXPR --> EXPR less EXPR

The value of the first EXPR must be a set. The value is that set with the value of the second EXPR removed, if it was present; the value of the first EXPR, if the second was not present.

Pointer expressions may be defined as follows:
EXPR --> EXPR -> 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.

EXPR --> EXPR = EXPR
The test for equality of any two Cantor values.

EXPR --> EXPR /= EXPR
Negation of EXPR=EXPR.

EXPR --> EXPR < EXPR
EXPR --> EXPR > EXPR
EXPR --> EXPR <= EXPR
EXPR --> EXPR >= EXPR

For all the above inequalities, both instances of EXPR must evaluate to the same type, which must be number or string. For numbers, this is the test for the standard arithmetic ordering; for strings, it is the test for lexicographic ordering.

EXPR --> EXPR in EXPR

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

EXPR --> EXPR notin EXPR
Negation of EXPR in EXPR.

EXPR --> EXPR subset EXPR

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

EXPR --> EXPR and EXPR

Logical conjunction. Both instances of EXPR should evaluate to a Boolean.

If the left operand is false, the right operand is not evaluated. Actually returns the second argument, if the first is true. While the user may depend on the left-to-right evaluation order, it is recommended that they not depend on the behavior when the second argument is not Boolean.

EXPR --> EXPR or EXPR

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

EXPR --> EXPR impl EXPR

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

EXPR --> EXPR iff EXPR

Logical equivalence. Both instances of EXPR should evaluate to a Boolean.

It actually checks for equality, like =, but it has a different precedence. It is recommended that the user not depend on iff to work with arguments other than Booleans.

EXPR --> % 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 bycommas.

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
%	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-orientd 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 sofar.

!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.

```
> 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);
! Evaluate: fact(10) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(9) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(8) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(7) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(6) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(5) eval time: Wed Oct 26 18:41:27 1994

! Evaluate: fact(4) eval time: Wed Oct 26 18:41:28 1994

! Evaluate: fact(3) eval time: Wed Oct 26 18:41:28 1994

! Evaluate: fact(2) eval time: Wed Oct 26 18:41:28 1994

! Evaluate: fact(1) eval time: Wed Oct 26 18:41:28 1994

! fact returns: 1 return time: Wed Oct 26 18:41:28 1994

! fact(2) := 2;
! Evaluate: fact(2) eval time: Wed Oct 26 18:41:28 1994

! Yields: 2;
! fact returns: 2 return time: Wed Oct 26 18:41:29 1994

! fact(3) := 6;
! Evaluate: fact(3) eval time: Wed Oct 26 18:41:29 1994

! Yields: 6;
! fact returns: 6 return time: Wed Oct 26 18:41:29 1994

! fact(4) := 24;
! Evaluate: fact(4) eval time: Wed Oct 26 18:41:29 1994

! Yields: 24;
! fact returns: 24 return time: Wed Oct 26 18:41:29 1994

! fact(5) := 120;
! Evaluate: fact(5) eval time: Wed Oct 26 18:41:30 1994

! Yields: 120;
! fact returns: 120 return time: Wed Oct 26 18:41:30 1994

! fact(6) := 720;
! Evaluate: fact(6) eval time: Wed Oct 26 18:41:30 1994

! Yields: 720;
! fact returns: 720 return time: Wed Oct 26 18:41:30 1994

! fact(7) := 5040;
! Evaluate: fact(7) eval time: Wed Oct 26 18:41:31 1994

! Yields: 5040;
! fact returns: 5040 return time: Wed Oct 26 18:41:31 1994

! fact(8) := 40320;
! Evaluate: fact(8) eval time: Wed Oct 26 18:41:31 1994

! Yields: 40320;
! fact returns: 40320 return time: Wed Oct 26 18:41:31 1994

! fact(9) := 362880;
! Evaluate: fact(9) eval time: Wed Oct 26 18:41:31 1994

! Yields: 362880;
! fact returns: 362880 return time: Wed Oct 26 18:41:32 1994

! fact(10) := 3628800;
! Evaluate: fact(10) eval time: Wed Oct 26 18:41:32 1994

! Yields: 3628800;
! fact returns: 3628800 return time: Wed Oct 26 18:41:32 1994

3628800;
> fact(12);
! Evaluate: fact(12) eval time: Wed Oct 26 18:43:37 1994

! Evaluate: fact(11) eval time: Wed Oct 26 18:43:37 1994

! Evaluate: fact(10) eval time: Wed Oct 26 18:43:37 1994

! Yields: 3628800;
! fact(11) := 39916800;
! Evaluate: fact(11) eval time: Wed Oct 26 18:43:38 1994

! Yields: 39916800;
! fact returns: 39916800 return time: Wed Oct 26 18:43:38 1994

! fact(12) := 479001600;
! Evaluate: fact(12) eval time: Wed Oct 26 18:43:38 1994

! Yields: 479001600;
! fact returns: 479001600 return time: Wed Oct 26 18:43:39 1994

479001600;
>
```

