Unicon Language Reference

Clinton L. Jeffery
Shamim Mohamed
August 5, 2002
Unicon Technical Report #8

Abstract

Unicon is a very high level application programming language with particular strengths in the areas of complex data structure and algorithm development, text processing, graphics and network savvy utilities. This language reference is adapted for free distribution and on-line documentation purposes from Appendix A of “Programming with Unicon” by Jeffery, Mohamed, Pereda, and Parlett.

Introduction

Unicon is expression-based. Nearly everything is an expression, including the common control structures such as while loops. The only things that are not expressions are declarations for procedures, methods, variables, records, classes, and linked libraries.

In the reference, types are listed for parameters and results. If an identifier is used, any type is allowed. For results, generator expressions are further annotated with an asterisk (*****) and non-generators that can fail are annotated with a question mark (?). A question mark by itself is shorthand for null? and denotes a predicate whose success or failure is what matters; the predicate return value (&null) is not significant.

Immutable Types

Unicon's immutable types are integers, real numbers, strings, and csets. Values of these types cannot change. Operators and functions on immutable types produce new values rather than modify existing ones. The simplest expressions are literal values, which occur only for immutable types. A literal value evaluates to itself.

Integer

Integers are of arbitrary precision. Decimal integer literals are contiguous sequences of the digits 0 through 9, optionally preceded by a + or - sign. Radix integer literals use the format *radix*R*digits*, where *radix* is a base in the range 2 through 36, and *digits* consists of one or more numerals in the supplied radix. After values 0-9, the letters A-Z are used for values 10-35. Radix literals are case insensitive, unlike the rest of the language, so the R may be upper or lower case, as may the following alphabetic digits.

Real

Reals are double-precision floating-point values. Real decimal literals are contiguous sequences of the digits 0 through 9, with a decimal point (a period) somewhere within or at either end of the digits. Real exponent literals use the format *number*E*integer*; E may be upper or lower case.

String

Strings are sequences of 0 or more characters, where a character is a value with a platform-dependent size and symbolic representation. On platforms with multi-byte character sets, multiple Icon characters represent a single symbol using a platform-dependent encoding.

String literals consist of 0 or more characters enclosed in double quotes. A string literal may include escape sequences that use multiple characters to encode special characters. The escape sequences are given in Table 3-1. Incomplete string literals may be continued on the next line if the last character on a line is an underscore (_). In that case, the underscore, the newline, and any whitespace at the beginning of the next line are not part of the string literal.

Cset

Csets are sets of 0 or more characters. Cset literals consist of 0 or more characters enclosed in single quotes. As with strings, a cset literal may include escape sequences that use multiple characters to encode special characters.

Mutable Types

Mutable types have values that may be altered. Changes to a mutable value affect its allocated memory or its associated persistent storage. Mutable types include lists, tables, sets, records, objects, and files, including windows, network connections and databases. These types are described in this Appendix in the entries for constructors that create mutable values. Structure types hold collections of elements that may be of arbitrary, mixed type.

List

Lists are dynamically sized, ordered sequences of zero or more values. They are constructed by function, by an explicit operator, or implicitly by a call to a variable argument procedure. They change size by stack and queue functions.

Table

Tables are dynamically sized, unordered mappings from keys to elements. They are constructed by function. The keys may be of arbitrary, mixed type.

Set

Sets are unordered collections. They are constructed by function.

Record

Records are ordered, fixed length sequences of elements that may be accessed via named fields.

Object

Objects are ordered, fixed length sequences of elements that may be accessed via named fields and methods. Field access to object state is a widely deprecated practice.

File

Files are system interface values that correspond to data on secondary storage, areas on users' displays, network connections, or databases. Operations on files are input or output functions that cause side effects on the system outside of the program execution.

Variables

Variables are names for locations in memory where values can be stored. Values are stored in variables by assignment operators. A variable name begins with a letter or underscore, followed by zero or more letters, underscores, or digits. A variable name cannot be the same as one of Icon's reserved words, nor can it be the same as one of Icon's keywords if it follows an adjacent ampersand character. Variables can hold values

of any type, and may hold different types of values at different times during program execution.

There are three kinds of variables: global, local, and static. Variables are declared by introducing one of the three reserved words (global, local, or static) followed by a comma-separated list of variable names. Global variables are declared outside of any procedure or method body, while local and static variables are declared at the beginning of procedure and method bodies.

Aliasing occurs when two or more variables refer to the same value, such that operations on one variable might affect the other. Aliasing is a common source of program bugs. Variables holding integer, real, string, or cset values are never aliased, because those types are immutable.

Global

Global variables are visible everywhere in the program, and exist at the same location for the entire program execution. Declaring a procedure declares a global variable and preinitializes it to the procedure value that corresponds to the code for that procedure.

Local

Local variables are visible only within a single procedure or method, and exist at any particular location only for the duration of a single procedure invocation, including suspensions and resumptions, until the procedure returns, fails, or is *vanquished* by the return or failure of one of its ancestor invocations. Variables that are undeclared in any scope are implicitly local, but this dangerous practice is deprecated and should be avoided in large programs.

Variables that are declared as *parameters* are local variables that are preinitialized to the values of actual parameters at the time of a procedure or method invocation. The semantics of parameter passing are the same as those of assignment.

Static

Static variables are visible only within a single procedure or method, but exist at the same location for the entire program execution. The value stored in a static variable is preserved between multiple calls to the procedure in which it is declared.

Class

Class variables are visible within the methods of a declared class. A set of class variables is created for each instance (object) of the class. The lifespan of class variables is the same as the life span of the instance to which they belong. The value stored in a class variable is preserved between multiple calls to the methods of the class in which it is declared.

Keywords

Keywords are names with global scope and special semantics within the language. They begin with an ampersand character. Some keywords are names of common constant values, while others are names of variables that play a special role in Icon's control

structures. The name of the keyword is followed by a : if it is read-only, or a := if it is a variable, followed by the type of value the keyword holds.

&allocated : integer* report memory use

&allocated generates the cumulative number of bytes allocated in heap, static, string, and block regions during the entire program execution.

&ascii : cset ASCII character set

&ascii produces a cset corresponding to the ASCII characters.

&clock : string time of day

&clock produces a string consisting of the current time of day in hh:mm:ss format. See also keyword &now.

&collections : integer* garbage collection activity

&collections generates the number of times memory has been reclaimed in heap, static, string, and block regions.

&column : integer source code column

&column returns the source code column number of the current execution point. This is especially useful for execution monitoring.

&cset : cset universal character set

&cset produces a cset constant corresponding to the universal set of all characters.

&current :co-expression current co-expression

&current produces the co-expression that is currently executing.

&date : string today's date

&date produces the current date in yyyy/mm/dd format.

&dateline : string time stamp

&dateline produces a human-readable timestamp that includes the day of the week, the date, and the current time, down to the minute.

&digits : cset digit characters

&digits produces a cset constant corresponding to the set of digit characters 0-9.

&dump := integer termination dump

&dump controls whether the program dumps information on program termination or not. If &dump is nonzero when the program halts, a dump of local and global variables and their values is produced.

&e : real natural log e

&e is the base of the natural logarithms, 2.7182818?

&error := integer — fail on error

&error controls whether runtime errors are converted into expression failure. By assigning to this keyword, error conversion can be enabled or disabled for specific sections of code. The integer &error is decremented by one on each error, and if it reaches zero, a runtime error is generated. Assigning a value of -1 effectively disables runtime errors indefinitely. See also `&syserr`.

&errornumber : integer? — runtime error code

&errornumber is the error number of the last runtime error that was converted to failure, if there was one.

&errortext : string? — runtime error message

&errortext is the error message of the last error that was converted to failure.

