

Icon programming language quick reference



General conventions

The comment symbol is #; comments extend to the end of the line.

Numeric constants can be written in exponential notation (e.g, $1.7e-4$ for 1.7×10^{-4}) or in the radix notation $\langle radix \rangle r \langle constant \rangle$, e.g., `16r0DOA` for hexadecimal `0DOA`.

String constants are written between double quotes ("`...`"). Escapes include: `\b` backspace, `\e` escape, `\f` formfeed, `\l` linefeed, `\n` newline, `\r` return, `\t` horizontal tab, `\'` single quote, `\"` double quote, `\ddd` for octal code `ddd`, and `\xdd` for hexadecimal code `dd`.

Variable names must start with a letter, followed by letters, digits, or underbar (`_`). They can be any length.

Built-in types

Fundamental types

- **integer**: Signed integers. They may have many digits (thousands, even) but extended-precision arithmetic can be slow.
- **real**: Real numbers.
- **string**: Strings of characters. Any character can be stored in a string, including nulls.
- **cset**: An improper subset of the character set.

Structural types

- **file**: An open file handle.
- **list**: A list of zero or more values of any type or mixture of types. Written as: $[e_1, e_2, \dots, e_n]$
- **null**: The type of variables to which no value is assigned.
- **procedure**: An Icon procedure.
- **table**: An associative array, containing a set of ordered pairs (k, v) where each k is a unique key value and v is any value associated with that key.
- **co-expression**: See the manual.

Reserved words

```
break by case create default do else end every fail global if initial
link local next not of procedure record repeat return static suspend
then to until while
```

Declarations

```
global <name>, ...
record <name>{<field>, ...}# Create with: var := <name>()
local <name>, ...
static <name>, ...
```

Control structures

break <i>e</i>	Exit the enclosing loop and produce <i>e</i>
case <i>n</i> of{ <i>x</i> ₁ : <i>e</i> ₁ <i>x</i> ₂ : <i>e</i> ₂ ... default: <i>e</i> _{<i>d</i>} }	Produces the <i>e</i> _{<i>i</i>} whose <i>x</i> _{<i>i</i>} matches <i>n</i>
create <i>x</i>	Create a co-expression for <i>x</i>
every <i>e</i> ₁ do <i>e</i> ₂	Iterate over outcomes of <i>e</i> ₁
fail	Fail out of the current procedure
if <i>e</i> ₁ then <i>e</i> ₂ [else <i>e</i> ₃]	Produces <i>e</i> ₂ if <i>e</i> ₁ succeeds, else <i>e</i> ₃
next	Go to the top of the loop
not <i>e</i>	Fails if <i>e</i> succeeds
repeat <i>e</i>	Repeat forever (or until break)
return <i>e</i>	Return <i>e</i> to the caller
suspend <i>e</i>	Suspend procedure, return <i>e</i>
until <i>e</i> ₁ do <i>e</i> ₂	Loop while <i>e</i> ₁ is false
while <i>e</i> ₁ do <i>e</i> ₂	Loop while <i>e</i> ₁ is true
<i>e</i> ₁ <i>e</i> ₂	Outcome of <i>e</i> ₁ followed by outcome of <i>e</i> ₂
<i>e</i>	Produce <i>e</i> forever
<i>e</i> \ <i>n</i>	Produce no more than <i>n</i> results from <i>e</i>
<i>s</i> ? <i>e</i>	Establish <i>s</i> as &subject, evaluate <i>e</i>

“Preprocessor” features

\$define <i>n text</i>	Define a textual substitution
\$else	Else construct for \$ifdef and \$ifndef
\$endif	End of conditional compilation
\$error	Signal a fatal compilation error
\$ifdef <i>n</i>	Conditional compilation when <i>n</i> is defined
\$ifndef <i>n</i>	Conditional compilation when <i>n</i> is undefined
\$include " <i>f</i> "	Include file <i>f</i>
\$undef <i>n</i>	Undefine a name

Operators

Precedence

(e) {e;...} [e,...]	Highest precedence
e[e]	
e(e)	
All unary operators	
\ @ !	
^	
* / % **	
+ - ++ --	
All relationals	
e e e to e [by e]	
:= ⊙ :=	where ⊙ is any binary operator
?	
&	
; ,	Lowest precedence

Unary operators

+n	Numeric value of n
-n	Negative of n
?n	Pseudorandom number $\in [1, n]$ for integers, $\in [0, 1)$ for $n \equiv 0$
?x	Randomly selected element of x
~c	Set complement of cset c
=s	tab(match(s))
!x	Sequence of elements from x
@e	Outcome of activating co-expression e
^e	Refresh co-expression e
*x	Size of x
.x	Value of x
/x	Is x undefined?
\x	Is x defined?