Figure 5: Tracing and timing

obsolete: in_debug

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 cheking '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.

```
> !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.

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

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

> f(2) := 1;

!  f(2) := 1;

> f(4);
!  Evaluate:  f(4);
!  Evaluate:  f(3);

!  Evaluate:  f(2);
!  Yields:  1;
!  Evaluate:  f(1);
!  Yields:  1;

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

Figure 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 frome LHS ;
STMT --> **take** LHS fromb LHS ;

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

STMTS --> 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+
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]

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

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. λ-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 whithin 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 accomodate 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.

```
> !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.

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

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 withinan 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",
```

"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 wiil 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 vraible 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 mesuré par clock
f := func(x); return x+10; end;
!watch f

!breakpoint on
```

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

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

```
! 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;

> RET_CLOSURE_bkPt;
"noop;";

> S_MAP_ACCESS_bkPt;
"noop;";

> S_MAP_ACCESS_bkPt := 'entry_time := clock();';
> RET_CLOSURE_bkPt := 'delta := clock() - entry_time;'+
' printf delta;';
> f := func(x); return x+10; end;

> !watch f
!'f' watched

> !breakpoint on

> f(10);Var(444) 'f'

! Evaluate: f(10);
Var(444) 'f'
!break point: S_MAP_ACCESS_bkPt!
;
!break point: RET_CLOSURE_bkPt!
;
! f returns: 20;
20;
    10
> entry_time;
1531779;

> delta;
10;

> f(15);Var(444) 'f'

! Evaluate: f(15);
Var(444) 'f'
!break point: S_MAP_ACCESS_bkPt!
;
!break point: RET_CLOSURE_bkPt!
;
! f returns: 25;
25;
    10
!include bkPt_session completed


!verbose on
```

```
> !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;

> RET_CLOSURE_bkPt;
"noop;";

> S_MAP_ACCESS_bkPt;
"noop;";

> S_MAP_ACCESS_bkPt := 'entry_time := clock();';
> RET_CLOSURE_bkPt := 'delta := clock() - entry_time;'+
' printf delta;';
> f := func(x); return x+10; end;
! f := !FUNC(33939a/341bba)!;

> !watch f
!'f' watched

> !breakpoint on

> f(10);Var(444) 'f'

! Evaluate: f(10);
Var(444) 'f'
!break point: S_MAP_ACCESS_bkPt: entry_time := clock();!
;
!break point: RET_CLOSURE_bkPt:  delta := clock() -
entry_time;
printf delta;!
;
! f returns: 20;
20;
    24
> entry_time;
1532863;

> delta;
24;

> f(15);Var(444) 'f'

! Evaluate: f(15);
Var(444) 'f'
!break point: S_MAP_ACCESS_bkPt: entry_time := clock();!
;
!break point: RET_CLOSURE_bkPt:  delta := clock() -
entry_time;
printf delta;!
;
! f returns: 25;
25;
    13
!include bkPt_session completed
```

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

primitive name	arg nbr	invocation, comment
basic math		
exp	1	r := exp(x);$ power of e (inverse: see ln)
ln	1	r := ln(x); $ neper. log (inverse: see exp)
log	1	r := log(x); $ base 10 log
max	2	x := max(a,b);
min	2	x := min(a,b);
abs	1	y := abs(x); $ absolute value
ceil	1	n := ceil(real); $ int. approx. of a real, see also floor, fix
fix	1	n := fix(real); $ int. approx. of a real, see also ceil,floor
floor	1	n := floor(real); $ int. approx. of a real, see also ceil,fix
even	1	bool := even(n);
odd	1	bool := odd(n);
random	1	n := random(root); $ type of root is the returned type
randomize	1	x := randomize(seed); $ set new seed for random gen.
round	1	n := round(x);
sgn	1	n := sgn(int); $ sign
sqrt	1	r := sqrt(x); $ square root
trigonometry		
cos	1	r := cos(x);
sin	1	r := sin(x);
tan	1	r := tan(x);
acos	1	r := acos(x);
asin	1	r := asin(x);
atan	1	r := atan(x);
cosh	1	r := cosh(x);
sinh	1	r := sinh(x);
tanh	1	r := tanh(x);
acosh	1	r := acosh(x);
asinh	1	r := asinh(x);
atanh	1	r := atanh(x);
basic set primitives		
arb	1	x:= arb(set); $ choice function
image	1	set := image(map); $ same as range
npow	2	set_collection := npow(set,nmax); $ the subsets of atmost nmax elts
pow	1	power_set := pow(set);
range	1	set := range(map); $ same as image