&errorvalue : any? — offending value

&errorvalue is the erroneous value of the last error that was converted to failure.

&errout : file — standard error file

&errout is the standard error file. It is the default destination to which runtime errors and program termination messages are written.

&eventcode := integer — program execution event

&eventcode indicates the kind of behavior that occurred in a monitored program at the time of the most recent call to EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&eventsource := co-expression — source of program execution events

&eventsource is the co-expression that transmitted the most recent event to the current program. This keyword is null unless the program is an execution monitor. See also &source. Under a monitor coordinator, &eventsource is the coordinator and global variable Monitored is the target program.

&eventvalue := any — program execution value

&eventvalue is a value from the monitored program that was being processed at the time of the last program event returned by EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&fail : none — expression failure

&fail never produces a result. Evaluating it always fails.

&features : string* — platform features

&features generates strings that indicate the nonportable features supported on the current platform.

&file : string? current source file

&file is the name of the source file for the current execution point, if there is one. This is especially useful for execution monitoring.

&host : string host machine name

&host is a string that identifies the host computer Icon is running on.

&input : file standard input file

&input is a standard input file. It is the default source for file input functions.

&lcase : cset lowercase letters

&lcase is a cset consisting of the lowercase letters from a to z.

&letters : cset letters

&letters is a cset consisting of the upper and lowercase letters A-Z and a-z.

&level : integer call depth

&level gives the nesting level of the currently active procedure call. This keyword is not supported under the optimizing compiler, iconc.

&line : integer current source line number

&line is the line number in the source code that is currently executing.

&main : co-expression main task

&main is the co-expression in which program execution began.

&now : integer current time

&now produces the current time as the number of seconds since the epoch beginning 00:00:00 GMT, January 1, 1970. See also &clock

&null : null null value

&null produces the null value.

&output : file standard output file

&output is the standard output file. It is the default destination for file output.

&phi : real golden ratio

&phi is the golden ratio, 1.618033988...

&pi : real pi

&pi is the value of pi, 3.141592653...

&pos := integer — string scanning position

&pos is the position within the current subject of string scanning. It is assigned implicitly by entering a string scanning environment, moving or tabbing within the environment, or assigning a new value to &subject. &pos may not be assigned a value that is outside the range of legal indices for the current &subject string.

&progname := string — program name

&progname is the name of the current executing program.

&random := integer — random number seed

&random is the seed for random numbers produced by the random operator, unary ?

&regions : integer* — region sizes

&regions produces the sizes of the static region, the string region, and the block region. The first result is always zero and is included for backward compatibility reasons.

&source : co-expression — invoking co-expression

&source is the co-expression that activated the current co-expression.

&storage : integer* — memory in use

&storage gives the amount of memory currently used within the static region, the string region, and the block region. The first result is always zero and is included for backward compatibility reasons.

&subject := string — string scanning subject

&subject holds the default value used in string scanning and analysis functions. Assigning to &subject implicitly assigns the value 1 to &pos.

&syserr := integer — halt on system error

&syserr controls whether a system error causes execution to halt with a runtime error. System errors cause expression failure by default. If &syserr is set to a non-zero value, system errors are converted to runtime errors and halt execution with an error traceback.

&time : integer — elapsed time

&time gives the number of milliseconds of CPU time that have elapsed since the program execution began. For wall clock time see &now or &clock.

&trace := integer — trace program

&trace indicates the number of nesting levels to which the program execution should be traced. 0 means no tracing. A negative value turns on tracing to an infinite depth. &trace is set outside the program using the TRACE environment variable or the -t compiler option.

&ucase : cset — upper case letters

&ucase is a cset consisting of all the upper case letters from A to Z.

&version : string version

&version is a string that indicates which version of Unicon or Icon is executing.

Graphics Keywords

Graphics keywords are described in [Griswold98]. They are listed here so that you will know to consult that book when reading Icon programs that use graphics.

&col := integer	column location of pointer
&control : ?	control key state
&interval : integer	time between input events
&ldrag : integer	left button drag
&lpress : integer	left button press
&lrelease : integer	left button release
&mdrag : integer	middle button drag
&meta : ?	meta key state
&mpress : integer	middle button press
&mrelease : integer	middle button release
&rdrag : integer	right button drag
&resize : integer	window resize
&row := integer	row location of pointer
&rpress : integer	right button press
&rrelease : integer	right button release
&shift : ?	shift key state
&window := window	standard window
&x := integer	pointer horizontal position
&y := integer	pointer vertical position

Control Structures and Reserved Words

Icon has many reserved words. Some are used in declarations, but most are used in control structures. This section summarizes the syntax and semantics introduced by all the reserved words of the language. The reserved word under discussion is written in a bold font. The surrounding syntax uses square brackets for optional items and an asterisk for items that may repeat.

break *expr* exit loop

The break expression exits the nearest enclosing loop. *expr* is evaluated and treated as the result of the entire loop expression. If *expr* is another break expression, multiple loops will be exited.

expr1 to *expr2* **by** *expr3* step increment

The by reserved word supplies a step increment to a to-expression (the default is 1).

case *expr* of { ? } select expression

The case expression selects one of several branches of code to be executed.

class name [: superclass]* (fields) methods [initially] end — class declaration

The class declaration introduces a new object type into the program. The class declaration may include a list of superclasses, fields, methods, and an optional initially section.

---

create *expr* — create co-expression

The create expression produces a new co-expression to evaluate *expr*.

---

default : *expr* — default case branch

The default branch of a case expression is taken if no other case branch is taken.

---

do *expr* — iteration expression

The do reserved word specifies an expression to be executed for each iteration of a preceding while, every, or suspend loop (yes, suspend is a looping construct).

---

if expr1 then expr2 **else** expr3 — else branch

The else expression is executed if expr1 fails to produce a result.

---

end — end of declared body

The reserved word end signifies the end of a procedure, method, or class body.

---

every *expr1* [do *expr2*] — generate all results

The every expression always fails, causing expr1 to be resumed for all its results.

---

fail — produce no results

The fail reserved word causes the enclosing procedure or method invocation to terminate immediately and produce no results. The invocation may not be resumed. See also the keyword &fail, which produces a less drastic expression failure. fail is equivalent to return &fail

---

global *var* [, *var*]* — declare global variables

Reserved word global introduces one or more global variables.

---

if *expr* then *expr2* [else *expr3*] — conditional expression

The if expression evaluates expr2 only if expr1 produces a result.

---

import *name* [, *name*]* — import package

The import declaration introduces the names from package *name* so that they may be used without prefixing them with the package name.