Binary operators

+ - * /	Ambition, distraction, uglification, derision
%	Remainder
a^b	a^b
< <= = >= > ~ =	Numerical comparison
++	Set union
--	Set difference
**	Set intersection
	String concatenation
<< <<= == >>= >> ~ ==	Lexical comparison
	Concatenate two lists
x @ e	Activate co-expression e , transmit x to it
:=	Assignment

<code><-</code>	Reversible assignment
<code>:=:</code>	Exchange
<code><-></code>	Reversible exchange
<code>=== ~===</code>	Value comparison
<code>x & y</code>	Produces <code>y</code>
<code>x.y</code>	Structure field reference
<code>x ⊙ := y</code>	Same as <code>x := x ⊙ y</code>
<code>s?e</code>	Establish <code>s</code> as subject within <code>e</code>

Built-in functions

Numeric functions

<code>abs(x)</code>	Absolute value of integer or real
<code>acos(x)</code>	Arccos, returns radians
<code>asin(x)</code>	Arcsin, returns radians
<code>atan(y, x)</code>	Arctan, default 2nd arg is 1
<code>dtor(x)</code>	Degrees to radians
<code>exp(x)</code>	Exponential
<code>cos(x)</code>	Argument in radians
<code>iand(a, b)</code>	Bitwise and
<code>icom(a)</code>	Bitwise not (one's complement)
<code>ior(x, y)</code>	Bitwise or
<code>ishift(x, n)</code>	Shift <code>x</code> left <code>n</code> ; negative <code>n</code> for right shift
<code>ixor(x, y)</code>	Bitwise exclusive or
<code>log(x, b)</code>	$\log_b x$; default <code>b</code> is <code>e</code>
<code>rtod(x)</code>	Radians to degrees
<code>sin(x)</code>	Argument in radians
<code>sqrt(x)</code>	Square root
<code>tan(x)</code>	Argument in radians

String functions

<code>any(c, s, i, j)</code>	Fails if <code>s[i] ∉ c</code> , else produces <code>i</code>
<code>bal(c, c_l, c_r, s, i, j)</code>	Balances <code>c_l, c_r</code> up to <code>c</code>
<code>center(s, n, p)</code>	Centers <code>s</code> in field of <code>n</code> , padded with <code>p</code>
<code>detab(s, i_1, i_2, ..., i_n)</code>	Untabify; default (9)
<code>entab(s, i_1, i_2, ..., i_n)</code>	Tabify
<code>find(t, s, i, j)</code>	Find target <code>t</code> in source <code>s</code>
<code>left(s, n, p)</code>	Flush-left <code>s</code> in field of <code>n</code> , padded with <code>p</code>
<code>many(c, s, i, j)</code>	Scan <code>s</code> while <code>∈ c</code>
<code>map(s, f, t)</code>	Map characters from <code>f</code> to <code>t</code> in <code>s</code> . E.g., to upshift: <code>map(s, &lcase, &ucase)</code>
<code>match(t, s, i, j)</code>	Is target <code>t</code> a prefix of <code>s</code> ?
<code>move(n)</code>	Advance <code>&pos</code> by <code>n</code> , return part skipped
<code>pos(n)</code>	Is <code>&pos</code> at <code>n</code> ?
<code>repl(s, n)</code>	Replicate <code>n</code> copies of <code>s</code>

<code>reverse(s)</code>	Reverse string
<code>right(s, n, p)</code>	Flush right s in field of width n , pad p
<code>tab(n)</code>	Set <code>&pos</code> to n , return string from old <code>&pos</code> to there
<code>trim(s, c)</code>	Trim trailing characters $\in c$
<code>upto(c, s, i, j)</code>	Scan s until a character $\in c$

Structural functions

<code>char(o)</code>	Character whose ordinal is o
<code>copy(x)</code>	Structure copy
<code>cset(x)</code>	Cast to cset
<code>delete(s, x)</code>	Delete element x from set s
<code>get(L)</code>	Pop first element of list L
<code>image(x)</code>	String image of x
<code>insert(s, x)</code>	Insert x into set s
<code>integer(x)</code>	Cast to integer
<code>list(n, x)</code>	List of n elements equal to x
<code>member(s, x)</code>	Set membership: is $x \in s$?
<code>numeric(x)</code>	Cast to integer or real
<code>ord(s)</code>	Ordinal of 1-character strings
<code>pop(L)</code>	Pop first element of list L
<code>pull(L)</code>	Pop last element of list L
<code>push(L, x)</code>	Push x as first element of L
<code>put(L, x)</code>	Push x as last element of L
<code>real(x)</code>	Cast to real
<code>seq(i, j)</code>	Infinite sequence: $i, i + j, i + 2j, \dots$
<code>set(L)</code>	Set of distinct elements in list L
<code>sort(a)</code>	Sort a list
<code>sort(t, k)</code>	Sort table t . Produces a list of 2-elt lists (i, v) of indices and values; sort by i if $k = 1$, by v if $k = 2$.
<code>sortf(x, k)</code>	Sort lists and records, using the k th field as key
<code>string(x)</code>	Cast to string
<code>table(x)</code>	New table with initial value x
<code>type(x)</code>	String describing the type of x