type testing		
is_atom	1	bool := is_atom(x);
is_ast	1	bool := is_ast(x);
is_boolean	1	bool := is_boolean(x);

is_bignum	1	bool := is_bignum(x);
is_defined	1	bool := is_defined(x);
is_integer	1	bool := is_integer(x);
is_file	1	bool := is_file(x);
is_floating	1	bool := is_floating(x);
is_func	1	bool := is_func(x);
is_map	1	bool := is_map(x);
is_func	1	bool := is_func(x);
is_number	1	bool := is_number(x);
is_om	1	bool := is_om(x);
is_set	1	bool := is_set(x);
is_string	1	bool := is_string(x);
is_table	1	bool := is_table(x); $ a map with domain and range elements of simple type
is_textpane	1	bool := is_textpane(x);
is_tuple	1	bool := is_tuple(x);
type	1	str := type(x);

source & binary		
interp	1	interp(s); $ interpret string s as a Cantor instruction or directive
include	1	include(filename); $ program command for source inclusion (see also the directive !include)
components	0	tuple := components(); $ tuple of currently restored components name
ids_in	1	idList := ids_in(component);
newids	0	tuple := newids();$ list of defined obj and their type
newsymbols	1	tuple := newsymbols(x) $x= om: list of defined obj x= 0: list of defined and their type; x>0 :list of defined obj their type and their value
allchanges	1	tuple := allchanges(x); $ x means the same as for newsymbols. tuple contains all ids which have changed
visiblechanges	1	tuple := visiblechanges(x); $ x means the same as for newsymbols. tuple contains visible ids
resetchanges	0	resetchanges();
ids	1	tuple := ids(x); $ids(om): tuple of defined ids, ids(0): tuple of [type, defined-id] ids(1): tuple of [type, defined-id, value]
gc	0	gc(); $ garbage collection invokation
restore	1	restore(filename|File|om);$ restore var. identifier and its value if file was 'save'-d or only a value if the file was created by a 'store'
save	2	save(varname,filename|File|om);$ if filename is OM or '': interactive selection of file
store	2	store(expression,filename|File|om);$ if filename is OM or '': interactive selection of file
ref	1	ref(func); $ prints a list of the identitiers referenced by func
refcollect	1	refcollect(true|false); $ sets -resp resets- ref collection
workComp	1	workComp(compName); $ cur env is that of compName

misc		
rank	2	x := rank(x,L); $ x is an item in L (list or tuple), or a substring if L is a string. returns 0 if x notin L
sorted	1	sorted(true|false); $ for displaying sets in h-sorted | unsorted order
size	1	n := size(object); $ size in byte of the object and its dependants
blockcount	1	n := blockcount(object); $ size in # of basic elements in the object and its dependants
npow2	2	x := npow2(set,nmax); $the set of all subsets with exactly nmax elements! equivalent to npow, but much faster
qsort	1	tuple := qsort(collection); $ tuple contains the collection in a sorted order
sort_index	1	tuple := sort_index(collection); $ tuple contains the permutation index map to sort collection
break	0	break();$ force premature (loop) exit
setallbreakpt	0	setallbreakpt();$ all breakpoint commands set to OM
ignoreallbreakpt	0	ignoreallbreakpt();$ all breakpoint commands set to noop
allbreaktype	0	allbreaktype();$ all breakpoint types
breaktype	0	breaktype();$ current breakpoint type
clock	0	tickCount := clock();$ approx. 1 Tick evry 16 msec
pause	1	pause(nb_sec); $ suspend all procssing for nb_sec
tellfunc	0	aFunc := tellfunc();$ attempts to tell within which func is current progr ptr
setBaseAtom	1	setBaseAtom(atom|om); $ set Base_Atom to atom | !0!