---

initial *expr* — execute on first invocation

The initial expression is executed the first time a procedure or method is invoked.

initially [(parameters)] initialize object

The initially section defines a special method that is invoked automatically when an object is created. If the initially section has declared parameters, they are used as the parameters of the constructor for objects of that class.

invocable *procedure* [, *procedure*]* allow string invocation
invocable all allow string invocation

The invocable declaration indicates that procedures may be used in string invocation.

link *filename* [, *filename*]* link code module

The link declaration directs that the code in *filename* will be added to the executable when this program is linked. *filename* may be an identifier or a string literal file path.

local *var* [, *var*]* declare local variables

The local declaration introduces one or more local variables into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

method *name* (*params*) *body* end declare method

The method declaration introduces a procedure that is only invoked with respect to instances of an enclosing class declaration. The *params* and *body* are as in procedures, described below.

next *expr* iterate loop

The next expression causes a loop to immediate skip to its next iteration.

not *expr* negate expression failure

The not expression fails if expr succeeds, and succeeds (producing null) if expr fails.

case *expr* **of** { ? } introduce case branches

The of reserved word precedes a special compound expression consisting if a sequence of case branches of the form *expr* : *expr*. Case branches are evaluated in sequence until one is found that matches the expression given between the word case and the of.

package *name* declare package

The package declaration segregates the global names in the current source file. In order to refer to them, client code must either import the package, or prepend *name* . (the package name followed by a period) onto the front of a name in the package.

procedure *name* (*params*) *body* end declare procedure

The procedure declaration specifies a procedure with the specified parameters and code body. The parameters are a comma-separated list of zero or more variable names. The last parameter may optionally be suffixed by [] indicating that any following parameters will be supplied to the procedure in a list. The body is an optional sequence of local and static variable declarations, followed by a sequence of zero or more expressions.

record *name* (*fields*) declare record

The record declaration introduces a new record type into the program.

repeat *expr* infinite loop

The repeat expression introduces an infinite loop that will reevaluate *expr* forever. Of course, *expr* may exit the loop or terminate the program in any number of ways.

return *expr* return from invocation

The return expression exits a procedure or method invocation, producing *expr* as its result. The invocation may not be resumed.

static *var* [, *var*]* declare static variables

The static declaration introduces one or more local variables into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

suspend *expr* [do *expr*] produce result from invocation

The suspend expression produces one or more results from an invocation for use by the calling expression. The procedure or method may be resumed for additional results if the calling expression needs them. Execution in the suspended invocation resumes where it left off, in the suspend expression. A single evaluation of a suspend expression may produce multiple results for the caller if *expr* is a generator. An optional do expression is evaluated each time the suspend is resumed.

if *expr1* **then** *expr2* conditional expression

The *expr2* following a then is evaluated only if *expr1* following an if succeeds. In that case, the result of the whole expression is the result of *expr2*.

expr1 **to** *expr2* generate arithmetic sequence

The to expression produces the integer sequence from *expr1* to *expr2*.

until *expr1* [do *expr2*] loop until success

The until expression loops as long as *expr1* fails.

while *expr1* [do *expr2*] loop until failure

The while expression loops as long as *expr1* succeeds.

Operators and Built-in Functions

Icon's built-ins operators and functions utilize automatic type conversion to provide flexibility and ease of programming. Automatic type conversions are limited to integer, real, string, and cset data types. Conversions to a "number" will convert to either an integer or a real, depending whether the value to be converted has a decimal. Conversions between numeric types and csets go through an intermediate conversion to a string value and are not generally useful.

Indexes start at 1. Index 0 is the position after the last element of a string or list. Negative indexes are positions relative to the end. Subscripting operators and string analysis functions can take two indices to specify a section of the string or list. When two indices are supplied, they select the same string section whether they are in ascending or descending order.

Operators

The result types of operators are the same as the operand types except as noted.

Unary Operators

! *x* : any* — generate elements

The generate operator produces the elements of x. It is a generator. If x is a string variable or refers to a structure value, the generated elements are variables that may be assigned. !i is equivalent to (1 to i) for integer i. List, record, string, and file elements are generated in order, with string elements consisting of one-letter substrings. Set and table elements are generated in an undefined order. If x is a messaging connection to a POP server, !x produces complete messages as strings. Other types of files, including network connections, produce elements consisting of text lines.

/ x — null test
\ x — nonnull test

The null and nonnull tests succeed and produce their operand if it satisfies the test.

- number — negate
+ number — numeric identity

Negation reverses the sign of its operand. Numeric identity does not change its operand's value other than to convert to a required numeric type.

= string — tab/match

The tab/match operator is equivalent to calling tab(match(s)) on its operand.

---

* x : integer — size

The size operator returns the number of elements in structure x.

. x : x — dereference

The dereference operator returns the value x.

? x : any — random element

The random operator produces a random element from structure x. The result is a variable that may be assigned. If x is a string, ?x produces a random one-letter substring.

| x : x* — repeated alternation

The repeated alternation operator generates results from evaluating its operand over and over again in an infinite loop.

~ cset — cset complement

The complement operator produces a cset consisting of all characters not in its operand.

^ co-expression — refresh co-expression

The refresh operator restarts a co-expression so the next time it is activated it will begin with its first result.

Binary Operators

Most binary operators may be augmented with an assignment. If such an operator is followed by a := the left operand must be a variable, and the expression `x` *op*`:= y` is equivalent to `x := x` *op* `y`. For example, `x +:= 5` is equivalent but faster than the expression `x := x+5.`

number1 ^ *number2*	power
number1 * *number2*	multiply
number1 / *number2*	divide
number1 % *number2*	modulo
number1 + *number2*	add
number1 - *number2*	subtract

The arithmetic operators may be augmented.

set1 ** set2	intersection
set1 ++ set2	union
set1 -- set2	difference

The set operators work on sets or csets. They may be augmented.

x . name	field
object . name (params)	method invocation
object $ superclass .name (params)	superclass method invocation

The field operator selects field name out of a record, object, or package. For objects, *name* may be a method, in which case the field operator is being used as part of a method invocation. Superclass method invocation consists of a dollar sign and superclass name prior to the field operator.

number1 = number2	equal
number1 ~= number2	not equal
number1 < number2	less than
number1 <= number2	less or equal
number1 > number2	greater than
number1 >= number2	greater or equal
string1 == string2	string equal
string1 ~== string2	string not equal
string1 << string2	string less than
string1 <<= string2	string less or equal
string1 >> string2	string greater than
string1 >>= string2	string greater or equal
x1 === x2	equivalence
x1 ~=== x2	non equivalence

Relational operators produce their right operand if they succeed. They may be augmented.

var := expr	assign
var1 :=: var2	swap
var <- expr	reversible assignment
var1 <-> var2	reversible swap

The several assignment operators all require variables for their left operands, and swap operators also require variables for their right operands.

string ? expr	scan string

The string scanning operator evaluates *expr* with &subject equal to string and &pos starting at 1. It may be augmented.

[x] @ co-expression	activate co-expression

The activate operator transfers execution control from the current co-expression to its right operand co-expression. The transmitted value is x, or &null if no left operand is supplied. Activation may be augmented.

string1 || string2	concatenation
list1 ||| list2	list concatenation

The concatenation operators produce new values containing a copy of the left operand followed by a copy of the right operand. They may be augmented.

x1 & x2	conjunction
expr1 | expr2	alternation

The conjunction operator produces x2 if x1 succeeds. The alternation operator produces the results of expr1 followed by the results of expr2; it is a generator. These operators may be augmented.

x1 \ integer	limitation

The limitation operator fails if it is resumed after its left operand has produced a number of results equal to its right operand.

(expr [, expr]*)	mutual evaluation
p (expr [, expr]*)	invocation

By themselves, parentheses are used to override operator precedence in surrounding expressions. A comma-separated list of expressions is evaluated left to right, and fails if any operand fails. Its value is the right of the rightmost operand.

When preceded by an operand, parentheses form an invocation. The operand may be a procedure, a method, a string that is converted to a procedure name, or an integer that selects the parameter to use as the result of the entire expression.

[ ]	empty list creation
[expr [, expr]* **]**	list creation
expr1 **[** expr2 [, expr]* **]**	subscript
expr1 **[** expr2 : expr3 **]**	subsection
expr1 **[** expr2 +: expr3 **]**	forward relative subsection
expr1 **[** expr2 -: expr3 **]**	backward relative subsection

With no preceding operand, square brackets create and initialize lists. When preceded by an operand, square brackets form a subscript or subsection. Multiple comma-separated subscript operands are equivalent to separate subscript operations with repeating square brackets, so x[y,z] is equivalent to x[y][z].