System functions

<code>chdir(s)</code>	Change directory to s
<code>close(f)</code>	Close file
<code>delay(i)</code>	Delay i milliseconds
<code>display(i, f)</code>	Stack traceback with locals, i levels
<code>exit(i)</code>	Terminate with status i
<code>flush(f)</code>	Flush output to file f
<code>function()</code>	Generates names of all builtin functions
<code>getch()</code>	Get next character
<code>getche()</code>	Get next character, no echo
<code>getenv(s)</code>	Value of environmental variable s
<code>kbhit()</code>	Is there a character available for <code>getch()</code> ?

<code>open(name, options)</code>	Open a file: options taken from r (read), w (write), a (append), b (read/write), c (create), t (enable line terminator translation), and u (inhibit line terminator translation).
<code>name(v)</code>	Name of variable <i>v</i>
<code>read(f)</code>	Return next line from <i>f</i>
<code>reads(f, n)</code>	Read next <i>n</i> characters from <i>f</i> , may be short, fails only if no characters remain
<code>remove(s)</code>	Delete file whose name is <i>s</i>
<code>rename(s_o, s_n)</code>	Rename file <i>s_o</i> as <i>s_n</i>
<code>seek(f, p)</code>	Set file position, origin 1; negative = before end
<code>stop(e₁, ..., e_n)</code>	Write the expressions and stop with exit status 1
<code>system(s)</code>	Execute a command line
<code>variable(s)</code>	Returns the variable named <i>s</i>
<code>where(f)</code>	Returns a file position
<code>write(e₁, e₂, ...)</code>	Write expressions, then newline
<code>writes(e₁, e₂, ...)</code>	Write expressions, no newline

Other operations

<code>i to j [by k]</code>	Arithmetic progression generator
<code>e₀(e₁, e₂, ...)</code>	The outcome of <i>e₀</i>
<code>p(e₁, e₂, ...)</code>	Procedure or function call
<code>s[i]</code>	<i>i</i> th element of <i>s</i>
<code>s[i:j]</code>	Elements <i>i</i> through <i>j</i>
<code>s[i+:j]</code>	Elements <i>i</i> through <i>i+j</i>
<code>s[i-:j]</code>	Elements <i>i-j</i> through <i>i</i>
<code>[e₁, e₂, ...]</code>	Form a list from expressions

Keywords (built-in variables)

<code>&ascii</code>	a cset of 7-bit ASCII
<code>&clock</code>	hh:mm:ss
<code>&cset</code>	Full character set, universe for csets
<code>&date</code>	yyyy/mm/dd
<code>&dateline</code>	Sunday, December 7, 1941 7:01 am
<code>&digits</code>	012...9
<code>&e</code>	2.71828...
<code>&errout</code>	Standard error stream
<code>&fail</code>	Fails
<code>&host</code>	Host system identification
<code>&input</code>	Standard input stream
<code>&letters</code>	a...zA...Z
<code>&lcase</code>	a...z
<code>&level</code>	Current procedure call level
<code>&main</code>	Co-expression for the initial call of main()
<code>&null</code>	The null value
<code>&output</code>	Standard output stream

<code>&phi</code>	Golden ratio ϕ , 1.61803...
<code>&pi</code>	π , 3.14159...
<code>&pos</code>	Current scan position
<code>&program</code>	File name of executing program
<code>&random</code>	Pseudorandom seed
<code>&source</code>	Co-expression for the activator of the current co-expression
<code>&subject</code>	Current scan string
<code>&time</code>	Milliseconds of cpu time since program start
<code>&trace</code>	Controls procedure trace: 0 for none, -1 for ∞
<code>&ucase</code>	A..Z
<code>&version</code>	Current Icon version

Written by John W. Shipman (tcc-doc@nmt.edu). This version printed 2003-06-30. Copyright © 2001 by the New Mexico Institute of Mining and Technology.