text-number conversion		
ator	1	real := ator(floatingNbrString);
rtoa	1	str := rtoa(realNbr);
atoi	1	n := atoi(nbrAsString);
itoat	1	atom(resp. int) := itoat(int(resp. atom));$ int (resp atom-value) to atom-value (resp int) conversion
itoa	1	str := itoa(n);$ integer (or atom-value) to string conversion
ord	1	n := ord(char); $ integer value of a character
char	1	s := char(s); $ (ascii) char value of an integer
hash	1	int = hash(x) $ hash value
date	0	str := date(); $ current date, with the precision of a second
uclcase	2	s := uclcase('(Uu)|(Ll)',string); $ convert string into upper (resp. lower) case
strsubst	3	string := strsubst(pat, string,by);$replace all occurrences of pat in string with by
max_line	1	i := max_line(int);$ control output line size in console
show_mode	1	show_mode(int|bool); $ arg 1 or true sets raw mode, i.e. w/o quotes,semi-colon
show	n	str := show(args); $ returns a string or a tuple of str. to construct what would be printed by print args

basic file processing		
close	1	close(File);
eof	1	bool := eof(File);
opena	1	File := opena(file_name); $open append text file
openab	1	File := openab(file_name); $open append binary file
openr	1	File := openr(file_name); $open read text file
openrb	1	File := openrb(file_name); $open read binary file
openrw	1	File := openrw(file_name); $open read-write text file
openrwb	1	File := openrwb(file_name); $open read-write binary file
openw	1	File := openw(file_name); $open write text file
openwb	1	File := openwb(file_name); $open write binary file
fwrite	2	fwrite(item,File);
fread	3	value_read := fread(File,'int'|'str',count);
fseek	2	fseek(File,f_position);
ftell	1	f_position := ftell(File);
rewind	1	rewind(File);
toend	1	toend(File);
flen	1	n := flen(File|fileName); $ file size
lc	1	n := lc(File|fileName|om); $ text file line count
fgets	2	string := fgets(n,file); $read a line of at most n char;
advanced file primitives		
fcopy	2	n := fcopy(from,to); $ copy from a File to a File
finsert	2	n := finsert(from,to); $ copy from a File to a text pane
panecopy	2	n := panecopy(from,to); $ copy from a text pane fo a File
file_find_str	2	bool := file_find_str(aFile,string); $ find a string within a file
rename	2	bool := rename(from,to); $ rename from a File to a File; returns true iff successfull
fdelete	1	bool := fdelete(filename); $ delete a File; returns true iff successfull

scope control		
applyEnv	2	fn1 := applyEnv(fn,optEnv); $set fn1 env to optEnv if specified, otherwise to the current env
applyNilEnv	1	fn1 := applyNilEnv(fn); $set fn1 env to the current env
hasNilEnv	1	bool := hasNilEnv(fn); $true if fn is a func with Nil Env
detachEnv	0	env := detachEnv(); $unlink the current func's env from creator's
codeOf	1	code := codeOf(func);$ code is a non-printable object
overrideOf	1	aMap := overrideOf(func);$ aMap is a s-map
envOf	1	env := envOf(func);$ env is a non-printable object
mkLocal	2	mkLocal(idName,aFunc);$ creates a local var in the environmen of aFunc

drawing		
clearscreen	0	clearscreen(); $ clears the screen

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] | x, y, wx, wy); $ frame a box at (x,y) having: width = wx, height = wy
fbox	4	fbox([x,y,wx,wy] | x, y, wx, wy); $ fill box
ellipse	4	ellipse([x,y,wx,wy] | 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] | 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],..,[xn,yn]]
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
alloccolor	3	alloccolor(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, [x1,y1], [x2,y2]); $ aCodeStr is 'invert'|'rect'|'vector'|'ellipse'
compute_rect_text	4	[[x,y],[w,h]] := compute_rect_text(fontname, x, y, str);

windows		
openwindow	1	idWin := openwindow(anAttrMap);
closewindow	0	closewindow(); $ closes current window
show_hide	2	win_no := show_hide(win_no | -1, true|false); $ 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

ask_long	4	anAnswer := ask_long(x, y, aMsg, aQuestion); $ a dialog returning a bignum
settitle	1	settitle(title); $ change window title
tell	1	tell(msg); $ open a warning window

events		
wait_mask_event	2	win_no := wait_mask_event(win_no,event_mask_set); $ nondiscriminating wait: wait_mask_event(-1,{});
check_mask_event	2	anEventMap := check_mask_event(win_no, event_mask_set); $check all events:check_mask_event(-1,{});
PostHEvent	2	PostHEvent(win_no, event_map); $ post the event defined in event_map:)
interface_task	0	interface_task();$ insert everywhere to allow queued events processing
event_mask	0	event_mask();$ help: the event type structure
event_map	0	event_map();$ help: the event map structure
event_domains	0	event_domains();$ help: more on event map structure
event_convert	1	event_typeName|event_typeNbr := event_convert (event_typeNbr|event_typeName); $ for CantorDrvrMap, convert type Nbr into name and conversely