Subscripting selects an element from a structure and allows that element to be assigned or for its value to be used. Lists and strings are subscripted using 1-based integer indices, tables are subscripted using arbitrary keys, and records may be subscripted by either string fieldname or 1-based integer index. Message connections may be subscripted by string header to obtain server responses; POP connections may also be subscripted by 1-based integer message numbers.

Subsectioning works on strings and lists. For strings, the subsection is a variable if the string was a variable, and assignment to the subsection modifies the original variable. For lists, a subsection is a new list that contains a copy of the elements from the original list.

expr1 ; expr2	bound expression

A semicolon bounds expr1. Once expr2 is entered expr1 cannot be resumed for more results. The result of expr2 is the result of the entire expression. Semicolons are automatically inserted at ends of lines wherever it is syntactically allowable to do so. This results in many *implicitly bounded* expressions.

{ expr [; expr]* }	compound expression
p { expr [; expr]* }	programmer defined control structure

Curly brackets typically cause a sequence of bounded expressions to be treated as a single expression. Preceded by a procedure value, curly brackets introduce a programmer defined control structure in which a co-expression is created for each argument; the procedure is called with these co-expressions as its parameters, and can determine for itself whether, and in what order, to activate its parameters to obtain values.

Built-in Functions

Unicon's built-in functions are a key element of its ease of learning and use. They provide substantial functionality in a consistent and memorizable manner.

In addition to automatic type conversion, built-in functions make extensive use of optional parameters with default values. Default values are indicated in the function descriptions, with the exception of string scanning functions. String scanning functions end with three parameters that default to the string `&subject`, the integer `&pos`, and the end of string (0) respectively. The position argument defaults to 1 when the string argument is supplied rather than defaulted.

---

abs(number) : number — absolute value

abs(N) produces the maximum of N or -N.

---

acos(real) : real — arc cosine

acos(r1) produces the arc cosine of r1. The argument is given in radians.

---

any(cset, string, integer, integer) : integer? — cset membership

String scanning function any(c,s,i1,i2) produces i1+1 if s[i1:i2][1] is in cset c, but fails otherwise.

---

args(procedure) : integer — number of arguments

args(p) produces the number of arguments expected by p. If p takes a variable number of arguments, args(p) returns a negative number to indicate that the final argument is a list conversion of an arbitrary number of arguments. For example, args(p) for a procedure p with formal parameters (x, y, z[]) returns a -3.

---

asin(real) : real — arc sine

asin(r1) produces the arc sine of r1. The argument is given in radians.

---

atan(real, real:1.0) : real — arc tangent

atan(r1) produces the arc tangent of r1. atan(r1,r2) produces the arc tangent of r1 and r2. Arguments are given in radians.

---

bal(cset:&cset, cset:'(', cset:')', string, integer, integer) : integer* — balance string

String scanning function bal(c1,c2,c3,s,i1,i2) generates the integer positions in s at which a member of c1 in s[i1:i2] is balanced with respect to characters in c2 and c3.

---

center(string, integer:1, string:" ") : string — center string

center(s1,i,s2) produces a string of i characters. If i > *s1 then s1 is padded equally on the left and right with s2 to length i. If i < *s1 then the center i characters of s1 are produced.

---

char(integer) : string — encode character

char(i) produces a string consisting of the character encoded by integer i.

---

chmod(f, m) : ? — file permissions

chmod(f, m) sets the access permissions ("mode") of a string filename (or on UNIX systems, an open file) f to a string or integer mode m. The mode indicates the change to be performed. The string is of the form

[ugoa]*[+-=][rwxRWXstugo]*

The first group describes the set of mode bits to be changed: u is the owner set, g is the group and o is the set of all others. The character a designates all the fields. The operator (+-=) describes the operation to be performed: + adds a permission, - removes a permission, and = sets a permission. The permissions themselves are:

r	read
w	write
x	execute
R	read if any other set already has r
W	write if any other set already has w
X	execute if any other set already has x
s	setuid (if the first part contains u and/or setgid if the first part contains g
t	sticky if the first part has o
u	the u bits on the same file
g	the g bits on the same file
o	the o bits on the same file

If the first group is missing, then it is treated as "all" except that any bits in the user's umask will not be modified in the mode. Not all platforms make use of all mode bits described here; the mode bits that are used is a property of the filesystem on which the file resides.

---

close(file) : file | integer — close file

close(f) closes file, pipe, window, network or message connection, or database f and returns any resources associated with it to the operating system. If f was a pipe or network connection, close() returns the integer exit status of the connection, otherwise it returns the closed file.

---

cofail(co-expression) : any — transmit co-expression failure

cofail(CE) causes the current activation of co-expression CE to fail.

---

collect(integer:0, integer:0) : null — collect garbage

collect(i1,i2) calls the garbage collector to ensure i2 bytes in region i1. i1 can be 0 (no region in particular) 1 (static region) 2 (string region) or 3 (block region).

---

copy(any) : any — copy value

copy(x) produces a copy of x. For immutable types (numbers, strings, csets, procedures) this is a no-op. For mutable types (lists, tables, sets, records, objects) a one-level deep copy of the object is made.

---

cos(real) : real — cosine

cos(r1) produces the cosine of r1. The argument is given in radians.

---

cset(any) : cset? — convert to cset

cset(x) converts x to a cset, or fails if the conversion cannot be performed.

ctime(integer) : string — format a time value into local time

ctime(i) converts an integer time given in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in the local timezone. See also keywords &clock and &dateline.

dbcolumns(database,string) : list — ODBC column information

dbcolumns(db,tablename) produces a list of record (catalog, schema, tablename, colname, datatype, typename, colsize, buflen, decdigits, numprecradix, nullable, remarks) entries. Datatype and typename are SQL-dependent and data source dependent, respectively. Colsize gives the maximum length in characters for SQL_CHAR or SQL_VARCHAR columns.. Decdigits gives the number of significant digits right of the decimal. Numprecradix specifies whether colsize and decdigits are specified in bits or decimal digits.. Nullable is 0 if the column does not accept null values, 1 if it does accept null values, and 2 if it is not known whether the column accepts null values.

dbdriver(database) : record — ODBC driver information

dbdriver(db) produces a record driver(name, ver, odbcver, connections, statements, dsn) that describes the details of the ODBC driver used to connect to database db. Connections and statements are the maximums the driver can support. Ver and odbcver are the driver and ODBC version numbers. Name and dsn are the driver filename and Windows Data Source Name associated with the connection.

dbkeys(database,string) : list — ODBC key information

dbkeys(db,tablename) produces a list of record (columnname, sequencenumber) pairs containing information about the primary keys in tablename.

dblimits(database) : record — ODBC operation limits

dblimits(db) produces a record (maxbinlitlen, maxcharlitlen, maxcolnamelen, maxgroupbycols, maxorderbycols, maxindexcols, maxselectcols, maxtblcols, maxcursnamelen, maxindexsize, maxrownamelen, maxprocnamelen, maxqualnamelen, maxrowsize, maxrowsizelong, maxstmtlen, maxtblnamelen, maxselecttbls, maxusernamelen) that contains the upper bounds of the database for many parameters.

dbproduct(database) : record — database name

dbproduct(db) produces a record (name, ver) that gives the name and the version of the DBMS product containing db.

dbtables(database) : list — ODBC table information

dbtables(db) returns a list of record (qualifier, owner, name, type, remarks) entries that describe all of the tables in the database associated with db.

delay(integer) : null — delay for i milliseconds

delay(i) pauses the program for at least i milliseconds.

delete(x1, x2, ...) : x1 — delete element

delete(x1, x2) deletes elements denoted by the 2nd and following parameters from set, table, list, DBM database, or POP connection x1 if it is there. In any case, it returns x1. If x1 is a table or set, elements xi denote keys of arbitrary type. If x1 is a DBM database, indices must be strings. If x1 is a list or a POP messaging connection, elements xi are integer indices of the element to be deleted. POP messages are actually deleted when the close() operation closes that connection.