menus		
install_menu	4	menuID := install_menu(win_no, ['item1',..,'itemN'], [ox,oy], [w,h]); $ ox,oy], [w,h] designates the active rectangle
remove_menu	1	remove_menu(win_no, menuID);
popupmenu	3	itemID := popupmenu(win_no, menuID, [ox,oy]); $ [ox,oy] indicates where to display the menu
getpopup	1	[[x,y],itemId] := getpopup(['item1',..,'itemN']); $ returns the selection within popup item list
set_item	3	set_item(menuID,menuItem, item_string);$ replace in menu menuID the item # menuItem by item_string
set_item_status	3	set_item_status(menuID,menuItem, bool |0|1|om);$ 3rd arg: bool->check (resp uncheck) item, 0(resp 1,om) -> disable(resp enable) item

regions		
clipwindow	1	clipwindow(region);
create_region	0	region := create_region();
rect_region	2	region := rect_region([ox,oy],[w,h]);
destroy_region	1	destroy_region(region);
union_region	2	region := union_region(region1, region2);

buttons		
create_button	1	button := create_button(anAttrMap);
set_button_attributes	2	set_button_attributes(button, anAttrMap);
get_button_attributes	2	anAttrMap := get_button_attributes(button, anAttrSet);
destroy_button	1	destroy_button(button);

draw_buttons	1	draw_buttons(win_no);
get_button_value	1	button_value := get_button_value(button);
button_attributes	0	button_attributes();$ help: the button attribute map structure

text pane		
create_text_pane	2	paneID := create_text_pane(win_no, tp_map);
delete_text_pane	1	delete_text_pane(paneID);
tp_get_select	1	[selStart, selEnd] := tp_get_select(paneID);$ok for grids
tp_set_select	2	tp_set_select(paneID,[selStart, selEnd]); $ok for grids
tp_return_select	1	selStr := tp_return_select(paneID); $ok for grids
tp_show_selection	1	tp_show_selection(paneID);
tp_delete_selection	1	tp_delete_selection(paneID);
tp_insert	2	tp_insert(paneID, aString);$ at most 8000 char long $ok for grids
tp_text_length	1	tp_text_length(paneID);
tp_set_prompt	2	tp_set_prompt(paneID, aString);
tp_select_paragraph	1	tp_select_paragraph(paneID);
tp_set_focus	1	tp_set_focus(paneID);
tp_set_handler	2	tp_set_handler(paneId,'console'|'edit'|'field'|'data'|'sel'); $ set (change) the key_handler
tp_find_string	2	found := tp_find_string(paneID, aString);
text_pane_attributes	0	text_pane_attributes();$ the text_pane attribute map structure
tp_set_handler	2	tp_set_handler(paneId,'console'|'edit'|'field'|'data'|'sel'); $ set (change) the key_handler

pane files		
open_pane_file	3	pane_file := open_pane_file(paneID, win_no, mode); $--> mode == 'r' or 'w'
get_pane_id	1	paneID := get_pane_id(pane_file); $--> return -1 if pane_file is a disk file
get_file_window	1	win_no := get_file_window(pane_file); $--> return -1 if pane_file is a disk file
echo_pane_file	2	echo_pane_file(pane_file,echo_file);
remove_echo_file	1	remove_echo_file(echo_file);

logo turtle		
cclearscreen,cs	0	clearscreen(); $ clears the screen cs(); $ clears the screen
round	1	n := round(x);
set_world	2	set_world([[x,y],[w,h]]); $ rect: [x,y],[w,h]
set_view	2	set_view([[x,y],[w,h]]);
wtov	1	v_pt := wtov(w_pt);
vtow	1	w_pt := vtow(v_pt);
move_to	1	move_to(w_pt);
line_to	1	line_to(w_pt);
dot	1	dot(w_pt);

cligne_line	2	cligne_line(pt1,pt2);
is_pendown	0	bool := is_pendown();
set_pendown	1	set_pendown(bool);
pendown,pd	0	pendown(); pd(); $ pd same as pendown
penup,pu	0	penup(); pu(); $ pusame as penup
pos	0	w_pt := pos();
set_pos	1	set_pos(w_pt);
stdpt	1	w_pt := stdpt(alfa_pt);
home	0	home();
std_dir	1	dir := std_dir(angle);
forward, fd	1	forward(dist); fd(dist); $ fd same as forward
backward, bk	1	backward(dist); bk(dist); $ bk same as backward
left, lt	1	left(angle); lt(angle); $ lt same as left
right, rt	1	right(angle); rt(angle); $ rt same as right
set_turtle	3	set_turtle(thePos,theDir,is_down);
ngon	2	ngon(n, edge);
init_turtle	0	init_turtle();
open_std_turtle	0	open_std_turtle();
cngon	2	cngon(n, radius);

abstract syntax		
parsetree	0	parsetree(); $ returns the accumulated parsetrees
resetParses	0	resetParses(); $ set Parses to Nil
scan	3	scan(File|fileName|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

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	2	findAll(atyp,ast); $produces a tuple of all the ast sub-trees 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 = int|string|set|om;\ produces a tuple of all the ast sub-trees with the given type
match	2	match(ast1,ast2); $true if ast1, ast2 are eq or if ast2 contains pattern matching ast1 subtrees or if ast1 contains patterns matching ast2 subtrees which will then be subsituted into ast1
well_defined	2	bool := well_defined1(af opt sw);$ 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); $ S 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|t_id1,astt2]); $ 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		designates target Cantor types for C data or data collections converted to or from Cantor (see below *mk cantor obj*)

type		#define Boolean 2 #define Integer 3 #define Bignum 4 #define File 5 #define Real 6 #define Set 7 #define String 8 #define Tuple 9 #define SetOfShortInt 72 #define SetOfInt 73 #define SetOfBn 74 #define SetOfFile 75 #define SetOfReal 76 #define SetOfStr 78 #define TupleOfShortInt 92 #define TupleOfInt 93 #define TupleOfBn 94 #define TupleOfFile 95 #define TupleOfReal 96 #define TupleOfStr 98
C types		C_Val /* a structure for converting data types */ Cantor_Ptr /* the Cantor data structure pointer */
interoperability functions		C-primitives for C-to-Cantor or Cantor-to-C
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 */
invocation (from Cantor) of C-procedures		a specific packaging is required
invocation (from C) of Cantor		self_interp (see *interp*)

macintosh specific misc.		
get_in_file_name	1	filename := get_in_file_name(filetype); $ interactive dialog to locate input file
get_out_file_name	1	filename := get_out_file_name(filetype); $ interactive dialog to locate output file
setfilecreator	3	setfilecreator(filename, creator, filetype);
add_mac_menu	3	menuID := add_mac_menu(title, ['item1',..,'itemN'], om | -1);$ 3rd arg is optional; if present menu is hierarchical
check_menu_item	3	check_menu_item(menuID, item_no, check); $--> check == 1 or 0

openresfile	1	refnum := openresfile(resFileName); $ resFileName is a string
closeresfile	1	closeresfile(refnum); $ refnum is an int
getnewdialog	1	dialog := getnewdialog(dialogResId); $ dialogResId is an int
closedialog	1	closedialog(dialog); $ dialog was returned by getnewdialog(..)
getditem	2	[itemType,itemHandle,box]:= getditem(dialog,itemNo); $ dialog and itemHandle are Bignum; box is [x,y,w,h]
setditem	5	setditem(dialog,itemNo,itemType,itemHandle,box); $ dialog and itemHandle are Bignum; box is [x,y,w,h]
getitext	1	atext := getitext(itemHandle); $ itemHandle is a Bignum;
setitext	2	setitext(itemHandle,atext); $ itemHandle is a Bignum;
getctitle	1	atext := getctitle(ControlItemHandle); $ ControlItemHandle is a Bignum;
setctitle	2	setctitle(ControlItemHandle,atext); $ ControlItemHandle is a Bignum;
hidecontrol	1	hidecontrol(ControlItemHandle); $ ControlItemHandle is a Bignum;
showcontrol	1	showcontrol(ControlItemHandle); $ ControlItemHandle is a Bignum;
getctlvalue	1	anInt := getctlvalue(ControlItemHandle); $ ControlItemHandle is a Bignum;
setctlvalue	2	setctlvalue(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;
getctlmin	1	anInt := getctlmin(ControlItemHandle); $ ControlItemHandle is a Bignum;
setctlmin	2	setctlmin(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;
getctlmax	1	anInt := getctlmax(ControlItemHandle); $ ControlItemHandle is a Bignum;
setctlmax	2	setctlmax(ControlItemHandle,anInt); $ ControlItemHandle is a Bignum;
hitdialog	2	hitdialog(dialog,itemHit); $ dialog is a Bignum; itemHit is an int
dialog	1	dialog(dialog); $ dialog is a Bignum; itemHit is an int

macintosh vector graphics		QuickDraw graphics based: an experimental set of primitives, not optimally operational
getobj	3	[x,y,w,h] := getobj(om|'r[ect]'|'e[llipse]', [a,b]|om, scale|0|-1|-2|om); $ create interactively an object: if scale>0: h = w * scale, $if -1: take [a,b] as origin,if -2: take [a,b] as extent
getrect	3	[x,y,w,h] := getrect(om|'r[ect]'|'e[llipse]', [a,b]|om, scale|0|-1|-2|om); $ create interactively a rect: if scale>0: h = w * scale, $if -1: take [a,b] as origin,if -2: take [a,b] as extent
getvector	3	[x1,y1,x2,y2,win_no] := getvector([x1,y1]|om, [x1,y1]|om, len|0|-2|-4|-8); $ create interactively a vector: if len > 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