---

detab(string, integer:9,...) : string — replace tabs

detab(s,i,...) replaces tabs with spaces, with stops at columns indicated by the second and following parameters, which must all be integers. Tab stops are extended infinitely using the interval between the last two specified tab stops.

---

display(integer:&level, file:&errout, co-expression:&current) : null — write variables

display(i,f) writes the local variables of i most recent procedure activations, plus global variables, to file f.

---

dtor(real) : real — convert degrees to radians

dtor(r1) produces the equivalent of r1 degrees, expressed in radians.

---

entab(string, integer:9,...) : string — replace spaces

entab(s,i,...) replaces spaces with tabs, with stops at columns indicated. Tab stops are extended infinitely using the interval between the last two specified tab stops.

---

errorclear() : null — clear error condition

errorclear() resets keywords &errornumber, &errortext, and &errorvalue to indicate that no error is present.

---

event(integer, any, co-expression) : any — transmit event

event(x, y, C) transmits an event with event code x and event value y to a monitoring co-expression C.

---

eventmask(co-expression, cset) : cset | null — get/set event mask

eventmask(ce) returns the event mask associated with the program that created ce, or &null if there is no eventmask. eventmask(ce,cs) sets that program's event mask to cs.

---

exit(integer:*normalexit*) — exit process

exit(i) terminates the current program execution, returning status code i. The default is the platform-dependent exit code that indicates normal termination (0 on most systems).

---

exp(real) : real — exponential

exp(r) produces the result of &e ^ r.

fetch(database, string) : string | row? — fetch database value

fetch(d, k) fetches the value corresponding to key k from a DBM or SQL database d. The result is a string (for DBM databases) or a row (for SQL databases). For SQL databases, when the string k is omitted, fetch(d) produces the next row in the current selection, and advances the cursor to the next row. A row is a record whose field names and types are determined by the columns specified in the current query. fetch(d) fails if there are no more rows to return from the current query. Typically a call to dbselect() will be followed by a while-loop that calls fetch() repeatedly until it fails.

---

fieldnames(record) : string* — get field names

fieldnames(r) produces the names of the fields in record r.

---

find(string, string, integer, integer) : integer* — find string

String scanning function find(s1,s2,i1,i2) generates the positions in s2 at which s1 occurs as a substring in s2[i1:i2].

---

flock(file, string) : ? — apply or remove file lock

flock(f,s) applies an advisory lock to the file. Advisory locks enable processes to cooperate when accessing a shared file, but do not enforce exclusive access. The following characters can be used to make up the operation string:

- s shared lock
- x exclusive lock
- b don't block when locking
- u unlock

Locks cannot be applied to windows, directories or database files. A file may not simultaneously have shared and exclusive locks.

---

flush(file) : file — flush file

flush(f) flushes all pending or buffered output to file f.

---

function() : string* — name the functions

function() generates the names of the built-in functions.

---

get(list) : any? — get element from queue

get(L) returns an element which is removed from the head of the queue L.

---

getch() : string? — get character from console

getch() waits for (if necessary) and returns a character typed at the keyboard, even if standard input was redirected. The character is not displayed.

---

getche() : string? — get and echo character from console

getche() waits for (if necessary) and returns a character typed at the console keyboard, even if standard input was redirected. The character is echoed to the screen.

getenv(string) : string? — get environment variable

getenv(s) returns the value of environment variable s from the operating system.

gettimeofday() : record — time of day

Returns the current time in seconds and microseconds since the epoch, Jan 1, 1970 00:00:00. The sec value may be converted to a date string with ctime or gtime. See also keywords &now, &clock, and &dateline.

Return value: record posix_timeval(sec, usec)

globalnames(co-expression) : string* — name the global variables

globalnames(ce) generates the names of the global variables in the program that created co-expression ce.

gtime(integer) : string — format a time value into UTC

Converts an integer time in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in Coordinated Universal Time (UTC).

iand(integer, integer) : integer — bitwise and

iand(i1, i2) produces the bitwise and of i1 and i2.

icom(integer) : integer — bitwise complement

icom(i) produces the bitwise complement (one's complement) of i.

image(any) : integer — string image

image(x) returns the string image of the value x.

insert(x1, x2, x3:&null) : x1 — insert element

insert(x1, x2, x3) inserts element x2 into set, table, or list or DBM database x1 if not already there. Unless x1 is a set, the assigned value for element x2 is x3. For lists, x2 is an integer index; for other types, it is a key. insert() always succeeds and returns x1.

integer(any) : integer? — convert to integer

integer(x) converts value x to an integer, or fails if the conversion cannot be performed.

ior(integer, integer) : integer — bitwise or

ior(i1, i2) produces the bitwise or of i1 and i2.

ishift(integer, integer) : integer — bitwise shift

ishift(i, j) produces the value obtained by shifting i by j bit positions. Shifting is to the left if j<0, or to the right if j>0. j zero bits are introduced at the end opposite the shift direction.

ixor(integer, integer) : integer — bitwise xor

ixor(i1, i2) produces the bitwise exclusive or of i1 and i2.

kbhit() : ? — check for console input

kbhit() checks to see if there is a keyboard character waiting to be read.

key(x) : any* — table keys

key(T) generates the key (entry) values from table T. key(L) generates the indices from 1 to *L in list L. key(R) generates the string fieldnames of record R.

left(string, integer:1, string:" ") : string — left format string

left(s1,i,s2) formats s1 to be a string of length i. If s1 is more than i characters, it is truncated. If s1 is fewer than i characters it is padded on the right with as many copies of s2 as needed to increase it to length i.

list(integer:0, any:&null) : list — create list

list(i, x) creates a list of size i, in which all elements have the initial value x. If x is a mutable value such as a list, all elements refer to the *same* value, not a separate copy of the value for each element.

load(string, list, file:&input, file:&output, file:&errout, int, int, int) : co-expression — load Unicon program

load(s,arglist,input,output,error,blocksize,stringsize,stacksize) loads the icode file named s and returns that program's execution as a co-expression ready to start its main() procedure with parameter arglist as its command line arguments. The three file parameters are used as that program's &input, &output, and &errout. The three integers are used as its initial memory region sizes.

loadfunc(string, string) : procedure — load C function

loadfunc(filename,funcname) dynamically loads a compiled C function from the object library file given by filename. funcname must be a specially written interface function that handles Icon data representations and calling conventions.

localnames(co-expression, integer) : string* — local variable names

localnames(ce,i) generates the names of local variables in co-expression ce, i levels up from the curent procedure invocation.

log(real, real:&e) : real — logarithm

log(r1,r2) produces the logarithm of r1 to base r2.

many(cset, string, integer, integer) : integer? — many characters

String scanning function many(c,s,i1,i2) produces the position in s after the longest initial sequence of members of c within s[i1:i2].

map(string, string:&ucase, string:&lcase) : string — map string

map(s1,s2,s3) maps s1, using s2 and s3. The resulting string will be a copy of s1, with the exception that any of s1's characters that appear in s2 are replaced by characters at the same position in s3.

match(string, string, integer, integer) : integer — match string

String scanning function match(s1,s2,i1,i2) produces i1+*s1 if s1==s2[i1+:*s1], but fails otherwise.

member(x, ...) : x? — test membership

member(x, ...) returns x if its second and subsequent arguments are all members of set, cset, list or table x but fails otherwise. If x is a cset, all of the characters in subsequent string arguments must be present in x in order to succeed.