getpoly	2	[poly,[y1,x1,y2,x2,....yn,xn]] := getpoly(1|0|om, win|om); $ poly is a pict ref (bignum) $ arg is: free-style (1 | om), constrained to manhattan motion : 0 $ moveto(t(2),t(1)); for i in [3,5..#t] do lineto(t(i+1),t(i)); end;
saveimage	2	saveimage(filename|om);$ save image from cur window
restoreimage	2	restoreimage(filename|om);$ restore image to cur window
spextra	1	spextra(x);$ widen spaces in texts by x pixels
DrawPicture	3	DrawPicture(picture, frameRect | om,1|0|om); $ both arg are deref ptrs (bignums) 3rd arg is: acquire (1 or om) or not acquire(0)
erasepict	2	erasepict(picture,win_no | om); $ arg picture: deref ptrs (bignum)
changepicset	2	changepicset(picset,win_no | om); $ picset: set or tuple of picture (deref ptrs (bignum))
getselpict	1	picture := getselpict(win_no | om); $ value returned is deref ptr (bignum)
getportpict	1	picture | [picture,picset] := getportpict(win_no | om); $ value returned is deref ptr (bignum) if no background picset, otherwise a pair:[porpict,picset]
setselpict	2	picture := setselpict(win_no | om, picture); $ arg picture & value returned are deref ptrs (bignums)
setportpict	2	picture := setportpict(win_no | om, picture|picset); $ arg picture & value returned are deref ptrs (bignums)
getselinrect	3	picture := getselinrect(win_no | om, [x,y,w,h] | rect-ref, kind); $ arg rect & value returned are deref ptrs (bignums)-- kind: 0:text, 1:line, 2:rect, 3:oval, 4:arc, 5:poly, nokind: om|-1
getselatpoint	3	picture := getselatpoint(win_no | om, [x,y] | point-as bignum, kind); $ value returned is deref ptr (bignum)-- kind: 0:text, 1:line, 2:rect, 3:oval, 4:arc, 5:poly, nokind: om|-1
movefocusto	2	movefocusto([x,y] | point-as bignum,win_no | om); $ program-controlled scrolling for graphic windows
setorigin	2	setorigin(x,y); $ change coordinates of portRect
moveselectpict	2	picture := moveselectpict(win_no | om, [dh,dv] | point-as bignum); $ value returned is deref ptr (bignum)
movepict	3	movepict(pict,win_no | om, [dh,dv] | point-as bignum); $ superposes moved pict to portpict
translatepict	3	picture := translatepict(picture,win_no | om, [dh,dv] | point-as bignum); $ value returned is deref ptr (bignum)
enumerpict	1	map := enumerpict(picture); $ map = pic_contents (the picture's components)
killpict	1	killpict(picture); $ no automated garbage col. for pictures!
duplpict	1	pic := duplpict(picture); $ duplicate (copy) a picture!
pictrect	1	[x,y,w,h] := pictrect(picture); $ a rectangle enclosing the picture!
clearselpict	1	clearselpict(win_no | om); $ clear window's selpict if any
cutselpict	1	cutselpict(win_no | om); $ cut window's selpict if any
copyselpict	1	copyselpict(win_no | om); $ copy window's selpict if any

pastepict	1	pastepict(win_no | om); $ paste clipboard pict in window
portrect	1	[x,y,w,h] := portrect(win_no | om); $ window's port rectangle
portframe	1	[x,y,w,h] := portframe(win_no | om); $ window's picture frame
viewframe	1	[x,y,w,h] := viewframe(win_no | om); $ window's view frame
screen	0	[x,y,w,h] := screen(); $ screen frame
textbox	4	pic := textbox(text,box,'center'|'right'|'left',win_no | om); $ box is [x,y,w,h]
dragpict	2	[[du,dv],[win_no,moved_pict]] := dragpict (pict, win_no | om); $ drag picture pict in window win_no, returns translation [du,dv] and translated pict
deselect	1	win_no := deselect(win_no | om); $ deselect all items in window
redraw	1	win_no := redraw(win_no | om); $ redraw all items in window
inrect	2	bool := point|rect .inrect r; $ [x,y] .inrect [u,v,w,h];$ [x,y,larg,haut] .inrect [u,v,w,h];