mkdir(string, mode) : ? — create directory

mkdir(path, mode) creates a new directory named path with mode **mode**. The mode can be numeric or a string of the form accepted by chmod().

move(integer:1) : string — move scanning position

move(i) moves &pos i characters from the current position and returns the substring of &subject between the old and new positions. This function reverses its effects by resetting the position to its old value if it is resumed.

name(v, co-expression:&current) : string — variable name

name(v) returns the name of variable v within the program that created co-expression c. Keyword variables are recognized and named correctly. name() returns the base type and subscript or field information for variables that are elements within other values, but does not produce the source code variable name for such variables.

numeric(any) : number — convert to number

numeric(x) produces an integer or real number resulting from the type conversion of x, but fails if the conversion is not possible.

open(string, string:"rt", ...) : file? — open file

open(s1, s2, ...) opens a file named s1 with mode s2 and attributes given in trailing arguments. The modes recognized by open() are:

"r"	open the file for reading
"w"	open the file for writing
"a"	open the file for appending
"b"	open the file for both reading and writing (b does not mean binary!)
"p"	execute a program given by command line s1 and open a pipe to it
"c"	create a new file and open it
"t"	open the file in text mode, with newlines translated
"u"	open the file in a binary untranslated mode
"na"	listen on a TCP network socket
"n"	connect to a TCP network socket
"nau"	listen on a UDP network socket
"nu"	connect to a UDP network socket
"d"	open a DBM database

"m" connect to a messaging server (HTTP, SMTP, POP, ...)
"o" open an ODBC connection to a (typically SQL) database

Directories may only be opened for reading, and produce the names of all files, one per line. Pipes may be opened for reading or writing, but not both.

When opening a network socket: the first argument s1 is the name of the socket to connect to: if of the form "s:i", it is an Internet domain socket on host s and port i; otherwise, it's the name of a Unix domain socket. If the host name is null, it represents the current host.

For a UDP socket, `connect' means that any writes to that file will send a datagram to that address, so that the address doesn't have to be specified each time. Also, read or reads cannot be performed on a UDP socket; use receive. UDP sockets must be in the INET domain, i.e. the address must have a colon.

For a DBM database, only one modifier character may be used: if s1 is "dr" it indicates that the database should be opened in read-only mode.

The filename argument is a Uniform Resource Indicator (URI) when opening a messaging connection. Arguments after the mode "m" are sent as headers. The HTTP User-Agent header defaults to "Unicon Messaging/10.0" and Host defaults to the host and port indicated in the URI. The SMTP From: header obtains its default from a UNICON_USERADDRESS environment variable if it is present.

---

opmask(co-expression, cset) : cset — opcode mask

opmask(ce) gets ce's program's opcode mask. The function returns &null if there is no opcode mask. opmask(ce,cs) sets ce's program's opcode mask to cs. This function is part of the execution monitoring facilities.

---

ord(string) : integer — ordinal value

ord(s) produces the integer ordinal (value) of s, which must be of size 1.

---

paramnames(co-expression, integer:0) : string* — parameter names

paramnames(ce,i) produces the names of the parameters in the procedure activation i levels above the current activation in ce

---

parent(co-expression) : co-expression — parent program

parent(ce) given a ce, return &main for that ce's parent. This is interesting only when programs are dynamically loaded using the load() function.

---

pipe() : list — create pipe

pipe() creates a pipe and returns a list of two file objects. The first is for reading, the second is for writing. See also function filepair().

---

pop(list|Message) : any? — pop from stack

pop(L) removes an element from the top of the stack (L[1]) and returns it. pop(M) removes and returns the first message in POP mailbox connection M; the actual deletion occurs when the messaging connection is closed.

pos(integer) : integer? — test scanning position

pos(i) tests whether &pos is at position i in &subject.

proc(any, integer:1) : procedure? — convert to procedure

proc(x,i) converts x to a procedure if that is possible. Parameter i is used to resolve ambiguous string names. i must be either 0, 1, 2, or 3. If i is 0, a built-in function is returned if it is available, even if the global identifier by that name has been assigned differently. If i is 1, 2, or 3, the procedure for an operator with that number of operands is produced. For example, proc("-",2) produces the procedure for subtraction, while proc("-") produces the procedure for unary negation.

pull(list) : any? — remove from list end

pull(L) removes and produces an element from the end of list L, if L is nonempty.

push(list, any, ...) : list — push on to stack

push(L, x1, ..., xN) pushes elements onto the beginning of list L. The order of the elements added to the list is the reverse of the order they are supplied as parameters to the call to push(). push() returns the same list that is passed as its first parameter, with the new elements added.

put(list, x1, ..., xN) : list — add to list end

put(L, x1, ..., xN) puts elements onto the end of list L.

read(file:&input) : string? — read line

read(f) reads a line from file f. The end of line marker is discarded.

reads(file:&input, integer:1) : string? — read characters

reads(f,i) reads up to i characters from file f. It fails on end of file. If f is a network connection, reads() returns as soon as it has input available, even if fewer than i characters were delivered. If i is -1, reads() reads and produces the entire file as a string. Care should be exercised when using this feature to read very large files.

real(any) : real? — convert to real

real(x) converts x to a real, or fails if the conversion cannot be performed.

receive(file) : record — receive datagram

receive(f) reads a datagram addressed to the port associated with f, waiting if necessary. The returned value is a record of type posix_message(addr, msg), containing the address of the sender and the contents of the message respectively.

remove(string) : ? — remove file

remove(s) removes the file named s.

rename(string, string) : ? — rename file

rename(s1,s2) renames the file named s1 to have the name s2.

repl(x, integer) : x — replicate

repl(x, i) concatenates and returns i copies of string or list x.

reverse(x) : x — reverse sequence

reverse(x) returns a value that is the reverse of string or list x.

right(string, integer:1, string:" ") : string — right format string

right(s1,i,s2) produces a string of length i. If i<*s1, s1 is truncated. Otherwise, the function pads s1 on left with s2 to length i.

rmdir(string) : ? — remove directory

rmdir(d) removes directory d. rmdir() fails if directory d is not empty or does not exist.

rtod(real) : real — convert radians to degrees

rtod(r1) produces the equivalent of r1 radians, expressed in degrees.

runerr(integer, any) — runtime error

runerr(i,x) produces runtime error i with value x. Program execution is terminated.

seek(file, any) : file? — seek to file offset

seek(f,i) seeks to offset i in file f, if it is possible. If f is a regular file, i must be an integer. If f is a database, i seeks a position within the current set of selected rows. The position is selected numerically if i is convertible to an integer; otherwise i must be convertible to a string and the position is selected associatively by the priimary key.

select(x1, x2, ?) : list — files with available input

select(files?, timeout) waits for a input to become available on any of several files, including network connections or windows. Its arguments may be files or lists of files, ending with an optional integer timeout value in milliseconds. It returns a list of those files among its arguments that have input waiting.

If the final argument to select() is an integer, it is an upper bound on the time elapsed before select returns. A timeout of 0 causes select() to return immediately with a list of files on which input is currently pending. If no files are given, select() waits for its timeout to expire. If no timeout is given, select() waits forever for available input on one of its file arguments. Directories and databases cannot be arguments to select().

send(string, string) : null — send datagram

send(s1, s2) sends a UDP datagram to the address s1 (in host:port format) with the contents s2.

seq(integer:1, integer:1) : integer* — generate sequence

seq(i, j) generates the infinite sequence i, i+j, i+2*j, ...j may not be 0.

serial(x) : integer? structure serial number

serial(x) returns the serial number for structure x, if it has one. Serial numbers uniquely identify structure values.

set(x, ...) : set create set

set() creates a set. Arguments are inserted into the new set, with the exception of lists. set(L) creates a set whose members are the elements of list L.

setenv(s) : ? set environment variable

setenv() sets an environment variable s in the operating system.

sort(x, integer:1) : list sort structure

sort(x, i) sort structure x. If x is a table, parameter i is the sort method. If i is 1 or 2, the table is sorted into a list of lists of the form [key, value]. If i is 3 or 4, the table is sorted into a list of alternating keys and values. Sorting is by keys for odd-values of i, and by table element values for even-values of i.

sortf(x, integer:1) : list sort by field

sortf(x,i) sorts a list, record, or set x using field i of each element that has one. Elements that don't have an i'th field are sorted in standard order and come before those that do have an i'th field.

sql(database, string) : integer execute SQL statement

sql(db, query) executes arbitrary SQL code on db. This function allows the program to do vendor-specific SQL and many SQL statements that cannot be expressed otherwise using the Unicon database facilities. sql() can leave the database in an arbitrary state and should be used with care.

sqrt(real) : real square root

sqrt(r) produces the square root of r.

stat(f) : record? get file information

stat(f) returns a record with information about the file f which may be a path or a file object. The return value is of type: record posix_stat(dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, blocks, symlink) Many of these fields are POSIX specific, but a number are supported across platforms, such as the file size in bytes (the size field), access permissions (the mode field), and the last modified time (the mtime field).

The atime, mtime, and ctime fields are integers that may be formatted with the ctime() and mtime() functions. The mode is a string similar to the long listing option of the UNIX ls(1) command. For example, "-rwxrwsr-x" represents a plain file with a mode of 2775 (octal). stat(f) fails if filename or path f does not exist.

staticnames(co-expression:&current, integer:0) : string* static variable names

staticnames(ce,i) generates the names of static variables in the procedure i levels above the current activation in ce.

stop(s|f, ...) : stop execution

stop(args) halts execution after writing out its string arguments, followed by a newline, to &errout. If any argument is a file, subsequent string arguments are written to that file instead of &errout. The program exit status indicates that an error has occurred.

string(x) : string? convert to string

string(x) converts x to a string and returns the result, or fails if the value cannot be converted.

system(x, file:&input, file:&output, file:&errout, string) : integer
execute system command

system(x, f1, f2, f3, waitflag) launches execution of a program in a separate process. x can be either a string or a list of strings. In the former case, whitespace is used to separate the arguments and the command is processed by the platform's command interpreter.. In the second case, each member of the list is an argument and the second and subsequent list elements are passed unmodified to the program named in the first element of the list.

The three file arguments are files that will be used for the new process' standard input, standard output and standard error. If the waitflag argument is "nowait", system will return immediately after spawning the new process. The default is for it to wait for the spawned process to exit. The return value is the exit status from the process, unless the waitflag was set to "nowait", in which case the return value is the process id of the new process.

tab(integer:0) : string? set scanning position

tab(i) sets &pos to i and returns the substring of &subject spanned by the former and new positions. tab(0) moves the position to the end of the string. This function reverses its effects by resetting the position to its old value if it is resumed.

table(x) : table create table

table(x) creates a table with default value x. If x is a mutable value such as a list, all references to the default value refer to the *same* value, not a separate copy for each key.

tan(real) : real tangent

tan(r) produces the tangent of r in radians.

trap(string, procedure) : procedure trap or untrap signal

trap(s, proc) sets up a signal handler for the signal s (the name of the signal). The old handler (if any) is returned. If proc is &null, the signal is reset to its default value.

Caveat: This is not supported by the optimizing compiler, iconc!

trim(string, cset:' ') : string — trim string

trim(s,c) removes trailing characters in c from s.

truncate(f, integer) : ? — truncate file

truncate(f, len) changes the file f (which may be a string filename, or an open file) to be no longer than length len. truncate() does not work on windows, network connections, pipes, or databases.

type(x) : string — type of value

type(x) returns a string that indicates the type of x.

upto(cset, string, integer, integer) : integer* — find characters in set

String scanning function upto(c,s,i1,i2) generates the sequence of integer positions in s up to a character in c in s[i2:i2], but fails if there is no such position.

utime(string, integer, integer) : null — file access/modification times

utime(f, atime, mtime) sets the access time for a file named f to atime and the modification time to mtime. The ctime is set to the current time. The effects of this function are platform specific. Some filesystems support only a subset of these times.

variable(string, co-expression:&current, integer:0) : any? — get variable

variable(s, c, i) finds the variable with name s and returns a variable descriptor that refers to its value. The name s is searched for within co-expression c, starting with local variables i levels above the current procedure frame, and then among the global variables in the program that created c.

where(file) : integer? — file position

where(f) returns the current offset position in file f. It fails on windows and networks. The beginning of the file is offset 1.

write(s|f, ...) : s|f — write text line

write(args) outputs strings, followed by a newline, to a file or files. Strings are written in order to their nearest preceding file, defaulting to &output. A newline is output to the preceding file after the last argument, as well as whenever a noninitial file argument directs output to a different file. write() returns its last argument.

writes(s|f, ...) : s|f — write strings

writes(args) outputs strings to one or more files. Each string argument is written to the nearest preceding file argument, defaulting to &output. writes() returns its last argument.

Graphics Functions

Unlike the other built-in functions, the names of built-in graphics functions begin with upper case. User functions, and library functions described in the next appendix, may begin with either upper or lower case. Built-in graphics functions are listed here so that you do not inadvertently use the names and get global identifiers where locals are

intended. These functions are described in [Griswold98]. You should consult that book when reading Icon programs that use graphics.

Active() - produce the next active window
Alert(w,volume) - alert the user
Bg(w,s) - background color
Clip(w, x, y, w, h) - set context clip rectangle
Clone(w, attribs...) - create a new context bound to w's canvas
Color(argv[]) - return or set color map entries
ColorValue(w,s) - produce RGB components from string color name
CopyArea(w,w2,x,y,width,height,x2,y2) - copy area
Couple(w,w2) - couple canvas to context
DrawArc(argv[]) - draw arc
DrawCircle(argv[]) - draw circle
DrawCurve(argv[]) - draw curve
DrawImage(w,x,y,s) - draw bitmapped figure
DrawLine(argv[]) - draw line
DrawPoint(argv[]) - draw point
DrawPolygon(argv[]) - draw polygon
DrawRectangle(argv[]) - draw rectangle
DrawSegment(argv[]) - draw line segment
DrawString(argv[]) - draw text
EraseArea(w,x,y,width,height) - clear an area of the window
EvGet(c,flag) - read through the next event token having a code matched by cset c.
Event(W, i) : a - return next window event
Fg(w,s) - foreground color
FillArc(arcs?)- fill arc
FillCircle(circles?)- draw filled circle
FillPolygon(points?) - fill polygon
FillRectangle(rectangles?)- draw filled rectangle
Font(w,s) - get/set font
FreeColor(colors?) - free colors
GotoRC(w,r,c) - move cursor to a particular text row and column
GotoXY(w,x,y) - move cursor to a particular pixel location

Lower(w) - lower w to the bottom of the window stack
NewColor(w,s) - allocate an entry in the color map
PaletteChars(w,p) - return the characters forming keys to palette p
PaletteColor(w,p,s) - return color of key s in palette p
PaletteKey(w,p,s) - return key of closest color to s in palette p
Pattern(w,s) - sets the context fill pattern by string name
Pending(w,x[]) - produce a list of events pending on window
Pixel(w,x,y,width,height) - produce the contents of some pixels
QueryPointer(w) - produce mouse position
Raise(w) - raise w to the top of the window stack
ReadImage(w, s, x, y, p) - load image file
TextWidth(w,s) - compute text pixel width
Uncouple(w) - uncouple window
WAttrib(argv[]) - read/write window attributes

WDefault(w,program,option) - get a default value from the environment
WFlush(w) - flush all output to window w
WSync(w) - synchronize with server
WriteImage(w,filename,x,y,width,height) - write an image to a file

One of the graphics functions, Event(w, i), has been extended in Unicon as follows. The new, second argument i is a timeout value in milliseconds. If the timeout expires before an event is available, then Event() fails. The default for i signifies that Event() should wait forever for the next window event.