grids and cells		an experimental set of primitives, extending text-pane primitives, not optimally operational
grid_attributes	0	grid_attributes();$ help: the grid attributes in a grid_map
grid_databounds	1	[x,y,w,h] := grid_databounds(grid); $ grid extent, cell range
create_grid_pane	2	grid_no := create_grid_pane(win_no,grid_map);$
add_col	3	add_col(grid,count,col); $ when arg is om use default values: current active pane, count = 1, current select. cell's col
del_col	3	del_col(grid,count,col); $ when arg is om use default values: current active pane, count = 1, current select. cell's col
add_row	3	add_row(grid,count,row); $ when arg is om use default values: current active pane, count = 1, current select. cell's row
del_row	3	del_row(grid,count,row); $ when arg is om use default values: current active pane, count = 1, current select. cell's row
gridcopy	3	n := gridcopy(from,to,om|[x,y]|[x,y,w,h]); $ copy from a text pane to a File the sub-grid starting at cur-sel | [x,y] | limited to [x,y,w,h] (see finsert for converse)
grid_deselect	1	grid_deselect(grid); $ deselect all cells in grid
grid_selflags	2	grid_selflags(grid,int_flags | om); $ modify grid selection flags
selected_cells	2	t := selected_cells(grid,kind|om); $ tuple of all selected cells in grid; kind = om: returns only cell list; kind = 3(int),4(long),8(string): the values are all of same kind; other kinds
grid_select_all	2	grid_select_all(grid,kind|om); $ kind = om: all cells selected; kind /= om: all non-empty cells selected
grid_add_to	3	grid_add_to(grid,x,om|[i,j]); $ append x to grid cell [i,j] or current selection

grid_get	2	x := grid_get(grid,om|[i,j]); $ returns the value of grid cell [i,j] or that of current selection
grid_set	3	grid_set(grid,x,om|[i,j]); $ set grid cell [i,j] or current selection to x

Cantor Global var		these variables have default values OM or { }
CantorDrvrMap		CantorDrvrMap([win_no,event_type]) := aFunc; $ if an event of type *event_type* occurs in window *win_no*, and has for event map *ev*, *aFunc(ev)* is run
MenuBarMap		MenuBarMap(win_no) := aFunc; $ if an event of type EVENT_MENU or EVENT_MENU_BAR occurs in window *win_no*, and with selected item number *isel*, then *aFunc(isel)* is run
StaticMenuDrvrMap		StaticMenuDrvrMap(selected_text) := aFunc; $ if an event of type EVENT_MENU occurs in any static window and selects the string *selected_text*, then *aFunc()* is run
cantor_AlphaNumSet		contains all non standard alphanumeric character (used by scan)

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

recordOutput		!recordOutput $ record all console output in file !recordOutput $ no filename arg --> stop recording
reset		!reset $ reset all program variables. Does'nt reset windows
save		!save filename $ create a prog. component file copy (see load)
suspend		!suspend $ suspend execution: same as interp(om); Resume execution by entering *return;*
time on/off		!time on/off $ enable/disable date display in traces
trace on/off		!trace on/off $ enable/disable Cantor Abstract Machine execution trace
unwatch		!unwatch name $ stop watching the object(s) identified by that name (see watch, breakpoint, allbreakpt())
verbose on/off		!verbose on/off $ enable/disable detailed display of data involved in traces
version		!version $ display current Cantor version
watch		!watch name $ start watching the object(s) identified by that name (see unwatch, breakpoint, allbreakpt())
work_ids		!work_ids $ lists all defined var created by the prog in the current component (see new_ids , oms, ids)

experimental Cantor directives		these are execution or compilation directives
annotate on/off		!annotate on/off $ enable/disable AST annotation
in_debug on/off		!in_debug on/off $ enable/disable graphic driver primitives trace
oldBin		!oldBin $ no longer applicable
oldReal		!oldReal $ for upward compatibility with binaries containing real nbrs in version V0.45 or before

pane & file experiments		
get_stdio	0	[stdin,stdout] := get_stdio(); $--> stdin , stdout : global variables in Cantor
stdin_from	2	stdin_from(paneID, win_no); $ redirects
stdout_to	2	stdout_to(paneID, win_no); $ redirects
get_file_value	1	file_value := get_file_value(file);
get_file_mode	1	file_value := get_file_mode(file);
get_pane_file	1	file_value := get_pane_file(paneID);
get_other_pane_file	2	file_value := get_other_pane_file(paneID, pane_file);

Cantor primitives index

Sorted Item 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
hitdialog
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_func
is_integer
is_map
is_number
is_om
is_pendown
is_set
is_string
is_table
is_textpane
is_tuple
itoa
itoat
killpict
kwd
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
parsetree
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

Index