Preprocessor

Unicon features a simple preprocessor that supports file inclusion and symbolic constants. It is a subset of the capabilities found in the C preprocessor, and is used primarily to support platform-specific code sections and large collections of symbols.

Preprocessor Commands

Preprocessor directives are lines beginning with a dollar sign. The available preprocessor commands are:

$define *symbol text* — symbolic substitution

All subsequent occurrences of *symbol* are replaced by the *text* within the current file. Note that $define does not support arguments, unlike C.

$include *filename* — insert source file

The named file is inserted into the compilation in place of the $include line.

$ifdef *symbol* — conditional compilation

$ifndef *symbol* — conditional compilation

$else — conditional alternative

$endif — end of conditional code

The subsequent lines of code, up to an $else or $endif, are discarded unless *symbol* is defined by some $define directive. $ifndef reverses this logic.

$error *text* — compile error

The compiler will emit an error with the supplied text as a message.

$line *number* [*filename*] — source code line

#line *number* [*filename*] — source code line

The subsequent lines of code are treated by the compiler as commencing from line *number* in the file *filename* or the current file if no filename is given.

$undef *symbol* — remove symbol definition

Subsequent occurrences of *symbol* are no longer replaced by any substitute text.

$(	**{**
$)	**}**
$<	**[**
$>	**]**

These character combinations are substitutes for curly and square brackets on keyboards that do not have these characters.

Predefined Symbols

Predefined symbols are provided for each platform and each feature that is optionally compiled in on some platforms. These symbols include:

Preprocessor Symbol	Feature
_V9	Version 9
_AMIGA	Amiga
_ACORN	Acorn Archimedes
_CMS	CMS
_MACINTOSH	Macintosh
_MSDOS_386	MS-DOS/386
_MS_WINDOWS_NT	MS Windows NT
_MSDOS	MS-DOS
_MVS	MVS
_OS2	OS/2
_PORT	PORT
_UNIX	UNIX
_POSIX	POSIX
_DBM	DBM
_VMS	VMS
_ASCII	ASCII
_EBCDIC	EBCDIC
_CO_EXPRESSIONS	co-expressions
_CONSOLE_WINDOW	console window
_DYNAMIC_LOADING	dynamic loading
_EVENT_MONITOR	event monitoring
_EXTERNAL_FUNCTIONS	external functions
_KEYBOARD_FUNCTIONS	keyboard functions
_LARGE_INTEGERS	large integers
_MULTITASKING	multiple programs
_PIPES	pipes
_RECORD_IO	record I/O
_SYSTEM_FUNCTION	system function
_MESSAGING	messaging
_GRAPHICS	graphics
_X_WINDOW_SYSTEM	X Windows
_MS_WINDOWS	MS Windows
_WIN32	Win32
_PRESENTATION_MGR	Presentation Manager

_ARM_FUNCTIONS	Archimedes extensions
_DOS_FUNCTIONS	MS-DOS extensions

Execution Errors

There are two kinds of errors that can occur during the execution of an Icon program: runtime errors and system errors. Runtime errors occur when a semantic or logic error in a program results in a computation that cannot perform as instructed. System errors occur when an operating system call fails to perform a required service.

Runtime Errors

By default, a runtime error causes program execution to abort. Runtime errors are reported by name as well as by number. They are accompanied by an error traceback that shows the procedure call stack and value that caused the error, if there is one. The errors are listed below to illustrate the kinds of situations that can cause execution to terminate.

The keyword &error allows a program to convert runtime errors into expression failure. In the event that an expression fails due to a converted runtime error, the keywords &errornumber, &errortext, and &errorvalue provide information about the nature of the error.

101	integer expected or out of range
102	numeric expected
103	string expected
104	cset expected
105	file expected
106	procedure or integer expected
107	record expected
108	list expected
109	string or file expected
110	string or list expected
111	variable expected
112	invalid type to size operation
113	invalid type to random operation
114	invalid type to subscript operation
115	structure expected
116	invalid type to element generator
117	missing main procedure
118	co-expression expected
119	set expected
120	two csets or two sets expected
121	function not supported
122	set or table expected
123	invalid type
124	table expected
125	list, record, or set expected
126	list or record expected
140	window expected
141	program terminated by window manager

142 attempt to read/write on closed window
143 malformed event queue
144 window system error
145 bad window attribute
146 incorrect number of arguments to drawing function
147 window attribute cannot be read or written as requested
160 cannot open file
161 bad file attribute
162 cannot open socket
170 string or integer expected
171 posix header file not included
172 posix record overridden by global value
173 directory opened for writing
174 directory or database invalid as file
175 invalid mode string
176 invalid signal
177 invalid operation to flock/fcntl
178 invalid procedure type
179 fdup of closed file
180 low-level read or select mixed with buffered read
181 not a network connection
182 not a UDP socket
183 invalid protocol name
184 invalid permission string for umask
190 database expected
201 division by zero
202 remaindering by zero
203 integer overflow
204 real overflow, underflow, or division by zero
205 invalid value
206 negative first argument to real exponentiation
207 invalid field name
208 second and third arguments to map of unequal length
209 invalid second argument to open
210 non-ascending arguments to detab/entab
211 by value equal to zero
212 attempt to read file not open for reading
213 attempt to write file not open for writing
214 input/output error
215 attempt to refresh &main
216 external function not found
301 evaluation stack overflow
302 memory violation
303 inadequate space for evaluation stack
304 inadequate space in qualifier list
305 inadequate space for static allocation
306 inadequate space in string region
307 inadequate space in block region

308 system stack overflow in co-expression
401 co-expressions not implemented
402 program not compiled with debugging option
500 program malfunction
600 vidget usage error

System Errors

If an error occurs during the execution of a system function, by default the function fails and keywords &errornumber, &errortext and &errorvalue will be set. This contrasts with runtime errors, which terminate execution by default. Whereas runtime errors can be converted to failure by setting &error, system errors can be converted to a runtime error by setting keyword &syserr to a non-null value.

The complete set of system errors is by definition platform specific. Error numbers above the value 1000 are used for system errors. Many of the POSIX standard system errors are supported across platforms, and error numbers between 1001 and 1040 are reserved for the system errors listed below. Platforms may report other system error codes so long as they do not conflict with existing runtime or system error codes.

1001 Operation not permitted
1002 No such file or directory
1003 No such process
1004 Interrupted system call
1005 I/O error
1006 No such device or address
1007 Arg list too long
1008 Exec format error
1009 Bad file number
1010 No child processes
1011 Try again
1012 Out of memory
1013 Permission denied
1014 Bad address
1016 Device or resource busy
1017 File exists
1018 Cross-device link
1019 No such device
1020 Not a directory
1021 Is a directory
1022 Invalid argument
1023 File table overflow
1024 Too many open files
1025 Not a typewriter
1027 File too large
1028 No space left on device
1029 Illegal seek
1030 Read-only file system
1031 Too many links
1032 Broken pipe

1033 Math argument out of domain of func
1034 Math result not representable
1035 Resource deadlock would occur
1036 File name too long
1037 No record locks available
1038 Function not implemented
1039 Directory not empty